TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

TUOMINEN JANNE

OLIOPERUSTAISET SUUNNITTELUMALLIT FUNKTIONAALISESSA OHJELMOINNISSA

Diplomityö

Tarkastaja:
professori Hannu-Matti Järvinen

Tarkastaja ja aihe hyväksytty:
28. marraskuuta 2018

TIIVISTELMÄ

JANNE TUOMINEN: Olioperustaiset suunnittelumallit funktionaalisessa ohjelmoinnissa
Tampereen teknillinen yliopisto
Diplomityö, 45 sivua, 24 liitesivua
Marraskuu 2018
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Hannu-Matti Järvinen

Avainsanat: suunnittelumalli, funktionaalinen ohjelmointi

Olioperustainen ohjelmointi on ottanut valta-aseman ohjelmistoalalla viimeisten parin vuosikymmenen aikana. Suosionsa myötä alan koulutus, tutkimus ja kehitys on keskittynyt pitkälti sen ympärille. Tämän johdosta olioperustaisen ohjelmistokehityksen tueksi on kehitetty lukuisia eri tekniikoita, menetelmiä ja prosesseja. Yksi tärkeimmistä, ellei tärkein, näistä on ollut suunnittelumallien käyttö.

Funktionaalinen ohjelmointi on ollut pitkään olioperustaisen ohjelmoinnin varjossa. Mielenkiinto sitä kohtaan on viime vuosina ollut kuitenkin kasvussa. Perinteisiin olio-ohjelmointikieliin on lisätty funktionaalisia ominaisuuksia ja myös kokonaan uusia funktionaalisia ohjelmointikieliä on julkaistu. Puhdas funktionaalinen ohjelmointi ratkaisee joitakin imperatiivisessa ohjelmoinnissa esiintyneitä ongelmia lähes automaattisesti. Tällaisia ovat muun muassa rinnakkaisuuden hallinta sekä testattavuus, jotka molemmat ovat korostuneet ohjelmistojen kompleksisuuden kasvaessa.

Työssä tutkittiin, miten 23 olioperustaista suunnittelumallia soveltuu funktionaalisen ohjelmoinnin asiayhteyteen. Johtuen olioperustaisen ja funktionaalisen ohjelmoinnin perustavaa laatua olevista eroista, suoraviivainen siirtäminen ohjelmointityylistä toiseen ei ollut mahdollista. Sen sijaan jokaisen suunnittelumallin kohdalla pyrittiin siirtämään sen ajatus funktionaaliseksi rakenteeksi.

Tutkielma osoitti, että funktionaalinen ohjelmointi korvaa huomattavan osan olioperustaisista suunnittelumalleista triviaaleilla rakenteilla. Huomattava osa suunnittelumalleista oli joko teennäisiä tai tarpeettomia funktionaalisessa asiayhteydessä. Huolimatta olioperustaisen ja funktionaalisen ohjelmoinnin suurista eroista, ainoastaan muutamaa suunnittelumallia ei ollut mahdollista soveltaa funktionaaliseen asiayhteyteen.

ABSTRACT

JANNE TUOMINEN: Object-oriented design patterns in functional programming
Tampere University of Technology
Master of Science Thesis, 45 pages, 24 Appendix pages
November 2018
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Professor Hannu-Matti Järvinen

Keywords: design pattern, functional programming

During the last few decades object-oriented programming has taken dominant role in software industry. Because its popularity, the education, research and development has mainly focused around it. Due to this, many techniques, methods and processes have been developed to aid object-oriented software development. One of the most important ones, if not the most important, has been design patterns.

Functional programming has been shadowed by object-oriented programming for a long time. However, interest towards it has been increasing during recent years. Functional features have been added to traditional object-oriented languages and some new functional languages have been published. Pure functional programming automatically solves some of the problems occurring in imperative programming. Examples of these are managing concurrency and testability, which both have become more important due to increasing complexity of software systems.

In this thesis, 23 object-oriented design patterns were explored in terms of their suitability to functional programming. It was impossible to directly transform design patterns to functional context, due to fundamental differences between these two programming paradigms. Instead, the goal was to transform the idea behind each of the design patterns to functional context.

The thesis demonstrates that functional programming replaces a considerable amount of object-oriented design patterns with trivial constructs. Large number of the design patterns were either irrelevant or artificial in functional context. Despite the fundamental differences between these two paradigms, only a few of the patterns did not fit at all to functional context.

ALKUSANAT

Tutustuin funktionaaliseen ohjelmointiin vasta hiljattain F#:n ja Haskellin kautta. Opiskellessani lisää aiheesta, huomasin väitteitä, joiden mukaan se tekee suunnittelumalleista pitkälti tarpeettomia. Asiasta ei kuitenkaan löytynyt selkeätä tieteellistä tutkielmaa. Tämä tarjosi oivan aiheen diplomityön tekemiselle.

Haluan esittää kiitokset koko Tampereen teknillisen yliopiston väelle sekä professori Hannu-Matti Järviselle, jonka kanssa sain käydä antoisia keskusteluita aiheesta ja sen vierestä.

Tampereella, 5.11.2018

Janne Tuominen

SISÄLLYSLUETTELO

.

1. JOHDANTO

Suunnittelumallit ovat olleet keskeisessä asemassa olioperustaista ohjelmointia. Ne ovat auttaneet luomaan modulaarisia, ylläpidettäviä ja helposti laajennettavia ohjelmia. Niiden hyödyt ovat olleet kiistattomat ohjelmointialalla. Muun muassa riippuvuusinjektio sekä SOLID-periaatteet nojaavat niiden käyttöön. Toisinaan suunnittelumallien on kritisoitu olevan vain paikkoja olioperustaisen ohjelmoinnin puutteille.

Funktionaalisen ohjelmoinnin historia on pitkä, mutta se on vuosikymmeniä pysynyt olio-ohjelmoinnin varjossa. Funktionaaliset ohjelmointikielet ovat kuitenkin viime vuosina saaneet yhä enemmän huomiota. Haskell on ollut jo pitkään suosittu varsinkin akateemisissa piireissä. Perinteisiin imperatiivisiin ohjelmointikieliin on lisätty funktionaalisia piirteitä ja lisäksi on julkaistu kokonaan uusia funktionaalisia ohjelmointikieliä.

Tutkielman tarkoitus on selvittää, miten nämä kaksi konseptia sopivat yhteen. Tutkielmassa tarkastellaan 23:a laajalti tunnettua olioperustaista suunnittelumallia funktionaalisessa asiayhteydessä. Valitut suunnittelumallit on alun perin esitetty *Design Patterns: Elements of Reusable Object-Oriented Software* -kirjassa. Vaikka kirjassa esitetyt suunnittelumallit on tarkoitettu olioperustaisen ohjelmoinnin tarpeisiin, niiden taustalla vaikuttavat ongelmat saattavat olla ohjelmointityylistä riippumattomia. Tutkielmassa selvitetään millaiset olioperustaiset suunnittelumallit ovat mielekkäitä funktionaalisessa asiayhteydessä sekä miten eri toteutustapa niihin vaikuttaa.

Tutkielma tehdään toteuttamalla jokainen suunnittelumalli funktionaalisen ohjelmoinnin menetelmin. Jokaisen toteutuksen yhteydessä analysoidaan, onko se mielekäs funktionaalisessa kontekstissa sekä millä tavoin se eroaa alkuperäisestä olioperustaisesta toteutuksesta. Analysoimalla suunnittelumallien funktionaalisia toteutuksia pyritään löytämään yhteisiä piirteitä, joiden perusteella voidaan arvioida millaiset suunnittelumallit ovat yleisesti ottaen mielekkäitä funktionaalisessa kontekstissa ja millä tavalla toteutukset poikkeavat olioperustaisista toteutuksista.

Työn on rakenteeltaan kolmiosainen:

- luvuissa 2–3 käydään läpi tutkielmaan liittyvät taustatiedot funktionaalisesta ohjelmoinnista sekä suunnittelumalleista
- luvuissa 4–5 esitellään tutkimusmenetelmä sekä sovelletaan sitä esimerkinomaisesti neljään suunnittelumalliin
- luvuissa 6–7 tarkastellaan tutkielman tuloksia ja suoritetaan yhteenveto.

Olioperustaisten suunnittelumallien soveltuvuutta muihin ohjelmointityyleihin on tutkittu jo aiemmin. Peter Norvig käsitteli niitä dynaamisen ohjelmoinnin kannalta vuonna 1998 julkaistussa työssään ”Design Patterns in Dynamic Languages”. Jan Hannemann ja Gregor Kiczales puolestaan tarkastelivat niitä aspektiperustaisessa ohjelmoinnissa julkaisussaan ”Design Pattern Implementation in Java and AspectJ”. Verkosta löytyy myös lukuisia vähemmän tieteellisiä artikkeleita ja blogikirjoituksia aiheesta.

2. SUUNNITTELUMALLIT

Suunnittelumallit ovat alun perin olioperustaisen ohjelmoinnin tarpeisiin dokumentoituja yleiskäyttöisiä ratkaisumalleja. Ensimmäiset formaalisti dokumentoidut suunnittelumallit on esitetty teoksessa *Design Patterns: Elements of Reusable Object-Oriented Softare*. Vaikka nykyisin suunnittelumalleja on dokumentoitu huomattavasti enemmän, tämä työ rajataan koskemaan ainoastaan kyseisessä teoksessa esitettyjä.

Luvun tarkoituksena on antaa pohjatieto siitä, mitä suunnittelumallit ovat ja millaisia haasteita niiden käyttämiseen olioperustaisen ohjelmointityylin ulkopuolella mahdollisesti liittyy. Luvussa esitellään suunnittelumallin käsite sekä nimetään tutkittavat 23 suunnittelumallia lyhyinen kuvauksineen. Yksi niistä on *Strategy*, johon tutustutaan tarkemmin esimerkissä. Lopuksi käsitellään suunnittelumallien suhdetta olioperustaiseen ohjelmointiin.

2.1 Yleiskatsaus

Design Patterns -kirjassa suunnittelumallien määritellään olevan ”*Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.*” Suunnittelumallilta on nimenomaan vaadittu suurempaa rakennetta, jossa useammat oliot tai luokat ovat keskinäisessä vuorovaikutuksessa. Niiden ulkopuolelle on rajattu yksinkertaisemmat rakenteet, kuten linkitetyt listat tai hajautustaulut. (Gamma 2000 s. 3)

Suunnittelumallien historia näkyy niiden määrittelystä. Oliot ja luokat eivät ole mielekkäitä käsitteitä olioperustaisen ohjelmoinnin ulkopuolella. Tämän tutkielman kannalta suunnittelumalli käsitetään ilman olioperustaista asiayhteyttä. Suunnittelumalli on kuvaus ratkaisumallista, jota voidaan soveltaa tietyntyyppisten suunnitteluongelmien ratkaisemiseen.

Suunnittelumallin dokumentaatio koostuu useista osista. Oleellisimmat niistä ovat suunnittelumallin nimi, ongelman kuvaus, ratkaisun kuvaus sekä käytön seuraukset (Gamma 2000 s. 3). Kirjassa nämä seikat on kuvattu järjestelmällisesti kunkin suunnittelumallin kohdalla. Jokainen suunnittelumalli on nimetty, ja sen käyttötarkoitus on kuvattu muutamalla lauseella. Suunnittelumallista on annettu myös sen rakenteen kuvaus erilaisin kaavioin sekä selitetty, miten sen eri osat ovat vuorovaikutuksessa keskenään. Suunnittelumallin käytön seurauksia on pohdittu omassa kohdassaan.

Suunnittelumalleja on esitetty kirjassa 23 kappaletta. Niistä jokainen ratkaisee erityyppisen ongelman, kuten rajapinnan muuntamisen soveltuvammaksi (*Adapter*) tai algoritmin

osavaiheiden varioinnin (*Template Method*). Suunnittelumallit on jaettu kolmeen eri pääkategoriaan: *luontimalleihin*, *rakennemalleihin* ja *käyttäytymismalleihin* niiden pääasiallisen tarkoituksen mukaan.

Esimerkki suunnittelumallista

Ohessa on esitelty suunnittelumalli *Strategy* mukaillen *Design Patterns* -kirjan esitystapaa. Suunnittelumallista on kirjassa esitetty myös muita kohtia, kuten esimerkkitoteutus sekä käyttötapaukset.

Nimi
Strategy

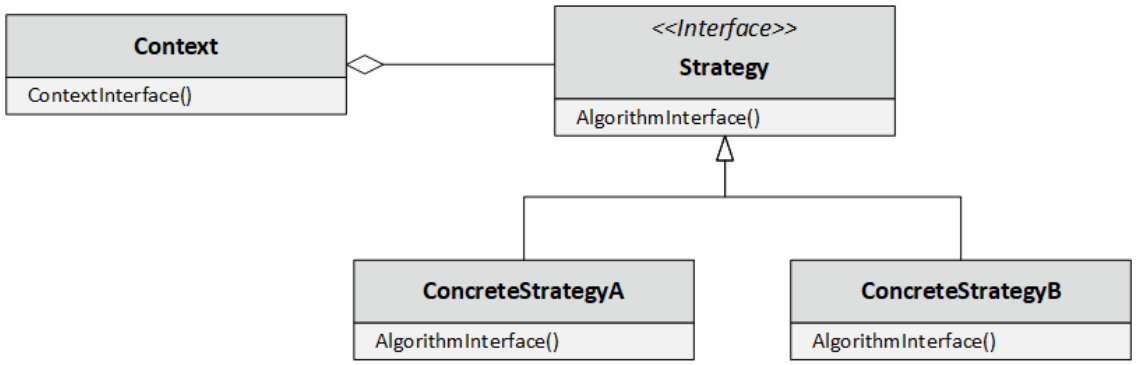
Luokittelu
Käyttäytymismalli

Tarkoitus
Toteuttaa joukko algoritmeja omiin luokkiinsa, jotka ovat vaihdannaisia keskenään yhteisen rajapinnan ansiosta. Strategy poistaa riippuvuuden asiakkaiden ja niiden käyttämien algoritmien väliltä.

Soveltuvuus
Mallia voidaan käyttää, kun

- halutaan konfiguroida luokan toiminnallisuus varioitavalla algoritmilla
- tarvitaan algoritmista useita muunnelmia, esimerkiksi nopeuden ja muistinkulutuksen suhteen
- halutaan tehdä asiakas riippumattomaksi algoritmin käyttämästä datasta
- halutaan erottaa algoritmit luokan sisältä omiksi strategialuokikseen.

Rakenne



Kuva 1. Strategy-suunnittelumallin luokkakaavio

Osallistujat

- *Strategy* määrittää yhteisen rajapinnan joukolle algoritmeja.
- *ConcreteStrategy* toteuttaa yksittäisen algoritmin.
- *Context* konfiguroidaan konkreettisella strategiaoliolla, jonka avulla toteuttaa osan toiminnallisuudestaan.

Seuraukset

- Erottaa algoritmikohtaisen toiminnallisuuden luokan sisältä.
- Eliminoi periytetyt luokat tilanteissa, joissa halutaan varioida ainoastaan tiettyä luokan sisäistä toiminnallisuutta.
- Selkeyttää luokan sisäistä logiikkaa korvaamalla mahdollisesti monimutkaiset toimintaan vaikuttavat ehtolausekkeet yhdellä oliolla.
- Vaatii kommunikointia asiakkaan ja strategian välillä, mikä voi heikentää suorituskykyä.
- Lisää suoritusaikaisten olioiden lukumäärää, mikä voi lisätä muistinkulutusta.

2.2 Luontimallit

Luontimallit liittyvät olioiden luomiseen sekä yhteenkuuluvuuden hallintaan. Niiden avulla on mahdollista luoda olioita, tuntematta luotujen olioiden konkreettisia luokkia. Taulukossa 1 on esitelty tähän kategoriaan kuuluvat suunnittelumallit lyhyine kuvauksineen. (Gamma 2000 s. 81-136)

Abstract Factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations.
Factory Method	Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
Prototype	Specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype.
Singleton	Ensure a class only has one instance and provide a global point of access to it.

Taulukko 1. Luontimallit

Luontimallit toimivat pohjana sekä osalle SOLID-periaatteista että riippuvuusinjektiolle. Piilottamalla käyttävältä koodilta olion konkreettinen luokka saadaan ohjelmoitua abstraktiota vasten. Tämä edistää oikeankaltaista modulaarisuutta ja vähentää luokkien välisiä riippuvuuksia. (Martin 2000; Seeman 2012)

Huomionarvoisesti *Singleton* on toisinaan saanut kritiikkiä osakseen. Sitä on pidetty anti-suunnittelumallina. Tällainen suunnittelumalli vaikuttaa houkuttelevalta, mutta tuo mukanaan piilotettuja ongelmia. (Safyan 2016)

2.3 Rakennemallit

Rakennemallien tehtävänä on jäsennellä, miten luokista ja olioista muodostetaan monimutkaisempia rakenteita. Niiden avulla ohjelman rakenteesta on mahdollista saada selkeämpi ja ylläpidettävämpi. Osa niistä keskittyy ohjelman suoritusaikaiseen muunneltavuuteen osan liittyessä puolestaan staattisen rakenteen määrittelyyn. Taulukossa 2 on esitetty tähän kategoriaan kuuluvat suunnittelumallit lyhyine kuvauksineen. (Gamma 2000 s. 137-220)

Adapter	Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
Bridge	Decouple an abstraction from its implementation so that the two can vary independently.
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Decorator	Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Flyweight	Use sharing to support large numbers of fine-grained objects efficiently.
Proxy	Provide a surrogate or placeholder for another object to control access to it

Taulukko 2. Rakennemallit

Rakennemallit voivat muun muassa helpottaa ohjelman testattavuutta (*Proxy*) sekä saattaa vanhentunut koodipohja yhteensopivaksi uuden kanssa (*Adapter* ja *Facade*). Aspektipohjaisessa ohjelmoinnissa *Decorator* voi puolestaan lisätä halutun toiminnallisuuden valittuihin luokkiin.

2.4 Käyttäytymismallit

Viimeinen suunnittelumallien pääkategoria on käyttäytymismallit. Niiden tehtävä on määrittää ohjelman suoritusaikaista toimintaa, mukaan lukien erilaiset algoritmit, tilan hallitseminen sekä osien välinen kommunikointi. Taulukossa 3 on esitetty tähän kategoriaan kuuluvat suunnittelumallit. (Gamma 2000 s. 221-349)

Chain of Responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
Memento	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
Observer	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Taulukko 3. Käyttäytymismallit

Käyttäytymismallit ovat kategorioista selkeästi suurin käsittäen noin puolet tutkielman suunnittelumalleista. Niiden käyttötarkoitusten vaihtelu on myös laajaa vaihdellen spesifisestä tietorakenteen läpikäynnistä (*Iterator*) aina muunneltavien algoritmirunkojen muodostamiseen (*Template Method*).

2.5 Sidos olioperustaisuuteen

Suunnittelumallien kuvaukset, rakenteet ja esimerkit liittyvät pitkälti olioperustaiseen ohjelmointiin. Ne on valittu nimenomaan olioperustaisen ohjelmoinnin tarpeisiin liittyen. Proseduraalisessa ohjelmoinnissa perintä, tiedon ja toiminnallisuuden kapselointi sekä monimuotoisuus olisivat saattaneet olla omia suunnittelumallejaan. Suunnittelumallit eivät ole täysin selkeitä edes olioperustaisen ohjelmointityylin sisällä. Osa suunnittelumalleista on sisäänrakennettu osaan olioperustaisista ohjelmointikielistä. (Gamma 2000 s. 4)

Suunnittelumallien on toisinaan arveltu olevan merkki olioperustaisen ohjelmoinnin puutteista. Peter Norvig on osoittanut, että 16 *Design Patterns* -kirjassa esitellyistä 23 suunnittelumalleista yksinkertaistuu tai katoaa toteutettaessa se Lisp- tai Dylan-ohjelmointikielillä. (Norvig 1996) Samankaltaisia tuloksia saivat myös Jan Hannemann ja Gregor Kiczales tarkastellessaan niitä aspektiperustaisessa asiayhteydessä. (Hannemann 2002)

3. FUNKTIONAALINEN OHJELMOINTI

Tutkielman menetelmien ja tulosten ymmärtämisen kannalta on oleellista ymmärtää funktionaalisen ohjelmoinnin perusteet. Tämän luvun tarkoituksena on esittää, millaisia menetelmiä ja tekniikoita suunnittelumallien muuntamisessa olioperustaisista funktionaalisiksi on käytetty. Lisäksi se tarjoaa pohjatiedot, miksi olioperustaisten suunnittelumallien suoraviivainen muuntaminen funktionaalisiksi ei usein ole mahdollista.

Luvun alussa tutustutaan ensin, mitä funktionaalinen ohjelmointi on, sekä käydään läpi siihen liittyvää termistöä. Puhtaan funktionaalisen ohjelmoinnin käsite asettaa reunaehdot tutkielmassa käytetylle ohjelmointikielelle. Seuraavaksi käydään läpi yksityiskohtaisesti, mitä funktionaalisen ohjelmoinnin ominaisuuksia tämän tutkielman puitteissa on käytetty. Vaikka työn esimerkit on ohjelmoitu Haskellilla, ovat työn tulokset yleistettävissä mille tahansa ohjelmointikielelle, joka täyttää tässä luvussa asetetut vaatimukset. Lopuksi suoritetaan lyhyt vertailu funktionaalisen ja olioperustaisen ohjelmoinnin välillä.

3.1 Funktionaalisen ohjelmoinnin esittely

Funktionaalinen ohjelmointi on tapa ohjelmoida käyttäen funktioita ohjelman rakennuspalikoina. Se perustuu lambdakalkyyliin, joka on formaali menetelmä laskennan kuvaamiseen. Funktionaalisessa ohjelmoinnissa funktio käsitetään nimenomaan matemaattisessa mielessä eli kuvauksena parametrin arvosta paluuarvoksi. Funktioiden monipuolinen yhdistely on avainasemassa funktionaalisessa ohjelmoinnissa. Funktiot ovat yhdenvertaisessa asemassa muiden arvojen kanssa, joten niitä voidaan välittää sekä parametreina että paluuarvoina. (Petricek 2010 s. 4-6; Hudak 1989; HaskellWiki 2014)

Funktionaalisen ohjelmoinnin tarkka määritteleminen on hankalaa. Funktionaalisiksi luokiteltavia ohjelmointikieliä on lukuisia eri ominaisuuksiin painottuen. Ei voida määrittää mitään tiettyä joukkoa ominaisuuksia, joita funktionaalisen ohjelmointikielen tulisi tukea. Tunnusomaista funktionaaliselle ohjelmoinnille ovat monipuoliset tavat yhdistellä funktioita, tilattomuus, rekursiivisuus sekä laiska suorittaminen. Tutkielman lähtökohtana on käyttää puhtaan funktionaalisen ohjelmointikielen ominaisuuksia. (Petricek 2010 s. 4-6; Hudak 1989; HaskelWiki 2014)

3.2 Termistö

Funktionaalisen ohjelmoinnin suomenkielinen termistö ei ole täysin vakiintunut. Sekaannusta saattaa aiheuttaa lisäksi termien eri merkitys imperatiivisen ja funktionaalisen ohjelmoinnin välillä. Tässä tutkielmassa funktionaalisen ohjelmoinnin yhteydessä käytetään usein termejä *funktio*, *data*, *arvo* ja *tunniste*.

Funktio on matemaattinen kuvaus, joka muuntaa parametrin arvon paluuarvoksi. Funktiolla ei ole sivuvaikutuksia, ja sen paluuarvo riippuu ainoastaan annetusta argumentista. Funktio on usein sidottu tunnisteeseen. Sitomaton funktio on anonyymi, ja siitä käytetään termiä *anonyymi funktio*.

Data on tietoa, kuten kokonaislukuja, totuusarvoja tai merkkijonoja. Termiä käytetään tässä työssä kuvaamaan sekä primitiivisiä literaaleja että tietorakenteen – kuten listan tai puun – sisältämää tietoa.

Arvo on yhteisnimitys funktioille ja datalle. Funktionaalisessa ohjelmoinnissa funktiot ja data ovat yhdenvertaisia, joten niistä puhuttaessa on luontevaa käyttää yhtenäistä termiä. Funktionaalisessa ohjelmoinnissa arvot ovat vakioita.

Tunniste on funktiolle tai datalle, eli arvolle, annettu nimi. Tunnisteen avulla voidaan yksikäsitteisesti viitata tiettyyn arvoon. Tunniste on aina sidottu arvoon, eikä sitä voida muuttaa viittamaan toiseen arvoon.

Tämän tutkielman asiayhteydessä funktionaalisella ohjelmoinnilla tarkoitetaan puhdasta funktionaalista ohjelmointia. Myös tähän liittyy eriäviä mielipiteitä, mikä on puhdasta ja mikä ei. Tämän tutkielman kannalta puhtauteen liittyvät termit määritellään seuraavasti.

Puhdas funktionaalinen ohjelma on *funktio*. Puhtaalla funktionaalisella ohjelmalla ei ole havaittavia sivuvaikutuksia eikä tilaa. Se koostuu ainoastaan toisista *funktioista*, muuttumattomasta *datasta* sekä lausekkeista.

Puhdas funktionaalinen ohjelmointi on nimitys prosessille, joka tuottaa puhtaita funktionaalisia ohjelmia.

Puhdas funktionaalinen ohjelmointikieli on ohjelmointikieli, jolla voi tuottaa ainoastaan puhtaita funktionaalisia ohjelmia.

Puhtaan funktionaalisen ohjelmoinnin määritelmä on samaan aikaan sekä erittäin rajoittava että salliva. Se ei esimerkiksi kerro millaisia tietorakenteita tai tyypitysjärjestelmää ohjelmointikielen tulee tukea. Toisaalta sen puitteissa ei voi suorittaa I/O-operaatioita eikä käyttää lainkaan muuttujia.

3.3 Tutkielmassa käytetty ohjelmointikieli

Tutkielman perustana on puhdas funktionaalinen ohjelmointikieli. Käsitteen laajuuden takia se rajataan koskemaan seuraavia ominaisuuksia:

- funktiot
    - sulkeuma (*closure*)
    - korkeamman tason funktiot (higher order function)

    - funktioiden osittaissoveltaminen (*partial application*)
    - funktioiden monimuotoisuus (*polymorphism*)
    - anonyymit funktiot (*lambda expression*)
    - rekursio
- tietotyypit
    - perustietotyypit
    - monikko (*tuple*)
    - tietue (*record*)
    - algebrallinen tietotyyppi (*algebraic data type*)
    - lista
- mallin sovittaminen (*pattern matching*)
- laiska suorittaminen.

Tutkielman menetelmät ja tulokset ovat yleistettävissä mille tahansa ohjelmointikielelle, joka tukee edellä esitettyjä ominaisuuksia. Huomioitavaa on, että myös täysin epäpudas ohjelmointikieli voi täyttää nämä vaatimukset

Haskell on moderni yleiskäyttöinen funktionaalinen ohjelmointikieli, joka täyttää edellä asetetut vaatimukset. Sitä voidaan pitää myös puhtaana funktionaalisena ohjelmointikielenä (Sebesta 1999 s. 595). Haskell on näin ollen valittu tässä työssä käytettäväksi esimerkkikieleksi. Haskellin IO-mekanismia ei tämän työn puitteissa käytetä.

Seuraavissa kohdissa käydään yksityiskohtaisesti esimerkkien avulla läpi kyseiset ominaisuudet. Koodiesimerkit ovat pseudokoodia, jotka noudattelevat Haskellin kielioppia.

3.3.1 Sulkeuma

Sulkeuma on funktion sisäisten ja ulkopuolisten arvojen yhdistelmä, johon funktio voi viitata. Tämä mahdollistaa viittaamisen myös niihin funktioihin ja arvoihin, joita ei ole määritelty funktion rungossa tai saatu parametrina. Oheinen esimerkki havainnollistaa asian.

```
area :: Int -> Int -> Int
area a b = a * b

volume :: Int -> Int -> Int -> Int
volume a b c = area a b * c
```

Funktion *volume* sulkeuma käsittää parametrit a, b ja c sekä ulkopuolella määritellyn funktion *area*. Se laskee ensin suorakulmion pinta-alan käyttäen hyväkseen ulkopuolista funktiota ja lopulta kertoo sen korkeudella palauttaen suorakulmaisen särmiön tilavuuden.

Sulkeuma on funktionaalisessa ohjelmoinnissa lähes välttämättömyys. Ilman sitä funktion käyttämät toiset funktiot olisi pakko määritellä joko paikallisesti funktion sisällä tai

antaa parametreina. Omiin moduuleihinsa toteutetut funktiot sekä muut arvot on mahdollista tuoda sulkeuman avulla muiden funktioiden käyttöön.

3.3.2 Korkeamman tason funktiot

Funktiota, joka ottaa funktion parametrinaan tai palauttaa funktion paluuarvonaan, kutsutaan korkeamman tason funktioksi. Antamalla funktiolle parametriksi toinen funktio on mahdollista tuottaa yleiskäyttöisiä funktioita.

Oheisen esimerkin funktio *sort* on korkeamman tason funktio, sillä se ottaa parametrinaan toisen funktion. Kyseisessä funktiossa toteutetaan järjestelyalgoritmi. Parametrina annettavasta funktioista riippuen se järjestelee listan alkiot halutulla tavalla, kuten laskevaan tai nousevaan järjestykseen.

```
sort :: (a -> a -> Int) -> [a] -> [a]
sort comparer a = ...
```

Korkeamman tason funktiot mahdollistavat koodin jäsentelyn toiminnallisuuden perusteella eri funktioihin. Oikein käytettynä tämä johtaa selkeisiin yleiskäyttöisiin funktioihin, joita voidaan käyttää rakennuspalikoina monimutkaisemman toiminnallisuuden aikaansaamiseksi. Korkeamman tason funktiot ovat funktionaalisessa ohjelmoinnissa avainasemassa.

3.3.3 Funktioiden soveltaminen osittain

Funktioiden soveltaminen osittain (*partial application*) on funktion parametrien kiinnittämistä tiettyyn arvoon. Sen tuloksena muodostuu uusi alkuperäistä vastaava, mutta vähäparametrisempi funktio.

Oheisessa listauksessa funktio *increase* on saatu soveltamalla osittain funktio *add*. Muodostettu funktio vastaa toiminnallisuudeltaan alkuperäistä funktiota, mutta sen ensimmäinen parametri on kiinnitetty arvoon 1. Vastaavalla tavalla voitaisiin muodostaa funktio *decrease*. Alkuperäisen funktion sijaan tarjolla olisi nyt kaksi täsmällisempään käyttöön soveltuvaa funktiota.

```
add :: Int -> Int -> Int
add a b = a + b

increase :: Int -> Int
increase = add 1
```

Yleisesti ottaen funktioiden soveltaminen osittain on tapa luoda geneerisestä funktiosta uusi funktio, joka soveltuu paremmin tarvittavaan laskentaan. Soveltaminen osittain ei ole välttämätön ominaisuus, sillä se voitaisiin korvata luomalla uusi funktio alkuperäisen

funktion ympärille. Se on kuitenkin luonnollinen ja helppokäyttöinen tapa johtaa uusia funktioita.

3.3.4 Monimuotoiset funktiot

Monimuotoinen funktion on funktio, joka toimii erityyppisillä parametreilla. Funktionaalinen monimuotoisuus voidaan jakaa kahteen alalajiin sen perusteella, tarvitaanko monimuotoisesta funktiosta yksi vai useampi toteutus.

Oheisessa esimerkin funktio *append* on monimuotoinen funktio, sillä se toimii millä tahansa tyypillä `a`. Parametrin tyypistä välittämätön monimuotoisuus ei vaadi funktiosta useampaa toteutusta.

```
append :: a -> [a] -> [a]
append x xs = x : xs
```

Seuraavassa listauksessa on esitelty funktio *area*, joka edustaa parametrin tyyppiin perustuvaa monimuotoisuutta. Kyseisestä funktiosta on laadittu eri toteutukset tyypeille *Circle* ja *Square*. Suoritettava funktio valitaan sille välitettävän parametrin tyypin perusteella.

```
-- Implementation is selected based on the
class Shape a where
    area :: a -> Int

instance Shape Circle where
    area c = (radius c) * (radius c) * pi

instance Shape Square where
    area s = (side s) * (side s)
```

Funktionaalinen monimuotoisuus on tehokas tapa tehdä funktioista yleiskäyttöisempiä. Se vähentää sekä koodin monistumista että tarvetta käyttää korkeamman tason funktioita.

3.3.5 Anonyymit funktiot

Anonyymi funktio on funktio, jota ei ole sidottu mihinkään tunnisteeseen. Sen toiminta on ekvivalentti samanrunkoisen nimetyn funktion kanssa. Tunnisteen puuttumisen takia sitä voidaan käyttää ainoastaan suoraan muun koodin seassa.

Anonyymia funktiota on seuraavassa esimerkissä käytetty funktion *filter* parametrina. Se ottaa yhden kokonaisluvun parametrinaan ja palauttaa totuusarvon, joka kuvaa, oliko luku parillinen.

```
filter (\x -> x `mod` 2 == 0) [1..]
```

Pienet kertakäyttöiset apufunktiot ovat usein tarpeen funktionaalisessa ohjelmoinnissa. Anonyymit funktiot soveltuvat kyseiseen tarkoitukseen. Oikeaoppisesti käytettynä ne sujuvoittavat ohjelmointia, tekevät koodista ymmärrettävämpää sekä auttavat pitämään sulkeuman roskattomana.

3.3.6 Rekursio

Rekursiossa funktio kutsuu itseään joko suoraan tai epäsuoraan. Laiska suorittaminen mahdollistaa päättymättömät rekursiot.

Oheisen esimerkin funktio *sum* laskee listan alkioiden summan rekursiivisesti. Se poimii listalta yhden alkion kerrallaan ja summaa lopuksi listasta puretut alkiot yhteen. Tyhjä lista – jonka alkioiden summan voidaan ajatella olevan 0 – toimii rekursion loppuehtona.

```
sum :: [Int] -> Int
sum [] = 0
sum (a:tail) = a + sum tail
```

Rekursio on tärkeä konsepti funktionaalisessa ohjelmoinnissa. Se mahdollistaa muun muassa useiden algoritmien toteutuksen sekä erilaisten tietorakenteiden tehokkaan läpikäynnin. Se on myös usein ainoa tapa toteuttaa tietty asia puhtaasti funktionaalisesti.

3.3.7 Perustietotyypit

Perustietotyypit ovat ohjelmointikielen tarjoamia tietotyyppejä, joita ei ole johdettu muista tietotyypeistä. Niitä voidaan käyttää monimutkaisempien tietorakenteiden rakennuspalikoina.

Alla olevassa listauksessa on esimerkkejä perinteisistä perustietotyypeistä, kuten kokonaisluvuista, liukuluvuista, totuusarvoista ja merkeistä. Perustietotyypit muodostavat funktionaalisessa ohjelmoinnissa pohjan tiedon esittämiselle.

```
[1, 2, 10]    :: [Int]   -- integer
[1.0, 2.0]    :: [Float] -- floating point number
[True, False] :: [Bool]  -- boolean value
[‘a’, ‘b’]    :: [Char]  -- character
```

3.3.8 Monikko

Monikko pitää sisällään järjestyksessä muita tietotyyppejä anonyymeissä kentissä. Monikon tyypin määrää sen sisältämien tietotyyppien lukumäärä sekä järjestys.

Oheisessa esimerkissä on luotu *monikko* edustamaan henkilötietoja, tässä tapauksessa nimeä ja ikää. Sen kentät ovat anonyymejä, mutta niiden järjestys on merkitsevä. Monikoista voidaan purkaa arvoja *pattern matching* -tekniikan avulla.

```
person :: (String, Int)
person = (“Jennifer”, 28)
```

Monikot toimivat luontevana osana funktionaalista ohjelmointia. Niitä voidaan käyttää joko sellaisenaan esittämään yksinkertaisia tietorakenteita tai toteuttamaan monimutkaisempia tietorakenteita kuten listoja. Monipuolisuudestaan huolimatta monikot ovet erittäin kevyitä luoda ja käyttää.

3.3.9 Tietue

Tietue pitää sisällään muita tietotyyppejä nimetyissä kentissä. Tietueen tyypin määrää sen sisältämien kenttien nimet ja tyypit.

Oheisessa esimerkissä tietuetta on käytetty määrittelemään henkilötiedot sisältävä tietotyyppi. Yksittäisen kentän arvo on mahdollista saada joko funktiolla tai malliin sovittamalla.

```
data Person = Person { name :: String, age :: Int }
person = Person “Jennifer” 28
```

Tietuetta voidaan käyttää funktionaalisessa ohjelmoinnissa esitettäessä yhteenkuuluvia tietoja. Sen etu verrattuna monikkoon on nimetyt kentät, mutta toisaalta sen käyttö on hieman raskaampaa.

3.3.10 Algebrallinen tietotyyppi

Algebrallinen tietotyyppi määrittelee joukon rakentajia, joilla voidaan luoda vaihtoehtoisia arvoja. Vasta rakentajia kutsumalla saadaan luotua tietotyyppiä vastaavia arvoja.

Oheisessa esimerkissä on esitetty algebrallinen tietotyyppi *Option*. Sillä on kaksi rakentajaa, joista toinen ottaa parametrinaan kokonaisluvun. Rakentajia kutsumalla luodaan kaksi arvoa, *ok* sekä *fail*.

```
data Option = Some Int | None

let ok = Some 5
let fail = None
```

Luettelotyyppi on algebrallisen tietotyypin erityistapaus. Sen kaikki rakentajat ovat parametrittomia. Oheisen esimerkin *Color* on luettelotyyppi.

```
data Color = Orange | White | Blue
```

Moni rakenne on mielekästä kuvata rekursiivisesti. Alla olevan esimerkin algebrallinen tietotyyppi *Tree* määrittelee puurakenteen rekursiivisesti. Puu voi olla joko yksittäinen lehti tai haara, jossa on kaksi alipuuta. Alipuut voivat taas olla joko lehtiä tai uusia haaroja.

```
data Tree = Leaf Int | Node Tree Tree
tree = Node (Leaf 1) (Node (Leaf 2) (Leaf 3))
```

3.3.11 Lista

Lista on joukko samantyyppisiä arvoja järjestettynä peräkkäin. Listan tyypin määrittää sen alkion tyyppi. Eri pituiset listat, jotka sisältävät saman tyyppisiä alkioita, ovat samaa tyyppiä.

Seuraavassa esimerkissä on kaksi eri tyyppistä listaa. Ensimmäinen listoista – *primes* – on tyypiltään kokonaislukulista, ja se sisältää kymmenen ensimmäistä alkulukua. Toinen on puolestaan liukulukulista, joka ei sisällä lainkaan arvoja. Myös tyhjällä listalla on tyyppi.

```
primes :: [Int]
primes = [ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ]

probabilities :: [Float]
probabilities = []
```

Listojen merkitys funktionaalisessa ohjelmoinnissa on merkittävä. Monet ongelmat voidaan ratkaista funktionaalisesti muuntamalla, suodattamalla tai yhdistelemällä listan arvoja. Huomionarvoista on, että lista voidaan toteuttaa monikon sekä rekursiivisen algebrallisen tietotyypin avulla.

3.3.12 Mallin sovittaminen

Mallin sovittaminen (*pattern matching*) on primitiivisen tai koostetun tiedon testaamista valittua mallia vasten. Sen avulla on mahdollista valita suoritettava koodi sekä purkaa tieto primitiivisiksi arvoiksi, kuten oheisessa koodissa havainnollistetaan.

```
let (name, age) = (”Jennifer”, 28) -- name = “Jennifer”, age = 28
let (Some a) = Some 13             -- a = 13
let a:b::tail = [ 1, 2, 3, 4 ]     -- a = 1, b = 2, tail = [ 3, 4 ]
```

Monikot, algebralliset tietotyypit ja listat on mahdollista purkaa osiin yksinkertaisella mallin sovittamisella. Mikäli annettu tieto ei vastaa mallia, syntyy virhetilanne.

Funktion rungon valinta on toinen tärkeä käyttökohde mallin sovittamiselle. Esimerkissä funktion *message* runko valitaan parametrina annetun luvun perusteella.

```
message :: Int -> String
message 0 = ”zero is nothing”
message 1 = ”one is something”
message a = ”what a number ” ++ (show a) ++ “ is”
```

Mallin sovittaminen on yksi tärkeimmistä konsepteista funktionaalisessa ohjelmoinnissa. Sen avulla voidaan toteuttaa muun muassa loppuehto rekursiolle sekä monipuolisia ehtolausekkeita. Esimerkkien tekniikat voidaan myös yhdistää, jolloin kompleksinen arvo pilkotaan osiin samalla kun osat välitetään tietylle funktion rungolle.

3.3.13 Laiska suorittaminen

Laiska suorittaminen on lausekkeen laskemista vasta, kun sen arvoa tarvitaan. Jos lausekkeen arvoa ei tarvita ohjelman suorituksen aikana, sitä ei lasketa lainkaan. Sen avulla on mahdollista luoda muun muassa äärettömiä tietorakenteita. Toteutustavasta riippuen se voi myös tehostaa ohjelman suoritusta välttämällä tarpeettomia laskutoimituksia.

Oheisen esimerkin lauseke luo loputtoman listan parittomia kokonaislukuja. Ymmärrettävistä syistä lauseke on mahdollinen vain laiskan suorittamisen ansiosta. Arvon *odds* luvut lasketaan sitä mukaan, kun niitä käytetään. Tämä on havainnollistettu toisella rivillä, joka pyytää 10 ensimmäistä paritonta lukua loputtomalta listalta. Huomioitavaa on, ettei myöskään sitä suoriteta ennen kuin arvolle *some* on tarvetta.

```
let odds = filter (\x -> x `mod` 2 == 0) [1..]
let some = take 10 odds
```

3.4 Vertailu olioperustaiseen ohjelmointiin

Funktionaalinen ja olioperustainen ohjelmointityyli poikkeavat toisistaan suuresti. Osittain niitä voidaan pitää toistensa vastakohtina. Selkein ero niiden välillä muodostuu käytetystä rakennuspalikasta, mutta myös muita merkittäviä eroavaisuuksia löytyy. Taulukossa 4Taulukko 4 on listattu tärkeimmät niistä.

	Funktionaalinen	**Olioperustainen**
Ylälaji	Deklaratiivinen	Imperatiivinen
Rakennuspalikka	Funktio	Olio
Arvot	Vakioita	Muuttuvia
Sivuvaikutukset	Ei sallittu	Sallittu

Taulukko 4. Funktionaalisen ja olioperustaisen ohjelmoinnin erot

Funktionaalinen ohjelmointityyli painottaa funktioiden käyttöä ohjelman rakennuspalikoina. Siinä funktioiden luonti ja monipuolinen yhdisteleminen ovat avainasemassa. Olioperustaisessa ohjelmoinnissa puolestaan oliot ovat keskiössä, ja niiden käsittelyyn on tarjolla monipuolisia mekanismeja. Ohjelmointityylien erojen ymmärtämiseksi on syytä tutkia funktioiden ja olioiden välisiä eroavaisuuksia, joista oleellisimmat on esitetty taulukossa 5.

	Funktio	**Olio**
Toiminnallisuus	Kyllä	Kyllä
Data	Ei	Kyllä
Tila	Ei	Kyllä
Tehtävä	Arvojen muuntaminen	Kapseloida yhteenkuuluva data ja toiminnallisuus

Taulukko 5. Funktion ja olion väliset erot

Funktionaalisessa ohjelmoinnissa funktiot ja data ovat selkeästi erillisiä kokonaisuuksia. Oliot puolestaan kapseloivat yhteenkuuluvan toiminnallisuuden ja datan. Siinä missä funktio vain muuntaa saamansa parametrin arvosta toiseen, ylläpitää olio sisäistä tilaa. Tämän eroavaisuuden pohjalta myös funktionaalisen ja olioperustaisen ohjelman rakenteet poikkeavat toisistaan suuresti. Funktionaalinen ohjelma on pitkälti arvojen muuntamista muodosta toiseen. Olioperustainen ohjelma puolestaan tilan ylläpitämistä olioissa sekä niiden välistä vuorovaikutusta.

Perustavaa laatua olevien erojen takia olioperustaisen ohjelmoinnin käsitteille ei useinkaan ole suoraa vastinetta funktionaalisessa ohjelmoinnissa. Osalle olioperustaisista ominaisuuksista voidaan löytää vastineita funktionaalisesta ohjelmoinnista, mutta osa on täysin spesifisiä kyseiselle ohjelmointityylille. Ensin mainittuun kategoriaan voidaan laskea muun muassa rajapinta sekä koostaminen ja jälkimmäiseen perintä. Nämä mekanismit ovat käytännössä osa jokaista olioperustaista suunnittelumallia, jonka takia niiden suoraviivainen muuntaminen olioperustaisesta funktionaalisiksi ei ole useinkaan mahdollista.

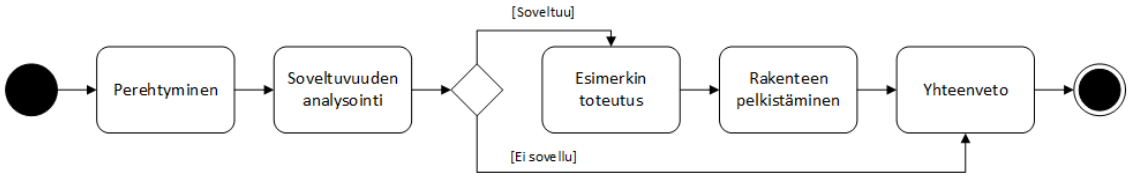
4. TUTKIMUSMENETELMÄ

Tutkimusmenetelmänä on tarkastella jokaista *Design Patterns* -kirjassa esitettyä suunnittelumallia funktionaalisessa asiayhteydessä. Suunnittelumallien soveltuvuus funktionaaliseen ohjelmointiin arvioidaan toteutettavan esimerkin perusteella. Aina tämä ei kuitenkaan ole mahdollista johtuen olioperustaisen ja funktionaalisen ohjelmoinnin eroista.

Tässä luvussa esitetään ensin yleiskuvaus analysointiprosessista. Tämän jälkeen sen eri vaiheet käydään läpi yksityiskohtaisemmin. Lisäksi määritellään kriteerit, joilla suunnittelumalli luokitellaan funktionaalisessa asiayhteydessä.

4.1 Suunnittelumallin analysointiprosessi

Suunnittelumallin analysointi funktionaalisen ohjelmoinnin asiayhteydessä on monivaiheinen prosessi. Tavoitteena on selvittää suunnittelumallin soveltuvuus ja rakenne funktionaalisessa kontekstissa yhteenvetoa varten. Kuva 2 havainnollistaa tätä prosessia.



Kuva 2. Suunnittelumallin analysointiprosessi

Perehtyminen suunnittelumalliin on ensimmäinen vaihe sen analysoinnissa. Perehtymällä *Design Patterns* -kirjassa esitettyyn lukuun suunnittelumallista – erityisesti sen tiivistettyyn tarkoitukseen, rakenteeseen sekä käyttökohteisiin – ymmärretään sen tarkoitus olioperustaisessa ohjelmoinnissa.

Seuraava vaihe on soveltuvuuden analysointi. Jos suunnittelumallin tarkoitus on yleistettävissä olioparadigmasta riippumattomaan muotoon, on se todennäköisesti sovellettavissa myös funktionaalisessa ohjelmoinnissa. Soveltuvuutta pohditaan esimerkein sen käyttökohteista.

Riippuen edellisen vaiheen tuloksesta edetään joko esimerkin toteutukseen tai suoraan yhteenvetoon. Esimerkiksi valitaan yksinkertainen mutta toimiva ohjelma. Esimerkin keskiössä on suunnittelumallin soveltaminen käytäntöön. Valmiin esimerkin koodi analysoidaan ja pelkistetään yleiskäyttöiseen muotoon.

Yhteenvedossa suunnittelumalli luokitellaan sen sovellettavuuden sekä pelkistetyn rakenteen perusteella. Luonnollisesti rakennetta ei analysoida, mikäli suunnittelumallia ei pystytä soveltamaan funktionaaliseen ohjelmointiin.

4.2 Perehtyminen

Perehtyminen on ensimmäinen vaihe suunnittelumallin analysointiprosessissa. Lähteenä toimii suunnittelumallia vastaava *Design Patterns* -kirjan luku. Luvussa kuvataan muun muassa suunnittelumallin tarkoitus (*intent*), motivaatio (*motivation*), sovellettavuus (*applicability*) sekä rakenne (*structure*). Kirjan lisäksi muita mahdollisia lähdemateriaaleja ei huomioida.

Suunnittelumallin alkuperäinen tarkoitus toimii tärkeimpänä perustana analyysille. Se tiivistää suunnittelumallin idean lyhyesti muutamalla lauseella. Esimerkiksi Adapterin tarkoitus on kuvattu seuraavasti.

”*Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.*” (Gamma 2000 s. 139)

Olioperustaisen ja funktionaalisen ohjelmoinnin eroista johtuen kuvausta ei useinkaan ole mielekästä ottaa kirjaimellisesti. Kuvaus saattaa sisältää vahvasti olioperustaiseen ohjelmointiin liittyviä käsitteitä, kuten *olio*, *luokka* tai *tila*. Kirjaimellisen tulkinnan sijaan pyritään ymmärtämään taustalla oleva ajatus.

Suunnittelumallin motivaatio, sovellettavuus sekä rakenne toimivat täydentävinä tekijöinä pohdittaessa suunnittelumallin tarkoitusta olioperustaisen asiayhteyden ulkopuolella. Vaikka esimerkiksi suunnittelumallista esitetty luokkakaavio liittyy vahvasti olioperustaiseen ohjelmointiin, auttaa se usein ymmärtämään suunnittelumallin tarkoitusta paremmin.

4.3 Soveltuvuuden analysointi

Soveltuvuuden analysointi on seuraava vaihe suunnittelumallin analysointiprosessissa. Se pohjautuu ensimmäisessä vaiheessa kerätylle tiedolle suunnittelumallin tarkoituksesta olioperustaisen ohjelmoinnin ulkopuolella. Tavoitteena on selvittää, voidaanko suunnittelumallin ideaa soveltaa funktionaalisessa asiayhteydessä.

Soveltuvuus selvitetään pohtimalla esimerkkitapauksia, joissa suunnittelumalli esiintyy osana puhtaasti funktionaalista ohjelmaa. Mikäli esimerkkitapauksia löytyy, valitaan niistä yksi ja siirrytään esimerkin toteutusvaiheeseen. Muussa tapauksessa siirrytään suorittamaan yhteenveto.

4.4 Esimerkin toteutus

Suunnittelumallin soveltamiseksi suunnitellaan yksinkertainen esimerkkiohjelma. Tarkoituksena on havainnoida, millaisen rakenteen suunnittelumalli saa funktionaalisessa

asiayhteydessä. Suunnittelumalli pidetään esimerkkiohjelman pääosassa ja muu rakenne karsitaan mahdollisimman vähäiseksi. Tästä huolimatta esimerkkiohjelmasta pyritään laatimaan oikeata ohjelmaa muistuttava kuten vaikkapa pinta-aloja laskeva ohjelma tai pakkausalgoritmi.

Suunnitelma toteutetaan lopuksi Haskellilla. Esimerkkiohjelmat ovat kaksiosaisia. Suunnittelumallia sovelletaan puhtaassa funktionaalisessa kontekstissa. Sen tulokset saatetaan kuitenkin tulostaa main-funktiossa konsoliin I/O-operaatiolla. Huomioitavaa on, ettei tulostaminen ole suunnittelumallin kannalta olennainen asia. Se ainoastaan helpottaa havainnollistamaan esimerkkiohjelman suoritusta.

4.5 Rakenteen pelkistäminen

Pelkistämisellä pyritään löytämään suunnittelumallin rakenteellinen ydin. Siinä esimerkin rakenne karsitaan siten, että ainoastaan suunnittelumallin toiminnan kannalta oleelliset seikat jätetään jäljelle. Tämä auttaa havainnollistamaan, millaisen muodon suunnittelumalli saa funktionaalisessa asiayhteydessä.

Rakenne esitetään abstraktina Haskelliin pohjautuvana pseudokoodina. Pelkistetyn rakenteen lisäksi suunnittelumallin toiminta kuvataan lyhyesti. Tämä vastaa pitkälti kirjassa esitettyjä kohtia rakenne (*structure*), osallistujat (*participants*) ja yhteistyö (*collaborations*).

4.6 Yhteenveto

Viimeinen vaihe analysointiprosessissa on yhteenvedon suorittaminen. Siinä suunnittelumalli luokitellaan aiempien kohtien havaintojen perusteella. Suunnittelumalli saa aina luokituksen sen soveltuvuuden mukaan. Sovellettavissa olevasta suunnittelumallista esitetään myös sen pelkistetty rakenne sekä arvioidaan sen kompleksisuus. Kompleksisuus perustuu suunnittelumallin pelkistettyyn rakenteeseen.

SOVELTUVUUS

Asteikko on kolmiportainen: *luonnollinen*, *teennäinen* ja *soveltumaton*.

Soveltumaton soveltuvuus tarkoittaa, ettei suunnittelumallin tarkoitus ole mielekäs puhtaasti funktionaalisessa asiayhteydessä. Suunnittelumalleja, joiden soveltuvuus on soveltumaton, ei ole myöskään toteutettu esimerkkeinä.

Teennäinen soveltuvuus tarkoittaa, että suunnittelumallin tarkoitus on sovellettavissa puhtaasti funktionaalisessa asiayhteydessä. Tällainen suunnittelumalli tuntuu kuitenkin teennäiseltä, eikä tuo lisäarvoa funktionaaliseen suunnitteluun.

Luonnollinen soveltuvuus tarkoittaa, että suunnittelumalli soveltuu hyvin puhtaaseen funktionaaliseen ohjelmointiin. Sen alkuperäinen tarkoitus säilyy, ja sen käyttö tuntuu luonnolliselta osana funktionaalista ohjelmaa.

KOMPLEKSISUUS

Asteikko on kolmiportainen: *triviaali*, *normaali* ja *monimutkainen*.

Triviaali suunnittelumalli on pelkistynyt funktionaalisessa ohjelmoinnissa perusoperaatioiksi, kuten osittain sovellettavaksi funktioksi tai funktioiden kompositioksi. Pelkistynyt rakenne on niin yksinkertainen, ettei sitä voida mieltää suunnittelumalliksi.

Normaali kompleksisuus tarkoittaa, että suunnittelumalli käyttää useita funktionaalisen ohjelmoinnin ominaisuuksia yhdessä. Tällainen tapaus voi olla esimerkiksi rekursiivisen tietorakenteen sekä sitä selaavan algoritmin yhteistyö.

Monimutkainen termi on varattu suunnittelumalleille, joiden toteuttaminen vaatii erityisiä funktionaalisia tekniikoita. Tällaisia voivat olla muun muassa, *lense*, *memoization* tai *zipper*. Sitä käytetään myös, jos suunnittelumallin rakenne on huomattavan monimutkainen.

5. TAPAUSTUTKIMUKSET

Tässä luvussa sovelletaan tutkimusmenetelmää neljään erityyliseen suunnittelumalliin. Tarkoituksena on havainnollistaa yksityiskohtaisesti vaihe vaiheelta, miten työn tulokset on johdettu.

Valitut suunnittelumallit ovat *Template Method*, *Composite*, *Builder* sekä *Observer*. Valituista suunnittelumalleista ensimmäinen soveltuu erinomaisesti funktionaaliseen ohjelmointiin, ja sen rakenne pelkistyy triviaaliksi. Kaksi seuraavaa suunnittelumallia ovat hieman monimutkaisempia sovellettavia. *Observer* edustaa suunnittelumalleja, jotka eivät sovellu lainkaan funktionaaliseen ohjelmointiin.

Suunnittelumallin kohdalla esitetään ensin lähtötiedot, jotka perustuvat *Design Patterns* -kirjan suunnittelumallia vastaavaan lukuun. Seuraavaksi pohditaan, onko kyseisen suunnittelumallin tarkoitus mielekäs funktionaalisessa ohjelmoinnissa. Jos on, siirrytään toteuttamaan esimerkkiohjelma, jossa suunnittelumalli on käytössä. Lopuksi analysointiprosessista suoritetaan yhteenveto.

5.1 Template Method

Template Method on yksinkertainen suunnittelumalli, jonka käyttötarkoitus on selkeä. Se soveltuu erinomaisesti ensimmäiseksi esimerkiksi tutkimusmenetelmän havainnollistamiseksi, sillä sen muuntaminen funktionaaliseen muotoon on yksinkertainen prosessi.

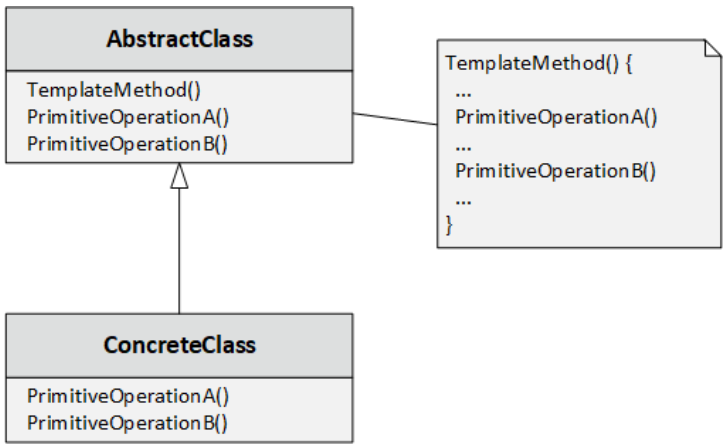
Luokittelu
Käyttäytymismalli

Tarkoitus
Määritellä algoritmin runko kantaluokassa, mutta antaa perityn luokan toteuttaa yhden tai useamman sen osavaiheista

Soveltuvuus
Mallia voidaan käyttää, jos jokin seuraavista kohdista toteutuu

- Halutaan toteuttaa algoritmin muuttumaton osa kertaalleen ja jätetään muuttuva toiminnallisuus toteutettavaksi aliluokkiin.
- Halutaan refaktoroida yhteinen koodi aliluokista yhteiseen kantaluokkaan koodin monistumisen välttämiseksi.
- Halutaan kontrolloida aliluokkien laajennettavuutta. Suunnittelumallin avulla voidaan valita kohdat, joihin aliluokat voivat vaikuttaa.

Rakenne



Kuva 3. Template Method -suunnittelumallin luokkakaavio

Toiminta

AbstractClass määrittää algoritmin rungon funktiossa *TemplateMethod()*. Kyseinen funktio puolestaan kutsuu virtuaalisia funktioita *PrimitiveOperation1()* ja *PrimitiveOperation2()* osavaiheidensa suorittamiseksi. *ConcreteClass* määrittää algoritmin toiminnan toteuttamalla kantaluokan virtuaaliset funktiot

5.1.1 Soveltuvuus funktionaaliseen ohjelmointiin

Suunnittelumallin tarkoitus on selkeästi esitetty, mutta siinä on viittauksia olioperustaisen ohjelmoinnin käsitteisiin. Avainasemassa vaikuttaa kuitenkin olevan algoritmin rungon ja konkreettisten osavaiheiden erottaminen. Algoritmit ovat hyödyllinen käsite myös ohjelmoinnin ulkopuolella, joten ne ovat selkeästi riippumattomia myös ohjelmointityylistä.

Algoritmin vaiheiden erottamiselle on helppo keksiä useita käyttötarkoituksia. Paketointialgoritmin runko voi olla otsikkotiedon lisääminen, varsinaisen tiedon pakkaaminen ja lopuksi tarkistusluvun asettaminen. Jos algoritmin eri vaiheet parametrisoidaan, saadaan rungosta abstrakti. Tällä tavoin samaa runkoa voidaan käyttää esimerkiksi kuvatiedoston häviölliseen tai tekstitiedoston häviöttömään pakkaukseen.

Esitetyn perusteella määritellään suunnittelumallin tarkoitus yleisemmällä tasolla seuraavasti. ”*Määritetään algoritmin runko, mutta mahdollistetaan sen osavaiheiden helppo muuttaminen.*” Suunnittelumallin tarkoitus soveltuu selkeästi myös funktionaaliseen ohjelmointiin, joten seuraavaksi toteutetaan esimerkkiohjelma.

5.1.2 Paketointialgoritmi

Suunnittelumallin sovelluskohteeksi valitaan datan paketointi. Sovelluskohteita sille ovat esimerkiksi kuvien, tekstin ja äänen paketointi. Paketoinnin yhteydessä on mahdollista

tiivistää dataa joko häviöllisesti tai häviöttömästi. Algoritmin halutaan määrittävän paketoinnin osavaiheet sekä paketin formaatti. Sen sijaan tiivistysalgoritmi sekä tarkistusluvun laskeminen halutaan jättää helposti muunneltaviksi. Paketin formaatiksi valitaan seuraava:

```
-- compression format | checksum format | checksum | <compressed data>
```

Paketin alussa on kolmen kentän mittainen otsikkotieto. Siinä on merkittynä sekä datan tiivistämiseen että tarkistusluvun laskemiseen käytetty algoritmi, joten sen avulla paketti voidaan myöhemmin helposti purkaa. Lisäksi tarkistusluku mahdollistaa puretun tiedon eheyden varmistamisen. Varsinainen pakattu tieto on paketin lopussa. Eri kentät erotellaan pystypalkeilla.

Funktionaalisessa ohjelmoinnissa tiivistysalgoritmi on luontevaa esittää funktiona, joka ottaa datan parametrinaan ja palauttaa käsitellyn datan paluuarvonaan. Riippumatta millä tavoin tiivistysalgoritmi datan tiivistää, vai tiivistääkö lainkaan, sen tyyppi on seuraava:

```
type CompressionFunction = [Char] -> [Char]
```

Datan esitysmuodoksi valitaan merkkijonot, jotka saavat mallintaa tavujonoja. Tarkistusluvun laskentaan käytettävä algoritmi ottaa puolestaan parametrinaan alkuperäisen datan, ja palauttaa sitä vastaavan tarkistussumman. Tässä tapauksessa tarkistussummaksi on valittu kokonaisluku.

```
type ChecksumFunction = [Char] -> Int
```

Funktioiden tyyppien määrittely ei ole välttämätöntä, mutta se auttaa hahmottamaan, miten niitä on tarkoitus käyttää. Toistaiseksi varsinaista toteutusta näille funktioille ei anneta. Riittää, että niiden tyypit ovat tiedossa.

Seuraavaksi tarkastelun kohteena on varsinainen paketointitoiminnallisuus. Se on yksinkertaisimmillaan funktio, joka ottaa paketoitavan tiedon parametrinaan ja antaa paluuarvonaan paketoidun tiedon. Oheinen esimerkki havainnollistaa asian.

```
deflate :: CompressionFunction
deflate input = ...

crc32 :: ChecksumFunction
crc32 input = ...

staticPacker :: [Char] -> [Char]
staticPacker input = do
    -- use compression and checksum functions from the closure
    let compressed = deflate input
    let checksum = crc32 input

    -- merge partial results to create output packet
    “deflate | crc32 | “ ++ (show checksum) ++ “|” ++ compressed
```

Listaus esittää, miten paketointifunktio *staticPacker* voidaan toteuttaa sulkeuman avulla. Se käyttää rungossaan kahta ulkopuolista funktiota, *deflate* ja *crc32*. Ongelma toteutuksessa on kuitenkin, ettei sen toiminnan muuttaminen ole helppoa. Jos halutaan paketoida tietoa pakkaamatta, mutta käyttäen yhä *crc32* tarkistuslaskentaa, pitää koko funktio toteuttaa uudestaan. Selkeästikään tämä ei ole suunnittelumallin mukainen ratkaisu.

Jotta paketoinnin suorittavan funktion toimintaa on mahdollista muuttaa helposti, tulee sen ottaa osavaiheisiin käyttämänsä funktiot parametreina. Funktio omaa yhä saman rungon, mutta sen toiminta on helposti muutettavissa.

```
packerTemplate :: CompressionFunction -> String ->
                  ChecksumFunction -> String ->
                  [Char] -> [Char]
packerTemplate compressionFunction cpTag checksumFunction csTag input = do
    -- use the functions given as parameters
    let compressed = compressionFunction input
    let checksum = checksumFunction input

    -- merge partial results to create output packet
    cpTag ++ "|" ++ csTag ++ "|" ++ (show checksum) ++ "|" ++ compressed
```

Oheisessa toteutuksessa funktion *packTemplate* toimintaa on vihdoin helppo muuttaa. Sille voidaan antaa parametrina haluttu tiivistysalgoritmi sekä tarkistussumman laskeva funktio. Lisäksi otsikkotietoon kirjoitettavat pakkausalgoritmin ja tarkistussumman laskemiseen käytetyn algoritmin tiedot ovat nyt dynaamisesti parametrisoitavissa. Se on valmis kutsuttavaksi muusta koodista.

```
clientFunction ... = do
    let packer = packerTemplate id "none" crc32 "crc32"
    ...

    -- pack some data with the concrete packer
    let packet = packer someData
```

Asiakasfunktio soveltaa osittain *packerTemplatea* haluamillaan parametreilla. Tässä tapauksessa luodaan funktio, joka ei tiivistä dataa lainkaan, mutta laskee sille crc32-tarkistusluvun. Suunnittelumallin tarkoitus on saatu siirrettyä onnistuneesti puhtaasti funktionaaliseen kontekstiin.

5.1.3 Yhteenveto

Analysoitaessa suunnittelumallia puretaan ensin sen rakenne konkreettisesta esimerkistä yleisempään muotoon. *Template Method* on funktionaalisessa asiayhteydessä korkeamman tason funktio, joka ottaa parametrinaan yhden tai useamman funktion osavaiheidensa suorittamiseksi. Soveltamalla osittain siitä saadaan tehtyä asiakasfunktion käyttötarpeisiin soveltuva. Oheinen listaus havainnollistaa abstraktin rakenteen.

```
type PrimitiveOperationA = ...
```

```
type PrimitiveOperationB = ...

algorithmTemplate :: PrimitiveOperationA -> PrimitiveOperationB -> ...

...

clientFunction ... =
    ...
    let algorithm = algorithmTemplate operationA operationB
    -- usage of the concrete algorithm
```

Suunnittelumallin vaatimat funktionaaliset ominaisuudet ovat korkeamman tason funktiot sekä soveltaminen osittain. Kyseiset ominaisuudet ovat perusominaisuuksia funktionaalisessa ohjelmoinnissa. Sen rakenne on myös pelkistynyt erittäin yksinkertaiseksi, joten se luokitellaan kompleksisuudeltaan *triviaaliksi*. Soveltuvuus funktionaaliseen ohjelmointiin on *luonnollinen*.

5.2 Composite

Composite toimii esimerkkinä rakennemallista. Se on melko hyvin tunnettu suunnittelumalli, sillä esimerkiksi monet käyttöliittymäkirjastot perustuvat sen käyttöön. Sen toteuttaminen vaatii pelkkien funktioiden lisäksi myös yhteensopivia tietotyyppejä.

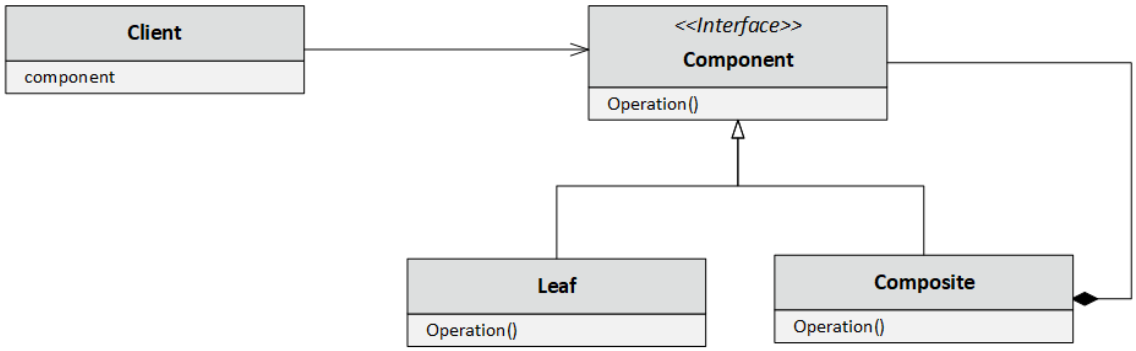
Luokittelu
Rakennemalli

Tarkoitus
Koostaa olioista puurakenteita esittämään osa-kokonaisuus-hierarkioita. *Composite* mahdollistaa yksittäisen osan ja kokonaisuuden käsittelyn yhtenäisesti.

Soveltuvuus
Mallia voidaan käyttää

- esittämään osa-kokonaisuus-hierarkioita
- piilottamaan, käytetäänkö yksittäistä osaa vai useampaa.

Rakenne



Kuva 4. Composite-suunnittelumallin luokkakaavio

Toiminta

Composite piilottaa puumaisen tietorakenteen yhtenäisen rajapinnan taakse. *Client* ei tämän ansiosta ole tietoinen, käyttääkö se yksittäistä komponenttia vai joukkoa niitä. Puumaisen tietorakenteen mallinnus on saatu käyttämällä rekursiivista luokkahierarkiaa, jossa aliluokka viittaa kantaluokkaansa.

5.2.1 Soveltuvuus funktionaaliseen ohjelmointiin

Suunnitelumallilla on kaksi toisiinsa liittyvää teemaa. Sen käyttökohteet ja esimerkit käsittelevät tavalla tai toisella puurakenteiden mallintamista. Toinen esille nouseva seikka on, miten sen avulla saadaan piilotettua, koostuuko kyseinen tietorakenne yhdestä vai useammasta osasta. Molemmat käsitteet ovat hyödyllisiä myös olioperustaisen ohjelmoinnin ulkopuolella. Tutkitaan seuraavaksi esimerkkiä, joka yhdistää molemmat teemat.

Auto koostuu useasta eri osasta, kuten vaihdelaatikosta ja moottorista. Osat koostuvat puolestaan useasta pienemmästä osasta, kuten akseleista, pulteista ja laakereista. Auto voidaankin mallintaa puurakenteena. Yksittäisten osien sijaan se kannattaa usein käsittää yhtenä kokonaisuutena. *Composite* soveltuu myös monien muiden osa-kokonaisuus-tietorakenteiden käsittelyyn.

Suunnittelumallin alkuperäinen tarkoituksen kuvaus on riittävä. Sen puitteissa esiintyvä olio ymmärretään sanan laajemmassa – ei olio-ohjelmointiin liittyvässä – merkityksessä. ”*Yhdenmukaistaa yksittäisen olion ja oliojoukon käsittely. Oliojoukko esitetään puurakenteena.*”

5.2.2 Vektorigrafiikkaohjelma

Esimerkkiohjelmaksi valitaan vektorigrafiikkaohjelma. Siinä esitetään yksittäisiä objekteja, kuten ympyröitä ja neliöitä, sekä niiden muodostamia joukkoja, eli koosteita. Yksittäistä objektia ja koostetta halutaan käsitellä yhdenmukaisesti. Ohjelman tulee osata sekä siirtää ja skaalata objekteja, että laskea niiden pinta-aloja.

```
scale :: Shape -> Float -> Shape
move :: Shape -> Float -> Float -> Shape
area :: Shape -> Float
```

Funktioiden tyypit on listattu yllä. Yksittäisen objektin tai koosteen skaalaaminen ja siirtäminen palauttavat uuden objektin tai koosteen, jossa vain koko tai sijainti on muuttunut. Pinta-alan laskeminen puolestaan palauttaa pinta-alaa vastaavan liukuluvun.

Rajapinta on yhtenäinen sekä yhdellä objektille että koosteella, koska funktioiden tyypit eivät salli muuta. Ongelmaksi kuitenkin muodostuu, mitä *Shape* itseasiassa on? Sen tulisi mallintaa sekä yksittäistä objektia että puurakennetta. Luonteva rakenne ongelman ratkaisuksi on kohdassa 3.3.10 esitelty algebrallinen tietotyyppi. Se määrittää yhtenäisen tyypin sekä ympyrälle, neliölle että koosteelle.

```
type Position = (Float, Float)
type Radius = Float
type Size = Float

data Shape =
    Circle Position Radius | Square Position Size | Composite [Shape]
```

Shape on nyt puumainen rakenne. Sen sisältämä vaihtoehto *Composite* viittaa rekursiivisesti tyyppiin *Shape*. Aiemmin määritellyt funktiot – *scale*, *move* ja *area* – ovat nyt pakotettu käsittelemään sekä ympyröitä, neliöitä että koosteita.

```
scale (Circle position radius) factor =
    Circle position (radius * factor)
scale (Square position size) factor =
    Square position (size * factor)
scale (Composite shapes) =
    Composite (map (\s -> scale s factor) shapes)
```

Funktio käsittelee kaikki vaihtoehdot valiten oikean funktiorungon mallin sovituksen avulla. Kaksi muuta funktiota toteutetaan vastaavalla tavalla. Oleellinen kohta funktioissa on *Compositen* käsittely, joka kutsuu rekursiivisesti funktiota jokaiselle koosteen objektille.

5.2.3 Yhteenveto

Composite on hyödyllinen käsite myös funktionaalisessa ohjelmoinnissa. Sen esittämä puurakenne toteutuu luontevasti algebrallisen tietotyypin avulla. Toiminnallisuus erotetaan funktionaalisen ohjelmoinnin periaatteiden mukaan omiksi funktioikseen. Oleellista on, että kyseiset funktiot tukevat kaikkia algebrallisen tietotyypin vaihtoehtoja. Pelkistettynä rakenne näyttää seuraavalta.

```
data Component = Leaf X | Composite [Component]

operation1 :: Shape -> b
operation1 (Leaf x) = ...
operation1 (Composite components) =
    -- perform operation with fold

operation2 :: Shape -> Shape
operation (Leaf x) = ...
operation2 :: (Composite components) =
    -- perform operation with map
```

Riippuen funktion *operation* tyypistä sen vaihtoehtoa *Composite* käsitellään eri listaoperaatioilla. Tietorakenteen muuttamiseen soveltuu funktio *map* ja siitä johdetun arvon laskemiseen käy puolestaan funktio *fold*. Myös muut funktiot ovat käyttökelpoisia.

Composite soveltuu erinomaisesti myös funktionaaliseen ohjelmointiin. Sen toteuttaminen vaatii useampaa funktionaalisen kielen ominaisuutta ja niiden yhteistyötä. Suunnittelumalli saa soveltuvuudesta luokituksen *luonnollinen* ja rakenteensa kompleksisuudesta *normaali*.

5.3 Builder

Builder on monimutkaisten olioiden luomiseen tarkoitettu suunnittelumalli. Se havainnollistaa, miten olioiden luominen on verrattavissa funktionaalisten tietorakenteiden muodostamiseen.

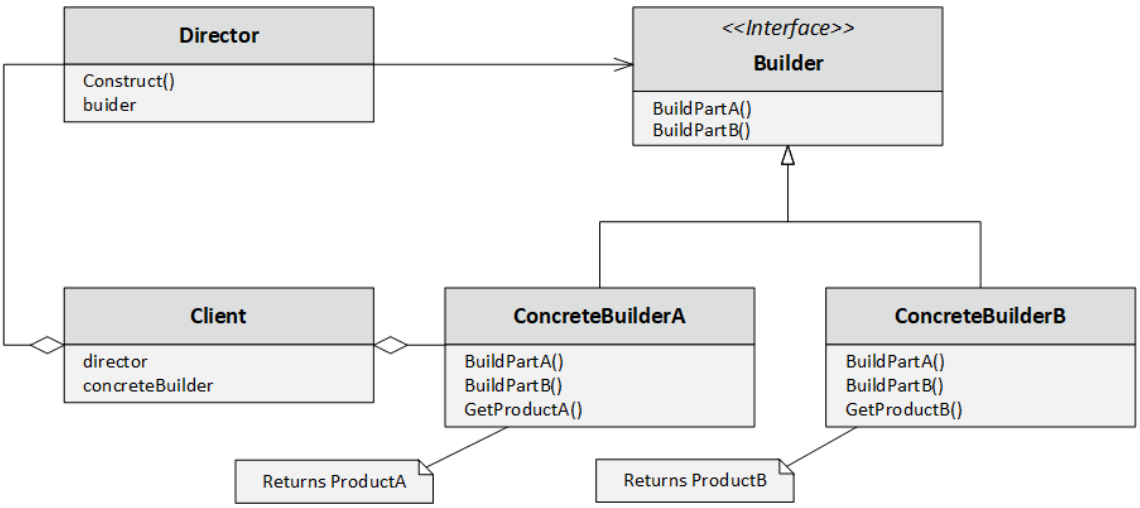
Luokittelu
Luontimalli

Tarkoitus
Erottaa monimutkaisen olion luontiprosessi sen rakenteesta siten, että sama luontiprosessi voi tuottaa erilaisia rakenteita.

Soveltuvuus

- Monimutkaisen olion rakentamiseen käytetyn algoritmin tulee olla riippumaton osista, joista olio koostuu ja miten ne yhdistyvät
- Algoritmilla tulee voida tuottaa erilaisia esityksiä rakennettavasta oliosta

Rakenne



Kuva 5. Builder-suunnittelumallin luokkakaavio

Toiminta

Builder määrittää rajapinnan, jonka avulla voidaan rakentaa monimutkainen olio. Siitä perityt konkreettiset rakentajat luokat (*ConcreteBuilderA* ja *ConcreteBuilderB*) toteuttavat rajapinnan haluamallaan tavalla. Rakennusprosessi on vaiheittainen, joten konkreettisen rakentajan täytyy ylläpitää rakennusvaiheen tilaa.

Client luo konkreettisen rakentajan ja välittää sen oliolle *Director*. Tämä puolestaan määrittää varsinaisen algoritmin, jolla rakennettava olio luodaan. *Director* käyttää konkreettista rakentajaa rajapinnan *Builder* kautta, joten se ei ole tietoinen konkreettisista rakentajista eikä niiden tuotteista. *Client* saa rakennetun tuotteen käyttöön konkreettisen rakentajan tarjoaman spesifisen funktion avulla.

5.3.1 Soveltuvuus funktionaaliseen ohjelmointiin

Suunnittelumallin teema liittyy monimutkaisten tietorakenteiden luomiseen. Tietoa ei synny tyhjästä, joten pohjimmiltaan se vain muuntaa tiedon yhdestä esitysmuodosta toiseen. Puhdas funktionaalinen ohjelma muuntaa saamansa argumentin paluuarvoksi. Tässä mielessä se muistuttaa paljolti *Builder*-suunnittelumallia. Monimutkaisen tietorakenteen muodostaminen yhden funktion sisällä voi johtaa vaikeasti ymmärrettävään koodiin. Lisäksi algoritmi ei ole tällöin yleiskäyttöinen, joten toisenlaisen tietorakenteen tuottaminen vaatii koko funktion muuttamista.

Tietojenkäsittelyssä on helppo keksiä esimerkkejä, missä *Builder* vaikuttaa hyödylliseltä ratkaisulta. Ohjelma voi esimerkiksi tarvita eri osiensa käyttöön samasta tiedosta eri esitysmuodon. Videoeditorin esikatselua varten voidaan tuottaa heikompilaatuinen versio,

kun puolestaan lopullisen videon halutaan olla mahdollisimman tarkka ja selkeä. Käyttämällä samasta tiedosta eri esitysmuotoja, saadaan mahdollistettua sekä sujuva editointi että tuotettua laadukas lopullinen videokuvatiedosto.

5.3.2 Pirtelöresepti ja -mainos

Tarkastellaan pirtelöbaaria, jonka automatisoitu linjasto osaa koota pirtelöannoksia. Pirtelöstä pitää toimittaa tiedot laitteelle, joka yhdistää eri ainekset ja valmistaa pirtelön. Samaista pirtelöä halutaan myös mainostaa nopeasti elektronisella näytöllä, joka ottaa mainokset tekstiformaatissa.

Aivan ensiksi täytyy määrittää, millaisista raaka-aineista pirtelömainokset ja varsinaiset pirtelöt voidaan koostaa. Raaka-aineet ovat esitettävissä kätevästi luettelotyypin avulla.

```
data Ingredient = Sugar | Stevia | Mango | Banana | Vanilla
```

Pirtelömainos laaditaan tarkoin määritellyllä algoritmilla. Se ottaa syötteenä kyseiseen pirtelöön tulevat raaka-aineet ja antaa tulosteena mainostekstin. Algoritmin ansiosta eri pirtelöiden mainosteksteillä on sama formaatti. Funktion tyyppi ja runko ovat seuraavanlaiset.

```
adBuilder :: [Ingredient] -> String
adBuilder :: ingredients = do
    -- define how ingredients are separated to different types
    let acceptedSweeteners = [Sugar, Stevia]
    let acceptedFlavors = [Mango, Banana, Vanilla]

    -- check if ingredients contain sugar
    let begin = if elem Sugar ingredients then “Tasty“ else “Healthy“

    -- combine all the flavors and separate with spaces
    let flavors = ingedients |> filter (elem acceptedFlavors) ...

    -- finalize the ad with product label and exclamation mark
    begin ++ flavors ++ “ Milkshake!”
```

Funktiota tarkasteltaessa se ei vaikuta kovin geneeriseltä. Ensinnäkin sen eri osavaiheet ovat kiinteästi osa funktion runkoa, eivätkä siten helposti muutettavissa. Toisekseen sen avulla on mahdollista tuottaa ainoastaan pirtelömainoksia, ei muita tuotteita. Kummatkin seikat sotivat suunnittelumallin tarkoitusta vastaan, joten funktio vaatii jalostamista.

Algoritmin osavaiheiden erottaminen algoritmin rungosta on jo tuttua *Template Methodin* yhteydestä. Sovellettaessa samaa ajatusta funktioon *adBuilder* saadaan siitä erotettua kolme osaa. Nämä ovat makeutusaineiden käsittely, makujen käsittely sekä pirtelömainoksen viimeistely. Funktioiden tyypit määritellään seuraavasti.

```
-- function to add sweetener to the advertisement
type SweetenerFunction = Ingredient -> String -> String
```

```
-- function to add the flavors to the advertisement
type FlavorFunction = Ingredient -> String -> String

-- function to finalize the advertisement with some decoration
type FinalizeFunction = String -> String
```

Jokainen funktiotyyppi määrittää funktion, joka ottaa parametrinaan tilan ja palauttaa sen muokattuna. Tämän ansiosta raaka-aineiden käsittely on mahdollista ketjuttaa esimerkiksi *fold*-funktiolla.

Vastaavasti funktion *adBuilder* tyyppi ja runko muutetaan käyttämään parametrina annettavia funktioita. Ensimmäisenä parametrina annetaan aihio, josta mainoslausetta aletaan rakentaa.

```
adBuilderTemplate :: String ->
                     SweetenerFunction ->
                     FlavorFunction ->
                     FinalizeFunction ->
                     [Ingredient] ->
                     String
```

Muutosten jälkeen *adBuilder* voidaan parametrisoida käyttämään eri funktioita pirtelömainosten kokoamiseen. Funktion yleiskäyttöisyys on lisääntynyt, mutta se tuottaa yhä vain pirtelömainoksia. Parametrina annettavien funktioiden tyypit eivät yksinkertaisesti salli muun tyyppistä tuotetta, kuin merkkijonon.

Ongelma saadaan ratkaistua parametrisoimalla myös funktioiden tyypit. Funktiot voivat tämän jälkeen tuottaa erilaisia tuotteita, kuten pirtelömainoksia ja pirtelöohjeita.

```
-- function to add sweetener to any state
type SweetenerFunction state = Ingredient -> state -> state

-- function to add flavor to any state
type FlavorFunction state = Ingredient -> state -> state

-- function to convert a state to a finalized product
type FinalizeFunction state product = state -> product
```

Viimeisin versio rakentajasta on nimetty geneerisesti *builderTemplate* nimiseksi. Sen tyyppi on parametrisoitavissa funktiotyypeillä, joiden tyyppi on parametrisoitavissa tilaa ja tuotetta kuvaavilla tyypeillä. Funktio määritellään seuraavasti.

```
builderTemplate :: state ->
                   SweetenerFunction state ->
                   FlavorFunction state ->
                   FinalizeFunction state product ->
                   product
builderTemplate seed
                sweetenerFunction
                flavorFunction
                finalizeFunction
                ingredients =
```

```
    let acceptedSweeteners = [Sugar, Stevia]
    let acceptedFlavors = [Mango, Banana, Vanilla]

    -- folder function used to pass state from one phase to another
    -- so that all the ingredients get handled
    let folder seed ingredient
                | elem ingredient acceptedFlavors =
                        flavorFunction ingredient state
                | elem ingredient acceptedSweeteners =
                        sweetenerFunction ingredient state
                | otherwise = state


    ingredients |> foldl folder seed |> finalizeFunction
```

Funktion tyypin lisäksi myös sen runkoa on selkeytetty alkuperäisestä. Käyttämällä listan *fold*-funktiota saadaan tila kuljetettua vaihe vaiheelta algoritmin läpi samalla, kun siihen lisätään makeutusaine sekä makuja. Viimeistely suoritetaan erillisenä vaiheena.

Funktiota on nyt mahdollista käyttää rakentamaan erilaisia pirtelöihin liittyviä tuotteita, kuten pirtelömainoksia ja pirtelöohjeita. Seuraavassa havainnollistetaan sen käyttö.

```
data Advertisement = String

data Milkshake = Milkshake { name :: String,
                             flavors :: [Ingredient],
                             sweetener :: Ingredient }
```

Tyyppinä *Milkshake* poikkeaa huomattavasti pirtelömainoksen käyttämästä tyypistä, joka on *String*. Tästä huolimatta *builderTemplate* on helppo parametrisoida valmistamaan myös pirtelöitä.

```
-- Advertisement specific functions
adSweetener :: SweetenerFunction Advertisement
adSweetener ingredient state = ...

adFlavor:: FlavorFunction Advertisement
adFlavor ingredient state = ...

...

let adSeed = “”
let adBuilder = builderTemplate adSeed adSweetener adFlavor adFinalizer


-- Milkshake specific functions
milkshakeSweetener :: SweetenerFunction Milkshake
milkshakeSweetener ingredient state = ...

milkshakeFlavor :: FlavorFunction Milkshake
milkshakeFlavor ingredient state = ...

...

let milkshakeSeed = Milkshake “” [] Sugar
```

```
milkshakeBuilder = builderTemplate milkshakeSeed ... milkshakeFinalizer
```

Osittaissovelletut rakentajat *adBuilder* ja *milkshakeBuilder* ottavat nyt listan raaka-aineita parametrinaan ja tuottavat vastaavasti pirtelömainoksia sekä pirtelöohjeita. *Builder*-suunnittelumallin tavoitteet ovat täten täytetty.

5.3.3 Yhteenveto

Builder muistuttaa funktionaalisesti toteutettuna paljolti *Template Methodin* rakennetta. Se toteutetaan korkeamman tason funktiona, jotta sen osavaiheet sekä lopputuotteen tyyppi voidaan parametrisoida. Konkreettinen rakentaja luodaan soveltamalla osittain mallifunktio yhteensopivilla osien rakentamiseen keskittyvillä funktioilla. Pelkistettynä sen rakenne on seuraava.

```
-- type defines for functions producing required parts
type BuildPartA state = IngredientA -> state -> state
type BuildPartB state = IngredientB -> state -> state
type Finalize state product = state -> product

-- builder template defining common parts of the build algorithm
builderTemplate :: Seed ->
                   BuildPartA Seed ->
                   BuildPartB Seed ->
                   Finalize Seed Product ->
                   [Ingredient] ->
                   Product

-- partial application to create concrete builder
concreteBuilder :: [Ingredient] -> ConcreteProduct
concreteBuilder = builderTemplate seed buildPartA buildPartB finalize
```

Yllä esitetty rakenne on vain yleinen malli. Se soveltuu tilanteisiin, joissa raaka-aineet ovat samaa tyyppiä ja näin ollen esitettävissä listalla. Jos raaka-aineet annetaan jossakin muussa tietorakenteessa, niin mallifunktion tyyppiä on muutettava vastaavasti. Jos osia rakentavat funktiot voivat palauttaa suoraan valmiin tuotteen, niin *finalize* on tarpeeton parametri mallifunktiolle. Toisin sanoen suunnittelumallin rakenne voi vaihdella jonkin verran riippuen käyttötilanteesta.

Suunnittelumalli on toteutettavissa funktionaalisen ohjelmoinnin perusominaisuuksilla. Sen rakenteen ydin on korkeamman tason funktio, jonka osavaiheiden tulee olla yhteensopivia rakennettavan tuotteen kanssa. Suunnittelumalli luokitellaan *luonnolliseksi* ja monimutkaisuudeltaan *triviaaliksi*.

5.4 Observer

Observer on yksi käytetyimmistä olioperustaisen ohjelmoinnin suunnittelumalleista. Se mahdollistaa tiedon tuottajan ja tiedon kuluttajien erottamisen toisistaan, joka on erittäin hyödyllinen ominaisuus monissa olioperustaisissa ohjelmissa. Toistaiseksi tarkastellut

suunnittelumallit ovat taipuneet helposti funktionaaliseen asiayhteyteen. *Observer* valitaan esimerkiksi suunnittelumallista, jonka soveltaminen funktionaalisessa ohjelmoinnissa on ongelmallista.

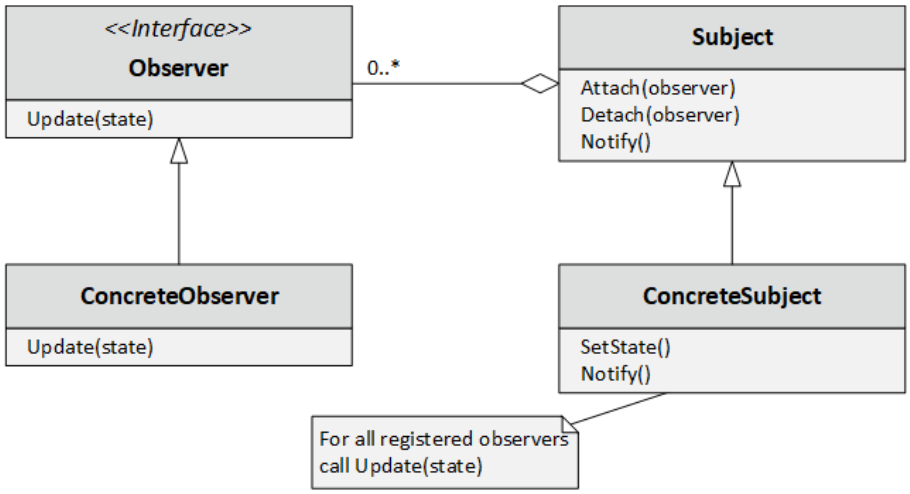
Luokittelu
Käyttäytymismalli

Tarkoitus
Mahdollistaa yksi-moneen-suhde tarkkailtavan ja tarkkailevien olioiden välillä siten, että tarkkailtavan tilan muuttuessa kaikki tarkkailijat saavat siitä tiedon ja päivitetään automaattisesti.

Soveltuvuus
Mallia voidaan käyttää, kun

- toiminnallisuudessa on kaksi osaa, joista toinen on riippuvainen toisesta. Kapseloimalla osat omiin olioihinsa, on mahdollista muuttaa ja uudelleen käyttää niitä toisistaan riippumatta
- muutos yhteen olioon vaatii muutoksia muihin olioihin, eikä ole tiedossa moneenko olioon muutos vaikuttaa
- olion tulee voida ilmoittaa muita olioita muutoksistaan, ilman tiukkaa sidettä olioiden välillä.

Rakenne



Kuva 6. Observer-suunnittelumallin push-variantin luokkakaavio

Toiminta
Toiminta perustuu kahteen rajapintaan, *Objectiin* ja *Subjectiin*, joiden instansseja kutsutaan vastaavasti tarkkailijaolioiksi ja kohdeolioksi. Tarkkailijaoliot rekisteröidään kuuntelemaan kohdeolion muutoksia rajapinnan *Subject* kautta. Kohdeolion vastuulla on kutsua *Notify()*-funktiota, kun se huomaa tilansa muuttuneen. Tämä aiheuttaa kaikkien tarkkailijaolioden päivityksen *Update(state)*-funktiokutsulla.

5.4.1 Soveltuvuus funktionaaliseen ohjelmointiin

Oberver liittyy keskeisesti muuttuvan tilan tarkkailuun ja muutoksiin reagoimiseen. Olion tila muuttuu erilaisten I/O-tapahtumien johdosta, kuten lämpötila-anturin lukeman muuttuessa tai ajan kuluessa. Sen avulla joukko tarkkailijaolioita voidaan löyhästi sitoa tarkkailtavaan kohdeolioon.

Puhtaat funktionaaliset ohjelmat ovat edellä esitetyssä mielessä tilattomia. Niissä ei myöskään ole I/O-operaatioita, jotka spontaanisti generoisivat tapahtumia. Vaikka osaa suunnittelumallin rakenteesta ja toiminnasta on mahdollista jäljitellä funktionaalisen ohjelmoinnin keinoin, sen idea ei kuitenkaan sovi lainkaan funktionaaliseen asiayhteyteen.

5.4.2 Yhteenveto

Funktionaalisessa ohjelmoinnissa ei ole tilankäsitettä eikä siten tarvetta myöskään ilmoittaa tilan muutoksista. Suunnittelumallin tarkoitus ei ole sovellettavissa funktionaalisessa asiayhteydessä, joten sen soveltuvuus on *soveltumaton*. Sille ei voida määrittää rakennetta, eikä näin ollen arvioida rakenteen kompleksisuutta.

6. TULOKSET JA NIIDEN TARKASTELU

Tutkielman tulokset esitellään tässä luvussa. Tulokset perustuvat tietoihin, jotka on saatu soveltamalla luvussa 4 esiteltyä tutkimusmenetelmää valittuihin suunnittelumalleihin. Taulukossa 6 on esitetty yhteenveto saaduista tuloksista yksittäisten suunnittelumallien osalta. Yksityiskohtainen analyysi jokaisesta suunnittelumallista on esitetty liitteessä 1.

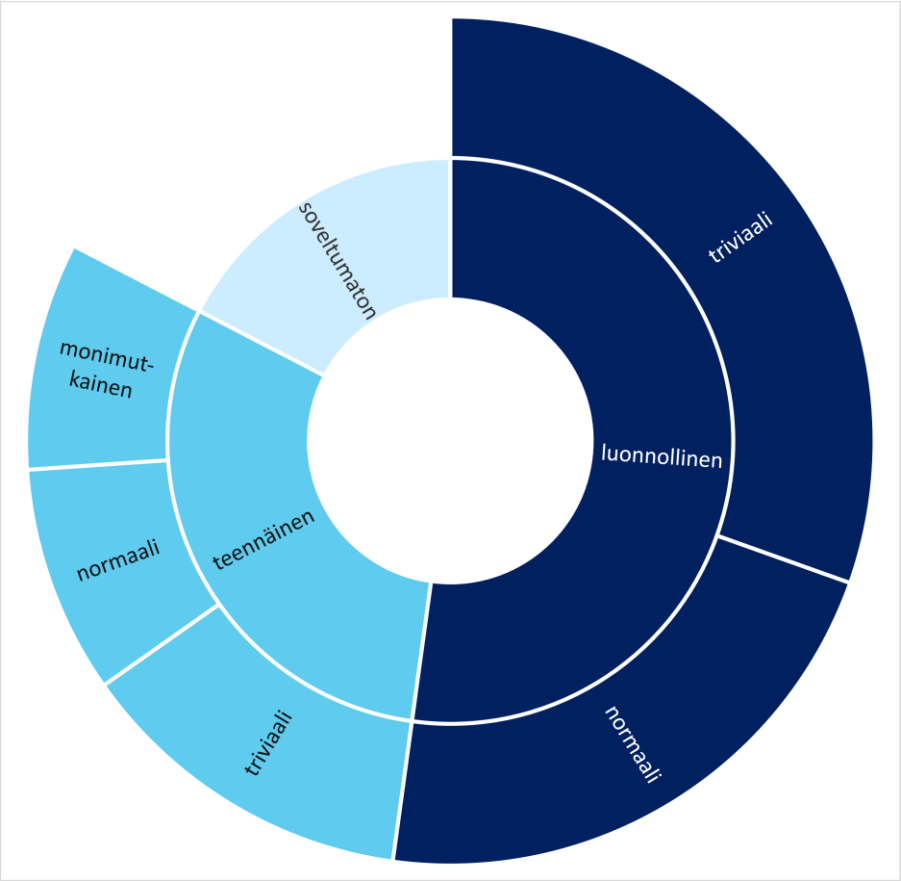
	Soveltuvuus	**Kompleksisuus**
Rakennemallit		
Abstract Factory	teennäinen	normaali
Builder	luonnollinen	triviaali
Factory Method	teennäinen	normaali
Prototype	teennäinen	triviaali
Singleton	soveltumaton	-
Rakennemallit		
Adapter	luonnollinen	triviaali
Bridge	luonnollinen	normaali
Composite	luonnollinen	normaali
Decorator	luonnollinen	triviaali
Facade	luonnollinen	triviaali
Flyweight	teennäinen	monimutkainen
Proxy	luonnollinen	triviaali
Käyttäytymismallit		
Chain of Responsibility	luonnollinen	normaali
Command	teennäinen	triviaali
Interpreter	luonnollinen	normaali
Iterator	luonnollinen	triviaali
Mediator	soveltumaton	-
Memento	soveltumaton	-
Observer	soveltumaton	-
State	luonnollinen	triviaali
Strategy	luonnollinen	triviaali
Template Method	luonnollinen	triviaali
Visitor	luonnollinen	normaali

Taulukko 6. Suunnittelumallien funktionaalinen luokittelu

Luvussa tarkastellaan ensin yleisesti suunnittelumallien soveltuvuutta funktionaaliseen ohjelmointiin. Tämän jälkeen analysoidaan niiden soveltuvuus pääkategorian mukaan ja pyritään etsimään myös muita yhtäläisyyksiä. Lopuksi vertaillaan suunnittelumallien olioperustaisia ja funktionaalisia toteutuksia.

6.1 Yleiskatsaus tuloksiin

Pääosin tutkitut olioperustaiset suunnittelumallit on mahdollista esittää myös funktionaalisen ohjelmoinnin menetelmin. Suunnittelumalleista noin puolet sopii luonnollisena osana funktionaaliseen ohjelmointiin, ja vain alle viidennes on täysin sopimattomia. Kuva 7 esittää suunnittelumallien jakautumisen soveltuvuuden sekä kompleksisuuden mukaan.



Kuva 7: Suunnittelumallit jaoteltuna soveltuvuuden sekä kompleksisuuden mukaan

Segmenteistä *luonnollinen* on suurin ja edustaa suunnittelumalleja, jotka sopivat ideansa puolesta erinomaisesti myös funktionaaliseen ohjelmointiin. Kyseinen segmentti jakaantuu kahteen alisegmenttiin: *triviaali* ja *normaali*. Alisegmenteistä triviaali edustaa suunnittelumalleja, jotka ovat pelkistyneet huomaamattomiksi osiksi funktionaalista ohjelmaa. Toinen alisegmenteistä, eli normaali, edustaa puolestaan joukkoa, jonka ymmärtämisestä on mahdollisesti hyötyä myös funktionaalisessa asiayhteydessä. Sen edustamat suunnittelumallit ovat sekä hyödyllisiä että vaativat monipuolisempaa rakennetta myös funktionaalisessa ohjelmoinnissa.

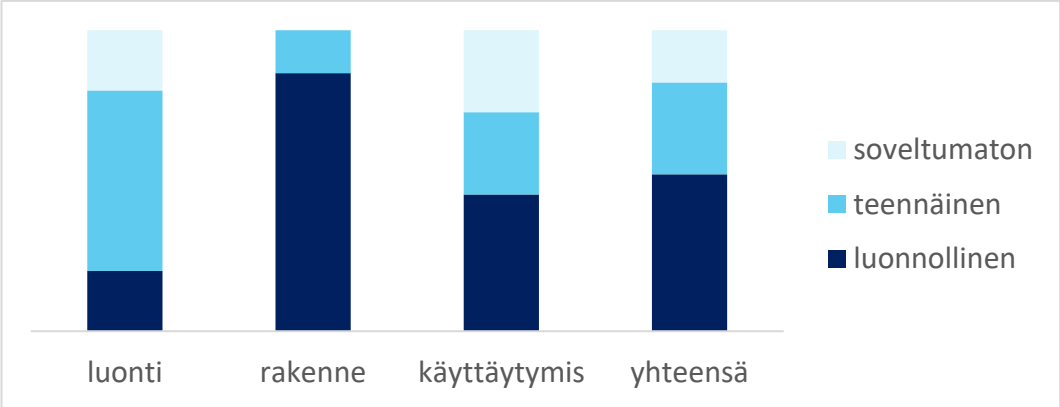
Segmenteistä *teennäinen* on toiseksi suurin. Sen edustamat suunnittelumallit on mahdollista toteuttaa myös funktionaalisessa ohjelmoinnissa, mutta niiden hyöty on kyseenalainen. Huomionarvoisesti yli kolmannes teennäisistä suunnittelumalleista on samalla myös triviaaleja. Vaikka ne olisivat hyödyllisiä funktionaalisessa ohjelmoinnissa, niiden käyttö

olisi yhä sisäänrakennettua. Alisegmenteistä kaksi muuta edustavat rakenteita, joiden hyöty on kyseenalainen ja joiden rakenne vaatii jokseenkin suunnittelua. Todennäköisesti muut rakenteet funktionaalisessa ohjelmoinnissa voivat tarvittaessa korvata nämä.

Viimeinen segmenteistä on *soveltumaton*, johon kuuluvien suunnittelumallien alkuperäinen idea ei ole siirrettävissä funktionaaliseen ohjelmointiin. Kyseisiä suunnittelumalleja oli vain alle viidennes tutkituista suunnittelumalleista. Nämä suunnittelumallit käsitellään yksityiskohtaisemmin kohdassa 6.3.3.

6.2 Soveltuvuus alkuperäisen kategorian mukaan

Suunnittelumallit on alun perin jaoteltu kolmeen pääkategoriaan: luontimalleihin, rakennemalleihin sekä käyttäytymismalleihin. Muunnettaessa suunnittelumallit funktionaaliseen ohjelmointiin soveltuviksi, esiintyi eri kategorioiden välillä jonkin verran eroja.



Kuva 8. Soveltuvuus kategorioittain

Kuvassa 8 on esitetty suunnittelumallien soveltuvuus funktionaaliseen ohjelmointiin kategorioittain. Palkit on esitetty suhteellisina osuuksina johtuen kategorioiden koon epätasaisuudesta. Parhaiten suunnittelumalleista paradigmasta toiseen siirtyivät rakennemallit ja selkeästi huonoiten luontimallit.

Luontimallien kohdalla ainoastaan *Builder* on funktionaaliseen ohjelmointiin sopiva ratkaisumalli. Se on käytännössä algoritmirunko, jonka vaiheet on mahdollista parametrisoida. Luontimalleista valtaosan tarkoitus oli tavalla tai toisella piilottaa luodut oliot rajapinnan taakse. Nämä suunnittelumallit tuntuivat funktionaalisessa asiayhteydessä teennäisiltä. *Singleton* poikkesi muista kategorian suunnittelumalleista keskittyen luotujen olioiden lukumäärän hallintaan. Se on käsitelty tarkemmin kohdassa 6.3.3 yhdessä muiden soveltumattomien suunnittelumallien kanssa.

Rakennemallit ovat yhtä lukuun ottamatta luonteva osa funktionaalista ohjelmointia. Ainoastaan *Flyweight* tuntuu teennäiseltä. Se keskittyy luotujen olioiden jakamiseen, ja on tässä mielessä rinnastettavissa luontimalleihin.

Viimeisessä kolmesta kategoriasta, eli käyttäytymismalleissa, esiintyi eniten vaihtelua. Sen edustamat suunnittelumallit jakaantuivat tasaisesti sekä luonteviin, teennäisiin että soveltumattomiin. Kategorian suunnittelumalleista *Strategy* ja *Template Method* olivat malliesimerkkejä funktionaalisen ohjelmoinnin eduista. Puolestaan olioiden suoritusaikaiseen kommunikointiin liittyvät *Observer* ja *Mediator* eivät soveltuneet lainkaan siirrettäväksi funktionaaliseen ohjelmointiin.

6.3 Tunnistetut yhtäläisyydet

Suunnittelumallit sopivat vaihtelevasti funktionaaliseen ohjelmointiin. Funktionaalisessa muodossa osassa niistä paljastuu selkeitä yhtenäisiä piirteitä, joiden perusteella niitä voidaan luokitella. Suuri osa suunnittelumalleista on kuitenkin niin yksilöllisiä, että merkittäviä yhtäläisyyksiä muihin suunnittelumalleihin ei niistä löydy. Tässä kappaleessa tarkastellaan sekä luonnollisesti soveltuvien, teennäisestä soveltuvien täysin soveltumattomien suunnittelumallien samankaltaisuuksia.

6.3.1 Luonnollisesti soveltuvat suunnittelumallit

Suunnittelumallit, jotka parhaiten soveltuvat funktionaaliseen ohjelmointiin ja pelkistyvät triviaaleiksi, voidaan jakaa rakenteensa suhteen karkeasti kahteen joukkoon. Ensimmäisen joukoista muodostavat suunnittelumallit, jotka on mahdollista esittää algoritmirungon variointina. Toisena joukkona ovat puolestaan suunnittelumallit, jotka muuntuvat erinäisiä tehtäviä suorittaviksi funktioiksi. Seuraavaksi tarkastellaan näitä kahta joukkoa erikseen.

Algoritmin variointia edustavat suunnittelumallit ovat yksinkertaisia toteuttaa funktionaalisin menetelmin. Sopivat tekniikat tähän ovat korkeamman tason funktiot sekä osittaissoveltaminen. Suunnittelumalleista *Builder*, *Template Method* ja *Strategy* kuuluvat tähän joukkoon.

Yksinkertaisiksi funktioiksi pelkistyvät suunnittelumallit muodostavat toisen joukon. *Adapter*, *Decorator* ja *Proxy* lisäävät toiminnallisuutta ydinfunktioon. *Facade* puolestaan yhdistää joukon funktioita yhden funktion taakse luoden selkeämmän rajapinnan. Tämän tapaiset muunnokset ja toimintojen lisäämiset ovat triviaaleja toteuttaa funktioilla.

Funktionaaliseen ohjelmointiin luonnollisesti soveltuvia suunnittelumalleja, joiden rakenne ei kuitenkaan pelkisty triviaaliksi, on viisi kappaletta. *Bridge* ja *Chain of Responsibility* vaativat monipuolista funktioiden yhdistelemistä halutun toiminnallisuuden saavuttamiseksi. *Composite*, *Interpreter* ja *Visitor* puolestaan vaativat funktioiden ja tietotyyppien yhteensovittamista.

6.3.2 Teennäisesti soveltuvat suunnittelumallit

Teennäiset suunnittelumallit vaihtelevat monimutkaisuudeltaan laidasta laitaan, ja niitä esiintyy kaikissa kolmessa suunnittelumallien kategoriassa. Luontimalleihin kuuluvat *Abstract Factory*, *Proxy* ja *Template Method* muodostavat ainoana selkeän ryhmän. Funktionaalisen monimuotoisuuden avulla niiden luomien tietotyyppien käsittely on mahdollista mutta tuntuu silti teennäiseltä.

6.3.3 Soveltumattomat suunnittelumallit

Funktionaaliseen ohjelmointiin soveltumattomista suunnittelumalleista *Mediator* ja *Observer* omaavat selkeitä yhtäläisyyksiä. Molemmat suunnittelumallit ovat kategorialtaan käyttäytymismalleja, ja niiden tarkoitus on auttaa ohjelman eri osien välistä kommunikaatiota. Puhtaassa funktionaalisessa ohjelmoinnissa vastaavaa käsitettä ei ole.

Memento kuuluu kategorialtaan myös käyttäytymismalleihin. Poiketen kuitenkin aiemmin mainituista kahdesta suunnittelumallista se ei liity olioiden väliseen kommunikointiin. Sen sijaan sen tehtävä on mahdollistaa aiemman tilan palautus. Funktionaalisilla ohjelmilla ei varsinaista tilaa ole, joten myös siihen liittyvät suunnittelumallit saattavat usein olla ongelmallisia.

Neljäs soveltumaton suunnittelumalli on *Singleton*. Se on ainoa funktionaaliseen ohjelmointiin soveltumaton luontimalli. Poiketen muista kategoriansa suunnittelumalleista, sen pääasiallinen tehtävä ei ole rakentaa olioita tai varmistaa olioiden yhteenkuuluvuutta. Sen sijaan sen tarkoitus on hallita edustamansa tyypin instanssien lukumäärää. Tässä mielessä se on lähellä suunnittelumallia *Flyweight*, jolla on myös samantapaiset ongelmat. Funktionaalisen ohjelmoinnin muuttumattomien arvojen takia käsite samaisen arvon lukumäärästä on yksiselitteisesti hyödytön.

6.4 Vertailu olioperustaiseen ohjelmointiin

Funktionaaliset versiot olioperustaisista suunnittelumalleista ovat pääsääntöisesti lyhyempiä ja kevyempiä rakenteiltaan. Ainoastaan *Iterator* ja *Flyweight* muuttuvat hankaliksi toteuttaa ilman talletettavaa tilaa. Huomionarvoista on, että *Iterator* pelkistyisi triviaaliksi, mikäli mahdollisuus selata kokoelmaa askel askeleelta poistettaisiin.

Funktionaaliset vastineet ovat osittain myös olioperustaisia esikuviaan monipuolisempia ja joustavampia. Tarkasteltaessa esimerkiksi *Template Methodia* huomataan funktionaalisessa versiossa tiettyjä etuja. Olioperustainen versio nojaa perintään, jolloin algoritmin kaikki osavaiheet toteutetaan kerralla aliluokassa. Jos näistä osavaiheista halutaan varioida yhtä, tarvitaan *Strategy*-suunnittelumalli avuksi. Funktionaalisessa versiossa korkeamman tason ydinfunktio määrittelee algoritmin rungon, ja on suoraan parametrisoitavissa osavaiheittain. Samoja piirteitä on havaittavissa myös muista suunnittelumalleista.

Olioperustaisen ja funktionaalisen suunnittelumallin rakenteet eivät vaikuta korreloivan vahvasti. Vaikka esimerkiksi *Composite* ja *Decorator* omaavat yhtenevät olioperustaiset rakenteet, on niiden funktionaalisilla versioilla varsin vähän yhteistä (Gamma 2000 s. 10). Suunnittelumallin käyttötarkoitus tuntuu olevan erottava tekijä ainakin osassa tapauksista.

7. YHTEENVETO

Tutkielman tarkoitus oli selvittää, miten olioperustaiset suunnittelumallit soveltuvat funktionaaliseen ohjelmointiin. Suunnittelumallit ovat tärkeä osa hyvää olioperustaista ohjelmointia. Funktionaalinen ohjelmointi on puolestaan viime vuosina kasvattanut suosiotaan. Funktionaalisen ohjelmoinnin oli esitetty tekevän suunnittelumalleista pitkälti tarpeettomia, mutta tieteellistä tutkielmaa aiheesta ei löytynyt.

Tutkielman kohteeksi valittiin *Design Patterns : Elements of Reusable Object-Oriented Programming* -kirjassa esitetyt 23 suunnittelumallia. Niiden taustalla vaikuttava idea ja teoria käsiteltiin sekä luotiin yleiskatsaus niihin. Lisäksi tarkasteltiin, miksi ne liittyvät vahvasti nimenomaan olioperustaiseen ohjelmointiin.

Funktionaalinen ohjelmointi on käsitteenä laaja. Tietyt piirteet esiintyvät useissa funktionaalisissa ohjelmointikielissä, mutta mitään tarkkaa määritelmää niiden suhteen ei ole. Funktionaalinen ohjelmointi määriteltiin työssä tarkoittamaan tilatonta ja sivuvaikutuksetonta ohjelmointia, jossa käytettiin muutamia yleisiä funktionaalisista ohjelmointikielistä löytyviä ominaisuksia.

Suunnittelumallit pyrittiin sovittamaan määriteltyyn funktionaaliseen ohjelmointiin yksitellen. Jokaista sovellettavissa olevaa suunnittelumallia kohden laadittiin lyhyt esimerkkiohjelma, joka havainnollisti sen käyttö funktionaalisessa asiayhteydessä. Suunnittelumalli luokiteltiin sen soveltuvuuden perusteella kolmiportaisella asteikolla (luonnollinen, teennäinen, soveltumaton). Soveltuvien suunnittelumallien rakenne pelkistettiin ja arvioitiin niin ikään kolmiportaisella asteikolla (triviaali, normaali, monimutkainen).

Suunnittelumalleista 11 soveltui luonnollisena osana funktionaaliseen ohjelmointiin, 7 teennäisesti ja 4 ei soveltunut lainkaan. Soveltuvista suunnittelumalleista 10 rakenne pelkistyi triviaaliksi, 7 todettiin normaaliksi ja 2 monimutkaiseksi. Suunnittelumalleista 5 sai luokituksen luonnollinen-normaali, ja niitä voidaan pitää jokseenkin hyödyllisinä myös funktionaalisessa ohjelmoinnissa.

Tutkielman perusteella todetaan, että funktionaalinen ohjelmointi tekee tarpeettomaksi suuren osan tutkituista suunnittelumalleista. Funktionaalisessa muodossa suunnittelumallien rakenne on usein yksinkertaisempi sekä toisinaan myös olioperustaista vastinettaan joustavampi.

LÄHTEET

Gamma, E., Helm, R., Johnson R., Vlissides J. (2000). Design Patterns : Elements of reusable Object-Oriented Software. Addison-Wesley.

Hannemann, J., Kiczales, G. (2002). Design Pattern Implementation in Java and AspectJ.
Saatavissa: https://github.com/functional-patterns/gof/blob/master/material/Design_Pattern_Implementation_in_Java_and_AspectJ.pdf

HaskellWiki. (2014). Functional Programming.
Saatavissa: https://wiki.haskell.org/Functional_programming
Viitattu: 2018-11-05

Hudak, P. (1989).The Conception, Evolution, and Application of Functional Programming Language. Yale University Department of Computer Science.
Saatavissa: http://haskell.cs.yale.edu/wp-content/uploads/2011/01/cs.pdf

Martin, R.C. (2000). Design Principles and Design Patterns.
Saatavissa: https://github.com/functional-patterns/gof/blob/master/material/Principles_and_Patterns.pdf

Norvig, P. (1996). Design Patterns in Dynamic Languages.
Saatavissa: http://www.norvig.com/design-patterns/
Viitattu: 2018-11-05

Petricek, T. (2010). Real-World Functional Programming. Manning Publications Co.

Safyan, M. (2016). Singleton Anti-Pattern.
Saatavissa: https://www.michaelsafyan.com/tech/design/patterns/singleton
Viitattu: 2018-11-05

Sebesta, R. (1999). Concepts of Programming Languages. Addison-Wesley

Seeman, M. (2012). Dependency Injection in .NET. Manning Publications Co.

LIITE 1: SUUNNITTELUMALLIEN ANALYYSIT

Tässä liitteessä on esitetty jokainen *Design Patterns* -kirjan suunnittelumalleista funktionaalisessa asiayhteydessä. Jokaisesta suunnittelumallista on esitetty seuraavat tiedot:

- nimi
- tarkoitus funktionaalisessa asiayhteydessä
- rakenteen ja toiminnan kuvaus
- yhteenveto
    - pohdinta suunnittelumallista funktionaalisessa asiayhteydessä
    - soveltuvuuden luokittelu kolmiportaisella asteikolla
    - kompleksisuuden luokittelu komiportaisella asteikolla.

ABSTRACT FACTORY

Tarkoitus

Tarjota rajapinta yhteensopivien tietorakenteiden luomiseen paljastamatta niiden konkreettisia tyyppejä

Rakenne

```
-- A and B presents product types, 1 and 2 presents product families
data ProductA = ProductA1 | ProductA2
data ProductB = ProductB1 | ProductB2

data Factory = Factory { createA :: () -> ProductA,
                         createB :: () -> ProductB }


-- Context
doSomething :: Factory -> ...
doSomething (Factory createA createB) ... = do
    let productA = createA
    let productB = createA
    ...
```

Tuotetyypit määritellään algebrallisilla tietotyypeillä. Niiden arvot kuvaavat tuotteen eri versioita. Tuotteiden yhteensopivuus varmistetaan tietueen *Factory* avulla, jonka jäsenet ovat eri tuotteita luovia funktioita. Menetelmä vastaa pitkälti riippuvuusinjektiota.

Haskellin tarjoamien tyyppiluokkien avulla tuotteiden eri versioiden ei tarvitse olla osa samaa algebrallista tietotyyppiä. Tämä puolestaan mahdollistaa uusien tuoteperheiden lisäämisen siten, ettei olemassa olevaan koodiin tarvitse tehdä muutoksia.

Yhteenveto

Abstract Factory on sovellettavissa myös funktionaaliseen ohjelmointiin. Algebralliset tietotyypit ovat luonnollinen tapa määrittää tuotetyypit eri versioineen. Konkreettisten tietotyyppien piilottaminen tuntuu kuitenkin osittain luonnottomalta funktionaalisessa ohjelmoinnissa.

soveltuvuus	teennäinen
kompleksisuus	normaali

ADAPTER

Tarkoitus

Muuntaa tietotyyppi – mukaan lukien funktion tyyppi – asiakasfunktiolle sopivaksi.

Rakenne

Tietotyypin muuntaminen on pari yksinkertaisia funktioita, jotka muuntavat tietotyypin esitysmuodosta X esitysmuotoon Y sekä takaisin.

```
-- Adapting data type
type Original = ...
type Adapted = ...

fromOriginal :: Original -> Adapted
toOriginal :: Adapted -> Original
```

Funktion tyypin muuntaminen on sen ottamien parametrien tai paluuarvon tyypin muuntamista. Myös parametrien järjestystä voidaan muokata.

```
-- Adapting parameter types
originalFunction :: Original -> Int

adaptedFunction :: Adapted -> Int
adaptedFunction adapted = originalFunction (toOriginal adapted)toOriginal

-- Adapting parameter ordering
originalFunction :: Int -> Float -> Int

adaptedFunction :: Float -> Int -> Int
adaptedFunction a b = originalFunction b a
```

Yhteenveto

Adapter on osa käytännössä jokaista funktionaalista ohjelmaa. Se on funktionaalisesti erittäin kevyt rakenteinen sekä joustava. Tarkka rakenne riippuu, mitä halutaan muuntaa.

soveltuvuus	luonnollinen
kompleksisuus	triviaali

BRIDGE

Tarkoitus

Erottaa abstraktio ja toteutus siten, että ne voivat varioida toisistaan riippumatta.

Rakenne

Rajapintaosassa on funktiomalli *operationTemplate*, sekä joukko muita sen kanssa yhteensopivia funktiomalleja.

```
-- Abstraction part
operationTemplate :: PrimitiveOperation -> ...

-- Primitive operation templates
primitiveOperationTemplate :: ImplementationFunction -> PrimitiveOperation
...
```

Toteutusosassa on joukko yleiskäyttöisiä funktioita, joilla ei ole suoraa riippuvuutta funktioon *operationTemplate*.

```
-- Implementation part
implementationFunction :: ImplementationFunction
...
```

Funktio *operationTemplate* sovelletaan osittain rajapintaosan muilla mallifunktioilla, jotka puolestaan sovelletaan osittain halutun toteutusosan funktioilla. Funktio *primitiveOperationTemplate* toimii siltana varsinaisen rajapintafunktion *operation* ja toteutusfunktion *implementationFunction* välillä, jolloin niiden toteutusta voidaan muuttaa toisistaan riippumatta.

Yhteenveto

Bridge toimii hyvin myös funktionaalisessa ohjelmoinnissa. Sen rakenne hyödyntää korkeamman tason funktioita sekä osittaissoveltamista. Sijoittamalla funktio rajapintaosan ja toteutusosan väliin, saadaan suora riippuvuus poistettua.

soveltuvuus	luonnollinen
kompleksisuus	triviaali

BUILDER

Tarkoitus

Erottaa monimutkaisen tietorakenteen esitystapa sen luontiprosessista.

Rakenne

Builder on rakenteeltaan korkeamman tason funktio. Se ottaa parametrinaan funktiot, jotka muuntavat tilan ja raaka-aineen tilaksi. Tällä tavoin funktioiden tuotokset voidaan ketjuttaa tuotteen muodostamiseksi.

```
-- type defines for functions producing required parts
type BuildPartA state = IngredientA -> state -> state
type BuildPartB state = IngredientB -> state -> state
type Finalize state product = state -> product

-- builder template defining common parts of the build algorithm
builderTemplate :: Seed ->
                   BuildPartA Seed ->
                   BuildPartB Seed ->
                   Finalize Seed Product ->
                   [Ingredient] ->
                   Product

-- partial application to create concrete builder
concreteBuilder :: [Ingredient] -> ConcreteProduct
concreteBuilder = builderTemplate seed buildPartA buildPartB finalize
```

Yllä esitetty rakenne sopii tilanteisiin, joissa raaka-aineet ovat samaa tyyppiä. Muussa tapauksessa raaka-aineet joudutaan antamaan erikseen listan sijaan.

Jos osia rakentavat funktiot voivat palauttaa suoraan valmiin tuotteen, niin *finalize* on tarpeeton parametri mallifunktiolle. Suunnittelumallin rakenne voi vaihdella jonkin verran riippuen käyttötilanteesta.

Yhteenveto

Builder on käyttökelpoinen suunnittelumalli funktionaalisessa ohjelmoinnissa. Se eriyttää luontialgoritmin tuotteen rakenteesta samaan tapaan kuin olioperustainen esikuvansa.

soveltuvuus	luonnollinen
kompleksisuus	triviaali

CHAIN OF RESPONSIBILITY

Tarkoitus

Mahdollistaa funktioiden ketjuttamisen siten, että yksi tai useampi niistä voi käsitellä parametrin.

Rakenne

Chain of Responsibility on joukko yhteen ketjutettuja funktioita, joiden parametri ja paluuarvo ovat yhteensopivat. Parametriin voidaan lisätä tarvitaessa konteksti monadilla.

```
regularHandler :: a -> a
monadicHandler :: a -> m a
```

Yhteenveto

Chain of Responsibility on usein käytetty rakenne funktionaalisessa, ja on läheistä sukua funktioiden putkittamiselle. Sen monadinen versio soveltuu muun muassa virhetilan välittämiseen.

soveltuvuus	luonnollinen
kompleksisuus	triviaali / normaali (monadinen versio)

COMMAND

Tarkoitus

Määrittää rajapinnan toistensa kanssa vaihdannaisille komennoille.

Rakenne

Command on funktio, joka ottaa ja palauttaa tilan. Monadista versiosta voidaan käyttää, jos komentojen sallitaan myös epäonnistua. Huomioitavaa on, ettei tavallisia ja monadisia komentoja voi yhdistellä.

```
regularCommand :: state -> state
monadicCommand :: state -> Maybe state
```

Komentoja on mahdollista myös yhdistää makrokomennoksi. Makrokomento on mallifunktio, joka ottaa listana suoritettavat komennot. Täten se voidaan soveltaa osittain halutulla komentojoukolla konkreettiseksi komennoksi.

```
macroCommandTemplate :: [command] -> state -> state
```

Komennot voidaan välittää asiakasfunktiolle assosiaatiotauluna. Asiakasfunktio etsii tarvittaessa suoritustaan vastaavan komennon ja kutsuu sitä.

Yhteenveto

Command pelkistyy funktionaalisessa ohjelmoinnissa yksinkertaiseksi funktioksi. Sen idea ei kuitenkaan sovellu hyvin puhtaaseen funktionaaliseen ohjelmointiin. IO-operaatioiden sekä tilan puuttuminen tekevät sen sovelluskohteista vähäisiä.

soveltuvuus	teennäinen
kompleksisuus	triviaali

COMPOSITE

Tarkoitus

Koostaa osista puumaisia osa-kokonaisuus-hierarkioita. Composite mahdollistaa yksittäisen osan ja kokonaisuuden käsittelyn yhtenäisesti.

Rakenne

Composite mallinnetaan rekursiivisen algebrallisen tietotyypin avulla, joka mahdollistaa puurakenteen. Funktiot toteutetaan purkamaan koostearvot mallin sovittamisen avulla.

```
data Component = Leaf X | Composite [Component]

operation1 :: Shape -> b
operation1 (Leaf x) = ...
operation1 (Composite components) =
    -- perform operation with fold

operation2 :: Shape -> Shape
operation (Leaf x) = ...
operation2 :: (Composite components) =
    -- perform operation with map
```

Funktiot voivat olla eri tyyppisiä – kuten fold, map tai filter – jolloin niillä voidaan käsitellä tietorakennetta eri tavoin. Käyttäjän ei tarvitse olla tietoinen kuuluuko koosteeseen yksi arvo vai onko se suurempi kokonaisuus.

Yhteenveto

Compositen merkitys on sama funktionaalisessa ohjelmoinnissa kuin olioperustaisen esikuvansa. Sen toteutus vaatii yhteenkuuluvia tietorakenteita sekä funktioita.

soveltuvuus	luonnollinen
kompleksisuus	normaali

DECORATOR

Tarkoitus

Lisätä toiminnallisuutta funktioihin muuttamatta niiden tyyppiä.

Rakenne

Funktion kuorrutus on mahdollista tehdä usealla tavalla. Tärkeää kuitenkin on pitää funktion tyyppi muuttumattomana.

```
coreFunction :: a -> b -> c
decoratedFunction :: a -> b -> c
```

Kuorrutus voidaan lisätä joko ennen tai jälkeen funktiokutsua, tai molemmin puolin sitä.

Etuliitteinen kuorruttaja muuntaa parametrit ennen niiden välittämistä *coreFunctio*lle. Jos parametreja on monta, niistä tehdään monikko.

```
preDecorator :: (a -> b) -> (a, b)
```

Jälkiliitteinen kuorruttaja muuntaa paluuarvon ennen sen palauttamista.

```
postDecorator :: c -> c
```

Sekä etu- että jälkiliitteisiä kuorruttajia on mahdollista liittää mielivaltainen määrä yhteen. Apuna on mahdollista käyttää funktioita *pre* ja *post*.

```
pre :: (a, b) -> (a -> b -> c) -> c
pre ab f = let (a, b) = ab in f a b

post :: c -> (c -> c) -> c
post c f = f cc
```

Kuorrutettu funktio voidaan nyt koostaa seuraavasti.

```
decorated :: a -> b -> c
decorated a b = (preDecorator a b) `pre` coreFunction `post` postDecorator
```

Yhteenveto

Decorator on luonnollinen osa funktionaalista ohjelmointia. Se lisää toiminnallisuutta funktioon. Sitä on mahdollista käyttää myös täydentämään tietotyyppejä rikkaammiksi.

soveltuvuus	luonnollinen
kompleksisuus	triviaali

FACADE

Tarkoitus

Tarjota korkeamman tason rajapinta alijärjestelmän käyttämiseksi.

Rakenne

Facadella ei ole tarkoin määriteltyä rakennetta funktionaalisessa ohjelmoinnissa. Se koostaa joukon alijärjestelmän funktioita yksittäisen funktion taakse, jolloin niiden käyttäminen on helpompaa.

```
lowlevelA :: a -> b
lowlevelB :: a -> c
lowlevelC :: b c -> d

facade :: a -> d
facade a =
  let b = lowlevelA
      c = lowlevelB
      in
      lowLevel b c
```

Facade voi määrittää myös lisätoiminnallisuutta alemman tason funktiokutsuja ennen, jälkeen tai välissä. Jos ainoastaan yksi alemman tason funktio kätketään *Facaden* taakse, on rakenne yhteneväinen *Adapterin* kanssa.

Yhteenveto

Facaden tarkoitus säilyy muuttumattomana funktionaalisessa ohjelmoinnissa. Sen käyttö selkeyttää alemman tason funktioista koostuvan alijärjestelmän käyttöä.

soveltuvuus	luonnollinen
kompleksisuus	triviaali

FACTORY METHOD

Tarkoitus

Määrittää rajapinta tiedon luomiselle funktiotyyppinä, joka tukee erilaisia luotavia arvoja.

Rakenne

Factory Method on funktio, joka ottaa yhden tai useamman parametrin ja palauttaa arvon. Paluuarvon tulee tukea toiminnallisuutta, jota asiakaskoodi odottaa. Tämä voidaan saavuttaa kahdella tavalla.

1. Paluuarvo on algebrallinen tietotyyppi.

```
data Product = ProductA a | ProductB b
productFactory :: x -> Product
```

2. Paluuarvo on tyyppiluokan instanssi.

```
productFactory :: (Product product) => x -> Product
productFactory x = ...
```

Yhteenveto

Factory Methodin idea ei sovi kovin hyvin funktionaaliseen ohjelmointiin. Vaikka abstraktin tiedon luominen on helppoa, on sen käyttäminen jokseenkin hankalaa.

Factory Methodilla on mahdollista luoda myös funktioita soveltamalla osittain. Tämä ei kuitenkaan tunnu lainkaan suunnittelumallin alkuperäisen idean mukaiselta.

soveltuvuus	teennäinen
kompleksisuus	normaali

FLYWEIGHT

Tarkoitus

Tukea suurta määrää tietoalkioita jakamalla niiden yhteiset osat.

Rakenne

Flyweight on funktio, joka palauttaa parametrina annettua avainta vastaavan arvon. Sama arvo on mahdollista liittää moneen muuhun koostettuun arvoon. Jakamalla yhteinen osa voidaan potentiaalisesti säästää muistia.

```
flyweightFactory :: key -> item
```

Pienelle joukolle ennalta määrättyjä arvoja assosiaatiotaulu on riittävä rakenne. Muutoin tulee käyttää laiskaa laskemista sekä memoization -tekniikkaa.

```
memoize :: (Int -> a) -> (Int -> a)
memoize f = (map f [0 ..] !!)
```

Esimerkiksi Haskellin laiskuus mahdollistaa loputtoman tietorakenteen luomisen, jolla memoization voidaan toteuttaa.

Tehtaan tuottamat arvot liitetään useampaan koostettuun arvoon.

```
let sharedPart = flyweightFactory key

let item = -- use shared part with unique part to construct an item
```

Yhteenveto

Flyweight on ongelmallinen funktionaalisessa ohjelmoinnissa, ja ohjelmoinnissa ylipäätään. Sen toteutus vaatii laiskaa laskemista, ja on rakenteeltaan imperatiivista vastinetta monimutkaisempi.

Suurin ongelma *Flyweightin* kohdalla on sen riippuvuus ohjelmointikielen ja siitä käännettävän tai tulkattavan ohjelman välillä. Suunnittelumallin hyödyllisyys riippuu puhtaasti ohjelmointityylin ulkopuolisista toteutusteknisistä seikoista.

soveltuvuus	teennäinen
kompleksisuus	monimutkainen

INTERPRETER

Tarkoitus

Määrittää annetulle kielelle tulkki, jonka avulla kielen lauseet voidaan evaluoida arvoiksi.

Rakenne

Kieli sekä kielioppi määritellään algebrallisten tietotyyppien avulla.

```
data Expression = Literal Literal | Operation Operation
data Operation = Unary Expression | Binary Expression Expression
type Literal = ...
```

Kielen lauseet evaluoidaan funktioiden avulla.

```
evaluateExpression :: Expression -> Literal
evaluateOperation :: Operation -> Literal
evaluateLiteral :: Literal -> Literal
```

Yhteenveto

Funktionaalisessa ohjelmoinnissa kielen ja kieliopin määritteleminen on helppoa. Sen evaluoiminen vaatii kielen tietorakenteiden kanssa yhteensopivia funktioita.

Interpreterin toteutus funktionaalisesti on yksinkertaisempaa kuin olioperustaisen ohjelmoinnin menetelmin. Suunnittelumalli ei vaadi luokkia tai muuttuvaa tilaa.

soveltuvuus	luonnollinen
kompleksisuus	normaali

ITERATOR

Tarkoitus

Mahdollistaa tietorakenteen alkioiden läpikäynti järjestyksessä, paljastamatta sen konkreettista rakennetta.

Rakenne

Iterator koostuu monimuotoisista funktioista ja niiden kanssa yhteensopivista tietorakenteista.

```
-- define interface for all Iterable containers
class Iterable t where
    forward :: t a -> t a
    value :: t a -> a

-- specify a container to belong to Iterable class
instance Iterable T where
    forward container = ...
    value container = ...
```

Asiakasfunktio vaatii ainoastaan, että sille annettu tietorakenne toteuttaa Iterablen määrittämät funktiot.

```
clietnFunction :: (Iterable Container) => Container ...
clientFunction container ... =
    -- forward container or value container calls are possible
```

Haskellissa funktioiden monimuotoisuuden työkaluna ovat tyyppiluokat. Mikäli ohjelmointikieli ei tue lainkaan funktioiden monimuotoisuutta, pitää funktiolle antaa parametreina tietorakenteen käyttöön tarvittavat funktiot.

Yhteenveto

Iteratorin toteutus funktionaalisessa ohjelmoinnissa on hankalaa, mikäli sen halutaan selaavan tietorakennetta läpi askel askeleelta. Tämä vaatii käytännössä tilan ylläpitämistä funktiossa *forward,* joka funktionaalisesti tapahtuu luomalla uusi tietorakenne. Tekniikkaa kutsutaan nimellä *Zipper*.

Funktionaalisessa ohjelmoinnissa samantyyppisiä alkioita sisältäviä tietorakenteita on luonnollisempaa käyttää askeltamatta. Riippuen käyttötarkoituksesta, tähän voidaan käyttää esimerkiksi funktioita *map* ja *filter*, jotka käsittelevät koko tietorakenteen kerralla.

soveltuvuus	teennäinen (luonnollinen, jos ei askellusta)
kompleksisuus	monimutkainen (normaali, jos ei askellusta)

MEDIATOR

Tarkoitus

-

Rakenne

-

Yhteenveto

Mediator liittyy olioiden väliseen kommunikointiin. Funktionaalisessa ohjelmoinnissa käsite ei ole mielekäs, sillä funktiot eivät kommunikoi samaan tapaan keskenään.

soveltuvuus	soveltumaton
kompleksisuus	-

MEMENTO

Tarkoitus

-

Rakenne

-

Yhteenveto

Mementon kuvaama tilan käsite ei sovellu funktionaaliseen ohjelmointiin. Sen käyttökohteet ovat IO-operaatioissa, joissa tila halutaan palauttaa.

soveltuvuus	soveltumaton
kompleksisuus	-

OBSERVER

Tarkoitus

-

Rakenne

-

Yhteenveto

Observer liittyy olioiden väliseen kommunikointiin. Funktionaalisessa ohjelmoinnissa käsite ei ole mielekäs, sillä funktiot eivät kommunikoi samaan tapaan keskenään.

soveltuvuus	soveltumaton
kompleksisuus	-

PROTOTYPE

Tarkoitus

Luoda uusia tietoalkioita kopioimalla prototyyppiä.

Rakenne

Prototypen ydin on mallifunktio, joka ottaa parametreinaan prototyypin sekä funktion, joka osaa muuntaa injektoitua prototyyppiä.

```
clonerTemplate :: Prototype -> (Prototype -> a -> Prototype) -> a -> Proto-
type
clonerTemplate prototype function parameter = function prototype parameter

cloner :: a -> Prototype
cloner parameter = clonerTemplate someProptotype someFunction
```

Perustuen funktiolle *cloner* annettuun parametriin, muutetaan prototyypistä kloonattua arvoa. Sen tyyppi pysyy kuitenkin muuttumattomana.

Yhteenveto

Funktionaalisessa ohjelmoinnissa arvot ovat vakioita. Arvon muuttaminen on mahdollista ainoastaan luomalla siitä ensin kopio. Tässä mielessä *Prototypen* idea on sisäänrakennettu funktionaaliseen ohjelmointiin. Sen alkuperäinen tarkoitus on kuitenkin kätkeä asiakaskoodilta luotavien tietoalkioiden konkreettinen tyyppi. Tämä ajatus ei tunnu sopivan luonnollisena osana funktionaaliseen ohjelmointiin.

soveltuvuus	teennäinen
kompleksisuus	triviaali

PROXY

Tarkoitus

Asettaa funktiolle edustaja, joka kontrolloi sen käyttöä. Alkuperäisen funktion ja edustetun funktion tyypit ovat samat.

Rakenne

Proxy on edustajafunktion ja ydinfunktion yhdistelmä. Oleellista on, että kyseisten funktioilla on sama tyyppi. Tämän ansiosta asiakas ei tiedä onko välissä edustaja.

```
coreFunction :: a -> b -> c
proxyFunction :: a -> b -> c
```

Edustajafunktio on mahdollista toteuttaa mallifunktion tai sulkeuman avulla.

```
proxyFunctionTemplate :: (a -> b -> c) -> a -> b -> c
proxyFunctionTemplate coreFunction a b =
    -- do something before calling the coreFunction
    let c = coreFunction a b
    -- do something after calling the core function
```

Muuttamalla miten *proxyFunctionTemplate* toteutetaan, saadaan edustaja tekemään erilaisia asioita. Se voi esimerkiksi rajoittaa edustamansa funktion käyttöä tai tallentaa tulokset välimuistiin suorituskyvyn parantamiseksi.

Yhteenveto

Proxyn käyttökohteet funktionaalisessa ohjelmoinnissa vastaavat sen olioperustaista esikuvaa. Se on erittäin kevyt rakenteinen, ja sen käyttö on yksinkertaista.

soveltuvuus	luonnollinen
kompleksisuus	triviaali

SINGLETON

Tarkoitus

-

Rakenne

-

Yhteenveto

Singleton pyrkii tarjoamaan yksittäisen instanssin edustamastaan tyypistä. Funktionaalisessa ohjelmoinnissa sekä data että funktiot ovat muuttumattomia, joten instanssi ei ole mielekäs käsitteenä.

soveltuvuus	soveltumaton
kompleksisuus	-

STATE

Tarkoitus

Muuttaa toimintaa riippuen laskennan tilasta.

Rakenne

Asiayhteys vaikuttaa suunnittelumallin rakenteeseen. Pelkän laskennan välituloksen lisäksi (*result*) tilaan liitetään seuraavan alkion käsittelevä funktio (*handler*).

```
data State = State { handler :: State -> T -> State; result :: ResultType }
```

Laskennan edetessä tilanmuutos tapahtuu, kun funktio *handler* palauttaa tilan, jonka funktio *handler* on eri kuin se itse.

Yhteenveto

State liittyy vahvasti tilan käsitteeseen. Funktionaalisella ohjelmalla ei ole tilaa, mutta sen laskennan vaiheilla voi sellainen olla. Muuttamalla laskennan seuraavan alkion käsittelyä edellisten vaiheiden välituloksen perusteella, saadaan suunnittelumallin tarkoitus hyödynnettyä myös funktionaalisessa ohjelmoinnissa.

soveltuvuus	teennäinen
kompleksisuus	triviaali

STRATEGY

Tarkoitus

Määrittää joukko toistensa kanssa vaihdannaisia algoritmeja kapseloimalle ne omiin funktioihinsa.

Rakenne

Strategyn rakenne on joukko funktioita, joilla on sama tyyppi ja jotka suorittavat loogisesti saman tehtävän.

```
type Strategy :: a -> b ->

fooStrategy :: Strategy
fooStrategy a b = ...

barStrategy :: Strategy
barStrategy a b = ...
```

Yksittäinen algoritmi on mahdollista tämän jälkeen injektoida asiakasfunktioon joko soveltamalla osittain tai välittämällä se argumenttina. On mahdollista myös välittää listallinen algoritmeja asiakasfunktiolle, joka valitsee haluamansa algoritmin tarpeensa mukaan.

Yhteenveto

Strategy on funktionaalisen ohjelmoinnin perusrakenteita. Sen käyttö on triviaalia ja joustavaa.

soveltuvuus	luonnollinen
kompleksisuus	triviaali

TEMPLATE METHOD

Tarkoitus

Määrittää algoritmin runko, jonka osavaiheet ovat parametrisoitavissa.

Rakenne

Template Method on korkeamman tason funktio. Se ottaa parametreinaan funktiot, joita se kutsuu suorittaakseen halutut osavaiheet.

```
type FunctionA :: a -> b
type FunctionB :: c -> d -> e

templateMethod :: FunctionA -> FunctionB -> x -> y -> z
```

Sen runko voi koostua kokonaan tai osittaan kutsuttavista välivaiheista.

```
templateMethod functionA functionB x y =
    -- do something
    -- call functionA
    -- do something more
    -- call functionB
    -- do final calculation
```

Yhteenveto

Template Method pelkistyy funktionaalisessa ohjelmoinnissa korkeamman tason funktioksi. Sen rakenne on joustavampi kuin olioperustaisen esikuvansa, sillä se voidaan parametrisoida millä tahansa kombinaatiolla funktioita, jotka suorittavat sen osavaiheet.

soveltuvuus	luonnollinen
kompleksisuus	triviaali

VISITOR

Tarkoitus

Mahdollistaa operaatioiden lisäämisen tietorakenteen alkioille, muuttamatta tietorakennetta.

Rakenne

Visitor koostuu kahdesta osasta, alkioille suoritettavasta operaatiosta (*visitor*) sekä tietorakenteen läpikäyvästä funktiosta (*traverser*).

Funktio *visitor* omaa seuraavan tyypin, kun vierailtavan tietorakenteenalkiot ovat tyyppiä `T`.

```
visitor :: State -> T -> State
visitor state item =
    -- alter the state according to the received item
```

Funktion *traverser* tyyppi riippuu vierailtavasta tietorakenteesta. Listalle, joka sisältää alkioita `T`, sen tyyppi on seuraava.

```
traverser :: (State -> T -> State) -> State -> [T] -> State
traverser visitor initialState dataStructure =
    -- recursively go through every item on the data structure and
    -- pass it to the visitor
```

Sama funktio *visitor* toimii myös muille tietorakenteille, joiden tyyppi on `T`. Tällöin ainoastaan funktio *traverser* joudutaan muuttamaan.

Yhteenveto

Visitor tunnetaan funktionaalisessa ohjelmoinnissa funktiona *fold*. Se on kahden asian yhdistelmä. Sen ensimmäinen osa on funktio, joka selaa tietorakenteen läpi, ja toinen osa on funktio, joka suorittaa operaation jokaiselle tietorakenteen alkiolle.

Operaation suorittava funktio on luonnostaan yleiskäyttöinen, ja toimii kaikille tietorakenteille, joiden alkiot ovat samaa tyyppiä. Funktionaaliset ohjelmointikielet tarjoavat listoille oletuksena rekursiiviset funktiot, joilla ne on mahdollista käydä läpi.

soveltuvuus	luonnollinen
kompleksisuus	normaali