Axen Jari
Hyvärinen Jaakko

# SUUNNITTELUMALLIT OHJELMISTOTUOTANNOSSA

## TEORIA, KÄYTÄNTÖ JA KOKEMUKSET

Tietojärjestelmätieteen
kandidaatintutkielma
19.5.2004

Jyväskylän yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä

## TIIVISTELMÄ

Jari Axen <jari.axen@iki.fi>

Jaakko Hyvärinen <japahyva@cc.jyu.fi>

Suunnittelumallit ohjelmistotuotannossa teoria, käytäntö, kokemukset

Jyväskylä: Jyväskylän yliopisto, 2004.

80 s.

Kandidaatintutkielma

Suunnittelumallit ovat nimettyjä ja hyvin dokumentoituja ratkaisuja toistuviin ongelmiin tietyssä asiayhteydessä. Suunnittelumallit voidaan kuvata ja luokitella yhtenäisellä tavalla.

Tässä tutkielmassa esitetään tapa, jolla suunnittelumallit yleensä kuvataan ja erilaisia vaihtoehtoisia tapoja luokitella suunnittelumalleja. Suunnittelumallien käyttöä kokeillaan ja havainnollistetaan esimerkkitapauksella. Esimerkkitapauksessa ilmenneet havainnot esitetään tutkielmassa ja niitä verrataan alan kirjallisuudessa esitettyyn suunnittelumalleja koskevaan teoriaan.

Tutkielmassa pohditaan suunnittelumallien käyttöä ja niiden merkitystä ohjelmistotuotannossa. Tutkielman lopuksi esitellään suunnittelumallien pohjalta kehitetyt vastamallit (engl. antipatterns), jotka kuvaavat yleisiä ja useasti toistuvia ohjelmistotuotannon ongelmia.

**AVAINSANAT:** Suunnittelumalli, Design Pattern, Suunnittelumallien luokittelu ja kuvaaminen.

# SISÄLLYSLUETTELO

1 JOHDANTO ........ 1
1.1 Mikä on suunnittelumalli ........ 3
1.2 Alexanderin näkemys suunnittelumalleista ........ 5

2 SUUNNITTELUMALLIEN KUVAAMINEN ........ 7
2.1 Gamma muoto ........ 7
2.2 Suunnittelumalliesimerkki ........ 10
2.3 Yhteenveto ........ 12

3 GRAAFISTEN KÄYTTÖLIITTYMIEN OHJELMOINTI KURSSI ........ 14
3.1 Delphi ........ 15
3.2 Harjoitustyö ........ 17
3.3 Yhteenveto ........ 18

4 SUUNNITTELUMALLEJA TRIVIAALISSA PURSUITISSA ........ 19
4.1 Strategia (Strategy) ........ 20
4.2 Rakentaja (Builder) ........ 28
4.3 Ainokainen (Singleton) ........ 34
4.4 Vierailija (Visitor) ........ 37
4.5 Double-dispatch ........ 46
4.6 Yhteenveto ........ 46

5 SUUNNITTELUMALLIEN LUOKITTELU ........ 48
5.1 Gamma-luokittelu ........ 48
5.2 Siemensin luokittelu ........ 51
5.3 Luokittelun merkitys ........ 55
5.4 Suunnittelumallien kaltaiset mallit ........ 56
5.5 Yhteenveto ........ 59

6 MALLIEN KÄYTTÄMINEN OHJELMISTOTUOTANNOSSA ........ 60
6.1 Ohjelmistotuotannon kehittyminen ........ 60
6.2 Mallien käyttäminen ........ 62
6.3 Malleista oppiminen ........ 63
6.4 Suunnittelumallien louhiminen ........ 64
6.5 Suunnittelumallien yleisiä käyttökokemuksia ........ 65
6.6 Omia käyttökokemuksia ........ 67
6.7 Suunnittelumallien käytön etuja ja haitat ........ 68
6.8 Yhteenveto ........ 71

7 VASTAMALLIT ........ 72

7.1 Mitä ovat vastamallit ..... 72
7.2 Vastamallien kuvaaminen ..... 72
7.3 Vastamallien käyttäminen ..... 73
7.4 Vastamalliesimerkki: leikkaa ja liitä -ohjelmointi ..... 73
7.5 Yhteenveto ..... 75

8 YHTEENVETO ..... 76

9 LÄHTEET ..... 78

# 1 JOHDANTO

Ohjelmistokriisin (Naur ja Randell 1969) keskeisinä aiheuttajina voidaan pitää seuraavia Brooksin (1987) artikkelissa ”No silver bullet – essence and accidents of software engineering” esittämiä ohjelmistoja koskevia asioita:

**Monimutkaisuus** (engl. complexity), joka tarkoittaa ohjelman osien välisten riippuvuuksien ja tilojen suurta määrää.

**Muunnettavuus** (engl. changeability), joka tarkoittaa ohjelmistojen ”helppoa” muunnettavuutta siihen kohdistuvien vaatimusten muuttuessa.

**Mukautuvuus** (engl. conformity), jolla viitataan siihen, että ohjelmisto on järjestelmän mukautuvin osa.

**Näkymättömyys** (engl. invisibility), joka tarkoittaa sitä, että ei ole mahdollista kuvata ohjelmistoa helposti käyttäen yleiskäyttöistä esitystä.

Nämä asiat muodostavat yhdessä ohjelmistoja koskevan yleisen ongelman, joka voidaan Brooksin (1987) mukaan nähdä ihmissutena. Ihmissuden tappamiseksi tarvitaan kansanperinteen mukaan hopealuoti. Olio-ohjelmointi on yksi Brooksin esittämistä toivoista löytää hopeaa tämän luodin valmistamiseen.

Olio-ohjelmointi on ollut yksi viime vuosikymmenen suosituimmista ohjelmointiparadigmoista. Se on muuttunut uudesta ja ihmeellisestä ohjelmointitavasta valtavirran ohjelmointimenetelmäksi. (Rintala ja ym. 2000)

Vesterholmin ja ym. (2003) mukaan olioperustaisuuden läpilyöntiin vaikuttaneista tekijöistä yksi on ollut ohjelmakoodin uudelleenkäyttö. Olio-ohjelmissa ohjelmakoodi ei ole kuitenkaan ainoa asia, jota voidaan käyttää uudelleen. Olio-

ohjelmien suunnitteluun on olemassa valmiita uudelleenkäytettäviä malleja. Näitä valmiita malleja kutsutaan suunnittelumalleiksi. Suunnittelumalli on yleisesti hyväksi todettu ratkaisu jonkin suunnitteluongelman ratkaisemiseen.

Kokeneet suunnittelijat yleensä tuntevat nämä ”patenttiratkaisut” ja soveltavat niitä jopa huomaamattaan. Kun mallit nimetään ja dokumentoidaan vakiomuotoon, niistä tulee suunnittelijoiden yhteinen ”kieli” ohjelmiston suunnitteluun ja dokumentointiin. (Haikala ja ym. 2000)

Tässä tutkielmassa esitellään suunnittelumallikäsite ja muutamia yleisiä suunnittelumalleja. Lisäksi kirjoitelmassa pohditaan suunnittelumallien käyttöä ja merkitystä ohjelmistotuotannossa. Kirjoitelmassa esitetyt ajatukset ja mallien kuvaukset pohjautuvat pitkälti Gamman ja ym. (1995) ”Olio-ohjelmointi suunnittelumallit” (Design Patterns Elements of Reusable Object-oriented Software) -kirjassa esiteltyihin ajatuksiin. Tämä alan ”raamattu” toi suunnittelumallit ohjelmistotuotantoon. Vaikka kirjassa esitetään vain oliomalleja, voidaan kirjan ajatukset malleista yleistää koskemaan koko ohjelmistotuotantoa. Esimerkkinä yleistämisestä voidaan mainita muut ohjelmointiparadigmat tai ohjelmistoarkkitehtuurit.

Toinen merkittävä tiedonantaja ja mielipiteiden muokkaaja on ollut Siemensin kirjaksi nimetty alan perusteos ”Pattern-Oriented Software Architecture – A System of Patterns” jonka on kirjoittanut Buschmann ja ym. (2002).

Kirjallisuudessa esitettyjä suunnittelumalleja tutkittiin esimerkkitapauksen avulla. Esimerkissä esitetään neljän mallin käyttäminen Triviaali Pursuitti –pelissä. Triviaali Pursuitti on kirjoittajien ohjelmoima peli, jossa on tietoisesti käytetty suunnittelumalleja.

## 1.1 Mikä on suunnittelumalli

Ohjelmistotuotannossa käytettävien suunnittelumallien ensimmäisenä ja suurimpana innoittajana pidetään yleisesti itävaltalaissyntyistä, nyttemmin yhdysvaltalaista, arkkitehtia Christopher Alexanderia. Alexander ja hänen kollegansa kirjoittivat 1970–luvun lopulla teoksen ”A Pattern Language: Towns, Buildings, Construction”. Kirjassa he esittävät useita malleja kaupunkien ja talojen rakentamiseen. Kirjan mallit (kaikkiaan 253) muodostavat niin sanotun Alexanderin mallikielen, jonka avulla ”jokainen” voi suunnitella itselleen esimerkiksi asuintalon. Pattern Language (mallikieli) voidaan suomentaa myös muotokieleksi ja Routio (2004) käyttääkin tuota termiä Taideteollisen korkeakoulun verkkojulkaisussa, jossa hän esittelee mm. Alexanderin malleja.

Alexander määrittää ohjeensa jokaisen mallin kohdalla samanlaisella formaatilla. Jokaisessa mallissa on kolmiosainen sääntö, joka ilmaisee ongelman kontekstin ja ratkaisun välisen suhteen. Tätä ohjeiden joukkoa voidaan kutsua mallijärjestelmäksi. Vaikka Alexanderin mallit tarkoittavatkin rakennusten ja kaupunkien suunnittelussa käytettäviä malleja, niin hänen määritelmänsä sopivat myös oliopohjaisiin suunnittelumalleihin. (Gamma ja ym. 1995)

Ohjelmistotuotannossa on monesti ongelmana se, että samat asiat keksitään monissa projekteissa uudelleen ja uudelleen. Toisin sanoen samankaltaiset ongelmat ja asiat toistuvat projektista toiseen. Vastaavan asian havaitsi myös Alexander malleja kehittäessään. Esimerkiksi useampikerroksisessa talossa on yleensä portaat kerroksien välillä – miten nämä portaat tulisi tehdä? Useasti kokemattomat ohjelmistojenkehittäjät tekevät samat virheet, jotka kaikkein kokeneimmatkin ohjelmistojenkehittäjät ovat aikanaan tehneet. Suunnittelijoiden kokemus ja taito kasvaa hiljalleen ajan kuluessa. Tätä taitoa tulisi pystyä käyt-

tämään tulevissa projekteissa ja taitoa tulisi pystyä siirtämään seuraaville suunnittelijasukupolville.

Ohjelmistotuotannon suunnittelumallit ovat nimettyjä ja hyvin dokumentoituja ratkaisuja toistuviin ongelmiin tietyssä kontekstissa. Tämä on kiteytetty vastaus, kun esitetään kysymys: mikä on suunnittelumalli? Tästä eteenpäin kirjoitelmassa suunnittelumalli sanalla tarkoitetaan ohjelmistotuotannon suunnittelumallia.

Suunnittelumallilla tulee olla jokin kuvaava nimi, josta suunnittelijat tunnistavat ko. mallin ja jonka avulla he pystyvät keskustelemaan samalla ”kielellä”. Suunnittelumallit kuvaavat ratkaisuja toistuviin ongelmiin. Tämä kuvaaminen tehdään tietyn määrätyn dokumentoinnin mukaisesti. Kirjallisuudessa (Gamma ja ym. 1995; Buschmann ja ym. 2002) esitetään, että suunnitteluratkaisua voidaan nimittää suunnittelumalliksi vasta, kun voidaan todistettavasti osoittaa, että mallia on käytetty kolmessa eri sovelluksessa ratkaisemaan sama suunnitteluongelma.

Kontekstilla voidaan ymmärtää, että suunnittelumallilla on aina tietty vuorovaikutteinen asema järjestelmässä. Puhutaan järjestelmän ja mallin välisistä voimista. Tietyn tyyppinen suunnitteluongelma ajaa käyttämään tiettyä mallia, joka taas vaikuttaa järjestelmään tietyllä tavalla.

Gamma ja ym. (1995) määrittelevät suunnittelumallin seuraavasti: ”Suunnittelumalli abstrahoi ja määrittää yleisen suunnittelurakenteen avainkohdat. Avainkohdat hyödyttävät uudelleenkäytettävän suunnitteluratkaisun luomista. Suunnittelumalli määrittää osallistuvat luokat ja ilmentymät, niiden roolit ja yhteistyön sekä vastuiden jakamisen.”

Suunnittelumalli on siis heidän mukaansa kuvaus keskenään vuorovaikutuksessa olevista olioista ja luokista, jotka on muotoiltu ratkaisemaan tietyssä yhteydessä esiintyvä yleinen suunnitteluongelma.

Kuvattiinpa mallia miten tahansa, tulisi kuvauksesta selvitä ainakin seuraavat asiat: 1) suunnittelumallin nimi, 2) konteksti, eli suunnittelutilanne, jossa ko. ongelma esiintyy, 3) ongelman kuvaus, jossa selitetään mitkä voimat ongelmaan liittyvät tässä kontekstissa, 4) ratkaisu, jossa selitetään voimien tasapainotus mahdollisimman järkevästi (komponenttien rakenne ja niiden väliset suhteet ja ratkaisun ajonaikainen käyttäytyminen).

## 1.2 Alexanderin näkemys suunnittelumalleista

Alexander kertoo 1996 OOPSLA:n puheessaan malleistaan seuraavasti: mallit luotiin auttamaan ihmisiä luomaan laadukas elinympäristö. Hänen mielestään on vähintäänkin outoa, kun hänen esittämät mallit pystytään näkemään myös ohjelmistoissa.

Alexander on kuitenkin mielissään siitä, että hänen ja kollegoidensa esittämiä ajatuksia malleista käytetään ohjelmistotuotannon suunnittelumallien yhteydessä. Tästä huolimatta hän toivoisi, että ohjelmistosuunnittelijat ja mallien kehittäjät ymmärtäisivät syvällisemmin hänen ajatuksensa malleista.

Alexander esitti mallit tarkoituksena parantaa ihmisten elämänlaatua. Mallien perimmäisenä tarkoituksena on auttaa tuottamaan elävää rakennetta. Alexander haluaisikin, että hänen mallit nähtäisiin myös muussa valossa, kuin vain yhtenäisenä mallien käsitteistönä (engl. pattern concept).

Malleilla on moraalinen puoli (engl. moral component), ja toiseksi ne pyrkivät luomaan yhtenäisen muoto-opillisen kokonaisuuden asioihin, jotka rakentuvat niillä. Lisäksi ne ovat generatiivisia: ne mahdollistavat ihmisten tuottaa näitä nykyään kadoksissa olevia yhteneväisiä muoto-opillisia kokonaisuuksia. Alexander ymmärtää ohjelmistotuotannossa käytettävät suunnittelumallit sikäli, kun ne käsittelevät esimerkiksi luokkia ja olioita ja näin ollen tekevät ohjelmistoista parempia. Mutta hän haluaa tehdä selkeän eron siitä, että hänen mallinsa ovat selkeästi osa jotain suurempaa kuin ohjelmistotuotannossa käytössä olevat suunnittelumallit.

Suunnittelumallit on saatettu nykypäivänä nostaa suuremmiksi ja ihmeellisimmiksi asioiksi kuin ne välttämättä todellisuudessa ovat. Buddin (2002) mukaan, suunnittelumallit eivät ole mitään muuta kuin pyrkimys dokumentoida hyväksi todettu ratkaisu ongelmaan, jotta tulevaisuudessa ongelmat voidaan helpommin ratkaista samalla yhtenäisellä tavalla. Näin ollen ohjelmistotuotannossa käytettävien suunnittelumallien rinnastaminen Alexanderin (1979) esittämiin malleihin ei välttämättä ole perusteltua.

# 2 SUUNNITTELUMALLIEN KUVAAMINEN

Graafiset esitystavat ja ohjelmakoodi ovat tärkeitä ja hyödyllisiä kuvaustapoja. Ne eivät yksistään silti ole riittäviä, sillä ne ainoastaan tallentavat suunnitteluprosessin lopputuotteen luokkien ja olion välisinä suhteina. Voidaksemme käyttää suunnitteluratkaisua uudelleen on meidän kirjattava ylös myös ratkaisuun johtaneet päätökset, suunnittelun aikana eteen tulleet vaihtoehdot sekä ratkaisun hyvät ja huonot puolet. (Gamma ja ym. 1995)

Suunnittelumalleja voidaan kuvata usealla tavalla. Yhtenäinen tapa, jolla mallit kuvataan muodostaa mallien kuvauskielen. Kokoamalla ja luokittelemalla malleja voidaan malleista muodostaa mallikieliä (engl. Pattern Language).

## 2.1 Gamma muoto

Yleisimpänä tapana kuvata suunnittelumalleja voidaan pitää Gamman ja ym. (1995) kirjassa esiteltyä tapaa. Tätä tapaa nimitetään yleisesti Gamma-muodoksi. Gamma-muodossa suunnittelumallin kuvaamisessa on 12 kohtaa. Siemens-kirjassa esitetään myös käyttökelpoinen tapa kuvata malleja, jossa Gamma-muotoa on joiltain osin jalostettu. Tässä kirjoitelmassa esimerkkimallit kuvataan kuitenkin Gamma-muodossa.

Gamma ja ym. (1995) esittelee mallit käyttäen yhtenäistä formaattia. Mallien kuvaaminen tapahtuu siten, että ensin annetaan spesifinen esimerkki mallin käytöstä ja sen jälkeen mallin käyttäminen yleistetään.

Jokainen malli jaetaan osiin seuraavan rungon mukaisesti. Runko luo tiedoille yhtenäisen rakenteen, joka helpottaa suunnittelumallien oppimista, ymmärtämistä ja käyttämistä.

## Mallin nimi

Mallin nimi kertoo olemuksen ytimekkäästi.

## Tarkoitus

Lyhyt lause, joka kertoo, mitä suunnittelumalli tekee ja mikä on sen perusajatus ja tarkoitus. Lisäksi kuvataan, mihin tiettyyn suunnittelukohteeseen tai ongelmaan se soveltuu.

## Alias

Mallin muita yleisesti tunnettuja nimiä, jos sellaisia on.

## Perustelut

Esimerkkitilanne, jossa kuvataan suunnitteluongelma ja se, miten mallin luokka- ja oliorakenteet ongelman ratkaisevat. Esimerkki auttaa ymmärtämään jäljessä seuraavan abstraktimman kuvauksen.

## Soveltuvuus

Mihin tilanteisiin suunnittelumallia voidaan soveltaa. Esimerkkejä huonoista suunnitteluratkaisuista, joita mallilla voidaan korjata ja siitä, miten nämä tilanteet tunnistetaan.

## Rakenne

Mallin luokkien graafinen esitys OMT-notaatiolla. Lisäksi vuorovaikutussuhteita kuvataan sekvenssikaavioilla.

### Osallistujat

Malliin liittyvät luokat ja/tai oliot sekä niiden vastuut.

### Yhteistyösuhteet

Miten osallistujat toimivat yhteistyössä toteuttaakseen vastuunsa?

### Seuraukset

Miten hyvin malli tukee tavoitteitaan? Mitä etuja mallin käytöllä saavutetaan ja mitä menetetään? Mitä systeemirakenteita malli sallii muutettavan riippumattomasti?

### Toteutus

Mitkä sudenkuopat, vinkit ja tekniikat on tiedettävä mallia toteuttaessa? Onko eri ohjelmointikielillä tehtyjen toteutusten välillä ohjelmointikielistä johtuvia eroja?

### Mallikoodia

Koodiesimerkkejä mallin toteutuksesta C++:lla tai Smalltalkilla.

### Tunnettuja käyttökohteita

Sisältää esimerkkejä mallin käytöstä todellisissa järjestelmissä. Annetaan vähintään kaksi esimerkkiä, jotka ovat eri alueilta.

### Läheiset mallit

Mitkä suunnittelumallit liittyvät läheisesti tähän malliin? Mitkä ovat tärkeimmät erot? Minkä muiden mallien kanssa tätä mallia tulisi käyttää?

## 2.2 Suunnittelumalliesimerkki

Seuraava esimerkki on otettu Rintalan ja Jokisen (2000) Olioiden ohjelmointi C++:lla –kirjasta (s.195-196). Alun perin suunnittelumalli esitettiin Gamma ja ym. (1995) kirjassa (s.257-271). Rintala ja Jokinen ovat supistaneet suunnittelumallin kuvausta tietyin osin.

### Iteraattori (engl. Iterator)

**Tarkoitus**

Tarjoaa rajapinnan kokoelman alkioihin viittaamiseen, joka on erillään itse kokoelman toteutuksesta.

**Tunnetaan myös nimellä:**

Kohdistin (engl. Cursor).

**Perustelu**

Listan tai muun alkiokokoelman rajapinta kasvaa helposti hyvin suureksi, jos siihen liitetään sekä rakenteen muokkaukseen että läpikäyntiin kuuluvat operaatiot. Yksi joukko läpiviennin operaatioita (alkuun, seuraava, edellinen ynnä muut) mahdollistaa vain yhden läpikäynnin olemassaolon kerrallaan, koska alkiokokoelman toteuttava olio säilyttää itse operaatioiden vaatiman tilatiedon. Kun vastuu alkiokokoelman läpikäynnistä irrotetaan erilliseksi (mutta kokoelmaan läheisesti liittyväksi) olioksi, saadaan ratkaistuksi molemmat ongelmat.

### Soveltuvuus

Tarvitaan useita yhtäaikaisia läpikäyntejä kokoelmaan tietoa tai halutaan tarjota yhtenäinen rajapinta useiden eri tavalla toteutettujen kokoelmien läpikäyntiin.

### Rakenne:

Katso kuvio 1.

### Osallistujat:

**Asiakas**

Käyttää rajapintaa Iteraattori luokan Kokoelma sisällä olevien alkioiden osoittamiseen ja läpikäyntiin.

**Kokoelma**

Määrittelee alkiokokoelman rajapinnan (erityisen tavan, jolla kokoelmaan saadaan luoduksi iteraattori).

**KokoelmaToteutus**

Toteuttaa edellä mainitun rajapinnan. Tarvitsee keinon luoda iteraattoriolion.

**Iteraattori**

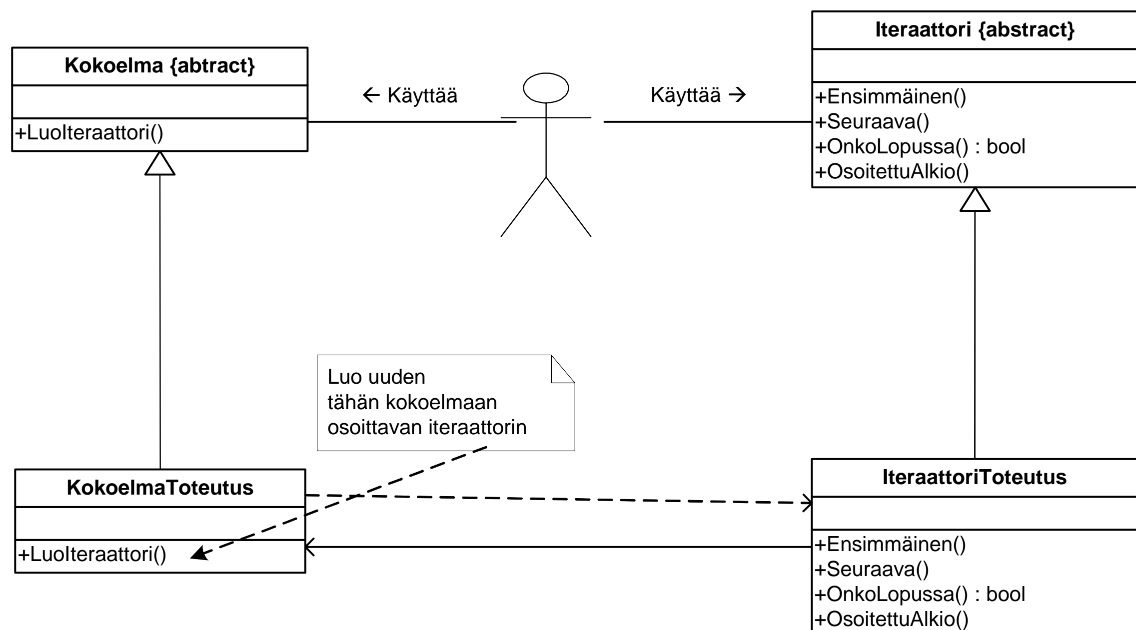
Määrittelee rajapinnan alkiokokoelman läpikäyntiin.

**IteraattoriToteutus**

Toteuttaa edellä mainitun rajapinnan. (KokoelmaToteutus palauttaa tämän luokan instanssin.)

**Seuraukset:**

Kokoelma pystyy tarjoamaan useita erilaisia tapoja alkioiden läpikäyntiin tarjoamalla useita iteraattoreita. Erillään oleva iteraattori yksinkertaistaa kokoelman rajapintaa. Kokoelman käyttäjä pystyy pitämään tarvittaessa useita osoituksia eli läpikäyntejä yhtä aikaa käynnissä.




**Kuvio 1. Iteraattori-suunnittelumalli**

## 2.3 Yhteenveto

Tässä luvussa esitettiin Gamman ja ym. (1995) esittämä tapa kuvata suunnittelumalleja. Luvun alussa kerrottiin, että on olemassa myös muita tapoja kuvata malleja. Gamma-muoto on saavuttanut alalla vakiintuneen aseman. Gamma-muodossa jokainen malli jaetaan tietyn rungon mukaisesti pieniin osiin. Tämä

runko luo suunnittelumalleihin liittyvästä tiedosta yhtenevän rakenteen, joka selkeyttää ja helpottaa mallien oppimista, ymmärtämistä ja käyttämistä. Luvun lopuksi esiteltiin esimerkkinä Iteraattori-suunnittelumalli.

# 3 GRAAFISTEN KÄYTTÖLIITTYMIEN OHJELMOINTI KURSSI

Graafisten käyttöliittymien ohjelmointi kurssi on Jyväskylän yliopistossa järjestettävä ohjelmoinnin jatkokurssi. Kurssin nimi ei kerro kurssin sisältöä, joten kurssin luennoitsijan Vesa Lappalaisen, sanoin mitä kurssi sisältää: ”Graafisten käyttöliittymien ohjelmointi - ehkä hämäävä, koska pääpaino on komponenteissa ja olio-ohjelmoinnissa”. Kurssilla käytettävä ohjelmointikieli on Delphi.

Olio-ohjelmointia kurssilla painotetaan, vaikkakin kaikkein äärimmäiset oliopa-radigman puolustajat pitävät Delphi-komponentteja lähinnä olio-ohjelmoinnin vastaisina niiden asetus- ja saantimetodien (engl. Get ja Set) takia. Allen Holub (2003) vastustaa asetus- ja saantimetodeita, koska niitä parempi tapa on minimoida olioiden välisen tiedon vaihto. Hänen mukaansa ylläpidettävyys on kääntäen verrannollinen suhteessa olioiden väliseen tiedon vaihdon määrään.

Perinteisesti olio-ohjelmissa olioiden ei pitäisi muulle ohjelmalle paljastaa sisäistä toteutustaan. Asetus- ja saantimetodit Holubin (2003) mukaan paljastavat olion toteutuksen rikkomalla tiedon piilotuksen (engl. Information hiding). Kurssiin kuuluu pakollinen harjoitustyö, josta Vesa Lappalainen kirjoittaa kurssin WWW-sivuilla seuraavasti: ”Harjoitustyössä pitäisi keskittyä nimenomaan tuottamaan komponentteja, joita muutkin voivat käyttää.” Harjoitustyössä siis yritetään luoda mahdollisimman yleiskäyttöisiä komponentteja.

## 3.1 Delphi

Object Pascal -ohjelmointikieltä kutsutaan nykyään Delphiksi (Borland, 2002). Graafinen ohjelmointityökalu kuten Delphi on ohjelmankehitysympäristönä (IDE) tärkeämpi kuin itse ohjelmointikieli (Cantù, 2003). Ohjelmankehitysympäristö on tehokas työkalu, totuttuaan siihen on hankala kuvitella palaavansa pelkkään tekstieditoriin. Mielestämme Delphi-ympäristö on työvälineenä tehokas ja käytettävä, kun sen hallitsee.

Delphi ohjelman rakenteesta kirjoittaa Swan (1999) näin: ”Delphi-sovellus koostuu yhdestä tai useammasta lomakeikkunasta, joihin visuaaliset komponentit sijoitetaan”. Komponentit ovat Delphin ydin ja ympäristön tehokkuus perustuu paljolti hyviin valmiisiin komponentteihin. Kukin komponentti, kuten esimerkiksi painike tai menu on itsessään olio. Kun komponentteja laitetaan lomakkeelle, saadaan uusia olioita (Lappalainen, 1998).

Delphin syntaksi ei ole kovin vaikea oppia ja Delphissä ajattelutapa on hyvin samanlainen kuin Javassa. Delphin lähdekoodia esittelemme perinteisellä ja yksinkertaisella ”hello world”-sovelluksella. Sovelluksessa on luotu nimiökomponentti (engl. `label`) ja sen seloste-ominaisuuden (engl. `caption`) arvoksi sijoitettu ”hello world”-teksti.

Aluksi Delphi ohjelmassa on rajapintaosa (engl. `interface`), joka esittelee luokan rajapinnan. `Uses`-lause tarkoittaa käytännössä samaa kuin `include` C++:ssa ja `import` Java:ssa, eli tuodaan kaikki yksikön (engl. `unit`) käyttämät muut yksiköt näkyviin. Sitten esitellään uusi luokka ja sen näkyvyysmääreet. Toinen osa on toteutusosa (engl. `implementation`), jossa toteutetaan luokan esittelyssä määrätyt metodit. `{$R *.dfm}` tarkoittaa, että yksikön mukaan

käännetään sitä vastaava dfm-tiedosto. Dfm-tiedostosta syntyy käännöksen tuloksena resurssitiedosto, joka linkitetään .exe-tiedostoon. Linkitettävästä resurssitiedostosta löytyy lomakkeen visuaalinen kuvaus. Delphissä metodit alkavat `begin`-sanalla ja päättyvät `end`-sanaan.

Sovelluksen `FormCreate`-metodissa nimiö luodaan ja sen omistajaksi määrätään lomake (engl. `form`). Tämän jälkeen nimiölle asetetaan isäntä (engl. `parent`) Isäntä määrittää minkä komponentin päällä nimiö sijaitsee. Delphin muuttuja on vain viite kuten Javassakin.

Ohjelmakoodi 1:ssa esitetään edellä mainittu esimerkki.

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
    ExampleLabel:TLabel;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
    ExampleLabel:=TLabel.Create(self);
    ExampleLabel.Parent:=self;
    ExampleLabel.Caption:='Hello world';
end;

end.
```

**Ohjelmakoodi 1. Hello world esimerkki**

## 3.2 Harjoitustyö

Harjoitustyönämme oli Triviali Pursuitti -sovellus, jossa perinteinen Trivial Pursuit -lautapeli, muutettiin tietokonepeliksi. Triviaali Pursuitti on tietokonepeli, jossa vastaillaan kysymyksiin ja yritetään kerätä värejä. Pelin voittaa pelaaja, joka ensimmäisenä kerää kaikki värit. Erilaiset värit merkitsevät erilaisia kysymyksiä, ja pelaaja voi kerätä värejä vastaamalla kysymykseen oikein tietyssä ruudussa. Mikäli pelaaja on tavallisessa ruudussa, niin oikealla vastauksella saa uuden vuoron. Sovellus on Windowsissa toimiva ohjelma, joka on rakennettu perinteisistä Windows-komponenteista ja harjoitustyötä varten tuotetuista komponenteista.

Ohjelma rakentuu erilaisista moduuleista: pelilauta, noppa, pelaaja ja kysymykset. Moduulit on rakennettu niin vähäisellä kytkennällä toisiinsa, että niitä voi käyttää yksinään eri sovelluksissa.

Noppa-moduuli on kolmiulotteinen graafinen noppa. Pelaaja-moduulissa hoidetaan pelaajan tietojen tallentaminen ja näyttäminen. Kysymys-moduulilla voidaan ottaa ohjelmaan erilaisia kysymyskorttipakkoja, jotka siis sisältävät kysymyksiä. Kysymykset luetaan tiedostoista, jotka määrittelevät kysymyssarjat. Kysymykset ovat aina monivalintakysymyksiä. Yksittäinen kysymys on olio, joka luo itse käyttöliittymänsä riippuen siitä, kuinka paljon sille on asetettu vastausvaihtoehtoja. Pelilauta-moduuli määrittelee pelilaudan. Harjoitustyössä toteutettiin Trivial Pursuit -pelilauta. Pelilauta-moduuli mahdollistaa kohtuullisen pienillä muutoksilla esimerkiksi Monopoli-pelilaudan tekemisen. Pelilauta-moduuli määrittelee myös pelilaudan geometrian, ja harjoitustyössä tuotettiin neliön- ja ympyränmuotoiset pelilaudat.

## 3.3 Yhteenveto

Tässä luvussa ensiksi esiteltiin kurssia johon esimerkkisovellus tuotettiin. Sitten käytiin toteutuksessa käytetyn Delphi-ohjelmointikielen peruskäsitteitä läpi. Lopuksi esiteltiin harjoitustyötä yleisesti ja esimerkkisovelluksen rakenne.

# 4 SUUNNITTELUMALLEJA TRIVIAALISSA PURSUITISSA

Jotta lukija saisi paremman käsityksen suunnittelumalleista, käsitellään tässä luvussa neljä erilaista Gamman ja ym. (1995) esittelemää suunnittelumallia. Suunnittelumalleja kokeiltiin edellisessä luvussa kuvatussa peliesimerkissä. Esimerkissä käytetyt mallit kuvataan Gamma-muodossa. Käytetyt suunnittelumallit ovat mikrosovelluskehyksiä.

Esimerkkimalleina esitetään Strategia (engl. Strategy), Rakentaja (engl. Builder), Ainokainen (engl. Singleton) ja Vierailija (engl. Visitor). Strategia- ja Rakentaja-malleja on käytetty pelilautakomponentissa.

Koska pelin toteuttamiskieli oli Delphi, niin Strategia- ja Rakentaja-mallin ilmentymät toteutettiin itsenäisinä komponentteina, jotka voidaan asentaa osaksi kehitysympäristöä. Komponentit nimettiin mallien mukaan, mikä ei ole hyvä ratkaisu. Strategia-komponentille kuvaavampi nimi olisi ollut esimerkiksi geometria. Pelilaudasta ohjelmoitiin myös vastaava komponentti. Strategian tehtävänä on ohjata Rakentajaa rakentamaan oikeisiin koordinaatteihin peliruutuja laudalle. Kun pelilautakomponenttia halutaan käyttää, voidaan se vetää visuaalisesti lomakkeelle ja tämän jälkeen siihen kytketään vastaavasti lomakkeelle vedetyt Strategia ja Rakentaja.

Ainokainen-suunnittelumallia käytettiin toteuttamaan luokka, joka sekoittaa pelin kysymyspakat ja jokaisen yksittäisen kysymyksen vastausvaihtoehtojen järjestyksen. Ainokainen-suunnittelumallia käyttämällä voitiin luopua kokonaan funktioista. Yksittäisen funktioiden käyttö sotii olio-ohjelmointiparadigmaa vastaan.

Vierailija on otettu esimerkkinä huonosta mallin käytöstä. Vierailija-suunnittelumallia yritettiin soveltaa pelilaudan ruutujen piirtämiseen ja pelilogiikan toteuttamiseen. Idea Vierailijan käyttämiseen tuli harjoitustyötä ohjanneelta ohjaajalta, ja sen käyttäminen osoittaa kuinka ongelman kannalta väärän mallin soveltaminen hankaloittaa ohjelmointia tuottamalla vaikeaselkoisia rakenteita ja ylimääräisiä rivejä.

Seuraavat suunnittelumallien kuvaukset perustuvat Gamman ja ym. (1995) kirjassa esitettyihin kuvauksiin, mutta niitä on tarkennettu tietyiltä osin.

## 4.1 Strategia (Strategy)

Tällä mallilla määritetään pelilaudan geometria.

### Tarkoitus

Määrittelee algoritmiperheen, kapseloi kunkin algoritmin ja tekee niistä keskenään vaihdettavia. Algoritmia voidaan muuttaa muuttamatta sovellusta, joka sitä käyttää.

### Perustelut

On olemassa useita erilaisia algoritmeja, joilla pelilaudan geometria voidaan muodostaa. Kaikkien tällaisten algoritmien sisällyttäminen pelisovellukseen on huono ratkaisu useasta syystä:

Pelilautakomponentista tulee erittäin monimutkainen, jos siihen liitetään kiinteästi geometrian luontialgoritmi.

Eri geometria-algoritmit sopivat eri peleille. Esimerkiksi monopolipeliin olisi järjetöntä sisällyttää Afrikan tähti -laudan geometrian luomisalgoritmi. Peli-

lautakomponenttiin ei kannata sisällyttää kaikkia geometria-algoritmeja, mikäli niitä ei käytetä.

Mikäli geometria algoritmi liitetään kiinteästi pelilautaan, on sen muuntaminen hankalaa. Monopolipelilauta esimerkiksi tarjoaa mahdollisuuden tuottaa vain Monopolipelilautoja, eikä toimi yleisenä geometriasta vapaana pelilautana.

Usein Strategian käytölle löytyy perusteluja jos algoritmi on tarpeellista vaihtaa ajon aikana. Triviaalissa Pursuitissa tämä on mahdollista, mutta ei kovin olennaista pelin pelaamisen kannalta.

**Soveltuvuus**

Joukko toisiinsa liittyviä luokkia eroaa toisistaan vain käyttäytymiseltään. Strategian avulla luokkaan saadaan liitettyä yksi useasta tarjolla olevasta käyttäytymisistä.

Algoritmista tarvitaan erilaisia variaatioita. Algoritmeilla voi esimerkiksi olla erilaisia muisti- ja suoritusaikavaatimuksia. Strategia-mallia voidaan käyttää, jos variaatiot toteutetaan algoritmeista muodostetussa luokkahierarkiassa.

Algoritmi käyttää tietoja, joista sovelluksen ei pidä tietää mitään. Strategia-mallia käyttäessä voidaan välttää paljastamasta monimutkaista algoritmikohtaisia tietorakenteita.

Luokka määrittelee erilaisia käyttäytymisiä ja käyttäytymiset ilmenevät sen operaatioissa useina ehtolausekkeina. Useiden ehtolausekkeiden käytön sijaan siirretään ehdolliset haarat omiin strategialuokkiinsa.

### Osallistujat

**Strategia,”geometria”**

Määrittelee kaikille algoritmeille yhteisen rajapinnan.

**YmpyräStrategia, NelioStrategia**

Toteuttaa yksittäisen algoritmin Strategia-rajapinnan mukaisesti.

**Pelilauta**

Konfiguroidaan yhteen konkreettisen Strategia-olion kanssa.

Vie esimerkiksi Rakentajan Strategialle.

Pitää yllä viitettä Strategiaan.

### Seuraukset

**Strategiat tekevät ehtolauseet tarpeettomiksi.**

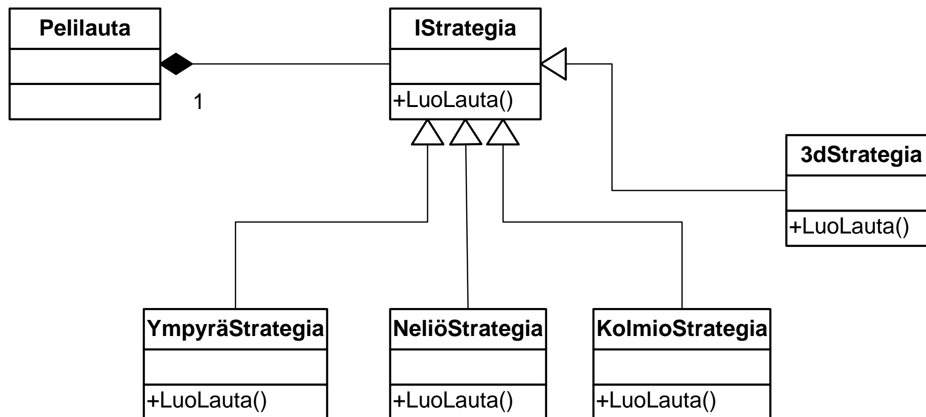
Strategia-malli tarjoaa vaihtoehdon ehtolausekkeiden käytölle halutun käyttäytymisen valinnassa. Jos yhteen luokkaan on kasattu erilaisia käyttäytymistapoja, on vaikea välttyä käyttämästä ehtolausekkeita oikean käyttäytymisen valitsemiseksi. Käyttäytymisen kapselointi erilaisiin Strategia-luokkiin tekee ehtolausekkeet tarpeettomiksi.

**Yhteen liittyvien algoritmien perheet.**

Strategia-luokkien hierarkia määrittelee algoritmiperheen ja käyttäytymisjoukon, jota konteksti voi käyttää.

**Vaihtoehto periytymisen käyttämiselle.**

Kuviossa 2 esitetään Strategia -mallin luokkien väliset suhteet.




**Kuvio 2. Strategia-mallin luokkien väliset suhteet**

Ohjelmakoodissa 2 esitetään `IStrategia`-rajapinnan toteutus.

```
unit strategia;
interface
uses rakentaja,ruutu,Classes,types;
type IStrategia= interface
    procedure LuoLauta;
    function AnnaLista:TList;
    procedure AsetaKoko(IClientrect: Trect);
end;

implementation
end.
```

**Ohjelmakoodi 2. IStrategia-rajapinta**

Ohjelmakoodissa 3 esitetään `TNelioStrategia`-komponentin toteutus suunnittelumallin kannalta olennaisin osin.

```
unit trivaalistrategia;
interface
uses
  SysUtils, Classes, strategia, ruutu, rakentaja,types;

type
  TNelioStrategia = class(TComponent,iStrategia)
private
    FRuutujaSivulla:integer;
    FKoko:integer;
    aloitus:TTriviaaliRuutu;
    ruudut:TList;
    FRakentaja :IRakentaja;
    procedure SetRakentaja(const Value: IRakentaja);
public
  constructor Create (AOwner: Tcomponent);overload; override;
  constructor Create (AOwner: Tcomponent; Rakentaja:IRakentaja); overload;
procedure LuoLauta;
  function AnnaAloitus:TRuutu;
  function AnnaLista:TList;
  procedure AsetaKoko(IClientrect: Trect);
published
  property Rakentaja: IRakentaja read FRakentaja write SetRakentaja;
  property Ruudunkoko: integer read FKoko write FKoko default 60;
  property Ruutujasivulla: integer
           read FRuutujaSivulla write FRuutujasivulla default 0;
end;

procedure Register;

implementation

.
.
.

function TNelioStrategia.AnnaLista: TList;
begin
 result := nil;
 if not assigned (Rakentaja) then exit;
 ruudut:= Rakentaja.AnnaRuudut;
 result :=ruudut;
end;

procedure TNelioStrategia.LuoLauta;
var i:integer;
begin
  if not assigned (Rakentaja) then exit;
  for i:=0 to RuutujaSivulla do begin
    Rakentaja.RakennaRuutu(ruudunkoko*i,0);
   end;
  for i:=1 to RuutujaSivulla-1 do begin
   Rakentaja.RakennaRuutu(ruudunkoko*RuutujaSivulla,Ruudunkoko*i);
  end;
  for i:=0 to RuutujaSivulla do begin
   Rakentaja.RakennaRuutu
(Ruudunkoko*Ruutujasivulla-Ruudunkoko*i,Ruudunkoko*RuutujaSivulla);
  end;
  for i:=1 to RuutujaSivulla-1 do begin
    Rakentaja.RakennaRuutu(0,Ruudunkoko*RuutujaSivulla-Ruudunkoko*i);
  end;
end;

procedure TNelioStrategia.SetRakentaja(const Value: IRakentaja);
begin
  FRakentaja := Value;
end;
end.
```

**Ohjelmakoodi 3. TNeliostrategia**

Ohjelmakoodissa 4 esitetään `TYmpyraStrategia`-komponentin `LuoLauta`-metodin toteutus.

```
procedure TYmpyraStrategia.LuoLauta;
var x:double;
    apu:TPoint;
    sade:double;
    askellusvali:integer;
begin
  Scale;
  sade:=round((2*PI*((clientRect.BottomRight.X-clientRect.TopLeft.X)/2)));
  askellusvali:=round(sade/ruudunkoko)-2;
  x := 0 ;
  while ( x < 360 ) do begin
    apu:= A.Map(cos(x*PI/180)*8,sin(x*PI/180)*8);
    rakentaja.rakennaRuutu(apu.X-round(ruudunkoko/2),apu.Y-round(ruudunkoko/2));
    x := x + 360/askellusvali*1.7;
  end;
end;
```

**Ohjelmakoodi 4. TYmpyraStrategia.LuoLauta**

## Ohjelmakoodin tulkinta

Strategia toteutettiin määrittämällä Strategia -rajapinta. Erilaisten strategioiden pitää toteuttaa tämä rajapinta. Rajapinnassa määriteltiin `LuoLauta`-, `AnnaLista`- ja `AsetaKoko`-metodit. Rajapinta tarvitaan, kun halutaan käyttää erilaisia strategioita. Strategia määrää pelilaudan geometrian. Toteutimme kaksi erilaista geometriaa (ympyrän ja neliön). `LuoLauta`-metodi rakentaa laudan tietynlaisella algoritmilla ottamatta kantaa millaisesta pelilaudasta (esimerkiksi Triviaali Pursuitti tai Monopoli) on kyse. Strategialla on Rakentaja ominaisuutena (engl. `property`), jota se kutsuu `LuoLauta`-metodissa, ja tämä Rakentaja tietää millainen laudasta tulee, ottamatta kantaa laudan geometriaan. `LuoLauta`-metodissa toteutetaan vain pelilaudan geometria. Tässä yhteydessä suunnittelumalli toimii hyvin, koska tällöin on mahdollista dynaamisesti vaihtaa pelilaudan geometria. Laudan luontialgoritmeja emme selitä, koska suunnittelumallien kannalta se olisi turhaa.

`AnnaLista`-metodi palauttaa listan niistä ruuduista, jotka on rakennettu, ja joissa Pelilauta koostuu. Strategia-suunnittelumallin käyttäminen yksinkertaistaa ohjelmakoodia huomattavasti. Ohjelmakoodissa 5 esitetään tilanne, jossa hahmotelma pelilaudan geometrian luontialgoritmista (Strategia) on ohjelmoitu kiinteästi pelilauta-moduuliin.

```
unit pelilauta;

type
  TForm1 = class(TForm)
implementation

procedure TForm1.Button1Click(Sender: TObject);
var apu,apu2:TRuutu; tyyppiCounter,i,j:integer;
begin
 rakentaja:=TRakentaja.luoRakentaja(tyyppienLukumaara);
 tyyppiCounter:=1;
 aloitus:=rakentaja.rakenna(tyyppiCounter,0,0);
 tyyppiCounter:=tyyppiCounter+1;
 koko:=round(form1.Width/20);
 i:=1;
 apu:=rakentaja.rakenna(tyyppiCounter,koko*i,0);
 tyyppiCounter:=tyyppiCounter+1;
 aloitus.oikea:=apu;
   for i:=2 to ruutujaSivulla do begin
     if(tyyppiCounter<tyyppienLukumaara) then apu2:= rakentaja.rakenna(3,koko*i,0);
     if(tyyppiCounter=tyyppienLukumaara) then apu2:= rakentaja.rakenna(2,koko*i,0);
     if(tyyppiCounter>tyyppienLukumaara) then
     begin apu2:= rakentaja.rakenna(1,koko*i,0);
       tyyppiCounter:=-1;
     end;
     tyyppiCounter:=tyyppiCounter+1;
     apu.oikea:=apu2;
     apu:=apu2;
    end;
  for i:=1 to ruutujaSivulla-1 do begin
    if(tyyppiCounter<tyyppienLukumaara) then
      apu2:= rakentaja.rakenna(3,koko*ruutujaSivulla,koko*i);
     if(tyyppiCounter=tyyppienLukumaara) then
      apu2:= rakentaja.rakenna(2,koko*ruutujaSivulla,koko*i);
     if(tyyppiCounter>tyyppienLukumaara) then begin
      apu2:= rakentaja.rakenna(1,koko*ruutujaSivulla,koko*i);
       tyyppiCounter:=-1;
     end;
     tyyppiCounter:=tyyppiCounter+1;
     apu.alas:=apu2;
     apu:=apu2;
    end;
  for i:=0 to ruutujaSivulla+1 do begin
     if(tyyppiCounter<tyyppienLukumaara) then
         apu2:= rakentaja.rakenna(3,koko*ruutujasivulla-koko*i,koko*ruutujaSivulla);
     if(tyyppiCounter=tyyppienLukumaara) then
         apu2:= rakentaja.rakenna(2,koko*ruutujasivulla-koko*i,koko*ruutujaSivulla);
     if(tyyppiCounter>tyyppienLukumaara) then begin
         apu2:= rakentaja.rakenna(1,koko*ruutujasivulla-koko*i,koko*ruutujaSivulla);
         tyyppiCounter:=-1;
     end;
     tyyppiCounter:=tyyppiCounter+1;
     apu.vasen:=apu2;
     apu:=apu2;
    end;
 for i:=1 to ruutujaSivulla-1 do begin
     if(tyyppiCounter<tyyppienLukumaara) then
         apu2:= rakentaja.rakenna(3,0,koko*ruutujaSivulla-koko*i);
     if(tyyppiCounter=tyyppienLukumaara) then
         apu2:= rakentaja.rakenna(2,0,koko*ruutujaSivulla-koko*i);
     if(tyyppiCounter>tyyppienLukumaara) then begin
         apu2:= rakentaja.rakenna(1,0,koko*ruutujaSivulla-koko*i);
         tyyppiCounter:=-1;
     end;
     tyyppiCounter:=tyyppiCounter+1;
     apu.ylos:=apu2;
     apu:=apu2;
    end;
end;
```

**Ohjelmakoodi 5. Tilanne ennen Strategia-suunnittelumallin hyödyntämistä**

## 4.2 Rakentaja (Builder)

Tätä suunnittelumallia käyttämällä luodaan Strategian ohjaamana pelilaudan ruutuja.

### Tarkoitus

Malli erottaa toisistaan monimutkaisen olion rakentamisessa käytetyn prosessin ja olion esitysmuodon, jolloin samalla rakentamisprosessilla voidaan tuottaa erilaisia esitysmuotoja.

### Perustelut

Saadaksemme pelilautakomponentista yleisen pelilautakomponentin täytyy pelilaudan ruutujen luominen erottaa itse pelilaudasta. Strategia(”geometria”) ohjaa pelilaudan ruutujen rakentamiseen ohjelmoitua rakentajaa. Erilaisia peliruutuja on yhtä paljon kuin erilaisia pelejä, joten pelilautaan on pystyttävä helposti määrittelemään erilaisten pelien ruutujen luominen. Ongelma ratkaistaan siten, että määritellään strategia, joka luo pelilaudan ruudut rakentajan avulla. Erilaiset rakentajat erikoistuvat eri peleihin esimerkiksi Triviaali -rakentaja on erikoistunut Triviaali Pursuitti -ruutujen rakentamiseen, kun taas Monopolirakentaja on erikoistunut Monopolin ruutujen rakentamiseen.

### Soveltuvuus

Rakentaja -mallia on hyvä käyttää silloin, kun halutaan pitää erillään algoritmi, jolla luodaan monimutkainen osista koostuva olio ja tämän olion osat. Rakentaja tarjoaa lisäksi tavan, jolla osat yhdistetään.

Rakentamisprosessin on pystyttävä tuottamaan esitysmuodoltaan erilaisia olioita. Holubin (2004) mukaan Rakentaja toimii hyvin myös silloin, kun sovellus rakennetaan siten, että erotetaan liiketoimintalogiikka käyttöliittymistä. Oliojärjestelmissä tämä voi tuottaa ongelmia, koska olion ei tulisi paljastaa olion sisäistä toteutusta. Kuten edellä mainitsimme, hyvin tehty oliosuunnittelu pyrkii välttämään saanti-metodeja, jotka paljastavat olion sisäisen tilan. Tästä johtuen olion on itse tarjottava käyttöliittymänsä. Joskus on tarpeen, että olion on tuotettava useampia kuin yksi esitystapa itsestään. Tällöin ei haluta sekoittaa liiketoimintalogiikkaa yksityiskohtiin, joita tarvitaan tuotettaessa erilaisia esityksiä. Rakentajalla ongelma voidaan ratkaista panemalla esitystapakohtainen ohjelmakoodi Rakentaja-luokkaan jota ohjataan ohjaajalla (engl. director) oliolla. Rakentaja siis rakentaa olion käyttöliittymän.

## Osallistujat

### Rakentaja

Määrittelee abstraktin rajapinnan, jolla luodaan pelilaudan osat.

### TriviaaliRakentaja

Toteuttaa Rakentaja-rajapinnan ja käyttää rajapintaa tuotteen rakentamiseen osista kokoamalla. Pitää kirjaa tuotteen rakenteen edistymisestä rakentamisen kuluessa (kerää peliruudut listaan). Toteuttaa tuotteen hakuoperaation (`AnnaRuudut`).

### Strategia

Kokoaa laudan käyttäen Rakentaja-rajapintaa.

**Pelilauta**

Kuvaa rakentamisen kohteena olevan tuotteen. TriviaaliRakentaja rakentaa tuotteen sisäisen esitysmuodon ja määrittelee prosessin, jonka mukaisesti kokoaminen tehdään.

## Seuraukset

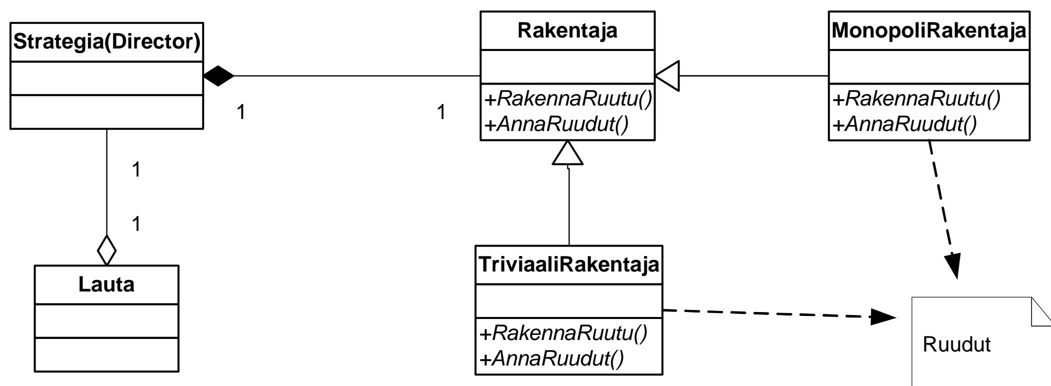
Mahdollistaa tuotteen (pelilaudan) sisäisen esitysmuodon varioinnin.

Erottaa toisistaan tuotteen rakentamiseen ja esittämiseen käytetyn logiikan.

Tarjoaa hyvän kontrollin rakentamisprosessille.

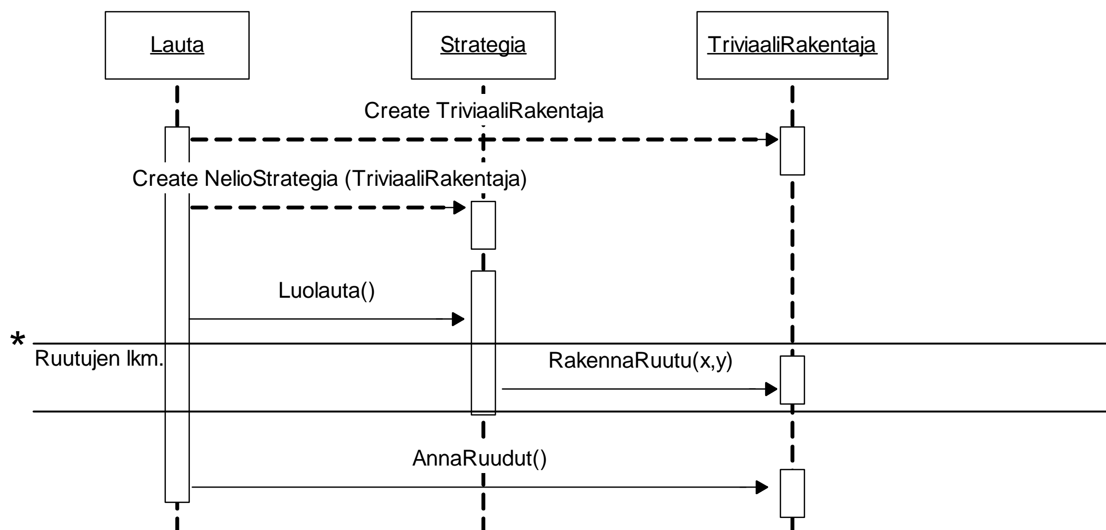
Mahdollistaa pelilautakomponentin tapauksessa helpon tavan vaihtaa erityyppinen rakentaja erilaisille peleille ja näiden ruuduille.

Kuviossa kolme esitetään Rakentaja-mallin luokkakaavio.




**Kuvio 3. Rakentajamallin luokkien väliset suhteet**

Kuviossa 4 esitetään rakentajan toiminta sekvenssikaaviolla.




**Kuvio 4. Sekvenssikaavio**

Ohjelmakoodi 6 esitetään `IRakentaja`-rajapinnan toteutus.

```
unit rakentaja;

interface

uses Classes;

type

IRakentaja = interface
   function AnnaRuudut:TList;
   procedure RakennaRuutu(x:integer;y:integer);
end;
```

**Ohjelmakoodi 6. IRakentaja**

Ohjelmakoodissa 7 esitetään `TTriviaaliRakentajan` toteutus.

```
unit triviaaliRakentaja;

interface
uses
SysUtils, Classes, ruutu, rakentaja, lauta;
type
  TRakennusMetodi = procedure(x:integer;y:integer) of object; //osoitintaulukko
   //että voidaan helposti lisätä
  TTriviiaaliRakentaja = class(TComponent,Irakentaja)
  private
    FRuutujenKuvat: TStringlist;
    FKysymysTyyppienLukumaara:integer;
    kysymysCounter:integer;   //hoitaa kysymysruutujen syklin
    variRuutuCounter:integer; //hoitaa väriruutujen syklin laudalle(nappula)
    variCounter:integer; //hoitaa kaikkien ruutujen syklin
    ruudut:TList;
    FMesta:TLauta;
    RakennusMetodit : array of TRakennusMetodi;
    procedure RakennaJatkaMatkaaRuutu(x:integer;y:integer);
    procedure RakennaVariRuutu(x:integer;y:integer);
    procedure RakennaKysymysRuutu(x:integer;y:integer);
    procedure Linkita(kohdalla:TTriviaaliRuutu);
  public
    function AnnaRuudut:TList;
    constructor Create (AOwner:TComponent); override;
    procedure RakennaRuutu(x:integer;y:integer);
    destructor Destroy; override;
  published
   property RuutujenKuvat: TStringlist read FRuutujenKuvat write FRuutujenKuvat;
   property KysymysTyyppienLukumaara : integer read fkysymysTyyppienLukumaara
            write fkysymysTyyppienLukumaara default 4;
   property Mesta:TLauta read FMesta write FMesta;
 end;

procedure Register;

implementation

function TTriviiaaliRakentaja.AnnaRuudut: TList;
var i:integer;
begin
  i:=ruudut.Count;
  TTriviaaliRuutu(ruudut.Items[0]).alas:=TTriviaaliRuutu(ruudut.Items[i-1]);
  TTriviaaliRuutu(ruudut.Items[i-1]).ylos:=TTriviaaliRuutu(ruudut.Items[0]);
  result:=ruudut;
end;
constructor TTriviiaaliRakentaja.Create(AOwner: TComponent);
var i:integer;
begin
  .
  .
  .
  for i:=0 to kysymysTyyppienLukumaara do  begin
  RakennusMetodit[i] := rakennaKysymysRuutu;
  end;
  i:=kysymysTyyppienLukumaara;
  RakennusMetodit[i] := RakennaVariRuutu;
  RakennusMetodit[i+1] := RakennaJatkaMatkaaRuutu;
  ruudut:=TList.Create;
end;
procedure TTriviiaaliRakentaja.RakennaRuutu(x, y: integer);
begin
  RakennusMetodit[variCounter](x,y);  //rakentaa tietyn ruudun
  inc(variCounter);
  if(variCounter = kysymysTyyppienLukumaara+2) then //jos mennään loppuun, niin 0
  variCounter :=0;
end;
end.
```

**Ohjelmakoodi 7. TTriviaaliRakentaja**

## Ohjelmakoodin tulkinta

Konkreettiset Rakentaja-luokat toteuttavat `IRakentaja`-rajapinnan ja siinä esitellyt metodit `AnnaRuudut` ja `RakennaRuutu`. Rakentajat ovat logiikaltaan pelilaudan tapauksessa suhteellisen komplekseja luokkia, joten emme esitä tässä täydellistä toteutusta. Luokkien ohjelmakoodeista on myös pyritty esittämään ainoastaan suunnittelumallin kannalta olennaiset osat. Rakentaja toimii Strategian ohjastamana ja tuottaa peliruutuja aina, kun sille kutsutaan `RakennaRuutu`-metodia. Konstruktorissa alustetaan Rakentaja-metodit, jotka on ruutujen rakentamisen vaiheistamisen vuoksi sijoitettu taulukkoon. Kun `RakennaRuutu`-metodia kutsutaan rakentajalle, se valitsee oman sisäisen tilansa perusteella, mitä metodia todellisuudessa kutsutaan ja millainen ruutu rakennetaan. `AnnaRuudut`-metodi toimii siten, että sitä kutsuttaessa rakentajalta saadaan ulos sen rakentamat ruudut tietorakenteessa. Rakentajan sisäinen `Linkita`-metodi mahdollistaa sen, että rakennetut ruudut ovat toivotunlaisessa linkitetyssä rakenteessa.

## 4.3 Ainokainen (Singleton)

### Tarkoitus

Varmistaa, että luokasta luodaan vain yksi ilmentymä, ja tarjoaa globaalin tavan päästä käsiksi tähän ilmentymään.

### Perustelut

Olio-ohjelmoinnissa on luokkia, joiden rooli on sellainen, että niistä voidaan luoda vain yksi ilmentymä. Pelissä esimerkiksi sekoittelun hoitava `TSotkija`-luokka on tällainen. Sotkijaa käyttävät sekä kysymyskorttipakat että itse kysymykset. Perinteisesti proseduraalisessa ohjelmoinnissa `TSotkija` olisi funktio, mutta olio-ohjelmoinnissa emme niitä käytä.

Miten varmistetaan, että luokalla on vain yksi ilmentymä ja että muut sovelluksen osat pääsevät tähän ilmentymään helposti käsiksi? Olioon päästään käsiksi globaalin muuttujan kautta, mutta globaalin muuttujan käyttö ei yksin riitä estämään useampien ilmentymien luontia. Parempi ratkaisu on antaa olion itse pitää huolta siitä, että useampia ilmentymiä ei luoda. Ainokainen-malli toimii juuri näin.

### Soveltuvuus

Käytä Ainokainen-mallia, kun luokalla täytyy olla täsmälleen yksi ilmentymä ja sen täytyy olla kaikkien sovellusten saatavilla.

Ilmentymää on voitava laajentaa aliluokkien avulla ja sovellusten on kyettävä käyttämään laajennettua ilmentymää ilman, että niiden koodia pitää muuttaa.

### Osallistujat

#### Sotkija

Määrittelee `instance`-operaation (`rakenna`), jolla sovellukset pääsevät käsiksi sen uniikkiin ilmentymään. `Instance` on luokka-operaatio.

### Seuraukset

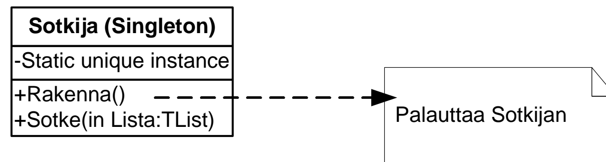
Ainoan ilmentymän käyttöä kontrolloidaan.

Nimiavaruus pienenee.

Operaatioita ja esitysmuotoa voidaan jalostaa.

Ratkaisu on joustavampi kuin luokkaoperaation käyttö.

Kuviossa 5 kuvataan Ainokainen-mallin luokkarakenne.




**Kuvio 5. Sotkija**

Ohjelmakoodissa kahdeksan esitetään `TSotkijan` toteutus.

```
unit sotkija;

//sekoittaa tietorakenteen

interface

uses
Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;

type

  Tsotkija = class
  private
    tietorakenne : TList;
    function AnnaObject(i: integer):TObject;
    procedure LisaaLoppuun(Objecti: TObject);
  protected
    constructor Create;
  public
    procedure Sotke (itietorakenne: TList);
    class function Rakenna:TSotkija;

  end;

  implementation

var Globaalisotkija:Tsotkija;  // Delphi ei tue luokkamuuttujia, siksi tällainen
                               // ratkaisu

constructor Tsotkija.Create;
begin
  Globaalisotkija := self;
end;

class function Tsotkija.Rakenna: TSotkija;
begin
   if not assigned(Globaalisotkija) then self.create;
   result := Globaalisotkija;
end;
end.
```

**Ohjelmakoodi 8. TSotkija**

### Ohjelmakoodin tulkinta

Ainokainen-mallin ideana on antaa luokan käyttäjälle vain yksi ilmentymä luokasta. Tämän takia luokan konstruktorista tehdään suojattu (engl. `protected`), jolloin luokan käyttäjä ei pääse sitä suoraan kutsumaan. Luokan ilmentymän luonti onnistuu siten, että luokalle tehdään luokkametodi (`Rakenna`), joka palauttaa luokan ilmentymän. Ensimmäisellä kerralla `Rakenna`-metodia kutsuttaessa luodaan luokasta ilmentymä ja seuraavilla kutsukerroilla palautetaan tämä ilmentymä. Delphi ei tue luokkamuuttujia, joten jouduimme käyttämään `sotkija`-yksikössä globaalimuuttujaa. Tämä ei ole kovin vaarallista, koska muuttuja näkyy vain ko. yksikössä oleville luokille.

Esimerkkitapauksessa Ainokainen-mallin hyöty jää kohtuullisen pieneksi, mutta mallin idea tulee hyvin esille. Halusimme tehdä oliomaisemman ratkaisun kuin funktio tai pelkkä luokka, ja luokalle pelkkiä luokkametodeita ja siksi päädyimme käyttämään Ainokainen-mallia.

## 4.4 Vierailija (Visitor)

Vierailija-suunnittelumalli ei kuulu Gamman ja ym. (1995) mukaan kaikkein yksinkertaisimpien suunnittelumallien joukkoon. Tämän havaitsimme myös itse sitä soveltaessamme. Vierailija-mallia yritettiin käyttää Triviaalissa Pursuitissa pelilaudan ruutujen piirtämiseen ja pelin logiikan toteuttamiseen. Malli onnistuttiin toteuttamaan jollain tasolla ohjelmakoodiksi, mutta sitä tehtäessä havaitsimme selkeästi, että olemme tekemässä toteutusta väärällä tavalla. Tämä tulee huomioida lukiessa seuraavia kuvauksia. Mallin käyttö ruutujen piirtotarkoitukseen oli sinänsä ratkaisuna väärä, koska näin menetimme esimerkiksi Delphin tarjoaman tuen komponenttien `onClick`-tapahtumille, eli Vierailijan

piirtämät ruudut olivat vain kuvia lomakkeella ilman mahdollisesti muita hyödyllisiä ja tarvittavia ominaisuuksia.

### Tarkoitus

Vierailija-suunnittelumallin tarkoitus on liittää operaatioita isäntäolioon tarjoamalla tie Vierailijalle. Yleisesti Vierailijaa käytetään muokkaamaan tai tutkimaan kokoelmarakenteen elementtejä (Holub 2003).

Vierailija-malli siis edustaa operaatiota, joka suoritetaan oliorakenteen elementeille. Elementtien luokkia ei tarvitse Vierailija-mallia käyttäessä muuttaa, kun luodaan niihin kohdistuva uusi operaatio.

### Perustelut

Vierailija-mallin avulla pystytään lisäämään laudan ruutuihin operaatioita koskematta itse ruutuihin. Riittää, että määritellään uusia Vierailija-olioita ja näin saadaan uusia operaatioita. Esimerkiksi jos halutaan ruudun näkyvän eri päätelaitteissa ja ei tiedetä tekohetkellä kaikkia mahdollisia päätelaitteita, niin Vierailijaa käytettäessä ei tarvitse mahdollisia uusia näyttörutiineja ohjelmoitaessa koskea itse ruutuihin.

Useiden erilaisten ja toisistaan poikkeavien operaatioiden sijoittaminen tietorakenteen luokkiin saattaa tehdä järjestelmästä vaikeasti ymmärrettävän ja ylläpidettävän. Vierailija-mallia käyttämällä tämä monimutkaisuus voidaan piilottaa erilaisiin vierailijoihin. Monimutkaisuus ei toteudu ja ole näin havaittavissa ruutu-luokan tapauksessa, koska kyseessä on pieni luokka.

## Soveltuvuus

Gamman ja ym. (1995) mukaan Vierailija-mallia on viisasta käyttää silloin, kun oliorakenne sisältää useita luokkia, joilla on erilaiset rajapinnat, ja halutaan suorittaa näiden ilmentymiin kohdistuvia operaatioita, jotka riippuvat niiden konkreettisista luokista.

Oliorakenteen olioille on suoritettava useita erillisiä ja toisiinsa liittymättömiä operaatioita, ja olioluokkia ei haluta ”roskata” näillä operaatioilla. Vierailija-malli auttaa pitämään toisiinsa liittyvät operaatiot yhdessä kokoamalla ne yhteen luokkaan. Jos useat sovellukset jakavat oliorakennetta, voidaan Vierailija-mallia käyttää sovelluskohtaisten operaatioiden niputtamiseen.

Oliorakenteen luokat muuttuvat harvoin, mutta uusia rakenteeseen kohdistuvia operaatioita lisätään usein. Oliorakenteen luokkien muuttaminen vaatii kaikkien vierailijoiden rajapintojen muuttamista, mikä voi olla kallista. Jos oliorakenteen luokat muuttuvat usein, on luultavasti parempi määritellä operaatiot itse luokissa.

## Osallistujat

### TVisitor

Määrittelee `visit`-operaation kullekin peliruudulle.

### TPiirtoVisitor

Toteuttaa jokaisen Visitor-luokan operaation.

### TRuutu

Määrittelee `accept`-operaation Vierailija-parametrilla.

**TVariRuutu**

Toteuttaa accept-operaation Vierailija-parametrilla.

**Oliorakenne (TLauta)**

Osaa luetella alkionsa(Ruudut).

Tarjoaa vierailijalle rajapinnan, jonka avulla vierailija voi vierailla alkioissa.

Usein kooste (rekursiokooste) tai kokoelma kuten lista.

## Seuraukset

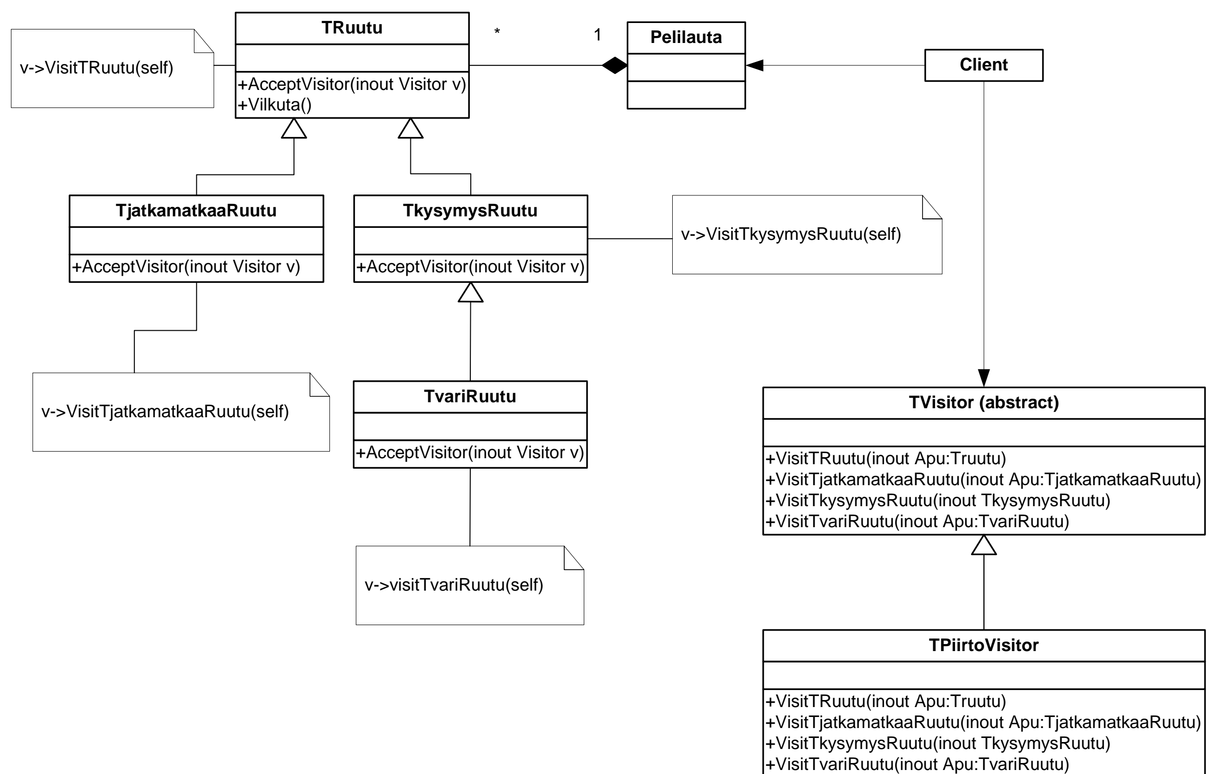
Uusien operaatioiden lisääminen on helppoa.

Vierailija kokoaa yhteenkuuluvat operaatiot ja eristää yhteen kuulumattomat operaatiot.

Uuden konkreettisen elementtiluokan lisääminen on vaikeaa. Vierailu luokkahierarkioiden yli mahdollista.

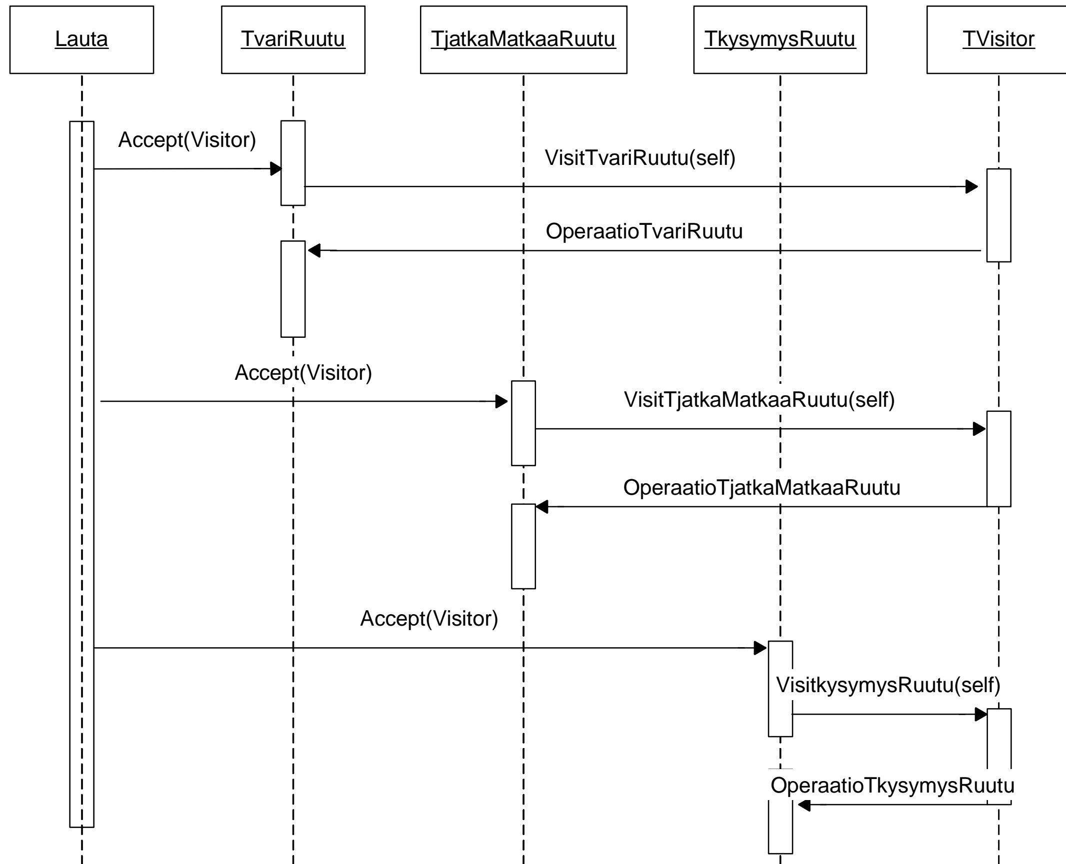
Soveltuvuuskohdassa Vierailija-mallin käytölle Gamma ja ym. (1995) antavat yhdeksi perusteluksi sen, että sen käyttö on perusteltua tapauksessa, jolloin tietorakenne sisältää luokkia, joilla ei ole sama yläluokka. Pelilaudan ruutujen tapauksessa ko. hyöty on häviävä koska ruudut on peritty samasta kantaluokasta (`TRuutu`).

Kuviossa 6 esitetään Vierailija-mallin liittyminen pelilaudan ruutuihin luokkakaaviona.




**Kuvio 6. Vierailija-mallin luokkakaavio**

Kuviossa 7 esitetään sekvenssikaavion avulla vierailijan toiminta pelilaudan ruuduissa.




**Kuvio 7. Vierailija-mallin sekvenssikaavio**

Ohjelmakoodissa 9 esitetään abstrakti Vierailija-yliluokka

```
unit visitor;
interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, Ruutu;

type
TVisitor = class
  public
   procedure VisitTJatkaMatkaaRuutu(apu:TJatkaMatkaaRuutu);virtual; abstract;
   procedure VisitTVariRuutu(apu:TVariRuutu);virtual; abstract;
   procedure VisitTKysymysRuutu(apu:TKysymysRuutu); virtual; abstract;
   procedure VisitTRuutu(apu:TRuutu);virtual; abstract;
   procedure VisitTTriviaaliRuutu(apu:TTriviaaliRuutu);virtual; abstract;
end;
```

**Ohjelmakoodi 9. Abstrakti TVisitor**

Ohjelmakoodissa 10 esitetään `TPiirtoVisitor`, joka on toteutus abstraktille `TVisitor`-luokalle.

```
TPiirtoVisitor = class(TVisitor)
  private
    kanvas:TCanvas;
    koko:integer;
    id:integer;
  public
   constructor Create(apu:TCanvas;ikoko:integer);
   procedure VisitTJatkaMatkaaRuutu(apu:TJatkaMatkaaRuutu);  override;
   procedure VisitTVariRuutu(apu:TVariRuutu);  override;
   procedure VisitTKysymysRuutu(apu:TKysymysRuutu); override;
   procedure VisitTRuutu(apu:TRuutu);  override;
   procedure VisitTTriviaaliRuutu(apu:TTriviaaliRuutu);  override;
end;

implementation

{ TVisitor }

constructor TPiirtoVisitor.Create(apu:TCanvas;ikoko:integer);
begin
 kanvas:=apu;
 koko:=ikoko;
 id:=0;
end;

procedure TPiirtoVisitor.VisitTRuutu(apu: TRuutu);
begin
end;

procedure TPiirtoVisitor.VisitTJatkaMatkaaRuutu(apu: TJatkaMatkaaRuutu);
begin
  kanvas.Rectangle(apu.x,apu.y,apu.x+koko,apu.y+koko);
  kanvas.TextOut(apu.x,apu.y,'-'+IntToStr(id));
  id:=id+1;
end;

procedure TPiirtoVisitor.VisitTVariRuutu(apu: TVariRuutu);
begin
  kanvas.Rectangle(apu.x,apu.y,apu.x+koko,apu.y+koko);
  kanvas.TextOut(apu.x,apu.y,IntToStr(id)+' Vari '+IntToStr(apu.tyyppi));
  id:=id+1;
end;

procedure TPiirtoVisitor.VisitTKysymysRuutu(apu: TKysymysRuutu);
begin
  kanvas.Rectangle(apu.x,apu.y,apu.x+koko,apu.y+koko);
  kanvas.TextOut(apu.x,apu.y,IntToStr(id)+' '+IntToStr(apu.tyyppi));
  id:=id+1;
end;

procedure TPiirtoVisitor.VisitTTriviaaliRuutu(apu: TTriviaaliRuutu);
begin
  inherited;
end;

end.
```

**Ohjelmakoodi 10. TPiirtoVisitor**

Ohjelmakoodissa 11 esitetään `TKysymysRuutu`-luokan `AcceptVisitor`-metodin toteutus.

```
TKysymysRuutu = class(TTriviaaliRuutu)
  private

  public
    tyyppi:integer;
    constructor Create(ix:integer;iy:integer;ityyppi:integer);reintroduce;virtual;
    procedure AcceptVisitor(visitori: TObject); override;

end;


procedure TKysymysRuutu.AcceptVisitor(visitori: TObject);
begin
  if(visitori is TVisitor) then
  (visitori as TVisitor).VisitTKysymysRuutu(self);
end;
```

**Ohjelmakoodi 11. TKysymysRuutu-luokan AcceptVisitor-metodi**

## Ohjelmakoodin tulkinta

Idea Vierailijan ohjelmakoodissa on seuraava: On olemassa abstrakti yläluokka `TVisitor`, jossa on esitelty vierailijaluokkien tarvitsemat metodi. Nämä metodit voitaisiin haluttaessa toteuttaa myös funktioiden parametrien kuormitusta (engl. function overloading) käyttäen, mikäli käytettävä ohjelmointikieli tätä ominaisuutta tukisi. Pelilauta-esimerkissä jokainen metodi haluttiin kuitenkin selvyyden vuoksi nimetä eri nimillä.

Laudan ruuduilla on kaikilla metodi `AcceptVisitor`, jolla se pystyy vastaanottamaan vierailijan. Vastaanotettuaan Vierailijan ruutu kutsuu Vierailijan ruutua vastaavaa `visit`-metodia vieden itsensä (ruutu) metodille parametrina.

Se että Vierailija tuodaan `TObject` tyyppisenä, johtuu Delphi-kielen rajoitteista. Mikäli Vierailija olisi tuotu `TVisitor`-tyyppisenä olisi tuloksena ollut `circular reference` –virhe, eli yksiköt olisivat viitanneet toisiinsa syklisesti.

## 4.5 Double-dispatch

Vierailija mallin tapaa kyetä lisäämään operaatiota luokkiin muuntelematta silti itse luokkia kutsutaan nimellä double-dispatch (Gamma ja ym. 1995). Double-dispatch tarkoitaa sitä, että suoritettava operaatio riippuu pyynnön tyypistä ja kahden vastaanottajan tyypistä. Joissain ohjelmointikielissä tämä tekniikka on sisäänrakennettuna. Ruuduissa oleva Vierailijan vastaanottava operaatio `AcceptVisitor` on double-dispatch–operaatio. Peliesimerkissä sen merkitys muodostuu Vierailijasta ja Ruutu-luokasta yhdessä. Sen sijaan, että Peliruudun operaatiot sidottaisiin staattisesti siihen itseensä, voidaan ne kasata Vierailijaan ja suorittaa sidonta ajon aikana `AcceptVisitor`-operaatiolla (Gamma ja ym. 1995).

## 4.6 Yhteenveto

Tässä luvussa käsiteltiin neljän suunnittelumallin käyttö esimerkin kautta. Esimerkkinä käytettiin Triviaali Pursuitti -peliä. Esimerkkitapauksessa käsiteltiin Strategia-, Rakentaja-, Ainokainen- ja Vierailija-suunnittelumallit. Mallien käyttö kuvattiin mukaillen Gamman ja ym. (1995) kirjassa esitettyä tapaa. Jokaisen mallin kohdalta pohdittiin esimerkin kautta mallin tarkoitusta, perusteluja käytölle, soveltuvuutta ja erilaisia osallistujia. Kuvauksien tueksi esitettiin graafisia kuvauksia malleissa käytettävien luokkien suhteista sekä luokkien vuorovaikutuksesta. Keskeisimmät toteutukseen liittyvät ratkaisut ohjelmakoodiesimerkkeineen esitettiin myös.

Esimerkissä suunnittelumallien käytöllä saavutettiin Vierailija-mallia lukuun ottamatta rakenteellisesti hyviä ja selkeitä ohjelmaratkaisuja. Luvussa esitetyt suunnitteluratkaisut voisi toteuttaa myös toisin, mutta nähdäksemme ainakin

pelilautakomponenttiin liittyvät Strategia ja Rakentaja ovat niin perusteltuja ratkaisuja, että ilmenisivät jossain muodossa ohjelmassa vaikka suunnittelumalleja ei olisi tietoisesti käytettykään.

# 5 SUUNNITTELUMALLIEN LUOKITTELU

Koska malleja on paljon, täytyy niiden luokittelemiseksi olla jotain tapoja. Kaikilla sovellusalueilla on omat vakiintuneet mallinsa. Jotkut näistä malleista ovat toistensa kanssa yhteneväisiä, toiset omia kokonaisuuksiaan. Luokittelulla pyritään siihen, että pystyttäisiin löytämään oikeita malleja tiettyihin suunnitteluongelmiin. Luokittelu nopeuttaa myös mallien opiskelua ja auttaa uusien mallien louhintaa. Malleille löytyy kirjallisuudessa useita luokittelumalleja, joista tässä esitetään kaksi. Ensimmäisenä esitetään Gamman ja ym. (1995) ja toisena Buschmannin ja ym. (2002) eli Siemens-kirjassa esitelty tapa luokitella malleja.

## 5.1 Gamma-luokittelu

Suunnittelumallit luokitellaan Gamman ja ym. (1995) kirjassa kahden pääkriteerin mukaan: kohde ja tarkoitus. Kohdekriteeri jakaa mallit sen mukaan, koskeeko malli kokonaista luokkaa vai yksittäisiä olioita. Luokkia koskevat mallit käsittelevät luokkia ja niiden aliluokkien välisiä suhteita. Luokkia koskevat mallit ovat staattisia. Olioita koskevat mallit ovat enemmänkin dynaamisia eli ajonaikaisia.

Tarkoituksen mukaan suunniteltumallit voidaan jakaa kolmeen luokkaan: luonti-, rakenne- ja käyttäytymismalleihin. Tarkoituskriteerin tehtävänä on yksinkertaisesti kertoa, mitä varten kyseinen malli on olemassa. Luontimallit koskevat olioiden luomisprosessia, rakennemalleilla kuvataan olioiden ja luokkien rakenteita ja käyttäytymismallit kuvaavat olioiden ja luokkien vuorovaikutussuhteita ja vastuiden jakamista.

Gamma ja ym. (1995) kirjan suunnittelumallien luokittelua pidetään yleisesti huonona. Luokittelun tekee huonoksi esimerkiksi se, että luokittelu keskittyy liikaa olioparadigman mukaiseen jaotteluun.

Mikäli esimerkiksi perinteistä proseduraalista kieltä käytettäisiin toteutusvälineenä, saattaisivat periytyminen ja dynaaminen sidonta olla suunnittelumalleja, jolloin ohjelmoija joutuisi ne tarvittaessa toteuttamaan itse kyseisen mallin mukaisesti (Koskimies 2001).

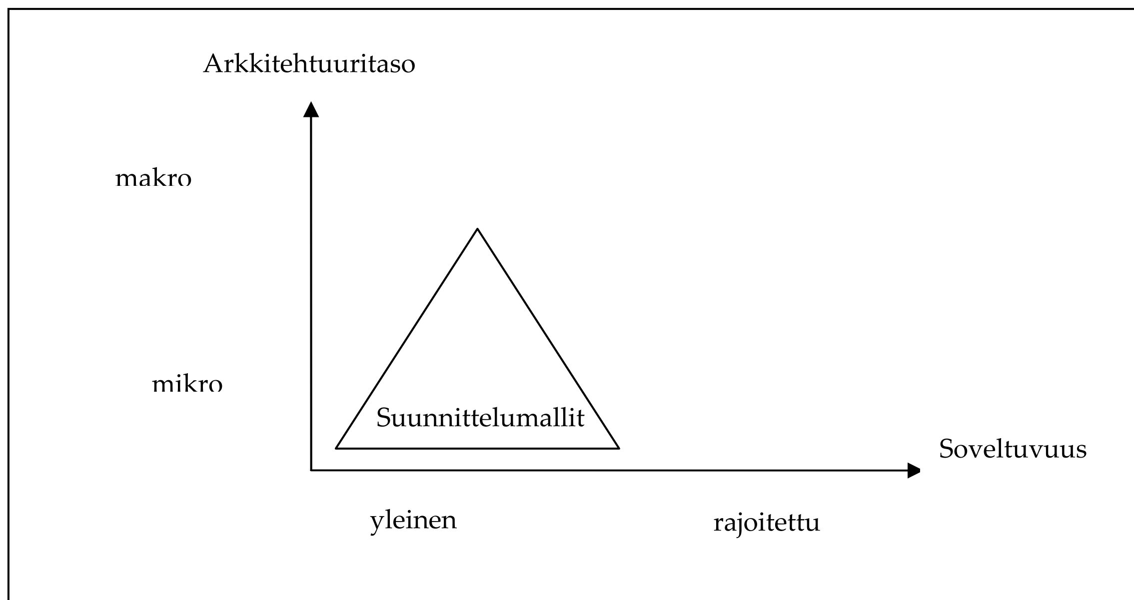
Suunnittelumalleja on muitakin kuin oliomalleja, joten mallien luokittelu ei voi lähteä ohjelmointiparadigmasta. Lisäksi Gamman ja ym. (1995) luokittelu on suhteellisen karkea, koska luokittelu kohdistetaan vain ylimalkaisella tasolla luontiin, rakenteeseen ja käyttäytymiseen. Luokitteluun kohdistetun yleisen kritiikin vuoksi ko. luokittelun kaikkien yksityiskohtien esittäminen sivuutetaan tässä.

Taulukossa 1 esitetään Gamman ja ym. (1995) esittämä suunnittelumallien luokittelu.

**Taulukko 1. Gamma-luokittelu**

| | | TARKOITUS | | |
|---|---|---|---|---|
| | | Luonti | Rakenne | Käyttäytyminen |
| KOHDE | Luokka | Tehdasmetodi | Sovitin(luokka) | Tulkki<br>Operaatiorunko |
| | Olio | Abstrakti tehdas<br>Rakentaja<br>Prototyyppi<br>Ainokainen | Sovitin(olio)<br>Silta<br>Rekursiokooste<br>Kuorruttaja<br>Julkisivu<br>Hiutale<br>Edustaja | Vastuuketju<br>Komento<br>Iteraattori<br>Välittäjä<br>Muistio<br>Tarkkailija<br>Tila<br>Strategia<br>Vierailija |

Kuviossa 8 esitetään Gamman ja ym. (1995) suunnittelumallien käyttöalueet ohjelmistontuotannossa.




**Kuvio 8. Gamman ja ym. (1995) suunnittelumallit ohjelmistontuotannossa**

## 5.2 Siemensin luokittelu

Buschmannin ja ym. (2002) kirjassa esitetty luokittelu eli Siemens-luokittelu jakaa suunnittelumallit kahden pääkriteerin mukaan. Luokittelukriteereistä kirjassa tärkeimpänä esitetään ohjelmistotuotannon vaihe. Ohjelmistotuotannon vaiheen mukaan suunnittelumallit voidaan jakaa arkkitehtuurimalleihin, suunnitteluvaiheen malleihin ja idiomeihin. Toinen pääkriteeri Buschmann ja ym. (2002) kirjan mukaan on ongelmaluokitus. Tässä luokittelussa jaotellaan suunnittelumallit niiden ratkaisemien ongelmien mukaan. Erityyppiset ongelmat on kategorisoitu omiksi luokikseen ja nimetty siten, että ohjelmistotuotannossa esiintyvät erilaiset ongelmat voidaan tunnistaa niistä. Seuraavat ongelmaluokat esitetään (Buschmann ja ym. 2002):

**Taulukko 2. Ongelmaluokat Buschmann ja ym.**

| Ongelmaluokka | Mallien tarkoitus |
| --- | --- |
| Mudasta rakenteeseen | Auttaa jakamaan koko järjestelmän yhdessä työskenteleviin alitehtäviin. |
| Hajautetut järjestelmät | Antaa infrastruktuurin järjestelmille, joiden osat on sijoitettu eri prosesseihin tai useisiin alijärjestelmiin ja komponentteihin. |
| Interaktiiviset järjestelmät | Mahdollistaa HCI-järjestelmien rakentamisen. |
| Mukautuvat järjestelmät | Voidaan muodostaa infrastruktuuri kehittyville ja muuttuville systeemeille |
| Rakenteellinen jakaminen | Voidaan jakaa alijärjestelmät ja monimutkaiset komponentit keskenään työskenteleviksi osiksi |
| Työn organisointi | Määrittelevät monimutkaisia palveluita tarjoavien komponenttien työskentelyn keskenään |
| Kulunvalvonta | Vartioivat ja myös kontrolloivat palveluiden ja komponenttien käyttöä |
| Hallinta | Käsittelevät kokonaisuutena homogeenisten olioiden varastoja, palveluja ja komponentteja |
| Kommunikointi | Auttavat organisoimaan komponenttien välisen kommunikoinnin |
| Resurssien hallinta | Auttavat käsittelemään jaettuja komponentteja ja objekteja |

Kun ohjelmistokehityksen vaihe ja ongelmaluokitus yhdistetään, suunnittelumallit voidaan esittää taulukon 3 mukaisesti (vrt. Buschmann ja ym. 2002, s 366). Tässä yhteydessä on kuitenkin huomattava, että mallien sijoittelusta/taulukoinnista puuttuu osa Gamman ja ym. (1995) esittämistä malleista. Mikäli kaikki Gamman ja ym. (1995) esittämät mallit haluttaisiin esittää taulukossa, jouduttaisiin ongelmaluokkia lisäämään. Taulukossa 4 esitetään Buschmannin ja ym. laatima luokittelu siten, että se kattaa myös Gamman ja ym. esittämät mallit.

**Taulukko 3. Siemens-luokittelu**

| | Arkkitehtoniset mallit | Suunnittelumallit | Idiomit |
|---|---|---|---|
| Mudasta rakenteeseen | Layers<br>Pipes and Filters<br>Blackboard | | |
| Hajautetut järjestelmät | Broker<br>Pipes and Filters | | |
| Vuorovaikutteiset järjestelmät | MVC<br>PAC | | |
| Mukautuvat järjestelmät | Microkernel<br>Reflection | | |
| Rakenteellinen jakaminen | | Whole-Part | |
| Työn organisointi | | Master-Slave | |
| Kulunvalvonta | | Proxy | |
| Hallinta | | Command Processor<br>View Handler | |
| Kommunikointi | | Publisher-Subscriber<br>Forwarder-Receiver<br>Client-Dispatcher-Server | |
| Resurssien hallinta | | | Counted Pointer |

**Taulukko 4. Siemens-luokittelu Gamma ja ym. malleilla laajennettuna**

| | Arkkitehtoniset mallit | Suunnittelumallit | Idiomit |
|---|---|---|---|
| Mudasta rakenteeseen | Layers<br>Pipes and Filters<br>Blackboard | Interpreter | |
| Hajautetut järjestelmät | Broker<br>Pipes and Filters<br>Microkernel | | |
| Vuorovaikutteiset järjestel-mät | MVC<br>PAC | | |
| Mukautuvat järjestelmät | Microkernel<br>Reflection | | |
| Luonti | | Abstract Factory<br>Prototype<br>Builder | Singleton<br>Factory Method |
| Rakenteellinen jakaminen | | Whole-Part<br>Composite | |
| Työn organisointi | | Master-Slave<br>Chain of Responsibility<br>Command<br>Mediator | |
| Kulunvalvonta | | Proxy<br>Façade<br>Iterator | |
| Palvelun varioninti | | Bridge<br>Strategy<br>State | Template Method |
| Palvelun laajentamien | | Decorator<br>Visitor | |
| Hallinta | | Command Processor<br>View Handler<br>Memento | |
| Liittyminen | | Adapter | |
| Kommunikointi | | Publisher-Subscriber<br>Forwarder-Receiver<br>Client-Dispatcher-Server | |
| Resurssien hallinta | | Flyweight | Counted Pointer |

## 5.3 Luokittelun merkitys

Mallien luokittelulla saavutetaan mallien järkevä ryhmittely. Kun mallit on luokiteltu huolellisesti ja luokittelujärjestelmä on järkevä, on oikean mallin löytäminen helpompaa myös sellaiselle, joka ei muista kaikkia mahdollisia malleja ulkoa. Buschmannin ja ym. (2002) kirjassa esitelty mallien luokittelu on selvästi Gamman ja ym. (1995) luokittelua parempi. Sillä Gamman ja ym. (1995) luokittelu on enemmänkin akateeminen ja liian karkea, kun taas Buschmann ja ym. (2002) kirjan luokittelu on suhteellisen selkeä ja se myös ottaa kantaa myös siihen, mikä malli sopii mihinkin ohjelmistotyön eri vaiheeseen. Lisäksi Buschmannin ja ym. (2002) kirjan luokittelussa otetaan kantaa mallin ratkaisemaan ongelmaan.

## 5.4 Suunnittelumallien kaltaiset mallit

Suunnittelumalleja käytetään ohjelmistontuotannossa suunnitteluvaiheessa. On olemassa erilaisia malleja ohjelmistotuotannon eri vaiheisiin ja toimintoihin. Ohjelmiston arkkitehtuuriin suunnitteluun ja muodostamiseen on olemassa arkkitehtuurimallit (tyylit) ja ohjelmiston toteutukseen halutulla kielellä on olemassa idiomeja, eli malleja, joilla kuvataan tietyn ohjelmointikielen tapa ratkaista jokin ongelma.

### Toteutusvaiheen malli, idiomi

Buschmannin ja ym. (2002) esittämät idiomit ovat malleina pienempiä ja kieliriippuvaisempia kuin tässä tutkielmassa aikaisemmin esitetyt suunnittelumallit. Idiomit esittävät, kuinka voidaan toteuttaa tehokkaasti ja oikein tietynlaisia asioita tietyn kielen tarjoamilla ominaisuuksilla. Buschmannin mukaan ne eroavat selkeimmin suunnittelumalleista juuri siinä, että suunnittelumallit osoittavat yleisiä rakenteellisia periaatteita, ja idiomit kuvaavat, kuinka pystytään ratkaisemaan toteutuksen kannalta spesifejä ongelmia jollain tietyllä ohjelmointikielellä.

Idiomit voivat kuvata ohjelmointikielen tapoja nimetä ohjelman osia. Tai esimerkiksi tavan, jolla lähdekoodi tulee muotoilla ja kommentoida. Idiomit ovat hyvä tapa opettaa ja opetella ohjelmointikieliä.

Hyvänä esimerkkinä idiomi-tason mallista voidaan pitää esimerkiksi lähteessä Stroustroup (2000) esitetty osoittimen viittauksien laskeminen. Tämä idiomi liittyy C++-kielen muistinhallintaan ja osoittimien käyttöön. Idiomi esitetään esimerkissä, jossa sitä käytetään merkkijonoluokan yhteydessä siten, että merkkijonon todellinen esitys voidaan jakaa useamman luokan kesken. Eli jos meillä

on kaksi merkkijonoa ”kissa”, niin todellisuudessa meillä onkin vain yksi merkkijono ”kissa” ja osoittimet kahvaluokkaan, jossa on osoitinlaskuri ja osoitin itse esitykseen. Jos teemme merkkijonosta kopion, niin sen sijaan, että kopioisimme sitä, kasvatamme vain kahvaluokassa olevaa osoitinlaskuria ja osoitamme uudella osoittimella edelleen kahvaluokkaan. Samainen idiomi on kuvattu myös Buschmannin ja ym. (2002) kirjassa nimellä Counted pointer.

Hirvonen (2002) esittää kirjassaan C-kielessä merkkijonojen kopiointiin idiomin. Tässä idiomissa merkkijono kopioidaan ”oikeasti” seuraavalla tavalla:

```
while ( *p++ = *q++ );
```

Pointterin q osoittama kohde *q sijoitetaan pointterin p osoittamaan paikkaan eli *p:n sisällöksi. Sen jälkeen pointterien arvoja kasvatetaan yhdellä. … Tämä lause käy siis taulukkoa läpi alkio alkiolta

Triviaalissa Pursuitissa käytettiin tietoisesti Delphi-idiomeja ohjelmakoodin muotoiluun, esimerkiksi muuttujien ym. nimeämiseen. Ainokainen suunnittelumalli toteutettiin Delphillä idiomia käyttäen. Object Pascal kielessä ei ole Gamman ja ym. (1995) esittämän Ainokaisen C++-kielisen esimerkkitoteutuksen käyttämää luokkamuuttujaa. Ainokainen voidaan kuitenkin ohjelmoida Delphillä käyttämällä globaalia muuttujaa yksikössä. Tämä ratkaisu on riittävä koska muuttuja ei ole globaali näkyvyydeltään kuin yksikön sisällä. Tosin käytettäessä globaalia muuttujaa on Ainokaisen toteutuksen sisältävä yksikkö suojattava muulta ohjelmakoodilta.

### Arkkitehtuuritason malli (tyyli)

Arkkitehtuuritason mallit ovat eri asia kuin suunnittelumallit. Arkkitehtuuri voi käsittää useita suunnittelumalleja, ja se kuvaa järjestelmää astetta ylempää kuin suunnittelumallit. Buschmannin ja ym. (2002) kirjassa esitetyt arkkitehtuuritason suunnittelumallit ovat vastaavia kuin Garlanin ja Shaw'n (2000) esittämät arkkitehtuuriset tyylit. Arkkitehtuurimallit ilmaisevat perusrakenteen, järjestelmän kaavan ohjelmistoille (Buschmann ja ym. 2002).

Esimerkkinä arkkitehtuuritason mallista voidaan mainita kerrokset (engl. layers). Garlanin ja Shaw'n (2000) mukaan kerrosjärjestelmä on jäsennetty hierarkkisesti. Jokainen kerros tarjoaa palvelua ylemmälle kerrokselle ja on asiakkaana alemmalle kerrokselle. Muiden kerroksien välistä tiedon vaihtoa ei mallissa sallita. Yleinen esimerkki Kerrosarkkitehtuureista ovat tietoliikenne protokollat, esimerkiksi OSI:n referenssimalli.

Triviaalin Pursuitin toteutuksessa Delphi-ympäristö itsessään määrää tapahtumapohjaisen (engl. Event-based) arkkitehtuurin käyttöön ja näin ollen Triviialin Pursuitin arkkitehtuuri on osittain tämän mukainen. Tapahtumapohjainen arkkitehtuuri koostuu komponenteista, ja komponentit määrittelevät rajapinnat, eli kuinka ne lähettävät kutsuja ja kuuntelevat kutsuja. Harjoitustyön tarkoituksena oli luoda mahdollisimman yleiskäyttöisiä komponentteja, joilla voidaan rakentaa sovellus. Toteutimme yksittäiset komponentit sisäisesti olioarkkitehtuurimallin mukaisesti, koska halusimme ylläpidettävän ja helposti laajennettavissa olevan ohjelman.

## 5.5 Yhteenveto

Tässä luvussa käsiteltiin suunnittelumallien luokittelua. Suunnittelumalleja voidaan luokitella usealla eri tavalla. Luokitteluista esitettiin yksityiskohtaisesti Gamman ja ym. (1995) kirjassa kuvattu luokittelu ja Buschmannin ja ym. (2002) kirjassa kuvattu Siemens-luokittelu. Erilaiset luokittelut nopeuttavat mallien opiskelua ja mahdollistavat uusien mallien löytämisen. Gamma-luokittelussa luokittelukriteereinä pidetään kohdetta ja tarkoitusta. Siemens-luokittelussa kriteereinä ovat ohjelmistotuotannon vaihe ja ongelmaluokitus. Ohjelmistontuotannon kannalta Siemens-luokittelu on mielestämme hyvä. Suunnittelumallien käyttötapa kuitenkin ratkaisee mitä luokittelua on syytä käyttää.

Luvun lopuksi esitettiin esimerkkinä Buschmannin ja ym. (2002) luokitteluun kuuluvia suunnittelumallien kaltaisia malleja. Ohjelmiston toteutusvaiheen mallina eli idiomina esitettiin C++-kielinen idiomi merkkijonoluokan toteutuksesta. Arkkitehtuuritason mallinna mainittiin kerrokset.

# 6 MALLIEN KÄYTTÄMINEN OHJELMISTOTUOTANNOSSA

Suunnittelumalleja voidaan käyttää eri tarkoituksissa ohjelmistotuotannossa. Malleja voidaan käyttää uudelleenkäytön[1] lisäämiseen ja tehostamiseen. Malleilla voidaan opettaa uusia suunnittelijoita. Malleja voidaan myös löytää. Ohjelmistotuotannon kannalta mallien tehokas käyttö lähtee näistä näkökulmista.

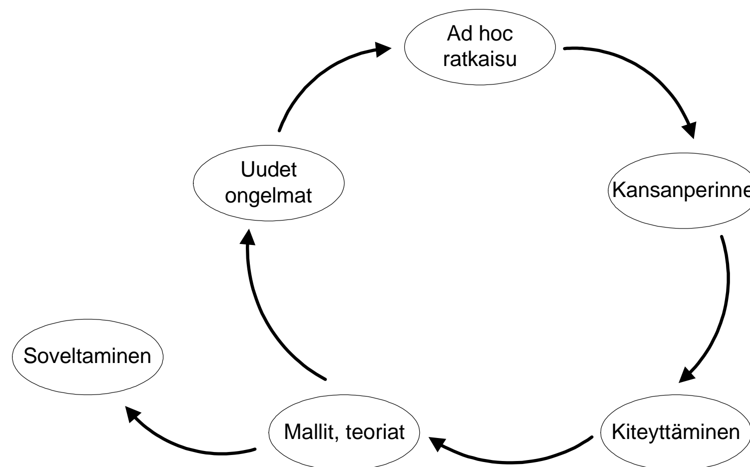
Sovelluksia tehtäessä tulee muistaa olemassa olevat mallit ja hyödyntää niiden tarjoamia ratkaisuja. Havaittaessa uusia toistuvia suunnitteluratkaisuja tulisi havaitut ratkaisut dokumentoida ja näin luoda uusia malleja.

## 6.1 Ohjelmistotuotannon kehittyminen

Ohjelmistotuotanto on nuori tieteenala. Termi ohjelmistotekniikka (engl. software engineering) otettiin käyttöön vasta Naton 1968 konferenssissa. Ohjelmistotuotannon historia on näin ollen nuori esimerkiksi verrattuna perinteisiin insinööritieteisiin. Shaw'n (1990) mukaan tekniikan alan kehitys noudattaa kehämäistä mallia, joka on esitetty kuviossa 9.

---

[1] Tässä uudelleenkäytöllä tarkoitetaan suunnitteluratkaisujen uudelleenkäyttöä.




**Kuvio 9. Shaw'n esittämä malli kansanperinteen kehittymisestä tieteeksi**

1. Ad hoc- menetelmät, toimiva ratkaisu.
2. Kansanperinne, tiettyä ratkaisua käytetään useassa tapauksessa.
3. Kiteyttäminen.
4. Mallien ja teorioiden kehittyminen.
5. Soveltaminen käytäntöön.

Ad-hoc ratkaisut ovat ensimmäisiä ratkaisuja ongelmaan. Näissä ratkaisuissa kukaan ratkaisun tuottajista ei ole aikaisemmin vastaavaa ongelmaa ratkaissut. Ko. ongelmaan kelpaa siis käytännössä mikä tahansa ratkaisu.

Kansanperinnevaiheessa ihmisillä alkaa olla kokemusta erilaisista ongelmista, ja ratkaisut tulevat henkilöiden omien kokemuksien kautta. Kokemus on tärkeää osaamista tässä vaiheessa. Kiteyttämisvaiheessa huomataan, että monessa ongelmassa käytetään samaa tai samankaltaista ratkaisua. Tämä yhteneväisyys

ratkaisuissa kiteyttää ihmisten kokemukset malleiksi ja teorioiksi. Mallien ja teorioiden soveltaminen käytäntöön onnistuu parhaiten siten, että ihmiset jotka eivät ole vastaavaa ongelmaa ratkaisseet, löytävät ongelmaan vastauksen malli-en ja teorioiden avulla.

Gamma ja ym. (1995) kuvaavat kuinka suunnittelumallit ovat löydetty valmiista järjestelmistä ja aina useammasta kuin yhdestä järjestelmästä. Löydetty ratkaisu on kiteytetty suunnittelumalliksi. Siksi suunnittelumallit ovat vahvasti ohjelmistotuotannon malli- ja teoriavaiheen osa. Tietyn ongelman ratkaisuun johtava tietotaito voidaan saada suunnittelumalleista, eikä näin tarvitse tuottaa ad hoc -ratkaisuja tai kansanperinnettä, eli uudetkin suunnittelijat voivat päästä hyväksi todettuun ratkaisuun helposti. Tämä auttaa ohjelmistontuotannon kehittymisen perinteisen tekniikan alan kaltaiseksi, jossa ratkaisut ovat rutiininomaisia

## 6.2 Mallien käyttäminen

Olemassa olevien mallien hyödyntämisessä ohjelmistotuotannossa voidaan käyttää seuraavaa menetelmää. Menetelmä mukailee yleisiä kirjallisuudessa esitettyjä tapoja. Menetelmä ei ota tässä kantaa siihen, missä ohjelmistotuotannon vaiheessa malleja sovelletaan. Näin ollen menetelmä on käyttökelpoinen suunniteltaessa järjestelmän kokonaisarkkitehtuuria, tai mahdollisesti yksittäisen moduulin sisäistä toteutusta. Mallikohtaisessa valinnassa ohjelmistotuotannon vaihe tulee tietenkin huomioida.

1. Pyri määrittelemään ja tunnistamaan ongelma
2. Tutki saatavilla olevia malleja
3. Valitse sopivat mallit tutustuen niiden kuvauksiin
4. Ratkaise ongelma käyttäen valittua mallia.

Vaikka suunnittelumallit ovat hyvä apu ohjelmistotuotannossa, niin on muistettava, ettei ole olemassa mitään hopealuotia. Kaikki ongelmat eivät välttämättä ole ratkaistavissa suunnittelumalleilla, koska jokainen ohjelmisto on itsenäinen kokonaisuus omine ongelmineen.

## 6.3 Malleista oppiminen

Gamma ja ym. (1995) kirjoittavat suunnittelumallien avulla oppimisesta seuraavasti: Suunnittelumallit tekevät sinusta myös paremman suunnittelijan. Ne tarjoavat ratkaisuja yleisiin ongelmiin. Jos työskentelet oliojärjestelmien parissa tarpeeksi pitkään, opit nämä mallit luultavasti itseksesi. Tämän kirjan lukeminen saa sinut oppimaan ne paljon nopeammin. Mallien opiskelu auttaa aloittelijaa työskentelemään enemmän kokeneen suunnittelijan tavoin.

Oppimisnäkökulma on mielestämme suunnittelumalleissa tärkeä, koska suunnittelumallit ovat ratkaisuja todellisiin ongelmiin. Suunnittelumalleissa on sitä suunnittelukokemusta, jota kokemattomilta suunnittelijoilta puuttuu, ja suunnittelumallien opiskelulla voidaan osittain korvata tämä kokemus.

Ajattelumaailman kehitys ei tapahdu vain yhden kirjan lukemisella, mutta tämä antaa hyvän lähtökohdan ajattelutavan muutokselle ja mielestämme nopeuttaa

oppimisprosessia paljon. Suunnittelumalleja ei voi mielestämme täysin ymmärtää, ellei niitä ole joskus soveltanut käytännössä.

Lähtökohtana voidaan pitää sitä, että mallien kuvaamien suunnitteluratkaisujen oppiminen aloittelijalle on mallien avulla paljon nopeampaa, kuin vastaavien ratkaisujen oppiminen kokemuksien kautta.

Suunnittelumallien opiskelu kannattaa aloittaa helpommista malleista. Gamma ja ym. (1995) kirjassaan ovat maininneet omista malleistaan helpoimmat (Abstrakti tehdas, Kuorruttaja, Operaatiorunko, Rekursiokooste, Sovitin, Strategia, Tarkkailija ja Tehdasmetodi), koska näiden mallien ymmärtäminen on yksinkertaista, ja näiden mallien ymmärtämisen jälkeen on helpompi oppia vaikeampia malleja.

## 6.4 Suunnittelumallien louhiminen

Mikäli samantyyppinen ongelma esiintyy usein eikä tämän ongelman ratkaisemiseksi ole olemassa sopivaa suunnittelumallia, voidaan yrittää löytää uusi suunnittelumalli. Buschmann ja ym. (2002) kutsuvat mallien löytämistä niiden louhimiseksi (engl. mining). Louhinta voi noudattaa seuraavaa Buschmannin ja ym. (2002) esittämää tapaa.

1. Löydä ainakin kolme esimerkkiä, joissa nimenomainen suunnittelu- tai toteutusongelma on ratkaistu tehokkaasti käyttäen samanlaista ratkaisukaavaa. Esimerkit pitää olla erilaisista käytössä olevista järjestelmistä, ja jokaisen järjestelmän tulisi olla luonut eri ryhmä.

2. Pura ratkaisukaava abstrahoimalla yleinen ratkaisukaava konkreettisista esimerkeistä. Kuvaa ongelma, johon ratkaisukaava osoittaa, ja voimat

jotka yhdistävät ongelman. Käytä soveltuvin osin mallien kuvaamisen sapluunaa. Listaa esimerkit, joista olet johtanut ratkaisukaavan tunnetuiksi käyttökohteiksi.

3. Julista ratkaisukaava mallikandidaatiksi.

4. Käynnistä kirjoittajien työpaja kehittääksesi mallikandidaattia ja jaa se kollegoillesi.

5. Sovella mallikandidaattia oikeassa sovellusprojektissa.

6. Julista mallikandidaatti malliksi, jos sovellus on onnistunut, ja integroi se mallijärjestelmääsi. Kehitä sen kuvausta käynnistämällä toinen kirjoittajien työpaja, ja lisää tämä uusi sovellus tunnettujen käyttökohteiden listaan. Jos mallikandidaatti epäonnistuu, kehitä sen kuvausta ottamalla opiksesi, ja kokeile soveltaa uudelleen. Vaihtoehtoisesti harkitse kandidaatin hylkäämistä kokonaan, ja etsi parempi ratkaisu alkuperäiseen ongelmaan.

## 6.5 Suunnittelumallien yleisiä käyttökokemuksia

Beck ym. (1996) esittää ohjelmistoteollisuuden kokemuksia suunnittelumalleista. Lähteessä esitetyt tulokset tukevat hyvin kirjallisuudessa esitettyä teoriaa.

**Mallit toimivat ohjelmistoteollisuudessa hyvin ryhmäviestinnän apuna.** Malleista tietoiset suunnittelijat keskustelevat suunnitteluongelmista käyttäen mallien nimiä pyrkiessään kuvaamaan ongelmaa.

**Suunnittelumallit ovat havaittavissa hyvin suunnitelluissa järjestelmissä**. Suunnittelumallit tavoittavat toimivan suunnittelun perusolemuksen muodos-

sa, joka mahdollistaa suunnitteluratkaisujen uudelleenkäytön tulevaisuuden projekteissa.

**Mallit kiinnittävät suunnitteluun liittyvä olennaiset osat kompaktiin esitysmuotoon**. Tämä esitysmuoto auttaa suunnittelijoita ja ylläpitäjiä ymmärtämään arkkitehtuuria sotkematta tai vääristämättä sitä. Malleilla voidaan tuottaa implisiittisestä nk. hiljaisesta tiedosta eksplisiittistä tietoa, ja näin tehostaa ohjelmistotuotantoa.

**Malleja voidaan käyttää tallentamaan ja edesauttamaan parhaiden toimintatapojen käyttämistä.** Tämä on erityisen tärkeää, että saadaan vähemmän kokeneet suunnittelijat tuottamaan parempaa suunnittelua nopeammin. Mallikirjat ovat hyvä apu opetuksessa, mutta niiden tarjoamia ohjeita ei saa noudattaa sokeasti.

**Suunnittelumallit eivät ole välttämättä olio-keskeisiä**. Huolimatta siitä, että suunnittelumallit tulevat olio-ohjelmointi–ihmisiltä, suunnittelumalleja voidaan teollisuudessa löytää kaikista eri ohjelmointiparadigmoilla toteutetuista ohjelmista.

**Suunnittelumallien hyväksynnän lisäämiseen ja niiden käytön tehostamiseen voidaan käyttää mentoreita.** Mentorit voivat auttaa kehittäjiä tunnistamaan suunnittelumallit, joita kehittäjät tietämättään käyttävät tuottamissaan ratkaisuissaan. Mentorien avulla kehittäjiä voidaan opastaa uudelleenkäyttämään kehittäjien tekemiä suunnitteluratkaisuja tulevissa projekteissa. Mentorien tärkeänä tehtävänä on lisäksi katsoa, ettei suunnitteluongelmiin yritetä väkisin käyttää malleja, jotka eivät sovellu niihin.

**Hyvät mallit ovat vaikeita ja aikaa vieviä tuottaa.** Mallien kirjoittaminen on taito, joka ei synny helposti. Mallien kirjoittamien on yleensä iteratiivinen prosessi, jossa ”löydetty” malli esitetään toisille ja yritetään liittää projekteihin. Malleja kommentoidaan, ja annetut kommentit pyritään hyödyntämän ja iterointia jatketaan toisissa projekteissa.

**Suunnittelumallien ymmärtämisessä käytäntö on kaikkein tärkeitä.** Kokemattoman suunnittelijan on hyvä tutkia ja opetella malleja käytännön esimerkkien kautta. Nähtyään esimerkkejä riittävästi, hän alkaa hiljalleen ymmärtää mallien todellisen arvon. Ymmärtämisen jälkeen suunnittelija pystyy tunnistamaan ja kirjoittamaan malleja itse.

## 6.6 Omia käyttökokemuksia

Yhdymme edellisessä alaluvussa esitettyihin ajatuksiin, toistamatta niitä. Yksi huomaamamme asia suunnittelumallien käytössä on se, että usein ainoastaan yhden mallin käyttämisellä ei onnistuta ratkaisemaan kokonaisongelmaa, vaan suunnittelumallit ovat luonteeltaan ehkäpä vastaavasti, kuten johdannossa mainitut Alexanderin mallit, generatiivisia. Malleja yhdistämällä voidaan ratkaista isompia ongelmia. Generatiivisuudesta seuraa useasti tiettyjen mallien käyttämisessä ajautuminen käyttämään toista suunnittelumallia ja tämä jälkeen seuraavaa. Triviaalissa Pursuitissa ruutujen luominen toteutettiin luontevasti käyttäen ensin Rakentaja-mallia ja tämän jälkeen Strategia-mallia.

Opiskellessa suunnittelumalleja on mentorilla tärkeä rooli. Mentoriksi tulisi asettaa henkilö, joka tietää miten malleilla voidaan suunnitteluongelmia ratkaista. Mikäli mentori ei ymmärrä itsekään malleja tarpeeksi syvällisesti, sorrutaan mallien käytössä helposti tekemään virhe, jossa suunnitteluongelmaa ei analy-

soida ja ymmärretä riittävän syvällisesti. Tällaisessa tilanteessa mallien käytöllä aiheutetaan enemmän haittaa kuin hyötyä. Vierailija on erinomainen esimerkki suunnitteluongelman vähäisestä ymmärtämisestä ja tilanteesta, jossa mallia yritetään väkisin soveltaa ongelmaan, johon se ei sovellu.

Ohjelmoitaessa suhteellisen vähällä kokemuksella, ohjelmointi on osaksi kielen rakenteiden ja syntaksin opettelemista. Ohjelman suunnittelu saattaa jäädä tällöin vähäisemmälle painoarvolle. Kuitenkin suunnittelulla tulisi olla iso merkitys ohjelman kehityksessä.

Suunnittelumallien kuvaamien rakenteiden tunnistamien ja mahdollinen soveltaminen omiin suunnitelmiin saattaa olla hyvinkin vaikeaa. Suunnittelumallien soveltamista tulisi yrittää käyttää vasta kun on varma, että malli sopii ko. ongelmaan.

Suunnittelumallit ovat mielestämme ohjelmoijalle hyvä apu ja niiden avulla pystyy ymmärtämään paremmin hyviä suunnittelutapoja. Suunnittelumallien opettaminen tulisi esittää kuitenkin viimeisenä opetettaessa suunnittelua, koska aloittelijan on liian helppoa ottaa oletettu malli nimen perusteella käyttöön ja soveltaa sitä ”väkisin” ongelman ratkaisuksi.

## 6.7 Suunnittelumallien käytön etuja ja haitat

Parhaimmillaan suunnittelumalli on abstrakti suunnitteludokumentti, joka voidaan ottaa käyttöön ohjelmiston suunnittelussa heti, kun suunnittelija tunnistaa ongelmasta kohdan, johon malli sopii. Yleistä tietämystä (osa ohjelmoijan yleissivistystä) olevien mallien lisäksi suunnittelumallit ovat erinomainen tapa kerätä talteen organisaatiossa olevaa sovellusaluekohtaista tietoa, joka näin säilyy, vaikka ihmiset vaihtuvatkin. (Rintala ja ym. 2001)

Suunnittelumallien käytöllä voidaan saavuttaa seuraavat edut:

- Voidaan tuoda testatut ja hyväksi todetut ratkaisut ohjelmistojen kehitysprosessiin.
- Ei tarvitse keksiä ruutia uudelleen.
- Voidaan jakaa tietoa, jolloin muistakin voi kehittyä eksperttejä.
- Mallien käyttö kasvattaa hyviä suunnittelijoita.

Suunnittelumallit voivat kattaa verraten suuren osan ohjelmistosuunnittelusta. Joidenkin asiantuntijoiden mukaan suunnittelumalleilla voidaan kuvata normaalitapauksessa 80% koko ohjelmistosta. (Haikala ja ym. 2000)

Haittoina voidaan pitää seuraavia:

- Väärän mallin käyttäminen.
- Mallien käytöllä ei aina välttämättä saavuteta selkeitä ratkaisuja.
- Ihmiset, joille suunnittelumallit eivät ole tuttuja, voivat kokea heille annetusta yksinkertaisesta koodista, että mallien käyttö on raskasta, ja abstraktio ei ole selvä heille. (MacDonald 1996) Tämän voi johtaa helposti siihen, että joudutaan tarkastelemaan jatkuvasti, mitä yläluokat määrittelevät ja kuinka malli toimii järjestelmässä, sillä useissa malleissa on käytetty perintää.

Suunnittelumalleilla sanotaan yleensä olevan vaikutusta ylläpidettävyyteen. Prechelt ja ym. (2001) tutkivat mallien vaikutusta ylläpidettävyyteen ja positiiviset asiat tulivat esille, mutta joissain tapauksissa yksinkertaisempi ratkaisu oli

ylläpidettävämpi kuin malleihin perustuva. Suunnittelumalliratkaisu saattoi olla liian raskas käyttää.

”Näyttää siltä, että suunnittelumallit, kuten kaikki ohjelmisto ja suunnittelutyökalut, pitää arvioida ennen käyttöä. Oikeassa ympäristössä suunnittelumallit voivat olla hyödyllinen työkalu suunnitteluun ja kommunikaatioon, mutta niiden käytöstä joutuu maksaan tietyn hinnan ja se pitää ottaa huomioon.” (MacDonald 1996)

## 6.8 Yhteenveto

Tässä luvussa käsiteltiin suunnittelumallien käyttämistä ohjelmistontuotannossa. Aluksi esiteltiin esitettiin Shaw'n (1990) kuvaama malli tekniikanalan kehittymisestä kansanperinteestä tieteeksi. Seuraavaksi esitettiin yleinen menetelmä miten suunnittelumalleja voidaan hyödyntää ohjelmistonkehityksessä. Tämän jälkeen käsiteltiin kuinka mallien avulla vähemmän kokeneet suunnittelijat voivat tehostaa oppimistaan.

Mikäli samantyyppinen ongelma esiintyy usein, eikä sen ratkaisemiseksi ole sopivaa suunnittelumallia voidaan suunnittelumalli louhia. Alaluvussa 6.4 esitettiin Buschmannin ja ym. (2002) kuvaama tapa louhia malleja. Luvun lopuksi esitettiin suunnittelumallien käyttökokemuksia ja suunnittelumallien käytöstä saavutettavat hyödyt ja mahdolliset haitat.

# 7 VASTAMALLIT

”Tavoitteena tunnistaa annetun kuvauksen perusteella ohjelmiston, projektin tai prosessin rakenteessa oleva perustavanlaatuinen ongelma, johon vastamallit (engl. antipattern) tarjoaa korjausehdotuksen tai ratkaisun vähentää ongelman kriittisyyttä.” (Brown 1998)

## 7.1 Mitä ovat vastamallit

Vastamalli on formaali dokumentointimuoto, joka kuvaa yleisiä ja useasti toistuvia ohjelmistotuotannon ongelmia, joilla on pitkäkestoisia seurauksia (Brown ja ym. 1998).

Vastamallit ovat malleja, mutta suunnittelumalleihin erona on, että vastamallit kuvaavat ratkaisun, joka selvästi tuottaa epäedullisen seurauksen ja antavat tähän ongelmaan paremman ratkaisun.

Vastamallitkaan eivät ole hopealuoti. Tosin ne ovat jossain määrin jopa toimivampi tapa ratkaista ongelmia kuin suunnittelumallit, koska ne auttavat organisaatiota laajempien ongelmien ratkaisuissa, eivätkä ainoastaan keskity suunnittelumallien tapaan kuvata suunnitteluratkaisuja.

Suunnittelumallit kuvaavat hyvät suunnitteluratkaisut. Vastamallit kuvaavat huonot suunnittelun, arkkitehtuurin tai projektinjohtamisen ratkaisut, eli tilanteet, joissa samat ongelmat toistuvat ohjelmistontuotannossa.

## 7.2 Vastamallien kuvaaminen

Suunnittelumallien luokittelu on jo edellä esitelty. Vastamallit luokitellaan samantyylisesti erilaisiin kategorioihin: ohjelmistonkehitys-, arkkitehtuuri- ja pro-

jektinhallintamalleihin. Jokaisen kategorian alla mallit on kuvattu Gamma-tyylistä kuvaustapaa käyttäen: nimi, aliakset, mittakaava, uudelleenmuotoillun ratkaisun nimi, uudelleenmuotoillun ratkaisun tyyppi, perussyyt, epäsuhtaiset resurssit, näytetarina, tausta, yleinen muoto vastamallista, oireet ja seuraukset, tyypilliset syyt, tunnetut poikkeukset, uudelleenmuotoiltu ratkaisu, muunnelmat, esimerkki ja yhteenkuuluva ratkaisu.

## 7.3 Vastamallien käyttäminen

Ohjelmistontuotannon projektien epäonnistuminen on suurin ongelma koko alalla ja tietysti koko ajan etsitään sitä uutta hopealuotia, joka auttaisi tähän ongelmaan. Yleensä organisaatiota kehitetään erilaisilla metodeilla, prosessien hallinnalla, projektin hallinnalla, uusilla prosesseilla, uudelleen käytön lisäämisellä tai jollain vastaavalla tavalla.

Vastamallien käyttö organisaatiossa voi olla todella vaikeaa, koska yleensä henkilöstöllä on todella kova muutosvastarinta. Muutosvastarintaa muodostuu luonnollisesti, kun aletaan arvostella ihmisten työtä ja toimintatapoja.

Osaamisen lisääminen ehkäisee vastamallien muodostumista organisaatioon. Esimerkiksi suunnittelumallien osaamaton käyttö väärien ongelmien ratkaisemiseen vain monimutkaistaa järjestelmää antamatta mitään apua uudelleenkäyttöön tai ylläpidettävyyteen.

## 7.4 Vastamalliesimerkki: leikkaa ja liitä -ohjelmointi

**Nimi**: leikkaa ja liitä -ohjelmointi (cut and paste programming).

**Alias**: Clipboard Coding, Software Cloning, Software Propagation.

**Aiheuttaja:** Laiskuus.

**Tausta:** Leikkaa ja liitä ohjelmointi on todella yleinen ongelma, joka samalla pahentaa ohjelman uudelleenkäyttöä ja tekee ylläpidon vaikeaksi. Malli johtuu siitä, että on helpompi käyttää vanhaa kuin ohjelmoida paremmalla ratkaisulla.

**Yleinen muoto**: Tämän vastamallin löytää usein monesta eri kohdasta ohjelmistoa ja se on siroteltu pitkin koodia, niin että sen ylläpito on todella raskasta. Yleensä projektissa on useita ohjelmoijia ja aloittelevat ohjelmoijat oppivat käytännön tapoja kokeneemmilta, olivat ne sitten hyviä tai huonoja. Tämä luo koodin monistusta, mikä saattaa lyhyellä ajalla vaikuttaa positiivisesti ainakin koodirivien määriä mitattaessa. Ohjelmakoodirivien määrä on Pressmanin (2000) mukaan yksi välitön ohjelmistotuotannon mittari.

**Oireet ja seuraukset:** Sama virhe toistuu useassa kohtaa ohjelmassa ja paikallinen korjaus ei poista virhettä. Koodirivien määrä kasvu ei lisää todellisuudessa suoraan tuottavuutta. Koodin katselmoinnit ovat turhaan laajempia.

**Tunnettu poikkeus:** Leikkaa ja liitä -ohjelmointi on sallittua, kun on ainoastaan tärkeää saada ohjelmisto valmiiksi välittämättä mitään sen ylläpidettävyydestä.

**Ratkaisu:** Perintä olio-ohjelmoinnissa ratkaisee yleisesti käytetyn leikkaa ja liitä ohjelmoinnin. Taulukoiden ja silmukoiden tehokas käyttö on toinen ongelman ratkaisu.

Esimerkki on otettu Brownin (1998) kirjasta suoraan ja sitä on lyhennetty. Esimerkistä saa hyvän kuvan siitä, mitä vastamallilla tarkoitetaan. Delphiä käytettäessä tulee helposti tehtyä Blob-vastamalli, sillä koko ohjelman logiikka on helposti IDE:n avustuksella rakennettavissa päälomakkeelle (eng. main form).

Tosin tämä on enemmän kiinni ajattelusta kuin ohjelmointikielestä. Blob-vastamallissa yksi olio tekee kaiken työn ja muut oliot toimivat vain tietovarastoina.

## 7.5 Yhteenveto

Tässä luvussa esiteltiin suunnittelumalleille läheinen, mutta vastakohtainen malli, vastamalli. Luvun alussa esitetään vastamalli-käsite yleisesti. Yleisen kuvauksen jälkeen esitetään vastamallien kuvaaminen ja tämän jälkeen kerrotaan mitenkä niitä voidaan käyttää. Luvun lopuksi esiteltiin esimerkkinä leikkaa ja liitä ohjelmointi -vastamalli.

# 8 YHTEENVETO

Tässä tutkielmassa esitettiin suunnittelumallikäsite, erilaisia tapoja luokitella suunnittelumalleja, suunnittelumallien soveltaminen esimerkkitapauksella, vastamallikäsite ja lisäksi tutkielmassa esitettiin suunnittelumallien hyödyntämistä ohjelmistotuotannossa.

Johdannossa esitettyihin ohjelmistokriisin ongelmiin suunnittelumallit tarjoavat vähän apua. Ihmissuden tappamiseksi tulee apua etsiä myös muualta. Seuraavassa esitetään näkökulmamme suunnittelumallien suhteista eri ongelmakohtiin.

### Monimutkaisuus

Mielestämme suunnittelumallit eivät poista ohjelmistoille tyypillistä monimutkaisuutta, mutta niiden avulla monimutkaisuutta kyetään hallitsemaan ja vähentämään. Suunnittelumallit ovat hyväksi koettuja suunnitteluratkaisuja. Huonoilla suunnitteluratkaisuilla ohjelman monimutkaisuus todennäköisesti kasvaa.

### Muunnettavuus

Useissa suunnittelumalleissa esiintyvät ratkaisut pohjautuvat dynaamisissa kielissä sisään rakennettuihin mekanismeihin. Pyrkimällä suunnittelussa dynaamisiin ratkaisuihin tuetaan ohjelmiston tulevaa ylläpitoa ja muunnettavuutta. Suunnittelumallien oikealla käytöllä mahdollistetaan järjestelmän parempi muunnettavuus. Vaarana on kuitenkin se, että jos ohjelmasta tehdään liian mukautuvainen, niin siitä tulee liikaa resursseja kuluttava. Lisäksi ohjelman tuottaminen vaatii enemmän ajattelua ja ohjelmointia. Mielestämme onnistuneessa

ohjelmiston suunnittelussa pitää päättää etukäteen, mistä osista ohjelmistoa tehdään joustava ylläpitoa silmällä pitäen ja mitkä osat jätetään staattisemmiksi.

### Mukautuvuus

Suunnittelumallit eivät ota nähdäksemme mukautuvuusongelmaan suoranaisesti kantaa.

### Näkymättömyys

Suunnittelumallien avulla voidaan osia ohjelmistosta nähdä abstrahoituina kokonaisuuksina. Käyttämällä suunnittelumalleja suunnittelijoiden välinen kommunikaatiota tehostaa. Suunnittelumalleilla on helppo kuvata ohjelmistojen yleinen rakenne, mutta yksityiskohtien kuvaamiseen tarvitaan muita kuvaustapoja.

Suunnittelumallien käytön kokemus yhdistettynä kirjallisuudessa esitettyihin teorioihin synnytti suunnittelumalleista seuraavan näkemyksen. Suunnittelumalleissa yhdistyy ohjelmistonkehitystä koskeva osaaminen ja suunnittelukokemus. Tämä osaaminen tarjotaan suunnittelijalle käyttäen yhtenäistä esitystapaa, jossa mallin kuvaava nimi on yksi keskeisimpiä ominaisuuksia. Luokiteltuna suunnittelumallit muodostavat ratkaisumallikokoelmia.

# 9 LÄHTEET

Alexander C., 1996 The Origin of Pattern Theory the Future of the Theory, And The Generation of a Living World. ACM Conference on Object-Oriented Programs, Systems, Languages an Applications [viitattu 3.2.2004]. Saatavilla www-muodossa <http://www.patternlanguage.com/archive/ieee/ieeetext.htm>

Alexander C., Ishikawa S., Silverstein M., Jacobson M., Fiksdahl-King I., Angel S., 1977. A Pattern Language: Towns, Buildings, Construction. Oxford University Press.

Alexander C. 1979, TheTimeless Way of Building. Oxford University Press.

Beck K., Coplien J., Crocker R., Dominick L., Meszaros G., Paulisch F., Vlissides J. 1996. Industrial Experiences with Design Patterns. IEEE Proceedings of ICSE -18.

Borland, 2002. Delphi Studio white paper. Technical Publications Borland Software Corporation [viitattu 13.4.2004]. Saatavilla www-muodossa <http://www.borland.com/products/white_papers/del_whats_new_in_borland_delphi7_studio.htm>

Brooks F. 1986. No Silver Bullet: Essence and Accidents of Software Engineering. Information Processing 86, 1069-1079. Elsevier [viitattu 18.4.2004]. Saatavilla www-muodossa <http://www-inst.eecs.berkeley.edu/~maratb/readings/NoSilverBullet.html tarkastettu 18.4.2004>

Brown W., Malveau R., McCormick III H., Mowbray T., 1998. Anti Patterns – Refactoring Software Architectures and Projects in Crisis. Wiley.

Budd T., 2002. An Introduction to Object Oriented Programming. Addison-Wesley.

Buchmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., 2002. Pattern-Oriented Software Architecture - a System of Patterns. Wiley.

Cantù M., 2003. Mastering Delphi 7. Sybex.

Gamma E., Helm R., Johnson R., Vlissides J., 1995. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley.

Garlan D., Shaw M., 1994. An Introduction to Software Architecture. Technical report CMU-CS sivut 94-166. Carnegie Mellon University [viitattu 2.3.2004]. Saatavilla www-muodossa <http://www-2.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf>

Haikala I., Märijärvi J., 2000. Ohjelmistotuotanto. Satku.

Hirvonen P. 2002. Ohjelmoinnin peruskurssi C-kieltä käyttäen. Jyväskylän yliopistopaino.

Holub A. 2003. Why getter and setter methods are evil – Make your code more maintainable by avoiding accessors. Java World [viitattu 2.3.2004]. Saatavilla www-muodossa <http://www.javaworld.com/javaworld/jw-09-2003/jw-0905-toolbox.html>

Holub A. 2004. More on getters and setters – Build user interfaces without getters and setters. Java World [viitattu 2.3.2004]. Saatavilla www-muodossa <http://www.javaworld.com/javaworld/jw-01-2004/jw-0102-toolbox.html>

Holub A. 2003 Design-Pattern quick reference [viitattu 14.4.2004]. Saatavilla www-muodossa <http://www.holub.com>

Itkonen J., 2004. TIE355 Ohjelmistoarkkitehtuurit, Bosch QASAR. Kopijyvä.

Koskimies K., 2001. Oliokirja. Satku.

Koskinen J., Sakkinen M., Paakki J., 2001. Ohjelmistotekniikka. Jyväskylän yliopiston yliopistopaino.

Lappalainen V., Malmirae P., 1998. Delphi 4 – Peruskurssi. Teknolit Oy.

Lappalainen V. 2003. Graafisten käyttöliittymien ohjelmointi 2003 [viitattu 21.3.2004]. <http://www.mit.jyu.fi/vesal/kurssit/winohj03/>

MacDonald S., 1996. Design Patterns in Enterprice. IBM Centre for Advanced Studies Conference. Toronto, Ontario, Canada. 12-14.11. 1996. IBM Press. sivu 25.

Prechelt L., Unger B., Tichy W., Brössler P., Votta L., 2001. A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions. IEEE Transactions on Software Engineering.

Pressman R., 2000. Software Engineering – A Practiotioner's Approach. MvGraw-Hill Publishing Company.

Naur P., Randell B., 1969. Software Engineering. Nato conference. Garmisch, Germany, 7-11.10. 1968. Scientific Affairs Division.

Rintala M., Jokinen J., 2001. Olioiden Ohjelmointi C++:lla. Satku.

Routio P., 2004. Metodi [viitattu 16.3.2004]. Saatavilla www-muodossa <http://www2.uiah.fi/projects/metodi/03l.htm>

Shaw M., 1990. Prospects for Engineering Discipline of Software. IEEE Software pp 15-24

Stroustroup B., 2000. C++-ohjelmointi. Teknolit Oy.

Swan T., 1999. Delphi 4. Teknolit Oy.

Vesterholm M., Kyppö J., 2003. Java-ohjelmointi. Talentum Media Oy.