KYMENLAAKSON AMMATTIKORKEAKOULU
Tietotekniikka / Ohjelmistotekniikka

Esa Knaapi

SUUNNITTELUMALLIT OHJELMOINNISSA

Opinnäytetyö 2013

TIIVISTELMÄ

KYMENLAAKSON AMMATTIKORKEAKOULU
Tietotekniikka

KNAAPI, ESA	Suunnittelumallit ohjelmoinnissa
Opinnäytetyö	61 sivua
Työn ohjaaja	Lehtori Teemu Saarelainen
Helmikuu 2013	
Avainsanat	ohjelmointi, suunnittelumallit, tietotekniikka, rakenne

Ohjelmoinnissa eräs suurimpia haasteita on ohjelmarakenteiden suunnittelu ja rakentaminen. Tämä huomattiin jo vuonna 1977, jolloin Christopher Alexander vertasi ohjelmien rakentamista talojen rakentamiseen ja totesi, että rakentaminen yleisesti perustuu toimivien ratkaisumallien keräämiseen, oppimiseen ja käyttämiseen. Ohjelmoinnin suunnittelumallit tulivat kuitenkin paremmin esiin vasta vuonna 1994, kun Erich Gamma julkaisi kirjansa suunnittelumalleista. Elämme nyt vuotta 2013, mutta kuinka moni ohjelmoija hallitsee tai tuntee suunnittelumallit? Ei kovinkaan moni ja se näkyy koodissa huonoina rakenteina ja tarpeettoman monimutkaisina ratkaisuina.

Tämä opinnäytetyö sisältää kaikki samat suunnittelumallit, mitä Erich Gamman kirjassa esiintyi. Näitä malleja on yhteensä 23 ja ne ovat: abstrakti tehdas, rakentaja, tehdasmetodi, prototyyppi, ainokainen, adapteri, silta, kooste, kuorruttaja, julkisivu, hiutale, edustaja, yleisfunktio, tulkki, tarkkailija, vastuuketju, välittäjä, iteraatio, komento, muisto, strategia, tila ja vierailija. Malleja on myöhemmin tullut lisääkin, mutta opinnäytetyössä vedettiin raja siihen, että käsitellään vain alkuperäiset suunnittelumallit. Opinnäytetyössä mainitaan lyhyesti myös antisuunnittelumallit ja annetaan niistä muutama esimerkki. Tämän jälkeen esitellään suunnittelumallit. Suunnittelumallin kohdalla kerrotaan lyhyesti, mitä sen on tarkoitettu tekevän, miten se rakentuu, ja jokainen sisältää myös yksinkertainen C#:lla tehdyn koodiesimerkin.

Suunnittelumallien esittelyn jälkeen niitä sovelletaan opinnäytetyön tekijän pelin uudelleenrakentamiseen. Alun perin peli muistutti rakenteellisesti lähinnä isoa klönttiä ja monet koodiratkaisuista oli toteutettu typerästi. Uudelleenrakentamisen jälkeen koodirivimäärä väheni yli 40 % ja ennen kaikkea se oli rakenteellisesti huomattavasti elegantimmin toteutettu.

ABSTRACT

KYMENLAAKSON AMMATTIKORKEAKOULU

University of Applied Sciences

Information Technology

KNAAPI, ESA	Software Design Patterns
Bachelor's Thesis	61 pages
Supervisor	Teemu Saarelainen, Senior Lecturer
February 2013	
Keywords	software, design, patterns, programming, it

In programming one of the major challenges is to design and build program structures smartly. This was noticed already in 1977, when Christopher Alexander compared programming to building houses and said that construction is commonly based on working design patterns, learning them and using them. Software design patterns became better-known after mid-1990s, when Erich Gamma published his book about software design patterns.

In this study all of the software design patterns, which were in Erich Gamma's book were discussed. Later people have invented even more patterns, but in this thesis only the original design patterns were included. In each pattern there were shortly told, what pattern has meant to do and how it is build. In addition short code examples about each pattern were given. Also anti-patterns were explained and a few examples of them were given.

In the end software design patterns were used to rebuild a thesis maker's game. Originally this game was poorly structured and there was much lousy coding work. After reconstructing the number of code lines decreased by 40 % and most importantly it is more elegant structurally.

Sisällysluettelo

Termit ja lyhenteet

OOPSLA-konferenssi

OOPSLA tulee sanoista: “Object-Oriented Programming, Systems, Languages & Applications”. Konferenssia on järjestetty vuodesta 1986 lähtien.

Pallopeli

Esa Knaapin projektina luotu pieni peli, joka on koodattu C#:lla ja käyttää Microsoftin luomaa XNA-pelikirjastoa.

1 JOHDANTO

Valitsin opinnäytetyökseni ohjelmoinnin suunnittelumallit, koska tajusin niiden suunnattoman hyödyn tutkiessani aihetta. Suunnittelumalleja ei yleisenä käsitteenä pidä rajoittaa pelkästään ohjelmointiin, sillä niitä on käytetty tuhansia vuosia rakentamisessa ja arkkitehtuurissa. Sitähän ohjelmointikin periaatteessa on, rakentamista ja arkkitehtuuria. Jos aletaan ohjelmoida isompaa kokonaisuutta, ilman minkäänlaisia suunnitelmia siitä, kuinka ohjelma tulisi rakentaa, niin tulos on lähtökohtaisesti huono, ellei peräti katastrofaalinen. Se on kuin lähtisi rakentamaan taloa ilman minkäänlaisia piirustuksia vasaraa heilutellen. Samalla tavalla monet myös ohjelmoivat, ikävä kyllä.

Tämän vuoksi suunnittelumallien osaaminen ja hallitseminen ohjelmoinnissa on hyvin tärkeää. Kun osaa ja ymmärtää eri suunnittelumallit, niin niitä voidaan käyttää ohjelman rakennetta mietittäessä tai ohjelmointiin liittyviä pulmia ratkaistaessa. Suunnittelumalleja ei kuitenkaan tule käyttää sellaisenaan. Näkisin ne pikemminkin suuntaviivoina, joita voidaan käyttää ratkaisujen kehittämisessä.

2 YLEISTÄ OHJELMOINNIN SUUNNITTELUMALLEISTA

Suunnittelumallit ovat sarja malleja, jotka on todettu hyväksi sekä teoriassa, että käytännössä. Suunnittelumalleja käytetään ratkaistaessa ohjelmoinnissa vastaan tulevia ongelmia ja tilanteita. Niitä voidaan käyttää myös työvälineenä suunniteltaessa ohjelman arkkitehtuuria. Niiden hyvä puoli on se, että niitä voidaan käyttää ilman mitään tiettyä tekniikkaa, suunnittelumenetelmää tai ohjelmointikieltä. Tosin suunnittelumallit on lähes aina toteutettu oliopohjaisesti, joten ohjelmointikielen olisi hyvä tukea sitä.

Suunnittelumalleja on keksitty suuri määrä ja joka päivä keksitään uusia ratkaisuja. Monet hyvät ratkaisut on mahdollista muuttaa sunnnittelumalliksi. Tässä opinnäytetyössä esitetyt suunnittelumallit ovat vanhoja ja pitkään hyväksi todettuja ratkaisuja, joita soveltamalla pääsee pitkälle.

2.1 Historia

Suunnittelumallien historia alkoi vuonna 1977, kun amerikkalainen arkkitehti Christopher Alexander esitti ajatuksen siitä, että rakentaminen yleisesti perustuu toimivien ratkaisumallien keräämiseen, oppimiseen ja käyttämiseen. Hän oli sitä mieltä, että näitä malleja tulisi käydä kehittämään myös ohjelmointiin liittyvien ongelmien ratkaisemiseen. (1, 15)

Myöhemmin Kent Beck ja Ward Cunningham tutustuivat Christopher Alexanderin ajatuksiin ja julkaisivat ensimmäiset suunnittelumallinsa OOPSLA-konfrenssissa vuonna 1987. Kyseisiä malleja he käyttivät olio-ohjelmoinnin suunnittelussa, mutta ne eivät olleet suunnittelumalleja sanan nykyisessä merkityksessä, vaan niitä voidaan pitää ohjelmoinnin käytettävyysmalleina. (1, 15)

Suunnittelumallien tärkeimpänä kehittäjänä pidetään kuitenkin Erich Gammaa, joka käsitteli suunnittelumalleja väitöskirjassaan vuonna 1991. Yhdessä Richard Helmin, Ralph Johnsonin ja John Vlissidesin kanssa hän julkaisi vuonna 1994 kirjan nimeltä Design Patterns: Elements of Reusable Object-Oriented Software ja siinä esiteltiin 23 suunnittelumallia.(1, 15)

2.2 Rakenne

Suunnittelumallien rakenne koostuu seitsemästä osasta. Nämä ovat: luokka, nimi, tarkoitus, rakenne, esimerkki, seuraukset ja käytettävyys.

1. **Luokka:** Mihin luokkaan suunnittelumalli kuuluu. Näitä luokkia on perinteisesti kolme: **luontimallit, rakennemallit ja käyttäytymismallit.** Myöhemmin on kuitenkin kehitetty malleja lisää. Esimerkiksi samanaikaisuusmallit.

2. **Nimi:** Kuvaava ja uniikki nimi auttaa tunnistamaan mallin, rakenteen ja sen käyttötarkoituksen.

3. **Tarkoitus:** Kuvaus siitä, miksi käyttää mallia ja mihin tarkoitukseen**.**

4. **Rakenne:** Kaavio ja selitys sille, miten malli muodostetaan.

5. **Esimerkki:** Ohjelmointikielinen esimerkki mallista.

6. **Seuraukset:** Mallin hyvät ja huonot puolet.

7. **Käytettävyys:** Milloin ja mihin mallia tulisi käyttää. (2, 6.)

2.3 Kritiikki

Epäillään, että suunnittelumallit on alun perin kehitetty korvaamaan ohjelmointikielien puuttuvia ominaisuuksia. Sellaisten kielien, kuten Java tai C++. Tietotekniikkatiedemies Peter Norvig onnistuikin demonstroimaan, miten 16 23 mallista yksinkertaistui tai hävisi kokonaan käytettäessä pidemmälle kehitettyjä kieliä, kuten Lispiä tai Dylania. (3.)

Peter Norvig on luultavasti oikeassa, mitä ohjelmointikielien puutteellisuuteen tulee. Ne todellakin kehittyvät hitaasti ja ovat lähtökohtaisesti puutteellisia. En kuitenkaan ole kovin vakuuttunut Lispin ja Dylanin paremmuudesta yleisemmin käytettyihin kieliin nähden. Tällä hetkellä oliopohjaiset kielet, kuten Java, C# ja C++ ovat maailman käytetyimpien ohjelmointikielten joukossa, eikä tähän näytä olevan tulossa muutosta kovin nopeasti, joten suunnittelumallien opettelemisesta ei voi pitää ajanhukkana.

Suunnittelumalleja on kritisoitu myös siitä, että ne voivat väärin käytettynä monimutkaistaa ohjelmaa tarpeettomasti(3).

Tämä pitää ilman muuta paikkansa. Vaikka suunnittelumallit ovatkin erinomaisia suuntaviittoja, niin ei niitä tietenkään pidä väkisin käyttää ohjelmoinnissa varsinkaan, jos ei tiedä mitä tekee. Se on itsestäänselvyys, että työkalua pitää osata käyttää ennen kuin käyttää sitä.

2.4 Antisuunnittelumallit

Antisuunnittelumallit ovat nimensä mukaan malleja siitä, miten asioita ei ainakaan kannata tehdä. Ne ovat joskus hyödyllisempiä kuin varsinaiset suunnittelumallit, koska niiden perusteella näkee helposti, jos jotain ollaan tekemässä väärin. Nämä mallit eivät rajoitu pelkästään ohjelmointiin, vaan niitä käytetään myös esimerkiksi yritystoiminnassa, projektien hallitsemisessa ja yleissuunnittelussa.

Pitäisin antisuunnittelumallienkin osaamista hyvin hyödyllisenä, koska ne osaamalla voidaan välttää sudenkuopat, joita ohjelmoinnissa tulee usein vastaan. Näitä sudenkuoppia ei ole keksimällä keksitty, vaan ne ovat tulleet vastaan käytännössä ja ne ovat johtaneet suuriin ongelmiin.

Antisuunnittelumalleja on todella paljon, mutta annan vain muutaman ohjelmointiin liittyvän esimerkin, joihin olen itse syyllistynyt.

1. Taikaluvut (Magic numbers): Käytetään literaaliarvoja nimettyjen vakioiden sijaan(4). Tähän syyllistyin työharjoittelussa tehdessäni ohjausta erääseen peliin. En todellakaan suosittele tekemään näin. Ohjelmasta tulee sekava, jos se sisältää lukuja, joiden alkuperää ei kuvata millään tavalla. Jos näitä joutuu käyttämään, niin ohjelman toteuttamista ei välttämättä ole mietitty tarpeeksi.

2. Neliskulmaisen pyörän uudelleenkeksiminen (Reinventing the square wheel): Luodaan huono ratkaisu, kun hyväkin olisi olemassa(5). Tähän päädytään usein, kun ei ole tarpeeksi tietoa tehtävästä asiasta. Olen oppinut kantapään kautta, että parempi opiskella dokumenteista materiaalia, ennen kuin tekee yhtään mitään. Jouduin tekemään ensimmäisessä pelissäni ison osan eräästä ratkaisusta uusiksi, koska se oli yksinkertaisesti typerä.

3. Tahaton monimutkaisuus (Accidental complexity): Väärä suunnittelu tai ongelman väärin ymmärtäminen lisää ratkaisuun yllättäviä ongelmakohtia(6). Tämä tuli vastaan täsmälleen samaan aikaan kuin edellinenkin. Kun ei ole tarpeeksi tietoa siitä, mitä tekee, niin tähän päädytään helposti. Hyvä ratkaisu on yleensä elegantin yksinkertainen.

4. Jumalaluokka (God class, God object): Luokka joka tekee tai tietää liian paljon, esimerkiksi toteuttaa suuren osan ohjelmasta. Nämä yhteenkasaumat tulee hajauttaa muihin luokkiin(7). Kohtasin tämänkin ongelman omaa peliä tehdessä. Koska peli ei ollut kovin laaja, ongelma ei päässyt kuitenkaan muodostumaan kovin suureksi. Olen kuitenkin varma, että niitä olisi alkanut ilmaantua, jos olisin jatkanut pelin kehittämistä pidemmälle.

5. Leikkaa ja liimaa -ohjelmointi (Cut and paste programming): Ohjelmakoodin hätäinen kopiointi ja sovitus aikaisemmasta toteutuksesta sen sijaan, että koodi

suunniteltaisiin uudelleenkäytettäväksi(8). Tähän olen syyllistynyt todella usein. Olen kuitenkin tätä kautta huomannut, että monissa tapauksissa olisi helposti ratkaistu ongelma yksinkertaisella funktiolla ilman, että koodia tarvitsee toistaa uudestaan ja uudestaan.

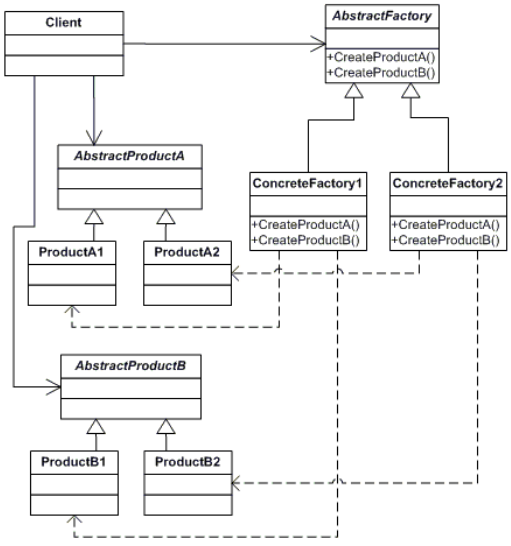
3 LUONTIMALLIT

Luontimalleissa keskitytään olioiden luomisprosesseihin. Näissä malleissa olioiden luominen siirretään toisille olioille eli olioiden luominenkin tullaan hoitamaan olio-ohjelmoinnin periaatteiden mukaisesti.

3.1 Abstrakti tehdas (Abstract Factory)

Abstrakti tehdas on olion luontimalli, jossa luomisluokkaa käytetään sitomaan yhteen samantyyppisiä luokkia.

Abstraktia tehdasta käytetään, kun halutaan luoda useita tuoteperheitä ja sovelluksen tulee valita, mitä niistä käytetään. Perheeseen kuuluvia olioita on tarkoitus käyttää ja yhdistellä vain keskenään ja ratkaisun on valvottava tätä toteutustapaa.



Kuva 1. Epäkonkreettinen tehdasrakenne. (9.)

Client lähettää tiedon AbstractFactorylle siitä, mitä olioperhettä kuuluu valmistaa ja tämä valmistaa oliot. Jos Client haluaisi esimerkiksi oliot luokista A1 ja B1, niin valittaisiin ConcreteFactory1.

Alla olevassa esimerkissä luodaan Asiakas, joka päättää, minkä maalaisia kulkuneuvoja valmistetaan. Jos Asiakkaalle lähetetään AmerikkaTehdas, niin ulos tulee Fordeja ja Baylinereitä. Jos Asiakkaalle lähetetään JapaniTehdas, niin ulos tulee Hondia ja Yamahoita.

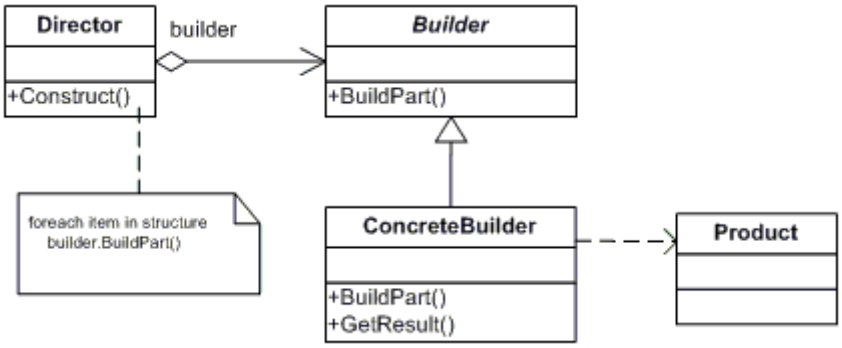
```
using System;
using System.Collections.Generic;

namespace Epäkonkreettinen_Tehdas
{
    class Program
    {
        static void Main(string[] args)
        {
            Asiakas asiakas = new Asiakas(new JapaniTehdas());
            asiakas.Toimi();
            Console.ReadKey();
        }
    }

    class Asiakas
    {
        private Auto auto;
        private Vene vene;

        public Asiakas(Kulkuneuvotehdas kulkuneuvotehdas)
        {
            auto = kulkuneuvotehdas.LuoAuto();
            vene = kulkuneuvotehdas.LuoVene();
        }

        public void Toimi()
        {
            auto.Toiminto();
            vene.Toiminto();
        }

    }

    abstract class Kulkuneuvotehdas
    {
        public abstract Vene LuoVene();
        public abstract Auto LuoAuto();
    }

    class AmerikkaTehdas : Kulkuneuvotehdas
    {
        public override Vene LuoVene()
        {
            Console.WriteLine("Tehtaasta tuli ulos Bayliner!");
            return new Bayliner();
        }

        public override Auto LuoAuto()
        {
            Console.WriteLine("Tehtaasta tuli ulos Ford!");
            return new Ford();
        }
    }

    class JapaniTehdas : Kulkuneuvotehdas
```

```
    {
        public override Vene LuoVene()
        {
            Console.WriteLine("Tehtaasta tuli ulos Yamaha!");
            return new Yamaha();
        }

        public override Auto LuoAuto()
        {
            Console.WriteLine("Tehtaasta tuli ulos Honda!");
            return new Honda();
        }
    }

    abstract class Vene
    {
        public abstract void Toiminto();
    }

    class Bayliner : Vene
    {
        public override void Toiminto()
        {
            Console.WriteLine("Bayliner liukuu ylväästi veden pintaa pitkin!");
        }
    }

    class Yamaha : Vene
    {
        public override void Toiminto()
        {
            Console.WriteLine("Yamaha liukuu ylväästi veden pintaa pitkin!");
        }
    }

    abstract class Auto
    {
        public abstract void Toiminto();
    }

    class Ford : Auto
    {
        public override void Toiminto()
        {
            Console.WriteLine("Ford kiitää uljaasti pitkin maanteitä!");
        }
    }

    class Honda : Auto
    {
        public override void Toiminto()
        {
            Console.WriteLine("Honda kiitää uljaasti pitkin maanteitä!");
        }
    }
}
```

(9.)

3.2 Rakentaja (Builder)

Rakentajaa käytetään, kun halutaan luoda olioita pala palalta ja pitää erillään luomisalgoritmi ja tapa, jolla osat yhdistetään.



Kuva 2. Rakentajan rakenne. (9.)

Director päättää, mitä tuotetta rakennetaan ja Builderi rakentaa Productin pala palalta.

Alla olevassa esimerkissä luodaan Lintutietelijä(Director), joka käyttää uusien lintulajien lisäämiseen KotkaRakentajaa ja PingviiniRakentajaa. Molemmissa tapauksissa lintu rakennetaan pala palalta. Tosin tässä tapauksessa palasia on vain kaksi: Nimi ja lentokyky.

```
using System;
using System.Collections.Generic;

namespace Rakentaja
{
    public class MainApp
    {
        public static void Main()
        {
            LintuRakentaja kotkarakentaja = new KotkaRakentaja();
            Lintutietelijä lintutietelijä = new Lintutietelijä();

            lintutietelijä.AsetaLintuRakentaja(kotkarakentaja);
            lintutietelijä.RakennaLintu();
            Lintu kotka = lintutietelijä.PalautaLintu();

            Console.WriteLine();

            LintuRakentaja pingviinirakentaja = new PingviiniRakentaja();
            lintutietelijä.AsetaLintuRakentaja(pingviinirakentaja);
            lintutietelijä.RakennaLintu();

            Lintu pingviini = lintutietelijä.PalautaLintu();
            Console.Read();
        }
    }
```

```
class Lintu
{
    public string Nimi = "";
    public string OsaakoLentää = "";
}


abstract class LintuRakentaja
{
    protected Lintu lintu;

    public Lintu PalautaLintu()
    {
        Console.WriteLine("Linnun nimi: " + lintu.Nimi);
        Console.WriteLine("Osaako lintu lentää: " + lintu.OsaakoLentää);
        return lintu;
    }

    public void LuoUusiLintu()
    {
        lintu = new Lintu();
    }

    public abstract void RakennaNimi();
    public abstract void RakennaOsaakoLentää();
}


class KotkaRakentaja : LintuRakentaja
{
    public override void RakennaNimi()
    {
       lintu.Nimi = "Merikotka";
    }

    public override void RakennaOsaakoLentää()
    {
        lintu.OsaakoLentää = "Osaa";
    }
}


class PingviiniRakentaja : LintuRakentaja
{
    public override void RakennaNimi()
    {
        lintu.Nimi = "Kuningaspingviini";
    }

    public override void RakennaOsaakoLentää()
    {
        lintu.OsaakoLentää = "Ei";
    }
}

//Lintutietelijä rakentaa linnun käyttämällä LintuRakentaja-luokkaa
class Lintutietelijä
{
    private LintuRakentaja _linturakentaja;

    public void AsetaLintuRakentaja(LintuRakentaja lr)
    {
        _linturakentaja = lr;
```

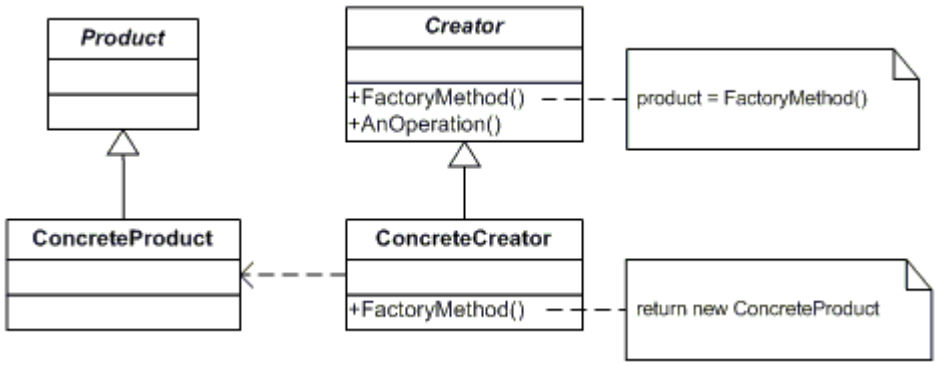
```
        }

        public Lintu PalautaLintu()
        {
            return _linturakentaja.PalautaLintu();
        }

        public void RakennaLintu()
        {
            _linturakentaja.LuoUusiLintu();
            _linturakentaja.RakennaNimi();
            _linturakentaja.RakennaOsaakoLentää();
        }
    }
}
```

(9.)

3.3 Tehdasmetodi (Factory Method)

Tehdasmetodi on olion luontimalli, jossa vastuu olioiden luomisesta siirretään toisille olioille.

Tehdasmetodi on muuten samanlainen kuin abstrakti tehdas, mutta tehdasmetodin tapauksessa luontiolioiden ei tarvitse valvoa sitä, että tietyt oliot luodaan samassa paketissa. Tehdasmetodi on suoraviivaisin kaikista luontimalleista. Se vain yksinkertaisesti palauttaa luotavan olion pyydettäessä.



Kuva 3. Tehdasmetodin rakenne. (9.)

ConcreteCreator palauttaa pyydettäessä ConcreteProduct-olion.

Valitaan halutaanko luoda Ferrari vai Lada. Jos halutaan luoda Ferrari, niin pyydetään FerrariLuojaa palauttamaan Ferrari. Jos halutaan luoda Lada, niin pyydetään Lada-Luojaa palauttamaan Lada.

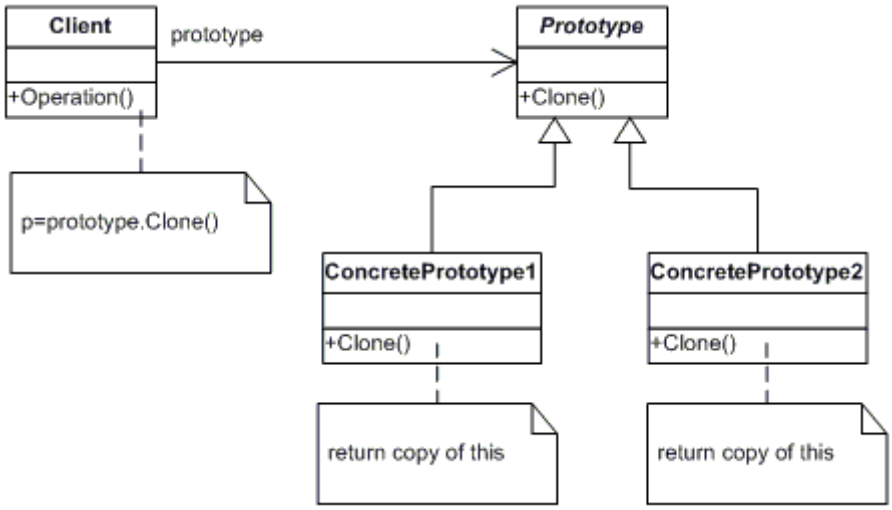
```
using System;

namespace DoFactory.GangOfFour.Factory.Structural
{
    class MainApp
    {
        static void Main()
        {
            Luoja LadaLuoja = new LadaLuoja();
            Luoja FerrariLuoja = new FerrariLuoja();

            Auto Ferrari = FerrariLuoja.TehdasMetodi();
            Auto Lada = LadaLuoja.TehdasMetodi();

            Console.ReadKey();
        }
    }

    abstract class Auto
    {
    }

    class Ferrari : Auto
    {
    }

    class Lada : Auto
    {
    }

    abstract class Luoja
    {
        public abstract Auto TehdasMetodi();
    }

    class FerrariLuoja : Luoja
    {
        public override Auto TehdasMetodi()
        {
            Console.WriteLine("Ferrari-olio palautettu");
            return new Ferrari();
        }
    }

    class LadaLuoja : Luoja
    {
        public override Auto TehdasMetodi()
        {
            Console.WriteLine("Lada-olio palautettu");
            return new Lada();
        }
    }
}
```

(9.)

3.4 Prototyyppimalli (Prototype)

Määrittelee prototyyppi-ilmentymää käyttämällä, millainen olio luodaan, ja luo uusia olioita tätä prototyyppiä kopioimalla.

Prototyyppimallia käytetään, kun sovellus halutaan pitää riippumattomana siitä, miten sen oliot luodaan, koostetaan ja esitetään. Sen lisäksi prototyyppimallia käytetään silloin, kun luotavien olioiden luokat halutaan määrittää vasta ajon aikana. Sitä käytetään myös silloin, kun ei haluta määritellä luokkahierarkian kanssa rinnakkaista tehtaiden luokkahierarkiaa. Ja lopuksi sitä kannattaa käyttää silloin, kun on vain muutama tila, jossa olio voi olla. On monesti tehokkaampaa luoda prototyypit eri tiloista ja kloonata niitä, kuin luoda luokista ilmentymiä manuaalisesti.



Kuva 4. Prototyypin rakenne. (9.)

Client luo uuden olion pyytämällä prototyyppiä kloonaamaan itsensä käyttämällä joko ConcretePrototype1:stä tai ConcretePrototype2:sta. Sen jälkeen ConcretePrototype1 tai 2 suorittaa itsensä kloonauksen ja palauttaa kloonin.

Esimerkissä luodaan prototyyppi eläinluokasta ja tätä prototyyppiä kloonaamalla luodaan klooni1 ja klooni2. Molemmat näistä ovat identtisiä kopioita prototyyppi-oliosta.

```
using System;
using System.Collections.Generic;

namespace Prototyyppi
{
    class Program
    {
        static void Main(string[] args)
        {
            //luodaan prototyyppi ja kloonataan se
            Eläin prototyyppi = new Eläin("Susi");
            Eläin klooni1 = prototyyppi.Kloonaa() as Eläin;
            Eläin klooni2 = klooni1.Kloonaa() as Eläin;

            klooni1.palautaNimi();
```

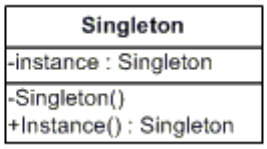
```
            klooni2.palautaNimi();

            Console.ReadLine();
        }
    }

    class Eläin
    {
        private string nimi = "";

        public Eläin(string nimi)
        {
            this.nimi = nimi;
        }

        public void palautaNimi()
        {
            Console.WriteLine("Hei olen "+nimi+"!");
        }

        public object Kloonaa()
        {
            return this.MemberwiseClone();
        }
    }
}
```

(9.)

3.5 Ainokainen (Singleton) (3)

Ainokainen varmistaa, että luokasta luodaan vain yksi ilmentymä, ja tarjoaa globaalin tavan päästä käsiksi tähän ilmentymään.(3)

Ainokaista voidaan käyttää, kun luokalla täytyy olla vain yksi ilmentymä ja sen täytyy olla kaikkien sovellutusten saatavilla. Ainokaista voidaan myös laajentaa käyttämättä aliluokkia, mutta se pitää tehdä niin, ettei ilmentymää käyttävien sovellutusten tarvitse muuttaa koodia.



Kuva 5. Ainokaisen rakenne. (9.)

Luodaan yksi luokka, jonka sisällä on yksi staattinen ilmentymä(Instance). Tätä instanssia kutsumalla voidaan luoda uusia olioita.

Esimerkissä luodaan olio nimeltä ainokainen käyttämällä Ainokaisluokan instanssia.

```
using System;
using System.Collections.Generic;

namespace Ainokainen
{
    class Program
    {
        static void Main(string[] args)
        {
            //olio voidaan luoda vain insanssin kautta, koska Ainokainen luokka
on protected
            Ainokainen ainokainen = Ainokainen.Instanssi();
        }
    }

    class Ainokainen
    {
        private static Ainokainen _instanssi;
        protected Ainokainen()
        {
        }

        public static Ainokainen Instanssi()
        {
            _instanssi = new Ainokainen();
            return _instanssi;
        }
    }
}
```

(9.)

3.6 Luontimallien yhteenveto

Välillä on vaikea päättää, mitä luontimallia tulisi käyttää. On kuitenkin yksi malli, mitä ei pidä käyttää. Se on ainokainen (Singelton). Äkkiseltään voi kuulostaa käytännölliseltä idealta, että on olemassa globaali instanssi, jota voi kutsua ihan mistä tahansa ja luoda sillä olioita. Juuri siinä piileekin ainokaisen suurin heikkous. Globaalit muuttujat eivät ole hyvä asia. Mitä isommaksi ohjelma muuttuu, sitä suurempi riski on, että jotain menee pahasti pieleen.

Abstraktia tehdasta käytetään, kun luotavat oliot halutaan luoda samassa paketissa.

Rakentajaa käytetään, kun halutaan luoda useammista palasista koostuva olio.

Tehdasmetodia käytetään, kun halutaan yksinkertaisesti vain siirtää vastuu olioiden luomisesta luokalle.

Prototyyppimallia käytetään, kun halutaan luoda dynaamisesti uusia olioita kloonaamalla valmiiksi olemassa olevia olioita.

Kun vastuu olioiden luomisesta siirretään oliolle, niin ollaan lähempänä oikeaa olio-ohjelmointia ja sitä paremmin ohjelma pysyy kasassa. Mitä vähemmän ohjelman sovellus tietää ja tekee, sitä parempi. Ja mitä suurempia oliokokonaisuuksia ohjelma tulee sisältämään, sitä enemmän luontimallien käyttämisestä on hyötyä ja sitä enemmän kannattaa miettiä, miten olioiden luominen tulee toteuttaa.

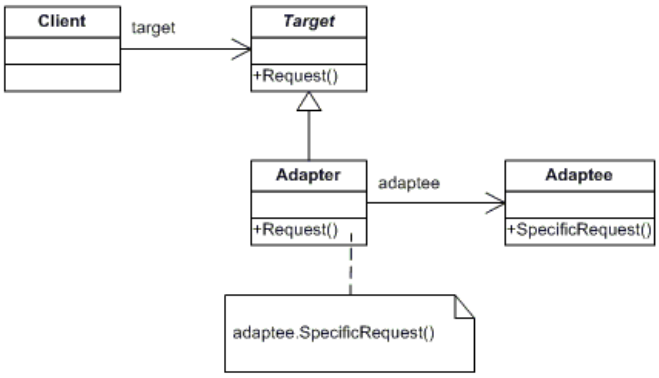
4 RAKENNEMALLIT

Nimensä mukaisesi rakennemallit antavat pohjan sille, minkälaisia rakenteita luokilla pystyy luomaan. Tietenkään rakennemallit eivät kata kaikkea mahdollista, mutta niitä soveltamalla päästään jo pitkälle.

4.1 Sovitin (Adapter, Wrapper)

Sovitinta käytetään toisiinsa sopimattomien luokkien toiminnallisuuksien yhdistämisessä.

Sovitin mallia ei käytetä kovinkaan usein. Monesti sovittimen käyttö voidaan välttää paremmilla luokkarakenteilla. Ylimääräinen sovitin-luokka voi nimittäin monimutkaistaa ohjelman rakennetta turhaan.



Kuva 6. Sovittimen rakenne. (9.)

Target-luokan ja Adaptee-luokan välille luodaan Adapteri, joka yhdistää näiden toiminnallisuuden keskenään. Adapter-malli muistuttaa seuraavaksi tulevaa Silta-mallia. Silta toimii tässä tapauksessa adapterina luokkien välillä.

Esimerkissä luodaan Eläin-luokka, jossa on metodina juokseminen. Koira perii Eläin-luokan ja juoksemisen. Siinä ei ole mitään ihmeellistä, että koira juoksee. Mitä jos halutaankin luoda lintu ja liittää sen toiminnallisuus Eläin-luokkaan? Linnut eivät juokse vaan lentävät! Siispä luodaan adapteri, joka yhdistää Eläin ja Lintu-luokat pistäen Linnun lentämään aina, kun adapterin käsketään juosta.

```
using System;
using System.Collections.Generic;

namespace Adapteri
{
    class Program
    {
        static void Main(string[] args)
        {
            Lintu lintu = new Lintu();
            Koira koira = new Koira();
            koira.Juoksee("Rekku");

             LintuAdapter lintuadapter = new LintuAdapter(lintu);
            lintuadapter.Juoksee("Kotka");
            Console.ReadLine();
        }
    }

    public abstract class Eläin
    {
        public void Juoksee(string nimi)
        {
            Console.WriteLine(nimi + " juoksee");
        }
    }

    class Koira : Eläin
    {
```

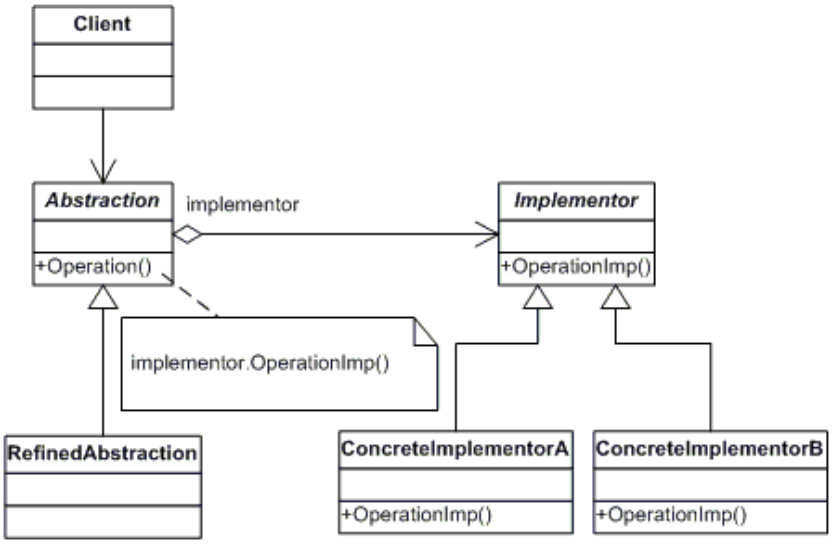
```
    }

    class LintuAdapter : Eläin
    {
        Lintu lintu;

        public LintuAdapter(Lintu newLintu)
        {
            lintu = newLintu;
        }

        public void Juoksee(string nimi)
        {
            lintu.Lentää(nimi);
        }
    }

    class Lintu
    {
        public void Lentää(string nimi)
        {
            Console.WriteLine(nimi + " lentää");
        }
    }
}
```

(9.)

4.2 Silta (Bridge, Handle, Body)

Erotetaan luokan toiminnallisuus toiseen luokkaan.

Muistuttaa rakenteellisesti hyvin paljon sovitinta, mutta käyttötarkoitus ja idea näissä luokissa ovat täysin erilaiset. Mitä enemmän luokan tilat ja sen toiminnallisuudet vaihtelevat, sitä hyödyllisemmäksi siltamallin käyttäminen tulee.



Kuva 7. Sillan rakenne. (9.)

RefinedAbstraction-oliolla voidaan toteuttaa operaatiot ConcreteImplementorA ja ConcreteImplementorB.

Silta muistuttaa rakenteeltaan hyvin paljon Sovitinta. Tätäkään asiaa ei kannata miettiä liian monimutkaisesti. Luokan yksi sisältä vain kutsutaan luokkaa kaksi. Tämä kutsuminen on ”silta”.

Esimerkissä luodaan Eläin-luokkia, joista halutaan tehdä luonto- ja ruokadokumentteja. Sen sijaan, että toteuttaisimme dokumentin Eläin-luokkien sisällä, se tapahtuukin Dokumentti-luokissa, joita kutsutaan Eläin-luokan sisältä. Tässä tapauksessa Dokumentti on Eläin-luokasta erotettua toiminnallisuutta.

```
using System;
using System.Collections.Generic;

namespace Silta
{
    class Program
    {
        static void Main(string[] args)
        {
            Eläin tiikeri = new Tiikeri();
            Eläin norsu = new Norsu();
            Dokumentti ruokadokumentti = new RuokaDokumentti();
            Dokumentti luontodokumentti = new LuontoDokumentti();

            tiikeri.Ohjelma(ruokadokumentti);
            norsu.Ohjelma(luontodokumentti);

            Console.ReadLine();
        }
    }
```

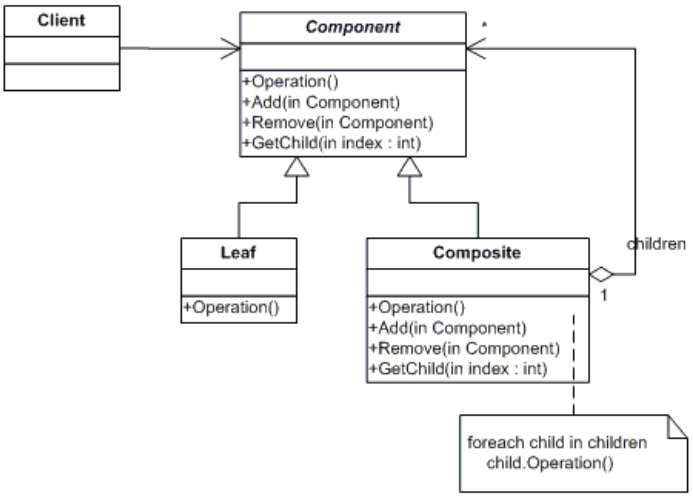
```
    abstract class Dokumentti
    {
        public abstract void Katso(string eläin);
    }

    class RuokaDokumentti : Dokumentti
    {
        public override void Katso(string eläin)
        {
            Console.WriteLine(eläin + " on hyvää, kun sen grillaa!");
        }
    }

    class LuontoDokumentti : Dokumentti
    {
        public override void Katso(string eläin)
        {
            Console.WriteLine(eläin + " elää luonnossa!");
        }
    }

    abstract class Eläin
    {
       protected Dokumentti dokumentti;

       public void Ohjelma(Dokumentti uusiDokumentti)
       {
           dokumentti = uusiDokumentti;
           dokumentti.Katso(GetType().Name);
       }


    }

    class Tiikeri : Eläin
    {
    }

    class Norsu : Eläin
    {
    }
}
```

(9.)

4.3 Kooste(Composite)

Koostemallissa luodaan olio-listoja, jotka voivat sisältää toisia samanlaisia olio-listoja tai ”olio-lehtiä”.

Yksinkertaistetusti koostemalli on puurakenteen luominen olioille. Tähän puuhun voidaan sitten lisätä ”oksia” ja ”lehtiä”. Tässä tapauksessa oksat ovat olioita, jotka sisältävät olio-listan ja lehdet niitä, jotka eivät sisällä.



Kuva 8. Koosteen rakenne. (9.)

Leaf ja Composite perivät Component luokan. Composite-luokka voi pitää sisällään sekä toisia Composite-luokkia, että Leaf-luokkia.

Jos listat ovat ohjelmoinnissa tuttuja, niin idean tajuaa helposti. Composite on luokka, jonka sisällä on lista ja tähän listaan voidaan varastoida Leaf- ja muita Composite-olioita.

Esimerkissä voidaan luoda eläinryhmiä ja niiden sisälle uusia eläinryhmiä tai eläimiä. Eläinryhmät ovat tässä tapauksessa puun ”oksia” ja eläimet ”lehtiä”.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Kooste
{
    class Program
    {
        static void Main(string[] args)
        {
            Eläinryhmä Nisäkkäät = new Eläinryhmä("Nisäkkäät");
            Nisäkkäät.Lisää(new Eläin("Susi"));
            Nisäkkäät.Lisää(new Eläin("Karhu"));
            Nisäkkäät.Tulosta();


            Eläinryhmä Linnut = new Eläinryhmä("Linnut");
            Eläinryhmä Petolinnut = new Eläinryhmä("Petolinnut");
            Eläinryhmä Pingviinit = new Eläinryhmä("Pingviinit");

            Petolinnut.Lisää(new Eläin("Kotka"));
            Pingviinit.Lisää(new Eläin("Keisaripingviini"));

            Linnut.Lisää(Petolinnut);
```

```
            Linnut.Lisää(Pingviinit);
            Linnut.Tulosta();

            Console.ReadLine();
        }
    }

    abstract class Eläimet
    {
        protected string nimi;

        public Eläimet(string uusiNimi)
        {
            nimi = uusiNimi;
        }

        public abstract void Lisää(Eläimet uusiEläin);
        public abstract void Poista(Eläimet uusiEläin);
        public abstract void Tulosta();
    }

    class Eläinryhmä : Eläimet
    {
        private List<Eläimet> _lapsi = new List<Eläimet>();

        public Eläinryhmä(string nimi)
            : base(nimi)
        {
        }

        public override void Lisää(Eläimet uusiEläin)
        {
            _lapsi.Add(uusiEläin);
        }

        public override void Poista(Eläimet uusiEläin)
        {
            _lapsi.Remove(uusiEläin);
        }

        public override void Tulosta()
        {
            Console.WriteLine(nimi);

            foreach (Eläimet eläimet in _lapsi)
            {
                eläimet.Tulosta();
            }
        }
    }

    class Eläin : Eläimet
    {
        public Eläin(string nimi)
            : base(nimi)
        {
        }

        public override void Lisää(Eläimet uusiEläin)
        {
            Console.WriteLine("Et voi lisätä eläimiä eläimeen");
        }

        public override void Poista(Eläimet uusiEläin)
        {
```

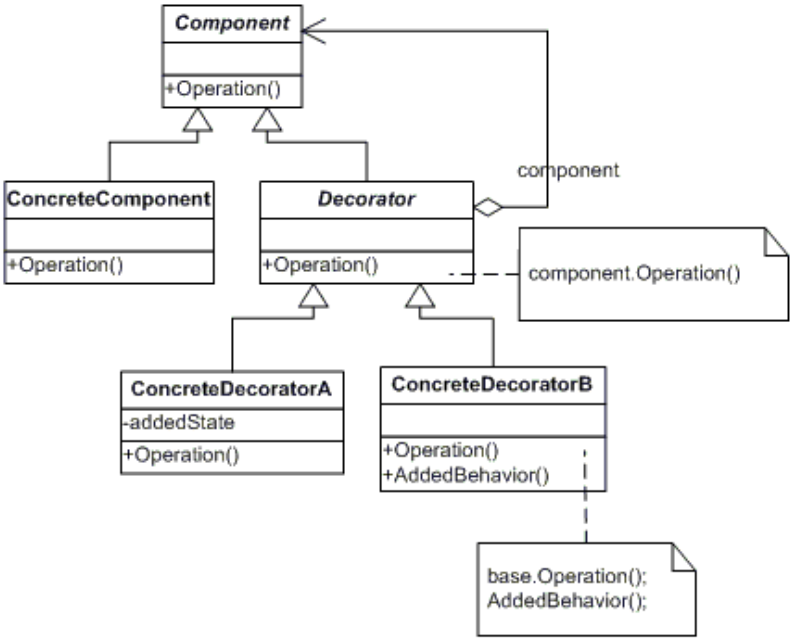
```
            Console.WriteLine("Et voi poistaa eläimiä eläimestä");
        }

        public override void Tulosta()
        {
            Console.WriteLine("-" + nimi);
        }
    }
}
```

(9.)

4.4 Kuorruttaja (Decorator)

Kuorruttajassa lisätään nimensä mukaisesti ”kuorrutteita” eli toiminnallisuutta kanta-olioon.

Kuorruttajamalli antaa mahdollisuuden muokata oliota dynaamisesti ajon aikana. Se on joutavampaa kuin periminen, koska olio voidaan kasata monista pienemmistä olioista ja eri kombinaatioiden määrät kasvavat eksponentiaalisesti kuorrutteiden määrän mukaan. Se muistuttaa rakenteellisesti hiukan koostemallia. Kuorruttajassa ei kuitenkaan voida muodostaa ”puuta” toisin kuin koosteessa.



Kuva 8. Kuorruttajan rakenne. (9.)

ConcreteComponenttiin voidaan lisätä ConcreteDecoratorA ja/tai ConcreteDecoratorB. Molemmat näistä sisältävät oman toiminnallisuutensa ja kun ne lisätään ConcreteComponenttiin, myös tämä saa sen saman toiminnallisuuden.

Esimerkissä Robotti Ruttuiselle annetaan tulipallohyökkäys ja murskaushyökkäys kuorruttajamallia käyttäen. Ruttusen luominen saattaa näyttää hiukan nurinkuriselta, koska Tulipallohyökkäyksen sisään menee Murskaushyökkäys ja sen sisään Perusrobotti. Tulipallohyökkäys siis sisältää Murskauksen ja Perusrobotin. Näin kuorruttajamallissa kuitenkin toimitaan ja peruskantaolio on aina sisimpänä.

```
using System;
using System.Collections.Generic;

namespace Kuorruttaja
{
    class Program
    {
        static void Main(string[] args)
        {
            Robotti RobottiRuttunen = new TulipalloHyökkäys(new
MurskausHyökkäys(new PerusRobotti()));

            RobottiRuttunen.Hyökkäys();
            Console.Read();
        }
    }

    abstract class Robotti
    {
        public abstract void Hyökkäys();
    }

    class PerusRobotti : Robotti
    {
        public override void Hyökkäys()
        {
            Console.WriteLine("Perushyökkäys!");
        }
    }

    abstract class RobottiKuorruttaja : Robotti
    {
        protected Robotti robotti;

        public RobottiKuorruttaja(Robotti robotti)
        {
            this.robotti = robotti;
        }

        public override void Hyökkäys()
        {
            if (robotti != null)
            {
                robotti.Hyökkäys();
            }
        }
    }
```

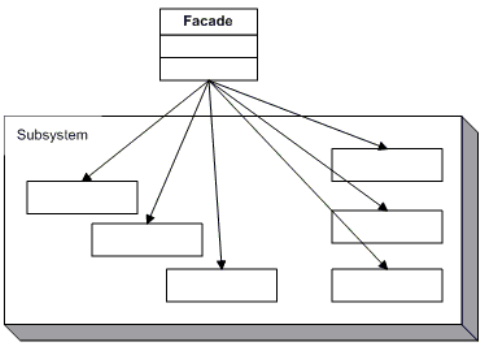
```
    class TulipalloHyökkäys : RobottiKuorruttaja
    {
        public TulipalloHyökkäys(Robotti robotti)
            : base(robotti)
        {
        }


        public override void Hyökkäys()
        {
            base.Hyökkäys();
            Console.WriteLine("Tulipallohyökkäys!");
        }
    }

    class MurskausHyökkäys : RobottiKuorruttaja
    {
        public MurskausHyökkäys(Robotti robotti)
            : base(robotti)
        {
        }

        public override void Hyökkäys()
        {
            base.Hyökkäys();
            Console.WriteLine("Murskaushyökkäys!");
        }
    }
}
```

(7.)

4.5 Julkisivu (Facade)

Julkisivussa kaikki toiminnallisuus laitetaan yhden luokan taakse. Tämä yksi luokka hallinnoi muita luokkia.

Julkisivua käytetään, kun halutaan luoda yksinkertainen käyttöönliittymä, joka suorittaa toimintoja taustalla. Julkisuvu-luokka sisältää muita luokkia, jotka suorittavat tarvittavia operaatioita.



Kuva 8. Julkisivun rakenne. (9.)

Yksinkertaisesti Julkisivu-luokka vain sisältää joukon muita luokkia ja metodeita.

Susi-luokka toimii esimerkissä julkisivuna. Susi-luokka sisältää Liikkuu-luokan ja Syö-luokan ja ne hoitavat nimensä mukaiset toiminnot.

```
using System;
using System.Collections.Generic;

namespace Julkisivu
{
    class Program
    {
        static void Main(string[] args)
        {
            Eläin Susi = new Eläin("Susi");

            Susi.Liikkuu();
            Susi.Syö();

            Console.Read();
        }
    }


    class Eläin
    {
        private string nimi;
        private Syö syö;
        private Liikkuu liikkuu;

        public Eläin(string nimi)
        {
            this.nimi = nimi;
            syö = new Syö(nimi);
            liikkuu = new Liikkuu(nimi);
        }

        public void Syö()
        {
            syö.Toiminto();
        }

        public void Liikkuu()
        {
            liikkuu.Toiminto();
        }
    }

    class Syö
    {
        private string nimi;

        public Syö(string nimi)
        {
            this.nimi = nimi;
        }

        public void Toiminto()
        {
            Console.WriteLine(nimi + " syö!");
        }
```

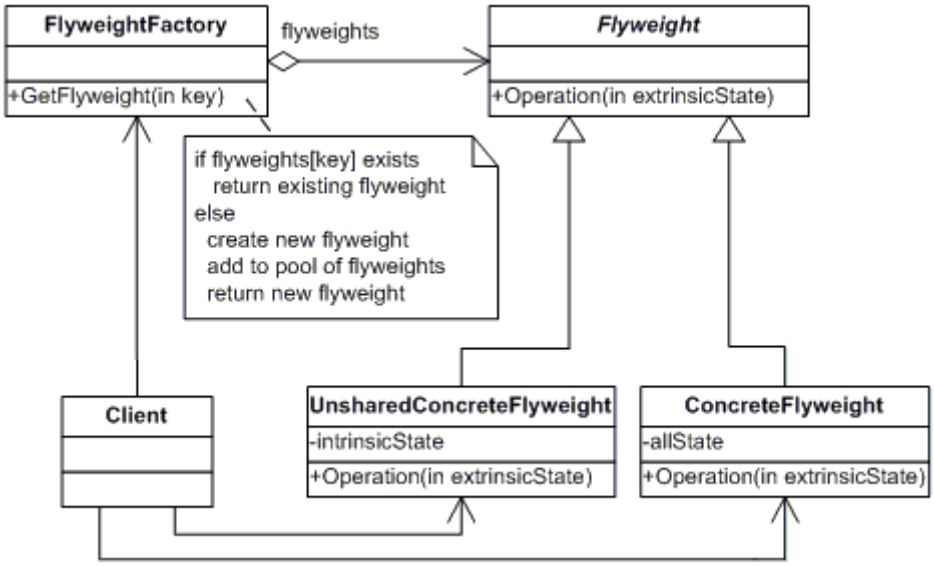
```
    }

    class Liikkuu
    {
        private string nimi;

        public Liikkuu(string nimi)
        {
            this.nimi = nimi;
        }

        public void Toiminto()
        {
            Console.WriteLine(nimi + " liikkuu!");
        }
    }
}
```

(9.)

4.6 Hiutale (Flyweight)

Hiutaleessa pyritään vähentämään olioiden luontiin kulutettuja tehoja.

Hiutalemallissa jaetaan valmiiksi luotuja olioita ja käytetään niitä toisten olioiden luomiseen, jolloin säästyy resursseja. Tämä on erityisen hyödyllistä, jos pitää luoda suuria määriä olijoja, jotka muistuttavat paljon toisiaan.



Kuva 9. Hiutaleen rakenne. (9.)

Client haluaa luoda ConcreteFlyweight-olion ja pyytää, että FlyweightFactory palauttaa vastaavan olion. FlyweightFactoryssä on vastaavanlainen luokka valmiiksi luotuna ja tätä luokkaa ”kopioimalla” uusi luokka luodaan. Koska luokka on jo valmiiksi luo-

tuna, niin säästyy resursseja. UnsharedConcreteFlyweight kuvaa sitä, että hiutaleessa voi myös olla aliluokkia, joita ei jaeta.

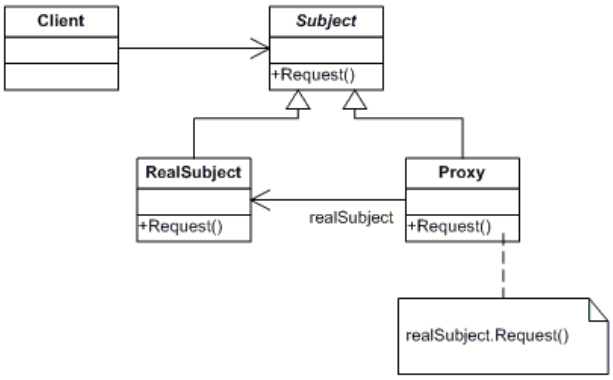
Esimerkissä luodaan ötököitä Ötökkätehtaan avulla. Kun halutaan käyttää ötökkä-oliota, pyydetään tehdasta palauttamaan kyseinen olio, joka on valmiiksi luotuna tehtaan sisällä. Esimerkki on todella yksinkertaistettu, eikä se sisällä abstrakteja luokkia, mutta siitä pitäisi selvitä hiutaleen ydin idea.

```
using System;
using System.Collections;

namespace Kärpässarjalainen
{
    class Program
    {
        static void Main(string[] args)
        {
            Ötökkätehdas tehdas = new Ötökkätehdas();
            Ötökkä ötökkä = tehdas.PalautaÖtökkä();

            ötökkä.Pörisee();

            Console.Read();
        }
    }

    class Ötökkätehdas
    {
        private Ötökkä ötökkä;

        public Ötökkätehdas()
        {
            ötökkä = new Ötökkä();
        }

        public Ötökkä PalautaÖtökkä()
        {
            return ötökkä;
        }
    }

    class Ötökkä
    {
        public void Pörisee()
        {
            Console.WriteLine("Pörr pörr!");
        }
    }
}
```

(9.)

4.7 Edustaja (Proxy, Surrogate)

Edustajaa käytetään vahtina luokkien välillä. Ainoastaan tämä vahti saa käsitellä vahdittavaa luokkaa.



Kuva 10. Edustajan rakenne. (9.)

Proxy ja RealSubject perivät Subject luokan. Proxy toimii RealSubjectin edustajana. Kun pyyntö Request() lähetetään Proxylle, niin Proxy kutsuu RealSubjectin Request()-metodia.

Aarre ja Vartija perivät Aarreluolan. Aarteeseen pääsee käsiksi ainoastaan Vartijan välityksellä.

```
using System;
using System.Collections.Generic;

namespace Edustaja
{
    class Program
    {
        static void Main(string[] args)
        {
            Aarreluola vartija = new Vartija();

            vartija.HaeAarre();

            Console.Read();
        }
    }

    abstract class Aarreluola
    {
        public abstract void HaeAarre();
    }

    class Aarre : Aarreluola
    {
        public override void HaeAarre()
        {
           Console.WriteLine("Aarre noudettu!");
```

```
        }
    }

    class Vartija : Aarreluola
    {
        private Aarre aarre;

        public override void HaeAarre()
        {
            if (aarre == null)
            {
                aarre = new Aarre();
            }

            aarre.HaeAarre();
        }
    }
}
```

(9.)

4.8 Rakennemallien yhteenveto

Sovitinmallia käytetään, kun halutaan yhdistää luokan toiminnallisuus toisen luokan toiminnallisuudeksi.

Siltamallia käytetään, kun tarvitaan luokalle tai luokille useampia metodirakenteita.

Koostemallia käytetään, kun olioista pitää saada muodostettua puurakenne.

Kuorruttajamallia käytetään, kun halutaan yhdistellä eri luokkien ominaisuuksia yhteen kantaluokkaan.

Julkisivumallia käytetään, kun halutaan luoda luokka, joka hoitaa hallinnoi muiden luokkien toimintaa.

Hiutalemallia käytetään, kun pitää luoda isoja määriä samantyyppisiä olioita ja halutaan vähentää käytettyjä resursseja.

Edustajaa käytetään, kun halutaan edustaja tai vahti, joka hallinnoi toiseen rinnakkaisluokkaan kohdistuvia pyyntöjä.

Yllä olevien esimerkkien tietämisestä pitäisi saada hyvät eväät ohjelmien luokkarakenteiden suunnittelulle ja miettimiselle.

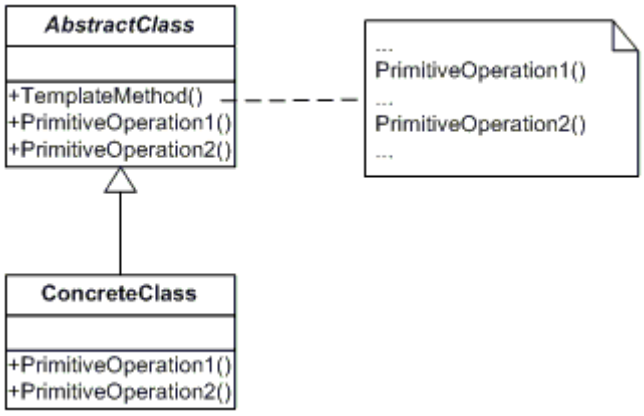
5 KÄYTTÄYTYMISMALLIT

Käyttäytymismalleissa keskitytään siihen, miten oliot voivat kommunikoida keskenään.

5.1 Yleisfunktio (Template Method)

Yleisfunktiossa kantaluokkaan luodaan yleisfunktio, joka kutsuu luokan muita metodeita.

Yleisfunktio on eräs yksinkertaisimmista suunnittelumalleista. Luokan muita metodeita kutsutaan, kun yleisfunktiota kutsutaan.



Kuva 11. Yleisfunktion rakenne. (9.)

TemplateMethodia() kutsuttaessa toteutetaan metodit PrimitiveOperation1() ja PrimitiveOperation2().

Esimerkissä Susi-luokka perii Eläin-luokalta Syö()- ja Juoksee()-metodit. Se perii myös Yleisfunktion(), jota kutsuttaessa toteutetaan Syö()- ja Juoksee()-metodit.

```
using System;
using System.Collections.Generic;

namespace Yleisfunktio
{
    class Program
```

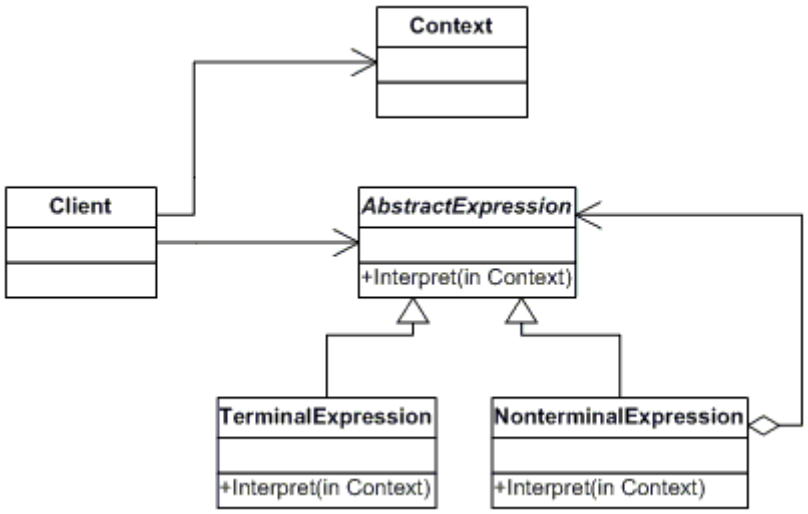
```
    {
        static void Main(string[] args)
        {
            Eläin Susi = new Susi();

            Susi.Yleisfunktio();

            Console.Read();
        }
    }

    abstract class Eläin
    {
        public abstract void Syö();
        public abstract void Juoksee();

        public void Yleisfunktio()
        {
            Syö();
            Juoksee();
        }
    }

    class Susi : Eläin
    {
        public override void Syö()
        {
            Console.WriteLine("Susi syö lihaa");
        }

        public override void Juoksee()
        {
            Console.WriteLine("Susi juoksee");
        }
    }
}
```

(9.)

5.2 Tulkki (Interpreter)

Tulkkia käytetään nimensä mukaisesti tulkkaamaan erilaisia sisältöjä.

Tulkkimallia käytetään useimmiten tulkkaamaan kielellisiä rakenteita, kuten esimerkiksi kielioppia. Tulkkaaminen tapahtuu ”parsimalla”. Parsijoita voi olla useampia ja ne voivat tulkata eri osia sisällöstä. Yksinkertaistettuna tulkkiin menee sisältö ”x” ja tulkin läpi mentyään se on tulkattu sisällöksi ”y”.



Kuva 12. Tulkin rakenne. (9.)

Context eli sisältö lähetetään TerminalExpressionille tulkattavaksi.

Esimerkki on tällä kertaa harvinaisen tynkä. Tulkin idea on kuitenkin yksinkertaistettuna sellainen, että Tulkkaajalle lähetetään sisältöä ja se tulkataan. Tässä tapauksessa tulkattava on luokkana, mutta aika usein se on tekstimuodossa.

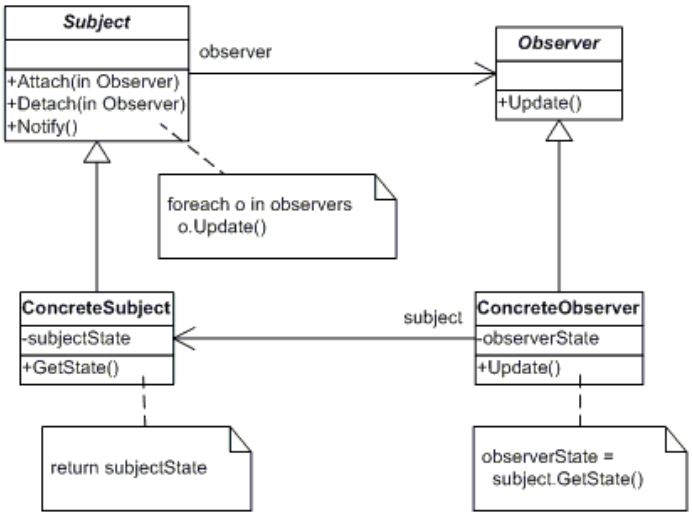
```
using System;
using System.Collections.Generic;

namespace Tulkki
{
    class Program
    {
        static void Main(string[] args)
        {
            Tulkattava tulkattava = new Tulkattava();
            Tulkkaaja tulkkaaja = new Tulkkaaja();

            tulkkaaja.Tulkkaa(tulkattava);

        }
    }

    class Tulkattava
    {
    }

    abstract class Tulkki
    {
        public abstract void Tulkkaa(Tulkattava tulkattava);
    }

    class Tulkkaaja : Tulkki
    {
        public override void Tulkkaa(Tulkattava tulkkaa)
        {
            Console.WriteLine("Tulkkaa Tulkattavan sisällön");
        }
```

```
    }
}
```
(9.)

5.3 Tarkkailija (Observer)

Tarkkailija tarkkailee jonkun toisen olion tilaa ja kun tämän olion tila muuttuu, niin tarkkailija toimii tilanteen mukaan.

Tarkkailijaa käytetään, kun halutaan muuttaa useiden olioiden tilaa, kun toinen olio muuttaa tilaansa.



Kuva 13. Tarkkailijan rakenne. (9.)

Subjectiin tulee voida lisätä ja poistaa tarkkailijoita. Ne Observerit, jotka on lisätty Subjectin listaan tarkkailevat Subjectin tilaa ja toimivat tilanteen mukaan.

Esimerkissä peura- ja jänis-oliot tarkkailevat susi-oliota. Jos susi on nälkäinen, niin ne juoksevat pakoon. Jos susi ei ole nälkäinen, ne syövät ruohoa.

```
using System;
using System.Collections.Generic;

namespace Tarkkailija
{
    class Program
    {
        static void Main(string[] args)
        {
            Susi susi = new Susi();
            Peura peura = new Peura();
            Jänis jänis = new Jänis();
```

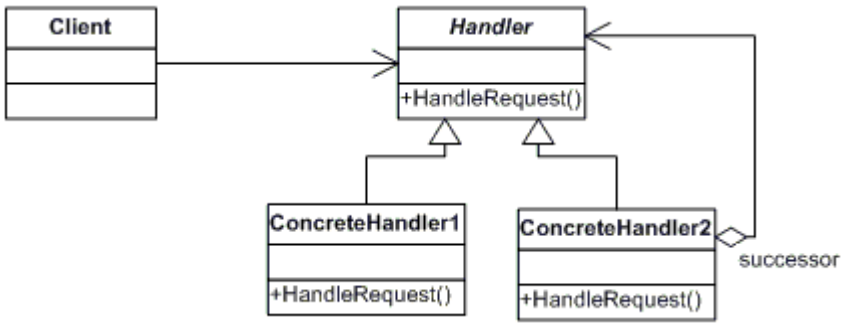
```
            susi.Lisää(peura);
            susi.Lisää(jänis);

            susi.HälytäTarkkailijat();

            susi.asetaNälkä(true);

            susi.HälytäTarkkailijat();

            Console.Read();
        }
    }

    abstract class Lihansyöjä
    {
        private List<Tarkkailija> tarkkailijat = new List<Tarkkailija>();
        private bool Nälkä = false;

        public void Lisää(Tarkkailija tarkkailija)
            {
                tarkkailijat.Add(tarkkailija);
            }
        public void Poista(Tarkkailija tarkkailija)
            {
                tarkkailijat.Remove(tarkkailija);
            }
        public void asetaNälkä(bool nälkä)
            {
                Nälkä = nälkä;
            }

        public void HälytäTarkkailijat()
        {
            foreach (Tarkkailija tarkkailija in tarkkailijat)
            {
                tarkkailija.Hälytä(Nälkä);
            }
        }
    }

    class Susi : Lihansyöjä
    {
    }

    abstract class Tarkkailija
    {
        public abstract void Hälytä(bool juokseKarkuun);
    }

    class Peura : Tarkkailija
    {
        public override void Hälytä(bool juokseKarkuun)
        {
            if (juokseKarkuun == true)
            {
                Console.WriteLine(this.GetType().Name + " juoksee karkuun!");
            }
            else
            {
                Console.WriteLine(this.GetType().Name + " mussuttaa ruohoa!");
            }
        }
    }

    class Jänis : Tarkkailija
```

```
    {
        public override void Hälytä(bool juokseKarkuun)
        {
            if (juokseKarkuun == true)
            {
                Console.WriteLine(this.GetType().Name + " loikkii karkuun!");
            }
            else
            {
                Console.WriteLine(this.GetType().Name + " mussuttaa ruohoa!");
            }
        }
    }
}
```

(9.)

5.4 Vastuuketju (Chain of Responsibility)

Vastuuketjussa ketjutetaan olioita peräkkäin. Jos ensimmäinen olio ei pysty käsittelemään jotakin pyyntöä, niin se siirretään seuraavalle niin kauan kunnes pyyntö käsitellään tai oliot loppuvat.



Kuva 14. Vastuuketjun rakenne. (9.)

ConcreteHandler1 käsittelee pyyntöä ensin, jonka jälkeen se voi halutessaan lähettää pyynnön käsiteltäväksi ConcreteHandler2:lle, jos se itse ei ole pystynyt käsittelemään pyyntöä.

Esimerkissä käsitellään Eläin-olioita. Vastuuketjun ensimmäisenä lenkkinä on susi-olio. Susi-oliolle lähetetään ruuaksi ruohoa, mutta sudet eivät tunnetusti syö ruhoa ja susi lähettää pyynnön edelleen peuralle, joka syö ruohon.

```
using System;
using System.Collections.Generic;

namespace Vastuuketju
{
    class Program
    {
```

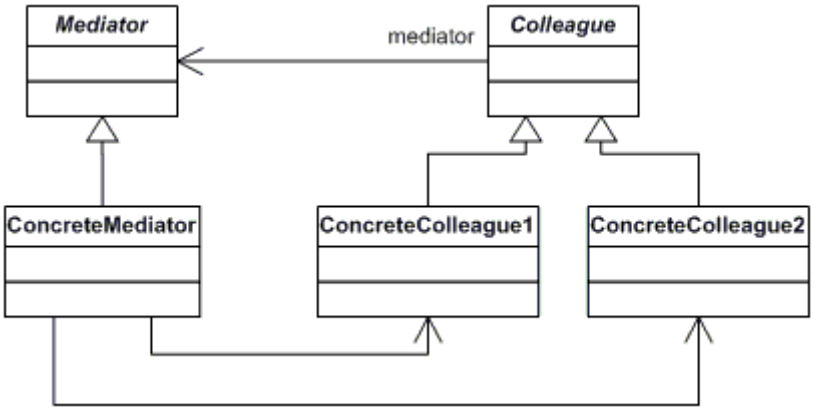
```
        static void Main(string[] args)
        {
            Susi susi = new Susi();
            Peura peura = new Peura();

            susi.asetaPerijä(peura);
            susi.Syö("ruohoa");

            Console.Read();
        }
    }

    abstract class Vastuuketju
    {
        protected Vastuuketju vastuuketju;

        public void asetaPerijä(Vastuuketju vastuuketju)
        {
            this.vastuuketju = vastuuketju;
        }

        public abstract void Syö(string ruoka);
    }

    class Susi : Vastuuketju
    {
        public override void Syö(string ruoka)
        {
            if (ruoka == "lihaa")
            {
                Console.WriteLine("Susi syö " + ruoka);
            }
            else if (vastuuketju != null)
            {
                Console.WriteLine("Susi ei syö "+ ruoka);
                vastuuketju.Syö(ruoka);
            }
        }
    }

    class Peura : Vastuuketju
    {
        public override void Syö(string ruoka)
        {
            if (ruoka == "ruohoa")
            {
                Console.WriteLine("Peura syö " + ruoka);
            }
            else if (vastuuketju != null)
            {
                Console.WriteLine("Peura ei syö " + ruoka);
                vastuuketju.Syö(ruoka);
            }
        }
    }
}
```

(9.)

5.5 Välittäjä (Mediator)

Välittäjä toimii linkkinä kahden olion välillä.

Välittäjämallissa kommunikoivien olioiden ei tarvitse tietää mitään toisistaan vaan välittäjä-olio hoitaa kommunikaation näiden välillä.



Kuva 15. Välittäjän rakenne. (9.)

ConcreteMediatoriin syötetään kollegat ConcreteColleague1 ja ConcreteColleague2, jonka jälkeen se hoitaa kommunikaatiota niiden välillä.

Esimerkissä luodaan metsä-olio, joka toimii välittäjänä orava-olion ja jänis-olion välillä. Kun orava lähettää viestin ”Moi!”, niin metsä-olio välittää viestin edelleen peuralle ja päinvastoin.

```
using System;
using System.Collections.Generic;

namespace Välittäjä
{
    class Program
    {
        static void Main(string[] args)
        {
            Metsä metsä = new Metsä();
            Orava orava = new Orava(metsä);
            Jänis jänis = new Jänis(metsä);

            metsä.asetaJänis(jänis);
            metsä.asetaOrava(orava);

            orava.LähetäViesti("Moi!");
            jänis.LähetäViesti("No moi!");

            Console.Read();
        }
    }

    abstract class Välittäjä
    {
        public abstract void Juttele(string viesti,Toverit toveri);
    }

    class Metsä : Välittäjä
```

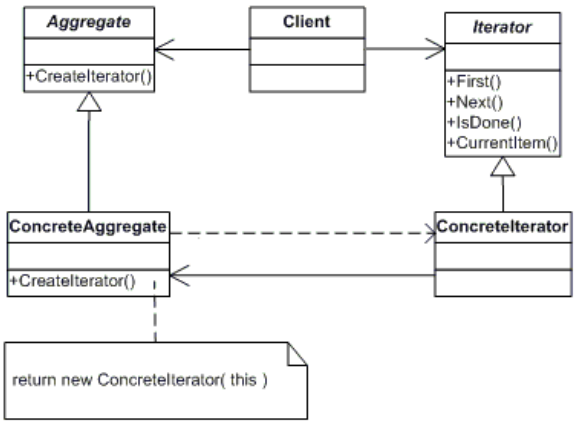
```
    {
        private Jänis jänis;
        private Orava orava;

        public void asetaJänis(Jänis jänis)
        {
            this.jänis = jänis;
        }

        public void asetaOrava(Orava orava)
        {
            this.orava = orava;
        }

        public override void Juttele(string viesti, Toverit toveri)
        {
            if (toveri == orava)
            {
                jänis.VastaanotaViesti(viesti);
            }
            else
            {
                orava.VastaanotaViesti(viesti);
            }
        }
    }
    abstract class Toverit
    {
        protected Välittäjä välittäjä;

        public Toverit(Välittäjä välittäjä)
        {
        this.välittäjä = välittäjä;
        }

        public void LähetäViesti(string viesti)
        {
            Console.WriteLine(GetType().Name + " lähettää viestin: "+ viesti);
            välittäjä.Juttele(viesti, this);
        }

        public void VastaanotaViesti(string viesti)
        {
            Console.WriteLine( GetType().Name + " vastaanottaa viestin: "
              + viesti);
        }
    }

    class Jänis : Toverit
    {
        public Jänis(Välittäjä välittäjä)
            : base(välittäjä)
        {
        }
    }

    class Orava : Toverit
    {
        public Orava(Välittäjä välittäjä)
            : base(välittäjä)
        {
        }
    }
}
```

(9.)

5.6 Iteraatio (Iterator)

Iteraatiota käytetään oliokokoelmien käsittelyyn.

Iteraatiossa ei ole väliä, onko kyseessä lista, taulukko vai pino. Se käsittelee niitä kaikkia samalla tavalla. Tämä on kätevää, jos ohjelma sisältää useita erilaisia kokonaisuuksia, joita pitää saada käsiteltyä.



Kuva 17. Iteraation rakenne. (9.)

Aggregate eli kooste lähetetään Iteratorille, jossa sitä voidaan käsitellä.

Esimerkissä luodaan postimerkki- ja perhoskoosteet, joista luodaan iteraatiot. Näitä iteraatioita hyväksikäyttämällä oliolistat voidaan kätevästi käydä läpi.

```
using System;
using System.Collections;

namespace Iteraatio
{
    class Program
    {
        static void Main(string[] args)
        {
            Kooste postimerkit = new Postimerkit();
            postimerkit[0] = "Susimerkki";
            postimerkit[1] = "Peuramerkki";
            postimerkit[2] = "Kotkamerkki";
            postimerkit[3] = "Mäyrämerkki";

            Kooste perhoset = new Perhoset();
            perhoset[0] = "Nokkosperhonen";
            perhoset[1] = "Kaunisperhonen";
            perhoset[2] = "Ritariperhonen";

            Iteraattori iteraatio = postimerkit.LuoIteraatio();
            Iteraattori iteraatio2 = perhoset.LuoIteraatio();
```

```
            object postimerkki = iteraatio.Ensimmäinen();
            object perhonen = iteraatio2.Ensimmäinen();

            while (postimerkki != null)
            {
                Console.WriteLine(postimerkki);
                postimerkki = iteraatio.Seuraava();
            }
            Console.WriteLine();
            while (perhonen != null)
            {
                Console.WriteLine(perhonen);
                perhonen = iteraatio2.Seuraava();
            }

            Console.Read();
        }
    }

    abstract class Kooste
    {
        protected ArrayList kooste = new ArrayList();
        public abstract Iteraattori LuoIteraatio();

        public int Määrä
        {
            get { return kooste.Count; }
        }

        public object this[int indeksi]
        {
            get { return kooste[indeksi]; }
            set { kooste.Insert(indeksi, value); }
        }
    }

    class Postimerkit : Kooste
    {
        public override Iteraattori LuoIteraatio()
        {
            return new Iteraattori(this);
        }
    }

    class Perhoset : Kooste
    {
        public override Iteraattori LuoIteraatio()
        {
            return new Iteraattori(this);
        }
    }

    abstract class Iteraattorit
    {
        public abstract object Ensimmäinen();
        public abstract object Seuraava();
        public abstract bool OnkoValmis();
        public abstract object SisältöNyt();
    }

    class Iteraattori : Iteraattorit
    {
        private Kooste kooste;
        private int indeksi;
```

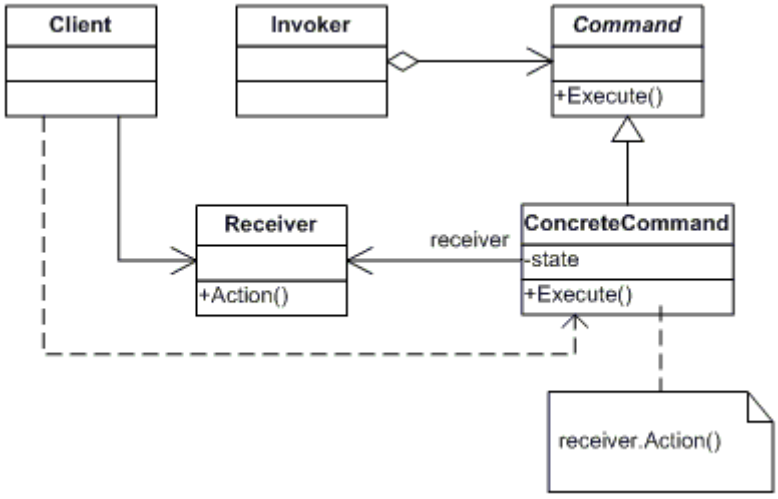
```
        public Iteraattori(Kooste kooste)
        {
            this.kooste = kooste;
        }

        public override object Ensimmäinen()
        {
            return kooste[0];
        }

        public override object Seuraava()
        {
            object sisältö = null;
            if (indeksi < kooste.Määrä - 1)
            {
                sisältö = kooste[++indeksi];
            }
            return sisältö;
        }

        public override bool OnkoValmis()
        {
            return indeksi >= kooste.Määrä;
        }

        public override object SisältöNyt()
        {
            return kooste[indeksi];
        }
    }
}
```

(9.)

5.7 Komento (Command)

Komentomallissa luokat käskyttävät toisiaan lähes armeijamaisesti. Käsky lähtee ylhäältä hierarkiaa pitkin alas asti.

Komentomalli voi olla kätevä, kun yhden asian toteuttaminen vaatii useampien palojen osallistumista toimintaan ja nämä palaset on pitänyt erotella toisistaan.



Kuva 18. Komennon rakenne. (9.)

Invoker käskyttää ConcreteCommandia ja ConcreteCommand käskee Receiveriä toimimaan. Tämä muistuttaa rakenteellisesti ehkä hiukan vastuuketjua.

Koska itselläni tuli ensimmäiseksi mieleen armeija, käytin kyseistä teemaa esimerkissä. Kenraali käskyttää luutnanttia ja luutnantti käskee alokkaan tehdä työn.

```
using System;
using System.Collections.Generic;

namespace Komento
{
    class Program
    {
        static void Main(string[] args)
        {
            Alokas alokas = new Alokas();
            Luutnantti luutnantti = new Luutnantti(alokas);
            Kenraali kenraali = new Kenraali(luutnantti);

            kenraali.Komenna();

            Console.Read();
        }
    }

    class Kenraali
    {
        protected Komentaja komentaja;

        public Kenraali(Komentaja komentaja)
        {
            this.komentaja = komentaja;
        }

        public void Komenna()
        {
            Console.WriteLine("Kenraali käskyttää Luutnanttia!");
            komentaja.Käskytä();
        }
```

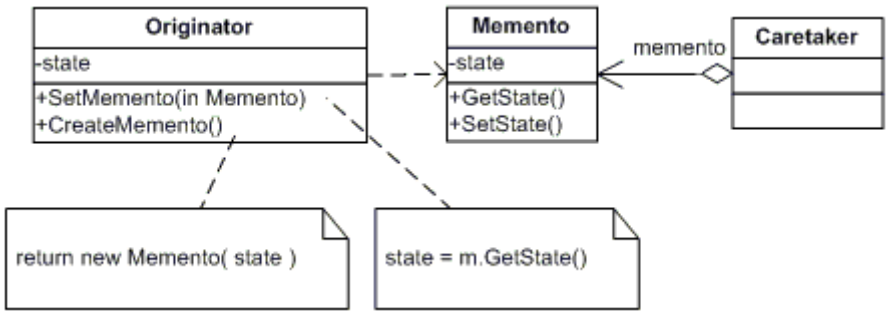
```
    }

    abstract class Komentaja
    {
        protected Alokas alokas;

        public Komentaja(Alokas alokas)
        {
            this.alokas = alokas;
        }

        public abstract void Käskytä();
    }

    class Luutnantti : Komentaja
    {
        public Luutnantti(Alokas alokas)
            : base(alokas) { }

        public override void Käskytä()
        {
            Console.WriteLine("Luutnantti käskyttää alokasta!");
            alokas.Toimii();
        }

    }

    class Alokas
    {
        public void Toimii()
        {
            Console.WriteLine("Alokas tekee nohevasti käskyn mukaan!");
        }
    }
}
```

(9.)

5.8 Muisto (Memento)

Muistomallissa säilötään olion tila muisto-olioon ja tämä tila voidaan halutessa palauttaa tai käyttää muihin tarkoituksiin.

Joskus ohjelmoinnissa tarvitsee palauttaa olion tila siihen, mitä se oli aikaisemmin. Jos tällainen tilanne tulee, niin muistomalli on erinomainen väline kyseisen toimenpiteen suorittamiseen.



Kuva 19. Muiston rakenne. (9.)

Caretaker huolehtii Mementojen eli muistojen organisoimisesta. Originator-olio palauttaa tiedot tilastaan, jonka jälkeen huolehtija-olio tallentaa kyseisen tilan muisto-olioon.

Esimerkissä säilötään numero-olion tilaa. Huolehtija-olio huolehtii säilyttämisestä ja tallentaa numero-olion tilan muisto-olioon.

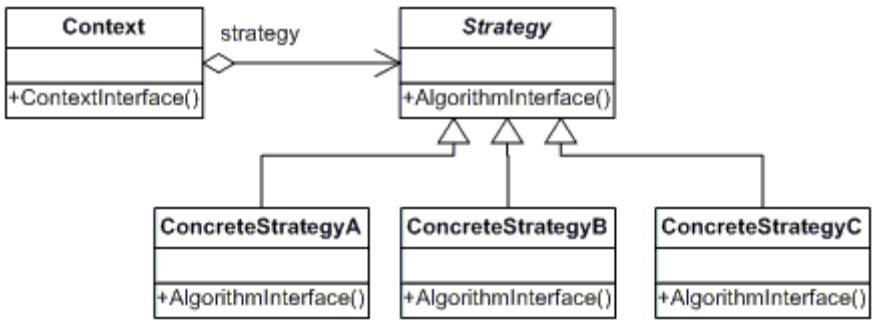
```
using System;
using System.Collections;

namespace Muisto
{
    class Program
    {
        static void Main(string[] args)
        {
            Numero numero = new Numero();
            Huolehtija huolehtija = new Huolehtija();

            numero.asetaNumero(42);
            huolehtija.asetaMuisto(numero.palautaMuisto());
            numero.asetaNumero(1);
            huolehtija.asetaMuisto(numero.palautaMuisto());
            numero.asetaNumero(28);
            huolehtija.asetaMuisto(numero.palautaMuisto());

            huolehtija.tulostaMuistot();
            Console.Read();
        }
    }

    class Muisto
    {
        private int numero;

        public Muisto(int numero)
        {
            this.numero = numero;
        }

        public int palautaNumero()
        {
            return numero;
        }
    }
```

```
    class Numero
    {
        private int numero;

        public void asetaNumero(int numero)
        {
            this.numero = numero;
        }

        public Muisto palautaMuisto()
        {
            return new Muisto(numero);
        }
    }

    class Huolehtija
    {
        ArrayList numeroLista = new ArrayList();

        public void asetaMuisto(Muisto muisto)
        {
            numeroLista.Add(muisto);
        }

        public ArrayList palautaMuistot()
        {
            return numeroLista;
        }

        public void tulostaMuistot()
        {
            foreach (Muisto muisto in numeroLista)
            {
                Console.WriteLine(muisto.palautaNumero());
            }
        }
    }
}
```

(9.)

5.9 Strategia (Strategy)

Strategiamallissa olio sisältää nimensä mukaisesti useita eri strategioita. Näistä strategioista valitaan tilanteen mukaan käyttöön yksi ja käytetään sitä haluttujen toimintojen suorittamiseen.



Kuva 20. Strategian rakenne. (9.)

Strategia-luokka sisältää strategiat A, B ja C. Contex-luokka, joka sisältää Strategy-olion valitsee, mitä näistä strategioista tulee käyttää ja suorittaa valitun strategian mukaiset toiminnot.

Esimerkissä susi-oliolle asetetaan strategia tilanteen mukaan. Kun susi aistii vaaran, strategia on pakoonjuokseminen ja kun susi näkee saaliin, se hyökkää sen kimppuun.

```
using System;
using System.Collections.Generic;

namespace Strategia
{
    class Program
    {
        static void Main(string[] args)
        {
            Susi susi = new Susi();

            susi.aistiiVaaran();
            susi.näkeeSaaliin();

            Console.Read();
        }
    }

    abstract class Strategia
    {
        public abstract void Toiminto();
    }

    class Metsästys : Strategia
    {
        public override void Toiminto()
        {
            Console.WriteLine("Susi hyökkää saaliin kimppuun!");
        }
    }

    class Pako : Strategia
    {
        public override void Toiminto()
        {
            Console.WriteLine("Susi juoksee karkuun!");
        }
```

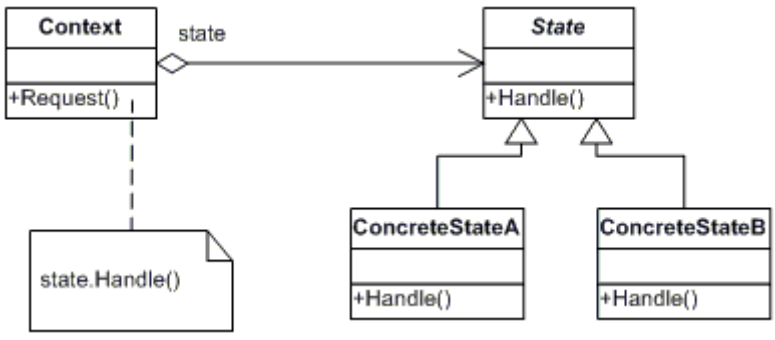
```
    }

    class Susi
    {
        private Strategia strategia;

        public void asetaStrategia(Strategia strategia)
        {
            this.strategia = strategia;
        }

        public void toteutaSuunnitelma()
        {
            if (strategia != null)
            {
                strategia.Toiminto();
            }
        }

        public void näkeeSaaliin()
        {
            Console.WriteLine("Susi näkee poron!");
            asetaStrategia(new Metsästys());
            toteutaSuunnitelma();
        }

        public void aistiiVaaran()
        {
            Console.WriteLine("Hullu metsästäjä lähestyy!");
            asetaStrategia(new Pako());
            toteutaSuunnitelma();
        }
    }
}
```

(9.)

5.10 Tila (State)

Tilassa toteutetaan erilaisia tiloja. Mitä tilaa kulloinkin toteutetaan, riippuu tilamallin sen hetkisestä tilasta.

Jos oliolle tarvitaan useita erilaisia toimintamalleja, niin tilamalli on siihen hyödyllinen työkalu.



Kuva 21. Tilan rakenne. (9.)

Tila muistuttaa rakenteellisesti hyvin paljon vastuuketjua. Ainoa ero näiden kahden välillä on, että tilassa ei siirrytä tilojen välillä ketjutetusti ja järjestyksessä, kuten vastuuketjussa.

Context-luokka sisältää state-olion, jota kutsutaan. Ensimmäisellä kerralla state-olio toteuttaa ConcreteStateA:n ja toisella kerralla ConcreteStateB:n.

Esimerkissä susi-oliolla on kolme erilaista tilaa: Aamu, päivä ja yö. Kun susi toimii ensimmäisen kerran, on aamu ja se herää. Kun susi toimii toisen kerran, siirrytään toiseen tilaan eli päivään, jolloin metsästetään. Yö-tilassa ollessaan susi nukkuu.

```
using System;
using System.Collections.Generic;

namespace Tila
{
    class Program
    {
        static void Main(string[] args)
        {
            Susi susi = new Susi(new Aamu());

            susi.Toimi();
            susi.Toimi();
            susi.Toimi();
            susi.Toimi();
            susi.Toimi();

            Console.Read();
        }
    }

    abstract class Tila
    {
        public abstract void Toiminto(Susi susi);
    }

    class Aamu : Tila
    {
        public override void Toiminto(Susi susi)
        {
             Console.WriteLine("Susi herää");
            susi.Tila = new Päivä();
        }
    }

    class Päivä : Tila
    {
        public override void Toiminto(Susi susi)
        {
                    Console.WriteLine("Susi metsästää");
            susi.Tila = new Yö();
        }
    }

    class Yö : Tila
```

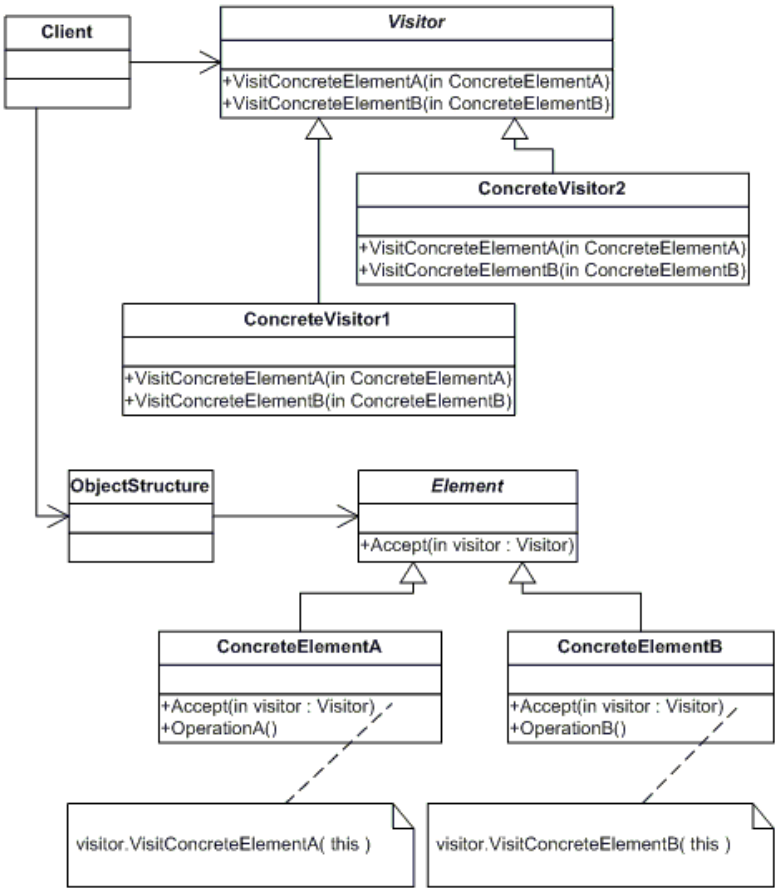
```
    {
        public override void Toiminto(Susi susi)
        {
                    Console.WriteLine("Susi nukkuu");
            susi.Tila = new Aamu();
        }
    }

    class Susi
    {
        private Tila tila;

        public Susi(Tila tila)
        {
            this.tila = tila;
        }

        public Tila Tila
        {
            get { return tila; }
            set { tila = value; }
        }

        public void Toimi()
        {
            tila.Toiminto(this);
        }
    }
}
```

(9.)

5.11 Vierailija (Visitor)

Vierailijamallissa luodaan luokkarakenne, jossa vierailija-olio käy toisten olioiden sisällä suorittamassa toimintoja.

Monet ovat sitä mieltä, että se on turhan monimutkainen, enkä ihmettele, kun katsoo millainen rakennehirviö alapuolella on. Muihin malleihin verrattuna se on monimutkainen ja samat asiat voidaan useimmissa tapauksissa tehdä huomattavasti yksinkertaisemmin ja vaivattomammin, joten suosittelen suhtautumaan vierailijamalliin suurin varauksin.



Kuva 22. Vierailijan rakenne. (9.)

Muihin malleihin verrattuna vierailijan rakenne näyttää aluksi hurjalta, mutta loppujen lopuksi tämäkin malli on varsin yksinkertainen. ConcreteElementA kutsuu vierailijaa ConcreteVisitor1 lähettäen tälle oman tilansa. ConcreteVisitor1 ottaa tilan vastaan ja käyttää sitä toimintojen suorittamiseen.

Äkkiseltään vierailijamalli voi vaikuttaa nurinkuriselta, koska Visitorit eli vierailijat eivät ole todellisuudessa niitä, jotka vierailevat vaan niiden luona vieraillaan. Älä anna tämän hämätä.

Esimerkissä Esa-olio hyväksyy vierailuita kierteleviltä partureilta ja kaupustelijoilta. Kun Esa hyväksyy Kauppiaan vierailun, Esa-olio lähettää tietonsa kauppiaalle ja kauppias antaa Esalle ruokaa.

```
using System;
using System.Collections.Generic;

namespace Vierailija
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            Vierailija Parturi = new Parturi();
            Vierailija Kauppias = new Kauppias();
            Ihminen Esa = new Ihminen("Esa");

            Esa.HyväksyVierailu(Kauppias);
            Esa.HyväksyVierailu(Parturi);

            Console.Read();
        }
    }

    abstract class Vierailija
    {
        public abstract void tapaaIhminen(Ihminen ihminen);
    }

    class Parturi : Vierailija
    {
        public override void tapaaIhminen(Ihminen ihminen)
        {
            Console.WriteLine("Parturi leikkaa " + ihminen.palautaNimi() + "n
hiukset!");
        }
    }

    class Kauppias : Vierailija
    {
        public override void tapaaIhminen(Ihminen ihminen)
        {
            Console.WriteLine("Kauppias antaa " + ihminen.palautaNimi() + "lle
ruokaa!");
        }
    }

    abstract class Elementti
    {
        public abstract void HyväksyVierailu(Vierailija vierailija);
    }

    class Ihminen : Elementti
    {
        private string nimi;

        public Ihminen(string nimi)
        {
            this.nimi = nimi;
        }

        public override void HyväksyVierailu(Vierailija vierailija)
        {
            vierailija.tapaaIhminen(this);
        }

        public string palautaNimi()
        {
            return nimi;
        }
    }
}
```

(9.)

5.12 Käyttäytymismallien yhteenveto

Yleisfunktiota käytetään, kun halutaan luoda metodirakenteet valmiiksi aliluokille ja tarvitaan yleisfunktiota, joka toteuttaa useampia luokan metodeista.

Tulkkia käytetään, kun halutaan tulkata tekstiä.

Tarkkailijaa käytetään, kun pitää muuttaa useiden olioiden tilaa yhden olion tilan muuttuessa.

Vastuuketjua käytetään, kun halutaan ketjuttaa olioita keskenään. Jos ensimmäinen olio ei pysty käsittelemään pyyntöä, niin se siirretään eteenpäin jne.

Välittäjää käytetään, kun halutaan välittäjä kahden toisiinsa sopimattoman olion välille.

Iteraatiota käytetään, kun halutaan erilaisten oliokokoelmien käsittelijä. Nämä kokoelmat voivat olla listoja, pinoja tai tauluja.

Komentoa käytetään, kun halutaan, että yhdellä komennolla toteutetaan useita eri metodeita.

Muistoa käytetään, kun halutaan tallettaa olion tilaa ja palauttaa se myöhemmin.

Strategiaa käytetään, kun halutaan luoda useita toimintokokonaisuuksia, joista valitaan tilanteen mukaan yksi.

Tilaa käytetään, kun halutaan olion muuttavan tilaansa joka kerta, kun sitä kutsutaan ja toimivan jokaisessa tilassa eri tavalla.

Vierailijaa käytetään, kun halutaan liittää olioon uusi vieraileva metodi muuttamatta itse oliota.

Ymmärtämällä nämä mallit saat tehtyä ohjelmistasi rakenteellisesti joustavampia.

6 SUUNNITTELUMALLIEN SOVELTAMINEN

Sovelsin suunnittelumalleja käytäntöön ja rakensin viime vuonna tekemäni Pallopelin uusiksi. Nyt peli on rakenteellisesti huomattavasti ehyempi ja rivimäärältään pienempi.

Luokka	Vanha Pallopeli rivimäärä	Uusi/Optimoitu Pallopeli rivimäärä
Pääluokka	905	241
Aarre	65	53
Pelaaja	145	99
Reunavihu	129	106
Vihu	124	107
Pahisneliö	73	48
Handlaaja	0	45
Pallo	0	58
Pallotehdas	0	56
	1441	**813**

Vanhassa versiossa rivimäärä oli 1441 ja uudemmassa 813. Tämä tarkoittaa, että rivimäärä laski yli 40 %. Tietenkään ohjelmoinnissa pelkkä vähäinen rivimäärä ei tarkoita, että koodi olisi hyvää. Suurin ongelma vanhemmassa versiossa oli, että Pääluokka hoiti lähes kaiken. Uudemmassa versiossa jaoin vastuuta muille olio-ohjelmoinnin periaatteiden mukaisesti. Sen lisäksi toteutin huomattavasti järkevämmin muutamia asioita Pääluokassa. Uudemmassa versiossa käytettiin myös vahvasti periytymistä hyväkseni. Periytin Pallo-luokasta kaiken tarvittavan tiedon alaspäin Vihulle, Pelaajalle, Aarteelle, Pahisneliölle ja Reunavihulle. Se vähensi koodin toistamista huomattavasti.

Mitä suunnittelumalleja sitten käytettiin uuden Pallopelin tekemiseen? Ei tarkkaan ottaen yhtään! Opinnäytetyön alkupuolella selitettiinkin, että mitään mallia ei pidä suoraan sellaisenaan ruveta käyttämään, vaan niitä pitää soveltaa ratkaisuja mietittäessä.

Uudessa versiossa oleva Pallotehdas hoitaa kaikkien niiden olioiden luomisen, jotka perivät Pallo-luokan. Tämä muistuttaa ehkä eniten tehdasmetodia.

Sen lisäksi on Handlaaja-luokka. Se muistuttaa luultavasti eniten Julkisivua. Alunperin oli tarkoitus, että se käsittelisi enemmänkin asioita, mutta tällä hetkellä se hoitaa ainoastaan törmäystarkastelut ja palauttaa ”true”, jos törmäys tapahtuu.

Molemmat versiot Pallopelistä löytyvät täältä: http://www.fileswap.com/dl/oZstfjAvXa/. Knaapallo on uudelleen rakennettu versio ja Pallopeli vanha. Molemmat tarvitsevat Visual Studio 2010 ja Microsoftin XNA:n kääntyäkseen. Microsoft lopetti XNA:n kehittämisen ja tukemisen, joten se ei toimi enää Visual Studio 2012:sta eteenpäin.

7 LOPPUYHTEENVETO

Kun valitsin opinnäytetyökseni suunnittelumallit, minulla ei ollut niistä kovin tarkkaa käsitystä. Tiesin, että ne ovat jonkinlaisia ratkaisumalleja ohjelmoinnissa esiintyviin ongelmiin. Aloittaessani suunnittelumallien opettelua kohtasin suuria ongelmia, koska ajattelin niitä liian vaikeasti. Kyseessähän on kuitenkin varsin yksinkertainen asia. Kuvittelin todennäköisesti, että suunnittelumalleissa on jotain taianomaisen nerokkaita ja vaikeasti käsitettäviä ratkaisuja. Koko kuvitelma on typerä, koska suunnittelumalleja ei olisi mahdollista soveltaa ohjelmoinnin ratkaisuihin, jos ne olisivat monimutkaisia. Luontimallit läpikäytyäni pääsin paremmin sisään suunnittelumalleihin ja loppua kohden työ kävi helpommaksi ja helpommaksi. Suunnittelumallit läpikäytyäni ajattelin, että opinnäytetyöstä puuttuu käytäntö kokonaan, joten rakensin pallopelini uudelleen uudella kokemuksella ja työjälki oli tällä kertaa huomattavasti kauniimpaa katsottavaa.

Olen opinnäytetyöni lopputulokseen tyytyväinen, vaikka minulle jäikin kaihertava tunne, että jotain puuttuu. Suunnittelumalleja olisi voinut analysoida niin monella eri tapaa, että materiaalia olisi helposti saanut tuotettua enemmänkin. Toisaalta mallien ylianalysointi olisi ollut vielä typerämpää, joten jätin sen tekemättä. Opin paljon olio-ohjelmoinnista ja ohjelmoinnista yleisesti opinnäytetyötä tehdessäni ja pelkästään sen takia saa ja pitää olla ylpeä.

LÄHTEET

1. **Koskimies, Kai.** *Oliokirja*. 2000. *Helsinki. Talentum oyj. ISBN-13: 9789517627207*

2. *Software Design Patterns. Department of Computer Science. Kent State University.* [Online] [Viitattu 11.3.2013] http://www.sdml.info/collard/se/notes/Software%20Design%20Patterns.ppt

3. **Paul Graham.** *Revenge of the Nerds.* Essee. [Online] 2002. [Viitattu: 11.3.2013.] http://www.paulgraham.com/icad.html

4. Scott Klement. *Anti-Patterns: Avoid the Programming Dark Side.* [Online] 2008. [Viitattu 11.3.2013.] http://www.iprodeveloper.com/article/rpg-programming/anti-patterns-avoid-the-programming-dark-side-67563

5. *Reinventing The Wheel* [Online] 2012. [Viitattu: 11.3.2013] http://c2.com/cgi/wiki?ReinventingTheWheel

6. *Accidental complexity anti-pattern.* [Online] 2007. [Viitattu: 11.3.2013] http://dugrocker.blogspot.fi/2007/01/accidental-complexity-anti-pattern.html

7. *Anti-Pattern: The god object.* [Online] 2009. [Viitattu: 11.3.2013.] http://blog.decayingcode.com/post/anti-pattern-god-object.aspx

8. *Cut-And-Paste Programming.* [Online] [Viitattu: 11.3.2013.] http://sourcemaking.com/antipatterns/cut-and-paste-programming

9. *Dofactory Patterns. .NET Design Patterns.* [Online] 2001. [Viitattu: 6.3.2013.] http://www.dofactory.com/Patterns/Patterns.aspx

10. **Derek Banas.** Suunnittelumallioppaat. [Online] 2012. [viitattu 6.3.2013.] http://www.youtube.com/playlist?list=PLF206E906175C7E07

11. **Mikael Kujanpää.** Suunnittelumallit. *Oulun yliopisto. Tietojenkäsittelytieteiden laitos. Oliosuuntautunut analyysi ja –suunnittelu.* [Online] 2003. [viitattu 6.3.2013.] http://koti.kapsi.fi/~mahead/yo-tekstit/oas-essee.pdf

12. *Design Patterns by Metaphors.* [Online] 2010. [viitattu 6.3.2013.] http://kirangudipudi.blogspot.fi/2010/08/design-patterns-by-metaphors-part-i.html

13. **Steve McConnell.** *Code Complete. 2004. ISBN13: 978-0735619678.*

14. **William Brown.** *AntiPatterns – Refactoring Software, Architectures, and Projects in Crisis.* 1998. *ISBN-10: 0471197130.*