SUUNNITTELUMALLIT

Mirja Immonen TKO4
Erikoistyö
Kuopion yliopisto
Tietojenkäsittelytieteen
ja sov.matem. laitos
24.9.2002

SISÄLLYSLUETTELO

1 JOHDANTO

Erikoistyössäni esittelen yleisimmin käytettyjä suunnittelumalleja, joihin olen myös pyrkinyt löytämään tai toteuttamaan ymmärtämistä helpottavan koodiesimerkin. Kaikista malleista ei ole koodiesimerkkiä, koska joissakin tapauksissa ei voida esittää yksinkertaista esimerkkiä, vaan sellaisen toteuttaminen olisi ollut tähän tarkoitukseen liian monimutkaista. Kaikki erikoistyössäni esittelemäni koodiesimerkit ovat testattuja ja toimivia.

1.1 Yleistä

Suunnittelumallit ovat malleja, jotka kuvaavat suunnitelmatason ratkaisun johonkin toistuvasti esiintyvään ohjelman tai ohjelmiston toteutusongelmaan. Suunnittelumallit ovat komponentin sisällä oliokeskeisissä menetelmissä käytettäviä välineitä. Suunnittelumallit ovat hyvin dokumentoituja, kokeiltuja, toimiviksi todettuja ja yleisesti hyväksyttyjä ratkaisutapoja. Suunnittelumalleja käyttämällä voidaan välttää ohjelmistovirheitä, joita jokainen ohjelmoija tekee. Tavoitteena siis on, että kaikkea ei tarvitse keksiä uudelleen, vaan hyviksi todettuja ratkaisuja saa, ja on suositeltavaakin käyttää. Hyvä suunnittelumalli kuvaa yksittäisten moduulien lisäksi myös niiden välisiä yhteyksiä.

Ohjelmoinnin alkuajoista lähtien on käytössä ollut aliohjelmakirjastoja. Luokkakirjastot tulivat oliokielten mukana ja viime aikoina on alettu puhua sovelluskehyksistä (*frameworks*) ja suunnittelumalleista (*design patterns*). Erityisesti juuri oliolähestymistavassa on helppo toteuttaa suunnittelutapojen ja komponenttien uudelleenkäyttöä. Vaikka suunnittelumalleja käytetäänkin yleensä oliosuuntautuneiden menetelmien yhteydessä, niitä ei ole millään tavalla sidottu oliomenetelmiin, vaan ne soveltuvat käytettäväksi myös muiden menetelmien kanssa.

Suunnittelumallit tukevat suunnittelu- ja kehitystyötä ensisijaisesti yleisellä tasolla, sillä keskipisteessä eivät niinkään ole tekniikka tai koodi, vaan paremminkin dokumentointikulttuurin ja laadukkaan suunnittelun tukeminen. Näin kokeneet suunnittelijat voivat levittää kokemuksiaan ja edesauttaa tällä tavalla yhteisten käytäntöjen syntymistä.

Suunnittelumallit eivät kuitenkaan ratkaise kaikkia ongelmia. Ne helpottavat ohjelmistojen toteutusta ja suunnittelua, sekä mahdollistavat uudelleenkäytön tehokkaan hyödyntämisen, mutta koska ongelmat ovat ainutlaatuisia, kaikkiin ongelmiin ei välttämättä löydy sopivaa valmista suunnittelumallia.

1.2 Suunnittelumallin määritelmä

Suunnittelumallin käsite on lainattu rakennusarkkitehtuurista. Suunnittelumalli voidaan määritellä vaikka seuraavasti:

Suunnittelumalli nimeää, tarjoaa taustatietoa ja selittää käytännössä hyväksi todetun tavan ratkaista jokin järjestelmissä usein esiintyvä arkkitehtuuri- tai suunnitteluongelma. Ratkaisu on yleinen kokoelma olioita ja luokkia, joita sovelletaan ja muovataan ratkaisemaan ongelma erilaisissa käytännön tilanteissa.

1.3 Suunnittelumallien luokittelu

Suunnittelumallit luokitellaan usein kahden kriteerin perusteella. Ensimmäinen kriteeri on tarkoitus, joka kuvaa mitä suunnittelumalli tekee. Toinen kriteeri on ympäristö, joka puolestaan kuvaa, soveltuuko suunnittelumalli käytettäväksi pääasiassa luokkiin vai olioihin. Käyttötarkoituksen mukaan suunnittelumallit voidaan jaotella kolmeen luokkaan [GaH00]:

- **Toiminnalliset-/käyttäytymismallit** (*behavioral patterns*):Toiminnalliset luokkamallit käyttävät perintää jakamaan toiminnot eri luokkiin. Toiminnallisten oliomallien perusmenetelmänä on sitä vastoin olioiden yhdistäminen. Ne auttavat ratkaisemaan mm. seuraavanlaisia ongelmia: Kuinka olion tila voidaan tallentaa ja palauttaa tarvittaessa myöhemmin. Kuinka välittää tietoja muille olioille ilman, että oliot tietävät toistensa luokkia.
- **Luontimallit** (*creational patterns*): Luontimallit auttavat ratkaisemaan mm. seuraavanlaisia käytännön ongelmia: Kuinka luoda olioita ympäristössä, jossa liittymänä on joukko tapauskohtaisia abstrakteja luokkia tai kuinka varmistetaan, että luokalla on vain yksi ilmentymä, joka on kaikkien muitten luokkien saatavissa.
- **Rakennemallit** (*structural patterns*): Rakennemallit kuvaavat olioiden välisiä suhteita ja sitä, kuinka olioita yhdistelemällä saadaan uusia toimintoja. Ne auttavat rat-

kaisemaan esimerkiksi muistinkäytön vähentämiseen tai tehokkuuden lisäämiseen liittyviä ongelmia, kun joudutaan luomaan miljoonia olioita tai kuinka voidaan sovittaa yhteen kaksi yhteensopimatonta järjestelmää.

1.4 Dokumentointi

Suunnittelumallien yhtenä tarkoituksena on auttaa ymmärtämään aikaisemmin onnistuneita ratkaisuja. Tästä syystä suunnittelumallin on oltava hyvin kuvattu ja dokumentoitu. Hyvin kirjoitetun ja dokumentoidun suunnittelumallin kuvaus sisältää seuraavanlaisia asioita:

- **Nimi:** Suunnittelumallille tulee antaa lyhyt ja kuvaava nimi (substantiivi). Nimen tulee olla kuvaava, jotta ratkaisua etsivän suunnittelijan olisi helpompi löytää ongelmaansa soveltuva suunnittelumalli. Toisaalta hyvä nimi helpottaa suunnittelijoiden välistä keskustelua mallista.
- **Tarkoitus:** Tarkoitus-kappale kuvaa yhteenvedonomaisesti, mitä suunnittelumalli tekee ja kuvaa sen suunnitteluasian tai – ongelman, johon suunnittelumalli keskittyy.
- **Ongelma:** Kuvaa ratkaistavan ongelman kontekstissaan. Lyhyt ongelman kuvaus helpottaa suunnittelijaa päättämään, onko suunnittelumalli sopiva juuri hänen kohtaamaansa ongelmaan.
- **Konteksti:** Sisältää historian niistä malleista, joita on käytetty ongelman ratkaisemiseksi aikaisemmin. Kuvaa voimien tilan.
- **Voimat:** Suunnittelumallit eivät ole sääntöjä, joita pitäisi seurata sokeasti, vaan suunnittelumalli täytyy ymmärtää ja soveltaa omaan ongelmaan sopivaksi. Jos ymmärtää mallin voimat, ymmärtää myös ongelman ja sen ratkaisun, tämän vuoksi voimat ovat suunnittelumallin ydin. Systeemissä siis vaikuttaa aina voimia ja näillä voimilla on sivuvaikutuksia. Suunnittelumalli kapseloi toisiinsa vaikuttavat voimat.
- **Ratkaisu:** Ratkaisu tasapainottaa voimat ja vaikuttaa systeemiin. Hyvä ratkaisu on tarpeeksi yksityiskohtainen, jotta suunnittelija tietää mitä tehdä, mutta tarpeeksi yleinen, jotta se soveltuu monenlaisiin konteksteihin. Oliokeskeisissä suunnittelumalleissa kuvataan luokkien ja olioiden väliset suhteet ratkaisussa. Myös sellaiset voimat, jotka jäävät ratkaisematta, tulee dokumentoida.

- **Seuraukset:** Tässä kappaleessa kuvataan suunnittelumallilla saavutettavat hyödyt ja haitat.
- **Toteutus:** Kuvataan suunnittelumallin toteutusvaihtoehdot (koodiesimerkkejä). Toteutusvaihtoehdot voivat olla ohjelmointikielestä riippuvia. Kappaleen tulisi sisältää myös soveltamisesimerkki.

Lisäksi suunnittelumallin dokumentissa voidaan kuvata tunnettuja käyttökohteita ja suunnittelumallille läheisiä, muita suunnittelumalleja.

1.5 Tämän dokumentin sisältö

Tässä dokumentissa esiteltävät suunnittelumallit on jaoteltu Gamman et.al [GaH00] mukaan toiminnallisiin-, rakenne- ja luontimalleihin. Lisäksi kahdessa viimeisessä luvussa esitellään tapahtumamalleja ja hajautetun arkkitehtuurin malleja Grandin [Gra02] mukaan. Muutamia poikkeuksia lukuun ottamatta kustakin suunnittelumallista kuvataan soveltuvuus, toteutus ja tärkeimmät ominaisuudet. Mahdollisimman monesta mallista on esitetty myös kaaviokuva ja/tai Java-kielinen koodiesimerkki. Esimerkit ovat testattuja ja toimivia.

2 TOIMINNALLISET SUUNNITTELUMALLIT

Toiminnallisten suunnittelumallien kohdealueena ovat algoritmit ja vastuun jakaminen olioiden välillä. Toiminnalliset mallit eivät siis tarjoa apua pelkkien olioiden mallintamiseen vaan myös olioiden väliseen kommunikaatioon. Jos jokin ohjelman toiminto muuttuu usein, toiminnalliset mallit tarjoavat keinon kapseloida tällainen toiminto yhteen olioon, jolloin muutosten tekeminen on helpompaa ja muutosten seuraukset hallittavampia.

2.1 Template Method

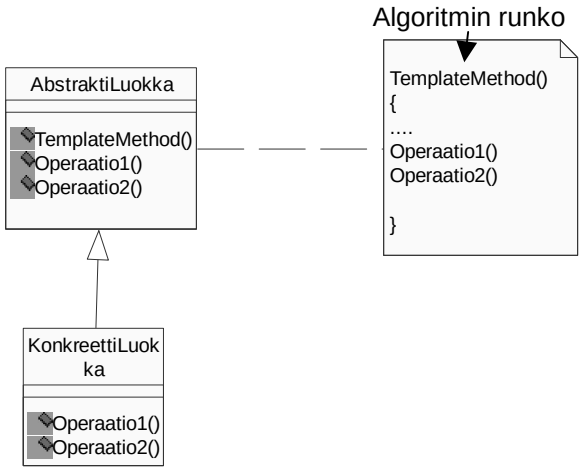
Kehysmetodi-suunnittelumalli (template method) on yksi ohjelmoinnin perustekniikoista, joka löytyy lähes jokaisesta abstraktista luokasta. Kehysmetodin tarkoituksena on määritellä operaatiossa algoritmin runko ja jättää jotkin sen osat aliluokkien toteutettavaksi. Kehysmetodi sallii siis tiettyjen algoritmin osien uudelleenmäärittelyn aliluokissa algoritmin rakenteen pysyessä muuttumattomana.

2.1.1 Soveltuvuus

Kehysmetodi soveltuu käytettäväksi seuraavanlaisissa tilanteissa:

- Kun halutaan toteuttaa algoritmin muuttumattomat osat vain kerran ja jättää aliluokkien tehtäväksi toteuttaa algoritmin muuttuvat osat.
- Kun halutaan välttää koodin toistamista, eli kootaan aliluokkien yhteinen käyttäytyminen yhteiseen luokkaan. Keshysmetodi soveltuu näin hyvin luokkakirjastoihin.
- Kun halutaan valvoa aliluokkien laajenemista. Laajennukset voidaan sallia vain tietyissä kohdissa kutsumalla "*koukkuoperaatioita*".
- Sovelluskehykset (Frameworks) toimivat Kehysmetodi-periaatteella

2.1.2 Toteutus



Kuva 1: Kehysmetodi-suunnittelumallin rakenne.

Kehysmetodi-suunnittelumallin osat:

- **AbstraktiLuokka:** Määrittelee algoritmin ns. primitiivioperaatiot, joilla konkreettiset aliluokat toteuttavat algoritmin askeleet. Toteuttaa myös kehysmetodin, joka määrittelee algoritmin rungon.
- **KonkreettiLuokka:** Toteuttaa primitiivioperaatiot, jotka vastaavat algoritmin aliluokkakohtaisia askeleita. Konkreettiluokka luottaa siihen, että AbstraktiLuokka toteuttaa algoritmin muuttumattomat osat.

Seuraavassa toteutusesimerkissä TemplateMethod-algoritmin muuttumattomassa osassa pyydetään käyttäjää antamaan kaksi kokonaislukua. Koukkumetodit koukku1 ja koukku2

käsittelevät lukuja eri tavoin: koukku1-metodissa lasketaan annettujen lukujen summa ja koukku2-metodissa samojen lukujen erotus, eli vähennetään pienempi luku suuremmasta. Algoritmin loppuosa on muuttumatonta koodia. Toimintaidealtaan esimerkki on sellainen, että esimerkin ensimmäinen osa voidaan toimittaa asiakkaalle käännettynä. Asiakas voi itse tehdä tai muokata koukkukohdat omiin tarkoitusperiinsä sopiviksi ja liittää kokonaisuuteen.

----------Ensimmäinen osa alkaa----------

```
import java.io.*;

abstract class ApplicationFramework {
          static BufferedReader stdin = new BufferedReader(new InputStream-
Reader(System.in));
          int ekaLuku = 0;
          int tokaLuku = 0;
          int summa = 0;
          int erotus = 0;
          boolean ok;
  public ApplicationFramework() {
    templateMethod();
  }
  abstract void koukku1();
  abstract void koukku2();
  // "private" tarkoittaa automaattisesti myös "final":
  private void templateMethod() {
          System.out.println("Alussa muuttumatonta koodia ennen 'koukkuja'");
          System.out.println("Pelataan luvuilla. Anna kokonaisluku.");
          do {
              try {
              ekaLuku = Integer.parseInt(stdin.readLine());
              System.out.println("Anna toinen kokonaisluku.");
              tokaLuku = Integer.parseInt(stdin.readLine());

//Tästä alkaa se osa koodia, jolle voidaan määritellä myöhemmin millainen toteu-
tus vain

              koukku1();
              koukku2();
              //Muutettava osa päättyy tähän.
              ok = true;
          } catch (Exception e){ //varmistetaan, että luvut kokonaislukuja.
              System.out.println("Kelvoton kokonaisluku. Anna uusi!");
              ok = false;
                    }
          }
          while (!ok);
          System.out.println("Taas voisi olla muuttumatonta koodia");
          }
  }
```

----------Ensimmäinen osa loppuu tähän----------

```
// Luodaan uusi "application":
class MyApp extends ApplicationFramework {
  void koukku1() {
          System.out.println("Ollaan 1.koukussa...");
          summa = ekaLuku + tokaLuku;
```

```
    System.out.println("Antamiesi lukujen summa on: " +summa);
  }
  void koukku2() {
          System.out.println("Ollaan 2.koukussa...");
          if (ekaLuku > tokaLuku) {
          erotus = ekaLuku - tokaLuku;
          System.out.println("Eka oli suurempi, erotus:" +erotus);
          }
          if (ekaLuku < tokaLuku) {
                    erotus = tokaLuku - ekaLuku;
          System.out.println("Toinen oli suurempi, erotus:" +erotus);
    }
    if (ekaLuku == tokaLuku)
          System.out.println("Antamasi luvut olivat samat.");
  }
}
public class TemplateMethod {
  MyApp app = new MyApp();
  public void test() {
  }
  public static void main(String args[]) {
    new TemplateMethod().test();
  }
}
```

2.1.3 Ominaisuuksia

Kehysmetodi on oleellinen käsite puhuttaessa sovelluskehyksistä. Sovelluskehyksissä (kehysmetodissa) on käänteinen kontrollirakenne luokkakirjastoihin verrattuna. Tätä kutsutaan "Hollywood-periaatteeksi". Tämä tarkoittaa sitä, että yliluokka kutsuu aliluokkien operaatioita eikä toisinpäin kuten luokkakirjastoissa. Sovelluskehysten ideana on se, että peritään luokka tai joukko luokkia ja käytetään suurinta osaa valmiista koodista hyväksi uudessa sovelluksessa. Yksi tai useampi metodi voidaan määritellä uudelleen, jotta saadaan räätälöidyksi sellainen uusi sovellus kuin halutaan. Esimerkiksi applet-mekanismi on kehysmetodi. Ensiksi peritään init() -metodi JApplet yliluokasta ja määritellään se sitten uudelleen käyttötarkoitusta vastaavaksi.

2.2 State

Tila-mallia (State) käytetään, kun halutaan olion muuttavan käyttäytymistään, kun sen sisäinen tila muuttuu. Antaa vaikutelman, että olio olisi muuttanut luokkaansa.

2.2.1 Soveltuvuus

Tila-malli soveltuu tilanteisiin, joissa olion käyttäytyminen riippuu sen tilasta ja olion on vaihdettava käyttäytymistään ajon aikana tilansa mukaisesti. Hyvänä esimerkkinä tällaisesta on TCP-yhteys, jonka tiloja voivat olla suljettu, muodostettu ja kuuntelee. Tiettyyn saapuvaan pyyntöön vastataan eri tavalla riippuen siitä tilasta, joka yhteydellä on pyynnön saapuessa. Malli soveltuu myös tilanteisiin, joissa operaatiot sisältävät laajoja ja monimutkaisia ehtolausekkeita, jotka riippuvat olion tilasta. Tällöin jokainen ehtohaara muutetaan erilliseksi luokaksi ja näin olion tilaa voidaan kohdella myös oliona ja tämä olio voi muuttaa toimintaansa tilaoliota vaihtamalla. Tällöin myös tilaolion toimintaa voidaan muuttaa muista olioista riippumatta.

2.2.2 Toteutus

Siirtymät tilojen välillä voidaan toteuttaa joko sisäisesti tai ulkoisesti. Toisin sanoen Tila-malli ei ota kantaa siihen, kuka määrittelee tilasiirtymien ehdot. Jos tilasiirtymien ehdot ovat kiinteät, ne voidaan määritellä ns. konteksti-oliossa. On myös mahdollista, että jokainen tila voi itse automaattisesti päättää, mikä sen seuraajatila on. Toteutuksessa on myös päätettävä, miten tilaolioita luodaan ja tuhotaan. Tilaoliot voidaan luoda vasta kun niitä tarvitaan ja tuhota heti käytön jälkeen. Tilaoliot voidaan luoda myös etukäteen ja niitä ei tuhota ollenkaan.

Esimerkissä kuvataan, kuinka opiskelijasta tulee erilaisten "operaatioiden" kautta tohtori.

```
interface StateBase { //Rajapintamäärittly
  void f();
  void g();
  void h();
}
class State implements StateBase { //Tila-luokka, toteuttaa rajapinnan
  private StateBase implementation;
  public State(StateBase imp) {
    implementation = imp;
  }
  public void changeImp(StateBase newImp) { //Ilmentymän vaihtaminen
    implementation = newImp;
  }
  // Tässä määritellään metodeja, joita kutsutaan.
  public void f() { implementation.f(); }
  public void g() { implementation.g(); }
  public void h() { implementation.h(); }
}
class Opiskelija implements StateBase { //Toteuttaa myöskin rajapinnan
  public void f() {
```

```
    System.out.println("Opiskelujaan aloitteleva opiskelija.");
  }
  public void g() {
    System.out.println("Gradun kimpussa puurtava opiskelija.");
  }
  public void h() {
    System.out.println("Valmistuminen maisteriksi.");
  }
}
class Maisteri implements StateBase { //Toteuttaa rajapinnan omalla tavallaan.
  public void f() {
    System.out.println("Jatko-opintojaan suunnitteleva maisteri.");
  }
  public void g() {
    System.out.println("Tutkimusta aloittava maisteri.");
  }
  public void h() {
    System.out.println("Tutkimusta jatkava maisteri.");
  }
}
class Tohtori implements StateBase { //Toteuttaa rajapinnan hieman eri tavalla.
  public void f() {
    System.out.println("Tutkimusta tekevä tohtori!");
  }
  public void g() {
    System.out.println("Tutkimusta dosenttina.!");
  }
  public void h() {
    System.out.println("Tutkimusta professorina");
  }
}
public class StateDemo {
  static void run(State b) {
    b.f();
    b.g();
    b.h();
  }
  State b = new State(new Opiskelija());    //Asetetaan alkutilaksi opiskelija.
  public void test() {
    run(b); //"Ajetaan" tila.
    b.changeImp(new Maisteri()); //Tilasiirtymä (ilmentymän muuttaminen).
    run(b); //"Ajetaan" uusi tila.
    b.changeImp(new Tohtori()); //Tilasiirtymä (ilmentymän muuttaminen).
    run(b); //"Ajetaan" uusi tila.
  }
  public static void main(String args[]) { //Testataan tilamallia.
    new StateDemo().test();
  }
} ///:~
```

2.2.3 Ominaisuuksia

Malli kokoaa tilakohtaisen käyttäytymisen yhteen olioon. Koska erillisille tiloille on omat olionsa, tilasiirtymistä tulee tarkkoja. Tilaoliot voidaan myös jakaa, jos niillä ei ole ilmentymämuuttujia. Tila-malli lähestymistapana lisää pienten luokkaolioiden määrää, mutta itse prosessin aikana se yksinkertaistaa ja selkiyttää ohjelmaa. Javassa kaikki tilat täytyy periä

yleisestä pääluokasta (base class) ja niillä täytyy olla yleiset metodit, vaikka jotkut näistä metodeista voivat olla myös tyhjiä.

2.3 State Machine

Tila-mallin salliessa ohjelmoijan muuttaa toteutusta eri tilojen mukaan, *tilakone* määrää rakenteen, jolla toteutus automaattisesti siirretään oliolta toiselle. Systeemin sen hetkinen toteutus kuvaa systeemin tilan ja systeemi toimii eri tavalla eri tiloissa. Koodi, joka muuttaa systeemin tilasta toiseen, on usein toteutettu käyttämällä Template Method -mallia (kehysmetodia).

2.3.1 Toteutus

Seuraavassa esimerkissä mallinnetaan pesukonetta, jonka tilat vaihtuvat automaattisesti pesun, linkouksen ja huuhteluiden välillä. Tässä esimerkissä luokka (StateMachine-luokka), joka kontrolloi tiloja, on vastuussa myös tilojen muuttamisesta. Tämä koodi on toteutettu käyttämällä Template Method -mallia. Toinen mahdollinen lähestymistapa olisi sallia tilaolioiden itse päättää siitä, mikä sen seuraava tila on. Jälkimmäinen ratkaisutapa olisi joustavampi.

```
package c04.statemachine;
import java.util.*;

//Seuraava esimerkki mallintaa pesukonetta, jossa mahdollisina tiloina on
//pesu, linkous ja huuhtelu

interface State {
  void run();
}

abstract class StateMachine {
  protected State currentState;
  abstract protected boolean changeState();
  // Tässä kohtaa on kehysmetodi (Template method:)
  protected final void runAll() {
           System.out.println("Pesuoperaatio alkaa, pyykit koneeseen");
    while(changeState()) // Tilan muuttaminen
      currentState.run();
    System.out.println("Pyykit pesty.");
  }
}

// Seuraavassa määritellään erilainen aliluokka jokaiselle tilalle
```

```
class Wash implements State {
  public void run() {
    System.out.println("Pesee...");
  }
}

class Spin implements State {
  public void run() {
    System.out.println("Linkoaa...");
  }
}

class Rinse implements State {
  public void run() {
    System.out.println("Huuhtelee...");
  }
}
class Washer extends StateMachine {
  private int i = 0;
  // Tilataulukko:
  private State states[] = {
    new Wash(), new Spin(),
    new Rinse(), new Spin(),
  };
  public Washer() { runAll(); }
  public boolean changeState() {
    if(i < states.length) {
      // Muuttaa tilan asettamalla
      // korvikeviittauksen uuteen olioon
      currentState = states[i++];
      return true;
    } else
      return false;
  }
}
public class StateMachineDemo {
  Washer w = new Washer();
  public void test() {
    // Konstruktori tekee työn, mutta tämä
    // varmistaa, että suoritus päättyy ilman
    // poikkeuksia
  }
  public static void main(String args[]) {
    new StateMachineDemo().test();
  }
} ///:~
```

2.4 Observer

Tarkkailija-suunnittelumallissa ajatellaan, että maailma koostuu kahdenlaisista olioista: *Subjekteista*, joita tarkkaillaan, ja *Tarkkailijoista,* jotka tarkkailevat. Jokaisella Subjektilla voi olla useita Tarkkailijoita, mutta Subjekti ei tunne niiden laatua. Kun Tarkkailija haluaa ryhtyä tarkkailemaan tiettyä Subjektia, se ilmoittaa tälle siitä. Aina, kun Subjektin tila muuttuu, se ilmoittaa tilamuutoksesta kaikille Tarkkailijoilleen. Tarkkailijat reagoivat muu-

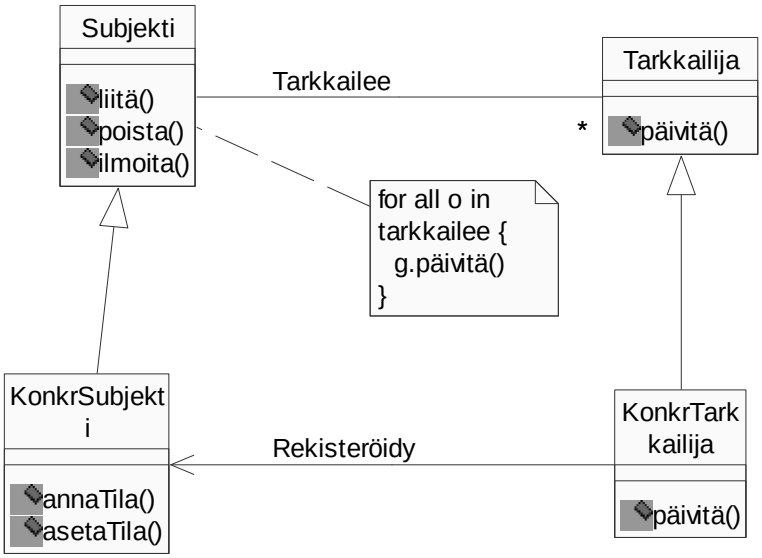
tokseen kukin omalla tavallaan. Usein systeemeissä aiheuttaa ongelmia tilanne, jossa tieto muuttuu yhdessä osassa ja muut osat ovat riippuvaisia tästä tiedosta. Juuri tämän ongelman ratkaisemiseksi Tarkkailija-suunnittelumalli on kehitetty. Se auttaa pitämään yhdessä toimivien osien tilat tasapainossa.

2.4.1 Soveltuvuus

Tarkkailija-suunnittelumalli soveltuu käytettäväksi tilanteissa, joissa toisistaan riippuvat oliot halutaan toteuttaa mahdollisimman riippumattomasti. Mallia kannattaa käyttää esimerkiksi seuraavanlaisissa tilanteissa:

- Kun käsiteltävällä abstraktiolla on kaksi kokonaisuutta, jotka riippuvat toisistaan. Kapseloimalla kumpikin osa omaksi oliokseen, niiden vaihtaminen ja uudelleenkäyttö itsenäisinä kokonaisuuksina tulee myös mahdolliseksi.
- Kun tietyn olion muutoksesta aiheutuu muutostarve myös muihin olioihin. Ei myöskään tiedetä niiden olioiden määrää, joihin muutostarve kohdistuu.
- Kun olion tulee ilmoittaa tilastaan muille olioille tietämättä tarkasti, mitä nämä oliot ovat.

2.4.2 Toteutus



Kuva 2: Tarkkailija-suunnittelumallin rakenne

Malliin osallistujat:

- **Subjekti:** Abstrakti luokka, joka sisältää Tarkkailija-olioiden perusoperaatiot. Tietää tarkkailijansa, joita voi olla lukematon määrä. Tarjoaa rajapinnan Tarkkailija-olioiden liittymiselle ja poistumiselle.
- **Tarkkailija:** Tarkkailijoiden rajapinnan kuvaava abstrakti luokka, joka sisältää päivityksistä huolehtivan operaation. Määrittelee niiden olioiden rajapinnan, joille tiedotetaan Subjektin tilojen muutoksista.
- **Konkreettinen subjekti:** Subjektin jonkin konkreettisen aliluokan ilmentymä. Tiedottaa tarkkailijoitaan muutoksistaan.
- **Konkreettinen tarkkailija:** Tämä olio on jonkin konkreettisen aliluokan ilmentymä. Kukin päivitysoperaatio on toteutettu omalla tavallaan. Pitää yllä viittausta konkreettiseen subjekti-olioon ja tallentaa tilan, jos sen pitää pysyä samana kuin Subjektin tilan.

Tarkkailija-suunnittelumallia voidaan käyttää hyväksi esimerkiksi käyttöliittymäarkkitehtuurissa. Tällöin subjektina on itse sovellus tai sen osa ja tarkkailijana toimii tämän näkymä näytöllä. Kun sovelluksen tila muuttuu, kaikille sen näkymille ilmoitetaan muutoksesta. Vaikka eri elementit riippuvatkin toisistaan, niiden ei kuitenkaan tarvitse tuntea toisiaan. Sovelluksen ei puolestaan tarvitse tuntea näyttöjen tarkempaa laatua eikä toteutustapaa. Tarvitaan ainoastaan tarkkailijarajapinta (operaatio), jolla muutoksesta ilmoitetaan käyttöliittymäelementeille, jotta ne voisivat päivittää näkymänsä.

Tarkkailija-suunnittelumalli siis antaa ratkaisun ongelmaan, jossa koko oliojoukon täytyy reagoida, jos jokin olioista muuttaa tilaansa. Ongelma on niin yleinen, että sen ratkaisu on otettu osaksi **java.util** library -standardia. Javassa on kahdentyyppisiä olioita, joita käytetään toteuttamaan Tarkkailija-malli. **Observable**-luokka pitää lukua kaikista niistä, jotka haluavat saada tiedon, kun jokin muutos tapahtuu. Kun jossakin tapahtuu muutos, Observable kutsuu **notifyObservers()**-metodiaan jokaisen listalla olevan olion kohdalla. **Observer**-luokka on rajapintaluokka, jolla on ainoastaan yksi jäsenfunktio, **update()**. Se olio, jota tarkkaillaan, kutsuu tätä metodia, kun tarkkailijoiden on aika päivittää tilansa.

Observable:ssa on lippukohta, joka osoittaa onko muutosta tapahtunut. Suurimman osan työstä tekee notifyObservers(), joka ei kuitenkaan tee mitään, jos lippukohdan muutosta ei ole asetettu. Pelkkä yksinkertainen Observable-olion käyttäminen ei kuitenkaan riitä hoitamaan muutoksia, vaan sen luokan on perittävä Observable-luokka ja jossakin kohtaa tässä perivässä luokassa on kutsuttava Observable-luokan **setChanged()**-metodia. Tämä

jäsenfunktio asettaa muutoksen lippukohtaan, mikä tarkoittaa sitä, että kun kutsutaan notifyObservers()-metodia, kaikki tarkkailijat saavat tiedon muutoksesta.

Seuraavassa esimerkissä käytetään näitä luokkia ja metodeita. Esimerkki on graafinen ohjelma, jossa on alussa ruutu täynnä erivärisiä kuvioita. Kun jonkin kuvion kohtaa napsauttaa hiirellä, tämän värin alue suurenee näytöllä.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

// Uusi tarkkailijatyyppi on perittävä Observable-luokasta
class BoxObservable extends Observable {
  public void notifyObservers(Object b) {
    // Muuten se ei levitä muutosta
    setChanged();
    super.notifyObservers(b);
  }
}
public class BoxObserver extends JFrame {
  Observable notifier = new BoxObservable();
  public BoxObserver(int grid) {
    setTitle("Mallinnetaan tarkkailija-suunnittelumallia");
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(grid, grid)); //Ruudukko näytölle
    for(int x = 0; x < grid; x++)
      for(int y = 0; y < grid; y++)
        cp.add(new OCBox(x, y, notifier));
  }
  public static void main(String[] args) {
    int grid = 8;
    if(args.length > 0)
      grid = Integer.parseInt(args[0]);
    JFrame f = new BoxObserver(grid);
    f.setSize(500, 400); //Näytön (frame) koon määrittäminen
    f.setVisible(true);
    // JDK 1.3:
    f.setDefaultCloseOperation(EXIT_ON_CLOSE);
    // Liitetään WindowAdapter, jos käytössä JDK 1.2
  }
}
class OCBox extends JPanel implements Observer {
  Observable notifier;
  int x, y; // Paikat ruudukossa
  Color cColor = newColor();
  static final Color[] colors = { //Määritellään käytettävien värien taulukko
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  static final Color newColor() { //Arvotaan uusi väri taulukosta
    return colors[
      (int)(Math.random() * colors.length)
```

```
    ];
  }
  OCBox(int x, int y, Observable notifier) {
    this.x = x;
    this.y = y;
    notifier.addObserver(this);
    this.notifier = notifier;
    addMouseListener(new ML());
  }
  public void paintComponent(Graphics g) { //Piirretään kuvio
    super.paintComponent(g);
    g.setColor(cColor);          //Asetetaan kuviolle väri
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
  class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) { //Tapahtuman "kuuntelija", tark
                                             //kaillaan hiiren painalluksia
      notifier.notifyObservers(OCBox.this);
    }
  }
  public void update(Observable o, Object arg) { //Päivitysmetodi
    OCBox clicked = (OCBox)arg;
    if(nextTo(clicked)) {
      cColor = clicked.cColor;   //Asetetaan se väri, jota hiirellä napsautettu
      repaint(); //Päivitetään piirros
    }
  }
  private final boolean nextTo(OCBox b) {
    return Math.abs(x - b.x) <= 1 &&
           Math.abs(y - b.y) <= 1;
  }
} ///:~
```

2.4.3 Ominaisuuksia

Tarkkailija-suunnittelumalli sallii Subjektien ja Tarkkailijoiden riippumattoman vaihtelun. Subjekteja voidaan uudelleenkäyttää ilman, että niiden Tarkkailijoita tarvitsisi käyttää. Sama pätee myös toisinpäin. Malli sallii myös lisätä Tarkkailijoita ilman, että Subjektia tai muita Tarkkailijoita tarvitsisi muuttaa. Muita ominaisuuksia:

- Subjektin ja Tarkkailijan välillä on abstrakti kytkentä. Tämä tarkoittaa sitä, että Subjekti tietää vain sen, että sillä on tarkkailijoita, joista jokainen käyttää Tarkkailija-luokan yksinkertaista rajapintaa. Subjekti ei sen sijaan tiedä yhdenkään Tarkkailijan luokkaa. Kytkennän tulee olla myös mahdollisimman vähäistä. Kun kytkentä ei ole tiukkaa, Subjekti ja Tarkkailija voivat kuulua systeemin eri abstraktiokerroksiin, eli alemman tason subjekti voi esimerkiksi informoida ylemmän tason Tarkkailijaa.
- Subjektin lähettämälle tiedolle ei tarvitse määritellä saajaa. Tieto lähetetään automaattisesti kaikille olioille, joille sillä on merkitystä. Koska Subjektille ei ole merkitystä kuin-

ka paljon sillä on Tarkkailijoita, tarkkailijoiden määrää voidaan vaihdella missä vaiheessa tahansa.

- Koska tarkkailijoilla ei ole lainkaan tietoa toistensa olemassaolosta, ne eivät ymmärrä miten suuri kokonaisvaikutus Subjektin muuttamisesta voi aiheutua. Huonosti määritellyt ja ylläpidetyt riippuvuudet voivat myös aiheuttaa ongelmia päivityksissä. Nämä päivitykset voivat olla hyvin vaikeasti jäljitettävissä, koska tavallinen päivitysmenetelmä ei sisällä tietoa siitä, mikä Subjektissa on muuttunut.
- Malli voi olla kaksisuuntainen.

2.5 Strategy

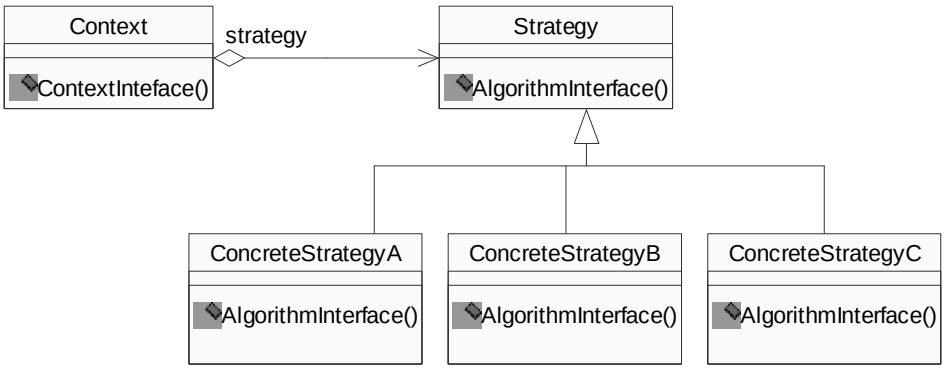
Strategia-suunnittelumallin ideana on kapseloida kukin erillinen algoritmi omaan luokkaansa siten, että algoritmit ovat helposti vaihdettavissa. Algoritmien käyttäminen tapahtuu niin sanottujen *Strategia-olioiden* avulla. Mallin päätarkoituksena on siis algoritmien vaihdettavuus niiden käyttäjistä riippumatta.

2.5.1 Soveltuvuus

Strategia-suunnittelumalli on käyttökelpoinen esimerkiksi seuraavanlaisissa tilanteissa:

- Kun joukko toisiinsa liittyviä luokkia eroaa toisistaan vain käyttäytymiseltään. Mallin avulla luokkaan saadaan liitettyä yksi useasta tarjolla olevasta käyttäytymistavasta.
- Algoritmista tarvitaan useita erilaisia variaatioita. Algoritmeilla voi esimerkiksi olla erilaisia muisti- ja suorituskykyvaatimuksia. Strategia-mallia voidaan käyttää, jos variaatiot toteutetaan algoritmeista muodostetussa luokkahierarkiassa.
- Algoritmi käyttää hyväkseen sellaisia tietoja, joista sovelluksen ei pidä tietää mitään. Strategia-mallia käytetään siis kätkemään monimutkaisia algoritmikohtaisia tietorakenteita.
- Kun luokka määrittelee erilaisia käyttäytymistapoja ja nämä käyttäytymistavat ilmenevät sen operaatioissa useina ehtolauseina. Useiden ehtolauseiden käytön sijasta Strategia-mallissa ehdolliset haarat siirretään omiin Strategia-luokkiinsa.

2.5.2 Toteutus



Kuva 3: Strategia-suunnittelumallin rakenne.

Strategia-suunnittelumallin osallistujat:

- **Context:** Konfiguroidaan yhteen ConcreteStrategy-olion kanssa. Pitää yllä viitettä Strategy-olioon ja voi määritellä rajapinnan, jonka kautta Strategy pääsee käsiksi Context-olion tietoihin.
- **Strategy:** Määrittelee kaikille algoritmeille yhteisen rajapinnan. Esimerkiksi Java-kielen toteutuksessa tämä voidaan määritellä rajapinnaksi tai vaikka abstraktiksi luokaksi. Asiakkaat käyttävät algoritmeja tämän rajapinnan kautta.
- **ConcreteStrategy:** Tässä roolissa olevat luokat tarjoavat toteutukset eri algoritmeille tai ne voivat tarjota erilaiset toteutukset samallekin algoritmille. Näiden luokkien on toteutettava yhteinen rajapinta, jota käytetään algoritmien kutsumiseen.

Strategia-suunnittelumalli eristää eri algoritmit omiin luokkiinsa, joita kutsutaan Strategia-luokiksi. Algoritmeja käyttävät oliot puolestaan sisältävät viitteet Strategia-olioihin ja algoritmien vaihtaminen tapahtuu vaihtamalla Strategia-oliota.

Seuraava koodiesimerkki on toteutettu Strategia-suunnittelumallia hyväksikäyttäen. Esimerkissä Strategia-olio kontrolloi sitä, minkä olioperheen oliota kulloinkin kutsutaan. Jokainen metodi on omassa luokassaan, jotka ovat yleisen perusluokan aliluokkia.

```
//Abstrakti Strategia-luokka.
public abstract class DvdNameStrategy {
   public abstract String formatDvdName(String dvdName, char charIn);
}

//Konkreettinen Strategia-luokka, toteutukset abstraktin luokan metodeille.
public class DvdNameAllCapStrategy extends DvdNameStrategy {
   public String formatDvdName(String dvdName, char charIn) {
       return dvdName.toUpperCase(); //Muutetaan DVD:n nimen kirjaimet isoiksi.
   }
}
//Toinen konkreettinen Strategia-luokka, abstraktin luokan aliluokka.
public class DvdNameTheAtEndStrategy extends DvdNameStrategy {

//Erilainen toteutus abstraktin yliluokan metodille.
   public String formatDvdName(String dvdName, char charIn) {
       if (dvdName.startsWith("The ")) { //Jos nimi alkaa the-artikkelilla…
                                         //Vaihdetaan artikkeli nimen loppuun.
          return new String(dvdName.substring(4, (dvdName.length())) + ", The");
       }
       if (dvdName.startsWith("THE ")) {
          return new String(dvdName.substring(4, (dvdName.length())) + ", THE");
       }
       if (dvdName.startsWith("the ")) {
          return new String(dvdName.substring(4, (dvdName.length())) + ", the");
       }
       return dvdName;
   }
}
//Kolmas konkreettinen Strategia-luokka.
public class DvdNameReplaceSpacesStrategy extends DvdNameStrategy {
           //Vielä erilainen toteutus DVD:n nimen muokkaamiselle
   public String formatDvdName(String dvdName, char charIn) {
       return dvdName.replace(' ', charIn); //Korvataan välilyönnit parametrina
                                            // saadulla merkillä.
   }
}
//Context-luokka
public class DvdNameContext {
   private DvdNameStrategy dvdNameStrategy;

   public DvdNameContext(char strategyTypeIn) { // Mihin kategoriaan DVD kuuluu.
       switch (strategyTypeIn) { //Kategoria saadaan parametrina.
           case 'C' : this.dvdNameStrategy = new DvdNameAllCapStrategy();
                      break;
           case 'E' : this.dvdNameStrategy = new DvdNameTheAtEndStrategy();
                      break;
           case 'S' : this.dvdNameStrategy = new DvdNameReplaceSpacesStrategy();
                      break;
           default  : this.dvdNameStrategy = new DvdNameTheAtEndStrategy();
       }
   }
   public String[] formatDvdNames(String[] namesIn) {
       return this.formatDvdNames(namesIn, ' ');
   }
   public String[] formatDvdNames(String[] namesIn, char replacementIn) {
       String[] namesOut = new String[namesIn.length]; //Taulukko nimille.
       for (int i = 0; i < namesIn.length; i++) { //käydään taulukko lävitse.
           namesOut[i] = dvdNameStrategy.formatDvdName(namesIn[i],
                                                       replacementIn);
           System.out.println("Dvd:n nimi ennen muutoksia: " + namesIn[i]);
```

```
            System.out.println("Dvd:n nimi, kun on tehty muutoksia:  "
                                + namesOut[i]);
            System.out.println("=========================");
        }
        return namesOut;
    }
}
//Strategia-suunnittelumallin testiluokka
class TestDvdStrategy {

    public static void main(String[] args) {
            //Annetaan kategoria parametrina
        DvdNameContext allCapContext = new DvdNameContext('C');
        DvdNameContext theEndContext = new DvdNameContext('E');
        DvdNameContext spacesContext = new DvdNameContext('S');

        String dvdNames[] = new String[3]; //Luodaan DVD:n nimistä taulukko.
        dvdNames[0] = "Jay and Silent Bob Strike Back";
        dvdNames[1] = "The Fast and the Furious";
        dvdNames[2] = "The Others";

        char replaceChar = '*'; //Merkki, jolla välilyönnit korvataan.

        System.out.println("Aloitetaan muutosten tekeminen...");
        String[] dvdCapNames = allCapContext.formatDvdNames(dvdNames);

        System.out.println(" ");
        System.out.println("Testataan muuntamalla alusta loppuun");
        String[] dvdEndNames = theEndContext.formatDvdNames(dvdNames);

        System.out.println(" ");
        System.out.println("Testataan laittamalla tyhjiin paikkoihin "
                               + replaceChar);
        String[] dvdSpcNames = spacesContext.formatDvdNames(dvdNames,
                                                            replaceChar);
    }
}
```

2.5.3 Ominaisuuksia

Strategia-suunnittelumallilla on omat hyvät ja huonot puolensa:

- Strategia-luokkien hierarkia määrittelee algoritmiperheen tai käyttäytymisjoukon, joita Context voi käyttää uudelleen. Perintä auttaa löytämään algoritmien yhteiset ominaisuudet.
- Kun kapseloidaan algoritmit omiin Strategia-luokkiinsa, pystytään algoritmeja muuttamaan käyttöyhteydestä riippumattomasti. Tämän seurauksena algoritmeja on helpompi ymmärtää, laajentaa ja vaihtaa.
- Voidaan käyttää eri toteutusvaihtoehtoja. Sama algoritmi voidaan siis toteuttaa usealla eri tavalla.

- Strategia-luokkien käyttö eliminoi ehtolauseiden käytön. Suunnittelumalli tarjoaa vaihtoehdon ehtolauseiden käytölle tietyn ominaisuuden valitsemiseksi. Jos eri toiminnallisuudet olisivat samassa luokassa, niiden käyttöä olisi vaikea estää oikean toiminnallisuuden löytämiseksi.
- Asiakkaiden on oltava tietoisia eri strategioista. Haittapuolena mallissa siis on, että asiakkaiden on ymmärrettävä kuinka eri strategiat eroavat toisistaan, jotta ne pystyisivät valitsemaan algoritmeista sen, jota haluavat käyttää.
- Asiakkaan ja strategian välisestä kommunikoinnista aiheutuu kustannuksia. Saattaa syntyä tilanteita, joissa kaikki konkreettiset Strategia-luokat eivät välttämättä käytä kaikkia parametreja, jotka saadaan yhteisen rajapinnan kautta.
- Strategia-suunnittelumallin käyttäminen lisää sovelluksessa luotavien olioiden määrää. Joissain tilanteissa olioiden määrää voidaan pienentää toteuttamalla Strategia-oliot tilattomina.

2.6 Chain of Responsibility

Vastuuketju-suunnittelumallin perusajatuksena on ketjuttaa suorituspyynnön mahdolliset toteuttajat siten, että lähettäjä ei tiedä, mikä ketjussa oleva luokka tai olio suorituspyynnön tulee toteuttamaan. Mallin avulla voidaan useammalle objektille tarjota mahdollisuus käsitellä suorituspyyntö. Ellei luokka halua toteuttaa suorituspyyntöä, luokalla on vastuu lähettää pyyntö edelleen seuraavalle ketjussa olevalle luokalle. Ketjutus toimii joko "automaattisesti" joko periytyvyyttä hyväksikäyttäen tai ylläpitämällä viitettä tai linkkiä ketjussa seuraavana olevaan objektiin tai molemmat keinot yhdistämällä. Tavallisimmin ketju rakennetaan tarkimmasta yleisimpään – periaatteella.

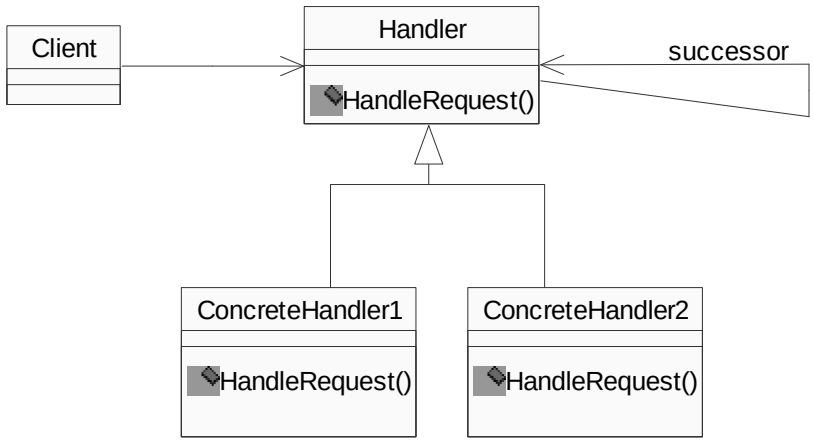
2.6.1 Soveltuvuus

Vastuuketju suunnittelumalli soveltuu käytettäväksi seuraavanlaisissa tilanteissa:

- Kun useampi kuin yksi objekti voi käsitellä suorituspyynnön ja käsittelijää ei tiedetä ennakolta. Käsittelijä tulisi varmistaa automaattisesti.
- Halutaan tehdä pyyntö yhdelle useasta objektista ilman, että vastaanottajaa määritellään tarkasti.
- Objektien joukko, joka pyyntöä voi käsitellä, tulisi määritellä dynaamisesti.

2.6.2 Toteutus

Vastuuketju-malli voidaan siis toteuttaa *periytyvyyden* tai *linkkien* avulla. Malli erottaa lähettäjät ja vastaajat toisistaan, täten useammat objektit saavat mahdollisuuden käsitellä pyyntö. Pyyntöä siirretään ketjussa eteenpäin, kunnes joku objekteista käsittelee pyynnön. Ensimmäinen ketjun objekti vastaanottaa pyynnön ja joko käsittelee tai jatkaa sen seuraavalle ehdokkaalle ketjussa, näin jatketaan, kunnes joku objekteista käsittelee pyynnön. Sillä objektilla, joka teki pyynnön, ei ole tarkkaa tietoa siitä, kuka pyynnön tulee käsittelemään, sanotaankin, että pyynnöllä on epäsuora vastaanottaja. Toimittaakseen pyynnön eteenpäin ketjussa ja varmistaakseen vastaanottajien pysymisen epäsuorana, jokainen objekti ketjussa jakaa yhteisen rajapinnan.



Kuva 4: Vastuuketju-mallin rakenne.

Malliin osallistujat:

- **Handler:** Käsittelijä. Määrittelee rajapinnan käsittelypyynnöille ja toteuttaa seuraajalinkin.
- **ConcreteHandler:** Todellinen käsittelijä. Käsittelee ne pyynnöt, joista se on vastuussa. Voi olla yhteydessä seuraajaan (successor). Jos todellinen käsittelijä voi käsitellä pyynnön, se käsittelee pyynnön. Muussa tapauksessa se siirtää pyynnön seuraajalleen.
- **Client:** (Toimeksiantaja). Tekee pyynnön konkreettiselle kohteelle ketjussa.

Kun toimeksiantaja (Client) tekee pyynnön, se etenee ketjussa, kunnes todellinen käsittelijä (ConcreteHandler) ottaa vastuun pyynnön käsittelemisestä. Seuraavassa koodiesimer-

kissä pyritään määrittämään dvd-komedian koko otsikko. Yhdestä luokasta kutsuttu metodi voi "liikkua" luokkahierarkiassa löytääkseen lopulta oikean olion, joka voi käsitellä pyynnön asianmukaisella tavalla.

```
//Rajapinta kaikille Vastuuketju-mallin luokille.
public interface TopTitle {
   public String getTopTitle();
   public String getAllCategories();
}
//Ylin luokka ketjussa. Toteuttaa rajapinnan.
public class DvdCategory implements TopTitle {
   private String category;
   private String topCategoryTitle;

   public DvdCategory(String category) {
        this.setCategory(category);
   }
   public void setCategory(String categoryIn) { Kategoriaksi parametrina saatu
           this.category = categoryIn;
   }
   public String getCategory() { //Palauttaa kategorian
           return this.category;
   }
   public String getAllCategories() {
           return getCategory();
   }
   //Asetetaan  pääkategorian nimeksi parametrina saatu String.
   public void setTopCategoryTitle(String topCategoryTitleIn) {
           this.topCategoryTitle = topCategoryTitleIn;
   }
   public String getTopCategoryTitle() { //Palaittaa kategorian otsikon.
           return this.topCategoryTitle;
   }
   public String getTopTitle() {
           return this.topCategoryTitle;
   }
}
//Keskimmäinen vastuuketju-esimerkin luokista.
public class DvdSubCategory implements TopTitle {  //määrittelee alakategoriat
   private String subCategory;
   private String topSubCategoryTitle;
   private DvdCategory parent;

   public DvdSubCategory(DvdCategory dvdCategory, String subCategory) {
        this.setSubCategory(subCategory); //Asetetaan alakategoria.
        this.parent = dvdCategory;        //Määritellään kuka on vanhempi.
   }
   public void setSubCategory(String subCategoryIn) {
           this.subCategory = subCategoryIn;}
   public String getSubCategory() {
           return this.subCategory;}
   public void setCategory(String categoryIn) { //Asetetaan kategoria.
           parent.setCategory(categoryIn);}
   public String getCategory() {
           return parent.getCategory();}
   public String getAllCategories() { //Palauttaa kaikki kategoriat.
           return (getCategory() + "/" + getSubCategory());}
```

```
   public void setTopSubCategoryTitle(String topSubCategoryTitleIn) {
           //Asetetaan alakategorian otsikoksi parametrina saatu.
           this.topSubCategoryTitle = topSubCategoryTitleIn;
   }
   public String getTopSubCategoryTitle() { //palautetaan otsikko.
           return this.topSubCategoryTitle;}
   public void setTopCategoryTitle(String topCategoryTitleIn){
           parent.setTopCategoryTitle(topCategoryTitleIn);}
   public String getTopCategoryTitle() {
           return parent.getTopCategoryTitle();}

   public String getTopTitle() { //Palautetaan ylin kategoriaotsikko
       if (null != getTopSubCategoryTitle()){
           return this.getTopSubCategoryTitle();
       }
       else {
           System.out.println("ei korkeampaa otsikkoa Category/SubCategory " +
                              getAllCategories()+":ssa");
           return parent.getTopTitle();
       }
   }
}
//Viimeinen lenkki vastuuketjua.
public class DvdSubSubCategory implements TopTitle {   //ala-alakategoria
   private String subSubCategory;
   private String topSubSubCategoryTitle;
   private DvdSubCategory parent; //vanhempi alakategoriassa

   public DvdSubSubCategory(DvdSubCategory dvdSubCategory, String subCategory) {
        this.setSubSubCategory(subCategory);
        this.parent = dvdSubCategory;
   }
   public void setSubSubCategory(String subSubCategoryIn) {
           //astetaan ala-alakategoria
           this.subSubCategory = subSubCategoryIn;
   }
   public String getSubSubCategory() { //Palautetaan.
           return this.subSubCategory;}
   public void setSubCategory(String subCategoryIn) { //Asetetaan alakategoria.
           parent.setSubCategory(subCategoryIn);
   }
   public String getSubCategory() { //Palautetaan
           return parent.getSubCategory();
   }
   public void setCategory(String categoryIn) { //Asetetaan ylin kategoria.
           parent.setCategory(categoryIn);
   }
   public String getCategory() {
           return parent.getCategory();
   }
   public String getAllCategories() {  //Palautetaan kaikki kategoriat.
return (getCategory() + "/" + getSubCategory() + "/" + getSubSubCategory());}

   public void setTopSubSubCategoryTitle(String topSubSubCategoryTitleIn) {
           //alin otsikko
           this.topSubSubCategoryTitle = topSubSubCategoryTitleIn;
   }
   public String getTopSubSubCategoryTitle() {
           return this.topSubSubCategoryTitle;}
   public void setTopSubCategoryTitle(String topSubCategoryTitleIn) {
           //Seuraava otsikko
```

```
            parent.setTopSubCategoryTitle(topSubCategoryTitleIn);}
    public String getTopSubCategoryTitle() {
            return parent.getTopSubCategoryTitle();}
    public void setTopCategoryTitle(String topCategoryTitleIn) {
            //ylin otsikko.
            parent.setTopCategoryTitle(topCategoryTitleIn);}
    public String getTopCategoryTitle() {
            return parent.getTopCategoryTitle();}

    public String getTopTitle() {
        if (null != getTopSubSubCategoryTitle()) {
            return this.getTopSubSubCategoryTitle();
        }
        else {
            System.out.println("ei korkeampaa otsikkoa
            Category/SubCategory/SubSubCategory " + getAllCategories()+ ":ssa");
            return parent.getTopTitle();
        }
    }
}
//Testiluokka vastuuketjun toimivuuden testaamiseksi.
class TestChainOfResponsibility {
    public static void main(String[] args) {
         String topTitle;
         DvdCategory comedy = new DvdCategory("Komedia");
         comedy.setTopCategoryTitle("Ghost World");

         DvdSubCategory comedyChildrens = new DvdSubCategory(comedy, "Chil
                                                             drens");

         DvdSubSubCategory comedyChildrensAquatic =
                      new DvdSubSubCategory(comedyChildrens, "Aquatic");
         comedyChildrensAquatic.setTopSubSubCategoryTitle("Sponge Bob Square
                                                             pants");

         System.out.println("");
         System.out.println("Kohotaan otsikkohierarkiassa:");
         topTitle = comedy.getTopTitle();
         System.out.println("Ylin otsikko: " + comedy.getAllCategories()
                                + " on " + topTitle);

         System.out.println("");
         System.out.println("Kohotaan comedy/childrens otsikoissa:");
         topTitle = comedyChildrens.getTopTitle();
         System.out.println("Ylin otsikko " + comedyChildrens.getAllCategories()
                                + ":lle on " + topTitle);

         System.out.println("");
         System.out.println("Kohotaan comedy/childrens/aquatic otsikoissa:");
         topTitle = comedyChildrensAquatic.getTopTitle();
         System.out.println("Ylin otsikko " + comedyChildrensA
                      quatic.getAllCategories() + ":lle on " + topTitle);
    }
}
```

2.6.3 Ominaisuuksia

Vastuuketju-suunnittelumallilla on seuraavanlaisia ominaisuuksia:

- Väljä kytkentä. Objektin ei siis tarvitse tietää, mikä objekti tulee käsittelemään suoritus-pyynnön. Lähettäjällä tai vastaanottajalla ei ole tarkkaa tietoa toisistaan eikä ketjussa olevan objektin tarvitse tietää ketjun rakennetta. Näin ollen suunnittelumalli yksinkertaistaa olioiden välisiä yhteyksiä. Sen sijaan, että oliot ylläpitäisivät viitteitä kaikkiin mahdollisiin vastaanottajiin, ne ylläpitävät vain yhtä viitettä seuraajaansa.
- Vastuuketjun käyttäminen lisää joustavuutta, sillä lisäämällä tai muuten muuttamalla ketjua ajon aikana voidaan muuttaa pyynnön käsittelyn vastuita.
- Koska pyynnöllä ei ole selvää vastaanottajaa, ei voida taata, että pyyntö tullaan käsittelemään. Pyyntö voi tulla ketjun päähän ilman, että sitä on käsitelty. Pyyntö voi jäädä käsittelemättä myös tilanteissa, joissa ketju ei ole oikein muodostettu.

3 LUONTIMALLIT

Luontimallit kapseloivat tiedon siitä, mitä konkreettisia luokkia ohjelma käyttää, sekä sen, miten luokan ilmentymiä luodaan ja koostetaan. Luokkakohtaiset luontimallit käyttävät perintää muuttamaan sitä tapaa, jolla luokka luodaan. Oliokohtaiset luontimallit delegoivat luomisen toisille olioille.

3.1 Factory Method

Tehdasmetodi-suunnittelumalli tunnetaan myös nimellä *Virtual Constructor*. Tehdasmetodia käyttämällä luokka voi siirtää ilmentymien luonnin aliluokille. Se määrittelee rajapinnan, jonka avulla voidaan luoda tietyn luokan olio. Tehdasmetodi on yleensä abstrakti, jolloin periytynyt luokka määrää, minkä luokan olio luodaan. Tehdasmetodi siis määrittelee olion luontioperaation, mutta jättää aliluokkien tehtäväksi päättää, mistä luokasta ilmentymä luodaan.

Tehdasmetodia käytetään yleisesti työkalupakeissa ja sovelluskehyksissä. Sovelluskehykset käyttävät abstrakteja luokkia olioiden välisten relaatioiden määrittelemiseen ja toisaalta sovelluskehys on usein vastuussa myös olioiden luomisesta. Sovelluskehyksen käyttäjän on puolestaan laajennettava sovelluskehyksen abstrakteja luokkia toteuttaakseen omat toimintonsa. Tehdasmetodi tarjoaa ratkaisun tähän ongelmaan kapseloimalla tiedon siitä,

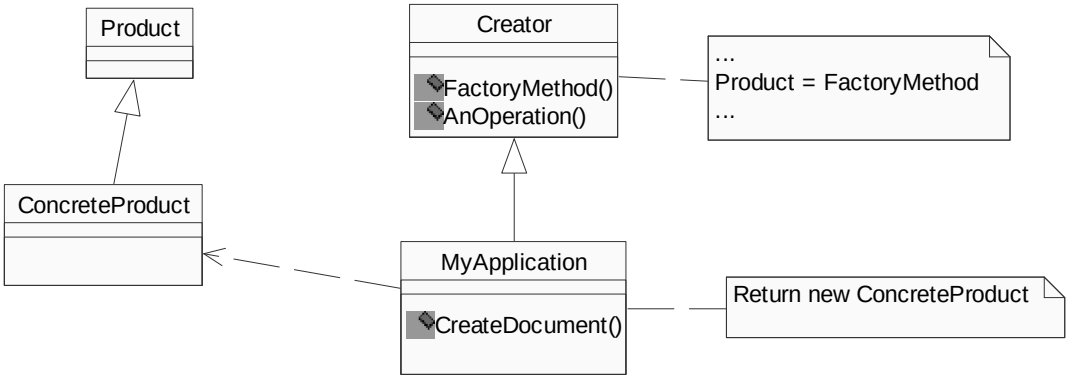
millaisen luokan ilmentymä kussakin tilanteessa luodaan. Näin sovelluskehyksen tietosisältöä voidaan laajentaa.

3.1.1 Soveltuvuus

Tehdasmetodi soveltuu käytettäväksi seuraavanlaisissa tilanteissa:

- Kun luokka ei tiedä etukäteen, mistä luokasta sen on luotava ilmentymiä.
- Kun halutaan, että luokan aliluokat määrittelevät luotavat oliot.
- Luokat delegoivat luontivastuun jollekin apuluokalle ja halutaan keskittää tieto siitä, mikä apuluokka tehtävän hoitaa.

3.1.2 Toteutus



Kuva 5: Tehdasmetodi-mallin rakenne.

Tehdasmetodi-suunnittelumalliin kuuluvat seuraavat luokat (kuva 5):

- **Product:** Määrittelee tehdasmetodin luomien olioiden rajapinnan.
- **ConcreteProduct:** Toteuttaa Product-rajapinnan.
- **Creator:** Määrittelee tehdasmetodin, joka palauttaa Procuct-tyyppisen olion. Voi määritellä tehdasmetodin oletustoteutuksen. Voi myös kutsua tehdasmetodia luodakseen Product-olion.
- **ConcreteCreator:** Korvaa tehdasmetodin metodilla, joka palauttaa ConcreteProduct-ilmentymän.

Tehdasmetodin toteutuksessa on kaksi erilaista variaatiota. Ensinnäkin Creator-luokka voi olla abstrakti, eli tällöin se ei tarjoa tehdasmetodin oletustoteutusta, tai Creator-luokka voi olla konkreettinen tarjoten samalla oletustoteutuksen. On myös mahdollista määritellä sellainen abstrakti luokka, joka määrittelee oletustoteutuksen, mutta tätä tapaa käytetään harvoin.

Seuraavassa yksinkertaisessa esimerkissä piirretään kahdenlaisia kuvioita. Metodi **tehdas()** välittää parametrin, jonka avulla määritellään se, millainen **Kuvio** kulloinkin piirretään. Esimerkkitapauksessa kuvio on vain rivi tekstiä, mutta se voisi olla minkätyyppistä tietoa tahansa. Jos esimerkkiin haluttaisiin liittää lisää piirrettäviä kuvioita, tarvitsisi vain **tehdas()** -kohtaa muuttaa.

```
package c05.shapefact1;
import c05.badshape.*;
import java.util.*;
import com.bruceeckel.test.*;

abstract class Kuvio {
  public abstract void piirra();
  public abstract void pyyhi();

  public static Kuvio tehdas(String type) //Muodostetaan olioita listan mukaan
    throws BadShapeCreation {
    if(type.equals("Pallo")) return new Pallo();
    if(type.equals("Kantikas")) return new Kantikas();
    throw new BadShapeCreation(type);
  }
}

class Pallo extends Kuvio {
  Pallo() {}
  public void piirra() {
    System.out.println("Pallo-piirros");
  }
  public void pyyhi() {
    System.out.println("Pallo poissa");
  }
}

class Kantikas extends Kuvio {
  Kantikas() {}
  public void piirra() {
    System.out.println("Kantikas piirretty");
  }
  public void pyyhi() {
    System.out.println("Kantikas kuvio poissa");
  }
}

public class KuvioTehdas extends UnitTest {    //Muodostetaan kuviolista
  String kuviolista[] = { "Pallo", "Kantikas",
```

```
    "Kantikas", "Pallo", "Pallo", "Kantikas" };
  List kuviot = new ArrayList();
  public void test() {
    try {
      for(int i = 0; i < kuviolista.length; i++)
        kuviot.add(Kuvio.tehdas(kuviolista[i]));
    } catch(BadShapeCreation e) {
      e.printStackTrace(System.err);
      assert(false);
    }
    Iterator i = kuviot.iterator();
    while(i.hasNext()) {
      Kuvio s = (Kuvio)i.next();              //Olioille pyyntö piirtää itsensä
      s.piirra();
      s.pyyhi();
    }
  }
  public static void main(String args[]) {
    new KuvioTehdas().test();
  }
}
```

3.1.3 Ominaisuuksia

Tehdasmetodia käyttämällä vältytään sitomasta sovellusriippuvaisia luokkia koodiin. Koodi käsittelee vain Product-rajapintaa, joten siihen voidaan liittää mikä tahansa ConcreteProduct-luokka. Tehdasmetodin haittapuolena on aliluokkien määrän kasvaminen, sillä aina kun lisätään uusi ConcreteProduct-luokka, joudutaan perimään uusi aliluokka Creator-luokasta.

Tehdasmetodi tarjoaa aliluokille koukun, eli aliluokat voivat lisätä koukkukohtiin oman laajennetun version oliostaan. Olioiden luominen luokan sisällä tehdasmetodia käyttämällä on joustavampaa kuin olioiden luominen suoraan. Tehdasmetodi-suunnittelumalli myös yhdistää rinnakkaiset luokkahierarkiat, koska sitä voidaan käyttää muutenkin kuin Creator-luokan kutsumana. Rinnakkainen luokkahierarkia syntyy, kun luokka delegoi osan vastuistaan toiselle erilliselle luokalle.

3.2 Abstract Factory

Abstrakti tehdas-suunnittelumalli tunnetaan myös nimellä *Kit*. Abstraktin tehtaan tarkoituksena on tarjota rajapinta, jolla luodaan toisiinsa liittyvien olioiden muodostamia olioperheitä. Abstrakti tehdas ei määrittele konkreettisia luokkia. Abstrakti tehdas toimii siis vastaa-

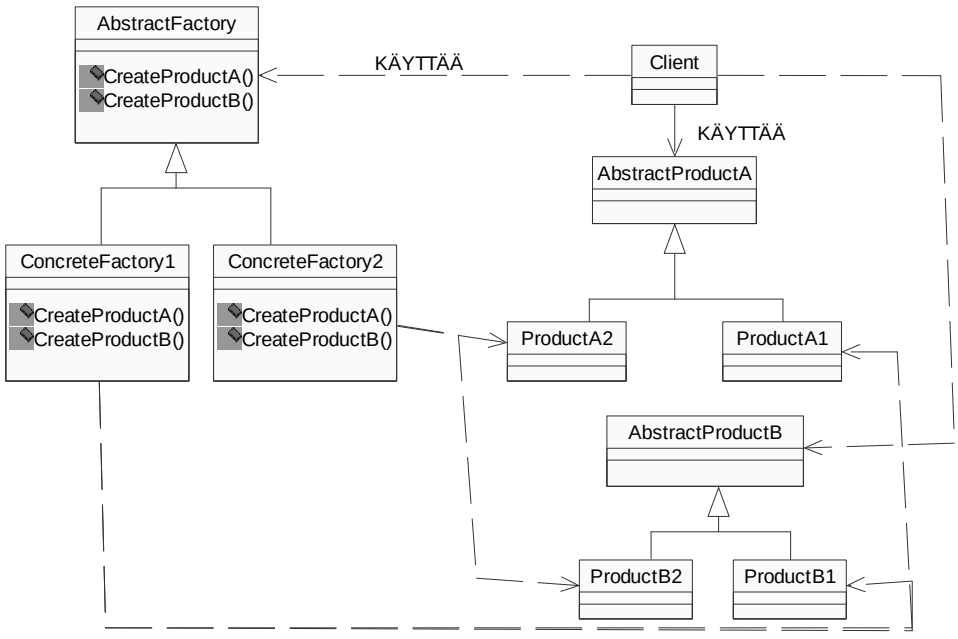
valla tavalla kuin tehdasmetodi, mutta yhden olion luomisen sijasta luodaankin useita toisistaan riippuvia olioita.

3.2.1 Soveltuvuus

Abstrakti tehdas-suunnittelumalli soveltuu käytettäväksi seuraavanlaisissa tilanteissa:

- Kun halutaan, että sovellus on riippumaton siitä, miten tuotteet luodaan, koostetaan ja esitetään
- Käytössä on useita tuoteperheitä ja halutaan koota sovellus käyttämään vain yhtä niistä.
- Perheeseen kuuluvia tuoteolioita on tarkoitus käyttää ja yhdistellä vain keskenään.
- Halutaan luoda tuotteista luokkakirjasto, jonka toteutustapaa ei haluta paljastaa.

3.2.2 Toteutus



Kuva 6: Abstrakti tehdas -mallin rakenne.

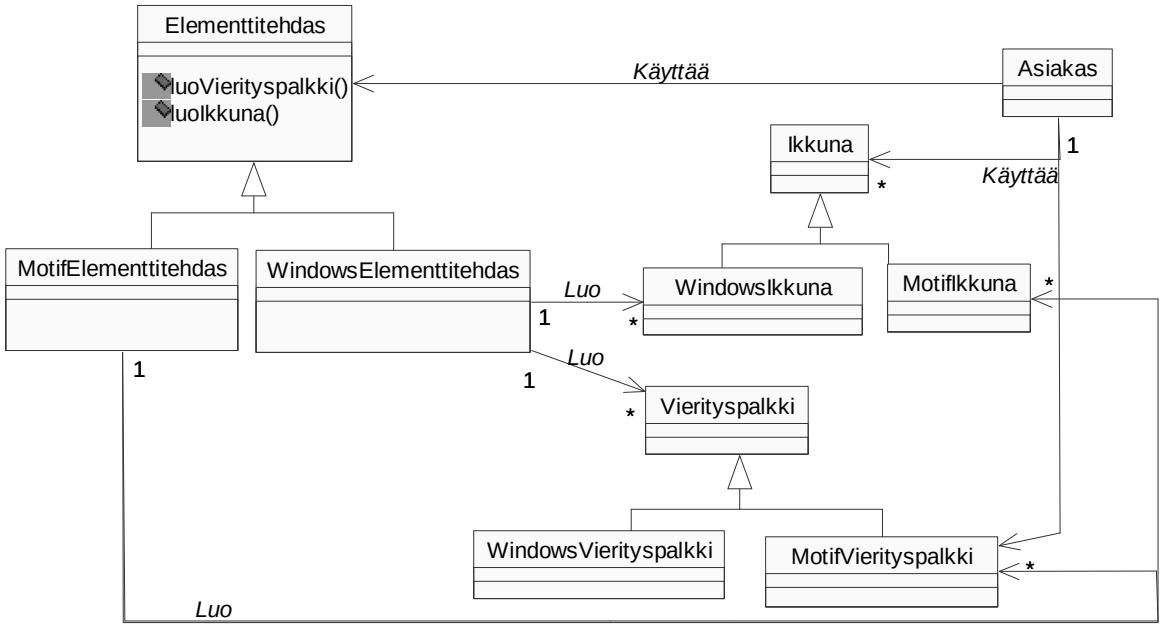
Abstrakti tehdas suunnittelumalliin kuuluvat seuraavat luokat (kuva 6):

- **AbstractFactory:** Määrittelee abstraktien tuoteolioiden luontioperaatioiden kutsumuodot.

- **ConcreteFactory:** Toteuttaa konkreettisten tuoteolioiden luontioperaatiot.
- **AbstractProduct:** Määrittelee tietyntyyppisen tuotteen rajapinnan.
- **ConcreteProduct:** Kuvaa sellaisen tuoteolion, jonka konkreettinen tehdas luo ja joka toteuttaa AbstractProduct-rajapinnan.
- **Client:** käyttää AbstractFactory- ja AbstractProduct -luokkien määrittelemiä rajapintoja.

Abstraktilla tehtaalla on useita erilaisia toteutustekniikoita. Yhtenä mahdollisuutena on toteuttaa tehdas singleton-suunnittelumallia hyväksikäyttäen. Tämä toteutustapa soveltuu parhaiten silloin, kun sovellus tarvitsee vain yhden ConcreteFactory-ilmentymän tuoteperhettä kohden. Toinen mahdollisuus on käyttää tehdasmetodia. Tällöin abstrakti tehdas määrittelee ainoastaan rajapinnan tuotteiden luomista varten. Jokaiselle tuoteperheelle määritellään oma tehdasmetodi. Kolmas vaihtoehtoinen tapa on käyttää prototyyppi-mallia, joka soveltuu sellaisiin tilanteisiin, joissa tuoteperheitä on paljon.

3.2.3 Esimerkkejä



Kuva 7: Ikkunaesimerkki (soveltaen [GaH00]).

Tarkastellaan esimerkkinä käyttöliittymän tekemiseen tarvittavaa välineistöä, eräänlaista "käyttöliittymätyökalupakkia", jonka avulla voidaan luoda ympäristökohtaisia (tässä Modif

ja Windows) käyttöliittymäelementtejä (esim. painikkeet ja ikkunat). Nämä elementit ovat molemmissa standardeissa toisiaan vastaavia, mutta voivat erota toisistaan ulkoasultaan ja käyttäytymiseltään. Määrittelyjä ei kannata koodata sovellukseen, jos halutaan säilyttää käyttöliittymämäärittelyn joustavuus muutostilanteissa.
Abstrakti tehdas -suunnittelumallin tarjoama ratkaisu on seuraavanlainen (kuva 7): Koska käyttöliittymäkomponentteja on pystyttävä vaihtamaan joustavasti, määritellään erillinen abstrakti rakentajaluokka (Elementtitehdas), jonka avulla voidaan luoda kunkin käyttöliittymätyypin tarvitsema elementtikokoelma. Ympäristökohtaiset elementit saadaan toteutettua määrittelemällä jokaiselle elementille aliluokka, joka tarvittaessa peritään. Abstraktin luokan (Elementtitehdas) rajapinnassa on operaatiot kaikille tarvittaville elementeille. Asiakas kommunikoi ainoastaan abstraktin luokan rajapinnassa olevien operaatioiden kautta, tuntematta taustalla olevia toteutuksesta vastaavia aliluokkia.

Alla olevassa esimerkissä **Pelaaja**-oliot ovat vuorovaikutuksessa **Este**-olioiden kanssa. Pelaaja- ja Este- oliot ovat erilaisia riippuen siitä, mitä peliä ollaan pelaamassa. Peli, jota pelataan, määritellään valitsemalla tietty **PeliElementtiTehdas.** Tämän jälkeen **PeliAlue** kontrolloi pelaamista ja pelin asetuksia.

```
 interface Este {
  void action();
}

interface Pelaaja {
  void interactWith(Este o);
}

class Kissanpentu implements Pelaaja {  //Lasten peliä varten kissanpentu
  public void interactWith(Este ob) {
    System.out.print("Kissa kohtasi esteen, sen on ratkaistava ");
    ob.action();
  }
}

class KungFuGuy implements Pelaaja { //Aikuisten peliä varten KugFu
  public void interactWith(Este ob) {
    System.out.print("KungFuGuy joutuu taistelemaan ");
    ob.action();
  }
}

class Arvoitus implements Este {
  public void action() {
    System.out.println("Arvoitus");
  }
}
```

```
class PahaAse implements Este {
  public void action() {
    System.out.println("Pahaa asetta vastaan");
  }
}

// Abstrakti tehdas (Abstract Factory)
//  Määritellään  rajapinnassa  metodien  otsikot,  jotka  saavat  toteutuksensa
ali//luokissa.

interface PeliElementtiTehdas {
  Pelaaja makePlayer();
  Este makeObstacle();
}

// Konkreettiset "tehtaat"
// Määritellään molemmille aliluokille erilainen peliväline ja este.

class PennutJaArvoitukset
implements PeliElementtiTehdas {
  public Pelaaja makePlayer() {
    return new Kissanpentu();
  }
  public Este makeObstacle() {
    return new Arvoitus();
  }
}

class TapaJaSilvo
implements PeliElementtiTehdas {
  public Pelaaja makePlayer() {
    return new KungFuGuy();
  }
  public Este makeObstacle() {
    return new PahaAse();
  }
}

class PeliAlue {
  private PeliElementtiTehdas gef;
  private Pelaaja p;
  private Este ob;
  public PeliAlue (
    PeliElementtiTehdas tehdas) {
    gef = tehdas;
    p = tehdas.makePlayer();
    ob = tehdas.makeObstacle();
  }
  public void play() { p.interactWith(ob); }
}

public class Pelit {
  PeliElementtiTehdas
    kp = new PennutJaArvoitukset(),
    kd = new TapaJaSilvo();
  PeliAlue
    g1 = new PeliAlue(kp),
    g2 = new PeliAlue(kd);

  public void test1() { g1.play(); }
  public void test2() { g2.play(); }
```

```
  public static void main(String args[]) {
    Pelit g = new Pelit();
    g.test1();
    g.test2();
  }
}
```

3.2.4 Ominaisuuksia

Abstraktin tehtaan avulla voidaan kontrolloida, mitä luokkia sovellus luo, eli näin eristetään konkreettiset luokat sovelluksesta. Sovellus käyttää tuoteilmentymiä rajapintojen kautta. Konkreettisen tehtaan luokka esiintyy sovelluksessa vain silloin kuin se luodaan. Näin ollen sovellus voi vaihtaa tuoteperhettä vaihtamalla konkreettisen tehtaan toiseksi, jolloin koko tuoteperhe vaihtuu toiseksi.

Abstrakti tehdas valvoo, että sovellus käyttää oliota vain yhdestä tuoteperheestä kerrallaan, koska tuoteperheiden tuotteiden on tarkoitus toimia yhdessä. Uudenlaisten tuotteiden luominen vaatii rajapinnan laajentamista, koska abstrakti tehdas kiinnittää tuotejoukon, joka voidaan luoda. Näin ollen abstraktin tehtaan laajentaminen siten, että se tukisi uudenlaisia tuotteita, on vaikeaa.

3.3 Singleton

Singleton (ainokainen) -suunnittelumalli on malleista ehkä helpoin ja yksinkertaisin. Sen tarkoituksena on varmistaa, että luokasta luodaan vain yksi ilmentymä, joka on saavutettavissa yhdestä tietystä paikasta ja jota voidaan periä. On olemassa sellaisia luokkia, joista voidaan tarjota vain yksi ilmentymä. Järjestelmässä voi esimerkiksi olla useita kirjoittimia, mutta kaikki käyttävät samaa tulostusjonoa. Mallia käytetään myös Javan luokkakirjastoissa. Olioon päästään käsiksi vain globaalin muuttujan kautta, mutta tämä ei vielä yksin pysty estämään useampien ilmentymien luontia. Sigleton-mallissa ongelma ratkaistaan antamalla luokan itse huolehtia siitä, että luokasta ei luoda useampia ilmentymiä. Muut sovellukset käyttävät sigleton-luokkaa vain tarjotun rajapinnan kautta.

3.3.1 Soveltuvuus

Singleton-malli soveltuu käytettäväksi silloin, kun luokasta halutaan luoda vain yksi ilmentymä, joka on muiden sovellusten saatavilla. Malli soveltuu käyttöön myös silloin, kun il-

mentymää on voitava laajentaa aliluokkien avulla ja sovellusten on kyettävä käyttämään tätä laajennettua ilmentymää koodia muuttamatta.

3.3.2 Toteutus

Singleton-mallin tarkoituksena on siis varmistaa, että luokasta ei luoda kuin yksi ilmentymä. Toteutuksessa on tärkeää suojata kaikki konstruktorit määrittelemällä ne yksityisiksi (private). On myös tärkeää määritellä ainakin yksi konstruktori, ettei kääntäjä tekisi sellaista automaattisesti.

```
final class Singleton {
  private static Singleton s = new Singleton(47);
  private int i;
  private Singleton(int x) { i = x; }
  public static Singleton getReference() {
    return s;
  }
  public int getValue() { return i; }
  public void setValue(int x) { i = x; }
}
public class SingletonPattern {
  public static void main(String[] args) {
    Singleton s = Singleton.getReference();
    System.out.println(s.getValue());
    Singleton s2 = Singleton.getReference();
    s2.setValue(9);
    System.out.println(s.getValue());
    try {
      // Can't do this: compile-time error.
      // Singleton s3 = (Singleton)s2.clone();
    } catch(Exception e) {
      e.printStackTrace(System.err);
    }
  }
} ///:~
```

3.3.3 Ominaisuuksia

Singleton-suunnittelumallilla on seuraavanlaisia ominaisuuksia:

- Singleton-luokka kapseloi ainoan ilmentymänsä, joten se pystyy kontrolloimaan sitä, milloin ja miten sovellukset käyttävät tätä ilmentymää.
- Malli pienentää nimiavaruutta, koska sitä ei tuhlata globaaleihin muuttujiin.
- Operaatioita ja esitysmuotoa voidaan jalostaa, sillä Sigleton-luokasta voidaan luoda aliluokkia ja sovellukseen voidaan kiinnittää aliluokkia myös ajoaikana.

- Ilmentymien lukumäärää voidaan myös vaihdella, sillä mallin avulla on helppo kontrolloida tilanteita, joissa halutaan käyttää useampia ilmentymiä. Lukumäärärajoitus voidaan sijoittaa singleton-ilmentymän käyttöä valvovaan operaatioon.
- Ratkaisu on joustavampi kuin luokkaoperaation käyttö.

4 RAKENNEMALLIT

Rakennemalleja käytetään apuna, kun muodostetaan jo olemassa olevista luokka- ja oliorakenteista suurempia, toiminnaltaan monipuolisempia ja joustavampia rakenteita. Rakennemallien käyttäminen mahdollistaa uusien ominaisuuksien lisäämisen helpommin ja myös tulevaisuudessa toteutettavat laajennokset on mahdollista tehdä vähällä vaivalla. Kuten muutkin suunnittelumallit (luonti- ja toiminnallisetmallit), myös rakennemallit jakaantuvat soveltamiskohteensa mukaan luokka- ja oliomalleihin. Rakenteelliset luokkamallit keskittyvät luokkien välisten riippuvuussuhteiden muokkaamiseen esim. perinnän avulla, kun taas oliomallit muodostavat uusia ominaisuuksia uusien olioiden avulla.

4.1 Proxy

Proxyn suomenkielinen nimi on *Edustaja*. Proxy-suunnittelumallin tarkoituksena on luoda yhteys komponentin sijasta komponentin edustajaan. Proxy-mallin avulla voidaan lisätä tehokkuutta, parantaa palvelujen saatavuutta ja tehostaa suojausta. Malli tarjoaa ratkaisun ongelmaan, jossa asiakkaan tarvitsee päästä käsiksi toisen komponentin tarjoamiin palveluihin ja suora yhteyskin olisi teknisesti mahdollista toteuttaa, mutta se ei ole paras ratkaisumalli. Suora rajoittamaton yhteys komponenttiin voi olla epätarkoituksenmukaista ja tehotonta, eikä se välttämättä ole turvallistakaan.

4.1.1 Soveltuvuus

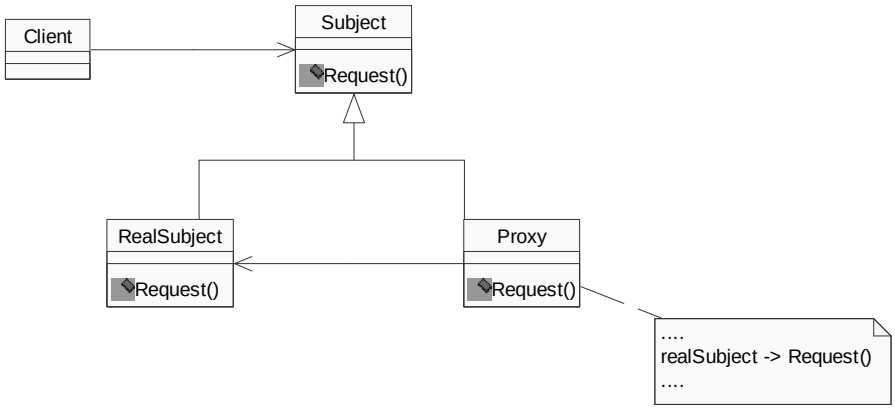
Proxy on käyttökelpoinen ratkaisumalli, kun tarvitaan ratkaisu seuraavanlaisten voimien tasapainottamiseen:

- Komponenttiin pääsyn tulee olla ajoaikaisesti tehokasta, edullista ja turvallista sekä asiakkaalle että komponentille.

- Komponenttiin pääsyn tulee olla asiakkaalle läpinäkyvää ja yksinkertaista. Asiakas ei kuitenkaan muuttaa kutsukäytäntöä tai syntaksia. Täydellisen läpinäkyvyyden avulla voidaan peittää palveluiden väliset erot.

Yleisesti ottaen proxyt ovat käyttökelpoisia aina, kun tarvitaan kehittyneempi tapa viitata olioon kuin pelkkä yksinkertainen viite tai osoitin.

4.1.2 Rakenne



Kuva 8: Proxy-suunnittelumallin rakenne.

Proxy-malliin osallistujat:

- **Subject:** Määrittelee yleiset rajapinnat RealSubject:lle ja Proxylle.
- **Proxy:** Pitää yllä viittausta, joka sallii Proxyn päästä käsiksi todelliseen kohteeseen (real subject). Tarjoaa rajapinnan, joka on identtinen Subject:n kanssa, jotta Proxy voi edustaa todellista kohdetta (real subject). Kontrolloi todelliseen kohteeseen pääsyä ja voi olla vastuussa myös sen luomisesta ja poistamisesta. Muut vastuut riippuvat Proxyn tyypistä (tehtävästä).
    - **RealSubject:** Määrittelee todellisen olion, jota Proxy edustaa.

4.1.3 Toteutusvaihtoehdot

Proxy-mallilla on erilaisia toteutusvaihtoehtoja. Seuraavissa kappaleissa kuvaus kustakin vaihtoehdosta ja siitä, millaisiin tilanteisiin kukin niistä parhaiten sopii.

4.1.3.1 Remote Proxy

Sopii tilanteisiin, joissa etäkomponenttien asiakkaat täytyy suojata verkon osoitteilta ja prosessien välisiltä kommunikointiprotokollilta. Edustaa siis toisessa nimiavaruudessa olevaa oliota. *Remote Proxyt* ovat vastuussa koodin muuntamisesta pyynnöksi ja sen muuttujiksi sekä pyynnön lähettämisestä todelliselle subjektille (real subject) tai toiselle proxylle eri nimiavaruudessa. Jokaista subjektia kohden on oma proxy.

4.1.3.2 Protection Proxy

Sopii käytettäväksi tilanteissa, joissa komponentit täytyy suojata tunnistamattomilta pääsyiltä (muistissa käynneiltä) ja kun olioilla pitäisi olla erilaisia oikeuksia päästä käsiksi todelliseen olioon. *Protection Proxy* siis valvoo yhteyttä alkuperäiseen olioon tarkistamalla jokaisen asiakkaan pääsyoikeudet. Helpoiten tämä voidaan toteuttaa käyttämällä pääsytarkistus -mekanismeja, joita laitealusta tarjoaa.

Seuraavassa esimerkki Protection Proxy -mallista. Esimerkissä todellinen olio tallentaa salaisen sanan. Ainoastaan ne asiakkaat, jotka tietävät tietyn salasanan pääsevät käsiksi tähän sanaan. Todellinen olio on suojattu proxylla, joka tietää salasanan. Jos asiakas tahtoo saada selville salaisen sanan, proxy pyytää ensin asiakasta todistamaan aitoutensa. Jos asiakas antaa oikean tiedon proxylle, se kutsuu todellista oliota ja välittää salaisen sanan asiakkaalle.

```
import java.io.*;

//Asiakkaalle näkyvä rajapinta

abstract class Safe{
//Kysyy salaisuutta
           public abstract String GetSecret();
}

//Turvallisuuden toteuttaminen

class RealSafe extends Safe{

           public RealSafe(){
                      System.out.println("RealSafe.RealSafe()");
           }
//Lukee salaisuuden

           public String GetSecret(){
                      System.out.println("RealSafe.GetSecret()");
```

```
                    return new String("Secret");
          }
}

//Protection proxy turvallisuuden takaamiseksi

class ProtectionProxy extends Safe{
//Salasana
          String Password=null;
//Olioviite
          RealSafe realobj=null;
//Käyttöönottovaihe
          public ProtectionProxy(String pwd){
                    System.out.println("ProtectionProxy.ProtectionProxy()");
                    Password=pwd;
                    realobj=new RealSafe();
          }
//Aitouden todistaminen, kysytään salasanaa

          public String GetSecret(){
                    System.out.println("ProtectionProxy.GetSecret()");
                    System.out.print("Password: ");
          BufferedReader inp=new BufferedReader(new InputStream
                                                  Reader(System.in));
                    String tmp=null;
                    try {
                              tmp=inp.readLine();
                    }catch(IOException e){
                              System.out.print("---"+e.toString());}

                    if(tmp.equals(Password))
                              return realobj.GetSecret();
                    System.out.println("Illegal password!");
                    return new String("");
          }
}

//Testiasiakas Protection Proxylle

public class protection{
          public static void main(String args[]){
                    Safe s=new ProtectionProxy("qwer");
                    System.out.println("main received: "+s.GetSecret());

                    System.out.println("main received: "+s.GetSecret());

          }
}
```

4.1.3.3 Cache Proxy

Tarjoaa väliaikaisen varaston kalliiden kohdeoperaatioiden tuloksille siten, että useat asiakkaat voivat jakaa tulokset. Toteutettaessa *Cache Proxy*, laajennetaan proxya tilapäisesti säilytettävien tulosten data-alueella. Välimuistin ylläpitämiseen ja virkistämiseen on kehitettävä oma strategiansa.

Esimerkiksi sellaiseen tilanteeseen, jossa välimuisti on täynnä ja täytyy vapauttaa tilaa uusia pääsyjä varten, on olemassa useita erilaisia strategioita. Pääsyt voidaan esimerkiksi toteuttaa käyttämällä "siirrä eteen" -strategiaa, jolloin pääsyt (merkinnät) voidaan poistaa listan lopusta. Vaihtoehtoisesti voidaan käyttää myös "läpikirjoitus" -strategiaa, jota käytetään mikroprosessorin välimuistin suunnittelussa. Miten tahansa alkuperäistä oliota on muutettu, sen kopiot muutetaan samalla tavalla. Tämä tulee kuitenkin monimutkaiseksi kun kopioiden määrä kasvaa tai kun kopion ovat etäällä verrattuna yksinkertaiseen mikroprosessorin välimuistiin. Jos asiakas hyväksyy lievästi vanhentunutta tietoa, voidaan yksilölliset välimuistimerkinnät leimata vanhentumispäivämäärällä. Esimerkkinä tästä strategiasta ovat www-selaimet.

4.1.3.4 Virtual Proxy

Kun komponentin lataaminen tai prosessointi kokonaan on keskeistä, vaikka osakin komponentin informaatiosta riittäisi, luo oliot vasta kun niitä todella tarvitaan. Oliot ovat yleensä monimutkaisia.

Alla olevassa esimerkissä FileHandler on todellinen olio (Subject). Sen konstruktori lataa tiedoston muistiin. Pääluokan ja metodien avulla voidaan kysyä kyseisen tiedoston nimeä ja sisältöä. Kun asiakas muodostaa todellisen olion, se antaa tiedoston nimen. Siten proxy voi palauttaa sen lataamatta tiedostoa. Kun asiakas ensin kysyy tiedoston sisältöä, proxyn täytyy muodostaa FileHandler-olio ja pyytää siltä tiedoston sisältöä.

```
import java.io.*;
// Rajapinta, joka tarjotaan Clientille

class FileHandler{
        String Filename=null;

//Alustetaan tiedoston nimi
        public FileHandler(String fn){
                System.out.println("FileHandler.Filehandler()");
                Filename=fn;
        }

//Palauttaa tiedoston nimen
        public String GetFilename(){
                System.out.println("FileHandler.GetFilename()");
                return Filename;
        }
```

```
//Palauttaa tiedoston sisällön
        public byte[] GetFile(){return null;};
}

//Tiedoston käsittelijän alustaminen
class RealFileHandler extends FileHandler{
        FileInputStream FileS=null;

//Alustaa tiedoston käsittelijän tiedoston nimellä
        public RealFileHandler(String fn){
                super(fn);
                System.out.println("RealFileHandler.RealFileHandler()");
                try{
                        FileS=new FileInputStream(fn);
                }catch(FileNotFoundException e){}
        }

//Lukee tiedoston muistista
        public byte[] GetFile(){
                System.out.println("RealFileHandler.GetFile()");
                byte tmp[]=new byte[200];
                try{
                                FileS.read(tmp,0,200);
                }catch(IOException e){}
                return tmp;
        }
}

//Virtual proxy tiedostonkäsittelijälle
class VirtualProxy extends FileHandler{
//Viitattu tiedostonkäsittelijä-olio
        RealFileHandler realobj=null;

        public VirtualProxy(String fn){  //Alustus
                super(fn);
                System.out.println("VirtualProxy.VirtualProxy()");
        }

//Tiedoston lataaminen, luo todellisen tiedostonkäsittelijän
//ja pyytää sitä lataamaan tiedoston
        public byte[] GetFile(){
                System.out.println("VirtualProxy.GetFile()");
                if(realobj==null)
                        realobj=new RealFileHandler(Filename);
                return realobj.GetFile();
        }
}

//Testi-client Virtual proxylle
public class Virtual{
        public static void main(String[] args){
/*              if(args.size==0){
                        System.out.println("usage: virtual filename.txt");
                        System.exit(0);
                }
*/              FileHandler fh=new VirtualProxy(args[0]);
                System.out.println("Filename: "+fh.GetFilename());
                System.out.println("Contents (first 200 byte):"+new
String(fh.GetFile())); }
}
```

4.1.3.5 Counting Proxy

Käytetään tilanteissa, joissa komponenttien tuhoaminen vahingossa on voitava estää tai kun halutaan kerätä käyttötilastoja. *Counting Proxyn* avulla voidaan myös toteuttaa viittauslaskenta, jolla voidaan automaattisesti tuhota vanhentuneita kohteita. Tällöin Counting Proxy pitää yllä viittausten määrää, jotka viittaavat todellisiin olioihin ja tuhoaa olion, kun tämän arvo saavuttaa nollan.

4.1.3.6 Synchronization Proxy

Synchronization Proxya kannattaa käyttää, kun lukuisat samanaikaiset pääsyt komponenttiin täytyy synkronoida (tahdistaa). Jos on tärkeää, että vain yksi asiakas tai määrätty määrä asiakkaita voi kerrallaan merkata todellisen olion, välipalvelin voi toteuttaa keskinäisen poissulkemisen opastimen avulla. Vaihtoehtoisesti se voi käyttää mitä tahansa tahdistuksen keinoja, joita käyttöjärjestelmä tarjoaa. Joskus saatetaan myös erotella pääsyt lukemisen ja kirjoittamisen välillä. Tällöin voidaan esimerkiksi sallia tietty määrä lukuja silloin, kun yhtään kirjoitusoperaatiota ei ole aktiivisena tai jonottamassa.

4.1.3.7 Firewall Proxy

Soveltuu käytettäväksi, kun paikalliset asiakkaat täytyy suojata ulkomaailmalta. *Firewall Proxy* sisältää verkkoratkaisun ja suojauskoodin pystyäkseen välittämään tietoa mahdollisesti epäedullisen ympäristön kanssa. Tavallisesti Firewall Proxy on toteutettu kuten daemon-prosessi palomuurikoneessa, joka viittaa proxy-palvelimeen. Kaikki asiakkaat, jotka välittävät pyyntöjä ulkopuoliseen maailmaan, viittaavat tähän palvelimeen. Välipalvelin työskentelee kulissien takana tarkastamalla ulosmenevät pyynnöt ja saapuvat vastaukset, sekä muuttamalla pyynnöt sisäiseen turvallisuuteen ja hyväksymis-menettelytapoihin sopiviksi. Se kieltää pääsyn, jos pyyntö ei mukaudu tällaiseen menettelytapaan, tai jos resurssit eivät ole riittävät.

4.1.3.8 Smart Reference Proxy

Kontrolloi oliota, pitää esimerkiksi lukua olioon viittauksista, lataa olion muistiin, kun siihen viitataan ensimmäisen kerran ja varmistaa, että alkuperäinen olio on lukittu, ennen kuin siihen viitataan, jotta kukaan ei pysty muuttamaan sitä.

Seuraavassa esimerkissä asiakkaat käyttävät *todellista oliota* (subject) ikään kuin se olisi niiden oma olio. Ne voivat luoda, käyttää ja poistaa olion. Jokaisella asiakkaalla on oma proxy, joka tekee täydentäviä tehtäviä. Proxy on asiakkaalle läpinäkyvä (tuntumaton). Proxyt kommunikoivat keskenään managerin välityksellä, joka laskee todellisen olion ilmentymien määrää ja tarvittaessa luo tai hävittää niitä. Tätä voidaan käyttää hyväksi kuormantasauksessa. *Viisas osoitin* (smart pointer) kasvattaa viittauslaskuria konstruktorissa ja vähentää sitä destruktorissa. Kun viittauslaskurin arvo tulee nollaksi, manageri poistaa olion muistista. Kun pyyntö saapuu ja todellista oliota ei ole olemassa, manageri luo sellaisen.

```
import java.util.*;

//Asiakkaan rajapinta
abstract class Subject{
//Näyttää "stringin"
        public abstract void Use(String s);
}

//Toteuttaa rajapinnan
class RealSubject extends Subject{
        public RealSubject(){
                System.out.println("RealSubject.RealSubject()");
        }
//Näyttää "stringin"
        public void Use(String s){
                System.out.println("RealSubject.Use() ");
                      System.out.println(s);
        }
}

//Subjektin "hallinnoija"
class Manager{
//Viittauslaskuri
           int Refs=0;
//Viitattu Subject-olio
           RealSubject Ref=null;

//Käyttöönotto
        public Manager(){
                System.out.println("Manager.Manager()");
        }
//Saadaan viittaus
           public RealSubject Get(){
```

```
                    Refs++;
               System.out.println("Manager.Get() Refs:"+Refs);
                    if(Ref==null){
                              Ref=new RealSubject();
                    }
               return Ref;
          }
//Hävitetään viittaus
          public void Drop(){
                    Refs--;
               System.out.println("Manager.Drop() Refs:"+Refs);
                    if(Refs==0){
                              Ref=null;
                         System.out.println("Manager: killing realsubject");

                    }
          }
}

//Smart reference (viisas viittaus) Subjectiin
class Proxy extends Subject{
//subjectin hallinnoija
          Manager man=null;
//viitattu Subject-olio
        RealSubject realobj=null;
//Käyttöönotto
        public Proxy(Manager m){
                System.out.println("Proxy.Proxy()");
                    man=m;
                    realobj=man.Get();
        }
//Käytetään todellista oliota (subjektia)
          public void Use(String s){
               System.out.println("Proxy.Use()");
                    realobj.Use(s);
          }
//Tapetaan proxy
          public void Delete(){
               System.out.println("Proxy.Delete()");
                    man.Drop();
          }
}

//Asiakas, joka käyttää proxya
class Client{
          Proxy p=null;
          Manager man=null;
          String str=null;
          public Client(String s, Manager m){
               System.out.println(s+"Client.Clinet()");
                    man=m;
                    str=s;
          }
          public void Run(){
               System.out.println(str+"Client.Run()");
                    p=new Proxy(man);
                    p.Use(str+"Running...\n");
          }
          public void Stop(){
               System.out.println(str+"Client.Stop()");
                    p.Delete();
```

```
        }
}

//Testisovellus
public class smart{
        public static void main(String args[]){
                Manager m=new Manager();
              Client c1=new Client("1",m);
                Client c2=new Client("2",m);
                Client c3=new Client("3",m);
                c1.Run();
                c2.Run();
                c2.Stop();
                c3.Run();
                c1.Stop();
                c3.Stop();
                c2.Run();
                c2.Stop();
        }
}
```

4.2 Adapter

Tunnetaan myös nimellä *Wrapper* (kääre). *Sovitin* (adapter) -suunnittelumallia käytetään, kun halutaan saada toimimaan yhdessä sellaiset luokat, joilla on yhteensopimattomat rajapinnat. Sovitin muuttaa luokan rajapinnan toiseksi rajapinnaksi, joka vastaa sovelluksen tarpeita. Yleensä ainoastaan toinen luokka haluaa käyttää toisen luokan tarjoamia palveluita, jolloin toinen luokista on palvelun tarjoaja ja toinen niiden käyttäjä. Tällöin puhutaan yksisuuntaisesta sovittimesta. Kaksisuuntainen sovitin puolestaan yhtenäistää luokkien väliset rajapinnat molempiin suuntiin.

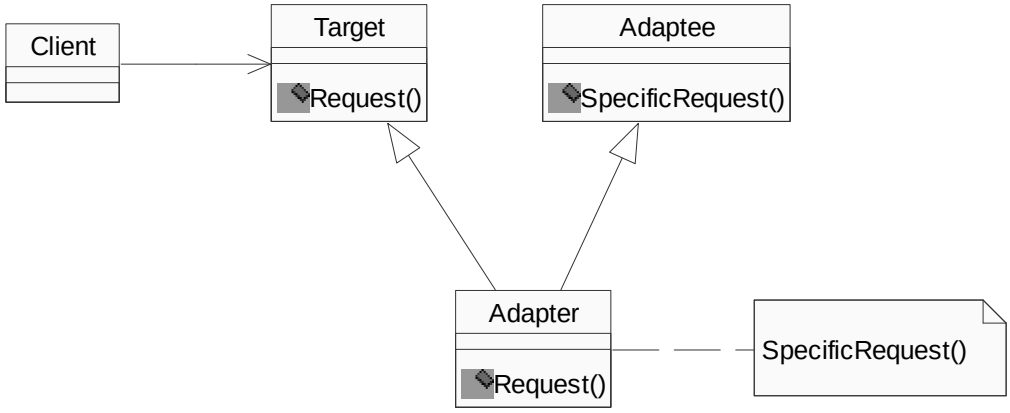
4.2.1 Soveltuvuus

Sovitin-suunnittelumallia käytetään seuraavanlaisissa tilanteissa:

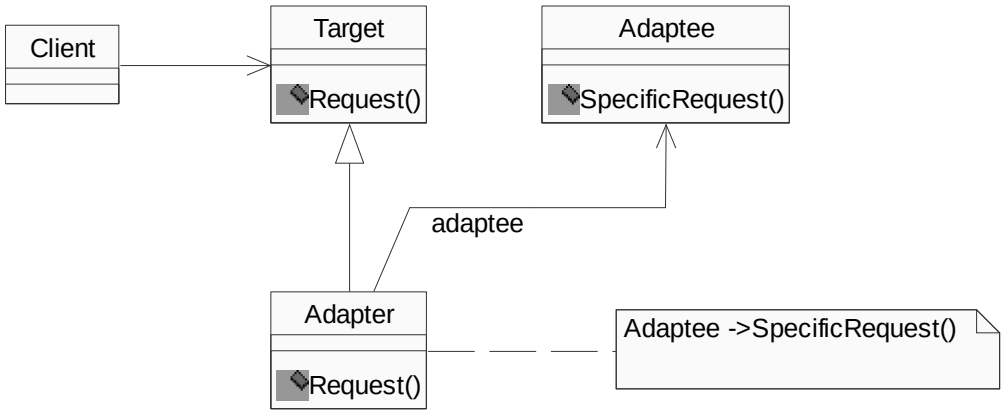
- Kun halutaan käyttää hyväksi olemassa olevaa luokkaa, mutta sen rajapinta ei ole sopiva.
- Tarvittavan luokan rajapintaa ei voida muuttaa, koska sen lähdekoodi ei ole saatavilla.
- Kun halutaan luoda uudelleenkäytettävä luokka, joka toimii yhdessä vielä määrittelemättömien luokkien kanssa, tai sellaisten luokkien kanssa, joilla on erilaiset rajapinnat.
- Oliosovitinta voidaan käyttää tilanteissa, joissa on käytettävä useita olemassa olevia aliluokkia, mutta ei ole käytännöllistä luoda jokaisesta omaa aliluokkaa rajapintojen yhdistämiseksi. Oliosovitin voi tehdä sovituksen yliluokkansa rajapinnalle.

4.2.2 Toteutus

Sovitustarpeet vaihtelevat suuresti. Yksinkertaisimmissa tapauksissa riittää, kun nimetään operaatiot uudelleen eri systeemeihin sopiviksi. Monimutkaisimmissa tapauksissa taas tarvitaan kokonaan uudenlaisten operaatioiden joukko. Toteutusta mietittäessä on myös ratkaistava, hoidetaanko rajapintojen kytkentä perivää sovitinta (luokkasovitin) vai ulkoista oliosovitinta käyttäen.



Kuva 9: Luokkasovitin.



Kuva 10: Oliosovitin.

Luokkasovittimen (kuva 9) ja *oliosovittimen* (kuva 10) ero on siinä, että luokkasovitin käyttää hyväkseen moniperintää rajapintojen yhdistämisessä, kun taas oliosovitin käyttää hyväkseen olioiden koostamista. Ulkoinen oliosovitin voidaan kytkeä suoritusaikaisesti. Koska luokkasovitin toteutetaan perimällä Adaptee-luokalta, ei Adaptee-luokan aliluokkien sovittaminen Target-luokan rajapintaan onnistu. Toisaalta Adaptee-luokan ominaisuuksien korvaaminen onnistuu, koska Adapter on Adaptee-luokan aliluokka. Oliosovitin sen sijaan pystyy sovittamaan myös Adaptee-luokan aliluokat Target-luokan rajapintaan, mutta Adaptee-luokan ominaisuuksia/funktioita on vaikeampi korvata.

Sovitin-malliin osallistujat:

- **Target:** Määrittelee sovelluskohtaisen rajapinnan, jota Client käyttää
- **Client:** Käyttää Target-luokan mukaista rajapintaa.
- **Adaptee:** Määrittelee olemassa olevan rajapinnan, joka tarvitsee sovitusta.
- **Adapter:** Sovittaa Adaptee-luokan rajapinnan Target-luokan rajapintaan.

Client kutsuu sovittimen (Adapter-luokan) operaatioita. Sovitin kutsuu puolestaan sovitettavan olion (Adaptee-luokan) operaatioita, jotka sitten toteuttavat pyynnön.

Seuraavassa esimerkissä kuvataan erilaisia tapoja toteuttaa sovittaminen.

```
class WhatIHave {
  public void g() {}
  public void h() {}
}

interface WhatIWant {
  void f();
}

class ProxyAdapter implements WhatIWant {
  WhatIHave whatIHave;
  public ProxyAdapter(WhatIHave wih) {
    whatIHave = wih;
  }
  public void f() { // Toteuttaa oman käyttäytymisensä käyttämällä WhatIHave-
rajapinnan
    whatIHave.g();  // "valmiita" metodeja
    System.out.println("metodissa f(), whatIhave.g()");
    whatIHave.h();
    System.out.println("metodissa f(), whatIhave.g()");
  }
}

class WhatIUse {
  public void op(WhatIWant wiw) {
```

```
            System.out.println("WhatIUse");
    wiw.f();
  }
}

// Tapa 2: rakennetaan sovitin op():lle:
class WhatIUse2 extends WhatIUse {          //WhatIUse-luokan aliluokka,
  public void op(WhatIHave wih) {           //Määritellään op() uudelleen.
            System.out.println("metodissa op()");
    new ProxyAdapter(wih).f();
  }
}

// Tapa 3: rakennetaan sovitin WhatIHave:lle:
class WhatIHave2 extends WhatIHave          //WhatIHave-luokan aliluokka
implements WhatIWant {                      //Toteuttaa WhatIWant-rajapinnan
  public void f() {
            System.out.println("Tapa 3: metodissa f()");
    g();
    h();
  }
}

// Tapa 4: käytetään hyväksi sisempää luokkaa:
class WhatIHave3 extends WhatIHave {        //WhatIHave-luokan aliluokka
  private class InnerAdapter implements WhatIWant{ //Luokka toteuttaa WhatIWant-
                                                    //rajapinnan
    public void f() {
      System.out.println("Tapa 4: metodissa f()");
      g();
      h();
    }
  }
  public WhatIWant whatIWant() {
    return new InnerAdapter();
  }
}

public class Adapter {                      //Adapter-luokka
  WhatIUse whatIUse = new WhatIUse();       //Luodaan tarvittavat oliot…
  WhatIHave whatIHave = new WhatIHave();
  WhatIWant adapt= new ProxyAdapter(whatIHave);
  WhatIUse2 whatIUse2 = new WhatIUse2();
  WhatIHave2 whatIHave2 = new WhatIHave2();
  WhatIHave3 whatIHave3 = new WhatIHave3();
  public void test() {
    System.out.println("Tapa 1:");       //.. ja testataan erilaiset tavat
                                         // toteuttaa sovittaminen
    whatIUse.op(adapt);
    // Tapa 2:
    System.out.println("Tapa 2:");
    whatIUse2.op(whatIHave);
    // Tapa 3:
    System.out.println("Tapa 3:");
    whatIUse.op(whatIHave2);
    // Tapa 4:
    System.out.println("Tapa 4:");
    whatIUse.op(whatIHave3.whatIWant());
  }
  public static void main(String args[]) {
    new Adapter().test();
```

```
  }
} ///:~
```

4.2.3 Ominaisuuksia

Luokka- ja oliosovittimilla on omat hyvät ja huonot puolensa.

- Luokkasovitin sovittaa Adaptee-rajapinnan Target-rajapintaan sitoutumalla konkreettiseen Adaptee-luokkaan. Tästä johtuen luokkasovitin ei sovellu tilanteisiin, joissa sekä luokka että kaikki sen aliluokat on sovitettava.
- Luokkasovitin sallii Adaptee-luokan operaatioiden korvaamisen, koska Adapter on Adaptee-luokan aliluokka.
- Luokkasovitin esittelee vain yhden uuden olion, eikä sovitettavan luokan käyttämiseen tarvita osoittimien uudelleenohjausta.
- Oliosovitin puolestaan mahdollistaa yhden sovittimen käytön monen sovitettavan luokan kanssa. Adapter-luokka voi lisätä toiminnallisuutta kaikkiin Adaptee-olioihin.
- Oliosovitin ei pysty helposti korvaamaan sovitettavan luokan operaatioita. Korvaus voidaan toteuttaa määrittelemällä Adaptee-aliluokka ja kiinnittämällä sovitin tähän aliluokkaan Adaptee-luokan sijasta.

4.3 Facade

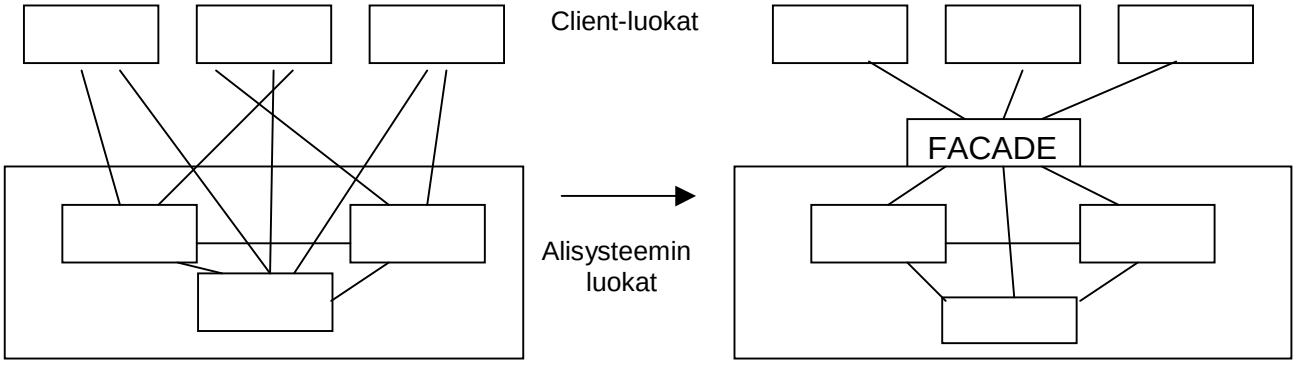
Julkisivu (Facade) -suunnittelumalli tarjoaa yhtenäisen rajapinnan alijärjestelmän rajapintajoukolle. Pilkkomalla systeemi pienempiin osiin, alisysteemeihin, voidaan vähentää systeemin monimutkaisuutta. Kun kullekin alisysteemille annetaan tarkkaan rajattu tehtävänsä, päästään siihen, että alisysteemien välillä on mahdollisimman vähän kommunikointia ja riippuvuutta. Tähän tarkoitukseen voidaan käyttää julkisivu-oliota, joka tarjoaa yhden yksinkertaisen rajapinnan alijärjestelmien yleisimmille palveluille. Toisin sanoen julkisivu-olio välittää itse eteenpäin kaikki alisysteemistä ulospäin lähtevät ja sinne tulevat pyynnöt. Kun keskitetään kaikki pyynnöt samaan yleiseen rajapintaan, yksinkertaistetaan alisysteemin eri palveluiden käyttöä. Yleinen rajapinta ei kuitenkaan estä käyttämästä alisysteemin tarjoamia alemman tason (vaikeammin käytettäviä) palveluja, joita jokin asiakasohjelma saattaa myös tarvita.

4.3.1 Soveltuvuus

Julkisivu-suunnittelumallia kannattaa käyttää seuraavanlaisissa tilanteissa:

- Halutaan tarjota yksinkertainen ja helppo rajapinta monimutkaisille alisysteemeille. Julkisivu tarjoaa yksinkertaisen oletusrajapinnan, joka sopii useimmille sovelluksille. Ne sovellukset, joita tämä rajapinta ei tyydytä, voivat kommunikoida suoraan alisysteemin kanssa ohi Julkisivu-mallin.
- Sovellusten ja toteutusluokkien välillä on paljon riippuvuuksia. Julkisivu-olion avulla voidaan irrottaa alisysteemit itse sovelluksesta ja toisista alisysteemeistä.
- Kun halutaan kerrostaa alisysteemejä. Julkisivu-oliota voidaan käyttää tällöin yhtenäisenä rajapintana kuhunkin kerrokseen. Tällä tavalla voidaan yksinkertaistaa monimutkaisen alisysteemin sisällä olevia riippuvuuksia.

4.3.2 Toteutus



Kuva 11: Julkisivu-mallin käyttöönotto.

Malliin osallistujat:

- **Facade (julkisivu):** Tietää, mitkä alisysteemin luokat ovat vastuussa mistäkin pyynnöstä. Delegoi sovellusten pyynnöt toisille alisysteemin osille ja tekee mahdollisesti sovitusta oman rajapintansa ja alisysteemin olion välillä.
- **Alisysteemin luokat:** Toteuttavat alisysteemin toiminnot ja käsittelevät julkisivu-olion pyynnöt, mutta eivät tunne julkisivu-oliota (eivät sisällä viittauksia siihen).

Seuraava esimerkki kuvaa teen keittämistä. Toteutuksessa on julkisivu-luokka Facade-CuppaMaker, joka hoitaa yhteydet muihin luokkiin.

```
//Julkisivu-luokka, hoitaa yhteydet muihin luokkiin

public class FacadeCuppaMaker {
   boolean teaBagIsSteeped;
   public FacadeCuppaMaker() {
       System.out.println("FacadeCuppaMaker valmiina tarjoamaan teekupposen!");
   }
   public FacadeTeaCup makeACuppa() {
       FacadeTeaCup cup = new FacadeTeaCup(); //Luodaan teekuppi
       FacadeTeaBag teaBag = new FacadeTeaBag(); //Luodaan teepussi
       FacadeWater water = new FacadeWater(); //Luodaan teevesi
       cup.addFacadeTeaBag(teaBag); //Lisätään teepussi kuppiin
       water.boilFacadeWater();  //Kiehutetaan vettä
       cup.addFacadeWater(water); //Lisätään vesi kuppiin
       cup.steepTeaBag(); //Liotetaan teepussia
       return cup;
   }
}
//Luokka, jota julkisivu-luokka kutsuu

public class FacadeTeaCup {
   boolean teaBagIsSteeped;
   FacadeWater facadeWater;
   FacadeTeaBag facadeTeaBag;

   public FacadeTeaCup() {
       setTeaBagIsSteeped(false); //Jos teepussi ei ole vielä kupissa
       System.out.println("Pitelet kaunista kuppia!");
   }

//Lisätään teepussi liukenemaan kuppiin
public void setTeaBagIsSteeped(boolean isTeaBagSteeped)
                                          {teaBagIsSteeped = isTeaBagSteeped;}
   public boolean getTeaBagIsSteeped() {return teaBagIsSteeped;}

   public void addFacadeTeaBag(FacadeTeaBag facadeTeaBagIn) {
       facadeTeaBag = facadeTeaBagIn;
       System.out.println("Teepussi on kupissa");
   }
   public void addFacadeWater(FacadeWater facadeWaterIn) {
       facadeWater = facadeWaterIn;
       System.out.println("Vesi on kupissa");
   }

//Tarkastetaan, onko teepussi kupissa vai ei.
public void steepTeaBag(){
       if ( (facadeTeaBag != null) && ((facadeWater != null) && (facadeWa
                                          ter.getWaterIsBoiling())) )
       {
          System.out.println("Tee on liukenemassa kupissa");
          setTeaBagIsSteeped(true);
       }
       else {
          System.out.println("Tee ei ole liukenemassa kupissa");
          setTeaBagIsSteeped(false);
       }
```

```
    }
    public String toString() {
            //Jos teepussi on liukenemassa kupissa…
        if (this.getTeaBagIsSteeped()) {return ("Maistuvaa juomaa teekupissa!");}
        else {
            //…muuten tarkastetaan onko kupissa vettä…
            String tempString = new String("Kupissa ");
            if (facadeWater != null) {
                 if (facadeWater.getWaterIsBoiling()) {
                     tempString = (tempString + "kiehuva vesi ");
                 }
                 else {
                     tempString = (tempString + "haalea vesi ");
                 }
            }
            else {
                 tempString = (tempString + "ei vettä ");
            }
            //…ja teepussi
            if (facadeTeaBag != null) {
                 tempString = (tempString + "ja teepussi");
            }
            else {
                 tempString = (tempString + "ja ei teepussia");
            }
            return tempString;
        }
    }
}
//Luokka, jota julkisivu-luokka kutsuu

public class FacadeWater {  //Onko vesi kiehumassa vai ei
    boolean waterIsBoiling;

    public FacadeWater() {
        setWaterIsBoiling(false); //Vesi ei ole kiehumassa
        System.out.println("Pitelet ihmeellistä vettä");
    }
    public void boilFacadeWater() {
        setWaterIsBoiling(true);  //Vesi on kiehumassa
        System.out.println("Vesi on kiehumassa");
    }
    public void setWaterIsBoiling(boolean isWaterBoiling)
                                              {waterIsBoiling = isWaterBoiling;}
    public boolean getWaterIsBoiling() {return waterIsBoiling;}
}
//Luokka, jota julkisivu-luokka kutsuu

public class FacadeTeaBag {
    public FacadeTeaBag(){
        System.out.println("pitelet ihanaa teepussia");
    }
}
//Julkisivun testiluokka
class TestFacade {
    public static void main(String[] args) {
        FacadeCuppaMaker cuppaMaker = new FacadeCuppaMaker();
        FacadeTeaCup teaCup = cuppaMaker.makeACuppa();
        System.out.println(teaCup);
    }
}
```

4.3.3 Ominaisuuksia

Julkisivu-mallilla on seuraavanlaisia ominaisuuksia:

- Malli eristää sovelluksen alijärjestelmän osista, joten se vähentää niiden olioiden määrää, joiden kanssa sovellus kommunikoi. Näin ollen alisysteemien käyttäminen on helpompaa ja suoraviivaisempaa.
- Saa aikaan heikon sidoksen alisysteemin ja sitä käyttävien sovellusten välille. Tämä mahdollistaa alisysteemin komponenttien muuttamisen tai jopa vaihtamisen ilman, että muutos vaikuttaa kutsuviin sovelluksiin.
- Helpottaa järjestelmien kerrostamista, sekä vähentää ja keskittää olioiden välisiä riippuvuuksia. Esimerkiksi laajoissa järjestelmissä voidaan vähentää aliluokkien muutoksista aiheutuvaa uudelleenkäännettävien osien määrää.
- Helpottaa järjestelmän siirtämistä uudelle alustalle, koska malli vähentää sen todennäköisyyttä, että yhden alisysteemin pystytys vaatisi kaikkien muiden kääntämistä.
- Ei estä sovelluksia käyttämästä alisysteemin luokkia tarvittaessa. Jos suora alisysteemin luokkien käyttäminen sallitaan, sovellus voi olla helppo- ja yleiskäyttöinen.

4.4 Bridge

Silta (Bridge) -suunnittelumallin tarkoituksena on erottaa rajapinta toteutuksesta niin, että kumpaakin voidaan muunnella toisistaan riippumatta. Kun abstraktilla kantaluokalla voi olla useita erilaisia toteutuksia, olisi helpointa toteuttaa ne käyttämällä apuna luokkien perintää. Abstrakti kantaluokka määrittelee tällöin rajapinnan abstraktioon ja konkreettiset aliluokat toteuttavat nämä abstraktiot haluamallaan tavalla. Perinnässä toteutus kuitenkin sitoutuu abstraktiin rajapintaan pysyvästi, jolloin rajapintaa ja sen toteutusta ei voida muuttaa toisistaan riippumattomasti. Paitsi että Silta-malli erottaa rajapinnan toteutuksesta, se mahdollistaa myös aivan uudenlaisen toteutuksen liittämisen järjestelmään ilman, että rajapintaan tarvitsee tehdä mitään muutoksia.

4.4.1 Soveltuvuus

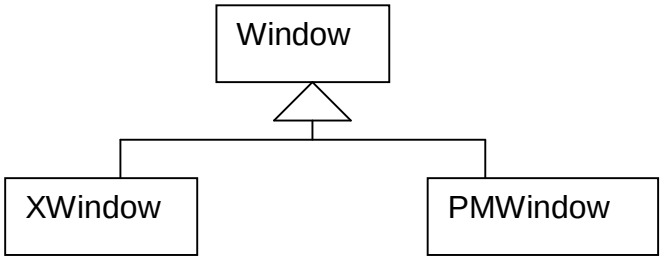
Silta-mallia kannattaa käyttää seuraavanlaisissa tilanteissa:

- Kun halutaan välttää rajapinnan ja toteutuksen välisen riippuvuuden sitominen esimerkiksi silloin, kun on tarvetta valita tai jopa vaihtaa toteutusta ohjelman ajon aikana.
- Kun sekä rajapinnan että toteutuksen tulisi olla laajennettavissa lisäämällä luokkahierarkiaan uusia aliluokkia. Tällaisissa tilanteissa Silta-malli siis mahdollistaa rajapinnan ja toteutuksen toisistaan riippumattoman muokkauksen ja laajentamisen.
- Kun toteutuksen muuttaminen ei saisi näkyä käyttäjälle.
- Jos luokkien lukumäärä kasvaa voimakkaasti.

4.4.2 Toteutus

Tarkastellaan seuraavanlaista tilannetta (kuva 9): *Window* on abstrakti kantaluokka, joka määrittelee rajapinnan Window-abstraktiolle. Kuvan mukaisilla määrittelyillä tulisi pystyä kirjoittamaan sekä XWindow että Presentation Manager -ympäristössä (IBM:n ikkunointijärjestelmä) toimivia sovellusohjelmia. Perinnän käyttämisessä tällaisissa tilanteissa on kaksi haittapuolta:

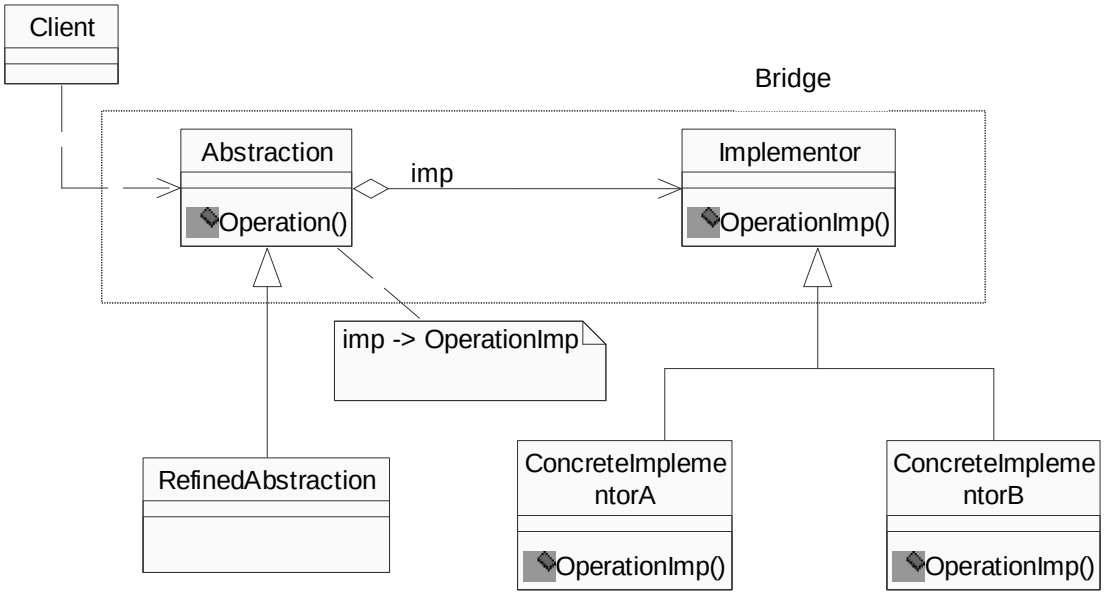
- Window-abstraktion laajentaminen uusilla ominaisuuksilla toimimaan molemmissa ympäristöissä on vaikeaa. Jokaista uutta ominaisuutta kohden on aina luotava kaksi uutta luokkaa, molemmille ympäristöille omansa.
- Perintä tekee koodista ympäristöriippuvaista.



Kuva 12: Ikkunat perintää hyväksikäyttäen.

Silta-malli pyrkii ratkaisemaan nämä kaksi ongelmaa sijoittamalla Windows-abstraktion ja sen toteutuksen kokonaan eri luokkahierarkioihin (kuva 12). Window-abstraktio ja siihen liittyvien ominaisuuksien rajapintamäärittelyt (*RefinedAbstraction*) ovat omassa luokkahierarkiassaan ja vastaavasti ympäristöstä riippuvat (esim. XWindow ja PM) omassaan *Implementor*-luokan aliluokkina. Kaikki Window-aliluokkien operaatiot toteutetaan abstraktin *Implementor*-luokan määrittelemien abstraktien operaatioiden (esim. OperationImp()

) avulla. Tällä tavalla pystytään erottamaan toisistaan Window-abstraktio ja mahdollisesti useat erilaiset ympäristöstä riippuvat toteutukset. Juuri tätä Window-luokan (kuvassa 13 *Abstraction*) ja Implementor-luokan välistä riippuvuutta kutsutaan sillaksi (bridge), koska se ikään kuin muodostaa sillan abstraktion ja sen toteutuksen välille mahdollistaen molempien muokkaamisen toisistaan riippumatta.



Kuva 13: Silta-mallin rakenne.

Silta-mallin osallistujat:

- **Abstraction:** Määrittelee abstraktion rajapinnan, joka näkyy Clientille (esim. Window).
- **RefinedAbstraction:** Laajentaa Abstraction-luokan määrittelemää rajapintaa uusien ominaisuuksien vaatimusten mukaiseksi.
- **Implementor:** Määrittelee rajapinnan toteutusluokille (ConcreteImplementor, esim. XWindow). Implementor-luokan rajapinnan ei tarvitse täysin vastata Abstraction-luokan rajapintaa, vaan yleensä Abstraction-luokka määrittelee korkeamman tason operaatiot.
- **ConcreteImplementor:** Toteuttaa Implementor-luokan määrittelemän rajapinnan mukaisen toteutuksen ja oman ympäristöstä riippuvaisen toimintansa (esim. XWindow).

Mikäli käytössä on vain yksi toteutus, abstraktia Implementor-luokkaa ei välttämättä tarvita. Tällöin Abstraction- ja Implementor- luokkien välillä on yksi-yhteen vastaavuus. Luok-

kien erottaminen toisistaan on tarpeellista, jos toteutukseen tehtävät muutokset eivät saa aiheuttaa sovellusten uudelleenkääntämistä.

Yleensä toteutuksen vaikein kohta on oikean ConcreteImplementorin valinta. Jos Abstraction tuntee kaikki ConcreteImplementor-luokat, se voi valita niistä yhden konstruktorissaan esimerkiksi parametrien kautta. Toinen mahdollisuus on valita aluksi yksi oletustoteutus ja muuttaa sitä myöhemmin, jos tulee tarvetta.
Seuraavassa ohjelmallinen esimerkki Silta-mallista.

```
//Abstraktin perusluokan määrittely.
public abstract class Soda
{
   SodaImp sodaImp;

   public void setSodaImp() {this.sodaImp = SodaImpSingleton.getTheSodaImp();}
   public SodaImp getSodaImp() {return this.sodaImp;}

   public abstract void pourSoda();
}
//Laajentaa Abstraktia, keskikokoinen soda
public class MediumSoda extends Soda
{
   public MediumSoda() {setSodaImp();}

   public void pourSoda() //Soodan kaataminen, määritellään yksi toteutus
   {
       SodaImp sodaImp = this.getSodaImp();
       for (int i = 0; i < 2; i++)
       {
           System.out.print("...glug...");
           sodaImp.pourSodaImp();
       }
       System.out.println(" ");
   }
}
//Laajentaa Abstraktia, isokokoinen sooda
public class SuperSizeSoda extends Soda
{
   public SuperSizeSoda() {setSodaImp();}

   public void pourSoda() //soodan kaataminen, määritellään erilainen toteutus
   {
       SodaImp sodaImp = this.getSodaImp();
       for (int i = 0; i < 5; i++)
       {
           System.out.print("...glug...");
           sodaImp.pourSodaImp();
       }
       System.out.println(" ");
   }
}
//Perusluokan toteutus
public abstract class SodaImp
{
```

```
   public abstract void pourSodaImp();
}
//Laajentaa toteutusta, kirsikkasooda.
public class CherrySodaImp extends SodaImp
{
   CherrySodaImp() {}

   public void pourSodaImp()
   {
       System.out.println("Yummy Cherry Soda!");
   }
}

//Laajentaa toteutusta, appelsiinisooda.
public class OrangeSodaImp extends SodaImp
{
   OrangeSodaImp() {}

   public void pourSodaImp()
   {
       System.out.println("Citrusy Orange Soda!");
   }
}
//Laajentaa toteutusta, greippisooda.
public class GrapeSodaImp extends SodaImp
{
   GrapeSodaImp() {}

   public void pourSodaImp()
   {
       System.out.println("Delicious Grape Soda!");
   }
}
//Pitää tiedon nykyisestä SodaImp-oliosta
public class SodaImpSingleton
{
    private static SodaImp sodaImp;

    public SodaImpSingleton(SodaImp sodaImpIn) {this.sodaImp = sodaImpIn;}

    public static SodaImp getTheSodaImp()
    {
        return sodaImp;
    }
}
//Testiluokka Silta-mallin testaamiseen
class TestBridge {
   public static void testCherryPlatform()
   {
 SodaImpSingleton sodaImpSingleton = new SodaImpSingleton(new CherrySodaImp());
       System.out.println("testing medium soda on the cherry platform");
       MediumSoda mediumSoda = new MediumSoda();
       mediumSoda.pourSoda();
       System.out.println("testing super size soda on the cherry platform");
       SuperSizeSoda superSizeSoda = new SuperSizeSoda();
       superSizeSoda.pourSoda();
   }

   public static void testGrapePlatform()
   {
  SodaImpSingleton sodaImpSingleton = new SodaImpSingleton(new GrapeSodaImp());
```

```
        System.out.println("testing medium soda on the grape platform");
        MediumSoda mediumSoda = new MediumSoda();
        mediumSoda.pourSoda();
        System.out.println("testing super size soda on the grape platform");
        SuperSizeSoda superSizeSoda = new SuperSizeSoda();
        superSizeSoda.pourSoda();
    }

    public static void testOrangePlatform(){
SodaImpSingleton sodaImpSingleton = new SodaImpSingleton(new OrangeSodaImp());
        System.out.println("testing medium soda on the orange platform");
        MediumSoda mediumSoda = new MediumSoda();
        mediumSoda.pourSoda();
        System.out.println("testing super size soda on the orange platform");
        SuperSizeSoda superSizeSoda = new SuperSizeSoda();
        superSizeSoda.pourSoda();
    }

    public static void main(String[] args)
    {
         testCherryPlatform();
         testGrapePlatform();
         testOrangePlatform();
    }
}
```

4.4.3 Ominaisuuksia

Silta-mallin ominaisuuksia:

- Toteutus ei sitoudu pysyvästi rajapintaan, vaan toteutus voidaan konfiguroida tai jopa vaihtaa ajon aikana.
- Toteutuksen muuttaminen ei vaadi abstraktion eikä asiakaskoodin uudelleenkääntämistä.
- Kumpiakin hierarkioita voidaan laajentaa toisistaan riippumattomasti.
- Asiakaskoodi eristetään toteutuksen yksityiskohdilta.

4.5 Composite

Kooste-suunnittelumallia voidaan käyttää hierarkkisten rakenteiden esittämiseen. Hierarkkinen rakenne koostuu komponenteista, joista kukin voi olla joko yksittäinen komponentti eli lehti tai hierarkkinen rakenne itsessään eli solmu. Tällaisen rakenteen läpikäyminen voi olla hankalaa, varsinkin, jos rakenteessa on paljon erilaisia komponentteja. Usein lehti- ja solmukomponenteilla on yhteisiä operaatioita, jotka olisi hyvä toteuttaa vain yhdessä paikassa, eikä erikseen jokaisessa komponentissa.

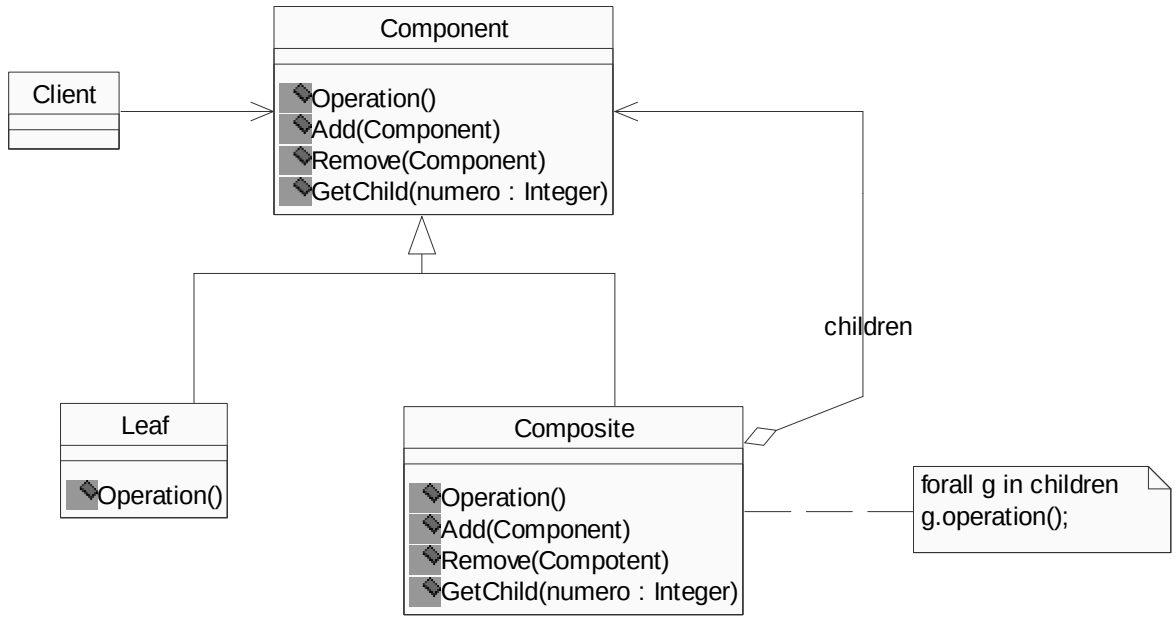
Kooste-suunnittelumalli ratkaisee edellä esitetyn ongelman määrittelemällä jokaiselle rakenteessa olevalle komponentille yhteisen abstraktin yliluokan, joka toteuttaa yhteiset operaatiot. Lisäksi jokaisessa aliluokassa voidaan luonnollisesti toteuttaa vain kyseiselle aliluokalle tarpeelliset operaatiot.
Mallin avulla voidaan siis esittää olio hierarkkisesti toisista olioista koostuvana siten, että koosteolioita ja niiden osaolioita voidaan käsitellä samalla tavalla.

4.5.1 Soveltuvuus

Kooste-mallia kannattaa käyttää seuraavanlaisissa tilanteissa:

- Halutaan esittää hierarkkisesti osaolioista koostuvia olioita.
- Asiakasohjelman kannalta on tärkeää olla erottelematta yksittäisiä olioita ja toisaalta olioista muodostettuja koosteita toisistaan. Tällöin asiakasohjelma käsittelee kaikkia olioita, sekä yksittäisiä että koosteita, samalla tavalla.

4.5.2 Toteutus



Kuva 14: Kooste-mallin rakenne.

Kooste-mallissa on seuraavanlaisia osallistujia:

- **Component:** Määrittelee rajapinnan Kooste-mallissa mukana oleville olioille. Toteuttaa kaikille luokille yhteiset operaatiot. Määrittelee myös rajapinnan lapsikomponenttien käsittelyyn ja hallintaan.
- **Leaf:** Esimerkiksi kirjain, rivi, jne..) Määrittelee kunkin primitiiviolion (esim. kirjain) toiminnan. Ei voi olla lapsikomponenttia/oliota.
- **Composite:** Määrittelee niiden komponenttien toiminnan, joilla on lapsia. Ylläpitää jotakin rakennetta, jossa on talletettuna lapsikomponentit/oliot. Määrittelee lapsikomponenttien liittyvät operaatiot, esim. lapsiolion luonti.
- **Client:** Manipuloi Kooste-mallissa mukana olevia olioita haluamallaan tavalla Component-luokan määrittelemän rajapinnan mukaisesti.

Asiakasohjelmat käyttävät Component-luokan määrittelemää rajapintaa kommunikoidakseen Composite-rakenteessa olevien olioiden kanssa. Asiakasohjelman ei tarvitse tietää, minkätyyppisen olion kanssa se kommunikoi. Jos asiakasohjelman pyynnön kohde on lehti, se käsittelee pyynnön. Mikäli pyynnön kohde taas on kooste (composite), välittää kooste pyynnön edelleen lapsikomponenteilleen suorittaen itse samalla pyynnön toteuttamiseen liittyvät toimenpiteet joko ennen tai jälkeen pyynnönvälityksen.

Kooste-malli rakentuu Silta-mallin (ks. s 56) tavoin tietyn avainabstraktion ympärille. Abstraktio (Kuva 14: *Component*) määrittelee rajapinnan kullekin komponentille ominaisille toiminnoille sekä toteuttaa ne operaatiot, jotka ovat yhteisiä kaikille alikomponenteille. Kullekin alikomponentille on oma aliluokkansa (*Leaf*), jotka määrittelevät itselleen ominaiset toimintonsa.

Composite-luokka on varsinainen yhtenäisen käsittelyn mahdollistava rakenne. Composite-luokan on mahdollista luoda rekursiivisesti uusia Component-olioita, koska luokilla Component ja Composite on yhteensopiva rajapinta, ja koska Composite-luokka voi muodostaa koosteen. Tällä tavoin muodostuu puumainen rakenne, jonka solmuina on Component-olioita ja lehtinä vaihteleva määrä Leaf-olioita.

Kooste-mallia toteutettaessa on otettava huomioon monenlaisia asioita:

- Kuinka muodostuvan oliopuun läpikäynti ratkaistaan?

- Onko järkevää luoda jokaiselle primitiivioliolle kokonaan oma olio (esim. yhden kirjaimen tallettamista varten)?
- Missä luokassa määritellään lapsikomponenttien hallintaan liittyvät toimenpiteet?
- Mikä on paras tietorakenne komponenttien tallentamiseen?

Seuraavassa esimerkki Kooste-suunnittelumallin käytöstä. Esimerkissä "koostetaan" teepusseista suurempia kokonaisuuksia, eli teepussipurkkeja. Myös pienempiä purkkeja voidaan liittää osaksi suurempaa teepussipurkkia.

```
import java.util.LinkedList;
import java.util.ListIterator;

//Abstrakti koosteluokka. Määritellään mm. metodien otsikot,
//joille määritellään myöhemmin toiminta.
public abstract class TeaBags
{
   LinkedList teaBagList;
   TeaBags parent;
   String name;

   public abstract int countTeaBags();

   public abstract boolean add(TeaBags teaBagsToAdd);
   public abstract boolean remove(TeaBags teaBagsToRemove);
   public abstract ListIterator createListIterator();

   public void setParent(TeaBags parentIn) {parent = parentIn;}
   public TeaBags getParent() {return parent;}

   public void setName(String nameIn) {name = nameIn;}
   public String getName() {return name;}
}
//Laajentava luokka "Leaf-luokka" (ks. Kuva 14)
import java.util.ListIterator;

public class OneTeaBag extends TeaBags // Abstraktin TeaBags-luokan aliluokka
{
    public OneTeaBag(String nameIn) { //Asetetaan parametrina saatu String
                                      //nimeksi
          this.setName(nameIn);
    }

    public int countTeaBags() { //Asetetaan teepussien lukumäärä ykköseksi.
          return 1;
    }

    public boolean add(TeaBags teaBagsToAdd) {return false;}
    public boolean remove(TeaBags teaBagsToRemove) {return false;}
    public ListIterator createListIterator() {return null;}
}
//"Solmukohta"
import java.util.LinkedList;
import java.util.ListIterator;
```

```
public class TinOfTeaBags extends TeaBags //TeaBags-luokan aliluokka.
{
   public TinOfTeaBags(String nameIn)
   {
       teaBagList = new LinkedList(); //Luodaan uusi linkitetty lista.
       this.setName(nameIn);    //Nimeksi parametrina saatu String.
   }

   public int countTeaBags() //Laskee teepussien määrän(listan alkioiden lkm).
   {
       int totalTeaBags = 0;
       ListIterator listIterator = this.createListIterator();
       TeaBags tempTeaBags;
       while (listIterator.hasNext())
       {
           tempTeaBags = (TeaBags)listIterator.next();
           totalTeaBags += tempTeaBags.countTeaBags();
       }
       return totalTeaBags;
   }

   public boolean add(TeaBags teaBagsToAdd) {
                                          //Lisätään teepussi (alkio listaan)
       teaBagsToAdd.setParent(this);
       return teaBagList.add(teaBagsToAdd);
   }

   public boolean remove(TeaBags teaBagsToRemove) {
                                       //Poistetaan teepussit (alkiot listasta)
       ListIterator listIterator = this.createListIterator();
       TeaBags tempTeaBags;
       while (listIterator.hasNext())
       {
           tempTeaBags = (TeaBags)listIterator.next();
           if (tempTeaBags == teaBagsToRemove)
           {
               listIterator.remove();
               return true;
           }
       }
       return false;
   }

   public ListIterator createListIterator() { //Luodaan listan iteroija.
       ListIterator listIterator = teaBagList.listIterator();
       return listIterator;
   }
}
//Testiluokka kooste-mallin testaamiseen.
class TestTeaBagsComposite {

   public static void main(String[] args)
   {
       System.out.println("Valmistetaan purkki teepusseille");
       TeaBags tinOfTeaBags = new TinOfTeaBags("purkki teepusseille");
      System.out.println("Teepurkissa on " + tinOfTeaBags.countTeaBags() + "
                               teepusseja.");

       System.out.println(" ");
```

```
        System.out.println("Luodaan teePurkki1");
        TeaBags teaBag1 = new OneTeaBag("1.teepussi");
        System.out.println("TeePurkki1:ssa on   " + teaBag1.countTeaBags() + "
                                teepusseja.");

        System.out.println(" ");

        System.out.println("Luodaan teePurkki2");
        TeaBags teaBag2 = new OneTeaBag("2. teepussi");
        System.out.println("Teepurkki2:ssa on  " + teaBag2.countTeaBags() + "
                                kpl teepusseja."); //Tulostetaan pussien lkm.

        System.out.println(" ");

        System.out.println("Laiteaan 1. teepussi ja 2. teepussi
                                teepussipurkkiin");
        if (tinOfTeaBags.add(teaBag1))
        {System.out.println("1. teepussi onnistuttu laittamaan
                                teepussipurkkiin");}
        else
        {System.out.println("1. teepussia ei onnistuttu laittamaan
                                teepussipurkkiin");}
        if (tinOfTeaBags.add(teaBag2))
        {System.out.println("2. teepussi onnistuttu laittamaan
                                teepussipurkkiin");}
        else
        {System.out.println("1. teepussia ei onnistuttu laittamaan
                                teepussipurkkiin");}
        System.out.println("Teeepussipurkissa on nyt " + tinOf-
TeaBags.countTeaBags() + " kpl teepusseja.");

        System.out.println(" ");

        System.out.println("Luodaan pieniTeePurkki");
        TeaBags smallTinOfTeaBags = new TinOfTeaBags("pieni purkki
                                        teepusseille");
        System.out.println("pieniTeePurkki " + smallTinOfTeaBags.countTeaBags()
                                        + ", jossa teepusseja");
        System.out.println("Luodaan teePurkki3");
        TeaBags teaBag3 = new OneTeaBag("3. teepussi");
        System.out.println("TeePurkki3:ssa " + teaBag3.countTeaBags() + " on
                                teepusseja.");
        System.out.println("Laitetaan 3. teepussi pieniTeePurkkiin");
        if (smallTinOfTeaBags.add(teaBag3)) //Jos lisääminen onnistuu…
        {System.out.println("3. teepussi onnistuttu laittamaan
                                pieniTeePurkkiin");}
        else {
           System.out.println("1. teepussia ei onnistuttu laittamaan
                                pieniTeePurkkiin");}
        System.out.println("pieniTeePurkissa on " + smallTinOf
                                TeaBags.countTeaBags() + " kpl teepusseja.");

        System.out.println(" ");

        System.out.println("Laitetaan pieniTeePurkki teepussipurkkiin");
        if (tinOfTeaBags.add(smallTinOfTeaBags))
        {System.out.println("pieniTeePurkki laitettu menestykselllisesti
                                teepussipurkkiin");}
        else {
           System.out.println("pieniTeePurkki ei onnistuttu laittamaan
                                teepussipurkkiin");}
```

```
        System.out.println("Teepussipurkissa on " + tinOfTeaBags.countTeaBags()
                                + " kpl teepusseja.");

        System.out.println(" ");

        System.out.println("Otetaan 2. teepussi teepussipurkista");
        if (tinOfTeaBags.remove(teaBag2))   //Jos voidaan positaa…
        {System.out.println("Nyt se on otettu teepussipurkista");}
        else
        {System.out.println("2. teepussin poistaminen ei onnistunut");}
        System.out.println("Teepussipurkissa on nyt " + tinOf
                                TeaBags.countTeaBags() + " kpl teepusseja.");
    }
}
```

4.5.3 Ominaisuuksia

Kooste-suunnittelumalli määrittelee luokkahierarkian, joka koostuu sekä primitiiviolioista että primitiiviolioista rekursiivisesti muodostetuista koosteista. Asiakasohjelma ei näe eroa kooste- ja primitiiviolioiden välillä.

- Asiakasohjelman toteutus yksinkertaistuu ja helpottuu, kun samalla ohjelmapätkällä voidaan suorittaa kaikki pyynnöt kohdeolion tyypistä riippumatta.
- Uusien ominaisuuksien lisääminen järjestelmään on helppoa, koska lisäys ei vaikuta järjestelmään muuten kuin lisäämällä luokkarakenteeseen uuden primitiiviluokan. Uudet primitiiviluokat toimivat luokkarakenteessa sellaisenaan.

4.6 Decorator

Koristelija-suunnittelumallin avulla voidaan liittää olioon lisäominaisuuksia dynaamisesti. Malli on perintään verrattuna joustavampi tapa laajentaa toiminnallisuutta. Joskus on tarpeen lisätä ominaisuuksia luokkaa pienemmille kokonaisuuksille. Esimerkiksi tekstinkäsittelyohjelmassa on voitava laittaa vain yhden kehyksen (frame) ääriviivat näkyviin ilman, että tiedoston kaikkien kehysten ääriviivat muuttuisivat näkyviksi. Tällainen toiminnallisuus voitaisiin toteuttaa perintää hyväksikäyttäen, mutta jos reunus peritään joltakin luokalta, kiinnitettään sama reunus myös aliluokkiin. Joustavampi tapa ratkaista ongelma, on kapseloida komponentti toisen olion sisään, joka lisää halutut ominaisuudet. Tällaista kapseloijaa kutsutaan *decoratoriksi* eli *koristelijaksi*. Koristelija-komponentilla on sama kutsurajapinta kuin kapseloitavalla komponentilla, joten se on käyttäjälleen näkymätön. Koristelija välittää komponenteille osoitetut viestit ja pyynnöt eteenpäin ja suorittaa samalla asiaan-

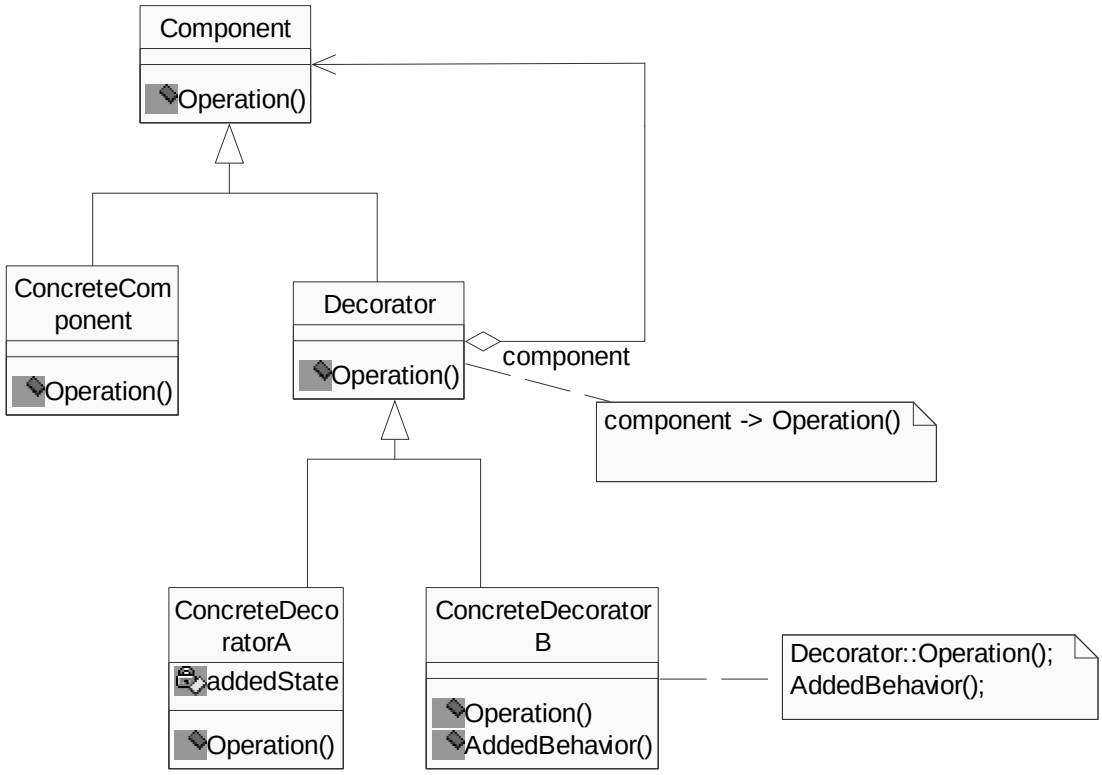
kuuluvat toimenpiteet joko ennen tai jälkeen viestinvälityksen. Koska koristelija on läpinäkyvä asiakasohjelmilleen, Koristelija-olioita voidaan luoda rekursiivisesti mielivaltainen määrä sisäkkäinkin.

4.6.1 Soveltuvuus

Koristelija-suunnittelumallia voidaan käyttää seuraavanlaisissa tilanteissa:

- Kun halutaan liittää oliolle ominaisuuksia dynaamisesti siten, että se ei vaikuta komponentin käyttöön.
- Kun halutaan lisätä vaihtuvia ominaisuuksia.
- Kun laajennus perintää hyväksikäyttäen ei ole käytännöllistä, eli aliluokkia on esimerkiksi suuri määrä tai luokkaa ei voida jostain syystä periä.

4.6.2 Toteutus



Kuva 15: Koristelija-mallin rakenne.

Koristelija-malliin osallistujat:

- **Component:** Abstrakti kantaluokka Decorator- ja kehysluokille. Määrittelee rajapinnan niihin olioihin, joihin voidaan lisätä ominaisuuksia (esim. ääriviivojen piirtäminen kehyksiin) dynaamisesti ajonaikana.
- **ConcreteComponent:** Ohjelman omia luokkia, esim. kehysluokka. Määrittelee olion, johon ominaisuudet on tarkoitus liittää.
- **Decorator:** Abstrakti kantaluokka erilaisille Decorator-luokille, esim. viivan piirtäminen, varjostus, jne. Ylläpitää viitettä Component-olioon ja määrittelee Component-luokan rajapintaa vastaavan rajapinnan, joita ConcreteDecorator-luokkien oliot käyttävät toteuttaessaan tehtäväänsä.
- **ConcreteDecorator:** lisäävät ominaisuuksia komponenttiin (ConcreteComponent).

Decorator-olio välittää asiakasohjelman pyynnöt eteenpäin kyseiselle ConcreteComponent-oliolle tekemättä pyynnölle itse mitään. Samalla Decorator-olio voi kuitenkin suorittaa myös haluamiaan operaatioita.

Seuraava esimerkki kuvaa Kuorruttaja-mallia. Ohjelmassa on abstrakti kantaluokka (Tea), johon muut luokat lisäävät oman toiminnallisuuden, eli tässä tapauksessa oman mausteensa teehen.

```
//Abstrakti kantaluokka
public abstract class Tea
{
   boolean teaIsSteeped;

   public abstract void steepTea();
}
//Kuorruttaja "the decoratee", Tea-luokan aliluokka. Toteuttaa abstraktissa
//yliluokassa määritellyt metodit.

public class TeaLeaves extends Tea {
   public TeaLeaves() {
       teaIsSteeped = false;
   }

   public void steepTea() {
       teaIsSteeped = true;
       System.out.println("teelehdet ovat likoamassa");
   }
}
//Kuorruttaja "decorator"
import java.util.ArrayList;
import java.util.ListIterator;

public class ChaiDecorator extends Tea {
                     //Tea-luokan toinen aliluokka, laajentaa toteutusta
    private Tea teaToMakeChai;
```

```
    private ArrayList chaiIngredients = new ArrayList();

    public ChaiDecorator(Tea teaToMakeChai) { //"kuorrutetaan" teetä mausteilla.
        this.addTea(teaToMakeChai);
        chaiIngredients.add("laakerinlehti");
        chaiIngredients.add("kanelitanko");
        chaiIngredients.add("inkivääri");
        chaiIngredients.add("hunaja");
        chaiIngredients.add("maito");
        chaiIngredients.add("vaniljansiemen");
    }

    private void addTea(Tea teaToMakeChaiIn){
        this.teaToMakeChai = teaToMakeChaiIn;
    }


    public void steepTea() { //Määritellään toteutus yliluokan metodille.
        this.steepChai();
    }

    public void steepChai(){
        teaToMakeChai.steepTea();
        this.steepChaiIngredients();
        System.out.println("teessä on ketjun ainekset");
    }

    public void steepChaiIngredients() {
        ListIterator listIterator = chaiIngredients.listIterator();

    //Käydään läpi maustelistaa, kunnes kaikki on sekoitettu joukkoon.
        while (listIterator.hasNext()) {
         System.out.println(((String)(listIterator.next())) + " on likoamassa");
        }
        System.out.println("ketjun ainekset ovat likoamassa");
    }
}
//Testiluokka kuorruttajan testaamiseen
class TestChaiDecorator {

   public static void main(String[] args) {
       Tea teaLeaves = new TeaLeaves();
       Tea chaiDecorator = new ChaiDecorator(teaLeaves);
       chaiDecorator.steepTea();
   }
}
```

4.6.3 Ominaisuuksia

Koristelija-suunnittelumallin ominaisuuksia:

- Joustavampi kuin staattinen perintä. Koristelijan avulla voidaan joustavasti laajentaa olioiden toiminnallisuutta. Ominaisuuksia voidaan lisätä tai poistaa ajonaikaisesti. Perintää käytettäessä jouduttaisiin luomaan luokkia jokaista lisäominaisuutta kohden. Tämä voi johtaa luokkien määrän lisääntymisen lisäksi myös järjestelmän monimutkaistumiseen.

- Helpottaa suunnittelua. Suunnitteluvaiheessa ei tarvitse tietää kaikkia mahdollisia tapauksia, joissa laajennuksia tarvitaan. Koristelijoiden avulla toiminnallisuutta voidaan laajentaa silloin kun siihen tulee tarvetta. Toiminnallisuus voidaan rakentaa useiden koristelijoiden avulla halutunlaiseksi.
- Koristelija ja sen komponentti eivät ole identtisiä. Koristelija toimii läpinäkyvänä kapseloijana. Kuitenkin komponentin näkökulmasta katsottuna koristelijalla kapseloitu komponentti ei ole identtinen komponentin kanssa. Koristelijoita käytettäessä ei tämän vuoksi tulisi luottaa komponentin identiteettiin.
- Koristelijoita käytettäessä järjestelmään tulee usein paljon pieniä olioita, jotka ovat toistensa kaltaisia. Nämä oliot eroavat toisistaan ainoastaan kytkentätavaltaan, eivät luokaltaan tai muuttujiensa arvoilta. Tällaiset järjestelmät ovat helposti muokattavia, jos ne ymmärtää, mutta niiden opettelu ja virheenjäljitys voi olla vaikeaa.

5 TAPAHTUMAMALLIT

Tapahtuma on sarja toimintoja, jotka muuttavat olion tai olioryhmän tilaa hyvin määritellyllä tavalla siten, että toimintojen tulokset (outcome) ovat yhtenäiset. Tapahtumamallit ovat käyttökelpoisia, koska niiden avulla voidaan määritellä olion tilalle rajoitteita, joiden on oltava voimassa ennen tapahtumaa, tapahtuman aikana tai tapahtuman jälkeen. Voidaan esimerkiksi asettaa rajoite, jonka mukaan olion jonkin attribuutin arvon on oltava tapahtuman jälkeen suurempi, kuin se oli ennen tapahtuman alkamista. Tapahtumat ovat tärkeässä asemassa monissa sovelluksissa.

5.1 ACID-Transactions

ACID-tapahtuma (ACID transaction) -suunnittelumalli. Tapahtuma koostuu peräkkäisistä operaatioista, jotka muuttavat olion tai oliokokoelman tilaa hyvin määritellyllä tavalla. Mallin tarkoituksena on varmistaa, että tapahtuma ei koskaan ole odottamaton tai muuten ristiriitainen. Tämä saadaan aikaan varmistamalla ACID-ominaisuudet, joita ovat:

- **Jakamattomuus** (*atomicity*), joka varmistaa sen, että kaikki tapahtuman osittain valmistuneet tulokset perutaan, jos tapahtuma keskeytyy.
- **Johdonmukaisuus/yhtenäisyys** (*consistency*). Tapahtumat tuottavat vain yhtenäisiä tuloksia. Tapahtumia voidaan suorittaa rinnakkain, mutta tulokset ovat samat,

vaikka tapahtumat suoritettaisiin peräkkäin. Johdonmukaisuudesta käytetään myös termiä sarjamuotoisuus (serializability).

- **Eristyneisyys** (*isolation*). Tapahtumat ovat eristettyjä muista tapahtumista. Tämä tarkoittaa sitä, että tapahtumien välitulokset eivät näy muille tapahtumille, eivätkä tapahtumien tekemät toiminnot vaikuta muihin tapahtumiin.
- **Kestävyys** *(durability*). Jos tapahtuman suoritus onnistuu, sen tulokset eivät voi enää kadota.

ACID-ominaisuudet on otettava huomioon hajautetuissa järjestelmissä, joissa on useita tietokantoja. Useimmat tietokannan hallintajärjestelmät varmistavat ACID-ominaisuuksien täyttymisen, mutta tietokantojen lisäksi on tärkeää huomioida kyseiset ominaisuudet myös sovellustasolla. Esimerkiksi huoltoasemilla bensiinin hinta voi muuttua nopeasti ja jopa useita kertoja päivässä. Tällöin on tärkeää, että siinä ohjelmassa, jota käytetään hinta-muutosten tekemiseen kassakoneisiin, bensiinimittareihin, mainosteluihin jne. on huomioitu ACID-ominaisuudet, jotta hintamuutos tapahtuu varmasti kaikissa näissä samanaikaisesti ja hallitusti, ettei bensiinimittareissa ole jollakin hetkellä eri hinta kuin kassakoneissa jne.

5.1.1 Toteutus

Yksinkertaisin tapa varmistaa ACID-ominaisuudet tapahtumissa on muuttaa olioiden tiloja jonkin työkalun avulla, esimerkiksi käyttämällä tietokannan hallintajärjestelmää, joka automaattisesti varmistaa ACID-ominaisuudet. Joskus ei ole mahdollista käyttää valmista työkalua kyseisten ominaisuuksien varmistamiseen. Syynä voi olla esimerkiksi suorituskykyvaatimukset tai tarve pitää sulautetun ohjelmiston koko mahdollisimman pienenä. Tapahtumalogiikan siirtäminen sovellukseen ACID-ominaisuuksien turvaamiseksi voi aiheuttaa sen, että suunnittelusta tulee monimutkainen tehtävä. Käytettäessä jotain valmista työkalua työkalu huolehtii täytäntöönpanon monimutkaisuudesta, eikä se näin ole enää osana suunnittelua. Valmis työkalu huolehtii myös usein virheidenkäsittelystä. Yleisesti ottaen voidaan todeta, että useimmiten on parempi ostaa kuin rakentaa itse ACID-tuki, jos suinkin mahdollista. Seuraavissa kappaleissa on kerrottu lyhyesti, mitä kuhunkin ACID-ominaisuuteen liittyy ja miten se voitaisiin toteuttaa.

5.1.1.1 Jakamattomuus (Atomicity)

Jakamattomuudella tarkoitetaan sitä, että joko kaikki muutokset tapahtuvat tarkalleen kerran tai muutoksia ei tapahdu ollenkaan (esim. muutos useaan tietokantaan). Nämä muutokset sisältävät sisäiset tilamuutokset, tietokantamuutokset, viestien välityksen ja muiden ohjelmien näkyvät sivuvaikutukset. Jos tapahtuma ei ole jakamaton, tapahtumien sivuvaikutukset voivat tapahtua useammin kuin kerran, esimerkiksi pankkiasiakkaan tililtä otettaisiin yhden tilitapahtuman jälkeen useamman kerran sama summa rahaa.

Jakamattomuuden toteuttamisen kannalta on tärkeää se, että olion alkuperäinen tila tallennetaan sillä tavalla, että se voidaan helposti palauttaa, jos tapahtuu jokin toimintahäiriö. Olion alkuperäisen tilan tallentamiseksi on erilaisia vaihtoehtoja. Ensinnäkin ennen kuin tehdään mitään, mikä voisi muuttaa sen olion tilaa, jota käsitellään, tapahtuman hallinnoija (transaction manager) voi käyttää toisen luokan ilmentymää olion tilan tallentamiseen. Toinen yksinkertainen tapa tallentaa olion tila on kloonata olio. Tällöin on oltava käytössä myös metodi, jolla olion alkuperäinen tila voidaan palauttaa.

On mahdollista myös luoda luokka, jonka ilmentymät ovat vastuussa olioiden tilojen tallentamisesta ja palauttamisesta käyttämällä julkisesti käytettäviä metodeja. Jos halutaan tallentaa sellaisia tiloja, joita tarvitaan loputtomasti, voidaan käyttää Javan sarjallistamispalvelua (Java's serialization facility). Tämä sarjallistamispalvelu voi tallentaa ja palauttaa koko olion tilan, jos sen luokka antaa ilmentymilleen luvan sarjallistua. Luokat antavat sarjallistamisluvan toteuttamalla *java.io.Serializable* -rajapinnan. Tämä rajapinta ei määrittele mitään muuttujia tai metodeja, vaan se ainoastaan ilmaisee, että luokan ilmentymät voidaan sarjallistaa.

Joissain tapauksissa olion koko tilan tallentaminen ei ole paras vaihtoehto. Jos tapahtumat suorittavat monien olioiden operaatioita, olioilla on paljon tilatietoa ja tapahtuma muuttaa vain osaa olion tilatiedoista, olion koko tilatiedon tallentaminen on tuhlausta. Tällaisissa tilanteissa on tehokkaampaa olla muuttamatta olion alkuperäistä tilaa niin kauan kunnes tapahtuma on loppunut. Tapahtuman loputtua käytetään kääreolioita arvojen uusimiseen. Jos tapahtuma päättyy virheeseen, kääreoliot vain hylätään.

5.1.1.2 Johdonmukaisuus (Consistency)

Tapahtuma on oikea siirtymä olion tilassa. Tapahtuman alkaessa oliot, joita tapahtuma operoi, ovat johdonmukaisia eheysrajoitteiden kanssa. Tapahtuman päättyessä, huolimatta sen onnistumisesta tai epäonnistumisesta, olioiden tulee olla johdonmukaisia eheysrajoitteiden kanssa. Jos kaikki esiehdot läpikäydään ennen tapahtumaa, kaikki jälkiehdot käydään läpi onnistuneen tapahtuman jälkeen.

Johdonmukaisuuden toteuttamiseksi ei ole olemassa mitään erityistä tekniikkaa, vaan siihen tarkoitukseen soveltuvat kaikki sellaiset tekniikat, jotka helpottavat varmistamaan ohjelman oikeellisuuden. Monipuolinen testaus on luonnollisesti avainasemassa oikeellisuuden varmistamisessa ja todentamisessa.

5.1.1.3 Eristyneisyys (Isolation)

Eristyneisyydellä tarkoitetaan sitä, että vaikka useampia tapahtumia suoritettaisiin samanaikaisesti, jokaisen tapahtuman T kohdalla pätee sama sääntö: muut tapahtumat suoritetaan joko ennen tai jälkeen T:n suorittamisen. Tämä tarkoittaa sitä, että jos olio, joka on osallisena tapahtumassa, hakee olion attribuutin, ei ole väliä missä vaiheessa se tehdään. Toisin sanoen tapahtuman elinaikana ne oliot, jotka ovat osallisena tapahtumassa, eivät välitä muiden samanaikaisten tapahtumien olioiden tilamuutoksista.

Eristyneisyydestä on huolehdittava erityisesti silloin, kun olio sisältyy samanaikaisiin tapahtumiin ja jokin tapahtumista voi muuttaa olion tilaa. Tapahtuman luonne määrää kuhunkin tilanteeseen sopivimman tekniikan. Tekniikkavaihtoehtoja on useita.

Sellaisissa tilanteissa, joissa kaikki tapahtumat voivat muokata olion tilaa, täytyy varmistua siitä, että tapahtumat eivät pääse käsiksi olioon samanaikaisesti. Tämä voidaan varmistaa sillä, että synkronoidaan ne metodit, jotka muokkaavat olion tilaa.

On olemassa myös tilanteita, joissa toiset tapahtumat muokkaavat ja toiset tapahtumat vain käyttävät olion tilaa. Tällaisissa tilanteissa kannattaa käyttää yksisäikeisyyttä suorituskyvyn parantamiseksi. Tällöin voidaan sallia olion tilaa muuttamattomien tapahtumien

päästä käsiksi olioon samanaikaisesti kun sallitaan olion tilaa muuttavien käsitellä olion tilaa yksisäikeisellä tavalla.

Tapahtumien ollessa suhteellisen pitkäkestoisia, voi sellaisten tapahtumien suorituskyvyn parantaminen olla mahdollista, mitkä käyttävät, mutta eivät muuta olion tilaa. Tämä voidaan toteuttaa siten, että tällaiset tapahtumat käyttävät olion kopiota. Tämä voidaan toteuttaa käyttämällä tällaisiin tapauksiin soveltuvaa suunnittelumallia. Jos soveltuvaa suunnittelumallia ei löydy, on myös mahdollista pilkkoa pitkäkestoisia tapahtumia lyhyemmiksi tai luopua joistain ACID-ominaisuuksista.

5.1.1.4 Kestävyys (Durability)

Kestävyydellä tai pysyvyydellä tarkoitetaan sitä, että kun tapahtuma on suoritettu oikein, ne muutokset, jotka olion tilaan on tehty, tulevat suhteellisen pysyviksi. Ne säilyvät ainakin niin kauan kunnes jokin olio huomaa jonkin muutoksen tapahtuneen.

Tärkeitä tapahtuman kestävyyden varmistamisessa on se, että tulosten on säilyttävä niin kauan kun muut oliot käsittelevät olion tilaa. Jos tapahtuman tuloksia ei tarvita yksittäisen ohjelman suorituksen jälkeen, usein riittää kun talletetaan tapahtuman tulos samaan muistiin kuin ne oliot, jotka käyttävät tuloksia. Mikäli muut oliot käyttävät tapahtuman tuloksia loputtomasti, tulokset pitäisi tallentaa hajaantumattomalle tallennusvälineelle, kuten magneettiselle levylle (magnetic disk).

5.1.2 Tunnettuja käyttökohteita

Monet Internetin jälleenmyyntisovellukset käyttävät ACID-tapahtumia. Lisäksi tietokannan hallintajärjestelmät takaavat ACID-ominaisuudet tapahtumissa.

5.2 Composite Transaction

Composite Transaction (yhdistetty tapahtuma) – suunnittelumalli. Käytetään silloin, kun halutaan suunnitella ja toteuttaa tapahtumat oikein ja mahdollisimman pienin ponnistuksin. Yksinkertaiset tapahtumat ovat helpompia toteuttaa ja tehdä oikein kuin monimutkaisemmat tapahtumat. Tällöin on mahdollista suunnitella ja toteuttaa monimutkaisemmat tapahtumat helpommista ACID-tapahtumista ikään kuin valmiita rakennuspalikoita käyttäen. Vaikka monimutkaisempien tapahtumien muodostamiseen käytettäisiin olemassa olevia ACID-tapahtumia, se ei takaa automaattisesti yhdistetyn tapahtuman ACID-ominaisuuksia.

5.2.1 Toteutus

Kannattaa suunnitella luokat, jotka toteuttavat monimutkaisia tapahtumia siten, että ne delegoivat mahdollisimman paljon toteutusta yksinkertaisemmille tapahtumille. Kun valitaan luokkia, joka toteuttavat tapahtumat liittämällä monimutkaisemmiksi tapahtumiksi, kannattaa käyttää luokkia, jotka ovat jo olemassa, ja joiden tiedetään olevan oikeita, tai sitten kannattaa valita kyseiseen tarkoitukseen sellaiset luokat, joilla on mahdollisimman paljon käyttöä.

Yksinkertaisilla tapahtumilla tulisi olla ACID-ominaisuudet, sillä se helpottaa varmistamaan ennustettavat yhdistetyn tapahtuman ominaisuudet. Joskus olosuhteet tekevät ACID-ominaisuuksien varmistamisen vaikeaksi yhdistetyissä tapahtumissa. Jos tapahtumien kokoelman ACID-ominaisuudet on toteutettu käyttämällä yksinkertaista mekanismia, joka tukee sisäkkäisiä tapahtumia, myös yhdistetyn tapahtuman ACID-ominaisuuksien turvaaminen tapahtumia yhdistettäessä on hyvin helppoa. Yksinkertaisinta on käyttää sellaista työkalua tapahtumien hoitamiseen, joka tukee sisäkkäisiä tapahtumia. Toinen mahdollisuus on käyttää tekniikoita, jotka kuvataan ACID-tapahtuma -suunnittelumallina yhteydessä. Jos komponenttien tapahtumat on hoidettu sellaisella mekanismilla, joka ei tue sisäkkäisiä tapahtumia, tarvitaan erilaisia tapoja turvata tapahtumien ennustettavat tulokset. Sellaisissa tapauksissa, joissa tapahtumat hoidetaan erilaisilla mekanismeilla, täytyy myös löytää tapa turvata yhdistetyn tapahtuman tulosten ennustettavuus.

Joissakin tapauksissa ACID-ominaisuuksien turvaaminen voi olla epäkäytännöllistä tai jopa mahdotonta. Joskus voi olla esimerkiksi sellainen tilanne, että on mahdollista jättää

eristyneisyys (isolation) -ominaisuus huomioimatta. Jos tapahtuma on esimerkiksi luonteeltaan sellainen, että se varmistaa, että samanaikaiset tapahtumat eivät muokkaa samaa oliota, nämä ominaisuudet voidaan jättää vähemmälle huomiolle.

Tapahtumien rakeisuuteen kannattaa kiinnittää huomiota suunnitteluvaiheessa, sillä on järkevää pitää yksinkertaiset tapahtumat yksinkertaisina ja monimutkaisemmat tapahtumat ymmärrettävinä.

5.3 Two Phase Commit

Kaksivaiheinen tiedostoon kirjoittaminen. Käytetään silloin, jos tapahtuma muodostuu yksinkertaisemmista tapahtumista ja jakaantuu useammille tietokannan hallinnoijille (database manager). Tällaisissa tapauksissa usein halutaan, että joko tapahtumat päätetään menestyksellisesti tai ne hylätään kaikki. Tämä voidaan toteuttaa tekemällä olio, joka vastaa tapahtumien yhdistämisestä siten, että joko kaikki suoritetaan menestyksellisesti tai kaikki tapahtumat hylätään.

5.3.1 Soveltuvuus

Kaksivaiheinen tiedostoon kirjoittaminen soveltuu käyttöön seuraavanlaisissa tilanteissa:

- Kun muulla tavalla itsenäisten jakamattomien (atomic) tapahtumien täytyy olla osallisina yhdistetyssä jakamattomassa tapahtumassa.
- Jos mikä tahansa niistä tapahtumista, jotka osallistuvat yhdistettyyn tapahtumaan jää tapahtumatta tai epäonnistuu, kaikki muutkin tapahtumat jäävät tapahtumatta. Tämä merkitsee sitä, että tapahtumat täytyy yhdistää jotenkin.
- Kun tapahtuman tulosten täytyisi säilyä niin kauan kuin joku olio on kiinnostunut tuloksista tai jos joku muu tapahtuma muuttaa olion tilaa. Jos yhdistetyillä tapahtumilla on ACID-ominaisuudet, tämä asia on hoidossa automaattisesti.

5.3.2 Toteutus

Tehdään olio, joka on vastuussa itsenäisten ACID-tapahtumien yhdistämisestä yhdistetyksi tapahtumaksi siten, että myös tällä yhdistetyllä tapahtumalla on ACID-ominaisuudet.

Tämä yhdistäjäolio suorittaa yhdistetyn tapahtuman täydentämisen kahdessa vaiheessa. Ensiksi se määrittelee, onko jokainen tapahtuma suorittanut tehtävänsä menestyksellisesti. Jos yksikin tapahtuma on epäonnistunut, olio aiheuttaa koko yhdistetyn tapahtuman hylkäämisen. Vain siinä tapauksessa, että kaikki osatapahtumat on suoritettu menestyksellisesti, yhdistäjäolio sallii osatapahtumien kirjoittaa tulokset tiedostoon (commit).

5.4 Audit Trail

Jäljitysketjua käytetään tilanteissa, joissa halutaan varmistaa, että tapahtumat on suoritettu oikein ja totuudenmukaisesti. Toteutetaan pitämällä yllä historiarekisteriä olioista tai oliokokoelmista. Rekisterin tulisi sisältää tarpeeksi yksityiskohtaista tietoa, jotta voitaisiin määritellä kuinka oliot käyttäytyivät tapahtumissa saavuttaessaan nykyisen tilansa.

5.4.1 Soveltuvuus

Jäljitysketju soveltuu käytettäväksi esimerkiksi seuraavanlaisissa tilanteissa:

- Kun on tarvetta tallentaa tapahtumat, jotka ovat muuttaneet olion tai oliokokoelman tilaa, jotta voidaan määritellä onko olion nykyinen tila oikea.
- Kun halutaan laskea olion toimenpiteiden lukumäärä, esimerkiksi turvallisuustarkoituksissa tai virheiden jäljityksessä.

Kun tapahtumat on tallennettu, niitä ei voi muuttaa. Jos tapahtumia olisi mahdollista muuttaa jälkikäteen, ei voitaisi olla varmoja siitä, mitä todellisuudessa tapahtui.

5.4.2 Toteutus

Historiarekisterin tulisi sisältää kaikki ne tapahtumat, jotka muuttavat tarkastelussa olevan olion tilaa. Jotta voitaisiin käyttää historiarekisteriä apuna määriteltäessä sitä, onko olion senhetkinen tila oikea, täytyy myös olion alkuperäinen tila tallentaa rekisteriin. Olisi syytä tallettaa rekisteriin myös ne tapahtumat, jotka tarkasteltavana oleva olio on aloittanut, koska usein ne on tärkeää tietää, jotta olion käyttäytymistä voitaisiin arvioida.

Historiarekisterin ei välttämättä tarvitse sisältää epäonnistuneita tapahtumia, jotta pystyttäisiin vahvistamaan olion nykyinen tila. Kuitenkin myös epäonnistuneitten tapahtumien

tallentamisella voidaan saavuttaa joitakin etuja. Se voi esimerkiksi helpottaa virheiden jäljittämistä ja turvallisuusongelmien esille tulemista. Jos myös epäonnistuneet tapahtumat tallennetaan joksikin aikaa historiarekisteriin, jokaisen tapahtuman kohdalle rekisterissä on talletettava tieto siitä, suoritettiinko tapahtuma menestyksellisesti loppuun vai ei. Jos tapahtumien määrä sovelluksessa muodostuu suureksi, on kaikkia tapahtumia fyysisesti mahdotonta tarkastella jäljitysketjussa. Tällaisissa tilanteissa kannattaa käyttää osittaista jäljitysketjua.

6 HAJAUTETUN ARKKITEHTUURIN MALLIT

Hajautetun arkkitehtuurin malleja (Distributed Architecture Patterns) voidaan käyttää suunniteltaessa systeemin korkean tason arkkitehtuuria. Hajautetun arkkitehtuurin mallit kuvaavat nimensä mukaisesti arkkitehtuuritasoa, joten sen takia näistä malleista ei ole koodiesimerkkejä.

6.1 Shared Object

Jaetun olion -malli (shared object) soveltuu käytettäväksi tilanteissa, joissa on saatavilla vain vähän tietoa tai resurssien määrä on rajoitettu. Kun jaetaan oliot monien asiakkaiden (client) kesken, saadaan jaettua myös kapseloitu tieto ja piilevät resurssit.

6.1.1 Soveltuvuus

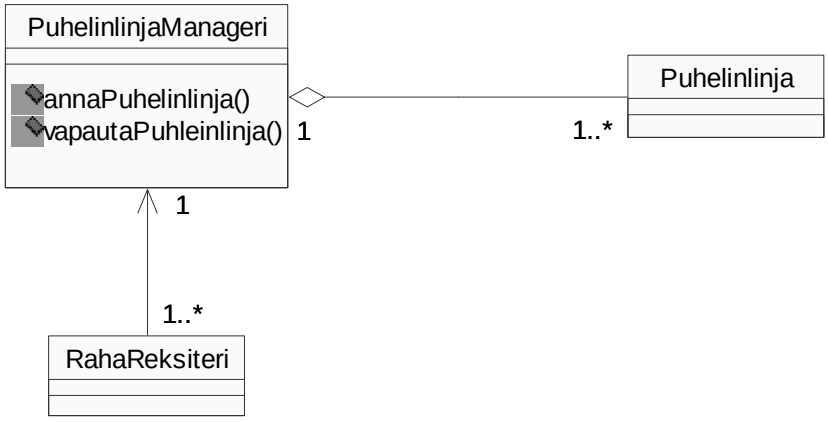
Jaettu olio -mallia kannattaa soveltaa esimerkiksi seuraavanlaisissa tilanteissa:

- Monet asiakas-oliot vaativat päästä käsiksi saman luokan ilmentymään tai olioon, joka toteuttaa määrätyn rajapinnan.
- Asiakas-oliot haluavat päästä käsiksi resurssiolioon, joka on ainutlaatuinen tai jota on saatavilla vain rajoitettu määrä.
- Asiakas-olion hallitsematon resurssiolion käsittely voi aiheuttaa pahoja tai arvaamattomia tuloksia.
- Halutaan saada se hyöty, joka saavutetaan, kun monet oliot jakavat samat resurssit.
- Halutaan mahdollistaa se, että jotkut resurssioliot hallinnoivat asiakkaitaan itsenäisesti.

6.1.2 Toteutus

Seuraavassa esimerkissä kuvataan luottokorttimaksu-systeemiä. Kun asiakas maksaa luottokortilla, systeemin täytyy ottaa yhteyttä maksujen käsittelykeskukseen, jotta voidaan varmistua siitä, että luottokortin numero on oikea ja antaa hyväksyntä tapahtumaan. On kuitenkin otettava huomioon, että samaan aikaan useampi luottokorttimaksu-systeemi voi ottaa yhteyttä käsittelykeskukseen. Yleisin tapa yhteydenottoon on tehdä se puhelimitse, mutta jos yhteydenottoja on samanaikaisesti useita ja linjoja vain yksi, muut joutuvat odottamaan vuoroaan. Jotta tällaiset odotusajat voitaisiin poistaa, linjoja on oltava useita.

Mallin mukainen ratkaisu ongelmaan on seuraavanlainen:



Kuva 16: Jaetut puhelinlinjat

RahaRekisterit toimivat PuhelinlinjaManageri-olion välityksellä. Kun RahaRekisteri-olio kutsuu annaPuhelinlinja-metodia, PuhelinlinjaManageri varaa yhden Puhelinlinja-olion käyttöönsä. Jos vapaita linjoja ei ole saatavilla PuhelinlinjaManageri odottaa, kunnes jokin linjoista vapautuu. Kun RahaRekisteri-olio ei enää tarvitse puhelinyhteyttä, se kutsuu vapautaPuhelinlija-metodia ja Puhelinlinja-olio vapautuu muiden RahaRekisteri-olioiden käyttöön.

PuhelinlijaManageri-olio on jaettu olio, joka hallinnoi useita RahaRekisteri-olioita. Kun Asiakas-olio tarvitsee RahaResurssi-oliota, PuhelinlinjaManageri päättää, mitä RahaResurssi-oliota Asiakas voi käyttää. Toteutuksessa on myös määriteltävä se, kuinka varmistetaan RahaRekisteri-olion oikeanlainen käyttö ja toiminta sellaisissa tilanteissa, kun yhtään RahaRekisteri-oliota ei ole saatavilla.

6.1.3 Ominaisuuksia

Jaettu olio -suunnittelumallilla on seuraavanlaisia ominaisuuksia:

- Olioiden jakaminen monien asiakkaitten kanssa voi olla tehokas tapa hallita rajoitettuja resursseja.
- Jos muut oliot jakavat olion, ne jakavat samalla epäsuorasti myös olion käyttämät resurssit.
- Jos monet asiakkaat jakavat olion, Manageri-oliosta voi muodostua toteutuksen pullonkaula.
- Voi olla hankalaa toteuttaa olion jakaminen, jos olio ja sen jakavat asiakkaat sijaitsevat hajautetusti eri koneilla.

6.2 Object Replication

Olioin kahdentaminen. Object Replication -mallia tarvitaan, kun halutaan parantaa hajautetun laskennan suoritustehoa tai käytettävyyttä. Hajautetut laskennat sisältävät olioita, joissa on monia hajautetun laskennan elementtejä. Nämä elementit kommunikoivat mallissa keskenään. Joskus hajautetun laskennan suoritustehoa ja käytettävyyttä on mahdollista parantaa kopioimalla olio moniin laskentaelementteihin ja pitämällä samalla yllä olion asiakkaille illuusiota, että kysymyksessä on yksittäinen olio.

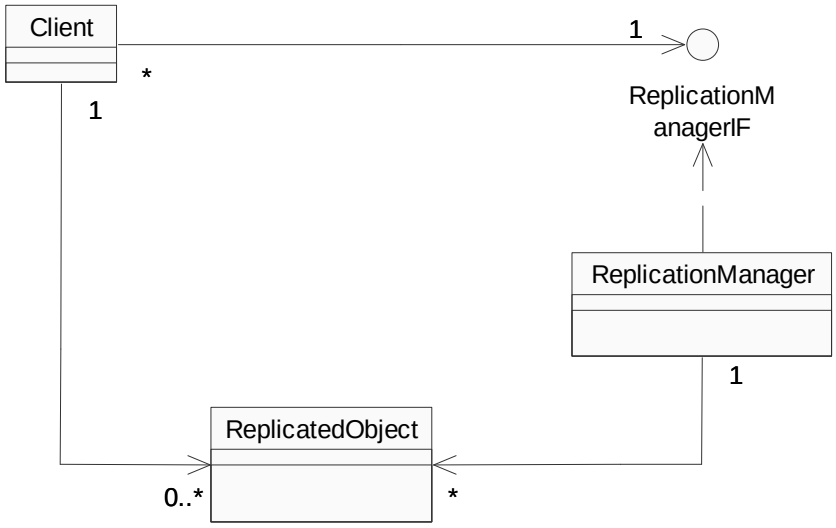
6.2.1 Voimat

Olion kahdentamis -mallilla on seuraavanlaisia voimia:

- Kopioimalla olio moniin laskentaelementteihin, saadaan parannettua systeemin viansietokykyä. Jos olio on vain yhdessä laskentaelementissä, oliosta tulee käyttökelvoton, jos laskentaelementti ei ole saatavilla. Kopioimalla olio moniin elementteihin, olio on saatavilla vaikka jokin laskentaelementeistä olisikin saamattomissa.
- Mitä lähempänä olio on liitännäisiään (accessors), sitä nopeammin liitännäiset yleensä voivat päästä siihen käsiksi. Esimerkiksi olio pääsee yleensä nopeammin käsiksi sellaiseen toiseen olioon, joka sijaitsee saman koneen muistissa. Jos eri sijaintipaikoissa olevat oliot käsittelevät samaa oliota, käsiksipääsyä voidaan usein nopeuttaa kopioimalla jokainen olio niin, että ne sijaitsevat lähellä liitännäisiään.

- Mallin avulla saavutetaan suuri parannus suorituskykyyn ja käsittelyyn tilanteissa, joissa laskentaelementtejä sisältävät oliot on yhdistetty muihin laskentaelementteihin vain osan aikaa. Esimerkiksi kannettava tietokone voi olla suurimman osan ajasta niin, että sitä ei ole yhdistetty lainkaan verkkoon.
- Sen mekanismin, joka pitää yllä molemminpuolisesta yhtenäisyydestä pitäisi olla mahdollisimman läpinäkyvä. Tällä tavalla saadaan aikaan vaikutelma, että jokaisen kopioidun olion asiakas on yksittäisen olion asiakas.

6.2.2 Toteutus



Kuva 17: Olion kahdentamis -malli.

Malliin osallistujat:

- **ReplicatedObject:** Tässä roolissa olevat luokat on kahdennettu, jotta ne olisivat lähempänä asiakkaitaan.
- **ReplicationManagerIF:** Asiakas-oliot ovat vuorovaikutuksessa tässä roolissa olevan rajapinnan kanssa päästäkseen käsiksi ReplicatedObject-luokan ilmentymään.
- **ReplicationManager:** On vastuussa kopioiden luomisesta ja olinpaikasta. Löytää kahdennetun olion lähimmän kopion. Jos kopio on liian kaukana, se luo paikallisen kopion ja mahdollistaa, että Asiakas-olio voi käyttää sitä.
- **Client:** (Asiakas-olio). Käyttävät ReplicatedObject-luokkaa. Pääsevät käsiksi tämän luokan olioihin kutsumalla ReplicationManagerIF-rajapinnan metodeja. Jos Asiakas-olio saa kutsuunsa vastaukseksi tiedon, että ReplicatedObject-olio ei ole saatavilla, se pyytää ReplicationManagerIF-oliolta toista ReplicatedObject-ilmentymää.

6.2.2.1 Replication management

Kopioimisen hallinta alkaa, kun Asiakas-olio kutsuu ReplicationManager-olion metodia pyytämään kopioidun olion käyttöä. Jos paikallinen kopio löytyy, käytetään sitä, mutta jos paikallista kopiota ei löydy, ReplicationManager-olio löytää jonkin naapurikopioista. Jos jokin niistä tarpeeksi lähellä, se valitsee yhden kopioista käyttöönsä. Jos taas kaikki naapurikopiot ovat liian kaukana, se luo uuden paikallisen kopion.

Suurimpia ongelmia mallissa on se, kuinka ylläpidetään tieto siitä, missä kopiot sijaitsevat. Jos vain muutama kopio on samalla paikallisella alueella, voidaan sijaintitieto hyvin pitää keskushakemistossa. Myös nämä hakemistot voidaan kopioida lähelle tarvitsijaa, jos järjestelmässä on useita koneita, jotka yrittävät paikallistaa kopion sijainnin samanaikaisesti. Jos ei ole mahdollista etukäteen tietää, missä kopiohakemistoja tullaan tarvitsemaan, voidaan käyttää olemassa olevaa strategiaa. Se sisältää sellaisen arkkitehtuurin, jonka avulla ReplicationManager-oliot voivat löytää toisensa myös verkon yli. Sitä käytetään, jos lähin kopiohakemisto ei ole samassa paikallisessa verkossa kuin ReplicationManager-olio, joka tahtoo käyttää sitä.

6.2.2.2 Change Replication

On olemassa useita *kopioiden muuttamismekanismeja*. Mikään niistä ei ole kuitenkaan toisiin nähden ylivoimainen, vaan kaikilla niistä on omat hyvät ja huonot puolensa. Usein mekanismin valinta on kompromissi, kaikkia hyviä puolia ei voi saavuttaa samanaikaisesti. Seuraavassa esitellään lyhyesti muutamia tekniikoita.

- **Naive pessimistic concurrency:** Ennen kuin vaihto hyväksytään yhdelle kopiolle, lukitaan muut kopiot. Tällä tavalla varmistetaan, että kaikki kopiot saavat tiedon muutoksesta samaan aikaan, ja ettei muita muutoksia tule samaan aikaan. Huonoja puolia mallissa on useita. Esimerkiksi kaikkien kopioiden lukitseminen vie aikaa, samoin muutoksen ilmoittaminen. Muutokset eivät voi myöskään tapahtua samanaikaisesti. Lisäksi kaikkien kopioiden tulee olla toistensa saavutettavissa. Jos yksikin kopio ei ole jonkin toisen kopion saavutettavissa, muutoksia ei voida tehdä mihinkään kopiosta, koska kaikkia kopioita ei saada lukittua.

- **Primary backup:** Yksi kopioista valitaan alkuperäiseksi kopioksi ja muut kopiot ovat ns. harkittuja varakopioita. Kun varakopio saa pyynnön muuttaa tilaansa, se lähettää pyynnön edelleen alkuperäiselle kopiolle. Kun muutos on tehty alkuperäiseen kopioon, se lähettää sen varakopioille. Jos alkuperäinen kopio on saavuttamattomissa, varakopiosta tulee uusi alkuperäinen kopio. Tällä tekniikalla on sama huono puoli kuin edellisellä, eli muutoksia ei tehdä, jollei muutosta voida tehdä kaikkiin kopioihin. Lisäksi päivityksen kestolle ei ole annettu rajaa.
- **Majority voting:** Kun yksi kopioita muuttuu, muutos tehdään automaattisesti valtaosaan kopioista. Tässä tapauksessa valtaosalla kuitenkin tarkoitetaan sitä, että jos oliosta on 1000 kopiota, muutos pitää tehdä automaattisesti 501 kopioon.
- **Timestamping:** Tässä lähestymistavassa kopioidun olion attribuutteihin tehdään aikamerkintä. Kopiot levittävät muutokset muihin kopioihin pareittain. Jos toisella kopiolla on pariaan vanhempi aikamerkintä attribuuttinaan, korvataan vanhempi arvo uudella arvolla ja aikamerkinnällä. Jos sama kopio on osana useammassa parissa, kaikkiin osasiin muutetaan sama arvo. Huonona puolena mainittakoon esimerkiksi se, että kaikkien koneiden kellojen on oltava samassa ajassa. Toisaalta muutoksen levittämisellekään ei ole määritelty mitään aikarajoitusta.
- **Optimistic replication:** Saa nimensä siitä oletuksesta, että muutosta ei tehdä lainkaan olion kopioihin. Muutokset tehdään lukitsematta ensin kopioita. Kun muutokset on tehty, tarkastetaan, ettei tehty mitään ristiriitaisia muutoksia. Lähestymistapa on nopea, jos mitään ristiriitaisuuksia ei ilmene. Jos ristiriitaisuuksia havaitaan, aikaa voi kulua paljonkin, sillä tällöin kaikki kopiot on taas yhdenmukaistettava. Tämä merkitsee usein kaikkien kopioiden lukitsemista ja ristiriitaisen muutoksen perumista.

6.2.3 Ominaisuuksia

Mallilla on seuraavanlaisia ominaisuuksia:

- Jos oliosta on kopioita yhtä paljon kuin oliolla on asiakkaita, voidaan maksimoida kopioiden paikallisuus niiden asiakkaille ja parantaa kokonaisuudessaan systeemin suorituskykyä.
- Kaikissa olion kopioissa on oltava sama tilatieto. Tämän tiedon ylläpitäminen on vaikeaa ja virhealtista.

- Operaatioiden päivittämisessä kaikkiin kopioihin tulisi turvata ACID-ominaisuudet (ks. sivu 70), tämä vie kuitenkin paljon aikaa.
- Erillisyys (Isolation) -ominaisuuden turvaaminen tapahtumassa, joka muuttaa kopioitua oliota, on ongelmallista.

6.3 Redundant Independent Objects

Redundanttien itsenäisten olioiden malli kuuluu muutaman edellä esitellyn mallin tavoin myöskin jaettujen arkkitehtuurien malleihin. Malli on käyttökelpoinen, kun halutaan varmistaa, että olio on saatavilla vaikka se tai sen alusta olisi juuri suorittamassa jotakin virheellistä toimintoa. Tämä ominaisuus aikaansaadaan tarjoamalla ylimääräiset oliot, jotka eivät ole riippuvaisia mistään yksittäisestä yleisestä resurssista.

6.3.1 Voimat

Seuraavanlaisten voimien vaikutuksia on mallin kohdalla harkittava:

- Kaksi itsenäistä systeemin komponenttia menee toimintakyvyttömiksi todennäköisemmin eri aikaan kuin samanaikaisesti.
- Komponentteja kutsutaan *redundanteiksi*, jos ne suorittavat samaa tehtävää ja jos muut tehtävästä riippuvaiset komponentit voivat toimia, jos yksi komponenteista jatkaa tehtävän suorittamista.
- Mitä suuremmat kustannukset häiriöajasta aiheutuu, sitä helpompi on perustella sitä, että häiriöt pyritään ennaltaehkäisemään.
- Redundanttien itsenäisten olioiden käyttäminen lisää systeemin monimutkaisuutta ja suunnittelusta, integroinnista ja kokoonpanosta tulee vaikeampaa.
- Mallin käyttäminen lisää rakennuskustannuksia.
- Mallin käyttäminen vähentää, mutta ei eliminoi sitä, että systeemi voi joskus mennä toimintakyvyttömäksi.

6.3.2 Toteutus

Systeemin toimintakykyisyyttä voidaan parantaa rakentamalla se redundanteilla itsenäisillä komponenteilla. Komponenttien tulee olla riittävän itsenäisiä, jotta yhden komponentin toimintahäiriö ei kasvata muiden komponenttien toimintahäiriön todennäköisyyttä. Jotta yksittäinen laitteistovirhe ei johtaisi koko redundanttien komponenttien kokoelman toimintahäiriöön, ne tulisi ajaa eri laitteistokomponenteissa.

On hyvin epätodennäköistä, että kaikki itsenäisesti toteutetut redundantit komponentit jakaisivat saman tavanomaisen virheen, joten tästä syystä nämä komponentit tulevat toimintakyvyttömiksi juuri samaan aikaan. Toteutusta ei kuitenkaan pidä vain kopioida komponentista toiseen, koska silloin samanlainen toimintahäiriö kaataa koko systeemin. Samat asiat on toteutettava eri komponenteissa eri tavalla.

Redundanttien itsenäisten komponenttien malli on erikoistunut muoto Olioiden kahdentamis -mallista (s.79), joten muuten toteutuksessa pätevät samat asiat kuin tuossa mallissa.

6.4 Prpompt Repair

Korjauskehote-malli on käyttökelpoinen tilanteissa, joissa halutaan varmistaa, että toimintahäiriön jälkeisen myöhemmän toimintahäiriön aiheuttama koko systeemin toimimattomuuden todennäköisyys olisi mahdollisimman pieni. Jotta minimoitaisiin katastrofaalisen toimintahäiriön todennäköisyys, toimintahäiriöinen olio on korjattava niin pian kuin mahdollista. Mallia käytetään esimerkiksi joillain web-sivuilla, jotta voidaan varmistua siitä, että sivut ovat aina nähtävissä.

6.4.1 Voimat

Mallin ongelma-alueeseen liittyviä jännitteitä:

- Systeemin kaksi itsenäistä komponenttia tulevat toimintakyvyttömiksi todennäköisemmin eri aikaan kuin juuri samanaikaisesti. Vaikka systeemi pystyisi jatkamaan toimintaansa yhden komponentin toimintahäiriön jälkeenkin, katastrofaalisen toimintahäiriön riski muissa komponenteissa kasvaa huomattavasti.

- Kun joku redundanteista itsenäisistä komponenteista on toimintakyvytön, systeemissä on vain vähän tai ei ollenkaan redundanssia (toistoa). Tämän vuoksi koko systeemin toimintakyvyttömyyden riski kasvaa. Mitä nopeammin toimintakyvytön komponentti korjataan sitä parempi.
- Komponentit täytyy rakentaa sillä tavalla, että toimintahäiriö huomataan heti kun sellainen ilmenee.

6.4.2 Toteutus

Komponentit on mallissa tärkeää rakentaa sillä tavalla, että jos siinä ilmenee toimintahäiriö, se huomataan mahdollisimman pian, jotta virhe voitaisiin myös korjata mahdollisimman nopeasti. Ohjelman ei ole yleensä mahdollista itse korjata omia virheitään, vaan komponentin korjaaminen tarkoittaa monissa tapauksissa sitä, että se käynnistetään uudelleen. On olemassa erilaisia uudelleenkäynnistysstrategioita, seuraavassa lyhyt kuvaus kahdesta erilaisesta strategiasta.

- **Cold start:** *Kylmä käynnistäminen* on uudelleenkäynnistystavoista yksinkertaisin. Sisältää vain komponentin käynnistämisen alkuperäiseen tilaansa.
- **Checkpoint restart:** *Tarkastuspistekäynnistys*. Käynnistetään komponentti siihen tilaan, joka oli lähimpänä toimintahäiriön ilmaantuessa vallinnutta tilaa.

Järjestelmän, jonka pitäisi toipua nopeasti virheistä, on oltava hyvin testattu. Lisäksi ei kannata olettaa, että virheet tulisivat aina nopeasti esille, vaan kannattaa käyttää vielä lisästrategioita toimintahäiriöiden jäljittämiseksi.

Yksinkertainen tapa huomata, mikä komponentti on lopettanut toimintansa, on määritellä komponenteille maksimiaika, jonka asiakas saa odottaa, että komponentti tekee jotakin. Kun aika on kulunut ja asiakas ei ole saanut vastaustaan, se olettaa automaattisesti, että komponentti on toimintakyvytön. Tämä tekniikka toimii parhaiten tilanteissa, joissa komponenttien toiminta tapahtuu yleensä tietyssä ajassa.

Voting (äänestys) on myös tavanomainen tapa selvittää, että jotkin komponentit eivät toimi oikein. Tekniikka tarkkailee toimintoja ja komponenttien antamia tuloksia. Samanlaisia

toimintoja ja tuloksia pidetään oikeina. Vastaavasti ne tulokset, jotka eroavat enemmistön tuloksista ajatellaan virheellisiksi.

6.5 Demilitarized Zone

Demilitarisoitu vyöhyke (DMZ). Malli on käyttökelpoinen tilanteissa, joissa on vaarana, että hakkerit voivat vaarantaa yleisesti saatavilla olevan palvelimen turvallisuuden. Ei myöskään haluta kohdata niitä seurauksia, joita tulee, jos hakkerit pääsevät käsiksi sellaisiin palvelimiin, jotka eivät ole yleisesti saatavissa, ja jotka sisältävät herkkää tietoa. Tällaisissa tapauksissa soveltuu käytettäväksi Demilitarisoidun vyöhykkeen malli. Mallissa kaikki ne verkon palvelimet, joihin päästään vapaasti käsiksi sijoitetaan reitittimen taakse mutta palomuurin etupuolelle. Tätä kutsutaan *Demilitarisoiduksi vyöhykkeeksi*. Ainoastaan muut palomuurin takana ja demilitarisoidulla alueella olevat koneet tietävät palomuurin takana olevien palvelimien verkko-osoitteet. Yleisen verkon eli Internetiin välityksellä päästään käsiksi vain demilitarisoidulla vyöhykkeellä oleviin palvelimiin. Internetin kautta ei päästä koskaan käsiksi palomuurin takana oleviin palvelimiin. Sitä vastoin demilitarisoidulla vyöhykkeellä olevat palvelimet voivat kommunikoida myös palomuurin takana olevien palvelimien kanssa, mutta ainoastaan, jos se on erittäin tarpeellista.

6.5.1 Soveltuvuus

Demilitarisoidun vyöhykkeen malli soveltuu käytettäväksi seuraavanlaisissa tilanteissa:

- Jos ei haluta, että tunkeilijat pääsevät helposti käsiksi muihin koneisiin, etenkin niihin, jotka sisältävät herkkää tietoa.
- Kun ei haluta, että tunkeilijat pääsevät käsiksi muihin palvelimiin palomuurin takana.

6.5.2 Ominaisuuksia

Mallilla on seuraavanlaisia ominaisuuksia:

- Ne asiakkaat, jotka kommunikoivat Internetin välityksellä, eivät suoraan pääse käsiksi palomuurin takana oleviin palvelimiin, joten ne ei voi myöskään vaarantaa niiden turvallisuutta.
- Epäsuora kommunikointi palomuurin takana olevien palvelimien kanssa on hitaampaa.

- Joissakin sovelluksissa mallin käyttäminen voi tehdä palvelimesta monimutkaisemman halkaisemalla sen kahteen osaan.

LÄHTEET

[GaH00] Gamma, E.; Helm,R.; Johnson, R.; Vlissides, J.: Design Patterns. Addison Wesley, Holland, 2000.

[Gra02] Grand, Mark: Java Enterprise Design Patterns. Patterns in Java. Volume 3. John Wiley & Sons, Inc., New York, 2002.

[Eck01] Eckel, Bruce: Thinking in Patterns with Java, Free Electronic book. 2001.

http://www.cs.helsinki.fi/u/laine/arkki/k01/esittelyt/

http://www.inf.bme.hu/ooret/1999osz/DesignPatterns

http://www.fluffycat.com/java/patterns.html

http://193.167.110.132/marko.forsell/Harjoitustyot/R_12.pdf

http://www.tml.hut.fi/Opinnot/T-109.450/1997/