Ryhmä 8

Riina Antikainen

Tuukka Mytty

Janne Valle

Simo Silander

OTP2/Suunnittelumallit

12.5.2019

REFAKTOROINTI RAPORTTI

VarastoSofta-ohjelman toteutuksessa on pyritty seuraamaan niin sanottua MVC-mallia (Model-View-Controller). MVC on ohjelman arkkitehtuuriin liittyvä suunnittelumalli, jota yleisesti käytetään käyttöliittymien kehitykseen. Se jakaa ohjelman sisäisen rakenteen kolmeen toisiinsa kytkeytyvään osaan. Osien erillään pito edistää tehokasta koodin uudelleen käyttöä sekä samanaikaista ohjelmiston kehitystä. Model osa pitää sisällään ohjelman tietorakenteen sekä hallinnoi tietoa, logiikkaa ja ohjelman sääntöjä. View on mikä tahansa ohjelman käsittelemän tiedon esitys kuten käyttöliittymä tai kartta, joita voi olla ohjelmassa useampia. Controller hyväksyy syötteen ja muuttaa sen komennoiksi joko Modelille tai Viewlle. Eri komponentteihin jaon lisäksi MVC-malli määrittelee, miten komponentit vuorovaikuttavat toistensa kanssa. Model huolehtii ohjelman informaatiosta ja saa käyttäjän syötteen Controllerin kautta. View esittää Modelin halutussa muodossa käyttäjälle. Controlleri vastaa käyttäjän syötteeseen ja vuorovaikuttaa Modelin objekteihin. Se vastaanottaa käyttäjän syötteen ja mahdollisesti validoi sen ennen kuin välittää sen Modelille.

Refaktorointi on tehty Martin Fowlerin kirjan Refatoring [1] tapoja käyttämällä sekä Metropolian kurssin suunnittelumallit tietoja hyödyntämällä. Kursiivilla olevat englannin kieliset termit viittaavat Fowlerin kirjan termeihin. Fowlerin ohjeita noudattamalla koodista on poistettu “Bad smells” eli huonoa ohjelmakoodia. Suunnittelumallit kurssilla käytiin läpi Gamman vuonna 1995 esittelemät 23 suunnittelumallia [2], joita on pyritty hyödyntämään koodin rakenteen parantamiseen. Lisäksi SonarQuben löytämät havainnot on poistettu parhaan mukaan koodista.

Tässä kappaleessa käydään läpi, miten VarastoSofta-ohjelman käyttäjän toimintojen varmistusdialogit ja loppuunsuorittamisdialogit refaktoroitiin. Lähtökohtana oli esimerkkikoodi 1 mukainen koodi, jossa tehdään varmistusdialogi ohjelman lopetuksen varmistamiseksi.

```
Alert alert = new Alert(AlertType.CONFIRMATION);
alert.setTitle("Confirmation Dialog");
alert.setHeaderText(null);
alert.setContentText("Are you sure want to Exit the program?");
Optional<ButtonType> result = alert.showAndWait();
if (result.isPresent() && result.get() == ButtonType.OK){
   return true;
} else {
   return false;
}
```

Esimerkkikoodi 1. Dialogin näyttämiseen tarvittava koodi.

Tämä, ja vastaavat dialogit, olivat alkuun tällaisenaan koodin siinä kohdassa, missä dialogia tarvittiin. Ensimmäinen refaktorointi, joka tehtiin oli tämän laittaminen metodin sisälle (*Extract Method)*. Tämän jälkeen nämä koottiin yhteen omaan luokkaansa *(Extract Class)*. Näin päästään hyödyntämään delegointia ja Gamman teesin numero kaksi "Suositaan koostamista mieluummin kuin periytymistä." mukaan myöskin koostamista. Refaktorointi ei kuitenkaan loppunut tähän, koska nyt oli useita metodeja, joilla oli runsaasti yhteistä koodia. Tämän parantamiseksi luotiin kaksi uutta metodia *(Extract Method),* joihin yhteiset osat koodia sijoitettiin. Yksi metodi oli varmistusta varten ja toinen informaatio ilmoituksia varten. Nämä varmasti saisi laitettua yhteen metodiin, mutta koodin luettavuuden sekä "Separation of Concerns" periaatteen perusteella nämä päätyivät kahteen eri metodiin. Näin noin 13 metodin toteutus saatiin mahdutettua kahteen. Tämä ratkaisu tukee erittäin hyvin koodin uudelleen käytettävyyttä, koska ohjelman lopetusvarmistusilmoitukseen voidaan aina käyttää yhdestä paikasta löytyvää metodia riippumatta näkymästä.

Suunnittelumallit

Refaktoroinnissa lokalisointi toteutettiin käyttämällä Singleton suunnittelumallia. Aikaisemmin koodissamme haettiin aina tarpeen mukaan käytössä olevan kielen mukainen kielitiedosto. Tässä toteutuksessa oli kuitenkin vaarana se, että käyttöliittymään olisi voinut päätyä kahden eri kielen tekstejä, koska ne olisi helposti voitu hakea vain laittamalla new Locale("en","GB") koodiesimerkin 2 parametrin istunto.getKieli() paikalle.

```
this.kaannokset = ResourceBundle.getBundle("MessagesBundle", istunto.getKieli(), new UTF8Control());
```

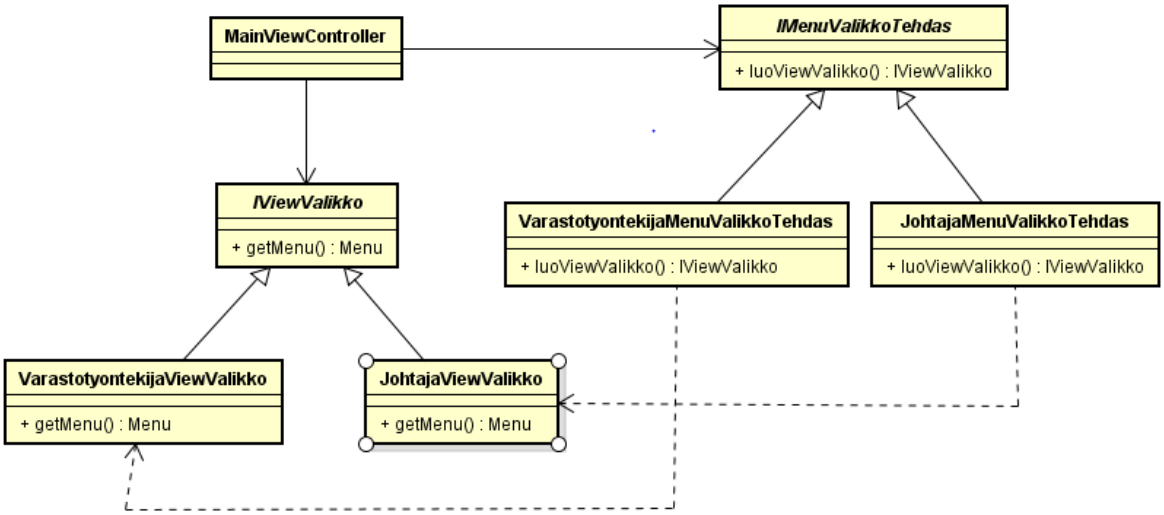
Esimerkkikoodi 2. Kielitiedoston hakemiseen tavittava koodi.

Singleton mallissa halutaan varmistaa, että luokalla on vain yksi ilmentymä, jota voidaan käyttää mistä tahansa. Tämä varmistetaan sillä, että konstruktorin näkyvyysmääre on private, jolloin mikään muu luokka ei voi luoda siitä uutta ilmentymää. Kielitiedostoa käyttäessä haluamme, että käytämme koodissa koko ajan samaa kielitiedostoa, joten singleton sopii erinomaisesti tähän tarkoitukseen. Luokat saavat Kaannokset luokan ilmentymän käyttöön Kaannokset kaannokset = Kaannokset.getInstance(); koodilla ja yhden käännoksen tekemiseen käytetään kaanna(String teksti)- metodia. Singletonia käyttämällä saamme Kaannokset-luokan avulla kielitiedoston käyttöön, missä tahansa osaa koodia ja voimme olla varmoja, että sama kielitiedosto vaikuttaa jokaiseen käännökseen.

```
public class Kaannokset {
   private final static Kaannokset KAANNOKSET = new Kaannokset();
   private Locale kieli;

   /**
    * Kaannokset konstruktori. Alustetaan default kieleksi englanti.
    */
   private Kaannokset() {
      this.kieli = new Locale("en","GB");
   }

   /**
    * Palautetaan Käännökset ilmentymä.
    * @return Käännökset ilmentymä
    */
   public static Kaannokset getInstance() {
      return KAANNOKSET;
   }

   /**
    * Asetetaan käytettävä kieli
    * @param kieli asennettava kieli
    */
   public void setKieli(Locale kieli) {
      this.kieli = kieli;
   }

   /**
    * Palautetaan käytössä oleva kielitiedosto.
    * @return kielitiedosto
    */
   public ResourceBundle haeKielitiedosto() {
      return ResourceBundle.getBundle("MessagesBundle", this.kieli, new UTF8Control());
   }

   /**
    * Käännä sana käytössä olevalle kielelle.
    * @param kaannettava sana tai lause, joka käännetään
    * @return käännetty sana tai lause
    */
   public String kaanna(String kaannettava) {
      return haeKielitiedosto().getString(kaannettava);
   }
}
```

Esimerkkikoodi 3. Lokalisointi Singleton suunnittelumallilla toteutettuna

Näkymiä vaihtava menuvalikko on luotu käyttämällä Abstract Factory suunnittelumallia kuvan 1 mukaisesti. Sovelluksessamme on käytössä kaksi käyttäjäryhmää, jotka ovat johtaja ja varastotyöntekijä. Kummallakin käyttäjäryhmällä on omat näkymänsä, jotka ne näkevät. Alkuperäisessä toteutuksessa teimme molemmille käyttäjille omat pohjanäkymänsä, joissa oli muuten samanlaiset menuvalikot, mutta näkymien vaihtolinkit olivat erilaiset.



Kuva 1. Näkymien menuvalikon luominen Abstract Factory suunnittelumallilla.

Abstract Factoryn periaatteena on tarjota rajapinta olioperheiden tai toisistaan riippuvien olioiden luontiin ilman, että näiden konkreettisia luokkia tarvitsee määritellä luonnin yhteydessä. Abstract Factoryn avulla saimme luotua yhteen pohjanäkymään (MainView.fxml) navigointivalikon, johon voimme luoda näkymien vaihtamiseen käyttäjäryhmän mukaisen osavalikon, joka sisältää ennalta määräämättömän määrän navigointilinkkejä.

Yhteenveto

Kun suunnittelumallit ovat hanskassa, niin se helpottaa huomattavasti suunnittelutyötä, koska kaikkea ei tarvitse miettiä alusta alkaen. Lisäksi koodi saa rakenteen, joka helposti sisäistettävä. Refaktoroinnissa koodimme monet osat muuttuivat ja sen avulla saimme poistettua koodista paljon toistoa. Olisimme halunneetkin käyttää enemmänkin suunnittelumalleja koodissamme, mutta suunnittelumallien runsauden ja sisäistämiseen kuluneen ajan takia aika loppui yksinkertaisesti kesken. Muun muassa tietokantojen käsittelyssä on paljon samanlaista koodia, joka olisi voitu yhtenäistää. Oli myös hienoa huomata, että ymmärtää paremmin esim. Javan omia toteutuksia, kuten salasanojen salauksessa käyttämämme Base64.getDecoder, jotka on toteutettu suunnittelumalleja käyttämällä. Jatkon kannalta suunnittelumallien tunteminen on erittäin hyödyllistä, kun niitä pääsee hyödyntämään jo ohjelman suunnitteluvaiheesta alkaen..

Lähteet

[1] https://www.csie.ntu.edu.tw/~r95004/Refactoring_improving_the_design_of_existing_code.pdf

[2] https://refactoring.guru/design-patterns/catalog