

PureMVC-ohjelmistokehityksen käyttöönotto



Ammattikorkeakoulun opinnäytetyö

Mediatekniikka

Riihimäki 3.11.2010



Mediatekniikka
Riihimäki

Työn nimi PureMVC-ohjelmistokehyksen käyttöönotto

Tekijä Janne Kauhanen

Ohjaava opettaja Raimo Hälinen

Hyväksytty _____._____.20_____

Hyväksyjä

RIIHIMÄKI
Mediatekniikka
Ohjelmistotekniikka

Tekijä	Janne Kauhanen	Vuosi 2010
Työn nimi	PureMVC-ohjelmistokehyksen käyttöönotto	

TIIVISTELMÄ

Tässä opinnäytetyössä tehtiin ohjeistus PureMVC-ohjelmistokehyksen käyttöönottoon ActionScript 3.0 ohjelmointikielelle. Työn toimeksiantajana oli Hämeen ammattikorkeakoulun Riihimäen yksikkö ja työ tehtiin Mediatekniikan verstaan tarpeisiin.

Työn tavoitteena oli luoda ohjeistus, jonka avulla mediatekniikan verstaan uudet työntekijät voivat ottaa käyttöön PureMVC-Ohjelmistokehyksen ohjelmointiprojektissa. Lisäksi työn tehtävänä oli selvittää MVC-mallin taustalla olevaa teoriaa sekä esitellä hyödyllisiä ohjelmointityökaluja.

Teoriaosuudessa selvitettiin MVC-arkkitehtuurin historiaa sekä toimintaperiaatetta. Osuudessa myös tutustuttiin PureMVC-ohjelmistokehyksen toimintaan, suunnittelumallien ja MVC-mallin teorian kautta.

Opinnäytetyössä tehtiin kohta kohdalta opastaen esimerkkisovellus PureMVC-kehystä hyväksi käyttäen. Esimerkkisovellus ohjelmoitiin Adobe AIR-ympäristöön, AS3.0-ohjelmointikielellä käyttäen Adobe Flex-kirjastoja.

Lopputuloksena syntyi ohjeistus, joka tutustuttaa lukijan MVC-arkkitehtuuriin, suunnittelumalleihin sekä havainnollisesti opettaa miten ottaa käyttöön PureMVC, MVC-arkkitehtuurin toteutus. Ohjeistuksessa esitellään myös lyhyesti teknologiavaihtoehtoja, kehitysympäristöjä sekä hyödyllisiä sovellustyökaluja.

Avainsanat MVC, ohjelmistokehys, PureMVC, suunnittelumallit, Adobe Flex

Sivut 42 s, + liitteet 7 s.

RIIHIMÄKI
Media Technology
Software development

Author	Janne Kauhanen	Year 2010
Subject of Bachelor's thesis	Deploying PureMVC	

ABSTRACT

The main aim of this Bachelor's thesis was to develop a user guide for the PureMVC-software framework for the ActionScript 3.0 programming languages. This user guide was commissioned by The Media technology workshop at HAMK University of Applied Sciences, Riihimäki.

The second objective of the thesis was to create a practical guide for new employees of the Media technology workshop. The third objective was to clarify the theory behind the MVC-model and introduce useful software tools.

The theory chapter introduces a history of the MVC-model and some functional principles of the MVC-pattern. The case chapter includes short examples, which are used, in order to show how to use the PureMVC-framework in the software projects. The software examples are programmed using Action Script3.0 language, Flex-framework and Adobe Air runtime.

The outcome of the thesis is a practical user guide for PureMVC-architecture and design models. It explains how to develop an application with using PureMVC-framework. The user guide briefly introduces different kinds of technological alternatives, which are available for use in software projects.

Keywords MVC, framework, PureMVC, design pattern, Adobe Flex

Pages 42 p + appendices 7 p.

SISÄLLYS

1	JOHDANTO.....	1
2	SUUNNITTELUMALLIT	2
2.1	Singleton-suunnittelumalli	2
2.2	Observer-suunnittelumalli	2
2.3	Composite-suunnittelumalli	3
2.4	Delegation-suunnittelumalli	4
2.5	Command-suunnittelumalli	5
2.6	Mediator-suunnittelumalli	5
2.7	Proxy-suunnittelumalli	6
2.8	Façade-suunnittelumalli	7
3	SOVELLUSARKKITEHTUURI	8
3.1	MVC-arkkitehtuuri.....	8
3.1.1	Malli	8
3.1.2	Näkymä.....	9
3.1.3	Ohjain	9
3.1.4	Arkkitehtuurin kommunikaatio ja toiminta	9
3.2	Ohjelmistokehys.....	10
3.3	PureMVC	10
3.4	Malli	12
3.5	Proxy	13
3.6	Näkymä	14
3.7	Mediator	14
3.8	Ohjain.....	15
3.9	Command	16
3.10	Facade.....	17
3.11	PureMVC-kehiksen kommunikaatio	18
3.11.1	Observer	20
3.11.2	Notification.....	20
4	SOVELLUSTYÖKALUT	21
4.1	Kehitysympäristöt	21
4.1.1	FlashDevelop	21
4.1.2	Flash Builder	21
4.1.3	Adobe Flex 4	22
4.1.4	Adobe AIR.....	22
4.1.5	SVN-palvelimen esittely	22
5	PUREMVC-TOTEUTUS.....	24
5.1	Työkalujen asennus	24
5.2	PureMVC asennus ja käyttöönotto.....	24
5.3	Esimerkkisovelluksen toteutus PureMVC-kehyksellä.....	25

5.3.1	Kansiorakenne	25
5.3.2	PureMVC-luokkien luominen	26
5.4	Sovelluksen ohjelmointi luokittain.....	27
5.4.1	Main.mxml	28
5.4.2	ApplicationFacade.as.....	29
5.4.3	StartupCommand.as.....	30
5.4.4	TextAreaProxy.as	30
5.4.5	TextAreaViewComponent.mxml	32
5.4.6	TextAreaMediator.as	34
5.4.7	TexAreaCommand.as	37
5.4.8	Sovelluksen toiminta	38
6	YHTEENVETO	40
6.1	Arkkitehtuurin hyödyt	40
6.2	Arkkitehtuurin ongelmat	40
6.3	Jatkokehitys	41
	LÄHTEET	42

Liite 1 Esimerkkisovelluksen lähdekoodi luokittain

1 JOHDANTO

Riihimäen HAMK:ssa toimivalla mediatekniikan verstaalla opiskelijat tekevät erilaisia multimedian sekä ohjelmistotekniikan projektitöitä, joissa tuotetaan ohjelmointikoodia. Projekteja tehdään monille eri toimeksiantajille eri puolille Suomea.

Mediaverstaalla käytettäviä ohjelmointikieliä ovat mm. ActionScript 3.0, PHP, Adobe Flex ja Javascript. Ohjelmointiprojektien suuruus vaihtelee, koodirivien määrissä mitattuna, muutaman sadan rivin mittaisesta usean tuhannen mittaiseen.

Mediaverstaan projekteissa tuotettua ohjelmointikoodia on tarve päivittää sekä käyttää uudelleen tulevissa projekteissa. Edellä mainittu tarve tuo ongelmia ja haasteita verstaan projekteihin, koska mediaverstaalla ei ole yleisesti käytössä olevaa ohjelma-arkkitehtuuria.

Yleisimmät vastaan tulleet ongelmat vanhojen verstaalla tehtyjen ohjelmien päivittämisessä ovat olleet mm. luokkien vääränlainen käyttö, liian isot luokat, vaikeasti ymmärrettävä logiikka ja ohjelmakoodin muokkaamisen vaikeus. Näiden ongelmien ratkaisemiseksi projekteihin tarvitsi tuoda yleisesti hyväksi havaittu ja hyvin dokumentoitu arkkitehtuurityyli, joka on käytössä laajalti.

MVC-mallien tuntemus on erittäin kysyttyä työelämässä, sillä esimerkiksi useat eri nettisivut käyttävät MVC-ohjelmistokehystä. Arkkitehtuuri on myös hyväksi havaittu graafisten työpöytäsovellusten suunnittelussa ja ohjelmoinnissa.

Tämän opinnäytetyön tarkoituksena on palvella mediatekniikan verstaan ohjelmointiprojekteja kehittämällä opiskelijoiden käyttämiä ohjelmointitapoja. Lisäksi tarkoituksena on standardisoida verstaan ohjelmointimalleja sekä syventää ohjelmoijien henkilökohtaista osaamista olio-ohjelmoinnista.

Opinnäytetyössä esitellään yleistä teoriaa MVC-arkkitehtuurista sekä tutustutaan suunnittelumalleihin. Ohjelmoijaa opastetaan MVC-mallin käyttöönotossa, sekä pohditaan MVC-ohjelmistokehyksen hyötyjä ja haittoja. Lisäksi tutustutaan PureMVC-ohjelmistokehyksen toimintaan ja luokkarakenteeseen. Lopuksi tehdään yksinkertainen esimerkkiohjelma käyttäen PureMVC-kehystä ja pohditaan, missä tapauksessa PureMVC:n käyttö on järkevää.

2 SUUNNITTELMALLIT

Suunnittelumallit ovat yleisiä, uudelleenkäytettäviä ratkaisuja toistuvasti esiintyviin ongelmiin. Suunnittelumalli ei ole valmis muotti tai ratkaisu, joka voidaan muuntaa suoraan ohjelmakoodiksi vaan se on kuvaus uudelleen sovellettavasta ongelmanratkaisutavasta. (Wikipedia 2010a.)

Tässä kappaleessa esitellään erilaisia, yleisesti käytettyjä suunnittelumalleja, jotka nousivat esille tämän opinnäytetyön aikana tai kuuluvat muuten oleellisesti osana MVC-arkkitehtuuriin.

2.1 Singleton-suunnittelumalli

Vaikka Singleton-malli ei välttämättä kuulu osana MVC-arkkitehtuuriin, sen esittely on perusteltua tässä opinnäytetyössä. Singleton kuuluu olennaisesti PureMVC-kehykseen ja siihen viitataan opinnäytetyössä useasti, jolloin mallin esittely on järkevää asian selkeyttämiseksi.

Singleton-mallissa on kaksi avainominaisuutta: Singleton-luokasta voidaan ottaa vain yksi ilmentymä sekä luokalla tulee olla yksi globaali muuttuja, jonka kautta hallita luokkaa.

Singleton-mallia käytetään todella paljon, sillä melkein jokaisessa ohjelmassa on tarve Singleton-suunnittelumallille. Esimerkiksi ohjelmoitaessa jonkinlaista peliä, missä tarvitaan pistelaskusta huolehtivaa luokkaa, tarvitaan yleensä vain yksi kappale pistelaskureita. (Sanders, Cumaranatunge, ActionScript 3.0 design Patterns, 2007, 101.)

2.2 Observer-suunnittelumalli

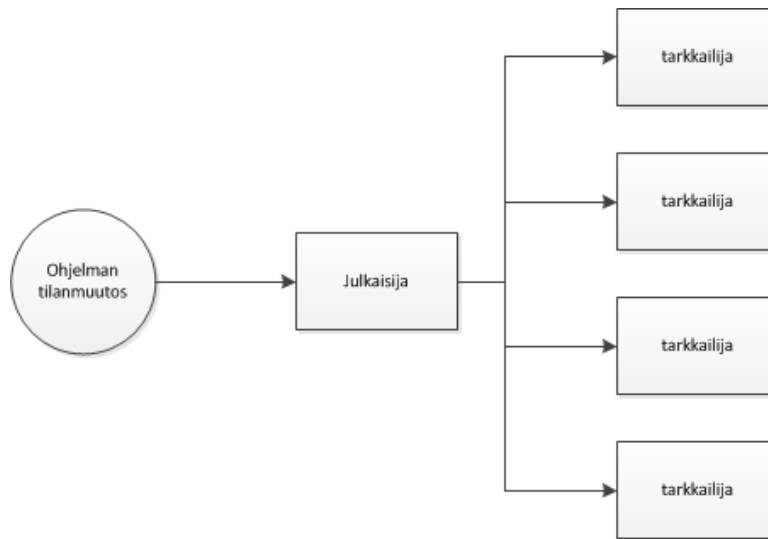
Tarkkailija-mallissa informaatiot jaetaan keskitetysti yhdestä paikasta usealle tilaajalle. Toiminta on samanlaista kuten esimerkiksi sanomalehden tilaus, sillä tilaajia on useita, mutta julkaisijoita vain yksi, jolta lehti tilataan. Jos ohjelman eri osille täytyy lähettää sama informaatio, on yhden, keskitetyn lähettäjän käyttö järkevämpää kuin saman tiedon lähettäminen usean eri luokan toimesta. Lisähyötynä Tarkkailija-mallin käytöstä tulee takuu siitä, että kaikille vastaanottajille menee sama informaatio.

Tarkkailija-mallista löytyy neljä tyypillistä ominaisuutta:

- sama informaatio lähetetään kaikille kuuntelijoille
- julkaisija voi rekisteröidä ja poistaa kuuntelijoita,
- julkaisijoita on vain yksi ja kuuntelijoita on ääretön määrä.

(Sanders, Cumaranatunge, ActionScript 3.0 design Patterns, 2007, 284.)

Kuvan 1 tarkkailija-mallin kaaviossa ohjelman tilanmuutoksesta menee tieto julkaisijalle, joka lähettää informaation muutoksesta kaikille tarkkailijoille.



Kuva 1 Tarkkailija-mallin toiminta.

2.3 Composite-suunnittelumalli

Rekursiokooste-malli tarjoaa toimivan ratkaisun pienistä komponenteista koostuvien isojen järjestelmien rakentamiseen. Esimerkkinä voidaan käyttää autoa, joka koostuu komponenteista kuten moottorista, renkaista, rungosta, istuimista jne.

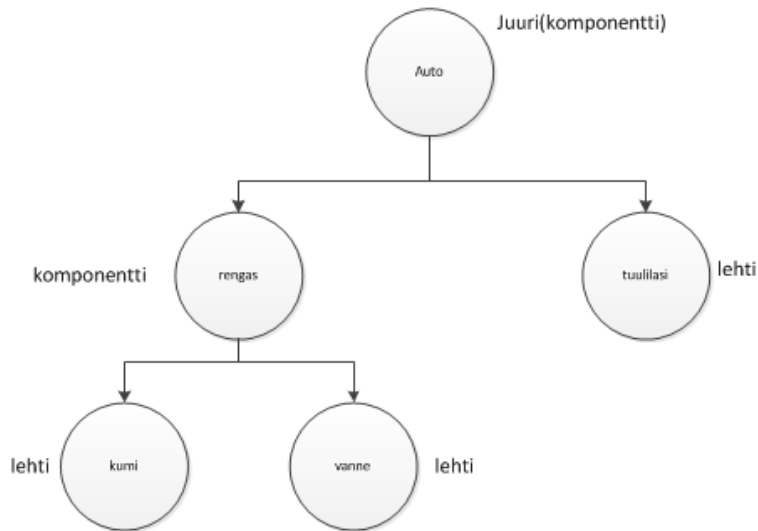
Rekursiokooste-mallin hyötynä on, että käyttäjä voi käsitellä kaikkia järjestelmän osia saman rajapinnan kautta. Esimerkiksi lisättäessä tai poistettaessa osia autosta, ei tarvitse tietää, onko poistettava osa yksittäinen pultti vai kokonainen rengas. Rengasta poistettaessa poistetaan kaikki renkaaseen kuuluvat osat, mukaan lukien pultit.

Rekursiokooste on hierarkkinen puu-rakenteinen malli, jossa komponentit voivat pitää sisällään äärettömän määrän lehtiä tai toisia komponentteja. Lehdet ovat yksittäisiä, primitiivisiä luokkia, jotka eivät pidä sisällään lapsi-objekteja. Lehteä, joka lähtee suoraan komponentista, kutsutaan lapsi-objektiksi (child), ja komponenttia lehden vanhemmaksi (parent). Esimerkiksi vanne on renkaan lapsi-objekti ja koska vanne on renkaan sisällä, on rengas vanteen vanhempi.

Rekursiokoosteella on kolme tyypillistä ominaisuutta:

- monimutkainen objektikokoelma on kasattu hierarkkiseen järjestykseen.
- järjestelmän komponentit voivat olla yksittäisiä objekteja tai kokoelmia useista eri objekteista.
- yksittäisiä luokkia sekä kokoelmaluokkia voidaan käsitellä samoilla komennoilla yhtä rajapintaa käyttäen.

(Sanders, Cumaranatunge, ActionScript 3.0 design Patterns, 2007, 204.)



Kuva 2 Rekursiivisen mallin hierarkkinen puu-rakenne.

2.4 Delegation-suunnittelumalli

Delegaatio-suunnittelumallissa kaksi luokkaa käsittelee yhdessä asiakkaan pyyntöjä. Toinen luokista saa käsiteltäväkseen toimintapyyntön, jonka saatuaan tämä delegoi toimintapyyntön suorittamisen toiselle luokalle. Yksinkertainen koodiesimerkki:

```
class A {
    void foo() {
        // "this" is also known under the names "current", "me" and "self" in
        // other languages
        this.bar()
    }

    void bar() {
        print("a.bar")
    }
}

class B {
    private A a; // delegationlink

    public B(A a)
    {
        this.a = a;
    }

    void foo() {
        a.foo() // call foo() on the a-instance
    }

    void bar() {
        print("b.bar")
    }
}

a = new A()
b = new B(a) // establish delegation between two objects
```

Kuva 3 koodiesimerkki delegaatiosta oliokielellä.

Kuvassa 3 luodaan kaksi luokkaa, A ja B, joihin molempiin luodaan samat metodit foo ja bar. B-luokalle annetaan viittaus A-luokan instanssiin, jolloin B-luokka delegoi Foo-metodillaan kirjoittamisen A-luokalle yksityisen a-muuttujan kautta.

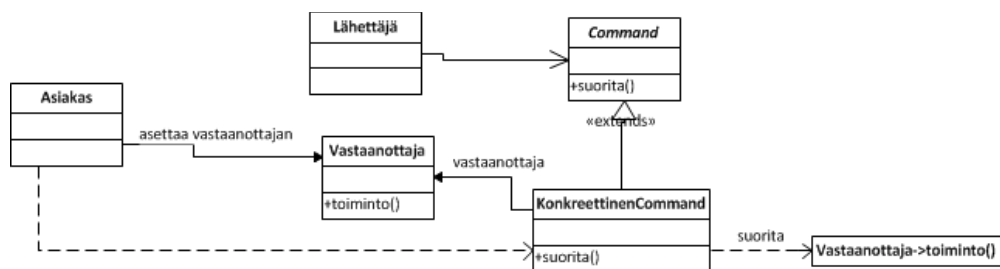
Delegaation etuna on mahdollisuus muuttaa koodin rakennetta ajon aikana. Delegointi tekee olioiden koostamisesta yhtä tehokkaan vaihtoehdon kuin periyttäminen, koska delegaatioon osallistuvilla luokilla on viittaukset toistensa metodeihin.

2.5 Command-suunnittelumalli

Command-suunnittelumallin tarkoituksena on kapseloida pyyntö olioksi, joka sisältää kaiken tarpeellisen informaation komennon myöhempää suorittamista varten. Kolme termiä, jotka esiintyvät aina Command-suunnittelumallissa ovat asiakas, lähettäjä ja vastaanottaja. Asiakas ottaa Command-luokasta instanssin ja antaa instanssille informaation komennon kutsumiselle myöhemmin. Lähettäjä on instanssi, joka päättää milloin komentoa kutsutaan. Vastaanottaja-luokka pitää sisällään suoritettavan ohjelmakoodin. (Wikipedia 2010b.)

Command-suunnittelumallin avulla asiakas voidaan parametroida lähettämään erilaisia pyyntöjä, laittaa pyynnöt jonoon, pitää niistä lokia ja peruuttaa operaatioita. (Gamma, Helm, Johnson & Vlissides 2001, 233.)

Kuvassa 4 on Command-suunnittelumallin rakenne. Command-rajapintaluokka määrittelee funktiot, jotka konkreettisen Command-luokan tulee toteuttaa. Konkreettinen Command-luokka sitoo yhteen vastaanottajan sekä toiminnon. Asiakas ottaa instanssin konkreettisesta Commandista, asettaa luokalle vastaanottajan ja lähettäjä käskää komentoa toteuttamaan toiminnon.



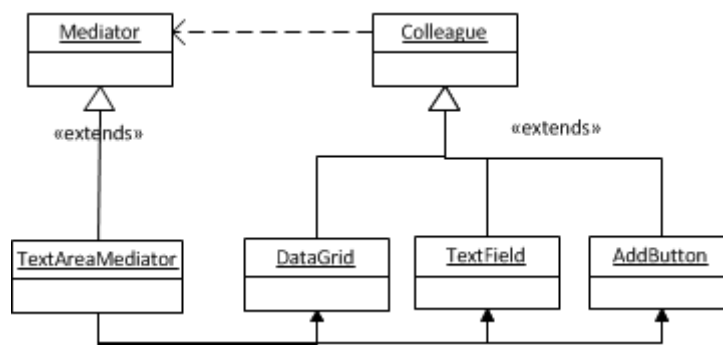
Kuva 4 Command-suunnittelumalli

2.6 Mediator-suunnittelumalli

Oliopohjainen sovellus on yleensä jaettu useaan luokkaan, joiden välisestä yhteistyöstä syntyy sovelluksen logiikka ja toiminta. Luokkien määrän

kasvaessa suureksi, niiden välinen kommunikaatio saattaa muuttua pirstaloituneeksi sekä monimutkaisemmaksi, mikä vaikeuttaa ohjelman ylläpitämistä. Lisäksi muutosten tekemisestä tulee ongelmallista, koska muutokset vaikuttavat useaan eri ohjelman kohtaan. Mediator-suunnittelumalli ratkaisee tämän ongelman kapseloimalla kommunikaation Mediator-objektin sisään. Ohjelman eri luokat eivät enää kommunikoi suoraan keskenään, vaan hoitavat kommunikoinnin välittäjän (mediator) kautta. Tämä vähentää luokkien välisiä riippuvuuksia ja yhteyksiä tehden ohjelmasta dynaamisemman. (Gamma, Helm, Johnson & Vlissides 2001, 273-275.)

Esimerkkinä voidaan ottaa opinnäytetyön käytännön osuudessa tehtävä sovelluksen käyttöliittymä. Kuvassa 5 Mediatorista luotu konkreettinen TextAreaMediator-luokka kuuntelee DataGrid-, TextField- ja AddButton-luokkia, jotka luovat ohjelman käyttöliittymän. Kaikki edellä mainitut luokat ilmoittavat muutoksista TextAreaMediator-luokalle, joka välittää muutokset eteenpäin PureMVC-kehykselle.



Kuva 5 Mediator-suunnittelumallin rakenne

2.7 Proxy-suunnittelumalli

Proxy-suunnittelumalli on yksinkertaisimmillaan luokka, joka toimii rajapintana joillekin muille resursseille. Proxy voi luoda rajapinnan mille tahansa. Se voi toimia rajapintana esim. yhteydelle etäpalvelimeen, isolle luokalle muistissa, tiedostolle kovalevyllä, tai jollekin muulle tapahtumalle, jonka kopioiminen on resursseja kuluttavaa tai mahdotonta. (Wikipedia 2010c.)

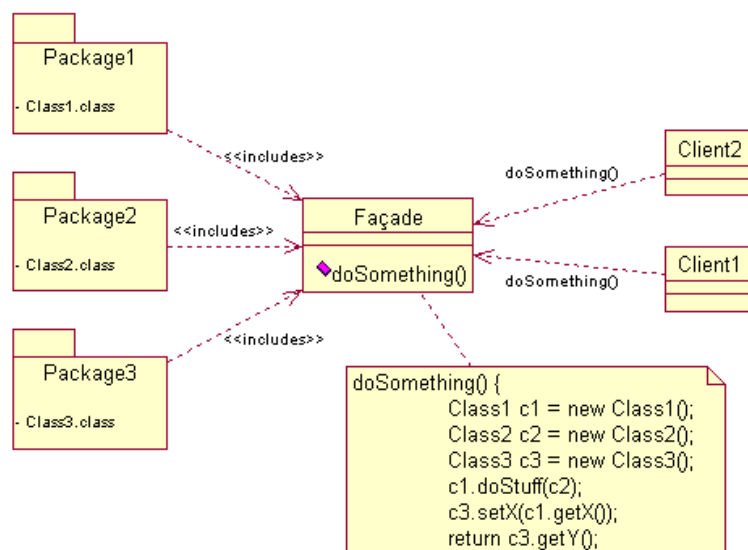
Usein on tarpeellista kontrolloida olioon kohdistuvia kutsuja, jotta olion luonnista ja alustuksesta aiheutuvat kustannukset voidaan lykätä siihen saakka kunnes oliota tarvitaan.

Esimerkkinä Proxy-mallista voidaan ottaa tekstieditori, joka käsittelee grafiikkaolioita. Joidenkin grafiikkaolioiden luominen, kuten suurien bittikarttakuvien, voi olla kuluttavaa. Teksti-dokumentin pitää avautua nopeasti, joten on järkevää olla luomatta kaikkia resursseja kuluttavia olioita samanaikaisesti. Olioiden luominen ei ole edes tarpeellista, koska dokumentin kaikki osat eivät näy käyttäjälle yhtäaikaaisesti. Ongelmana on,

mitä kuvan tilalle laitetaan? Oikean kuvan tilalla käytetään vähemmän muistia ja prosessoriakaa kuluttavaa oliota eli edustajaa (Proxy). Edustaja toimii samalla tavalla kuin oikea kuva ja huolehtii oikean kuvan lataamisesta kun sitä tarvitaan. (Gamma, Helm, Johnson & Vlissides 2001, 207-209.)

2.8 Façade-suunnittelumalli

Facaden tarkoituksena on tarjota rajapinta alijärjestelmän rajapintojen joukolle. Facade tarjoaa korkeamman tason rajapinnan, jonka kautta aliluokkien metodeja voi käyttää.



Kuva 6 Façade-luokan rakenne

Kuvassa 6 Façade-luokkaa käyttävät Client2 ja Client1-luokat pääsevät käyttämään package1-, package2- ja package3-luokkia yksinkertaistetusti Facaden kautta. Asiakas-luokat Client2 ja Client1 eivät ole sidottuja suoraan Class1-, Class2- ja Class3-aliluokkiin vaan suorittavat haluamansa toiminnot Facaden kautta. Façade-suunnittelumallin etuina on luokkien käytön helpottuminen. Koska alijärjestelmät yleensä ohjelman laajetessa monimutkaistuvat, Façade helpottaa alijärjestelmien yhteiskäyttöä. (Wikipedia 2010c.)

3 SOVELLUSARKKITEHTUURI

Sovellusarkkitehtuuri on osa nykyaikaista ohjelmistotuotantoa. Sovellusarkkitehtuuri määrittelee ohjelmointitavan, miten ohjelmoijien tulee sovelluksensa rakentaa. Arkkitehtuurin päämääränä yleisesti on ison sovelluksen osiin jakaminen, jolloin ohjelman ylläpito ja muokattavuus paranevat. Sovelluksen osiin jakamisella saadaan ohjelmointityö jaettua helpommin eri työntekijöille, ja lisäksi samankaltaisten ohjelmien sarjatuotanto nopeutuu.

Sovellusarkkitehtuureja on useita erilaisia, mutta tässä opinnäytetyössä paneudutaan MVC-arkkitehtuuriin ja sen toteutuksiin. Arkkitehtuuria on kuvattu järjestelmän perustuslaiksi, jota tulee noudattaa järjestelmää rakennettaessa, ja jonka muuttaminen ei ole suotavaa. Vertaus kertoo hyvin yhden arkkitehtuurin päämääristä eli se asettaa ohjelmoijille säännöt joiden puitteissa toimia, jolloin ohjelmoijien on helpompi ymmärtää toistensa ohjelmakoodia. (Koskimies & Mikkonen, Ohjelmistoarkkitehtuurit 2005, 18 - 19.)

3.1 MVC-arkkitehtuuri

MVC-arkkitehtuurin nimi tulee sanoista model-view-controller, joka tarkoittaa mallia, näkymää ja ohjainta. MVC-arkkitehtuuri on yhdistelmäarkkitehtuuri, jossa sovellus nimensä mukaisesti jaetaan kolmeen pääosiin. Malli, näkymä ja ohjain, joista jokainen huolehtii sovelluksen eri osa-alueiden hallinnasta. Yhdistelmäarkkitehtuurin MVC-mallista tekee usean eri suunnittelumallin yhteiskäyttö osana isoa sovellusta. (Sanders & Cumaranatunge, ActionScript 3.0 design Patterns, 2007, 427.) Näihin MVC-arkkitehtuurin osioihin ja suunnittelumalleihin tutustutaan paremmin myöhemmissä kappaleissa.

Arkkitehtuurimallin kehitti alunperin Trygve Reenskaug Xerox PARC:ssa Kaliforniassa ohjelmointikielelle nimeltä Smalltalk. MVC-arkkitehtuuri on kehittämisen jälkeen siirtynyt laajaan käyttöön ja se on suosittu erityisesti graafisten käyttöliittymien sekä web-sivujen ohjelmoinnissa. MVC-arkkitehtuuri on osoittautunut erittäin hyväksi tavaksi toteuttaa graafisia käyttöliittymiä. Tästä syystä siitä on tehty toteutuksia useille eri kielille ja alustoille (Walther, ASP.NET MVC Framework unleashed, 2010.)

3.1.1 Malli

MVC-arkkitehtuurissa malli vastaa ohjelman tilasta. Se käsittelee Sovelluksen datahakujen logiikkaa ja ilmoittaa näkymälle datan muuttumisesta. Malli käyttää hyväkseen Tarkkailija-suunnittelumallia, kertoakseen tilanmuutoksista siitä kiinnostuneille luokille. (Sanders & Cumaranatunge, 2007, 428.)

3.1.2 Näkymä

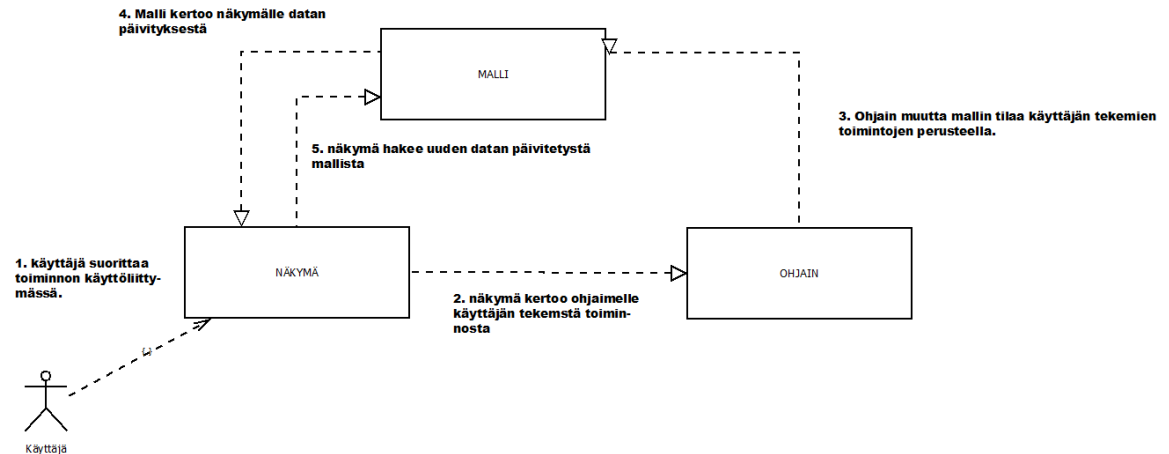
Näkymä on komponenteista rakennettu käyttäjälle näkyvä ohjelman osakokonaisuus, jonka kautta käyttäjä ohjaa sovellusta haluamallaan tavalla. Näkymä voi koostua monenlaisista eri komponenteista. Se voi olla esimerkiksi käyttäjän syötettä vastaanottava tekstikenttä tai kaiuttimista kuuluvaa musiikkia. (Sanders & Cumaratunge, ActionScript 3.0 design Patterns, 2007, 428.)

3.1.3 Ohjain

Ohjain on MVC-arkkitehtuurin osa, joka vastaanottaa ja käsittelee käyttäjän toiminnot. Lisäksi ohjain päättää, miten sovellus reagoi käyttäjän syötteisiin.

Esimerkiksi jos sovellus on musiikkisoitin, joka on rakennettu käyttäen MVC-arkkitehtuuria ja näkymässä on äänenvoimakkuuden säätöpainikkeet. Ohjaimen tehtävänä on päättää, kuinka paljon äänentasa nostetaan ja lasketaan yhdellä säätöpainikkeen painamisella. (Sanders & Cumaratunge, ActionScript 3.0 design Patterns, 2007, 428.)

3.1.4 Arkkitehtuurin kommunikaatio ja toiminta



Kuva 7 kaaviokuva MVC-mallin kommunikaatiosta

Malli, näkymä ja ohjain kommunikoivat keskenään erilaisilla tavoilla MVC-arkkitehtuurissa, joten luokkien täytyy olla tietoisia toisistaan eri tavoilla.

Mallin täytyy kertoa näkymälle datan päivityksestä, jolloin mallin tulee kommunikoida näkymän kanssa.

Näkymän tulee päivittää itseään mallin datan muuttuessa ja näkymän tulee pystyä kertomaan ohjaimelle käyttäjän interaktioista. Siksi näkymän on kommunikoitava mallin ja ohjaimen kanssa.

Kuvan 7 kaaviossa käyttäjä käynnistää tapahtumat suorittamalla toiminnon käyttöliittymässä, joka on osa näkymää. Interaktion jälkeen näkymä kertoo ohjaimelle käyttäjän suorittamasta toiminnosta. Ohjain saa tiedon tapahtuneesta toiminnosta ja tämän perusteella kertoo mallille datan päivitystarpeesta. Malli päivittää datan ja ilmoittaa päivitystapahtumasta näkymälle, joka hakee päivitetyn datan mallilta. (Sanders & Cumaranatunge, ActionScript 3.0 design Patterns, 2007, 428 - 429.)

3.2 Ohjelmistokehys

Ohjelmistokehykset ovat olio-ohjelmoinnille tyypillinen tapa toteuttaa sovelluksen runko. Kehyksen perustavoitteena on ohjelmistojen ja ohjelmakoodin laajamittainen ja systemaattinen uudelleenkäyttö. Tämä nopeuttaa ohjelmistojen sarjatuotantoa.

Ohjelmistokehys voidaan ymmärtää sovelluksen runkona, joka sisältää aukkoja. Kehys itsessään ei ole toimiva sovellus vaan siitä tehdään toimiva lisäämällä aukkoihin uusi koodi.

Konkreettisella tasolla olioperustainen ohjelmistokehys on kokoelma luokkia, komponentteja ja rajapintoja, jotka toteuttavat ohjelmistojoukon arkkitehtuurin ja perustoiminnallisuuden.

Kehysten käytön kannattavuus tulee esille parhaiten juuri systemaattisesti tapahtuvassa hyödyntämisessä, jolloin tehdään useita samantyyppisiä ohjelmia.

Huonona puolena ohjelmistojen uudelleenkäytöllä on suoritustehon laskeminen. Yleiskäyttöiset sovellusratkaisut eivät koskaan voi olla yhtä tehokkaita kuin tiettyä tarkoitusta varten räätälöidyt ratkaisut. (Koskimies & Mikkonen, Ohjelmistoarkkitehtuurit 2005, 188.)

3.3 PureMVC

PureMVC on kevyt ohjelmistokehys, joka perustuu suunnittelultaan MVC-arkkitehtuuriin. PureMVC-ohjelmistokehys on vapaata lähdekoodia ja alun perin ilmestyi ActionScript 3.0 -ohjelmointikielelle.

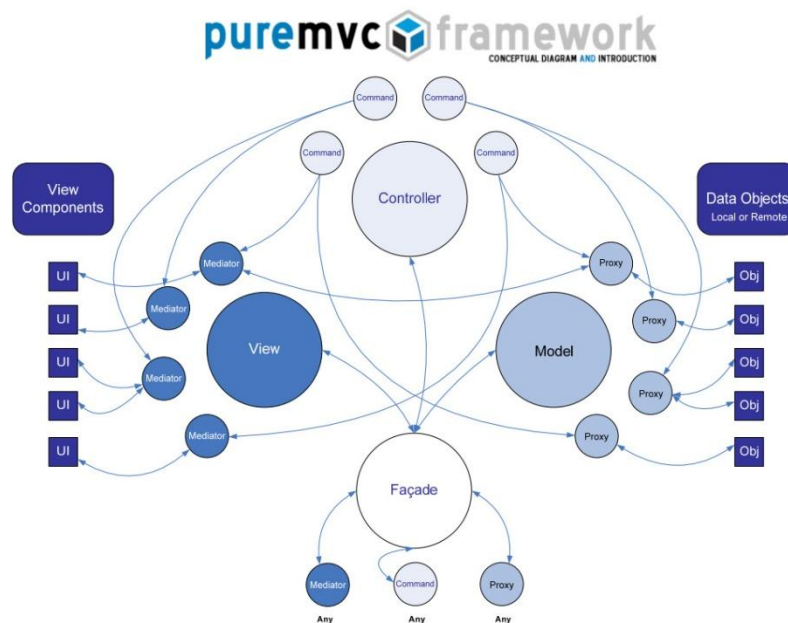
PureMVC-ohjelmistokehyksen ainoana päämääränä on jakaa ohjelmakoodi kolmeen tasoon, malliin, näkymään ja ohjaimeen. Tässä MVC-arkkitehtuurin toteutuksessa, kerroksia edustaa kolme Singleton-luokkaa, joiden yhteistoimintaa helpottaamaan on tuotu Facade-luokka. Nimensä mukaisesti Facade on Facade-suunnittelumallin toteutus.

PureMVC-toteutuksessa ohjelmoija ei koskaan käsittele suoraan mallin, ohjaimen tai näkymän Singleton-luokkia, vaan käsittely tapahtuu Facaden tarjoaman rajapinnan kautta. Ohjelmistokehyksen luokkien välinen

kommunikaatio tapahtuu käyttämällä Notifikaatio-rajapintaa, jonka sekä näkymä, malli, Facade ja ohjain toteuttavat. (Hall 2008.)

Kuvassa 8 olevasta käsitekaaviosta näkyy koko ohjelmistokehyksen rakenne. Näkymä, ohjain ja malli ovat Singleton-luokkia, joista otetaan vain yksi ilmentymä. Facade-luokka on Singleton, joka alustaa ja pitää sisällään viittaukset malliin, ohjaimeen ja näkymään. Lisäksi Facade tarjoaa rajapinnan, jonka kautta pääsee käsittelemään näkymän, ohjaimen ja mallin julkisia metodeja.

Näkymän, mallin ja ohjaimen Singleton-luokat pitävät sisällään useita luokkia. Näkymä, pitää sisällään Mediator-ilmentymiä, malli Proxy-ilmentymiä ja ohjain Command-ilmentymiä. Näihin ilmentymiin PureMVC:n käyttäjä kirjoittaa ohjelmakoodinsa, jakaen erityyppisen koodin eri tasoihin. (Hall 2008)



Kuva 8 PureMVC-ohjelmistokehyksen käsitekaavio

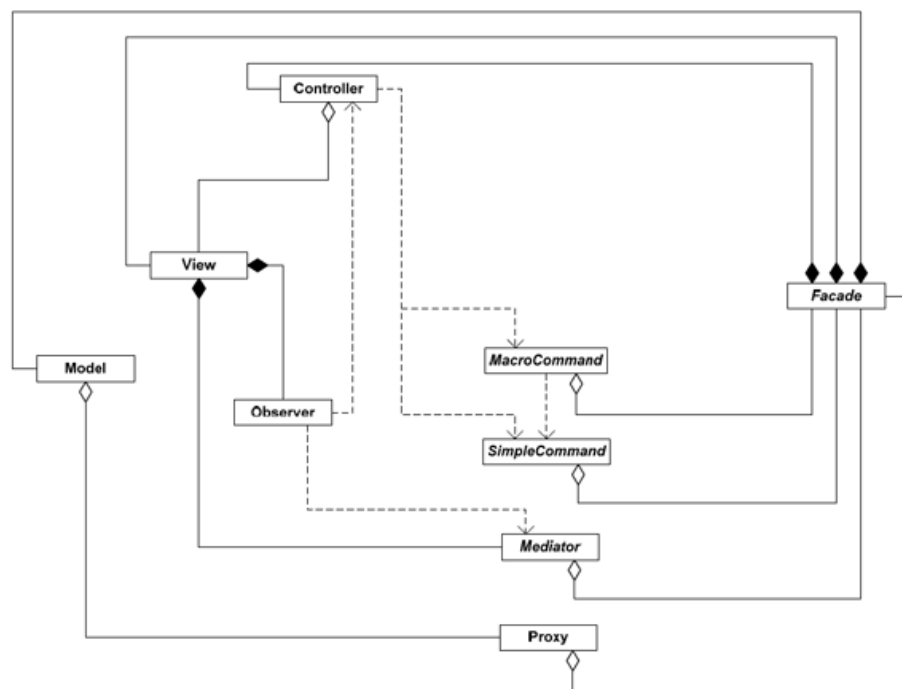
Kuvan 8 kaavion mukaisesti Facade-luokka pitää sisällään Controller-, View- ja Model-luokat, jotka luodaan ohjelman käynnistyessä. Controller-luokka pitää sisällään listan Notifikaatioista ja Command-luokkien viittauksista, jotka on linkitetty toisiinsa.

View-luokka pitää sisällään ilmentymiä Mediator-luokista, ja Model-luokka sisältää Proxy-luokan ilmentymiä. Kaikilla Command-, Mediator- sekä Proxy-luokilla on viittaus Facade-luokkaan, joka tarjoaa rajapinnan Notifikaatioiden lähettämiseksi mistä tahansa Command-, Mediator- tai Proxy-luokasta.

Notifikaatiot ovat ohjelmistokehyksen tärkein kommunikointitapa. Notifikaatioiden tehtävänä on lähettää tietoa ohjelman tapahtumista eri kerroksille, esim. näkymässä tapahtuvasta hiiren painalluksesta lähetetään

tieto ohjaimelle, joka käynnistää Notifikaatiolle rekisteröidyn komennon (Command).

On erittäin tärkeää, että sovellusta toteutettaessa PureMVC-kehyksellä, luokkien välinen kommunikaatio tapahtuu Notifikaatioita käyttämällä. Notifikaatioihin perustuva kommunikaatio pitää sovelluksen eri osat löysästi sidottuina, jolloin jälkepäin tehtävät muutokset onnistuvat helpommin.



Kuva 9 UML-kaavio PureMVC-ohjelmistokehyksen pääluokista.

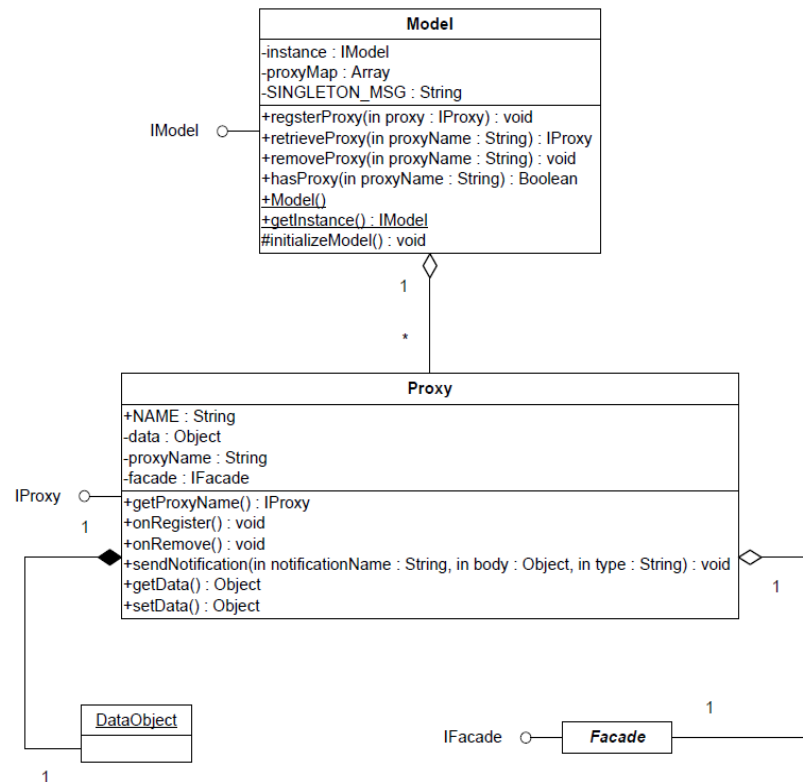
3.4 Malli

Malli-Singleton-luokka tallentaa sisäänsä viittaukset IProxy-rajapinnan toteuttaviin luokkiin. IProxy on rajapinta, joka tarjoaa toiminnallisuuden sovelluksen datan tallennusta, muokkausta ja hakua varten.

Kuten kuvassa 9 olevasta kaaviosta näkyy, malli ei ole yhteydessä suoraan mihinkään muuhun kuin Facade-luokkaan. Model-luokka on rakennettu rekursiokooste-suunnittelumallia käyttäen ja Facade tarjoaa rajapinnan Proxy-luokkien rekisteröinnille. Ohjelmoija kirjoittaa koodinsa Proxy-luokkiin, joihin malli pitää viittaukset taulukkomuuttujassa, ja joiden käsittelystä malli huolehtii.

Kuvassa 10 näkyvässä luokkakaaviossa näkyy kuinka malli ja Proxy liittyvät toisiinsa. Malli-luokka pitää sisällään ainoastaan taulukon Proxyistä, sekä Proxyn rekisteröintiin, noutamiseen ja poistamiseen tarkoitetut funktiot. Proxy-luokat ovat Model-luokan sisällä IProxy-

tyyppisinä luokkina taulukossa. IProxy on rajapintaluokka, joka tulee toteuttaa Proxy-luokkaa tehtäessä.



Kuva 10 Mallin ja Proxyn luokkakaavio

3.5 Proxy

Proxy-luokka on rajapinta, joka auttaa käsittelemään ja paljastamaan tietorakenteita sekä luokkia. Proxy-luokkia käyttämällä tietojen käsittelyyn tarkoitettuja rutiineja on helppo käyttää uudelleen sovelluksen eri osissa. Lisäksi Proxy-luokan datarakenteiden muuttaminen vaikuttaa muun sovelluksen toimintaan äärimmäisen vähän.

Yksinkertaisimmillaan Proxy-luokkaa käytetään esimerkiksi hakemaan tietoa tietokannasta tai palvelimelta. Haun jälkeen tieto kasataan sopivaan muotoon, jonka jälkeen muutoksesta ilmoitetaan siitä kiinnostuneelle näkymälle datapäivitystapahtumasta.

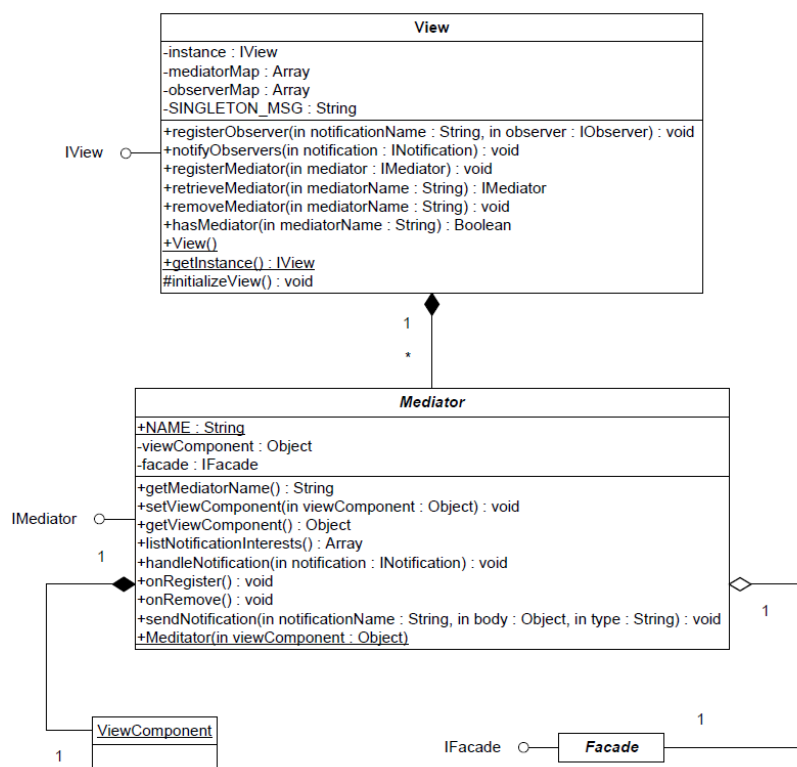
Datan muutos ilmoitetaan päivityksestä kiinnostuneille luokille käyttämällä Proxyn toteuttamaa Notifikaatorajapintaa. Käsitelty data tallennetaan Proxy-luokan data-muuttujaan näkymän noutoa varten. (Hall 2008.)

Kuvan 8 käsitekaaviosta näkyy, että Proxy-luokka pitää viittaukset data-objekteihin, joihin kyseisen Proxy-luokan data kasataan. Dataobjekti voi olla monimutkainen luokka, joka pitää sisällään monen tyyppisiä muuttujia ja luokkia. Dataobjektiin voidaan varastoida esim. taulukoita, kokonaislukuja, luokkia, toisia objekteja jne. Objekti voi myös olla yksinkertaisesti vain JPG-kuvan polku merkkijonona.

Proxy-luokan funktiot sekä muuttujat ovat nähtävissä kuvan 10 luokkakaaviosta. name-vakio on tarkoitettu Proxyn hakemiseen ja tunnistamiseen. Data pitää sisällään objektin, mihin datamalli rakennetaan ja Facade-muuttuja pitää sisällään viittauksen Facade-luokkaan. Facadessa on rekisteröity ohjelman kannalta tärkeimmät Notifikaatiot, ja se toimii rajapintana kaikille PureMVC:n tärkeimmille luokille.

3.6 Näkymä

Näkymä on PureMVC-kehyksessä toteutettu Singleton-tyyppisenä luokkana, joka tallentaa ja käsittelee viittauksia IMediator-rajapinnan toteutuksiin. Lisäksi Näkymä-luokan vastuulla on Tarkkailija (Observer)-luokkien hallinta. Näkymä rekisteröi listan Notifikaatio-Tarkkailija-pareista ja ilmoittaa kaikille Tarkkailija-luokille, kun sopiva Notifikaatio on lähetetty. (Hall 2008.)



Kuva 11 View- ja Mediator-luokan luokkakaavio.

3.7 Mediator

Mediator-luokka, joka on rekursiokooste, auttaa sovelluksen käyttöliittymän käsittelyssä sekä kommunikoi käyttöliittymän ja PureMVC:n välillä. Sovelluksen näkymä koostuu

käyttöliittymäkomponenteista, jotka paljastavat datan käyttäjälle ja kuuntelevat käyttäjän toimia. Näkymän komponentit eivät ole tietoisia PureMVC-kehyksestä ympärillään, vaikka Mediator-luokalla on viittaus komponentteihin.

Mediator-luokka kuuntelee näkymän tapahtumia käyttämällä Flash Events-luokkakokoelman tapahtuman lähetyksiä ja tapahtumankuuntelijoita. Mediatorin tehtävänä onkin yhdistää näkymään haluttu data sekä kommunikoida muun sovelluksen kanssa näkymän käyttöliittymän komponenttien puolesta.

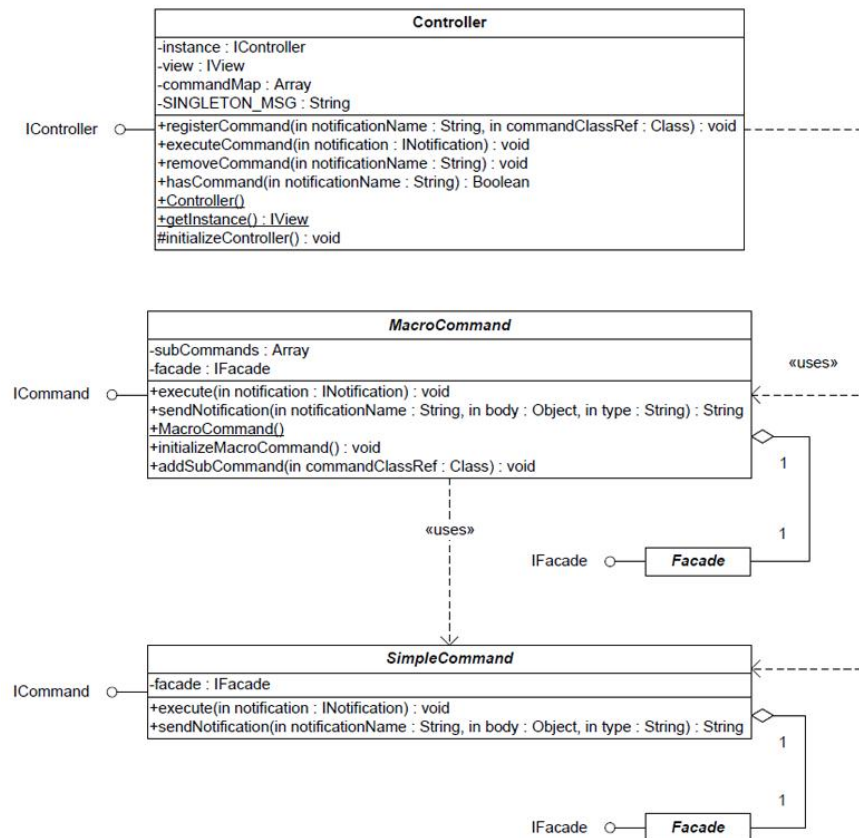
Mediator-luokka pitää sisällään yhden tai useamman näkymän komponentin. Näkymän komponentit ovat käyttäjälle näkyviä käyttöliittymän osia, joita käyttäjä käsittelee. (Hall 2008.)

Rekisteröintivaiheessa Mediatorilta tiedustellaan, mistä Notifikaatioista se on kiinnostunut, suorittamalla Mediatorin listNotifications-funktion, joka palauttaa taulukon Notifikaatioiden nimistä ja yhdistää niihin Notifikaatioista kiinnostuneet Mediattorit

Kuvassa 11 näkyvässä luokkakaaviossa ViewComponent-luokka on Mediator-luokan hallinnoima käyttäjälle näkyvän käyttöliittymän osa. Yleensä esimerkiksi Adobe Flex-sovelluksessa ViewComponent on mxml-tiedosto, jota käyttäjä käyttää ja Mediator kuuntelee sekä välittää käyttäjän käyttötapaukset ohjaimen hallittavaksi.

3.8 Ohjain

PureMVC-ohjelmistokehyksen ohjain-luokka sisältää kartoituksen tilattomista Command-luokista. Ohjain-luokka on singleton tyyppinen luokka ja se alustetaan ohjelman käynnistämisvaiheessa, jolloin myös rekisteröidään kaikki sovelluksen käynnistämisessä käytettävät Command-luokat.



Kuva 12 Luokkakaavio ohjaimesta sekä MacroCommand- ja SimpleCommand-luokista

3.9 Command

Command-luokat ovat luokkia, joissa suoritetaan ohjelman tehtävälogiikka. Ohjelman alustusvaiheessa kaikki sovelluksessa käytettävät Command-luokat sidotaan johonkin Notifikaatioon. Kun Command-luokka sidotaan Notifikaatioon, ohjain-luokka rekisteröi itselleen tarkkailijan annetulle Notifikaatiolle. Kun Notifikaatio suoritetaan, ohjain ottaa ilmentymän siihen sidotusta Command-luokasta ja suorittaa luokan sisällä olevan ohjelmakoodin. Command-luokat ovat tilattomia, ne syntyvät vain kun niitä kutsutaan, ja ne kuolevat suorituksen jälkeen.

Command-luokkien käyttötarkoituksena on monimutkaisien järjestelmän tilanmuutosten tekeminen halutussa järjestyksessä. Esimerkiksi Command-luokka voi rekisteröidä Proxy-luokan ja Mediatorin halutussa järjestyksessä. Rekisteröinnin jälkeen Command voi käyttää Proxy:n ilmentymää suoraan, suorittamalla Proxy-luokan julkisia funktioita.

Command voi esimerkiksi ottaa vastaan käyttäjän kirjautumisen, käskää Proxy:n ilmentymää tarkastamaan salasanan ja hakemaan tunnistetun käyttäjän tiedot tietokannasta. Salasan tarkastuksen jälkeen Proxy

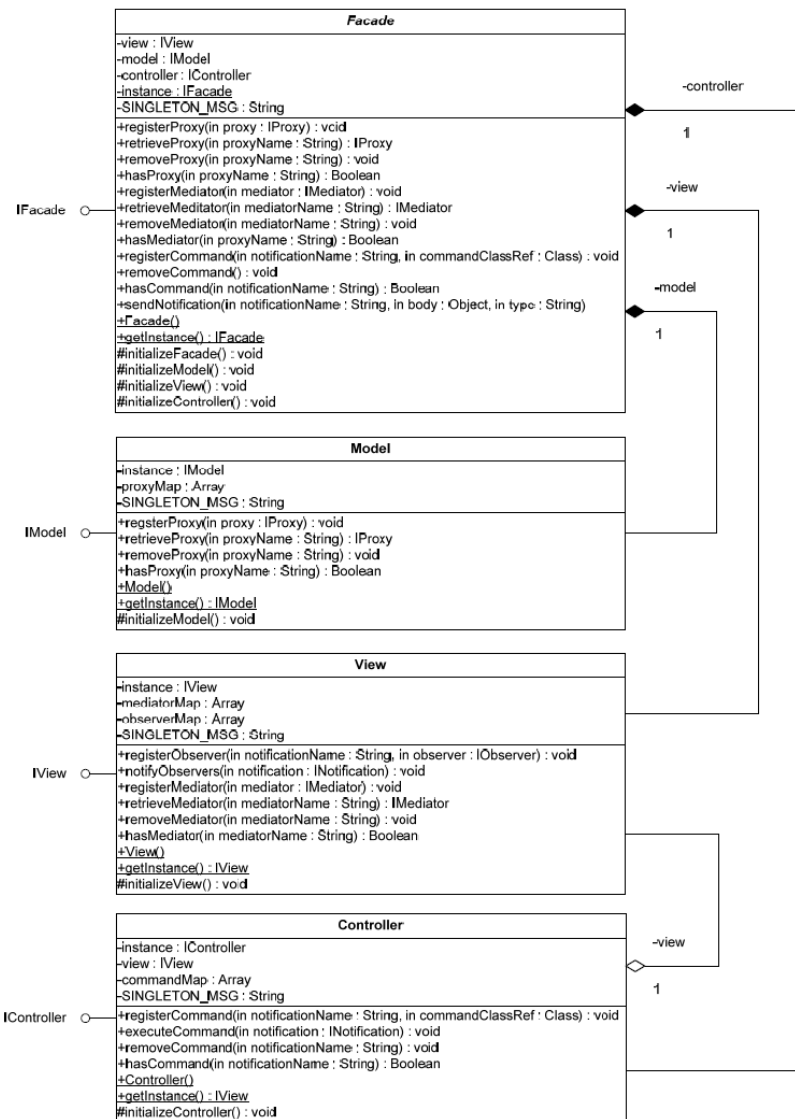
muuttaa näkymää vastaamaan ohjelman tilaa lähettämällä Notifikaation, joka kertoo datan muutoksesta. Jos salasana on oikein, käyttäjä pääsee ohjelmassa eteenpäin, muuten käyttäjän on syötettävä uusi salasana.

Kuvassa 12 näkyvästä luokkakaaviosta ilmenee, että ICommand-rajapinnan toteuttavia luokkia on kahta erilaista tyyppiä PureMVC-kehityksessä: SimpleCommand sekä MacroCommand. Näiden luokkien erona on, että SimpleCommand suorittaa yhden komennon kun taas MacroCommand pystyy suorittamaan useita Command-luokkia halutussa järjestyksessä. (Hall 2008.)

3.10 Facade

Facade on singleton luokka, jonka tarkoituksena on huolehtia sovelluksen käynnistyksestä ja mallin, näkymän sekä ohjaimen alustamisesta. Facade rekursiokooste, jonka komponentteja ovat Model-, View- ja Controller-luokat.

Kun Facadesta otetaan instanssi, Facade alustaa samalla mallin, ohjaimen, sekä näkymän. Lisäksi Facade rekisteröi Command-luokat ja ajaa tarvittavat ohjaimen komennot, mallin ja näkymän koostamiseksi. Facaden luomisen jälkeen Proxy-, Mediator- ja Command -luokat voivat käyttää Facade-ilmentymää keskinäisessä kommunikaatiossaan. (Hall 2008.)



Kuva 13 Facaden sekä PureMVC-päälluokkien luokkakaavio

3.11 PureMVC-kehiksen kommunikaatio

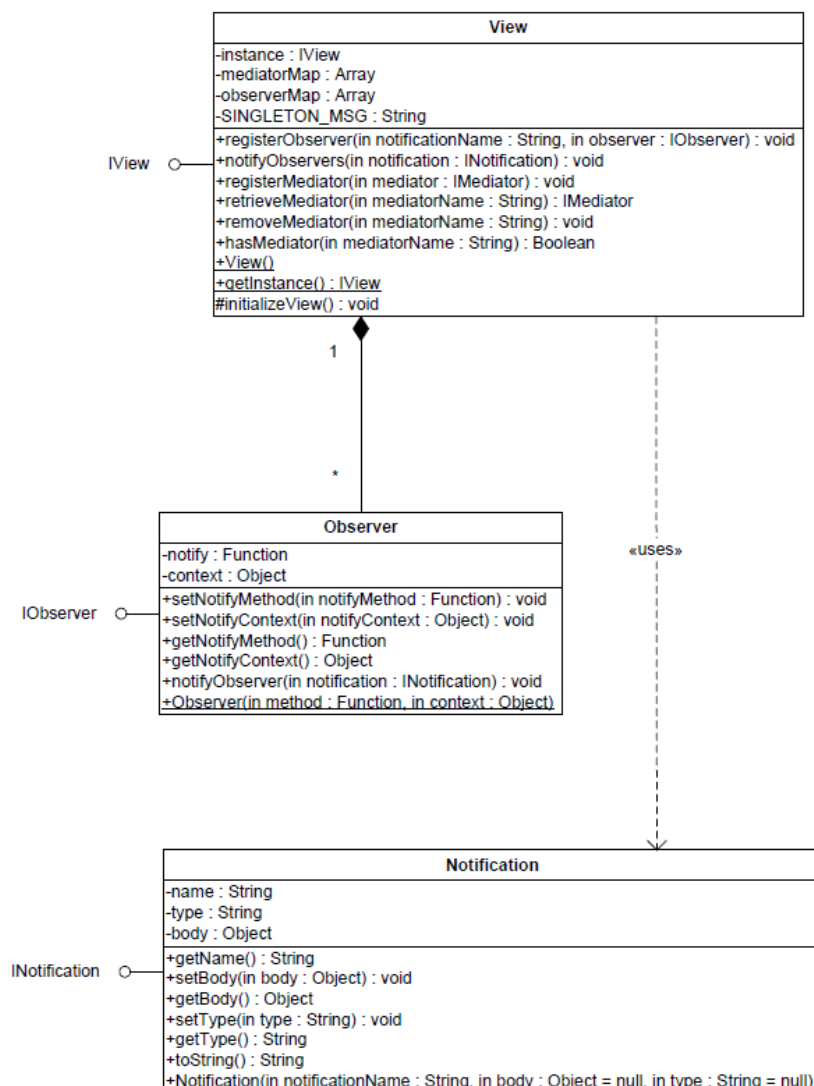
Aikaisemmin esiteltyt Proxy-, Mediator- ja Command-luokat keskustelevat keskenään Notifikaatioilla, jotka ovat ilmoituksia, joita kaikki edellä mainitut luokat voivat lähettää. Proxy lähettää, mutta ei tarkkaile Notifikaatioita. Mediator tarkkailee sekä lähettää Notifikaatioita ja Command-luokan laukaisijana toimii sille rekisteröity Notifikaatio. (Hall 2008.)

Koska PureMVC:n päätarkoituksena on pitää sovelluksen eri osat mahdollisimman itsenäisinä ja erillään toisistaan. Mediator, Command ja Proxy-luokat eivät ole tietoisia toisistaan. Sovelluksen eri osat ovat pilkottuna arkkitehtuurin eri kerroksiin, jotka eivät ole yhteydessä suoraan toisiinsa.

Ohjelman toiminnan kannalta datan siirto kerrosten välillä on välttämätöntä, jolloin tarvitaan datan kuljettamiseen erikoistunut luokka: Notification. PureMVC kehyksessä on toteutettu Tarkkailija-malli pääasiassa Observer- ja Notification-luokkien avulla.

PureMVC-kehyksessä Mediator-luokan instanssi luotaessa ilmoittaa View-luokalle Notifikaatio, joita haluaa tarkkailla. View lisää tarkkailijat a Notifikaatiot taulukkoon sekä pitää huolen, että oikeat Mediatorit saavat tiedon Notifikaatioiden lähetyksestä. (Hall 2008.)

Kuvan 14 luokkakaaviosta voi havaita, kuinka näkymän registerObserver funktio lisää observerMap-tilaukkomuuttujaan Notifikaation nimen sekä siitä kiinnostuneen Tarkkailija-luokan.



Kuva 14 Näkymän-, Tarkkailijan- ja Notifikaation-luokkakaavio.

3.11.1 Observer

Tarkkailija on ohjelmistokehyksen sisäänrakennettu luokka, josta ohjelmoijan ei tarvitse huolehtia. IObserver-rajapinnan toteuttava Observer luokka pitää sisällään viittauksen objektiin, joka on kiinnostunut notifi kaatiosta sekä kyseisen objektin funktioon, joka suoritetaan kun Notifi kaatio lähetetään.

Tarkkailijoiden toiminnasta vastaa näkymä-singleton-luokka, joka järjestää tarkkalijat taulukkoon ja lähettää tiedon kaikille tarkkailijoille Notifi kaation suorituksesta. (Hall 2008.)

3.11.2 Notification

PureMVC-kehyksen tarkoituksena on olla mahdollisimman kieliriippumaton, jolloin kehys ei voi käyttää minkään yksittäisen ohjelmointikielen luokkia sisäiseen kommunikaationsa. Tämän vuoksi PureMVC-ohjelmistokehykseen on tehty Notification- ja Observer-luokilla toteutettu Julkaise- ja tilaa-järjestelmä. (Hall 2008.)

4 SOVELLUSTYÖKALUT

Tässä osiossa esitellään PureMVC-kehiksen yhteydessä käytettäviä tekniikoita. Opinnäytetyö on tehty Riihimäen HAMK:n toimipisteessä toimivan mediatekniikan verstaan tarpeisiin, ja Adoben Flash, Flex sekä AIR ovat verstaalla pääasiassa käytettävät teknologiat. Siksi esittelyt keskittyvät pääasiassa Adoben teknologiaperheeseen. Esittelyssä käydään läpi lyhyesti vaihtoehtoisia ohjelmointiympäristöjä, sekä erilaisia ohjelmointirajapintoja ja hyödyllisiä teknologioita.

4.1 Kehitysympäristöt

PureMVC-kehys on alun perin tehty AS3-ohjelmointikielelle ja tarkoitettu käytettäväksi Adobe Flex-, Flash- ja Air-ympäristöissä. Nykyään kehys on kuitenkin saatavilla useille eri ohjelmointikielille kuten Java, C#, PHP, Python, JavaScript, jne. Tässä opinnäytetyössä keskitytään AS3-toteutuksiin ja tekniikoihin.

4.1.1 FlashDevelop

FlashDevelop on ilmainen ja avoimen lähdekoodin graafinen ohjelmointiympäristö, joka on alun perin suunniteltu halvaksi vaihtoehdoksi aloitteleville AS3-ohjelmoijille.

Ulkoasultaan FlashDevelop muistuttaa erittäin paljon Eclipse-ohjelmointiympäristöä. Ohjelmointiympäristö on kirjoitettu C#-ohjelmointikielellä ja tästä syystä on saatavilla vain Microsoft Windows-käyttöjärjestelmälle ja NET 2.0 kehikselle.

FlashDevelopin hyviä ominaisuuksia ovat mm. ilmaisuus, muokattavuus, hyvä syntaksikorostus ja koodin automaattinen täydennys. Kehitysympäristö on ilmaisuutensa ansiosta erittäin hyvä vaihtoehto aloittelevalle ohjelmoijalle ja yksityiselle pienyrittäjälle.

4.1.2 Flash Builder

Flash Builder on Adoben lanseeraama kehitysympäristö rikkaiden Internet-sovellusten kehittämiseen. Flash Builder on rakennettu Eclipsen päälle, minkä takia se muistuttaa erittäin paljon Eclipse-kehitysympäristöä. Flash Builder on selkeästi laadukkain, sekä ominaisuuksiltaan laajin vaihtoehto Flexin kanssa työskentelyyn.

Hyviä ominaisuuksia ovat mm. helppokäyttöinen, käyttöliittymien luomiseen tarkoitettu graafinen ympäristö, ASDOC-dokumentointityökalu sekä Debug-konsoli. Huonoina puolina Flash Builderista voi mainita kalliin hinnan, jonka takia se ei välttämättä sovellu pieneen, yksityiseen käyttöön.

4.1.3 Adobe Flex 4

Adobe Flex on ilmainen avoimen lähdekoodin kehys vuorovaikutteisten internet-sovellusten kehittämiseen. Flex-kehyksellä tehtyjä sovelluksia voi käyttää useimmissa eri selaimissa ja käyttöjärjestelmissä.

Flex-kehys sisältää oman MXML-kuvauskielen, joka on tarkoitettu käyttöliittymän asetteluun sekä ulkonäön ja toimintojen määrittelyyn. Asiakaspuolen ohjelmointi tapahtuu oliopohjaista AS3.0-ohjelmointikieltä käyttäen. Sovellusta käännettäessä MXML ja AS-tiedostot koostetaan yhdeksi SWF-tiedostoksi, joka muodostaa selaimessa käännettävän Flex-sovelluksen.

Kehyksen olennainen osa on valmiit MXML komponentit, joita on tarjolla yli 100 kappaletta. MXML-komponentteja voi tehdä ja yhdistellä helposti itse.

4.1.4 Adobe AIR

Adobe AIR on järjestelmäriippumaton ajoympäristö, joka mahdollistaa eri web-tekniikoiden kuten HTML:n, Ajaxin, Adoben Flashin ja Adobe Flexin käytön työpöytäsovelluksena. AIR-ajoympäristö käyttää ulkoasun renderöintiin WebKit HTML-renderöintimoottoria yhdistettynä Flash ja PDF-teknologioihin.

Sovelluksia Adobe AIR-ympäristöön voi ohjelmoida monella eri tapaa: Adobe Flash Builderilla, FlashDevelopilla, Dreamweaverilla, tai millä tahansa tekstinkäsittelyohjelmalla.

AIR-sovelluksia voidaan ohjelmoida myös kokonaisuudessaan käyttäen JavaScript-ohjelmointikieltä, mutta AIR-ympäristön JavaScriptin ajaminen eroaa hieman tavallisen selaimen ajamasta ohjelmointikoodista tietoturvasyistä. AIR-ympäristössä suoritettavat sisällöt laitetaan eri hiekkalaatikoihin (sandbox), jolloin suoritettavan sisällön oikeudet muuttuvat.

4.1.5 SVN-palvelimen esittely

Apache Subversion, lyhemmin SVN on version hallintajärjestelmä, jonka käyttäminen ohjelmistoprojektissa on erittäin suositeltavaa. Version hallinta pitää kopiot ohjelmoitavan lähdekoodin eri versioista automaattisesti ja versioiden välillä liikkuminen tapahtuu helposti.

SVN-palvelimen käyttö ei ole pakollista MVC-mallin yhteydessä. Tuskin mikään MVC-ohjelmistokehys vaatii suoraan Subversion-palvelimen käyttöä, mutta SVN helpottaa ja nopeuttaa ohjelmoijien työtä huomattavasti, varsinkin jos projektissa on useampi kuin yksi tekijä. Lisäksi SVN-palvelin ottaa tarvittavat varmuuskopiot lähdekoodista

automaattisesti ja pitää vanhat lähdekoodit saatavilla, jolloin mahdollisen virheen sattuessa voidaan palata vanhempiin versioihin helposti.

5 PUREMVC-TOTEUTUS

Tässä kappaleessa esitellään PureMVC:n käyttöönotto esimerkein ja yksityiskohtaisesti FlashDevelop-alustalle. Esimerkkikoodit toimivat myös Flash Builder 4 ympäristössä, lukuun ottamatta PureMVC-malliluokkia, jotka ovat ainostaan FlashDevelopille. Toteutusesimerkin tarkoituksena on, että PureMVC-ympäristöön tutustumaton käyttäjä ymmärtää kehityksen toimintaperiaatteen käytännössä ja pystyy soveltamaan kehystä omissa projekteissaan tämän ohjeistuksen pohjalta. Lisäksi päämääränä on, että lukija oppii tuntemaan PureMVC:n pääluokat ja niiden käyttötarkoitukset käytännön kautta.

5.1 Työkalujen asennus

Asennettuina työkaluina toteutusta varten tarvitaan FlashDevelop tai Flash Builder 4-kehitysympäristö, SVN-palvelin, sopivat versiot Flash player- ja Flash debug player-ohjelmista sekä uusin versio PureMVC-ohjelmistokehityksestä.

Kehys on saatavilla PureMVC-kehityksen kotisivuilta, josta tulee hakea yksityinen AS 3.0-versio. Lisäksi FlashDevelop-kehitysympäristöön on saatavilla valmiit mallit PureMVC-luokille. Malliluokkien käyttöönotto nopeuttaa ohjelmointityötä ja niiden käyttäminen on suositeltavaa.

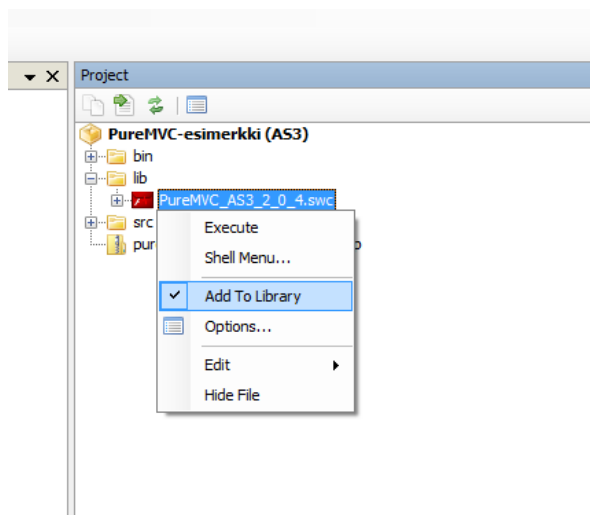
5.2 PureMVC asennus ja käyttöönotto.

PureMVC-kehityksen avulla toteutettu sovellus aloitettiin luomalla uusi Flex4-projekti, jonka nimeksi annettiin PureMVC-esimerkki. Projektin luomisen jälkeen FlashDevelop tekee automaattisesti peruskansiorakenteen, johon on erotettu:

- käännetty ohjelma bin-kansioon.
- lähdekoodi src-kansioon
- ulkopuoliset koodikirjastot lib-kansioon

<http://www.actionscriptdeveloper.co.uk/wpcontent/uploads/2008/05/puremvcflashdeveloptemplates.zip>-osoitteesta löytyvät malliluokat, joiden käyttöönotto on suositeltavaa, FlashDevelopille. Malliluokat otetaan käyttöön purkamalla osoitteesta saatavan zip-paketin sisältö C:\Users\AppData\Local\FlashDevelop\Templates\ProjectFiles\AS3Project\ -kansioon. Asennuksen jälkeen malliluokat ovat käytettävissä FlashDevelopin projekteissa hiiren oikeata nappia painamalla.

PureMVC-luokkien käyttöönotto tapahtuu helpoimmin ottamalla PureMVC:n kotisivuilta ladatusta paketista PureMVC_AS3_2_0_4.swc –tiedosto ja laittamalla tiedosto projektin lib-kansioon. Kopioinnin jälkeen .swc-tiedosto lisätään kirjastoon kuvan 15 osoittamalla tavalla oikeata hiiren nappia painamalla.



Kuva 15 Lisätään PureMVC_AS3_2_0_4.swc-tiedosto luokkakirjastoon FlashDevelop ympäristössä.

5.3 Esimerkkisovelluksen toteutus PureMVC-kehyksellä

Esimerkissä tehdään yksinkertainen sovellus PureMVC-kehyksellä, missä käyttäjä muokkaa DataGridin-taulukon soluja. Sovellukseen tarvitaan viisi luokkaa: ApplicationFacade, TextAreaProxy, TextAreaMediator, StartupCommand, ManipulateTextCommand. Näiden luokkien lisäksi tarvitaan TextAreaViewComponent.mxml-komponentti, joka määrittelee ohjelman käyttöliittymän.

5.3.1 Kansiorakenne

Kun kaikki tarvittavat työkalut on asennettu sekä toimintakunnossa, aloitetaan varsinaisen sovelluksen koostaminen. PureMVC:n kansiorakenne on mielivaltainen, mutta ohjelman selkeyden ja ylläpidon kannalta looginen kansiorakenne on suositeltava.

Kansiorakenteen luominen aloitetaan Luomalla src-kansion alle model-, view- ja controller-kansiot. Model-kansioon sisälle tulee konkreettiset Proxy-luokat, view-kansioon Mediator ja controller-kansioon Command-luokan ilmentymät.

Kun model-, view- ja controller-kansiot on luotu, tehdään vielä yksi oleellinen kansio: view-kansion alikansio components. Components-kansioon sijoitetaan MXML-kielellä määriteltävät käyttöliittymäkomponentit, joista luodaan ohjelman käyttöliittymä.

5.3.2 PureMVC-luokkien luominen

Uusia Luokkia luodaan helposti seuraavalla tavalla: valitaan kansio johon uusi luokka halutaan laittaa, jonka jälkeen valitaan add new class-toiminto hiiren oikealla napilla.

Toiminto avaa uuden luokan luomiseen tarkoitetun dialogin, jonka avulla ohjelmoija voi nopeuttaa työtään luomalla geneeristä ohjelmakoodia. Dialogista valitaan periytettävä luokka, toteutettava rajapinta sekä nimi uudelle luokalle, jonka jälkeen kehitysympäristö luo automaattisesti uuden luokan käyttäjän valintojen mukaisesti.

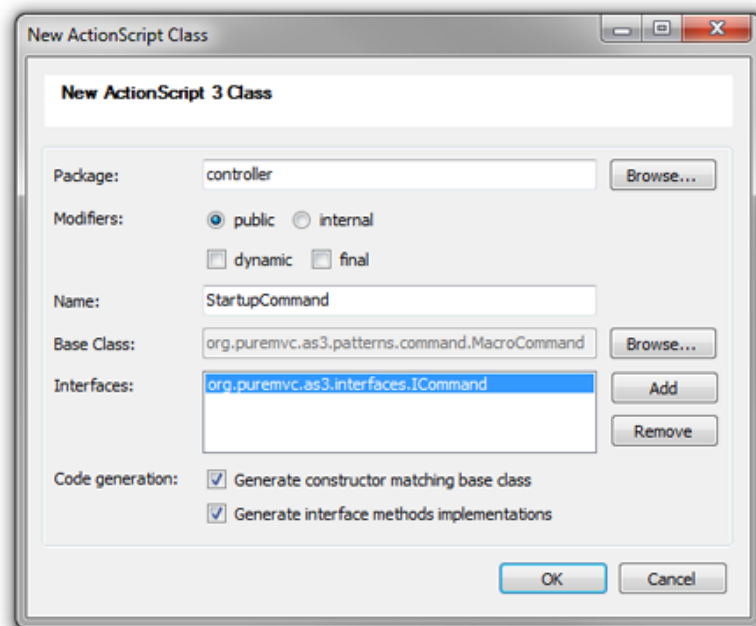
Jotta ohjelmoija voi käyttää PureMVC:n ominaisuuksia, uusien luokkien tulee periä PureMVC:n abstraktit luokat sekä rajapinnat. StartupCommand perii org.puremvc.as3.patterns.command.SimpleCommand-luokan ja toteuttaa ICommand-rajapinnan, TextAreaProxy perii Proxy-luokan sekä toteuttaa Iproxy-rajapinnan jne.

```
package controller
{
    import org.puremvc.as3.interfaces.ICommand;
    import org.puremvc.as3.interfaces.INotification;
    import org.puremvc.as3.patterns.command.SimpleCommand;
    import model.TextAreaProxy;
    import view.TextAreaMediator;
    /**
     * ...
     * @author Janne Kauhanen
     */
    public class StartupCommand extends SimpleCommand implements ICommand
    {
        /* INTERFACE org.puremvc.as3.interfaces.ICommand */
        override public function execute(note:INotification):void
        {
        }
    }
}
```

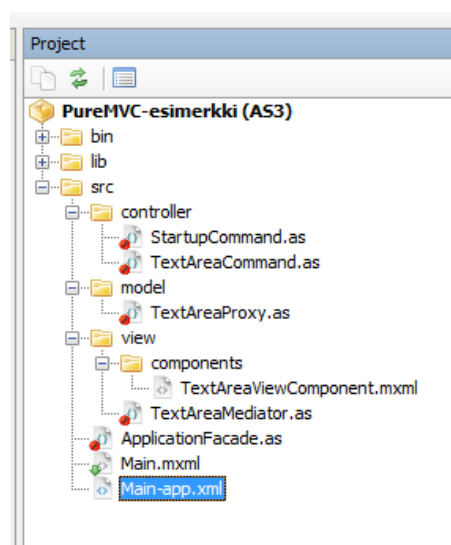
Mediator sekä Proxy-luokkien instanssit varastoidaan Model- ja View-luokkiin nimen perusteella, siksi on hyvä tapa tehdä luokan nimestä staattinen vakio alla olevan esimerkin osoittamalla tavalla. Koodiesimerkissä perittävälle luokalle lähetetään sen vaatimat nimimerkkijono, sekä data-objekti, super avainsanalla. Kun kaikki luokat on luotu, kansiorakenne on samanlainen kuin kuvassa 17.

```
public class TextAreaProxy extends Proxy implements IProxy
{
    public static const NAME:String = "TextAreaProxy";

    public function TextAreaProxy(data:Object = null)
    {
        super(NAME, data);
    }
}
```

Kuva 16 Uuden luokan luomiseen tarkoitettu dialogi



Kuva 17 Valmis luokka- ja -kansiorakenne.

5.4 Sovelluksen ohjelmointi luokittain

Koska PureMVC-kehyksen tarkoituksena on jakaa eri ohjelmakoodin osat eri kerroksiin, on katsottava tarkemmin jokaisen kerroksen koodia. Seuraavaksi esitellään ohjelman rakenne ja toiminta yksityiskohtaisesti ohjelman käynnistymisjärjestyksessä.

5.4.1 Main.mxml

Tämä on ohjelman pääluokka. Tässä luokassa käynnistetään PureMVC-kehys, sekä määritellään näkyvät komponentit ja sovelluksen pääikkunan koko. Kuten tiedoston päätteestä voi havaita, Main.mxml on Flex-komponentti ja se on tehty MXML-määrittelykielellä.

Sovelluksen ohjelmointi aloitetaan tekemällä käynnistysfunktio Main.mxml-tiedostoon. Käynnistysfunktiossa luodaan ensin app-niminen, ApplicationFacade-tyyppinen muuttuja, johon tallennetaan instanssi ApplicationFacade-luokasta. Koska luokka on singleton, instanssin ottamisessa täytyy käyttää staattista getInstance-metodia.

Jotta MXML-tagien sekaan voidaan upottaa AS3.0-ohjelmointikoodia, tarvitaan Script-tagit, joiden sisään ohjelmakoodi sijoitetaan. Kun instanssi ApplicationFacade-luokasta on otettu, tehdään startPureMVC-funktio. Funktion tarkoituksena on käynnistää sovellus ja pureMVC-kehys suorittamalla ApplicationFacaden startPureMVC-funktio. Funktion mukana lähetetään viittaus Main.mxml-luokkaan, joka on erittäin tärkeä ohjelman toiminnan kannalta, koska Main.mxml-luokka sisältää sovelluksen käyttöliittymän. Käyttäjän toimintojen kuuntelemista varten tarvitaan viittaus käyttöliittymään Mediatorissa, joka välittää käyttäjän eleet ohjaimelle.

```
<fx:Script>
    <![CDATA[
        private var app:ApplicationFacade = ApplicationFacade.getInstance();
        //otetaan singleton-luokasta otetaan yksi instanssi käyttämällä getInstance-metodia
        public function startPureMVC():void
        {
            //käynnistetään PureMVC
            app.startPureMVC(this);
        }
    ]]>
</fx:Script>
```

Kun StartPureMVC-funktio on valmis, lisätään Main-tiedostoon oleellisia määrittelyjä. Tiedostoon lisätään mm. tapahtumankuuntelija, joka kuuntelee sovelluksen käynnistymistä. Lisäksi Mainiin tulee nimiavaruus, joka sisältää aiemmin tehdyn TextAreaViewComponent.mxml-tiedoston sekä sovelluksen pääikkunan koon määrittelevät attribuutit.

CreationComplete-attribuutti luo automaattisesti tapahtumankuuntelijan, joka suorittaa halutun funktion ohjelmakoodin kääntämisen jälkeen. Xmlns-attribuutti tekee nimiavaruuden ja yhdistää sen haluttuun polkuun.

Tässä tapauksessa luodaan nimiavaruus view ja yhdistetään se kansiorakenteessa olevaan käyttöliittymäkomponenttikansioon src.view.components. Tähti-notaatiolla nimiavaruuteen yhdistetään kaikki kansiossa olevat komponentit.

```
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="startPureMVC();"
    xmlns:view="view.components.*" width="340" height="269" >
```

Lopuksi pääsovellukseen lisätään vielä näkymä, jota rakennetaan myöhemmin. Näkymän tunnisteeksi annetaan `TextAreaViewComponent` id-attribuutilla. Tunnisteen käyttäminen helpottaa komponenttiin viittaamista huomattavasti, joten tunnisteiden käyttö on suositeltavaa.

```
<view:TextAreaViewComponent id="textAreaViewComponent"></view:TextAreaViewComponent>
```

5.4.2 ApplicationFacade.as

Kuten opinnäytetyön teoriaosuudessa kerrottiin Facade-luokka on ylemmän tason rajapintaluokka, jonka kautta PureMVC:n muut luokat kommunikoivat keskenään. Tässä luokassa rekisteröidään tärkeimmät notifiikaatiot ja sidotaan notifiikaatiot haluttuihin Command-luokkiin.

Facaden ohjelmointi aloitetaan tekemällä MVC-kehityksen käynnistämiseen tarkoitettu `startPureMVC`- funktio. Funktion tehtävä on yksinkertainen: se lähettää Notifiikaation, joka käynnistää ensimmäisen ohjelmakomennon ottamalla instanssin `StartupCommand`-luokasta. Notifiikaation mukana lähetetään viittaus `Main.mxml`-luokkaan, `StartupCommand`-luokan käsiteltäväksi.

```
public function startPureMVC (app:Main):void
{
    sendNotification(ApplicationFacade.STARTUP, app);
}
```

Jotta PureMVC osaa yhdistää komennot Notifiikaatioihin, komennot täytyy rekisteröidä `ApplicationFacade`-luokassa. Komennot rekisteröidään yliajettavassa `initializeController`-funktiossa, joka alustaa Controller-singleton luokan. `RegisterCommand` yhdistää Command-luokan merkkijonoon, jonka Notifiikaatio lähettää.

```
override protected function initializeController():void
{
    super.initializeController();
    registerCommand(STARTUP, StartupCommand);
}
```

Komennon rekisteröinnin jälkeen on tarpeellista tehdä Notifiikaation mukana lähetettävä merkkijono, joka suorittaa `StartupCommand`. Hyvä tapa on tehdä merkkijonoista julkisia, staattisia, vakioita, jotka sijoitetaan `ApplicationFacade` luokan sisään. Tällöin ne ovat koottuna yhdessä paikassa kaikkien PureMVC-luokkien ulottuvilla.

```
public class ApplicationFacade extends Facade implements IFacade {  
  
    // Notification name constants  
    public static const STARTUP:String = "startup";  
}
```

5.4.3 StartupCommand.as

StartupCommand-luokan toiminnot ovat erittäin yksinkertaisia eivätkä vaadi montaa riviä ohjelmakoodia. Aluksi luodaan Main-tyyppinen muuttuja, johon tallennetaan Notifikaation mukana tuleva viittaus pääohjelmaan. Kun pääohjelma on tallennettu muuttujaan, rekisteröidään TextAreaProxy-luokka. Rekisteröitäessä Proxya tulee esille jälleen Facade-luokan monikäyttöisyys: kaikki PureMVC:n tärkeimmät toiminnot tapahtuvat yhden rajapinnan kautta.

Kun Proxy on rekisteröity ja data näkymää varten on valmisteltu, rekisteröidään ohjelman päänäkömää tarkkaileva luokka TextAreaMediator. Luokalle annetaan mukaan käyttöliittymän komponentti, jota Mediatorin halutaan tarkkailevan eli tässä tapauksessa TextAreaViewComponent.mxml.

```
override public function execute(note:INotification):void  
{  
    var mainApp:Main = note.getBody() as Main;  
    facade.registerProxy(new TextAreaProxy());  
    facade.registerMediator  
        (new TextAreaMediator(mainApp.textAreaViewComponent));  
}
```

5.4.4 TextAreaProxy.as

TextAreaProxy-luokka pitää tiedon päänäkömän datasta ja tarjoaa datan hallintaan tarkoitetut funktiot. Sovelluksen ideana on tekstikentän muokkaaminen päänäkömässä, joten Proxyn tarvitsee pitää sisällään tekstikentän muokkaamisen tarkoitetut funktiot.

Funktioita tekstikentän datan manipuloimiseen tarvitaan kolme kappaletta: tekstin lisäämiseen, tekstin poistamiseen sekä funktio koko taulukon alustamiseen. Jokainen tekstikentän sisältöä muokkaava funktio sisältää notifikaation lähetyksen. Notifikaatioiden tehtävänä on ilmoittaa näkymälle datan muutoksista, jotta näkymä tietää noutaa muokatun, ajan tasalla olevan datan.

```
public function addNewText(newText:String):void
{
    dataInArrayCollection.addItem({text:newText});
    sendNotification(ApplicationFacade.TEXTAREA_DATA_CHANGED);
}

public function resetData():void
{
    this.data = new ArrayCollection();
    sendNotification(ApplicationFacade.TEXTAREA_DATA_CHANGED);
}

public function removeText(selectedIndex:int):void
{
    dataInArrayCollection.removeItemAt(selectedIndex);
    sendNotification(ApplicationFacade.TEXTAREA_DATA_CHANGED);
}
```

Koska data-muuttuja Proxy-luokassa on Object-tyyppinen, tyyppin muuttaminen on mahdollista sekä tarpeellista. Esimerkkisovelluksessa helpoin tapa datan lisäämiseksi näkymän komponentteihin on ArrayCollection-luokka.

ArrayCollection on kääreluokka, jonka sisään laitetaan taulukkoja ja joka tarjoaa taulukon käsittelyyn tarkoitettuja metodeja. Proxy-luokkaan tehdään get-funktio, joka palauttaa datan ArrayCollection tyyppimuunnattuna sekä yliajetaan onRegister-funktio, joka suoritetaan aina automaattisesti Proxyn rekisteröinnin aikana.

```
override public function onRegister():void
{
    resetData();
}

public function get dataInArrayCollection():ArrayCollection
{
    return this.data as ArrayCollection;
}
```

Viimeisenä toimenpiteenä ennen seuraavan luokan ohjelmointiin siirtymistä on hyvä lisätä lähetettävien Notifikaatioiden nimiä ApplicationFacade-luokkaan, jotta Notifikaatioista kiinnostuneet luokat voivat tarkkailla lähetystä.

Facade sisältää jo aikaisemmin tehdyn Startup-merkkijono-vakion, josta mallia ottamalla voidaan luoda muutkin tarvittavat Notifikaatio-nimet. Toimiakseen halutulla tavalla, ohjelma tarvitsee lisäksi Notifikaatiot tekstidatan muokkaamiselle.

Näkymässä käyttäjän tekemät muutokset ja eleet täytyy kuljettaa ohjaimen kautta mallille. Notifikaatiot on suunniteltu käyttäjän toimintojen kuljetusta varten ja niiden käyttö pitää luokkien riippuvuussuhteet mahdollisimman vähäisinä.

```
public class ApplicationFacade extends Facade implements IFacade {
    // Notification name constants
    public static const STARTUP:String = "startup";
    public static const ADD_NEW_TEXT:String = "add_new_text";
    public static const RESET_TEXTAREA:String = "reset_text_area";
    public static const REMOVE_TEXT:String = "remove_text";
    public static const TEXTAREA_DATA_CHANGED:String = "textarea_data_changed";
}
```

5.4.5 TextAreaViewComponent.mxml

Käyttöliittymän määrittely aloitetaan lisäämällä komponentit mxml-tiedostoon ja määrittelemällä komponenttien sijainnit sekä toiminnot. Sovelluksen näkymä on komponenttiryhmä ja se määritellään group-tagilla, jonka sisään kaikki käyttöliittymän napit sekä muut komponentit tulevat. Group tagin sisällä lisätään käytettävät nimiavaruudet, jotka sisältävät Adoben tarjoamat komponenttikirjastot sekä määritellään näkymän koko.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  height="248" width="339">
</s:Group>
```

Kun group-tagin on saatu määriteltyä, aletaan tagin sisään kasata käyttöliittymää. Sovelluksessa lisätään taulukkoon merkkijonoja sekä poistetaan merkkijonoja, minkä vuoksi tarvitaan komponentti, johon listata merkkijonot. Tähän tarkoitukseen sopii erinomaisesti DataGrid-luokka, taulukkokomponentti, johon voi luoda ja nimettyjä sarakkeita.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  height="248" width="339">
</s:Group>
```

DataGridin lisäksi käyttöliittymään tarvitaan napit tekstin poistamiselle, lisäämiselle, DataGridin alustamiselle sekä tekstikenttä uuden tekstin lisäämiseksi.

```

<mx:DataGrid id="dataGrid" selectable = "true"
change="dataGridIndexChanged()" x="0" y="0" width="339" height="159">
<mx:columns>
  <mx:DataGridColumn dataField="text" headerText="Text"/></mx:columns>
</mx:DataGrid>

<s:TextInput id="newTextInput" change="textChanged();" selectable="true"
x="10" y="167" width="320">add new text here</s:TextInput>
<s:Button id="addButton" label="add new text" enabled="false"
click="addButtonClicked();" x="10" y="209"></s:Button>
<s:Button id="resetButton" label="reset" enabled="true"
click="resetButtonClicked();" x="240" y="209" width="89"></s:Button>
<s:Button id="removeButton" label="remove text" enabled="false"
click="removeButtonClicked();" x="125" y="209" width="89"></s:Button>

```

Nappeihin lisätään klikattaessa suoritettavien funktioiden nimet: addButtonClicked, resetButtonClicked ja removeButtonClicked. Nappien tila riippuu dataGrid-komponentin sekä TextInput-tekstikentän muutoksista, joten näiden kenttien muutoksia tulee tarkkailla. DataGrid ja TextInput-luokille määritellään muutoksen tapahtuessa suoritettavat funktiot dataGridIndexChanged sekä textChanged.

Komponenttien määrittelyn jälkeen ohjelmoidaan käyttöliittymän toiminnot. Toiminnallisuuden toteuttamiseen tarvitaan ActionScriptin EventDispatcher-luokkaa, joka toimii julkaise- ja tilaa-suunnittelumallin periaatteella.

AddButtonClicked-funktio lähettää instanssin Event-luokasta, jonka mukana lähetetään Add_button_pressed-vakio. Muut napit toimivat samalla tavalla, ainoastaan Eventin mukana lähetettävä vakio muuttuu.

RemoveButtonClicked-funktio suorittaa eventDispatchin jälkeen dataGridIndexChange-funktion, joka aktivoi tekstin poistoon tarkoitetun napin käyttöliittymässä. TextChanged-funktio huolehtii tekstin lisäysnapin aktivoinnista, tarkastamalla lisättävän tekstin pituuden. Lisättävän uuden tekstin tulee olla vähintään yhden merkin pituinen, jotta addButton-nappi aktivoituu.

DataGridIndexChanged-funktio huolehtii tekstin poistamiseen tarkoitetun napin tilasta. Funktio tarkistaa, onko dataGridistä valittuna jokin kenttä. Jos on, removeButton-napista tehdään aktiivinen.

```

<fx:Script>
<![CDATA[
import flash.events.Event;
import flashx.textLayout.events.SelectionEvent;

public static const ADD_BUTTON_PRESSED:String = "add_button_pressed";
public static const RESET_BUTTON_PRESSED:String = "reset_button_pressed";
public static const REMOVE_BUTTON_PRESSED:String = "remove_button_pressed";

private function addButtonClicked():void
{
    dispatchEvent(new Event(ADD_BUTTON_PRESSED));
}
private function resetButtonClicked():void
{
    dispatchEvent(new Event(RESET_BUTTON_PRESSED));
}
private function removeButtonClicked():void
{
    dispatchEvent(new Event(REMOVE_BUTTON_PRESSED));
    dataGrid.selectedIndexChanged();
}
private function textChanged():void
{
    if (newTextInput.text.length > 0) addButton.enabled = true;
    else if (newTextInput.text.length < 1) addButton.enabled = false;
}
private function dataGridIndexChanged():void
{
    if (dataGrid.selectedItem != null) removeButton.enabled = true;
    else if (dataGrid.selectedItem == null) removeButton.enabled = false;
}
]]>
</fx:Script>

```

5.4.6 TextAreaMediator.as

Mediator on PureMVC:n monimutkaisin luokka ja vaatii eniten ohjelmakoodia. Mediatorin tehtävänä on kuunnella käyttäjän tekemiä toimintoja ja välittää toiminnot eteenpäin.

Instanssin ottamisvaiheessa Mediator ottaa vastaan käyttöliittymän osan, jota sen on tarkoitus tarkkailla. Flex-ohjelmassa käyttöliittymän osa on mxml-tiedosto. Hyvä tapa käyttöliittymäkomponenttiin viittaamiseen on getter-funktion tekeminen.

Koska Mediator-luokan tarvitsee hakea useasti dataa TextAreaProxy-luokasta, on getter-funktion tekeminen myös Proxylle järkevää. TextAreaProxy-luokan instanssi haetaan Facaden retrieveProxy-funktiolla Facaden nimen perusteella ja palauttaa Objektin, jonka tyyppi muunnetaan TextAreaProxy-luokaksi.

```

private function get textAreaComponent():TextAreaViewComponent
{
    return this.viewComponent as TextAreaViewComponent;
}
private function get textAreaProxy():TextAreaProxy
{
    return facade.retrieveProxy(TextAreaProxy.NAME) as TextAreaProxy;
}

```


Jotta käyttäjän toimintoja voidaan kuunnella, tarvitsee rekisteröidä tapahtumankuuntelijat Mediatorin tarkkailemaan komponenttiin. Tapahtumankuuntelijat saattavat olla ongelmallisia sovelluksen muistinkulutuksen kannalta ja siksi niiden lisäämiseen ja poistoon tulee kiinnittää erityistä huomiota.

Jos tapahtumankuuntelija lisätään, mutta sen poistamisesta ei huolehdi, tulee helposti tilanne, jossa sama tapahtumankuuntelija lisätään useaan kertaan. Tämä on ongelmallista koska tapahtumankuuntelija ei yliaja vanhaa, vaan lisää samanlaisen kuuntelijan vanhan rinnalle ja kuunneltavan tapahtuman sattuessa, kuuntelijan toiminto suoritetaan useaan kertaan. Lisäksi tapahtumankuuntelijat lisäävät kuunneltavaan luokkaan vahvan viittauksen, joka estää roskien kerääjää tuhoamasta luokkaa, mikä aiheuttaa pitkässä juoksussa hallitsemattomia muistivuotoja.

Hyvä paikka tapahtumankuuntelijoiden lisäämiseen Mediator-luokassa on yliajettava `onRegister`-funktio, joka ajetaan yhden ainoan kerran Mediatorin rekisteröimisvaiheessa. Vastaavasti kuuntelijan poisto kannattaa tehdä `onRemove`-funktiossa, joka suoritetaan Mediatorin tuhoamisvaiheessa. Näillä toimenpiteillä varmistetaan, että Mediatorin tarvitsemat kuuntelijat luodaan vain kerran, eikä muistivuotoja synny. Samalla kuuntelijoista päästään eroon helposti poistamalla Mediator eikä kuuntelijoiden poistosta tarvitse huolehtia erikseen.

```
override public function onRegister():void
{
    textAreaComponent.addEventListener(TextAreaViewComponent.ADD_BUTTON_PRESSED, handleEvents);
    textAreaComponent.addEventListener(TextAreaViewComponent.RESET_BUTTON_PRESSED, handleEvents);
    textAreaComponent.addEventListener(TextAreaViewComponent.REMOVE_BUTTON_PRESSED, handleEvents);
    textAreaComponent.dataGrid.dataProvider = textAreaProxy.dataInArrayCollection;
}
override public function onRemove():void
{
    textAreaComponent.addEventListener(view.components.TextAreaViewComponent.ADD_BUTTON_PRESSED,
    handleEvents);

    textAreaComponent.addEventListener(view.components.TextAreaViewComponent.RESET_BUTTON_PRESSED,
    handleEvents);

    textAreaComponent.addEventListener(view.components.TextAreaViewComponent.REMOVE_BUTTON_PRESSED,
    handleEvents);

}
```

`TextAreaMediator` kuuntelee `TextAreaComponent`-luokan vakioita, jotka ohjelmoitiin edellisessä kappaleessa. Tekstin lisäysnapille, poistamisnapille sekä kentän alustamisnapille on omat tarkkailtavat vakionsa, joten tarvitaan `handleEvents`-funktio, joka ohjaa jatkotoimia eri tapahtumille. `HandleEvents`-funktion tehtävänä on ottaa vastaan tapahtuma, tarkistaa tapahtuman tyyppi sekä lähettää Notifikaatio tapahtumasta ohjaimelle.

```
private function handleEvents(e:Event):void
{
    switch(e.type)
    {
        case TextAreaViewComponent.ADD_BUTTON_PRESSED:
            sendNotification
            (ApplicationFacade.ADD_NEW_TEXT, textAreaComponent.newTextInput.text);
            break;

        case TextAreaViewComponent.RESET_BUTTON_PRESSED:
            sendNotification(ApplicationFacade.RESET_TEXTAREA);
            break;

        case TextAreaViewComponent.REMOVE_BUTTON_PRESSED:
            sendNotification
            (ApplicationFacade.REMOVE_TEXT, textAreaComponent.dataGrid.selectedIndex);
            break;
    }
}
```

MVC-mallin mukaisesti näkymän täytyy olla tietoinen mallin tilanmuutoksesta, jotta näkymä osaa näyttää ajantasalla olevan datan käyttäjälle. Mediator-luokka rekisteröi tarkkailtavien Notifikaatioiden nimet listNotificationInterests-funktiossa, joka palauttaa taulukon Notifikaatioiden nimistä, joita mediator haluaa kuunnella. Esimerkkisovelluksessa Mediator kuuntelee vain textarea_data_changed-Notifikaatiota, joka lähetetään proxysta aina kun dataa muutetaan.

```
override public function listNotificationInterests():Array
{
    return [ApplicationFacade.TEXTAREA_DATA_CHANGED];
}
```

Tarkkailijamallin mukaisesti PureMVC:ssä tarkkailijalle on annettu funktio, joka suoritetaan aina vastaanotettaessa Notifikaatiota. Mediator-luokassa suoritetaan aina handleNotification-funktio, jossa voidaan erotella esimerkiksi switch-case-lauseella eri Notifikaatiolle eri toimintamalli.

Esimerkkisovelluksessa tarkastetaan Notifikaation nimi, kun Notifikaation nimi on text_area_changed, textAreaComponent-näkymässä olevan dataGrid-komponentin data korvataan TextAreaProxysta haettavalla uudella datalla.

```
override public function handleNotification(notification:INotification):void
{
    switch(notification.getName())
    {
        case ApplicationFacade.TEXTAREA_DATA_CHANGED:
            textAreaComponent.dataGrid.dataProvider = textAreaProxy.dataInArrayCollection;
            break;
    }
}
```

5.4.7 TextAreaCommand.as

käyttöliittymässä käyttäjän tekemät eleet lähetetään Mediator-luokan avustamana ohjaimelle, joka komentojen avulla ohjaa sovelluksen jatkotoimia. Jotta PureMVC osaa käynnistää oikean komennon eri Notifikaatioille, tulee komennot ja Notifikaatioiden nimet rekisteröidä ApplicationFacade-luokassa.

InitializeController-funktiossa rekisteröidään, sovelluksen käynnistysvaiheessa, kaikki sovelluksessa käytetyt komennot ja sidotaan ne johonkin Notifikaatioon. Notifikaatioita voi ja kannattaa sitoa useita yhteen komentoon, jotta Command-luokkien lukumäärä ei paisu tarpeettomasti. Esimerkkisovelluksessa kaikki tekstin muuttamiseen tarkoitetut Notifikaatiot kutsuvat TextAreaCommand-luokkaa, joka vastaanottaa Ilmoituksen ja kutsuu Proxyn funktioita mallin muokkaamiseksi.

```
override protected function initializeController():void
{
    super.initializeController();
    registerCommand(STARTUP, StartupCommand);
    registerCommand(ADD_NEW_TEXT, TextAreaCommand);
    registerCommand(RESET_TEXTAREA, TextAreaCommand);
    registerCommand(REMOVE_TEXT, TextAreaCommand);
}
```

Kun TextAreaCommand-luokasta otetaan instanssi, suoritetaan luokasta aina automaattisesti execute-funktio, joka ottaa vastaan lähetetyn Notifikaation. Notifikaatiosta on hyvä tallentaa sekä nimi että mukana tuleva data muuttujiin. Muuttujien luonnin jälkeen kutsutaan chooseAction-funktiota, jolle lähetetään noteName sekä noteBody-muuttujat.

NoteName sisältää Notifikaation nimen ja noteBody mukana tulleen datan. Koska TextAreaProxyn jotkut funktiot ottavat sisäänsä muuttujan ja noteBody-muuttuja on object tyyppinen, on muuttujan tyyppi muunnettava funktiolle sopivaksi.

```
override public function execute(notification:INotification):void
{
    var noteName:String = notification.getName() as String;
    var noteBody:Object = notification.getBody();
    chooseAction(noteName,noteBody);
}
```

TextAreaCommand-luokan tarvitsee käsitellä toistuvasti TextAreaProxy-luokkaa ja luokan funktioita. Edellämäinitusta syystä viittauksen tekeminen omaksi getter-funktiokseen on suositeltavaa. Getter-funktion ohjelmointi tapahtuu kuten aiemmissa esimerkeissä: laittamalla funktion eteen get-avainsana, minkä jälkeen funktio toimii kuten muuttuja. Funktion sisällä haetaan Facade-luokalta Proxyn ilmentymä retrieveProxy-funktiota käyttämällä TextAreaProxyn Name-vakion perusteella.

```
private function get textAreaProxy():TextAreaProxy
{
    return facade.retrieveProxy(TextAreaProxy.NAME) as TextAreaProxy;
}
```

ChooseAction-funktion tehtävänä on muuttaa TextAreaProxyn dataa käyttäjän toiminnon mukaan. Notifikaation nimen ollessa add_new_text suoritetaan textAreaProxyn addNewText-funktio, jolle laitetaan Notifikaation mukana tullut uusi, lisättävä teksti. Reset_text_area suorittaa resetData-funktion, joka tyhjentää Proxyn datan. Remove_text suorittaa removeText-funktion, joka tarvitsee sisäänsä taulukosta poistettavan tekstin indeksin.

```
private function chooseAction(noteName:String, noteBody:Object):void
{
    switch (noteName)
    {
        case ApplicationFacade.ADD_NEW_TEXT:
            textAreaProxy.addNewText(noteBody as String);
            break;
        case ApplicationFacade.RESET_TEXTAREA:
            textAreaProxy.resetData();
            break;
        case ApplicationFacade.REMOVE_TEXT:
            textAreaProxy.removeText(noteBody as int);
            break;
    }
}
```

5.4.8 Sovelluksen toiminta

Lopuksi on syytä tarkastella esimerkisovelluksen toimintaa. Kuvassa 16 on kaavio sovelluksen toiminnasta. Ohjelman käynnistyessä kohdassa 1 Main.mxml-tiedostossa suoritetaan startPureMVC-funktio, joka ottaa instanssin ApplicationFacade-luokasta. Instanssin ottamisen jälkeen suoritetaan Facaden startPureMVC-funktio, ja annetaan funktion parametriksi viittaus Main.mxml tiedostoon.

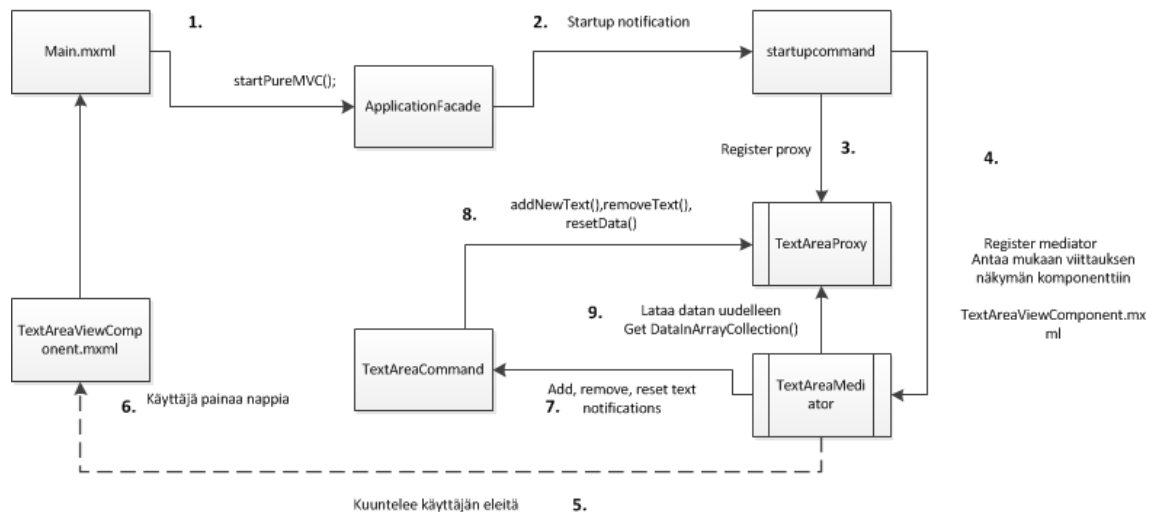
Kohdassa 2 ApplicationFacade-luokka lähettää ohjelman ensimmäisen Notifikaation, jonka nimi on Startup. Startup-Notifikaation ottaa instanssin StartupCommand-luokasta ja antaa Notifikaation mukana viittauksen Main.mxml-luokkaan.

Seuravassa kohdassa StartupCommand suorittaa execute-funktion, joka rekisteröi TextAreaProxyn sekä Mediatorin. TextAreaMediatorille annetaan viittaus tarkkailtavaan näkymään eli TextAreaViewComponent-luokkaan.

Kohdassa 5 TextAreaMediator lisää tarvittavat tapahtumankuuntelijat, minkä jälkeen se jää kuuntelemaan käyttäjän tekemiä valintoja. Kun kohdassa 6 käyttäjän painaessa käyttöliittymän nappia, Mediator lähettää Notifikaation. TextAreaMediatorin lähettämät Notifikaatiot ovat sidottuja TextAreaCommand-luokkaan, josta otetaan instanssi aina kun jokin kolmesta tekstin muokkaus-Notifikaatiosta lähetetään.

Kohdassa 8 TextAreaCommand käsittelee Notifikaation ja sen sisältämän datan. Notifikaatiosta riippuen Command-luokka suorittaa TextAreaProxyn datan käsittelyyn tarkoitettuja eri funktioita. Esimerkiksi, jos käyttäjä lisää uuden tekstin, Notifikaation mukana tullut uusi teksti annetaan mukaan TextAreaProxyn addNewText-funktion parametrinä. Kun TextAreaProxyn data-muuttujaa on muokattu, lähetetään textAreaData_changed-Notifikaatio.

Kohdassa 9 TextAreamediator kuuntelee textAreaChanged-Notifikaatiota, ja kun Notifikaatio lähetetään, hakee TextAreaMediator datan uudelleen Proxyltä. Kierroksen jälkeen Mediator jää kuuntelemaan käyttäjän eleitä kunnes sovellus sammutetaan. Jokainen napin painallus käyttöliittymässä suorittaa kaavion kohdat 6 – 9.



Kuva 18 Esimerkkisovelluksen toiminta

6 YHTEENVETO

Sovelluksen tekeminen hyödynämällä PureMVC-kehystä on monikäsitteinen asia, ja PureMVC:n käyttöönoton optimaalista tilannetta on vaikea määritellä. Ei ole olemassa selvää koodirivien määrää, joka ilmaisisi, milloin kehyksen käyttöönotto olisi järkevää. On monia eri syitä olla käyttämättä ja käyttää PureMVC-kehystä. Seuraavaksi tarkastellaan PureMVC-projekteissa ilmenneitä hyötyjä ja haittoja.

6.1 Arkkitehtuurin hyödyt

Nyrkkisääntönä voidaan sanoa, että jos projektissa on useita ohjelmoijia, ohjelmistokehyksen käytöstä on selvää hyötyä. PureMVC:n käyttö yhdessä SVN-palvelimen kanssa, nopeutti ja helpotti ohjelmoijien välistä yhteistyötä suuresti. PureMVC ja SVN mahdollisti sovelluksen eri näkymien toiminnan ohjelmoinnin ja testauksen itsenäisesti. Kun näkymä oli saatu toimivaksi, se lisättiin osaksi lopullista sovellusta.

Sovelluksen osien uudelleenkäytettävyys ja muunneltavuus oli selkeä etu PureMVC-kehystä käytettäessä. Jos sovelluksia tehdään paljon ja ohjelmoitavissa sovelluksissa on samanlaisia toimintoja, kehyksellä toteutetut osat voidaan pienellä suunnittelulla helposti käyttää uudelleen uusissa sovelluksissa.

Suurimmaksi eduksi kuitenkin PureMVC:llä toteutetuissa projekteissa nousi sovelluksen hallittavuus. Vaikka ohjelman koko ja koodimäärä kasvaisi, se tapahtuu hallitusti, koska sovelluksen koodi on jaettu pienempiin osiin. Oikein käytettynä PureMVC-ohjelmistokehys estää niin sanotun spagettikoodin ohjelmoimisen.

Hyvänä puolena voidaan myös mainita ohjelmistokehyksen hyvä dokumentaatio ja esimerkit. Hyötynä mediatekniikan verstaan opiskelijoiden kannalta voidaan pitää, että ohjelmistokehyksen opettelu syventää olio-ohjelmoinnin osaamista huomattavasti.

6.2 Arkkitehtuurin ongelmat

Useasti esiin nousseista ongelmista suurin oli ohjelmakoodin sekavuus ja pilkkoutuneisuus. Virheiden etsiminen osiin pilkotusta koodista saattaa olla erittäin vaikeaa. Lisäksi kehyksen oppimiskynnys on melkoisen suuri, jolloin PureMVC:n opettaminen uusille työntekijöille hidastaa projektia. Uuden ohjelmoijan on syytä tuntea olio-ohjelmoinnin peruskäsitteet hyvin, koska käsitteitä esiintyy todella paljon sekä dokumentaatiossa että esimerkeissä.

PureMVC-kehyksellä ohjelmointi lisää ohjelmointikoodin määrää ja projektin pituutta hieman koodin pilkkomisen takia. Lisäksi luokkia

syntyy projektin aikana huomattava määrä, mikä saattaa lisätä entisestään sovelluksen sekavuutta, ja saattaa olla ongelma luokkaohjelmointiin tottumattomalle ohjelmoijalle.

Tyypimuunnokset nousivat myös ongelmaksi PureMVC-kehyksellä tehtyjen projektien aikana. Tyypimuunnoksia pitää tehdä useita kertoja sovelluksen aikana, mikä kasvattaa ohjelmointikoodin määrää entisestään.

6.3 Jatkokehitys

Mediatekniikan verstaan näkökulmasta, jatkokehitystoimenpiteenä voidaan esimerkiksi tutustua johonkin toiseen yleisesti käytettyyn AS3.0-ohjelmistokehykseen. PureMVC on suhteellisen vaikea ohjelmistokehys varsinkin opiskelijalle, joka ei ole aikaisemmin tutustunut olio-ohjelmointiin tai ohjelmistokehyksiin.

PureMVC-kehyyksen käyttöönotto ei ole suositeltavaa, jos projekti on kovin pieni. Siksi jatkokehityksenä olisi hyvä tutustua pienempiin ja yksinkertaisempiin kehyksiin sekä suorittaa vertailua kehysten välillä. Oleellista olisi löytää eri tilanteisiin sopiva kehys, sekä tutkia, milloin kehyyksen käyttö on ylipäättään suositeltavaa.

Jatkokehitysmahdollisuutena on myös oman MVC-kehyyksen kehittäminen sekä dokumentointi mediatekniikan verstaan tarpeisiin. Mediatekniikan verstaalla tehdään paljon projekteja ulkopuolisille asiakkaille ja työntekijöinä toimivat pääasiassa koulun opiskelijat. Tästä syystä työntekijöiden vaihtuvuus on suuri ja työn jälki on vaihtelevaa. Olisi suuri etu verstaalle sekä asiakkaille, jos mediatekniikan verstaan ohjelmointitavat vakiinnutettaisiin. Tämä yhdenmukaistaminen onnistuisi parhaiten oikealla ohjelmistokehyyksen valinnalla.

Verstaan asiakkaat tilaavat usein myös verstaalla tehdyille sovelluksille päivityksiä sekä laajennuksia. Ohjelmistokehyyksen lisäetuna on juuri laajennusten ja päivitysten huomioon ottaminen. Ohjelmat ovat helpommin laajennettavissa sekä sovelluksen ohjelmointilogiikasta on kattava dokumentaatio.

LÄHTEET

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. 2001. Olio-ohjelmointi – suunnittelumallit. Helsinki: Oy Edita Ab. (Gamma,Helm,Johnson & Vlissides 2001)

Hall 2008, C. PureMVC Framework overview with UML. Viitattu 3.10.2010. http://puremvc.org/component/option,com_wrapper/Itemid,35/

Koskimies, K., & Mikkonen T. 2005. OhjelmistoArkkitehtuurit. Helsinki: Talentum Media Oy. (Koskimies & Mikkonen 2005)

Sanders, W. & Cumaranatunge, C. 2007. ActionScript 3.0 Design Patterns. Sebasopol: O'Reilly Media, Inc. (Sanders & Cumaranatunge 2007)

Wikipedia 2010a. Design pattern(computer science). Viitattu 1.10.2010. [http://en.wikipedia.org/wiki/Design_pattern_\(computer_science\)](http://en.wikipedia.org/wiki/Design_pattern_(computer_science)) (Wikipedia 2010a)

Wikipedia 2010b. Command pattern. Viitattu 4.10.2010. http://en.wikipedia.org/wiki/Command_pattern (Wikipedia 2010b)

Wikipedia 2010c. Proxy pattern. Viitattu 7.10.2010. http://en.wikipedia.org/wiki/Proxy_pattern (Wikipedia 2010c)

Wikipedia 2010d. Facade pattern. Viitattu 7.10.2010. http://en.wikipedia.org/wiki/Facade_pattern (Wikipedia 2010d)

Esimerkkisovelluksen lähdekoodi luokitattain

ApplicationFacade.as

```
package
{
    import org.puremvc.as3.interfaces.IFacade;
    import org.puremvc.as3.patterns.facade.Facade;
    import org.puremvc.as3.core.Model;
    import org.puremvc.as3.core.View;
    import org.puremvc.as3.core.Controller;
    import controller.*;

    public class ApplicationFacade extends Facade implements IFacade
    {
        // Notification name constants
        public static const STARTUP:String = "startup";
        public static const ADD_NEW_TEXT:String = "add_new_text";
        public static const RESET_TEXTAREA:String = "reset_text_area";
        public static const REMOVE_TEXT:String = "remove_text";
        public static const TEXTAREA_DATA_CHANGED:String = "textarea_data_changed";

        public static function getInstance():ApplicationFacade
        {
            if (instance == null) instance = new ApplicationFacade();
            return instance as ApplicationFacade;
        }
        // Register commands with the controller
        override protected function initializeController():void
        {
            super.initializeController();
            registerCommand(STARTUP, StartupCommand);
            registerCommand(ADD_NEW_TEXT, TextAreaCommand);
            registerCommand(RESET_TEXTAREA, TextAreaCommand);
            registerCommand(REMOVE_TEXT, TextAreaCommand);
        }
        public function startPureMVC(app:Main):void
        {
            sendNotification(ApplicationFacade.STARTUP, app);
        }
    }
}
```

Main.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
xmlns:s="library://ns.adobe.com/flex/spark"
xmlns:mx="library://ns.adobe.com/flex/mx" creationComplete="startPureMVC();"
xmlns:view="view.components.*" width="340" height="269" >

    <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>
    <fx:Script>
    <![CDATA[
        private var app:ApplicationFacade = ApplicationFacade.getInstance(); //otetaan
        singleton-luokasta yksi instanssi käyttämällä getInstance-metodia
        public function startPureMVC():void
        {
            app.startPureMVC(this);
        }
    ]]>
    </fx:Script>
    <view:TextAreaViewComponent id="textAreaViewComponent"></view:TextAreaViewComponent>
</s:WindowedApplication>
```

StartupCommand.as

```
package controller
{
    import org.puremvc.as3.interfaces.ICommand;
    import org.puremvc.as3.interfaces.INotification;
    import org.puremvc.as3.patterns.command.SimpleCommand;
    import model.TextAreaProxy;
    import view.TextAreaMediator;

    public class StartupCommand extends SimpleCommand implements ICommand
    {
        /* INTERFACE org.puremvc.as3.interfaces.ICommand */
        override public function execute(note:INotification):void
        {
            var mainApp:Main = note.getBody() as Main;
            facade.registerProxy(new TextAreaProxy());
            facade.registerMediator(new TextAreaMediator(mainApp.textAreaViewComponent));
        }
    }
}
```

TextAreaViewComponent.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx" height="248" width="339">

  <fx:Script>
    <![CDATA[
      import flash.events.Event;
      import flashx.textLayout.events.SelectionEvent;

      public static const ADD_BUTTON_PRESSED:String = "add_button_pressed";
      public static const RESET_BUTTON_PRESSED:String = "reset_button_pressed";
      public static const REMOVE_BUTTON_PRESSED:String = "remove_button_pressed";

      private function addButtonClicked():void
      {
        dispatchEvent(new Event(ADD_BUTTON_PRESSED, true));
      }
      private function resetButtonClicked():void
      {
        dispatchEvent(new Event(RESET_BUTTON_PRESSED, true));
      }
      private function removeButtonClicked():void
      {
        dispatchEvent(new Event(REMOVE_BUTTON_PRESSED, true));
      }
      private function textChanged():void
      {
        if (newTextInput.text.length > 0) addButton.enabled = true;
        else if (newTextInput.text.length < 1) addButton.enabled = false;
      }
      private function dataGrindIndexChanged(event:Event):void
      {
        if (dataGrid.selectedItem != null) removeButton.enabled = true;
        else if (dataGrid.selectedItem == null) removeButton.enabled = false;
      }
    ]]>
  </fx:Script>
  <mx:DataGrid id="dataGrid" selectable = "true"
    change="dataGrindIndexChanged(event)" x="0" y="0" width="339" height="159">
    <mx:columns>
      <mx:DataGridColumn dataField="text" headerText="Text"/>
    </mx:columns>
  </mx:DataGrid>
  <s:TextInput id="newTextInput" change="textChanged();" selectable="true" x="10"
    y="167" width="320">add new text here</s:TextInput>
  <s:Button id="addButton" label="add new text" enabled="false"
    click="addButtonClicked();" x="10" y="209"></s:Button>
  <s:Button id="resetButton" label="reset" enabled="true"
    click="resetButtonClicked();" x="240" y="209" width="89"></s:Button>
  <s:Button id="removeButton" label="remove text" enabled="false"
    click="removeButtonClicked();" x="125" y="209" width="89"></s:Button>
</s:Group>
```

TextAreaCommand.as

```
package controller
{
    import org.puremvc.as3.interfaces.ICommand;
    import org.puremvc.as3.interfaces.INotification;
    import org.puremvc.as3.patterns.command.SimpleCommand;
    import model.TextAreaProxy;

    public class TextAreaCommand extends SimpleCommand implements ICommand
    {
        /* INTERFACE org.puremvc.as3.interfaces.ICommand */
        override public function execute(notification:INotification):void
        {
            var noteName:String = notification.getName() as String;
            var noteBody:Object = notification.getBody();
            chooseAction(noteName,noteBody);
        }
        private function chooseAction(noteName:String, noteBody:Object):void
        {
            switch(noteName)
            {
                case ApplicationFacade.ADD_NEW_TEXT:
                    textAreaProxy.addNewText(noteBody as String);
                    break;
                case ApplicationFacade.RESET_TEXTAREA:
                    textAreaProxy.resetData();
                    break;
                case ApplicationFacade.REMOVE_TEXT:
                    textAreaProxy.removeText(noteBody as int);
                    break;
            }
        }
        private function get textAreaProxy():TextAreaProxy
        {
            return facade.retrieveProxy(TextAreaProxy.NAME) as TextAreaProxy;
            // haetaan proxy nimen perusteella, tähän käytetään NAME-vakiota.
        }
    }
}
```

TextAreaProxy.as

```
package model
{
    import mx.collections.ArrayCollection;
    import org.puremvc.as3.interfaces.IProxy;
    import org.puremvc.as3.patterns.proxy.Proxy;

    public class TextAreaProxy extends Proxy implements IProxy
    {
        public static const NAME:String = "TextAreaProxy";
        public function TextAreaProxy(proxyName:String = null, data:Object = null)
        {
            super(NAME, data);
        }
        /* INTERFACE org.puremvc.as3.interfaces.IProxy */
        override public function onRegister():void
        {
            resetData();
        }
        public function get dataInArrayCollection():ArrayCollection
        {
            return this.data as ArrayCollection;
        }
        public function addNewText(newText:String):void
        {
            dataInArrayCollection.addItem({text:newText});
            sendNotification(ApplicationFacade.TEXTAREA_DATA_CHANGED);
        }
        public function resetData():void
        {
            this.data = new ArrayCollection();
            sendNotification(ApplicationFacade.TEXTAREA_DATA_CHANGED);
        }
        public function removeText(selectedIndex:int):void
        {
            dataInArrayCollection.removeItemAt(selectedIndex);
            sendNotification(ApplicationFacade.TEXTAREA_DATA_CHANGED);
        }
    }
}
```

TextAreaMediator.as

```

package view
{
    import flash.events.Event;
    import org.puremvc.as3.interfaces.IMediator;
    import org.puremvc.as3.interfaces.INotification;
    import org.puremvc.as3.patterns.mediator.Mediator;
    import view.components.TextAreaViewComponent;
    import model.TextAreaProxy;

    public class TextAreaMediator extends Mediator implements IMediator
    {
        public static const NAME:String = "TextAreaMediator";
        public function TextAreaMediator(viewComponent:Object = null)
        {
            super(NAME, viewComponent);
        }
        /* INTERFACE org.puremvc.as3.interfaces.IMediator */
        override public function listNotificationInterests():Array
        {
            return [ApplicationFacade.TEXTAREA_DATA_CHANGED];
        }
        override public function onRegister():void
        {
            textArea-
            Comment.addEventListener(TextAreaViewComponent.ADD_BUTTON_PRESSED,handleEvents);
            textAreaComponent.addEventListener(TextAreaViewComponent.RESET_BUTTON_PRESSED,
            handleEvents);
            textAreaComponent.addEventListener(TextAreaViewComponent.REMOVE_BUTTON_PRESSED,
            handleEvents);
            textAreaComponent.dataGrid.dataProvider = textAreaProxy.dataInArrayCollection;
        }

        private function handleEvents(e:Event):void
        {
            switch(e.type)
            {
                case TextAreaViewComponent.ADD_BUTTON_PRESSED:
                    sendNotifica-
                    tion(ApplicationFacade.ADD_NEW_TEXT,textAreaComponent.newTextInput.text);
                    break;
                case TextAreaViewComponent.RESET_BUTTON_PRESSED:
                    sendNotification(ApplicationFacade.RESET_TEXTAREA);
                    break;
                case TextAreaViewComponent.REMOVE_BUTTON_PRESSED:
                    sendNotifica-
                    tion(ApplicationFacade.REMOVE_TEXT,textAreaComponent.dataGrid.selectedIndex);
                    break;
            }
        }

        override public function handleNotification(notification:INotification):void
        {
            switch(notification.getName())
            {
                case ApplicationFacade.TEXTAREA_DATA_CHANGED:
                    textAreaComponent.dataGrid.dataProvider = tex-
                    tAreaProxy.dataInArrayCollection;
                    break;
            }
        }
    }
}

```

TextAreaMediator.as

```
override public function handleNotification(notification:INotification):void
{
    switch(notification.getName())
    {
        case ApplicationFacade.TEXTAREA_DATA_CHANGED:
            textAreaComponent.dataGrid.dataProvider = textAreaProxy.dataInArrayCollection;
            break;
    }
}

override public function onRemove():void
{
    textAreaComponent.addEventListener(view.components.TextAreaViewComponent.ADD_BUTTON_PRESSED,
    handleEvents);
    textAreaComponent.addEventListener(view.components.TextAreaViewComponent.RESET_BUTTON_PRESSED,
    handleEvents);
    textAreaComponent.addEventListener(view.components.TextAreaViewComponent.REMOVE_BUTTON_PRESSED,
    handleEvents);
}

private function get textAreaComponent():TextAreaViewComponent
{
    return this.viewComponent as TextAreaViewComponent;
}

private function get textAreaProxy():TextAreaProxy
{
    return facade.retrieveProxy(TextAreaProxy.NAME) as TextAreaProxy;
}
}
```