Olio-ohjelmointi käytännössä käyttäen hyväksi avointa tietoa, graafista käyttöliittymää ja karttaviitekehystä

Versio 1.0

Antti Herala, Erno Vanhala ja Uolevi Nikula

Lappeenrannan teknillinen yliopisto
LUT School of Business and Management
Laboratory of Innovation and Software
PL 20
53851 Lappeenranta

Lappeenrannan teknillinen yliopisto
Lappeenranta University of Technology

LUT School of Business and Management
Software Engineering and Information Management

LUT Scientific and Expertise Publications

Oppimateriaalit – Lecture Notes 6

Antti Herala, Erno Vanhala, Uolevi Nikula

Olio-ohjelmointi käytännössä käyttäen hyväksi avointa tietoa, graafista käyttöliittymää ja karttaviitekehitystä

ISBN 978-952-265-755-8
ISBN 978-952-265-756-5 (PDF)
ISSN-L 2243-3392
ISSN 2243-3392

Lappeenranta 2015

Sisällysluettelo

1. Johdanto

Tässä opassarjan toisessa osassa siirrytään olio-ohjelmoinnista käsittelemään enemmän oikean elämän esimerkkejä. Näitä esimerkkejä käydään läpi käyttäen avointa dataa, siihen liittyvää XML:ää ja niiden yhteen liittämistä karttapohjien rakentamisessa. Lisäksi opas käsittelee myös graafisen käyttöliittymän luomisen Javan graafista kirjastoa käyttäen.

Opas sisältää graafisen käyttöliittymän luonnissa Javaa sekä karttaviitekehyksen yhteydessä JavaScriptiä. Tämän oppaan tarkoitus ei ole enää opettaa ohjelmointia vaan esittää esimerkkien kautta nykyaikaisia sovellutuksia tietotekniikan saralla. Käytetyt konseptit ovat linkitettyjä toisiinsa niin, että opas lähtee liikkelle matalan tason datasta ja kasvaa lopulta toimivaksi karttajärjestelmäksi. Karttajärjestelmän jälkeen käydään myös läpi vielä graafisen käyttöliittymän perusperiaatteita tarkoitukseen, jossa verkkoa ei ole mahdollista käyttää tai halutaan käyttää työpöytäsovellusta.

2. Versionhallinta

Versionhallinta on järjestelmä, joka pitää kirjaa tiedostoon tai tiedostoihin liittyvistä muutoksista niin, että tiedostosta on mahdollista palauttaa tietty versio tarpeen mukaan. Tavallisesti versionhallinta yhdistetään lähdekoodin tallentamiseen ja hallinnointiin, mutta versionhallinta kattaa myös kaiken muun projektiin liittyvän, esimerkiksi dokumentaatiot, tarvittavat tiedostot ja testaustapaukset.

Versionhallinnan avulla on mahdollista säilyttää kaikki projektin aikana syntyneet välivaiheet, esimerkiksi graafisen käyttöliittymän kaikki erilaiset asetelmat on mahdollista tallentaa ja hakea nopeasti ja vaivattomasti. Versionhallinnan avulla tiedostojen katoaminen ja korruptoituminen voidaan korjata helposti päivittämällä paikalliselle työasemalle tarvittavat tiedostot. Järjestelmän avulla on myös mahdollista kehittää ohjelman eri osia useassa paikassa yhtäaikaisesti ja sen avulla voidaan valvoa sitä, kuka teki minkäkin päivityksen versionhallintaan, jolloin virheitä ja yksityiskohtia voi tiedustella suoraan kyseiseltä kehittäjältä, joka muutoksen tuotti.

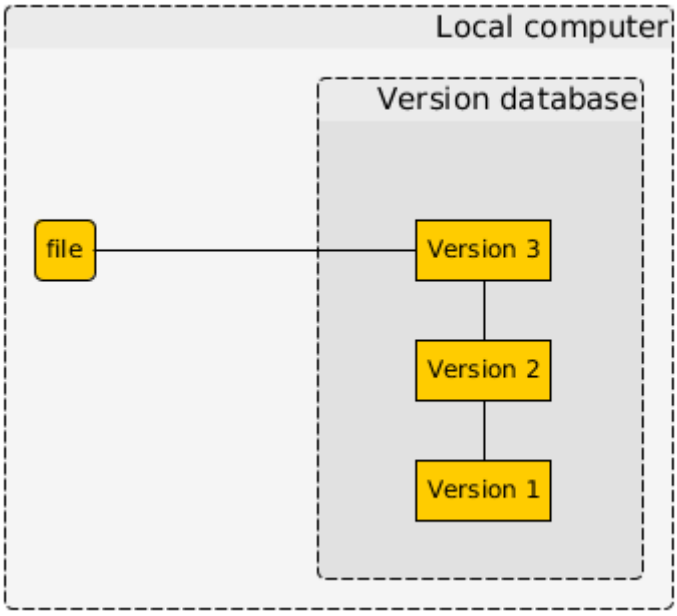
2.1. Versionhallintajärjestelmät

Versionhallintajärjestelmät voidaan jakaa kolmeen luokkaan: paikalliset, keskitetyt ja hajautetut versionhallinnat. Seuraavassa käydään hieman läpi näitä kolmea eri luokkaa.

Paikallinen versionhallinta on nimensä mukaisesti toiminnassa vain yhdessä järjestelmässä, eikä se sisällä mitään kommunikaatiota muiden järjestelmien kanssa. Paikallisen versionhallinnan toiminta on esitetty kuvassa 1.1. Paikallinen järjestelmä kehitettiin alkujaan yhden järjestelmän sisäiseksi versionhallinnaksi, missä järjestelmä pitää sisällään versiotietokannan, josta käyttäjä voi hakea tarvetta vastaavan version tiedostoista. Järjestelmä on kyllä yksinkertainen, mutta samalla hyvin virheherkkä, sillä se vaatii käyttäjää ylläpitämään ja päivittämään kaikkia tiedostoja manuaalisesti, jolloin inhimillinen virhe on suuri vaikuttaja. Käyttäjä voi esimerkiksi unohtaa, missä versiossa ominaisuus on tai voi päivittää vahingossa väärän tiedoston sisällön, tuhoten kyseisen version joko osin tai täysin.

Suosittu työkalu paikalliseen versionhallintaan on RCS, joka on asennettavissa vielä jopa Mac OS X-käyttöjärjestelmään. RCS toimii niin, että se hallinnoi tiettyä osaa kovalevystä,

jonne se tallentaa versioita erillisinä joukkoina, jolloin se voi hakea minkä tahansa joukon tarpeen mukaan. Samankaltainen toiminto on Windowsin palautuspisteominaisuus, jolla on mahdollista palauttaa järjestelmä siihen tilaan, missä se vielä toimi vakaasti.

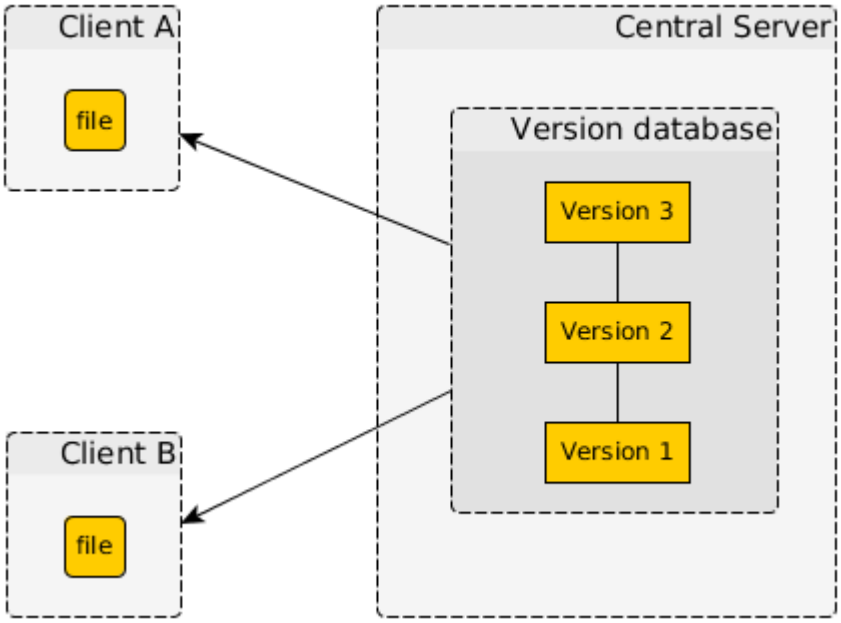


Kuva 2.1. Paikallinen versionhallintajärjestelmä

Keskitetty versionhallintajärjestelmä tarkoittaa mallia, jossa versiot on tallennettu erilliselle palvelimelle, josta asiakasohjelmistot pääsevät käyttämään niitä. Keskitetty malli tarjoaa kehittäjille mahdollisuuden työskennellä saman projektin parissa eri työpisteiltä niin, että he voivat työstää eri osia ohjelmistosta samanaikaisesti ja lopuksi yhdistää ne. Molemmat kehittäjät kuitenkin päivittäessä tiedostoja luovat oman versionsa hallintajärjestelmään. Haittapuolena järjestelmässä on juuri kyseinen ominaisuus, missä kaksi kehittäjää muuttaa samaa tiedostoa samanaikaisesti ja päivittävät nämä versionhallintaan. Tällöin on vaarana, että toisen kehittäjän tekemät muutokset eivät päivity viimeisimpään versioon, josta modernit versionhallintajärjestelmät tosin huomauttavat. On myös huomattava, että riski koko projektin katoamiselle on suuri, kun kaikki tiedostot on tallennettu vain yhdelle palvelimelle. Keskitetty versionhallintajärjestelmä tuo kuitenkin huomattavia etuja varsinkin projektin ylläpitoon, kun tuotetta voidaan tarkastella jatkuvasti jo kehitysvaiheessa.

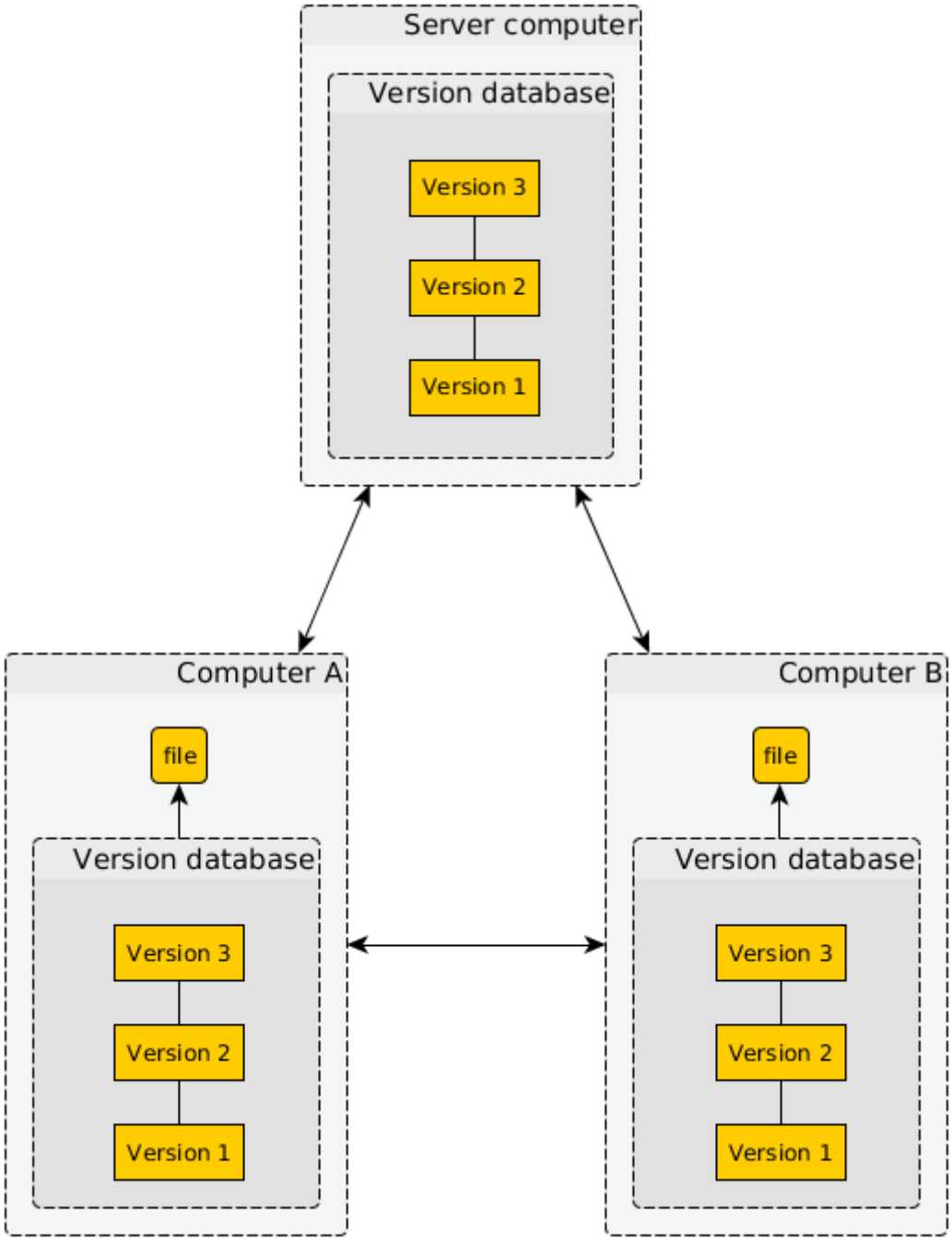
Suosittuja keskitetyn mallin työkaluja ovat Subversion (SVN), CVS ja Perforce. Työkalut toimivat yksinkertaistettuna niin, että tiedostot ovat tallennettuna yhdelle, kaikki versiot

sisältävälle, keskitetylle palvelimelle, josta asiakasohjelmistot voivat käydä hakemassa tiedostoja tarpeen mukaan. Keskitetty versionhallinta on ollut pitkään laajalti käytössä versioiden hallinnassa ja sen toiminta on esitetty kuvassa 2.2.



Kuva 2.2. Keskitetty versionhallintajärjestelmä (tiedoston lataaminen palvelimelta)

Keskitettyjen versionhallintajärjestelmien tilalle on viime aikoina alkanut nousta hajautetut versionhallintajärjestelmät (kuva 2.3.), joissa asiakas ei enää kopioi yhtä pientä osaa versiosta, vaan jokainen lataaminen palvelimelta tarkoittaa koko tietovaraston (*repository*) lataamista. Tämän avulla estetään keskitetyn järjestelmän heikkous, jossa palvelimen kaatuminen tarkoittaa kaiken lähdekoodin häviämistä. Hajautetussa versionhallinnassa jokainen asiakasohjelmisto pitää siis sisällään kaiken sen tiedon, mitä alkuperäinen palvelinkin, jolloin palvelin voidaan alustaa uudelleen minkä tahansa asiakasohjelmiston kautta. Jokainen lataus on siis käytännössä varmuuskopio alkuperäisestä datasta. Tällaisia työkaluja ovat esimerkiksi Git, Mercurial, Bazaar ja Darcs.



Kuva 2.3. Hajautettu versionhallintajärjestelmä

2.2. Versionhallintapalveluita

Edellä esitetyt versionhallintajärjestelmät olisivat hyvin hankalia ja raskaita ottaa käyttöön varsinkin pienissä ohjelmistoyrityksissä, koska yritys joutuisi itse hankkimaan vaadittavat palvelimet, palvelinsovellukset ja asiakasohjelmistot. Tehtävää helpottamaan on kehitetty

verkossa toimivia palvelualustoja, joita on olemassa useampia ja ne tukevat malliltaan hajautettuja versionhallintajärjestelmiä, kuten esimerkiksi Gitiä ja Mercurialia. Nämä palvelualustat nojaavat tallennuksessa aina verkkoon ja niitä on mahdollista käyttää myös verkkoselaimen läpi.

Bitbucket on verkossa toimiva versionhallintapalvelu, joka tukee sekä Mercurialia että Gitiä. Palvelualusta tukee sekä ilmaisia että maksullisia tilejä, joista ilmaiset tilit ovat käyttöoikeuksiltaan hieman rajatumpia kuin maksulliset. Käyttäjät voivat Bitbucketissa luoda sekä yksityisiä että julkisia tietovarastoja (*repository*), joista yksityisiä voi käyttää vain käyttäjä itse ja julkisia tietovarastoja käytetään tavallisesti avoimen lähdekoodin projekteissa.

GitHub on Bitbucketin tavoin verkossa toimiva versionhallintapalvelu, mutta se tukee ainoastaan Gitiä, kuten nimestä voi päätellä. GitHub tarjoaa käyttäjille ilmaisia tilejä julkisiin projekteihin ja yksityisiä tietovarastoja maksua vastaan. GitHub on tällä hetkellä maailman suurin versionhallintapalvelu, hallinnoiden jo yli 13,1 miljoonaa tietovarastoa (17.6.2014). GitHubia käytetään myös muuhun kuin koodin hallinnoimiseen; esimerkiksi lakialoitteita ja avoimia oppikirjoja on tallennettuna GitHubin tietovarastoissa.

Google Code on Googlen tarjoama verkkopalvelu, jossa on mahdollista luoda ohjelmistoprojekteja käyttäen Googlen tarjoamia kehitystyökaluja ja useita ohjelmointirajapintoja. Sivusto sisältää dokumentaation kaikista Googlen tarjoamista rajapinnoista, kuten Google Maps, Youtube ja Google Apps. Palveluna se ei ole yhtä kattava kuin edellä mainitut, mutta Googlen työkaluja käytettäessä se tarjoaa tukea ja tarpeellista säilytystilaa avoimen lähdekoodin projekteihin. Tietovarastona Google Code tukee Gitiä, Mercurialia ja SVN:ää.

2.3. Versionhallinta ja IDE

Koska versionhallinta on ollut jatkuvasti kasvava tapa säilöä projekteja, on se havaittu myös kehitystyökalujen kehityksessä. Esimerkiksi NetBeans tarjoaa täyden tuen Git-, Mercurial- ja SVN-versionhallintojen käyttöön ja muitakin järjestelmiä tuetaan erillisten asennusten kautta. Tällöin suoraan IDE:stä on mahdollista alustaa, ladata, muokata, päivittää ja tallentaa muutoksia versionhallintaan. Tällöin eliminoidaan hidastava askel, jossa ensin koodia on muutettava, se on tallennettava tiedostoon ja tiedostot on lisättävä versionhallintaan erikseen.

Tämä oli käytännössä versionhallinnan menetelmä aikaisemmin ja varsinkin keskitetyssä versionhallinnassa hyvin yleinen tapa. Tarkemmat ohjeet kiinnostuineille NetBeansin ja Gitin yhteiskäyttöön löytyvät englanniksi osoitteesta: https://netbeans.org/kb/docs/ide/git.html

3. Avoin data (open data)

"*Open data is data that can be freely used, reused and redistributed by anyone - subject only, at most, to the requirement to attribute and sharealike.*" - Open Knowledge Foundation [URL: http://opendatahandbook.org/en/what-is-open-data/]

Avoimella datalla ymmärretään kaikki sellainen data, jonka käyttöä ei ole millään tasolla rajoitettu. Sen käyttö, uudelleenkäyttö ja jakaminen on siis mahdollista kaikille riippumatta siitä, missä ja miten dataa käyttää. Jotta data olisi avointa, tulee sen täyttää tietyt kriteerit:

- **Helppo saatavuus ja hyvä käytettävyys**
    - Datan on oltava helposti saatavilla, jotta jokainen data tarvitseva voi päästä siihen käsiksi niin helposti kuin mahdollista, esimerkiksi ladattavissa verkosta. Data on oltava myös käytännöllisessä muodossa niin, että sitä on mahdollista hyödyntää helposti ja ilman hankinnasta johtuvia kustannuksia.
- **Uudelleenkäyttö ja jakaminen**
    - Datan on oltava saatavilla niin, että se mahdollistaa uudelleenkäytön ja datan jakamisen eteenpäin, jotta dataa voi käyttää myös muiden tietojoukkojen kanssa. Datan jakaminen ei saa sisältää ehtoja, joissa kiellettäisiin mahdollinen uusiokäyttö tai yhdistäminen.
- **Yleinen osallistuminen**
    - Jokaiselle on oltava mahdollista käyttää, uudelleenkäyttää ja jakaa dataa, sitä ei saa rajoittaa esimerkiksi tietyltä alalta, henkilöltä tai ryhmältä. Esimerkiksi data, jota saa käyttää vain tuottamattomassa toiminnassa, kuten opetus tai tutkimus, on rajoitus tuottavalta toiminnalta. Samalla tavalla datan käyttö vain koulutukseen on vastoin avoimen datan määritelmää. Vaikka tarkoitus olisi estää vain suuria yrityksiä hyötymästä datasta ja antaa tällä etua pienemmille yrityksille, ei puhuta enää avoimesta datasta.

Miksi nämä säännöt ja kriteerit ovat tärkeitä? Avointa dataa hyödynnetään juuri sen vuoksi, että voidaan luoda ja kehittää järjestelmien ja organisaatioiden välistä yhteentoimivuutta. Täysin rajoittamaton data tarjoaa mahdollisuuksia innovoida erilaisia tuotteita ja prosesseja kattavasti eri alojen kesken eikä data olisi enää yhteistyötä rajoittava tekijä. Yhteentoimivuus tarjoaa paremman mahdollisuuden suurten ja kompleksisten ongelmien ja järjestelmien kehittämiseen. Avoimen datan avulla järjestelmä voidaan jakaa komponentteihin uudella

tavalla ja omalla tavallaan se voi myös parantaa ja muokata komponenttien luomista rakenteellisemmaksi, poistamalla saatavan datan aiheuttamat rajoitteet jo suunnitteluvaiheessa. Esimerkkinä datan avoimuudesta voidaan muistella myyttiä Baabelin tornista, jossa kommunikoinnin eli yhteentoimimisen mahdottomuus (yhteisen datan puute) kaatoi koko projektin [Raamattu, 1 Moos 11:1-9].

Jotta avoimuutta voitaisiin käyttää hyväksi myös avoimen tai suljetun lähdekoodin ohjelmistoja ja palveluita tuottaessa, on avoin data tärkeä osa kokonaisuutta. Avointa materiaalia tulisi voida liittää toiseen avoimeen materiaaliin, jolloin nämä kaksi irrallista komponenttia luovat laajemman kokonaisuuden kuin kummastakaan voisi rakentua itsenäisesti. Tästä hyvänä esimerkkinä toimii palveluorientoitunut arkkitehtuuri (*service-oriented architecture*), missä on tarkoituksena luoda kattavia ja helposti käytettäviä palveluita, jotka toimivat keskenään omien tietojoukkojensa avulla.

Avoimuuden avulla mahdollistetaan suurten kokonaisuuksien luonti pienistä tietojoukoista ja avoimuuden määrittelyn ansiosta nämä tietojoukot on mahdollista liittää yhdeksi toimivaksi kokonaisuudeksi ilman ongelmia. Näin saadaan aikaan pienistä komponenteista laaja toiminnallisuus, joka tuottaa enemmän arvoa käyttäjille kuin kaikki pienet, erilliset osaset yhteensä. Avoin data on vain pieni osa suurta avoimuuden kokonaisuutta, johon lukeutuu myös esimerkiksi avoimen lähdekoodin ohjelmistot.

Avoimen tiedon puolestapuhujat ovat esittäneet syitä, miksi avointa dataa pitäisi olla tarjolla. Esimerkkeinä näistä syistä ovat julkilausumat, missä toistetaan kaiken datan kuuluvan koko ihmiskunnalle. Tämän lisäksi valtion tuottama data tulisi olla julkista, sillä datan hankinta, luominen ja kehitys on rahoitettu pääasiassa verorahoilla, jolloin kaikkien tuloksien tulisi olla julkisesti käytettävissä. On myös valitettavan selvää, että luonnon lakeja ja yleisiä totuuksia ei voi suojata tekijänoikeudella, sillä ne ovat olleet käytännössä aina olemassa ja ne on vain löydetty. Siksi kenelläkään ei tulisi olla niihin yksinoikeutta. Myös tieteellinen tutkimus hyötyisi datan avoimuudesta, jolloin samaa asiaa ei tarvitsisi tutkia useaan kertaan. On tosin totta, että osana tieteellistä tutkimusta on muiden tekemän tutkimuksen uudelleen luominen ja saman tuloksen tavoittelu (replikointi), mutta jos näistä samankaltaisia tutkimuksista ei ole tietoa tutkimusta aloittaessa, on se työtä hidastavaa.

Avointa dataa on myös kritisoitu, sillä avoimen datan avulla luotu tuote, joka tuottaa voittoa vain pienelle joukolle, varsinkin jos tiedon hankintaan on käytetty julkisia varoja, ei ole

julkiselta sektorilta vastuullista rahankäyttöä. Avoimuus on myös haitallista silloin, jos data sisältää yksityistä tietoa, esimerkiksi yrityssalaisuuksia, jolloin sen jakaminen voi aiheuttaa haittaa ja vahinkoa yrityksille. Datan etsintään annetut rahat eivät myöskään välttämättä tuota sellaista lopputulosta kuin rahoittaja tarvitsisi ja oletti saavansa. Lisäksi dataa ei ole aina mahdollista tarjota sellaisessa muodossa, että se olisi aina kaikkien luettavissa, tai jos se on mahdollista, se on hyvin epätuottavaa ja kallista. Data ei ole avointa myöskään silloin, jos siihen vaikuttaa tai mahdollisesti vaikuttaa jokin poliittinen paine, markkinoiden aiheuttama paine tai laillinen paine, tästä esimerkkinä valtion biasoima data omasta varallisuudestaan. On tietysti myös huomattava, että rajoitetulla pääsyllä varustettu data ei ole avointa, edes silloin, jos data on rajoitettuna tai tarjolla vain osan aikaa tai siihen kiinni pääseminen on maksullista. Jopa rekisteröitymisen vaatiminen, oli se sitten maksullinen tai maksuton, on avoimen datan vastaista.

Esimerkkejä avoimesta tiedosta löytyy:

- Julkisen hallinnon avoin data
  - http://www.suomi.fi/suomifi/tyohuone/yhteiset_palvelut/avoin_data/
  - https://www.avoindata.fi/
- Ilmatieteen laitoksen avoin data
  - https://ilmatieteenlaitos.fi/avoin-data

Avoin data voidaan luokitella niin sanotulla viiden tähden järjestelmällä [http://5stardata.info/]:

1. Data on jaettu verkossa noudattaen avoimen datan periaatteita missä tahansa formaatissa, esimerkiksi PDF
2. Data on jaettu niin, että se on rakenteellista ja koneluettavaa, esimerkiksi Excel
3. Data käyttää alustariippumatonta tallennusmuotoa, esimerkiksi CSV tai XML
4. Data löytyy verkosta tietyn osoitteen päästä, jolloin siihen on mahdollista viitata
5. Datan lisäksi tarjotaan linkitys muuhun dataan, jolloin lisätään kontekstin määrää

Tavallisimpia datan siirtorakenteita verkon välityksellä ovat CSV (Comma Separated Values) ja XML, mutta varsinkin suuremman datamäärän esittämiseen ja lähetykseen XML on hyvin paljon suositumpi tallennusmuoto. XML mahdollistaa myös tietynlaisen rakenteellisuuden vaatimisen, esimerkiksi varmistamaan ettei datassa oleva tärkeä tieto ole kadonnut kirjoitusvirheen vuoksi. Tällaista ominaisuutta ei CSV tallennusmuotona tarjoa. Seuraavassa

luvussa käsitellään tarkemmin XML:n perusteet, joiden avulla kyseistä tiedon tallennusmuotoa voidaan käyttää ja soveltaa.

4. XML

"*Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.*" - Wikipedia [URL: http://en.wikipedia.org/wiki/XML]

XML on lyhenne sanoista *eXtensible Markup Language* ja se on hyvin läheistä sukua HTML:lle eli *Hypertext Markup Language*lle. Molemmat ovat merkintäkieliä sekä hyvin tärkeitä nykyisen kaltaisen Webin olemassaololle. Merkintäkieli tarkoittaa sitä, että kielet perustuvat tageihin eli merkintöihin (oppaassa tästä lähin käytetään nimitystä tägi), joiden sisään kaikki tarvittava informaatio on sijoitettu. Alku- ja lopputägit sekä niiden väliin sijoitettu tieto on nimeltään elementti.

XML ja HTML kuitenkin eroavat toisistaan siinä, että HTML:ää käytetään verkkosivujen rakentamisen perusrunkona, eli sitä käytetään esittämään tietoa näytöllä ja sitä luodessa keskitytään pääasiassa siihen, miltä esitettävä tieto tulee näyttämään. XML taas on suunniteltu täysin tiedon siirtoa varten, jolloin sen sisältämää tietoa ei tulosteta mihinkään päätelaitteeseen, tietoa ainoastaan liikutellaan paikasta toiseen ja tallennetaan. XML siis mielletään enemmän tietokannaksi, kuten relaatiotietokanta, kuin ohjelmointi- tai kyselykieleksi.

Tässä yhteydessä voidaan ajatella, että tietokannat on luotu juuri sitä varten, että niihin voidaan varastoida tietoa ja tätä tietoa voidaan lähettää verkon yli helposti. Huomionarvoista kuitenkin on tavallisen tietokannan ja XML-tiedoston kokoerot sekä fyysisesti että rivimääräisesti. Tietokannat vaativat tavallisesti täysin oman suunnittelutyön ohjelmistoprojekteissa, sillä ne ovat usein hyvin massiivisia ja jokaisen taulun väliset yhteydet ovat hyvin merkityksellisiä. XML on hieman vapaampi muoto tietokannasta, sillä siihen tallennettu tieto pysyy aina tägien välissä ja tietoa voidaan hakea ja tallentaa tägien avulla. Tämä ei siis käytännössä mahdollista suuria tietokokonaisuuksia ja voidaan puhua useista sadoista tai ehkä jopa tuhansista riveistä, kun taas relaatiotietokanta sisältää jopa miljoonia rivejä. Toinen eroavaisuus XML:n ja tietokannan välillä on se, että tietokanta on vahvasti rakennettu, jonka vuoksi jo luotuja tauluja ei ole mahdollista dynaamisesti muokata, ainoastaan sen sisältämää dataa. Esimerkiksi tietokannassa on tallennettuna käyttäjän etunimi ja sukunimi, mutta haluttaisiin lisätä rivi, johon on lisätty myös käyttäjän toinen nimi. Tavallinen tietokanta ei tähän taivu, mutta XML on hieman hövelimpi sen osalta, jolloin

tarvittava elementti voidaan lisätä dynaamisesti silloin, jos se tulee materiaalissa vastaan. Esimerkkiä on yksinkertaistettu hieman, jonka vuoksi tehtävä ei toimi käytännössä aivan näin helppo. Pääasiana lukijan on siis huomattava se, että XML voi korvata tavallisen tietokannan erityisesti Web-sivujen ja -applikaatioiden tiedontallennus- ja siirtomuotona, jolloin tietoa on ketterämpää hakea ja vapaampaa tallentaa.

XML on siitä tavaton kieli, että sillä on hyvin löyhä valmis syntaksi, jonka tulee täyttyä. Esimerkiksi HTML on hyvin tarkka elementtien sisällöstä, esimerkiksi se vaatii, että jokainen tiedosto pitää sisällään <head></head>- ja <body></body>-elementtiparit. XML ei ole edes näin vaativa syntaktisesti, vaan tiedoston luojan on mahdollista ja jopa vaadittua itse keksiä, miten nimeää elementtinsä. Luodaan esimerkki note, joka pitää sisällään tiedon lähettäjästä, vastaanottajasta, otsikosta ja sisällöstä.

Esimerkki 4.1. Yksinkertainen XML-taltiointi

```
<note>
        <to>Erno</to>
        <from>Timo</from>
        <header>Hei, me ohjelmoidaan!</header>
        <message>Oletko muistanut ohjelmoida?</message>
</note>
```

Esimerkissä 4.1. nähdään, miten XML-notaatio toimii käytännössä. Jokaista avaavaa tägiä kohden on siis oltava suljettu tägi tai sen on lopetettava itse itsensä, josta myöhemmin. Tägin avaaminen tapahtuu kulmasulkeiden avulla, eli viestin avaava tägi on <note> ja se tarvitsee parikseen sulkevan tägin. Sulkeva tägi on muuten identtinen kulmasulkeiden välissä oleva tunniste, mutta se pitää sisällään myös päättymistä ilmaisevan /-merkin, eli </note>.

Jos esimerkissä olevia note-elementtejä olisi enemmän, puhuttaisiin suuremman tietomäärän tallennuksesta. On huomattava, että jokainen XML-taltiointi tarvitsee *juuren* (*root*). Eli on siis oltava jokin sellainen elementti (esimerkissä note), jonka sisälle kaikki informaatio on tallennettu. Tällöin tietoa purkaessa etsitään ensimmäisenä juurielementti, jonka sisältä kaikki tieto löytyy. Havainnollistetaan tätä hieman esimerkin avulla, jossa luomme kirjaston.

Esimerkki 4.2. Yksinkertainen kirjasto

```
<library>
        <book>
                <title>UML</title>
                <author>Hans-Erik Eriksson</author>
```

```
            <location>85.4</location>
        </book>
        <book>
            <title>e-Business 2.0 - Roadmap for Success</title>
            <author>Ravi Kalakota</author>
            <location>77.3</location>
        </book>
        <book>
            <title>Software engineering economics</title>
            <author>Barry W. Boehm</author>
            <location>21.4</location>
        </book>
</library>
```

Esimerkin 4.2. juurielementtinä toimii siis library. Jos esimerkkiä vietäisiin vielä edemmäs vaikka kuntatasolle, meillä olisi juurielementtinä kunta, jonka sisällä olisi useita kirjastoja. Esimerkki on yksinkertainen, jotta nähtäisiin, miten suuri määrä samankaltaista tietoa on mahdollista tallentaa XML-muodossa. XML myös sisältää toiminnallisuutta varmistaakseen, miten ja kuinka paljon tietoa on mahdollista lisätä. Tätä kutsutaan nimellä DTD eli *Document Type Definition*, jonka avulla on mahdollista rajoittaa sitä, mitä tietoa XML-taltiointiin voi tallentaa. Tämä tosin katoaa jo kauas kurssin aiheen ulkopuolelle, joten jätetään asia siihen, aiheeseen voi tutustua esimerkiksi W3Schoolsin kautta osoitteessa http://www.w3schools.com/dtd/.

Esimerkissä 4.2. on se harmillinen puoli, että joillakin kirjoilla on useampia kirjoittajia ja kirjoittajat olisi hyvä saada tallennettua niin, että heidän sukunimensä ja etunimensä voisi irrottaa erillisiksi elementeiksi sekä lisätä mahdollisuuden keskimmäiselle nimelle. Kehitetään siis kirjastoamme hieman lisää seuraavassa esimerkissä.

Esimerkki 4.3. Laajennettu kirjasto

```
<library>
        <book>
            <title>UML</title>
            <authors>
                <author>
                    <first-name>Hans-Erik</first-name>
                    <lastname>Eriksson</lastname>
                </author>
                <author>
                    <first-name>Magnus</first-name>
                    <lastname>Penker</lastname>
                </author>

            </authors>
```

```
            <location>85.4</location>
        </book>
        <book>
            <title>e-Business 2.0 - Roadmap for Success</title>
            <authors>
                <author>
                    <first-name>Ravi</first-name>
                    <lastname>Kalakota</lastname>
                </author>
                <author>
                    <first-name>Marcia</first-name>
                    <lastname>Robinson</lastname>
                </author>

            </authors>
            <location>77.3</location>
        </book>
        <book>
            <title>Software engineering economics</title>
            <authors>
                <author>
                    <first-name>Barry</first-name>
                    <middle>W</middle>
                    <lastname>Boehm</lastname>
                </author>
            </authors>
            <location>21.4</location>
        </book>
</library>
```

Kuten huomaamme, kirjasto laajeni huomattavasti. Nyt jokaiselle kirjalle lisättiin elementti authors, joiden sisälle on mahdollista laittaa niin monta kirjailijaa kuin tarve vaatii. Huomaa, kuinka radikaalisti XML-tiedoston pituus kasvaa sitä mukaa, kun dataa lisätään ja luokitellaan eri tavalla. Tämä ei kuitenkaan koneellista käsittelyä estä tai edes hidasta, sillä datan lukeminen on nopeaa lukea läpi. Kun ohjelman avulla käydään läpi XML-tietoa ja etsitään esimerkiksi juuri kirjojen nimiä, on suoritus hyvin nopea, sillä ohjelman tarvitsee lukea vain ja ainoastaan aloitustägi, jolloin se tietää jättää tiedon käsittelemättä, jos se ei ole pyydettyä tietoa.

Esimerkin lisäksi eri tägeihin on mahdollista lisätä vielä erilaisia attribuutteja, joiden avulla voidaan ilmoittaa esimerkiksi kirjan kieli tai kategoria. Nämä asiat liittyvät tavallisesti hyvin kiinteästi juuri tiettyyn kirjaan, jolloin ne on hyvä sijoittaa jo tägin sisään valmiiksi arvoksi eikä luoda erillistä kenttää tällaisen tiedon esittämiseen. Tämä on kuitenkin täysin luojan harkinnanvarainen valinta, luodaanko tiedosta erillinen elementti vai luodaanko siitä

attribuutti, kieli ei tähän ota kantaa. Otetaan vielä esimerkiksi vain yksittäinen kirja, jolle on lisätty attribuuttiarvoiksi kategoria sekä kirjoituskieli.

Esimerkki 4.4. Laajennettu kirja

```
<book category=”Engineering” lang=”en”>
        <title>Software engineering economics</title>
        <authors>
                <author>
                        <first-name>Barry</first-name>
                        <middle>W</middle>
                        <lastname>Boehm</lastname>
                </author>
        </authors>
        <location>21.4</location>
</book>
```

Kuten aikaisemmin sanottiin, XML on suosittu tiedon tallennusmuoto verkon yli jaettaessa, myös avoimen datan puolella. Tästä hyvänä ja kattavana esimerkkinä on Maanmittauslaitoksen tarjoama maastotietokanta (https://tiedostopalvelu.maanmittauslaitos.fi/tp/kartta), josta on mahdollista ladata mitä tahansa Suomen maastoon liittyvää dataa. Tämä avoin data on tallennettuna XML-muodossa ja sieltä on mahdollista etsiä esimerkiksi kaikki Suomen tiet ja osoitteet, mahdollistaen esimerkiksi oman navigointijärjestelmän kehittämisen varmasti ajankohtaisella ja tarkalla datalla. Seuraavassa luvussa käsitellään hieman enemmän karttojen ohjelmointiin liittyviä käsitteitä ja teknistä toteutusta.

5. Kartat ja ohjelmointi

"*Many maps are static two-dimensional, geometrically accurate (or approximately accurate) representations of three-dimensional space, while others are dynamic or interactive, even three-dimensional. Although most commonly used to depict geography, maps may represent any space, real or imagined, without regard to context or scale; e.g. brain mapping, DNA mapping and extraterrestrial mapping.*" - Wikipedia [URL: http://en.wikipedia.org/wiki/Map]

Kartat ovat olleet tärkeässä roolissa ihmisen jokaisessa historian vaiheessa. Karttojen avulla on luotu malleja ja suunnitelmia kaupungin infrastruktuurin mallintamiseen ja suunnitteluun, niiden avulla on suunniteltu sotastrategioita ja tärkeimpänä käyttökohteena on ollut niiden käyttö suunnistukseen sekä maalla että merellä.

Eniten käytettyjä karttoja ovat nykyisin tiekartat, jotka sisältävät myös omalta osaltaan navigointiin käytettävät kartat eli merikortit ja ilmailukartat. Näiden lisäksi tiekarttoihin lukeutuvat myös rautatieverkostoa esittävät kartat sekä lenkkeily- ja pyöräilykartat. Pääosa nykyisin toimitetuista piirretyistä kartoista tulee paikallisilta toimijoilta, kuten kunnilta ja julkishallinnon virastojen kautta, jotka toimittavat erilaisia maanmittausoperaatioita. Erityisesti karttojen paikkansapitävyydestä vastaa Maanmittauslaitos (http://www.maanmittauslaitos.fi/).

Paikallisen ja pysyvän tiedon lisäksi karttojen avulla on mahdollista esittää useita erilaisia maastonmuotoja sekä analyysin tuloksia. Nykyaikana paperiset kartat alkavat korvautua sähköisillä, joten erilaisen datan esittäminen kartan avulla on yksinkertaistunut, kun jokaiselle analyysille ei tarvitse tuottaa fyysistä karttaa, joista tietoa yhdistettäisiin. Tällainen analyysityökalu on esimerkiksi paikkatietoikkuna, joka löytyy osoitteesta http://www.paikkatietoikkuna.fi/web/fi/etusivu ja on Maanmittauslaitoksen ja Liikenneviraston yhdessä tuottama karttapalvelu.

5.1. Karttatyyppejä

Karttatyyppejä on useita erilaisia ja niiden sisällöt vaihtelevat suuresti sen mukaan, mitä sisältöä kartan on tarkoitus esittää. Tällaisia karttoja ovat poliittiset kartat, fyysiset kartat,

topografiset kartat, ilmastokartat (kuvaa ilmaston piirteitä, kuten sademääriä), talouskartta (luonnonvarat ja teollinen toiminta), geologiset kartat (maaperän ja kallioperän tieto), teemakartat (kuvataan aluetta teeman mukaisesti), kohokartat (käsin tunnusteltava kartta) sekä jo aikaisemmin mainitut merikartat (merenkulkua kuvastava kartta, ei merikortti) ja tiekartat.

Poliittiset kartat on suunniteltu hallinnollisten alueiden eli valtioiden ja osavaltioiden rajat, sekä suurimpien kaupunkien sijainnit. Kartoissa tavallisesti esitetään myös merkittävät vesivarannot, joita eri alueilla sijaitsee.

Fyysiset kartat sisältävät samoja piirteitä kuin poliittiset kartat, mutta niissä on lisäksi huomioitu maaston topografiaa, eli maaston muotoja, kuten vuoria, aavikkoja, jokia ja muita vesistöjä. Fyysinen kartta tuo maaston muodot paremmin saataville kartan avulla sekä siihen on merkitty huomattavien maaston muotojen nimet. Suurin osa tarjolla olevista maailman kartoista sekä karttapalloista on fyysisten ja poliittisten karttojen sekoituksia.

Topografiset kartat esittävät maaston vaihteluita tarkemmalla tasolla kuin fyysiset kartat eli käyttämällä korkeuskäyriä. Topografiset kartat ovat tunnettuja siitä, että ne pyrkivät mahdollisimman tarkkaan esitykseen kuvaamastaan maastonkohdasta ja niissä esitetään sekä luonnolliset yksityiskohdat että ihmisen rakentamat ominaisuudet. Topografisia karttoja käytetään nykyisin monella eri alalla, esimerkiksi maantieteellisessä suunnittelussa tai laaja-alaisessa arkkitehtuurissa, kaivostoiminnassa, rakentamisessa sekä lenkkeilyssä ja suunnistuksessa. Suomessa tuotettavat topografiset kartat ovat Maanmittauslaitoksen tuottamia ja niiden mittakaava on 1:25 000 tai 1:50 000, jonka lisäksi Maanmittauslaitos tarjoaa topografista dataa mittakaavavälillä 1:5 000-1:10 000. Maanmittauslaitoksen tarjoamia maastokarttoja voi katsella osoitteesta http://www.paikkatietoikkuna.fi/web/fi/kartta valitsemalla karttatasoksi maastokartan.

5.2. Koordinaatit

Geograafiset koordinaatit ovat koordinaattijärjestelmä, jonka avulla on mahdollista esittää tietty piste maapallolla. Nämä luvut tavallisesti määrittävät pituusasteen, leveysasteen sekä korkeuden ja ne vaihtelevat asteikon standardin sekä esitystavan mukaisesti. Maanmittauslaitos määrittää karttakoordinaatiston seuraavalla tavalla: "*UTM (Universal Transverse Mercator) jakaa koko maailman kaistoihin (ja ruutuihin) välillä 80° eteläistä ja*

84° pohjoista leveyttä. Yhden kaistan leveys on 6° ja ne on numeroitu 1 - 60:een. Yhden ruudun korkeus on 8° ja leveyspiirien mukaiset kaistat on merkitty kirjaimin C - X". Enemmän asiasta: http://www.maanmittauslaitos.fi/kartat/koordinaatit/tasokoordinaatistot /etrs-tm35fin

Koordinaattien toiminta on helppo ymmärtää, jos on joskus sattunut käyttämään GPS-laitetta (Global Positioning System) tai puhelimen GPS-ominaisuuksia. GPS-laitteen avulla on mahdollista visuaalisesti osoittaa kartalla kyseisen vastaanottimen (ja samalla käyttäjän) sijainti, sillä laite käyttää kartan tavoin samaa koordinaatistoa, jolloin sijoittaminen on mahdollista. Oletetaan käyttäjän olevan Helsingissä, jolloin hänen sijaintinsa koordinaatistolla olisi 60° 10′ 15″ N, 24° 56′ 15″ E tai koneellisesti ymmärrettävässä muodossa 60.170833, 24.9375 (WGS84-koordinaatistolla).

Suomessa on käytössä UTM:n pohjautuva ETRS-TM35FIN-koordinaattijärjestelmä [URL: http://fi.wikipedia.org/wiki/ETRS-TM35FIN] ja Googlen käytössä vastaavasti edelläkin jo kuvattu WGS84. Nämä kuulostavat hyvin kryptisiltä nimiltä, mutta käytännössä ne kertovat vain, miten kartan koordinaattipisteet esitetään. Edellistä esimerkkiä jatkaen Suomen käyttämän koordinaattijärjestelmän mukainen koordinaattipiste Helsingille on 385565, 6672224. Pisteet lasketaan eri järjestelmissä eri menetelmillä, mutta on olemassa työkaluja siihen, että pisteet voidaan muuntaa järjestelmästä toiseen. Tästä enemmän myöhemmin tässä luvussa.

5.3. paikkatietoikkuna.fi

"*Paikkatietoikkuna on julkinen, kaikille avoin ja maksuton verkkosivusto, jonne on koottu paikkatietoa ja asiaa paikkatiedosta.*" - Maanmittauslaitos
[URL: http://www.paikkatietoikkuna.fi/web/fi/mika-paikkatietoikkuna]

Paikkatietoikkuna on palvelu, jonka avulla on mahdollista mallintaa kaikki edellä mainitut karttatyypit (ei navigointikarttoja). Paikkatietoikkuna on analyysityökalu, jonne toimitetaan tietoa jokaiselta julkishallinnon osa-alueelta, jolla paikkatietoa on tallennettu. Näitä ovat erilaiset organisaatiot, kuten esimerkiksi kunnat, kaupungit, liitot, laitokset ja tutkimuskeskukset.

Palvelun tarkoituksena on luoda analyysityökalu, jonka avulla voidaan visuaalisesti esittää useita erilaisia tietoja ja etsiä näiden välisiä yhteyksiä. Hyödyllisin paikkatietoikkunan toiminnallisuus on mahdollisuus rakentaa täysin omia karttapohjia sekä ottaa niitä käyttöön omaan tarkoitukseensa. Tällä tarkoitetaan sitä, että käyttäjä voi luoda karttapohjan, johon hän kerää useita datakohteita ja ottaa tämän käyttöön suoraan omille verkkosivuilleen. Tästä esimerkkinä toimii Vihdin kunta, joka on ottanut paikkatietoikkunan avulla rakennetun karttapalvelun käyttöön omille verkkosivuilleen: http://www.vihti.fi/karttapalvelu.

Paikkatietoikkuna toimii avoimen paikkatiedon avulla, jota se saa yhteistyökumppaneilta, kuten Maanmittauslaitokselta, Ilmatieteen laitokselta, geologian tutkimuskeskukselta ja museovirastolta. Kaikki data tosin ei ole täysin avoimen datan määritelmän mukaista, sillä esimerkiksi Ilmatieteen laitos vaatii, että tiedon käyttäjä suorittaa kirjautumisen. Myös muut yhteistyökumppanit rajoittavat tiedon saamisen määrää sallimalla vain tietyn määrän dataa tai vain tietyn määrän kyselyitä aikavälillä. Täysi lista kaikista tietolähteistä löytyy osoitteesta http://www.paikkatietoikkuna.fi/web/fi/avoin-paikkatieto.

Paikkatietoikkuna on rakennettu käyttämällä taustalla Maanmittauslaitoksen tuottamia karttoja, maastotietokannasta (ja myös muualta) saatuja tietoja sekä Oskari-karttaviitekehystä.

5.4. Oskari (Open Source KArttaIkkuna)

Kuten edellä mainittiin, Paikkatietoikkuna perustaa toimintansa Oskari-karttaviitekehykseen (lisenssi: MIT/EUPL). Tämä tosin ei tarkoita, että Oskari olisi ollut olemassa ensin, vaan Oskari lähti kehittymään omaan suuntaansa sen jälkeen, kun paikkatietoikkuna oli saatu toimimaan, vaikkakin ne toimivat edelleen hyvin läheisessä yhteistyössä. Oskarista lähdettiin rakentamaan täysin omanlaista viitekehystä, joka tarjoaa kehittäjälle sekä käyttöliittymän että palvelualustan (sekä tulevaisuudessa palveluväylän eli *service bus*in) JavaScriptillä kirjoitettuna, jolloin se toimii selaimessa ilman erillisiä asennuksia. Tällä hetkellä paikkatietoikkunan karttaikkuna toimii testausalustana Oskarin uusille toiminnoille ja muulle kehitykselle.

Oskari hyödyntää mahdollisuuksien mukaan tuottajien rajapintoja tiedon hakemista ja esittämistä varten. Tällä hetkellä tuettuja rajapintapalveluja ovat WMS (*Web Map Service*), WMTS (*Web Map Tile Service*) ja WFS (*Web Feature Service*). Itse Oskarin käyttöliittymän toiminta on taattu käyttäen olemassa olevia JavaScript-kirjastoja, kuten jQuery, GeoTools,

OpenLayers ja Jackson. Palvelualusta toimii muusta viitekehyksestä poiketen Javalla, jolloin se on mahdollista saada toimimaan palvelimella ketterämmin kuin JavaScriptin kanssa.

Oskaria esittelevät verkkosivut löytyvät osoitteesta http://oskari.org/ ja kaikki siihen liittyvä lähdekoodi on saatavilla GitHubista osoittesta https://github.com/nls-oskari/oskari (käyttöliittymä) ja https://github.com/nls-oskari/oskari-server (palvelualusta).

Oskari on myös mahdollista ottaa testikäyttöön käyttämällä VirtualBoxia ja Vagrant-ohjelmistoa, jolloin kehittäjän ei ole tarvetta kuin ladata Vagrant-paketti ja asettaa sen kautta Oskari toimintakuntoon, johon ohjeet löytyvät osoitteesta http://oskari.org/documentation/vagrant-guide. Vagrant-paketti sisältää Oskarin täysin käyttövalmiina, jolloin kehittäjän ei tarvitse erikseen ladata tiedostoja tai asentaa muita ohjelmistoja käyttökunnon takaamiseksi. Huomionarvoista on, että tätä pakettia ei ole suotavaa käyttää tuotantokäytössä, se toimii parhaiten testauksessa ja kehityksessä.

5.5. WMS, WMTS ja WFS

WMS on norminmukainen protokolla geoviitattujen karttakuvien tuottamiseen verkon yli, joka on luotu käyttäen karttapalvelinta ja GIS (*Geographic Information System*) tietokantaa. Sen on kehittänyt ja tuottanut OGC (Open Geospatial Consortium) jo vuonna 1999, jolloin ensimmäisiä verkkoon liitettäviä karttoja alettiin kehittää. WMTS on vuorostaan norminmukainen protokolla, jonka avulla tuotetaan esirenderöityjä ja geoviitattuja karttaosia verkon yli. WMTS siis muistuttaa hyvin paljon jo aikaisemmin mainittua WMS:ää, mutta se kehitettiin vuonna 2010, kun WMS ei kyennyt saavuttamaan vaadittuja vasteaikatavoitteita, joita kartoilta vaadittiin. WMS tavallisesti vaatii yhden tai useamman prosessorisekunnin (*CPU second*) vastauksen tuottamiseen, mutta nykyisillä vaatimuksilla, kun puhutaan useista samanaikaisista ajoista eli useista samanaikaisista käyttäjistä, tämä vasteaika ei enää ole riittävä.

WMS ja WMTS tuottavat käytännössä vain ja ainoastaan käyttäjälle visualisoitavat kartat jättäen kaiken toiminnallisuuden pois. Tämä tarkoittaa sitä, että näiden kahden protokollan avulla käyttäjän on mahdollista saada kartta näkyville, mutta hän ei pysty itse vaikuttamaan siihen, mitä kartalla näytetään ja miten. Tämän ominaisuuden tuottamiseen on luotu rajapinta WFS, joka voidaan ymmärtää lähdekoodiksi karttaportaalin (kuten Paikkatietoikkuna) taustalla. WFS:n avulla käyttäjä voi siis muuttaa ja analysoida muiden protokollien tuottamaa

kartta-aineistoa verkon yli esimerkiksi lisäämällä kartalle haluamiaan merkkejä ja alueita sekä korostaa haluamaansa materiaalia. Tästä esimerkkinä toimivat Paikkatietoikkunan työkalut, joita käyttäjälle tarjotaan.

5.6. OpenLayers

Pääasiallisesti tärkein karttoihin liittyvä kirjasto Oskarin kehityksessä on OpenLayers. OpenLayers on avoin *JavaScript*-kirjasto, jonka avulla karttoja on mahdollista käyttää verkkosivujen osana, jolloin on mahdollista esittää mitä tahansa karttaosia ja havainnollistavia merkkejä kartan päällä. Tästä hyvä esimerkki on käytännössä maailmanlaajuisesti tunnettu Google Maps, joka käyttää Oskarin tavoin OpenLayers-rajapintaa tuottaakseen visuaalisia ja interaktiivisia karttoja.

Google Maps ja OpenLayers tarjoavat samankaltaisia rajapintoja kehittäjän käyttöön, tosin Googlen pitäessä kaiken koodinsa suljettuna, OpenLayers on kaikille avoin rajapinta (lisenssi: FreeBSD). OpenLayers siis tarjoaa kehittäjälle mahdollisuuden kehittää ilmaiseksi visuaalisesti toimivia karttoja verkkosivuille JavaScriptin avulla, jolloin erillistä palvelinsuoritusta ei tarvita.

Aikaisemmin tässä luvussa oli puhetta koordinaattien muunnoksista UTM:n ja WGS84:n välillä. OpenLayers tarjoaa tähän muunnokseen metodin, jonka avulla muuntaminen on mahdollista. Muuntaminen on tarpeellista aina silloin, kun tarjottava karttamateriaali ei täsmää koordinaattijärjestelmältään annettuihin koordinaatteihin. Myös tämän tarvittavan muunnoksen tekeminen on haastavaa, sillä jokaisella koordinaattijärjestelmällä on standardoitu EPSG-koodi, joka on Suomen järjestelmälle ESPG 3067 (http://spatialreference.org/ref/epsg/etrs89-etrs-tm35fin/). Edellä mainittu Googlen käyttämä WSG84 on taas standardoitu ESPG 4326:ksi (http://spatialreference.org/ref/epsg/4326/). Käydään muuntamista hieman läpi esimerkin avulla.

Esimerkki 5.1. OpenLayers ja koordinaattien muunnos

```
// ETRS-TM35FIN -> WSG84
var proj4326 = new OpenLayers.Projection("EPSG:4326");
var proj3067 = new OpenLayers.Projection("EPSG:3067");

var point = new OpenLayers.LonLat(385565, 6672224);
var newpoint = point.transform(proj3067, proj4326);
```

Esimerkin 4.1. avulla on mahdollista muuntaa Suomessa käytettävät koordinaattipisteet maailmanlaajuista formaattia käyttävään muotoon. Esimerkissä muunnetaan piste 385565, 6672224 pisteeksi 60.170833, 24.9375. Tällainen muunnos on tarpeellinen siis aina, kun annettu aineisto ei täsmää annetun kartan kanssa tai jos kartan ja aineiston käyttötarkoitus sitä vaatii.

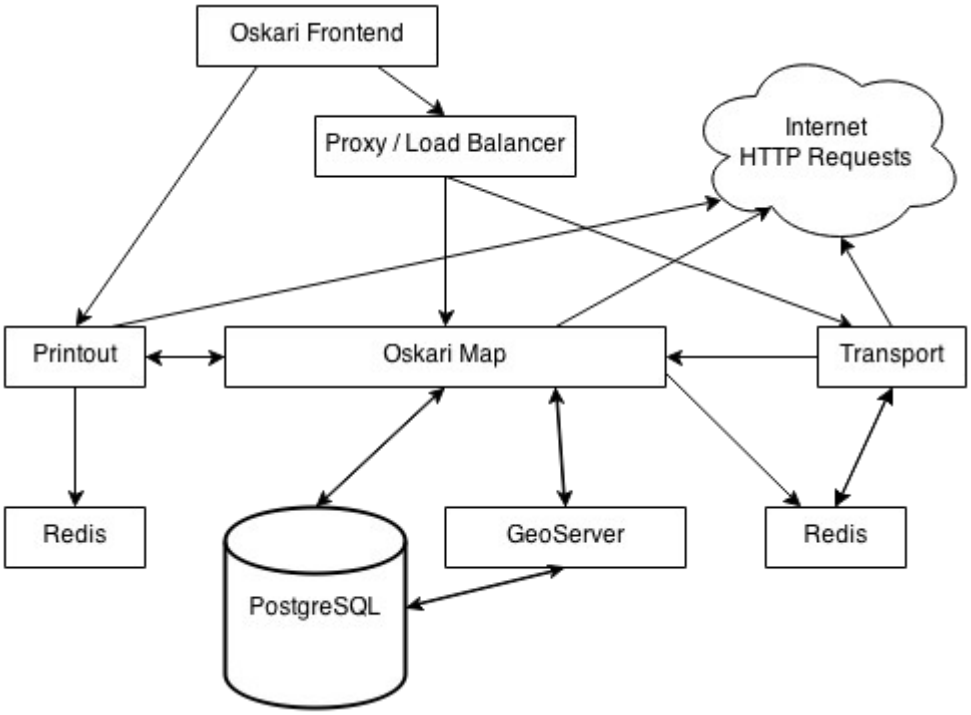
5.7. Oskarin tekninen toteutus

Tässä oppaassa ei käydä Oskarin teknistä toteutusta kooditasolla sen tarkemmin läpi, koska tarkat kuvaukset ovat löydettävissä Oskarin dokumentaatiosta. Kuvat, joissa kuvataan järjestelmän arkkitehtuuria löytyvät myös Oskarin dokumentaatiosta ja niitä lainataan tässä oppaassa selitysten kera.

Aluksi käydään läpi hieman koko Oskarin arkkitehtuuria, joka sisältää sekä käyttöliittymän että palvelualustan. Kaikki data, joka on Oskaria varten varastoitu, sijaitsee PostgreSQL-tietokannassa (http://www.postgresql.org/), johon kaikki tarvittavat kyselyt suoritetaan. Tietokantaa käyttävät osat ovat Oskarin karttakomponentti sekä GeoServer, jotka on luotu käyttämällä Apache Tomcatia (http://tomcat.apache.org/) ja Jettyä (http://www.eclipse.org/jetty/), sekä karttakomponentissa lisäksi Liferaytä (http://www.liferay.com/).

Tiedon siirtoon (*Transport*) ja tulostukseen (*Printout*) käytetään myös Jettyä, joka kommunikoi Redisin (http://redis.io/) eli tietorakennepalvelimen kanssa, jonne suoritukseen tarvittava tieto on tallennettu. Proxyyn ja kuormantasaukseen, joilla parannetaan käyttöliittymän responsiivisuutta ja käyttömukavuutta, käyteään työkaluja HAProxy (http://www.haproxy.org/), Apache (http://www.apache.org/), Nginx (http://nginx.org/) ja F5 load balancer (https://f5.com/glossary/load-balancer).

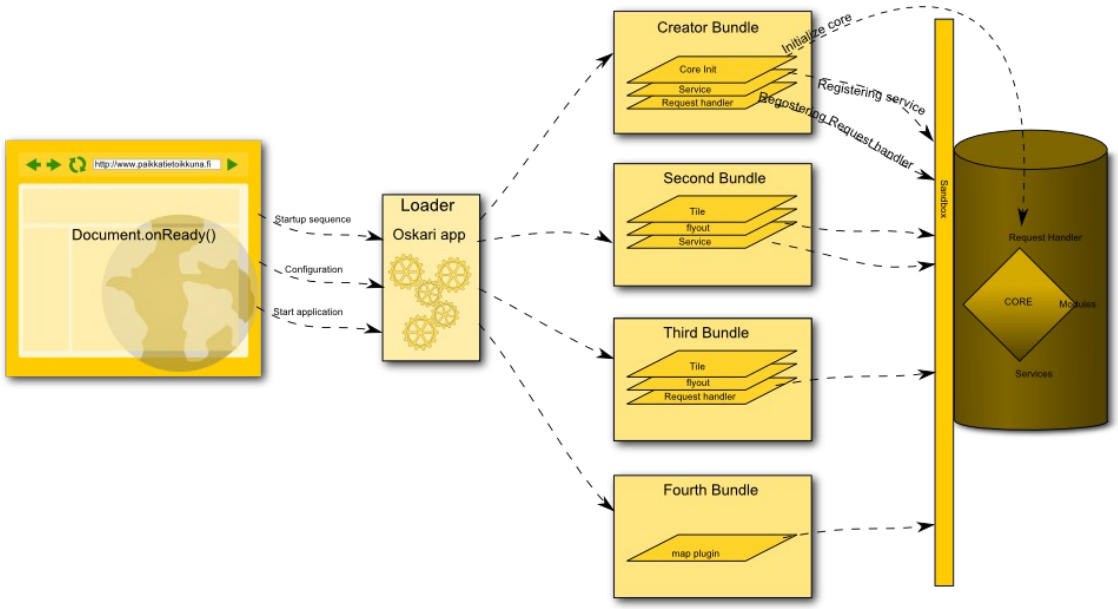
Oskarin käyttöliittymä ei tarvitse käytettäväkseen muita työkaluja kuin tuetun verkkoselaimen, joita ovat Firefox, Chrome, IE ja Safari.



Kuva 5.1. Oskarin arkkitehtuuri (http://oskari.org/documentation/architecture_basics)

Seuraavaksi käydään läpi hieman sitä, miten Oskarin käyttöliittymä toimii. Käyttöliittymän lataus käynnistyy siitä, kun käyttäjä lataa verkkosivun Oskaria käyttävän verkkosivun. Tällöin selain lähettää viitekehykselle tiedon siitä, että verkkosivun lataus on valmis. Tällöin käynnistys annetaan Oskarin Loader-komponentille, joka huolehtii käynnistyksen sujumisesta ja konfiguroinnista JSONin avulla. Lopuksi Loader-komponentti kutsuu startApplication()-metodia, joka käynnistää lopullisen käynnistyksen prosessoinnin.

Kun Oskari käynnistyy metodikutsun jälkeen, ladataan siihen kaikki bundlet eli komponentit, joita ohjelmakoodissa on pyydetty lataamaan, joista yksi bundle on ns. “luoja-bundle”, joka alustaa Oskarin ytimen (*Oskari core*). Kun ydin on alustettu, kaikki palvelut ja pyyntöjen käsittelijät (*request handler*) rekisteröidään ytimeen mistä tahansa bundlesta. Viittaukset karttamoduuliin voidaan hakea ytimestä ja mitkä tahansa liitännäiset (*plugin*) voidaan liittää tähän moduuliin missä tahansa bundlessa. Tämän jälkeen Oskari on käynnistynyt ja kartta ja sen komponentit ovat verkkoselaimessa käyttäjän käytettävissä.



Kuva 5.2. Oskarin käyttöliittymän arkkitehtuuri
(http://oskari.org/documentation/development/architecture)

Edellä mainittiin useaan otteeseen, että Oskari rakentuu dynaamisesti erilaisista bundleista. Nämä bundlet ovat kokoelma Oskarin luokista, jotka muodostavat toimivan komponentin ja komponentti tarjoaa toiminnallisuuksia ohjelmalle. Bundle voi rakentua niin, että se tarjoaa useita toiminnallisuuksia modulaarisesti ja tarjoaa erilaisille tarpeille samankaltaista toiminnallisuutta, eli bundle voi tarjota useita näkymiä samalle toiminnallisuudelle. Esimerkkinä annettakoon karttaikkunan hakutoiminto, joka voi toimia sekä tekstipohjaisesti erillisellä kentällä tai integroituna kartan käyttöliittymään, käyttäen samaa bundlea. Bundlen rakenteeseen päästään myöhemmin arkkitehtuurikuvauksen jälkeen liitteessä 1.

Oskarin palvelualusta koostuu kolmesta kerroksesta: palvelu-, hallinta- ja käyttöliittymäkerroksesta. Tiedon siirto ja tulostus käyttävät osaltaan kaikkia kolmea kerrosta, joissa siirto keskittyy lähinnä WFS-toiminnallisuuksien rakentamiseen ja tulostus tuottaa materiaalin tulostuksen PNG/PDF-muotoon ja esikatselumahdollisuudet.

Palvelukerros pitää sisällään useita eri toiminnallisuuksia, jotka on esitetty kuvassa 5.3. Käydään kerroksen sisältöä läpi moduuli kerrallaan:

- Palvelukartta (*service map*)

    - Sisältää suuren osan palvelun toiminnallisuudesta, joka tarjoaa Oskarin toiminnallisuuksia
- Jaetut testausresurssit (*shared test resources*)
    - Sisältää tarpeellisia aputoimintoja kaikkien kerrostem testausta varten
- Palveluoikeudet (*service permission*)
    - Geneerinen käyttöoikeuksien tarkistus käyttäjäkohtaisesti
- Palveluhaku (*service search*)
    - Geneerinen hakutoiminto, jota on mahdollista laajentaa lisäämällä ja rekisteröimällä uusia hakukanavia
- Palveluhallinta (*service control*)
    - Rakentaa pohjan hallintakerroksen toiminnalle määrittämällä rakenteet ja rajapinat
- Palvelun jalusta (*service-base*)
    - Sisältää koko palvelualustan yhteisiä apuluokkia

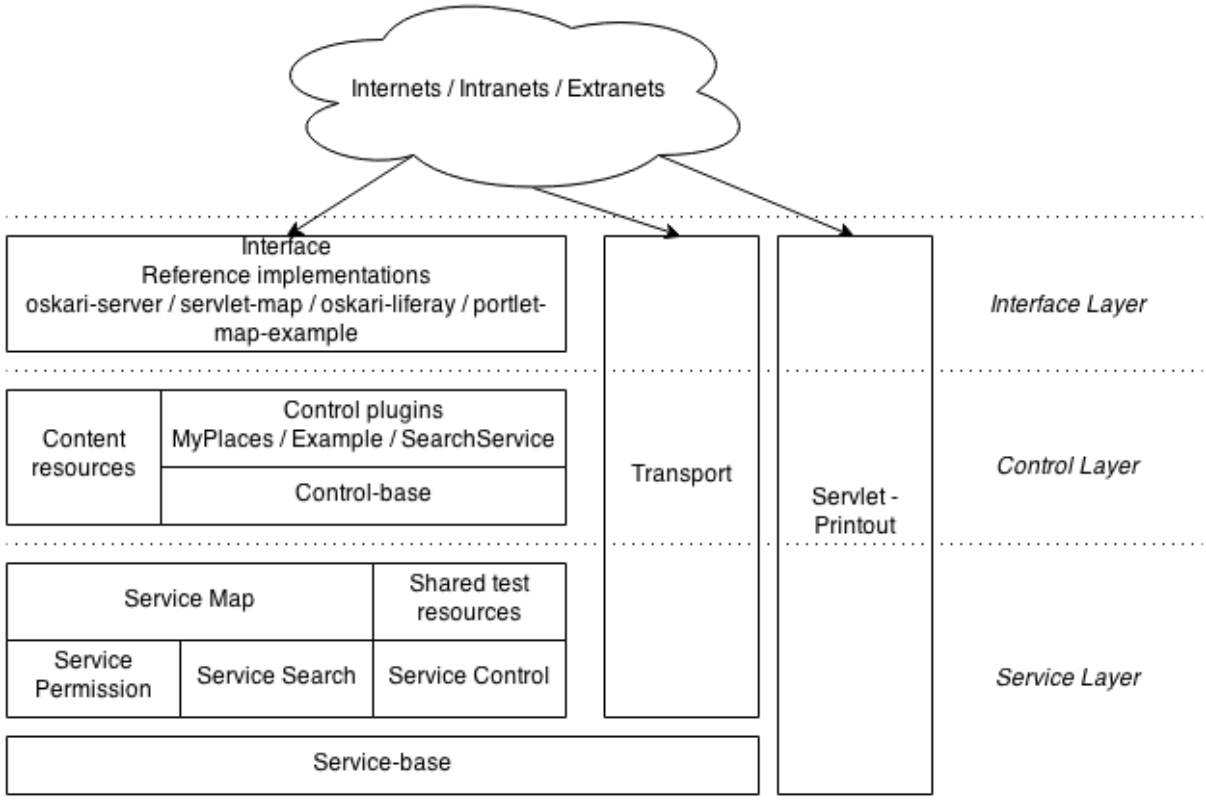
Hallintakerros rakentuu puhtaasti palvelukerroksen päälle ja välittää omaa toimintaansa alemmalle kerrokselle.

- Hallintajalusta (*control-base*)
    - Muodostaa pohjan kaikille hallintamoduuleille ja sisältää suurimman osan kaikista *AJAX*-käsittelijöistä Oskarin käyttöliittymän tarpeisiin
- Hallintaliitännäiset (*control plugins*)
    - Sisältää esimerkkejä ja toimintaohjeita implementaatiota varten
- Sisältöresurssit (*content resources*)
    - Työkalut, mallit ja skriptit tietokantojen rakentamiseen ja päivittämiseen

Hallintakerroksen pääasialliset toiminnallisuudet ovat käyttöliittymän AJAX-komentojen käsittely, parsia käyttäjän syötteitä palveluiden parametreiksi, kutsua palveluita suorittamaan tarvittavat toiminnallisuudet sekä muotoilla palveluiden palauttama informaatio luettavaan muotoon.

Käyttöliittymäkerros on käytännössä vain HTTP-rajapinta, joka on rakennettu hallintakerroksen päälle. Rajapinta sisältää lähdetoteutuksen HTTP Servletille (*oskari-server/servlet-map*), Webappille (*oskari-server/webapp-map*) ja portletille (*oskari-liferay/portlet-map-example*). Käyttöliittymäkerros on vastuussa käyttäjien

sessionhallinnasta, hallintakerrokselle ymmärrettävän pyynnön luomisesta saapuvasta pyynnöstä ja pyynnön ohjaaminen hallintakerrokselle.



Kuva 5.3. Oskarin palvelualustan arkkitehtuuri (http://oskari.org/documentation/architecture/components)

Karttoja on mahdollista esittää verkkoselaimessa, kuten tässä luvussa on esitetty. Kartat kuitenkin vaativat aina, jonkinlaisen graafisen käyttöliittymän, sillä helposti käytettäviä karttoja on käytännössä mahdotonta esittää tekstipohjaisessa käyttöliittymässä. Tämän vuoksi tarvitaan myös karttoja esittävä graafinen käyttöliittymä, jonka perusperiaatteita käydään läpi seuraavassa luvussa. Seuraavan luvun esimerkit eivät puhtaasti käsittele enää karttoja vaan esimerkit ovat pieniä, arkielämää helpottavia ohjelmia.

6. Graafinen käyttöliittymä

"*A program interface that takes advantage of the computer's graphics capabilities to make the program easier to use. Well-designed graphical user interfaces can free the user from learning complex command languages. On the other hand, many users find that they work more effectively with a command-driven interface, especially if they already know the command language.*" - Webopedia
[URL: http://www.webopedia.com/TERM/G/Graphical_User_Interface_GUI.html]

Graafinen käyttöliittymä mahdollisti tietokoneiden yleistymisen tarjoamalla helposti käytettäviä, *point-and-click*-pohjaisia sovelluksia. Graafisten käyttöliittymien esiinmarssin aloittivat Microsoft ja Apple jo niinkin aikaisin kuin 1980-luvulla ja siitä on tultu pitkä matka eteepäin kehityksen tietä. Nykyään valtaosa käytetyistä tietoteknisistä sovelluksista toimii graafisen käyttöliittymän avulla, sillä käytännössä kaikki web-sovellukset ovat graafisia. Tällä viitataan myös suureen osaan mobiilisovelluksista ja osin jopa työpöytäsovelluksiinkin, sillä osa niistä toimii suoraan selaimessa ilman erikseen asennettavia ohjelmia. Tämä on ollut viimeisin suuri mullistus käyttöliittymien suunnittelussa, missä erillisiä painikkeita ja alustoja ei ole tarvinnut erikseen ohjelmoida, ainoastaan käyttöliittymän komponenttien sijoittelu on tarvinnut suunnitella.

Web-sovelluksista huolimatta työpöytäsovellukset, jotka eivät käytä Web-teknologioita vaativat edelleen erillisen käyttöliittymän rakentamisen. Tähän on kehitetty ratkaisuja esimerkiksi Linuxin ja Macin puolella, missä komentoriviltä ajettavat ohjelmat ovat yleisempiä ja välillä ne jopa toimivat paremmin ja tehokkaammin kuin graafiset vastineensa. Varsinkin Windowsin puolella graafisen käyttöliittymän tarve korostuu huomattavasti, sillä Windows on pysynyt uskollisena itselleen siinä, että käyttäjälle tarjotaan varmasti toimiva graafinen käyttöliittymä niin, ettei komentoriville ole mitään tarvetta. Toki tämä Windowsin puolelta löytyy, mutta sen tehokkuus ja käytettävyys on kyseenalainen.

Graafisia sovelluksia on työläämpää ja monimutkaisempaa lähteä luomaan työpöytäkäyttöön kuin Web-käyttöön, mutta tarve on huomattu ja siihen on vastattu. Lähes jokainen ohjelmointikieli on saanut tuekseen graafisia kirjastoja, joiden avulla jopa yhdellä rivillä koodia on mahdollista luoda ruudulle jo ohjelmaikkuna. Tätä on laajennettu myös niin, että jokainen vaadittava objekti, oli se sitten valikko, tekstikenttä, painike, valintalaatikko etc. on helppo asettaa mukaan käyttöliittymään ja sen kustomointi on mahdollistettu. Tämän tekivät

mahdolliseksi graafiset käyttöliittymäkirjastot, joita ovat esimerkiksi Javalle Swing ja JavaFX tai laajemmin useille ohjelmointikielille palveluja tarjoavat Qt ja GTK+. Nämä kirjastot tarjoavat työkalut helppoon käyttöliittymän rakentamiseen, jolloin on mahdollista luoda vain pohja eli sovellusikkuna ja sen sisään hiiren avulla siirtää tarvittavia palikoita. Tällaisia työkaluja ovat Swingin WindowBuilder, JavaFX:n Scene Builder, Qt:n Qt Designer ja GTK+:n Glade.

WindowBuilder: http://www.eclipse.org/windowbuilder/
Scene Builder:
http://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-info-2157684.html
Qt Designer: http://qt-project.org/doc/qt-4.8/designer-manual.html
Glade: https://glade.gnome.org/

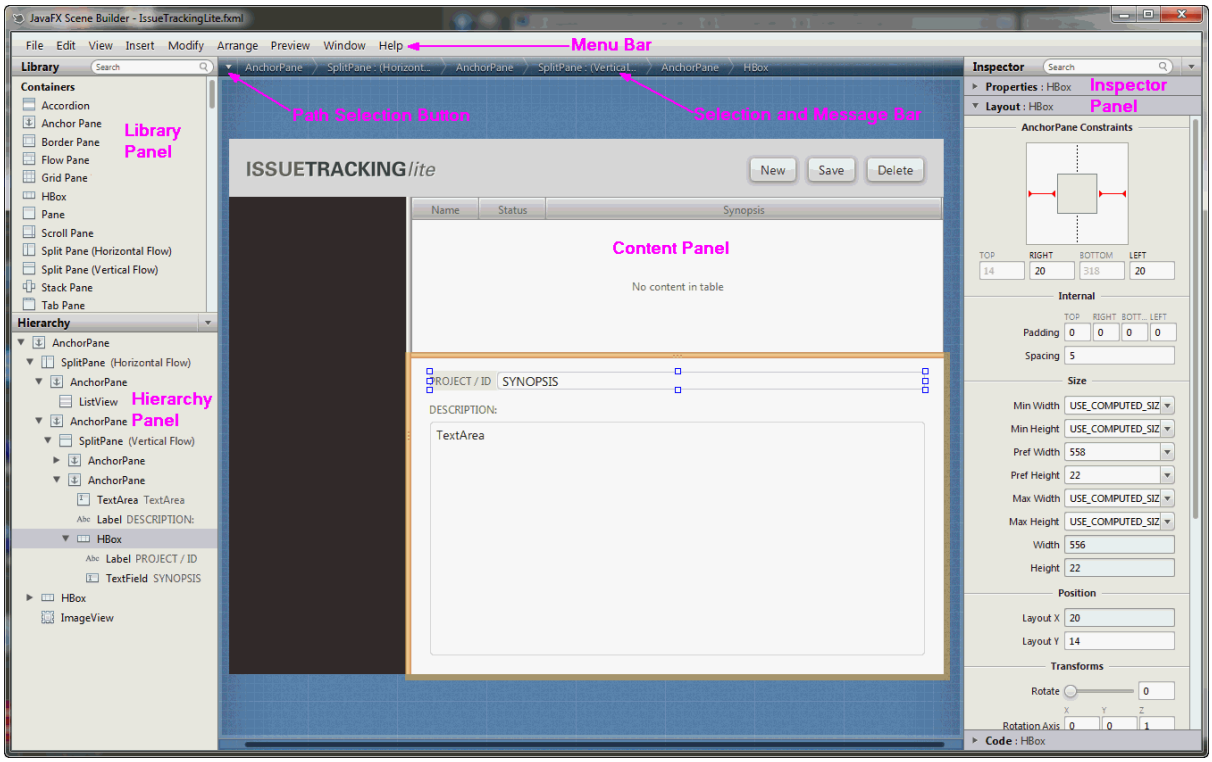
Tämä opas käsittelee pääasiassa JavaFX-kirjastoa ja sen tuomaa Scene Builder-työkalua, mutta graafiset kirjastot muistuttavat hyvin paljon toisiaan ja osaamalla yhden kirjaston toiminnan on helppo oppia käyttämään toistakin kirjastoa.

6.1. JavaFX ja Scene Builder

"*JavaFX is the next step in the evolution of Java as a rich client platform. It is designed to provide a lightweight, hardware-accelerated Java UI platform for enterprise business applications. With JavaFX, developers can preserve existing investments by reusing Java libraries in their applications.*" - Oracle
[URL: http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html]

Scene Builder perustuu hyvin paljon JavaFX:n tarjoamaan FXML-rakenteeseen, joka on XML-pohjainen merkintäkieli. Se mahdollistaa komponenttien asettelun koko ikkunaan sekä päällekkäin niin, että yksi komponentti (esimerkiksi Layout) pitää sisällään useita erilaisia komponentteja. Scene Builder on siis graafinen työkalu, joka muokkaa FXML-dokumenttia, ja tästä dokumentista ohjelma hakee tarvitsemansa tiedon käyttöliittymää varten. Huomaathan myös sen, että JavaFX ja Scene Builder vaativat asennetuksi NetBeans 7.4 tai uudemman tai muun tuetun IDE:n. Tämän lisäksi Scene Builder on vielä asennettava erikseen

Oraclen sivuilta, jonka jälkeen sen käyttö onnistuu suoraan IDE:stä tavalla, joka on kuvattu tässä luvussa.



Kuva 6.1. JavaFX pääikkuna (http://docs.oracle.com/javafx/scenebuilder/1/get_started/prepare-for-tutorial.htm#CEGJBHHA)

Lähdetään esimerkkinä luomaan yksinkertaista graafista ohjelmaa, johon lisäämme erilaisia komponentteja tarpeen mukaan. Esimerkissä käydään myös läpi, miten komponentteja voi luoda ja miten niitä voidaan muuttaa tai käyttää pelkällä koodilla. Näemme siis, miten painikkeet saavat jotakin aikaan ja luovat jopa uutta sisältöä.

Esimerkki luodaan käyttäen NetBeans 8:aa, mutta se käytännössä toimii myös muilla tuetuilla IDE:illä. Jotta voisimme aloittaa kehittämisen, luodaan NetBeansissa uusi projekti. Valitaan siis JavaFX-kategorian alta JavaFX FXML Application. Tämä avaa meille kolme kooditiedostoa, pääohjelman, FXML-dokumentin sekä FXMLDocumentControllerin. Aloitetaan pääohjelmasta.

Esimerkki 6.1. JavaFX esimerkkiohjelma

```
import javafx.application.Application;
```

```
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class JavaFXSample extends Application {

    @Override
    public void start(Stage stage) throws Exception {
        Parent root =
 FXMLLoader.load(getClass().getResource("FXMLDocument.fxml"));

        Scene scene = new Scene(root);

        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Ohjelma sisältää main-funktion, jossa ei suuremmin suoritusta ole. *launch()*-komennon avulla ohjelma käynnistyy ja ottaa tarvittaessa komentoriviparametreja suoritukseen mukaan. *start()*-metodi pitää sisällään tarvittavat komponentit ja niiden väliset yhteydet. Metodin sisälle voidaan lisätä komponentteja kooditasolla, jos halutaan luoda staattisia komponentteja, mutta Scene Builderin ansiosta tätä ei tarvitse tehdä. *stage.show()*-komento lopuksi näyttää koko komeuden käyttäjän ruudulla.

root-muuttuja on nimensä mukaisesti ohjelman juuri, mistä kaikki suoritus lähtee ja minkä päälle koko ohjelma rakentuu ja kuten huomataan, se lataa itseensä FXMLDocument-tiedoston. Katsellaan seuraavaksi sitä tiedostoa.

Esimerkki 6.2. FXMLDocument

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<AnchorPane id="AnchorPane" prefHeight="200"
```

```
        prefWidth="320" xmlns:fx="http://javafx.com/fxml/1"
        fx:controller="javafxsample.FXMLDocumentController">
    <children>
      <Button layoutX="126" layoutY="90" text="Click Me!"
           onAction="#handleButtonAction" fx:id="button" />
      <Label layoutX="126" layoutY="120" minHeight="16"
          minWidth="69" fx:id="label" />
    </children>
</AnchorPane>
```

FXMLDocument päivittyy sitä mukaa, mitä Scene Builderissa rakennetaan ja tallennetaan, eli tallentamaton muutos ei vielä näy tiedostossa. Tiedoston sisältö näyttää hyvin samalta kuin aiemman luvun XML-esimerkit, mutta nyt kentät eivät sisällä elementtejä vaan kaikki data on attribuutteina. Lähdetään liikkeelle pohjalta, eli ohjelma rakentuu AnchorPanen päälle, jolla on oma id, jonka avulla siihen päästään käsiksi. Lisäksi sille on määritetty leveys ja korkeus, jotka ovat ikkunan mitat sekä tiedostoa kontrolloiva luokka FXMLDocumentController eli jos halutaan päästä käsiksi ohjelman juureen, se on tehtävä tiedostossa FXMLDocumentController ja silloin on käytettävä kentälle tarkoitettua id:tä.

AnchorPane-kentän sisällä näemme kentän *children*, joka korostaa kaikkien ohjelman komponenttien keräämistä juuren sisälle. Tässä ohjelmassa komponentteja on kaksi, painike (Button) sekä tekstikenttä (Label). Painikkeen id nähdään viimeisenä attribuuttina eli *"button"* ja sen sisältämä teksti on "Click Me!". Lisäksi painike sisältää kentän *onAction*, jolla viitataan siihen, mitä ohjelma suorittaa silloin kun painiketta painetaan. Tekstikenttä on nimeltään *"label"* ja sille on määritetty myös sijainti sekä koko. Dokumentti on siis hyvin yksinkertainen ja pitää sisällään kaiken tarvittavan informaation ohjelmasta.

Esimerkki 6.3. FXMLDocumentController

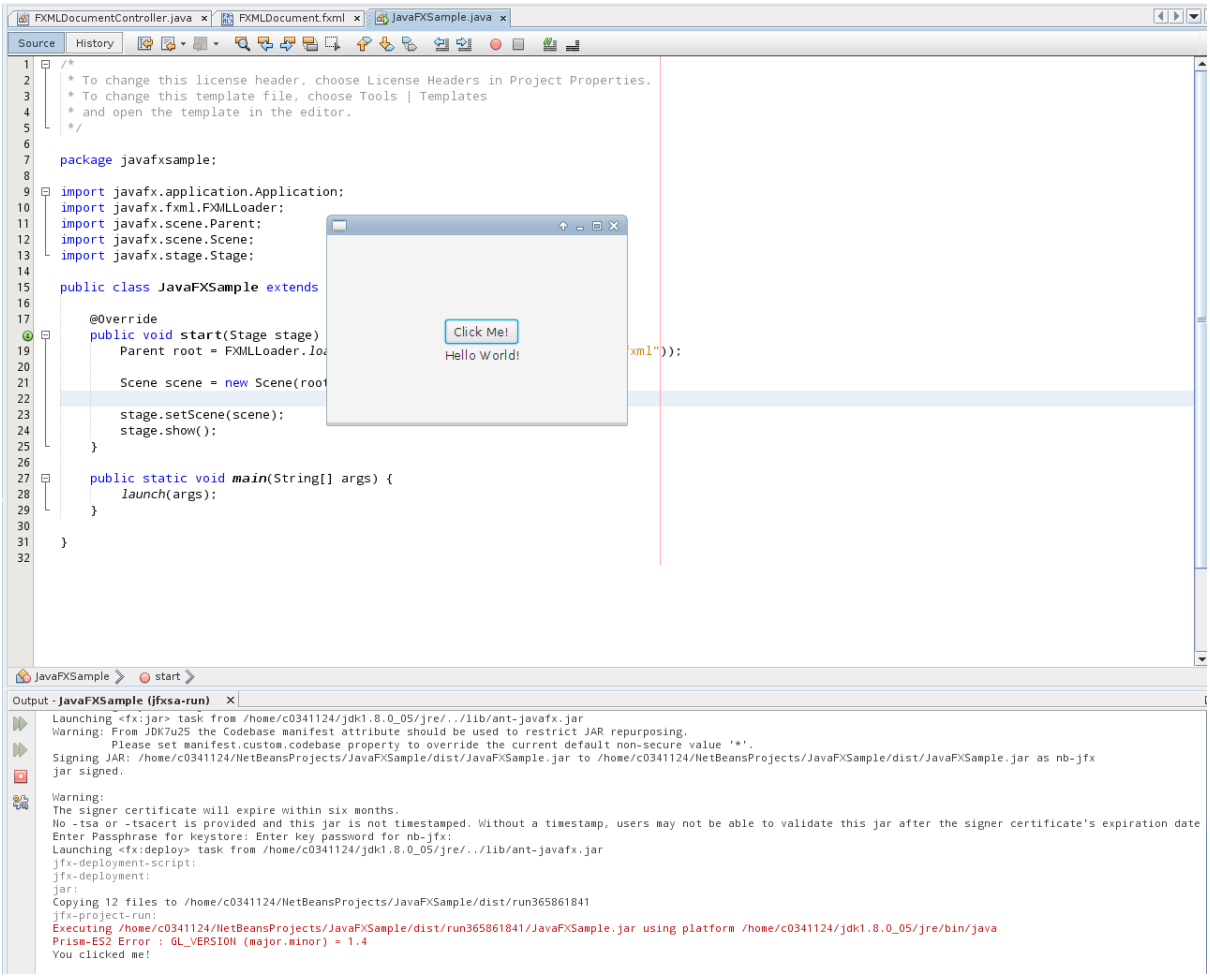
```
import java.net.URL;
import java.util.ResourceBundle;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Label;

public class FXMLDocumentController implements Initializable {

    @FXML
    private Label label;

    @FXML
```

```
    private void handleButtonAction(ActionEvent event) {
        System.out.println("You clicked me!");
        label.setText("Hello World!");
    }

    @Override
    public void initialize(URL url, ResourceBundle rb) {
    }

}
```

FXMLDocumetnController pitää sisällään dynaamisen hallinnoinnin liittyen FXMLDocumentin toimintaan. Luokan alussa nähdään yksityiseksi määritetty Label, jonka nimi on sama kuin dokumentissa olevan kentän id. Näin siis päästään käsiksi staattisesti asetettuihin komponentteihin, jolloin tekstikenttää voidaan muokata aina tarpeen mukaan kun tietty toimenpide tapahtuu.

Seuraavana luokassa on yksityiseksi määritelty metodi *handleButtonAction()*, jonka nimi on sama kuin dokumentissa määritelty *button*-painikkeen *onAction*-kenttä. Tässä sidotaan painikkeen painallusoperaatio dynaamisesti niin, että jokaisella painalluksella tapahtuu tietty toiminto. Esimerkin tapauksessa aina painiketta painaessa ohjelma tulostaa konsoliin rivin "*You clicked me!*" ja asettaa tekstikenttään tekstin "*Hello World!*".

Lisäksi luokalla on metodi *initialize()*, joka on alustusmetodi eli se suoritetaan aina silloin, kun ohjelma käynnistyy. Jos komponentille halutaan alustaa alkuarvoja, se voidaan tehdä tässä metodissa. Huomaa myös merkintä @FXML, sillä se on aina lisättävä muuttujan eteen, jos sitä halutaan käyttää dynaamisesti ohjelmassa. Metodin edessä tämän tulee myös olla, jos metodia kutsutaan *onAction*-kentän avulla. Katsotaan, millainen ohjelmaikkuna tällä esimerkkiohjelmalla saatiin aikaiseksi.



Kuva 6.2. Hello world-ohjelma ajettuna

Kuvassa 6.2. ohjelmassa on klikattu painiketta, jolloin painikkeen alle ilmestyi teksti “Hello World!” ja NetBeans-konsoliin teksti “You clicked me!”, aivan kuin koodista ymmärrettiin.

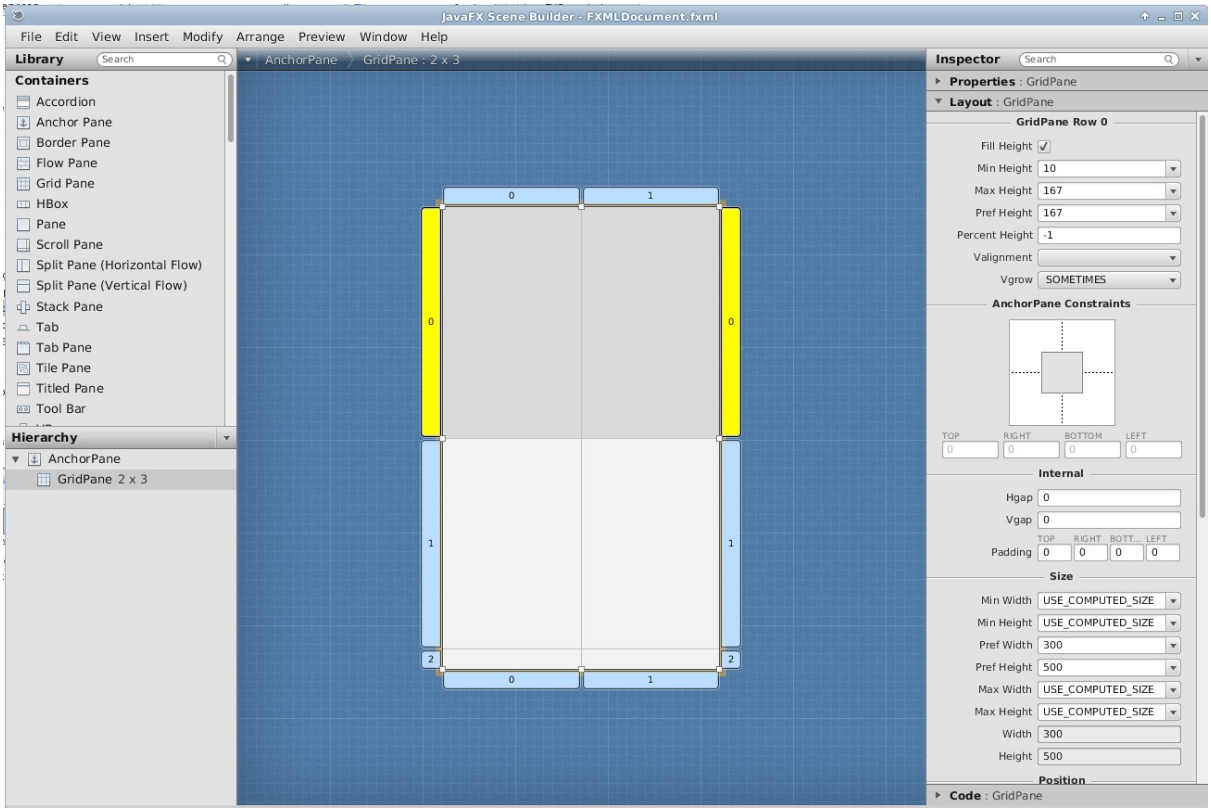
6.2. Junalipun tilauslomake

Lähdetään seuraavaksi luomaan esimerkin avulla junalipun varausjärjestelmää. Pohditaan aluksi, mitä tuo järjestelmä voisi tarvita käyttöliittymältä. Painikkeita (Button) olisi ainakin hyvä olla perumiseen ja lähettämiseen, tekstitenttiä (Label) selventämään valikkoja ja komponentteja, valintanappeja (Radio Button) määrittämään lipun tyyppiä ja tiputusvalikkoja (Combo Box), joilla määrätään lähtö- ja kohdekaupungit. Nämä kaikki komponentit olisi myös mukava koneellisesti organisoida järkevään järjestykseen, joten käytämme erilaisia keinoja siirtää komponentit paikalleen. Lisäksi lipun ostaja voi lisätä kommentteja lipun

ostoon liittyen, joten on lisättävä myös tekstikenttä (Text Area). Listataan vielä tarvittavat komponentit:

- Button
- Label
- Radio Button
- Combo Box
- Text Area
- Layout

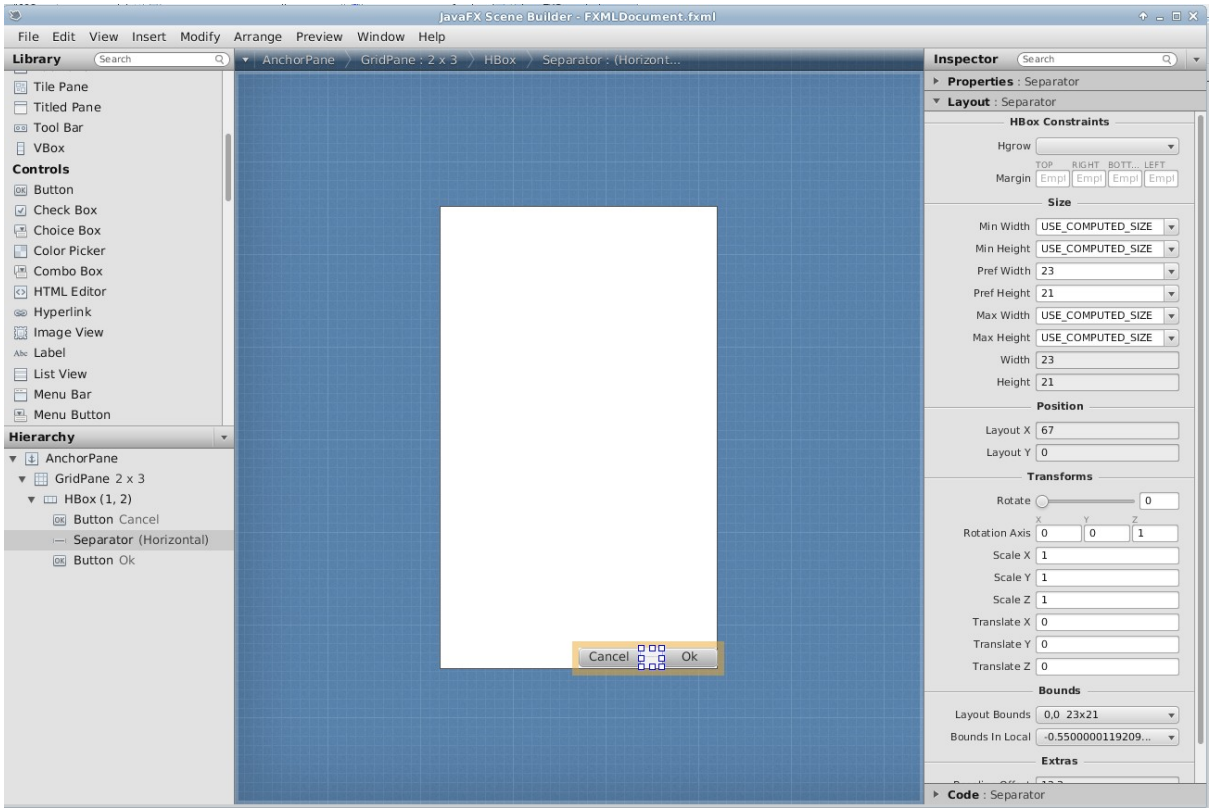
Lähdetään aluksi kasaamaan käyttöliittymää pohjalta ylöspäin, eli aluksi lisätään alustalle pohja eli *layout*. Käytetään pohjana Grid Pane-komponenttia, joka mahdollistaa useiden komponenttiosioiden rakentamisen erillisiksi kokonaisuuksiksi.



Kuva 6.3. Grid Panen lisääminen

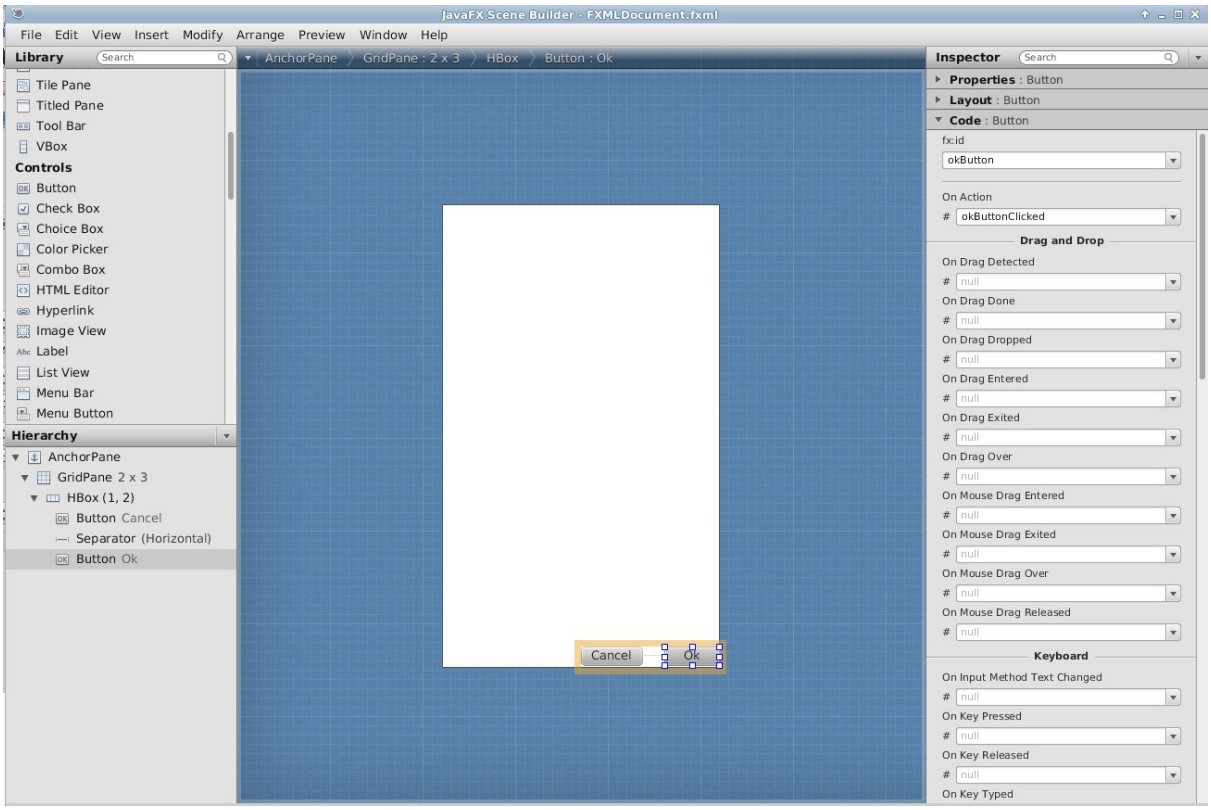
Luodaan pohjan päälle ensimmäiset näkyvät komponentit, eli hyväksymis- ja hylkäyspainikkeet. Niiden kanssa käytetään alustana HBox-komponenttia (tulee sanoista Horizontal Box), jonka avulla on mahdollista lisätä useita komponentteja rinnakkain. Jos kuitenkin komponenttiin lisätään vain painikkeita, liimautuvat ne toisiinsa kiinni ja niitä ei

voi asettaa käsin vetämällä laatikkoon, vaan joudumme käyttämään Separator-komponenttia, jolla painikkeet voidaan erottaa. Monimutkaisen kuuloista, joten katsotaan asiaa esimerkistä.



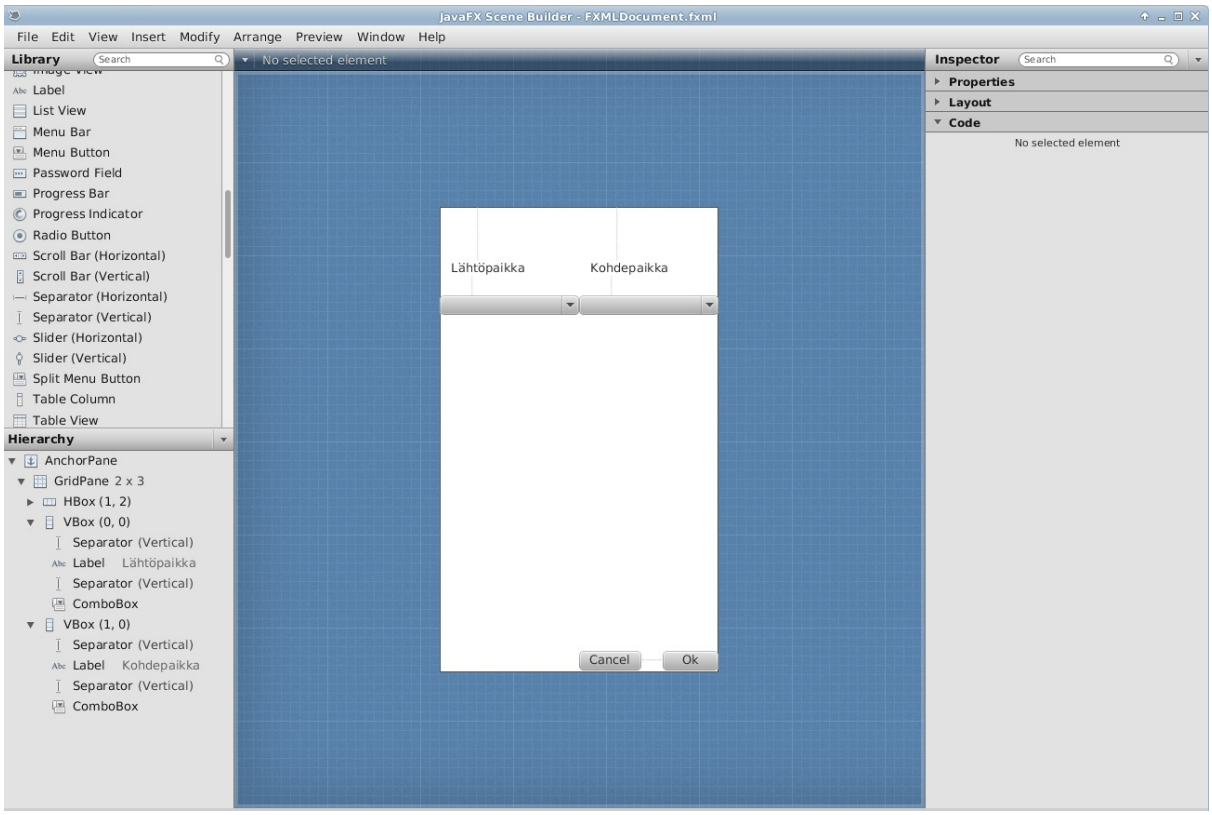
Kuva 6.4. HBox ja painikkeet

Kuva 6.4. ei täysin havainnollista sitä, miten komponentit on asetettu alustan päälle, mutta Scene Builderin vasemmassa alareunassa oleva puunäkymä (Hierarchy) antaa kuvan siitä, mikä komponentti sisältää muita. Nähdään siis, että Grid Panen sisällä on HBox sijainnissa 1,2, eli alaoikealla, origon (0,0) ollessa ylävasemmalla. HBox taas sisältää painikkeet ja niiden välissä olevan Separator-komponentin. Tässä vaiheessa lisätään painikkeille id:t, jotta niihin voidaan liittää toiminnallisuutta. Käytetään painikkeiden nimeämiseen järkeviä nimiä, eli *okButton* ja *cancelButton*. Lisäksi lisätään molemmille painikkeille *onAction*-metodit, joita kutsutaan painiketta painettaessa. Tämä voidaan tehdä oikeassa reunassa näkyvän Code-välilehden kautta, joka näkyy ohjelman oikeassa reunassa kuvassa 6.5.. Käytetään myös metodeille järkeviä nimiä, jotka on helppo yhdistää painikkeisiin, eli *okButtonClicked()* ja *cancelButtonClicked()*.



Kuva 6.5. Toiminnallisuuden lisääminen painikkeille

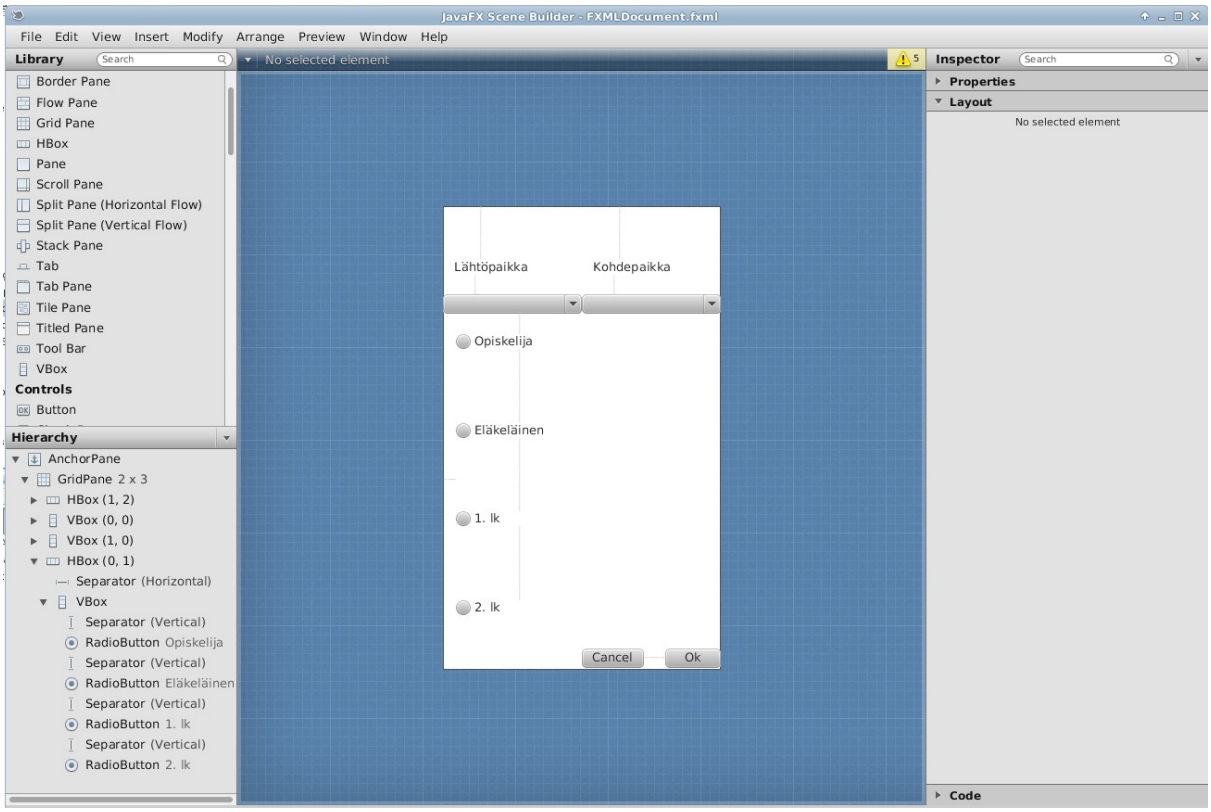
Seuraavaksi rakennetaan pudotusvalikot käyttöliittymän yläreunaan, josta voimme valita lähtöpaikan ja kohteen. Tämä alustus vaatii myös ohjelman koodiin lisäyksiä, joista puhumme kun käyttöliittymä on valmis. Etsitään vasemmalla olevasta valikosta Combo Box-komponentit, ja lisätään ne Grid Panen ylimpiin lokeroihin kuvassa 6.6.. Lisäksi on suotavaa lisätä otsikkokentät (Label) molemmille valikoille, jotta käyttäjä tietää mistä valitaan mitäkin. Nyt voimme käyttää komponenttien järjestämiseen VBox-komponenttia (Vertical Box), sillä haluamme komponentit asettumaan päällekkäin. Lisäksi voimme käyttää pystysuuntaista Separator-komponenttia, jotta saamme luontevan näköisen välin komponenttien välille.



Kuva 6.6. Lähtö- ja kohdekomponentit

Kuvassa 6.6. nähdään, kuinka Grip Panen osiin 0,0 (vasen ylä) ja 1,0 (oikea ylä) lisättiin lähes identtiset komponentit. Pohjalle asetettiin VBox, jotta komponentit saadaan näkymään järkevästi päällekkäin. Tähän lisättiin tekstikenttä ja tiputusvalikko separaattoreiden avulla eroteltuina. Taas vasemmassa alakulmassa on Hierarchy-laatikko, jossa puunäkymä esittää käytetyt komponentit. Huomaathan myös sen, että tiputusvalikkokomponentit on nimetty nyt nimillä *fromBox* ja *toBox*.

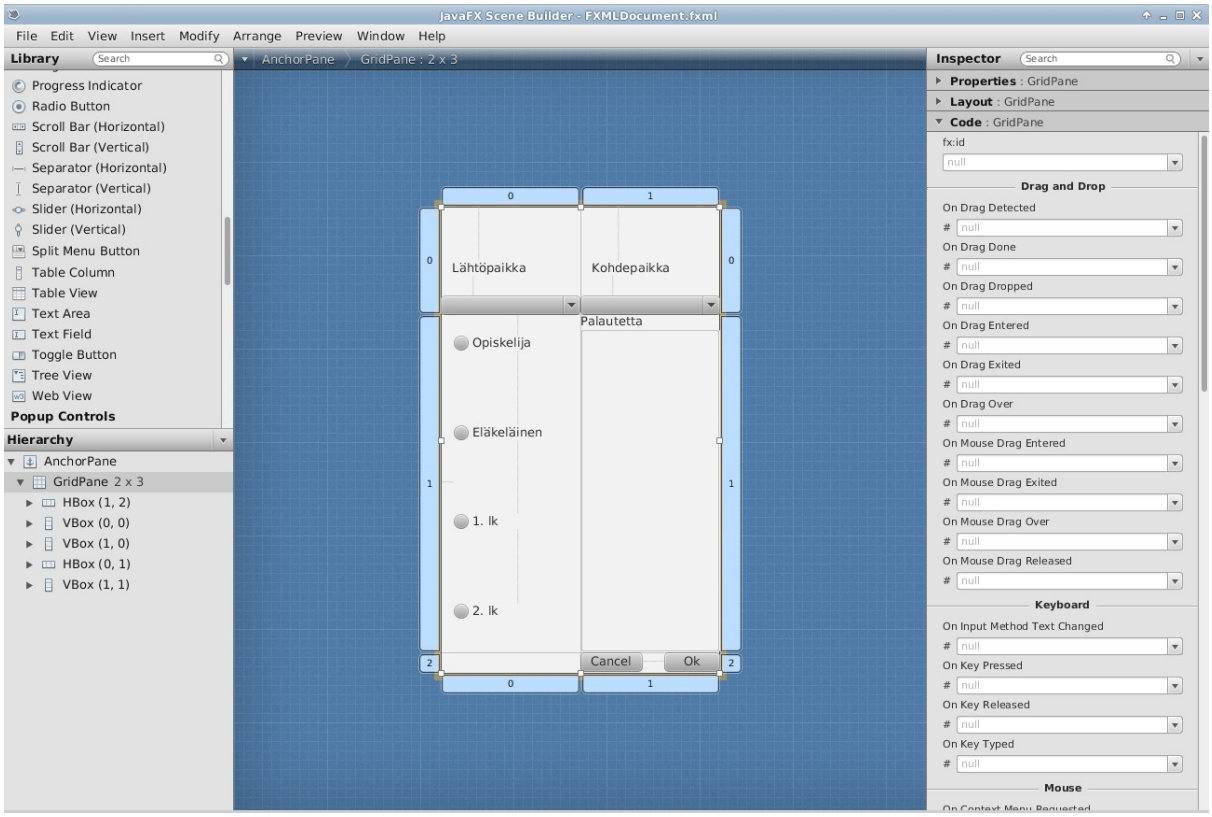
Nyt voimme lisätä seuraavat komponentit eli valintanapit. Valintanapeilla voidaan määrittää, onko ostaja opiskelija, eläkeläinen vaiko 1. luokan vai 2. luokan matkustaja. Lisätään valikkonapit vasemmalle keskelle, eli laatikkoon 0,1. Käytetään taas hyödyksi VBox- ja Separator- komponentteja samalla tavalla kuin ennenkin. Valintanapit on myös hyvä nimetä järkevästi, käytännössä kaikki komponentit kannattaa nimetä selkeästi.



Kuva 6.7. Valintanappien lisäys

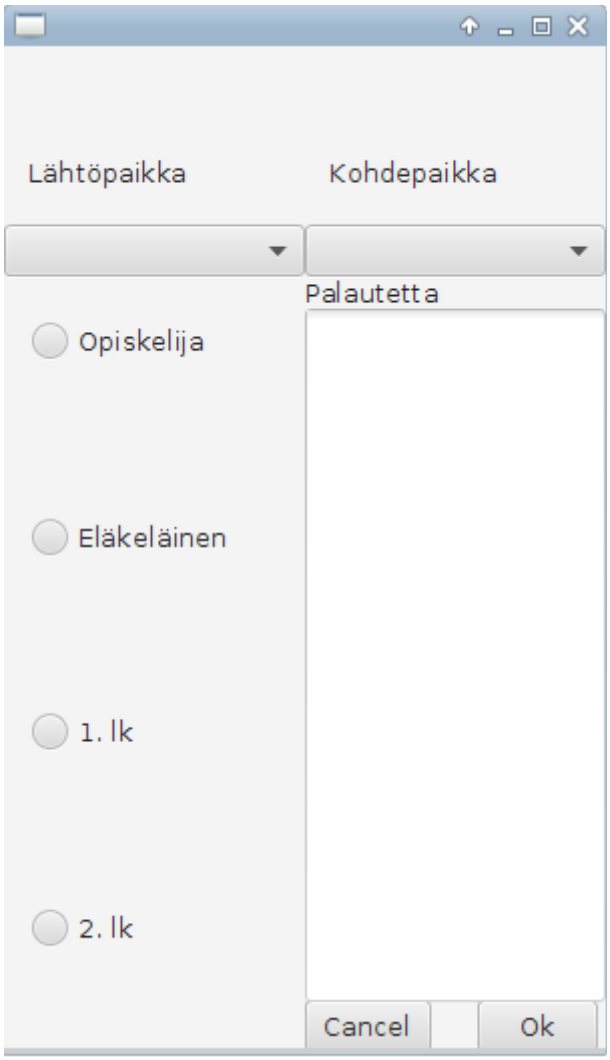
Käyttöliittymään lisättiin siis valintanappeja neljä kappaletta ja ne asetettiin separaattoreiden avulla tasaisin välein toisistaan. Kuten Hierarchy-ikkunasta voidaan havaita, Grid Panen ruutuun 0,1 on nyt asetettu ensin HBox, jonka sisällä on separaattori ja VBox, mikä pitää sisällään halutut valikkonapit. Ilman vaakasuuntaista separaattoria valintanapit olisivat aivan kiinni ruudun vasemmassa reunassa, mikä ei ole esteettisesti kovin suotavaa.

Lopuksi lisätään käyttöliittymään vielä tekstikenttä, jonka avulla käyttäjältä voidaan kerätä palautetta ja kommentteja lippuun ja sen tilaamiseen liittyen. Tämä vaatii taas oman VBox-komponentin, jonka sisälle sijoitamme tekstikentän ja tekstinsyöttöalueen (Text Area).



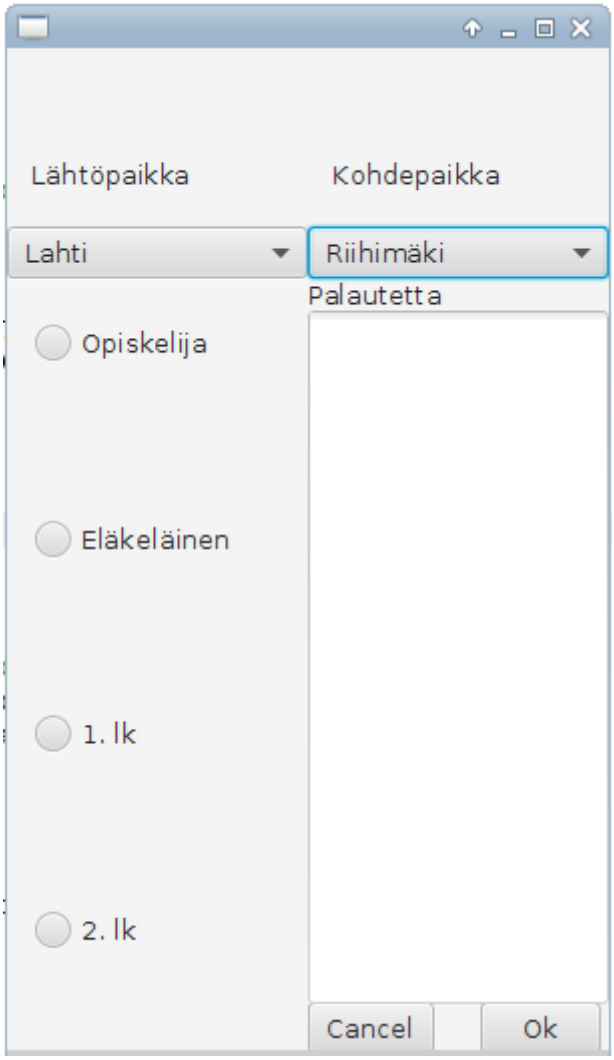
Kuva 6.8. Viimeiset komponentit

Kuvassa 6.9. on esitys siitä, millainen käyttöliittymä valmiista suunnitelmasta tuli ajamalla projekti. Huomattavaa on myös, että Separator-komponentit tulisi piilottaa Properties-välilehdestä (Visible-checkbox), jotta komponentit eivät näytä suunniteltaessa näkyviä viivoja lopullisessa käyttöliittymässä.



Kuva 6.9. Lopullinen käyttöliittymä

Lopullista käyttöliittymää on tietysti mukava katsella kuvassa 6.9., mutta siinä ei tällä hetkellä ole mitään toiminnallisuutta. Painikkeista ei tapahdu mitään, pudotusvalikot ovat tyhjiä ja palautetta voi antaa, mutta se pysyy vain käyttäjän ruudulla. On siis tarpeellista alkaa rakentamaan toiminnallisuutta käyttöliittymälle. Aloitetaan alustamalla pudotusvalikot ja luomalla tyhjät metodit painikkeille.



Kuva 6.10. Ensimmäiset toiminnallisuudet

Pudotusvalikoissa näkyy nyt valittuna kaksi kaupunkia, mutta tämä ei ollut pelkän käyttöliittymän kautta mahdollista. Miten tämä saatiin aikaan? Katsotaan koodia tiedostosta FXMLDocumentController.

Esimerkki 6.4. Pudotusvalikkojen alustus ja painikemetodit

```
public class FXMLDocumentController implements Initializable {

    @FXML
    private ComboBox fromBox;
    @FXML
    private ComboBox toBox;

    private ArrayList<String> locations;

    @Override
```

```
public void initialize(URL url, ResourceBundle rb) {
        locations = new ArrayList<String>();
        locations.add("Lappeenranta");
        locations.add("Helsinki");
        locations.add("Lahti");
        locations.add("Kouvola");
        locations.add("Riihimäki");
        populateBoxes();
    }

    public void populateBoxes() {
        for(String location : locations) {
            fromBox.getItems().add(location);
            toBox.getItems().add(location);
        }
    }

    @FXML
    public void okButtonClicked(ActionEvent event) {

    }

    @FXML
    public void cancelButtonClicked(ActionEvent event) {

    }

}
```

Ohjelmaan lisättiin aluksi pudotusvalikot noudattamalla samaa ohjeistusta kuin aikaisemmin. Lisäksi ohjelmaan täydennettiin yksityinen lista, johon lisättiin alkioita *initialize()*-metodissa ja erillinen metodi *populateBoxes()*, joka ei ole suoraan tekemisissä käyttöliittymän kanssa. Metodissa käydään läpi jo alustettu lista ja sijoitetaan kaikki listan alkiot molempiin valikoihin. Lisäksi ohjelmaan lisättiin toiminnallisuudet molemmille painikkeille, jotka on nimetty täysin samalla tavalla kuin käyttöliittymän puolella. Ohjelmassa siis kaikki @FXML-notaation jälkeen tulevat muuttujat tai metodit on alustettu käyttöliittymän puolella ja niihin päästään käsiksi Controller-luokassa.

Lisätään lopuksi painikkeille toiminnallisuuksia. Käyttöliittymän toimintaa ajatellen peruutuspainike vain sulkee ikkunan ja tyhjentää tiedon tietokoneen muistista, mutta hyväksymispainikkeella haluamme tehdä kerätyllä tiedolla jotakin. Pitääksemme esimerkin yksinkertaisena, painiketta painettaessa kerätään tieto kaikista komponenteista ja tulostetaan niiden sisältämä tieto konsoliin.

Esimerkki 6.5. Tulosten kerääminen komponenteista

```
public class FXMLDocumentController implements Initializable {

    @FXML private ComboBox fromBox;
    @FXML private ComboBox toBox;
    @FXML private Button cancelButton;
    @FXML private Button okButton;
    @FXML private TextArea commentsArea;

    @FXML private RadioButton student;
    @FXML private RadioButton elderly;
    @FXML private RadioButton firstClass;
    @FXML private RadioButton secondClass;


    private ArrayList<String> locations;

    @Override
    public void initialize(URL url, ResourceBundle rb) {
        locations = new ArrayList<String>();
        locations.add("Lappeenranta");
        locations.add("Helsinki");
        locations.add("Lahti");
        locations.add("Kouvola");
        locations.add("Riihimäki");
        populateBoxes();
    }

    public void populateBoxes() {
        for(String location : locations) {
            fromBox.getItems().add(location);
            toBox.getItems().add(location);
        }
    }

    @FXML public void okButtonClicked(ActionEvent event) {
        System.out.println("*** SYÖTTEET ***");
        System.out.println("Lähtöpaikka: " + fromBox.getValue());
        System.out.println("Kohdepaikka: " + toBox.getValue());
        System.out.println("Luokka: " + getTravelClass());
        System.out.println("Kommentit: " + commentsArea.getText());
        closeWindow(okButton);
    }

    public String getTravelClass() {
        String returnValue = "";
        if(student.isSelected()) {returnValue = "Opiskelija";}
        else if(elderly.isSelected()) {returnValue = "Eläkeläinen";}
        else if(firstClass.isSelected()) {returnValue = "Ensimmäinen luokka";}
        else if(secondClass.isSelected()) {returnValue = "Toinen luokka";}
        else {returnValue = null;}
```

```
        return returnValue;
    }

    @FXML public void cancelButtonClicked(ActionEvent event) {
        closeWindow(cancelButton);
    }

    public void closeWindow(Button button) {
        System.out.println("Kiitos käytöstä.");
        Stage scene = (Stage) button.getScene().getWindow();
        scene.close();
    }

}
```

Tuloste

```
*** SYÖTTEET ***
Lähtöpaikka: Helsinki
Kohdepaikka: Riihimäki
Luokka: Eläkeläinen
Kommentit: Näkisin, että tarvitsen ikkunapaikan välittömästi. En lähde muuten
matkaan ollenkaan.
Kiitos käytöstä.
```

Esimerkistä voidaan huomata, että se käyttää useita metodeja, jotka eivät liity suoraan käyttöliittymän toimintaan, mutta ovat tarpeellisia, jotta ohjelma toimisi. Suoritus on mahdollista myös luoda niin, että rakennetaan ohjelman taustalle erillinen toiminnallisuusluokka, joka käsittelee kaiken informaation, mitä käyttöliittymä käyttää. Tällöin käyttöliittymän koodi olisi vain ja ainoastaan tarkoitettu tiedon esittämiseen. Esimerkki on vain suppea esitys siitä, millaisia käyttöliittymiä työkalulla on mahdollista rakentaa, mutta JavaFX sisältää vielä monia erilaisia komponentteja, joiden käyttö on hyvin dokumentoitua.

Hieman pidemmälle vietynä esimerkki sisältää vielä ikkunointimahdollisuuden, josta painikkeen avulla on mahdollista lisätä uusi ikkuna, jonka kautta tietoa kerätään lisää. Eli avataan toinen ikkuna ensimmäisen ikkunan päälle ja kerätään käyttäjältä syötettä tästäkin ikkunasta. Metodeja tämän ominaisuuden luomiseksi on muutamia, joista seuraavassa Singleton.

Esimerkki 6.6. Toisen ikkunan avaaminen ohjelmassa

```
    @FXML public void askMoreInfo(ActionEvent event) {
```

```
        try {
            moreinfo = new Stage();
            Parent page =
                    FXMLLoader.load(getClass().getResource("FXMLMoreInfo.fxml"));
            Scene scene = new Scene(page);
            moreinfo.setScene(scene);
            moreinfo.show();
        } catch (IOException ex) {
            System.err.println("Error occured.");
        }
    }
```

Esimerkkiin on lisätty painike, jonka avulla on mahdollisuus kerätä lisätietoja käyttäjltä. Painikkeen toiminnallisuudessa luodaan uusi Stage-olio, ladataan Scene Builderilla luotu fxml-tiedosto ja tehdään sama operaatio kuin tehdään *start()*-metodissa, kun ohjelma käynnistetään.

FXMLMoreInfo.fxml on tiedosto, joka on luotu Scene Builderilla ja sille on rakennettu oma Controller-luokka, joka on esimerkissä 6.7.

Esimerkki 6.7. FXMLMoreInfon Controller

```
public class FXMLMoreInfoController implements Initializable {

    @FXML private Button okButton;
    @FXML private Button cancelButton;
    @FXML private TextArea text;

    @Override
    public void initialize(URL url, ResourceBundle rb) {
    }

    @FXML public void okButtonClicked(ActionEvent event) {
        MoreInfoObject.getInstance().setText(text.getText());
        closeWindow(okButton);
    }

    @FXML public void cancelButtonClicked(ActionEvent event) {
        closeWindow(cancelButton);
    }

    public void closeWindow(Button button) {
        Stage scene = (Stage) button.getScene().getWindow();
        scene.close();
    }
}
```

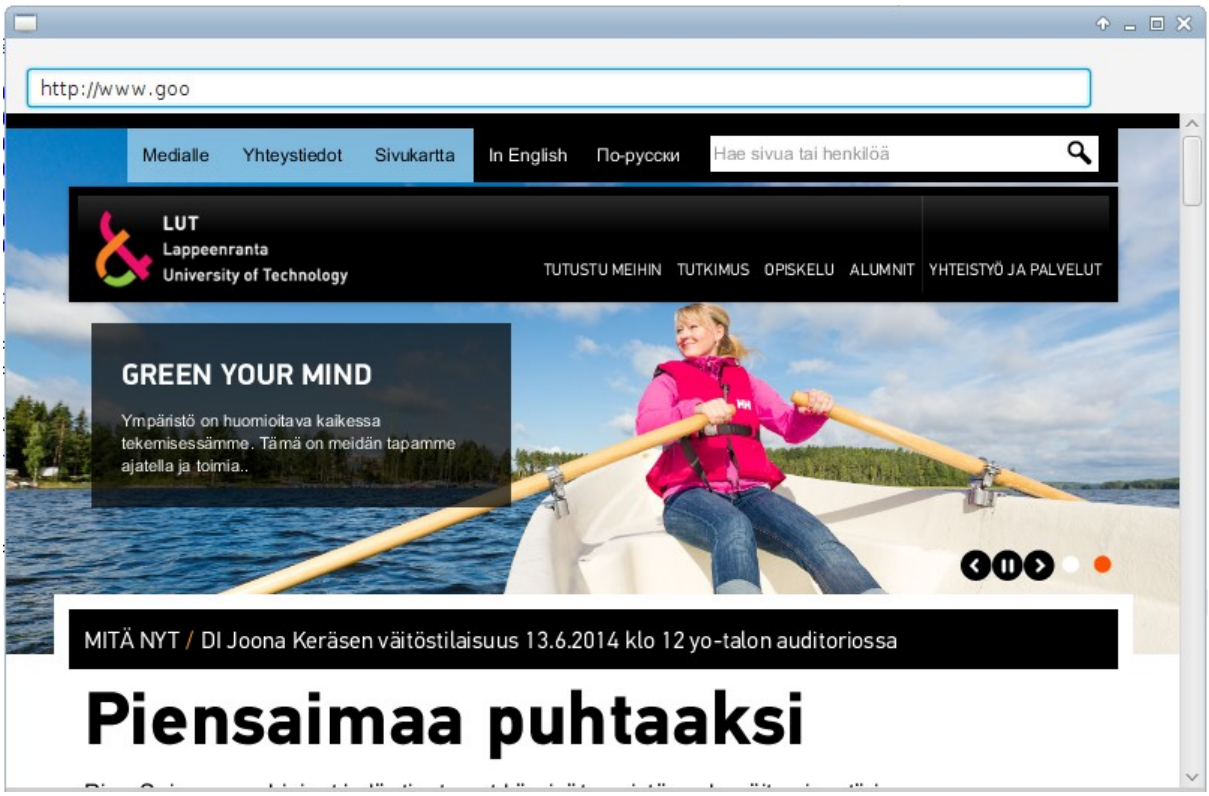
MoreInfoObject.java

```
public class MoreInfoObject {
   private static MoreInfoObject instance = null;
   private String text;

   protected MoreInfoObject() {}

   public static MoreInfoObject getInstance() {
      if(instance == null) {
         instance = new MoreInfoObject();
      }
      return instance;
   }

   public String getText() {
      return text;
   }

   public void setText(String input) {
      text = input;
   }
}
```

Esimerkissä on luotu uusi käyttöliittymäikkuna, sillä Controller, jonka avulla käyttöliittymän toiminnallisuutta hallitaan sekä erillinen tieto-olio, joka pitää tallessa tietoa siitä, mitä tekstiä käyttäjä syötti, jotta syötettä voidaan käyttää myös toisessa ikkunassa. Huomaa, että tämän ohjelman tarkoitus ei ole kerätä toisesta ikkunasta kuin yksi merkkijono, jonka vuoksi taustaluokka on luotu käyttäen Singleton-suunnitteluperiaatetta, joka on selitetty olioihin keskittyvässä oppaassa luvussa 7. Tätä luokkaa voidaan kyllä muuttaa niin, että se pitää sisällään listan merkkijonoista tai muista tietosäilöistä, joita se tarjoaa molemmille ikkunoille samanaikaiseen käyttöön.

6.3. Verkkoselain-esimerkki

Tehdään vielä toinen esimerkki, jossa havainnollistetaan JavaFX:n erästä hyvin näppärää komponenttia, WebView-komponenttia. Komponentin avulla on mahdollista luoda verkkoselain vain yhden komponentin avulla ja se mahdollistaa myös rajapinnan JavaScriptin kanssa, eli ohjelma voi keskustella verkkosivun kanssa, eli lähettää sinne tietoa ja pyytää tietoa takaisin olemassa olevan rajapinnan läpi.



Kuva 6.11. Yksinkertainen verkkoselain

Verkkoselain ei sisällä muuta toiminnallisuutta kuin verkkosivun alustamisen heti ohjelman käynnistyessä sekä tekstikentästä osoitteen ottamisen. Tämä esimerkki ei ole kovin laaja, mutta selainta on hyvin helppo lähteä laajentamaan esimerkiksi eri painikkeilla, sillä JavaFX tarjoaa käytännöllisiä työkaluja tarpeen mukaan. Ohjelma on toiminnallisuuden puutteen vuoksi hyvin yksinkertainen ja se pitää sisällään vain muutaman rivin koodia.

Esimerkki 6.8. Verkkoselaimen toiminnallisuus

```
public class FXMLDocumentController implements Initializable {

    @FXML private WebView webWindow;
    @FXML private TextField urlField;

    @Override
    public void initialize(URL url, ResourceBundle rb) {
        webWindow.getEngine().load("http://www.google.fi");
    }

    @FXML public void onEnterPressed(ActionEvent event) {
        webWindow.getEngine().load(urlField.getText());
        urlField.clear();
    }
}
```

Jos ohjelmoija haluaisi käyttää paikallisia tiedostoja, eli omalla työpisteellä olevia .html-tiedostoja, on ne parsittava muotoon, jonka WebView tunnistaa. Tällöin osoite on laadittava rivinä:

```
webWindow.getEngine().load(getClass().getResource("index.html").toExternalForm());
```

Koska *load()*-metodi ottaa sisäänsä merkkijonona URL-osoitteen, on se haettava staattisen metodin kautta resurssina ja lisättävä myös ohjelman juurikansioon, eli NetBeansin tapauksessa projektin kansioon. Tällöin ohjelma osaa etsiä tarvittavan tiedoston ja voi sen esittää ruudulla.

6.4. Komponenttien dynaaminen lisääminen

Joskus voi tulla tarpeeseen lisätä komponentteja ruudulle vain ja ainoastaan tarpeen mukaan, esimerkiksi painiketta painettaessa tai hiiren avulla. Seuraavassa käydään lyhyt esimerkki, miten muotoja voidaan lisätä yksinkertaisen, tyhjän alustan päälle hiiren avulla.

Esimerkki 6.9. Muotojen dynaaminen lisääminen

```
public class FXMLDocumentController implements Initializable {

    @FXML private AnchorPane base;

    @Override
    public void initialize(URL url, ResourceBundle rb) {
    }

    @FXML public void baseMouseClicked(MouseEvent event) {
        Circle c = new Circle();
        c.setLayoutX(event.getX());
        c.setLayoutY(event.getY());
        c.setRadius(10);
        c.setFill(Color.BLUE);
            c.setOnMouseClicked(new EventHandler<MouseEvent>() {
            @Override
            public void handle(MouseEvent event) {
                System.out.println("X: " + event.getX() +
                                " , Y: " + event.getY());
            }
        });

        base.getChildren().add(c);
    }
```

```
}
```

Esimerkissä käytetään tavallista ympyrämuotoa, jonka avulla voidaan esittää pisteitä alustalla. Esimerkin tarkoitus on kyetä lisäämään pohjan päälle ympyräolio aina, kun hiirtä klikataan alustan päällä. *event*-olion avulla saadaan tieto siitä, missä hiiren kursori on klikkaushetkellä ollut ja sen avulla asetetaan ympyrä klikattuun kohtaan. *baseMouseClicked()* on AnchorPane-alustan ominaisuus, joka on liitetty tapahtumaan, jossa hiirtä klikataan.

Lisäksi ympyräoliolle lisätään tapahtuman käsittelijä, joka reagoi silloin, kun ympyrää käsitellään. Tämä käsittelijä voidaan lisätä suoraan Scene Builderin kautta, jos olio on staattinen, mutta dynaamisessa lisäämisessä on käytettävä esimerkin keinoa.

6.5. JavaScript-rajapinta

JavaFX mahdollistaa WebEngine-komponentin kautta ominaisuuden, jonka avulla voidaan lähettää verkkosivulle JavaScript-muotoisia käskyjä. Tämä tarkoittaa käytännössä sitä, että käyttöliittymän kautta on mahdollista kirjoittaa JavaScript-käskyjä verkkosivulle, joilla on mahdollista muokata, karsia ja lisätä toiminnallisuuksia sivustojen sisältöön, mutta muutokset ovat näkyvissä vain itse ajetussa instanssissa. Tämä voi kuulostaa hyvin tutulta menetelmältä, jos esimerkiksi on käyttänyt mainosten estämiseen tarkoitettua selainlisäystä (AdBlocker tai jokin muu).

Käytännössä käskyjä on yksinkertaista suorittaa käyttämällä itse tehtyä html-tiedostoa, jonne lisätään JavaScript-metodeita ja niitä käytetään käyttöliittymän kautta. Tehdään näin esimerkin avulla, jossa käydään läpi myös hieman JavaScriptiä selvyyden vuoksi.

Esimerkki 6.10. JavaScript-rajapinta

```
@FXML public void shoutBtnClicked() {
      webWindow.getEngine().executeScript("document.shoutOut()");
}
```

index.html

```
<html>
<head>
<script type=”text/javascript”>
        function initialize() {
```

```
                document.getElementById("random").innerHTML = "Moi Maailma!";
        }

        document.shoutOut = function shoutOut() {
                document.getElementById("random").innerHTML = "Hello World!";
        }
</script>
</head>
<body onload=”initialize()”>
        <p id=”random”></p>
</body>
</html>
```

Esimerkissä kannattaa kiinnittää huomiota elementtien <script> ja </script> väliin, missä kaikki verkkosivun JavaScript-suoritus sijaitsee. Tiedostoon on määritelty kaksi metodia, joista initialize() on alustusmetodi, joka ajetaan heti kun sivusto ladataan eli avataan. document.shoutOut() on metodi, joka muuttaa alustuksessa määritellyn tekstin käyttämällä samaa elementtiä ja samaa keinoa tekstin muuttamiseen. index.html-tiedoston yläpuolella näkyy JavaFX-painikemetodi, jossa on kutsuttu haluttua JavaScript-metodia WebEngine-luokan executeScript()-metodia käyttäen.

executeScript()-metodi ottaa parametrikseen funktiokutsun tai koodirivin vain merkkijonona, mutta ei hyväksy esimerkiksi rivinvaihtoja. Tämä on suotavaa huomata varsinkin silloin, jos pyritään lähettämään parametreja funktiokutsun mukana. Tällöin numeromuotoiset muuttujat voidaan liittää vain +-merkin avulla mukaan lähetettävään merkkijonoon, mutta merkkijonoissa on muistettava lisätä uudet heittomerkit mukaan merkkijonoa parsittaessa. Esimerkkinä käytetään metodia, joka ottaa kolme parametria: taulukon (path), merkkijonon (pathColor) sekä numeron (choice).

```
engine.executeScript(
   "document.createPath("+path +", '"+pathColor+"', "+choice+")");
```

Taulukkoa ei ole tarpeen erotella, kuten ei ole numeroakaan, mutta merkkijonosta huomaamme, että sen ympärille on asetettu ‘’-merkit, kuten esimerkissä '"+pathColor+"' on kiedottu yksinkertaisilla lainausmerkeillä. Tällöin JavaScript ymmärtää, että kyseessä on merkkijono ja osaa toimia oikein parametrin kanssa.

7. Lähteet

http://opendatahandbook.org/en/what-is-open-data/

http://en.wikipedia.org/wiki/Open_data

http://oskari.org/

http://www.paikkatietoikkuna.fi/web/fi

8. Liite 1. Oskarin käyttöönotto

Oskarin käyttöönotto aloitetaan hakemalla tarvittavat tiedostot githubista, jossa voi valita ottaako käyttöön vain käyttöliittymän vaiko myös palvelualustan. Käyttöliittymän lähdekoodi löytyy osoitteesta https://github.com/nls-oskari/oskari ja palvelualustan lähdekoodi osoitteesta https://github.com/nls-oskari/oskari-server. Käydään aluksi tässä oppaassa läpi käyttöliittymän käyttöönotto ja lopuksi palvelualustan käyttöönotto. Huomaathan, että oppaan esimerkeissä on käytetty Oskarin versiota 1.23.0

Kuten edellä mainittiin käyttöliittymän arkkitehtuurin yhteydessä, Oskari rakentuu koottavista bundleista sekä tarvittavista konfiguraatiotiedostoista. Käyttöliittymä käynnistetään avaamalla index.html-tiedosto, joka lataa tarvittavia tiedostoja, ensimmäisenä index.js-tiedoston. Esimerkeissä käytetään Oskarissa valmiina tulevia esimerkkitiedostoja tiedostopolusta Oskari/applications/sample/myfirst. Tiedostossa on kolme tiedostoa: index.html, index.js ja appsetup.json. index.html on perustiedosto, jonka avulla karttakäyttöliittymä käynnistetään ja se lataa toimintaansa ensimmäisenä jQuery-tiedostot, bundlen konfiguraatioon tarvittavan bundle.js, OpenLayers-alustuksen startup.js ja lopuksi index.js:n.

Esimerkki L1.1. index.html

```
<!DOCTYPE html>
<html>
<head>
<title>demo</title>
<!-- ############## css ################## -->

<script type="text/javascript"
src="http://code.jquery.com/jquery-1.7.2.min.js">
</script>

<link rel="stylesheet" type="text/css" href="../css/icons.css" />
<style type="text/css">
@media screen {
 body {
  margin : 0;
  padding: 0;
 }
 #mapdiv {
  width: 100%;
 }
 #contentMap {
  height: 100%;
```

```
 }
}
</style>
<!-- ############## /css ################## -->
</head>
<body>

<div id="contentMap">
<div id="mapdiv"></div>
</div>


<!-- ############## Javascript ################## -->

<!--  OSKARI -->
<script type="text/javascript"
        src="../../../bundles/bundle.js">
</script>

<!--  OPENLAYERS -->
<script type="text/javascript"
        src="../../../packages/openlayers/startup.js">
</script>

<script type="text/javascript" src="index.js">
</script>

<!-- ############## /Javascript ################## -->
</body>
</html>
```

index.js-tiedosto sisältää tarpeellisten konfiguraatiotiedostojen appsetup.json ja config.json lataamisen sekä lopulta käyttöliittymän käynnistämisen. Tiedostossa on oltava tarkkana, että tiedostopolut ovat molempien konfiguraatioiden kanssa tismalleen oikein. Esimerkissä oleva config.json löytyy kansiosta Oskari/applications/sample, jonka vuoksi kansiopoluksi on asetettu ../config.json, eli ajokansion alakansio.

Esimerkki L1.2. index.js

```
jQuery(document).ready(function () {
    Oskari.setLang('en');
    Oskari.setLoaderMode('dev');
    var appSetup,
        appConfig,
        downloadConfig = function (notifyCallback) {
            jQuery.ajax({
                type: 'GET',
                dataType: 'json',
                url: '../config.json',
```

```
            beforeSend: function (x) {
                if (x && x.overrideMimeType) {
                    x.overrideMimeType("application/j-son;charset=UTF-8");
                }
            },
            success: function (config) {
                appConfig = config;
                notifyCallback();
            }
        });
    },
    downloadAppSetup = function (notifyCallback) {
        jQuery.ajax({
            type: 'GET',
            dataType: 'json',
            url: 'appsetup.json',
            beforeSend: function (x) {
                if (x && x.overrideMimeType) {
                    x.overrideMimeType("application/j-son;charset=UTF-8");
                }
            },
            success: function (setup) {
                appSetup = setup;
                notifyCallback();
            }
        });
    },
    startApplication = function () {
        // check that both setup and config are loaded
        // before actually starting the application
        if (appSetup && appConfig) {
            var app = Oskari.app;
            app.setApplicationSetup(appSetup);
            app.setConfiguration(appConfig);
            app.startApplication(function (startupInfos) {
                // all bundles have been loaded
            });
        }
    };
    downloadAppSetup(startApplication);
    downloadConfig(startApplication);
});
```

config.json sisältää peruskonfiguraation karttojen toimintaan, eli se lataa tarvittavat perusliitännäiset, karttavalinnat (*map options*) sekä tarvittavat karttatasot: taustakartta, maastokartta ja ortokuvakartta. Tiedosto on useita satoja rivejä, joten sitä ei tässä oppaassa esitetä. Tavallisesti projektia varten config.json-tiedostoa ei ole tarpeen muokata, vaan se toimii karttapohjalla aina.

Lopulta käynnistys pääsee jo startSequence-metodiin asti, kun appsetup.json-tiedosto ladataan. Tähän tiedostoon lisätään kaikki käyttöliittymän tarvitsemat bundlet, josta käydään esimerkkiä läpi seuraavassa.

Esimerkki L1.3. appsetup.json

```
{
 "startupSequence" : [
    {
     "title" : "OpenLayers",
     "en" : "OpenLayers",
     "fi" : "OpenLayers",
     "sv" : "OpenLayers",
     "bundleinstancename" : "openlayers-default-theme",
     "bundlename" : "openlayers-default-theme",
     "instanceProps" : {  },
     "metadata" : {
        "Import-Bundle" : {
           "openlayers-default-theme" :
{ "bundlePath" : "../../../packages/openlayers/bundle/" },
           "openlayers-single-full" :
{ "bundlePath" : "../../../packages/openlayers/bundle/" }
        },
        "Require-Bundle-Instance" : [  ]
     }
    },
    {
     "title" : "Map",
     "en" : "Map",
     "fi" : "Map",
     "sv" : "Map",
     "bundleinstancename" : "mapfull",
     "bundlename" : "mapfull",
     "instanceProps" : {  },
     "metadata" : {
        "Import-Bundle" : {
              "oskariui" : { "bundlePath" :
 "../../../packages/framework/bundle/" },
           "core-base" : { "bundlePath" :
 "../../../packages/framework/bundle/" },
           "core-map" : { "bundlePath" :
 "../../../packages/framework/bundle/" },
           "domain" : { "bundlePath" :
 "../../../packages/framework/bundle/" },
           "event-base" : { "bundlePath" :
"../../../packages/framework/bundle/" },
           "event-map" : { "bundlePath" :
"../../../packages/framework/bundle/" },
           "event-map-layer" : { "bundlePath" :
```

```
 "../../../packages/framework/bundle/" },
            "mapfull" : { "bundlePath" :
"../../../packages/framework/bundle/" },
            "mapmodule-plugin" : { "bundlePath" :
"../../../packages/framework/bundle/" },
            "mapwmts" : { "bundlePath" :
"../../../packages/framework/bundle/" },
            "request-base" : { "bundlePath" :
"../../../packages/framework/bundle/" },
            "request-map" : { "bundlePath" :
"../../../packages/framework/bundle/" },
            "request-map-layer" : { "bundlePath" :
 "../../../packages/framework/bundle/" },
            "sandbox-base" : { "bundlePath" :
"../../../packages/framework/bundle/" },
            "sandbox-map" : { "bundlePath" :
"../../../packages/framework/bundle/" },
            "service-base" : { "bundlePath" :
"../../../packages/framework/bundle/" },
            "service-map" : { "bundlePath" :
"../../../packages/framework/bundle/" }
          },
          "Require-Bundle-Instance" : [  ]
       }
     },
     {
       "title" : "My1st",
       "en" : "My1st",
       "fi" : "My1st",
       "sv" : "My1st",
       "bundleinstancename" : "myfirstbundle",
       "bundlename" : "myfirstbundle",
       "instanceProps" : {  },
       "metadata" : {
          "Import-Bundle" : {
             "myfirstbundle" : { "bundlePath" :
"../../../packages/sample/bundle/" }
          },
          "Require-Bundle-Instance" : [  ]
       }
     }
  ]
}
```

Edellä oleva esimerkki on edelleen samasta sample-projektista myfirst, jossa ladataan tarvittavat bundlet OpenLayers ja karttakomponentti. Kuten esimerkissä voidaan huomata, tiedostopolut ovat kaikki kovakoodattuina tähän tiedostoon, joten komponentin tai tämän tiedoston siirtäminen paikasta toiseen vaativat muutoksia myös näihin tiedostopolkuihin. Jos ohjelmaan halutaan lisätä muita bundleja, onnistuu se etsimällä bundle

Oskari/bundles/framework/bundle-kansiosta ja valitsemalla haluttu bundle, esimerkiksi divmanager, jolla hallinnoidaan käyttöliittymän työkaluja.

Esimerkki L1.4. Divmanagerin lisääminen appsetup.jsoniin

```
{
     "title" : "Oskari DIV Manazer",
     "en" : "Oskari DIV Manazer",
     "fi" : "Oskari DIV Manazer",
     "sv" : "Oskari DIV Manazer",
     "bundleinstancename" : "divmanazer",
"bundlename" : "divmanazer",
     "instanceProps" : {  },
     "metadata" : {
        "Import-Bundle" : {
           "divmanazer" : { "bundlePath" :
"../../../packages/framework/bundle/" }
        },
        "Require-Bundle-Instance" : [  ]
     }
    }
```

Huomaa myös, että eri bundlet on eriteltävä tiedostossa pilkun avulla, jotta ne voidaan huomioida ja ottaa käyttöön.

Kolmas ladattava bundle esimerkissä L1.3. on esimerkkibundle My1st, joka otetaan myös käyttöön yksinkertaisen suorituksen mukaan saamiseksi. Etsitään siis tarvittava bundle annetun tiedostopolun Oskari/packages/sample/bundle/myfirstbundle päästä ja katsotaan, mitä se sisältää. Kansiosta löytyy yksi tiedosto, bundle.js.

Esimerkki L1.5. bundle.js

```
/**
 * @class Oskari.sample.bundle.myfirstbundle.SimpleHelloWorldBundle
 *
 * Definition for bundle. See source for details.
 */
Oskari.clazz.define("Oskari.sample.bundle.myfirstbundle.SimpleHelloWo
rldBundle",

/**
 * @method create called automatically on construction
 * @static
 */
function() {

}, {
    "create" : function() {
        var me = this;
```

```
        var inst =
Oskari.clazz.create("Oskari.sample.bundle.myfirstbundle.SimpleHelloWo
rldBundleInstance");
        return inst;

    },
    "update" : function(manager, bundle, bi, info) {

    }
}, {

    "protocol" : [ "Oskari.bundle.Bundle" ],
    "source" : {

        "scripts" : [{
            "type" : "text/javascript",
            "src" : "../../../../bundles/sample/bundle/" +
                        "myfirstbundle/instance.js"
        }]
    },
    "bundle" : {
        "manifest" : {
            "Bundle-Identifier" : "myfirstbundle",
            "Bundle-Name" : "myfirstbundle",
            "Bundle-Author" : [{
                "Name" : "ev",
                "Organisation" : "nls.fi",
                "Temporal" : {
                    "Start" : "2009",
                    "End" : "2011"
                },
                "Copyleft" : {
                    "License" : {
                        "License-Name" : "EUPL",
                        "License-Online-Resource" :
                                    "http://www.paikkatietoikkuna.fi/
license"
                    }
                }
            }],
            "Bundle-Name-Locale" : {
                "fi" : {
                    "Name" : " style-1",
                    "Title" : " style-1"
                },
                "en" : {}
            },
            "Bundle-Version" : "1.0.0",
            "Import-Namespace" : ["Oskari"],
            "Import-Bundle" : {}

        }
    },

    /**
    * @static
    * @property dependencies
    */
    "dependencies" : []
```

```
});

Oskari.bundle_manager.installBundleClass("myfirstbundle",
        "Oskari.sample.bundle.myfirstbundle.SimpleHelloWorldBundle");
```

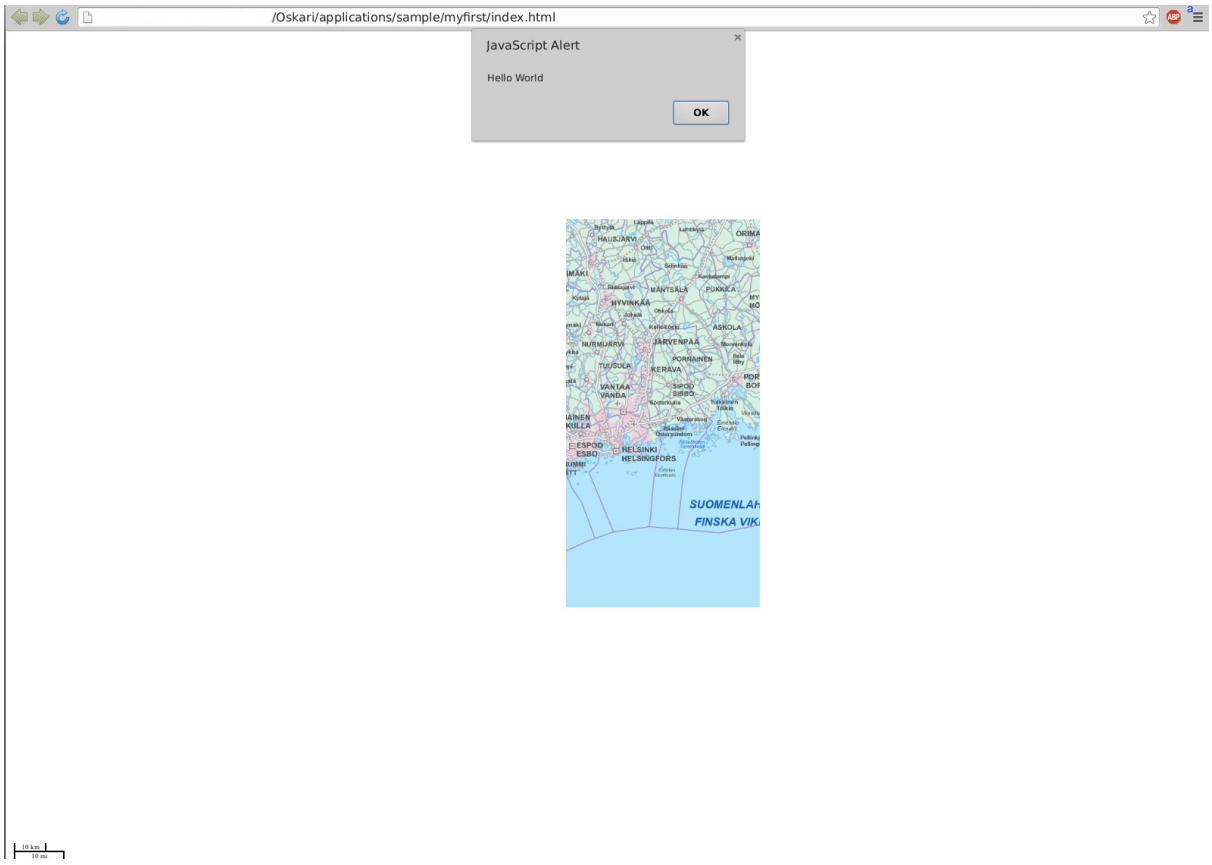
Tämän tiedoston tarkoituksena on määritellä create ja update toiminnallisuudet sekä protokolla, tarjota bundlen lähdekooditiedostot sekä määrittää bundlen käyttötiedot. Tiedoston sisältämät tiedot pysyvät muuten muuttumattomina, mutta instanse.js-lähdekooditiedoston tiedostopolku on tarkistettava aina, jotta bundle varmasti lataa toiminnallisuudet. Käydään tämän tiedoston muokkaamista tarkemmin läpi, kun luodaan esimerkkinä uusi bundle. Seuraavana käydään läpi, mitä sisältää tiedosto instance.js tiedostopolusta Oskari/bundles/sample/bundle/myfirstbundle/instance.js, sillä edellä läpikäyty tiedosto ei vielä sisällä kartan päällä tapahtuvaa toiminnallisuutta.

Esimerkki L1.6. instance.js

```
/**
 * @class Oskari.sample.bundle.myfirstbundle.SimpleHelloWorldBundleInstance
 *
 * This bundle demonstrates a simplest possible bundle
 * that will just alert a Hello World message on startup.
 *
 * Add this to startupsequence to get this bundle started
 {
        title : 'myfirstbundle',
        fi : 'myfirstbundle',
        sv : '?',
        en : '?',
        bundlename : 'myfirstbundle',
        bundleinstancename : 'myfirstbundle',
        metadata : {
           "Import-Bundle" : {
              "myfirstbundle" : {
                 bundlePath : '/<path to>/packages/sample/bundle/'
              }
           },
           "Require-Bundle-Instance" : []
        },
        instanceProps : {}
     }
 */
Oskari.clazz.define("Oskari.sample.bundle.myfirstbundle.SimpleHelloWorldBundleI
nstance",

   /**
 * @method create called automatically on construction
    * @static
    */
```

```
function () {}, {
    /**
     * @method start
     * BundleInstance protocol method
     */
    start: function () {
        // ************************************
        //   Your code here
        // ************************************
        alert('Hello World');
        // ************************************
        //   Your code ends
        // ************************************
    },

    /**
     * @method stop
     * BundleInstance protocol method
     */
    stop: function () {},
    /**
     * @method update
     * BundleInstance protocol method
     */
    update: function () {}
}, {
    protocol: ['Oskari.bundle.BundleInstance']
});
```

Tiedoston alussa on kommenttikentän avulla ilmoitettu, mitä on tarpeen lisätä aikaisemmin nähtyyn startupSequence-metodiin, eli appsetup.json-tiedostoon, jotta bundle on mahdollista ottaa käyttöön. Tämä yksinkertainen esimerkki osoittaa, että bundlen toiminnallisuudesta on aina löydyttävä metodit start, stop ja update, jotka määrittävät toiminnallisuudet tarvittavissa tilanteissa. Esimerkissä oleva toiminnallisuus sisältää vain rivin start-metodissa, jossa kuulutetaan selaimessa JavaScriptin avulla “Hello World!”. Jos bundleen halutaan lisätä lisätoiminnallisuutta, voidaan se tehdä tässä tiedostossa ja se välittyy suoraan käyttöliittymälle aikaisemmin esiteltyjen tiedostojen avulla.



Kuva L1.1. Oskarin käyttöliittymän esimerkki selaimessa ajettuna

Huomaathan, että kartta ei ole latautunut, koska selain vaatii JavaScriptin alert-metodin sulkemisen ennen kuin kaikki loput karttakomponentit ladataan.

Rakennetaan esimerkin vuoksi täysin oma bundle, joka sisältää muutamia piirto-ominaisuuksia. Luodaan samalla uusi applikaatio, johon nämä ominaisuudet sidotaan. Aluksi käytetään jo esimerkistä L1.1. tuttua index.html-tiedostoa, johon vaihdetaan vain otsikon (*title*) tilalle oma otsikko.

index.js-tiedosto ei tarvitse muutoksia, jos käytetään samanlaista kansiorakennetta kuin esimerkissä, mutta pienessä projektissa on loogisempaa pitää konfiguraatio samassa tiedostossa, jolloin index.js-tiedostoon vaihdetaan config.json-tiedoston kansiopolku osoittamaan käytössä olevaan kansioon.

config.json-tiedosto ei itsessään tarvitse muutoksia, sillä se kuvaa kartan ruudulle piirtymistä ja reunoja, jolloin se voi olla tavallisessa projektissa aina samanlainen. appsetup.json-

tiedostoon tulee lisätä mukaan liitäntä, jossa otetaan käyttöön itse rakennettu bundle, tämän esimerkin tapauksessa MyDrawSample.

Esimerkki L1.7. MyDrawSamplen käyttöönotto appsetup.json-tiedostossa

```
{
      "title" : "MyDrawSample",
      "en" : "MyDrawSample",
      "fi" : "MyDrawSample",
      "sv" : "MyDrawSample",
      "bundleinstancename" : "mysearchdraw",
      "bundlename" : "mysearchdraw",
      "instanceProps" : {  },
      "metadata" : {
         "Import-Bundle" : {
            "mysearchdraw" : { "bundlePath" :
 "../../packages/framework/bundle/" }
         },
         "Require-Bundle-Instance" : [  ]
      }
    }
```

Jotta bundle olisi mahdollista rakentaa, on sille luotava omat kansionsa sekä Oskari/packages/framework/bundle- ja Oskari/bundles/framework/bundle-kansioihin. Kansion nimi on määritelty edellä olevassa esimerkissä L1.7., jossa bundlename ja bundleinstancename kertovat tarvittavan kansion nimen. instance.js-tiedosto luodaan siis kansiopolun Oskari/bundles/framework/bundle/mysearchdraw päähän ja bundle.js-tiedosto kansoiopolun Oskari/packages/framework/bundle/mysearchdraw.

Esimerkki L1.8. bundle.js-tiedoston sisältö

```
/**
 * Definition for bundle. See source for details.
 *
 * @class Oskari.<mynamespace>.bundle.<bundle-identifier>.MyBundle
 */
Oskari.clazz.define("Oskari.myframe.bundle.mysearchdraw.MyDrawBundle",

/**
 * Called automatically on construction. At this stage bundle sources have been
 * loaded, if bundle is loaded dynamically.
 *
 * @contructor
* @static
 */
function() {
```

```
}, {
    /*
     * called when a bundle instance will be created
     *
     * @method create
     */
    "create" : function() {
                                                                        return
Oskari.clazz.create("Oskari.myframe.bundle.mysearchdraw.MyDrawBundleInstanc
e");
    },
    /**
     * Called by Bundle Manager to provide state information to
     *
     * @method update
     * bundle
     */
    "update" : function(manager, bundle, bi, info) {
    }
},

/**
 * metadata
 */
{
    "protocol" : ["Oskari.bundle.Bundle"],
    "source" : {
        "scripts" : [{
            "type" : "text/javascript",
            "src" : "../../../../bundles/framework/bundle/mysearchdraw/instance.js"
        }]
    },
    "bundle" : {
        "manifest" : {
            "Bundle-Identifier" : "mysearchdraw",
            "Bundle-Name" : "mysearchdraw",
            "Bundle-Author" : [{
                "Name" : "ev",
                "Organisation" : "nls.fi",
                "Temporal" : {
                    "Start" : "2009",
                    "End" : "2011"
                },
                "Copyleft" : {
                    "License" : {
                        "License-Name" : "EUPL",
                                                        "License-Online-Resource"    :
"http://www.paikkatietoikkuna.fi/license"
                    }
                }
            }],
            "Bundle-Name-Locale" : {
```

```
            "fi" : {
               "Name" : " style-1",
               "Title" : " style-1"
            },
            "en" : {}
         },
         "Bundle-Version" : "1.0.0",
         "Import-Namespace" : ["Oskari"],
         "Import-Bundle" : {}
      }
   },

   /**
    * @static
    * @property dependencies
    */
   "dependencies" : []
});

// Install this bundle by instantating the Bundle Class
Oskari.bundle_manager.installBundleClass("mysearchdraw",
"Oskari.myframe.bundle.mysearchdraw.MyDrawBundle");
```

Tärkeimpiä huomioita tästä tiedostosta ovat rivit, joissa määritetään bundlen nimeä ja sen käyttämää nimiavaruutta. Tiedoston alussa oleva kommenttikenttä määrää, että jokaisen luotavan bundlen on noudatettava seuraavaa luokkapolkua, “@class Oskari.<mynamespace>.bundle.<bundle-identifier>.MyBundle”, joka esimerkin tapauksessa on "Oskari.myframe.bundle.mysearchdraw.MyDrawBundle". Tämä luokkapolku on tarpeen määritellä tässä tiedostossa, kun bundle halutaan luoda. Nimiavaruudeksi kehittäjä voi asettaa minkä tahansa vapaavalintaisen nimiavaruuden, jona tässä on käytetty avaruutta “myframe”.

appsetup.json-tiedostossa määriteltiin myös bundleinstancename, joka etsii instance.js-tiedostoa nimen ja tiedostopolut avulla. Seuraavassa on esimerkki siitä, millainen instance.js on mahdollista luoda tähän esimerkkiin, eli pisteiden piirtämiseen liittyen.

Esimerkki L1.9. instance.js-tiedoston sisältö

```
Oskari.clazz.define("Oskari.myframe.bundle.mysearchdraw.MyDrawBundleInstanc
e",

/**
 * @method create called automatically on construction
 * @static
 */
function() {
   this.sandbox = null;
```

```
    this.drawPlugin = null;
}, {
    /**
     * @static
     * @property __name
     */
    __name : 'MyDrawBundle',

    /**
     * @method getName
     * Module protocol method
     */
    getName : function() {
        return this.__name;
    },

    /**
     * @method update
     * BundleInstance protocol method
     */
    update : function() {
    },
    /**
     * @method start
     * BundleInstance protocol method
     */
    start : function() {
        var me = this;

        sandbox = Oskari.getSandbox();

        // register to sandbox as a module
        sandbox.register(me);
        // register to listening events
        for (var p in me.eventHandlers) {
            if (p) {
                sandbox.registerForEventByName(me, p);
            }
        }

        // get reference to mapmodule
        var mapModule = sandbox.findRegisteredModuleInstance('MainMapModule');

            // create drawplugin with conf (id should be used if we have multiple
drawplugins, defaults to "DrawPlugin")
        var pluginConfig = {
            id: 'Something',
            multipart: true
        };
        drawPlugin =
                    Oskari.clazz.create(
```

```
                        'Oskari.mapframework.ui.module.common.mapmodule.Draw
                Plugin', pluginConfig);
    // register drawplugin for map
    mapModule.registerPlugin(drawPlugin);
    mapModule.startPlugin(drawPlugin);

    drawPlugin.startDrawing({ drawMode : 'point' }); // or 'line' or 'area'
},

/**
 * @method init
 * Module protocol method
 */
init : function() {
    // headless module so nothing to return
    return null;
},

/**
 * @method onEvent
 * Module protocol method/Event dispatch
 */
onEvent : function(event) {
    var me = this;
    var handler = me.eventHandlers[event.getName()];
    if (!handler) {
        return;
    }

    return handler.apply(this, [event]);
},

/**
 * @static
 * @property eventHandlers
 * Best practices: defining which
 * events bundle is listening and how bundle reacts to them
 */
eventHandlers : {
    "DrawPlugin.FinishedDrawingEvent" : function(event) {
        // preferred to use sandbox.printDebug()
        console.log("User drew:", event.getDrawing());
    }
},

/**
 * @method stop
 * BundleInstance protocol method
 */
stop : function() {
    var me = this;
    var sandbox = me.sandbox();
```

```
        // unregister from listening events
        for (var p in me.eventHandlers) {
            if (p) {
                sandbox.unregisterFromEventByName(me, p);
            }
        }
        // unregister module from sandbox
        me.sandbox.unregister(me);
    }
}, {
    protocol : [ 'Oskari.bundle.BundleInstance' ]
});
```

Esimerkin tärkeimpiä metodeja ovat start ja onEvent sekä eventHandlers-rekisteröinti. start-metodissa otetaan käyttöön Oskarin sandbox, jonka avulla kaikki bundlet pyörivät ja jossa on kaikki yhteinen data. Tämän jälkeen sandboxin avulla rekisteröidään kaikki ne tapahtumat, joita bundlen on tarpeen kuunnella. Sandboxin kautta haetaan sitten tarpeellinen karttamoduuli, johon kaikki käytettävät liitännäiset on rekisteröitävä. Liitännäinen on myös konfiguroitava ennen käyttöönottoa, jonka jälkeen liitännäinen (*drawPlugin*) luodaan ja rekisteröidään kartan käyttöön. Lopuksi start-metodi käynnistää käyttöön otetun liitännäisen tietyllä piirtotilalla, joka on esimerkissä piste.

Kuten aluksi mainittiin, tärkeä ominaisuus on myös onEvent, joka pitää kirjaa kaikista rekisteröidyistä käsittelijöistä ja käsittelee tapahtumakohtaisesti kaikki käyttöliittymän kautta saapuvat tapahtumat. Tässä esimerkissä tapahtumien käsittelijöitä on vain yksi, "DrawPlugin.FinishedDrawingEvent", jolloin käyttöliittymä tulostaa piirron lopuksi piirretyn kuvan. Tämä tosin ei ole tarpeellinen, koska tapahtumaa ei koskaan käsitellä kun kyseessä on usean pisteen toteutus, eli pisteitä voidaan piirtää niin monta kuin halutaan. Jos usean pisteen toteutus haluttaisiin muuttaa yhden pisteen toteutukseksi, tulisi ohjelmasta piirtoliitännäisen konfiguraatiosta muuttaa “*multipart*”-muuttujan arvo falseksi. Huomaa myös, että stop-metodissa kaikki rekisteröidyt käsittelijät poistetaan sandboxista, jottei päällekkäisyyksiä tule.

8.1. Palvelualustan käyttö

Yksinkertaisin tapa luoda toimiva palvelualusta Oskaria varten on ladata tarvittava lähdekoodi suoraan gitistä ja ajaa palvelu paikallisella laitteella. Latauksen jälkeen kehittäjän tulee asettaa palvelualusta käyttämään jo ladattua käyttöliittymää muuttamalla käyttöliittymän kansion nimeksi Oskari. Tämän jälkeen kehittäjän ei tarvitse muuta kuin

asentaa palvelu pyörimään Mavenin avulla. Tarkat ohjeet ja komennot tämän suorittamiseen on löydettävissä Oskarin kotisivuilta: http://oskari.org/documentation/backend/server-embedded-developer. Palvelualusta ei itsessään tuo suurempia lisätoiminnallisuuksia käyttöliittymän toimintaan, mutta mahdollistaa Oskarin käyttämisen julkisena palveluna palvelimen kautta.

8.2. Yhteenveto

Pääasiallisesti Oskarin tavoitteena on luoda työkalu, jonka avulla on mahdollista esittää ja käyttää useista lähteistä saapuvaa avointa paikkatietoa visuaalisesti jokaiselle käyttäjälle. Paikkatiedon esitys ja analysointi selaimessa on hyvin toteutettu, mutta dynaamisen paikkatiedon käyttämiseen se ei sovellu yhtä hyvin, joten se soveltuu staattisten karttojen ja suurten datamäärien esittämiseen ja analysointiin.