Eini Saarivesi

Liitännäisarkkitehtuuri Knowledge Practices Environment -järjestelmään

Metropolia Ammattikorkeakoulu
Insinööri (AMK)
Mediatekniikan koulutusohjelma
Insinöörityö
6.6.2011

Tekijä Otsikko Sivumäärä Aika	Eini Saarivesi Liitännäisarkkitehtuuri Knowledge Practices Environment -järjestelmään 45 sivua + 1 liite 6.6.2011
Tutkinto	insinööri (AMK)
Koulutusohjelma	mediatekniikka
Suuntautumisvaihtoehto	digitaalinen media
Ohjaajat	projektipäällikkö Hannu Markkanen tutkijaopettaja Olli Alm

Insinöörityössä oli tavoitteena selvittää liitännäisten (plug-in) olemus ja liitännäisarkkitehtuurin (plug-in architecture) määritelmä sekä toteuttaa määritelmän mukainen liitännäisarkkitehtuuri Knowledge Practices Environment (KPE) -järjestelmään. KPE-järjestelmä on Knowledge Practices Laboratory (KP-Lab) -projektissa kehitetty oppimis- ja työskentely-ympäristö. KP-Lab-projekti oli vuonna 2006 aloitettu viisivuotinen EU-rahoitteinen projekti, joka keskittyi luomaan modulaarista, joustavaa ja laajennettavaa tietojärjestelmää. KPE-järjestelmään oli aikaisemmin toteutettu liitännäisarkkitehtuuri, mutta suurelta osin se ei vastannut liitännäisarkkitehtuurin vaatimuksia.

Insinöörityössä uudistettiin aikaisempi liitännäisarkkitehtuuri vastaamaan liitännäisarkkitehtuurin asettamia vaatimuksia. Arkkitehtuurin vaatimuksena oli mahdollistaa kolmansien osapuolien kehitystyö ja helpottaa uusien työkalujen ja tietosolmutyyppien kehitystä. Arkkitehtuuri toteutettiin tukemaan KPE-järjestelmän perustoiminnallisuuksia. Siksi olikin tärkeää kehittää se niin, että kehitystä voidaan helposti jatkaa ja tuki rajauksen ulkopuolelle jääneille toiminnallisuuksille voidaan toteuttaa helposti myöhemmin. Työssä uudistettiin aikaisemmat liitännäisarkkitehtuurin rajapinnat ja määriteltiin konfiguraatio liitännäisille ja KPE-järjestelmän tietosolmuille. Konfiguraation avulla liitännäiset pystytään lisäämään järjestelmään ajon aikana, ja ne voidaan käsitellä dynaamisesti. Lisäksi liitännäisten tietoa varten toteutettiin tietokanta ja web service -rajapinta.

Insinöörityön lopputuloksena syntyi liitännäisarkkitehtuuri, joka vastaa arkkitehtuurin ja projektin asettamia vaatimuksia. Arkkitehtuuri osoittautui järjestelmälle hyödylliseksi, ja se liitetään osaksi KPE-järjestelmää. Liitännäisten kehitystä varten kirjoitettiin englanninkielinen kehittäjänopas, jonka avulla järjestelmään on jo alettu kehittää uusia liitännäisiä. Arkkitehtuurin jatkokehitystä varten insinöörityössä selvitettiin myös muutamia seuraavia vaiheita arkkitehtuurin optimoimiseksi ja laajentamiseksi, kuten esimerkiksi rajapintojen versiointi ja liitännäisten jakelu.

Avainsanat	liitännäisarkkitehtuuri, liitännäinen, laajennettavuus, modulaarisuus

Author	Eini Saarivesi
Title	Plug-in architecture into Knowledge Practices Environment
Number of Pages	45 pages + 1 appendices
Date	6 June 2011
Degree	Bachelor of Engineering
Degree Programme	Media Technology
Specialisation option	Digital Media
Instructors	Hannu Markkanen, Project Manager Olli Alm, Researching Lecturer

The aim of this thesis was to clarify the essence of plug-in and the definition of the plug-in architecture, and implement plug-in architecture into Knowledge Practices Environment (KPE) system based on definition. KPE system is a learning and working environment and it was developed in Knowledge Practices Laboratory (KP-Lab) project. KP-Lab project was launched in 2006 and it was a five-year-long EU-funded project that focused on creating a modular, flexible and scalable information system. Plug-in architecture had already been developed into KPE system, but it did not meet the requirements of the plug-in architecture.

In this thesis the former architecture was revised to meet the requirements of the plug-in architecture. The requirements were to allow third-party development, and facilitate the development of new tools and data node types. The architecture was implemented to support the basic functionalities of KPE. It was, therefore, important to develop it in such a manner that further development can continue without difficulties. Also, it should be easy to implement later such architecture that was excluded during this project. In this work the earlier plug-in architecture interfaces were revised and the configuration for the plug-ins and KPE system data nodes was defined. Configuration allows adding plug-ins during the run-time and they can be processed dynamically. In addition, database and web service interface were implemented for the data of the plug-ins.

The result of this thesis is a plug-in architecture, which matches the requirements set for the architecture and the project. The architecture turned out to be useful, and it will be integrated into the KPE system. An English developer guide was written for plug-in development. With the help of the guide, the development of the plug-ins has already started. This thesis also presents ideas for further development of the architecture, such as the versioning of the plug-in interfaces, and plug-in distribution.

Keywords	plug-in architecture, plug-in, modularity, expandability

Sisällys

Lyhenteet, käsitteet ja määritelmät

Isäntäsovellus	Liitännäisarkkitehtuurin yhteydessä isäntäsovelluksella tarkoitetaan pääsovellusta, johon liitännäiset lisätään.
KPE-järjestelmä	*Knowledge Practices Environment* -järjestelmä on verkossa toimiva yhteisöllinen oppimis- ja työskentely-ympäristö.
Liitännäinen	Liitännäinen (plug-in) on ohjelmistokomponentti, jolla voidaan laajentaa isäntäsovelluksen toiminnallisuuksia. Liitännäiset eivät ole irrallisia sovelluksia, vaan ne aina vaativat isäntäsovelluksen toimiakseen. Isäntäsovellus tarjoaa liitännäisille rajapinnat, joiden avulla liitännäiset voivat keskustella ja toimia isäntäsovelluksen kanssa.
Liitännäisarkkitehtuuri	Liitännäisarkkitehtuuri (plug-in architecture) koostuu isäntäsovelluksesta ja isäntäsovellukseen liitettävistä liitännäisistä. Liitännäisarkkitehtuuri mahdollistaa sovelluksen laajennettavuuden.
RSLs	*Runtime Shared Libraries.* RSLs-kirjastot ovat Adobe Flex -ohjelman kirjastoja, jotka voidaan ladata ja tallentaa käyttäjän selaimen välimuistiin. RSLs-kirjastot ladataan vain kerran, ja monet eri sovellukset voivat käyttää niitä.
Tietosolmu	Tietosolmulla (node) tarkoitetaan kaikkia KPE-järjestelmässä olevia tieto-objekteja. Tietosolmuilla on eri tyyppejä, jotka sisältävät eri sisältöä. Tietosolmutyyppejä ovat muun muassa Shared Space, Task, Note ja Web link.

1 Johdanto

Tekniikka kehittyy nopeasti, ja nykypäivän tietotekniikan ohjelmistoilta vaaditaan joustavuutta, nopeaa muunneltavuutta ja laajennettavuutta. Usein halutaan, että ne pystyvät omaksumaan uusia toiminnallisuuksia helposti ja nopeasti. Modulaarisuus ja komponentti- ja liitännäisarkkitehtuuri (plug-in architecture) mahdollistavat ohjelmiston nopean muunneltavuuden, tosin kattavan liitännäisarkkitehtuurin muodostaminen saattaa olla hyvin kallista ja aikaavievää.

Useat ohjelmistot hyödyntävät liitännäisarkkitehtuurin tarjoamia etuja. Sen avulla sovellukset voidaan pilkkoa pienempiin osiin ja niiden toiminnallisuudet voidaan toteuttaa liitännäisinä (plug-in). Liitännäisarkkitehtuurista hyötyvät niin loppukäyttäjät kuin kehittäjätkin. Usein loppukäyttäjille annetaan mahdollisuus räätälöidä sovellusta mielensä mukaiseksi. Esimerkiksi Mozilla Firefox -selain mahdollistaa erilaisten liitännäisten lisäämisen, ja näin käyttäjät voivat helposti parantaa selaimen käyttömukavuutta. Lisäksi liitännäisarkkitehtuuri antaa kenelle tahansa mahdollisuuden kehittää sovellukseen uusia toiminnallisuuksia. Tämä tarjoaa sovelluksen valmistajalle suuren edun, sillä sovellukseen saadaan laajennuksia hyvin pienillä resursseilla.

Insinöörityössä tutkitaan liitännäisiä ja liitännäisarkkitehtuureja ja tehdään selvitys niiden määritelmistä ja vaatimuksista. Selvitystyön perusteella määritellään Knowledge Practices Environment (KPE) -järjestelmän liitännäisarkkitehtuurin vaatimukset ja uudistetaan sen aikaisempi liitännäisarkkitehtuuri vastaamaan määrittelyjä. Insinöörityö tehdään osana Knowledge Practices Laboratory (KP-Lab) -projektia, joka on vuonna 2006 alkanut viisivuotinen EU-rahoitteinen projekti. Sen yhtenä teknisenä tavoitteena on suunnitella ja toteuttaa modulaarinen, joustava ja laajennettava tietojärjestelmä. KP-Lab-projekti keskittyy luomaan opetusjärjestelmää, joka helpottaa innovatiivisten käytäntöjen luomista ja jakamista sekä työskentelyä tietotaidon kanssa koulussa ja työpaikoilla. KPE-järjestelmä on KP-Lab-projektissa kehitetty verkossa toimiva virtuaalinen opetusjärjestelmä, joka on Adobe Flexillä toteutettu RIA (Rich Internet Applications) -sovellus. Järjestelmä tukee yhteisöllistä ja samanaikaista työskentelyä. Se sisäl-

tää monia eri työkaluja ja toiminnallisuuksia, joiden avulla voidaan käsitellä ja analysoida järjestelmässä olevaa sisältöä.

Insinöörityön tavoitteena on toteuttaa liitännäisarkkitehtuuri, jolla saadaan KP-Lab-projektin tekninen tavoite paremmin toteutettua, ja lisäksi halutaan helpottaa uusien työkalujen ja toiminnallisuuksien kehitystä. Aikataulun puitteissa ei ole mahdollista toteuttaa liitännäisarkkitehtuuria tukemaan kaikkia KPE-järjestelmän toiminnallisuuksia. Arkkitehtuuri tulee toteuttaa niin, että tuki rajauksen ulkopuolelle jääville toiminnallisuuksille voidaan myöhemmin helposti toteuttaa. Työhön kuuluu myös englanninkielisen liitännäisten kehittäjänoppaan kirjoittaminen, jonka tarkoituksena on helpottaa liitännäisten kehitystyötä.

2 Liitännäisarkkitehtuuri

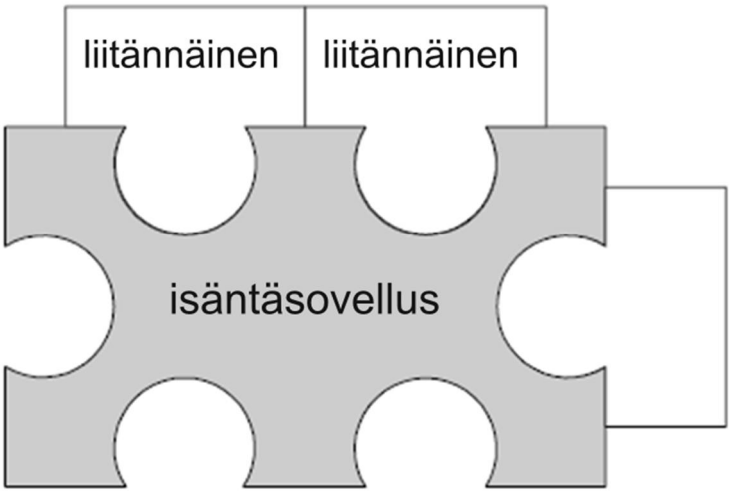
2.1 Liitännäisarkkitehtuurin perusajatus

Liitännäisarkkitehtuuri tarjoaa kehittäjille tavat toteuttaa sovelluksia, jotka ovat helposti laajennettavissa ja muokattavassa uusiin käyttötilanteisiin. Liitännäisarkkitehtuurin perusajatuksena on tarjota isäntäsovellus, jota voidaan helposti laajentaa liitännäisten avulla. [1, s. 21.]

Isäntäsovellusta laajennetaan liitännäisillä, jotka liitetään siihen joko ajon aikana tai sovelluksen käynnistyksen yhteydessä. Liitännäisarkkitehtuurin peruspiirteenä on, että isäntäsovellukseen voidaan lisätä toiminnallisuuksia ja ominaisuuksia ilman, että isäntäsovellusta tarvitsee muuttaa. Näin ollen, kun uusi liitännäinen lisätään isäntäsovellukseen, ei sovellusta tarvitse kääntää uudelleen. Isäntäsovellus ei myöskään saisi tietää mitään liitännäisten toiminnallisuuksista, ennen kuin liitännäinen ladataan isäntäsovellukseen. Hyvin toteutetussa liitännäisarkkitehtuurissa isäntäsovellus ei sisällä mitään tiettyyn liitännäiseen liittyvää tietoa. Isäntäsovelluksen täytyy tarjota rajapinnat liitännäisille, jotta kommunikointi ja työskentely liitännäisten ja isäntäsovelluksen välillä onnistuu. [2, s. 49; 3, s. 1324; 4, s. 88, s. 92.]

Liitännäinen on itsenäinen ohjelmistokomponentti, joka liitetään isäntäsovellukseen. Jos liitännäiseen tehdään muutos, tarvitsee vain liitännäinen kääntää uudelleen. Liitännäiset kehitetään erillään isäntäsovelluksesta, mutta ne eivät kuitenkaan ole irrallisia sovelluksia. Liitännäiset ovat riippuvaisia isäntäsovelluksesta, eivätkä ne voi toimia ilman sitä. Kehittäjän näkökulmasta voi olla melko haastavaa, että liitännäiset kehitetään erillään isäntäsovelluksesta. Kehitystyö helpottuu huomattavasti, jos pystytään esimerkiksi lokaalisti ajamaan ja testaamaan liitännäisten toimivuutta isäntäsovelluksessa. Isäntäsovelluksen rajapinnat määrittelevät, miten liitännäinen voi toimia isäntäsovelluksen ja muiden liitännäisten kanssa. Rajapinnat myös määrittelevät, mitä tietoa liitännäiset voivat saada isäntäsovellukselta. Liitännäiset tarjoavat isäntäsovellukselle konfiguraation, jonka avulla isäntäsovellus osaa liittää liitännäiset osaksi järjestelmää. [2, s. 21; 4, s. 88, s. 92; 5, s. 301–303.]

Liitännäisarkkitehtuuri voidaan ajatella palapelinä: isäntäsovellus sisältää koloja, joihin liitännäiset voidaan lisätä, jos ne ovat oikean muotoisia ja kokoisia [2, s. 50–51]. Kuviossa 1 havainnollistetaan liitännäisarkkitehtuuria palapelinä.

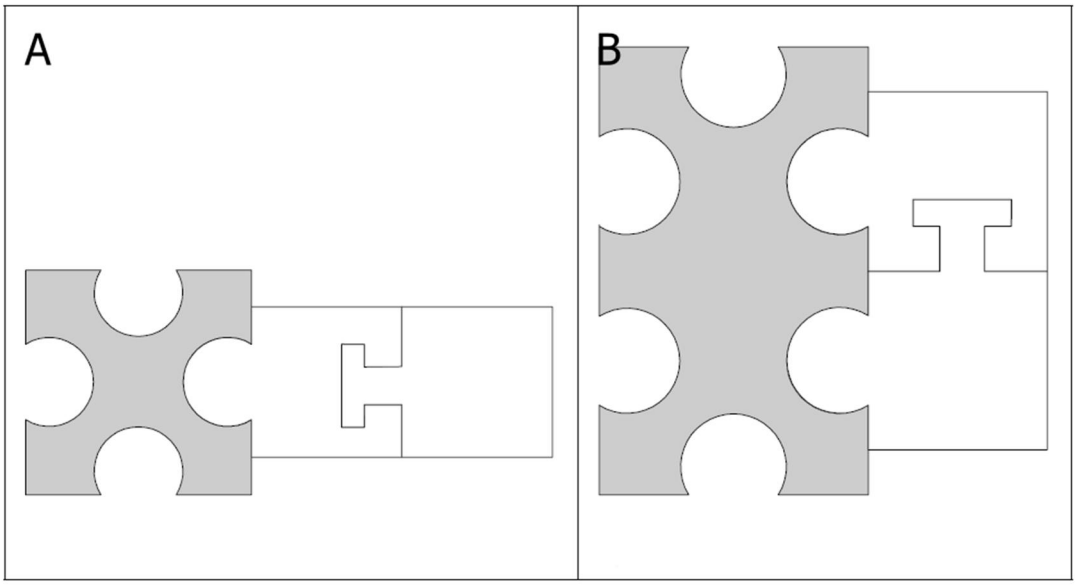


Kuvio 1. Liitännäisarkkitehtuuri voidaan havainnollistaa palapelinä [2, s. 50–51].

Kuviossa 1 harmaa pala kuvaa isäntäsovellusta ja valkoiset palat liitännäisiä. Isäntäsovelluksen palassa olevat kolot kuvaavat sen määrittelemiä rajapintoja, ja liitännäisten

paloissa oleva uloke kuvaa liitännäisen luokkia, jotka toteuttavat vaaditut rajapinnat. Jos palapelin palaset sopivat yhteen, liitännäinen toteuttaa rajapinnat isäntäsovelluksen haluamalla tavalla ja liitännäinen voidaan liittää sovellukseen. [2, s. 50–51.]

Riippuen siitä, miten liitännäisarkkitehtuuri on toteutettu, voi olla mahdollista, että liitännäiset sisältävät koloja, joihin voidaan liittää muita liitännäisiä. Tämä mahdollistaa sen, että liitännäiset voivat hyödyntää toisten liitännäisten toiminnallisuuksia ja ominaisuuksia. [2, s. 50–51.] Kuviossa 2 on esitetty liitännäisarkkitehtuurin vaihtoehtoisia malleja liitännäisten suhteiden määrittelyyn.



Kuvio 2. Liitännäisarkkitehtuurin vaihtoehtoisia malleja liitännäisten suhteiden määrittelyyn [2, s. 50–51].

Kuvion 2 A-kohdassa liitännäinen voi laajentaa toista liitännäistä laajentamatta isäntäsovelluksen toiminnallisuuksia ja B-kohdassa liitännäiset voivat laajentaa samanaikaisesti isäntäsovellusta sekä toista liitännäistä. Jos arkkitehtuuri sallii liitännäisten välisen laajentamisen, se yleensä toteutetaan käyttämällä isäntäsovelluksen määrittelemillä rajapinnoilla. Liitännäiset eivät sisällä rajapintoja toisia liitännäisiä varten, vaan liitännäinen kertoo isäntäsovellukselle, miten sitä voidaan laajentaa. Isäntäsovellus toteuttaa liitännäisten väliset laajennukset määrittelyiden perusteella. Näin isäntäsovellus pystyy hallitsemaan kaikkia liitännäisiä ja se pystyy tunnistamaan kaikki sovellukseen

lisätyt liitännäiset. Isäntäsovellus pystyy esimerkiksi ilmoittamaan käyttäjälle, jos käyttäjän aktivoima liitännäinen vaatii toisen liitännäisen aktivoimisen. [2, s. 50–51.]

2.2 Liitännäisarkkitehtuurin hyödyt

Liitännäisarkkitehtuuri on hyvä ratkaisu kevyemmän ohjelmistorakenteen toteutukseen. Sovelluksen laajennettavuus ja modulaarisuus on tärkeää nykypäivän ohjelmistokehityksessä, sillä ei voida tietää eikä ennustaa kaikkia järjestelmän tulevia vaatimuksia sovelluksen kehityksen alkuvaiheessa. Hyvä esimerkki aiheeseen liittyen on Internet-selaimet. Uusia tiedostoformaatteja kehitetään jatkuvasti, ja käyttäjät haluavat käyttää niitä selaimella. Jotta selain pystyy näyttämään uudet tiedostoformaatit, täytyy siihen lisätä toiminnallisuus niiden esittämistä varten. Selaimen kehitysvaiheessa ei ole mahdollista tietää tulevia tiedostomuotoja, ja välttämättä ei ole kovin mielekästä julkaista selaimesta kokonaan uutta versiota aina, kun uuden tiedostoformaatin tuki halutaan lisätä selaimeen. Toteuttamalla liitännäisarkkitehtuuri selaimeen tuki uusiin tiedostoformaatteihin voidaan lisätä portaittain. [2, s. 50–51; 6, s. 1–2; 7, s. 217–218.]

Usein liitännäisarkkitehtuuria hyödyntävät sovellukset antavat loppukäyttäjälle mahdollisuuden vaikuttaa järjestelmän toimintaan ja toiminnallisuuksiin. Käyttäjät voivat aktivoida ja poistaa liitännäisiä käytöstä, ja näin käyttäjät pystyvät muokkaamaan sovelluksen mieleisekseen. Myös sovelluksen räätälöiminen eri käyttötarkoituksiin helpottuu, sillä sovellus voidaan toimittaa asiakkaille täysin riisuttuna. Sovelluksen käyttöarvo (use value) voidaan lisätä jälkikäteen liitännäisinä. Lisäksi sovelluksen päivittäminen saattaa helpottua, sillä päivitykset voidaan tarjota asiakkaille liitännäisinä. [1, s. 21; 5, s. 301–302; 7, s. 217–218; 8, s. 287.]

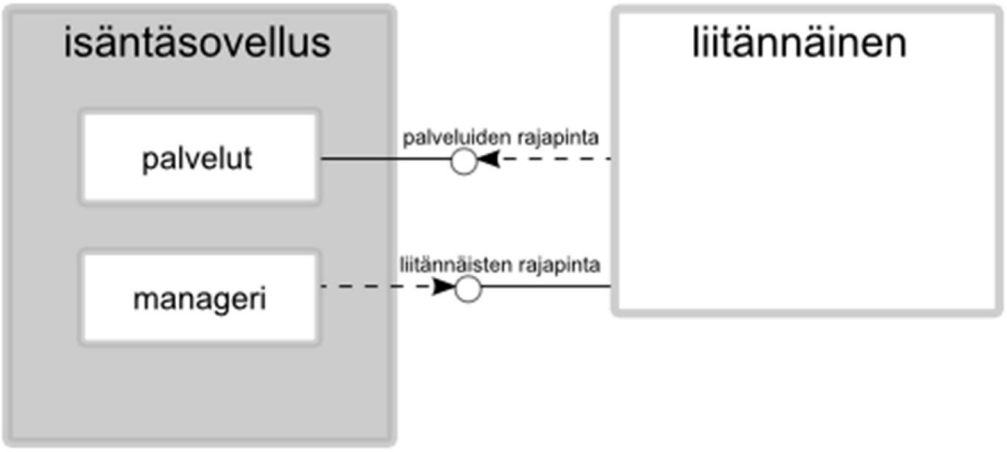
Kolmansien osapuolien kehitys pystytään mahdollistamaan liitännäisarkkitehtuurin avulla. Kuten jo aikaisemmin on mainittu, liitännäisarkkitehtuuri tarjoaa liitännäisille hyvin tarkasti suunnitellut ja rajatut rajapinnat. Tämä yleensä johtaa siihen, että liitännäisten rajapinnat on hyvin dokumentoitu, mikä puolestaan helpottaa liitännäisten kehittämistä. [2, s. 49; 7, s. 218; 8, s. 288.]

Kun liitännäisarkkitehtuuri toteutetaan, voidaan isäntäsovelluksen kokoa pienentää toteuttamalla sovelluksen toiminnallisuuksia liitännäisinä. Isäntäsovelluksen pienentyminen lyhentää sovelluksen käynnistymisaikaa ja sovelluksen levityspaketin koko pienenee. Ajatuksena on, että isäntäsovellus sisältää ainoastaan sovelluksen toiminnan kannalta tärkeimmät toiminnallisuudet. Muut valinnaiset toiminnallisuudet on toteutettu liitännäisinä, jotka voidaan tarvittaessa lisätä järjestelmään tai poistaa siitä. [2, s. 49; 5, s. 303.]

Liitännäisarkkitehtuuriin liittyy myös haittapuolia. Liitännäisiä varten täytyy määritellä rajapinnat etukäteen, joten järjestelmään liittyvät vaatimukset ja rajoitukset täytyy suunnitella ja määritellä tarkasti. Hyvän liitännäisarkkitehtuurin suunnitteluun ja määrittelyyn joudutaan käyttämään paljon aikaa, ja näin ollen suunnittelu ja määrittely voi olla kallista. [5, s. 301.]

2.3 Liitännäisarkkitehtuurin rakenne

Isäntäsovellus tarjoaa rajapinnat, joihin liitännäinen kytketään. Rajapintojen avulla isäntäsovelluksen ja liitännäisen välinen kommunikaatio on kaksisuuntaista; Kun liitännäinen kytketään isäntäsovellukseen, isäntäsovellus pyytää liitännäistä esittelemään itsensä. Koska liitännäinen ei ole itsenäinen sovellus, kaikki pyynnöt tulevat isännältä liitännäisille. Kun liitännäinen ladataan isäntäsovellukseen, isäntäsovellus kysyy liitännäiseltä esimerkiksi liitännäisen nimen ja konfiguraation. [4, s. 89–92.] Kuviossa 3 on esitetty liitännäisarkkitehtuurin isäntäsovelluksen ja liitännäisten väliset rajapinnat.



Kuvio 3. Isäntäsovelluksen ja liitännäisen väliset rajapinnat [4, s. 92].

Isäntäsovelluksessa oleva manageri on vastuussa liitännäisten käsittelystä. Manageri on tietoinen liitännäisten tiloista, ja se on vastuussa liitännäisten lataamisesta ja poistamisesta, kun se on tarpeellista. Manageri pystyy kysymään tarvittavat tiedot liitännäiseltä kuviossa 3 esitetyn *liitännäisten rajapinnan* avulla. *Palveluiden rajapinta* puolestaan tarjoaa liitännäisille tietoa isäntäsovelluksesta ja mahdollisuuden käyttää isäntäsovelluksen toiminnallisuuksia. Palveluiden rajapinta on turvallisuussyiden vuoksi usein melko rajoitettu, sillä se tarjoaa kolmansien osapuolien kehittäjille tietoa isäntäsovelluksesta. Siksi halutaankin olla varmoja, ettei mihinkään arkaluontoiseen tietoon päästä käsiksi. Usein liitännäiset saavat yhteyden palveluiden rajapintaan liitännäisten rajapinnan avulla. Näin pystytään varmistamaan, että vain isäntäsovelluksen tunnistamat liitännäiset pääsevät käsiksi isäntäsovelluksen tietoon ja toiminnallisuuksiin. [4, s. 89–92.]

Tietyt ohjelmointikielet tarjoavat valmiita työkaluja liitännäisarkkitehtuurin kehittämiseen. Työkalut sisältävät valmiita luokkia ja kirjastoja, joiden avulla pystytään kehittämään arkkitehtuuri, joka tukee liitännäisten hallintaa ja käsittelyä. Esimerkiksi Java-ohjelmointikieli tarjoaa Java plug-in framework (JPF) -ohjelmistokehyksen liitännäisten kehitystä varten ja Flex-ohjelma tarjoaa module- ja sub-application-tekniikat irrallisten ohjelmistokomponenttien dynaamiseen lataamiseen. [9; 10, s. 985; 11, s. 1.]

Liitännäisarkkitehtuurin kehittäjät ovat vastuussa rajapintojen määrittelystä isäntäsovelluksen ja liitännäisten välille. Näin ollen kuviossa 3 esitettyä rajapintaratkaisua ei voida pitää ainoana oikeana ratkaisuna rajapintojen toteuttamiseksi. Todellisuudessa isäntäsovelluksen ja liitännäisen välisiä rajapintoja saattaa olla enemmän tai jopa vähemmän. Yleisenä piirteenä liitännäisarkkitehtuureille kuitenkin on, että isäntäsovellus määrittelee ainakin yhden rajapinnan, jonka avulla isäntäsovellus tunnistaa liitännäisen ja pystyy liittämään liitännäisen järjestelmään. Jos halutaan, että liitännäinen pystyy hyödyntämään isäntäsovelluksen toiminnallisuuksia ja saamaan isäntäsovelluksesta tietoa, täytyy isäntäsovelluksen määritellä rajapinta sitä varten. [4, s. 92.]

2.4 Liitännäisarkkitehtuurin erityisongelmat

Ajatuksen tasolla liitännäisarkkitehtuurin perusrakenne on varsin yksinkertainen, mutta käytännössä arkkitehtuurin toteuttaminen on vaikeaa ja aikaavievää. Liitännäisarkkitehtuuriin liittyy tiettyjä ongelmia, jotka täytyy ratkaista liitännäisarkkitehtuuria toteutettaessa.

Isäntäsovelluksen ja liitännäisen välisen s*opimuksen määrittely* eli rajapintojen määrittely*.* Isäntäsovellus määrittelee sopimuksen eli miten sitä voidaan laajentaa ja mitä sen sisältämiä toiminnallisuuksia liitännäiset voivat käyttää. Liitännäisten kehittäjät luottavat isäntäsovelluksen määrittelemään sopimukseen ja toteuttavat liitännäisen sovitulla tavalla. Rajapintojen yhtenä ominaispiirteenä on, että niiden toteuttajan pitää toteuttaa rajapinta täysin samalla tavalla kuin rajapinta on määritelty. Tämän vuoksi liitännäisarkkitehtuurin rajapintojen täytyy olla muuttumattomat. Hyvin todennäköistä on, että rajapintoja kuitenkin joudutaan jossakin vaiheessa muuttamaan, joten hyvin toteutettu liitännäisarkkitehtuuri pystyy mahdollistamaan rajapintojen muuttamisen rajapintojen versioinnin avulla. [7, s. 218–219.]

Rajapintojen versiointi. Kun isäntäsovelluksesta julkaistaan uusi versio, on toivottavaa, että vanhat liitännäiset toimivat uudessa versiossa. Versioinnissa täytyy nimenomaan ottaa huomioon liitännäisarkkitehtuurin rajapinnat. Vanhalla rajapinnalla määritelty liitännäinen ei voi toimia uudella rajapinnalla, sillä vanha rajapinta ei täytä uuden rajapinnan vaatimuksia. Näin ollen jos halutaan mahdollistaa vanhojen liitännäisten toimi-

vuus uudessa rajapinnassa, täytyy rajapinnat versioida. Rajapintojen tulisi olla ainakin taaksepäin yhteensopivia, eli vanhaa rajapintaa käyttävän liitännäisen tulisi toimia uudessa rajapinnassa. Mutta voidaan myös haluta, että rajapinnat ovat eteenpäin yhteensopivia eli uudella rajapinnalla toteutetun liitännäisen tulisi myös toimia järjestelmässä, joka käyttää vanhaa rajapintaa. [7, s. 218–219.]

Liitännäisten kehittäminen ja jakaminen. Liitännäisten kehittäminen ja paketoiminen vaihtelee ohjelmointikielestä riippuen. Liitännäiset voivat olla esimerkiksi JAR (Java Archive), DLL (Dynamic Link Library) -paketteja tai SWF (ShockWave Flash) -tiedostoja. Liitännäisten jakelun voi puolestaan toteuttaa keskitetysti tai villisti. Keskitetyssä jakelussa liitännäiset kootaan yhteen paikkaan, mistä käyttäjät voivat ne hakea. Esimerkiksi Mozilla Firefox -selain jakaa liitännäisiä Firefox Addons -sivustolla ja Eclipse-ohjelmointiympäristö jakaa liitännäisiä Eclipse Marketplace -sivustolla. Liitännäisten villi jakelu on keskitetyn jakelun vastakohta, eli liitännäisten kehittäjät jakavat liitännäisiä esimerkiksi omilta sivuiltaan. Keskitetty jakelu näyttää olevan suosituin vaihtoehto liitännäisten jakeluun, mutta villiä jakelua käytettään myös. Jakelutavat eivät kuitenkaan poissulje toisiaan; keskitetyn jakelun lisäksi voidaan liitännäisiä jakaa myös villisti. Esimerkiksi Eclipse-ohjelmointiympäristö liitännäisten kehittäjät jakavat hyvin paljon liitännäisiä omilta sivuiltaan Eclipse Marketplace -sivuston ohella. [7, s. 218–219.]

Liitännäisten laadun hyväksyminen. Isäntäsovelluksen täytyy olla varma, että liitännäinen on toteutettu oikein, jotta se voidaan lisätä isäntäsovellukseen. Liitännäinen ei saa aiheuttaa järjestelmälle virhetiloja tai kaataa järjestelmää. Liitännäiset ovat osa isäntäsovellusta, joten ne myös vaikuttavat siihen, onko isäntäsovellus hyvä vai huono. Eclipse-ohjelmointiympäristö tarjoaa liitännäisten kehittämistä varten kehitystyökalun, jonka avulla liitännäisten kehittäjät voivat myös testata liitännäisen toimivuutta kehitystyön aikana. Näin pystytään vähentämään huonolaatuisten ja virheellisten liitännäisten kehitystä. Liitännäisten laadun hyväksymiseen on useita eri vaihtoehtoja. Jos liitännäiset jaellaan keskitetysti, voidaan jakelusivulle toteuttaa esimerkiksi sivuston ylläpidon puolesta liitännäisen hyväksyminen, ennen kuin se hyväksytään jaeltavaksi. Toinen vaihtoehto on antaa käyttäjien arvostella ja kommentoida liitännäisten toimivuutta ja näin selvittää niiden toimivuus. Jos järjestelmään voi lisätä liitännäisiä ainoastaan järjestel-

män ylläpitäjä, voidaan vaatia, että ylläpitäjä tarkistaa liitännäisen toimivuuden, ennen kuin se lisätään järjestelmään. [7, s. 218–219; 12, s. 112–127.]

Liitännäisten turvallisuus. Usein yksi liitännäisarkkitehtuurin toteuttamisen tärkeimmistä syistä on kolmansien osapuolien kehityksen mahdollistaminen. Tällöin on syytä rajata, mitä tietoa liitännäinen saa isäntäsovelluksessa ja mitä se saa tehdä isäntäsovelluksessa. Vastuu liitännäisten turvallisuudesta tai ainakin turvallisuuden tarkistamisesta on isäntäsovelluksella. Tietoturva on liitännäisarkkitehtuurin kehittämisen kannalta suuri riski, sillä jos isäntäsovellus liitännäisten takia menee rikki tai siinä on liitännäisten takia tietoturvavuotoja, se voi olla haitallista sovellukselle, yrityksen brändille ja maineelle sekä varsinaisille käyttäjillekin. Liitännäisten turvallisuus liittyy vahvasti niiden laadun hyväksymiseen, joten liitännäisen laatua hyväksyttäessä tulisi myös tarkistaa sen turvallisuus. [7, s. 218–219.]

Riippuen käytetystä ohjelmointikielestä myös seuraavat asiat saattavat asettaa haasteita liitännäisarkkitehtuurin toteuttamiselle. *Liitännäisen lisääminen* ja *tunnistaminen*: miten liitännäinen lisätään järjestelmään ja kuka liitännäisen voi lisätä. Esimerkiksi voivatko käyttäjät lisätä liitännäisiä järjestelmään vai voiko lisäyksen tehdä vain järjestelmän ylläpitäjä. *Liitännäisen liittäminen* osaksi järjestelmää: miten isäntäsovellus liittää liitännäiset järjestelmään ja tehdäänkö liittäminen ajon aikana vai käynnistyksen yhteydessä. *Liitännäisten aktivoiminen ja poistaminen käytöstä*: voivatko käyttäjät itse räätälöidä sovellusta. [7, s. 218–219.]

2.5 Sovellukset

Useat Internet-selaimet tukevat liitännäisiä. Esimerkiksi Mozilla antaa kehittäjille mahdollisuuden luoda uusia liitännäisiä muun muassa Mozilla Firefox -selaimeen. Mozilla määrittelee sovelluksen käyttöliittymän XML (eXtensible Markup Language) -merkintäkielen avulla, ja näin liitännäiset pystyvät määrittelemään omat lisäykset käyttöliittymään XUL (XML User Interface Language) -määrittelykielen avulla. Mozilla lisää liitännäiset JavaScript-komentosarjakielen ja XPCOM (Cross Platform Component Object Model) -komponenttimallin avulla dynaamisesti osaksi sovellusta. Mozilla ei rajoita liitännäisten lisäämistä pelkästään selaimiin vaan sen XPCOM-komponenttimalli mahdol-

listaa liitännäisten lisäämisen myös esimerkiksi Mozillan sähköpostiohjelmaan Thunderbirdiin ja Sunbird-kalenterisovellukseen. [8, s. 289.]

Eclipse-ohjelmointiympäristö on alun perin kehitetty Java-pohjaisten ohjelmistojen kehittämiseen. Eclipse-ohjelmointiympäristöä voidaan kuitenkin laajentaa tukemaan uusia ohjelmointikieliä liitännäisillä. Eclipse hallinnoi liitännäisiä OSGi (Open Service Gateway Initiative) -rajapinnan avulla. OSGi on Java-pohjainen rajapinta, joka mahdollistaa liitännäisten hallinnoimisen ja liittämisen sovellukseen. Eclipsen liitännäiset kirjoitetaan Java-ohjelmointikielellä, ja ne jaellaan JAR-paketteina. Liitännäiset määrittelevät liitännäisen yhteyden järjestelmään ja muiden liitännäisten välille XML-tiedoston avulla. Eclipse ei pysty käsittelemään liitännäisten lisäystä ajonaikaisesti, vaan uuden liitännäisen lisäys vaatii aina sovelluksen uudelleen käynnistämisen. [2, s. 7; 6 s. 9; 8, s. 289.]

NetBeans-ohjelmointiympäristöön voi Eclipse-ohjelmistoympäristön tavoin lisätä liitännäisiä. NetBeansin laajennusmekanismi ja liitännäisten yhdistäminen osaksi järjestelmää toteutetaan virtuaalisella tiedostojärjestelmällä (virtual file system) ja liitännäiset määritellään XML-tiedoston avulla. Kuten Eclipsen, myös NetBeansin liitännäiset ovat JAR-paketteja. [8, s. 289–290.]

Adobe mahdollistaa liitännäisten liittämisen sen omiin sovelluksiin, esimerkiksi Photoshopiin, Illustratoriin ja Acrobat Suiteen. Adobe Photoshopiin voidaan esimerkiksi lisätä uusia suodattimia kuvien käsittelyä varten. Liitännäiset käsitellään adapterin avulla, joka myös mahdollistaa liitännäisen ja sovelluksen välisen kommunikoinnin. Liitännäiset määritellään PiPL (Plugin Property List) -tietomallin avulla. PiPL-tietomalli sisältää liitännäisen metatiedon, ja se kertoo adapterille, kuinka liitännäinen tulisi käsitellä. Kuten Mozilla, myös Adobe tukee liitännäisten kehitystä monissa eri sovelluksissa. Adobe on kehittänyt PICA (Plug-in Component Architecture) -rajapinnan pystyäkseen toteuttamaan liitännäisiä alustariippumattomasti omissa sovelluksissaan. [4, s. 98; 5, s. 304.]

2.6 Liitännäisarkkitehtuurin toteuttaminen Adobe Flexissä

Insinöörityössä toteutettiin liitännäisarkkitehtuuri KPE-järjestelmään, joka on Flexillä toteutettu RIA (Rich Internet Applications) -sovellus. Tämän vuoksi työssä vertailtiin Flexin tarjoamia vaihtoehtoja liitännäisarkkitehtuurin toteuttamiseksi. Flex tarjoaa kaksi tapaa ladata sovellukseen dynaamisesti ohjelmistokomponentteja: module ja sub-application. Seuraavassa esitellään Flexin tarjoamat tekniikat ja pohditaan, miten ne sopivat liitännäisarkkitehtuurin asettamiin määrityksiin.

Module

Module-komponenttien avulla Flex tarjoaa tavan irrottaa Flexin pääsovelluksen toiminnallisuuksia eri tiedostoihin, ja näin pääsovelluksesta voidaan luoda modulaarisempi. Module-komponentit ovat SWF-tiedostoja eli flash-tiedostoja, jotka ladataan dynaamisesti pääsovellukseen. Ne ovat tiukasti sidottuja pääsovellukseen, eivätkä ne voi toimia ilman sitä. Pääsovelluksen ei tarvitse ladata module-komponentteja sovelluksen käynnistyessä, vaan komponentit voidaan tarpeen mukaan ladata ja poistaa ajon aikana. Module-komponentit ovat täysin riippuvaisia pääsovelluksesta, vaikka ne kehitetään siitä erillään. Näin ollen kun module-komponenttiin tehdään muutos riittää, kun pelkästään module käännetään uudelleen. [10, s. 985–986.]

Module-komponenttien käyttö lyhentää pääsovelluksen käynnistysaikaa, sillä ne ladataan vasta, kun niitä tarvitaan. Module-komponentit myös käyttävät pääsovelluksen perusluokkia (manager luokkia), joten niillä saadaan pienennettyä modulen ja pääsovelluksen tiedostokokoa. [10, s. 985–986.]

Module voi olla visuaalinen komponentti tai pelkästään toiminnallinen tai prosessoiva komponentti. Tämä mahdollistaa sen, että module-komponenttien ei tarvitse olla pelkästään visuaalisia, vaan ne voivat laajentaa isäntäsovellusta pelkästään uusilla toiminnallisuuksilla. Module-komponentin rakenne mahdollistaa sen, että pääsovellus voi luoda modulen luokka-instanssin ilman, että luokkien toteutukset täytyisi liittää pääsovellukseen sovelluksen kääntämisen yhteydessä. [10, s. 986; 11, s. 13–14.]

Module-komponentit vastaavat luvussa 2.1 esiteltyä liitännäisten perusrakennetta: Ne ladataan dynaamisesti ajon aikana sovellukseen, ja ne ovat täysin riippuvaisia pääsovelluksesta. Ne myös kehitetään erillään pääsovelluksesta, ja ne keskustelevat ModuleManagerin kanssa rajapintojen avulla.

Sub-application

Sub-application-komponentit ovat pääsovellukseen ladattavia niin sanottuja alisovelluksia. Sub-application-komponentit ovat module-komponenttien tavoin SWF-tiedostoja, jotka ladataan ja poistetaan dynaamisesti ajon aikana, ja ne kehitetään erillään pääsovelluksesta. Sub-application-komponentit eroavat module-komponenteista siinä, että ne voivat toimia myös itsenäisinä sovelluksina. Lisäksi ne voidaan ladata useaan eri sovellukseen riippumatta siitä, miten pääsovellus on määritelty. [11, s. 1, s. 13–14.]

Myös sub-application-komponentit pienentävät pääsovelluksen kokoa, koska osa pääsovelluksen toiminnallisuuksista on kirjoitettu sub-application-komponenttiin. Ne ovat kuitenkin tiedostokokonsa puolesta suurempia kuin module-komponentit, sillä ne eivät pysty käyttämään pääsovelluksen luokkia hyväkseen samalla tavalla kuin module-komponentit, vaan ne sisältävät samat luokat, jotka itse pääsovellus sisältää. [11, s. 1, s. 13–14.]

Sub-application-komponentit ladataan SWFLoaderilla pääsovellukseen. Se, mitä parametrejä SWFloaderiin asetetaan, ratkaisee pääsovelluksen ja sub-application-komponenttien yhteensoveltuvuuden. SWFloaderin trustContent-ominaisuus määrittelee, onko sovellus luotettu vai ei. *Luotetulla sovelluksella* on parempi yhteensopivuus pääsovelluksen kanssa kuin luottamattomilla sovelluksilla. SWFloaderin loadForCompatibility-ominaisuus mahdollistaa sub-application-komponenttien *versioinnin*. Näin Flex tarjoaa tavan ladata sovelluksia, jotka on käännetty eri Flexin SDK (software development kit) -versiolla. Versioinnin etuna on, että se lisää sovelluksen joustavuutta. [11, s. 13–14.]

Liitännäisarkkitehtuurin kannalta sub-application-komponentit eivät ole niin hyvä vaihtoehto kuin module-komponentit, vaikkakin niillä on samoja piirteitä kuin luvussa 2.1 määritellyillä liitännäisillä: dynaaminen lataus ja kehitys erillään pääsovelluksesta. Lii-

tännäisarkkitehtuurin kannalta huonoja piirteitä on sub-application-komponenttien itsenäisyys, sillä ne voivat toimia ilman pääsovellusta, ja pääsovelluksen ja sub-applicationin välisen kommunikoinnin rajoitteet. Toisaalta module-komponentteihin verrattuna sub-application-komponenttien luotettavuus ja versiointi voidaan määritellä paremmin, ja nämä ovat tärkeitä asioita liitännäisarkkitehtuurissa. Olemukseltaan sub-application-komponentit sopivat paremmin komponenttiarkkitehtuurisovelluksiin kuin liitännäisarkkitehtuurisovelluksiin.

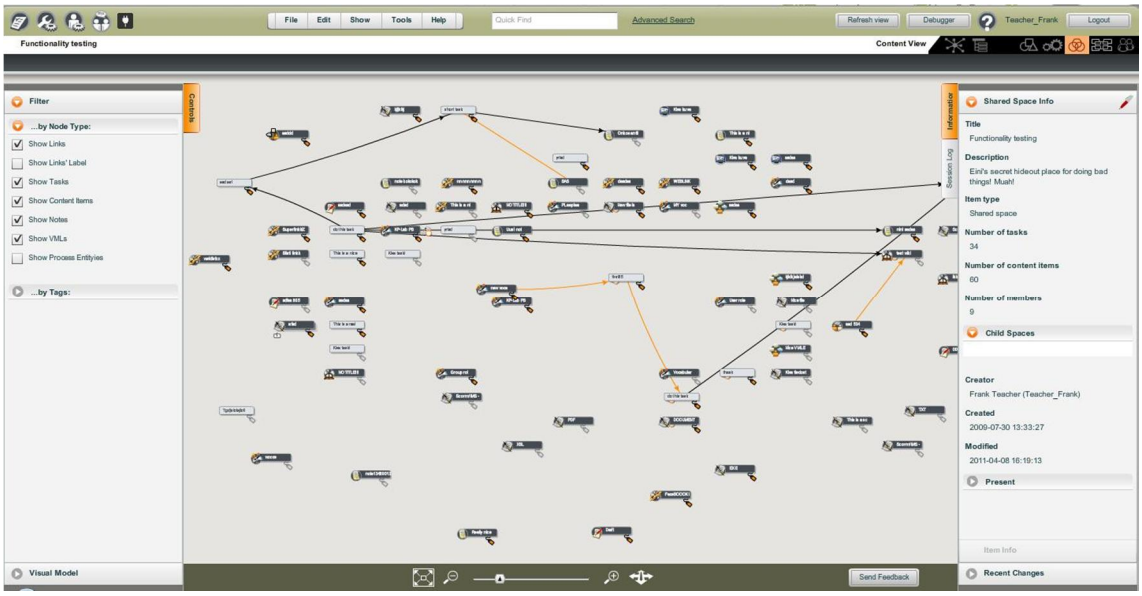
3 Knowledge Practices Environment -järjestelmä

3.1 KPE-järjestelmä yleisesti

Insinöörityössä toteutettiin liitännäisarkkitehtuuri KPE (Knowledge Practices Environment) -järjestelmään. Se on KP-Lab-projektissa kehitetty virtuaalinen työskentelyympäristö, joka tukee yhteisöllistä oppimista ja muuta työskentelyä. Se on verkossa toimiva järjestelmä, ja se tarjoaa työkaluja, toiminnallisuuksia ja ominaisuuksia, jotka mahdollistavat yhteisöllisen ja reaaliaikaisen työskentelyn. KPE-järjestelmä tarjoaa virtuaalisia yhteisiä työtiloja (shared space), joissa käyttäjät voivat työskennellä. Työtilat jakautuvat näkymiin, jotka mahdollistavat työtilan sisällön tarkastelun eri näkökulmista. Tärkeimmät näkymät ovat Content View, Process View ja Community View -näkymät.

Työtilat sisältävät monia eri työkaluja, joiden avulla voidaan hallita ja analysoida työtilan sisältöä. Työkaluja on muun muassa HPA (real-time and history based awareness) -työkalu, muistiinpanojen muokkaustyökalu (Note Editor), kommentointityökalu (Commenting) ja semanttinen haku (Semantic Search).

Ensisijainen näkymä KPE-järjestelmässä on Content View, ja sen sisältö esitetään varsin visuaalisesti. Kuvio 4 esittää KPE-järjestelmän Content View -näkymän.



Kuvio 4. KPE-järjestelmän Content View -näkymä.

Kuviosta 4 nähdään, että sisällön visuaalinen esittäminen muistuttaa hieman tietokoneen työpöytää. Näkymän sisältö eli tietosolmut esitetään kuvakkeina, ja niitä voidaan yhdistellä linkeillä. Käyttäjät voivat järjestää sisällön hyvin vapaasti ja joustavasti. KPE-järjestelmä poikkeaa tavanomaisista oppimis- ja työskentely-ympäristöistä siten, että se ei esitä sisältöään kansiorakenteellisesti tai hierarkkisesti, kuten esimerkiksi Moodle-oppimisympäristö.

Tietosolmutyyppejä on KPE-järjestelmässä useita, ja niillä voidaan luoda erilaista sisältöä järjestelmään. Tietosolmut eroavat toisistaan, joten niille on määritelty erilaisia tietosolmutyyppejä. Tietosolmutyyppejä voidaan verrata esimerkiksi käyttöjärjestelmän tiedostotyyppeihin, joiden luomista ja muokkaamista varten on omat työkalunsa. KPE-järjestelmän tietosolmutyyppejä ovat esimerkiksi Task (tehtävä), Note (muistiinpano), Web Link (Internet-linkki), Chat (keskustelu) ja Shared Space (jaettu työtila).

3.2 KPE ja liitännäiset

KPE-järjestelmään on toteutettu aikaisemmin liitännäisarkkitehtuuri, jolla oli järjestelmään saatu lisättyä yksi liitännäinen. Aikaisemman liitännäisarkkitehtuurin liitännäisten lataaminen, hallinnointi ja käsittely ei kaikilta osin toteuttanut liitännäisten ja liitän-

näisarkkitehtuurin asettamia määritelmiä, joten arkkitehtuuria uudistettiin ja parannettiin. Esimerkiksi KPE-järjestelmä ei osannut käsitellä liitännäisiä dynaamisesti, vaan niihin liittyvät toiminnallisuudet oli kirjoitettu suoraan järjestelmän lähdekoodiin. Lisäksi liitännäisen aktivoiminen käyttöliittymässä vaati aina järjestelmän uudelleen käynnistämisen. Kun järjestelmän aloituslatausaika on melko pitkä, on käyttäjän kannalta miellyttävämpää, jos järjestelmään pystytään lisäämään liitännäiset ajon aikana.

Aikaisempaan järjestelmään ei pystytty joustavasti ja helposti lisäämään uusia työkaluja ja tietosolmutyyppejä. Näin ollen liitännäisarkkitehtuurilla haluttiin helpottaa tietosolmutyyppien ja työkalujen kehittämistä ja niiden kehitys haluttiin mahdollistaa myös kolmansien osapuolien kehittäjille. Lisäksi haluttiin helpottaa sovelluksen räätälöintiä eri käyttäjäkunnille.

Flex-ohjelmointiympäristö ei tarjonnut vielä KPE-järjestelmän aikaisemman liitännäisarkkitehtuurin toteutusvaiheessa sub-application-tekniikkaa, joten liitännäisarkkitehtuuri toteutettiin module-komponentteja käyttäen. Flexin 3.2-versio lisäsi SWFloaderiin turvallisuuteen ja versiointiin liittyviä ominaisuuksia, ja näin ollen vasta Flexin 3.2-version jälkeen sub-application-tekniikan käyttäminen tuli Flex-sovelluksille hyödylliseksi [11].

Module- ja sub-application-komponentteja vertailtiin keskenään ja huomattiin, että molemmissa tekniikoissa oli liitännäisarkkitehtuurin kannalta sekä hyötyjä että haittoja. Module-tekniikalla rakennettu arkkitehtuuri haluttiin kuitenkin säilyttää, vaikka sub-application-komponentit olisivat tarjonneet paremman tuen liitännäisten turvallisuuteen ja versiointiin liittyviin ongelmiin. Sub-application-tekniikalla ei voida luoda ei-visuaalisia komponentteja, sillä ne ovat nimensä mukaisesti alisovelluksia eli samanlaisia kuin itse pääsovellus. Sub-application-tekniikalla ei voitaisi siis toteuttaa liitännäistä, joka muuttaa vain ohjelman toiminnallisuuksia lisäämättä visuaalisia elementtejä käyttöliittymään.

3.3 KPE-järjestelmän liitännäisarkkitehtuuriin liittyvät ongelmat

Seuraavassa esitellään KPE-järjestelmän liitännäisarkkitehtuurin toteuttamiseen liittyvät ongelmat, joihin piti arkkitehtuurin toteutusvaiheessa kiinnittää huomiota. Varsinaiset järjestelmän liitännäisarkkitehtuurin vaatimukset ja määrittelyt esitellään luvussa 4.

Liitännäisarkkitehtuurin rajoittaminen

Ihanteellisessa liitännäisarkkitehtuurissa KPE-järjestelmän kaikki tietosolmut, näkymät ja työkalut olisivat liitännäisiä, jolloin KPE-järjestelmä voitaisiin räätälöidä tietylle kohderyhmälle tai tietylle käyttötapaukselle sopivaksi. Työn tavoitteena oli kuitenkin toteuttaa liitännäisarkkitehtuuri KPE-järjestelmän perustoiminnallisuuksille ja toteuttaa arkkitehtuuri niin, että se voidaan myöhemmin laajentaa tukemaan muita järjestelmän toiminnallisuuksia.

Liitännäisten dynaaminen käsittely

KPE-järjestelmä ei voi etukäteen tietää, mitä uutta tietoa ja mitä uusia asioita liitännäiset lisäävät järjestelmään. Järjestelmä voi vain rajoittaa, mitkä sen toiminnallisuudet toimivat ja tukevat liitännäisiä, ja määritellä, mitä asioita järjestelmässä voidaan laajentaa. Koska järjestelmä ei tiedä etukäteen, mitä liitännäiset lisäävät järjestelmään, täytyy liitännäisten kertoa, miten järjestelmän pitää käsitellä ne. Konfiguraation avulla liitännäiset pystyvät kertomaan KPE-järjestelmälle, mitä asioita ne laajentavat, ja näin KPE-järjestelmä osaa luoda ja näyttää liitännäisen tarjoaman uuden tiedon ja toiminnallisuudet järjestelmän käyttöliittymässä.

Tietokanta ja tiedon tallentaminen

KPE-järjestelmässä on semanttinen tietokanta. Tietokannan tietomalli määritellään RDF (Resource description framework) -metakielen avulla. RDF-metakieli on semanttiseen webiin liittyvä ontologioiden mallinnuskieli [13]. Koska liitännäisarkkitehtuurin tavoitteena on mahdollistaa uuden ja ennalta määrittelemättömän tiedon lisääminen järjes-

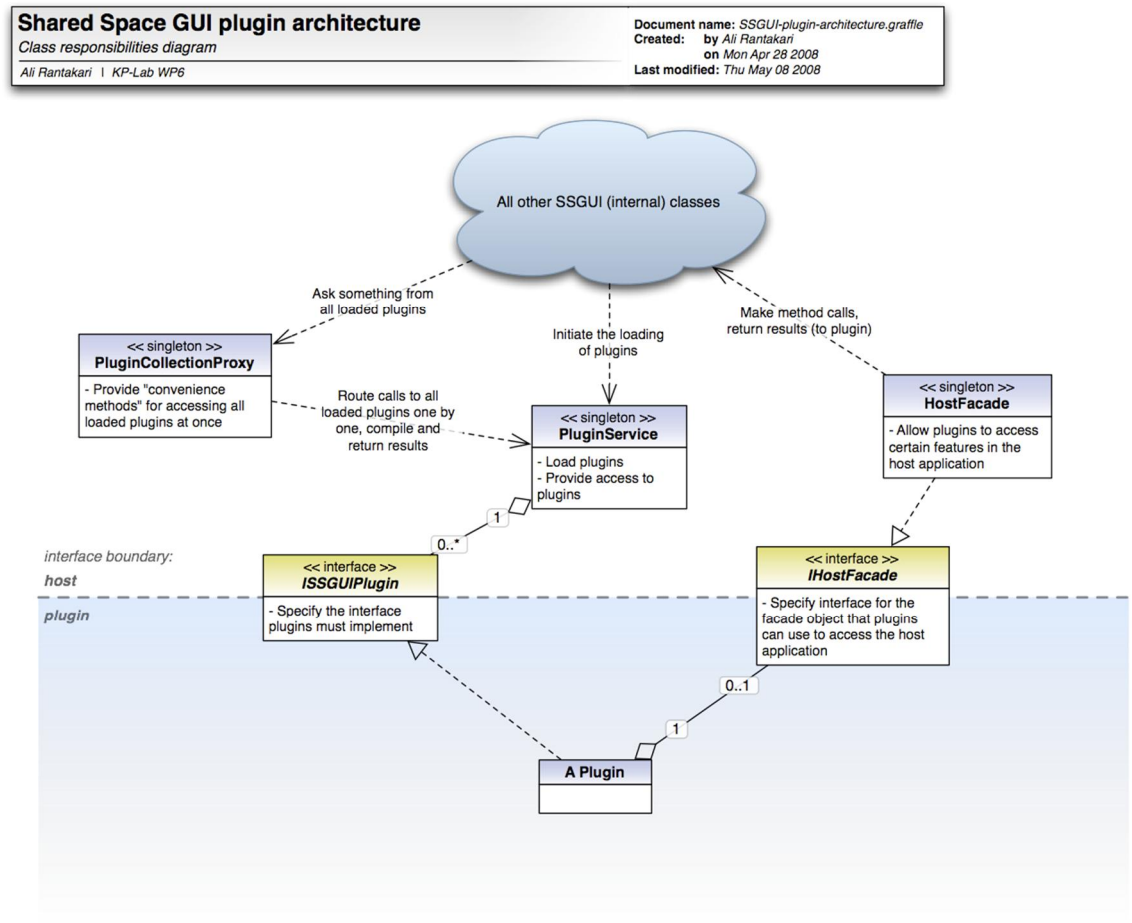
telmään, olisi erittäin vaikeaa, ellei mahdotonta sallia mielivaltaisen tiedon lisääminen ja käsittely KPE-järjestelmän semanttiseen tietokantaan. Liitännäisiä varten täytyi toteuttaa tietokanta, joka pystyy vaivattomasti skaalautumaan liitännäisen tarpeiden mukaisesti.

Seuraavia liitännäisarkkitehtuurin toteutukseen liittyviä ongelmia ei toteutettu työn puitteissa: liitännäisten turvallisuus, versiointi ja jakelu. Näiden ongelmien mahdollisia toteutustapoja pohditaan kuitenkin luvussa 5.

4 KPE-järjestelmän liitännäisarkkitehtuuri

4.1 Aikaisempi liitännäisarkkitehtuuri

KPE-järjestelmän ensimmäinen liitännäisarkkitehtuuri toteutettiin vuosina 2007–2008, ja sen rakenne ja käytetyt tekniset ratkaisut olivat hyviä, mutta arkkitehtuurin toteutus KPE-järjestelmän puolella ei ollut riittävä. Kuviosta 5 nähdään KPE-järjestelmän aikaisempi liitännäisarkkitehtuuri.



Kuvio 5. KPE-järjestelmän ensimmäinen liitännäisarkkitehtuuri [14].

Kuviosta 5 nähdään, että KPE-järjestelmän liitännäisarkkitehtuurin rakenne on samanlainen kuin luvussa 2.3 esitetty liitännäisarkkitehtuurin rakenne (kuvio 3). Kuviossa 5 ISSGUIPlugin-rajapinta vastaa kuvion 3 liitännäisten rajapintaa ja IHostFacade-rajapinta vastaa kuvion 3 palveluiden rajapintaa. Muut kuviossa 5 esitetyt luokat ovat KPE-järjestelmän sisällä olevia luokkia, jotka muun muassa hallinnoivat liitännäisiä ja tarjoavat niille tietoa.

Aikaisemman liitännäisarkkitehtuurin rajapintaratkaisu oli hyvä, joten sen rakennetta ei työssä muutettu. Työn puitteissa sitä ainoastaan uudistettiin tukemaan monipuolisemmin KPE-järjestelmän toiminnallisuuksia. Järjestelmän sisälle ei kuitenkaan oltu toteutettu liitännäisten käsittelyä liitännäisarkkitehtuurin vaatimalla tavalla. Työ keskittyi näin ollen toteuttamaan liitännäisarkkitehtuurin KPE-järjestelmän sisälle.

4.2 Liitännäisarkkitehtuurin vaatimukset ja määrittely

4.2.1 Vaatimukset

KPE-järjestelmän liitännäisarkkitehtuurin haluttiin mahdollistavan uusien tietosolmutyyppien ja työkalujen luominen, ja haluttiin, että järjestelmää pystytään räätälöimään eri käyttäjäryhmille. Arkkitehtuurille asetettiin siksi seuraavat vaatimukset:

1. Liitännäisten kehittäjän pitää pystyä luomaan uusia tietosolmutyyppejä.
2. Liitännäisten kehittäjän pitää pystyä luomaan uusia työkaluja.
3. Liitännäisten tietosolmut ja työkalut voivat käyttää KPE-järjestelmän perusominaisuuksia.
4. Liitännäisten tieto tallennetaan KPE-järjestelmään liitännäisten omaan tietokantaan.
5. Käyttäjät voivat itse ajon aikana määritellä, mitä liitännäisiä haluavat KPE-järjestelmässä käyttää.

Kun liitännäisarkkitehtuuri toteuttaa yllä mainitut vaatimukset, liitännäiset pystyvät toimimaan vaaditulla tavalla KPE-järjestelmässä. Lisäksi järjestelmän käyttäjät voivat halutessaan räätälöidä KPE-järjestelmää aktivoimalla ja poistamalla liitännäisiä käytöstä.

Liitännäisarkkitehtuurille täytyi asettaa muutamia rajoituksia, sillä KPE-järjestelmä haluttiin pitää yksinkertaisena mutta myös haluttiin tarjota tarpeeksi monipuoliset mahdollisuudet liitännäisten luomiseen. Pitämällä liitännäisarkkitehtuuri yksinkertaisena voidaan todennäköisesti hieman helpottaa liitännäisten kehittämistä. Liitännäisarkkitehtuurille asetettiin seuraavat vaatimukset:

- Liitännäisten kehittäjät eivät voi luoda uusia tietosolmutyyppejä Network View -näkymään.
- Liitännäisten kehittäjät eivät voi luoda KPE-järjestelmään uusia näkymiä.

Rajoituksiksi määriteltyjen toiminnallisuuksien toteuttaminen aikataulun puitteissa olisi ollut myös haastavaa, joten yksinkertaisuuden vuoksi nämä toiminnallisuudet päätettiin rajata arkkitehtuurin tukemien toiminnallisuuksien ulkopuolelle.

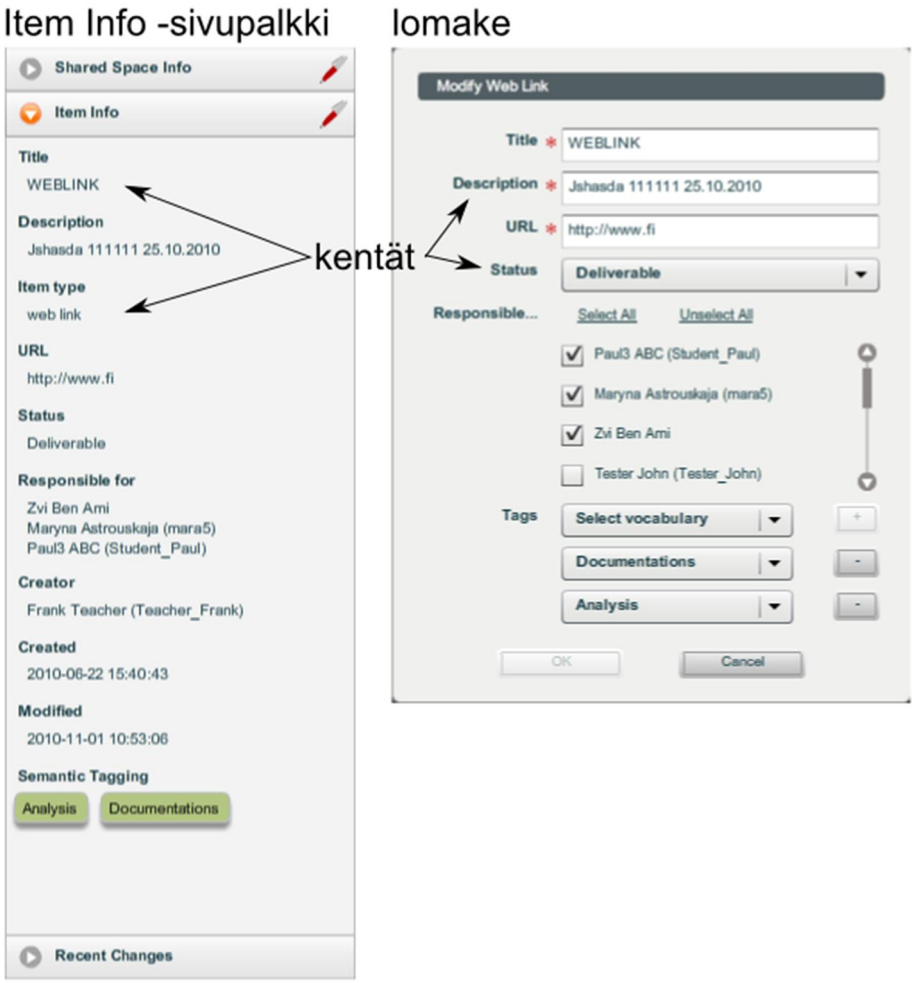
4.2.2 Liitännäisarkkitehtuurin tukemat toiminnallisuudet

Kun liitännäinen lisätään järjestelmään, täytyy isäntäsovelluksen pystyä reagoimaan lisäykseen näyttämällä liitännäisen läsnäolo käyttöliittymäkomponenteissa. Käyttöliittymäkomponentit näyttävät tietoa liitännäisten tietosolmuista tai työkalusta. Toteutetun arkkitehtuurin tuetuiksi perustoiminnallisuuksiksi määriteltiin Item Info -sivupalkki ja lomakkeet, pää- ja pikavalikko (context menu) ja tietosolmujen ulkoasu.

Item Info -sivupalkki ja lomakkeet

KPE-järjestelmän Item Info -sivupalkki on komponentti, joka näyttää tietosolmujen tiedon. Lomake puolestaan on komponentti, jonka avulla syötetään ja muokataan tietosolmun tietoa. Molempien komponenttien täytyy ymmärtää liitännäisiä, jotka lisäävät uuden tietosolmutyypin.

Toteutetussa arkkitehtuurissa Item Info -sivupalkin ja lomakkeiden kentät luodaan dynaamisesti tietosolmujen konfiguraation avulla. Kuvio 6 esittää KPE-järjestelmän Item Info -sivupalkin ja lomakkeen.



Kuvio 6. Item Info -sivupalkki ja lomake KPE-järjestelmässä.

Aikaisemmassa arkkitehtuurissa lähes jokaiselle KPE-järjestelmän tietosolmutyypille oli määritelty oma lomake ja Item Info -sivupalkissa näytettävä tieto. Aikaisemman arkkitehtuurin luokat olivat tehty niin sanotusti ”käsityönä”, siten että toteutukset olivat toisistaan irrallisia, eivät yleistäviä. Konfiguraation avulla nykyinen arkkitehtuuri käyttää vain yhtä lomaketta kaikkien tietosolmutyyppien esittämiseen. Kun konfiguraatiotiedostossa tehdään muutos molemmat, Item Info -sivupalkki ja lomake, näyttävät oikean tiedon ilman, että muutoksia tarvitsee erikseen tehdä lähdekoodiin.

Aikaisemmassa liitännäisarkkitehtuurissa liitännäiset loivat itse tietosolmujen lisäys- ja muokkauslomakkeet, koska KPE-järjestelmä ei tiennyt liitännäisten tietosolmuista mitään. Lisäksi liitännäisten lomakkeiden luomista ei ollut toteutettu KPE-järjestelmän lähdekoodiin. Näin ollen liitännäisten lomakkeet olivat toiminnallisuuksiltaan ja ulko-

asuiltaan aivan erilaisia kuin KPE-järjestelmän tietosolmujen lomakkeet. Myöskään liitännäisten tietoa ei voitu näyttää Item Info -sivupalkissa ollenkaan. Item Info -sivupalkki on KPE-järjestelmään toteutettu komponentti, eivätkä liitännäiset pysty itsenäisesti asettamaan KPE-järjestelmän komponentteihin tietoa.

Nykyisessä arkkitehtuurissa liitännäisten tietosolmujen tieto esitetään konfiguraation avulla samoilla käyttöliittymäkomponenteilla kuin KPE-järjestelmän tietosolmujen tieto. Lomakkeiden ja Item Info -sivupalkin yhtenäinen toteutus parantaa myös järjestelmän yhdenmukaisuutta. Aikaisemmin järjestelmässä on ollut paljon pieniä eriäväisyyksiä eri tietosolmutyyppien lomakkeissa ja Item Info -sivupalkissa näytettävissä tiedoissa.

Valikot

Järjestelmä sisältää kaksi eri valikkoa: pää- ja pikavalikon. Kuvio 7 esittää KPE-järjestelmässä näytettävät valikot.

Kuvio 7. KPE-järjestelmän pää- ja pikavalikko.

Käyttäjän kannalta on tärkeää, että KPE-järjestelmän valikoita pystytään muuttamaan dynaamisesti: jos käyttäjä ottaa liitännäisen pois käytöstä, niin liitännäiseen liittyvien toimintojen täytyy poistua valikoista.

Aikaisemmassa arkkitehtuurissa valikoiden dynaamista muuttamista ei ollut huomioitu ollenkaan: pää- ja pikavalikko määritelty kiinteästi valikoiden komponentteihin. Esimerkiksi KPE-järjestelmän päävalikkokomponenttiin oli suoraan kirjoitettu XML-tieto valikon määrittelyä varten. XML-tietoa ei voitu dynaamisesti muokata, vaan jos päävalikkoon haluttiin tehdä muutos, se täytyi kirjoittaa komponenttiin ja koko sovellus piti kääntää ja ajaa uudelleen. Lisäksi liitännäiset joutuivat itse luomaan tietosolmujen pikavalikot, ja näin ollen liitännäisten tietosolmujen pikavalikot erosivat suuresti KPE-järjestelmän tietosolmujen pikavalikoista. Liitännäisten pikavalikot eivät voineet käyttää KPE-järjestelmän toiminnallisuuksia, vaan liitännäiset joutuivat itse toteuttamaan liitännäisen tietosolmuun liittyvät toiminnallisuudet kuten tietosolmun avaamisen, muokkaamisen ja poistamisen. Koska päävalikko ei pystynyt ajon aikana muuttumaan, se ei osannut näyttää liitännäisten tietosolmuihin liittyviä toiminnallisuuksia ja näin ollen päävalikossa ei ollut mitään liitännäisten tietosolmuihin liittyviä toiminnallisuuksia.

Nykyisen arkkitehtuurin konfiguroiminen mahdollistaa, että molemmat, pää- ja pikavalikko, näyttävät oikean informaation ja molempia voi päivittää ajon aikaisesti.

Tietosolmujen ulkoasu

Aikaisemmassa liitännäisarkkitehtuurissa liitännäiset olivat itse vastuussa tietosolmujen ulkoasujen luomisesta. Kehittäjän kannalta miellyttävämpi toteutus on tarjota liitännäisten tietosolmuille oletusulkoasu mutta myös mahdollistaa liitännäisten ulkoasujen muokkaamisen. Näin saadaan hieman helpotettua liitännäisten kehittämistä, mutta silti pystytään mahdollistamaan omien ulkoasujen luonti, jos liitännäisen kehittäjä niin haluaa tehdä.

4.2.3 Liitännäisarkkitehtuurin rajapinnat

Aikaisemmassa liitännäisarkkitehtuurissa kehittäjien täytyi itse lisätä KPE-järjestelmän vaatimat rajapinnat luokka luokalta Flex-projektiin. Tämä tapa jakaa liitännäisarkkitehtuurin rajapinnat johtaa helposti siihen, että rajapintoihin saattaa tulla virheitä, ja näin ollen liitännäiset eivät toimi KPE-järjestelmässä. Parempi ja helpompi vaihtoehto on

tarjota rajapinnat RSLs (Runtime Shared Libraries) -kirjastona, jonka kehittäjät voivat helposti lisätä Flex-projektiin. Myös KPE-järjestelmä lataa liitännäisten rajapinnat RSLs-kirjastona. Näin varmistetaan, että liitännäisten ja KPE-järjestelmän rajapinnat ovat samat ja pystytään välttämään tilanteet, joissa vahingossa saatettaisiin muuttaa liitännäisten rajapintoja ja näin rikkoa liitännäisten toimiminen järjestelmässä.

Liitännäisarkkitehtuurin rajapintojen jakaminen RSLs-kirjastona on helpompaa, ja lisäksi ne vähentävät kokonaissovelluksen koodin määrää. RSLs-kirjastojen ominaisuutena on, että ne tallennetaan selaimen välimuistiin ja samaa kirjastoa voi käyttää usea sovellus. Näin ollen, jos sovelluksessa on useita liitännäisiä, liitännäisarkkitehtuurin rajapinnat ladataan vain kerran, eikä jokaiselle liitännäiselle erikseen, kuten aikaisemmassa arkkitehtuurissa tehtiin. [15, s. 461–461.]

Aikaisempia rajapintoja laajennettiin tukemaan liitännäisten konfiguraatiota ja rajapintaan toteutettiin työkalutyyppisille liitännäisille tarvittavat toiminnallisuudet eli miten työkalu avataan ja miten sen sisältö näytetään KPE-järjestelmässä. Uusi rajapinta vaatii muun muassa liitännäistä tarjoamaan konfiguraatiotiedoston, jonka avulla KPE-järjestelmä pystyy liittämään liitännäisen toiminnallisuuden järjestelmään dynaamisesti.

4.3 KPE-järjestelmän ja liitännäisten konfiguroiminen

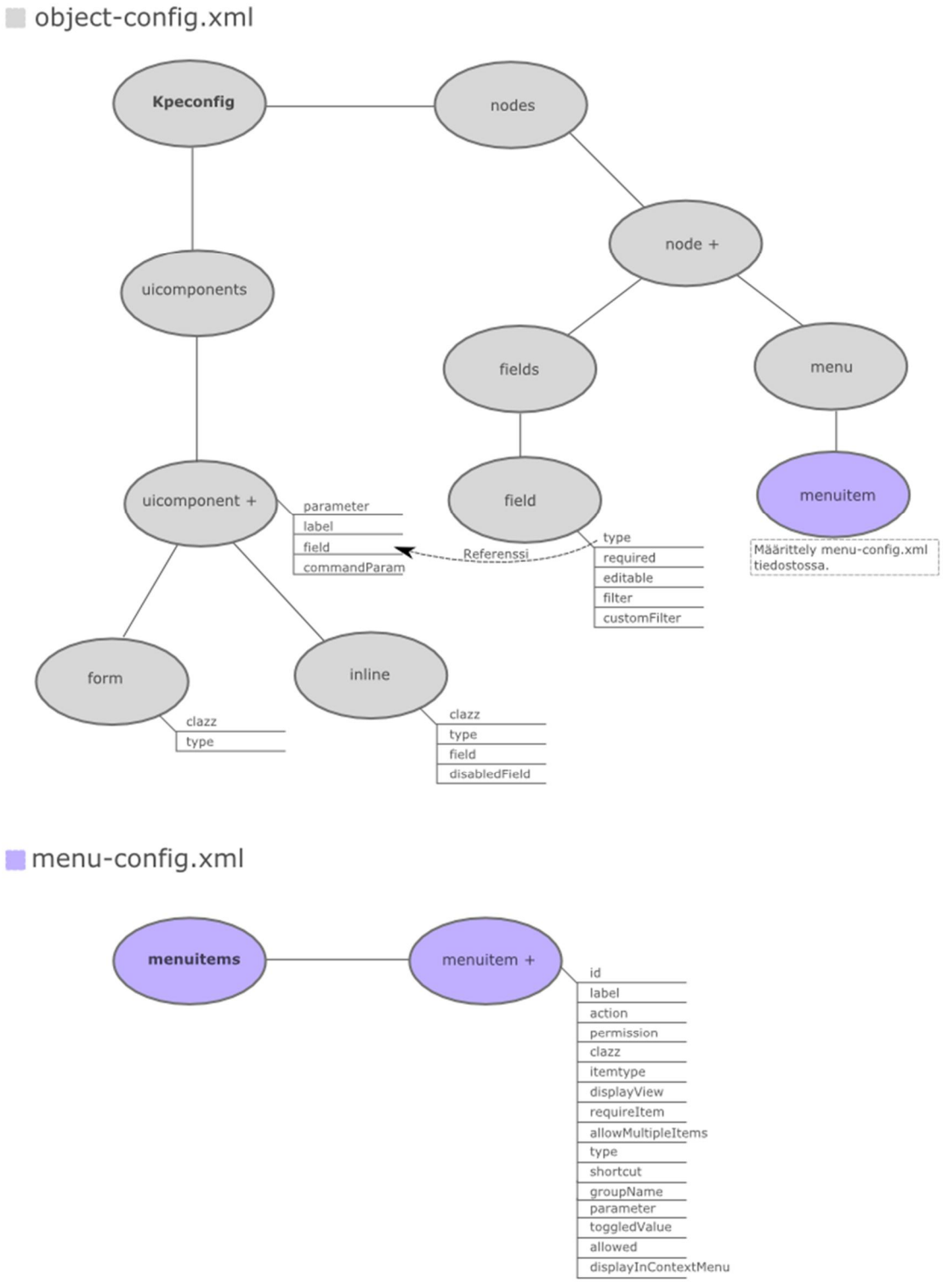
4.3.1 KPE-järjestelmän konfigurointi

Liitännäisarkkitehtuurin toteuttamisen yhteydessä haluttiin yksinkertaistaa ja selkeyttää KPE-järjestelmää, joten oli mielekästä muuttaa myös järjestelmän omien tietosolmujen käsittely dynaamiseksi. Näin KPE-järjestelmä pystyy käyttämään muun muassa samaa lomaketta niin omien tietosolmujen kuin liitännäisten tietosolmujen esittämiseen. Lisäksi pystyttiin muuttamaan visuaalisten komponenttien käsittely korkeammalle tasolle. Esimerkiksi päävalikkoa ei määritellä näkymäkohtaisesti, vaan valikko määritellään vain kerran ja se on kaikille näkymille yhteinen; näkymät ja tietosolmut voivat tarvittaessa muuttaa valikossa näytettävää tietoa esimerkiksi näkymäkohtaiseksi.

Yhtenäistämällä KPE-järjestelmän ja liitännäisten tietosolmujen käsittely saadaan parannettua järjestelmän yhdenmukaisuutta, sillä kaikki tietosolmut ja näkymien valikot käyttävät samoja käyttöliittymäkomponentteja tiedon esittämiseen. Lisäksi järjestelmän virheensietokyky paranee ja mahdollisesti virheenkorjaukseen käytettävä aika lyhenee, sillä mahdollista virhettä voidaan etsiä yhdestä lomakkeesta tai valikosta, kun aikaisemmin kehittäjä saattoi joutua käymään läpi kaikki näkymät ja lomakkeet löytääkseen virheen.

Konfiguraatio mahdollistaa myös sen, että liitännäisarkkitehtuuria voidaan helposti laajentaa tukemaan paremmin KPE-järjestelmän muita toiminnallisuuksia. Esimerkiksi KPE-järjestelmän omien tietosolmujen laajentamisen mahdollistaminen olisi hyvin helposti toteutettavissa. Tietosolmujen laajentamisella tarkoitetaan, että liitännäisillä voitaisiin luoda uusia kenttiä KPE-järjestelmän omiin tietosolmuihin.

KPE-järjestelmään on lisätty kaksi konfiguraatiotiedostoa: *object-config.xml* ja *menu-config.xml*. Tiedostot mahdollistavat sen, että luvussa 4.2 mainitut toiminnot ja työkalut hallitaan dynaamisesti. Kuvio 8 esittää visuaalisesti tiedostojen rakenteet ja riippuvuudet toisiinsa.



Kuvio 8. KPE-järjestelmän konfiguraatiotiedostojen rakenteet ja riippuvuudet.

Seuraavassa kuvataan pintapuolisesti konfiguraatiotiedostojen rakenteet ja käyttötarkoitukset. Tarkemmin konfiguraatiotiedostot on esitelty KPE-järjestelmän liitännäisten kehittäjänoppaassa (liite 1).

Tietosolmutyyppien konfigurointi

Object-config.xml-tiedosto konfiguroi kaikki KPE-järjestelmässä olevat tietosolmutyypit, ja se myös määrittelee, mitä käyttöliittymäkomponentteja tietosolmut käyttävät lomakkeissa ja Item Info -sivupalkissa. Lisäksi tiedosto sisältää tietosolmukohtaisen valikko-konfiguraation eli sen, mitä toiminnallisuuksia näytetään KPE-järjestelmän pää- ja pikavalikossa tietosolmun ollessa valittuna. Tiedoston avulla järjestelmä on tietoinen, mitä tietoja näytetään miltäkin tietosolmulta ja kuinka tieto tulisi visuaalisesti esittää käyttöliittymässä.

Järjestelmän käyttöliittymäkomponentit määritellään uicomponent-elementteihin. Elementillä on kaksi alielementtiä: form ja inline. Form-elementti määrittelee, mitä käyttöliittymäkomponenttia käytetään lomakkeissa, ja inline-elementti määrittelee, mitä komponenttia käytetään Item Info -sivupalkissa. Kuviossa 9 näytetään konfiguraatio Title-käyttöliittymäkomponentille.

```
<uicomponent parameter="title" label="Title" field="title" commandParam="title">
    <form clazz="ssgui.components.popups.forms.components" type="TitleInput" />
    <inline clazz="ssgui.libraries.adapter" type="AdapterField" field="EDITABLE_TEXT" disabledField="LABEL"/>
</uicomponent>
```

Kuvio 9. Käyttöliittymäkomponentin konfigurointi.

Kuvion 9 käyttöliittymäkomponenttia voidaan käyttää tietosolmujen otsikkotiedon näyttämiseen käyttöliittymässä.

KPE-järjestelmän tietosolmut määritellään node-elementteihin. Elementillä on kaksi alielementtiä: fields ja menu. Fields-elementti sisältää field-elementtejä, jotka määrittelevät, mitä tietoa tietosolmu voi sisältää: esimerkiksi otsikon, kuvauksen ja tilatiedon (status). Menu-elementti sisältää menuitem-elementtejä, jotka määritelevät tietosolmun valikon, eli mitä toiminnallisuuksia näytetään pää- ja pikavalikossa. Kuviossa 10 nähdään esimerkkikonfiguraatio Shared Space -tietosolmuille.

```
<node name="SharedSpace" parent="RootSpace" label="LB_SSP_S"
      displayView="NETWORK_VIEW,LIST_VIEW"
      clazz="ssgui.libraries.graphModel.SharedSpace"
      type="TYPE" itemtype="SharedSpace">
    <fields>
        <field type="title" required="true" editable="false" filter="SSP_TITLE_REGEXP"/>
        <field type="description" required="true" />
        <field type="itemType" required="false" editable="false"/>
        <field type="numOfTasks" editable="false" />
        <field type="numOfContentItems" editable="false" />
        <field type="numOfMembers" editable="false" />
        <field type="hierarchicalParent" editable="false" />
        <field type="childSpaces" editable="false" />
        <field type="creator" editable="false" />
        <field type="created" editable="false" />
        <field type="modified" editable="false" />
        <field type="presence" editable="false" />
    </fields>
    <menu>
        <menuitem id="open" displayInContextMenu="true" />
        <menuitem id="modify" displayInContextMenu="true"/>
        <menuitem id="delete" displayInContextMenu="true"/>
        <menuitem id="add_childSSP" displayInContextMenu="true"/>
        <menuitem id="copy" displayInContextMenu="true"/>
        <menuitem id="comment" displayInContextMenu="true"/>
        <menuitem id="chat" allowed="false" />
        <menuitem id="tag" allowed="false" />
    </menu>
</node>
```

Kuvio 10. Shared Space -tietosolmun konfigurointi.

Kuviosta 10 voidaan nähdä, että se sisältää kaiken Shared Space -tietosolmun esittämiseen tarvittavan tiedon: tietokentät ja tietosolmukohtaisen valikkokonfiguraation. Tarkemmin tietosolmujen konfiguraatio ja elementtien väliset suhteet on esitelty KPE-järjestelmän kehittäjän oppaassa (liite 1).

Valikoiden konfigurointi

Menu-config.xml-tiedosto määrittelee oletus-pää- ja -pikavalikot KPE-järjestelmään. Object-config.xml-tiedostossa on määritelty tietosolmukohtaiset valikot, joten valikot pystyvät muuttumaan tarpeen vaatiessa näyttämään valitun tietosolmun tarvitsemat tiedot. Kuviossa 11 nähdään KPE-järjestelmän konfiguraatio Help (apu) -valikolle.

```
<menuitem id="help" label="Help">
    <menuitem label="KP-Lab System Help" action="help" data="openHelp" shortcut="F1" />
    <menuitem label="Help Mode" action="helpMode" data="helpMode" />
    <menuitem label="Shortcuts Help" action="shortcutHelp" data="shortcutshelp" />
    <menuitem label="" data="" type="separator" />
    <menuitem label="Send Feedback" action="feedback" data="feedbackForm" />
    <menuitem label="About" action="about" data="about" />
</menuitem>
```

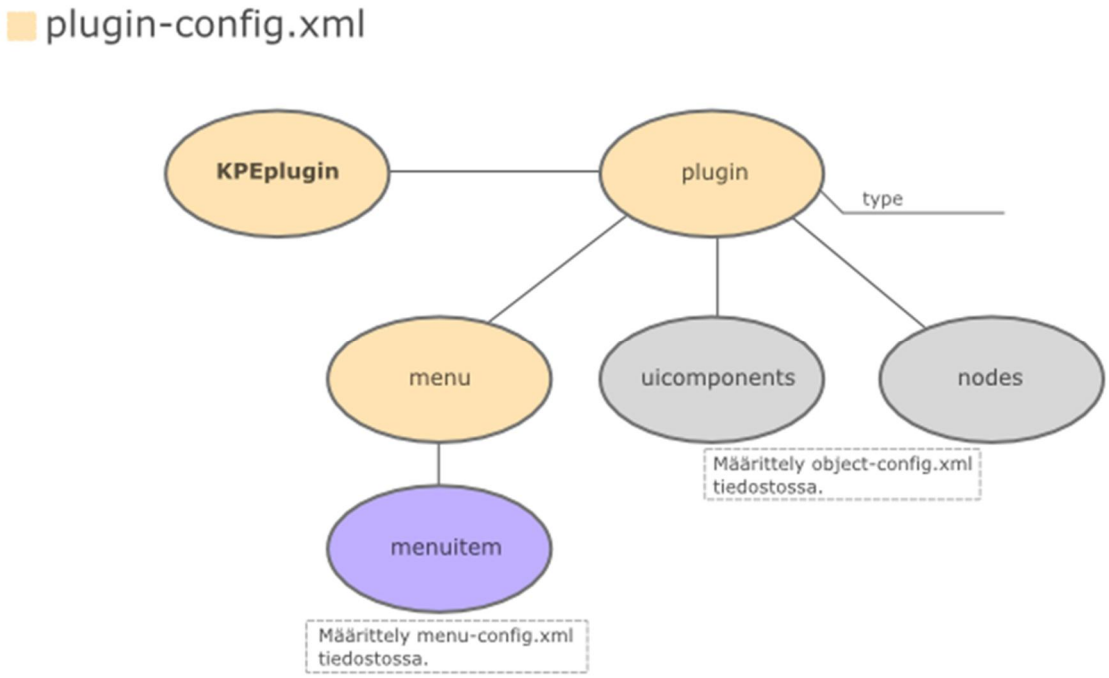
Kuvio 11. Help-valikon konfiguraatio.

Help-valikon konfiguraatiossa määritellään valikossa näkyvä teksti, toiminto, joka suoritetaan, kun valikon toiminto (menunitem) valitaan, ja mahdollisesti pikatoiminnon (shortcut) teksti. KPE-järjestelmän Help-valikko on jokaisessa näkymässä sama, eikä se aseta näkymäkohtaisia rajoitteita tai käyttäjien oikeuksille rajoitteita. Valikoiden konfiguraatiorakenne ja mahdolliset rajoitteet on esitetty tarkemmin KPE-järjestelmän kehittäjän oppaassa (katso liite 1).

Menu-config.xml-tiedostoa voidaan käyttää melkein suoraan KPE-järjestelmän päävalikossa. Järjestelmä tarkistaa ainoastaan, mitkä valikon toiminnot ovat sallittuja avatussa näkymässä ja onko käyttäjällä oikeutta käyttää valikon toimintoa.

4.3.2 Liitännäisten konfigurointi

Liitännäisten levityspaketin täytyy sisältää konfiguraatiotiedosto, jotta KPE-järjestelmä pystyy käsittelemään liitännäisen tarjoaman uuden tiedon. Kuvio 12 esittää visuaalisesti *plugin-config.xml*-tiedoston rakenteen ja suhteet.



Kuvio 12. Plugin-config.xml-tiedoston rakenne ja elementtien väliset suhteet.

Kuviossa 12 plugin-elementti sisältää kolme alielementtiä: menu, uicomponents ja nodes. Plugin-elementti sisältää myös type-attribuutin, joka määrittelee, käsitelläänkö liitännäinen työkaluna vai ei. Uicomponents- ja nodes-elementit määritellään ja käytetään samalla lailla kuin käyttöliittymäkomponentit ja tietosolmut määriteltiin object-config.xml tiedostossa. Plugin-elementin menu-elementti voi sisältää vain yhden menuitem-elementin, jolla määritellään, näytetäänkö liitännäinen KPE-järjestelmän Tools-valikossa (työkalu-valikossa). Jos liitännäiselle ei ole plugin-elementtiin määritelty type-attribuuttia, edellä mainittu menu-elementti jätetään kokonaan huomioimatta. Kuvio 13 esittää esimerkin liitännäisen konfiguraatiosta.

```
<KPEplugin>
    <plugin type="tool">
        <menu>
            <menuitem   label="Open KPE Plugin"
                        displayView="PROCESS_VIEW,CONTENT_VIEW,ALTERNATIVE_PROCESS_VIEW,TAILORED_VIEW,COMMUNITY_VIEW"
                        action="openTool" data="kpePlugin"/>
        </menu>
        <uicomponents>
            <uicomponent field="PluginName" parameter="plugin,pluginName" label="Plugin">
                <form clazz="" type=""/>
                <inline clazz="ssgui.libraries.adapter" type="AdapterField" field="LABEL" disabledField="LABEL" />
            </uicomponent>
            <uicomponent field="Summary" parameter="dataEntries,summary" label="Summary">
                <form clazz="ssgui.components.popups.forms.components" type="DescriptionInput" />
                <inline clazz="ssgui.libraries.adapter" type="AdapterField" field="EDITABLE_TEXT" disabledField="LABEL" />
            </uicomponent>
        </uicomponents>
        <nodes>
            <node   name="PluginNode" typeID="http://www.kpePlugin.org/ontologies/PluginNode#BaseNode"
                    label="KPEPlugin node" displayView="CONTENT_VIEW,ALTERNATIVE_PROCESS_VIEW,TAILORED_VIEW"
                    clazz="ssgui.libraries.graphModel.PluginNode" type="TYPE" itemtype="PluginNode">
                <fields>
                    <field type="title" required="true" />
                    <field type="description" required="true" />
                    <field type="itemType" required="false" editable="false"/>
                    <field type="PluginName" required="false" editable="false" />
                    <field type="creator" editable="false" />
                    <field type="created" editable="false" />
                    <field type="modified" editable="false" />
                    <field type="Summary" editable = "true" required = "false" />
                </fields>
                <menu>
                    <menuitem id="open" displayInContextMenu="true" />
                    <menuitem id="modify" displayInContextMenu="true" />
                    <menuitem id="delete" displayInContextMenu="true" />
                    <menuitem id="comment" allowed="false" />
                    <menuitem id="tag" allowed="false" />
                    <menuitem id="chat" allowed="false" />
                </menu>
            </node>
        </nodes>
    </plugin>
</KPEplugin>
```

Kuvio 13. Esimerkki liitännäisen konfiguraatiosta.

Liitännäisten ei tarvitse määritellä käyttöliittymäkomponentteja ollenkaan, vaan ne voivat käyttää suoraan KPE-järjestelmään määriteltyjä käyttöliittymäkomponentteja. Kuviossa 13 seuraavat field-elementit eli kentät käyttävät KPE-järjestelmään määriteltyjä käyttöliittymäkomponentteja: title, description, itemType, creator, created ja modified. PluginName- ja Summary-kentät vaativat uudet käyttöliittymäkomponentit, ja ne on määritelty uicomponent-elementin sisälle. Tarkempaa tietoa KPE-järjestelmän valmiiksi määritellyistä käyttöliittymäkomponenteista löytyy liitteestä 1.

4.4 Konfiguraatiotiedostojen käsittely

KPE-järjestelmän käynnistyessä ladataan ja käsitellään KPE-järjestelmän ja liitännäisten konfiguraatiotiedostot. Tieto object-config.xml-tiedostosta tallennetaan ObjectConfigSigleton-luokkaan, ja tieto valikoista tallennetaan MenuConfigSingleton-luokkaan. Ob-

jectConfigSingleton-luokka varastoi käyttöliittymäkomponenttien ja tietosolmujen tiedot. Item Info -sivupalkki ja lomake kysyvät luokalta esitettävän tietosolmun käyttöliittymäkomponentteja ja tietoja silloin, kun ne niitä tarvitsevat. MenuConfigSingleton-luokka tallentaa perusvalikon määrittelyn ja tarjoaa sen KPE-järjestelmän pää- ja pikavalikolle. Kun käyttäjä valitsee tietosolmun käyttöliittymässä, MenuConfigSingleton-luokka päivittää valikot ObjectConfigSingleton-luokan avulla tietosolmun määritelmien mukaiseksi.

PluginConfigSingleton-luokka käsittelee liitännäisen tarjoaman konfiguraatiotiedoston, kun liitännäinen ladataan KPE-järjestelmään. Luokka tallentaa liitännäisen tietosolmuihin ja käyttöliittymäkomponentteihin liittyvät määrittelyt ObjectConfigSingleton-luokkaan, ja tarvittaessa se tallentaa liitännäisen valikkoihin liittyvän määrittelyn MenuConfigSingleton-luokkaan. Itse PluginConfigSingleton-luokkaan tallennetaan viittaukset liitännäisiin ja muihin luokkiin (ObjectConfigSingleton-luokkaan ja MenuConfigSingleton-luokkaan), jotta liitännäiseen liittyvä tieto on helposti poistettavissa, jos liitännäinen otetaan pois käytöstä.

Nykyistä arkkitehtuuria voidaan helpommin laajentaa, koska liitännäisten ja KPE-järjestelmän valikkojen, tietosolmujen ja käyttöliittymäkomponenttien liittyvät määrittelyt tallennetaan samoihin luokkiin. Arkkitehtuuri voidaan esimerkiksi laajentaa tukemaan liitännäisiä, jotka laajentavat KPE-järjestelmän sisäisiä tietosolmutyyppejä eli lisäävät tietosolmuihin uusia kenttiä.

4.5 Liitännäisten tietokanta ja tiedon tallentaminen

4.5.1 Liitännäisten tietokanta

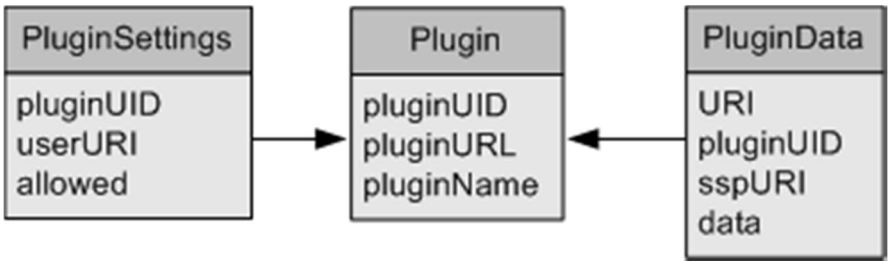
Liitännäisten tietoa ei ole mahdollista tallentaa järjestelmän semanttiseen tietokantaan tietokannan asettamien rajoitusten vuoksi. Lisäksi turvallisuussyiden vuoksi ei haluta, että liitännäisten kehittäjät pystyivät lisäämään ja hakemaan tietoa KPE-järjestelmän tietokannasta. Myöskään tiedon tallentamisen vastuun jättäminen liitännäisten kehittäjille ei ole mahdollista, sillä ei voida taata, kuinka kauan kehittäjä pystyy ylläpitämään

tietokantaa, ja lisäksi ei voida tietää, käyttääkö liitännäisen kehittäjä liitännäisellä tallennettuja tietoja hyväksi.

Mielekäs ratkaisu liitännäisten tietokantatoteutukseksi on tarjota KPE-järjestelmässä erillinen ja skaalautuva tietokanta. NoSQL-tietokannat ovat skaalautuvia ja suhteellisen nopeita. Ne sopivat erinomaisesti KPE-järjestelmän liitännäisten tietokannaksi. NoSQL-tietokannat eivät vaadi tallennettavalle tiedolle skeemaa, ja tällä tavoin KPE-järjestelmä ei rajoita mitenkään liitännäisten mahdollisuuksia tallentaa tietoa. Nopea vertailu eri NoSQL-tietokantojen välillä osoitti, että MongoDB on hyvä vaihtoehto liitännäisten tietokannaksi. [16, s. 247–248; 16.]

MongoDB on C-ohjelmointikielellä kirjoitettu ja avoimella lähdekoodilla toteutettu nopea tietokanta, jonka skeema on rajoittamaton eli skeema muuttuu tallennetun tiedon mukaiseksi. MongoDB:tä kehittää 10gen-yritys. MongoDB sopii hyvin järjestelmiin, jotka sisältävät paljon tietoa ja joiden tieto voi muuttua usein. Se tallentaa tiedon tietokantaan avain-arvo (key-value) -pareina, ja tietokanta skaalautuu tietokantaan tallennetun tiedon mukaan. MongoDB:n tietomalli on hyvin SQL (Structured Query Language) -tietomallin tapainen. MongoDB-järjestelmä muodostuu tietokannoista, jotka sisältävät kokoelmia. SQL-tietokannat sisältävät puolestaan tauluja. [17.]

KPE-järjestelmän liitännäisten tietokanta on toteutettu mahdollisimman yksinkertaisesti. Kuvio 14 esittää liitännäisten tietokantamallin.



Kuvio 14. KPE-järjestelmän liitännäisten tietokantamalli.

Kuvio 14 esittää liitännäisten tietokannan sisältämät kolme kokoelmaa: PluginSettings, Plugin ja PluginData. PluginSettings-kokoelmaan tallennetaan käyttäjän asetukset liitännäisiä varten; eli näytetäänkö käyttäjälle liitännäinen vai ei. Plugin-kokoelma sisäl-

tää liitännäisen perustiedot: tunnisteen, liitännäisen URL:n (Uniform Resource Locator) ja liitännäisen nimen. Kun KPE-järjestelmä käynnistyy, se lataa liitännäiset Plugin-kokoelmassa olevan tiedon avulla. PluginData-kokoelmaan tallennetaan tietosolmukohtaiset tiedot. PluginData-kokoelman datakenttään tallennetaan liitännäisen luoma tieto. Datakenttä sisältää kokoelman objekteja, ja näin ollen KPE-järjestelmä ei aseta rajoituksia liitännäisen tallentaman tiedon määrälle tai sen rakenteelle.

4.5.2 Liitännäisten web service -rajapinta

Tieto tallennetaan liitännäisten tietokantaan liitännäisten web service -rajapinnan avulla. Liitännäisten web service -rajapinta on toteutettu Java-ohjelmointikielellä, ja web service -rajapinnan avulla voidaan tehdä tietokantaan yksinkertaisia kyselyjä:

- uusien liitännäisten lisääminen
- liitännäisen poistaminen
- asennettujen liitännäisten hakeminen
- liitännäisten asetusten hakeminen (näytetäänkö liitännäinen käyttäjälle vai ei)
- liitännäisten tietosolmujen tiedon hakeminen
- liitännäisten tietosolmujen lisääminen
- liitännäisten tietosolmujen muokkaaminen
- liitännäisten tietosolmujen poistaminen.

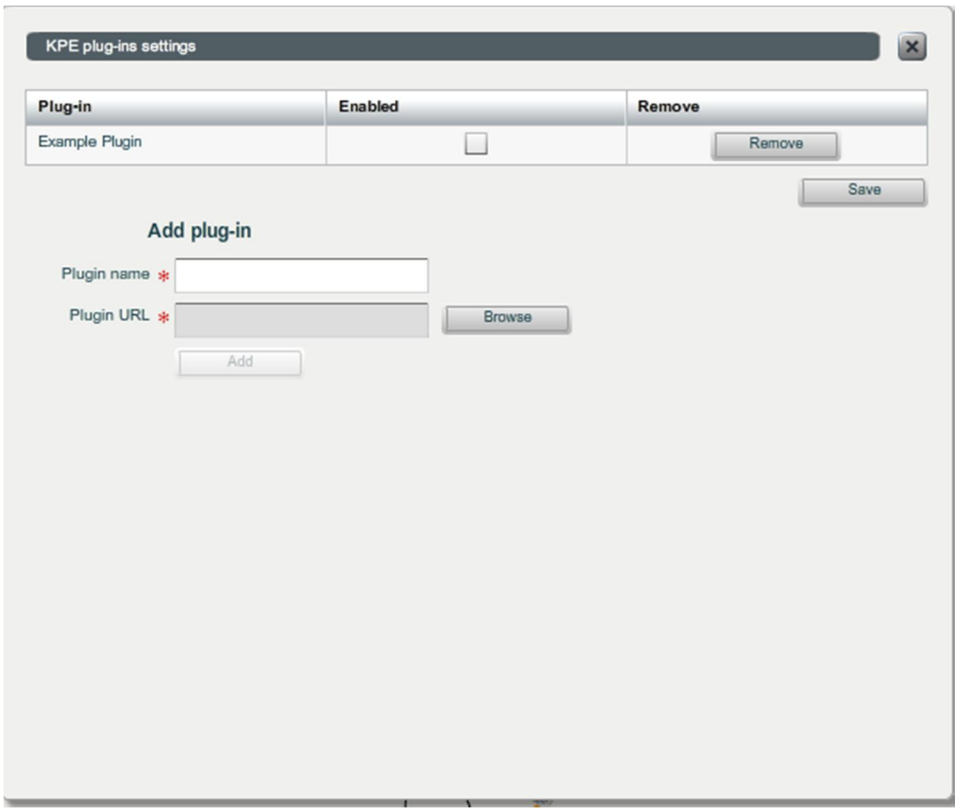
Tällä hetkellä ainoastaan liitännäisten tietosolmujen tieto tallennetaan tietokantaan ja liitännäisten asetuksista tallennetaan vain, onko käyttäjä aktivoinut liitännäisen vai ei. Liitännäisten kehittäjien kannalta voisi olla mieluisaa, jos liitännäisten työkalujen tietoja voitaisiin tallentaa tarvittaessa tietokantaan. Lisäksi käyttäjän näkökulmasta voisi olla mukavaa, jos liitännäiset voisivat tarjota muitakin asetuksia, kuin pelkästään sen, onko liitännäinen käytössä vai ei.

4.6 Liitännäisten näkyminen KPE-järjestelmässä

4.6.1 Liitännäisten hallintatyökalu

Ainoastaan järjestelmän ylläpitäjä voi lisätä uusia liitännäisiä KPE-järjestelmään. Käyttäjät eivät voi lisätä järjestelmään liitännäisiä, sillä KPE-järjestelmään ei ole toteutettu liitännäisen laadun tai turvallisuuden tarkastusta. Näin järjestelmän ylläpitäjän vastuulla on tarkastaa liitännäisen toimivuus järjestelmässä. Käyttäjät voivat ainoastaan ottaa ylläpitäjän hyväksymiä liitännäisiä käyttöön tai poistaa ne käytöstä. Jos järjestelmään myöhemmin kehitetään liitännäisten jakelua vasten sivusto, jonka avulla voidaan tarkastaa tai arvioida liitännäisen laatu ja turvallisuus, voidaan uudelleen harkita, annetaanko myös käyttäjien lisätä liitännäisiä järjestelmään.

Ylläpitäjä voi ladata järjestelmään uusia liitännäisiä liitännäisten hallintatyökalun avulla. Kuvio 15 esittää järjestelmän ylläpitäjälle näytettävän liitännäisten hallintatyökalun.

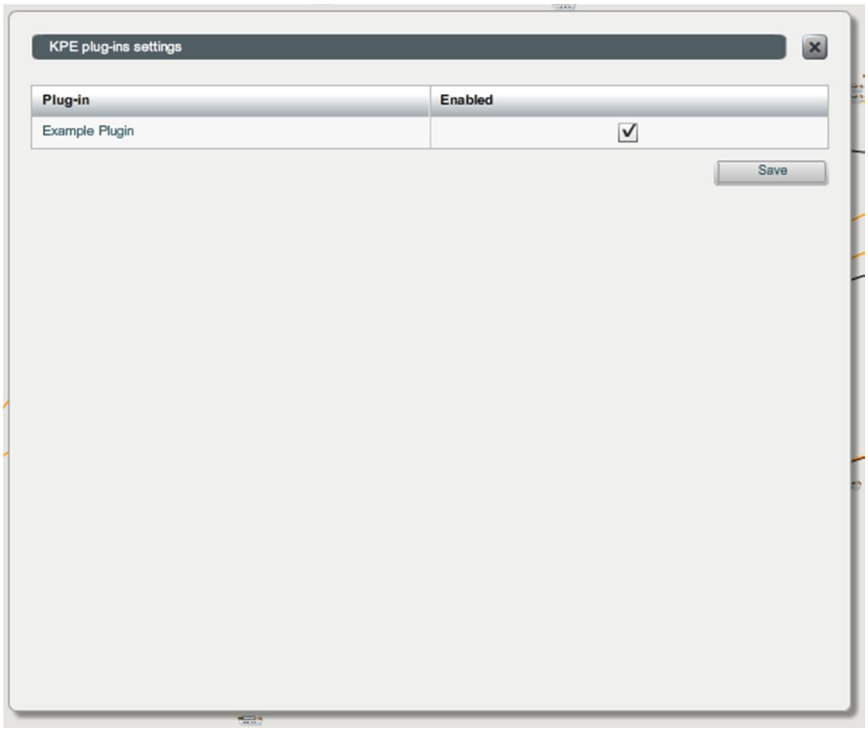


Kuvio 15. Järjestelmän ylläpitäjän liitännäisten hallintatyökalu.

Kuvio 15 esittää liitännäisten hallintatyökalun. Järjestelmään lisätyt liitännäiset esitetään liitännäisten hallintatyökalun yläosassa sijaitsevassa taulukossa. Kuviossa 15 olevasta taulukosta nähdään, että KPE-järjestelmä sisältää yhden Example Plugin -nimisen liitännäisen. Liitännäisen voi ottaa käyttöön tai pois käytöstä vaihtamalla Enabled-sarakkeessa olevaa arvoa. Liitännäisten poistaminen KPE-järjestelmästä on sallittu ainoastaan järjestelmän ylläpitäjälle. Liitännäisen voi poistaa järjestelmästä Remove-napilla.

Ylläpitäjä voi lisätä uuden liitännäisen kuviossa 15 näkyvällä Add plug-in -lomakkeella. Lomakkeeseen syötetään liitännäisen nimi ja liitännäisen flash-tiedosto eli modulen SWF -tiedosto ylläpitäjän paikalliselta levyltä. Tiedosto ladataan palvelimelle, ja liitännäiseen liittyvä tieto tallennetaan tietokantaan. Liitännäiset lisätään järjestelmään ajon aikana, ja käyttäjät voivat ottaa liitännäiset heti käyttöön.

Tavalliselle käyttäjälle ei näytetä liitännäisten lisäyslomaketta, eikä käyttäjä myöskään voi poistaa liitännäisiä järjestelmästä. Kuvio 16 näyttää tavalliselle käyttäjälle esitettävän KPE plug-ins settings -hallintatyökalun.

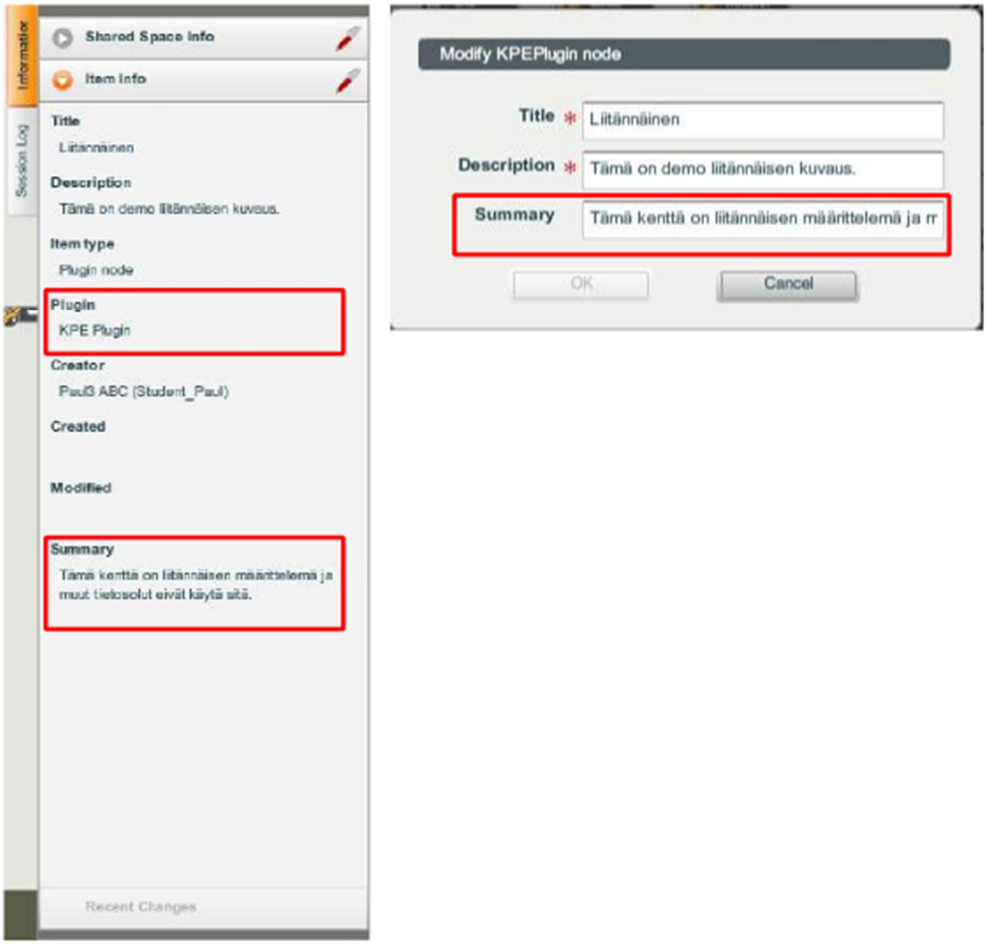


Kuvio 16. Tavallisille käyttäjille esitettävä liitännäisten hallintatyökalu.

Kuten kuviosta 16 nähdään, tavallinen käyttäjä voi hallita liitännäisiä ainoastaan ottamalla ne käyttöön tai poistamalla ne käytöstä. Oletusarvoisesti kaikki liitännäiset ovat pois käytöstä.

4.6.2 Liitännäisten lomakkeet ja tietosolmut

Nykyisen arkkitehtuurin avulla liitännäiset liitetään järjestelmään saumattomasti, eivätkä ne enää korostu kokonaisuudesta. Liitännäiset eivät erotu KPE-järjestelmän muista toiminnallisuuksista ja tietosolmuista, sillä ne käyttävät samoja toteutuskomponentteja kuin KPE-järjestelmän tietosolmut ja työkalut. Kuviosta 17 nähdään, kuinka liitännäisten tietosolmut näytetään KPE-järjestelmän Item Info -sivupalkissa ja lomakkeissa.



Kuvio 17. Liitännäisten uuden tiedon näyttäminen Item Info -sivupalkissa ja lomakkeessa.

Kuviossa 17 on punaisilla suorakulmioilla korostettu liitännäisen uusia kenttiä. Esimerkkinä käytettävä liitännäisen tietosolmu on lisännyt KPE-järjestelmään kaksi uutta kenttää: Plugin ja Summary. Kuviossa 17 käytetty liitännäinen käyttää kuviossa 13 esitettyä konfiguraatiotiedostoa uusien kenttien määrittelyyn.

Liitännäisten tietosolmuille on määritelty oletusulkoasu, jonka avulla liitännäiset erottuvat KPE-järjestelmän muista tietosolmuista. Kuviossa 17 esimerkkinä käytetty tietosolmu esitetään kuvion 18 mukaisesti KPE-järjestelmässä.

Kuvio 18. Liitännäisten tietosolmun graafinen esitys KPE-järjestelmässä.

Kuvion 18 liitännäinen käyttää KPE-järjestelmän määrittelemää oletusulkoasua tietosolmuissaan. Liitännäisten tietosolmut erottaa helposti muista KPE-järjestelmän tietosolmuista palapelikuvakkeen ansiosta. Liitännäisen kehittäjän ei tarvitse käyttää oletusulkoasuja, vaan kehittäjä voi luoda myös oman ulkoasun liitännäisen tietosolmulle. Oletusulkoasu on toteutettu helpottamaan liitännäisten kehitystyötä.

5 Liitännäisten jatkokehitys

Liitännäisten tukeminen muissa KPE-järjestelmän työkaluissa

Toteutettu liitännäisarkkitehtuuri sai hyvän vastaanoton niin projektin kuin EU:n arvioijilta. Liitännäisarkkitehtuuri on otettu käyttöön, ja siihen kehitetään jo liitännäisiä.

Nykyinen liitännäisarkkitehtuuri toteutettiin tukemaan vain muutamaa KPE-järjestelmän sisältämää työkalua, mutta KPE-järjestelmä sisältää runsaasti muita käyttäjille varsin hyödyllisiä työkaluja. Suurin osa KPE-järjestelmän työkaluista tallentaa tietoja

KPE-järjestelmän semanttiseen tietokantaan, ja koska semanttisen tietokannan rajoitukset pakottavat tallentamaan liitännäisten tiedon erilliseen tietokantaan, ei kaikkia työkaluja voida käyttää liitännäisten tietosolmujen osalta. Seuraavat työkalut ovat käyttäjälle varsin hyödyllisiä, joten niiden tuki liitännäisille olisi hyvä toteuttaa.

Semanttisen haku -työkalun avulla voidaan hakea varsin kattavasti tietoa KPE-järjestelmän sisällöstä. Työkalun tulisi esittää myös liitännäisiin liittyvä tieto hakutuloksissa. Välttämättä ei tarvitsisi toteuttaa semanttista hakua tukemaan täydellisesti liitännäisten hakemista, vaan riittäisi, jos käyttäjä voisi hakea haulla liitännäisten tietosolmujen otsikoiden, kuvauksen ja tietosolmutyypin mukaisesti.

HPA (History and Participation Awareness) -työkalulla näytetään käyttäjälle ilmoituksia uusista muutoksista ja pystytään tallentamaan ja näyttämään tiedot viimeisimmistä muutoksista. Loppukäyttäjän näkökulmasta tämä työkalu on erittäin tärkeä, ja olisikin hyvä, jos HPA-työkalu pystyisi seuraamaan liitännäisten tietosolmujen muutoksia.

Kommentointi (Commenting) -työkalulla voidaan liitää tietosolmuihin kommentteja. Kommentointityökalua ei voida tällä hetkellä käyttää liitännäisten tietosolmuissa. Se kuitenkin on käyttäjien kannalta hyödyllinen työkalu, ja olisikin tärkeää, että liitännäisten tietosolmuja voisi kommentoida.

Taggaus (Tagging) -työkalulla voidaan tietosolmuihin liittää tagejä (tag). Tagien käyttö helpottaa KPE-järjestelmään lisätyn tiedon hahmottamista, koska tagejä hyödynnetään esimerkiksi tietosolmujen suodattamisessa (filter) ja muutamassa työkalussa. Olisikin hyvä, jos tagejä voisi lisätä liitännäisten tietosolmuihin ja näin ollen liitännäisten tietosolmut voitaisiin ainakin suodattaa.

KPE-järjestelmässä käyttäjät työskentelevät synkronoidusti. Käyttäjät siis näkevät toisten käyttäjien toiminnan reaaliajassa. Esimerkiksi tietosolmujen liikuttaminen käyttöliittymässä näkyy kaikille käyttäjille. Liitännäisiin liittyviä toimia ei ole kaikilta osin synkronoitu. Esimerkiksi liitännäisten tietosolmujen liikuttaminen ja lukitseminen on synkronoitu, mutta tietosolmujen muokkaaminen, poistaminen ja lisääminen eivät ole.

Tietokannan ja web service -rajapinnan optimointi ja turvallisuus

Liitännäisten tietokanta on pyritty toteuttamaan mahdollisimman yksinkertaisesti. Kuitenkaan tietokannan ja web service -rajapinnan nopeutta ja turvallisuutta ei ole testattu kovin järjestelmällisesti. Lisäksi tietokanta sisältää tällä hetkellä hyvin pienen määrän tietoa, joten olisi hyvä testata tietokantaa ja optimoida web service -rajapintaa myös suuremmalla tietomäärällä.

Liitännäisten jakaminen

Liitännäisille ei ole toteutettu jakelujärjestelmää. Tällä hetkellä voidaan sanoa, että KPE-järjestelmän liitännäiset jaellaan villisti, koska niille ei ole toteutettu sivustoa, jonka avulla liitännäisten jakelu voitaisiin keskittää. Keskitetty jakelu olisi käyttäjille ja kehittäjille helpompi, mutta sivuston toteuttaminen ja ylläpito vaativat enemmän töitä. Lisäksi liitännäisten valvominen keskitetyssä jakelussa onnistuu helpommin, kun liitännäiset on keskitetty yhtyeen paikkaan. Tällöin voi sivuston ylläpitäjä tarkistaa, että liitännäiset ovat toimivia ja luotettavia, tai vaihtoehtoisesti luotettavuus ja toimivuus voidaan mitata käyttäjien arvioinnin mukaan.

Liitännäisten ja rajapintojen versiointi

Liitännäisten yhteensopivuus taaksepäin on erittäin tärkeä asia. Flex-module-komponenttien käyttäminen asettaa joitakin rajoituksia taaksepäinyhteensopivuuteen. Flexin SDK:n vaihtaminen isäntäsovelluksessa voi vaikuttaa liitännäisiin niin, että niiden Flex SDK täytyy myös vaihtaa samaan kuin isäntäsovelluksen. Tähän ongelmaan ei varmaankaan löydy yksiselitteistä ratkaisua, ennen kuin Adobe kehittää sellaisen.

Rajapintojen muuttuminen aiheuttaa sen, että vanhalla rajapinnalla toteutetut liitännäiset eivät toimi isäntäsovelluksessa. Tämän vuoksi rajapinnat täytyy versioida. Flex RSLs -kirjastot saattavat ratkaista rajapintojen muuttumisen, sillä isäntäsovellukseen voidaan lisätä useita RSLs-kirjastoja. Tätä ei kuitenkaan ole vielä testattu ja todettu toimivaksi.

6 Yhteenveto

Insinöörityössä selvitettiin liitännäisten olemus ja liitännäisarkkitehtuurin määritelmä sekä toteutettiin liitännäisarkkitehtuurin määritelmää vastaava arkkitehtuuri KPE-järjestelmään. KPE-järjestelmä on verkossa toimiva yhteisöllinen ja virtuaalinen oppimis- ja työskentely-ympäristö, joka kehitettiin KP-Lab-projektissa. Projektin teknisenä tavoitteena oli toteuttaa modulaarinen, joustava ja laajennettava tietojärjestelmä, mutta aikaisemmalla arkkitehtuurilla toteutettu järjestelmä ei ollut kovin modulaarinen ja laajennettava.

Liitännäisarkkitehtuurin avulla järjestelmästä saadaan modulaarinen ja laajennettava, mutta laajennettavuuden toteuttaminen vaatii hyvin paljon suunnittelu- ja määrittelytyötä. Liitännäisarkkitehtuuri muodostuu isäntäsovelluksesta ja liitännäisistä. Liitännäiset liitetään isäntäsovellukseen dynaamisesti, joko ajon aikana tai järjestelmän käynnistyksen yhteydessä. Liitännäiset eivät ole itsenäisiä sovelluksia, joten isäntäsovellus määrittelee tarvittavat rajapinnat, joiden avulla liitännäiset pystyvät toimimaan isäntäsovelluksessa ja kommunikoimaan isäntäsovelluksen kanssa. Ominaispiirteenä liitännäisarkkitehtuurin isäntäsovellukselle on, että liitännäiset pystytään lisäämään ilman, että isäntäsovelluksen lähdekoodia joudutaan muuttamaan.

Liitännäisarkkitehtuurin toteuttaminen sisältää paljon ongelmallisia asioita, joihin täytyy löytyä ratkaisu liitännäisarkkitehtuuria toteuttaessa: liitännäisten ja isäntäsovelluksen sopimuksen määrittely eli rajapintojen määrittely, liitännäisten rajapintojen versiointi, liitännäisten kehitys ja lataus, liitännäisten laadullinen hyväksyminen ja liitännäisten turvallisuus. Lisäksi liitännäisarkkitehtuuri toteuttaminen sisältää muita ongelmia, jotka liittyvät liitännäisten lataamiseen, tunnistamiseen ja hallitsemiseen itse järjestelmässä. Liitännäisarkkitehtuurin toteutus on hyvin alustariippuvainen; jotkin ohjelmointikielet tarjoavat valmiita luokkia ja kirjastoja liitännäisarkkitehtuurin toteuttamista varten, toiset ohjelmointikielet eivät. Siispä liitännäisarkkitehtuurin toteuttamisen vaatima työn määrä saattaa vaihdella ohjelmointikielestä riippuen.

Liitännäisarkkitehtuureista ja liitännäisistä tehdyn selvitystyön perusteella määriteltiin ja toteutettiin liitännäisarkkitehtuuri KPE-järjestelmään. Arkkitehtuuri ei ole täydellinen, eikä se ole aivan valmis. KPE-järjestelmän liitännäisarkkitehtuuri toteutettiin vain järjes-

telmän perustoiminnallisuuksien osalta. Myös liitännäisarkkitehtuurin vaatimukset asetettiin mahdollisimman yksinkertaisiksi, mutta myös tarpeeksi laajoiksi, jotta arkkitehtuurista olisi järjestelmälle ja kehittäjille jotakin hyötyä.

KPE-järjestelmän liitännäisarkkitehtuurin toteuttamisessa oli haasteita, joiden ratkaisu piti tarkkaan miettiä. Adobe Flex tarjoaa valmiin ratkaisun liitännäisten dynaamiselle lataamiselle module-tekniikalla, mutta liitännäisten dynaamiseen käsittelyyn ja uuden tiedon liittämiseen täytyi määritellä siihen sopiva konfiguraatio. Myös liitännäisten tiedon tallentamisen toteuttaminen oli haastavaa, sillä KPE-järjestelmän tietokanta ei nykyisellä mallillaan sovi liitännäisten tiedon tallentamiseen. Liitännäiset tarvitsevat nopean ja skaalautuvan tietokannan, jolle ei ole määritelty valmista tietomallia. NoSQL-tietokannat skaalautuvat tallennettavan tiedon mukaan, joten ne sopivat hyvin ennalta määrittelemättömän tiedon tallentamiseen. Liitännäisten tiedolle ei voida määritellä valmista tietomallia, sillä ei voida etukäteen tietää, mitä ne haluavat tallentaa.

Toteutettu liitännäisarkkitehtuuri osoittautui menestykseksi, ja se aiotaan virallisesti liitää osaksi KPE-järjestelmää. Järjestelmään on jo alettu kehittää uusia liitännäisiä englanninkielisen kehittäjänoppaan avulla. Arkkitehtuuria suunniteltaessa ja toteutettaessa otettiin huomioon myös sen mahdollinen laajennettavuus. Työssä toteutetun arkkitehtuurin rajausten ulkopuolelle jäi paljon KPE-järjestelmän toiminnallisuuksia ja työkaluja, joiden tuki tulee pystyä lisäämään arkkitehtuuriin suhteellisen helposti. Työssä myös esiteltiin arkkitehtuurin jatkokehityksen kannalta varteenotettavia asioita, kuten esimerkiksi liitännäisten ja rajapintojen versiointi ja liitännäisten jakelu.

Lähteet

1 Wolfinger, R., Reiter, S., Dhungana, D., Grunbacher, P., Prahofer, H. 2008. Supporting Runtime System Adaptation through Product Line Engineering and Plug-in Techiques. Teoksessa Seventh International Conference on Composition-Based Software Systems, 2008. ICCBSS 2008. IEEE.

2 Chatley, R., Eisenback, S., Magee, J. 2003. Modelling a Framework for Plugins. Teoksessa Proceedings of the Workshop on Specification and Verification of Component-Based Systems, ser. Technical Report 03-11. Iowa State University.

3 Kharrat, D., Qadri, S. 2005. Self-registering plug-ins: an architecture for extensible software. Teoksessa Canadian Conference on Electrical and Computer Engineering, 2005. IEEE.

4 Mayer, J., Melzer, I., Schweiggert, F. 2002. Lightweight Plug-In-Based Application Development. Teoksessa Aksit, M., Mezini, M., Unland, R. (eds.). Objects, components, architectures, services, and applications for a networked world: International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7-10, 2002: revised papers. Springer.

5 Marquardt, Klaus. 2006. Patters for plug-ins. Teoksessa Manolescu, Dragos-Anton, Voelter, Markus, Noble, James. Pattern Languages of Program Design 5. Boston: Pearson Education, Inc.

6 Chatley, R., Eisenback, S., Magee, J. 2003. Painless Plugins. Verkkodokumentti. Imperial College London. <http://www.doc.ic.ac.uk/~rbc/writings/pp.pdf>. Luettu 21.1.2011.

7 Wolfinger, R., Prähofer, H. 2007. Integration Models in a.NET Plug-in Framefork. Teoksessa SE 2007 - the Conference on Software Engineering, Hamburg, Germany, March, 27-30, 2007.

8 Wolfinger, R., Dhungana, D., Prähofer, H.; Mössenböck, H. 2006. A Component Plug-in Architecture for the .NET Platform. Teoksessa Lightfoot, D., Szyperski, C. (eds.). Modular Programming Languages 7th Joint Modular Languages Conference, JMLC 2006 Oxford, UK, September 13-15, 2006 Proceedings. Berlin: Springer.

9 Java Plug-in Framework (JPF) Project. 2007. Verkkodokumentti. JPF Team. <http://jpf.sourceforge.net/>. Päivitetty 2.7.2007. Luettu 21.1.2011.

10 Adobe Flex 3 Developer Guide. 2008. Verkkodokumentti. Adobe System Incorporated. <http://livedocs.adobe.com/flex/3/devguide_flex3.pdf>. Luettu 23.1.2011.

11 Using Adobe® Flex™ 3.2. 2008. Verkkodokumentti. Adobe System Incorporated. <http://livedocs.adobe.com/flex/3/loading_applications.pdf>. Luettu 21.1.2011.

12 Clayberg, E., Rubel, D. 2004. Eclipse Building Commercial Quality Plug-ins. Boston: Pearson Education.

13 RDF Primer. 2004. Verkkodokumentti. W3C. <http://www.w3.org/TR/rdf-syntax/>. Luettu 16.5.2011.

14 Rantakari, Ali. 2008. SSGUI plugin architecture. Verkkodokumentti. KP-Lab. <http://trac.kp-lab.org/browser/wp6/docs/SSGUI/SSGUI-plugin-architecture.png>. 28.4.2008. Luettu 21.1.2011.

15 Noble, J., Anderson, T., Braithwaite, G., Casario, M., Tretola, R. 2010. Flex 4 Cookbook. Sebastobol, California: O'Reilly Media.

16 Florenzano, Eric. 2010. Nonrelational Databases. Teoksessa Allspaw, John, Robbins, Jesse (eds.). Web Operations: Keeping the Data on Time. Sebastobol, California: O'Reilly Media.

17 MongoDB. Verkkodokumentti. 10gen, Inc. <http://www.mongodb.org>. Luettu 23.1.2011.

KPE-järjestelmän liitännäisten kehittäjänopas

KNOWLEDGE PRACTICES LABORATORY

KPE plug-in developer's guide

Guide for developing KPE plug-ins

Eini Saarivesi
27.3.2011

Table of Contents

Attachments

1 Introduction

1.1 Overview

KPE plug-ins are made with Flex Modules. Modules are .swf files that can be loaded and unloaded during the run-time. KPE plug-ins require KPE plug-in framework library. KPE plug-in framework is a RSLs file that contains all the required interfaces and classes for developing KPE plug-ins. KPE plug-in must implement ISSGUIPlugin interface (which is included into KPE plug-in framework) and the plug-in must provide configuration file so that it can be loaded into KPE.

Plug-in's node data will be saved into NoSQL database, which is located in KPE. The plug-in developer doesn't have to worry about saving and retrieving data from the database; the KPE does that for you.

The Flex Builder was used when this document was made so if you are using Flash Builder, there might be some differences how things are done in Flash Builder.

What you can do with KPE plug-ins:

- Create new tools into KPE.
- Create new types of nodes into KPE.

1.2 Links

- More information about Adobe Flex: Flex online help <http://livedocs.adobe.com/flex/3/html/help.html?content=Part2_DevApps_1.html> and Flex developer guide <http://livedocs.adobe.com/flex/3/devguide_flex3.pdf>.
- The "Creating Modular Applications" document <http://blogs.adobe.com/flexdoc/files/flexdoc/pdfs/modular.pdf > is one chapter of the Flex Developer's Guide. It contains very useful information about how to create modules.

1.3 Requirements

- Adobe Flex builder or Adobe Flash builder.

- To create plug-ins you have to use Flex 3.5 SDK. If you don't have SDK in your Flex Builder or Flash builder you can download it from <http://opensource.adobe.com/wiki/display/flexsdk/Download+Flex+3>.
- KPE in some server (e.g. Mielikki). (Modules cannot be used as stand-alone applications, so you have to upload it to KPE to test and use your plug-in).
- You have to use KPE plug-in framework in your plug-in project and your plug-in must implement ISSGUIPlugin interface. The KPE plug-in framework library contains the ISSGUI interface.

IMPORTANT: You have to use Flex 3.5 SDK in your Flex project. If you are using some other Flex SDK, the plug-ins will not work. Check your SDK from Flex upper menu: "Project"-> "Properties" and select "Flex Compiler" link. The first section of the window will tell you what the Flex SDK is. If it is Flex 3.5 you are good to go forward.

2 Creating new KPE plug-in

2.1 Creating new Flex project

To create new flex project in Flex Builder click "File" -> "New" -> "Flex Project". The "New Flex project" – window is opened. Window is displayed in figure 1.

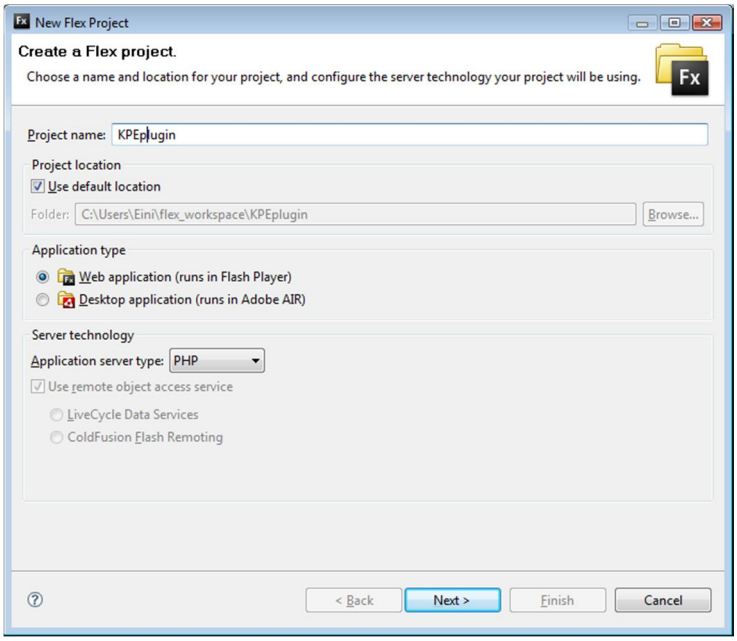


Figure 1: The "New flex project" - window.

Give your project a name and click “Next” button. In next window, define your server location (most probably localhost) and output folder. If you don’t know what to put in here, check the Flex developer guide or Flex online help (links can be found in introduction chapter).

After you have defined your server location and output folder click “Next” button. In next window you will add the KPE plug-in framework library into your project. The figure 2 displays the window where you can add the KPE plug-in framework library into your project.

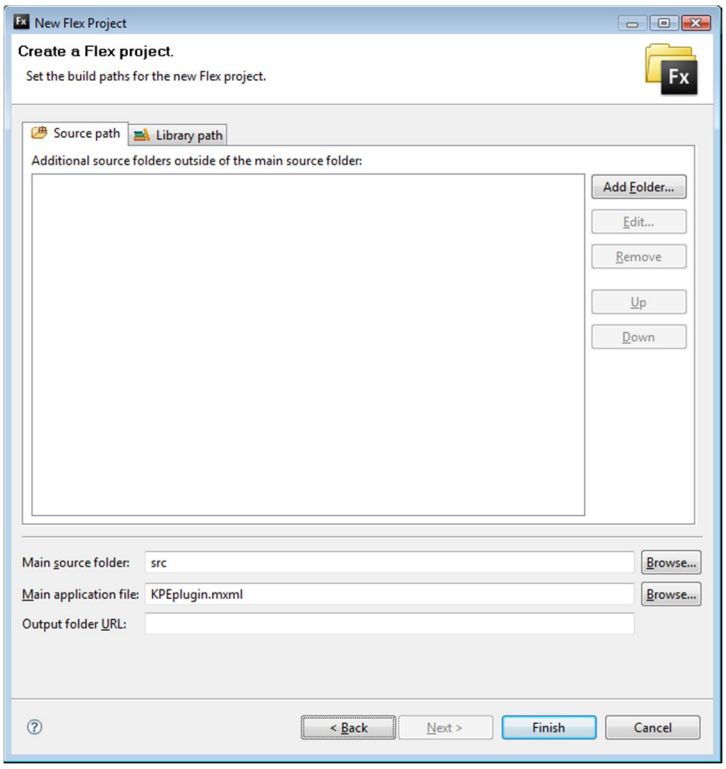


Figure 2: Window for adding the KPE plug-in framework into the project.

Change to the “Library path” tab and click “Add SWC…” button. Browse the KPE plug-in framework SWC file from your computer and click “OK”. Your “New Flex Project” window should look now like figure 3.

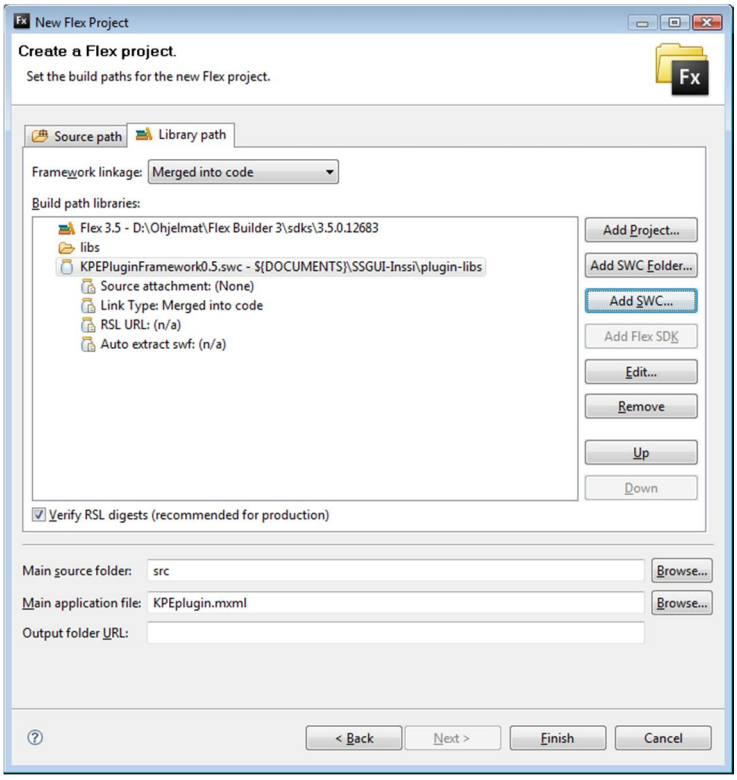


Figure 3: KPE plug-in framework is added into project.

Okay, you are almost done. You still need to define that the KPE plug-in framework is RSLs file. To do that, double click the “Link Type” text. And following window should open:

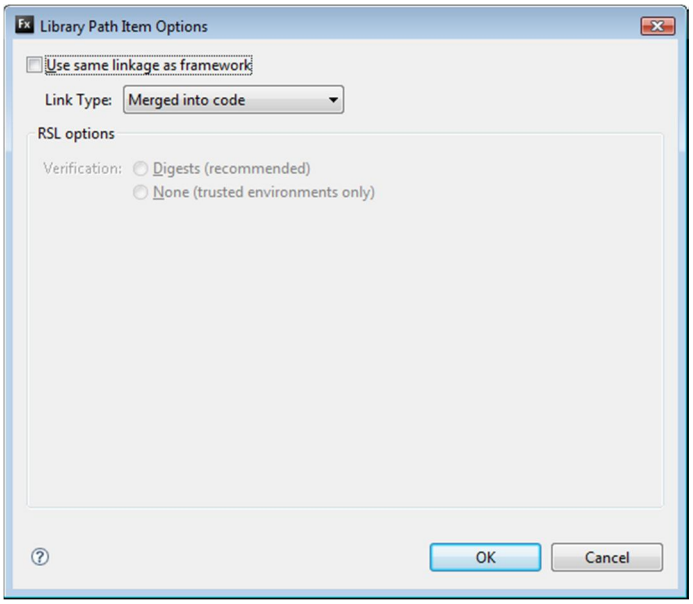


Figure 4: Defining the link type for the KPE plug-in framework.

Change the link type from the “Merged into code” to “Runtime Shared Library (RSL)”. Keep the Verification settings in “None (trusted environments only)” option and into the Deployment path/URL write “plugins/<and the swc file name>”. Uncheck the check box “Automatically extract the swf to deployment path”. The figure 5 displays how your settings should look now.

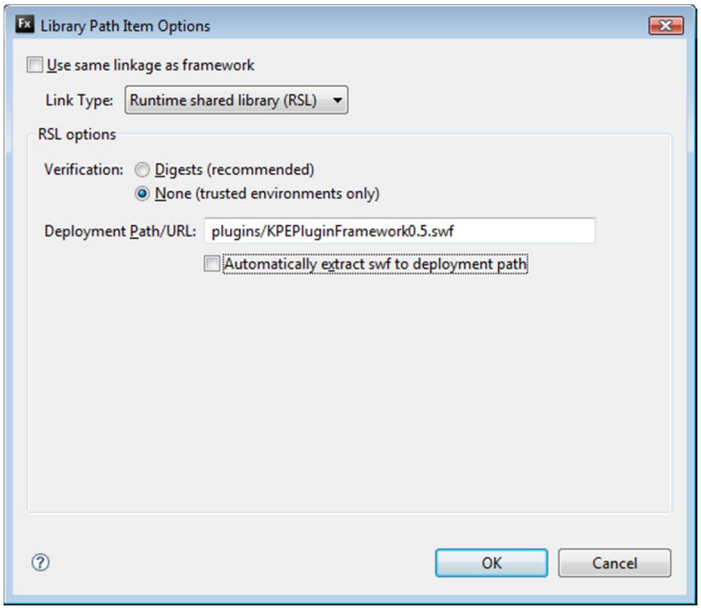


Figure 5: Settings for the KPE plug-in framework.

If your settings for KPE plug-in framework look like in figure 5, click “Ok” button. Click “Finish” button in “Add Flex Project” window and you are done for creating Flex project.

Now you can create the Flex module. Next chapters will briefly give instructions how to create module that implement ISSGUI interface. If you need more information how to create and develop modules, please check *Flex developer guide*, *Creating modular application document* or/and *Flex online help* (links can be found from the Introduction chapter).

2.2 Creating ActionScript based module

It is highly recommended to create your KPE plug-ins by using ActionScript based modules, because when you create new (module) class, the Flex creates required interface methods into your class. You can also create mxml based module but you have

to manually define the methods that are required to be implanted in ISSGUIPlugin interface. (Check Flex developer guide, Flex online help and/or “Creating Modular Applications” document to get more information about how to create mxml modules if you want to).

To create new ActionScript based module select “File” -> “New” -> “ActionScript Class”. Image 6 displays the settings you have to add to add new module.

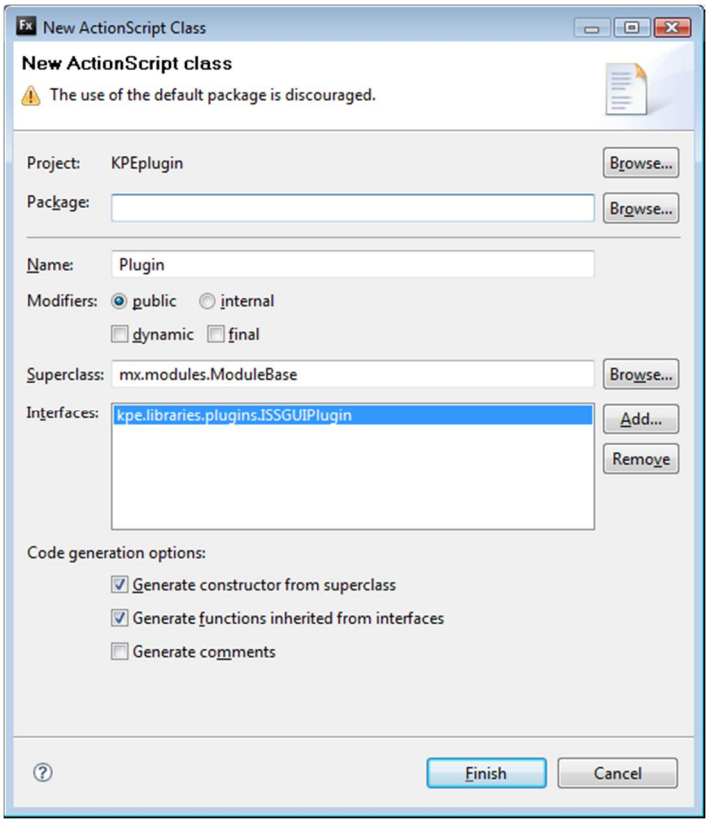


Figure 6: Displays create new ActionSript module class.

As the figure 6 displays, you have to type your module name into “Name” field. This name will be the name of your KPE plug-in. In the “Superclass” field browse and select ModuleBase option. And in the “Interfaces” field select ISSGUIPlugin class by pressing the “Add” button and selecting ISSGUIPlugin from the list. Check that the “Generate functions inherited from interfaces” is checked and click “Finnish” button.

Now you have created KPE module.

2.3 Defining the module to be build when project is build

Your module must be build to create .swf file that you can add into KPE. To make module buildable you have to configure your project properties. From the Flex builder upper menu, select “Project” -> “Properties”. From the properties window click “Flex module” link. Your properties window should look like figure 7.

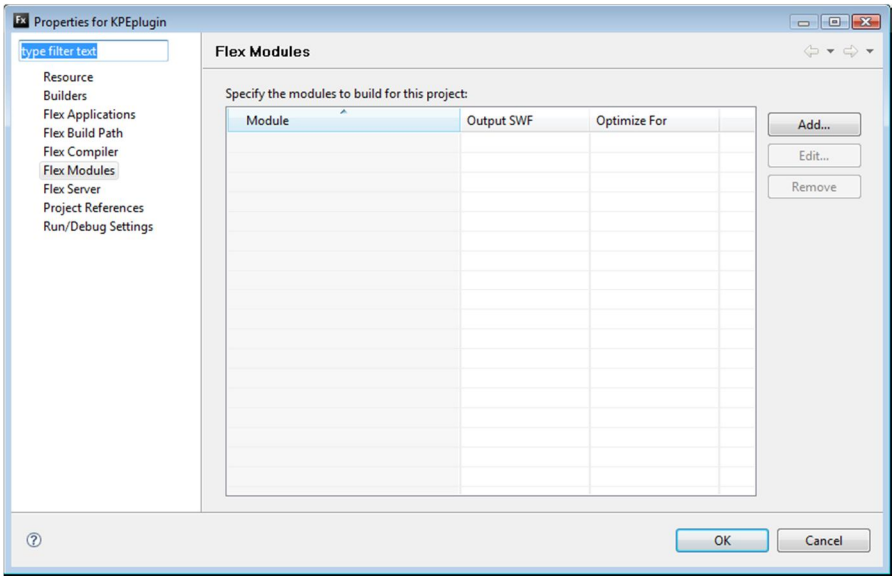


Figure 7: Project properties window when Flex module option is selected.

From the right side of the window click “Add...” button. Browse your newly created module from the list and click “Ok” button. Select “Do not optimize (module can be loaded by multiple applications)” option. Figure 8 displays the settings for the module.

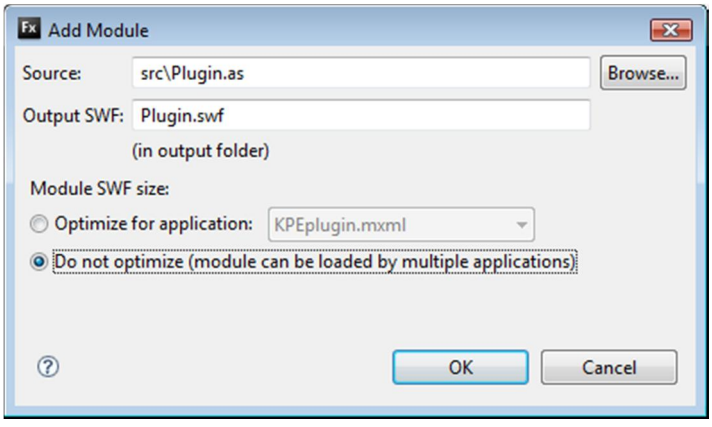


Figure 8: Module settings for KPE plug-in.

If your settings are same as in figure 8, then press “OK” button. Now your module will be build into .swf file when you build your project.

When you created your Flex project, the Flex created .mxml file that is named as your project. Because you need only the module for KPE you should delete the .mxml file from your project (you don’t need it). Now if you build your project (right click above your project name and select “Build Project”), the .sfw file should appear into your output folder (the folder you defined when you created the project, by default it is “bin-debug” folder).

Next step is configuring your KPE plug-in.

3 Configuring the KPE plug-in

Because KPE is not aware what the plug-in does in KPE you have to create configuration file for your plug-in. Create “plugin-config.xml” file into your project. Create xml file from “File”-> “New” -> “File”. Type “plugin-config.xml” into text field and click “Finish” button. The syntax for plug-in configuration file is displayed in figure 9.

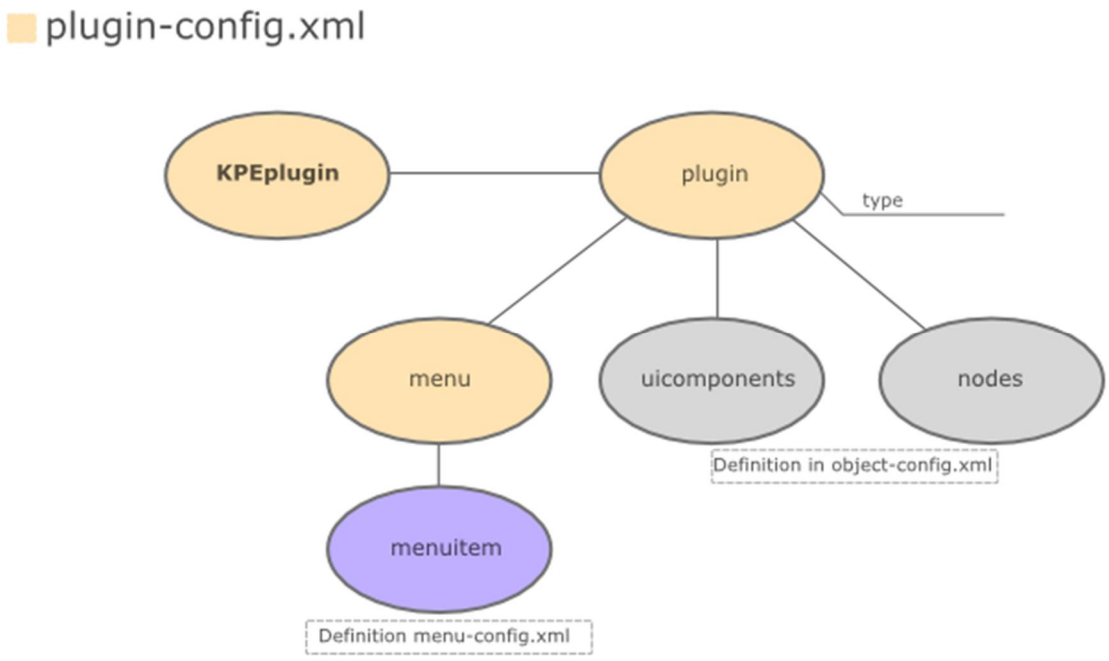


Figure 9: The syntax for plug-in configuration file.

The root element for the configuration file is *KPEplugin*. The root element contains *plugin* child element that contains configuration for tool menu, UI components and plug-in nodes.

Figure 10 displays example configuration for plugin-config.xml file.

```
<KPEplugin>
    <plugin type="tool">
        <menu>
            <menuitem   label="Open Example Plugin"
                        displayView="PROCESS_VIEW,CONTENT_VIEW,ALTERNATIVE_PROCESS_VIEW,TAILORED_VIEW,COMMUNITY_VIEW"
                        action="openTool" data="examplePlugin"/>
        </menu>
        <uicomponents>
            <uicomponent field="PluginName" parameter="plugin,pluginName" label="Plugin">
                <form clazz="" type=""/>
                <inline clazz="ssgui.libraries.adapter" type="AdapterField" field="LABEL" disabledField="LABEL" />
            </uicomponent>
        </uicomponents>
        <nodes>
            <node   name="PluginNode"
                    typeID="http://www.examplePlugin.org/ontologies/PluginNode#BaseNode"
                    label="Excample Plugin node"
                    displayView="CONTENT_VIEW,ALTERNATIVE_PROCESS_VIEW,TAILORED_VIEW"
                    clazz="ssgui.libraries.graphModel.PluginNode" type="TYPE" itemtype="PluginNode">
                <fields>
                    <field type="title" required="true" />
                    <field type="description" required="true" />
                    <field type="itemType" required="false" editable="false"/>
                    <field type="PluginName" required="false" editable="false" />
                    <field type="creator" editable="false" />
                    <field type="created" editable="false" />
                    <field type="modified" editable="false" />
                </fields>
                <menu>
                    <menuitem id="comment" allowed="false" />
                    <menuitem id="tag" allowed="false" />
                </menu>
            </node>
        </nodes>
    </plugin>
</KPEplugin>
```

Figure 10: Example for plug-in configuration.

Following chapters will go though the elements in configuration file. Figure 11 displays the more detailed structure of the elements and relations for the elements used in configuration file.

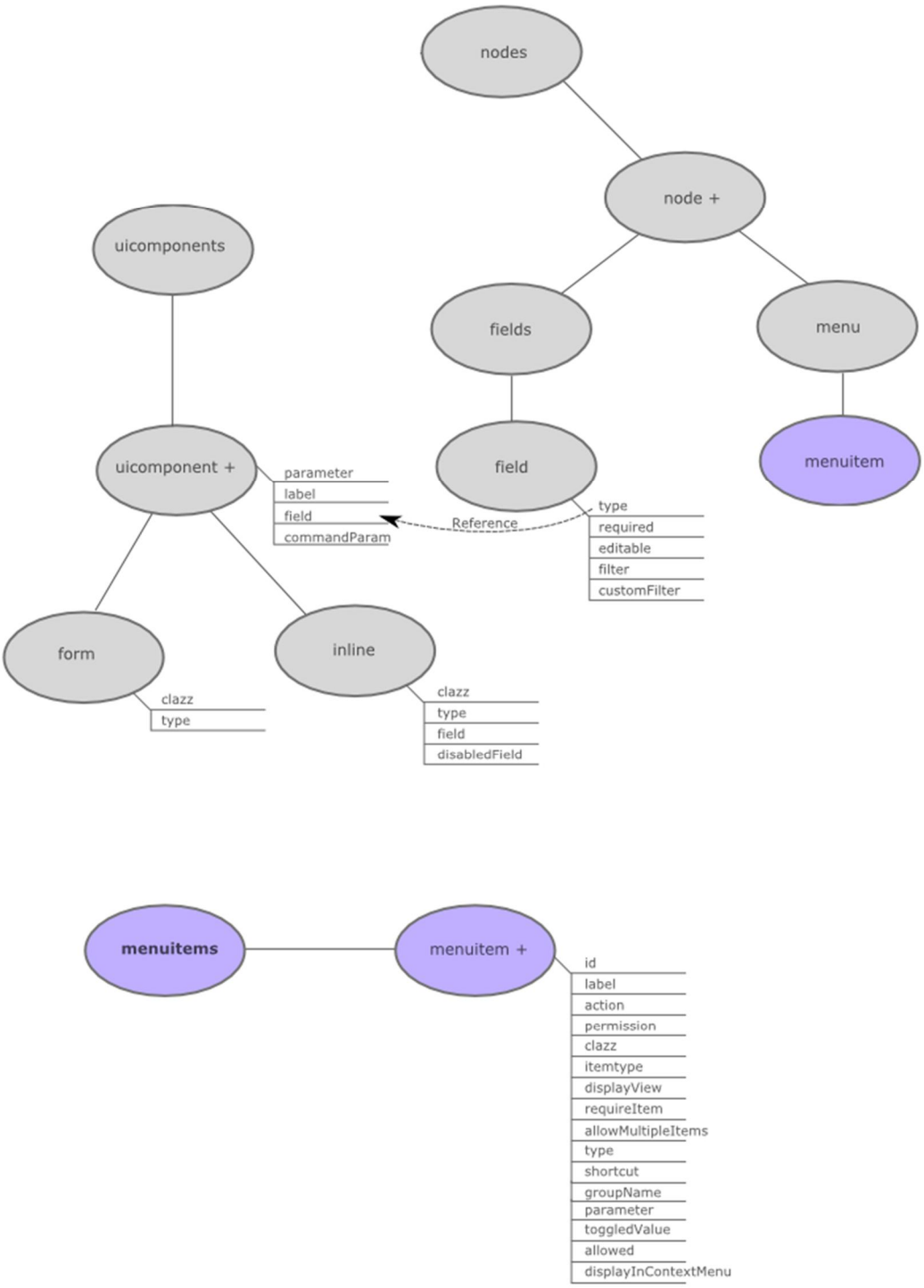


Figure 11: The structure and references between elements used in configuration file.

3.1 Plug-in element

All the plug-in related configuration data is inside the *plugin* element. The plug-in element can contain only one attribute: *type*. The *type* attribute defines if the plug-in is displayed in KPE tools menu. If your plug-in is tool, just add *type="tool"* text into *plugin* element. If the type attribute is not defined or the type attribute has some other value than "tool" the attribute will be ignored and the plug-in is not displayed in KPE Tools menu.

3.2 Menu element

The *menu* element is for the KPE Tools menu, if the *plugin* element "type" attribute is "tool". The *menu* element can have only one *menuitem* element and the *menuitem* element defines configuration for the KPE tools menu. You can add following attributes to *menuitem* element: *label, displayView, action* and *data*.

Label attributes defines the text that is displayed in Tools menu. For example in figure 10, the label attribute for the KPE Tools menu is "Open Example Plugin" and in KPE it will be displayed like in figure 12.

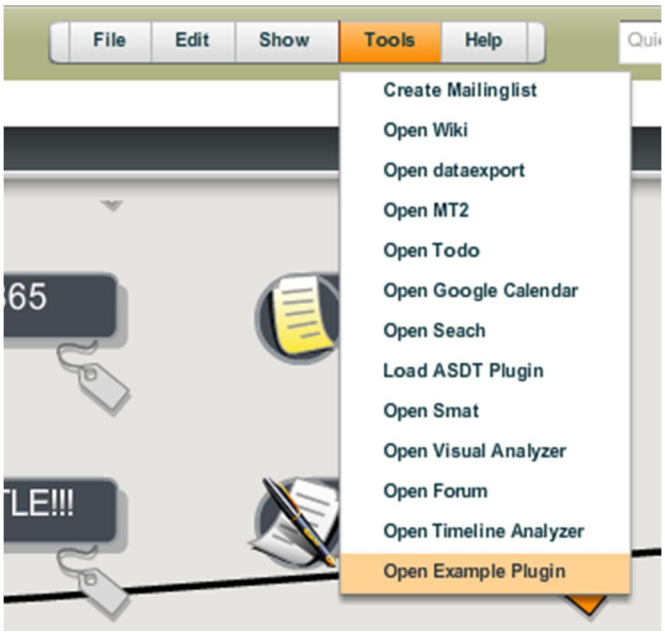


Figure 12: Tools menu configuration for plug-ins.

DisplayView attribute defines in what views the plug-in tool menu is displayed. The possible values for the *displayView* attribute are PROCESS_VIEW, CONTENT_VIEW, ALTERNATIVE_PROCESS_VIEW, TAILORED_VIEW and COMMUNITY_VIEW. Type the views with capital letters and separate views with comma. Do not leave spaces between the views.

Action attribute defines what the KPE does when the Plug-in tool menu is clicked. Always define *action* attribute value as "*openTool*".

Data field is to identify the menu item from the other KPE menu items. Use your plug-in id value for the *data* element value. (The plug-in id will be defined when the ISSGUI interface is explained more detailed.)

3.3 UI components element

UIcomponets element contains UI components that are tailored for the plug-in nodes. The UI components are used in forms and item info when the plug-in node data is displayed. Plug-in doesn't have to have any tailored UI components; it can use the predefined UI components to display its data.

The *UIcomponents* element can contain multiple *UIcomponent* elements. You can define UI components for your plug-in node data (fields in forms and item info). *UIcomponent* element contains following attributes: *parameter*, *label* and *field*. Table 1 lists *UIcomponent* attributes and the attributes for the child elements of the *UIcomponent* element.

XML element	Attributes	Comment
Uicomponent	parameter	Parameter is the reference to the plug-in node KPE class where the parameter same as the variable where the data is stored. Following variables can be used: **URI**, **title, x, y created**, **modifier** and **description.** (Also plug-in name, URI and URL can be used.) If your plug-in creates new variables into plug-in node, you can access them by typing "**dataEntries,<your variable>**". E.g. if your plug-in node contains

		"summary" field you refer to it: "dataEntries, summary". If you want to get the plug-in name you can use "plugin, pluginName". Check the getters from the ISSGUIPlugin interface and you know what data you can get from plug-in.
	label	Label field contains text and it will be the label for the form items in forms and item info.
	field	Field attribute is the ID of the UI component and the node fields can refer to this attribute.
form (child element)	clazz	Always: "ssgui.components.popups.forms.components"
	type	Select type from the predefined KPE UI components, e.g. TitleInput
inline (child element)	clazz	Always: "ssgui.libraries.adapter"
	type	Always: "AdabterField"
	field	Select type from the predefined KPE inline UI components.
	disabledField	Select type from the predefined KPE inline UI components.

Table 1: The list of UIcomponent element attributes and child element attributes.

Parameter attribute links to the variable of the KPE plug-in node class. *Label* attribute defines the form item name for forms and item info. *Field* attribute is the ID of the *UIcomponent* element.

UIcomponent can contain two child elements: *form* and *inline*. The *form* element defines the UI component for KPE form and *inline* element defines the UI component for the KPE item info.

The *form* element can contain two different attributes: *clazz* and *type*. The *clazz* attribute is always *"ssgui.components.popups. forms.components"* and the *type* attribute is reference to the Predefined KPE UI component class name. Table 2 displays list of predefined UI form components.

UI components in forms		
Clazz	**Type**	**Comment**
ssgui.components.popups.forms.components	TitleInput	Title input field that can be can be filtered.
ssgui.components.popups.forms.components	DescriptionInput	Multiline text area.
ssgui.components.popups.forms.components	StatusTagCom-ponent	Component for status tags.
ssgui.components.popups.forms.components	WebLinkInput	We link component uses default web link validator.
ssgui.components.popups.forms.components	DateRangeInput	Date range component accepts only an object that has start date and end date values.
ssgui.components.popups.forms.components	DueDateInput	UI component to display date.

Table 2: Lists predefined KPE UI components for forms.

The *inline* element can contain four different attributes: *clazz*, *type*, *field* and *disabledField*. The *clazz* attribute for *inline* element is always "*ssgui.libraries.adabter*" and the *type* attribute is always *"AdabterField"*. The *clazz* and *type* attributes defines where the KPE can found the UI components.

Field attribute defines the inline UI component when the item info is in "edit mode" and the *disabledField* attribute defines inline UI component when the item info is in "read mode". If the plug-in node field is not editable (defined in node configuration), then the *disabledField* is displayed in edit and read mode. Table 3 lists the UI components defined for the item info.

Inline components			
Clazz	Type	Field/ Disabledfield attribute	Comment
ssgui.libraries.adapter	AdapterField	EDITABLE_TEXT	Text that can be edited
ssgui.libraries.adapter	AdapterField	COLLAPSIBLE	List, displayed in item info must contain array collec-tion. Cannot be edited.
ssgui.libraries.adapter	AdapterField	EDITAB-LE_COMBOBOX	Editable Compobox

ssgui.libraries.adapter	AdapterField	EDITABLE_LABEL	Editable Label
ssgui.libraries.adapter	AdapterField	EDITAB-LE_DATERANGE	Editable Daterange
ssgui.libraries.adapter	AdapterField	EDITABLE_DATE	Editable date
ssgui.libraries.adapter	AdapterField	LABEL	Label that cannot be edited.

Table 3: Lists predefined KPE UI components for item info.

Above information is needed when you have to configure your own UI components. Table 4 lists KPE UI components that can be used by referring the "*field*" attribute of the UI component.

Predefined UI components

Field attribute	Form UI com-ponent	Inline field attribute	Inline disabledField Attribute	(node) pa-rameter
title	TitleInput	EDITABLE_TEXT	LABEL	title
name	TitleInput	EDITABLE_TEXT	LABEL	name
descripti-on	DescriptionInput	EDITABLE_TEXT	LABEL	description
status	StatusTagCom-ponent	EDITAB-LE_COMBOBOX	EDITAB-LE_COMBOBOX	status
weblink	WebLinkInput	EDITABLE_LABEL	LABEL	contentURL
duration	DateRangeInput	EDITAB-LE_DATERANGE	EDITAB-LE_DATERANGE	dateRange
DueDate	DueDateInput	EDITABLE_DATE	LABEL	dueDate
creator	-	LABEL	LABEL	crea-tor,fullname
created	-	LABEL	LABEL	created
modifier	-	LABEL	LABEL	modifier
itemType	-	LABEL	LABEL	itemType

Table 4: The list of predefined UI components and its parameters.

Note that predefined UI components are using certain node parameters. If you have created new data fields for your plug-in nodes, you have to define the UI component so that it refers to your plug-in data field parameter. For example in figure 10 (what

displayed the plugin-config.xml example) has one new UI component defined. The figure 13 displays the configuration for that UI component.

```
<uicomponent field="PluginName" parameter="plugin,pluginName" label="Plugin">
    <form clazz="" type=""/>
    <inline clazz="ssgui.libraries.adapter" type="AdapterField" field="LABEL" disabledField="LABEL" />
</uicomponent>
```

Figure 13: Displays the configuration example for new UI component.

In figure 13 new UI component is made for displaying pluginName variable. The *form* element is empty because the field is not displayed in forms. The *inline* element defines that the field is using LABEL UI component in item info.

The *TitleInput* UI component for forms can have on extra configuration; it can be filtered. More detailed instructions how to add filter into UI component is explained in next chapter.

3.4 Nodes element

The *nodes* element can contain multiple *node* elements. *Node* elements define your plug-in nodes if your plug-in adds new nodes to the KPE.

Node element can contain following attributes: *name*, *typeID*, *label*, *displayView*, *clazz*, *type* and *itemtype*. Table 5 represents the attributes of the *node* element and its child elements attributes.

XML element	Attributes	Comment
node	name	The name of the plug-in node. Value is required.
	typeID	ID of the plug-in type. The typeID attribute indentifies different plug-in nodes from each others. The value must be unique and value is required.
	label	The label used in forms, item info and menus. Value is required.
	displayView	Defines in what views the plug-in node is displayed. Possible values are: CONTENT_VIEW, ALTERNATIVE_PROCESS_VIEW and TAILORED_VIEW. Attribute can contain multiple values and values are separated with comma. If no views are set, then the plug-in node will not

		be displayed.
	clazz	Always: "*ssgui.libraries.graphModel.PluginNode*". Value is required.
	type	Always: "*TYPE"*. Value is required.
	itemtype	Can be "*PluginNode"* or "*ContentItem".* If value is "*PuginNode*", then the plug-in node creation is displayed separately in creation menus. If the value is "*ContentItem*", then the creation of the plug-in node is displayed in "Create Content Item" form. Value is required.
fields (child of node)		Does not have any attributes. Fields element can only contain multiple Field elements.
field (child of fields)	type	Type is the reference to the UI component element (can be KPE predefined UI component or plug-in UI component). Value is required.
	required	Value can be true or false. If set to true, the form will validate the value. Default value is false.
	editable	Value can be true or false. Editable attribute defines if the field can be edited in forms and item info. Default value is true.
	filter	Filter value can be used only if the UI component contains TitleInput component in form element. The filter value can be reference to the predefined KPE filters or it can be plain Regular Expression.
menu (child of node)		Does not have any attributes. Menu element can only contain multiple menuitem elements.
menuitem (child of menu)	id	The id of the predefined KPE menuitem (e.g. open). Value is required.
	allowed	If allowed is true, then the item will be displayed in menus. Value can be true or false. Default value is true.
	label	The label of the menuitem if it needs to be changed. If label is not defined, then menuitem will use predefined value.
	displayInContextMenu	Every menuitem that needs to be displayed in node's context menu has to be defined. Set value to true and the menuitem will be displayed in context menu. If value is not

		set, defined menuitem will not be displayed in context menu.

Table 5: The list of node element attributes and child element attributes.

Name attribute defines the name of your node.

TypeID attribute separates the node from other KPE nodes and other plug-in nodes, so the *typeID* have to be unique. Good way to name your plug-in is use your plug-in name in typeID e.g. http://www.examplePlugin.org/ontologies/PluginNode#BaseNode. Change the "examplePlugin" to your plug-in name.

Label attribute is displayed in forms, item info and menus, so use human readable label. Figure 14 displays example where the label attribute is used in KPE.

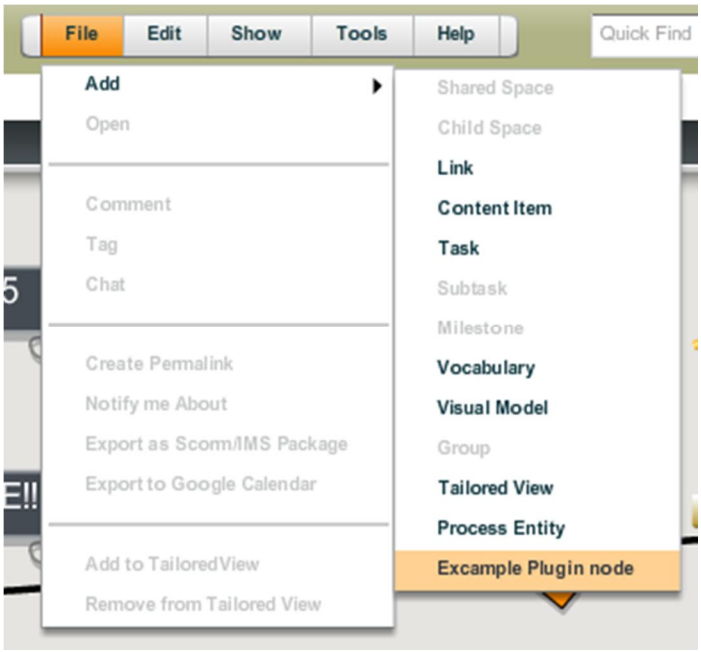


Figure 14: Label attribute is used e.g. in menus.

DisplayView attribute defines in what views the node element is displayed and where the user can create new plug-in elements. Possible values for the *displayView* attribute are: CONTENT_VIEW, ALTERNATIVE_PROCESS_VIEW and TAILORED_VIEW. Type the views with capital letters and separate views with comma. Do not leave spaces between the views.

Clazz and *type* cannot be changed and they always have to have value what is displayed in figure 10. *Clazz* element has to be “ssgui.libraries.graphModel.PluginNode” and *type* element has to be “TYPE”. These values connect the plug-in node to the other KPE nodes and if these values are not defined the KPE cannot create plug-in nodes.

Itemtype element defines is the plug-in node threaded as plug-in node or Content Item. The difference between plug-in node and content item itemtype is that if the plug-in node is defined as Content Item, the creation of the plug-in node is found only inside Content Item form, but if the itemtype is defined as plug-in node, then the creation for plug-in node is in same level as creation of other nodes (e.g. tasks, links, vocabulary etc.) The possible values for *itemtype* attribute are: PluginNode and ContentItem.

3.4.1 Fields element: the child element for node element

The *node* element contains *fields* child element. The *fields* element can contain multiple *field* elements and every *field* element defines one data field for the plug-in node. For example if the *fields* element contains three *field* elements: title, description and creator. Those three fields are displayed when the plug-in node is created, modified of viewed with the item info or in form.

The *field* element can contain following attributes: *type*, *required*, *editable* and *filter*.

The *type* attribute is reference to the UI component element *field* attribute. You can refer directly to predefined KPE UI components or refer to your own UI component. The type attribute defines what UI component is displayed in form and in item info when the plug-in node is created, modified or viewed in item info.

Required attribute defines is the field required or not. If *required* attribute is defined “true” then red asterisk (*) is displayed in form and the form will not be valid if the field is not filled. Default value for the attribute is “false”.

Editable attribute defines is the field editable or not. For example the “creator” field for the node should be non editable. The *editable* attribute can be “true” or “false”. The default value for the attribute is “true”.

Filter attribute can be used only if the UI component that uses TitleInput component in forms. KPE has predefined UI component named “title” and it can be used for title field. To use filter attribute in other UI components, you have to create new UI component that uses TitleInput component in form. Check chapter “UI components element” how to do that.

The filter value can contain one of the three predefined regular expressions: WIKI_TITLE_REGEXP, SSP_TITLE_REGEXP or GLOBAL_REGEXP. You can also create and use your own filter by writing the regular expression directly into the *filter* attribute. Table 6 displays predefined regular expressions for filters.

Predefined KPE filters	
WIKI_TITLE_REGEXP	^[^'\"&`´=]$
SSP_TITLE_REGEXP	^[_a-z0-9]$
GLOBAL_REGEXP	^[^<>]$

Table 6: List of predefined KPE filters.

If you are not familiar with regular expressions, check Flex 3 regular expression explorer to get started <http://ryanswanson.com/regexp/#start>.

3.4.2 Menu element: the child element for node element

Menu element can contain multiple *menuitems* elements. The purpose of the *menu* element is to define elements that are disabled for the plug-in node. If you want to change the label from the default menu item for the plug-in node (for example the “open” menu item for the task node is changed to “open in process view”). Also the context menu is defined in *menu* element for the plug-in nodes.

The *menu* element doesn’t contain any attributes but the child elements *menuitem* can contain following attributes: *id*, *label*, *allowed* and *displayInContextMenu*.

Id attribute is reference to the predefined KPE menuitem. Table 7 lists predefined KPE menuitems.

Id	Description	Comment
open	Menuitem for open functionality.	-
comment	Menuitem for comment functinality.	Not implemented for plug-in nodes.
tag	Menuitem for Semantic tagging functionality.	Not implemented for plug-in nodes.
chat	Menuitem for chat functionality.	Not implemented for plug-in nodes.
AddTOTV	Menuitem for Add to Tailored View functionality.	Not tested. Might not work.
modify	Menuitem for modify functionality.	-
delete	Menutiem for delete functionality.	-
straightenLink	Menuitem for straigthenLink functionality.	Not implemented for plug-in nodes.
copy	Menuitem for copy functionality.	Not implemented for plug-in nodes.
paste	Menuitem for paste functionality.	Not implemented for plug-in nodes.

Table 7: List of predefined KPE menuitems.

Label attribute defines the label for the menu item. The label is displayed in main menu and in context menu if the menu item is displayed in context menu.

Allowed attribute defines if the menu item is disabled or enabled for the plug-in node. By default all menu items are enabled, but for example if you don't want to display "Open" menu item for the plug-in node, you can define "open" menu item to be disabled by setting *allowed* attribute to "false". Tagging and commenting are not available for the plug-in nodes, so those menu items should be disabled.

DisplayInContextMenu attribute defines if the menu item is displayed in context menu or not. By setting attribute to "false" the menu item is not displayed in context menu.

```
<menu>
    <menuitem id=open" displayInContextMenu="true" />
    <menuitem id="comment" allowed="false" />
    <menuitem id="tag" allowed="false" />
</menu>
```

Table 8: Defining menu for plug-in node.

Now that you have created plugin-config.xml file for your plug-in, you have to embed the xml to your module.

4 KPE plug-in interfaces and embedding configuration file

4.1 ISSGUIPlugin interface

The *ISSGUIPlugin* interface defines all methods that can be used between KPE and plug-in. The main purpose of the interface is to provide information about the plug-in to KPE. All the plug-ins in KPE are processed as *ISSGUIPlugin*, so if the plug-in doesn't implement *ISSGUIPlugin* interface or the interface is not implemented in correct way, the plug-in cannot be loaded into KPE. Table 9 presents the list of methods in *ISSGUIPlugin* interface.

Method Summary	
String	get pluginName():String The human readable name of the plug-in.

String	get pluginId():String The id of the plug-in.
String	get pluginUID():String The UID of the plug-in.
String	get pluginVersion():String The human-readable version information of the plug-in.
XML	get pluginConfig():XML Returns plug-in configuration when the host application asks for it.
Void	pluginUID(value:String):void Sets the plug-in UID when service returns the UID.
Array	getZUIObjectsForSSP(aSSPURI:String):Array Returns all nodes the plug-in wishes to show in the content view of the specified shared space.
IHostFacade	get hostFacade():IHostFacade The reference to the HostFacade object.
Void	set hostFacade(aVal:IHostFacade):void The reference to the HostFacade object.
EventDispatcher	loadSSPContents(aSSPURI:String, aCompleteEventHandler:Function = null):EventDispatcher Asks the plug-in to initiate the loading of any data it wants to load for a specific shared space.
Array	getContextMenuItems_ContentViewPlane(aSSPURI:String, aContext-Menu:ContextMenu):Array Gets context menu items that the plug-in wishes to add to the context menu of the content view plane.
Array	getLinkTypes():Array Gets objects describing the ZUIPlane link types that the plug-in wishes to allow the user to create.
Void	linkCreated(aStartObjectURI:String, aEndObjectURI:String, aLinkType:String):void A notification that a link has been created in the host application.
Void	setPluginContent(data:ArrayCollection):void The KPE returns the plug-in nodes of the plug-in.

DisplayObject	getContent():DisplayObject Returns the content of the plug-in tool window.
Void	open(type:String = ""):void KPE wants to open plug-in tool or plug-in node.

Table 9: The method summary of the ISSGUIPlugin interface.

The table 10 displays ISSGUIPlugin interface methods more detailed.

Method Detail
public function get pluginName():String Returns human readable name of the plug-in. Define plug-in name in ISSGUIplugin constructor.
public function get pluginId():String Returns id of the plug-in. Define plug-in id in ISSGUIPlugin constructor.
public function get pluginUID():String Returns plug-in UID.
public function get pluginVersion():String The human-readable version information of the plug-in. Define plug-in version in ISSGUIPlugin constructor.
public function get pluginConfig():XML Returns plug-in configuration when the host application asks for it. Plug-in configuration xml have to be embedded into module. Load xml in ISSGUIPlugin

constructor.
public function set pluginUID(value:String):void Sets the plug-in UID when server returns the UID.
public function getZUIObjectsForSSP(aSSPURI:String):Array Returns all nodes the plug-in wishes to show in the content view of the specified shared space. *@param* aSSPURI is The URI of the shared space, or null to designate the shared space network
public function get hostFacade():IHostFacade The reference to the HostFacade object. *@return* IHostFacade
public function set hostFacade(aVal:IHostFacade):void The reference to the HostFacade object. *@param* is IHostFacade
public function loadSSPContents(aSSPURI:String, aCompleteEventHandler:Function = null):EventDispatcher Asks the plug-in to initiate the loading of any data it wants to load for a specific shared space. The plug-in should return an *EventDispatcher* which will dispatch an *Event.COMPLETE* event when this loading procedure has completed in the plug-in. If the plug-in returns *null* for this method call, the host can assume that the plug-in doesn't want to load anything.

@param aSSPURI is the URI of the shared space the contents of which to begin loading
@param aCompleteEventHandler is a Reference to an event handler function that the plug-in should attach to the *EventDispatcher* right away (preferably before actually initiating the data loading operation so that events that are dispatched with almost no delay would also trigger this event handler).
@return An object to dispatch an *Event.COMPLETE* event when the loading has completed, or *null* if the plug-in doesn't want to load anything

public function getContextMenuItems_ContentViewPlane(aSSPURI:String, aContextMenu:ContextMenu):Array

Gets context menu items that the plugin wishes to add to the context menu of the content view plane.
@param aSSPURI is the URI of the current shared space that the current view is showing.
@param aContextMenu is the context menu for which these items are meant. Note that the plugins should add the items into this context menu themselves.
@return An array of ContextMenuItems.

public function getLinkTypes():Array

Gets objects describing the ZUIPlane link types that the plug-in wishes to allow the user to create.
The returned array should contain objects with the properties "*title*" (the human-readable name of this link type, e.g. "explosive link"), *"type"* (a string identifier for this link type, e.g. "explosiveLink") and *"color"* (an integer specifying the color value for the link).
@return An array of objects describing the link types.

public function linkCreated(aStartObjectURI:String, aEndObjectURI:String, aLinkType:String):void

A notification that a link has been created in the host application.

@*param* aStartObjectURI is the URI of the link's start node @*param* aEndObjectURI is the URI of the link's end node @*param* aLinkType is the type of the link
public function setPluginContent(data:ArrayCollection):void The KPE returns the plugin nodes of the plug-in. @*param* data is the ArrayCollection that contains plug-in data.
public function getContent():DisplayObject Returns the content of the plug-in tool window. @*return* DisplayObject for plug-in tool window.
public function open(type:String = ""):void KPE wants to open plug-in tool or plug-in node. @*param* type is the type of the node, or if not set, then tool window is opened.

Table 10: Method details of the ISSGUIPlugin interface.

4.2 IHostFacade interface

The IHostFacade interface allows plug-in to get information from KPE. When the KPE loads Plug-in, it will use set IHostFacade function (implemented in ISSGUIPlugin interface). And after plug-in have got the reference to IHostFacade, it will be able to ask certain information from the KPE.

The implementation of the IHostFacade is displayed in Attachment 1. The attachment displays the comments that will explain how the interface can be used.

4.3 Embedding configuration file

The Flex does not automatically build xml files into .swf file when the project is build. That’s why you have to embed your plug-in configuration file into your plug-in class. Figure 15 displays how the xml file is embedded into your plug-in class.

```
//embed the plug-in config.xml file
[Embed("plugin-config.xml", mimeType="application/octet-stream")]
private static const Config:Class;
```

Figure 15: Embedding configuration file into private class.

When the plug-in configuration file is embedded you have to save the xml into private variable *_pluginConfig* so that KPE is able to get the xml file when the plug-in is loaded (by using *get pluginConfig*() method). Figure 16 displays example method for adding xml file into class variable.

```
private function loadPluginXML():XML{
    var ba:ByteArrayAsset = ByteArrayAsset(new Config());
    var xml:XML = new XML(ba.readUTFBytes(ba.length));
    return xml;
}
```

Figure 16: Loading xml into private class.

Method *loadPluginXML* is called in plug-in class constructor as the figure 17 displays.

```
public function ExamplePlugin()
{
    super();
    _pluginName = "Excample Plugin";
    _pluginId = "examplePlugin";
    _pluginConfig = null;
    _pluginConfig = loadPluginXML();
}
```

Figure 17: Setting the private variables in plug-in constructor.

5 Uploading new plug-in to KPE

The plug-in KPE is located at http://mielikki.mobile.metropolia.fi/shared-space-plugin/. Log in as *Student_Paul* if you want to add or remove plug-ins from the KPE. Other users can only enable or disable the visibility of the plug-in in their KPE.

To open plug-in settings window click  plug-in icon from the top left part of the KPE. Image 18 displays the plug-in settings window for Student_Paul.

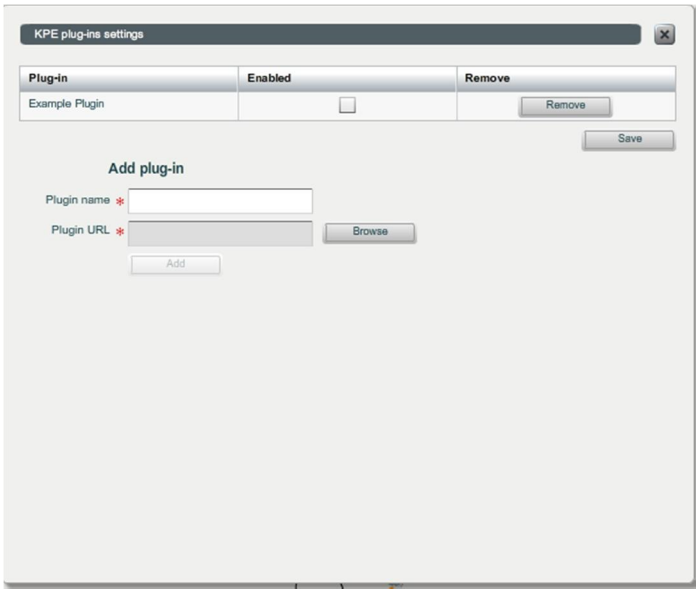


Figure 18: Displays plug-in settings window for Student_Paul.

Image 19 displays the plug-in settings window for other users (other than Student_Paul).

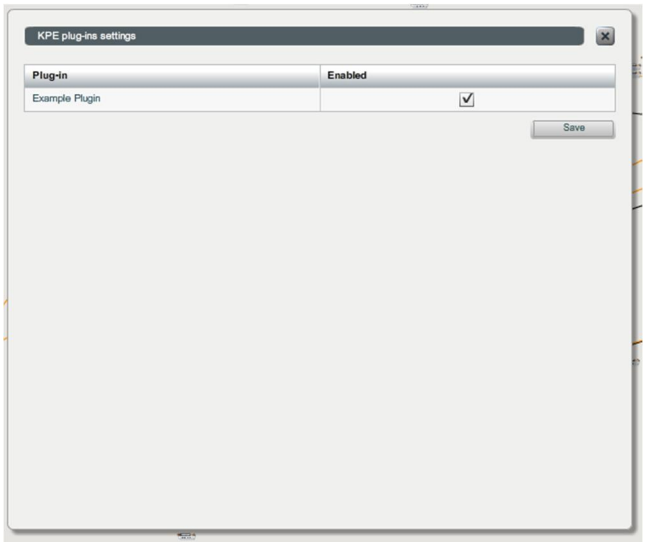


Figure 19: Displays plug-in settings window for other users.

To add new plug-ins log in as “Student_Paul”. Open the KPE Plug-in Settings window and you should see the “Add Plug-in” form (see figure 15). Write your plug-in name

into “Plug-in name” field and browse your plug-in .swf file in the “Plug-in URL” field. When you have filled these two fields, the “Add” button enables and pressing the “Add” button, the plug-in is uploaded and saved into KPE plug-in database. After the plug-in is successfully uploaded the plug-in appears into “Plug-in settings” –list (above the form). By checking the enabled checkbox for your plug-in and clicking “Save” button, your plug-in will be loaded into KPE.

The plug-in settings are personal, so all user can enable and disable the plug-ins displayed in KPE. By default all the plug-ins are disabled, until the user enables them.

Removing the Plug-ins is easy. You have to be logged as “Student-Paul” to see the “Remove” button in “Plug-in Settings” –list. Clicking the “Remove” button the plug-in will be removed from the KPE and all the data in database will be deleted.

Attachment 1: IHostFacade class

```
package kpe.libraries.plugins

{
            import flash.geom.Point;

            /**

            * This is the interface the host facade object will
            * implement.
            * The host facade is an object that is passed on to all
            * plugins once they have been loaded, and it allows the
            * plugins to make calls to the host application.
            *
            */

            public interface IHostFacade {

                        /**

                        * Writes a message to the log output via the
                        * logging system in the host application.
                        *
                        * @param aMessage The message to write to the log
                        * @param aType The type of the message
                        *
                        *@see org.kplab.common.libraries.LoggingService
                        */

                        function log(aMessage:String, aType:String):void


                        /**
                        * Creates an IPluginZUINode object based on some
                        * initialization properties.
                        *
                        * @param aInitProperties The initialization
                        * properties. Each property should correspond, by
                        * name and by type of the value, to properties in
                        * <code>IPluginZUINode</code>.
                        *
                        * @return The created IPluginZUINode object
                        */

                        function createZUINode(aInitProperties:Object,
                        plugin:ISSGUIPlugin,
                        typeID:String):IPluginZUINode



                        /**
                        * Creates an IPluginGANTTItem object based on
                        * some initialization properties.
                        *
                        * @param aInitProperties The initialization
```

```
* properties. Each property should correspond, by
* name and by type of the value, to properties in
* <code>IPluginGANTTItem</code>.
*
* @return  The created IPluginGANTTItem object
*/

function createGANTTItem(aInitProperties:Object):
IPluginGANTTItem



/**
* Creates an IPluginZUILink object based on some
* initialization properties.
*
* @param aInitProperties The initialization
* properties. Each property should correspond, by
* name and by type of the value, to properties in
* <code>IPluginZUILink</code>.
*
* @return The created IPluginZUILink object
*/

function createZUILink(aInitProperties:Object):
IPluginZUILink



/**
* Adds the specified ZUINode to the current view,
* if the current view is a ZUIPlane-based view.
*
* @param aZUINode The node to add
*/

function addZUINodeToCurrentView(aZUINode:
IPluginZUINode): void



/**
* Adds the specified ZUILink to the current view,
* if the current view is a ZUIPlane-based view.
*
* @param aZUILink     The link to add
*/

function addZUILinkToCurrentView(aZUILink:
IPluginZUILink): void



/**
* Creates a shared space and adds it to the
* application model.
*/
```

```
function createSharedSpace(aURI:String,
aGraphElementURI:String, aTitle:String,
aXCoord:Number, aYCoord:Number, aCreated:Date,
aModified:Date, aCreatorURI:String,
aAllowOpen:Boolean, aAllowModify:Boolean,
aAllowDelete:Boolean, aAllowAddObjects:Boolean,
aAllowCreateChildSpace:Boolean,aAllowInOutBoundLi
nk:Boolean, aAllowAdministrateMembers:Boolean,
aAllowModifyContent:Boolean,
aAllowDeleteContent:Boolean, aClassURI:String,
aDescription:String = "", aOtherProperties:Object
= null):void



/**
* Returns info about the current user. The
* returned object
* will be in the following format:
*
* <code>
* {username:"", firstName:"", lastName:"",
* URI:""}
* </code>
*
* @return  An object describing the user
*/

function getUserInfo():Object



function openPluginToolWindow(plugin:
ISSGUIPlugin):void



/**
* Changes the current shared space.
*
* @param aSSPURIThe URI of the space to go to
*/

function goToSharedSpace(aSSPURI:String):void




/**
* Returns the URI of the currently shown shared
* space.
*
* @return  The URI of the currently shown shared
* space
*/

function getCurrentSSPURI():String
```

```
                    /**
                    * Returns the name of the current view.
                    *
                    * @return  The name of the current view
                    */

                    function getCurrentViewName():String



                    /**
                    * Returns the current mouse cursor coordinates in
                    * the coordinate space of the ZUICanvas of the
                    * ZUIPlane of the current view, if the current
                    * view has a ZUIPlane.
                    */

                    function getCurrentZUIPlaneMouseCoordinates():
                    Point



                    /**
                    * Enters the mode where the user can create a new
                    * link of the specified type.
                    */

                    function enterLinkingMode(aLinkType:String,
                    aLinkColor:int = 0):void


            }

}
```