Mikael Koskinen

Plugin-pohjaiset sovellukset – arkkitehtuurit

Tietojärjestelmätieteen
Kandidaatin tutkielma
27.3.2005

Jyväskylän yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä

TIIVISTELMÄ

Koskinen, Mikael Janne Petteri

Plugin-pohjaiset sovellukset - arkkitehtuurit / Mikael Koskinen

Jyväskylä: Jyväskylän yliopisto, 2005.

28 s.

Tutkielma

Tässä tutkielmassa tarkastellaan plugin-pohjaisia sovelluksia sekä plugin-arkkitehtuureja. Päätavoitteena on selvittää pluginien käytöstä johtuvat mahdolliset hyödyt sovelluksen kehityksen näkökulmasta.

Plugineihin on tutustuttu tieteellisten kirjoitusten pohjalta. Kolme ehdotusta plugin-arkkitehtuuriksi valittiin tarkempaa tutkimusta varten. Arkkitehtuurit on esitelty yleisellä tasolla, ilman koodiesimerkkejä. Arkkitehtuureja verrataan keskenään ja niiden hyödyt sekä heikkoudet tuodaan esille.

AVAINSANAT: plugin, komponentit

SISÄLLYSLUETTELO

1 JOHDANTO 4

2 PLUGINIT 6
2.1 Yleistä 6
2.2 Määritelmä 7
2.3 Ominaisuudet 7
2.4 Käyttötilanteet 8

3 PLUGIN-ARKKITEHTUURIT 11
3.1 Yleistä arkkitehtuureista 11
3.2 PluggableComponent 14
3.3 Lightweight Plug-In-Based Application Development 18
3.4 MagicBeans 20
3.5 Vertailua arkkitehtuurien välillä 23

4 YHTEENVETO 25

LÄHDELUETTELO 27

1 JOHDANTO

Pluginit ovat sovellukseen kytkettäviä lisäpalikoita, jotka tarjoavat ohjelmaan uusia ominaisuuksia. Plugineja voidaan hyödyntää tilanteissa, joissa sovellukseen täytyy lisätä ominaisuuksia julkaisun jälkeen sekä tilanteissa, joissa suuri sovellus halutaan jakaa pienempiin, helpommin hallittaviin osiin. Ne mahdollistavat myös jatkuvasti päällä olevien sovellusten päivittämisen ilman sammuttamista sekä kolmansien osapuolien tekemien ominaisuuksien lisäämisen sovellukseen.

Plugin-pohjaiset sovellukset ovat kuin palapelejä. Sovellus tarjoaa avoimia pisteitä, joihin oikean muotoiset pluginit voivat kytkeytyä. Valittu plugin-arkkitehtuuri määrittää, kuinka kytkös tehdään.

Tässä tutkimuksessa käsitellään plugin-pohjaisia sovelluksia sekä niiden arkkitehtuureja. Plugineita hyödynnetään nykyisin lukuisissa eri ohjelmissa. Tämän tutkimuksen tarkoituksena on selvittää lukijalle pluginien tarjoamat mahdolliset hyödyt sekä käyttötilanteet. Plugin-arkkitehtuurien ominaisuuksia esitellään yleisellä tasolla ja lisäksi perehdytään tarkemmin kolmeen erilaiseen plugin-arkkitehtuurin. Arkkitehtuurit on valittu niiden erilaisuuden perusteella. Jokainen niistä hyödyntää olio-ohjelmoinnin perustoimintoja, mutta erilailla. Käsiteltävät arkkitehtuurit ovat Chatleyn ym. (2004) MagicBeans, Mayerin ym. (2002) LPA sekä Völterin (1999) PluggableComponent.

Plugin-pohjaiset sovellukset ovat tutkimuksen alana uusi kohde. Ensimmäisen ehdotuksen plugin-arkkitehtuurista antoi Marquardt vuonna 1999. Tämä on ensimmäinen katsaus, joka kokoaa yhteen eri tutkimuksissa plugin-arkkitehtuureihin esitetyt suunnittelumallit ja vertailee niiden ominaisuuksia.

Tutkimuksen sisältö jakaantuu kahteen lukuun. Luvussa 2 käsitellään plugineja yleisemmällä tasolla. Termi plugin määritellään ja pluginien käyttömahdollisuuksia esitellään. Luvussa 3 tutustutaan tarkemmin plugin-arkkitehtuuriin. Aluksi esitellään arkkitehtuurien yleisiä piirteitä, jonka jälkeen tutustutaan tarkemmin kolmeen eri arkkitehtuurin. Lopuksi ratkaisujen ominaisuuksia vertaillaan keskenään. Tutkielman päättää yhteenvetoluku.

2 PLUGINIT

Tässä luvussa käsitellään plugineja yleisellä tasolla. Ensin selvitetään ohjelmistonkehityksen taustoja, jonka jälkeen määritellään tarkemmin termi plugin. Seuraavaksi tuodaan esiin pluginien ominaisuudet ja niiden mahdolliset käyttötilanteet.

2.1 Yleistä

Ohjelmistojen ylläpito on tärkeä osa ohjelmiston kehitysprosessia. Suurinta osaa sovelluksista joudutaan päivittämään vielä niiden julkaisemisenkin jälkeen. Päivityksellä voidaan korjata esimerkiksi havaittuja vikoja tai lisätä uusia ominaisuuksia. Normaalisti muutokset ohjelmaan tehdään muuttamalla sen lähdekoodia. Muutokset tallennetaan, lähdekoodi käännetään ja uudet versiot binääri-tiedostoista jaetaan ohjelman käyttäjille. Tämä on hankala prosessi etenkin kriittisissä järjestelmissä, joiden pitäisi olla jatkuvasti päällä. Uuden version asentaminen vaatii järjestelmän sammuttamisen, päivittämisen ja uudelleenkäynnistämisen. (Chatley ym. 2003, 1).

Suurin osa sovelluksista hyötyisi siitä, jos niiden osia pystyttäisiin helposti vaihtamaan julkaisun jälkeen (Völter 1999, 1). Marquardtin (1999, 1) mukaan ohjelmistoja käyttävät asiakkaat vaativat sovelluksilta mukautumiskykyä sekä laajennettavuutta. Laajennuksien täytyy istua saumattomasti olemassa oleviin sovelluksiin. Käyttäjät odottavat ohjelmistojen tukevan huomisen teknologiaa ja heidän kasvavat vaatimukset ovat aiheuttaneet sovellusten koon kasvun. Vaatimukset saattavat muuttua sovelluksen ilmestymisen jälkeen. Näihin muutoksiin on pyritty varautumaan esimerkiksi komponentti-pohjaisilla järjestelmillä (Chatley ym. 2003, 1). Komponenteista koostettujen sovellusten luvattiin tulevan markkinoille olio-ohjelmoinnin yleistymisen myötä, mutta näin ei ole käynyt. Ongelmia on aiheuttanut muuan muassa standardien puutteellisuus. (Marquardt 1999, 1).

Mayerin ym. tutkimuksessa (2002, 1) todetaan, että uusien sovellusten muistivaatimukset ovat kasvaneet reilusti. Yksi syy tähän on se, että ohjelmistoala on hyväksynyt Mooren lain nopeasti kasvavista tietokonetehoista. Toinen tekijä on sovelluksiin lisättävät uudet ominaisuudet. Ominaisuudet voivat olla tarpeellisia, mutta samalla vanhoja ominaisuuksia ei uskalleta poistaa, koska niiden käyttötarkoitusta ei tiedetä. Kiireellinen aikataulu haittaa usein ohjelmiston tarkkaa suunnittelua. Sovelluksen modulaarisella suunnittelulla voidaan saavuttaa rakenne, jossa tiettyjä ohjelman osia voidaan korvata uusilla, niin kutsutuilla plugineilla. (Mayer ym. 2002, 1).

2.2 Määritelmä

Plugin on ohjelma, joka integroituu tiettyyn sovellukseen, tarjoten siihen uusia ominaisuuksia (Mayer ym. 2002, 2). Pluginit ovat alkuperäisen ohjelman lisäosia. Ne sisältävät tietyn toiminnallisuuden, joka ajetaan tarvittaessa sovelluksen suorituksen aikana. (Marquardt 1999, 2). Sovellus täytyy suunnitella niin, että se tukee plugineja ja usein sovelluksen tekijät julkaisevat tekniset asiakirjat siitä, kuinka plugineja voidaan kirjoittaa. Plugin on sovellukselle tuntematon sen käännösvaiheessa. Sovellukseen ei ole ohjelmoitu mitään joka liittyisi tiettyyn pluginiin. Pluginit vaativat ohjelman, jota varten ne on suunniteltu. (Mayer ym. 2002, 2). Sovellukseen on määritetty avoimia kohtia, joihin vaatimukset täyttävä plugin voi kiinnittyä. Pluginit täydentävät ohjelman ominaisuuksia. Ohjelma ei ole riippuvainen pluginista, mutta plugin itsessään ei toimi ilman ohjelmaa. (Marquardt 1999, 2).

2.3 Ominaisuudet

Pluginit mahdollistavat sovelluksen laajentamisen ajonaikana. Ne ovat dynaamisia komponentteja, joita sovelluksen käännösvaiheessa ei tunneta. Plugin alustetaan vain tarpeen vaatiessa. Tämän takia sovellus käynnistyy aluksi nopeasti, sillä yhtään pluginia ei ladata. Tämä vähentää sovelluksen

käyttämää muistin määrää. (Mayer ym. 2002, 3). Plugin-arkkitehtuuri myös mahdollistaa yksittäisen osan muokkaamisen ilman, että muihin osiin täytyy tehdä muutoksia. Sovellus ei vaadi pluginia toimiakseen ja toisessa osapuolessa tapahtuvat muutos ei vaikuta toiseen osapuoleen. (Marquardt 1999, 12). Mayer ym. (2002, 12) toteavat, että pluginit yksinkertaistavat sovelluksen rakennetta ja tekevät siitä helpommin hahmotettavan. Kuitenkaan koko järjestelmää ei ole mahdollista rakentaa pluginien pohjalta. Esimerkiksi sovellusta, joka hyödyntää plugineja, ei voida toteuttaa erillisenä komponenttina. Toinen ongelma on se, että plugineja voi kertyä lukuisia, jolloin niiden hallinnointi vaikeutuu.

2.4 Käyttötilanteet

Sovelluksen kokoaminen komponenteista on havaittu hyödylliseksi. Sovelluksen osakokonaisuuksien jakaminen moduuleihin muodostui komponentti-pohjaiseksi ohjelmiston kehitykseksi. Komponentteja yhdistelemällä voidaan luoda kokonaisia ohjelmistoja. Pluginit ja komponentit eroavat siinä, että pluginit ovat valinnaisia. Plugin-pohjaisen sovelluksen pitäisi toimia normaalisti ilman pluginejä. (Chatley ym. 2003, 1). Marquardt (1999, 21) toteaakin, että plugin-pohjaiset sovellukset ovat verrattavissa komponentti-pohjaiseen ohjelmistonkehitykseen. Molemmat ratkaisut perustuvat olio-ohjelmoinnin tarjoamiin tekniikoihin ja mahdollistavat suuren määrän ohjelmakoodin uudelleenkäyttöä. Pluginien käyttö on kuitenkin yleensä sidottu tiettyyn sovellukseen, toisin kuin komponentit. Pluginit ovat rajoittuneempia kuin komponentit, mutta niiden käyttöalue on laajempi. Plugineja hyödynnetään esimerkiksi mobiililaitteiden sovelluksissa, siinä missä komponentit ovat yleisesti rajoittuneet hajautettuihin sovelluksiin.

Chatleyn ym. (2003, 1) ja Mayerin ym. (2003, 3) mukaan plugineja voidaan hyödyntää seuraavissa tapauksissa:

- sovelluksen ominaisuuksien lisäämisessä julkaisun jälkeen

- suurten sovellusten jakamisessa osiin
- jatkuvasti päällä olevien sovellusten päivittämisessä ilman sammuttamista
- kolmansien osapuolien tekemien ominaisuuksien lisäämisessä sovellukseen.

Usein on hyödyllistä, jos sovellusta on mahdollista laajentaa pluginien avulla. Kaikkia vaatimuksia ei voida tietää, kun sovellusta ensi kertaa kehitetään. Jos sovellus tarjoaa mekanismin, jolla siihen voidaan lisätä uusia ominaisuuksia pluginien avulla, voidaan sitä kehittää tarpeen vaatiessa pienissä sykäyksissä. Suurissa sovelluksissa on yleistä, että tietty käyttäjäryhmä tarvitsee vain tiettyjä sovelluksen osia, siinä missä toinen ryhmä tarvitsee toisia. Jos sovellus tarjoaa vakiona jokaisen ominaisuuden, voi se johtaa ylimääräiseen muistin käyttöön. Jakamalla ohjelma moduuleihin, voidaan siitä koostaa eri käyttäjille räätälöityjä kokonaisuuksia. Tämän avulla voidaan minimoida resurssien käyttö. Lisäksi ohjelman tarjoamaa käyttöliittymää voidaan muokata käyttäjän tarpeiden mukaan ja sovelluksen eri osakokonaisuuksia voidaan myydä erikseen. (Chatley ym. 2003, 2). Marquardt (1999, 11) toteaakin, että plugineista saatava suurin hyöty on sovellusten räätälöinti; Plugin-tuen omaava sovellus on erittäin mukautumiskykyinen.

Chatley ym. (2003, 2) esittävät, että jatkuvasti päällä olevien sovellusten päivittäminen on ongelmallista. Etenkin uusien ominaisuuksien lisääminen niihin vaatii yleensä sovelluksen uudelleenkäynnistämisen. Plugin-pohjaiset sovellukset mahdollistavat uusien ominaisuuksien lisäämisen ilman uudelleenkäynnistämistä. Pluginit mahdollistavat myös kolmansien osapuolien tarjoamien ominaisuuksien käyttämisen sovelluksessa. Useinkaan sovelluksen tekijät eivät halua jakaa ohjelmansa lähdekoodiaan ulkopuolisille kehittäjille, mutta haluavat kuitenkin sovelluksensa olevan laajennettavissa. Pluginit mahdollistavat laajennusten ostamisen ja lisäämisen sovellukseen. (Chatley ym. 2003, 2). Ongelmaksi saattaa muodostua ohjelman tarjoamien avoimien pisteiden määrittely. Avoimet pisteet pitää määritellä niin, että pluginien avulla

voidaan mahdollistaa sovelluksen käyttö ennalta arvaamattomilla tavoilla. (Marquardt 1999, 2-3).

Plugin-tuen omaavia sovelluksia on erilaisia. Jotkin sovellukset sisältävät useita erilaisia avoimia pisteitä, joihin plugin voidaan liittää. Jotkin ohjelmat mahdollistavat vain yhden aktiivisen pluginin kerrallaan, kun taas toiset tukevat käytännössä loputtomasti eri plugineja. (Marquardt 1999, 3).

Tässä luvussa esiteltiin plugineja yleisellä tasolla. Aluksi kerrottiin syitä sille, miksi sovellukset ovat menossa komponentti-painotteisempaan suuntaan. Tämän jälkeen määritettiin termi plugin ja kuvattiin sen ominaisuuksia. Lopuksi annettiin mahdollisia käyttötapauksia, joissa plugineja voitaisiin hyödyntää. Seuraavassa kappaleessa tullaan tutustumaan plugin-arkkitehtuureihin tarkemmin. Arkkitehtuurien yleiset piirteet esitellään, jonka jälkeen perehdytään kolmeen arkkitehtuuri-ehdotukseen.

3 PLUGIN-ARKKITEHTUURIT

Tässä luvussa perehdytään tarkemmin plugin-arkkitehtuureihin. Aluksi kerrotaan arkkitehtuurien yleisistä periaatteista, jonka jälkeen tutustaan kolmeen arkkitehtuuriin.

3.1 Yleistä arkkitehtuureista

Chatley ym. (2003, 1) uskovat, että on mahdollista luoda joustava plugin-arkkitehtuuri, joka mahdollistaa sovellukseen tehdyt muutokset ajonaikana. Plugineja käytetään tällä hetkellä useissa eri sovelluksissa, mutta monesti varsin rajoitetusti. Ohjelma voi rajoittaa pluginin ominaisuuksia tai pluginin ohjelmointi voi olla hankalaa. Oriezy ym. (1998, 3-4) esittävät, että ohjelmisto-arkkitehtuurilta vaaditaan kolme ominaisuutta, jotta sitä käyttävää sovellusta voitaisiin huoltaa ilman sammuttamista. Nämä ominaisuudet ovat:

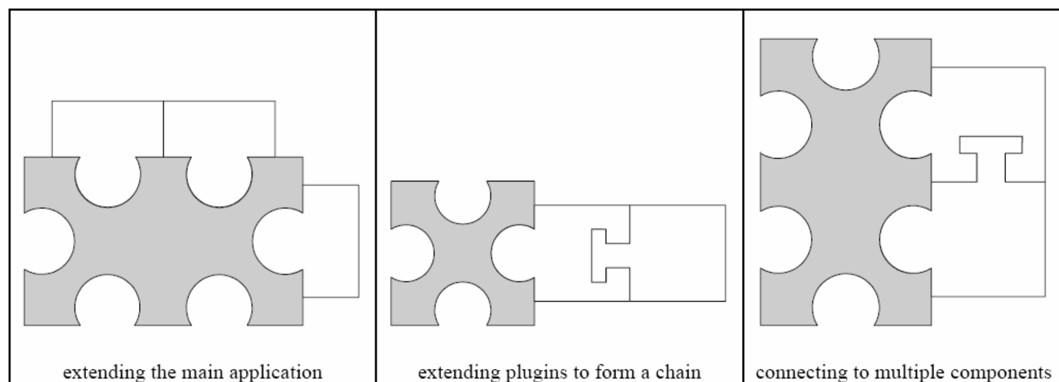
- komponenttien lisääminen sovelluksen suorituksen aikana
- komponenttien poistaminen sovelluksen suorituksen aikana
- komponenttien korvaaminen uusilla versioilla suorituksen aikana.

Pluginit vaativat toimiakseen sovelluksen, joka huolehtii niiden lataamisesta ja käynnistämisestä. Sovellus ajetaan yleensä kehyksen sisässä. Sovellus määrittää avoimia pisteitä, joihin pluginit voivat kytkeytyä ja kehys huolehtii pluginien hallinnoinnista. Pluginin täytyy toteuttaa avoimen pisteen vaatimukset, jotta se pystyy kytkeytymään tiettyyn pisteeseen. Avoimia pisteitä kutsutaan yleensä aktiivialueiksi. Jos useampi eri sovellus käyttää samaa kehystä pluginien hallintaan, plugineja voidaan käyttää eri sovelluksissa. Pluginit osaavat yleensä käyttää ulkopuolisten resurssien lisäksi myös sovelluksen tarjoamia ominaisuuksia. (Marquardt 1999, 4).

Chatley ym. (2003, 3) näkevät plugin-arkkitehtuurin olevan kuin palapeli, jossa oikean muotoiset komponentit sopivat yhteen (KUVA 1). Komponentin muoto

ratkaisee, minne sen voi pistää. Plugineja tukeva sovellus tarjoaa pohjan palapelille. Siinä on joukko avoimia pisteitä, joihin pluginit voivat kiinnittyä tappien avulla. Pluginit ovat valinnaisia, sovellus toimii myös ilman niitä. Avoimet pisteet edustavat sovelluksen tuntemia rajapintoja ja tapit plugineissa olevia luokkia, jotka toteuttavat nämä rajapinnat. Jos sovelluksessa on avoin rajapinta ja plugin toteuttaa rajapinnan vaatimukset (esimerkiksi tietyt metodit), voidaan niiden välille tehdä kytkös. (Chatley ym. 2003, 3). Völter (1999, 2) tarkentaa, että kaikki samantyyppiset pluginit toteuttavat saman rajapinnan, joten ne käyvät samaan sovelluksen tarjoamaan avoimeen pisteeseen. Jokaisen pluginin toteutus voi kuitenkin erota toisesta.

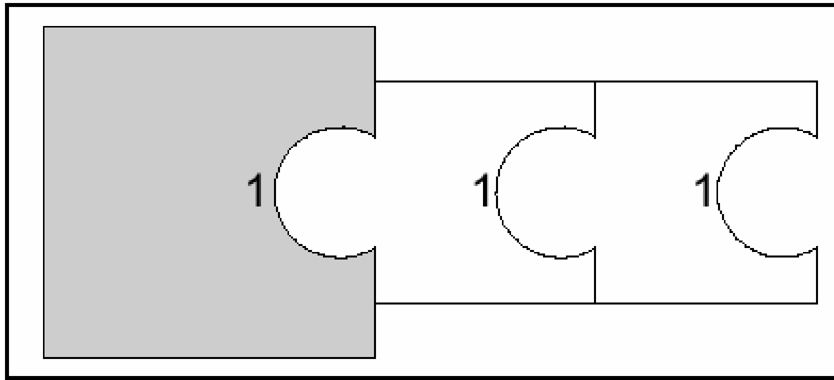
Edellä esiteltyä arkkitehtuuri-esimerkkiä voidaan laajentaa, jos myös pluginit voivat sisältää omia avoimia pisteitä. Tämä mahdollistaisi pluginit, jotka laajentaisivat toisia plugineja, eivät sovellusta (KUVA 1). Lisäksi on mahdollista, että komponentilla on lukuisia avoimia pisteitä ja tappeja. Näin komponenttien rakenteesta voi tulla monimutkainen. Palapeli-esimerkki ei riitä kuvaamaan kaikkia mahdollisia vaihtoehtoja, joita eri komponentteja yhdistelemällä voi syntyä. (Chatley ym. 2003, 3).



KUVA 1. Plugin-pohjaisen sovelluksen arkkitehtuuri. Chatley ym. (2003, 1)

Palapeli-esimerkkiä tarkastelemalla käy selväksi, että olisi käytännössä mahdollista luoda sovelluksia, joissa olisi loputon määrä plugineja. Tietyissä

tapauksissa voidaan kuitenkin haluta rajoittaa pluginien lukumäärää. Tähän voidaan joutua esimerkiksi silloin, kun käytettävät järjestelmäresurssit ovat rajalliset ja jokainen asennettu plugin vie tietyn määrän vapaista resursseista. Rajoitteiden avulla voidaan tarkemmin määritellä miten komponentteja voidaan lisätä. Käytännössä rajoitteet toimivat niin, että tiettyyn koloon kytkeytyvien pluginien määrä määritetään esimerkiksi kahteen (KUVA 2). Usein kuitenkin rajoitteita ei ole, vaan koloon voidaan kytkeä loputtomasti tappeja. (Chatley ym. 2003, 3).



KUVA 2. Rajoitteet plugin-pohjaisissa sovelluksissa. Chatley ym. (2003, 3)

Plugin-pohjainen sovellus voidaan jakaa kahteen osaan: fyysiseen ja loogiseen. Fyysiseen osaan kuuluu ohjelman ajaminen. Looginen osa sisältää ohjelman toiminnallisuuden. Fyysinen osa kuuluu sovellukselle, joka päättää milloin mitäkin pluginia käytetään. Sovellus saattaa myös sisältää hieman loogista toiminnallisuutta. Suurin osa toiminnallisuudesta on kuitenkin pluginien vastuulla. Sovellus ei välitä pluginin sisäisestä logiikasta. Plugin voidaan toteuttaa esimerkiksi irrallisena Dll-tiedostona. Pluginia hyödyntävä sovellus päättää, koska plugin aktivoidaan ja millä ehdoilla. Plugin ei siis toimi yksinään, vaan se on aina riippuvainen sitä ajavasta sovelluksesta. Koska sovellus on itsestään toimiva kokonaisuus, voidaan se toimittaa ilman

plugineja. Jos kuitenkin tietty plugin vaaditaan, jotta sovellusta voitaisiin käyttää, pitää plugin toimittaa sovelluksen rinnalla. (Marquardt 1999, 3).

Sovellus määrittää esimerkiksi kovalevylle hakemiston, josta se etsii saatavilla olevia plugineja. Jokainen uusi plugin asennetaan kyseiseen hakemistoon. Tämä on helpoin ratkaisu toteuttaa plugineja tukeva sovellus. (Marquardt 1999, 15).

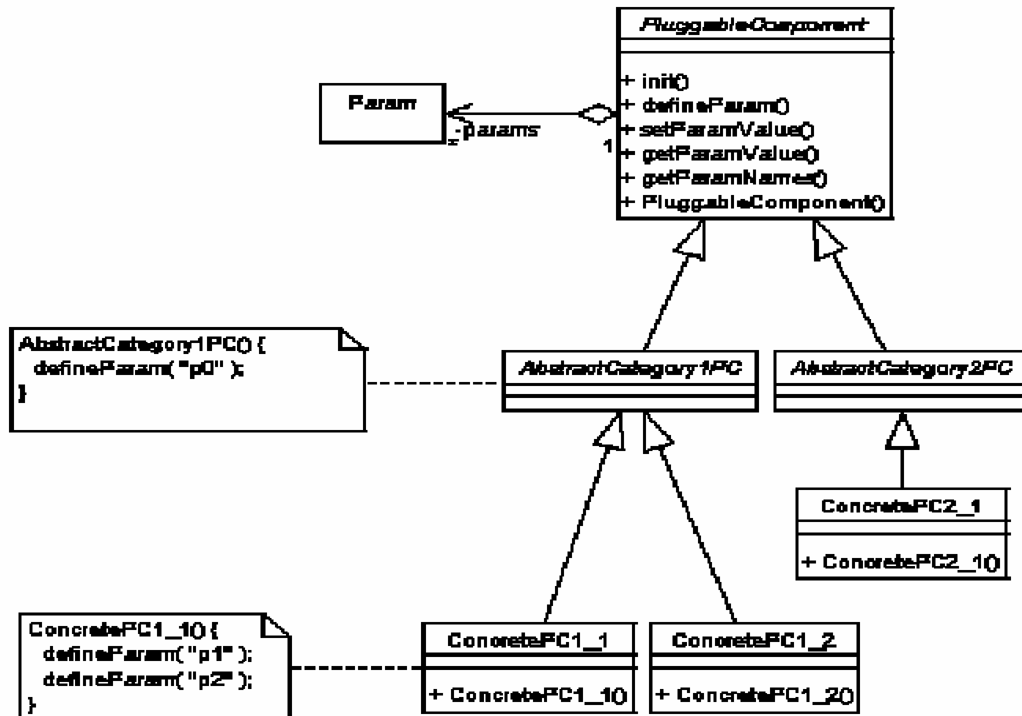
3.2 PluggableComponent

Seuraavassa esitellään Völterin (1999) näkemys plugin-arkkitehtuurista. PluggableComponent-arkkitehtuuri mahdollistaa yksittäisten komponenttien lisäämisen sovellukseen sekä näiden asetusten muokkaamisen parametrien avulla. Komponenttia kutsutaan PluggableComponentiksi. Arkkitehtuuri sisältää rekisterin, joka hallinnoi erityyppisiä plugineja. PluggableComponent sisältää myös tavan tallentaa ja siirtää komponentteja. Tätä arkkitehtuuria voidaan hyödyntää tilanteissa, joissa sovellus sisältää monia avoimia pisteitä, joiden toiminnallisuutta pitäisi pystyä hallinnoimaan ajonaikana. Jokainen komponentti voi tarvita eri parametrit asetuksiaan varten. Arkkitehtuuri hyödyntää luokkien lataamista sovellukseen dynaamisesti. Tämän takia ohjelmointikielen täytyy tukea tätä ominaisuutta. PluggableComponent on toteutettu Java-kielellä.

Jokaista sovelluksen tarjoamaa avointa pistettä kohden on yksi abstrakti luokka, joka periytyy PluggableComponent-luokasta (KUVA 3). Jokainen avoimen pisteen toteuttava konkreettinen luokka, eli plugin, periytyy abstraktista luokasta. Abstrakti luokka toimii rajapintana sovelluksen tarjoamaan aukkoon. Siihen voi olla useita erilaisia konkreettisia toteutuksia. Jokainen konkreettinen luokka voi määritellä sen tarvitsemat asetus-parametrit. Kun komponentin asetukset ovat kunnossa, serialisoidaan se ja tallennetaan yhdessä asetustensa

kanssa. Kun komponenttia seuraavan kerran käytetään, palautetaan se ensin takaisin normaaliin muotoonsa.

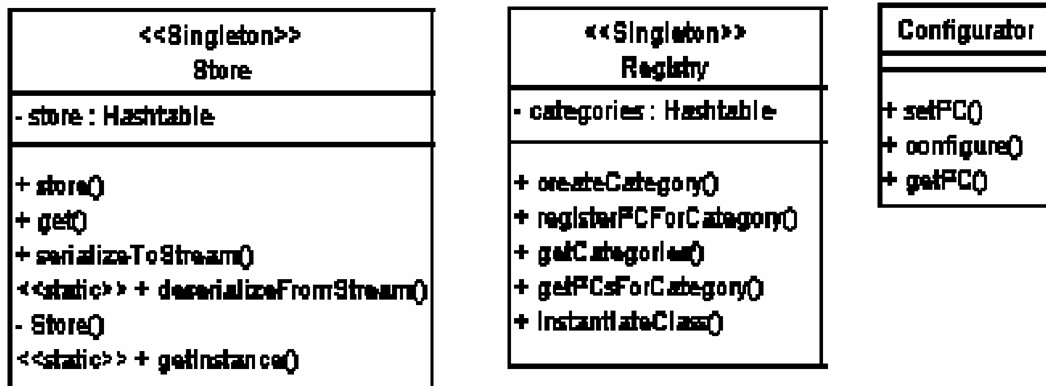
Abstrakti luokka PluggableComponent toimii pohjana jokaiselle komponentille. Se sisältää kokoelman parametreja ja sisältää toiminnot, jotka huolehtivat parametrien hallinnasta. Nämä sisältävät tiedot joita tarvitaan, kun sovellus koittaa ajaa sitä. Jokainen parametri täytyy nimetä. Tämä voidaan toteuttaa esimerkiksi konkreettisen luokan konstruktorissa tai abstraktissa luokassa, jos jokaisella konkreettisella osalla on samat parametrit. Kun komponentit parametrien nimet on kerrottu, toimitetaan sille parametrien arvot.



KUVA 3. PluggableComponent-arkkitehtuurin luokkakaavio. Völter (1999, 4)

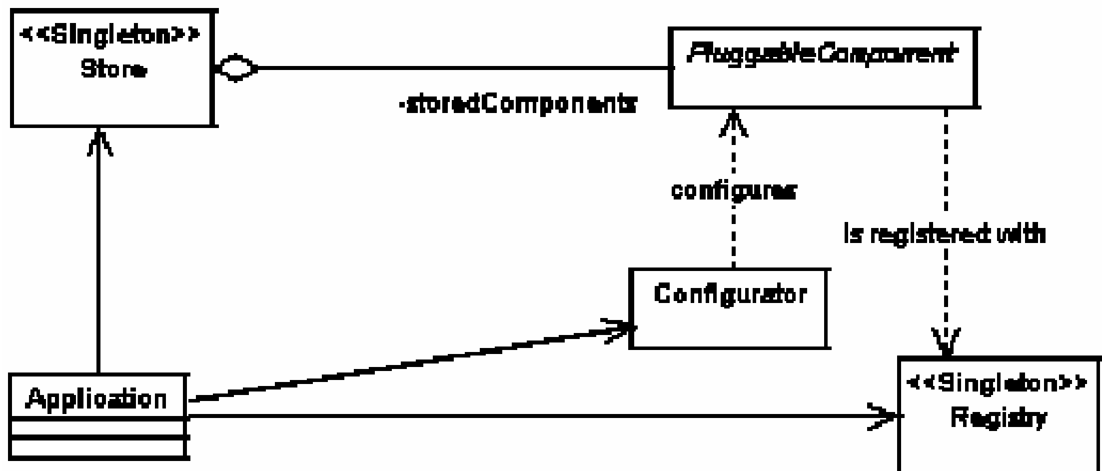
Sovelluksessa, joka käyttää PluggableComponenttia on todennäköisesti monia avoimia pisteitä. Jokaista avointa pistettä kohden on yksi abstrakti luokka, joka toteuttaa avoimen pisteen vaatimukset. Konkreettinen luokka toteuttaa

abstraktin luokan. Jokaista abstraktia luokkaa kohden voi olla lukuisia konkreettisia luokkia, joista jokainen voi käyttäytyä erilailla ja hyödyntää eri parametreja. Nämä pluginit voidaan kytkeä sovelluksen avoimiin pisteisiin hyödyntämällä Configurator-luokkaa. Konkreettiset luokat voivat määritellä tarvitsemansa parametrit defineParam-metodin avulla, omassa konstruktorissaan. Parametrien arvot annetaan myöhemmin Configurator-luokan avulla. Völterin mukaan yksittäisen pluginin asetusten hallinta on tämän arkkitehtuurin tärkein ominaisuus. Configurator-luokan täytyy olla tietoinen tällä hetkellä sovellukseen kytketyistä kategorioista (avoimen pisteen täyttävistä abstrakteista luokista). Tätä varten hyödynnetään Registry-luokkaa. Luokka toimii sovelluksen rekisterinä eri plugineihin. Se sisältää kaikki sovelluksen plugin-kategoriat sekä saatavilla olevat konkreettiset luokat. Registry-luokka sisältää metodit, joiden avulla voidaan sovellukseen lisätä uusia kategorioita ja plugineja, sekä palauttaa tietoa sovelluksen tämän hetkisistä plugineista. Kun plugineille on annettu näiden tarvitsemat parametrien arvot, tallennetaan pluginit. Tähän hyödynnetään Store-luokkaa. Se sisältää kaikki pluginit. Jokaisella pluginilla on Store-luokassa uniikki nimi. KUVA 1**Error! Reference source not found.** sisältää PluggableComponent-arkkitehtuurissa käytettävät luokat ja KUVA 5 sisältää arkkitehtuuria hyödyntävän sovelluksen yleisen rakenteen.

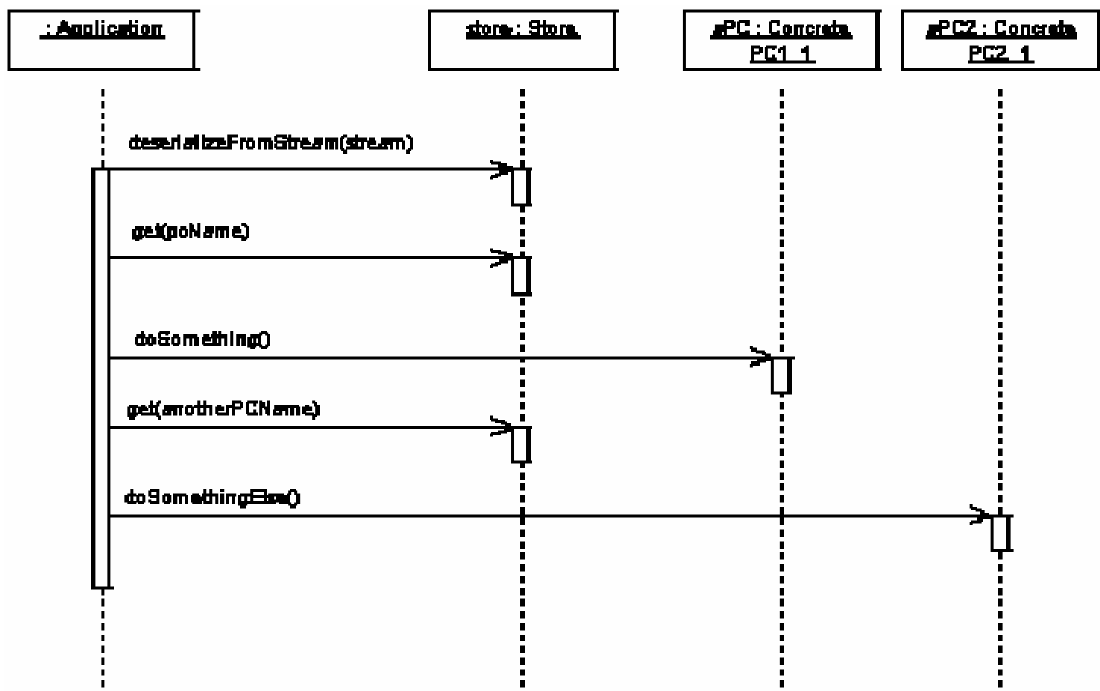


KUVA 4. PluggableComponent-arkkitehtuurin muut luokat. Völter (1999, 5)

Kun sovellus haluaa käyttää tiettyä pluginia, se lataa ensin Store-luokan sisällön. Tämän jälkeen se voi kutsua tiettyä pluginia komponentin nimen perusteella. KUVA 6 sisältää pluginin käyttämisen prosessin.



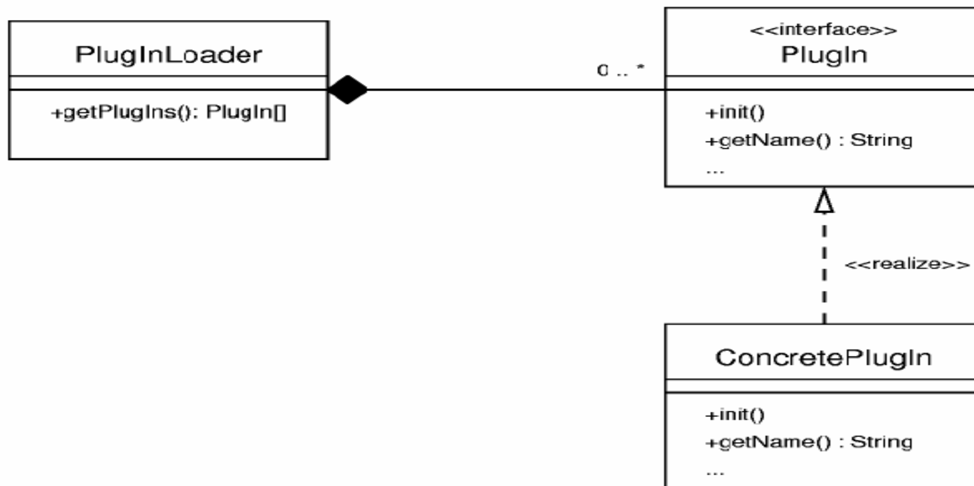
KUVA 5. PluggableComponent-pohjaisen sovelluksen yleinen arkkitehtuuri. Völter (1999, 5)



KUVA 6. Pluginin käyttäminen PluggableComponent-arkkitehtuurissa. Völter (1999, 9)

3.3 Lightweight Plug-In-Based Application Development

Seuraavassa esitellään Mayerin ym. (2002) ehdotus plugin-pohjaisen sovelluksen arkkitehtuurista. Arkkitehtuurista käytetään myöhemmin lyhennettä LPA. He esittävät, että arkkitehtuuri koostuu pääasiassa kolmesta eri luokasta. PlugIn-luokka toimii rajapintana, joka määrittelee tietyn tyyppisen pluginin ominaisuudet. Se sisältää metodin getName, jonka avulla päästään käsiksi pluginin nimeen sekä metodin init. Mahdollisesti PlugIn-luokkaa voidaan laajentaa kattamaan tilanne, jossa sille halutaan antaa mahdollisuus päättää suoritetaanko se tietyssä tilanteessa. ConcretePlugIn-luokka toteuttaa PlugIn-rajapinnan. Se täyttää PlugIn-luokan asettamat vaatimukset. Se voi periytyä muista luokista ja mahdollisesti myös käyttää sovelluksen tarjoamia omia palveluja hyväkseen. PlugInLoader-luokka huolehtii sovelluksen ja pluginien yhteydestä. Se tarpeen vaatiessa etsii oikean tyyppiset pluginit ja kysyy niiltä, mikä suoritetaan. Katso KUVA 7.

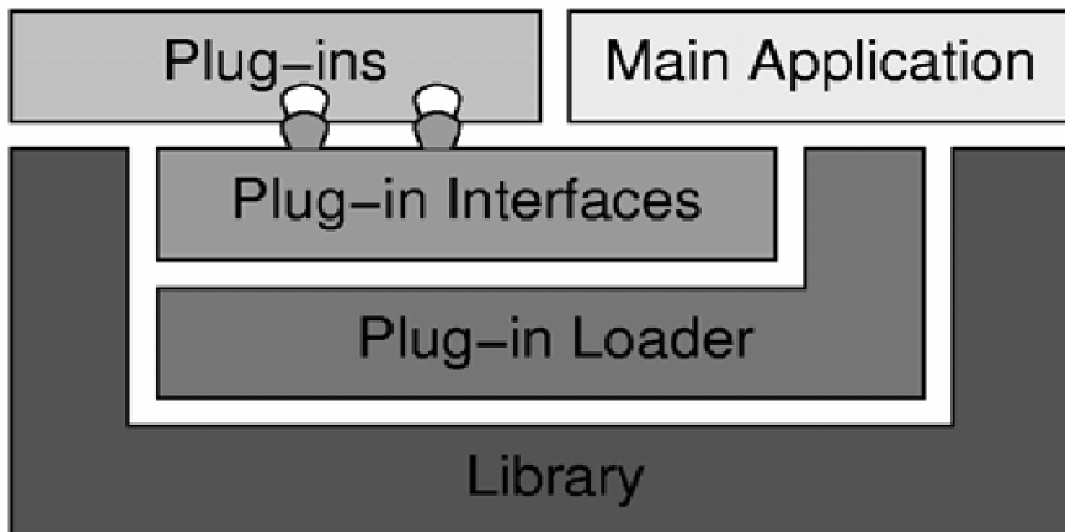


KUVA 7. LPA-arkkitehtuurin luokkarakenne. Mayer ym. (2002, 4)

PlugInLoader-luokka päättelee pluginien nimet kutsumalla niiden getName-metodeja ja mahdollistaa niiden suorittamisen init-metodin avulla. Sovellus, joka hyödyntää PlugInLoader-luokkaa pääsee pluginien toimintoihin käsiksi

hyödyntämällä PlugInLoader- ja PlugIn-luokkia. Sovellus voi tarjota plugineille ominaisuuksia käyttöönsä, jolloin ConcretePlugin voi hyödyntää näitä.

LPA-arkkitehtuuri mahdollistaa useita eri plugin-rajapintoja, mutta yleensä vain yksi plugin kerrallaan toteuttaa kyseisen rajapinnan. Harvinaisissa tapauksissa plugin toteuttaa monta rajapintaa. Sovellus voi tarjota pluginien käyttöön luokkakirjaston. Luokkakirjasto sisältää yleisiä toimintoja, joita pluginit voivat hyödyntää. Ilman tätä kirjastoa, useat pluginit tai sovellus joutuisivat toteuttamaan nämä toiminnot. Sovelluksen lisäominaisuudet on toteutettu yksittäisillä plugineilla. Pluginien tyyppi vaihtelee niiden toteuttamien rajapintojen mukaan. Sovelluksen suorituksen aikana pluginit ladataan PlugInLoaderin avulla, mahdollisesti sen jälkeen, kun sovellus pyytää tiettyä toimintoa. PlugInLoader-luokka hallinnoi kaikkia plugineja, tosin erilaisia plugineja varten voi olla erilainen PlugInLoader. Yleensä pluginit aluksi alustetaan ja tämän jälkeen niiden nimet haetaan. Kun sovellus haluaa kommunikoida tietyn pluginin kanssa, kutsuu se PlugInLoaderia. Kaikki sovelluksen osat, joita ei pystytä mallintamaan plugineina, kuuluvat sovellukseen.



KUVA 8. LPA-pohjaisen sovelluksen yleinen arkkitehtuuri. Mayer ym. (2002, 6)

Jotta arkkitehtuuri toimisi, tietyn rajapinnan toteuttavat luokat pitää pystyä päättelemään sovelluksen suorituksen aikana. Usein tämä voidaan tehdä tutkimalla tietyn hakemiston tiedostoja. Kuitenkin on välttämätöntä, että ohjelmointikieli tukee käännösvaiheessa tuntemattomana pysyneen luokan lataamista suorituksen aikana. Ohjelma lukee hakemistoja ja Jar-tiedostoja. Luokalla, joka toteuttaa tietyn rajapinnan pitäisi olla parametriton konstruktori.

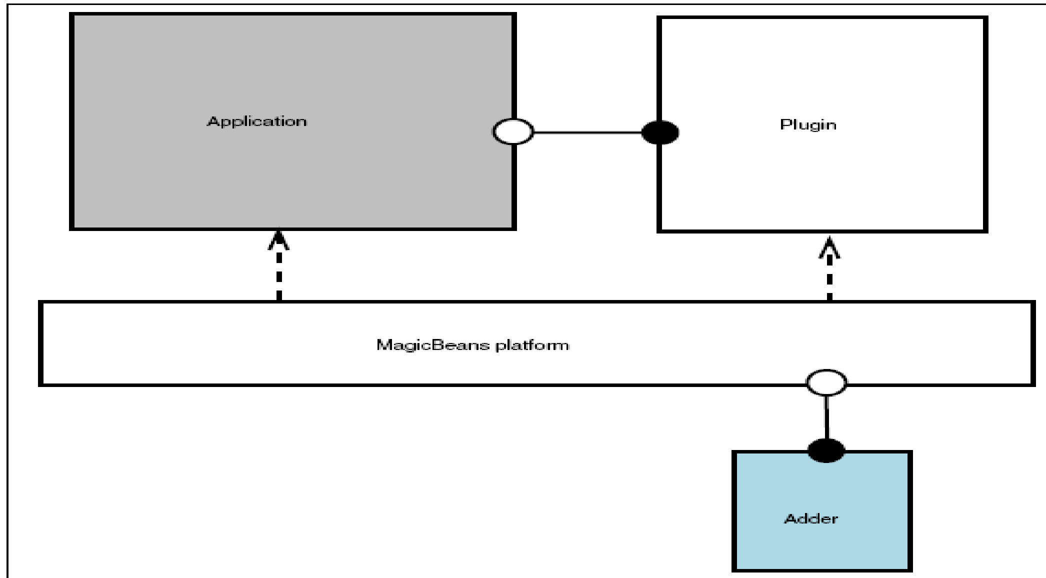
3.4 MagicBeans

Chatley ym. (2004) esittävät plugin-arkkitehtuuriksi kehittämäänsä MagicBeans-järjestelmää. Tavoitteena heillä oli luoda arkkitehtuuri, joka mahdollistaisi uusien komponenttien lisäyksen sovellukseen jälkikäteen. Lisäysten pitäisi tapahtua automaattisesti, ilman, että käyttäjä joutuu kertomaan järjestelmälle tietoja komponenttien sisällöstä. Arkkitehtuuri pystyisi siis päättelemään komponentin sijoituspaikan sen tarjoamien luokkien perusteella. Muina vaatimuksina Chatley ym. (2003, 4) esittävät plugin-arkkitehtuurilleen muuan muassa sen, että MagicBeansia käyttävä sovellus käynnistyy sen sisään. MagicBeans-alusta käynnistää sovelluksen ja tämän jälkeen hallinnoi sovelluksen käyttämiä plugineja. Vaatimuksena on lisäksi, että alusta hoitaa automaattisesti pluginien kytkemisen sovellukseen. Pluginien pitäisi pystyä työskentelemään keskenään. Useita plugineja voitaisiin siis ketjuttaa. Uusia plugineja ei pitäisi joutua erikseen asentamaan, vaan riittäisi, jos alustalle kerrottaisiin uusien komponenttien sijainti. Resurssien hallinnan vuoksi pitäisi myös olla mahdollista määrittää rajapinnoille rajoitteita sen suhteen kuinka monta pluginia niihin voidaan kytkeä. Viimeisenä vaatimuksena on, että tilanteissa, joissa plugin on mahdollista kytkeä useaan eri avoimeen pisteeseen, plugin-alusta osaisi päätellä, mihin yhteys tehdään.

MagicBeans-järjestelmä on kehitetty Java-alustalle. MagicBeans mahdollistaa sovelluksen koostamisen komponenteista, joista jokainen koostuu tietystä määrästä Java-luokkia sekä resurssitiedostoja. Luokat ja niiden käyttämät

tiedostot on pakattu Jar-tiedostoon. Uuden pluginin lisäyksen yhteydessä järjestelmä käy läpi komponentin tarjoamat luokat ja rajapinnat. Näiden tietojen pohjalta se päättelee, mitä palveluita plugin tarjoaa ja miten se voidaan kytkeä tällä hetkellä järjestelmässä oleviin komponentteihin. Luokan toteuttama rajapinta määrittää sen, mihin osaan järjestelmää uusi plugin kytketään.

Tapaus, jossa plugin kytketään toiseen pluginiin, on hankalampi. Uuden pluginin palvelujen hyödyntämistä varten täytyy ensimmäisen pluginin saada viite toiseen pluginiin. Tämä on toteutettu hyödyntämällä ilmoitus-mekanismia. Plugin voi staattista metodia kutsumalla ilmoittaa olevansa kiinnostunut uusista plugineista, jotka toteuttavat tietyn rajapinnan. Uuden pluginin lisäyksen yhteydessä kaikkia siitä kiinnostuneita plugineja informoidaan muutoksesta. Niille välitetään viite uuden pluginin välipalvelimeen eli proxyyn. Tämän avulla pluginit voidaan tarvittaessa poistaa turvallisesti. Järjestelmään kytkettävän uuden pluginin täytyy toteuttaa määritelty rajapinta eikä luokka saa olla abstrakti. Uusien luokkien lisäys suoritetaan Javan reflection-menetelmällä, joka mahdollistaa luokan tietojen tutkimisen sovelluksen suorituksen aikana. Kun vastaavuus rajapinnan ja luokan välillä löytyy, uusi plugin lisätään järjestelmään. Uudesta luokasta luodaan instanssi ja muita, jo järjestelmään kytkeytyneitä plugineja informoidaan uudesta pluginista. MagicBeans-arkkitehtuuri ei ota kantaa siihen, kuinka uudet pluginit esitellään järjestelmälle. Mahdollisia vaihtoehtoja ovat esimerkiksi pluginin lataaminen graafisen valikon kautta ja tietyn hakemiston sisällön seuraaminen. Mekanismit, joilla uudet pluginit havaitaan, voidaan toteuttaa järjestelmään omina plugeina. Tietyn rajapinnan toteuttava luokka voi kytkeytyä MagicBeans-arkkitehtuurin ja informoida järjestelmää, kun havaitsee uuden pluginin. Tämän jälkeen uusi osa lisätään järjestelmään edellä esitellyllä tavalla.



KUVA 9. MagicBeans-pohjaisen sovelluksen yleinen rakenne. Chatley ym. (2004, 3)

MagicBeans-arkkitehtuuri osaa myös poistaa pluginеja. Poistamisen yhteydessä järjestelmä raportoi asiasta jokaiselle siitä kiinnostuneelle pluginille. Tämä aiheuttaa ongelman, sillä plugin ei välttämättä ilmoita olevansa kiinnostunut muiden osien poistosta. Näin plugineihin saattaa jäädä viittauksia jo poistettuihin osiin. Tämän takia uuden pluginin lisäyksen yhteydessä asiasta kiinnostuneille muille komponenteille välitetään viite pluginin proxyyn, ei itse pluginiin. Komponentin poiston yhteydessä proxyn viite komponenttiin voidaan katkaista, jolloin muihin plugineihin ei voi jäädä viittauksia jo poistettuihin osiin. Jotta plugin voitaisiin poistaa turvallisesti se ei saa olla poistamisen aikana suorittamassa tehtävää, jonka toinen komponentti on pyytänyt sitä tekemään. MagicBeans kiertää tämän ongelman synkronoimalla proxy-olioiden tarjoamat metodit niin, että komponenttia ei voida poistaa, jos sen suorittama tehtävä on kesken.

Chatleyn ym. (2004) kehittämä MagicBeans-järjestelmä tukee pluginin korvaamista uudella. Jotta uusi versio pluginista voisi korvata vanhan, täytyy varmistaa, että se tarjoaa vähintään samat palvelut kuin vanha, aikaisemmin

käytössä ollut versio. Käytössä olevat proxy-oliot mahdollistavat sen, että muita plugineja ei tarvitse informoida uudesta versiosta. Proxyn viittaukset ohjataan uuteen komponenttiin, jolloin muut järjestelmän osat eivät huomaa yhden komponentin vaihtuneen.

3.5 Vertailua arkkitehtuurien välillä

Edellä esitellyt kolme arkkitehtuuria eroavat niin toteutuksen kuin ominaisuuksien puolesta toisistaan. Oriezy ym. (1998, 3-4) esittämät kolme vaatimusta ohjelmisto-arkkitehtuurilta, joka mahdollistaisi sovelluksen huollon ilman sovelluksen päivittämistä, olivat komponenttien lisääminen sovelluksen suorituksen aikana, komponenttien poistaminen sovelluksen suorituksen aikana sekä komponenttien korvaaminen uusilla versioilla suorituksen aikana.

Yksi plugin-arkkitehtuurien käyttökohteista on jatkuvasti päällä olevat sovellukset, kuten kappaleessa 2.4 esitettiin. Vertailtaessa yllä olevia vaatimuksia edellisissä kappaleissa esiteltyihin arkkitehtuureihin, käy ilmi, että ainoastaan Chatleyn ym. (2004) MagicBeans-arkkitehtuuri toteuttaa yllä esitetyt ominaisuudet. MagicBeans mahdollistaa ainoana useiden pluginien ketjuttamisen. Kahdessa muussa arkkitehtuurissa pluginit voidaan kytkeä ainoastaan sovelluksen tarjoamiin aukkoihin.

Kaikki kolme arkkitehtuuria hyödyntävät toiminnassaan teoksessa Budd (1999) esiteltyjä olio-ohjelmoinnin toimintoja, tosin hieman erilailla. PluggableComponent on ainoa, joka nojaa abstraktien luokkien käyttöön, MagicBeans ja LPA hyödyntävät rajapinta-luokkia. (Chatley ym. 2004), (Mayer ym. 2002), (Völter 1999). Tämä rajoittaa PluggableComponent-arkkitehtuurin käyttöä, sillä plugin joudutaan periyttämään aina samasta luokasta. MagicBeans ja LPA mahdollistavat pluginin periytymisen mistä tahansa luokasta. Riittää, että pluginit toteuttavat tietyn rajapinnan. PluggableComponent vaatii, että sovelluksen käyttäjä hallinnoi pluginin

ominaisuuksia manuaalisesti. MagicBeans- ja LPA-arkkitehtuureissa ominaisuuksien hallinnointi tapahtuu automaattisesti.

Jokainen arkkitehtuuri hyödyntää Gamman ym. (1995) esittämiä suunnittelumalleja. PluggableComponent hyödyntää Registry-luokassa Singleton-suunnittelumallia (Völter 1999, 4), (Gamma ym. 1995, 127). LPA-arkkitehtuurin ConretePlugin-luokan pohjana toimii Gamman ym. (1995, 185) Façade-malli (Mayer ym. 2002, 5). MagicBeansissa toteutetut rajoitteet mukailevat Gamman ym. (1995, 175) Decorator-suunnittelumallia (Chatley ym. 2003, 4).

Arkkitehtuureista ainoastaan MagicBeans mahdollistaa pluginien ketjuttamisen. Sen voidaan todeta olevan esitellyistä arkkitehtuureista monipuolisin. Se on ainoa tämän kappaleen alussa esiteyt vaatimukset täyttävä arkkitehtuuri, joka mahdollistaa sovelluksen päivittämisen ilman sen uudelleenkäynnistämistä.

Tässä luvussa käytiin läpi tarkemmin plugin-arkkitehtuureja. Arkkitehtuureista annettiin ensin kuvaus yleisemmällä tasolla, jonka jälkeen perehdyttiin tarkasti kolmeen erilaiseen plugin-arkkitehtuuriin. Lopuksi arkkitehtuurien välisiä eroja tutkittiin.

4 YHTEENVETO

Tässä tutkielmassa on perehdytty plugin-pohjaisten sovellusten arkkitehtuureihin. Aluksi on käsitelty tarkemmin termiä plugin, jonka jälkeen on annettu yleisellä tasolla kuvaus plugin-arkkitehtuureista sekä esitelty kolme erilaista ratkaisua. Ratkaisuja on myös vertailtu toisiinsa.

Tutkimuksessa on todettu, että plugineja voidaan hyödyntää tilanteissa, joissa sovellukseen täytyy lisätä ominaisuuksia julkaisun jälkeen sekä tilanteissa, joissa suuri sovellus halutaan jakaa pienempiin, helpommin hallittaviin osiin. Lisäksi todettiin niiden mahdollistavan jatkuvasti päällä olevien sovellusten päivittämisen ilman sammuttamista sekä kolmansien osapuolien tekemien ominaisuuksien lisäämisen sovellukseen. Toisin kuin perinteiset komponentit, pluginit toimivat vain yhdessä sovelluksessa. Arkkitehtuureista yleisimmäksi lähestymistavaksi on todettu kehyksen hyödyntäminen. Kehys toimii pluginien hallintatyökaluna, jonka sisään plugineja käyttävä sovellus käynnistetään.

Vaikka plugineita hyödynnetään nykyisin lukuisissa sovelluksissa, on niitä tutkittu vähän. Vasta vuonna 1999 annettiin ensimmäinen suunnittelumalli plugin-pohjaiselle sovellukselle. Aihealueen nuoruudesta johtuen siitä ei ole saatavilla paljoa lähdemateriaalia. Tässä tutkimuksessa on keskitetty esittelemään kolme erilaista arkkitehtuuria. Ainoastaan MagicBeans-arkkitehtuuri mahdollistaa pluginien lisäämisen, poistamisen sekä korvaamisen sovelluksen suorituksen aikana. PluggableComponent-arkkitehtuurin suurin rajoitus on sen vaatima jokaisen pluginin manuaalinen asetusten hallinta. LPA on ominaisuuksiltaan lähempänä MagicBeans-järjestelmää. LPA ei kuitenkaan mahdollista pluginien ketjuttamista. Ominaisuuksiensa puolesta on mahdollista pistää arkkitehtuurit järjestykseen: MagicBeans on kattavin, LPA:n seuratessa toisena perässä. PluggableComponent on ratkaisuista rajoitetuin. On kuitenkin huomattava, että arkkitehtuurien iät kulkevat rinnakkain niiden

ominaisuuksien kanssa. MagicBeans on nuorin ja PluggableComponent vanhin. Ikäeroa näillä on 5 vuotta.

Selväksi ongelmaksi plugin-pohjaisten sovellusten suunnittelussa on noussut esiin avoimien pisteiden tekeminen. Pluginit kytketään sovelluksen tarjoamiin avoimiin pisteisiin. Sovelluksen luontivaiheessa ohjelmoija ei välttämättä osaa kuvitella jokaista mahdollista tapaa laajentaa ohjelmaansa, jolloin häneltä saattaa jäädä määrittämättä tärkeitäkin avoimia pisteitä. Jos avoin piste puuttuu, sovelluksen ominaisuuksia ei voi päivittää tai laajentaa.

Tässä tutkimuksessa on käsitelty vain kolme eri arkkitehtuuria. Ne ovat ainoat, joiden mallit ovat yleisesti saatavilla. Jatkokehityksenä tai toisena lähestymistapana tutkimukselle on perehtyminen nykyisin käytössä oleviin sovelluksiin, jotka hyödyntävät plugineja. Yksi eniten plugineja hyödyntävä ohjelma on Eclipse. Eclipse on kehitysympäristö, joka itsessään koostuu vain kehyksestä (Object 2003). Jokainen sen toiminto on toteutettu erillisenä pluginina. Olisi hyödyllistä verrata Eclipsen plugin-arkkitehtuuria ja se ominaisuuksia tässä tutkielmassa esiteltyihin malleihin.

LÄHDELUETTELO

Budd T. 1999. An Introduction to Object-Oriented Programming. Addison-Wesley, Reading, Massachusetts.

Chatley R., Eisenbach S., Magee J. 2003. Painless Plugins. Imperial College London. Saatavilla www-muodossa <http://www.doc.ic.ac.uk/~rbc/writings/pp.pdf>

Chatley R., Eisenbach S., Magee J. 2004. MagicBeans: a Platform for Deploying Plugin Components. Teoksessa Proceedings of the CD'04, 99-106. Saatavilla www-muodossa <http://www.doc.ic.ac.uk/~rbc/writings/cd04.pdf>

Gamma E., Helm R., Johnson R., Vlissides J. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Marquardt K. 1999. Patterns for Plug-Ins. Teoksessa Proceedings of the Fourth European Conference on Pattern Languages of Programming and Computing, EuroPLoP '99, Bad Irsee, Germany. Saatavilla www-muodossa < http://st-www.cs.uiuc.edu/~plop/plop99/initial-papers/Marquardt/pluginpatterns3.pdf>

Mayer J., Melzer I., Schweiggert F. 2002. Lightweight Plug-in-Based Application Development. Saatavilla www-muodossa <http://www.springerlink.com/index/L3BQLWC6PKPEA57M.pdf>

Object Tehnology International, Inc. 2003. Eclipse Platform Overview. Saatavilla www-muodossa <http://www.eclipse.org/whitepapers/eclipse-overview.pdf>

Völter M. 1999. PluggableComponent – A Pattern for Interactive System Configuration. Saatavilla www-muodossa <http://hillside.net/europlop/HillsideEurope/Papers/PluggableComponent.pdf>

Oriezy P., Medvidovic N., Taylor R. 1998. Arhitecture-based runtime software evolution. Teoksessa ICSE '98. Saatavilla www-muodossa <http://www.ics.uci.edu/~peymano/papers/ICSE98.pdf>