Sääntöpohjainen pääsynvalvonta Javan web-sovelluksissa

Sami Leino

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi / Ohjelmistokehitys
Pro gradu -tutkielma
Ohjaaja: Jyrki Nummenmaa
Toukokuu 2008

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi / Ohjelmistokehitys
Tekijän Nimi: Sami Leino
Pro gradu -tutkielma, 71 sivua
Toukokuu 2008

Tutkielman tarkoituksena on tutkia ja kehittää Javalla toteutettujen web-sovellusten pääsynvalvontaan liittyviä ratkaisuja tilanteessa, jossa erilaisista ryhmätyösovelluksista ja käyttäjien välisestä yhteistoiminnasta on tullut verkossa arkipäivää. Työssä tutkitaan erityisesti deklaratiivisen ohjelmointimallin ja sääntökoneiden soveltuvuutta käyttöoikeuksien mallintamiseen ja hallintaan. Tutkielmassa kuvataan myös karkealla tasolla joitakin aiempia ratkaisuja, joita muun muassa käyttöjärjestelmät, relaatiotietokantajärjestelmät ja Java-ohjelmistot ovat pääsynvalvontaan esittäneet. Sääntökoneiden käyttökelpoisuutta pääsynvalvonnan toteuttamismenetelmänä testataan rakentamalla Drools-sääntökonetta hyödyntävä esimerkkisovellus, ja pohditaan minkälaisin toimenpitein tästä esimerkkisovelluksesta voitaisiin jatkojalostaa tuotantokäyttöön soveltuva ohjelmistokehys.

Avainsanat ja -sanonnat: Java, pääsynvalvonta, web-sovellukset, sääntökoneet, Drools

Sisällys

1. Johdanto ... 1
2. Pääsynvalvonnan menetelmät ... 3
   2.1. Mitä pääsynvalvonta on? ... 3
      2.1.1. Harkinnanvarainen pääsynvalvonta ... 3
      2.1.2. Pakollinen pääsynvalvonta ... 4
      2.1.3. Roolipohjainen pääsynvalvonta ... 5
      2.1.4. Sääntöpohjainen pääsynvalvonta ... 6
   2.2. Perinteisiä pääsynvalvonnan menetelmiä ... 7
      2.2.1. UNIX-tiedostojärjestelmä ... 7
      2.2.2. Relaatiotietokantajärjestelmät ... 8
   2.3. Java-sovellusten pääsynvalvonta ... 10
      2.3.1. Järjestelmätason pääsynvalvonta Java 2 -ympäristössä ... 11
      2.3.2. Java Authentication and Authorization Service (JAAS) ... 12
      2.3.3. Web-sovellusten pääsynvalvonta ... 16
3. Deklaratiivinen ohjelmointi ja sääntökoneet ... 18
   3.1. Deklaratiivisen ohjelmoinnin määrittelyä ... 18
   3.2. Sääntökoneet ... 18
      3.2.1. Eteenpäin ketjuttavat sääntökoneet ... 19
      3.2.2. Taaksepäin ketjuttavat sääntökoneet ... 19
      3.2.3. Milloin käyttää sääntökonetta? ... 20
   3.3. Drools-sääntökone ... 22
      3.3.1. Historia ... 22
      3.3.2. Yleiskuvaus ... 23
      3.3.3. Drools-sääntöjen esitystavoista ... 28
4. Esimerkkisovellus: web-kalenterin pääsynvalvonta ... 35
   4.1. Esimerkkisovelluksen määrittelyä ... 35
      4.1.1. Toiminnalliset vaatimukset ... 35
      4.1.2. Tekniset vaatimukset ... 40
   4.2. Arkkitehtuurikuvaus ... 41
      4.2.1. Toimintotason pääsynvalvonta ... 42
      4.2.2. Instanssitason pääsynvalvonta ... 46
      4.2.3. Pääsynvalvonta käyttöliittymäkerroksessa ... 52
   4.3. Pääsynvalvontasääntöjen hallinta ja esitystavat ... 53
   4.4. Prototyypin arviointia ... 57
      4.4.1. Arkkitehtuuri ... 57
      4.4.2. Sääntöjen esitystapa ja ilmaisuvoima ... 61

4.4.3. Suorituskyky 63
4.5. Valitun ratkaisumallin kritiikkiä 64
5. Yhteenveto 67
5.1. Jatkotutkimusideoita 67

Viiteluettelo 70

1. Johdanto

Vuoden 2004 tienoilla termi ”Web 2.0” alkoi nousta yhä useammin esiin tietojärjestelmiin ja verkkojulkaisemiseen liittyvissä keskusteluissa. Termillä ei niinkään viitattu mihinkään erityiseen teknologiseen edistysaskeleeseen, vaan sen ajatuksena oli pikemminkin kuvata muutosta tavoissa, joilla Internetiä käytämme. Jos World Wide Web oli aiemmin nähty lähinnä kokoelmana erilaisia tekniikoita informaation esittämiseksi tietoverkossa, nousi Web 2.0:n myötä keskeisempään asemaan itse informaatiosisältö ja käyttäjien välinen *yhteistoiminta* (kollaboraatio) [Wikipedia, 2008a].

Aiemmin oli ollut tavallista, että web-sovellusten käyttäjät oli jaettavissa joko *ylläpitäjiin* (administrators) tai *peruskäyttäjiin* (users). Käyttäjien voitiin täten katsoa toimivan järjestelmässä tietyssä *roolissa* (role), joka määritteli sen minkälaisia *toimintoja* (actions) käyttäjä pystyi tietojärjestelmän avulla suorittamaan. Ylläpitäjät hallinnoivat järjestelmässä olevaa informaatiota, ja peruskäyttäjät lähinnä katselivat järjestelmään tallennettua informaatiota tai pystyivät korkeintaan muokkaamaan valikoitua osaa itseään koskevasta informaatiosta. Yhteistoiminta peruskäyttäjien välillä oli harvinaista, usein peruskäyttäjät eivät ylipäätään edes tienneet toistensa olemassaolosta. Koska käyttäjien jaottelu rooleihin oli näin selkeä, oli myös pääsynvalvonnan toteuttaminen varsin vaivatonta.

Web 2.0 -sovellusten (esimerkiksi blogit, wikit ja yhteisösovellukset) yleistymisen myötä myös loppukäyttäjästä tuli yhä useammin sisällöntuottaja. Tämä muutos johti ensinnäkin tietoverkossa olevan informaatiomäärän valtavaan kasvuun ja toisaalta myös tiedonhallinnallisiin ongelmiin. Uusiin mahdollisuuksiin ihastuneet käyttäjät hyödynsivät tietoverkkojen suomia palveluita varsin huolettomasti ymmärtämättä täysin tietoverkossa piilevien riskien laajuutta, ja asettivat tuottamansa informaatiosisällön alttiiksi väärinkäytölle. Tämän ongelman ratkaisemiseksi oli tarpeen kehittää uusia tapoja käyttöoikeuksien hallintaan, ja loppukäyttäjille oli kyettävä tarjoamaan helppo tapa kontrolloida sitä, mitä heidän syöttämällään informaatiolla on luvallista tehdä.

Kuten aiemmin todettiin, web-sovellusten *pääsynvalvonta* (access control) oli aiemmin hoidettu tyypillisesti *roolipohjaisen pääsynvalvonnan* (role-based access control, RBAC) avulla; kullekin käyttäjälle (tai käyttäjäryhmälle) myönnettiin yksi tai useampi rooli, joka määritteli henkilön valtuudet järjestelmässä. Se,

mitä kullakin roolilla oli mahdollista tehdä, oli usein sovellusten ohjelmakoodiin upotettua staattista informaatiota, jonka muuttaminen ei onnistunut ilman järjestelmäpäivitystä. Uudessa tilanteessa, jossa yhä useampi pääsi syöttämään informaatiota järjestelmään, kävi nopeasti selväksi, ettei tämä lähestymistapa enää parhaalla mahdollisella tavalla toimisi.

Voidaankin sanoa, että kun aiemmin käyttäjän rooli oli ollut jossain määrin staattinen ja kohdistunut itse järjestelmään kokonaisuutena, tulisi käyttäjän rooli jatkossa olemaan enemmänkin dynaaminen ja suhteessa järjestelmässä olevaan informaatioon, ei järjestelmään itseensä. Toki edelleen esiintyy tarvetta rajoittaa roolipohjaisesti loppukäyttäjien pääsyä joihinkin järjestelmän toimintoihin, mutta tärkeämpää on kyetä rajoittamaan loppukäyttäjien pääsyä sellaiseen informaatioon, joihin heillä ei pitäisi olla oikeutta päästä käsiksi.

Tässä tutkielmassa ongelman ratkaisutavaksi ehdotetaan *deklaratiivista ohjelmointia* (declarative programming) ja *sääntökoneiden* (rule engines) käyttöä. Valittua lähestymistapaa koetellaan konstruoimalla sääntökoneen ympärille rakentuva toiminnallinen prototyyppi, josta olisi mahdollista jatkojalostaa tuotantokäyttöön soveltuva ohjelmistokehys. Tuotettua prototyyppiä arvioidaan sekä kvantitatiivisilla että kvalitatiivisilla menetelmillä, pääpainon kohdistuessa jälkimmäisiin.

Tämän tutkielman luvussa 2 luodaan yleiskatsaus olemassa oleviin pääsynvalvonnan menetelmiin, ja tarkastellaan esimerkkeinä tiedostojärjestelmän pääsynvalvontaa UNIX-käyttöjärjestelmässä sekä relaatiotietokantajärjestelmien pääsynvalvontaa SQL-92 -standardin puitteissa. Lisäksi tarkastellaan Java-ympäristön tarjoamia ratkaisuja pääsynvalvonnan toteuttamiseen. Luvussa 3 pohditaan deklaratiivisen ohjelmointitavan ja sääntökoneiden roolia nykyaikaisissa tietojärjestelmissä, sekä esitellään Drools-sääntökone esimerkkinä Javalla toteutetusta avoimen lähdekoodin sääntökonetoteutuksesta. Luvussa 4 dokumentoidaan tutkielman yhteydessä syntynyt esimerkkisovellus, ja arvioidaan siinä käytettyjen menetelmien soveltuvuus kohdealueen ongelmien ratkaisemiseen. Luku 5 sisältää yhteenvedon, jossa tutkielman anti nivotaan yhteen.

2. Pääsynvalvonnan menetelmät

2.1. Mitä pääsynvalvonta on?

ATK-sanakirja [2004, s. 186] määrittelee *pääsynvalvonnan* (access control) olevan ne *"toiminnot ja menettelyt, joiden avulla tietojärjestelmään pääsy tai tiedon saanti sallitaan vain valtuutetuille henkilöille tai sovelluksille"*. Tämä tietojärjestelmäkeskeinen näkökulma soveltuu erinomaisesti tämän tutkielman yhteydessä käytettäväksi, mutta kenties on syytä tarkastella pääsynvalvonnan käsitettä aluksi myös hieman laajemmassa kontekstissa.

Yleisesti ottaen voidaan sanoa, että pääsynvalvonnan (josta käytetään monessa asiayhteydessä synonyyminä myös termiä pääsynhallinta) avulla voidaan kontrolloida tietyn toimijan oikeutta käyttää jotakin tiettyä resurssia. Resurssi voi olla esimerkiksi fyysinen resurssi (kuten vaikkapa elokuvateatteri, jonne pääsemiseksi tarvitaan pääsylippu), looginen resurssi (pankkitili, josta vain tilin omistaja ja omistajan valtuuttamana työskentelevät pankkitoimihenkilöt voivat tehdä nostoja) tai digitaalinen resurssi (vaikkapa tietokoneella oleva tekstidokumentti, jota vain tietyt henkilöt saavat lukea). Tässä tutkielmassa keskitymme digitaalisten resurssien pääsynvalvonnan mahdollistaviin menetelmiin.

Pääsynhallinnan voidaan katsoa koostuvan *käyttöoikeuksien hallinnasta* (access rights management) sekä *pääsynvalvonnan toimeenpanosta* (enforcement). Termiä *autorisointi* (authorization) käytetään yleisesti synonyyminä pääsynvalvonnalle, etenkin silloin kun puhutaan pääsynvalvonnan toimeenpanosta. Pääsynvalvontaan liittyy olennaisesti myös *todentaminen* (authentication), jonka avulla varmistetaan, että käyttölupaa pyytävä toimija on todellakin se, joka hän väittää olevansa. Pääsynvalvontaa ei voida toimeenpanna ilman luotettavia todentamismenetelmiä. Todentamismenetelmät jäävät tämän tutkielman rajauksen ulkopuolelle, mutta niitä sivutaan hiukan Javan pääsynvalvontamenetelmiä koskevassa osiossa Javan JAAS-laajennoksen (Java Authentication and Authorization Service) yhteydessä. Pääpaino on kuitenkin pääsynvalvontamenetelmillä, joista kerrotaan tarkemmin seuraavissa alakohdissa.

2.1.1. Harkinnanvarainen pääsynvalvonta

Harkinnanvarainen pääsynvalvonta (discretionary access control, DAC) perustuu järjestelmän *toimijoille* (käyttäjät, käyttäjäryhmät) myönnettäviin eksplisiittisiin *käyttöoikeuksiin* (access rights) ja *lupiin* (permissions). Harkinnanvaraisen

pääsynvalvonnan yhteydessä käytetään myös yleisesti käsitettä *pääsynvalvontalista* (access control list, ACL), joka sisältää kaikki tiettyyn kohteena olevaan resurssiin kohdistuvat luvat.

Harkinnanvaraisessa pääsynvalvonnassa resurssiin kohdistuvan käyttöoikeuden myöntää yleensä kohteena olevan tiedon *omistaja* (owner), joka esimerkiksi tiedoston tapauksessa on yleensä se käyttäjä, joka on tiedoston luonut. Kohteen lisäksi luvassa määritetään yleensä ne toiminnot, joita käyttöoikeuden saanut toimija saa tiedolle suorittaa. Joissakin järjestelmissä on myös mahdollista, että käyttöoikeuden saanut voi myöntää edelleen käyttöoikeuksia muille toimijoille. Esimerkkinä harkinnanvaraisen pääsynvalvonnan toteutuksesta toimii UNIX:in tiedostojärjestelmä, joka on esitelty alakohdassa 2.2.1.

Harkinnanvaraisen pääsynvalvonnan etuna on se, että kunkin yksittäisen toimijan käyttöoikeudet on mahdollista määrittää varsin tarkasti. Tämä voi kuitenkin joissakin tilanteissa olla samalla myös haittatekijä, eli erillisten käyttöoikeusmääritysten määrä voi järjestelmätasolla kasvaa hyvinkin suureksi ja laajan käyttöoikeuskokoelman hallinta voi tällöin olla varsin työlästä. Tätä pidetään yleisesti yhtenä suurena syynä web-sovelluksissa laajaa suosiota nauttivan roolipohjaisen pääsynvalvonnan menestymiseen.

Silloin kun tietojärjestelmä sisältää luottamuksellista tai salaiseksi luokiteltavaa informaatiota, harkinnanvaraista pääsynvalvontaa pidetään riittämättömänä ja riskialttiina menetelmänä. Harkinnanvaraisessa pääsynvalvonnassa käyttäjä saattaa huolimattomuuttaan tai väärinymmärryksen seurauksena saattaa hallinnassaan olevaa informaatiota vääriin käsiin. Tämä on mahdollista erityisesti silloin, jos tietojärjestelmä on otettu käyttöön liian nopealla aikataululla, eikä käyttäjien kouluttamiseen ole panostettu riittävästi. Ryhmätyösovellusten tapauksessa harkinnanvarainen pääsynvalvonta on kuitenkin välttämätön menetelmä, koska käyttäjien on pystyttävä antamaan toisilleen eksplisiittisiä oikeuksia tietojärjestelmään syötettyyn tietoon.

2.1.2. Pakollinen pääsynvalvonta

Pakollista pääsynvalvontaa (mandatory access control, MAC) sovelletaan useimmiten korkeiden tietoturvavaatimusten järjestelmissä, esimerkiksi valtionhallinnon tietojärjestelmissä ja sotilastietojärjestelmissä. Tällaisille järjestelmille on tyypillistä se, että tietojärjestelmään syötettävä tieto jaotellaan eri turvallisuusluokkiin, ja tietoa janoavan käyttäjän turvallisuusluokituksen tulee olla vähintään samaa tasoa kohteena olevan tiedon

turvallisuusluokituksen kanssa, jotta käyttölupa voidaan myöntää. Voitaneen siis sanoa, että sekä tietoon että käyttäjään ”lyödään leima”, joka määrittelee hänen valtuutensa järjestelmässä.

Pakollisen pääsynvalvonnan suurimpana erona harkinnanvaraiseen pääsynvalvontaan on se, että käyttäjillä ei ole mahdollisuutta myöntää eksplisiittisiä käyttöoikeuksia tietoihin vaan käyttöoikeuspolitiikka määritetään järjestelmätasolla. Tällä pyritään varmistamaan se, ettei käyttäjä edes huolimattomuuttaan voi saattaa korkean turvallisuusluokituksen alaista tietoa sellaisten tahojen saataville, joilta riittävä turvallisuusluokitus puuttuu.

Pakollinen pääsynvalvonta on tutkimuksen kannalta sikäli mielenkiintoinen menetelmä, että se toteutetaan usein sääntöpohjaisen pääsynvalvonnan avulla. Sääntöpohjaista pääsynvalvontaa ei kuitenkaan tule tulkita pakollisen pääsynvalvonnan erityistapaukseksi, vaan se on vain eräs keino pakollisen pääsynvalvonnan toteuttamiseen. Sääntöpohjaisen pääsynvalvontamenetelmän ilmaisuvoiman avulla voidaan siis toteuttaa pakollista pääsynvalvontaa, mutta myös muita pääsynvalvontamalleja.

2.1.3. Roolipohjainen pääsynvalvonta

Roolipohjaisessa pääsynvalvonnassa käyttäjän oikeus käyttää jotakin tietojärjestelmän toimintoa riippuu käyttäjälle myönnetystä roolista tai rooleista (näitä voi olla useampia kuin yksi). Roolin avulla voidaan kuvata henkilön tehtävää tai toimenkuvaa organisaatiossa, jolloin puhutaan henkilön *liiketoiminnallisesta roolista* (business role). Toisaalta tarkastelunäkökulmana voi olla sovelluksen näkökulma, jolloin puhutaan *sovellusroolista* (application role). Liiketoimintarooleja voisivat olla esimerkiksi ”palkanlaskija” ja ”toimitusjohtaja”, sovellusrooleja vaikkapa ”ylläpitäjä” ja ”peruskäyttäjä”. Usein nämä eri roolityypit sekoittuvat sovelluksissa enemmän tai vähemmän kiinteästi toisiinsa, eikä kaikissa järjestelmässä edes tehdä eroa niiden välille. Tekniseltä kannalta katsottuna roolin voidaan katsoa olevan nimetty joukko yksittäisiä käyttöoikeuksia. Rooleista voidaan muodostaa *roolihierarkioita* (role hierarchies), joiden avulla tietty rooli voidaan omien käyttöoikeuksiensa ohella määrittää sisältämään myös muita rooleja, jolloin määriteltävä rooli perii myös näiden muiden roolien sisältämät käyttöoikeudet.

Myös roolipohjainen pääsynvalvonta määritetään järjestelmätasolla, eli tässäkään mallissa peruskäyttäjillä ei ole mahdollisuutta myöntää eksplisiittisiä käyttöoikeuksia toisilleen. Roolipohjainen pääsynvalvonta on erityisen suosittu tapa pääsynvalvonnan hoitamiseen web-sovelluksissa, koska se on

yksinkertainen ja ylläpidollisesti vähätöinen ratkaisu. Jos esimerkiksi henkilön toimenkuva yrityksessä muuttuu, ylläpitohenkilöstön ei tarvitse muokata kymmeniä tai jopa satoja yksittäisiä käyttöoikeuksia järjestelmässä, vaan käyttäjän roolin vaihtaminen riittää.

Roolipohjainen pääsynvalvonta eroaa siinä mielessä muista pääsynvalvontamalleista, että siinä käyttöoikeuden kohteena ei yleensä ole itse informaatio, vaan toiminto, jolla informaatiota hallitaan. Harkinnanvaraisessa pääsynvalvonnassahan kohteena olevaan informaatioon annettiin eksplisiittisiä käyttöoikeuksia, ja pakollisessa pääsynvalvonnassa kohteena oleva tieto ”leimattiin” turvallisuusluokituksella. Tässä suhteessa roolipohjainen pääsynvalvonta on fundamentaalisesti erilainen pääsynvalvontatapa kuin muut kuvatut pääsynvalvontamenetelmät.

2.1.4. Sääntöpohjainen pääsynvalvonta

Tämän tutkielman kannalta mielenkiintoisin pääsynvalvontatyyppi on kuitenkin sääntöpohjainen pääsynvalvonta (rule-based access control). Se, pääseekö käyttäjä käsiksi haluamaansa tietoon, riippuu järjestelmään ylläpitäjien toimesta syötetyistä säännöistä. Sääntöpohjainen pääsynvalvonta ei siis – pakollisen ja roolipohjaisen pääsynvalvonnan tavoin – tarjoa käyttäjälle mahdollisuutta vaikuttaa itse siihen, kenellä järjestelmän käyttäjällä on oikeus päästä käsiksi hänen luomaansa informaatioon.

Sääntöpohjainen pääsynvalvonta on mielenkiintoinen pääsynvalvontamalli erityisesti siksi, että se tarjoaa huomattavasti muita pääsynvalvontamalleja ilmaisuvoimaisemman tavan vaikuttaa siihen, mihin toimenpiteisiin tai mihin informaatioon käyttäjällä on järjestelmässä pääsy. Näkisin, että voimallisimmillaan sääntöpohjainen pääsynvalvonta olisi nimenomaan täydentäessään muita pääsynvalvontamalleja, ei tulemalla niiden tilalle. Sääntöpohjaisuuden avulla pääsynvalvontaan saadaan lisää monipuolisuutta ja ehdollisuutta; kun esimerkiksi roolipohjaisen pääsynvalvonnan avulla voisimme määrittää, että ”käyttäjienhallintaliittymää saa käyttää vain SUPERADMIN-roolin omaava käyttäjä”, voimme sääntöpohjaisella pääsynvalvonnalla tarkentaa tätä määrittämällä, että ”käyttäjienhallintaliittymää saa käyttää vain SUPERADMIN-roolin omaava käyttäjä, jos hänen IP-osoitteensa on 127.0.0.1 ja hän pyytää käyttöoikeutta aikavälillä 16-08.”.

2.2. Perinteisiä pääsynvalvonnan menetelmiä

Esimerkkinä pääsynvalvonnan menetelmistä esitellään UNIX-tiedostojärjestelmä sekä relaatiotietokantajärjestelmä SQL-92 -standardin puitteissa. Pääpiirteissään molemmat ovat lukijalle varmasti hyvin tuttuja, ja siksi ne soveltuvatkin erinomaisesti sillaksi siirryttäessä pääsynvalvonnan teoriasta pääsynvalvonnan soveltamiseen.

2.2.1. UNIX-tiedostojärjestelmä

UNIX-tiedostojärjestelmän pääsynvalvontamalli perustuu harkinnanvaraisen pääsynvalvonnan menetelmiin. Kullakin tiedostojärjestelmän objektilla (tiedostot ja hakemistot) on eksplisiittiset *tiedosto-oikeudet* (file permissions), joiden avulla määritetään, kuka tiedostoja saa käyttää, ja mitä tiedostoilla voi tehdä. Käyttöoikeuksia voidaan myöntää tiedoston omistajalle (owner), ryhmälle (group) tai muille käyttäjille (others). Jälkimmäiseen joukkoon kuuluvat kaikki muut käyttöjärjestelmän käyttäjät paitsi tiedoston omistaja sekä nimettyyn ryhmään kuuluvat käyttäjät.

Tiedostojen käyttöoikeudet erotellaan lukuoikeuksiin (r), kirjoitusoikeuksiin (w) ja suoritusoikeuksiin (x). Hakemiston kyseessä ollessa lukuoikeus tarkoittaa oikeutta listata hakemiston sisältö, kirjoitusoikeus oikeutta luoda, muokata ja poistaa hakemistossa olevia tiedostoja ja suoritusoikeus muun muassa oikeutta vaihtaa kohteena oleva hakemisto työhakemistoksi (komennolla cd). Suuri osa toiminnoista (kuten hakemistossa olevan tiedoston lukeminen) vaatii hakemistoon suoritusoikeuden, koska käyttöjärjestelmätasolla on pystyttävä etsimään hakemistossa olevan tiedoston kuvaustietue (inode) ja tätä tietoa ei voi saada ilman suoritusoikeutta. Hakemistojen tapauksessa suoritusoikeutta kutsutaankin yleensä hakuoikeudeksi (search). Havainnollistamme tiedosto-oikeuksia hakemistolistauksen (Esimerkki 1) avulla:

```
Wed 09 Apr 23:18 slite:~/workarea>ls -l
total 32
-rw-r--r--   1 slite    admingrp     450 Apr  9 23:17 README.txt
-rwxr-x---   1 slite    admingrp      33 Apr  9 23:18 findwords.sh
drwxr-x---   2 slite    admingrp      80 Apr  9 23:19 storage
```

Esimerkki 1. Hakemistolistaus UNIX-käyttöjärjestelmässä

Tiedosto-oikeudet esitetään 10 merkin pituisena merkkijonona, joka on hakemistolistauksessa kunkin tiedostorivin ensimmäinen alkio. Merkkijonon ensimmäinen merkki (`-` tai `d`) kuvaa sen, onko kyseessä tavallinen tiedosto vai hakemisto. Sen jälkeen tulevat kolmen merkin rykelmät kuvaavat tiedosto-

oikeudet tiedoston omistajalle (`slite`), ryhmälle (`admingrp`) ja muille käyttäjille. Esimerkiksi tiedoston `findwords.sh` tiedosto-oikeudet tarkoittavat, että kyseessä on tavallinen tiedosto (`-`), johon omistajalla on lukuoikeus (`r`), kirjoitusoikeus (`w`) ja suoritusoikeus (`x`). Vastaavasti ryhmällä on tiedostoon lukuoikeus (`r`) ja suoritusoikeus (`x`), mutta ei kirjoitusoikeutta; näin ollen ryhmään `admingrp` kuuluvat käyttäjät saavat suorittaa tiedoston, mutta eivät muokata tai poistaa sitä. Järjestelmän muilla käyttäjillä ei puolestaan ole tiedostoon minkäänlaisia oikeuksia.

UNIX-järjestelmissä järjestelmän pääkäyttäjällä (`root`) on normaalitilanteessa aina täydet valtuudet kaikkeen järjestelmään tallennettuun tietoon. Jos esimerkiksi huolimaton käyttäjä sattuisi vahingossa poistamaan oikeutensa omistamiinsa tiedostoihin (tämä on UNIX-järjestelmässä mahdollista, vaikkapa antamalla esimerkkitapauksessa komennon `chmod 000 findwords.sh`), voi järjestelmän pääkäyttäjä palauttaa entiset käyttöoikeudet voimaan. Tämä pääkäyttäjän implisiittinen oikeus kaikkeen järjestelmään tallennettuun tietoon on kuitenkin toisaalta myös valtava riski, koska pääkäyttäjän käyttäjätilin vaarantuminen vaarantaa samalla koko tietojärjestelmän turvallisuuden. Lisäksi pääkäyttäjän implisiittinen oikeus tarkoittaa sitä, ettei pakollisen pääsynvalvonnan toteuttaminen järjestelmätasolla ole suoraan mahdollista.

UNIX-käyttöjärjestelmän tiedosto-oikeudet tarjoavat kompaktin ja tekstuaalisen tavan määrittää oikeuksia tiedostoihin. Tiedosto-oikeuksien ilmaisuvoima on kuitenkin rajallinen ja niiden puutteet – muun muassa se, että oikeuksia voi antaa vain yhdelle nimetylle ryhmälle eikä oikeuksia voi myöntää yksittäisille käyttäjille – ovat johtaneet vaihtoehtoisten pääsynvalvontatapojen käyttöönottoon eri UNIX- ja Linux-toteutuksissa. Suurimmassa osassa nykyaikaisista UNIX- ja Linux-toteutuksista onkin olemassa jonkinlainen ACL-mekanismi, joka täydentää perinteistä tiedosto-oikeusmallia tarjoamalla mahdollisuuden luoda käyttöoikeuslistoja, joissa tiedostoon kohdistuvat käyttöoikeudet luetellaan eksplisiittisesti. Lisäksi muun muassa NSA on kehittänyt Linux-ytimiin ominaisuuksia (näistä käytetään nimitystä Security-Enhanced Linux eli SELinux), joiden avulla muun muassa pakollisen pääsynvalvonnan toteuttaminen on Linux-järjestelmissä mahdollista [NSA, 2008].

2.2.2. Relaatiotietokantajärjestelmät

Myös relaatiotietokantajärjestelmien pääsynvalvonta pohjautuu pitkälti harkinnanvaraisen pääsynvalvonnan periaatteisiin, mutta niissä on lisäksi roolipohjaiseen pääsynvalvontaan kuuluvia piirteitä. Erityisesti

tietokantajärjestelmän pääkäyttäjän (database administrator, DBA) implisiittiset järjestelmänhallintaoikeudet voidaan mielestäni katsoa kuuluvan roolipohjaisen pääsynvalvonnan piiriin.

Tietokantajärjestelmän pääkäyttäjä vastaa monella tapaa UNIX-käyttöjärjestelmän pääkäyttäjää (root) [Elmasri and Navathe, 1994, s. 597]. Pääkäyttäjällä on tietokantajärjestelmässä oikeus luoda/poistaa tietokantakäyttäjiä ja antaa näille eritasoisia oikeuksia tietokantajärjestelmään. Pääkäyttäjän oikeudet ovat implisiittisiä sikäli, että niitä ei yleisesti erikseen määritellä tietokantajärjestelmien asetustiedostoissa, vaan ne ovat ennalta määrättyjä.

Relaatiotietokantajärjestelmät voivat olla varsin erilaisia, joten tässä yhteydessä keskityn lähinnä siihen, mitä SQL-92 -standardi sanoo pääsynvalvonnasta. SQL-92 -standardin mukaan käyttöoikeudet voidaan määritellä *käyttäjätilin tasolla* (account level) tai *relaation tasolla* (relation level). Käyttäjätiliin kohdistuvat oikeudet pätevät relaatiosta riippumatta, mutta relaatioon kohdistuvat oikeudet vain kyseisen relaation sisällä. SQL-92 mahdollistaa pääsynvalvonnan jopa attribuuttitasolla, sillä käyttäjälle voidaan myöntää oikeus päivittää vain tiettyjä relaation attribuutteja (eli tietokantataulun sarakkeita). Vastaavasti käyttäjälle voidaan tarjota lukuoikeus vain joihinkin relaation attribuutteihin; SQL-92:ssa tällainen rajoite laaditaan *näkymien* (views) avulla.

SQL-92 -standardi määrittelee *omistajan* (owner) käsitteen. Esimerkiksi kullakin tietokantataululla on omistaja; omistajatieto määräytyy aluksi sen perusteella, kuka tietokantataulun on luonut, mutta omistusoikeus on mahdollista siirtää toiselle tietokantakäyttäjälle. SQL-92 määrittelee myös *tietokantakaavion* (database schema) käsitteen, jolla voidaan nivoa yhteen loogisesti yhteenkuuluvat tietokantataulut; myös tietokantakaaviolle voidaan määrittää omistaja.

Tietokantataulun omistaja voi myöntää tauluun käyttöoikeuksia muille tietokantakäyttäjille. Käyttöoikeus voi olla täydellinen, tai mahdollistaa vain tiettyjä toimenpiteitä. Esimerkiksi seuraava lause antaa tietokantakäyttäjälle `VILLE` mahdollisuuden tehdä kyselyitä kohdistuen `EMPLOYEE`-tietokantatauluun:

```
GRANT SELECT ON EMPLOYEE TO VILLE;
```

SQL-92 -standardin mukaisessa käyttöoikeuden myöntämislauseessa kerrotaan siis käyttöoikeuden kohde (EMPLOYEE), se kenelle lupa myönnetään (VILLE) ja se minkä toiminnon (SELECT) käyttöoikeus mahdollistaa. Jos VILLE-käyttäjälle halutaan antaa käyttöoikeuksien edelleenmyöntämisoikeus, voidaan edellinen lause kirjoittaa muodossa:

```
GRANT SELECT ON EMPLOYEE TO VILLE WITH GRANT OPTION;
```

Nyt VILLE voi myöntää samantasoisen oikeuden eteenpäin myös muille tietokantakäyttäjille. Jos tietokannan pääkäyttäjä kuitenkin nyt poistaa VILLE-käyttäjältä oikeuden tehdä kyselyitä EMPLOYEE-tauluun, samalla peruuntuvat myös VILLE-käyttäjän edelleen myöntämät oikeudet. Peruutus voidaan tehdä seuraavalla lauseella:

```
REVOKE SELECT ON EMPLOYEE FROM VILLE;
```

Kuten esimerkeistä nähdään, SQL-92 -standardin määrittämä tapa käyttöoikeuksien hallintaan pohjautuu hyvin vahvasti harkinnanvaraisen pääsynvalvonnan periaatteisiin. Toimijat antavat toisilleen eksplisiittisiä käyttöoikeuksia, joiden avulla määritellään käyttöoikeuden saaja, kohde sekä toiminto, jonka kohteelle voi suorittaa. SQL-92 -standardi ei määrittele tapoja pakollisen pääsynvalvonnan toteuttamiseen, ja varsin harva relaatiotietokantavalmistaja on ylipäätään tällaisia ominaisuuksia tietokantajärjestelmiinsä edes toteuttanut. Rjaibi ja Bird [2004] mainitsevat, että ainakin Trusted Oracle, Informix OnLine/Secure ja Sybase Secure SQL sisältävät pakollisen pääsynvalvonnan mekanismeja, mutta nämä järjestelmät eivät ole kaupallisessa käytössä vaan niiden käyttö on rajattu muun muassa Yhdysvaltain tiedustelupalveluorganisaatioiden sekä puolustusministeriön käyttöön.

2.3. Java-sovellusten pääsynvalvonta

Java-arkkitehtuuri on historiallisesti ollut tunnettu tiukoista tietoturvarajoitteistaan. JDK 1.0:n tietoturvamallina oli niin sanottu *hiekkalaatikkomalli* (sandbox model) [Gong, 2002], jossa verkosta ladattu ja selaimessa suoritettava ohjelmakoodi ei saanut juuri minkäänlaisia oikeuksia käyttöjärjestelmätasolla (muun muassa paikallisten tiedostojen lukeminen tai kirjoittaminen ei ollut tällaiselle ohjelmakoodille sallittua). JDK 1.1 laajensi tätä mallia tuomalla mukaan *allekirjoitetun sovelman* (signed applet) käsitteen; jos sovelma oli allekirjoituksen perusteella luotetun tahon valmistama, se sai

järjestelmään täydet valtuudet. Jos näin ei ollut, sovelma suoritettiin edelleen hiekkalaatikossa.

2.3.1. Järjestelmätason pääsynvalvonta Java 2 -ympäristössä

Vuoden 1998 joulukuussa julkaistu Java 2 (JDK 1.2) muutti Javan pääsynvalvontamallin radikaalilla tavalla. Java 2 esitteli *politiikkatiedostoihin* (policy files) perustuvan pääsynvalvontamallin, jonka avulla pääsyä järjestelmäresursseihin voitiin nyt rajoittaa deklaratiivisesti.

Politiikkatiedostot ovat syntaktisesti varsin yksinkertaisia tekstitiedostoja, joten niitä on helppo muokata aivan tavallisella tekstieditorilla, olkoonkin, että Java-ympäristö tarjoaa tätä varten myös oman työkalun (Policy Tool). Java-ympäristön oletusarvoinen tietoturvapolitiikka on määritetty tiedostossa `$java.home/lib/security/java.policy`; oletuspolitiikka ei juuri rajoita pääsyä tiedostoihin, antaa ohjelmien ”kuunnella” kaikkia muita paitsi yleisesti tunnettuja portteja (portit 1024:sta ylöspäin) ja määrittää joukon ympäristömuuttujia, joita Java-ohjelmat saavat lukea. Järjestelmän oletusarvoisen politiikkatiedoston lisäksi käyttäjän kotihakemistossa voi olla tiedosto `$user.home/.java.policy`, jolla voidaan käyttäjäkohtaisesti hienosäätää järjestelmän oletuspolitiikkaa.

Politiikkatiedostojen rakenteen seikkaperäinen kuvaaminen ei ole tämän tutkielman puitteissa järkevää, mutta lienee kuitenkin paikallaan esittää pieni näyte (Esimerkki 2) Java 2 -arkkitehtuurin mukaisesta käyttöoikeussäännöstä:

```
grant codeBase "file:/home/users/sami/myApp"
{
    permission java.io.FilePermission "/tmp/app.log", "write";
};
```

Esimerkki 2. Javan politiikkatiedostossa määritelty käyttöoikeus

Yllä oleva sääntö määrittää, että kaikki hakemistosta `/home/users/sami/myApp` ladatut Java-luokat saavat kirjoittaa tiedostoon `/tmp/app.log`. On kuitenkin huomionarvoista, että nämä määritykset eivät tietenkään voi syrjäyttää käyttöjärjestelmätason rajoitteita; jos hakemistossa `/home/users/sami/myApp` sijaitsevaa Java-ohjelmaa ajetaan `sami`-käyttäjän oikeuksilla ja tällä käyttäjällä ei ole kirjoitusoikeutta `/tmp` -hakemistoon, ei Java-ohjelmakaan pysty kirjoittamaan kyseiseen hakemistoon, vaan seurauksena on poikkeus.

Politiikkatiedostot tarjoavat siis tavan kuvata deklaratiivisesti sovellusten käyttöoikeuksia. Sovelluksia suoritettaessa näiden käyttöoikeussääntöjen soveltamisesta huolehtii Javan turvamanageri (`java.lang.SecurityManager`), joka delegoi pääosan käyttöoikeuksien tarkistustyöstä JDK 1.2:ssa esitellylle pääsynvalvontakomponentille (`java.security.AccessController`) [Jaworski and Perrone, 2000, s. 63]. Laittoman toiminnon yrittämisestä seuraa poikkeus (`java.lang.SecurityException`), jonka asiakaskoodi voi halutessaan käsitellä; SecurityException on *tarkistamaton poikkeus* (unchecked exception), jota asiakaskoodin ei välttämättä tarvitse käsitellä, vaan asiakaskoodi voi antaa poikkeuksen valua ylemmälle tasolle ohjelmakoodissa. Poikkeus on kuitenkin hyvä käsitellä viimeistään ohjelman päätasolla, jolloin käyttäjälle voidaan antaa ymmärrettävässä muodossa oleva kuvaus virheen syystä.

Turvamanageri voidaan määritellä Javan virtuaalikoneelle sen käynnistymisvaiheessa, joten on täysin mahdollista vaikkapa korvata järjestelmän oletusarvoinen turvamanageri täysin omalla toteutuksella. Tällaisen ratkaisun avulla voitaisiin esimerkiksi käyttää Javan politiikkatiedostoista eroavaa syntaksia käyttöoikeussääntöjen esittämiseen, jos jokin muu syntaksi tuntuisi soveltuvan kohdealueen käyttöoikeussääntöjen esittämiseen perussyntaksia paremmin. Javan politiikkatiedostot soveltuvat erinomaisesti sovellusten ja niiden toimintojen käyttöoikeuksien hallintaan, mutta niiden avulla ei kuitenkaan voida antaa oikeuksia yksittäisille käyttäjille tai käyttäjäryhmille, minkä vuoksi politiikkatiedostot eivät sovellu kovinkaan hyvin web-sovellusten pääsynvalvontaan.

2.3.2. Java Authentication and Authorization Service (JAAS)

Java Authentication and Authorization Service [JAAS, 2008] esiteltiin erikseen ladattavana lisäpakettina JDK-versiossa 1.3, ja integroitiin pysyväksi osaksi Java-arkkitehtuuria JDK-versiossa 1.4. Se laajentaa Javan perustietoturvamallia mahdollistaen muun muassa käyttöoikeuksien myöntämisen yksittäisille käyttäjille ja käyttäjäryhmille. Nimensä mukaisesti JAAS tarjoaa ratkaisuja sekä todentamiseen (authentication), että pääsynvalvontaan (authorization).

JAAS:in todentamismekanismi pohjautuu PAM:iin (Pluggable Authentication Module) [Samar and Schemers, 1995]. Se tarjoaa ohjelmistokehittäjille toteutustavasta riippumattoman rajapinnan käyttäjän todentamiseen; käyttäjä voidaan todentaa esimerkiksi LDAP-hakemistorakennetta vasten ilman, että asiakasohjelmaan tarvitsee kirjoittaa LDAP-spesifistä ohjelmakoodia. Todentamista voidaan yrittää useaa eri menetelmää käyttäen; seuraavassa esimerkissä (Esimerkki 3) menetelminä ovat Kerberos ja LDAP:

```
TEST {

  fi.slite.jaas.auth.module.Krb5LoginModule sufficient
        debug=FALSE;

  fi.slite.jaas.auth.module.LDAPLoginModule sufficient
        java.naming.provider.url="ldap://127.0.0.1:389/o=slite,c=fi"
        java.naming.security.principal="cn=admin,o=slite,c=fi"
        java.naming.security.credentials="secret"
        Attribute="uid"
        startTLS="false";

};
```

Esimerkki 3. JAAS-konfigurointitiedosto

Esimerkkitapauksessa *kirjautumismoduulin* (login module) yhteydessä käytetty `sufficient`-määre tarkoittaa, että onnistuneeseen todentamiseen riittää joko Kerberos- tai LDAP-todentamisen onnistuminen. JAAS on mahdollista konfiguroida myös siten, että kaikkien kirjautumismoduulien on onnistuttava käyttäjien todentamisessa, jotta todentaminen olisi asiakasohjelman näkökulmasta onnistunut. Useimmiten tällaiselle ei kuitenkaan ole tarvetta, vaan riittää, että käyttäjän todentaminen onnistuu yhden kirjautumismoduulin avulla.

Onnistuneen todentamisen seurauksena JAAS toimittaa asiakassovellukselle subjektin (luokan `javax.security.auth.Subject` esiintymä), joka edustaa järjestelmään sisäänkirjautunutta käyttäjää. Subjekti sisältää yhden tai useamman `java.security.Principal`-luokan esiintymän; `Principal`:in avulla kuvataan käyttäjän yksilölliset *identiteetit* (identities) järjestelmässä. Eri sovelluksissa voidaan käyttää eri identiteettiä sen mukaan, mikä on sovelluksen kannalta tarkoituksenmukaisinta; identiteettinä voi toimia esimerkiksi käyttäjän sosiaaliturvatunnus, käyttäjätunnus tai X.500-hakemistopalvelussa käyttäjätietueen yksilöivä nimi (distinguished name, DN).

Pelkkä subjektin välittyminen asiakaskoodille ei vielä johda siihen, että sisäänkirjautunut käyttäjä tulisi sovellusta suoritettaessa pääsynvalvontatoimenpiteiden piiriin. Asiakasohjelman on suoritettava kohteena oleva toiminto sisäänkirjautuneen käyttäjän (eli subjektin) ominaisuudessa, mikä tapahtuu käyttämällä seuraavia `javax.security.auth.Subject`-olion tarjoamia staattisia metodeita:

- `static Object doAs(Subject subject, PrivilegedAction action)`
- `static Object doAs(Subject subject, PrivilegedExceptionAction action)`

Kohdetoiminnon tulee toteuttaa rajapinta `java.security.PrivilegedAction` tai vaihtoehtoisesti rajapinta `java.security.PrivilegedExceptionAction`, jotka määrittelevät ainoastaan yhden `run()`-metodin kohteena olevan toiminnon suorittamiseksi. Näiden kahden rajapinnan erona on se, että `PrivilegedAction`-rajapinnan toteuttava toiminto voi suorituksensa aikana heittää vain tarkistamattoman poikkeuksen, mutta `PrivilegedExceptionAction`-rajapinnan toteuttava toiminto voi heittää myös tarkistetun poikkeuksen.

Kun asiakaskoodi suorittaa kohteena olevan toiminnon `doAs`-metodia kutsumalla, se siis ikään kuin suorittaa toiminnon käyttäjän oikeuksin. Näitä ei tule sotkea varsinaisen Java-prosessin (eli käyttöjärjestelmässä ajettavan prosessin) ajo-oikeuksiin; Java-prosessia ajetaan toki sovelluksen käynnistäneen käyttäjän oikeuksilla, eikä suinkaan sen käyttäjän oikeuksilla, joka JAAS:in avulla sovellukseen kirjautui. Sovelluksen käyttäjällähän ei välttämättä edes ole käyttöjärjestelmätason tunnusta järjestelmään, vaan sovelluksen käyttäjää voi usein edustaa pelkkä yksittäinen tietue relaatiotietokantajärjestelmässä tai vaikkapa LDAP-hakemistopalvelussa. Tämä näkökanta on syytä pitää kirkkaana mielessä silloin, kun ollaan aikeissa käyttää JAAS:in tarjoamia palveluita sovelluksen toimintojen pääsynvalvonnan hoitamiseen.

JAAS siis täydentää Javan tietoturvamallia tuomalla mukaan subjektin käsitteen. JAAS myös laajentaa Javan politiikkatiedostojen syntaksia tuomalla mukaan `principal`-elementin, jonka avulla voimme nyt täydentää alakohdassa 2.3.1 esittelemäämme Javan politiikkatiedostoa seuraavasti (Esimerkki 4):

```
grant codeBase "file:/home/users/sami/myApp",
    principal javax.security.auth.x500.X500Principal "cn=Toni"
{
    permission java.io.FilePermission "/tmp/app.log", "write";
};
```

Esimerkki 4. JAAS-määreen sisältävä politiikkatiedosto

Päivitetty sääntö määrittää, että kaikki hakemistosta `/home/users/sami/myApp` ladatut Java-luokat saavat kirjoittaa tiedostoon `/tmp/app.log`, kunhan kirjoitusoikeutta pyytävä ohjelmakoodi ajetaan `Toni`-nimisen sovelluskäyttäjän

oikeuksilla. `Toni`-käyttäjän tulee olla kirjautunut sovellukseen JAAS:in todentamispalveluiden avulla, ja sovelluksen pitää käyttää todentamisen seurauksena saamaansa `Toni`-käyttäjää edustavaa subjektioliota kutsuessaan kirjoitusoikeutta pyytävää koodia `javax.security.auth.Subject`-luokan `doAs`-metodien avulla.

Pohtiessamme JAAS:in soveltuvuutta web-sovellusten (ja erityisesti monimutkaisempien ryhmätyösovellusten) pääsynvalvontaan on kuitenkin tultava siihen tulokseen, että JAAS:in ilmaisuvoima ei sittenkään riitä. Ensinnäkin, JAAS pakottaa kutsumaan kohteena olevaa komponenttia sangen rajoitetulla tavalla (kutsuttavalla oliolla oltava sellainen `run()`-metodi joka niin sanotusti ”tekee kaiken”), minkä vuoksi esimerkiksi hienostuneemman elinkaarimallin käyttäminen kohdekomponentin toteutuksessa osoittautuu varsin vaikeaksi. Paljon merkittävämpi syy JAAS:in soveltumattomuuteen on kuitenkin se, että JAAS ei tarjoa *instanssitason* käyttöoikeuksia kohteena olevaan informaatioon; voimme JAAS:in avulla kyllä estää käyttäjältä tietyn Java-luokan käytön, mutta emme voi kieltää vain tietyn Java-luokan ajonaikaisen instanssin eli esiintymän käyttöä.

Havainnollistakaamme edellä lausuttua konkreettisen esimerkin avulla. Jos olisimme kiinnostuneita vaikkapa Juuso-nimisen henkilön oikeudesta ajaa autoa, jonka rekisterinumero on ABC-123, JAAS ei tarjoaisi meille mahdollisuutta viitata tähän auton yksilöivään informaatioon. JAAS:in avulla voisimme määrittää, että ”*Juuso saa ajaa autoa*” tai ”*Juuso ei saa ajaa autoa*”, mutta emme voisi sanoa, että ”*Juuso saa ajaa autoa jonka rekisterinumero on ABC-123, mutta ei muita autoja*”. Meillä ei ole mitään tapaa viitata JAAS:in konfiguraatiotiedostosta käsin tietyn Java-luokan ajonaikaiseen instanssiin.

Tilanne olisi erilainen esimerkiksi silloin, jos jokaista rajoitettavaa Java-luokan instanssia edustaisi esimerkiksi tiedostojärjestelmään tallennettu tiedosto. Javan politiikkatiedostoista kertovassa esimerkissähän annoimme eksplisiittisen oikeuden tiedostoon `/tmp/app.log`, ja tällaisia oikeuksiahan voidaan siis myös JAAS:in avulla myöntää. Käytännön sovelluksissa arvoaluemallin pysyvyys toteutetaan kuitenkin useimmiten relaatiotietokantajärjestelmien avulla, eikä ajonaikaisten objektien tilaa tallenneta tavanomaisiin tiedostoihin. Mikään ei tietysti estä meitä tekemästä omaa `java.io.FilePermission`-luokan kaltaista lupatyyppiä (kutsukaamme sitä vaikkapa nimellä `fi.slite.security.DatabaseTuplePermission`) tietokantatietueiden osoittamiseen, mutta käytännön tasolla tämä menisi melkoisen hankalaksi.

Esimerkiksi sääntökone ei näkemykseni mukaan ole ylläpidollisesti ainakaan hankalampi ratkaisu, ja voimme olla suhteellisen vakuuttuneita siitä, että sääntökoneen ilmaisuvoima on varmasti aivan eri luokkaa suhteessa JAAS:iin ja omaan lupatyyppitoteutukseen.

JAAS-palvelulla on kuitenkin oma roolinsa web-sovelluksissa. Monet sovellukset käyttävät JAAS:ista vain sen tarjoamia todentamispalveluita, ja näen tämän aivan hyvänä ratkaisuna. JAAS piilottaa alla käytettävän todentamismekanismin asiakassovellukselta, ja tarjoaa valmiina joitakin suhteellisen työläiksi luokiteltavissa olevia todentamismekanismeja (kuten Kerberos V5), joita asiakassovelluksen laatijan ei tällöin tarvitse omatoimisesti toteuttaa. Ryhmätyösovellusten pääsynvalvontaan en sen käyttöä kuitenkaan suosittelisi.

2.3.3. Web-sovellusten pääsynvalvonta

Edellisessä alakohdassa sivuttiin JAAS:in yhteydessä hivenen jo web-sovellustenkin pääsynvalvontaa. Tarkastellaan seuraavaksi minkälaista ominaisuuksia Java 2 -ympäristö muilta osin tarjoaa web-sovellusten pääsynvalvonnan tarpeisiin, ja pohditaan riittäisivätkö nämä ominaisuudet esimerkiksi ryhmätyösovellusten pääsynvalvonnan toteuttamiseen.

Java 2 -ympäristö itsessään tarjoaa toistaiseksi hyvin suppean tuen web-sovellusten pääsynvalvontaan. Käyttäjän on mahdollista asettaa tietyt web-sovelluksen toiminnot pääsynvalvonnan piiriin *käytönkuvaimen* (deployment descriptor) avulla, ja toisaalta `javax.servlet.http.HttpServletRequest`-rajapinta tarjoaa asiakaskoodille mahdollisuuden selvittää, omaako käyttäjä sellaisen roolin, jolla kohdetoiminnon suorittaminen olisi luvallista. Kunnollista tukea harkinnanvaraiselle pääsynvalvonnalle ei Javan web-sovelluksille ole määritelty.

Käytönkuvaimella (web-sovellusten tiedosto `web.xml`) voidaan määritellä rajoituksia web-sovelluksen toimintojen käyttöön seuraavasti (Esimerkki 5):

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>AdminGUI</web-resource-name>
        <url-pattern>/admin/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>SUPERADMIN</role-name>
    </auth-constraint>
</security-constraint>
```

Esimerkki 5. Web-sovelluksen käytönkuvain (`web.xml`)

Jos sovelluksemme sijaitsisi www-osoitteessa `http://localhost:8080/myApp`, osoitteessa `http://localhost:8080/myApp/admin/adminConsole.do` sijaitseva sovelluksen hallintakonsoli ei olisi käyttäjän käytettävissä ilman hänelle myönnettyä `SUPERADMIN`-roolia.

Jos rajoitteita ei haluta määritellä käytönkuvaimessa, voidaan pääsyä web-sovelluksen eri osiin rajoittaa myös ohjelmallisesti. Seuraavassa esimerkissä (Esimerkki 6) näemme, miten tiedon poistotoiminnossa voidaan kysyä käyttäjän rooli `javax.servlet.http.HttpServletRequest`-rajapinnalta, ja sen perusteella joko sallia tai evätä käyttäjältä tiedon poistaminen:

```
protected void removeItem(HttpServletRequest request,
                          HttpServletResponse response)
    throws SecurityException
{
    String itemId = request.getParameter("itemId");

    if (request.isUserInRole("SUPERADMIN")) {
        PersistenceService.getItemDAO().removeItem(itemId);
    }
    else {
        throw new SecurityException("User does not have proper"
             + " privileges for removing item ‘" + itemId + "’");
    }
}
```

Esimerkki 6. Rooli-informaation hyödyntäminen sovelluskooditasolla

Kuten edellisistä esimerkeistä voi päätellä, on Javan web-sovellusten pääsynvalvonnan tuki varsin suppealla tasolla. Mainittuja ominaisuuksia ei vähänkään isommissa web-sovelluksissa siksi juuri käytetä, vaan niiden sijasta käytetään yleisesti sovelluskohtaisia pääsynvalvontamekanismeja. Joissakin web-sovelluskehyksissä on pääsynvalvontaan liittyvää toiminnallisuutta, mutta niissäkään ei yleisesti ole tarjolla sellaista ilmaisuvoimaa, jota esimerkiksi ryhmätyöohjelmistojen pääsynvalvonta edellyttäisi. Tästä syystä koenkin tärkeäksi tässä tutkielmassa selvittää, voisiko sääntökoneisiin perustuva pääsynvalvontamalli osoittautua toimivaksi pääsynvalvontamenetelmäksi juuri ryhmätyösovellusten yhteydessä.

3. Deklaratiivinen ohjelmointi ja sääntökoneet

Tässä luvussa luodaan yleiskatsaus deklaratiivisen ohjelmointiparadigman erityispiirteisiin, ja pohditaan, mihin tällainen ohjelmointimalli ja sääntökoneet soveltuvat. Luvussa myös esitellään prototyypissä käytettävä avoimen lähdekoodin Drools-sääntökone.

3.1. Deklaratiivisen ohjelmoinnin määrittelyä

Deklaratiivinen ohjelmointitapa määritellään yleensä niin, että sen avulla pyritään vastaamaan kysymykseen ”mitä”, eikä kysymykseen ”miten”, kuten imperatiivisessa ohjelmointimallissa (josta käytetään usein myös synonyyminä termiä proseduraalinen). Imperatiivisissa ohjelmointikielissä (muun muassa C ja Java) ohjelmien voidaan katsoa olevan sarja yksittäisiä lausekkeita, jotka suoritetaan yksi kerrallaan peräkkäisessä järjestyksessä. Korkeatasoisissa imperatiivisissa kielissä on toki olemassa huomattavan paljon monimutkaisempia rakenteita (silmukat, ehdollinen haarautuminen, aliohjelmakutsut, rinnakkaisuus jne.), mutta pohjimmiltaan kyse on aina siitä, että ohjelmat koostuvat yksittäisistä peräkkäin ajettavista lausekkeista.

Deklaratiivisessa ohjelmoinnissa ei yleisesti olla kiinnostuneita siitä miten ohjelma ratkaisee ongelman (eli pääsee alkutilasta lopputilaan), vaan algoritmitason toteutus jätetään järjestelmän vastuulle. Sen sijaan pyritään vastaamaan kysymykseen ”mitä”, eli kuvataan ongelma-alueen tietämys sellaiseen muotoon, jossa se on esimerkiksi sääntökoneen käsiteltävissä. Yleensä tämä tapahtuu kirjaamalla ylös kohdealuetta koskevia *faktoja* (facts) ja *sääntöjä* (rules).

Deklaratiivisen ohjelmoinnin katsotaan yleensä kattavan *funktionaalisen ohjelmoinnin* (functional programming), *logiikkaohjelmoinnin* (logic programming) sekä *rajoiteohjelmoinnin* (constraint programming). On kuitenkin huomioitava, että raja eri ohjelmointiparadigmojen välillä ei ole aina kovin selkeä, vaan paradigmat sisältävät usein toistensa piirteitä. Esimerkiksi proseduraalisten ohjelmointikielten ehtolausekkeilla ja logiikkaohjelmoinnilla on selvästi olemassa joitakin yhteisiä piirteitä.

3.2. Sääntökoneet

Mitä sitten ovat sääntökoneet? Voitaneen sanoa, että sääntökoneet ovat välineitä deklaratiivisen ohjelmointitavan toteuttamiseen. Sääntökoneet jaetaan yleensä *eteenpäin ketjuttaviin* (forward chaining) ja *taaksepäin ketjuttaviin* (backward chaining) sääntökoneisiin. Tämä sääntöjen käsittelytapaa koskeva

ero voi olla tietyissä tilanteissa hyvinkin merkitsevä, joten sääntökoneen käyttöönottoa pohtivan ohjelmistosuunnittelijan on tiedettävä, minkälaisella lähestymistavalla nämä erilaiset sääntökoneet toimivat.

3.2.1. Eteenpäin ketjuttavat sääntökoneet

Eteenpäin ketjuttavat sääntökoneet toimivat siten, että ne etsivät sääntökokoelmasta sääntöjä, joiden *alkuehto* (condition) on sääntökoneen kohdealueesta saaman tietämyksen perusteella totta. Tämän jälkeen sääntökoneet laukaisevat valitun säännön, eli suorittavat säännössä määritetyt *seuraukset* (consequences), jotka muuttavat kohdealueen tilaa.

Eteenpäin ketjuttava sääntökone on oikea valinta silloin, kun olemme kiinnostuneita siitä, minkälainen lopputulos jollakin tarkastelulla tulisi olemaan (vastakohtana sille, että tietäisimme haluamamme lopputuloksen jo valmiiksi, ja etsisimme vain keinoja siihen pääsemiseksi). Esimerkiksi tämän tutkielman yhteydessä syntyneessä esimerkkisovelluksessa olemme kiinnostuneita selvittämään, onko käyttäjällä K oikeus käyttää toimintoa T tai informaatiota I. Näin ollen oikea valinta esimerkkisovelluksen käyttämäksi sääntökoneeksi on eteenpäin ketjuttava sääntökone, kuten esimerkiksi luvussa 3.3 esiteltävä Drools.

3.2.2. Taaksepäin ketjuttavat sääntökoneet

Taaksepäin ketjuttavat sääntökoneet sen sijaan lähtevät liikkeelle etsimällä sääntökokoelmasta sääntöjä, joissa kuvatut seuraukset ovat yhteneväiset halutun lopputuloksen kanssa. Kun tällainen sääntö (käyttäkäämme tästä vaikkapa nimitystä Z) löytyy, sääntökone jatkaa edelleen etsimään edeltävää sääntöä Y, joka laukaistuaan muokkaisi kohdemaailman tilaa siten, että Z:n alkuehto tulee todeksi. Näin jatketaan siihen saakka, kunnes sääntökannasta löytyy riittävä määrä sääntöjä (A->B->….->Y->Z), joiden avulla on mahdollista päästä alkutilasta haluttuun lopputulokseen asti.

Taaksepäin ketjuttavat sääntökoneet ovat siis *päämäärävetoisia* (goal-driven), eli sääntökoneen tehtäväksi tulee etsiä keinot haluttuun päämäärään päätymiseksi, kun alkutila ja haluttu lopputulos ovat tiedossa. *Asiantuntijajärjestelmät* (expert systems) ovat useimmiten päämäärävetoisia, eli niissä haluttu lopputulos (esimerkiksi potilaan paraneminen sairaudesta) on tarkastelun lähtökohtana, ja asiantuntijajärjestelmän tehtävänä on löytää tämän mahdollistavat toimenpiteet (esimerkiksi potilaalle parhaiten soveltuvan lääkeyhdistelmän valinta), kun alkutila (sairastunut potilas) on tiedossa. Prolog -logiikkaohjelmointikieli on erinomainen esimerkki taaksepäin ketjuttavasta sääntökoneesta.

3.2.3. Milloin käyttää sääntökonetta?

Monet ohjelmistokehittäjät ovat jossain vaiheessa uraansa ajautuneet tilanteeseen, jossa he huomaavat ohjelmansa koostuvan valtavasta määrästä `if/else`-ehtologiikkaa. Mikäli ohjelmat sisältävät lisäksi runsaasti esimerkiksi `System.out.println()`-lausekkeita, joiden avulla yritetään pysyä selvillä ohjelman suorituksen etenemisestä, pitäisi viimeistään tämän tilanteen herättää ohjelmistokehittäjä pohtimaan, voisiko tarkasteltavana olevan ongelman ratkaista jollakin toisella menetelmällä. Jos tällaista ohjelmakoodia ei refaktoroinnin avulla onnistuta pilkkomaan pieniin ja ymmärrettäviin osiin, voisi olla hyvä ratkaisu kokeilla ongelman ratkaisemista sääntökoneen avulla.

Sääntökonetta kannattaa ehdottomasti käyttää silloin, jos sovelluksen liiketoimintalogiikka muuttuu usein. Jos liiketoimintalogiikka on upotettuna ohjelmakoodiin, vaativat muutokset uuden ohjelmistoversion rakentamista ja uutta käyttöönottoprosessia. Vähänkin isompien ohjelmistojen tapauksessa käyttöönottoprosessi on työläs ja aikaa vievä vaihe, koska uusi sovellusversio on testattava kauttaaltaan merkittävimpien virheiden löytämiseksi. Toki sääntökoneen tapauksessakin muuttuneet liiketoimintasäännöt on testattava huolellisesti, mutta tämä on kuitenkin huomattavasti pienempi vaiva kuin koko ohjelmiston testaaminen. On kuitenkin syytä huomioida se, että sääntökone ei suinkaan ole ainoa tapa tuoda dynaamisuutta liiketoimintalogiikan käsittelyyn; jos liiketoimintalogiikka koostuu enemmänkin sarjallisesti tietyssä järjestyksessä suoritettavista yksittäisistä operaatioista kuin irrallisista ja toisistaan riippumattomista säännöistä, voidaan dynaamisuutta tuoda sovellukseen esimerkiksi skriptikielien (vaikkapa BeanShell) avulla.

Sääntökoneen käyttäminen on perusteltua myös silloin, jos sovelluksen liiketoimintasäännöt laaditaan pääosin liiketoiminta-analyytikkojen toimesta, eikä ohjelmistokehittäjien itsensä toimesta. Analyytikoilla ei useinkaan ole kovin vahvaa kokemusta ohjelmistoarkkitehtuureista, joten heidän valmiutensa ymmärtää ohjelmistojen teknistä toimintaa on usein heikohko. Mikäli liiketoimintasäännöt onnistutaan erottamaan muusta sovelluslogiikasta ja mahdollisesti vielä esittämään mahdollisimman luonnollisen kaltaisella kielellä, on analyytikkojen helpompaa laatia liiketoimintasäännöt ja myös verifioida sääntöjen toiminnan oikeellisuus.

Myös silloin on syytä pohtia sääntökoneen käyttöönottoa, kun toteutettava sovellus on SaaS-mallin [Wikipedia, 2008b] mukainen tietojärjestelmä, jossa usea eri asiakas käyttää samaa sovellusinstanssia siten, että asiakkaiden data on

tietojärjestelmässä virtuaalisesti eroteltua (tällaisesta järjestelmästä voidaan käyttää myös nimitystä *monivuokralaissovellus* (multi-tenant application) tai moniorganisaatiotuen sisältävä sovellus). Vaikka eri asiakkaat olisivatkin käyttöönottovaiheessa kohtalaisen tyytyväisiä sovelluksen perustoimintoihin, asiakkaiden erilaiset organisaatiomallit johtavat käytön jatkuessa yleensä siihen, että asiakkaat haluavat käyttää joitakin järjestelmän piirteitä hieman toisistaan eroavalla tavalla. Iso ohjelmistotoimittaja voi piiloutua suuruutensa taakse ja luottaa volyymiin; yhden asiakkaan menettäminen ei ole ongelma, jos asiakaspotentiaali on riittävän iso. Sen sijaan pienten ohjelmistotoimittajien on kilpailtava joustavuudella ja palveluhalukkuudella, eivätkä ne voi kylmästi jättää asiakkaiden toiveita huomiotta. Sääntökoneen ja joustavan liiketoimintalogiikan avulla sama sovellusinstanssi voidaan tarvittaessa säätää toimimaan tietyissä tilanteissa hieman eri tavalla asiakaskohtaisesti, eivätkä kustannukset nouse kohtuuttomiksi.

Sääntökoneet ovat myös erittäin tehokkaita silloin, kun liiketoimintalogiikkaa on runsaasti. Erityisesti RETE-algoritmiin ja sen johdannaisiin perustuvien sääntökoneiden suorituskyky on teoreettisesti lähes riippumaton sääntöjen lukumäärästä [Wikipedia, 2008c], jolloin sääntökokoelman koon merkittäväkään kasvaminen ei yleensä johda järjestelmän suorituskykyongelmiin. Tehokkuushyödyt tulevat erityisen hyvin esiin silloin, kun kohdemaailman tila ei juurikaan muutu; sääntökoneissa on yleensä välimuistirakenteita, jotka säilyttävät tietoa aiemmin tehdyistä päätöksistä, ja tätä informaatiota voidaan käyttää hyväksi seuraavilla tarkastelukierroksilla.

Eräs sääntökoneiden hyöty on myös tietynlainen itsedokumentoituvuus: ideaalitilanteessa luonnollisen kielen kaltaisella syntaksilla esitetyt säännöt ovat niin selkeitä ja luettavia, että ne ovat lähes suoraan vietävissä osaksi järjestelmän teknistä dokumentaatiota. On kuitenkin huomioitava, että sääntöjä voidaan esittää hyvinkin erityyppisillä kielillä, joten mikään itsestäänselvyys tällainen itsedokumentoituvuusominaisuus ei suinkaan ole.

Sääntökoneet sisältävät usein myös toimintoja, joiden avulla sääntökone antaa perusteluja tekemistään päätöksistä. Tällainen informaatio on kullanarvoista ohjelmistokehittäjille, mutta se saattaa (sopivasti suodatettuna ja muotoiltuna) olla hyvin arvokasta myös järjestelmän käyttäjille. Esimerkiksi aiemmin mainitussa asiantuntijajärjestelmän lääkevalintaesimerkissä järjestelmä voisi tarvittaessa antaa perustelut sille, miksi juuri tiettyä lääkeyhdistelmää ehdotettiin potilaalle. Näin potilas voisi vakuuttua siitä, että hänelle myönnetyt

lääkkeet ovat juuri hänen tapauksessaan parhaat mahdolliset, eikä lääkepäätös ole syntynyt esimerkiksi siitä syystä, että lääkäri sattuu olemaan jonkin tietyn lääkevalmistajan merkittävä osakkeenomistaja. Kaikenlainen päätösten läpinäkyvyys on luottamuksen synnyttämisessä aina avainasemassa, mutta toisaalta on myös varottava sitä, ettei tämä syö prosessissa mukana olevien ammattilaisten omaa auktoriteettia. Potilashan voisi pahimmassa tapauksessa kokea lääkärin jopa jossain määrin tarpeettomaksi, jos asiantuntijajärjestelmän antama raportti olisi tarpeettomankin tasokas, ja potilas kokisi lääkärin toimivan vain pelkkänä asiantuntijajärjestelmän antamien tulosten esilukijana.

Lopuksi kannattanee vielä tähdentää sitä itsestään selvältä tuntuvaa seikkaa, että mikäli suunnitteilla oleva järjestelmä vaikuttaisi ilmiselvästi olevan niin sanottu asiantuntijajärjestelmä, kannattaa toteutuksessa ilman muuta tukeutua deklaratiivisiin ohjelmointimenetelmiin. Logiikkaohjelmointi ja sääntökoneet ovat näissä yhteyksissä koeteltuja ja hyväksi havaittuja ratkaisuja, joten niiden ohittamiseen vaaditaan melkoisen painavat perusteet.

3.3. Drools-sääntökone

Ennen tutkielman yhteydessä rakennettavan esimerkkisovelluksen esittelyä on tarpeen vielä tutustua esimerkkisovelluksessa käytettävään Drools-sääntökonetoteutukseen. Se on Charles S. Forgyn kehittämään RETE-algoritmiin perustuva eteenpäin ketjuttava sääntökone, ja saavuttanut viime vuosina suurta suosiota erityisesti Javalla toteutetuissa tietojärjestelmissä. Droolsista on olemassa myös .NET-ympäristöön sovitettu versio, mutta tutkielmassa käytämme Java-pohjaista toteutusta.

3.3.1. Historia

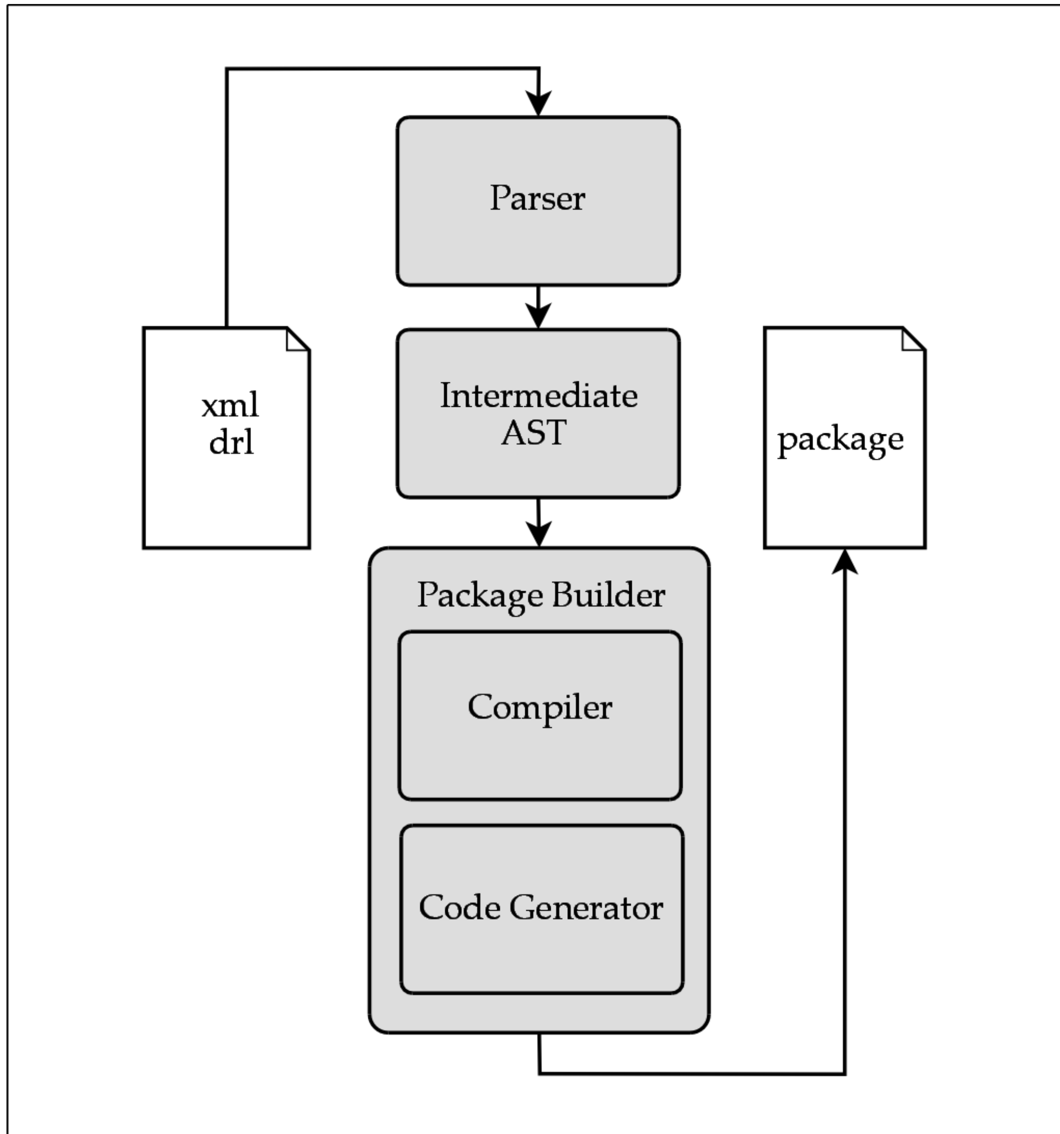
Bob McWhirter käynnisti Drools-projektin vuonna 2001 SourceForgessa. Alkuperäinen versio oli niin sanottuun brute force -menetelmään perustuva lineaarinen toteutus, joka törmäsi pian niin suuriin suorituskykyongelmiin, ettei versiota 1.0 ikinä julkaistu. Lineaarisuus tarkoittaa tässä yhteydessä sitä, että sääntökone evaluoi kaikki järjestelmästä löytyvät säännöt järjestyksessä läpi, eikä ota huomioon sitä, että aktivoituessaan jokin sääntö voi sulkea pois (eli eliminoida) suurenkin joukon muita sääntöjä, jolloin näiden sääntöjen evaluointi ei enää ole tarpeen. Valitun ratkaisutavan osoittauduttua ongelmalliseksi versiota 2.0 lähdettiinkin kehittämään löyhästi RETE-algoritmiin pohjautuen, samalla projekti siirrettiin Codehausin alaisuuteen ja sen vetäjäksi vaihtui Mark Proctor.

Suosion kasvamisen myötä projektille peräänkuulutettiin kaupallista tukea, ja vuonna 2005 JBoss otti Droolsin osaksi sen tuoteportfoliota. Lähtiessään tuotteistamaan Droolsia JBoss nimesi sen uudelleen nimellä JBoss Rules, mutta myös vanha nimi jäi sitkeästi elämään kehitysyhteisön keskuudessa, ja erityisesti juuri varsinaisesta sääntökoneen ytimestä puhuttaessa Drools-nimeä käytetään edelleen yleisesti. JBoss Rules on kuitenkin virallinen tuotenimi, jota käytetään erityisesti silloin, kun puhutaan koko tuotepaletista (sääntökone, siihen kehitetyt hallintatyökalut ja tukitoiminnot). Tässä tutkielmassa tullaan käyttämään yksinomaan Drools-nimeä tästä kohdasta eteenpäin lähinnä siitä syystä, että koen tekstin pysyvän näin luettavampana kuin pidempää JBoss Rules -nimeä käytettäessä.

JBossin alaisuuteen siirtymisen jälkeen julkaistiin versio 3.0, jossa oli kehittäjien mukaan nyt täysimittainen RETE-algoritmin toteutus. Sittemmin kehitys on edennyt nelossarjaan, ja uusin julkaistu versio on 4.0.4 (julkaistu 31.3.2008). Tämän tutkielman yhteydessä syntyneessä esimerkkisovelluksessa käytetään tätä versiota.

3.3.2. Yleiskuvaus

Järjestelmän operoinnin kannalta Drools jakautuu kahteen osioon: sääntökoneen ja sääntökokoelmien hallintaan (authoring) ja sääntöjen suoritukseen (runtime). Hallintatoimintojen avulla DRL- tai XML-tiedostojen avulla esitettävät säännöt jäsennetään, jonka jälkeen niistä rakennetaan ajokelpoisia paketteja seuraavan kuvan (Kuva 1) havainnollistamalla tavalla:



Kuva 1. Ajokelpoisten sääntöpakettien muodostaminen Droolsissa

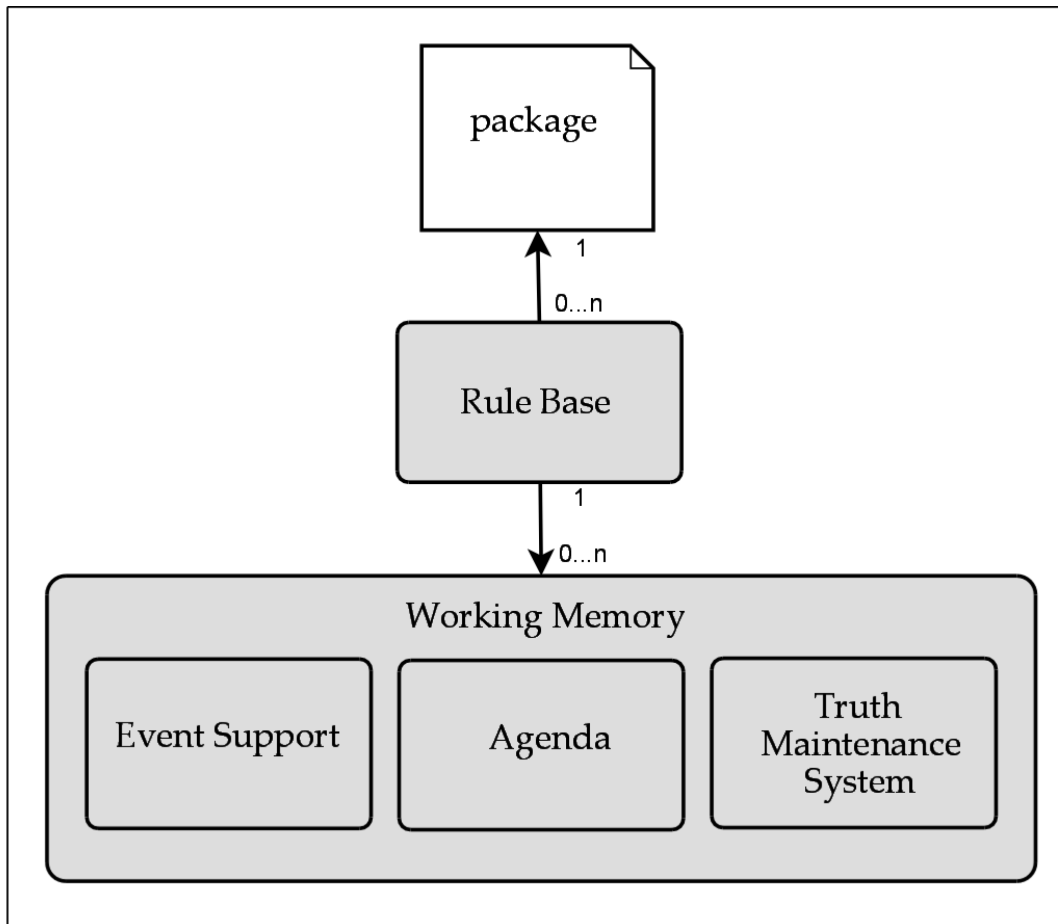
Prosessin tuloksena syntyvät paketit koostuvat sarjallistuvista (serializable) Java-luokista. Paketit ovat itseriittoisia sikäli, että sääntöjä suoritettaessa ei tarvita alkuperäisiä sääntötiedostoja (DRL) tai jäsentäjän tuottamia välitietorakenteita (AST), vaan sääntöpaketti itsessään riittää. Java-ohjelmoinnin parissa työskentelevät huomaavat varmasti prosessin yhtäläisyydet tavanomaiseen Java-ohjelmistokehitykseen; molemmissa tapauksissa lähdekoodista tuotetaan jäsentäjän ja kääntäjän avulla binäärimuotoisia rakenteita, jotka ovat suorituskelpoisia ilman alkuperäisiä lähdekooditiedostoja. Seuraava listaus (Esimerkki 7) esittää, miten sääntöpaketit muodostetaan lähdekooditasolla:

```
PackageBuilder builder = new PackageBuilder();
builder.addPackageFromDrl(new InputStreamReader(
    getClass().getResourceAsStream("package1.drl")));
Package pkg = builder.getPackage();
PackageBuilderErrors errors = builder.getErrors();

if (errors != null && errors.getErrors().length > 0) {
    // There were some errors in the rules, handle them here
}
```

Esimerkki 7. Sääntöpaketin muodostaminen lähdekooditasolla

Jos sääntöpaketin muodostaminen onnistuu virheettömästi, voidaan sääntöpaketin sisältämiä sääntöjä suorittaa. Sääntöjä suoritettaessa keskeinen komponentti on sääntökanta (RuleBase), johon voidaan kytkeä yksi tai useampi sääntöpaketti oheisen kuvan (Kuva 2) havainnollistamalla tavalla:



Kuva 2. Sääntökoneen suoritusaikainen arkkitehtuuri

Kun sääntökonetta pyydetään evaluoimaan sääntökannasta löytyviä sääntöjä, sääntökone alustaa yhden tai useamman työmuistin (Working Memory)

työskentelyn apuvälineeksi. Työmuisti sisältää päättelyn kohteena olevat faktat sekä työjärjestyksen (Agenda), jonka avulla sääntöjen valikointia ja aktivointia säädellään. Työmuisti sisältää myös toiminnallisuutta tapahtumankäsittelyn tueksi, minkä avulla asiakassovellus voi halutessaan saada informaatiota muun muassa käsittelyn etenemisestä sääntökoneessa.

Työmuisti sisältää myös totuuden ylläpitojärjestelmän (Truth Maintenance System). Tämä alijärjestelmä on tärkeä siitä syystä, että aktivoituvat säännöt voivat muokata työmuistissa olevien faktojen tilaa, jolloin sääntökoneen on sekä pidettävä kirjaa sääntöjen aktivoinnin tuloksista että tiedettävä, mikä faktojen alkuperäinen tila oli. Ilman tällaista mekanismia ei olisi takeita siitä, että sääntöjen evaluoinnin seurauksena saadut tulokset ovat luotettavia ja johdonmukaisia. Seuraavassa listauksessa (Esimerkki 8) esitetään, miten sääntökanta luodaan ja miten sen sisältämät säännöt suoritetaan:

```
RuleBase ruleBase = RuleBaseFactory.newRuleBase();
ruleBase.addPackage(pkg);

StatefulSession session = ruleBase.newStatefulSession();
session.insert(new Person("Rauno Mäkynen", "11.02.1943"));
session.fireAllRules();
session.dispose();
```

Esimerkki 8. Sääntöjen suorittaminen lähdekooditasolla

RuleBase on Droolsissa se komponentti, jonka kautta päästään suorittamaan varsinaisia sääntöjä. Se on varsin raskas komponentti, ja alustetaan tyypillisesti vain kerran sovelluksen käynnistymisvaiheessa. Se on myös säieturvallinen (thread-safe) komponentti, eli useampi säie voi käyttää samaa `RuleBase`-luokan esiintymää yhtäaikaisesti ilman pelkoa siitä, että samanaikainen käyttö sotkisi komponentin sisäistä tilaa. Sen sijaan `RuleBase`-luokalta pyydettävä istunto (session) ei ole säieturvallinen komponentti, joten sitä ei pidä jakaa eri säikeiden välillä. Istunto voi olla tilallinen tai tilaton, ja istunnon avulla asiakaskoodi voi muun muassa syöttää faktat työmuistiin sekä käynnistää sääntöjen evaluoinnin.

Esimerkistä nähdään, että sääntökoneita käytettäessä ajatuksena ei ole suorittaa yksittäisiä sääntökannan sääntöjä, vaan antaa sääntökoneen päätellä, mitkä säännöt pitää laukaista kohdemaailman tila huomioiden. Esimerkiksi seuraava sääntökannan sisältämä sääntö (Esimerkki 9) aktivoituisi testihenkilömme kohdalla (edellyttäen että `Person`-luokalla on iän palauttava `getAge()`-metodi sekä nimen kertova `getName()`-metodi):

```
rule "Person who is at least 65 years old is a senior citizen"
    when
        $p : Person (age >= 65)
    then
        System.out.println(p.getName() + " is a senior citizen.");
end
```

Esimerkki 9. Sääntökannan sisältämä esimerkkisääntö

Droolsissa on mahdollista rajoittaa jonkin verran sitä, mitkä säännöt tietyissä tilanteissa aktivoituvat. Seuraavassa aliluvussa kerrotaan sääntöjen kuvaustavoista tarkemmin, ja sivutaan hivenen myös tätä aihetta. Ennen sitä on kuitenkin syytä vielä tarkastella pintapuolisesti sitä, miten yllä mainittu toiminnallisuus jakautuu Droolsissa eri moduuleihin, ja mitä moduuleita sääntökoneen käyttäjä tarvitsee Droolsia käyttäessään.

Drools on jaettu neljään eri moduuliin, jotka ovat:

- `drools-core.jar`
- `drools-compiler.jar`
- `drools-decisiontables.jar`
- `drools-jsr94.jar`

Jos sovellus hyödyntää valmiiksi käännettyjä sääntöpaketteja (eli vain suorittaa sääntöjä), tarvitaan vain moduuli `drools-core.jar`. Jos sovelluksen tulee lisäksi kyetä jäsentämään ja kääntämään sääntöpaketteja tekstimuotoisista sääntötiedostoista, tarvitaan myös moduuli `drools-compiler.jar`. Moduuli `drools-decisiontables.jar` tarvitaan, mikäli säännöt laaditaan *päätöstaulujen* (decision tables) avulla (emme kuvaa päätöstauluja tarkemmin tässä yhteydessä, lisätietoa aiheesta voi etsiä Drools-käyttöoppaan [Proctor *et al.*, 2008] luvusta 4).

Moduuli `drools-jsr94.jar` mahdollistaa Droolsin käyttämisen JSR-94 -spesifikaation mukaisesti. JSR-94 (Java Rule Engine API) on sääntökoneiden käyttörajapinnan määrittävä spesifikaatio, joka määrittää sen, millä tavoin sääntökoneita käytetään sääntökoneriippumattomalla tavalla. Se vastaa käsitteellisesti JDBC-rajapintaa siinä mielessä, että JDBC esittelee järjestelmätoimittajasta riippumattoman tavan relaatiotietokantojen käyttämiseen Java-sovelluksista käsin, mutta ei määrittele sitä, minkälaisia tietokantakyselyitä kohdejärjestelmässä on mahdollista suorittaa. Vastaavasti

JSR-94 siis määrittelee tavan, jolla sääntökonetta voidaan kutsua Java-ohjelmista käsin, mutta ei määrittelee tapaa, jolla säännöt kohdejärjestelmässä esitetään.

Droolsin tekijät eivät erityisen vahvasti suosittele JSR-94 -spesifikaation mukaista käyttöä. He näkevät asian siten, että standardoitu käyttötapa jättää niin suuren osan järjestelmän ominaisuuksista huomiotta, että käyttötavan puutteet ovat selvästi etuja merkittävämmät. Oma näkemykseni on melko yhdenmukainen tämän kanssa, ja kun tarkastelemme vaikkapa aiemmin esiteltyjä esimerkkilistauksia (Esimerkki 7 ja Esimerkki 8), huomaamme miten yksinkertaista Drools-sääntökoneen käyttäminen lähdekooditasolla loppujen lopuksi on. Jos siis päättäisimme vaihtaa käytettävän sääntökoneen joksikin toiseksi, olisi työmäärä lähdekoodin muuttamisen osalta varsin vähäinen. Näin ollen en katso, että JSR-94 -spesifikaation mukainen lähdekooditason käyttö olisi sovellusten ylläpidettävyyden kannalta olennainen tekijä. Tilanne olisi toinen mikäli JSR-94 määrittelisi myös säännöissä käytettävän syntaksin, mutta tätä se ei tee, koska sääntöjen esittämistavoista ei vallitse kovinkaan suurta yhteisymmärrystä eri järjestelmätoimittajien välillä. Seuraavassa aliluvussa tarkastelemme, minkälaisen syntaksin Drools-sääntökone on valinnut sääntöjen esitystavaksi.

3.3.3. Drools-sääntöjen esitystavoista

Säännöt voidaan Droolsissa esittää joko DRL- tai XML-tiedostojen avulla. DRL-tiedosto on suositeltava esitystapa sääntöjen kuvaamiseen, ja sen rakenne on seuraavankaltainen (Esimerkki 10):

```
package <package-name>

imports

expander

globals

functions

queries

rules
```

Esimerkki 10. DRL-tiedoston rakenne

Vaihtoehtoinen XML-formaatti on tarkoitettu lähinnä tilanteisiin, jossa sääntöjä generoidaan ohjelmallisesti. Se on hyvä vaihtoehto myös tilanteissa, joissa osa järjestelmään tuotavista säännöistä on jo valmiiksi jonkinlaisessa XML-

formaatissa. Tällöin säännöt voidaan suhteellisen pienellä vaivalla muuntaa Droolsin omaan XML-formaattiin esimerkiksi XSLT-transformaation avulla.

Tämän tutkielman yhteydessä emme perehdy tarkemmin XML-formaattiin, mutta käymme läpi DRL-esitystavan. DRL-tiedostoissa elementtien keskinäisellä järjestyksellä ei ole merkitystä muilta osin, mutta package-elementin tulee olla ensimmäisenä silloin, kun se on määritelty. Mitään elementtiä ei ole pakko määritellä, vaan kaikki ovat jäsentäjän kannalta valinnaisia elementtejä.

Package-elementin avulla sääntökokonaisuus (eli pakkaus) voidaan nimetä yksilöllisesti. Jos sovellus käyttää vain yhtä pakkausta, ei nimeämisellä ole juurikaan merkitystä. Yhteen pakkaukseen kootaan tyypillisesti samaa arvoaluetta koskevat säännöt, ja pakkauksen nimi toimii XML-tyylisenä *nimiavaruutena* (namespace). Pakkauksen sisällä määritetyt säännöt eivät normaaliolosuhteissa voi viitata muissa pakkauksissa esiteltyihin sääntöihin.

Imports-elementti toimii kuten Javan lähdekooditiedostojen import-lausekkeet. Niiden avulla määritetään ne Java-luokat, joita säännöissä voidaan käyttää. Säännöissä suoritetaan siis tyypillisesti Java-koodia, joskin myös muiden kielten – esimerkkinä MVEL – käyttäminen on mahdollista. Kuten Javan lähdekooditiedostojen tapauksessakin, Javan `java.lang`-pakkauksesta käytettyjä luokkia ei tarvitse erikseen importoida. Expander-elementin avulla voidaan puolestaan lukea *arvoaluekohtaisen kielen* (domain-specific language, DSL) määritykset ulkoisesta tiedostosta; palaamme tähän aiheeseen tarkemmin perussääntökielen kuvauksen jälkeen.

Globals-elementin avulla voidaan määritellä globaaleja muuttujia, jotka ovat pakkauksen kaikkien sääntöjen käytettävissä. Nämä ovat asiakasohjelman sääntökoneelle välittämiä objekteja, joita säännöt voivat käyttää muun muassa tiedonhankinnan apuvälineenä (voidaan esimerkiksi välittää sääntökoneelle Hibernaten sessio, jota hyödyntämällä sääntökone voi noutaa tietokannasta informaatiota) tai päättelyiden tulosten tallentamiseen ja välittämiseen asiakasohjelmalle. Eräs tyypillinen käyttötapa globaaleille muuttujille on *takaisinkutsujen* (callbacks) tekeminen sääntökoneesta asiakassovellukseen päin.

Functions-elementti mahdollistaa semanttisten funktioiden kirjoittamisen sääntötiedostojen yhteyteen. Funktiot sisältävät tyypillisesti ohjelmakoodia,

joka Java-sovelluksissa kirjoitettaisiin niin sanottujen *apuluokkien* (helper classes) avulla. Funktioiden vieminen Java-koodista sääntötiedostoon tarjoaa kaksi etua: ensinnäkin, sääntöihin liittyvä funktioina esitettävissä oleva logiikka voidaan pitää sääntökuvausten välittömässä yhteydessä, ja toisaalta mekanismi tarjoaa mahdollisuuden muuttaa funktioiden toimintaa ilman, että asiakasohjelmaa tarvitsee tämän johdosta päivittää.

Queries-elementti tarjoaa mahdollisuuden tehdä kyselyjä kohdemaailman faktoihin liittyen. Ne sivuavat varsinaisia sääntöjä siten, että kyselyissä määritetään ehdot kohdemaailman sisältämien faktojen kohdistamiseen, mutta ei toimenpiteitä, joita kohdistuminen aiheuttaa. Kyselyt ovat käteviä silloin, kun haluamme vain valikoida kohdemaailmasta informaatiota tiettyjen kriteerien mukaan asiakasohjelman prosessoitavaksi ilman, että tulosjoukolle haluttaisiin suorittaa mitään toimenpiteitä (eli seurauksia) itse sääntökoneen toimesta.

Rules-elementti sisältää varsinaiset sääntölausekkeet. Esimerkki 11 kuvaa sääntölausekkeen rakenteen karkealla tasolla:

```
rule "<name>"
    <attribute>*
when
    <conditional element>*
then
    <action>*
end
```

Esimerkki 11. DRL-tiedoston sisältämän yksittäisen säännön rakenne

Attribute-elementtien avulla voidaan kontrolloida säännön aktivoitumista tai säännön suorittamista erilaisten direktiivien avulla. Ne ovat siis eräänlaisia sääntökoneelle annettavia ohjeita; esimerkiksi `date-effective`- ja `date-expires`-attribuuttien avulla sääntö voidaan määrittää aktivoitumaan vain tiettynä ajanjaksona. Attribuuttien tarkempi läpikäynti ei tämän tutkielman puitteissa ole erityisen tarpeellista; asiasta kiinnostunut lukija voi etsiä lisätietoa aiheesta Drools-käyttöoppaan [Proctor *et al.*, 2008] luvusta 6.

When-elementti sisältää ehtolausekkeet, joiden perusteella päätellään, aktivoituuko sääntö vai ei. Elementti voi olla tyhjä, jolloin sääntö aktivoituu

aina, kohdemaailman faktoista riippumatta. Useimmiten elementti kuitenkin sisältää vähintään yhden ehtolausekkeen. When-elementin sisältöä kutsutaan myös säännön *vasemmaksi puoleksi* (left-hand side, LHS).

Then-elementti, eli säännön *oikea puoli* (right-hand side, RHS), sisältää toimenpiteet, jotka suoritetaan mikäli sääntö *laukeaa* (fires). Toimenpiteet voivat muuttaa kohdealueen faktoja, tai vaikkapa kutsua takaisinkutsun avulla jotakin asiakasohjelmiston komponenttia. Jos toimenpiteet muuttavat kohdealueen faktoja, on sääntöjen laatijan päätettävä, pitääkö näistä muutoksista informoida sääntökonetta prosessoinnin kuluessa. Faktojen muuttaminen prosessoinnin aikana voi nimittäin johtaa sääntökoneen päättymättömään *rekursioon* (recursion); tämä voidaan kuitenkin ehkäistä `no-loop`-attribuutin avulla, jolloin sääntökone ei enää suorita jo aiemmin suorittamiaan toimenpiteitä uudestaan.

Tarkastelkaamme edellä kerrottua esimerkkilistauksen (Esimerkki 12) avulla:

```
package org.drools.examples

import org.drools.examples.HonestPoliticianExample.Politician;
import org.drools.examples.HonestPoliticianExample.Hope;

rule "We have an honest Politician"
    salience 10
    when
        exists( Politician( honest == true ) )
    then
        insertLogical( new Hope() );
end

rule "Hope Lives"
    salience 10
    when
        exists( Hope() )
    then
        System.out.println("Hurrah!!! Democracy Lives");
end

rule "Hope is Dead"
    when
        not( Hope() )
    then
        System.out.println( "We are all Doomed!!!”
           + " Democracy is Dead" );
end

rule "Corrupt the Honest"
    when
        politician : Politician( honest == true )
        exists( Hope() )
    then
        System.out.println( "I'm an evil corporation and I have"
           + " corrupted " + politician.getName() );
        politician.setHonest( false );
        update( politician );
end
```

Esimerkki 12. DRL-esimerkkitiedosto (`HonestPolitician.drl`)

Tarkastelemalla esimerkiksi `Corrupt the Honest` -säännön `when`-elementtiä saatamme tulla siihen tulokseen, että sääntöjen esitystapa tuntuu hivenen sekavalta (esimerkissä olevat säännöt ovat tosin sieltä yksinkertaisimmasta päästä). Sääntökoneiden käyttömahdollisuuksia pohtivassa alakohdassa esitin sellaisen näkemyksen, että mahdollisimman paljon luonnollista kieltä muistuttavat säännöt tarjoaisivat liiketoiminta-analyytikoille mahdollisuuden mallintaa liiketoimintasääntöjä ilman ohjelmistokehittäjien myötävaikutusta. Esimerkkisääntömme eivät tunnu luonnollisen kielen kaltaisilta – olenko siis esittänyt liian optimistisia näkemyksiä sääntökoneiden käyttöominaisuuksista?

Avuksi ongelmaan rientävät jo aiemmin mainitsemani arvoaluekohtaiset kielet (domain-specific languages, DSL). Arvoaluekohtaiset kielet mahdollistavat luonnollisen kaltaisen kielen käyttämisen säännöissä, ja niiden avulla säännöissä käytettävän syntaksin monimutkaisuutta pystytään huomattavasti peittämään. Esimerkki 13 havainnollistaa, miten edellä kuvattu esimerkki esitettäisiin DSL:n avulla:

```
package org.drools.examples

import org.drools.examples.HonestPoliticianExample.Politician;
import org.drools.examples.HonestPoliticianExample.Hope;

expander HonestPolitician.dsl

rule "We have an honest Politician"
    salience 10
    when
        there is an honest politician
    then
        there is hope
end

rule "Hope Lives"
    salience 10
    when
        there is hope
    then
        tell the world about it
end

rule "Hope is Dead"
    when
        there is no hope
    then
        tell that we are doomed
end

rule "Corrupt the Honest"
    when
        there is a specific honest politician
        there is hope
```

```
    then
        corrupt this politician and brag about it
end
```

Esimerkki 13. DRL-sääntöjen korvaaminen DSL-säännöillä

DRL-tiedosto viittaa nyt DSL-tiedostoon, jonka avulla voimme esittää muunnokset luonnollisen kaltaisen kielen ja Droolsin oman syntaksin välillä seuraavan esimerkin (Esimerkki 14) mukaisesti. Säännöt kuvataan `[condition]` ja `[consequence]`-elementtien avulla siten, että tiedostossa esiintyy yksi rivi per muunnos. Rivien pituuden vuoksi ne joudutaan esimerkkilistauksessa rivittämään, minkä vuoksi olen esimerkkilistaukseen lisännyt rivinumerot kutakin loogista (tiedostossa olevaa) riviä kohden.

```
1: [condition][] there is an honest politician = exists( Politician(
       honest == true ) )

2: [condition][] there is hope = exists( Hope() )

3: [condition][] there is no hope = not( Hope() )

4: [condition][] there is a specific honest politician = politician :
       Politician( honest == true )

5: [consequence][] there is hope = insertLogical( new Hope() );

6: [consequence][] tell the world about it =
       System.out.println("Hurrah!!! Democracy Lives");

7: [consequence][] tell that we are doomed = System.out.println("
       We are all Doomed!!! Democracy is Dead");

8: [consequence][] corrupt this politician and brag about it =
       System.out.println( "I'm an evil corporation and I have
       Corrupted " + politician.getName() ); politician.setHonest(
       false ); update( politician );
```

Esimerkki 14. DSL-tiedoston rakenne

Valmiin DRL-muotoisen tiedoston muuntaminen DSL-muotoisia sääntöjä sisältäväksi on siis varsin suoraviivainen toimenpide. DRL-tiedosto linkitetään DSL-tiedostoon `expander`-elementin avulla, ja DRL-tiedoston sisältämät lausekkeet ulkoistetaan osaksi DSL-tiedostoa ja korvataan käyttäjän laatimilla luonnollisen kielen kaltaisilla lausekkeilla. DSL on täysin käännösaikainen konstruktio, eli sitä hyödynnetään vain sääntöjä jäsennettäessä. Droolsin ajonaikaisessa toiminnassa DSL-tiedostoilla ei ole mitään roolia.

Kuten esimerkkitapauksesta nähdään, arvoaluekohtaisten kielten avulla pääsemme huomattavasti lähemmäksi luonnollisella kielellä määritettyjä

sääntöjä. Näkisinkin, että arvoaluekohtaisen kielen määrittäminen on kannattavaa lähestulkoon aina, silloinkin kun säännöt laaditaan ohjelmistokehittäjän itsensä eikä liiketoiminta-analyytikon toimesta. Luonnollisen kaltaisella kielellä esitetyt säännöt ovat helpompia laatia, mutta erityisesti arvoaluekohtainen kieli helpottaa sääntöjen logiikan seuraamista ja niiden toiminnan verifiointia. Tutkielman yhteydessä toteutetussa esimerkkisovelluksessa säännöt esitetään luonnollisen kielen kaltaisella esitystavalla.

4. Esimerkkisovellus: web-kalenterin pääsynvalvonta

Teoreettisen tarkastelun perusteella sääntöpohjainen pääsynvalvonta vaikuttaisi soveltuvan varsin mainiosti web-sovellusten pääsynvalvonnan toteuttamiseen. Varmistuaksemme tästä on kuitenkin tarpeen kokeilla lähestymistavan toimivuutta myös käytännössä, minkä teemme yksinkertaisen esimerkkisovelluksen avulla. Esimerkkisovellus esitellään ehyenä kokonaisuutena, ja samalla pohditaan, miten sen sisältämä toiminnallisuus voitaisiin lohkoa sovelluskohtaiseen ja ohjelmistokehyskohtaiseen osioon siten, että pääsynvalvontaan liittyvän toiminnallisuuden uudelleenkäyttö olisi jatkossa mahdollisimman vaivatonta. Koen, että valitun esittämistavan avulla lukijan on helpompi hahmottaa kokonaiskuva verrattuna siihen, että ohjelmistokehys esiteltäisiin ensin abstraktina mallina, jonka päälle sovelluskohtainen toiminnallisuus myöhemmin liitettäisiin.

Tässä luvussa esimerkkisovelluksen rakennetta ja ohjelmakoodia esitellään siinä määrin, että lukija pystyy ymmärtämään sovelluksessa toteutetut pääsynvalvontamenetelmät. Sovelluksen täysi lähdekoodi on ladattavissa esimerkkisovelluksen projektisivulta [Simplecal, 2008], josta löytyvät myös ohjeet sovelluksen käyttöön ja asentamiseen. Asiasta kiinnostunut lukija voi ohjeiden avulla käyttää esimerkkisovellusta pohjana omien kokeilujensa toteuttamiseen.

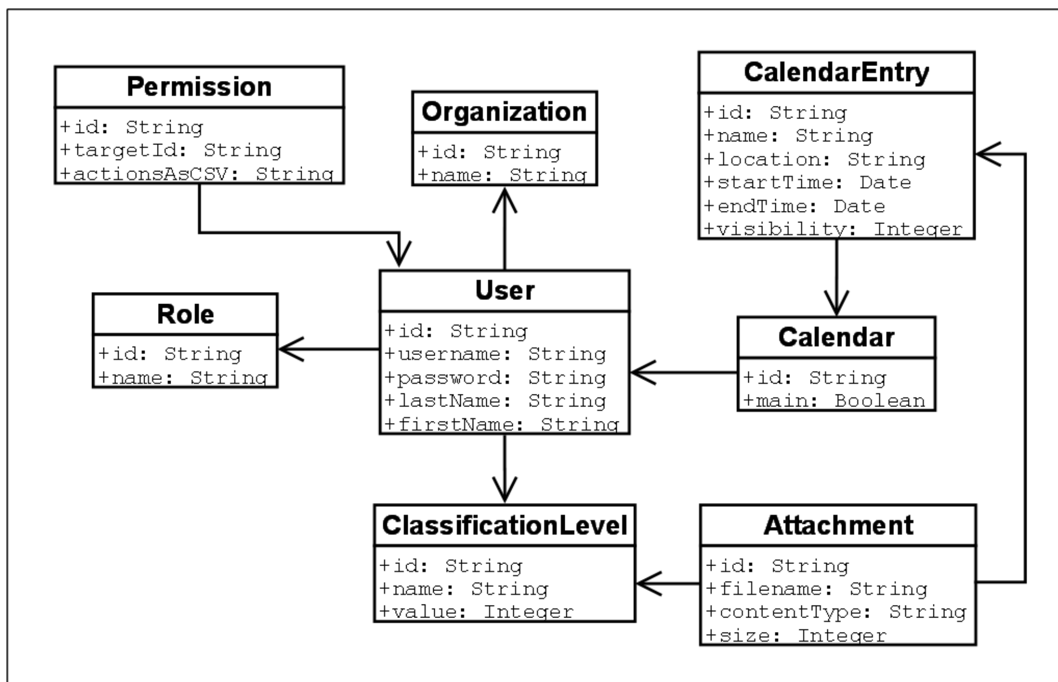
4.1. Esimerkkisovelluksen määrittelyä

Esimerkkisovellus on ajanhallinnan tueksi laadittu erittäin pelkistetty ryhmäkalenterisovellus, jonka avulla järjestelmän käyttäjät voivat kirjata järjestelmään kalenterimerkintöjä. Sovellus on alakohdassa 3.2.3 mainitun SaaS-mallin mukainen web-sovellus, jonka avulla useaan eri asiakasorganisaatioon kuuluvat käyttäjät voivat käyttää samaa sovellusinstanssia kalenterimerkintöjensä tallentamiseen. Organisaatiotieto toimii järjestelmässä eristämispisteenä, eli eri organisaatiot eivät normaaliolosuhteissa pääse käsiksi toistensa sovellusdataan. Ainoastaan `SUPERADMIN`-roolin omaavat käyttäjät näkevät järjestelmässä tietoa yli organisaatiorajojen.

4.1.1. Toiminnalliset vaatimukset

Ennen esimerkkisovelluksen toiminnallisten vaatimusten määrittelyä on paikallaan luoda silmäys esimerkkisovelluksemme käsitteelliseen malliin (Kuva 3). Se määrittelee karkealla tasolla sovelluksen tietosisällön, joka koostuu seuraavista käsitteistä:

- `User`: sovelluksen käyttäjä
- `Role`: käyttäjän rooli, joko `SUPERADMIN`, `ORGANIZATION_ADMIN` tai `USER`
- `Organization`: organisaatio, johon sovelluksen käyttäjä kuuluu
- `ClassificationLevel`: turvallisuusluokittelu, vaihtoehtoina `TOP_SECRET`, `SECRET`, `CONFIDENTIAL` ja `UNCLASSIFIED`
- `Permission`: eksplisiittinen käyttöoikeus johonkin resurssiin
- `Calendar`: loogisesti yhteenkuuluvat kalenterimerkinnät sisältävä säiliö
- `CalendarEntry`: kalenterimerkintä, sama merkintä voi liittyä useampaan kalenteriin (jolloin kyseessä on ryhmämerkintä, kuten vaikkapa kokous)
- `Attachment`: kalenterimerkintään mahdollisesti liittyvä liitetiedosto



Kuva 3. Esimerkkisovelluksen käsitteellinen malli

Arvoalue on mallinnettu siten, että kaikki mallin luokat kuuluvat pakettiin `fi.slite.simplecal.model`. Malliin kuuluvat luokat `User`, `Role`, `Organization`, `ClassificationLevel`, `Permission` ja `Attachment` ovat kuitenkin yleiskäyttöisiä, joten ne voitaisiin viedä ulos sovelluksesta ja kuvata vaikkapa paketissa `fi.slite.identitymanagement.model`. Tämän ratkaisun myötä luokat olisivat uudelleenkäytettävissä eri sovellusten välillä, ja vain selvästi sovelluskohtaiset luokat (`Calendar` ja `CalendarEntry`) jäisivät sovelluksen vastuulle.

Sovelluksen käyttäjät kuuluvat siis aina johonkin organisaatioon. Käyttäjä voi kuulua vain yhteen organisaatioon; jos henkilö liittyy useampaan eri

organisaatioon, on hänelle luotava erillinen edustajatunnus jokaista organisaatiota kohden. Kullakin sovelluksen käyttäjällä on (yksi) rooli, joka määrittelee hänen karkean tason oikeutensa järjestelmässä. Rooli `SUPERADMIN` mahdollistaa kaikkien hallintatoimintojen käytön, muun muassa uuden organisaation luomisen. Rooli `ORGANIZATION_ADMIN` mahdollistaa hallintatoiminnot organisaation sisällä, eli esimerkiksi käyttäjien lisäämisen organisaation alaisuuteen. Rooli `USER` mahdollistaa järjestelmän peruskäytön, eli kalenterimerkintöjen tekemisen ja muiden käyttäjien kalenterimerkintöjen tarkastelemisen sen mukaisesti, millaisia käyttöoikeuksia käyttäjällä kalenterimerkintöihin on.

Yhdellä käyttäjällä voi kalenterisovelluksessa olla useampia kalentereita, jolloin hän voi vaikkapa merkitä yhteen kalenteriin työmenonsa ja toiseen kalenteriin vapaa-ajan harrastuksensa. Oletusarvoisesti käyttäjä näkee aina koostenäkymän kaikista kalentereistaan, mutta hän voi kalenterimerkintöjä tarkastellessaan myös suodattaa jonkin kalenterin merkinnät kalenterinäkymästä pois. Yksi kalentereista pitää määritellä oletuskalenteriksi, johon muiden käyttäjien luomat ryhmämerkinnät sijoitetaan.

Kalenterimerkintä voi liittyä yhteen tai useampaan kalenteriin, jolloin puhutaan ryhmämerkinnästä (esimerkiksi kokous). Kullakin kalenterimerkinnällä on näkyvyysmääre, joka vaikuttaa siihen, miten muut kalenterisovelluksen käyttäjät näkevät kalenterimerkinnän tiedot. Kuka tahansa käyttäjä voi luoda kalenterimerkinnän, mutta vain merkinnän omistaja (eli se käyttäjä, jonka kalenteriin kalenterimerkintä on lisätty) voi oletusarvoisesti muokata merkintää tai poistaa sen, ellei hän ole erikseen antanut hallintaoikeutta merkintään jollekin sellaiselle käyttäjälle, joka myös on liitettynä kalenterimerkintään. Käyttäjä voi myös antaa hallintaoikeuksia koko kalenteriinsa muille organisaation käyttäjille, jolloin nämä saavat muokata kaikkia muita käyttäjän kalenterimerkintöjä paitsi niitä, jotka kalenterimerkinnän tekijä on merkinnyt rajoitetuiksi kalenterimerkinnän näkyvyysmääreen avulla.

Kalenterimerkintään voidaan liittää yksi tai useampi dokumentti liitetiedostona. Jos liitetiedoston lisääjälle on asetettuna turvaluokitus, myös liitetiedoston yhteyteen merkitään tiedoston avaamiseen vaadittava turvaluokitus, jolloin vain saman tai korkeamman turvaluokituksen omaava käyttäjä voi avata tiedoston. Perinteisen pakollisen pääsynvalvonnan mallin mukaisesti käyttäjän turvaluokitus määrää hänen luomansa liitetiedoston turvaluokituksen; käyttäjä voi tallentaa liitetiedoston joko sillä

turvaluokituksella, joka hänellä itselläänkin on, tai vaihtoehtoisesti asettaa sille korkeamman tason turvaluokituksen. Tällöin on huomionarvoista se, että liitetiedoston lisääjä menettää itsekin lukuoikeutensa liitetiedostoon. Tämä ominaisuus voi tuntua sovelluksen suhteen hieman ylimitoitetulta ja teennäiseltä, mutta se toimii esimerkkinä siitä, miten voimme sisällyttää sovellukseen myös pakollisen pääsynvalvonnan piiriin kuuluvia piirteitä.

Oheisessa taulukossa (Taulukko 1) on määritetty edellä olevan kuvauksen perusteella esimerkkisovelluksemme käyttötapaukset. Kunkin käyttötapauksen kohdalla on kerrottu minkälaiset toimijat saavat käyttötapauksen kuvaaman toiminnon suorittaa, ja kohdistuuko oikeuteen jotakin rajoitteita (rajoitteet listataan suluissa toimijakoodin jälkeen ja niiden merkitys on kuvattu taulukon alla). Tilan säästämiseksi käytämme taulukossa seuraavia lyhenteitä:

- SA = `SUPERADMIN`-roolin omaava käyttäjä
- OA = `ORGANIZATION_ADMIN`-roolin omaava käyttäjä
- OM = Kohteena olevan kalenterimerkinnän omistava käyttäjä
- OS = Osallistuja, eli ryhmämerkintään liitetty käyttäjä, joka ei ole merkinnän omistaja
- KKL = Eksplisiittisen käyttöluvan koko kalenteriin saanut käyttäjä
- MKL = Eksplisiittisen käyttöluvan merkintään saanut käyttäjä
- * = Kuka tahansa saa suorittaa toiminnon

Käyttötapauskoodi	Kuvaus	Oikeutetut toimijat
UC_MANAGE _ORGANIZATIONS	Organisaatioiden hallinta	SA
UC_LIST_USERS	Käyttäjien listaus	SA, OA (#1)
UC_CREATE_USER	Uuden käyttäjän lisääminen	SA, OA (#1)
UC_UPDATE_USER	Käyttäjän muokkaaminen	SA, OA (#1)
UC_SHOW_USER	Käyttäjän tietojen katselu	SA, OA (#1)
UC_REMOVE_USER	Käyttäjän poistaminen	SA(#2), OA (#1) (#2)
UC_LIST_ENTRIES	Kalenterimerkintöjen listaus (eli päiväkohtaisen varaustilanteen tarkastelu)	* (#3)
UC_CREATE_ENTRY	Kalenterimerkinnän lisäys	*
UC_UPDATE_ENTRY	Kalenterimerkinnän muokkaus	OM, MKL, KKL (#3)
UC_SHOW_ENTRY	Kalenterimerkinnän	* (#3)

	katselu	
UC_REMOVE_ENTRY	Kalenterimerkinnän poisto	OM, MKL, KKL (#3)
UC_ADD _ATTACHMENT	Liitetiedoston liittäminen kalenterimerkintään	OM, OS, MKL, KKL (#3)
UC_SHOW _ATTACHMENT	Liitetiedoston katsominen	* (#3) (#4)
UC_REMOVE _ATTACHMENT	Liitetiedoston poistaminen	OM, OS (#5), MKL, KKL (#3)
UC_ADD_ATTENDEE	Osallistujan liittäminen kalenterimerkintään	OM, MKL, KKL (#3)
UC_REMOVE_ATTENDEE	Osallistujan poistaminen kalenterimerkinnästä	OM, OS (#6), MKL, KKL (#3),
UC_LIST_CALENDARS	Kalenterien listaaminen	OM, KKL
UC_CREATE_CALENDAR	Uuden kalenterin luominen	OM
UC_UPDATE_CALENDAR	Olemassaolevan kalenterin tietojen päivitys	OM
UC_REMOVE_CALENDAR	Olemassaolevan kalenterin (ja sen sisältämien tapahtumien) poisto	OM (#7)

Taulukko 1. Sovelluksen käyttötapaukset

(#1) Käyttäjä saa hallinnoida vain oman organisaationsa käyttäjiä
(#2) Käyttäjä ei voi poistaa itseään järjestelmästä
(#3) Näkyvyysmääre voi rajoittaa merkinnän näyttämistä tai käsittelyä
(#4) Turvaluokitus voi rajoittaa tiedon näyttämistä tai käsittelyä
(#5) Jos osallistuja on itse lisännyt liitetiedoston eli on liitetiedoston omistaja
(#6) Osallistujalla on oikeus peruuttaa oma osallistumisensa
(#7) Oletuskalenteria ei saa poistaa

Listauksesta nähdään, että jo äärimmäisen pienellä ryhmätyösovelluksella voi olla hyvin mittava määrä erilaisia tietoturvarajoitteita, olkoonkin, että osa rajoitteista on käyttötapausten välillä samoja ja niitä voi siksi toteutuksessa uusiokäyttää. Perinteisissä web-sovelluksissa tietoturvarajoitteet ovat usein pirstaloituneet osaksi sovelluksen liiketoimintalogiikkaa ja käyttöliittymäkerrosta, jonka seurauksena ohjelmakoodista tulee helposti jo

ohjelmiston elinkaaren alkuvaiheessa hankalasti ylläpidettävää. Mikäli pääsynvalvontalogiikka pystytään esittämään keskitetyllä ja deklaratiivisella tavalla, mahdollisuudet menestyksekkääseen ohjelmistoprojektiin kasvavat suuresti.

4.1.2. Tekniset vaatimukset

Esimerkkisovelluksen on oltava standardi J2EE-mallin mukainen web-sovellus, jota voidaan ajaa missä tahansa Servlet API 2.4:n toteuttavassa sovelluspalvelimessa, jollainen on esimerkiksi Apache Tomcat 5.5. Sen on toimittava JDK-versiolla 1.5 (ja toki myös tätä uudemmilla); JDK 1.5:n myötä Java-kieleen tulivat muun muassa *annotaatiot* (annotations), joiden avulla Java-ohjelmakoodin yhteyteen voidaan liittää kontekstiin kuuluvaa metadataa. Pääsynvalvonnan tapauksessa luonnollinen tapa käyttää annotaatioita olisi esimerkiksi sovellettavan pääsynvalvontasäännön merkitseminen Java-servletin tai vaikkapa Strutsin `Action`-luokan yhteyteen.

Sen lisäksi, että esimerkkisovelluksen tulee olla J2EE-standardin mukainen web-sovellus, sen pitää myös olla autonominen sikäli, että sovellus ei saa vaatia sovelluksen ulkopuolisten konfiguraatiotiedostojen käsittelyä toimiakseen. Pidän sovelluksen ulkopuolista konfigurointia ongelmallisena sikäli, että web-sovelluksia siirretään elämänkaarensa aikana usein palvelinympäristöstä toiseen ja silloin sovelluksen ulkopuoliset konfiguraatiot voivat epähuomiossa jäädä viemättä uuteen ympäristöön, olkoonkin, että tällaiset konfiguraatiot ovat yleensä kuvattuna järjestelmien ylläpitodokumentaation yhteyteen. Parempi kuitenkin on, jos web-sovellus on tässä suhteessa autonominen.

Esimerkkisovelluksen avulla on pystyttävä rajoittamaan pääsyä sekä toimintoihin (actions) että informaatioon (data), ja sen avulla on voitava mallintaa kaikki luvussa 2 esitellyt pääsynvalvontatavat. Rajoituksien kohteena oleva informaatio on voitava esittää tavanomaisina JavaBean-luokkina ilman, että lähdekooditasolla olisi staattisia kytkentöjä ohjelmistokehykseen. Katson kuitenkin, että luokkia on voitava käsitellä dynaamisesti esimerkiksi AOP-menetelmillä, joiden avulla voimme jälkikäteen luoda jopa attribuuttitasolle menevän pääsynvalvonnan sensitiivisen informaation esittämisen estämiseksi.

Esimerkkisovelluksessa käytettävien pääsynvalvontamenetelmien on oltava modulaarisia siten, että menetelmiä voidaan uudelleenkäyttää eri sovelluksissa ja sovelluskehyksissä. Esimerkkisovellus toteutetaan Struts-sovelluskehyksen [Struts, 2008] avulla, mutta pääsynvalvontamekanismi on adapterien avulla oltava sovitettavissa myös muihin web-sovelluskehyksiin.

Pääsynvalvontamekanismin tulee myös olla riittävän suorituskykyinen, eli se ei saa aiheuttaa näkyvää kuormituksen kasvamista sovelluspalvelimella suhteessa tilanteeseen, jossa pääsynvalvonta ei ole kytkettynä päälle.

4.2. Arkkitehtuurikuvaus

Esimerkkisovellus on tyypillinen MVC-suunnittelumallin (Model-View-Controller) mukainen web-sovellus. Arvoalue (eli suunnittelumallin M-osa) on mallinnettu tavanomaisena JavaBean-luokkina, jotka eivät sisällä liiketoimintalogiikkaa. Mallin pysyvyys toteutetaan selkeyden vuoksi puhtaan JDBC:n avulla, eli Hibernaten kaltaista ORM-kehystä ei sovelluksen pienuuden vuoksi käytetä. Näkymät (eli suunnittelumallin V-osa) toteutetaan HTML:n ja Apache Velocityn [Velocity, 2008] avulla.

Kontrollerina (eli suunnittelumallin C-osana) toimii Apachen Struts-sovelluskehyksen versio 1.2.9. Olisimme toki voineet valita jonkin hieman nykyaikaisemman sovelluskehyksen, mutta Strutsin 1.2.x-sarja on esimerkkisovelluksellemme sikäli hyvä valinta, että se on web-sovelluskehityksen parissa työskennelleille yleensä varsin tuttu. Tämän ansiosta lukija voi esimerkkilistauksia tarkastellessaan keskittyä täysipainoisemmin itse pääsynvalvontaratkaisujen arviointiin, eikä lukijan keskittyminen häiriinny aiheemme kannalta epäolennaisten asioiden tarkastelun vuoksi.

Keskeisiä luokkia Strutsissa ovat `org.apache.struts.action.ActionServlet` ja `org.apache.struts.action.RequestProcessor`, jotka kontrolloivat palvelupyyntöjen käsittelyä ja välittämistä sovelluskohtaisille toiminnoille. Nämä luokat siis toteuttavat Strutsissa Front Controller -suunnittelumallin [Alur *et al.*, 2001, ss. 172-185] mukaisen pyyntöjen käsittelymekanismin, joka tarjoaa keskitetyn ja valvotun väylän sovelluskohtaisiin toimintoihin. Sovelluskohtaiset toiminnot toteutetaan perimällä Strutsin `org.apache.struts.action.Action`-luokka tai jokin sen aliluokista; suosittua on periä luokka `org.apache.struts.actions.DispatchAction`, jonka avulla loogisesti yhteenliittyvät operaatiot voidaan helposti toteuttaa saman luokan yksittäisinä metodeina `DispatchAction`-luokan huolehtiessa kontrollin ohjaamisesta oikealle metodille. Esimerkiksi CRUD-operaatiot [Grand, 2002, ss. 407-412] toteutetaan hyvin usein tämänkaltaisella tavalla.

Esimerkkisovelluksemme pääsynvalvonta toteutetaan kokonaisuudessaan Drools-sääntökoneen avulla. Toimintotason (actions) pääsynvalvonta ja

toimintojen kohteena olevan tietosisällön pääsynvalvonta (käytän tästä myöhemmin nimitystä *instanssitason pääsynvalvonta*) ovat kuitenkin teknisesti jossain määrin erilaisia prosesseja, joten käsittelen ne erillisissä osioissa. Ensiksi perehdymme toimintotason pääsynvalvonnan toteuttamiseen.

4.2.1. Toimintotason pääsynvalvonta

`RequestProcessor` on Struts-ohjelmistokehyksessä se komponentti, joka vastaa tavanomaisen web-sovelluksen toimintotason pääsynvalvonnan toteuttamisesta. Se on luonteva paikka pääsynvalvonnan hoitamiseen muun muassa seuraavista syistä:

- kaikki palvelupyynnöt kulkevat sen kautta;
- se tietää, kuka kohteena olevaa toimintoa on suorittamassa;
- se tietää, mitä toimintoa ollaan suorittamassa

Toinen varteenotettava vaihtoehto on kirjoittaa sellainen `DispatchAction`-luokkaa laajentava sovelluskohtainen kantaluokka, jonka kaikki sovelluksen konkreettiset toimintoluokat perivät. Molemmat ratkaisutavat ovat suurin piirtein yhtä työläitä toteuttaa, mutta `RequestProcessor`-luokan laajentamista puoltaa se seikka, että mikään toiminto ei voi sitä kovinkaan helposti ohittaa. Jos esimerkiksi laajentaisimme `DispatchAction`-luokkaa sovelluskohtaisella `fi.slite.simplecal.actions.CustomDispatchAction`-kantaluokalla, emme voisi pakottaa sovelluskehittäjää toteuttamaan toimintoluokkiaan siten, että ne perivät `CustomDispatchAction`-kantaluokan. Sen sijaan `RequestProcessor` on ohitettavissa vain Strutsin konfiguraatiotiedoston muokkaamisen kautta, joten `RequestProcessor`-luokan ohittaminen ei ainakaan vahingon seurauksena ole mahdollista, vaan se edellyttää aina tietoista pyrkimystä.

On olemassa myös kolmas vaihtoehto. Toimintotason pääsynvalvonta voidaan nimittäin viedä jopa kokonaan sovelluskehyksen ulkopuolelle Servlet API 2.3:ssa määriteltyjen *pyyntösuodattimien* (request filters) avulla. Pyyntösuodattimet ovat rajapinnan `javax.servlet.Filter` toteuttavia Java-luokkia, joita voidaan kytkeä web-sovelluksiin käytönkuvaimen avulla seuraavan esimerkin (Esimerkki 15) mukaisesti:

```
<web-app>
    <display-name>Test application</display-name>
    <filter>
        <filter-name>AccessControlFilter</filter-name>
        <filter-class>com.acme.AccessControlFilter</filter-class>
    </filter>
    <filter>
        <filter-name>TimeMeasuringFilter</filter-name>
        <filter-class>com.acme.TimeMeasuringFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>AccessControlFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <filter-mapping>
        <filter-name>TimeMeasuringFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

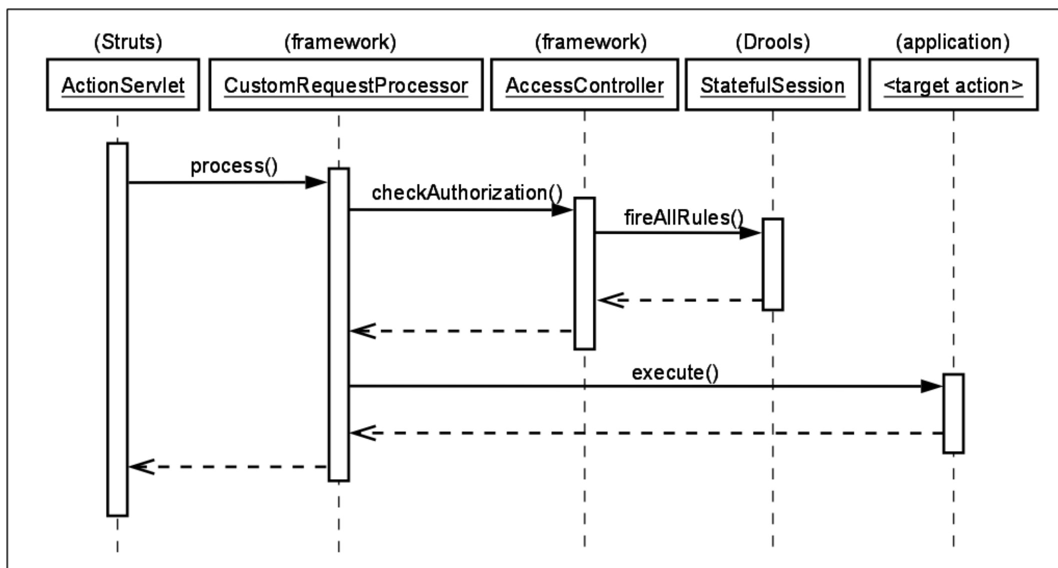
Esimerkki 15. Pyyntösuodattimien määrittely käytönkuvaimessa (`web.xml`)

Pyyntösuodattimet kytkeytyvät HTTP-pyynnön käsittelyketjuun siten, että niiden avulla voidaan suorittaa operaatioita sekä ennen kontrollin ohjaamista asiakaskoodille (joka voi olla esimerkiksi palvelinsovelma tai JSP-sivu), että kontrollin palattua takaisin asiakassovelluksen suorituksen jälkeen. Esimerkiksi yllä olevassa esimerkkilistauksessa määritelty `AccessControlFilter` tarkistaisi käyttäjän oikeudet ennen kontrollin siirtymistä Strutsin `ActionServlet`-luokalle, ja `TimeMeasuringFilter` puolestaan aloittaisi palvelupyynnön käsittelyyn kuluvan ajan mittaamisen ennen pyynnön ohjaamista asiakaskoodille ja lopettaisi mittaamisen sekä kirjoittaisi tulokset lokiin, kun kontrolli palautuisi asiakaskoodilta. Pyyntösuodattimet toteuttavat Chain of Responsibility -suunnittelumallin [Gamma *et al*. 1995, ss. 223-232] mukaisen käsittelyketjun, jossa ketjun osana oleva käsittelijä voi ottaa pyynnön käsiteltäväkseen, tai välittää sen ketjussa eteenpäin. Jos `AccessControlFilter` käyttöoikeudet tarkistettuaan huomaa, ettei käyttäjälle tulisi tarjota pääsyä kohteena olevaan toimintoon, se voi päättää käsitellä pyynnön itse ja olla välittämättä sitä ketjussa eteenpäin. Näin ollen pyyntö ei ikinä tule toiminnolle asti, ja `AccessControlFilter` vastaa pyynnön lähettäjälle että pyyntöä ei voida käsitellä, koska kohteena olevaan toimintoon ei ole käyttöoikeuksia.

Pyyntösuodattimien etuna on siis se, että niiden avulla toimintojen pääsynvalvonta voidaan tehdä web-sovelluskehyksistä riippumattomalla tavalla. Samalla tämä on myös niiden haittapuoli, eli menetämme samalla myös sovelluskehysten tarjoaman toiminnallisuuden. Pyyntösuodatin ei esimerkiksi pyyntöä ketjuttaessaan tiedä, mikä komponentti pyynnön lopulta tulee

käsittelemään, joten emme voisi viitata kohteena olevaan toimintoon (kuten Strutsin `Action`-luokan aliluokkiin) käyttöoikeusmäärityksissä. Pyyntösuodatin saa kyllä tietoonsa HTTP-pyynnön URL:in, mutta mikäli toimintojen käyttöoikeussäännöt eivät perustu URL:eihin (vaan esimerkiksi kohteena olevien toimintoluokkien nimiin) ei tästä informaatiosta ole kannaltamme kovinkaan suurta hyötyä. Näin ollen katson, että Strutsin `RequestProcessor`-luokan laajentaminen on paras tapa toteuttaa esimerkkisovelluksemme pääsynvalvonta.

Esimerkkisovelluksemme toimintokohtainen pääsynvalvonta toteutetaan näin ollen seuraavalla tavalla (Kuva 4):



Kuva 4. Toimintokohtainen pääsynvalvonta esimerkkisovelluksessa

Strutsin `ActionServlet`-luokka pyytää aluksi `CustomRequestProcessor`-luokkaa käsittelemään pyynnön. `CustomRequestProcessor` delegoi pääsynvalvonnan `AccessController`-komponentille, joka syöttää sääntökoneelle kohdealuetta koskevan tietämyksen (muun muassa tiedot käyttäjästä sekä kohteena olevasta toiminnosta) ja pyytää sääntökonetta evaluoimaan sääntökokoelmasta löytyvät säännöt. Lauetessaan säännöt täyttävät sääntökoneelle globaalina muuttujana välitettyä `AccessControlDecision`-oliota, joka päättelyn seurauksena sisältää tiedon siitä, onko kohteena olevan toiminnon käyttäminen luvallista käyttäjälle vaiko ei. Tämä tieto palautetaan `CustomRequestProcessor`-luokalle, joka tarkistaa tulokset ja kutsuu kohteena olevaa toimintoa mikäli toiminnon kutsuminen on sallittua. Jos näin ei ole, `CustomRequestProcessor` ohjaa käyttäjän virhesivulle, jossa häntä informoidaan laittomasta toiminnosta.

Toteutetut `CustomRequestProcessor`- ja `AccessController`-luokat ovat sellaisia, jotka voidaan viedä esimerkkisovelluksesta ohjelmistokehystasolle edellyttäen, että tietyt arvoalueen luokat (lähinnä `Role` ja `User`) on myös ulkoistettu sovelluksesta. `CustomRequestProcessor` on Strutsin versioon 1 liittyvä luokka; jos käyttäisimme jotakin muuta web-sovelluskehystä (vaikkapa Struts 2 tai Spring MVC), joutuisimme toteuttamaan siihen vastaavan liitännän. `AccessController`-luokka ei sen sijaan ole arkkitehtuurissamme sidoksissa mihinkään web-sovelluskehykseen, joten sen uudelleenkäyttö on vaivatonta.

Jotta sääntökone voisi päätellä, onko jonkin toiminnon suorittaminen laillista, pitää kunkin toiminnon yhteyteen kirjata tieto siitä, mitä tietoturvasääntöä toimintoon sovelletaan. Strutsin oletusmekanismi perustuu rooliperustaiseen pääsynvalvontaan, ja kunkin toiminnon edellyttämä rooli kirjataan Strutsin konfiguraatiotiedostoon (`struts-config.xml`). XDocletin [XDoclet, 2008] avulla roolitieto voidaan kuitenkin kirjata suoraan toimintoluokan yhteyteen seuraavasti (Esimerkki 16):

```
/**
 * @struts.action
 *     name="userManagementForm"
 *     path="/updateUser"
 *     scope="request"
 *     validate="false"
 *     parameter="method"
 *     roles="ORGANIZATION_ADMIN, SUPERADMIN"
 */
public class UpdateUserAction extends AbstractUserManagementAction
{
    // Action-specific code here
}
```

Esimerkki 16. Stutsin konfigurointi XDoclet-määreiden avulla

XDocletin avulla voidaan siis kirjoittaa ohjelmakoodin yhteyteen siihen loogisesti liittyvää metadataa. XDoclet jäsentää Javadoc-kommenteista tunnistamansa Struts-aiheiset määreet, ja tuottaa lopputuloksena XML-muotoisia konfiguraatioelementtejä, jotka se yhdistää Strutsin varsinaiseen konfiguraatiotiedostoon. JDK 1.5:sta lähtien sama asia olisi toteutettavissa myös Java-kieleen sisältyvien annotaatioiden avulla.

Esimerkissämme kiinnostavin attribuutti lienee `roles`-määre, sillä sen avulla on mahdollista määrittää pilkulla eroteltu lista rooleja, jotka oikeuttavat kyseisen toiminnon käyttöön. Koska haluamme korvata rooliperustaisen pääsynvalvonnan sääntöpohjaisella pääsynvalvonnalla, korvaamme roolilistan kirjoittamalla sen tilalle toimintoluokan toteuttaman käyttötapauksen koodin.

Attribuutin nimeksi jää kuitenkin tässä vaiheessa edelleen `roles`, koska emme halua lähteä muuttamaan Strutsin konfiguraatiotiedoston rakennetta tai säätämään XDoclet:in toimintaa sen tunnistamien attribuuttien suhteen. Esimerkki 17 sisältää päivitetyn määrittelyn:

```
/**
 * @struts.action
 *     name="userManagementForm"
 *     path="/updateUser"
 *     scope="request"
 *     validate="false"
 *     parameter="method"
 *     roles="UC_UPDATE_USER"
 */
public class UpdateUserAction extends AbstractUserManagementAction
{
    // Action-specific code here
}
```

Esimerkki 17. Roolimäärityksen korvaaminen käyttötapauskoodilla

Käyttötapauskoodin käyttäminen roolilistauksen asemesta lisää järjestelmän joustavuutta, koska voimme muuttaa toiminnon suorittamisen sallivia sääntöjä ajonaikaisesti ilman, että meidän täytyy kääntää sovellus uudestaan. Ja mikä tärkeintä, sääntöjen käyttäminen roolilistan asemesta tarjoaa meille mahdollisuuden säätää toimintojen käyttöoikeuksia asiakaskohtaisesti, jos tarvetta tällaiseen ilmenee. Jos tällaista ominaisuutta ei ole huomioitu järjestelmän suunnitteluvaiheessa, on sellainen toteuttaminen jälkikäteen hyvin raskas ja kallis prosessi.

Tässä alakohdassa kuvatulla tavalla pystymme siis toteuttamaan sovelluksiin toimintokohtaisen pääsynvalvonnan. Valitettavasti tämä ei useinkaan riitä, sillä oikeus käyttää jotakin järjestelmän toimintoa voi olla ehdollinen, eli riippua esimerkiksi siitä, mitä informaatiota kohteena olevalla toiminnolla on tarkoitus käsitellä. Tämän mahdollistamiseksi tarvitsemme instanssitason pääsynvalvontaa.

4.2.2. Instanssitason pääsynvalvonta

Instanssitason pääsynvalvonta tarkoittaa, että toiminnon kohteena oleva tieto vaikuttaa siihen, onko toiminnon suorittaminen sallittua vai ei. Esimerkiksi testisovelluksessamme käyttäjän muokkaaminen on sallittua, jos muokkaajan roolina on `SUPERADMIN`, tai jos muokkaajan roolina on `ORGANIZATION_ADMIN` ja kohteena oleva käyttäjä kuuluu samaan organisaatioon kuin muokkaajakin. Kohteena oleva tieto (muokattava käyttäjä) voi siis vaikuttaa siihen, saako toimintoa (`UpdateUserAction`) käyttää vai ei.

Jotta instanssitasoinen pääsynvalvonta olisi mahdollista, on pääsynvalvontakomponentin kyettävä selvittämään, mitä informaatiota kohteena oleva toiminto aikoo käsitellä. Tätä varten määrittelemme `GuardedAction`-nimisen rajapinnan (Esimerkki 18), jonka jokaisen sääntöpohjaista pääsynvalvontaa soveltavan toiminnon on toteutettava:

```
public interface GuardedAction
{
    public GuardContext getGuardContext(ActionMapping mapping,
                                        ActionForm form,
                                        HttpServletRequest request)
        throws Exception;
}
```

Esimerkki 18. `GuardedAction`-rajapinta

Jos kohteena oleva toiminto toteuttaa `GuardedAction`-rajapinnan, `CustomRequestProcessor` pyytää kohteena olevalta toiminnolta tätä rajapintaa hyväksikäyttämällä tiedon siitä, mitä informaatiota (Java-luokan instanssia) kohdetoiminto aikoo käsitellä. Juuri `GuardedAction`-rajapinnan käyttäminen määrittää viime kädessä myös sen, tulkitaanko aiemmin kuvattu `roles`-määre roolilistaukseksi vai käyttötapauskoodiksi. Näin ollen säilytämme sovelluksessa myös option käyttää jossakin toiminnossa perinteistä roolipohjaista pääsynvalvontaa, jos katsomme tämän tarpeelliseksi. Esimerkki 19 osoittaa, miten `GuardContext`-luokkaa käytetään kohdetiedon välittämiseen pääsynvalvontakomponentille:

```
public class UpdateUserAction extends AbstractUserManagementAction
{
    public GuardContext getGuardContext(ActionMapping mapping,
                                        ActionForm form,
                                        HttpServletRequest request)
        throws Exception
    {
        DynaActionForm daf = (DynaActionForm)form;
        return new GuardContext(
             DataFacade.findUserById(daf.getString("id")));
    }
}
```

Esimerkki 19. Toiminnon kohteena olevan instanssin välitys pääsynvalvontakomponentille

Esimerkkitoiminto perii `AbstractUserManagementAction`-luokan, joka puolestaan toteuttaa `GuardedAction`-rajapinnan. Se pyytää kohteena olevan tietueen tunnisteen `DynaActionForm`-luokalta (tämä on Struts-sovelluksissa käytettävä tietorakenne, jonka avulla voidaan kuvata HTML-lomakkeen

sisältämä informaatio), hakee tätä vastaavan käyttäjätietueen tietokannasta `DataFacade`-fasadiluokan avulla ja luo `GuardContext`-olion, jonka avulla käyttäjätietuetta vastaava `User`-objekti välitetään sovelluksen pääsynvalvontakomponentille. Pääsynvalvontakomponentti noutaa automaattisesti kohdeobjektiin mahdollisesti myönnetyt eksplisiittiset käyttöluvat (luokan `Permission` esiintymiä) ja välittää tiedot sääntökoneelle, joka ratkaisee sen, onko sovelluksen käyttäjällä oikeutta kohteena olevaan tietoon vaiko ei. Jos oikeutta ei ole, `CustomRequestProcessor`-luokka ei ikinä kutsu varsinaista kohteena olevaa toimintoa vaan esittää käyttäjälle virhesivun, jossa kerrotaan hänen yrittäneen suorittaa laittoman toiminnon.

Edellisessä kappaleessa mainittiin, että pääsynvalvontakomponentti osaa noutaa automaattisesti kohdeobjekteihin myönnetyt eksplisiittiset käyttöluvat. Jotta tämä olisi mahdollista, on kohteena olevien objektien oltava yksilöllisesti tunnistettavissa. Vaikka sovelluksen tietosisältöä kuvaavassa luokkakaaviossa ei sitä eksplisiittisesti ilmaistukaan, sovelluksen arvoalueeseen kuuluvat luokat toteuttavat `GloballyIdentifiable`-rajapinnan, jonka määrittämän `setId(String id)` -metodin kautta objektille voidaan asettaa universaalisti yksilöllinen tunniste seuraavan esimerkin (Esimerkki 20) mukaisesti:

```
public static void saveUser(User user)
    throws PersistenceException
{
    if (user.getId() == null || user.getId().trim().length() == 0) {
        user.setId(UUID.randomUUID().toString());
        UserManager.add(user);
    }
    else {
        UserManager.update(user);
    }
}
```

Esimerkki 20. Yksilöllisen tunnisteen antaminen kohdeobjekteille

Tunnisteita luotaessa käytetään apuna JDK 1.5:ssa esiteltyä `java.util.UUID`-luokkaa, jonka pitäisi taata, että luotavat tunnisteet ovat universaalisesti yksilöllisiä. Näin ollen `Permission`-luokan avulla voidaan viitata mihin tahansa `GloballyIdentifiable`-rajapinnan toteuttavan luokan esiintymään jopa ilman, että `Permission`-luokkaan tarvitsee kuvata kohteena olevan esiintymän luokkatietoa. Pelkkä tunniste riittää, edellyttäen että esiintymille myönnetyt tunnisteet on luotu esimerkin kuvaamalla tavalla. Esimerkkisovelluksessa tätä mekanismia hyödynnetään siten, että kalenterimerkintöihin tai kalentereihin voidaan myöntää sen avulla eksplisiittisiä hallintaoikeuksia.

Kuvatulla tavalla toimintokohtaista pääsynvalvontaa voidaan siis laajentaa siten, että pääsy toimintoon voi olla riippuvainen kohteena olevan objektin tunnisteesta tai tietosisällöstä. Tämä menettelytapa sopii erinomaisesti esimerkiksi CRUD-operaatioiden (Create, Read, Update, Delete) toteuttamiseen, koska kohteena on yleensä aina jokin tietty yksittäinen Java-luokan instanssi, johon voimme kooditasolla viitata. Mutta web-sovelluksissa on yleisesti paljon myös muunlaisia toimintoja, kuten hakutoimintoja, tietuelistauksia ja sellaisia tietoa suodattavia toimintoja, jotka voivat esittää käyttöliittymissä valikoidusti tietoa myös ”kielletyiksi” luokitelluista tietueista. Miten valittu pääsynvalvontamalli soveltuu tällaisten toimintojen pääsynvalvontaan?

Selvää on, että pääsynvalvontamenetelmän ilmaisuvoima ei edellä kuvatun kaltaisena tähän sovellu. Aiempana kuvattu menetelmä soveltuu parhaiten tilanteisiin, joissa pääsynvalvontatarkastelun lopputuloksena on selvästi joko päätös ”kyllä” tai ”ei”. Jos haluamme tuoda lisää hienojakoisuutta tähän menettelyyn, meidän on kyettävä välittämään pääsynvalvontakomponentille lisätietoa toiminnan avuksi, minkä voimme tehdä jo aiemmin sivuamamme `GuardContext`-objektin avulla. Seuraavassa listauksessa (Esimerkki 21) esitellään `GuardContext`-luokan tarjoamat rakentimet:

```
public class GuardContext
{
    public GuardContext(Object target);

    public GuardContext(List targets);

    public GuardContext(Object target,
                        ResolutionPolicy resolutionPolicy);

    public GuardContext(List targets,
                        ResolutionPolicy resolutionPolicy);

    public GuardContext(Object target,
                        ResolutionPolicy resolutionPolicy,
                        Transformer transformer);

    public GuardContext(List targets,
                        ResolutionPolicy resolutionPolicy,
                        Transformer transformer);
}
```

Esimerkki 21. `GuardContext`–luokan rakentimet

`GuardContext`-luokan esiintymä voidaan siis luoda välittämällä rakentimelle vain yksittäinen pääsynvalvonnan kohteena oleva kohdeobjekti, tai vaihtoehtoisesti lista kohteena olevia objekteja. Esimerkiksi käyttäjälistauksen tuottavassa toiminnossa (`ListUsersAction`) välitettäisiin siis joukko

käyttäjäobjekteja, mutta käyttäjän muokkaustoiminnossa (`UpdateUserAction`) vain yksi käyttäjäobjekti. Oletusarvoisesti pääsynvalvontakomponentti estää pääsyn kohdetoimintoon, mikäli yksikin käsiteltävinä olevista kohdeobjekteista on sellainen, johon toiminnon suorittajalla ei ole käyttöoikeutta, mutta voimme ohjeistaa pääsynvalvontakomponenttia toimimaan eri tavoin `ResolutionPolicy`-luokan avulla. Toistaiseksi olen identifioinut seuraavat kolme politiikkaa joiden avulla pääsynvalvontakomponentin toimintaa voidaan ohjailla:

- `ResolutionPolicy.DENY_ACCESS`
- `ResolutionPolicy.REMOVE_INSTANCE`
- `ResolutionPolicy.TRANSFORM_OR_REMOVE_INSTANCE`

Oletuspolitiikkana on `DENY_ACCESS`, joka toimii estämällä pääsyn kohteena olevaan toimintoon, mikäli yksikin toiminnon kohdeobjekteista on sellainen johon ei ole käyttöoikeutta. Suurin osa esimerkkisovelluksemme toiminnoista hyödyntää juuri tätä politiikkaa; esimerkiksi CRUD-operaatioiden tapauksessa ei ole järkevää tarjota minkäänlaista pääsyä kohteena olevaan toimintoon, jos kohteena olevaa tietuetta ei ole lupa käsitellä.

Politiikka `REMOVE_INSTANCE` toimii siten, että pääsynvalvontakomponentti poistaa kielletyt objektit kohdeobjektien joukosta, jolloin kohteena oleva toiminto saa listan, joka sisältää vain sallittuja kohdeobjekteja. Esimerkkisovelluksessamme tätä hyödyntää käyttäjät listaava toiminto; `ORGANIZATION_ADMIN`-roolin omaavat käyttäjät voivat hallinnoida vain omaan organisaatioonsa kuuluvia käyttäjiä, joten pääsynvalvontakomponentti suodattaa kohdejoukosta pois ne käyttäjät, joiden organisaatiotieto ei täsmää. Sen sijaan `SUPERADMIN`-roolin omaavan käyttäjän tapauksessa suodatusta ei tehdä, koska tällainen käyttäjä saa muokata kaikkia sovelluksen käyttäjiä näiden organisaatiosta riippumatta.

Kaikkein ilmaisuvoimaisin politiikka on `TRANSFORM_OR_REMOVE_INSTANCE`, joka tarjoaa täyden kontrollin siihen, mitä kielletyksi luokiteltavalle kohdeobjektille voidaan tehdä. Nimensä mukaisesti yksittäinen kohdeobjekti voidaan poistaa kohdeobjektien joukosta, tai siihen voidaan kohdistaa toimenpiteitä (eli transformaatioita), jotka muokkaavat kohdeobjektin tilaa siten, että kohdeobjekti säilyy edelleen kohdejoukossa mutta ei enää sisällä sellaista informaatiota, jonka esittäminen olisi erityisen haitallista. Esimerkkisovelluksessamme muiden kalenterikäyttäjien rajoitetun näkyvyyden

omaavat kalenterimerkinnät käsitellään tällaisella tavalla, jolloin merkinnät sisältävät transformaation seurauksena enää vain merkinnän alkamis- ja loppumisajan, mutta kaikki muu informaatio on transformaation seurauksena piilotettua. Näin ollen muut kalenterikäyttäjät näkevät, että käyttäjällä on tiettyyn aikaan kalenterissaan varauksia, mutta eivät näe varauksien tarkempia tietoja. Jos kohteena oleva tieto piilotettaisiin kokonaisuudessaan ja kyseinen ajankohta näytettäisiin kalenterissa vapaana, muut kalenterikäyttäjät voisivat tehdä kohdekäyttäjän kalenteriin varauksia, jotka menisivät päällekkäin käyttäjän omien henkilökohtaisten merkintöjen kanssa.

Politiikka `TRANSFORM_OR_REMOVE_INSTANCE` vaatii toimiakseen sen, että `GuardContext`-objektiin liitettävä `Transformer`-rajapinnan toteuttava käsittelijä. `Transformer`-rajapinta (Esimerkki 22) määrittää vain yhden metodin, joka konkreettisen luokan on toteutettava:

```
public interface Transformer
{
    public Object transform(Object obj)
        throws Exception;
}
```

Esimerkki 22. `Transformer`-rajapinta

Jos transformoija katsoo, että kohteena olevaa objektia ei ole lainkaan syytä sisällyttää tulosjoukkoon, se voi palauttaa käsittelyn seurauksena arvon `null`. Jos kohdeobjekti on muokattavissa sellaiseksi, että kohdejoukkoon kuuluminen on mahdollista, transformoija suorittaa kohdeobjektille haluamansa toimenpiteet ja palauttaa kohdeobjektin takaisin pääsynvalvontakomponentille. Mikäli transformoinnista aiheutuu jokin poikkeus, pääsynvalvontakomponentti poistaa kohdeobjektin tulosjoukosta automaattisesti.

Mainitut menetelmät lisäävät pääsynvalvontakomponentin ilmaisuvoimaa selvästi. Menetelmät kattavat valtaosan tyypillisten web-sovellusten pääsynvalvonnan tarpeista, ja niitä on edelleen mahdollista laajentaa esimerkiksi luomalla uusia `ResolutionPolicy`-määrityksiä. Jos käyttötapaus on eksoottinen ja vaikeasti toteutettavissa mainittujen menetelmien avulla, on toki edelleen mahdollista kirjoittaa liiketoimintalogiikka myös itse toimintokerrokselle ja sivuuttaa pääsynvalvontakomponentti kokonaan. Tähän ei kuitenkaan pitäisi tyypillisissä web-sovelluksissa olla juurikaan tarvetta.

4.2.3. Pääsynvalvonta käyttöliittymäkerroksessa

Edellä on kuvattu menetelmiä, joiden avulla pääsynvalvontaa toimeenpannaan liiketoimintalogiikkakerroksella. Jos tarkoituksenamme on laatia käyttäjäystävällisiä web-sovelluksia, emme kuitenkaan voi pitää pääsynvalvontaa pelkästään liiketoimintalogiikkakerroksen asiana. Jos siis esimerkiksi `UpdateUserAction`-toiminnon kutsuminen tietyllä syötteellä aikaansaisi laittoman toiminnon ja raportoitaisiin käyttäjälle virhesivun avulla, meidän tulisi jo lähtökohtaisesti estää mahdollisuus siihen, että tällainen toiminto olisi ylipäätään käyttäjän ajettavissa käyttöliittymätasolla.

Hyvän käyttökokemuksen takaamiseksi pyrimme siis siihen, että käyttäjälle tarjotaan käyttöliittymissä vain sellaisia toimintoja, joita käyttäjällä on oikeasti mahdollisuus käyttää. Käyttöliittymistä ei siis saa löytyä linkkejä tai painikkeita, joiden seurauksena käyttäjälle esitetään virhesivu, jossa kerrotaan käyttäjän yrittäneen laittomaa toimintoa. Jotta tämä olisi mahdollista, on käyttöliittymäkerrokselle tarjottava pääsynvalvontakomponentti käyttöön, minkä teemme `CustomDispatchAction`-kantaluokassa seuraavan listauksen (Esimerkki 23) kuvaamalla tavalla:

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
    throws Exception
{
    request.setAttribute("currentAction", this);
    request.setAttribute("now", new Date());
    request.setAttribute("dateFormatter", dateFormatter);
    request.setAttribute("dotFormatter", dotFormatter);
    request.setAttribute("commaFormatter", commaFormatter);
    request.setAttribute("guard", new AccessController());
    request.setAttribute("appUser", getUser(request));

    try {
        return super.execute(mapping, form, request, response);
    }
    catch (Exception e) {
        log.error("Error occurred during request processing", e);
        throw e;
    }
}
```

Esimerkki 23. `CustomDispatchAction`-luokan `execute`-metodi

Kantaluokka asettaa pyynnön attribuuteiksi erilaisia *näkymän avustajia* (view helper) [Alur *et al.*, 2001, ss. 186-202], joita käyttöliittymäkerros voi hyödyntää. Esimerkissä pääsynvalvontakomponentti `AccessController` välitetään

käyttöliittymäkerrokselle nimellä `guard`, jolloin voimme käyttää sitä vaikkapa Velocity-sivupohjassa seuraavan esimerkin (Esimerkki 24) mukaisesti:

```
<form name="showEntryForm">

  #if ($guard.check($request, "UC_UPDATE_ENTRY", $entry))
    <input type="button" value="Muokkaa"
     onClick="update('$entry.id')">
  #end

  #if ($guard.check($request, "UC_REMOVE_ENTRY", $entry))
    <input type="button" value="Poista"
     onClick="confirmRemove('$entry.id')">
  #end

  <input type="button" value="Takaisin" onClick="cancel()">
</form>
```

Esimerkki 24. Pääsynvalvontakomponentin kutsuminen Velocity-sivupohjasta

Avustajien avulla itse käyttöliittymät pysyvät siisteinä, emmekä joudu kirjoittamaan näkymiin monimutkaista liiketoimintalogiikkaa. Kaiken kaikkiaan pääsynvalvontamenetelmät näkyvät käyttöliittymäkerroksessa erittäin minimaalisesti, eli mitään edellisessä esimerkissä kuvattua monimutkaisempaa toimenpidettä ei käyttöliittymäkerroksessa tarvitse tehdä pääsynvalvonnan mahdollistamiseksi.

4.3. Pääsynvalvontasääntöjen hallinta ja esitystavat

Päätin, että esimerkkisovellukseen ei toteuteta pääsynvalvontasääntöjen hallintatoimintoja sen vuoksi, että Drools itsessään sisältää hyviä työkaluja sääntökokoelmien hallintaan. Sääntöjä voidaan laatia esimerkiksi Drools 4.0.4 BRMS -sovelluksen avulla, tai käyttäen Eclipseen tarkoitettua Drools IDE -laajennosta, jotka molemmat ovat ladattavissa Droolsin projektisivustolta. Itse päätin laatia pääsynvalvontasäännöt ilman työkaluja tekstieditorin avulla.

Pidän pääsynvalvontasääntöjen esityksessä keskeisenä sitä, että säännöt ovat selkeitä ja niiden logiikka on helposti seurattavissa. Täten päätin jo alkuvaiheessa toteuttaa säännöt arvoaluekohtaisen kielen (DSL) avulla. Tällä tavalla sääntötiedostossa määritetyt säännöt pysyvät selkeinä, kuten käyttötapausta UC_REMOVE_USER koskevasta esimerkkilistauksesta (Esimerkki 25) voimme nähdä:

```
#--------------------------#
# Use case: UC_REMOVE_USER #
#--------------------------#

rule "SUPERADMINs can remove users"
    agenda-group "UC_REMOVE_USER"
    salience 30
    when
        user has "SUPERADMIN" role
    then
        operation is permitted
end

rule "And ORGANIZATION_ADMINs can remove users from their own
organization"
    agenda-group "UC_REMOVE_USER"
    salience 20
    when
        user has "ORGANIZATION_ADMIN" role
        target belongs to same organization than user
    then
        operation is permitted
end

rule "But suicides are never allowed"
    agenda-group "UC_REMOVE_USER"
    salience 10
    when
        target is user
    then
        operation is not permitted
end
```

Esimerkki 25. Käyttötapauksen UC_REMOVE_USER pääsynvalvontasäännöt

Esimerkkilistauksessa esiintyy `salience`-attribuutti, jonka avulla sääntöjen keskinäiseen suoritusjärjestykseen on mahdollista vaikuttaa. Mitä isompi arvo attribuutilla on, sitä aikaisemmassa vaiheessa sääntö suoritetaan. Vaikka sääntökoneita käytettäessä pidetäänkin suotavana sitä, että säännöt pidetään mahdollisimman riippumattomana toisistaan, suoritusjärjestyksen asettaminen on aika ajoin tarpeellista. Jos emme esimerkkitapauksessa asettaisi suoritusjärjestystä kuvatun kaltaiseksi, saatettaisiin sääntö ”`But suicides are never allowed`” suorittaa aiemmin kuin sääntö ”`SUPERADMINs can remove users`”, jolloin jälkimmäinen sääntö ylikirjoittaisi ensimmäisen säännön tuloksen ja `SUPERADMIN`-roolin omaava käyttäjä pystyisikin poistamaan itsensä, mikä ei ole tarkoituksena.

Tästä päästäänkin pohtimaan *ristiriitatilanteiden hallinnan* (conflict resolution) problematiikkaa. Kun suorituksen kohteena on irrallisia sääntöjä, saattaa jonkin säännön evaluoiminen johtaa positiiviseen lopputulokseen ja jonkin muun säännön negatiiviseen lopputulokseen. Esimerkkisovelluksessa olen valinnut

ristiriitatilanteiden ratkaisutavaksi puhtaasti suoritusjärjestyksen määräämän tavan, eli viimeinen suoritettu sääntö on vastuussa lopullisesta päätöksestä. Toinen ratkaisuvaihtoehto voisi olla esimerkiksi sellainen, että kullekin päätökselle asetettaisiin painoarvot, ja korkeimman painoarvon saanut päätös olisi merkitsevä. Esimerkkisovelluksen tapauksessa valittu ratkaisutapa toimii kuitenkin varsin hyvin.

Joskus voi käydä niin, että yksikään sääntö ei laukea. Tästä syystä pitää valita oletuspolitiikka, joka määrää miten näissä tilanteissa tulee menetellä. Oletuspolitiikka voi olla salliva (jos jotain ei ole erikseen kielletty, se on sallittua) tai kieltävä (jos jotain ei ole erikseen sallittu, se on kiellettyä). Koska haluamme laatia korkean tietoturvatason sovelluksen, valitsemme jälkimmäisen politiikan. Konkreettisesti tämä toteutetaan siten, että päättelyn tulokset keräävä olio (`AccessControlDecision`) asetetaan ennen sääntökoneelle välitystä tilaan, jossa päätös on kielteinen.

Kerroin aiemmin, että Droolsissa on mahdollista rajata sitä, mitkä säännöt voivat suoritettaessa laueta. Koska teemme pääsynvalvontapäätöksiä sen perusteella mikä käyttötapaus on kyseessä, on luontevaa että vain kyseiseen käyttötapaukseen liittyvät säännöt voivat laueta. Droolsissa säännöt ryhmitellään käyttötapausten mukaisesti `agenda-group` -määreen avulla, jolloin voisimme suorittaa vain tähän käyttötapaukseen liittyvät säännöt seuraavan listauksen (Esimerkki 26) mukaisesti:

```
StatefulSession session = ruleBase.newStatefulSession();
session.setFocus("UC_REMOVE_USER");
session.fireAllRules();
```

Esimerkki 26. Valikoitavien sääntöjen rajoittaminen ennen sääntöjen suorittamista

Voimme myös ohjeistaa Droolsia vaihtamaan työjärjestystä (eli agendaa) sääntöjen suorituksen aikana. Tämä on erityisen hyödyllinen ominaisuus sikäli, että jos jonkin käyttötapauksen pääsynvalvontasäännöt ovat identtiset toisen tapauksen kanssa (tai toisen käyttötapauksen sääntöjoukko on kyseessä olevan käyttötapauksen sääntöjen osajoukko), voimme delegoida sääntöjen suorituksen toiselle sääntöjoukolle seuraavan esimerkin (Esimerkki 27) mukaisesti:

```
#--------------------------------#
# Use case: UC_REMOVE_ATTACHMENT #
#--------------------------------#

rule "User can remove attachment if he can remove entry"
    agenda-group "UC_REMOVE_ATTACHMENT"
    salience 20
    when
    then
        delegate to UC_REMOVE_ENTRY
end

rule "User can also remove attachment if he has created it"
    agenda-group "UC_REMOVE_ATTACHMENT"
    salience 10
    when
        user has created the attachment
    then
        operation is permitted
end
```

Esimerkki 27. Kontrollin siirtäminen sääntöjoukolta toiselle

Kun delegoinnin kohteena olevan työjärjestykseen (`UC_REMOVE_ENTRY`) kuuluvat säännöt on evaluoitu, Drools-sääntökone palaa takaisin suorittamaan edelliseen työjärjestykseen (`UC_REMOVE_ATTACHMENT`) sisältyviä sääntöjä. Koska työjärjestykset kuvataan Droolsissa sisäisesti pinorakenteen avulla, voi esimerkiksi työjärjestys `UC_REMOVE_ENTRY` edelleen hyödyntää työjärjestystä `UC_UPDATE_ENTRY`, joten mahdollisuudet ovat lähes rajattomat. Vaikka Drools ei suoraan sallikaan säännön kutsuvan suorituksensa aikana toista sääntöä, `agenda-group` -määre tuo joustavuutta käsittelyyn ja mahdollistaa riittävän tasoisen uudelleenkäytön sääntöjen välillä.

Esimerkkilistauksissa olen siis käyttänyt luonnollisen kaltaista kieltä sääntöjen esittämiseen. Se, mitä mikäkin sääntö tekee, kuvataan DSL-tiedostossa, josta olen poiminut joitakin esimerkkimäärityksiä seuraavaan listaukseen (Esimerkki 28):

```
[condition][] user has "{roleName}" role =
    rc1 : ReasoningContext (user.role.name == "{roleName}")

[condition][] target belongs to same organization than user =
    rc3 : ReasoningContext (target != null, targetType == "User",
          target.organization != null, user.organization.id ==
          target.organization.id)

[condition][] target is user =
    rc4 : ReasoningContext (target != null, user.id == target.id)

[consequence][] operation is permitted = decision.setPermitted(true);

[consequence][] operation is not permitted =
```

```
    decision.setPermitted(false);

[consequence][] delegate to "{agendaGroup}" =
    drools.setFocus("{agendaGroup}");
```

Esimerkki 28. Esimerkkilistauksissa käytettävien sääntömäärityksiä

Esimerkkilistausta tarkastellessamme huomaamme DRL-muotoisten sääntöjen sisältävän paljon esimerkiksi `null`-tarkistuksia, jotka voimme DSL:n käytön avulla varsin täydellisesti piilottaa. DSL:t pitävät varsinaiset säännöt siisteinä, ja parantavat sääntöjen luettavuutta olennaisesti. Niiden avulla voimme *kapseloida* sääntöihin liittyvän monimutkaisuuden yhteen paikkaan.

4.4. Prototyypin arviointia

Seuraavissa aliluvuissa arvioin kuvaamaani esimerkkisovellusta pääosin kvalitatiivisin menetelmin. Arvioin paitsi arkkitehtuurin onnistuneisuutta, myös sääntöjen määrittämisen vaikeutta sekä sääntökoneen käytön vaikutusta järjestelmän kokonaissuorituskykyyn.

4.4.1. Arkkitehtuuri

Toteutettu pääsynvalvonta-arkkitehtuuri soveltuu varsin hyvin web-sovellusten ja ryhmätyösovellusten pääsynvalvontaan. Arkkitehtuuri on myös suhteellisen modulaarinen, eli pääsynvalvontakomponentti on pienin muutoksin muunneltavissa myös sellaisten sovellusten yhteydessä käytettäväksi, jotka eivät ole varsinaisia web-sovelluksia. Samaten pääsynvalvontakomponentti ei pääosiltaan myöskään ole sidottu Drools-sääntökoneeseen, joten Drools voitaisiin suhteellisen helposti korvata esimerkiksi Javalle käännetyllä Prolog-toteutuksella.

Suurin osa web-sovellusten pääsynvalvontatarpeista voidaan tyydyttää tutkielmassa esitettyjen menetelmien avulla. On kuitenkin olemassa tilanteita, joissa pääsynvalvontaa ei voida automatisoida parhaalla mahdollisella tavalla. Esimerkiksi luontitoiminnot (UC_CREATE_USER, UC_CREATE_ENTRY jne.) ovat ongelmallisia sikäli, että kohteena olevaa objektia ei ole olemassa vielä ennen toiminnon suorittamista, koska toiminto itsessään vasta luo objektin. Jos pääsynvalvontaa siis sovelletaan toimintoihin ennaltaehkäisevästi (eli käyttäjää ei päästetä edes toimintoon asti jos oikeutta kohdeobjektiin ei ole), on pääsynvalvontakomponentin mahdotonta tehdä päätöstä toiminnon sallimisesta vielä tässä vaiheessa, kun käsillä ei ole konkreettista objektia jota tarkastella.

Edellä mainittuun ongelmaan löytyy ratkaisuja, mutta niissä on omat ongelmansa. Yksinkertaisin ratkaisu lienee se, että pääsynvalvontakomponentti ei kutsukaan kohdetoimintoa etukäteen, vaan päästää suorituksen etenemään kohdetoimintoon asti ja pääsynvalvontatarkistus tehdään eksplisiittisesti siellä. Näin ollen päätyisimme seuraavanlaiseen pääsynvalvontaratkaisuun (Esimerkki 29):

```
public ActionForward save(ActionMapping mapping,
                          ActionForm form,
                          HttpServletRequest request,
                          HttpServletResponse response)
    throws Exception
{
    DynaActionForm daf = (DynaActionForm)form;
    ActionErrors errors = validateFormData(daf);

    if (!errors.isEmpty()) {
        saveErrors(request, (ActionMessages)errors);
        return mapping.findForward(EDIT_USER_VIEW);
    }
    else {
        User user = new User();
        populateUserDetailsFromForm(user, daf);
        AccessController.authorize(request, "UC_CREATE_USER", user);
        DataFacade.saveUser(user);
        request.setAttribute("redirectMessageKey",
            "editUser.saveForwardText");
        request.setAttribute("redirectUrl", request.getContextPath() +
            mapping.findForward(USER_LIST_REDIRECT).getPath());
        return mapping.findForward(FORWARD_SCREEN_VIEW);
    }
}
```

Esimerkki 29. AccessController-komponentin suora kutsuminen toiminnoissa

Esimerkissä käytetty `authorize`-kutsu eroaa aiemmin käyttöliittymätasolla hyödynnetystä `check`-kutsusta siten, että `authorize` heittää poikkeuksen mikäli käyttöoikeutta kohdeobjektiin ei ole. Esimerkistä nähdään, että pääsynvalvontakomponentin eksplisiittinen kutsuminen ei aiheuta merkittävästi lisätyötä, mutta se on ongelmallinen ainakin kahdesta syystä. Ensinnäkin, ohjelmistokehittäjä saattaa unohtaa lisätä pääsynvalvontakutsun ohjelmakoodiinsa, ja toiseksi pääsynvalvontamekanismi ei enää ole yhdenmukainen eri toimintojen välillä. Joissakin toiminnoissa pääsynvalvonta suoritettaisiin siis jo ennen toimintoon saapumista, joissakin toiminnoissa vasta toiminnon sisällä. Tällainen epäsymmetrinen ratkaisu on ohjelmistoissa aina epätoivottava ratkaisu, mutta valitettavasti ohjelmistoja on varsin vaikea rakentaa ilman, että välillä joudutaan tekemään ikäviltäkin tuntuvia kompromisseja.

Toinen tapa olisi vaiheistaa toiminnot hienojakoisemmin. Voisimme laatia esimerkiksi `PhasedAction`-nimisen rajapinnan, jolla `save`-operaation kaltainen monoliittinen toiminto jaettaisiin selkeisiin vaiheisiin esimerkiksi erottamalla vaiheet `validate`, `populate`, `persist` ja `createView`. Tällöin pääsynvalvontakomponenttia voitaisiin kutsua vasta ennen `persist`-vaihetta, jolloin kohteena oleva objekti olisi jo rakennettu valmiiksi.

Kolmas vaihtoehto olisi se, että pääsynvalvonta viedään kokonaan eri ohjelmistokerrokselle. Pääsynvalvontaa ei tällöin suoritettaisi kontrollerissa, vaan kontrollerin ja tietokannan välissä sijaitsevalla liiketoimintakerroksella. Esimerkkisovelluksen tapauksessa tällainen komponentti olisi `UserManager`, joka ennen tallennusta kutsuisi pääsynvalvontakomponenttia selvittääkseen, voiko kohteena olevaa objektia todella tallentaa tietokantaan vaiko ei. Välikerroksen ongelmana on valitettavasti se, että välikerros ei tiedä kontekstia, jossa tietoa käytetään; joskus sama tieto voi olla sallittua tietyssä kontekstissa, mutta kiellettyä toisessa. Jos kyseessä on esimerkiksi tiedon noutaminen tietokannasta, välikerros ei tiedä aiotaanko tietoalkio välittää aina käyttöliittymäkerrokselle asti vai käytetäänkö sitä esimerkiksi jonkinlaisessa statistiikkaoperaatiossa, jolloin yksittäistä tietoalkiota ei sellaisenaan paljasteta käyttöliittymätasolla, vaan sitä käytetään vain yhteenvetotiedon muodostamiseen.

Mikään näistä ratkaisuvaihtoehdoista ei ole täysin ongelmaton. Esimerkkisovelluksessa on valittu ensimmäinen ratkaisuvaihtoehto, mutta näkisin kuitenkin toimintojen vaiheistamisen olevan selkeämpi ratkaisu. Ratkaisun käyttöönotto kuitenkin vaatisi erilaisten web-sovellusten toimintojen laajamittaista analysointia ja luokittelua, jotta pystyisimme määrittämään sellaisen vaiheistuksen, jonka avulla suurin osa ongelmallisista käyttötapauksista pystyttäisiin esittämään. Pääsynvalvontalogiikan vientiä (yksinomaan) liiketoimintalogiikkakerrokselle en sen sijaan suosittelisi.

Pääsyn salliminen kohteena olevaan tietoon voi joskus riippua myös sellaisesta informaatiosta, jota ei suoraan ole saatavissa kohdeobjektin kautta. Esimerkiksi sovelluksemme `CalendarEntry`-luokasta ei ole suoraa yhteyttä kalenterimerkinnän osallistujatietoihin (tällainen yhteys toki voisi ja pitäisikin olla, mutta se on esimerkkisovelluksessa jätetty tietoisesti tekemättä asian havainnollistamiseksi), mutta pääsyn salliminen kalenterimerkintään voi kuitenkin riippua juuri siitä, onko käyttäjä osallistujana kyseisessä merkinnässä vaiko ei. Jotta sääntö voisi selvittää onko käyttäjä osallistujana, sääntökoneen

pitää päästä käsiksi myös informaatioon, joka ei ole saatavissa suoraan kohdeobjektin kautta.

Ongelma voidaan ratkaista ainakin kahdella tavalla. Ensinnäkin, voimme muuttaa `GuardContext`-oliota niin, että sen avulla voi välittää myös oheistietoa (lisäparametreja) toimintoluokasta sääntökoneelle. Tämä on kuitenkin epätoivottava ratkaisu, koska ideaalitilanteessa kohdetoiminnon (esimerkiksi `ShowEntryAction`-luokka) ei tulisi olla tietoinen siitä, minkälaisten sääntöjen avulla pääsynvalvontapäätös sääntökoneessa tehdään (sen ei itse asiassa tarvitse tietää edes sitä, että pääsynvalvontapäätös ylipäätään syntyy sääntökoneen avulla). Näin ollen kohteena oleva toiminto ei myöskään voi tietää sitä, minkälaisia lisätietoja säännöt tarvitsevat, joten tätä vaihtoehtoa ei kannata liiemmin edes harkita.

Toinen vaihtoehto on se, että sääntökone käy itse noutamassa tarvitsemansa lisätiedot. Tämä on epämieluisaa sikäli, että sääntökoneen tulisi normaaliolosuhteissa kyetä tekemään päätökset sen tietämyksen perusteella joka sille on syötetty, eikä joutua hankkimaan tietoja itse. Tämä on kuitenkin kompromissi johon monissa tilanteissa ajaudutaan, ja käyttötavan yleisyydestä kertonee jo sekin, että Droolsin dokumentaatiossa erikseen mainitaan juuri Hibernaten session välittäminen sääntökoneelle globaalin muuttujan avulla. Näin ollen päädyin esimerkkisovelluksessakin tällaiseen ratkaisuun, mutta en kylläkään välitä sääntökoneelle varsinaista tietokantakomponenttia, vaan tarjoan pääsyn tietokantaan välikomponentin (`ReasoningContext`) avulla. Ratkaisun etuna on se, että välikomponentti voi pitää hakemiaan tietoja välimuistirakenteessa, jolloin toinen samaa tietoa mahdollisesti tarvitseva sääntö saakin tiedon suoraan välikomponentilta, eikä joudu käymään tietokannassa asti. Jos evaluoitavia sääntöjä on paljon, tämä voi olla merkittävä tekijä järjestelmän suorituskyvyn kannalta.

Yllä kuvaamieni ongelmien lisäksi arkkitehtuurissa on myös sellainen puute, että se ei tällä hetkellä vielä mahdollista suoraan attribuuttitasolle menevää pääsynvalvontaa. Vaikka pääsemmekin pääsynvalvonnassa instanssitasolle, joskus olisi tarvetta pystyä kuvaamaan käyttöoikeuksia jopa tätäkin hienojakoisemmalla tavalla. Voisimme esimerkiksi määrittää, että henkilön tietoja saa katsella kuka tahansa, mutta henkilön sosiaaliturvatunnuksen (`Person.getSocialSecurityNumber()`) kysyminen ei olisikaan kaikille sallittua. Pystymme toki käyttämään tutkielmassa esitettyä `Transformer`-mekanismia tiedon esittämisen estämiseen, mutta hienompaa olisi päästä kiinni suoraan

attribuuttitasolle ja estää luottamukselliseksi katsottavien attribuuttien näkyminen esimerkiksi AOP-menetelmien avulla.

Toteutettu arkkitehtuuri mahdollistaa täysin deklaratiivisen pääsynvalvonnan soveltamisen, ja pääsynvalvontapolitiikka saadaan myös erinomaisesti kapseloitua. Tätä voi demonstroida muun muassa vaihtamalla esimerkkisovelluksen sisäänkirjautussivulta pääsynvalvontapolitiikaksi tiedoston `looseAccessControlRules.drl` sisältämän pääsynvalvontapolitiikan, jolloin sovelluksessa ei rajoiteta mitään. Tällainen mekanismi on erinomainen apuväline ohjelmistokehittäjälle, sillä arkkitehtuurin avulla ohjelmoija voi jättää pääsynvalvontapolitiikan pohtimisen kehitysvaiheessa vähemmälle, eikä jää jumiin miettiessään sitä, kellä pitäisi olla oikeus käyttää kehitettävänä olevaa toimintoa. Riittää, että ohjelmistokehittäjä merkitsee kehittämänsä toiminnot esimerkiksi käyttötapauskoodilla, jolloin pääsynvalvontapolitiikka voidaan laatia vasta myöhemmin.

Yhteenvetona arkkitehtuurin onnistuneisuudesta sanoisin, että arkkitehtuurin avulla web-sovellusten tietoturvatarpeet on mahdollista kattaa sillä tarkkuudella, kuin ennen tutkielman aloittamista oletinkin. Arkkitehtuurin avulla on varsin helppoa rakentaa tietoturvallisia ryhmätyösovelluksia, mutta tietyt vaiheistamiseen ja hienojakoisuuteen liittyvät epävarmuustekijät rajoittavat vielä sen soveltuvuutta aivan kaikkien toimintojen yhteydessä käytettäväksi. Näkisin kuitenkin, että toteutettu ratkaisu tarjoaa erinomaisen pohjan ryhmätyösovellusten pääsynvalvontamenetelmien jatkokehitykselle, ja sääntötiedostoon ulkoistetut pääsynvalvontasäännöt helpottavat merkittävästi myös ohjelmistokehittäjän työtä.

4.4.2. Sääntöjen esitystapa ja ilmaisuvoima

Olen hieman tyytymätön Droolsin käyttämään sääntöjen esitystapaan ja erityisesti siihen, millä tavalla säännöissä voidaan viitata kohdemaailman faktoihin. Oletin nimittäin niin, että sääntökone olisi hyvin suvaitsevainen esimerkiksi erilaisia merkintätapoja kohtaan, mutta osoittautui, että näin ei välttämättä ole. Esimerkiksi Velocityn sivupohjissa voisimme viitata henkilön ikään vaikkapa merkinnöillä `$person.age`, `${person.age}` tai `$person.getAge()`, mutta Drools:issa esimerkiksi ehtolauseke `person : Person (getAge() >= 65)` aiheuttaa poikkeuksen, mutta `person : Person (age >= 65)` ei. Monille Javan syntaksiin tottuneille tämä voi tulla yllätyksenä. Droolsin antamat virheilmoitukset eivät myöskään aina parhaalla mahdollisella tavalla kuvaa sitä, mistä virhe pohjimmiltaan johtuu.

Oletin myös, ettei säännöissä tarvitsisi tehdä eksplisiittisiä `null`-tarkistuksia, mutta myös tämä käsitys osoittautui virheelliseksi. Joissakin skriptikielissä tai esimerkiksi Velocityn sivupohjissa `null`-arvojen tarkistus erikseen ei ole tarpeellista, vaan esimerkiksi vertailu `$user.role.name == ”SUPERADMIN”` antaa tuloksen `false`, jos käyttäjään ei ole lainkaan kytketty roolitietoa. Droolsin tapauksessa vastaavasta ilmauksesta seuraa poikkeus, joka pakottaa sääntöjen laatijan huomioimaan tiedon puuttumisen aina erikseen. Tämä on toki hyvän ohjelmointitavan mukaista, mutta tekee sääntöjen laatimisesta jossain määrin työlästä.

Tämä ongelma kuitenkin poistuu käyttämällä arvoaluekohtaisia kieliä. Niiden avulla säännöt voidaan kirjoittaa luonnollisen kielen kaltaisesti, ja DRL-muotoisten sääntöjen sisältämä monimutkaisuus saadaan varsin täydellisesti peitettyä. Kun DRL-muotoinen sääntö onnistutaan kerran saattamaan toimintakuntoon, voidaan sääntö ”unohtaa” DSL-kuvaustiedostoon eikä säännön mahdollisesti sisältämiä monimutkaisia operaatioita tarvitse enää käyttää sellaisenaan.

Jos Drools ei tukisi arvoaluekohtaisia kieliä, en luultavasti käyttäisi sitä pääsynvalvonnan toteuttamiseen. Korvaisin Droolsin esimerkiksi Velocityllä tai jollakin skriptikielellä, jonka avulla säännöt pystyttäisiin esittämään dynaamisesti, mutta merkintätapa olisi Droolsin käyttämää tiiviimpi. Luonnollisen kaltaisen kielen käyttäminen kuitenkin kallistaa vaakakupin Droolsin kannalle, olkoonkin, että sääntöjen kuvaustapa ei ole erityisen kompakti.

Kuten aiemmin mainitsin, sääntökantojen sisältämien sääntöjen tulisi normaaliolosuhteissa olla mahdollisimman irrallisia ja riippumattomia toisistaan. Jos säännöt joudutaan kytketään vahvasti toisiinsa ja niiden suoritusjärjestys lyödään usein lukkoon, on tämä merkki siitä, että säännöt voitaisiin todennäköisesti esittää myös proseduraalisten menetelmien avulla. Esimerkkisovelluksessa säännöt suoritetaan monissa käyttötapauksessa tietyssä järjestyksessä, mikä saattaa olla indikaattori siitä, että deklaratiivista pääsynvalvontaa voitaisiin aivan hyvin harjoittaa myös ilman varsinaista sääntökonetta.

Uskon kuitenkin, että valtaosa havaitsemistani hankaluuksista johtuu yksinomaan siitä, että perehtymiseni Drools-sääntökoneeseen on aivan liian pintapuolinen. Ennen tutkielman aloittamista minulla ei ollut siihen

minkäänlaista kosketuspintaa, ja uskon toisaalta olevani myös ajatustasolla liian lukkiutunut proseduraaliseen ohjelmointimalliin voidakseni laatia säännöt parhaalla mahdollisella tavalla. Uskonkin, että käytön jatkuessa Drools osoittautuu erinomaiseksi välineeksi deklaratiivisen ohjelmointimallin soveltamiseen. Jo nyt olen tullut vakuuttuneeksi siitä, että Drools tarjoaa erittäin ilmaisuvoimaisen sääntökoneen, jonka avulla on mahdollista toteuttaa hyvinkin monimutkaisia liiketoimintasovelluksia tehokkaalla ja elegantilla tavalla.

4.4.3. Suorituskyky

Suorittamieni minimaalisten kuormitustestien perusteella Drools-sääntökoneen käyttämisellä ei tuntuisi olevan kovinkaan merkittävää vaikutusta järjestelmän kokonaissuorituskykyyn. Esimerkiksi UC_MANAGE_ORGANIZATIONS-käyttötapauksen toteuttavan `OrganizationManagementAction`-luokan `update`-metodin suorittaminen HTTP-rajapinnan yli kesti testiympäristössä 2.000 kutsun sarjassa keskimäärin 6 millisekuntia silloin, kun sääntökone oli käytössä, ja 3 millisekuntia silloin, kun sääntökone ohitettiin kokonaan. Suhteellisen vähäinen lisäkuormitus on johdonmukaista sikäli, että Droolsin tapauksessa säännöistä muodostetaan viime kädessä Java-tavukoodia, jonka suorittaminen ei eroa tavanomaisesta Java-ohjelmasta millään tavoin.

Tuloksien perusteella voidaan siis tehdä varovainen päätelmä, että Droolsin käyttäminen sinänsä ei ole suorituskyvyn kannalta olennainen asia; olennaista onkin se, millä tavoin sääntökonetta käytetään, ja millä tavoin säännöt määritellään. Säännöthän voivat esimerkiksi suorittaa jopa tietokantakyselyjä, joten sääntöpohjaisen ratkaisun tehokkuus on viime kädessä kiinni sääntöjen laatijan osaamisesta ja arkkitehtonisista kokonaisratkaisuista.

On kuitenkin syytä varoa tekemästä liian pitkälle meneviä johtopäätöksiä tuloksien pohjalta. Kohteena oleva esimerkkisovellus on erittäin pelkistetty, eikä testiympäristössä ollut syötettynä tuotantokäyttöä vastaavaa määrää dataa. Luotettavien tulosten saamiseksi olisikin ensi tilassa syytä laatia kymmeniä tuhansia käyttäjiä ja satoja tuhansia kalenterimerkintöjä sisältävä testiaineisto, mutta valitettavasti käytettävissä oleva aika ei tällaisen generointiin tutkielman yhteydessä riittänyt. Lisäksi on syytä tähdentää sitä, että tulokset pätevät ainoastaan Drools-sääntökoneen tapauksessa, eikä niiden pohjalta voi tehdä yleistyksiä muiden sääntökoneiden suorituskyvystä.

4.5. Valitun ratkaisumallin kritiikkiä

Kern ja Walhorn [2005, ss. 130-138] kritikoivat sääntöpohjaista pääsynvalvontaa muun muassa siitä, että sääntöpohjaisuus muuttaa käyttöoikeudet liiankin dynaamisiksi, jonka vuoksi on vaikea hahmottaa helposti sitä, kenellä on ajonaikainen käyttöoikeus johonkin järjestelmän toimintoon. Tämän vuoksi sääntöjen ylläpidosta tulee haasteellista, koska etukäteen on hyvin vaikea arvioida, minkälaisia kerrannaisvaikutuksia sääntöjen muuttamisella voi olla. He pitävät myös perinteistä roolipohjaista pääsynvalvontaa ongelmallisena siitä syystä, että suurissa järjestelmissä erillisten roolimääritysten lukumäärä voi kasvaa hyvinkin suureksi, jolloin käyttöoikeuksien hallinta käy roolipohjaisuudesta huolimatta työlääksi. He ehdottavat ongelman ratkaisuksi sääntöpohjaista roolien provisiointia (Rule-Based Provisioning of RBAC).

Kernin ja Walhornin mallissa yksittäisen sovelluksen pääsynvalvonta on täysin roolipohjainen. Rooleja ei normaalitilanteessa määritellä manuaalisesti (tämä on kylläkin mahdollista), vaan roolimääritykset tuotetaan useimmiten automatisoidusti identiteetinhallintajärjestelmään kytketyn sääntökoneen avulla. Sääntökone luo sovelluskohtaiset roolit keskitetysti sille määritettyjen sääntöjen mukaan, ja käyttää päättelyssä hyväkseen esimerkiksi henkilötietojärjestelmään syötettyä dataa. Kern ja Walhorn näkevät, että juuri henkilötietojärjestelmä on isoissa organisaatioissa se järjestelmä, jossa henkilöihin liittyvä tieto on parhaiten ajan tasalla. Siksi henkilötietojärjestelmää tulisi käyttää ensisijaisena lähteenä myös kohdejärjestelmien (Target Systems) käyttöoikeuksia myönnettäessä, joka siis tapahtuu roolien provisioinnin avulla.

Kern ja Walhorn esittelevät artikkelissa myös muutaman tapausesimerkin. Eräänä esimerkkinä on eurooppalainen pankki, jonka kahtakymmentäviittä tietojärjestelmää käyttää yhteensä 46.000 käyttäjää. Noin kahdeksankymmentä prosenttia kaikista kohdejärjestelmien roolimäärityksistä on automaattisesti tuotettu noin tuhannen sääntökoneelle määritetyn säännön perusteella, joten kaksikymmentä prosenttia roolimäärityksistä tehdään manuaalisesti. Sääntökone käyttää sääntöjä evaluoidessaan henkilötietojärjestelmästä noin viittätoista eri attribuuttia, joista artikkelissa mainitaan ainakin kustannuspaikka, organisaatioyksikkö ja henkilön työskentelypaikkakunta. Tällaisten attribuuttien perusteella siis päätellään mitä rooleja henkilölle myönnetään, ja nämä roolit provisioidaan kohteena oleviin tietojärjestelmiin. Kohdejärjestelmä toimii tällöin siis puhtaasti roolipohjaisten pääsynvalvontamenetelmien avulla, mitä Kern ja Walhorn pitävät hyvänä

asiana, koska tällöin henkilöiden oikeudet kohdejärjestelmässä ovat varsin eksplisiittiset ja ne on siksi helppo tarkistaa.

Itse koen Kernin ja Walhornin kritiikin osittain oikeutetuksi. On totta, että säännöt eivät ole yhtä eksplisiittinen tapa esittää käyttöoikeuksia kuin roolit. Mitä heidän mainitsemaansa roolimääritysten hallinnan työläyteen tulee, on syytä tähdentää, että Kernin ja Walhornin kuvailemat järjestelmät voivat sisältää jopa tuhansia erilaisia rooleja. Tässä kohtaa meidän tuleekin ottaa tarkastelun kohteeksi roolin *karkeuden* (granularity) käsite; Kern ja Walhorn näkevät roolin hyvin hienojakoisena käsitteenä, mutta esimerkiksi itse näen roolin selvästi karkeamman tason käsitteeksi. Jos tarkastelun kohteeksi otetaan vaikkapa pankin tietojärjestelmät, voisi pankin työntekijän liiketoiminnallinen rooli olla karkealla tasolla ”pankkitoimihenkilö”, mutta hienojakoisella tasolla ”Tampereen Osuuspankin Lielahden konttorin kassalla työskentelevä pankkitoimihenkilö”.

Itse en pidä tällaista äärimmäisyyteen asti vietyä hienojakoisuutta hyvänä asiana. Mielestäni henkilön liiketoiminnallinen rooli on syytä pitää ”puhtaana”, eli erillään liiketoiminnallisesta kontekstista (yritys, organisaatioyksikkö, sijainti), ja liiketoiminnallinen konteksti pitää mallintaa attribuutteina eikä ympätä sitä osaksi roolimäärityksiä. Olen luottamuksellisissa keskusteluissa kuullut puhuttavan muun muassa sellaista, että jonkin tietyn pankkikonttorin toimihenkilö ei ole voinut tarjota asiakkaalleen kaikkia tämän haluamia palveluita, koska asiakkaan oma tili on sattunut olemaan saman yrityksen naapurikonttorissa. Pankin omana intressinä olisi ollut mahdollisimman suuri yhteistyö konttorien välillä, mutta tietojärjestelmien joustamattomuuden takia on jouduttu tyytymään kompromisseihin. Nykyaikana liiketoiminnalliset vaatimukset muuttuvat niin tiheästi, että tietojärjestelmien on pystyttävä mukautumaan tällaisiin muutoksiin nykyistä paremmin.

On myös esitettävä kysymys, että mitä hyötyä saavutetaan hienojakoisilla roolimäärityksillä, jos niiden seurauksena järjestelmässä on tuhansia tai kymmeniäkin tuhansia eri rooleja? Eikö tilanne silloin ole sellainen, että aivan yhtä hyvin voitaisiin suoraan myöntää yksittäisille käyttäjille yksittäisiä käyttöoikeuksia, koska melkeinpä jokaisella käyttäjällä on järjestelmässä joka tapauksessa erilainen rooli kuin toisella. Tällöin rooli ei enää ryhmittele tai luokittele juuri mitään, eikä sen olemassaololle ole enää perusteita. Ja vaikka ylläpitotyötä saataisiinkin identiteetinhallinnan puolelta vähennettyä provisioimalla roolit käyttäjille Kernin ja Walhornin kuvaamalla

automatisoidulla tavalla, ei se suinkaan poista ylläpitotarvetta toisesta päästä – kohdejärjestelmistä. Nämä tuhannet roolit pitää kohdentaa kohdejärjestelmien yksittäisiin toimintoihin, eli analysoida kunkin toiminnon osalta, mitkä näistä rooleista ovat sellaisia, joilla kyseistä toimintoa pitäisi saada käyttää. Eikö olisi helpompaa korvata tällainen mittava roolilistaus yksinkertaisella sääntömäärityksellä?

Pidän lisäksi lähtökohtaisesti mahdottomana sitä, että muutaman henkilötietojärjestelmään tallennetun attribuutin perusteella pystyttäisiin kiistattomasti päättelemään henkilön rooli kussakin kohteena olevassa tietojärjestelmässä. Henkilön toimenkuva ja tarve käyttää jotakin tiettyä tietojärjestelmän ominaisuutta tunnetaan yleensä parhaiten paikallisesti (esimerkiksi organisaatioyksikön sisällä), joten on vaikea kuvitella miten automatisoidulla ja keskitetyllä ratkaisulla päästäisiin tässä parempiin tuloksiin.

On kuitenkin syytä huomioida, että aina kysymys ei olekaan siitä, mikä on jollekulle parasta. Melko usein kysymys on pohjimmiltaan siitä, mikä on kyseisen asiakasorganisaation kyky sietää riskiä käyttöoikeuksien hallinnan osalta. Onko tarkoituksena ”pelata varman päälle”, eli tarjota organisaation työntekijälle vain aivan välttämättömimmät toiminnot sen kustannuksella, että työntekijän tuottavuus ei organisaatiossa nouse sellaiselle tasolle kuin se voisi parhaimmillaan olla? Vai onko tarkoituksena tarjota työntekijälle ”avaimet käteen”, eli parhaat mahdolliset työkalut työn harjoittamiseen silläkin uhalla, että jossakin tietyssä tilanteessa käyttäjä saattaa päästä käsiksi toimintoon, johon hänen ei kenties pitäisi. Tämä riippuu varmasti hyvin pitkälti kohdealueesta, ja ymmärränkin hyvin roolipohjaisen pääsynvalvonnan tai pakollisen pääsynvalvonnan soveltamisen esimerkiksi juuri pankkitietojärjestelmissä tai potilastietojärjestelmissä. Tavanomaisissa ryhmätyösovelluksissa (joissa ei liiku rahaa tai arkaluonteista henkilötietoa) näkisin kuitenkin sääntöpohjaisuuden olevan varsin käyttökelpoinen menetelmä.

5. Yhteenveto

Tutkielmassa on tarkasteltu pääsynvalvonnan merkitystä nykyaikaisten web-sovellusten arkkitehtuurissa sekä esitelty joitakin perinteitä menetelmiä, joita pääsynvalvontaan on tietojärjestelmissä yleisesti sovellettu. Erityisesti on tutkittu Javalla toteutettujen web-sovellusten pääsynvalvontaan liittyviä ratkaisuja ja tarkasteltu, miten deklaratiivinen ohjelmointiparadigma voisi soveltua kohdealueen ongelmien ratkaisemiseen. Tutkielmassa on esitelty karkealla tasolla Java 2 -ympäristön valmiina tarjoamat ratkaisut ja kuvattu niiden puutteita erityisesti web-ohjelmistojen ja ryhmätyösovellusten osalta.

Tutkielmassa on myös luotu yleiskatsaus deklaratiivisiin ohjelmointimenetelmiin sekä sääntökoneisiin, ja pohdittu milloin deklaratiivisia ohjelmointimenetelmiä ja sääntökoneita kannattaisi hyödyntää. Konkreettisena sääntökonetoteutuksena on esitelty Drools-sääntökone, jonka avulla on toteutettu pääsynvalvonta yksinkertaiseen ryhmäkalenterisovellukseen. Tämän jälkeen esimerkkisovelluksessa käytettävät menetelmät on dokumentoitu.

Tutkielman yhteydessä havaittiin, että deklaratiivinen ohjelmointiparadigma ja Drools-sääntökone soveltuvat varsin mainiosti web-sovellusten pääsynvalvonnan toteuttamiseen. Niiden avulla pääsynvalvonta on mahdollista toteuttaa kattavasti ja tehokkaasti, eikä pääsynvalvontalogiikka pirstaloidu osaksi staattista ohjelmakoodia ja käyttöliittymäkerrosta niin pahasti kuin perinteisillä pääsynvalvontamenetelmillä. Samalla kuitenkin havaittiin, että sääntöjen valtavan ilmaisuvoiman kääntöpuolena on tietynlainen läpinäkymättömyys, jolloin ei enää ole yksiselitteisesti ja nopeasti nähtävissä, kuka mitäkin järjestelmän toimintoa saa tietyssä tilanteessa käyttää. Tämän jälkeen arvioitiin vielä ratkaisua, jossa sääntöpohjainen pääsynvalvonta ja roolipohjainen pääsynvalvonta on yhdistetty, ja pohdittiin olisiko tämä lähestymistapa puhtaasti sääntöpohjaista ratkaisua parempi tapa pääsynvalvonnan toteuttamiseen.

5.1. Jatkotutkimusideoita

Tutkielman rajaamisen vuoksi eräitä aihealueeseen liittyviä asioita on jouduttu jättämään tutkielmassa käsittelemättä. Lisätutkimusta vaadittaisiin muun muassa attribuuttitasolle menevän pääsynvalvonnan toteuttamisesta, jonka avulla instanssitasolle menevää pääsynvalvontaa voitaisiin edelleen tarkentaa. Ensinnäkin olisi tarpeen kehittää menetelmiä attribuuttitasolle menevän

pääsynvalvonnan sääntöjen ja käyttölupien kuvaamiseen, ja toiseksi toteuttaa menetelmiä tällaisen pääsynvalvontatavan toimeenpanoon. Uskon, että AOP-menetelmillä olisi tässä oma sijansa. Lisäksi toteutusta pitäisi laajentaa mahdollistamaan myös negatiiviset käyttöluvat, joilla tietyn tiedon esittäminen voitaisiin eksplisiittisesti kieltää tietyltä toimijalta. Tällä hetkellä esitetyssä ratkaisussa ei ole tukea tällaisille ominaisuuksille.

Eräs mielenkiintoinen jatkotutkimusaihe olisi XACML (eXtensible Access Control Markup Language) [XACML, 2003]. XACML määrittelee menetelmiä pääsynvalvontapolitiikkojen kuvaamiseksi XML-esitystavan avulla, ja olisikin hyödyllistä tutkia mitä uusia ominaisuuksia erityisesti tällä hetkellä kehitteillä oleva standardin versio 3.0 tuo pääsynvalvontamenetelmiin. Erityisesti olisi tarpeen tutkia, soveltuisiko XACML sellaisenaan juuri web-sovellusten pääsynvalvontasääntöjen esittämiseen, tai olisiko sen käyttämistä esitystavoista hyötyä esimerkiksi eri sääntökoneiden yhteensopivuuden parantamisessa.

Lisäksi olisi tarpeen tarkastella, miten JACC (Java Authorization Contract for Containers) [JACC, 2008] istuu pääsynvalvonnan kokonaiskuvaan. Se laajentaa J2EE-sovellusten pääsynvalvontaa viemällä pääsynvalvontaan liittyvää toiminnallisuutta sovelluksista sovelluspalvelimien vastuulle, ja esittelee uusia lupatyyppejä (esimerkiksi `javax.security.jacc.WebUserDataPermission`) web-sovellusten tarpeisiin. Nähdäkseni JACC perustuu kuitenkin valtaosin siihen, että erityyppiset resurssit identifioidaan niiden URI:n perusteella, joten en usko, että JACC:ista on merkittävää apua perinteisten tietokantapohjaisten web-sovellusten pääsynvalvontaan. Tämä olisi kuitenkin syytä selvittää.

Olisi myös hyödyllistä tutkia sitä, millä tavoin sääntöpohjainen pääsynvalvonta olisi sovitettavissa Spring Security -ohjelmistokehykseen [Spring Security, 2008]. Spring Security (tunnetaan myös nimellä Acegi Security) sisältää pääsynvalvontalistoihin perustuvia piirteitä instanssitason pääsynvalvonnan toteuttamiseen, ja vireillä on ollut myös keskusteluja siitä, että pääsynvalvontamekanismia voitaisiin laajentaa sääntöpohjaisilla ominaisuuksilla. Spring-sovelluskehys on saavuttanut viimeisten vuosien aikana hyvin laajaa suosiota Javalla toteutettujen ohjelmistokehysten alueella, ja sen arkkitehtuuri on varsin modulaarinen mahdollistaen Springin osakomponenttien käyttämisen myös muissa asiayhteyksissä. Näin ollen Spring Securityn pääsynvalvontaan tehdyt laajennokset olisivat käyttökelpoisia myös silloin, kun Springin muita osia (esimerkiksi Spring MVC) ei käytettäisi.

Olisi myös mielenkiintoista kokeilla sitä, miten hyvin Prolog soveltuisi pääsynvalvonnan sääntöjen esittämiseen. Prologin prosessointia on perinteisesti pidetty varsin raskaana, mutta kaikki järjestelmät eivät suinkaan sisällä kymmeniätuhansia tai satojatuhansia käsiteltäviä tietoalkioita. Siksi olisi mielenkiintoista kokeilla minkälaiseen suorituskykyyn Javalle käännetyt Prolog-toteutukset pystyisivät, ja olisiko niistä hyötyä pääsynvalvonnan toimeenpanossa. Jos pääsynvalvontakomponentin käyttämä sääntökone abstrahoitaisiin täysin rajapintojen taakse, voitaisiin Prologia käyttää esimerkiksi sellaisissa järjestelmissä joissa käsiteltävä tietomäärä on rajallinen, ja vaihtaa pääsynvalvonnan toimeenpaneva toteutus esimerkiksi Drools-pohjaiseen toteutukseen silloin kun datamäärä kasvaa.

Viiteluettelo

[Alur *et al*., 2001] Deepak Alur, John Crupi and Dan Malks, *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2001.

[Atk-sanakirja, 2004] *Atk-sanakirja*. Talentum, Helsinki, 2004.

[Elmasri and Navathe, 1994] Ramez Elmasri and Shamkant B. Navathe, *Fundamentals of Database Systems, Second Edition.* Addison Wesley, 1994.

[Gamma *et al*., 1995] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns*. Addison-Wesley, 1995.

[Gong, 2002] Li Gong. Java™ 2 Platform Security Architecture, Version 1.2. Available as *http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html.*

[Grand, 2002] Mark Grand, *Java Enterprise Design Patterns.* John Wiley & Sons, Inc, 2002.

[JAAS, 2008] JAAS Reference Guide. Available as *http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html.*

[JACC, 2008] Java Authorization Contract for Containers. Available as *http://java.sun.com/j2ee/javaacc/index.html*.

[Jaworski and Perrone, 2000] Jamie Jaworski and Paul Perrone, *Java Security Handbook*. SAMS Publishing, 2000.

[Kern and Walhorn, 2005] Axel Kern and Claudia Walhorn. Rule Support for Role-Based Access Control. In *Proceedings of the tenth ACM symposium on Access Control Models and Technologies*, ACM Press, 2005.

[NSA, 2008] Security-Enhanced Linux. Available as *http://www.nsa.gov/selinux/.*

[Proctor *et al*., 2008] Mark Proctor, Michael Neale, Michael Frandsen, Sam Griffith Jr., Edson Tirelli, Fernando Meyer and Kris Verlaenen. Drools User Guide. Available as *http://downloads.jboss.com/drools/docs/4.0.4.17825.GA/html_single/index.html.*

[Rjaibi and Bird, 2004] A Multi-Purpose Implementation of Mandatory Access Control in Relational Database Management Systems. In *Proc. 30th Very Large Data Bases Conference (VLDB '04)*, August 2004. Available as *http://www.cs.toronto.edu/vldb04/protected/eProceedings/contents/pdf/IND1P3.PDF.*

[Samar and Schemers, 1995] Unified login with pluggable authentication modules (PAM). Open Software Foundation RFC 86.0. Available as *http://www.opengroup.org/rfc/rfc86.0.html.*

[Simplecal, 2008] Simplecal-esimerkkisovelluksen projektisivut. Saatavana osoitteesta *http://www.slite.fi/simplecal-project/.*

[Spring Security, 2008] Spring Security Project Home Page. Available as *http://www.acegisecurity.org*.

[Struts, 2008] The Apache Struts Project Home Page. Available as *http://struts.apache.org.*

[Velocity, 2008] The Apache Velocity Project Home Page. Available as *http://velocity.apache.org.*

[Wikipedia, 2008a] Wikipedia, The Free Encyclopedia. Web 2.0. Available as *http://en.wikipedia.org/wiki/Web_2.0*.

[Wikipedia, 2008b] Wikipedia, The Free Encyclopedia. Software as a service. Available as *http://en.wikipedia.org/wiki/Software_as_a_Service*.

[Wikipedia, 2008c] Wikipedia, The Free Encyclopedia. Rete algorithm. Available as *http://en.wikipedia.org/wiki/Rete_algorithm*.

[XACML, 2003] eXtensible Access Control Markup Language (XACML) Version 1.0. Available as *http://www.oasis-open.org/committees/download.php/2406/oasis-xacml-1.0.pdf.*

[XDoclet, 2008] XDoclet Project Home Page. Available as *http://xdoclet.sourceforge.net/xdoclet/index.html*.

Huom. Kaikki WWW-osoitteet tarkistettu 7.5.2008.