TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

TOPI OJALA

# REAKTIIVINEN SÄÄNTÖOHJATTU MONITOROINTIJÄRJESTELMÄ HAJAUTETTUUN YMPÄRISTÖÖN

Diplomityö

Tarkastaja: Tommi Mikkonen
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekunta-neuvoston kokouksessa 12. elokuuta 2015

# TIIVISTELMÄ

**TOPI OJALA**: Reaktiivinen sääntöohjattu monitorointijärjestelmä hajautettuun ympäristöön
Tampereen teknillinen yliopisto
Diplomityö, 43 sivua
Lokakuu 2015
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Tommi Mikkonen

Avainsanat: monitorointi, hajautetut järjestelmät, reaktiivinen suunnittelu

*Monitorointi* on sen kohteen tilan seuraamista, tilan vaihtumisen huomaamista ja siihen reagointia. Nykyisin monitoroinnin toteutus ohjelmistona on luontevaa, koska usein tarkkailtava kohde tai kohteen tilatietoja tuottavat sensorit ovat myös ohjelmia. Monitorointijärjestelmän toteutuksessa ohjelmistona voidaan hyödyntää hyväksi todettuja tekniikoita joita ovat mm. toiminta tapahtumakeskeisesti eli *reaktiivisuus,* konfiguroitavuutta edesauttava *sääntöohjaus* –malli, monta-yhteen suhdetta tukeva *julkaisija-tilaaja* –viestintämalli ja ohjelmien yhteistoiminnan rakentamiseen tarvittavia *integraatiokomponentteja.*

Eatechin asiakkaalla oli tarve automatisoida hajautetun ympäristön laitteiden reaaliaikainen monitorointi. Asiakas oli yrittänyt automaattisen monitoroinnin toteutusta jo aikaisemmin, mutta ratkaisussa ilmeni ongelmia ja sen käytöstä oli luovuttu. Asiakkaan ongelma oli viantunnistuskriteerien mallinnus automaattisessa monitoroinnissa. Vikatilanteiden tunnistuskriteerit ovat asiantuntijoiden määrittelemiä kaavoja, jotka on löydetty kokemuksen myötä. Ongelmaa lähdettiin ratkomaan tarpeiden perusteella. Ensimmäisessä vaiheessa määrittelimme järjestelmän laatuvaatimukset sekä keräsimme esimerkkejä vikatilanteiden tunnistuskriteereistä. Laatuvaatimukset ohjasivat järjestelmän toimintaperiaatteen lähtökohtaisesti tapahtumakeskeiseksi, eli reaktiiviseksi. Monitorointitarpeiden vaihtelevuuden sekä heterogeenisen toimintaympäristön vuoksi, järjestelmä rakennettiin tuottamaan hälytyksiä sääntöohjautuvasti. Seuraavaksi suunnittelimme yhtenäisen ja sopivan tavan kuvata vikatilanteita, jota käytimme pohjana järjestelmän suunnittelussa. Toteutukseen valitsimme tarpeisiin ja laatuvaatimuksiin osuvimmat teknologiat.

Tässä diplomityössä esitelty ratkaisu tunnistaa vikatilanteet kriteerien perusteella reaaliaikaisesti. Kun heterogeeninen tuotantoympäristö toimii odottamattomasti, reaktiivisen järjestelmän luotettavuus heikentyy, koska järjestelmän tilasiirtymien tulee varautua myös odottamattomiin syötteisiin. Järjestelmän tilasiirtymälogiikoita voitiin kuitenkin hienosäätää huomioimaan ympäristön kriittisimmät ominaisuudet, jonka jälkeen monitorointiratkaisut toimivat lähes kaikissa tapauksissa.

## ABSTRACT

**TOPI OJALA**: Reactive rule-based monitoring system for distributed environment
Tampere University of Technology
Master of Science Thesis, 43 pages
October 2015
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Professor Tommi Mikkonen

Keywords: monitoring, distributed systems, reactive design

*Monitoring* can be decsribed as tracking the state of its target, detecting a change of state and reacting to the state change. Implementing monitoring using a software is convenient since often the monitored targets, or sensors producing data of the monitoring targets, are software as well. When designing a monitoring software the following patterns are often on the table: *reactive* event-oriented design, *rule-based* design, which is advantageous in configurability, one-to-many relationship messaging pattern *publish-subscribe,* and *integration components* used in building functionality between interconnected software.

Eatech's customer initiated the development of a software system with the intention to automate real-time monitoring of devices within a distributed environment. The customer had attempted an automated monitoring system before, but the implementation was not without issues, and it was deactivated from production use. The customer had a problem applying complex fault recognition patterns to an automated monitoring system. These patterns have been detected by experts, based on their long history of manual monitoring.

Solving the problem began by finding out the real needs. First, we defined the quality requirements needed for such system and collected examples of fault recognition patterns in the production environment. Real-time requirement guided the system design towards reactive design, varying use cases and extensibility requirements towards rule-based design. Next, we defined a way to coherently model these patterns, which we used as a frame for architectural design. Finally for implementation, we picked the technologies most suited for our needs and quality requirements.

The solution presented in this thesis effectively monitors devices and recognizes fault situations. The reliability of a reactive system within a heterogenic environment is compromised when the environment behaves unexpectedly, and every possible input scenario should be accounted for. After fine-tuning the system's state transfer logic to take account the environment's most critical characteristics, the monitoring solutions work as expected in almost all cases.

# ALKUSANAT

Haluan kiittää tämän työn mahdollistamisesta ja valmistumisesta työn tarkastajaa prof. Tommi Mikkosta, Eatech Oyta ja esimiestäni Antti Kopposta, isääni Jormaa sekä avovaimoani Tiinaa.

Tampereella 23.8.2015

Topi Ojala

# SISÄLLYSLUETTELO

1. JOHDANTO ........ 1
2. MONITOROINTI JA SEN TOTEUTUSTEKNIIKAT ........ 2
 2.1 Monitorointi ........ 2
 2.2 Reaktiivisuus ........ 3
 2.3 Sääntöohjaus ........ 5
 2.4 Julkaisija-tilaaja ........ 5
 2.5 Integraatiokomponentit ........ 6
3. TOTEUTETTAVA JÄRJESTELMÄ ........ 7
 3.1 Toimintaympäristö ........ 7
 3.2 Valmiit ratkaisut ........ 8
 3.3 Järjestelmän laatuvaatimukset ........ 9
  3.3.1 Reaaliaikaisuus ........ 9
  3.3.2 Konfiguroitavuus ........ 9
  3.3.3 Laajennettavuus ........ 10
  3.3.4 Skaalautuvuus ........ 10
  3.3.5 Vikasietoisuus ........ 10
  3.3.6 Luotettavuus ........ 11
4. SUUNNITTELU ........ 12
 4.1 Määrittely ........ 12
 4.2 Käsitteet ........ 13
  4.2.1 Monitoroitava (yksikkö) ........ 13
  4.2.2 Kohde ........ 14
  4.2.3 Sanoma ........ 14
  4.2.4 Sääntö ........ 15
  4.2.5 Hälytys ........ 17
 4.3 Arkkitehtuuri ........ 18
 4.4 Tietokantaskeema ........ 19
 4.5 Luokkasuunnittelu ........ 21
  4.5.1 AlarmEngine –moduuli ........ 21
  4.5.2 Interface -moduuli ........ 23
  4.5.3 Communication -moduuli ........ 23
 4.6 Rajapinta ulkoisille tapahtumalähteille ........ 24
 4.7 Konfiguraatioiden päivitys ........ 26
5. KÄYTTÖTAPAUKSET ........ 28
 5.1 Laite Offline ........ 28
 5.2 Jakson täsmäytys ........ 31
 5.3 Käyttäjän tunnistus ........ 32
 5.4 Apusäännöt ........ 33
6. TOTEUTUS JA ARVIOINTI ........ 34
 6.1 Teknologiavalinnat ........ 34

6.2 Suorituskyvyn profilointi ... 35
6.3 Yksikkötestaus ja ajan hallinta ... 37
6.4 Järjestelmän isännöinti palveluna ... 39
6.5 Jatkokehitysideat ... 41
6.6 Arviointi ... 41
7. YHTEENVETO ... 43
LÄHDELUETTELO ... 44

# KUVALUETTELO


***Kuva 1.*** *Monitorointi kohteiden ja ylläpidon välillä ............ 2*
***Kuva 2.*** *Vedenkeitin mallinnettuna äärellisenä tilakoneena ............ 4*
***Kuva 3.*** *Synkroninen ja asynkroninen kommunikaatio, Hohpe & Woolfe [11] ............ 4*
***Kuva 4.*** *Julkaisija-tilaaja suunnittelumalli ............ 6*
***Kuva 5.*** *Toimintaympäristön tietoliikenneverkot ............ 8*
***Kuva 6.*** *AlarmSystem käsitekaavio ............ 13*
***Kuva 7.*** *Monitoroitava puurakenne ............ 14*
***Kuva 8.*** *Sanomien muodostus ............ 15*
***Kuva 9.*** *Tilakaavio sääntöjen toimintalogiikasta ............ 16*
***Kuva 10.*** *Sääntöjen abstrakti kantaluokka ja periytyminen ............ 17*
***Kuva 11.*** *Vuokaavio hälytysten toimintalogiikasta ............ 18*
***Kuva 12.*** *AlarmSystem arkkitehtuuri ............ 19*
***Kuva 13.*** *Vuokaavio AlarmSystemin moduulien yhteistoiminnasta ............ 20*
***Kuva 14.*** *AlarmSystem tietokantaskeema ............ 21*
***Kuva 15.*** *AlarmSystem.AlarmEngine moduulin luokkakaavio ............ 22*
***Kuva 16.*** *AlarmSystem.Interface moduulin luokkakaavio ............ 23*
***Kuva 17.*** *AlarmSystem.Communication moduulin luokkakaavio ............ 24*
***Kuva 18.*** *Ulkoisten tapahtumalähteiden integrointi ............ 25*
***Kuva 19.*** *ValueProvider luokan rajapinta ja assosiaatiot ............ 27*
***Kuva 20.*** *Tilakonekaavio Laite Offline Säännön 1 tilasiirtymälogiikalle. ............ 29*
***Kuva 21.*** *Tilakonekaavio Laite Offline Säännön 2 tilasiirtymälogiikalle. ............ 30*
***Kuva 22.*** *Tilakonekaavio Jakson täsmäytys –säännön tilasiirtymälogiikalle ............ 31*
***Kuva 23.*** *Tilakonekaavio Käyttäjän tunnistus –säännön tilasiirtymälogiikalle ............ 32*
***Kuva 24.*** *MessageConverter CPU-ajan profilointi ............ 36*
***Kuva 25.*** *AlarmEngine.Rules, CPU-ajan profilointi ............ 37*
***Kuva 26.*** *WCF-kommunikaatio palvelun ja asiakkaan välillä ............ 41*

# LYHENTEET JA MERKINNÄT

| | |
|---|---|
| CORBA | Common Object Request Broker Architecture. Object Management Groupin määrittelemä arkkitehtuuri, jonka avulla sovellukset kommunikoivat verkon yli. [1] |
| CRUD | Create-Read-Update-Delete. Operaatiot, jolla voidaan hallita resursseja. |
| DTO | Data Transfer Object. Ilmentymä luokasta, jota käytetään vain tiedonsiirtoon. |
| Endpoint | Palvelun tarjoama liityntä. Yleensä URI, joka määrittelee protokollan, verkkosijainnin ja portin. |
| Fire-and-forget | Yksisuuntainen kommunikaatio, ei odoteta vastausta. |
| FTP | File Transfer Protocol. Sovellusprotokolla tiedostojen siirtoon. |
| Forward-only | Vain eteenpäin suuntaava, ei mahdollisuutta palata taaksepäin aloittamatta alusta. |
| Full duplex | Kaksisuuntainen kommunikaatio. |
| HTTP | Hyper-Text Transfer Protocol. Sovellusprotokolla tekstin ja dokumenttien siirtoon. |
| Konfiguroitavuus | Suure, joka arvioi komponentin säädettävyyttä parametrisoinnilla. |
| Kätkö | Varasto välimuistissa, säilyttää resursseja joiden uudelleenhankinta on yleensä kallista. |
| LINQ | Language Integrated Query. .NET Frameworkin kirjasto, joka mahdollistaa SQL-tyylisiä kyselyjä tietorakenteisiin natiivisti. [2] |
| TCP/IP | Transport Control/Internet Protocol. Siirtoprotokollaperhe laitteiden väliseen kommunikointiin verkon ylitse. |
| ORM | Object-Relational Mapping. Siltaus olioista tietokantaan ja takaisin. |
| Overhead | Taakka. Viestinnän vaatima data, joka ei ole liiketoiminnan kannalta merkittävää. |
| RDBMS | Relational database management system. Tietokantahallintajärjestelmä joka perustuu relaatiotietokantamalliin. |
| Rx | Reactive Extensions. Kirjasto, joka tarjoaa muun muassa julkaisija -tilaaja mallin toteutuksen. |
| SOAP | Simple Object Access Protocol. XML-pohjainen sovellusprotokolla sovellusten väliseen kommunikaatioon. |
| UML | Unified Modeling Language. Ohjelmistosuunnittelussa käytetty visuaalinen kieli, jolla voidaan mallintaa suunnitelun osa-alueita. |
| URI | Uniform Resource Identifier. Tunnisteskeema resursseille. |
| WCF | Windows Communication Foundation, Microsoft .NET sovelluskehyksen tarjoama teknologia palvelujen välisten kommunikaation rakentamiseen. |
| Wildcard | Merkkijonovertailussa käytetty merkki, joka edustaa mitä tahansa merkkiä. |
| Worst case | Skenaario, joka kuvastaa järjestelmän tehottominta tilannetta. |
| WSDL | Web Service Description Language. XML-pohjainen kuvaus verkkopalvelun rajapinnasta. |

# 1. JOHDANTO

*Monitorointi* on sen kohteen tilan seuraamista, tilan vaihtumisen huomaamista ja siihen reagointia. Nykyisin monitoroinnin toteutus ohjelmistona on luontevaa, koska usein tarkkailtava kohde tai kohteen tilatietoja tuottavat sensorit ovat myös ohjelmia. Monitorointijärjestelmän toteutuksessa ohjelmistona voidaan hyödyntää hyväksi todettuja tekniikoita joita ovat mm. toiminta tapahtumakeskeisesti eli *reaktiivisuus,* konfiguroitavuutta edesauttava *sääntöohjaus* –malli, monta yhteen -suhdetta tukeva *julkaisija-tilaaja* –viestintämalli ja ohjelmien yhteistoiminnan rakentamiseen tarvittavia *integraatiokomponentteja.*

Eatechin asiakkaalla oli tarve automatisoida hajautetun ympäristön laitteiden reaaliaikainen monitorointi. Monitorointia toteuttaessa on erittäin tärkeää omata kattava ymmärrys monitorointijärjestelmän toimintaympäristöstä. Tästä syystä Eatech oli osuva valinta järjestelmän toteuttajaksi; yritys omaa pitkän kokemuksen hajautettujen järjestelmien hallinnasta sekä laiteohjelmistojen ylläpidosta ja monitoroinnista. Lisäksi, jatkuvan asiakkuuden myötä yritys pystyy myös tarjoamaan järjestelmän ylläpidon. Asiakkaan ongelma oli ei-triviaalien viantunnistuskriteerien mallinnus automaattisessa monitoroinnissa. Vikatilanteiden tunnistuskriteerit ovat asiantuntijoiden määrittelemiä kaavoja, jotka on löydetty kokemuksen myötä. Asiakas oli yrittänyt automaattisen monitoroinnin toteutusta jo aikaisemmin, mutta ratkaisussa ilmeni ongelmia ja sen käytöstä oli luovuttu. Ongelmaa lähdettiin ratkomaan tarpeiden perusteella. Aluksi määriteltiin järjestelmän laatuvaatimukset sekä kerättiin esimerkkejä vikatilanteiden tunnistuskriteereistä. Laatuvaatimukset ohjasivat järjestelmän toimintaperiaatteen lähtökohtaisesti *reaktiiviseksi*. Monitorointitarpeiden vaihtelevuuden sekä heterogeenisen toimintaympäristön vuoksi, järjestelmä rakennettiin tuottamaan hälytyksiä *sääntöohjautuvasti*. Järjestelmän sisäisen viestinvälityksen toteutuksessa hyödynnettiin *julkaisija-tilaaja* -mallia. Tilatietoja tuottavat sensorit yhdistettiin järjestelmään valitsemalla oikeat *integraatiokomponentit.* Seuraavaksi luotiin yhtenäinen sanasto sekä tapa kuvata vikatilanteita, joita käytettiin pohjana järjestelmän suunnittelussa. Toteutukseen valittiin tarpeisiin ja laatuvaatimuksiin osuvimmat teknologiat ja työkalut.

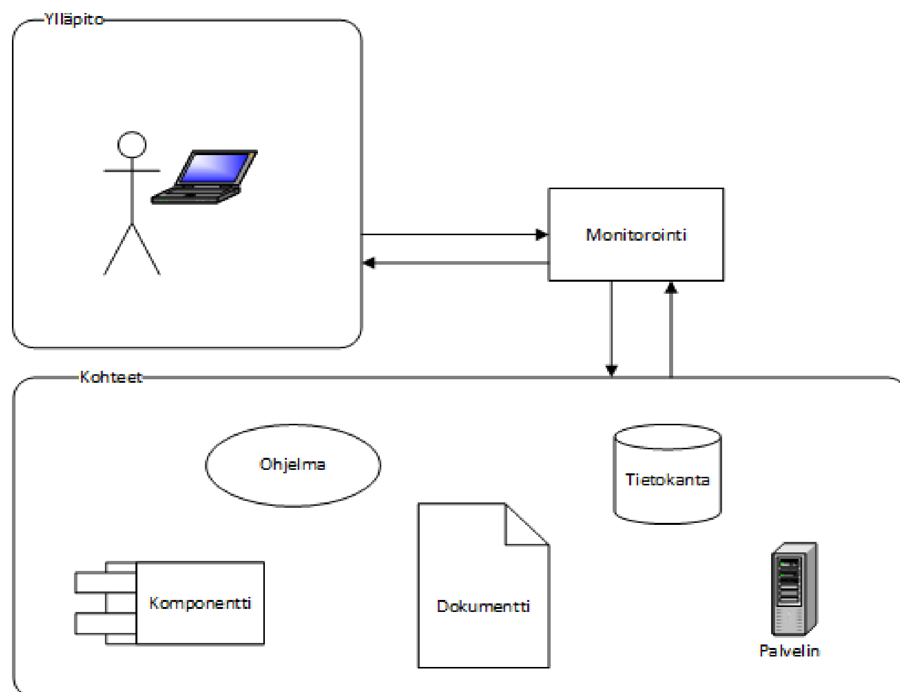
Tämän diplomityön luvussa 2 perehdytään monitoroinnin ja sen toteutuksen teoriaan. Luvussa 3 esitellään järjestelmän toimintaympäristö, punnitut ratkaisuvaihtoehdot ja sille asetetut laatuvaatimukset ja. Luvussa 4 kuvataan järjestelmän määrittelyä ja suunnittelua. Luvussa 5 esitellään esitutkimusvaiheen käyttötapauksia sovellettuna järjestelmään. Luku 6 käsittelee järjestelmän teknologiavalintoja, profilointia ja testausta, palveluisännöintiä sekä sisältää arviointia toteutetun järjestelmän ratkaisuista. Luku 7 on yhteenveto työssä käsitellyistä asioista.

# 2. MONITOROINTI JA SEN TOTEUTUSTEKNIIKAT

Tässä luvussa käsitellään monitorointia sekä sen suunnittelussa ja toteutuksessa käytettyjä tekniikoita. Kohdassa 2.1 määritellään monitorointi terminä ja mitä se nykypäivänä tarkoittaa. Kohdassa 2.2 käsitellään reaktiivista tapahtumakeskeisestä mallia ja mitä ominaisuuksia se tuo mukanaan. Kohdassa 2.3 esitellään viestinvälitykseen suunnattua julkaisija-tilaaja mallia. Kohdassa 2.4 käsitellään yleensä tekoälyn toteutuksessa käytettyä sääntöohjaus mallia ja miten sitä voidaan soveltaa monitorointijärjestelmän toteutukseen. Kohdassa 2.5. esitellään integraatiokomponentit ja pohditaan niiden valintaa.

## 2.1 Monitorointi

Monitoroinnilla tarkoitetaan sen kohteen tilan seuraamista. Monitoroinnin tarkoitus on saavuttaa ja ylläpitää tietoisuus monitoroitavan kohteen tilasta. Monitoroinnin kohteina ovat tyypillisesti järjestelmät, joiden tila voi muuttua ei-toivottuun tilaan. Tilamuutos ei-toivottuun tilaan pitää tunnistaa monitoroinnin keinoin, jotta siihen voidaan reagoida (Kuva 1).




**Kuva 1.** *Monitorointi kohteiden ja ylläpidon välillä*

Nykyaikainen monitorointi toteutetaan usein ohjelmistoina. Monitoroitavat kohteet tai dataa tuottavat sensorit ovat yleensä itsekin ohjelmistoja, data on useimmiten ohjelmallisesti käsiteltävässä muodossa sekä datan välitys monitorointijärjestelmille on integroitavissa ohjelmistotekniikan menetelmin, esimerkiksi rajapintojen ja verkkotekniikoiden avulla. Monitorointi on tyypillisesti jatkuva prosessi, joten sen toteutus ohjelmistona on luontevaa; ohjelmistot ovat hyvä ratkaisu automatisoida jokin toistuva toiminto.

Jatkuva monitorointi voi perustua joko ajastettuihin tarkastuksiin tai tapahtumien reaaliaikaiseen seurantaan. Ajastetun tarkastuksen mallissa määritellään ajoittain suoritettava tarkastuslogiikka, jonka perusteella vika tunnistetaan. Mallin hyvät puolet ovat toteutuksen yksinkertaisuus ja *robustisuus*. Mallin huonot puolet ovat virheentunnistuksen *worst case* viivästyminen sekä huono skaalautuvuus. Tapahtumin reagoimista niiden ilmetessä kutsutaan *reaktiivisuudeksi*, jota käsitellään seuraavassa kohdassa.

## 2.2 Reaktiivisuus

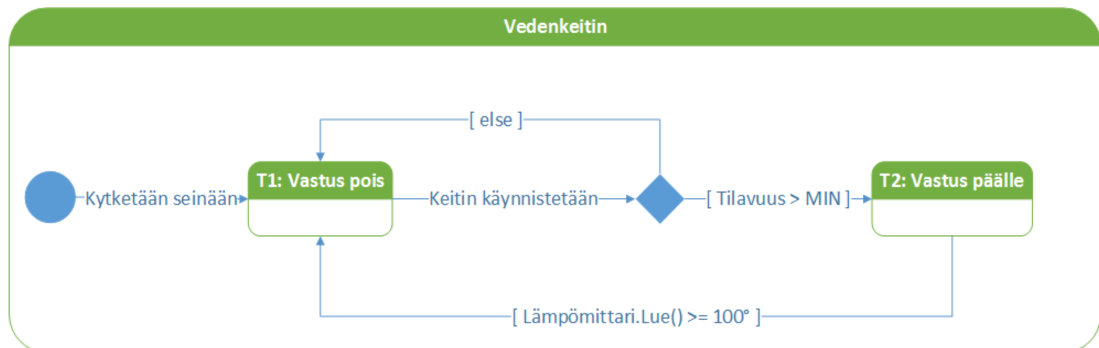
Reaaliaikaista monitorointijärjestelmää suunnitellessa voidaan hyödyntää reaktiivisuuden mallia. Järjestelmää, jonka toimintaa ohjaavat järjestelmän sisäiset ja ulkopuoliset tapahtumat, kutsutaan reaktiiviseksi [7]. Manna ja Pnuelli määrittelevät reaktiivisen ohjelman sellaiseksi, jonka rooli on ylläpitää jatkuva kanssakäyminen ympäristönsä kanssa, sen sijaan että se laskisi lopputuloksen ja lopettaisi toimintansa [9]. Yksi esimerkki reaaliaikaisesta reaktiivisesta järjestelmästä on liikennevalojen ohjausjärjestelmä. Tässä järjestelmässä sisäiset tapahtumat perustuvat ajastimiin ja ulkopuoliset tapahtumat perustuvat liikenteen käyttäjiin.

Reaktiivinen järjestelmä tarkkailee tapahtumia ja reagoi niihin. Tapahtuma (*event*) on ilmoitus siitä, että jotain on tapahtunut. Chandy määrittelee tapahtuman lähettäjänsä merkittäväksi tilamuutokseksi [10]. Tapahtumalla on aina yksi lähettäjä sekä *0-n* tarkkailijaa. Reaktiivista järjestelmää, jonka toimintaa ohjaavat paitsi tapahtumat, mutta myös tapahtumien historia, kutsutaan dynaamiseksi.

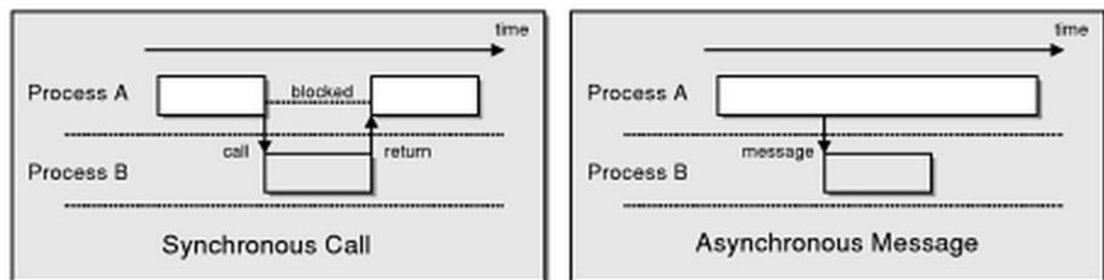
Reaktiivisen järjestelmän toimintaa voidaan kuvata äärellisillä tilakoneilla (*Finite-State Machine, FSM*), jossa näytetään järjestelmän mahdolliset tilat sekä tilasiirtymälogiikka sisäisten ja ulkoisten tapahtumien mukaan [7]. Tilakaaviossa (Kuva 2) esitetään esimerkin vuoksi vedenkeittimen UML-tilakonekaavio. Kaaviosta voidaan nähdä, että järjestelmällä on kaksi mahdollista tilaa, ja sen tilasiirtymälogiikat perustuvat ulkoisiin ja sisäisiin tapahtumiin

Reaktiiviset järjestelmät voivat käyttää hyväkseen asynkronista kommunikaatiota, jossa viestin lähettäjän ei tarvitse odottaa vastaanottajan kuittausta jatkaakseen toimiansa. Kuittaus saapuu *joskus tulevaisuudessa,* kun vastaanottaja on käsitellyt viestin. Kuittausta odottaessa lähettäjä voi suorittaa muita toimenpiteitä. Yleensä tämä toiminallisuus toteutetaan viestijonon avulla: vastaanottajalla on jono, johon viestit saapuessaan laitetaan ja

josta vastaanottaja käsittelee viestejä kun pystyy. Hohpen ja Woolfen kaavio (Kuva 3) esittää synkronisen ja asynkronisen kommunikaation eroavaisuuden [11].




**Kuva 2.** *Vedenkeitin mallinnettuna äärellisenä tilakoneena*




**Kuva 3.** *Synkroninen ja asynkroninen kommunikaatio, Hohpe & Woolfe [11]*

Arulanthun et. al. julkaisussa on mitattu synkronisen ja asynkronisen CORBA-kommunikaation eroavaisuuksia latenssin ja läpäisytehon suhteen [15]. Teknologiaeroista huolimatta mittaustuloksia voidaan hyödyntää järjestelmän suunnittelussa: teoreettinen periaate-ero synkronisen ja asynkronisen viestinnän välillä säilyy. Mittaustuloksista käy ilmi, että asynkronisen ja synkronisen kommunikaation latenssiajat eivät juurikaan eroa, mutta asynkronisuutta hyödyntäen viestien läpäisyaste kasvaa jopa 20 prosenttia suurilla lähetystaajuuksilla. Synkroninen kommunikaatio vaatii myös uusien säikeiden luomista, jotta rinnakkaista viestintää voidaan hyödyntää. Julkaisun mukaan asynkronisen viestinnän hyötyjä saavutetaan varsinkin reaaliaikaisuutta ja korkeaa läpäisyastetta vaativissa järjestelmissä.

Asynkroninen kommunikaatio reaktiivisen monitorointijärjestelmän komponenttien välillä on ensisijaisen tärkeää: komponenttien mahdollisuus toimia itsenäisesti, odottamatta toisten komponenttien työtä mahdollistaa skaalautumisen jolloin järjestelmä voi toimia reaaliaikaisesti myös kovan kuorman alla.

## 2.3 Sääntöohjaus

Sääntöohjatun järjestelmän toimintalogiikkaa voidaan ohjata asettamalla ennalta määriteltyjä sääntöjä joko käyttöön tai pois käytöstä. Sääntöohjattu järjestelmä koostuu ennalta määrätyistä säännöistä, syötteestä ja prosessista (*inference engine),* jonka avulla ennalta määriteltyjä sääntöjä sovelletaan syötteeseen [20]. Sääntöohjaus mallia voidaan hyödyntää, kun järjestelmän toimintaa tulee pystyä ohjaamaan suorituksen aikaisesti ennalta määritellyllä tavalla.

Esimerkki sääntöohjatusta järjestelmästä on sähköposti asiakasohjelmiston kansio-ohjaus. Vastaanotettujen sähköpostiviestien talletus asiakasohjelmiston kansioihin voidaan määritellä ennalta määrättyjen sääntöjen avulla. Näitä sääntöjä on asiakasohjelmistossa mahdollista yhdistää logiikkaoperaattoreiden avulla. Ohjelma 1 esitetään yksinkertaista sääntöohjausta sähköpostiohjelmistossa kahden säännön avulla.

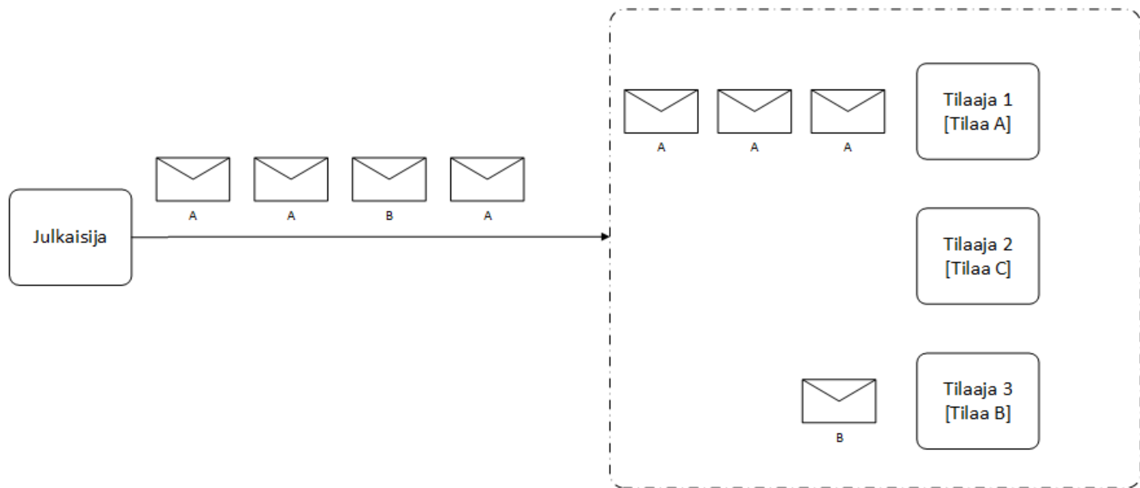
```
From( s ): return (mail.sender == s)
Subject( s ): return (mail.subject.Contains( s ))

IF( From( *@tut.fi ) && Subject(“Diplomityö”) )
THEN MovetoFolder( ”Important” )
```

***Ohjelma 1.*** *Sääntöohjatun järjestelmän toiminta*

Yksi mallin tarjoama ominaisuus on ennalta määriteltyjen sääntöjen luominen ja toiminnan ohjaaminen niiden avulla. Luomalla rajoitettu joukko uudelleenkäytettäviä sääntöjä, näistä säännöistä saadaan hyvin luotettavia. Monitorointijärjestelmälle luotettava ajonaikainen toiminnanohjaus on erittäin tärkeää konfiguroitavuuden näkökulmasta sekä myös arkkitehtuurillista syistä. Järjestelmä toimii pääasiassa muistin varassa, eikä tilakoneiden sisäisiä tiloja tallenneta tietokantaan, joten järjestelmän täydellistä alustusta halutaan välttää.

## 2.4 Julkaisija-tilaaja

Julkaisija-tilaaja (*Publish-subscribe*) on suunnittelumalli viestinvälitykseen kahden tai useamman komponentin välillä. Viestin lähettäjä toimii julkaisijana, joka suoran lähettämisen sijaan julkaisee viestin. Viestin vastaanottajat toimivat tilaajina, jotka tilaavat julkaisijoilta heitä kiinnostavat viestit. Mallin avainominaisuuksia ovat viestin lähettäjien ja vastaanottajien välinen aika- ja paikkariippumattomuus, asynkroninen viestinvälitys, monesta moneen suhteet sekä viestien suodatus [11]. Seuraavassa kaaviossa (Kuva 4) on esitelty julkaisija-tilaaja mallin toimintaa yhden julkaisijan ja kolmen tilaajan välillä.




**Kuva 4.** *Julkaisija-tilaaja suunnittelumalli*

Suunnittelumallin toteutukset yleensä mahdollistavat viestien suodattamisen, mahdollistamalla tilausten asettaminen viestien osajoukkoon. Tarkoman mukaan viestien suodatus parantaa julkaisija-tilaaja järjestelmän tehokkuutta estämällä viestien välityksen tilaajille, joilla ei ole aktiivista tilausta viesteihin [12]. Suodatus toteutetaan yleensä sallimalla tilaus julkaisijan aihealueisiin tai määrätyn sisällön omaaviin viesteihin. Aihealuetilauksessa on julkaisijan vastuulla on luokitella viestit aihealueisiin. Sisältötilauksessa on tilaajan vastuulla tietää viestiskeema ja asettaa tilaus viestin sisällön perusteella. Kuvan 4 tilanteeseen sovitetussa aihealuetilauksessa julkaisija ilmoittaa saatavilla olevat aihealueet (A, B, C) sekä luokittelee viestit näihin kategorioihin. Samaan tilanteeseen sovitetussa sisältötilauksessa Tilaaja 1 tilaa julkaisijalta viestit, joiden sisältö on A.

Monitorointijärjestelmässä julkaisija-tilaajamallia voidaan hyödyntää ulkopuolisten tapahtumien välityksessä niistä riippuvaisille tilakoneille, eli reaktiivisille hälytyssäännöille.

## 2.5 Integraatiokomponentit

Monitorointijärjestelmään tulee pystyä liittämään sensoreita, eli ulkoisia tapahtumalähteitä, jotka tuottavat tilatietoa monitoroitavista kohteista. Järjestelmien integraatio voidaan määritellä kahden tai useamman järjestelmän yhteistoiminnan rakentamisella tavalla, johon niitä ei ole alun perin suunniteltu. Ulkoisten tapahtumalähteiden liittäminen monitorointijärjestelmään täyttää tämän määrittelyn. Hohpe ja Woolfe määrittelevät teoksessaan yleisiä integraatiokomponentteja [11], joista voimme hyödyntää sopivimpia. Komponentit tulee valita teknologisten mahdollisuuksien, ulkoisten tapahtumalähteiden asettamien rajoitteiden sekä toteuttavan järjestelmän laatuvaatimusten perusteella.

# 3. TOTEUTETTAVA JÄRJESTELMÄ

Tässä luvussa tutustutaan tarkemmin toimintaympäristöön, ratkaisuvaihtoehtoihin sekä määriteltyihin laatuvaatimuksiin. Kohdassa 3.1 perehdytään toteutettavan järjestelmän toimintaympäristöön ja sen viestinvälitykseen. Kohdassa 3.2 esitellään lyhyesti valmis monitorointiratkaisu mitä harkittiin ratkaisun tueksi. Kohdassa 3.3 määritellään toteutettavalle järjestelmälle määritellyt laatuvaatimukset ja miten ne pyritään saavuttamaan suunnittelun avulla.

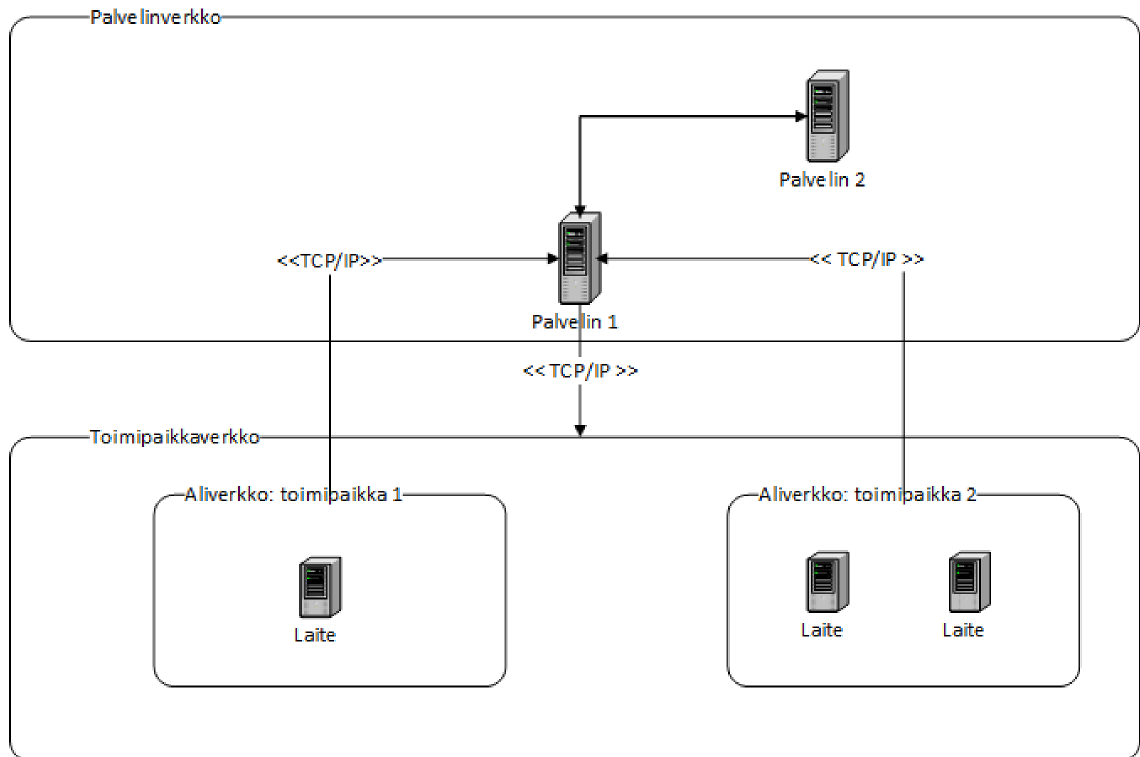
## 3.1 Toimintaympäristö

Toteutettavan monitorointijärjestelmän kohdeympäristö on maantieteellisesti hajautettu laitejärjestelmä. Coulouriksen määritelmän mukaan hajautettu järjestelmä on laitteisto- tai ohjelmistokomponentit, jotka toimivat tietoverkkoon liitetyissä tietokoneissa sekä kommunikoivat ja koordinoivat toimintojaan viestien avulla [8].

Ympäristön monitoroitaviin laitteisiin on kytketty liiketoiminnan vaatimat oheislaitteet erilaisilla protokollilla. Laitteilla ajetaan oheislaitteita ohjaavia ohjelmia, eli ajureita, ja ohjelmia jotka orkestroivat ajureiden yhteistoimintaa liiketoimintaprosessissa. Vaikka laitteiden käyttötarkoitus on pääasiassa sama, laitekanta on heterogeeninen; laitteita on useammalta eri toimittajalta ja oheislaitteiden sekä niiden ajurien määrä ja tyyppi voi vaihtua laitekohtaisesti. Yhteys laitteille muodostetaan TCP/IP-protokollaperheellä.

Järjestelmän osapuolten välinen tiedonsiirto tapahtuu yhteisessä verkossa, joka on jaettu aliverkkoihin maantieteellisen sijainnin, ns. toimipaikan perusteella. Eri toimipaikkojen verkoista ei ole näkyvyyttä toistensa aliverkkoihin. Keskitetty hallinta hoidetaan palvelimilta, jotka on määritelty yhteiseen verkkoon *full duplex* -yhteydelliseksi kaikkiin sen aliverkkoihin. Verkkokaavio (Kuva 5) havainnollistaa toimintaympäristön verkkokonfiguraatioita. Toteutettavan monitorointijärjestelmän tulee toimia palvelinverkossa.

Keskitetty hallinta palvelimen ja laitteiden välillä tapahtuu sanomavälityksellä. Laitteille voidaan lähettää ohjaussanomia. Ohjaussanomat ovat määrämuotoisia tiedostoja, joita laite vastaanottaessaan tulkitsee ja muuttaa toimintansa niihin perustuen. Ohjaussanomat siirretään laitteille vaihtelevilla siirtoprotokollilla, joihin lukeutuvat mm. FTP, SOAP ja *socket*-tiedonsiirto. Vastaavasti laitteet lähettävät infosanomia määrämuotoisia tiedostoja, jotka kertovat laitteella tapahtuvasta toiminnasta tai laitteen tilasta. Infosanomat siirretään palvelimelle vaihtelevilla tavoilla: laite voi joko lähettää tiedoston palvelimelle tai palvelin voi hakea tiedostoja laitteelta. Kuten ohjaussanomien siirrossa, myös infosanomien siirtoprotokolla vaihtelee laitekohtaisesti.




**Kuva 5.** *Toimintaympäristön tietoliikenneverkot*

Tässä toimintaympäristössä laitteiden toiminnan reaaliaikainen monitorointi perustuu laitteiden lähettämiin infosanomiin. Laitevalmistajat toteuttavat yleensä hyvin yksinkertaiset virhetilanteissa lähetettävät infosanomat. Esimerkiksi hetkelliset yhteyshäiriöt laitteisiin voivat aiheuttaa infosanoman lähetyksen. Kokemus on kuitenkin osoittanut, että laitteiden virhetilanteissa lähettämiin infosanomiin voi harvoin luottaa. Luotettavaan monitorointiin tarvitaan tällä hetkellä ylläpitäjä tai muu henkilö, joka virhetilanteen infosanoman saapuessa tarkistaa hälytyksen oikeellisuuden, muun muassa laitteen sanomahistorian perusteella. Sanomahistorian tulkinnan perusteella ylläpitäjät ja asiantuntijat ovat löytäneet kaavoja, joiden perusteella voidaan päätellä hälytyksen olevan todellinen.

## 3.2 Valmiit ratkaisut

Määrittelyn alussa kartoitettiin valmiita monitorointiratkaisuja jotka voisivat sopia tarpeisiimme. Yritysten IT-hallinnalle suunnattuja monitorointijärjestelmiä on olemassa valmiina tuotteina. Kartoitetut valmiit ratkaisut tarjosivat pääasiassa hyvin yksinkertaisia sääntöjä ja käyttöliittymiä monitorointiin. Ne eivät kuitenkaan tarjonneet ratkaisua kohdeympäristön tarpeisiin: mallintaa spesifiset hälytyskriteerit äärellisinä tilakoneina ja toteuttaa ne reaktiivisen mallin mukaisesti.

Yksi suosituimmista ja monipuolisimmista monitorointijärjestelmistä on Nagios [3]. Hyvänä puolena Nagioksessa on valmiit käyttöliittymät monitorointituloksiin, historian tarkasteluun ja konfiguraatioiden asetukseen. Nagios tarjoaa ratkaisuja myös hajautetun verkon laitteiden monitorointiin. Myös Nagioksen monitorointiratkaisut on rakennettu yhteisten teknologioiden ja protokollien tarpeisiin, jolloin ne ovat otettavissa käyttöön lähes missä tahansa IT-hallinnassa. Nagios hylättiin ratkaisuvaihtoehtona, koska sen käyttöönotossa ja räätälöinnissä arvioitiin kuluvan liikaa aikaa. Lisäksi monet sen tarjoamista ominaisuuksista on jo toteutettu sekä sen käyttöönotto olisi voinut vaatia maksullista konsultointia.

## 3.3 Järjestelmän laatuvaatimukset

Tässä kohdassa esitellään omina alakohtina monitorointijärjestelmälle asetetut laatuvaatimukset, syyt näille laatuvaatimuksille ja miten asetetut laatuvaatimukset pyritään saavuttamaan.

### 3.3.1 Reaaliaikaisuus

Monitorointijärjestelmän reaaliaikaisuus takaa virheentunnistuksen jatkuvasti ja mahdollisimman nopeasti. Tässä yhteydessä reaaliaikaisuudella ei tarkoita toimimista reaaliaikavaatimusten mukaisesti, vaan jatkuvaa prosessia jossa virheentunnistus tapahtuu mahdollisimman nopeasti eikä esimerkiksi ajastettuina ajoina. Reaaliaikainen virheentunnistus on yksi tärkeimmistä vaatimuksista, koska vakava laitevirhe voi pysäyttää liiketoiminnan kokonaan. Mitä nopeammin huoltotoimenpiteet voidaan aloittaa, sitä nopeammin liiketoimintaa voidaan taas käydä.

Reaaliaikaisuus saavutetaan noudattamalla reaktiivista tapahtumakeskeistä mallia järjestelmän arkkitehtuurin suunnittelussa.

### 3.3.2 Konfiguroitavuus

Monitorointijärjestelmän konfiguroitavuus mahdollistaa laitekohtaisen tarkkuuden. Eri valmistajien laitteet vaativat omat konfiguraatiorajansa, mutta konfiguraatioihin voi vaikuttaa myös ulkopuoliset tekijät, kuten laitteen käyttötaajuus. Konfiguraatioita on pystyttävä säätämään laitekohtaisesti, jotta ajan myötä jokaiselle laitteelle löydetään tehokkaimmat konfiguraatiot. Automaattisesti mukautuvaa konfigurointia ei kuulu toiminnalisiin vaatimuksiin eikä sitä toteuteta ensimmäisessä vaiheessa.

Hyvä konfiguroitavuus saavutetaan noudattamalla sääntöohjattua mallia järjestelmän arkkitehtuurin suunnittelussa sekä muistissa ajettujen hälytysten konfiguraatioiden ajoittaisella tietokantasynkronoinnilla.

### 3.3.3 Laajennettavuus

Laajennettavuus voidaan määritellä järjestelmän kyvyksi ottaa käyttöön uusia toiminallisuuksia niin, että vaikutukset sen sisäiseen rakenteeseen ja tiedonvälitykseen ovat minimaaliset tai olemattomat [21]. Monitorointijärjestelmän laajennettavuus takaa yhtenäisen toteutustavan monitorointitarpeille tulevaisuudessakin sekä järjestelmän pitkän elinkaaren. Järjestelmän on oltava laajennettavissa ulkopuolisten tapahtumalähteiden sekä sisäisten hälytyslogiikoiden suhteen.

Laajennettavuus saavutetaan tarjoamalla helposti käyttöönotettava rajapinta ulkoisille tapahtumalähteille, sekä tekemällä uusien sääntöjen toteutus mahdollisimman vaivattomaksi arkkitehtuurin avulla.

### 3.3.4 Skaalautuvuus

Skaalautuvuus voidaan määritellä järjestelmän kyvyksi suorittaa kasvava määrä työtä onnistuneesti tai tulla kasvatetuksi suhteessa työn määrään [22]. Monitorointijärjestelmän suorituskyvyn on skaalauduttava vaihtelevan kuormituksen alla. Skaalautuvuus tulee huomioida ulkoisten tapahtumien käsittelyssä sekä laitekohtaisten hälytystilojen ylläpidossa. Ulkoisten tapahtumien käsittely vaatii prosessorin laskenta-aikaa. Ulkoisia tapahtumia voidaan arvioida saapuvan järjestelmään maksimitaajuudella 20 tapahtumaa sekunnissa. Hälytysten tiloja ylläpidetään muistissa ja vain tilamuutokset kirjoitetaan tietokantaan. Alkuvaiheessa voidaan arvioida ylläpidettäväksi 500 – 1500 tilaa. Arvio perustuu laitemäärään kentällä kerrottuna toteuttavilla hälytystyypeillä. Tässä vaiheessa on vaikea arvioida yhden ylläpidettävän tilan muistinkulutusta.

Hyvä skaalautuvuus saavutetaan hyödyntämällä asynkronista viestinvälitystä järjestelmän komponenttien välillä sekä toteuttamalla sanomien harmonisointi ja sääntöjen tilasiirtymälogiikat mahdollisimman tehokkaiksi laskentatehon ja muistinkulutuksen suhteen.

### 3.3.5 Vikasietoisuus

Vikasietoinen järjestelmä voidaan määritellä sellaisena, joka jatkaa toimintaansa kelvollisesti virheiden ilmetessä [23]. Monitorointijärjestelmän virhetilanteet eivät saa johtaa väärien hälytysten muodostamiseen. Tilanteet, jossa järjestelmään saapuvat ulkoiset tapahtumat käyttäytyvät odottamattomalla tavalla, pitää tunnistaa ja hälytysten muodostusta tulee rajoittaa. Tilanteet, jossa järjestelmän sisäinen logiikka käyttäytyy odottamattomalla tavalla, pitää tunnistaa ja raportoida ylläpidollisia toimia varten.

Vikasietoisuus saavutetaan jatkuvalla yksikkötestauksella, varaamalla prosentuaalisesti suuri työmäärä järjestelmätestaukseen sekä pilotoimalla järjestelmää ennen täyttä tuotantoon vientiä.

### 3.3.6 Luotettavuus

Monitorointijärjestelmän tuottamien hälytysten oikeellisuuteen pitää pystyä luottamaan. Manna ja Pnuelli mainitsevat historian osoittaneen, että luotettavien reaktiivisten järjestelmien toteuttaminen on yksi vaikeimmista ohjelmointitehtävistä [9]. Vikatilanteiden tunnistuskriteerit on pystyttävä mallintamaan järjestelmässä niin hyvin, että jokaisen hälytyksen tiedetään olevan aiheellinen sekä jokaisesta virhetilanteesta luodaan hälytys. Toisin sanoen järjestelmä ei tuota *false positive* -hälytyksiä, eikä yksikään virhetilanne jää huomaamatta (*true negative)*.

Luotettavuus saavutetaan liiketoimintaprosessin ja reaalimaailman ongelman ymmärryksellä sekä kattavilla yksikkötestitapauksilla.

# 4. SUUNNITTELU

Tässä luvussa esitellään järjestelmän suunnittelua, arkkitehtuuria ja käsitteitä sekä miten ne mallinnetaan ohjelmistotekniikan menetelmin. Havainnollistamisen apuna käytetään UML-kaavioita. Tästä eteenpäin toteutettavasta monitorointijärjestelmästä käytetään sen projektinimeä, *AlarmSystem*. Kohdassa 4.1 määritellään *AlarmSystemin* suunnittelua ohjaavat päätökset. Kohdassa 4.2 esitellään järjestelmän suunnittelussa käytetyt reaalimaailmaa mallintavat käsitteet. Kohdassa 4.3 esitellään suunniteltu ohjelmistoarkkitehtuuri ja toiminnallisuuksien jako moduuleihin. Kohdassa 4.4 esitellään relaatiotietokantaskeema, joka on rakennettu käsitteiden suhteiden perusteella. Kohdassa 4.5 esitellään olio-ohjelmointiin kuuluvaa luokkajakoa, joka perustuu kohdan 4.3 käsitteiden toiminnallisuuksiin. Kohdassa 4.6 esitellään ulkoisten tapahtumalähteiden integraatiosuunnitelmaa. Kohdassa 4.7. määritellään muistin ja tietokannan synkronointiin toteutettu ratkaisu.

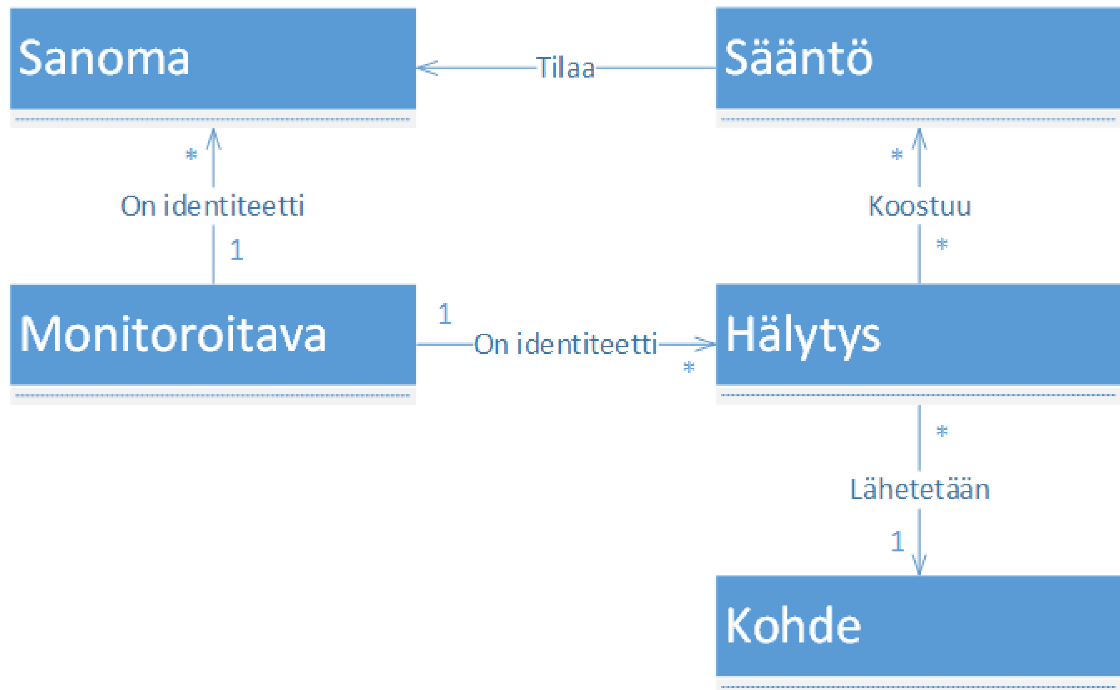
## 4.1 Määrittely

Tässä määrittelyssä kuvataan suunnittelua ohjaavat päätökset, jotka on tehty hyödynnettävien suunnittelumallien, toimintaympäristön sekä järjestelmän laatuvaatimusten mukaan.

1. *AlarmSystem* toteutetaan palveluna, johon voidaan konfiguroida hälytyksiä monitoroivaille kohteille.
2. *AlarmSystemin* hälytykset toteutetaan joukkoina sääntöjä, jotta hälytysten toimintaa voidaan ohjata sääntöohjaus mallin mukaisesti.
3. Säännöt ovat reaktiivisia äärellisiä tilakoneita, joiden tilasiirtymälogiikka perustuu tapahtumareagointiin.
4. *AlarmSystemiin* voidaan integroida ulkoisia tapahtumalähteitä ja *AlarmSystem* tulee välittämään niiden tapahtumat *julkaisija-tilaaja* -mallin mukaisesti niistä kiinnostuneille tilaajille, eli hälytysten säännöille, käyttäen asynkronista viestinvälitystä.
5. *AlarmSystem* tulee välittämään aktivoituneet hälytykset niistä kiinnostuneille osapuolille.
6. *AlarmSystem* toteutetaan olio-ohjelmoinnin periaatteiden mukaisesti.

## 4.2 Käsitteet

Tässä kohdassa listataan tärkeimmät käsitteet, joita käytetään järjestelmän suunnittelussa. Käsitekaavio (Kuva 6) esittelee käsitteet ja hahmottaa niiden suhteita.

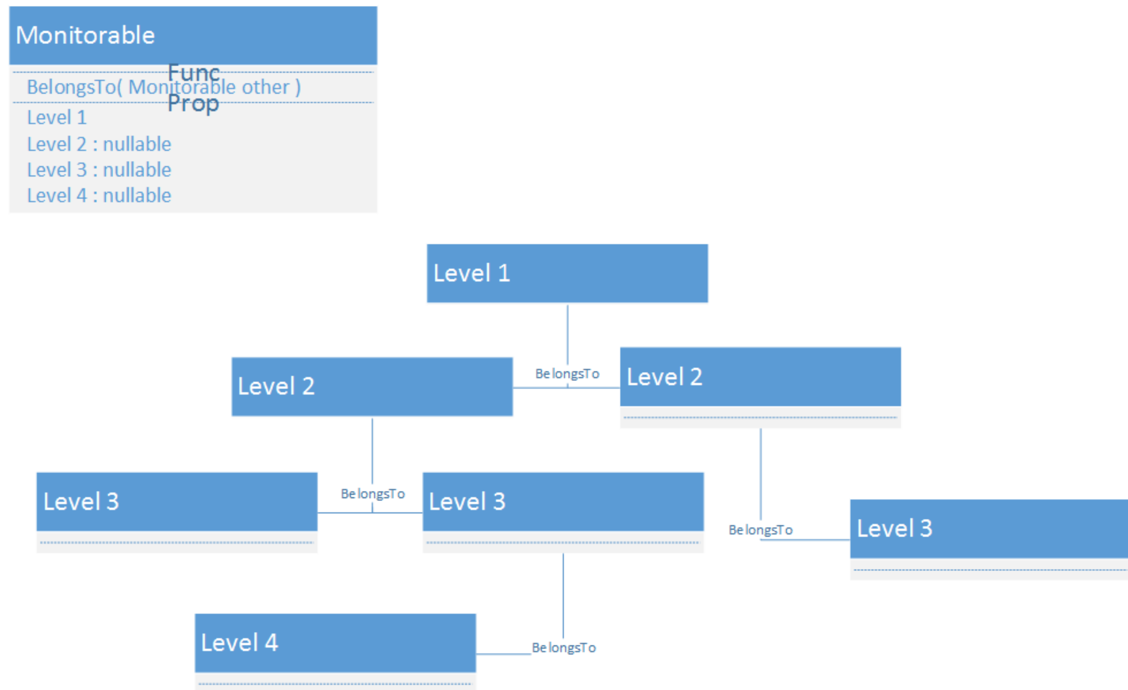



**Kuva 6.** *AlarmSystem käsitekaavio*

### 4.2.1 Monitoroitava (yksikkö)

Monitoroitavalla (yksiköllä) tarkoitetaan määriteltävää identiteettiä, jolle monitorointi voidaan asettaa ja räätälöidä. Monitoroitava mallintaa reaalimaailman laitetta, tai niiden yhdistelmää. Monitoroitavaksi on mahdollista määritellä useampi taso, esimerkiksi koko liikepaikka ja sen laitteet, liikepaikan tietyt laitteet tai yksi liikepaikan tietty laite. Monitoroitavat voivatkin muodostaa puurakenteen (Kuva 7), jossa alempien tasojen monitoroitavat kuuluvat ylemmän tason monitoroitaville.

Puurakenteesta on hyötyä sääntöjen tilausten hallinnassa tapahtumasekvenssin osajoukkoon. Monitoroinnin sääntö voidaan asettaa ylemmälle tasolle, jolloin sanomat joiden monitoroitava identiteetti on tämän ylemmän tason perillinen, välitetään luodulle säännölle.



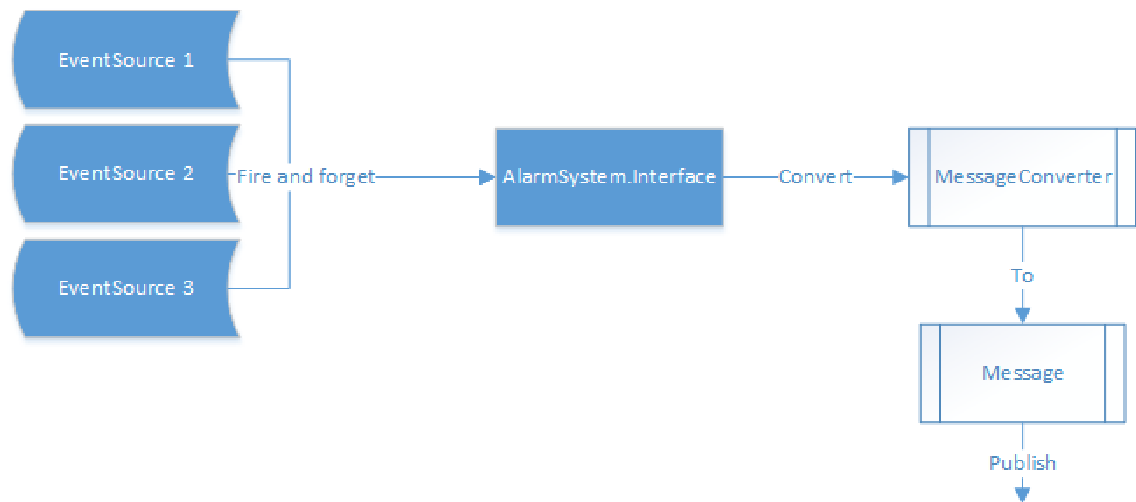

**Kuva 7.** *Monitoroitava puurakenne*

### 4.2.2 Kohde

Kohde mallintaa tyypillisesti toista palvelua, joka on kiinnostunut hälytyksistä. Kohteelle on konfiguroitu URI-osoitteita, joihin hälytys aktivoituaan lähetetään. URI määrittelee siirtoprotokollan, ja lisäkonfiguraatiolla määritellään mediatyyppi sekä viestiskeeman nimi mihin hälytysviesti tulee muuntaa ennen lähetystä. *AlarmSystem.Communication* -moduuli tarjoaa konvertoinnin viestiskeemoihin sekä lähetyksen määritellyillä protokollilla. Hälytyksellä on aina yksi kohde.

### 4.2.3 Sanoma

Sanoma mallintaa *AlarmSystem*iin saapuvaa ulkopuolista tapahtumaa: käytännössä monitoroitavan laitteen lähettämää infosanomaa, joka yleensä on XML-dokumentti. Ulkopuoliset tapahtumat konvertoidaan *AlarmSystem.Interface* -moduulin toimesta järjestelmän omaan yhtenäiseen muotoon, jotta eri laitevalmistajien lähettämiä sanomia voidaan käsitellä yhtenäisillä tavoilla. Jos vastaanotettu sanoma ei mukaudu järjestelmän yhdenmukaistettuun muotoon, sanoma hylätään. Yhdenmukaiset sanomat sitten julkaistaan tarkkailtavaan tapahtumasekvenssiin, johon säännöt voivat asettaa tilauksensa perustuen kiinnostukseen. Tilaus asetetaan yleensä sanoman identiteetin perusteella, joka on monitoroitava (yksikkö).




**Kuva 8.** *Sanomien muodostus*

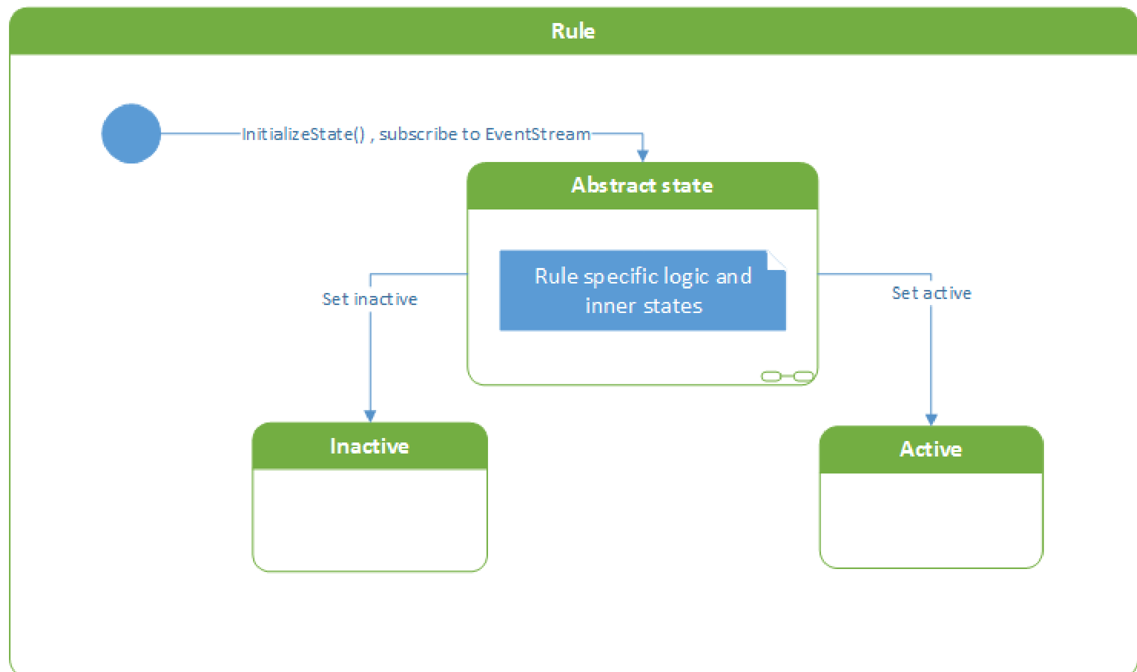
Erilaisia sanomatyyppejä *AlarmSystemissä* on kourallinen. Eri sanomatyypit omaavat erilaisia attribuutteja. Tästä syystä sanomien hallinnassa käytetään abstraktia kantaluokkaa *Message*, joka toteuttaa sanomien perustoiminnot sekä tarjoaa yhtenäisen luokan tapahtumasekvenssille ja muille tietorakenteille. Abstrakti kantaluokka mahdollistaa myös laajennettavuuden uusien sanomatyyppien osalta. Eri sanomatyyppejä ovat muun muassa *Info, Transaction* ja *Summary.*

### 4.2.4 Sääntö

Säännöt ovat äärellisiä tilakoneita, joiden sisäisiä tiloja on $n$ kappaletta, mutta ulospäin näkyy vain kaksi tilaa: aktiivinen ja epäaktiivinen. Säännöt mallinnetaan luokkina, joiden tilasiirtymälogiikka määritellään toteutuksessa ja konfiguroitavilla parametreillä. Sääntöjen tilasiirtymälogiikka perustuu *AlarmSystem*iin saapuviin ulkopuolisiin tapahtumiin, eli sanomiin, ja niihin reagoimiseen. Tilakaavio (Kuva 9) kuvastaa säännön perustoiminnallisuutta järjestelmässä.

*AlarmSystem*iin saapuvat sanomat ohjataan oikeille säännöille *julkaisija-tilaaja* -suunnittelumallin mukaisesti. *AlarmSystem.Interface* -moduuli julkaisee konvertoidut sanomat *IObservable* -rajapinnan toteuttamana tapahtumasekvenssinä, johon *IObserver* -rajapinnan toteuttavat säännöt voivat asettaa tilauksensa. Tilausparametreihin kuuluvat yleensä säännön omistavan hälytyksen identiteetti sekä säännöille spesifiset sanomatyypit.

Ohjelma 2 on esimerkki säännön toimintalogiikasta. Tapahtumasekvenssiin asetetaan tilaus sanomille, joiden identiteetti kuuluu hälytyksen monitoroitavalle sekä sanoma on *Info*-tyyppinen. Lopuksi tilaus perutaan funktiolla *Dispose()*, joka ilmoittaa julkaisijalle tilauksen peruutuksesta sekä vapauttaa resurssit.



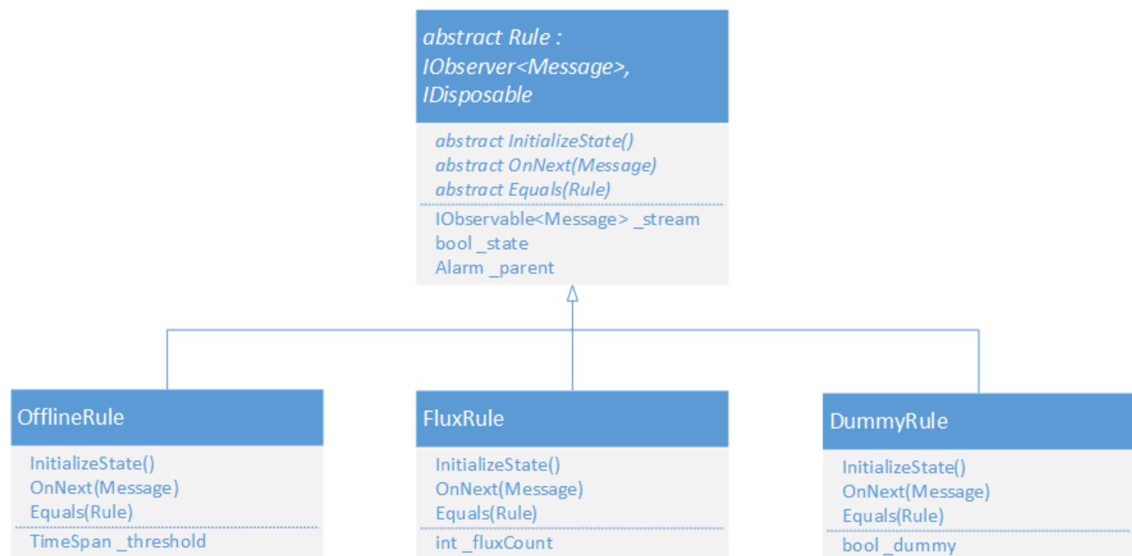

**Kuva 9.** *Tilakaavio sääntöjen toimintalogiikasta*

```
private IObservable<Message> _stream = _engine.GetEventStream();

var subscription = _stream
.Where(msg=> msg.Identity.BelongsTo(_alarm.Identity) &&
             msg.MessageType.Equals( MessageType.Information))
.Subscribe( this );

subscription.Dispose();
```

***Ohjelma 2.*** *Yksinkertainen sisältöperusteinen tilaus tapahtumasekvenssiin ja tilauksen peruminen Rx-kirjaston mukaisesti.*

Kaikki järjestelmän säännöt periytyvät abstraktista kantaluokasta *Rule*, joka toteuttaa sääntöjen perustoiminnallisuudet. Kantaluokka määrittelee kolme abstraktia funktiota toteutettavaksi periytyville luokille: *InitializeState(), OnNext(Message value)* sekä *Equals(Rule other).* Funktiossa *InitializeState()* määritellään säännön tilan alustus. Tilan alustukseen kuuluu tarvittaessa käynnistysvaiheen sisäisen tilan määritys tietokannan perusteella, sekä jatkuvan tilasiirtymälogiikan määrittely tapahtumasekvenssin sanomien perusteella. Funktio *OnNext(Message value)* on *IObserver*-rajapinnan funktio, jota kutsutaan funktiokutsulla *Subscribe(this)*. Sääntöjen tapauksessa sitä käytetään tilasiirtymälogiikan lopussa tilan aktivoimiseen. Toteutuksen jättäminen abstraktiksi antaa säännöille mahdollisuuden kustomoida toimintaansa sääntöjen aktivointitilanteessa. Funktio *Equals(Rule other)* on *object*-luokan vertailuoperaattorin ylikirjoitus. Tässä funktiossa sään-

töjen tulee toteuttaa arvovertailu, jota hyödynnetään konfiguraatioiden päivityksessä tietokannasta muistiin. Kuva 10 on esimerkin vuoksi osin puutteellinen UML-luokkakaavio sääntöjen periytymishierarkiasta.



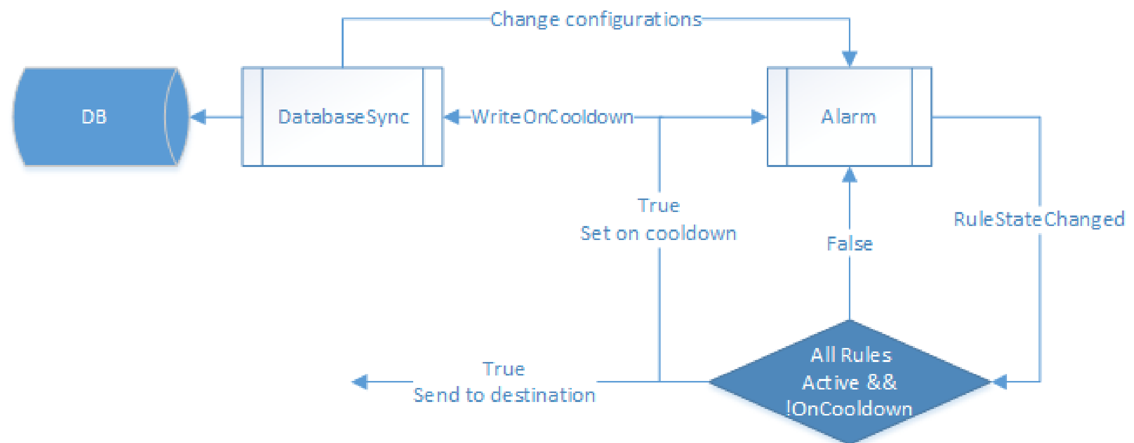

**Kuva 10.** *Sääntöjen abstrakti kantaluokka ja periytyminen*

Ratkaisu tukee laajennettavuutta, sillä uusia monitorointitarpeita voidaan toteuttaa periyttämällä uusia luokkia *Rule*-kantaluokasta. Nämä luokat voidaan sitten ottaa käyttöön hälytyksissä sääntöohjausmallin mukaisesti.

### 4.2.5 Hälytys

Hälytys koostaa joukon sääntöjä monitoroitavalle yksikölle. Hälytykset alustetaan tietokannasta. Hälytys seuraa sen sääntöjen tiloja *RuleStateChanged* tapahtumien perusteella. Hälytys noudattaa sääntöohjausmallia yhdistämällä sen säännöt *AND*-operaattorilla; kun hälytykselle määrätyt säännöt ovat samanaikaisesti aktiivisessa tilassa, myös hälytys aktivoituu. Hälytysten monipuolisuutta ja sääntöohjausmallin toteutusta voidaan jatkossa laajentaa tarjoamalla hälytyksille mahdollisuus koostaa sääntöjä muidenkin logiikkaoperaattoreiden, kuten *OR* tai *XOR*, avulla yhteen. Hälytyksen aktivointiin liittyy viestinvälitys hälytykselle määrättyyn kohteeseen, sekä hälytyksen asettaminen *viilentymisjaksolle*. Viilentymisjaksoa käytetään lukkomekanismina saman hälytyksen korkean taajuuden duplikaattilähetyksen estämiseksi.

Vuokaavio (Kuva 11) havainnollistaa hälytyksen toimintaa.




**Kuva 11.** *Vuokaavio hälytysten toimintalogiikasta*

## 4.3 Arkkitehtuuri

Ohjelmistoarkkitehtuuria käytetään määrittelemään sen kohteen kokonaisvaltainen rakenne, joka sisältää muun muassa toiminallisuuksien järjestämisen suunnitteluelementteihin ja elementtien väliset yhteydet [19]. Yksi suunnitteluelementti on *moduuli:* itsenäinen osa ohjelmaa, jonka vastuulla on määrätty osakokonaisuus.

*AlarmSystemin* ohjelmistoarkkitehtuuri voidaan jakaa kolmeen moduuliin, jotka ovat vastuussa järjestelmän eri osakokonaisuuksista (Kuva 12).

Moduuli *AlarmSystem.Interface* tarjoaa rajapinnan ulkopuolisille tapahtumalähteille. Ulkopuoliset tapahtumat ovat tyypillisesti laitevalmistajien infosanomia, jotka lähteet lähettävät moduulille. Ulkopuoliset tapahtumat yhdenmukaistetaan järjestelmälle yhtenäiseen muotoon, sanomiksi, jotka julkaistaan *julkaisija-tilaaja* suunnittelumallin mukaisesti tarkkailtavana tapahtumasekvenssinä.

Moduuli *AlarmSystem.AlarmEngine* tarjoaa hälytysten alustamisen ja päivittämisen tietokannan perusteella, hälytysten tilojen ylläpidon muistissa ja osittain tietokannassa sekä sääntöjen jatkuvan tilasiirtymälogiikan tarkkailtavan tapahtumasekvenssin tilausten perusteella.

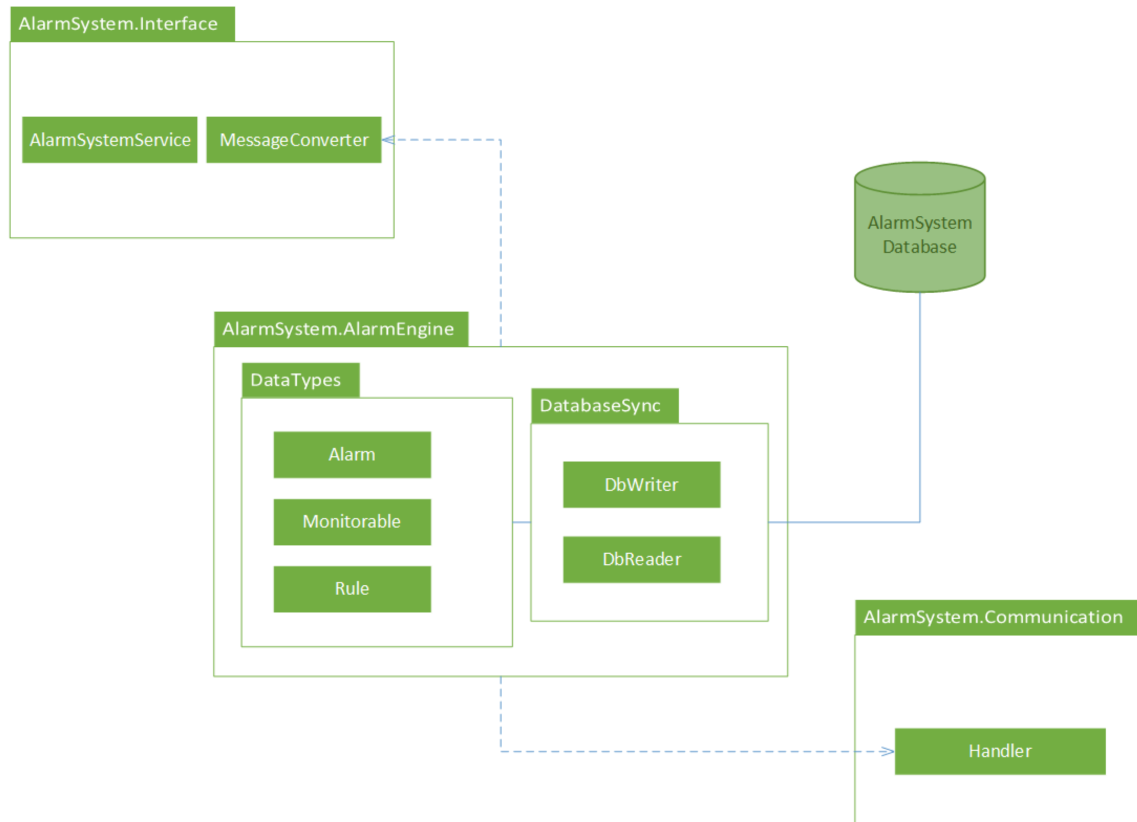
Moduuli *AlarmSystem.Communication* tarjoaa hälytysten lähetykseen kohteille tarvittavat toimenpiteet, joita ovat hälytyksen konvertointi kohteen skeemaan sekä viestinvälitys kohteelle asetetun protokollan mukaisesti.

Vapaamuotoinen vuokaavio (Kuva 13) havainnollistaa suunnitteluratkaisujen, ohjelmsitoarkkitehtuurin, moduulien ja käsitteiden yhteistoimintaa reaktiivisen järjestelmän jatkuvassa prosessissa.

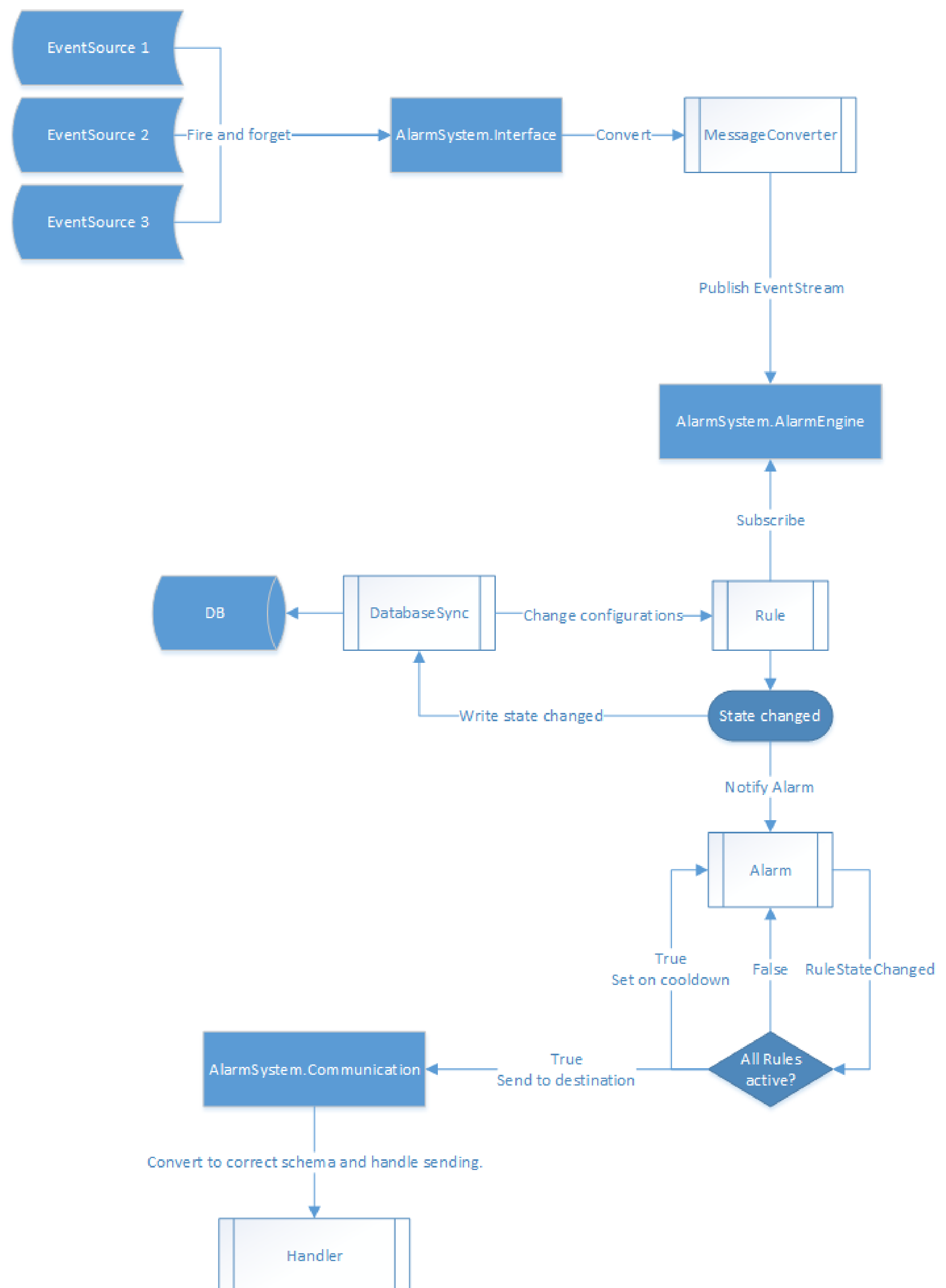
## 4.4 Tietokantaskeema

Tietokantaskeema on rakennettu relaatiotietokannalle. Kaaviossa (Kuva 14) on esitetty tietokantaskeema *AlarmSystem*in tietosisältöä varten.

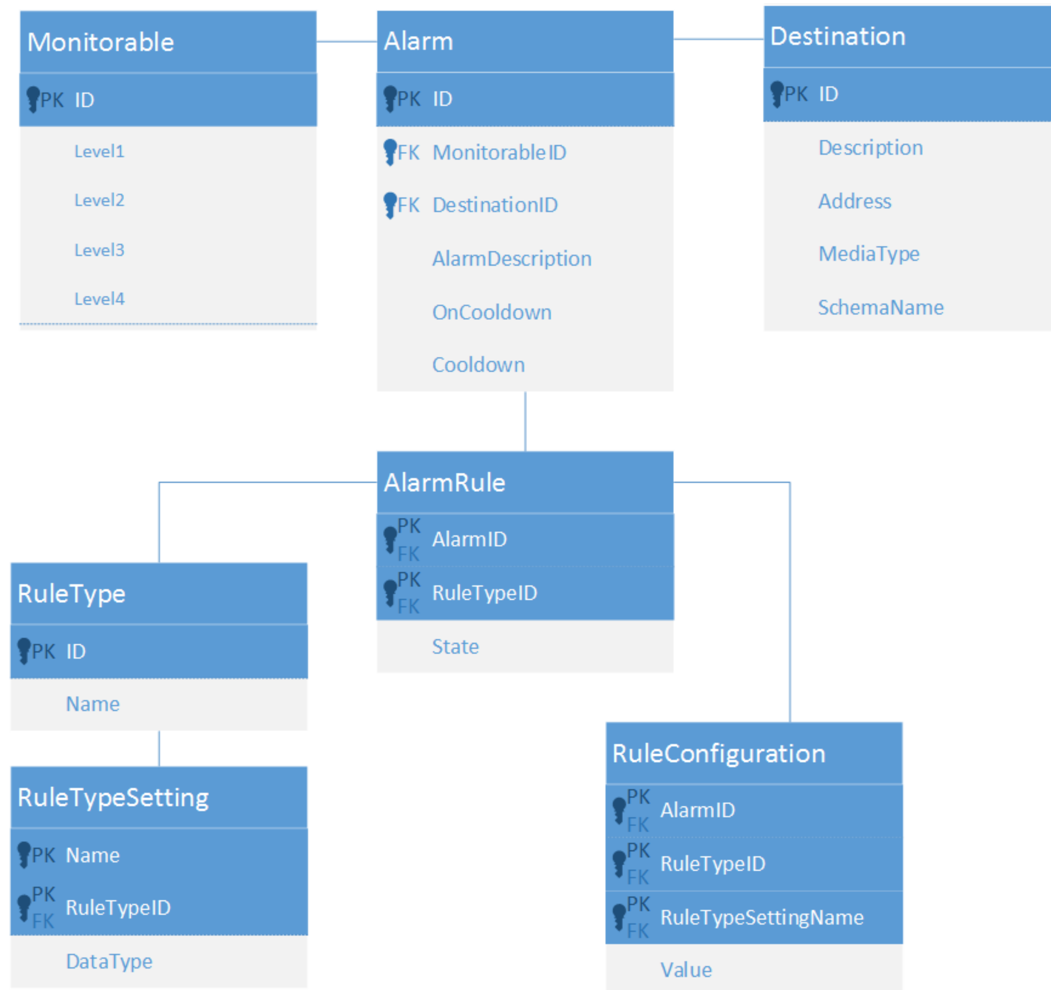
Tietokantataulut *Monitorable*, *Alarm* ja *Destination* ovat suoraviivaisia ja niiden tietosisältö heijastaa suoraan määriteltyjen käsitteiden ominaisuuksia. Sääntöjen tietosisällön hallintaan tarvitaan kuitenkin useampi taulu. Taulut *RuleType* ja *RuleTypeSetting* määrittelevät hälytyksille mahdollisesti käyttöönotettavat säännöt sekä sääntöjen pakolliset konfiguraatiot. Taulut *AlarmRule* ja *RuleConfiguration* yhdistävät säännöt hälytyksiin ja määrittävät hälytykselle spesifisten sääntökonfiguraatioiden arvot. Ratkaisu tukee konfiguroitavuutta; jokaiselle hälytykselle voidaan asettaa haluttu määrä sääntöjä ja jokaisen säännön konfiguraatiot voidaan asettaa halutuksi.




**Kuva 12.** *AlarmSystem arkkitehtuuri*




**Kuva 13.** *Vuokaavio AlarmSystemin moduulien yhteistoiminnasta*



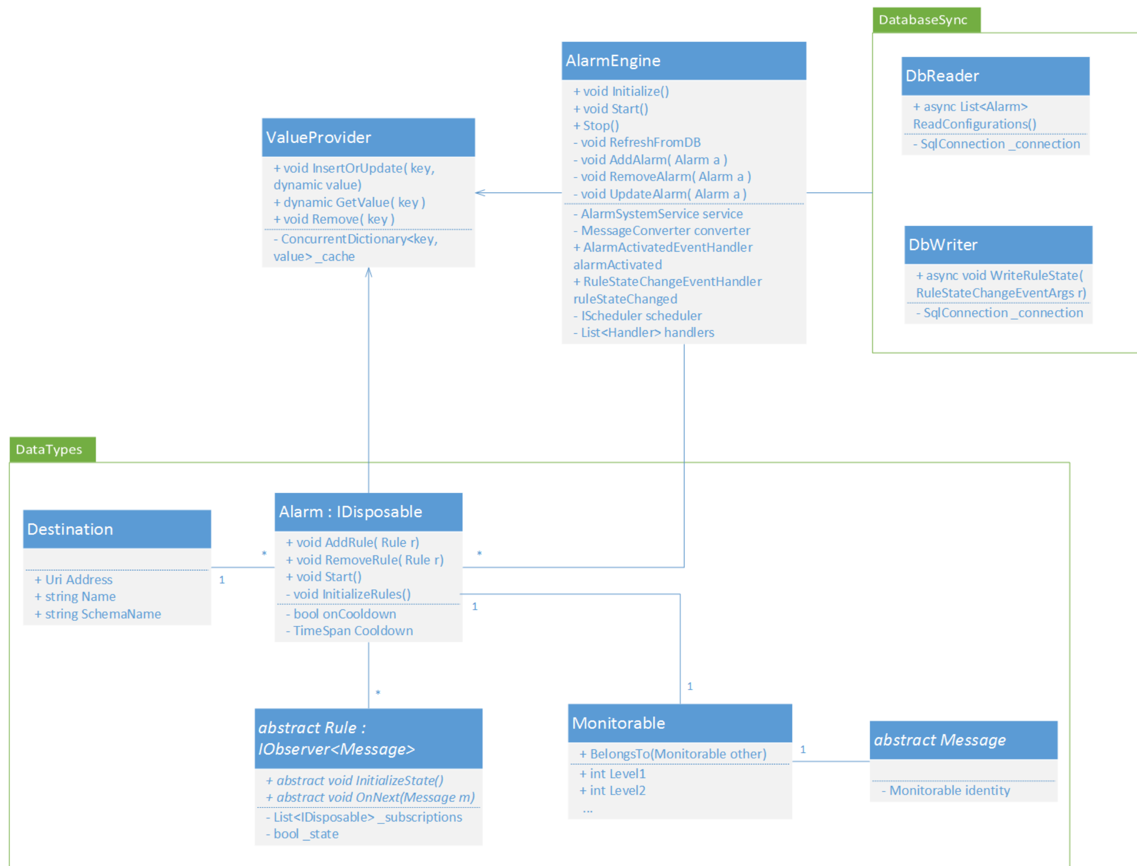

**Kuva 14.** *AlarmSystem tietokantaskeema*

## 4.5 Luokkasuunnittelu

Suunnitellessa ohjelmistoa olio-ohjelmoinnin periaatteiden mukaisesti, kuuluu suunnitteluvaiheeseen luokkasuunnittelu. Luokkasuunnittelu sisältää moduulien toiminallisuuksien jakamisen luokkiin, luokkien tietosisällön ja toiminallisuden määrittely sekä luokkien välinen kommunikaatio ja hierarkia. Luokkasuunnittelun lopputuloksena on yleensä UML-luokkakaavio. Alla on esillä *AlarmSystemin* moduulien UML-luokkakaaviot tärkeimpien luokkien ja ominaisuuksien osalta.

### 4.5.1 AlarmEngine –moduuli

Kuva 15 luokkakaaviossa on kuvattu *AlarmEngine* moduulin oleellisimmat luokat ja toiminnallisuudet.




**Kuva 15.** *AlarmSystem.AlarmEngine moduulin luokkakaavio*

Moduuli sisältää myös ali-nimiavaruudet *DataTypes* ja *DatabaseSync*. Luokka *AlarmEngine* on ylimmän tason toimija, jonka toiminta käynnistetään ja lopetetaan *Start* ja *Stop* kutsuilla Windowsin palvelurajapinnalta. *AlarmEnginen* vastuulla on alustaa ja synkronoida järjestelmän tila sekä käyttää muiden moduulien tarjoamia palveluja. *AlarmEngine* kuuntelee hälytysten ja sääntöjen tapahtumia *AlarmActivatedEventhandler* ja *RuleStateChangedEventHandler* tapahtumakäsittelijöiden avulla sekä välittää hälytysten aktivoinnit *AlarmSystem.Communication* moduulille ja sääntöjen tilamuutokset *AlarmEngine.DatabaseSync* moduulille. Luokka *ValueProvider* on kätkö, joka tarjoaa muistinvaraisen tietosisällön resursseille joiden uudelleenhankinta on kallista.

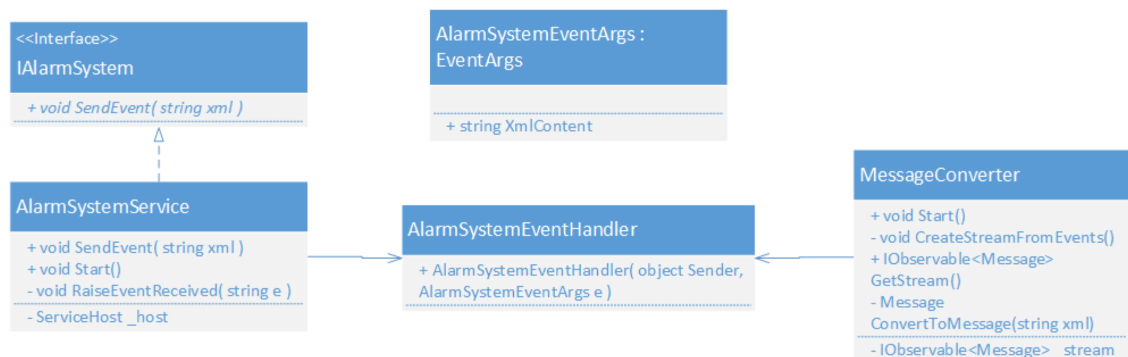
Nimiavaruus *DatabaseSync* sisältää luokat *DbReader* ja *DbWriter*, jotka tarjoavat *AlarmEnginelle* sen tarvitsemat tietokantayhteydet. Toiminallisuudet on hajautettu kahteen luokkaan operaatioiden tyypin, luku tai kirjoitus, mukaan. Tämä ei ollut täysin välttämätöntä, mutta hallinnan vuoksi toiminallisuuksia haluttiin hajauttaa jonkin kriteerin perusteella.

Nimiavaruus *DataTypes* sisältää tutut käsitteet ja taulut luokkina. Nimiavaruus käännetään omaan dynaamisesti linkitettävään kirjastoon, jotta muut moduulit voivat käyttää

luokkia; käsitteellisesti luokat kuitenkin kuuluvat *AlarmEngine* moduuliin. Luokkakaaviosta huomioitavaa on abstraktin kantaluokan *Rule* omistama tilauslista: säännöllä voi käytännössä olla useita tilauksia tarkkailtavaan tapahtumasekvenssiin, joka helpottaa haastavien tilasiirtymälogiikoiden toteutusta.

### 4.5.2 Interface -moduuli

Luokkakaaviossa (Kuva 16) on kuvattu *AlarmSystem.Interface* moduulin oleellisimmat luokat ja niiden toiminallisuudet.



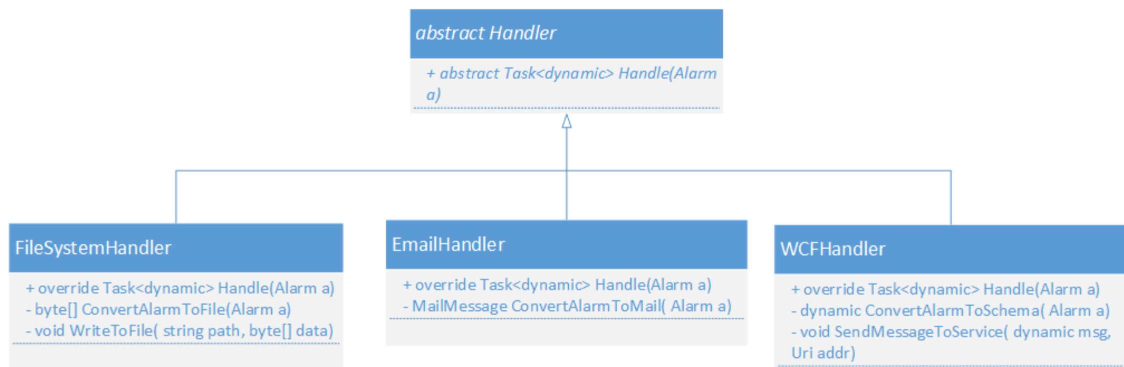

**Kuva 16.***AlarmSystem.Interface moduulin luokkakaavio*

Rajapinta *IAlarmSystem* määrittelee järjestelmän rajapinnan ulkoisille tapahtumalähteille. Luokka *AlarmSystemService* toteuttaa tämän rajapinnan ja isännoi sen WCF-palveluna *ServiceHost* luokan avulla. *AlarmSystemServiceen* saapuvat tapahtumat välitetään *MessageConverter*-luokalle asynkronisina tapahtumina. *MessageConverter* yrittää harmonisoida sanomat *Message*-luokan ilmentymiksi ja julkaisee ne *IObservable<Message>* tapahtumasekvenssinä.

### 4.5.3 Communication -moduuli

Luokkakaavio (Kuva 17) sisältää *AlarmSystem.Communication* moduulin luokat ja toiminallisuudet.

Moduulin vastuulla on *AlarmSystem*in hälytysten välitys niistä kiinnostuneille osapuolille, jotka ovat määritelty *Destination* -oliona hälytysten jäsenmuuttujina. Moduuli määrittää abstraktin kantaluokan *Handler*. Luokan funktiota *Handle* tulee kutsua, kun hälytys on aktivoitunut. Parametrinä funktio ottaa *Alarm*-luokan, joka on aktivoitunut. Paluuarvo on kääritty *Task*-luokkaan, joka hyödyntää geneeristä tyyppiparametrisointia ja mahdollistaa suoritettavan funktion ajamisen asynkronisesti [24]. Paluuarvo on määritelty dynaamiseksi, jotta periyttävät luokat voivat toimittaa kutsujille eri tietotyyppejä. Tämä ratkaisu vaatii sen, että kutsuja tietää mitä kukin *Handler* luokan toteuttaja voi palauttaa.




**Kuva 17.** *AlarmSystem.Communication moduulin luokkakaavio*

Toteutettuja käsittelijöitä ovat *FileSystemHandler*, joka tallentaa aktivoituneet hälytykset levylle, *EmailHandler,* joka lähettää aktivoituneet hälytykset sähköpostitse, sekä *WCFHandler,* joka lähettää aktivoituneet hälytykset toisille WCF-palveluille.

## 4.6 Rajapinta ulkoisille tapahtumalähteille

*AlarmSystem*in sääntöjen tilamuutoslogiikat ovat riippuvaisia ulkoisten tapahtumalähteiden tarjoamista sanomista. Tässä kohdassa kuvataan suunnitteluratkaisut näiden lähteiden liittämiseen *AlarmSystemiin* valittujen integraatiokomponenttien avulla. (Taulukko 1).

Integraatiossa voidaan hyödyntää lähettäjille helposti käyttöönotettavaa protokollaa, sovittua kirjekuorta johon lähettäjän tulee pakata viesti jonka vastaanottajan viestiadapteri purkaa ja yhdenmukaistaa järjestelmän ymmärtämään muotoon, sekä yksisuuntaista *fire-and-forget* viestintää niin, että viestin lähettäjän ei tarvitse huolehtia viestin toimituksen onnistumisesta.

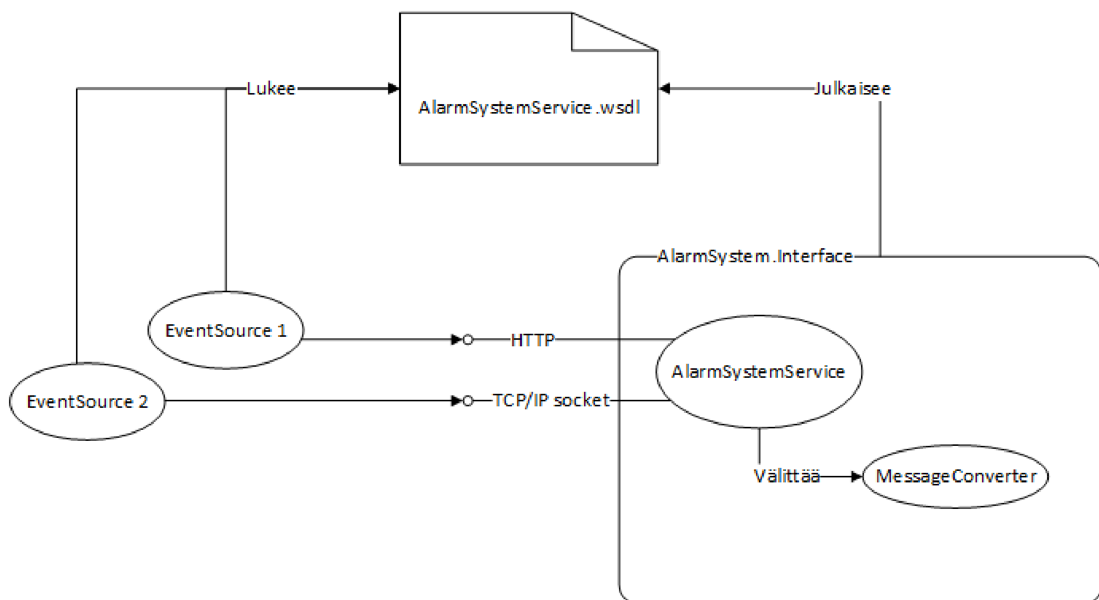
| **Vaatimus** | **Soveltuva integraatiokomponentti** |
|---|---|
| Monta yhteen. | Protokolla |
| Lähteiden lähdeviestiskeemat eroavat. | Kirjekuori, Adapteri |
| Lähteen tehokkuus ja vikasietoisuus. | Yksisuuntainen viestintä |

***Taulukko 1.*** *Soveltuvat integraatiokomponentit*

Käytettäväksi sovellusprotokollaksi valittiin SOAP, joka tarjoaa yksinkertaisen XML-muotoisen kirjekuoren sanomien pakkaamiseen. Valintaa helpotti myös toteutuksen yksinkertaisuus: *AlarmSystem.Interface* –moduulin *AlarmSystemService* voidaan toteuttaa

WCF palveluna, joka tuottaa SOAP protokollan WSDL-rajapintadokumentin automaattisesti. SOAP on myös riippumaton siirtoprotokollasta [11]: WCF palvelu voi tarjota *endpoint*it eri siirtoprotokollille, joiden yli SOAP-kirjekuoria voidaan siirtää. WCF palvelun rajapinta voidaan myös merkitä yksisuuntaiseksi, jolloin se merkitään myös generoituun WSDL-rajapintakuvaukseen. Yleinen sovellusten välisen kommunikaation siirtoprotokolla on HTTP. Se on pyyntö-vastaus protokolla, jonka avulla on mahdollista toteuttaa yksisuuntainen viestintä, mutta se ei sovellu protokollalle luonnollisesti. Järjestelmä voi tarjota myös TCP/IP-socket *endpointin*, jota käyttämällä siirrettävän tiedon *overhead* vähentyy näin ollen tehostaen tiedonsiirtoa.

Arkkitehtuurissa on otettu huomioon viestien harmonisointi Adapteri-komponentin avulla. Adapteri toteutetaan *AlarmSystem.Interface* –moduulin *MessageConverter* luokkana. Järjestelmäkaaviossa (Kuva 18) esitellään valitut komponentit *AlarmSystemin* ja tapahtumalähteiden välisessä integraatiossa. Integraation suunnittelussa nousi epäilys adapteri komponentin suorituskyvystä: ulkopuoliset tapahtumat pitää pystyä harmonisoimaan samalla nopeudella tai nopeammin kuin niitä järjestelmään saapuu. Ennaltaehkäisimme mahdollista suorituskykyongelmaa toteutuksessa sekä testausvaiheessa. Adapteri toteutettiin käsittelemään XML-dokumentteja vain muistin varassa sekä jäsentämään ne *forward-only* -lukijalla, joka ei alustuksessaan lue dokumentin puurakennetta muistiin. Dokumenttien skeemavalidointia ei tehdä niiden suhteellisen pienen koon vuoksi, sekä mahdollisten skeemamuutosten yhteensopimattomuuksien välttämiseksi [16]. Testausvaiheessa teimme suorituskyvyn profilointia vaihtelevalla syötedatalla.




**Kuva 18.** *Ulkoisten tapahtumalähteiden integrointi*

## 4.7 Konfiguraatioiden päivitys

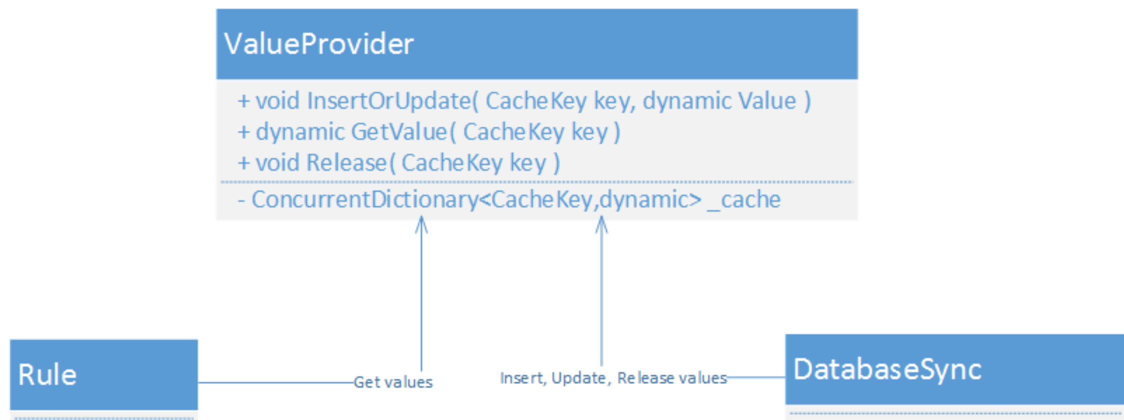
*AlarmSystemin* hälytyslogiikkaa suorittavien komponenttien konfigurointeja on pystyttävä muokkaamaan ajonaikaisesti. Järjestelmän konfiguraatioita säilytetään SQL Server tietokannassa, aikaisemmin määritellyssä tietokantaskeemassa. *AlarmSystem.AlarmEngine.DatabaseSync* –moduulin vastuulla on konfiguraatioiden synkronointi: sen tulee lukea konfiguraatiot tietokannasta määrätyin intervallein sekä muokata ohjelman ajonaikaista tilaa vastaamaan tietokantaan konfiguroituja arvoja. Arvot luetaan synkronointia varten tietokantanäkymän avulla. Tietokantanäkymä (*view*) on virtuaalinen taulu, jota ei välttämättä ole olemassa tietokannassa, mutta voidaan muodostaa olemassa olevien relaatioiden avulla [18]. Näkymä *vAlarmConfigurations* tarjoaa konfiguraatiot helposti käsiteltävässä muodossa synkronointia varten (Ohjelma 3).

```
CREATE VIEW [vAlarmConfigurations]
      AS
SELECT m.ID as MonitorableID,
      a.ID as AlarmID, a.Cooldown, a.OnCooldown, a.DestinationID,
      a.AlarmDescription, a.Priority, r.ID as RuleID, r.Name as RuleName,
      ar.State, c.RuleTypeSettingName, c.Value as SettingValue
FROM [Monitorable] as m
JOIN [Alarm] as a ON a.MonitorableID = m.ID
JOIN [AlarmRule] as ar ON [AlarmRule].AlarmID = a.ID
JOIN [RuleType] as r ON [AlarmRule].RuleTypeID = r.ID
LEFT JOIN [RuleTypeSetting] as s
      ON s.RuleTypeID = r.ID
LEFT JOIN [RuleConfiguration] as c
      ON (c.AlarmID = a.ID AND c.RuleTypeID = r.ID
            AND c.RuleTypeSettingName = s.Name)
```

***Ohjelma 3.*** *vAlarmConfigurations synkronointinäkymä*

Haasteeksi osoittautuvat järjestelmän tilalliset komponentit, ja näistä ehkä tärkeimpinä *Rule*-kantaluokasta periytetyt säännöt. Sääntöjen sisäisten tilasiirtymälogiikoiden halutaan olevan jatkuvasti ajossa reaaliaikaisen ja luotettavan monitoroinnin vuoksi, mutta sääntöspesifisiä konfiguraatioita pitää kuitenkin pystyä muuttamaan. Ratkaisuna käytettiin *caching* suunnittelumallia, jossa staattisen muuttujan omistamisen sijaan arvoja kysytään ajonaikaisesti *kätköstä* (*cache*). Kätkön tarkoitus on säilöä usein käytettävää tietoa välimuistissa, eli kätkössä, josta kätkön käyttäjä voi hakea tietoa. Kircher ja Prashantin mukaan tiedon säilöntä kätkössä vähentää kalliita resurssien uudelleenhankintaoperaatioita [17]. He myös määrittävät vaiheet, jotka tulisi suorittaa kätkön toteutuksessa: valita kätkettävät resurssit, määritellä kätkön rajapinta, määritellä poistostrategia ja varmistaa tiedon yhdenpitävyys.

Kätkön toteuttaa *AlarmEngine*-moduulin *ValueProvider*-luokka. *ValueProvideria* käytetään säilömään vain sellaista tietoa, jonka päivitys ei ole triviaalia ja vaatisi jonkin tilakoneen uudelleenalustuksen: käytännössä tämä tarkoittaa *Rule*-luokasta periytettyjen sääntöjen jäsenmuuttujien arvoja. Kätkön rajapinta on kuvattu alla olevassa kaaviossa (Kuva 19).




**Kuva 19.** *ValueProvider luokan rajapinta ja assosiaatiot*

Rajapinta mahdollistaa arvojen lisäämisen, päivittämisen, hakemisen ja poistamisen. Kaikki operaatiot suoritetaan avaimen *CacheKey* avulla. Poistostrategialla tarkoitetaan prosessia, jolla tarpeettomat avain-arvoparit vapautetaan kätköstä. *ValueProvider* -luokan funktiota *Release* kutsutaan kun *AlarmEngine.DatabaseSync* huomaa tietokantanäkymän kautta säännön poistuneen käytöstä. Tällöin tämän säännön purkuoperaatoiden ohella, vapautetaan sen *CacheKey*llä arvot myös *ValueProviderilta.* Tiedon yhdenpitävyydellä Kircher ja Prashant tarkoittavat kätkössä olevan tiedon ja alkuperäisen tietolähteen arvojen yhdenpitävyyttä [17]. He mainitsevat yhdenpitävyyden ratkaistuksi yleensä synkronoinnilla, joka on myös *AlarmSystemin* ratkaisu. Synkronointi-intervalilla voidaan säätää varmuutta yhdenpitävyydestä; *AlarmSystemin* tapauksessa konfiguraation synkronointi kätköön ei ole korkeimmalla prioriteetilla ja synkronointi-intervalli saa olla useita minuutteja.

# 5. KÄYTTÖTAPAUKSET

Tässä luvussa esitellään esitutkimusvaiheessa selvitettyjen monitorointitapojen soveltamista *AlarmSystem* monitorointijärjestelmään. Monitorointitavat toteutetaan luvussa 4 suunniteltuina sääntöinä, eli äärellisinä tilakoneina. Näiden tilakoneiden toimintaa kuvataan tilasiirtymämatriisien ja tilakonekaavioiden avulla. Säännön toteutuksen jälkeen, se voidaan sääntöohjausmallin avulla helposti kytkeä halutuille monitoroitaville osaksi olemassa olevaa hälytystä tai uutena hälytyksenä. Kohdassa 5.1 määritellään laitteen offline-tilan tunnistamisen mallinnus kahdella eri tavalla. Kohdassa 5.2 määritellään jakson yhteenvedon oikeellisuuden varmennus. Kohdassa 5.3 määritellään spesifisten käyttäjien tunnistus. Kohdassa 5.4 esitellään apusääntöjä, joita voidaan hyödyntää sääntöohjausmallissa.

## 5.1 Laite Offline

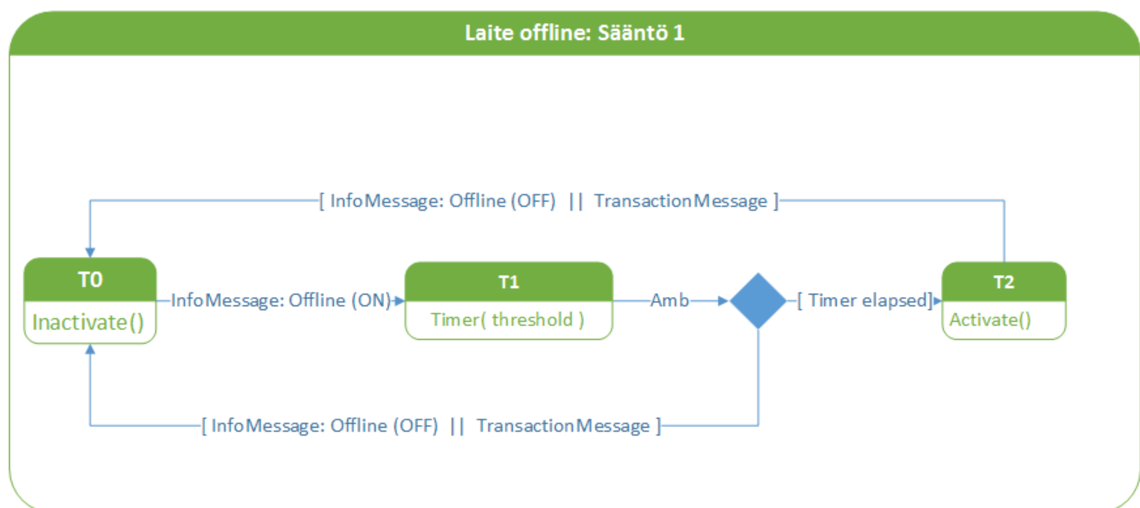
Tässä kohdassa määriteltävän monitoroinnin tarkoitus on tunnistaa laitteen offline-tila luotettavasti. Offline-tilalla tarkoitetaan sellaista tilaa, jossa laite on itse tunnistanut liiketoiminnan kannalta oleellisen komponentin olevan pois käytöstä. Tällöin laite voi lähettää sanomia normaalisti palvelimelle, mutta ei kykene normaaliin toimintaan. Tällä hetkellä monitorointia suoritetaan asiakkaan toimesta manuaalisesti. Vian tunnistamisen jälkeen, hälytys voidaan siirtää automaattisesti kenttähuollolle. Laite Offline –monitorointi on käyttötapauksista tärkein, koska virhetila pysäyttää laitteen liiketoiminnan täysin.

Monitoroinnin kriteerit perustuvat laitteen lähettämiin viesteihin ja niiden puutteeseen. Laitteen offline-tila on mahdollista huomata kahdella eri tavalla, joten monitorointi tulee mallintaa kahdella eri säännöllä, eli tilakoneella. Näitä sääntöjä ei kuitenkaan tulla käyttämään samassa hälytyksessä, koska hälytys yhdistää säännöt aina *AND*-operaattorilla ja vaatii aktivoituakseen kaikkien sen sääntöjen aktivoitumisen. Tässä kohdassa näistä säännöistä puhutaan nimillä sääntö 1 ja sääntö 2.

**Sääntö 1.** Säännön 1 tilasiirtymälogiikka kuvataan tilasiirtymämatriisilla (Taulukko 2) ja tilakonekaaviolla (Kuva 20).

| Tila | Kuvaus | Toimin-not | Mahdolliset tilasiirtymät |
|---|---|---|---|
| T0 | Monitoroitavan tila on toivottu. Kun monitoroitavalle saapuu *Info*-tyyppinen sanoma, jonka koodi kuvaa valmistajan määrittämä offline-tilaa ja status on ON, siirrytään tilaan T1. | Inaktivoi sääntö | T1 |
| T1 | Odotetaan konfiguroitava ajanjakso *Info*-tyyppistä sanomaa, joka kytkee offline-tilan pois (status: OFF) tai *Transaction*-tyyppistä sanomaa. Jos toinen sanomista saapuu konfiguroidun ajanjakson sisällä, siirrytään välittömästi tilaan T0. Jos ajanjakso kuluu ilman näitä sanomia, siirrytään tilaan T2. | | T0, T2 |
| T2 | Odotetaan *Info*-tyyppistä sanomaa, joka kytkee offline-tilan pois (status: OFF) tai *Transaction*-tyyppistä sanomaa. Jos sanoma saapuu, siirrytään tilaan T0. | Aktivoi sääntö | T0 |

***Taulukko 2.*** *Tilasiirtymäkaavio, Laite Offline Sääntö 1*

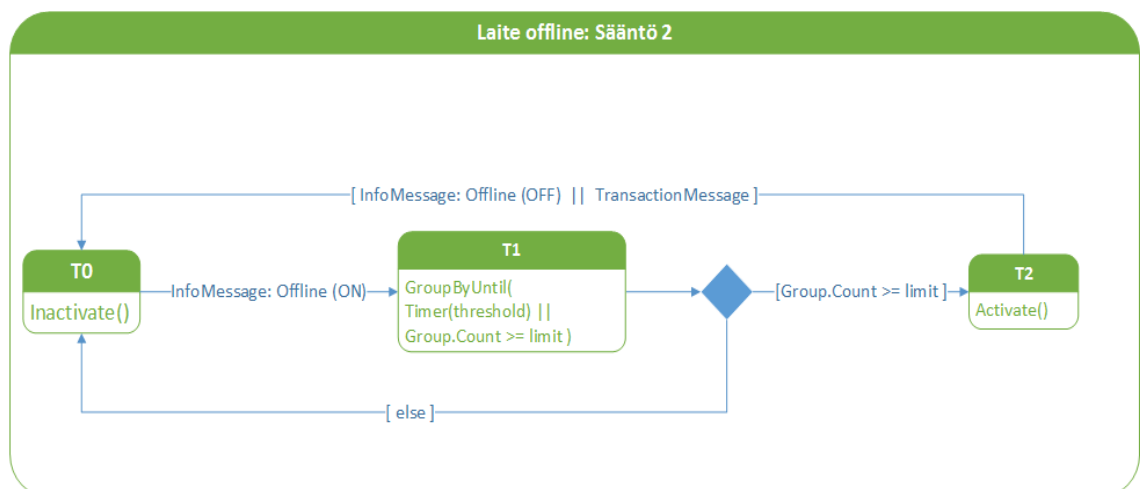



**Kuva 20.** *Tilakonekaavio Laite Offline Säännön 1 tilasiirtymälogiikalle.*

**Sääntö 2**. Säännön 2 tilasiirtymälogiikka kuvataan tilasiirtymämatriisilla (Taulukko 3) ja tilakonekaaviolla (Kuva 21).

| **Tila** | **Kuvaus** | **Toimin-not** | **Mahdolliset tilasiirtymät** |
|---|---|---|---|
| T0 | Monitoroitavan tila on toivottu. Kun monitoroitavalle saapuu *Info*-tyyppinen sanoma, jonka koodi kuvaa valmistajan määrittämä offline-tilaa ja status on ON, siirrytään tilaan T1. | Inaktivoi sääntö | T1 |
| T1 | Odotetaan konfiguroitu määrä vastaavien sanomien status vaihteluja ON ja OFF välillä, konfiguroitavan ajanjakson sisällä. Jos vaihtelujen määrä täyttyy ennen ajanjakson päättymistä, siirrytään tilaan T2. Jos vaihteluja ei tule konfiguroitua määrää ajanjakson sisään, siirrytään tilaan T0. | | T0, T2 |
| T2 | Sääntö aktivoidaan. Odotetaan *Info*-tyyppistä sanomaa, joka kytkee offline-tilan pois (status: OFF) tai *Transaction*-tyyppistä sanomaa. Jos sanoma saapuu, siirrytään tilaan 0. | Aktivoi sääntö | T0 |

***Taulukko 3.*** *Laite Offline Säännön 2 tilasiirtymämatriisi.*



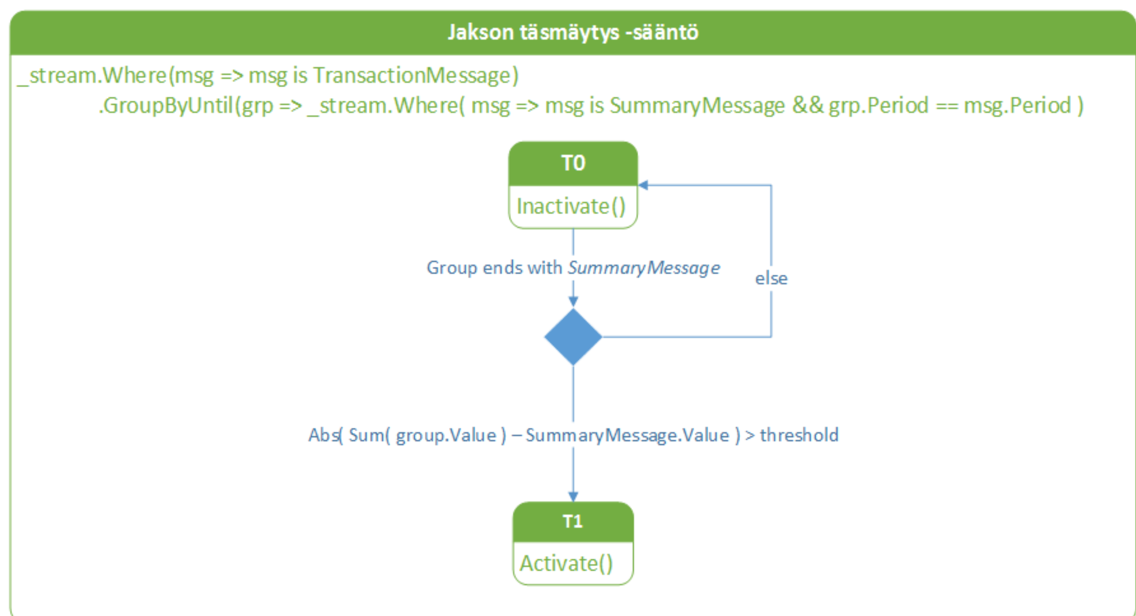

**Kuva 21.** *Tilakonekaavio Laite Offline Säännön 2 tilasiirtymälogiikalle.*

## 5.2 Jakson täsmäytys

Tämän kohdan monitoroinnin tarkoitus on tunnistaa eroavaisuudet jakson *Transaction*-tyyppisten sanomien ja jakson päättävän *Summary*-tyyppisen sanoman arvoissa. Monitoroitavalle saapuvat *Transaction*-tyyppiset sanomat ryhmitellään jatkuvasti (tilasta riippumatta) jakson tunnisteen mukaan. Jaksojen *Transaction*–tyyppisten sanomien arvojen summia pidetään muistissa. Kun jakson päättävä *Summary*-tyyppinen sanoma saapuu, verrataan yhteenvedon ilmoittamaa jakson kokonaissummaa ja jakson *Transaction*-sanomista kumuloitua summaa toisiinsa. Monitorointi voidaan toteuttaa yhdellä säännöllä, joka on esitelty tilasiirtymämatriisissa (Taulukko 4) ja tilakonekaaviossa (Kuva 22).

| Tila | Kuvaus | Toimin-not | Mahdolliset tilasiirtymät |
|---|---|---|---|
| T0 | Monitoroitavan tila on toivottu. Jakson täsmäyttävän *Summary*-tyyppisen sanoman saapuessa verrataan jakson täsmäyttävän *Summary*-tyyppisen sanoman summaa jakson *Transaction*-tyyppisten sanomien summaan. Jos eroavaisuus on yli konfiguroidun raja-arvon, siirrytään tilaan T1. | Inaktivoi sääntö | T1 |
| T1 | Huomattiin eroavaisuus. Uuden jakson täsmäyttävän *Summary*-tyyppisen sanoman saapuessa siirrytään tilaan T0. | Aktivoi sääntö. | T0 |

***Taulukko 4.*** *Täsmäytys-säännön tilasiirtymämatriisi*



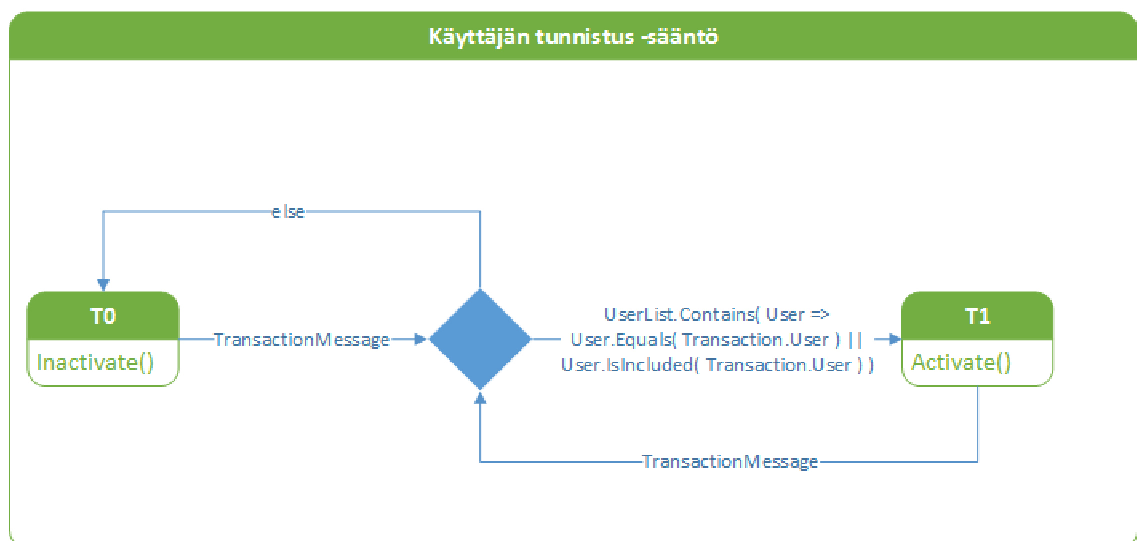

**Kuva 22.** *Tilakonekaavio Jakson täsmäytys –säännön tilasiirtymälogiikalle*

## 5.3 Käyttäjän tunnistus

Tämän kohdan monitoroinnin tarkoitus on huomata tapahtumat tietyillä käyttäjätunnisteilla ja ilmoittaa niistä kiinnostuneille osapuolille. Tämä voidaan toteuttaa kahdella eri tavalla: vertaamalla *Transaction* -sanoman käyttäjätunnistetta konfiguroituun käyttäjätunnistelistaan tai vertaamalla käyttäjätunnistetta *wildcard* merkeillä konfiguroituun tunnisteavaruuteen. Käyttäjän tunnistus -säännön tilasiirtymälogiikat esitellään tilasiirtymämatriisissa (Taulukko 5) ja tilakonekaaviossa (Kuva 23).

| **Tila** | **Kuvaus** | **Toiminnot** | **Mahdolliset tilasiirtymät** |
|---|---|---|---|
| T0 | Monitoroitavan tila on toivottu. Vastaanotettaessa *Transaction*-tyyppinen sanoma, jonka tunniste on konfiguroidulla listalla tai tunnisteavaruudella, siirrytään tilaan T1. | Inaktivoi sääntö | T1 |
| T1 | Käyttäjä tunnistettu. Minkä tahansa *Transaction*-tyyppisen sanoman saapuessa siirrytään tilaan T0. | Aktivoi sääntö | T0 |

***Taulukko 5.*** *Käyttäjän tunnistus –säännön tilasiirtymämatriisi*




**Kuva 23.** *Tilakonekaavio Käyttäjän tunnistus –säännön tilasiirtymälogiikalle*

## 5.4 Apusäännöt

Hälytysten toimintaa voidaan ohjata sääntöohjausmallin mukaisesti. Lisäksi on määritelty hälytysten aktivoituvan, kun sille määrätty joukko sääntöjä ovat samanaikaisesti aktiivisessa tilassa. Ominaisuuden tueksi, *AlarmSystemiin* toteutettiin käyttötapausten sääntöjen lisäksi yksinkertaisia apusääntöjä, joita voidaan tarvittaessa ottaa käyttöön. Tässä kohdassa esitellään apusäännöt *NeverActiveRule, SilentRule* ja *NotSilentRule,* jotka on toteuttu *Rule*-luokan perivinä luokkina.

*NeverActiveRule* on sääntö, joka alustaa tilansa epäaktiiviseksi, ei kuuntele tapahtumia eikä näin ollen ikinä aktivoidu. Hyödyllinen, jos toisen säännön toimintaa halutaan testata hälytyksessä, mutta hälytyksen ei haluta aktivoituvan.

*SilentRule* on sääntö, joka aktivoituu jos sen omistavan hälytyksen monitoroitavalta kohteelta ei ole saapunut tapahtumia nykyhetkestä konfiguroidun aikavälin sisällä. Hyödyllinen, jos hälytyksen aktivointikriteereihin halutaan lisätä monitoroitavan kohteen hiljaisuus.

*NotSilentRule* on sääntö, joka aktivoituu jos sen omistavan hälytyksen monitoroitavalta kohteelta on saapunut tapahtumia nykyhetkestä konfiguroidun aikavälin sisällä. Hyödyllinen, jos hälytyksen aktivointikriteereihin halutaan lisätä monitoroivan kohteen aktiivisuus.

# 6. TOTEUTUS JA ARVIOINTI

Tässä luvussa esitellään toteutusvaiheen teknologiavalintoja, toimintatapoja sekä tuotantoon vientiä. Kohdassa 6.1 esitellään tiettyyn käyttötarkoitukseen valittu teknologia ja sen perustelu. Kohdassa 6.2 esitellään toteutusvaiheessa suoritettua suorituskyvyn profilointia, jonka kandidaatit valittiin suunnitteluvaiheessa arvioinnin perusteella. Kohdassa 6.3 esitellään toteutusvaiheessa käyttöönotettu yksikkötestausprosessi sekä reaktiivisten aikariippuvaisten komponenttien testaukseen suunnattua *TestSchedulerin* käyttö. Kohdassa 6.4 määritellään mitä *AlarmSystemin* tapauksessa palveluna isännöinti tarkoittaa ja miten se on tehty. Kohdassa 6.5 luetellaan ja arvioidaan suunnittelun ja toteutuksen aikana syntyneitä jatkokehitysideoita. Lisäksi kohdassa 6.6 arvioidaan järjestelmän määrittely, suunnittelu ja toteutusvaiheiden ratkaisuja.

## 6.1 Teknologiavalinnat

*AlarmSystemin* toteutusteknologioiden valinnat perustuvat yrityksen sisäisiin toimintapoihin, palvelinympäristöön sekä laatuvaatimuksiin. Eatech toteuttaa ratkaisuja pääasiassa Microsoftin työkaluilla. Lisäksi kohdeympäristön palvelinkoneet käyttävät Microsoft Windows tuoteperheen käyttöjärjestelmiä. Toteutus tehtiin hyödyntäen Microsoft .NET Framework -sovelluskehystä. Käytettävä ohjelmointikieli oli C# ja kehitysympäristönä toimi Microsoft Visual Studio Premium 2013.

Reactive Extensions –kirjasto (*Rx)* tarjosi työkaluja asynkroniseen tapahtumapohjaiseen viestinvälitykseen ja julkaisija-tilaaja -mallin toteutukseen [4]. Rx-kirjasto tarjoaa LINQ-kyselyiden kirjoituksen tarkkailtaviin tapahtumasekvensseihin, joka mahdollistaa joukko-opin hyödyntämisen tapahtumien käsittelyssä vaivattomasti.

ORM-siltaukseksi valittiin *Dapper*, sillä komponentti on tehokas varsinkin SELECT-lukuoperaatioiden tulosten kääntämisessä olioiksi [5], joista konfiguraatioiden tietokantasynkronointi pääasiassa koostuu.

Relaatiotietokannanhallintajärjestelmänä (*RDBMS*) toimi Microsoft SQL Server 2012, koska muutkin palvelinverkon tietokannat on rakennettu sen varaan ja tietokannanhallintajärjestelmälle on rakennettu automaattiset ylläpitotoimet.

## 6.2 Suorituskyvyn profilointi

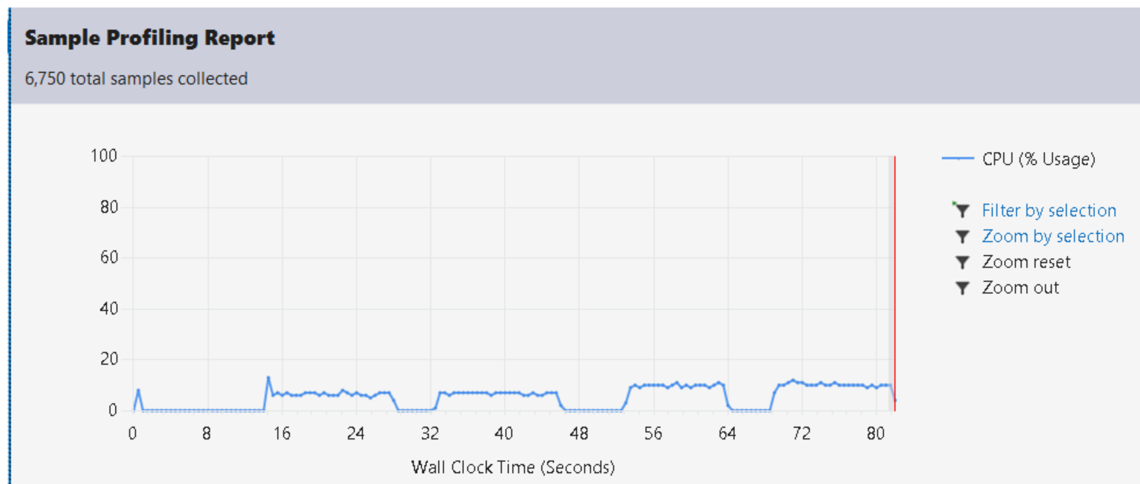
Järjestelmän yhdeksi laatuvaatimukseksi määriteltiin skaalautuvuus ja määrättiin se saavutettavaksi toteuttamalla sanomien harmonisointi ja sääntöjen tilasiirtymälogiikat mahdollisimman tehokkaiksi laskentatehon sekä muistinkulutuksen suhteen. Toteutustekniikoiden valintojen lisäksi järjestelmälle ajettiin myös kuormitustestitapauksia CPU-ajan käytön ja muistinkulutuksen suhteen. Kaikki profiloinnit suoritettiin neliytimisellä Intel i5-4200M, 2.5GHz prosessorilla, Windows 8.1 Enterprise -käyttöjärjestelmällä.

Laitteiden tuottamien XML-dokumenttien voidaan arvioida olevan kooltaan 5-20 kilotavua ja niiden maksimitaajuudeksi voidaan arvioida 20 tapahtumaa sekunnissa. Arvioiden pohjalta voidaan suorittaa kuormitustestaus, jossa profiloidaan adapterikomponentti *MessageConverterin* suorituskykyä ja mitataan sanomien läpäisyastetta. *MessageConverterin* kuormitustestitapaukset ja tulokset ovat taulukoitu alle (Taulukko 6).

| **Syöte** | **Oletettu tulos** | **Ajanhetki (Kuva 15)** | **Lopputulos** |
|---|---|---|---|
| 5kt dokumentteja. 2Hz taajuudella. | CPU-ajassa ei näkyvää piikkiä. 100 % läpäisy. | 15-27s | CPU-ajassa ei näkyvää piikkiä. 100 % läpäisy. |
| 5kt dokumentteja 20Hz taajuudella. | CPU-ajassa näkyvä piikki. 100% läpäisy | 33-46s | CPU-ajassa näkyvä piikki. 100% läpäisy |
| 20kt dokumentteja. 2Hz taajuudella. | CPU-ajassa näkyvä piikki. 100% läpäisy | 52-64s | CPU-ajassa ei näkyvää piikkiä. 100 % läpäisy. |
| 20kt dokumentteja 20Hz taajuudella. | Säie syö yhden prosessorin CPU-ajan kokonaan. Läpäisy <100% | 70-82s | CPU-ajassa ei näkyvää piikkiä. 100% läpäisy. |

***Taulukko 6.*** *MessageConverter -kuormitustestitapaukset*

Kuvaajassa (Kuva 24) on esillä suorittimen profilointikuvaaja taulukon (Taulukko 6) mukaisessa järjestyksessä: X-akselilla aika, Y-akselilla CPU-ajankäyttö prosentteina. Kuvasta nähdään, että suorittimen käyttöaika pysyi lähes vakiona, noin 10% luokassa, riippumatta syötedokumentin koosta tai lähetystaajuudesta. Tuloksista voidaan päätellä XML-dokumenttien käsittelyn olevan tarpeeksi tehokasta, eikä sen tulisi muodostaa pullonkaulaa suorituskykyyn.




**Kuva 24.** *MessageConverter CPU-ajan profilointi*

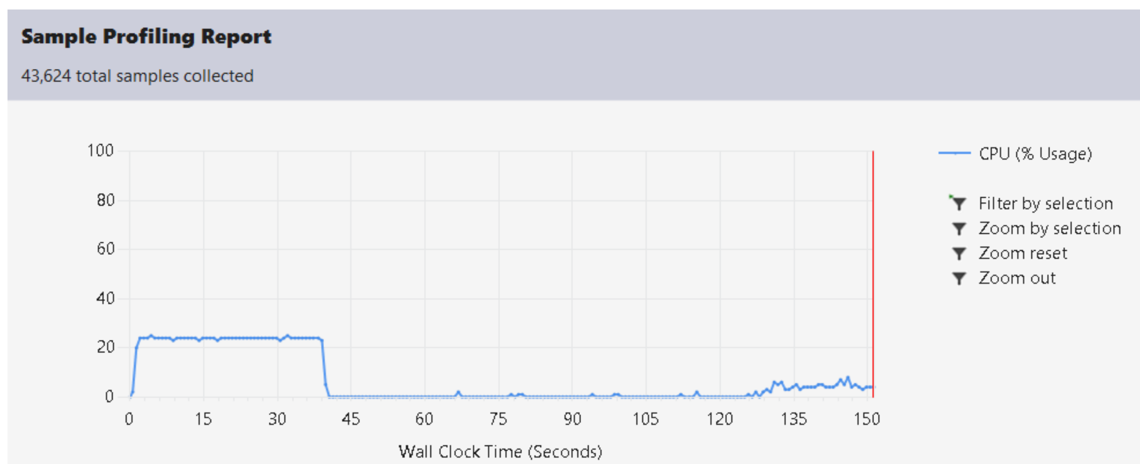
Sääntöjen suorituskyvyn profilointia suoritettiin kahdella tavalla. Ensiksi miten muistinkulutus ja CPU-aika suhtautuvat kasvavaan määrään sääntöjä, ja näin ollen julkaistun tapahtumasekvenssin tilauksiin, ja toiseksi miten sääntöjen muistinkulutus suhtautuu pitkän ajanjakson tilakoneen pyörittämiseen. Testitapaukset laadittiin arvioitujen sääntöjen määrän ja sanomataajuuden perusteella. Tapahtumasyötteenä käytettiin *MessageConverter*-luokan kuormitustestitapausten maksimia, 20 kilotavun dokumentteja vaihtelevalla taajuudella Testaukseen valittiin Käyttäjän tunnistus –sääntö. Testit ja niiden tulokset ovat esitetty taulukossa (Taulukko 7).

10 000 säännön alustus tietokannasta ja tietokantasynkronointi kesti testeissä 30-60 sekuntia. Kuvaajassa (Kuva 25) tämä näkyy ajanjaksona 0-40 sekunnin aikana, jolloin yksi CPU-ydin käytetään kokonaan. Ajanjaksolla 60-120s sääntöjä kuormitettiin 2Hz taajuudella ja ajanjaksolla 130-145s 20Hz taajuudella.

Tulokset osoittavat järjestelmän skaalautuvan odotettua paremmin sääntöjen määrään ja tapahtumataajuuteen. Säännöt varasivat hieman oletettua enemmän muistia, mutta ajonaikainen CPU-ajan käyttö ja muistinkulutus ovat hyväksytyissä rajoissa. Tärkein huomio kuormitustestistä on tietokantasynkronoinnin suorituskyky; sääntöjen alustukseen tarvittava CPU-aika kasvaa suhteessa niiden määrään. Tämä tulee huomioida tuotantoympäristön tietokantasynkronointi-intervallia asettaessa. On otettava huomioon että kuormitustestiympäristössä työtä suoritettiin vain yhdellä ytimellä. Tuotantoympäristö voi mahdollisesti suorittaa työtä useammalla ytimellä. Tähän ei kuitenkaan otettu kantaa toteutustekniikoiden avulla.

| Säännöt | Oletettu tulos | Lopputulos |
|---|---|---|
| 10 000 sääntöä, 2Hz taajuudella | CPU-aika 10-20%<br><br>Muistinkulutus: Vakio (10-15MB) + (10-20MB) | CPU-aika < 10%.<br><br>Muistinkulutus: 60-70MB |
| 10 000 sääntöä, 20Hz taajuudella | CPU-aika 20-50%<br><br>Muistinkulutus: Vakio (10-15MB) + (10-20MB) | CPU-aika < 10%.<br><br>Muistinkulutus: 60-70MB |

***Taulukko 7.** AlarmEngine.Rules kuormitustestitapaukset*




**Kuva 25.** *AlarmEngine.Rules, CPU-ajan profilointi*

## 6.3 Yksikkötestaus ja ajan hallinta

Yksikkötestauksella pyrittiin osittain täyttämään kaksi järjestelmälle asetettua laatuvaatimusta: järjestelmän luotettavuus ja vikasietoisuus. Toteutuksen alkuvaiheessa otettiin tavaksi kirjoittaa yksikkötestit kehitysympäristön tarjoamaan *Visual Studio Unit Test Projectiin*. Ominaisuus mahdollistaa yksikkötestien koostamisen saman projektin alle sekä niiden vaivattoman ajamisen, debuggaamisen ja kattavan raportoinnin [6]. Yksikkötestit kirjoitettiin käyttötapauskohtaisesti: pyrimme luomaan erilaisia skenaarioita, joista testattavan ominaisuuden tulee selviytyä tuotantoympäristössä. Tällä tavalla yksikkötestien koodikattavuus on vaihteleva, mutta järjestelmän oikeellinen toiminta saadaan varmennettua. Kehitysvaiheessa yksikkötestien koodikattavuus vaihteli 70 ja 85 prosentin välillä. Yksikkötestejä oli yhteensä 13 kappaletta, joista suurin osa oli *AlarmSystem.AlarmEngine* -moduulin sääntöjen ja tietokantasynkronoinnin testaukseen. Yksikkötestauksen kat-

tavuutta olisi voitu parantaa luomalla skenaariotapausten rinnalle hienojakoisempia testitapauksia, jotka olisi kohdistettu yksittäisiin luokkiin tai jopa funktioihin. Resurssien ja aikataulun vuoksi, hienojakoisempia testejä ei kirjoitettu kaikille luokille.

Tapahtumien lisäksi yleinen muuttuja reaktiivisissa järjestelmissä on aika. Tämä johtuu reaktiivisten järjestelmien luonteesta sekä siitä mihin ympäristöön tai ongelmanratkaisuun niitä usein tarvitaan. Myös *AlarmSystemin* sääntöjen liiketoimintavaatimukset ovat usein aikariippuvaisia, sillä säännön tilasiirtymälogiikka voi olla riippuvainen esimerkiksi kahden tapahtuman aikavälistä tai tietyn aikavälin sisällä ilmenevien tapahtumien lukumäärästä. Aikariippuvaisten komponenttien testaus osoittautuu usein ongelmalliseksi: jos aikaväli säädetään testauksen kannalta käytännölliseksi, ei välttämättä saada varmuutta siitä, että järjestelmä toimii täysin yhtenäisesti myös pitkillä aikaväleillä. Rx-kirjaston aikariippuvaisia komponentteja on mahdollista testata *ReactiveTest*-kirjaston tarjoamalla skeduloijalla *TestScheduler*, joka toteuttaa *IScheduler*-rajapinnan. Rx-kirjaston aikariippuvaiset funktiot on kuormitettu hyväksymään parametrinä skeduloijan, joka määrittää koska ja miten työtä tehdään. *TestScheduler* litistää pitkät aikavälit helposti testattaviksi, säilyttäen kuitenkin saman rinnakkaisuuden mitä funktio normaalilla skeduloijalla käyttäisi [26]. *TestScheduler*in ansiosta oli mahdollista kirjoittaa yksikkötestejä luonnollisille aikaintervalleille ja testata sääntöjen tilasiirtymälogiikan pitkän aikavälin käyttäytymistä. *TestScheduler* mahdollistaa myös simuloitujen tapahtumasekvenssien luomisen yksikkötesteihin, joita olisi muuten työläs luoda.

Ohjelma 4 esittää hieman yksinkertaistettuna reaktiivisen säännön yksikkötestitapausta. Yksikkötestin tarkoitus on varmistaa säännön oikea käyttäytyminen simuloidulla tapahtumasekvenssillä. Aluksi luodaan simuloitu tapahtumasekvenssi funktiolla *CreateHotObservable*, jonka parametreinä määritellään mitä sanomia järjestelmään saapuu ja millä ajanhetkellä kyseiset sanomat saapuvat. Sen jälkeen tapahtumalähde kytketään testattavaan sääntöön. Testin onnistuminen tarkistetaan luomalla testattavan säännön tilamuutostapahtumista tarkkailtava tapahtumalähde ja assertoimalla tämän lähteen toteutunutta viestijoukkoa odotettuihin viesteihin, sen jälkeen kun *TestScheduler*ia on ajettu eteenpäin määritelty aikaväli funktiolla *AdvanceBy*.

```
var scheduler = new TestScheduler();

ITestableObservable<StatusMessage> statusSource =
      scheduler.CreateHotObservable(
                OnNext(TimeSpan.FromSeconds(100).Ticks,
                    new StatusMessage()),
                OnNext(TimeSpan.FromSeconds(120).Ticks,
                    new StatusMessage()),
                OnNext(TimeSpan.FromSeconds(200).Ticks,
                    new StatusMessage()));

ITestableObservable<TransactionMessage> trxSource =
      scheduler.CreateHotObservable(
                OnNext(TimeSpan.FromSeconds(140).Ticks,
                    new TransactionMessage()),
                OnNext(TimeSpan.FromSeconds(253).Ticks,
                    new TransactionMessage()));

//Simulated event stream
IObservable<Message> source =
      Observable.Merge(scheduler, new IObservable<Message>[]
      {trxSource, statusSource});

var rule = new OfflineRule(1, TimeSpan.FromSeconds(51), source, scheduler);

ITestableObserver<bool> results = scheduler.CreateObserver<bool>();

Observable.FromEventPattern
      <PropertyChangedEventHandler, PropertyChangedEventArgs>(
                h => rule.PropertyChanged += h,
                h => rule.PropertyChanged -= h)
          .Subscribe(i => results.OnNext(rule.GetState()));

scheduler.AdvanceBy(TimeSpan.FromSeconds(300).Ticks);

results.Messages.AssertEqual(
                OnNext(TimeSpan.FromSeconds(251).Ticks, true),
                OnNext(TimeSpan.FromSeconds(253).Ticks, false));
```

***Ohjelma 4.*** *Reaktiivisen säännön yksikkötestitapaus*

## 6.4 Järjestelmän isännöinti palveluna

*AlarmSystem* on määritelty toteutettavaksi palveluna. Määrittelyn termi palvelu konkretisoituu tapauksessamme kahdeksi teknologiavalinnaksi: Windows Communication Foundation (WCF) -palveluksi sekä Windows-palveluksi.

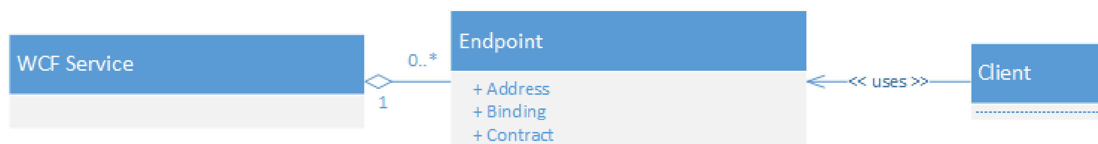
Windows Communication Foundation on sovellusten välisen kommunikaation rakentamiseen suunnattu kehys, mikä mahdollistaa asynkronisen viestinvälityksen sovellusten *endpoint*ien välillä [27]. Kuva 26 havainnollistaa WCF *endpointin* koostuvan kolmesta määriteltävästä ominaisuudesta: sopimuksesta (*contract),* sidonnasta (*binding*) ja osoitteesta (*address).* Sopimus on rajapintakuvaus, joka sisältää rajapinnan kautta käytettävät toiminallisuudet. Sidonta määrittelee protokollan minkä avulla *endpointiin* voidaan ottaa

yhteyttä. Osoite on URI-tunnus, joka määrittelee *endpointin* sijainnin. *AlarmSystem*in tapauksessa *Interface*-moduulin *AlarmSystemService*-luokka itseisännöi WCF-palvelua *ServiceHost*-luokan avulla. Palvelun avulla ulkoiset tapahtumalähteet voivat lähettää tapahtumia *AlarmSystemille*. Ohjelma 5 havainnollistaa WCF-palvelun isännöintiin tarvittavan konfiguraation ja ohjelmoinnin *AlarmSystemissä*. Kuten on suunniteltu, *AlarmSystem* tarjoaa *endpointit* kahdelle eri protokollalle, ja tapahtuman lähetys on merkitty rajapintaan yksisuuntaiseksi.

Windows-palvelu on Windows-käyttöjärjestelmän tarjoama ominaisuus hallinnoida sovelluksen suoritusta *Service Control Managerin* alaisuudessa. Windows-palveluksi määritelty sovellus toteuttaa määritellyn rajapinnan, jonka avulla *Service Control Manager* voi hallinnoida sovelluksen suoritusta esimerkiksi käynnistämällä sen aina käyttöjärjestelmän käynnistyksen yhteydessä [28]. *AlarmSystemin* asentaminen Windows-palveluksi mahdollistaa siirtämisen vastuun sovelluksen suorituksesta käyttöjärjestelmälle, joka vähentää ylläpitotoimien määrää.

```
<!-- App.config -->
<service name="AlarmEngine.Interface.AlarmSystemService">
    <host>
      <baseAddresses>
            <add name=”http”
            baseAddress="http://localhost:3333/AlarmSystemInterface" />
            <add name=”tcp”
            baseAddress=”net.tcp://localhost:3334/AlarmSystemInterface” />
      </baseAddresses>
     </host>
     <endpoint address=”http” binding="basicHttpBinding"
               contract="AlarmEngine.Interface.IAlarmSystem" />
     <endpoint address=”tcp” binding=”netTcpBinding”
               contract=”AlarmEngine.Interface.IAlarmSystem” />
</service>

// IAlarmSystem.cs
[ServiceContract(Name = "AlarmSystem”)]
public interface IAlarmSystem
{
   [OperationContract(IsOneWay = true)]
   void SendEvent( string xml );
}

// AlarmSystemService.cs
public class AlarmSystemService : IAlarmSystem
{
    private ServiceHost _host = new ServiceHost(typeof(AlarmSystemService));
    _host.Open();
}
```

***Ohjelma 5.*** *WCF-palvelun itseisännöinti AlarmSystemissä*




**Kuva 26.** *WCF-kommunikaatio palvelun ja asiakkaan välillä*

## 6.5 Jatkokehitysideat

*AlarmSystemin* suunnittelu- ja toteutusvaiheiden aikana ilmeni jatkokehitysideoita järjestelmälle. Tässä kohdassa listataan ja arvioidaan näitä ideoita, joita ei kuitenkaan vielä toteutettu.

***AlarmSystem.DataTypes* -kirjaston automaattinen uudelleenlataus.** Yksi korkean suorituskyvyn kriteereistä on järjestelmän muistinvarainen toiminta. Järjestelmän täydellistä uudelleenalustusta halutaan välttää tilatiedon menetyksen vuoksi. Kun uusia sääntöjä toteutetaan, niiden käyttöönotto vaatii *AlarmSystem.DataTypes* dynaamisesti linkitetyn kirjaston uudelleenlatauksen. Jatkokehitysidea on, että tämä kirjasto ladattaisiin käyttöön automaattisesti sen muuttuessa, määritellyin intervallein tai käyttäjän syötteestä. Ominaisuuden etu olisi uusien sääntöjen käyttöönotto järjestelmässä ilman tilatiedon menetystä.

**Tapahtumalähteiden integrointi jonopalvelun avulla.** Ulkoisten tapahtumien täysin luotettavan toimituksen takaamiseksi tulisi ottaa käyttöön jatkuvasti saavutettava jonopalvelu. Jatkuvasti saavutettava järjestelmä voidaan määritellä sellaiseksi, joka on toiminnassa 24 tuntia päivässä, seitsemän päivää viikossa ilman suunniteltuja tai suunnittelemattomia katkoja [25]. Ominaisuuden etu olisi tapahtumien varma toimitus *AlarmSystem*ille. Ominaisuuden huonoina puolina voidaan nähdä jonotusjärjestelmän toteutuksen ja käyttöönoton suuri työmäärä.

**Rajapinta konfigurointiin.** *AlarmSystem*in konfiguraatioiden päivitys on määritelty tehtävän tietokannasta suorituksen aikana. Tietokantaoperaatiot eivät kuitenkaan ole optimaalinen tapa tarjota mahdollisuus järjestelmän konfigurointiin. Jatkokehitysidea oli toteuttaa ohjelmoitava rajapinta *AlarmSystemiin*, joka tarjoaisi *CRUD*-operaatiot järjestelmän käsitteille. Ominaisuuden etu olisi tietokantaoperaatioiden abstrahointi käyttäjiltä sekä järjestelmän laajennusmahdollisuus, esimerkiksi mahdollisuus toteuttaa konfigurointikäyttöliittymä.

## 6.6 Arviointi

Valituista suunnittelumalleista hyödyllisimmäksi osoittautui julkaisija-tilaaja -malli. Rx-kirjaston mallin toteutus osoittautui tehokkaaksi ja sanaston opettelun jälkeen helppokäyttöiseksi. Ulkoisten tapahtumien haarauttaminen niitä koskeville hälytyssäännöille onnistui helposti. Sääntöohjausmallia ei vielä alkuvaiheessa hyödynnetty täysin, koska käyttötapausten hälytyskriteerit olivat kuvattavissa yhdellä säännöllä. Mallista oletetaan

olevan hyötyä tulevaisuudessa, kun *AlarmSystemiin* luodaan uusia monitorointitoteutuksia. Reaktiivinen tapahtumakeskeinen malli edistää *AlarmSystemin* reaaliaikaisuutta ja skaalautuvuutta.

Asetetuista laatuvaatimuksista parhaiten täyttyvät reaaliaikaisuus, skaalautuvuus sekä laajennettavuus. Konfiguroitavuutta voitaisiin kehittää toteuttamalla rajapinta ja käyttöliittymä konfiguraatioiden säätöön. Koska kaikki järjestelmän moduulit ovat riippuvaisia toisistaan, vikasietoisuuden kehittämiseksi tulisi toteuttaa automaattinen *AlarmSystemin* vikatilanteiden ilmoitus ylläpidolle. Luotettavuutta pyritään kehittämään vielä käyttöönoton jälkeen analysoimalla luotuja hälytyksiä ja varmentamalla tapahtumasekvenssi mikä johti hälytykseen. Tällä tavalla voidaan löytää *false positive* -hälytyksiä, mutta ei *true negative* -hälytyksiä. Huomioon on kuitenkin otettava luotettavuus reaktiivisissa järjestelmissä; on lähes mahdotonta varautua kaikkiin mahdollisiin tapahtumasekvensseihin.

Käyttötapausten toteutuksessa ilmeni ongelmia vain jakson täsmäytys -säännön kanssa: jos jaksolle ei ikinä saavu jakson päättävää *Summary*-sanomaa, ryhmä on elossa koko ohjelman suorituksen ajan aiheuttaen mahdollisesti nousevaa muistinkulutusta. Tässä tapauksessa ongelma tulisi ratkaista investoimalla varmaan toimitukseen jonotuspalvelun avulla tai luottamaan toimintaympäristön tuottavan dataa kuten määritelty.

Suurimmat haasteet käyttöönottovaiheessa ilmenivät ulkoisten tapahtumalähteiden integroinnissa *AlarmSystemiin*. Adapteri-mallin toteutus *forward-only* lukijalla osoittautui tuotannossa virhealttiiksi paikaksi, johtuen viestiskeemojen odotettua suuremmasta vaihtelevuudesta. Vaihtoehtoinen tapa integraation toteutuksessa olisi ollut käyttää *Data Transfer Objecteja* (DTO), eli valmiiksi määriteltyjä luokkia, joiden avulla ulkopuolisia tapahtumia olisi siirretty tapahtumalähteiltä *AlarmSystemille*. DTO:iden toteutus olisi vaatinut enemmän työtä tapahtumalähteiden toimesta jota haluttiin välttää integraatiosuunnitelmassa. Lisäksi ulkoisten tapahtumalähteiden reaaliaikaisuus osoittautui hyvin vaihtelevaksi. Esimerkiksi tapaukset, joissa ulkoinen tapahtumalähde lähettää tietoa useita kymmeniä minuutteja myöhässä, aiheutti odottamattomia luotettavuusongelmia *AlarmSystemissä*. Tämä käyttäytyminen olisi voitu huomata aikaisemmin *iteratiivisemmalla* kehitysmallilla, jossa tuotantoympäristöön vientejä olisi tehty iteraatiokierrosten aikana. Tämä olisi kuitenkin voinut tuoda uusia ongelmia tuotantoonviennin monimutkaisuuden ja työläyden vuoksi. Pian käyttöönoton jälkeen sääntöjä jatkokehitettiin ottamaan mahdollisimman hyvin huomioon tuotantoympäristön ominaisuudet, jotka vaikuttavat sääntöjen tilasiirtymälogiikoihin, jonka jälkeen *AlarmSystemin* luotettavuus parani huomattavasti.

# 7. YHTEENVETO

Tässä työssä käsiteltiin hajautetun laitejärjestelmän monitorointijärjestelmän suunnittelua ja toteutusta. Monitorointijärjestelmän nimeksi tuli *AlarmSystem.* Työ esittelee ratkaisun tarkasti: käsiteltävät aiheet ovat muun muassa valitut suunnittelumallit, järjestelmäarkkitehtuuri, suunnittelua ohjaavat päätökset ja laatuvaatimukset, käsitteiden toteutus luokkina ja tietokantoina, integraatiot, käyttötapauksien määrittely tilakoneina, teknologiavalinnat ja testimenetelmät, sekä vienti tuotantoon palveluksi. Työ siis esittelee ratkaisun kokonaisuudessaan, jättämättä mitään vaihetta käsittelemättä. Lukija voi työn perusteella ymmärtää reaktiivisen tapahtumakeskeisen mallin perusteet, sisäistää ohjelmistokehitysprosessin vaiheet sekä ymmärtää ja arvioida valittuja ratkaisuja. Suureksi osaksi työssä kerrotaan miten ongelmia on ratkaistu, perusteluineen ja ratkaisuvaihtoehtoineen. Lopuksi esiteltiin ratkaisuissa ilmenneistä ongelmista ja arvioitiin valittuja ratkaisuja.

Yhteenvetoa koostettaessa *AlarmSystem* on pilottikäytössä tuotantoympäristössä. Se monitoroi noin 500 kohdetta kolmen eri kriteeristön perusteella. Järjestelmän tuottamia automaattisia hälytyksiä validoidaan asiantuntijoiden kanssa järjestelmän luotettavuuden varmentamiseksi. *AlarmSystemin* moduulien toteutus on yhteensä pituudeltaan 1101 riviä. *AlarmSystemin* yksikkötestit ovat yhteensä pituudeltaan 675 riviä. Lisäksi asiakkaan toimittamia uusia monitorointitarpeita on jo sovitettu *AlarmSystemiin.*

Työn suunnittelu, toteutus ja testaus onnistuivat hyvin. Ongelmia ilmeni vasta, kun järjestelmää alettiin sovittamaan tuotantoympäristöön. Ympäristö ei aina käyttäydy kuten oletettiin, jolloin alkuperäiset esitutkimusvaiheessa määritellyt viantunnistuskriteerit olivat lähes käyttökelvottomia. Tämä on yleinen haaste reaktiivisten järjestelmien luotettavuudessa ympäristössä, josta ei ole hallintaa ja jonka käyttäytymistä voidaan vain arvioida parhaan mukaan. Ympäristön käyttäytyminen olisi voinut tulla tutuksi aikaisemmin *iteratiivisemmalla* kehitysmallilla, jossa tuotantoympäristöön vientejä olisi tehty nopeammin iteraatiokierrosten aikana. Tämä olisi kuitenkin voinut tuoda uusia ongelmia tuotantoonviennin monimutkaisuuden ja työläyden vuoksi. Sääntöjä on kuitenkin jatkokehitetty ottamaan huomioon ympäristön ominaisuudet, jotka vaikuttivat sääntöjen toimintaan kriitisesti. Tämän jälkeen sääntöinä mallinnetut viantunnistuskriteerit toimivat lähes kaikissa tapauksissa kuten määritelty.

# LÄHDELUETTELO

[1] CORBA FAQ. Object Management Group. [WWW] Viitattu 16.9.2015. Saatavissa: http://www.omg.org/gettingstarted/corbafaq.htm

[2] LINQ, Language Integrated query. Microsoft Developer Network. [WWW] Viitattu 16.9.2015. Saatavissa: https://msdn.microsoft.com/fi-fi/library/bb397926.aspx

[3] Nagios Solutions. [WWW] Viitattu 27.3.2015. Saatavissa: http://www.nagios.com/solutions

[4] Reactive Extensions. A brief intro. [WWW] Viitattu 12.4.2015, Saatavissa: https://rx.codeplex.com/

[5] Dapper .NET. Performance, [WWW] Viitattu 20.3.2015. Saatavissa: https://github.com/StackExchange/dapper-dot-net#performance

[6] Verifying code by using Unit Tests: MSDN. [WWW] Viitattu 25.3.2015. Saatavissa: https://msdn.microsoft.com/en-us/library/dd264975.aspx

[7] Harel David: Statecharts: A Visual Formalism of Complex Systems. Science of Computer Programming vol. 8. 1987.

[8] Coulouris George, Dollimore Jean & Kindberg Tim. Distributed Systems: Concepts and design. Addison-Wesley & Pearson Education Inc. 2005.

[9] Zohar Manna, Amir Pnuelli: Temporal logic of Reactive and Concurrent systems. Springer-Verlag. 1992.

[10] Chandy: Event-Driven Applications: Costs, Benefits and Design Approaches, California Institute of Technology. 2006.

[11] Hohpe G, Woolfe B: Enterprise Integration Patterns, Pearson Education Inc, 2004.

[12] Tarkoma, Sasu: Publish/subscribe systems: Design and principles, John Wiley & Sons limited. 2012.

[13] Schilling, Koldehofe, Rothermel: Efficient and Distributed Rule Placement in Heavy Constraint-Driven Event Systems. High Performance Computing and Communications (HPCC), IEEE 13th International Conference. 2011.

[14] Cugola, Margara: Deployment Strategies for Distributed Complex Event Processing, Department of Electronic Information. Milan, Italia. 2012.

[15] Arulanthu, O'Ryan, Schmidt, Kircher, Parsons: The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging. Department of Computer Science, Washington University. 2001.

[16] J.D. Meier, Srinath Vasireddy, Ashish Babbar, Alex Mackman: Improving .NET Application Performance and Scalability. Microsoft Press. 2004.

[17] Kircher, Prashant: Pattern-Oriented Software Architecture, Patterns for Resource Management. WICSA, The Working IEEE/IFIP Conference. 2007.

[18] Connolly, Begg: Database Systems, A Practical Approach to Design, Implementation, and Management. Neljäs painos. Addison-Wesley & Pearson Education Inc. 2005.

[19] Garlan, Shaw: An Introduction to Software Architecture. School of Computer Science, Carnegie Mellon University. 1994.

[20] Ireson-Paine: What is a rule-based system [WWW] Viitattu 23.5.2015. Saatavissa: http://www.j-paine.org/students/lectures/lect3/node5.html

[21] Johansson, Löfgren: Designing for Extensibility: An action research study of maximizing extensibility by means of design principle. Department of Applied Information Technology, University of Gothenburg. 2009

[22] Bondi, André: Characteristics of scalability and their impact on performance. WOSP2000: Second International Workshop on Software and Performance. 2000.

[23] Rennels. Fault Tolerant Computing. [WWW] Viitattu 24.5.2015. Saatavissa: http://www.cs.ucla.edu/~rennels/article98.pdf

[24] Task-based asynchronous pattern. [WWW] Viitattu 28.5.2015. Saatavissa: https://msdn.microsoft.com/en-us/library/hh873175(v=vs.110).aspx

[25] Piedad, Hawkins: High Availability: Design, Techniques, and Processes. Prentice-Hall Inc. 2001.

[26] Testing Rx Queries using Virtual Time Scheduling. [WWW] Viitattu 30.5.2015. Saatavissa: http://blogs.msdn.com/b/rxteam/archive/2012/06/14/testing-rx-queries-using-virtual-time-scheduling.aspx

[27] What Is Windows Communication Foundation [WWW] Viitattu 30.5.2015. Saatavissa: http://msdn.microsoft.com/en-us/library/ms731082%28v=vs.110%29.aspx

[28] Windows Dev Center: Services [WWW] Viitattu 30.5.2015. Saatavissa: https://msdn.microsoft.com/en-us/library/windows/desktop/ms685141(v=vs.85).aspx