Matti-Pekka Sivosuo

Tuotelinja-arkkitehtuurit ohjelmistokehityksessä

Tietotekniikan (Ohjelmistotekniikka)
pro gradu -tutkielma
31. toukokuuta 2004

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Matti-Pekka Sivosuo
Yhteystiedot:
Matti-Pekka Sivosuo
Survontie 46 A 53
40520 Jyväskylä
Email: mapes@iki.fi
Työn nimi: Tuotelinja-arkkitehtuurit ohjelmistokehityksessä
Title in English: Product Line Architectures in Software Development
Työ: Tietotekniikan (Ohjelmistotekniikka) pro gradu -tutkielma
Sivumäärä: 83
Tiivistelmä: Tässä tutkielmassa perehdytään tuotelinja-arkkitehtuureihin ja erityisesti tuotelinjan perustamisen vaiheisiin. Tarkoituksena on tutkia, miten tuotelinja perustetaan ja minkälaisia vaiheita tässä prosessissa kohdataan. Määriteltyä prosessia sovelletaan käytännössä suunnittelemalla demonstraatiotuotelinja. Tämän avulla pyritään havainnollistamamaan tuotelinjan kokoamista, asetettuja odotuksia sekä ongelmakohtia, joita mahdollisesti kohdataan tuotelinjaa kasattaessa. Määrittelyn aikana tehtyjen havaintojen pohjalta pohditaan tuotelinjojen ongelmia ja mahdollisuuksia.
English abstract: In this thesis product line architectures (PLA), especially phases in constructing a product line, will be reviewed. The purpose is to study how a product line architecture is constructed and which phases take place in this process. The defined process is then applied in practice by designing a demonstration product line. This will be used to illustrate the construction of a product line, anticipated expectations, and trouble spots likely to be encountered during the construction. Based on observations done during PLA definition problems and possibilities in product lines will be discussed.
Avainsanat: tuotelinjat, ohjelmistoarkkitehtuurit, koodin uudelleenkäyttö, ohjelmistoprosessi
Keywords: product lines, software architectures, code reuse, software process

Sisältö

Sisältö **i**

1 Johdanto **1**

2 Mitä ovat tuotelinjat? **3**
2.1 Tuotelinjoille asetettuja odotuksia 4
2.2 Tuotelinjan perustamisen aiheuttamia kustannuksia 5
2.3 Tuotelinjan perustamisen vaiheet 6
2.4 Yhteenveto 8

3 Tuoteperheen määrittely **9**
3.1 Sovellusalueanalyysi tuoteperheille 9
3.1.1 Sovellusaluemäärittely 10
3.1.2 Tiedon keräys 10
3.1.3 Analyysivaihe 10
3.1.4 Luokittelu 11
3.1.5 Huomioita 12
3.2 ODM 12
3.2.1 Deskriptiivinen vaihe 12
3.2.2 Preskriptiivinen vaihe 13
3.2.3 Huomioita 13
3.3 FeatuRSEB 13
3.3.1 Piirremalli 14
3.3.2 Käyttötapausmalli 15
3.4 FORM 16
3.4.1 Sovellusalueanalyysi 16
3.4.2 Piirremallin määrittely 17
3.4.3 Arkkitehtuurimallit ja niiden määrittely 17
3.5 FAST 18
3.5.1 Organisaatioroolit 19
3.5.2 Sovellusaluevaihe 19
3.5.3 Sovellusvaihe 20
3.6 PuLSE 20
3.6.1 Tukevat komponentit 21
3.6.2 Tekniset komponentit 21

3.6.3 Alustusvaihe . . . 22
3.6.4 Infrastruktuurin muodostaminen . . . 22
3.6.5 Infrastruktuurin käyttövaihe . . . 23
3.6.6 Hallinta- ja kehitysvaihe . . . 23
3.7 Määriteltyjen piirteiden hallinta . . . 23
3.7.1 Ristiriitaongelma . . . 23
3.7.2 Ratkaisuja . . . 24
3.8 Yhteenveto . . . 24

4 Arkkitehtuurin suunnittelu 26
4.1 Ohjelmistoarkkitehtuuri . . . 26
4.2 Arkkitehtuurityylit . . . 27
4.3 Arkkitehtuurille asetetut odotukset . . . 27
4.4 Arkkitehtuurin määrittäminen . . . 28
4.4.1 Arkkitehtuuristen tavoitteiden valinta . . . 28
4.4.2 Arkkitehtuurityylin valinta . . . 29
4.4.3 Moduulien instantiointi ja toiminnallisuuksien liittäminen niihin . . . 29
4.4.4 Moduulien rajapintojen määritys . . . 30
4.4.5 Validointi, käyttötapausten ja laatuskenaarioiden jalostus sekä rajoitteiden määrittely lapsimoduuleille . . . 30
4.5 Tuotelinjan vaikutus arkkitehtuuriin . . . 31
4.5.1 Variaatiopisteiden tunnistaminen . . . 31
4.5.2 Variaatiopisteiden tukeminen . . . 31
4.6 Yhteenveto . . . 32

5 Arkkitehtuurin arviointi 33
5.1 Arvioinnin tavoitteet . . . 33
5.2 Esiehdot . . . 34
5.3 Arviointimenetelmätyypit . . . 35
5.3.1 Skenaarioperustaiset menetelmät . . . 35
5.3.2 Simulaatioperustaiset menetelmät . . . 35
5.3.3 Matemaattisperustaiset menetelmät . . . 36
5.3.4 Kokemusperustaiset menetelmät . . . 36
5.4 ATAM . . . 36
5.4.1 Osanottajat . . . 37
5.4.2 Kumppanuus ja valmistautuminen . . . 38
5.4.3 Ensimäinen arviointivaihe . . . 38
5.4.4 Toinen arviointivaihe . . . 39
5.4.5 Jälkivaihe . . . 40
5.5 Yhteenveto . . . 41

6 Elementtien toteutus **42**
6.1 Elementtien rajapinnat . . . 42
6.1.1 Konfigurointirajapinnat . . . 42
6.1.2 Tarjottavat rajapinnat . . . 43
6.1.3 Vaadittavat rajapinnat . . . 43
6.2 Ohjelmointikieliin perustuvat variaatiomenetelmät . . . 43
6.2.1 Perintä . . . 43
6.2.2 Koostaminen . . . 44
6.2.3 Rajapinnat . . . 44
6.2.4 Makrot . . . 44
6.2.5 Suunnittelumallit . . . 45
6.2.6 Subjektiohjelmointi . . . 45
6.2.7 Aspektiohjelmointi . . . 47
6.2.8 Geneerinen ohjelmointi . . . 47
6.3 Järjestelmägeneraattorit . . . 48
6.3.1 GenVoca . . . 48
6.4 Yhteenveto . . . 49

7 Tuotelinjat ja sovelluskehykset **50**
7.1 Kehys tuotelinjana . . . 50
7.2 Kehykset elementteinä . . . 51
7.2.1 Tuotekohtainen laajennusmalli . . . 51
7.2.2 Standardikohtainen laajennusmalli . . . 51
7.2.3 Hienojakoinen laajennusmalli . . . 51
7.2.4 Generaattorimalli . . . 52
7.3 Yhteenveto . . . 52

8 Rannekellotuotelinja **53**
8.1 Sovellusalueen määritys . . . 53
8.1.1 Tiedon keräys . . . 54
8.2 Tiedon analysointi ja luokittelu . . . 54
8.2.1 Kellonaika . . . 54
8.2.2 Päivämäärä . . . 54
8.2.3 Ajanotto . . . 55
8.2.4 Muistutus . . . 55
8.2.5 Laatuvaatimukset . . . 55
8.2.6 Määritelty tuoteperhe . . . 55
8.3 Arkkitehtuurityyli . . . 56
8.3.1 Arkkitehtuuriset tavoitteet . . . 56
8.3.2 Arkkitehtuurityylin valinta . . . 57
8.4 Elementtien tunnistus . . . 58

8.5 Näyttöelementti . . . 59
8.5.1 Vastuualueet . . . 59
8.5.2 Luokkarakenne . . . 59
8.5.3 Riippuvuudet . . . 59
8.5.4 Konfiguraatio . . . 59
8.5.5 Rajapinnat . . . 59
8.6 Kellomoottorielementti . . . 60
8.6.1 Vastuualueet . . . 60
8.6.2 Luokkarakenne . . . 60
8.6.3 Riippuvuudet . . . 61
8.6.4 Konfiguraatio . . . 61
8.6.5 Rajapinnat . . . 61
8.7 Kellonaikaelementti . . . 61
8.7.1 Vastuualueet . . . 61
8.7.2 Luokkarakenne . . . 61
8.7.3 Riippuvuudet . . . 62
8.7.4 Konfiguraatio . . . 62
8.7.5 Rajapinnat . . . 62
8.8 Muistutuselementti . . . 62
8.8.1 Vastuualueet . . . 62
8.8.2 Luokkarakenne . . . 62
8.8.3 Riippuvuudet . . . 62
8.8.4 Konfiguraatio . . . 63
8.8.5 Rajapinnat . . . 63
8.9 Päivämääräelementti . . . 63
8.9.1 Vastuualueet . . . 63
8.9.2 Luokkarakenne . . . 63
8.9.3 Riippuvuudet . . . 63
8.9.4 Konfiguraatio . . . 63
8.9.5 Rajapinnat . . . 64
8.10 Ajanottoelementti . . . 64
8.10.1 Vastuualueet . . . 64
8.10.2 Luokkarakenne . . . 64
8.10.3 Riippuvuudet . . . 64
8.10.4 Konfiguraatio . . . 64
8.10.5 Rajapinnat . . . 65
8.11 Yhteistoiminnan mallinnus . . . 65
8.11.1 Rinnakkaisuusnäkymä . . . 65
8.11.2 Moduulinäkymä . . . 66
8.11.3 Sijoitusnäkymä . . . 67

8.12 Suuntaviivoja toteutukselle 67
8.13 Huomioita rannekellotuotelinjasta 68
8.13.1 Sovellusmahdollisuus 68
8.13.2 Useamman tuotteen monimutkaisuus 69
8.13.3 Arkkitehtuuri 69
8.13.4 Tulokset 70

9 Yhteenveto **71**

10 Kirjallisuutta **72**

1 Johdanto

Ohjelmistotekniikka on vielä nuori tieteenala. Viimeisimmän 50 vuoden aikana kehitys on kuitenkin ollut huimaa. Teknologian kehityksen myötä myös ohjelmointikielet ja ohjelmat ovat kehittyneet. Konekielestä alkanut kehitys on johtanut korkean tason ohjelmointikieliin ja erilaisiin ohjelmointiparadigmoihin. Tavoitteena on ollut helpottaa ohjelmien tekoa ja saavuttaa parempia tuloksia esimerkiksi laadun, tehokkuuden ja uudelleenkäytettävyyden alueella.

Ohjelmointikielten kehittyessä myös tieteenala kehittyy. Ohjelmista halutaan uudelleenkäytettäviä, modulaarisia ja laadukkaita. Tämä kehitys on johtanut esimerkiksi uudelleenkäytettäviin aliohjelmiin, koodikirjastoihin, olioparadigmaan ja ohjelmistoarkkitehtuureihin. Vähitellen ollaan siirrytty opportunistisesta uudelleenkäytöstä järjestelmälliseen. Vaikka kaikki edellämainitut kehitysaskeleet ovat olleet aikansa läpimurtoja, ei eteneminen ole pysähtynyt. Tieteenalan ja teollisuuden kehitys johtaa aina uusiin pyrkimyksiin saavuttaa asetetut tavoitteet ja tavoitella uusia mahdollisia edistysaskelia. Eräs viime vuosina sekä akatemian että teollisuuden puolella kiinnostusta herättänyt ja lupaava aihe on ollut tuotelinja-arkkitehtuurit.

Ohjelmien rakentaminen komponenteista ei ole uusi ajatus. Tähän on viitattu kirjallisuudessa jo 1960-luvun lopulla [43] ja 1970-luvulla [47], [48] . Ohjelmistoarkkitehtuurien myötä laatuvaatimusten täyttyminen ja ohjelmistojen rakenne nousivat enemmän esiin. Tuotelinja-arkkitehtuureissa ajatusta viedään askeleen pidemmälle laajentamalla sovellusaluetta koskemaan tuoteperhettä. Yhden tuotteen sijasta määritellään ja kehitetään tietyn tarkkaan määritetyn sovellusalueen kattava tuoteperhe. Tuoteperheen yhteinen arkkitehtuuri ohjaa kohti tietyt laadulliset tavoitteet täyttävää joukkoa. Perheen jäsenet koostuvat elementeistä, jotka ovat yhteisiä tuotteiden kesken. Eroavaisuudet tuotteiden kesken toteutetaan konfiguroimalla tuotteelle valittua elementtijoukkoa. Lähestymistavalla tavoitellaan korkeaa uudelleenkäyttöä, tehokkuutta, laadukkuutta ja tietysti taloudellista kannattavuutta. Tuotelinjan perustaminen on kuitenkin raskas ja aikaavievä operaatio, johon liittyy monia ongelmia ja monimutkaisuutta. Onkin järkevää tutkia lähestymistavan kannattavuutta ennen soveltamista.

Tässä tutkielmassa perehdytään tuotelinja-arkkitehtuureihin ja erityisesti tuotelinjan perustamisen vaiheisiin. Tarkoituksena on tutkia, miten tuotelinja perustetaan ja minkälaisia vaiheita tässä prosessissa kohdataan. Määriteltyä prosessia sovelletaan suunnittelemalla demonstraatiotuotelinja. Tämän avulla pyritään havainnollistamaan tuotelinjan kokoamista, asetettuja odotuksia sekä ongelmakohtia, joita mahdollisesti kohdataan tuotelinjaa kasattaessa. Määrittelyn aikana tehtyjen havainto-

jen pohjalta pohditaan tuotelinjojen ongelmia ja mahdollisuuksia. Toisessa luvussa tutustutaan yleisellä tasolla tuotelinjoihin ja niiden perustamiseen liittyviin vaiheisiin. Kolmannessa luvussa esitellään tuotelinjoja määriteltäessä käytettäviä sovellusalueteknisiä (engl. *domain engineering*) menetelmiä. Neljännessä luvussa keskitytään ohjelmistoarkkitehtuuriin ja sen suunnitteluun. Viidennessä luvussa käsitellään arkkitehtuurin arviointia. Kuudennessa luvussa tutustutaan elementtien toteuttamisessa käytettäviin menetelmiin. Seitsemännessä luvussa pohditaan lyhyesti tuotelinjojen ja sovelluskehysten välistä suhdetta. Kahdeksannessa luvussa demonstroidaan tuotelinjan perustamista, siihen liittyviä ongelmia ja saavutettavia tuloksia määrittelemällä rannekellotuotelinja. Yhdeksännessä kappaleessa kootaan tehdyt johtopäätökset yhteen.

2 Mitä ovat tuotelinjat?

Tuotelinja on menetelmä tuottaa ohjelmistoperheitä. Tuotelinjan määritelmiä on yhtä monta kuin alan asiantuntijoitakin. Martin L. Griss tiivistää artikkelissaan [23, sivu 3] tuotelinjan toisiinsa liittyväksi tuotejoukoksi, joka jakaa yhteisen vaatimusjoukon mutta sisältää myös tuotekohtaisia vaatimuksia. Jan Bosch kuvaa tuotelinjoja organisaation toisiinsa liittyvän tuotejoukon yhteiseksi arkkitehtuuriksi [11, sivu 162]. Erinomainen määritelmä [13, sivu 5] tuotelinjoille on:

> A set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market seqment or mission and that are developed from a common set of core assets in a prescribed way.

Ylläoleva määritelmä pitää sisällään kaikki tuontantolinja-arkkitehtuureihin liittyvät olennaiset elementit. Tuotelinjan jäsenet ovat samaan sovellusalueeseen kuuluvia yhteisen piirrejoukon jakavia ja samoista rakennusosista muodostuvia ohjelmistoja. Yhteisten piirteiden ohella tuotelinjan jäsenet jakavat yhteisen *arkkitehtuuri*n [6, sivu 353]. Ohjelmiston arkkitehtuuri määrittää tuotteen rakenteen ja rakenteiden väliset suhteet. Elementit ovat järjestelmän modularisoituvia kokonaisuuksia. Elementtien välillä vallitsee suhteita, jotka määrittelevät niiden välisen yhteistoiminnan. Elementit ja niiden väliset suhteet muodostavat järjestelmän rakenteen. Tiivis ja kuvaava määritelmä [6, sivu 3] ohjelmistoarkkitehtuurille on:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements and the relationships between them.

Arkkitehtuurin valinta on suuri päätös, joka vaikuttaa olennaisesti tulevan järjestelmän ominaisuuksiin. Arkkitehtuuri määrittelee paljolti järjestelmän vahvuudet ja heikkoudet, joiden tulisi vastata tuotelinjan jäsenten vaatimuksia. Sen suunnittelu vaatii paljon aikaa ja resursseja, mutta tuotelinjan tuotteet jakavat tämän kustannuksen.

Tuotelinjan jäsenet jakavat myös vankan yhteisen vaatimusjoukon. Tuotteiden täytyy olla vahvasti toisiinsa liittyviä ja valmiita sitoutumaan yhteiseen arkkitehtuuriin. Tuotteille asetettuihin vaatimuksiin vastataan ominaisuuksien avulla. Puhuttaessa tuotelinjan jäsenten erityisistä ja yhteisistä ominaisuuksista käytetään yleensä termiä *piirre* (engl. *feature*). Piirre on tuotteen ominaisuus, jonka asiakkaat ja käyttäjät kokevat tärkeäksi tuotelinjan jäsenten kuvaamisessa ja erottamisessa. Piirre voi

liittyä yksittäiseen vaatimukseen, valikoimaan vaihtoehtoisia tai valinnaisia vaatimuksia tai tuotteeseen tai toteutukseen liittyvään ominaisuuteen [23, sivu 3].

Saman tuotelinjan jäsenet käyttävät siis yhteistä arkkitehtuuria ja jakavat yhteisiä piirteitä. Tuotelinja-ajattelutavan looginen idea onkin koota kukin tuote yhteisestä elementtijoukosta valitsemalla tuotteen ominaisuuksia vastaavat elementit. Näitä elementtejä muokataan tarvittaessa täyttämään tuotekohtaiset vaatimukset. Tätä varten on luotu koko tuotelinjan elinkaaren kattavia prosessimalleja.

2.1 Tuotelinjoille asetettuja odotuksia

Tuotelinjojen vaikutukset ovat sekä taloudellisia että toteutukseen liittyviä. Järkevillä suunnittelupäätöksillä ja arkkitehtuurien käytöllä pyritään tekemään parempia ohjelmia. Vaikka laajennettavuus ja uudelleenkäytettävä lähdekoodi parantavat ohjelman laatua, on niillä myös taloudellista merkitystä. Tuotelinja-arkkitehtuurin vaikutukset ulottuvat suunnitteluun, työntekijöihin, kustannuksiin, aikatauluun ja tuleviin projekteihin. Odotettujen positiivisten ja kestävien vaikutustensa vuoksi tuotelinja-arkkitehtuurit ovatkin nykyään kasvavan mielenkiinnon kohteena.

Tärkein tekijä uutta ohjelmistoa kehitettäessä on tuotteen julkaisuaika (engl. *time-to-market*). Termillä tarkoitetaan vaatimusmäärittelyn ja valmiin tuotteen välistä aikaa. Yrityksen tuotteet kuuluvat usein samaan sovellusalueeseen ja liittyvät näin toisiinsa. Jo perustettu tuotelinja tarjoaa mahdollisesti valmiita elementtejä uudelle tuotteelle. Samankaltaisilla tuotteilla on toisiaan muistuttavia vaatimuksia. Osa vanhojen tuotteiden vaatimuksista voidaan sopivassa tapauksessa siirtää suoraan uudelle tuotteelle ja näin joudutaan määrittelemään vain uudet ominaisuudet [6, sivu 355]. Vaativa ja aikaavievä arkkitehtuurisuunnittelu on tehty tuotelinjan tuotteille jo aiemmin. Toteutusvaiheessa on mahdollista hyödyntää jo olemassaolevia elementtejä. Vain uudet piirteet joudutaan tällöin toteuttamaan alusta asti. Testausvaiheessa voidaan käyttää aikaisempaa testausohjelmaa samankaltaisille tuotteille. Yhdessä näistä tekijöistä voi kertyä huomattava ajansäästö.

Kaupallisia ohjelmia tehtäessä kehityskustannukset merkitsevät paljon. Tuotelinjojen tapauksessa kustannukset jakaantuvat useamman tuotteen kesken tuoteperheessä. Raskain vaihe onkin juuri perustaminen. Kun elementit on toteutettu, voidaan keskittyä niiden kehittämiseen. Tuotelinja-arkkitehtuureissa tuotelinjan elementtejä kehitettäessä vaikutukset ulottuvat koko tuoteperheeseen. Tämän keskitetyn kehittämisen tuloksena tuotelinjojen elementeistä kehkeytyy korkealaatuisia, mikä pienentää tulevia ylläpitokuluja [11, sivu 191].

Bosch nostaa esiin myös vaikutukset henkilökuntaan [11, sivu 191]. Perinteinen malli, jossa palkataan uusia työntekijöitä käynnistämään tuotetarjontaa laajentavia projekteja, ei välttämättä ole kustannustehokasta tai aina mahdollistakaan. Tarvetta kuitenkin uusille houkutteleville tuotteille olisi. Tuotelinjat mahdollistavat sen, et-

tä vanhat työntekijät toteuttavat ja kehittävät suurempaa tuotejoukkoa yhden tuotteen sijasta. Tämä vaatii tietysti osaavia työntekijöitä. Tämän osaamisen toivotaan vaikuttavan yhden tuotteen sijasta koko tuotelinjaan.

Prosessien kohdalla voidaan tarkemmin nostaa esiin testaus, mallintaminen, työkalut, projektin suunnittelu ja laadun parantuminen [6, sivu 355]. Samankaltaisia tuotteita testattaessa syntyy valmiita testitapauksia, valmiita tuloksia ja testitulosten käsittelyrutiineja, joita voidaan hyödyntää useammalle tuotteelle. Erilaiset tuotteen ominaisuuksien mallinnukset ja niiden tulosten analysointi voidaan hyödyntää koko tuotelinjalle. Tuotelinjoista saatava kokemus realisoituu projektin suunnittelussa esimerksi aikataulujen ja budjettien laadinnan yhteydessä. Tämä kehitys johtaa jalostuneisiin ja määriteltyihin prosesseihin. Yleinen laadun parantuminen seuraa suunnittelupäätösten ja ohjelmakoodin uudelleenkäytöstä. Niihin tehdyt parannukset siirtyvät aina uusille kehitettäville järjestelmille.

2.2 Tuotelinjan perustamisen aiheuttamia kustannuksia

Tuotelinjojen perustaminen vaatii aikaa, resursseja ja osaamista. Samoin siirtyminen vanhasta uuteen tuotelinjalähestymistapaan on monimutkainen prosessi, joka saattaa hidastaa jo käynnissä olevia projekteja ja aiheuttaa muutoksia organisaatiossa [11, sivu 165]. Tuotelinja-arkkitehtuurin suunnittelu on kallis operaatio [11, sivu 165], joten jo tuotelinjan aiheuttama taloudellinen riski on kustannus.

Uutta tuotelinjaa perustettaessa suoritetaan aluksi laaja ja huolellinen vaatimusmäärittely, jossa määritellään vaatimukset koko tuoteperheelle. Verrattuna yhdelle tuotteelle tehtävään vaatimusmäärittelyyn on tuoteperheen vaatimusmäärittely monimutkaisempi ja pitkäkestoisempi prosessi, johon liittyy vahvasti myös sovellusalueen määrittely. Tuoteperheen yhteiset piirteet on myös tunnistettava. Tuoteperheelle tehtyjen määritysten perusteella suunnitellaan sen yhteinen arkkitehtuuri. Arkkitehtuurin valinta on aikaavievä vaihe, joka vaikuttaa vahvasti tuoteperheen laadullisiin ominaisuuksiin. Yhden järjestelmän tapauksessa päätös on yleensä helpompi tai sitä ei edes tehdä. Elementtien suunnitteluun ja toteutukseen siirryttäessä on edessä tuoteperheen yhteisten elementtien toteutus. Sanomattakin on selvää, että tuotteiden piirteiden välisten riippuvuuksien käsittely on hankalamapaa kuin yhden sovelluksen kohdalla. Kokoamisvaiheessa tuote joudutaan kokoamaan eri elementeistä. Jokaiselle tuotteelle valitaan ensin sopivat elementit. Mahdollisesti elementtejä joudutaan konfiguroimaan tietylle tuotteelle sopivaksi. Testausvaihetta varten joudutaan kehittämään tuoteperhekohtaiset sekä tuotekohtaiset testitapaukset. Yhden tuotteen testaus on edelleen helpompaa kuin vastaavan tuoteperheen.

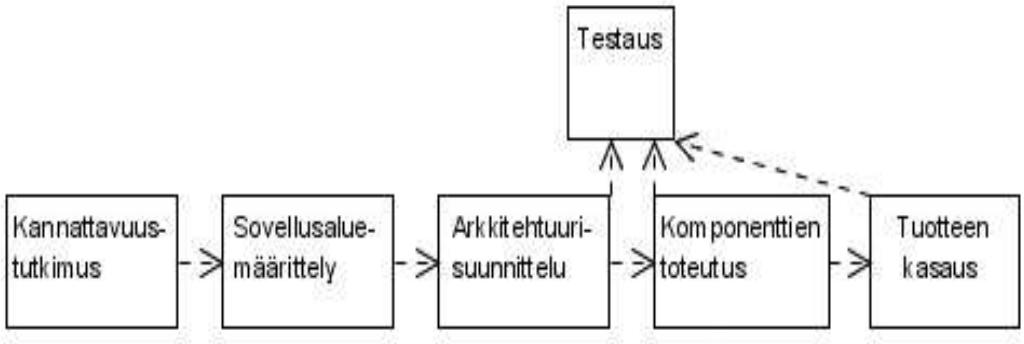
Omana perustamistapauksenaan voidaan ajatella siirtyminen vanhasta käytössäolevasta yhden tuotteen prosessista tuotelinjamaiseen prosessiin. Tämä voi tapahtua joko perustamalla kerralla uusi tuotelinja tai kehittämällä vähitellen jo olemas-

saolevista tuotteista tuotelinja [11, sivu 166]. Vanhan tuotejoukon muuttaminen tuotelinjaksi on pitkä prosessi. Valitaan arkkitehtuuri, tunnistetaan yhteiset elementit, jotka vähitellen yleistetään tuotekohtaisista vanhoista elementeistä yhteisiksi. Vaikka lähestymistapa minimoi riskejä ja hyödyntää vanhoja olemassaolevia tuotteita, hidastaa siirtymävaihe meneillään olevien projektien etenemistä [11, sivu 167]. Pitkällä aikavälillä myös menetelmän kokonaiskustannukset nousevat uuden linjan perustamiskustannuksia korkeammiksi, koska siirtymisvaiheessa joudutaan tekemään useita tilapäisiä ratkaisuja [11, sivu 168]. Siirtyminen kerralla uuden tuotelinjan käyttöön on suuri muutos. Se voi tapahtua joko yhdistämällä vanhat tuotteet uudeksi tuotelinjaksi tai perustamalla kokonaan uusi tuotelinja. Ensimmäinen tapa vaatii runsaasti aikaa ja vaivaa, joten yleensä on järkevämpää siirtyä kokonaan uuden linjan käyttöön [11, sivu 168]. Kummassakin tapauksessa siirtyminen vanhasta uuteen aiheuttaa työtä ja tuotantoaikojen venymistä [11, sivu 167-168].

2.3 Tuotelinjan perustamisen vaiheet

Tuotelinjan perustaminen jakaantuu useaan osaan [11, sivu 189], jotka voidaan taas jakaa omiin alaosioihinsa. Tuotelinjalähestymistapaan soveltuvia menetelmiä on useita. Eri menetelmillä on omat vaiheensa ja termistönsä. Toiset menetelmät ovat erikoistuneita vain tuoteperheen ja sovellusalueen määrittelyyn, kun taas toiset kattavat myös toteutusosion. Tutkimalla olemassaolevia menetelmiä ja kirjallisuutta voidaan tunnistaa tuotelinjalähestymistavalle ominaisia vaiheita. Näitä ovat sovellusaluemäärittely [3, sivu 42, 46], [51, sivu 196, 204], arkkitehtuurisuunnittelu [11, sivu 201], [32, sivu 143], [10, sivu 126], elementtien suunnittelu ja toteutus [11, sivu 214], [32, sivu 160], tuotteen kasaus [32, sivu 160] sekä testaus [11, sivu 189], [6, sivu 362], [45, sivu 2]. Ennen näitä kaikkia on mahdollista suorittaa erityistä kannattavuustutkimusta, jolla arvioidaan aloitettavan hankkeen kannattavuutta. Tämä vaihe ei kuitenkaan ole ominainen tuotelinjoille tai ohjelmistoprosessille, vaan erityisesti yrityselämässä harjoitettava käytäntö.

Sovellusaluemäärittelyn laajuus riippuu siitä, ollaanko perustamassa uutta vai laajentamassa olemassaolevaa tuotelinjaa. Uuden tuotelinjan tapauksessa joudutaan määrittelemään sekä tuoteperheen yhteiset että tuotekohtaiset vaatimukset ja selvittämään näiden väliset ristiriidat. Tuoteperheen tuotteiden on liityttävä riittävässä määrin toisiinsa, jotta tuotelinja olisi järkevä vaihtoehto. Tuoteperheen laajennuksessa yhteiset ominaisuudet on jo määritelty. Jäljelle jää vain tuotekohtaisten ominaisuuksien määrittely ja uusien ristiriitojen tarkastaminen. Kyseessä on tällöin enemmänkin yhden tuotteen vaatimusmäärittely. Arkkitehtuurin suunnitteluun on kehitetty prosesseja, jotka ottavat huomioon tuotteille asetetut laadulliset ja toiminnalliset vaatimukset. Tuotelinja-arkkitehtuuria suunniteltaessa tulee lisäksi ottaa huomioon variaatiopisteiden tunnistaminen, niiden tukeminen ja arkkitehtuurin sopi-



Kuva 2.1: Tuotelinjan vaiheet

vuus tuotelinjalle. Variaatiopisteet ovat kohtia, joissa tuoteperheen jäsenet voivat erota toisistaan.

Elementtien suunnittelu tapahtuu valitun arkkitehtuurin ja tuotteiden määriteltyjen piirteiden pohjalta. Uutta linjaa perustettaessa pitää suunnitella kaikki tarvittavat elementit, jotta vaatimukset täyttyvät. Tuotevalikoimaa laajennettaessa vain tuotekohtaiset ominaisuudet määritellään. Elementtien määrittely piirteiden pohjalta on sisällytetty erilaisiin prosesseihin (esim. PuLSE, KobrA). Muunneltavien elementtien toteutus tuotteille on vaativaa. Tähän käytetään useita erilaisia lähestymistapoja. Mahdollisia toteutusmenetelmiä ovat esimerkiksi aspektiohjelmointi (engl. *aspect-oriented programming*), subjektiohjelmointi (engl. *subject-oriented programming*), olio-ohjelmointi ja sovelluskehykset.

Näiden vaiheiden jälkeen tuotelinja on perustettu. Jotta tuote saadaan markkinoille, se pitää kasata valmiista elementeistä. Ensin tuotteelle valitaan sen ominaisuuksia vastaavat elementit. Elementit mahdollisesti konfiguroidaan ja tuloksena on toimiva sovellus.

Tuotelinjan testauksessa pitää pystyä käsittelemään tuotteiden vaihtelevuutta ja hyödyntämään toisaalta samankaltaisuutta. Tuotelinjassa voidaan ajatella tapahtuvan kolmenlaista testausta: arkkitehtuuritestausta, toiminnallista testausta ja regressiotestausta [45, sivu 4]. Arkkitehtuuritestauksella pyritään selvittämään, miten hyvin valittu arkkitehtuuri vastaa järjestelmälle asetettuja laadullisia ja toiminnallisia tavoitteita. Tämän avulla pyritään ehkäisemään suunnitteluvaiheessa tapahtuvia virheitä. Toiminnallisessa testauksessa elementtien toimintaa ja rakennetta käytetään toiminnallisten testitapausten pohjana, joita käytetään elementtien lähdekoodeille. Regressiotestausta käytetään kehitys- ja ylläpitovaiheessa. Kehitysvaiheessa sitä voidaan hyödyntää yleistämällä yhden tuoteperheen jäsenen yleisiä testituloksia muille jäsenille. Ylläpitovaiheessa sen avulla validoidaan muutoksia tuotelinjassa.

2.4 Yhteenveto

Tässä luvussa käsiteltiin tuotelinjoja, niihin liittyviä hyötyjä ja ongelmia sekä tuotelinjan vaiheita. Tuotelinjojen kannalta sovellusalueen määritys on tärkeä vaihe, koska tuoteperheen jäsenet määritellään siinä. Suunnittelussa olennainen arkkitehtuuri ohjaa tuoteperheen laadullisia ominaisuuksia ja auttaa ymmärtämään ohjelmistojen rakennetta. Jatkossa keskitytäänkin sovellusalueen määrittelyyn sekä arkkitehtuurin suunnitteluun ja arviointiin niiden merkityksen vuoksi. Myöhemmin esitellään myös tuotelinjojen mahdollisia toteutusmenetelmiä, joiden avulla pyritään havainnollistamaan, miten määritelty tuoteperhe on mahdollista toteuttaa.

3 Tuoteperheen määrittely

Toimivan tuotelinjan perustamisen edellytyksenä on, että sen tuoteperhe on hyvin määriteltävissä ja rajattavissa. Monimutkaista järjestelmää suunniteltaessa vaatimusten määrä kasvaa nopeasti. On tärkeää tunnistaa tuotteiden tärkeimmät vaatimukset [42, sivu 61]. *Sovellusalueanalyysi* on prosessi, jossa tuoteperhe määritellään ja sen vaatimukset tunnistetaan ja luokitellaan. Tarkoituksena on määrittää tuoteperheen laatuvaatimukset, piirteet, ja näiden kautta arkkitehtuuriset tavoitteet (engl. *architectural driver*). Sovellusaluetekniset (engl. *domain engineering*) menetelmät määrittelevät tuoteperheen, sovellusaluekielen, kehitysympäristön ja tuotantoprosessin [57, sivu 53]. Sovellusaluetekniset menetelmät käyttävät sovellusalueanalyysia tuoteperheen määrittelyssä. Tässä luvussa käsitellään tarkemmin sovellusalueanalyysia ja esitellään tunnettuja sovellusalueteknisiä menetelmiä.

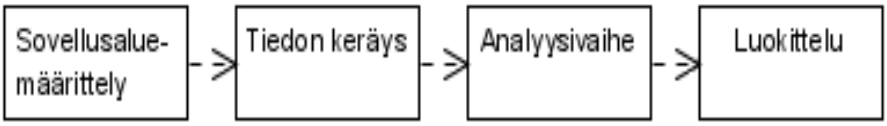
3.1 Sovellusalueanalyysi tuoteperheille

Sovellusalueanalyysin tarkoituksena on edistää uudelleenkäytettävyyttä tutkimalla rajattua samaan sovellusalueeseen kuuluvaa tuotejoukkoa, havaitsemalla yhteneväisyyksiä ja tuottamalla havainnoista malli. Guillermo Arango [3, sivu 43] määrittelee sovellusaluealyysin roolin seuraavasti

> The role of domain analysis is to acquire and consolidate information about applications in the domain so that such infrastructure can be designed reliably.

Sovellusalueen rooli on siis Arangon mukaan hankkia ja yhdistää tietoa sovellusalueen sovelluksista, ja tämän tiedon avulla mahdollistaa luotettava infrastruktuurin suunnittelu. Sovellusalueanalyysin tavoitteina voidaan pitää sovellusalueen sovellukset kuvaavan kielen (engl. *domain language*) kehittämistä, kielellä määriteltävien järjestelmien mahdollisten arkkitehtuurien joukon ja elementtien tunnistamista, kielellä tehtyjen määritysten yhdistämistä olennaisiin arkkitehtuureihin ja elementteihin sekä olennaisten arkkitehtuurien ja elementtien määrittelyä ennaltamääritellyt prosessit huomioiden [3, sivu 43]. Sovellusaluekielen avulla pitää pystyä määrittelemään sovellusalueen tuotteiden suhteet, rajoitteet, toiminnallisuus ja kokonaisuudet. Kokoonpanoon liittyvä vaihtoehtoisten arkkitehtuurien ja elementtien joukko on tunnistettava kaikille sovellusaluekielellä muodostettaville kokoonpanoille. Näin saadaan vankka pohja, jolle suunnittelupäätökset voidaan rakentaa. Tehdyt kielimääritykset pitää myös liittää vastaaviin elementteihin ja arkkitehtuu-

reihin. Havaitut arkkitehtuurit ja elementit on vielä määriteltävä sopiviksi ennalta määrättyä toteutusmekanismia ajatellen. Itse sovellusalueanalyysiprosessi voidaan jakaa neljään osaan: sovellusaluemäärittely (engl. *domain characterization*), tiedon keräys, tiedon analysointi ja tiedon luokittelu. Kaksi ensimmäistä vaihetta liittyvät alueen rajaamiseen ja tiedon keräämiseen siitä. Seuraavissa kahdessa vaiheessa käsitellään ja lajitellaan kerättyä tietoa.



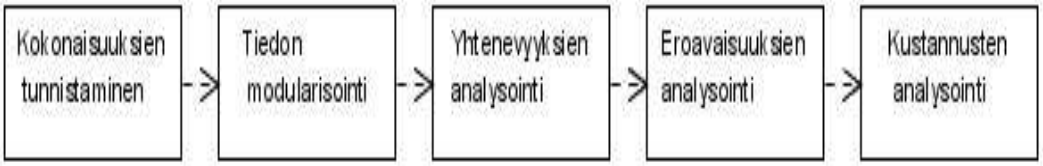
Kuva 3.1: Sovellusalueanalyysin vaiheet

3.1.1 Sovellusaluemäärittely Sovellusaluemäärittely [3, sivu 44] on vaativa vaihe, joka on erityisen tärkeä koko tuotelinjan perustamisen kannalta. Mitä rajatumpi alue on, sitä helpompaa on luoda siitä kattava malli. Toisaalta liian pieni sovellusalue tarkoittaa pientä tuotejoukkoa. Tämä taas voi johtaa raskaasta prosessista johtuen taloudelliseen kannattamattomuuteen. Tuotelinjalle on eduksi sovellusalueen pieni muuttuvuus. Mitä stabiilimpi sovellusalue on, sitä paremmin pystytään ennustamaan tulevia tarpeita. Olennaista on saada selkeä käsitys, mitä sovellusalueeseen kuuluu ja mitä siihen ei kuulu, ja onko tarkasteltavassa tapauksessa edes taloudellisesti järkevää rakentaa uudelleenkäytettävää kokonaisuutta.

3.1.2 Tiedon keräys Tiedon keräys [3, sivu 44] sisältää perinteisiä vaatimusmäärittelyyn liittyviä osioita. Tietoa kerätään asiakkailta ja asiantuntijoilta erilaisin menetelmin. Samoin tutkitaan kirjallisuutta ja mahdollisesti olemassaolevia järjestelmiä. Pyrkimyksenä on kerätä raakatietoa, jota voidaan suodattaa, yleistää, järjestää ja luokitella.

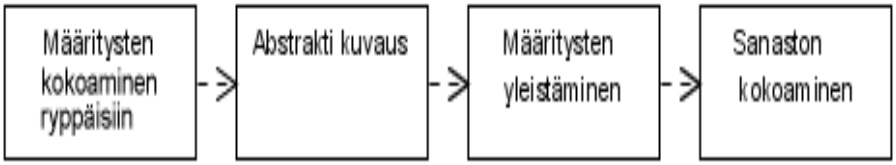
3.1.3 Analyysivaihe Analyysivaihe [3, sivu 45] saa syötteenään suuren määrän edellisessä vaiheessa kerättyä tietoa. Analysointi onkin keskeinen vaihe sovellusalueanalyysissa. Analyysivaiheen tarkoituksena on karsia tarpeeton informaatio ja määrittää tuotealueen kokonaisuudet, tuotteiden väliset suhteet, operaatiot ja tapahtumat. Tiedon analysointi on moniosainen putkimainen operaatio, joka voidaan iteroida läpi useampaan kertaan. Arango jakaa vaiheen kuuteen osioon: kokonaisuuksien tunnistaminen, tiedon modularisointi, yhtenevyyksien analysointi, eroavaisuuksien analysointi, kokoonpanojen analysointi ja kustannusten (engl. *trade-off*) analysointi. Kokonaisuuksien tunnistamisessa pyritään selvittämään juuri tuotealueen rakenteet ja niiden väliset yhteydet. Seuraavaksi pyritään saadut kokonaisuudet modularisoimaan käyttäen esimerkiksi oliomallinnusta. Tarkoituksena on eriyt-

tää kokonaisuudet järkevästi uudelleenkäyttöä ajatellen. Yhtenevyyksiä hakemalla pyritään yksinkertaistamaan mallia ja yhdistämään samankaltainen informaatio yhdeksi kokonaisuudeksi. Eroavaisuuksien etsinnällä pyritään kartoittamaan mahdollinen parametrisoinnin tai kapseloinnin mahdollisuus. Kokoonpanoja tutkimalla pyritään havitsemaan rakenteellisten mallien soveltamismahdollisuuksia. Tässä vaiheessa voidaan jo pohtia eri arkkitehtuurien sopivuutta sovellusalueeseen. Lopulta arvioidaan vielä mahdollisten eri suunnittelupäätösten etuja ja haittoja toisiinsa nähden.



Kuva 3.2: Analyysivaihe

3.1.4 Luokittelu Sovellusalueanalyysin viimeinen vaihe on luokittelu [3, sivu 44]. Siinä luokitellaan sovellusalueen määrityksiä ja pyritään kategorisoimaan niitä yläkäsitteiden alle. Yläkäsitteet muodostuvat, kun toisiaan muistuttavien määritysten tärkeimmät piirteet korvataan kuvaavalla yleisemmällä käsitteellä. Prosessin tuloksena saadaan sovellusalueen määrittelevän kielen sanasto. Kuten aikaisemmatkin vaiheet, jakautuu luokittelu useampaan alaosioon: määritysten ryppäisiinkokoaminen, abstrakti kuvaus, määritysten yleistäminen ja sanaston kokoaminen. Ryppäisiinkokoamisvaiheessa samankaltaiset piirteet kootaan omiin ryppäisiinsä. Jokaisesta ryppäästä valitaan tärkeimmät kuvaukset, jotka liitetään ylemmän tason alle. Muodostetut abstraktiot luokitellaan hierarkkisesti keskenään (engl. *generalization hierarchy*). Abstraktiohierarkioiden perusteella muodostetaan sanasto, joka on sovellusaluekielen perusta. Kielen tarkoituksena on liittää piirteet formaalisti määriteltyihin kuvauksiin sovellusalueesta.



Kuva 3.3: Luokitteluvaihe

3.1.5 Huomioita Arangon malli sovellusalueanalyysista on selkeästi jaettu pää- ja alavaiheisiin. Siinä on tarkasti kuvattu kukin osio ja sen tavoitteet. Tuoteperheet on otettu siinä huomioon. Kuitenkin paljon kohtia jätetään avoimiksi. Millaisia metodeja kussakin vaiheessa pitäisi käyttää? Minkä muotoista tiedon pitää olla eri vaiheissa? Minkä kriteerien mukaan tietoa jaotellaan? Kyseessä onkin enemmän abstrakti malli kuin toimiva metodi. Monia tarkemmin näihin kysymyksiin vastaavia metodeja on kehitetty tuotelinjoille pohjautuen sovellusalueanalyysiin. Seuraavaksi esitellään tunnetuimpia näistä menetelmistä.

3.2 ODM

ODM (*Organization Domain Modeling*) [51, sivu 196] on Marc A. Simosin 1990-luvun alussa kehittämä sovellusaluetekninen malli. Vaikka ODM ei ota kantaa yksityiskohtaisiin menetelmiin, käsittelee se huomattavasti Arangon luurankomallia tarkemmin eri vaiheita ja tarjoaa yleisellä tasolla ratkaisuja ongelmiin tuoteperheen määrittelyssä.

Luonteeltaan ODM on iteratiivinen malli, joka jakaantuu kahteen päävaiheeseen: deskriptiiviseen ja preskriptiiviseen. Deskpritiivisessä vaiheessa keskitytään sovellusalueen määrittelyyn käyttäen materiaalina perinnejärjestelmiä (engl. *legacy system*). Vaihe muistuttaakin paljolti käänteistekniikkamenetelmiä (engl. *reverse engineering*). Preskriptiivinen vaihe jakaantuu kolmeen alavaiheeseen: esimerkkisovellusalueiden tutkimiseen, näytejoukon tutkimiseen ja sovellusaluemallin luomiseen. Nämä vaiheet ovat sisäkkäisiä ja tavallaan jokainen vaihe pienentää tutkittavan informaation määrää jalostamalla sitä.

3.2.1 Deskriptiivinen vaihe Deskriptiivinen vaihe [51, sivu 200] aloitetaan valitsemalla esimerkkisovellusalueeseen mallijärjestelmiä, vastaesimerkkejä, ja rajatapauksia, jotka luokitellaan omiin luokkiinsa. Luokkien sisällä kiinnitetään huomiota myös järjestelmien välisiin historiallisiin (esim. teknologiat), toiminnallisiin (esim. alijärjestelmät) ja käsitteellisiin suhteisiin (esim. sovellusalueen laajuus). Vaiheen tuloksena syntyy sovellusalueyhteysmalli (engl. *domain interconnection model*), joka määrittää formaalisti valittuun sovellusalueeseen (engl. *domain of focus*) liittyvät suhteet eri luokkien välillä. Seuraavaksi valitaan tutkituista järjestelmistä edustava näytejoukko, johon perehdytään tarkemmin. Tästä näytejoukosta pyritään keräämään monipuolinen tietomäärä liittyen järjestelmän elinkaareen. Tiedon perusteella muodostetaan malleja, joita tulkitaan ja yhdistellään. Tuloksena syntyy sovellusalueen vaihtelevuutta kuvaavaa informaatiota. Deskriptiivisen vaiheen viimeisessä alavaiheessa syötteenä saatua tietoa muokataan ja sen pohjalta tehdään uudistavia muunnoksia muodostettavaan sovellusaluemalliin. Tämä malli on deskriptiivisen vaiheen tulos ja toimii pohjana siirryttäessä preskriptiiviseen vaiheeseen.

3.2.2 Preskriptiivinen vaihe Preskriptiivinen vaihe [51, sivu 200] liittyy lähemmin toteutukseen, sen suunnitteluun ja mallintamiseen. Vaihe jakaantuu kahteen alavaiheeseen: elementtimallin (engl. *asset base model*) ja elementtiarkkitehtuurin (engl. *asset base architecture*) luomiseen. Siirryttäessä elementtimallin luomiseen määritellään sovellusaluemallin pohjalta toiminnallisuus, jonka toteutettavat elementit tulevat kattamaan. Tämän dokumentin pohjalta luodaan arkkitehtuurimalli, jonka perusteella elementtejä aletaan toteuttaa.

3.2.3 Huomioita ODM menetelmänä vain linjaa vaiheet ja määrittää niiden tulokset. Itse vaiheiden toteutus ja tulosten muoto jätetään avoimiksi. Kuitenkin se ottaa huomioon monia yleisiä sovellusalueteknisiä ongelmia ja pyrkii ratkaisemaan ne. ODM ottaa kantaa siihen mitä tulee ottaa huomioon määriteltäessä tuoteperheen sovellusaluetta ja keiden tulisi ottaa osaa tähän prosessiin. ODM ottaa kantaa myös piirteiden käyttöön sovellusalueanalyyissa [51, sivu 203] ja määrittelee formaalien piirteiden avulla tapahtuvaa mallintamista.

3.3 FeatuRSEB

FeatuRSEB (*Featured RSEB*) [22, sivu 1] on kahden sovellusalueanalyysimetodin yhdistelmä. Siinä yhdistyvät piirreorientoitunut menetelmä FODA (*Feature-Oriented Domain Analysis*) [31] ja käyttötapauksiin perustuva menetelmä RSEB (*Reuse-Driven Software Engineering Business*). FODA on kehitetty SEI:ssä [50] ja sen perusteellinen sovellusalueanalyysiprosessi teki siitä yhden 1990-luvun käytetyimmistä metodeista. RSEB taas on tarkoitettu toisiinsa liittyvien sovellusten rakentamiseen uudelleenkäytettävistä elementeistä. Sille ominaista ovat selkeät käyttötapausmallit kaikissa vaiheissa. RSEB perustuu Jacobsonin *OO Software Engineering* - ja *Business Engineering* -teoksiin. RSEB on perinyt paljon piirteitä ODM-menetelmältä erityisesti sovellusalueen määrittelyä koskien. Erikoista FeatuRSEB-yhdistelmässä on käyttötapausten ja piirremallin käyttö rinnakkain. FeatuRSEB kehitettiin alunperin puhelinverkkosovelluksia varten.

FeatuRSEB koostuu seitsemästä vaiheesta: Sovellusalueen määrittely, tarpeiden, suuntausten ja esimerkkien valinta ja analyysi, piirrejoukkojen tunnistaminen, tekijöiden löytäminen (engl. *factoring*) ja ryhmittely, sovellusaluemallin tai geneerisen mallin ja arkkitehtuurin kehittäminen, hyödyllisten yhteneväisyyksien ja eriävyyksien esittäminen, niiden hyödyntäminen sekä uudelleenkäytettävien elementtien toteutus, sertifiointi ja pakkaus.

Vaihe	Nimi
1	Sovellusalueen määrittely
2	Tarpeiden, suuntausten ja esimerkkien valinta ja analyysi
3	Piirrejoukkojen tunnistaminen, tekijöiden löytäminen ja ryhmittely

Taulukko 3.1 – Jatkuu edelliseltä sivulta

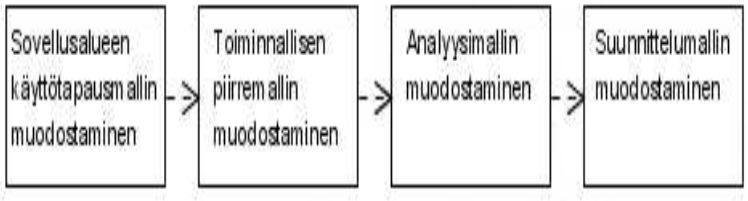
Vaihe	Nimi
4	Sovellusaluemallin tai geneerisen mallin ja arkkitehtuurin kehittäminen
5	Hyödyllisten yhteneväisyyksien ja eriävyyksien esittäminen
6	Yhteneväisyyksien ja eriävyyksien hyödyntäminen
7	Uudelleenkäytettävien elementtien toteutus, sertifiointi ja pakkaus

Taulukko 3.1: FeatuRSEBin vaiheet

FeatuRSEB muistuttaa alkuvaiheiltaan perussovellusalueanalyysia. Keskitytään tämän vuoksi tutkimaan käyttötapauksien ja piirteiden integrointia samaan metodiin ja sen tuomaa hyötyä. Käyttötapausmallilla mallinetaan toiminnallisuutta, ja siihen tallennetaan käyttäjien toimia järjestelmässä. Piirremallissa määritellään tuotteiden ominaisuudet uudelleenkäytettävyyttä silmällä pitäen. Käyttötapaukset siis vastaavat kysymykseen, mitä tuotteet tekevät, ja piirteet siihen, mistä järjestelmä koostuu.

3.3.1 Piirremalli Piirremalli [22, sivu 3] on itse asiassa keskeinen elementti menetelmässä. FeaturRSEB toteuttaa Philippe Kruchtenin kehittämää ”4+1”-mallia [40], jossa tuotetaan useita eri näkymiä arkkitehtuurista ja liitetään ne toisiinsa yhden keskeisen ”+1”-mallin avulla [22, sivu 2]. Määriteltäessä sovellusaluetta havaitaan esimerkkijärjestelmien kesken eroavaisuuksia. Jotta nämä eroavaisuudet pystyttäisiin luontevasti kuvaamaan, luokitellaan piirteet FeatuRSEB-menetelmässä välttämättömiin (engl. *mandatory*), valinnaisiin (engl. *optional*) ja vaihtoehtoisiin (engl. *variant*). Välttämättömät piirteet kuvaavat sovellusalueen ydinominaisuuksia ja luovat pohjan sovellusalueen infrastruktuurille. Valinnaiset piirteet ovat toisarvoisia piirteitä, jotka eivät esiinny kaikissa näytejoukon järjestelmissä. Vaihtoehtoiset piirteet kuvaavat erilaisia konfiguraatioita välttämättömille ja vaihtoehtoisille piirteille. Yhdessä käyttötapausmallin kanssa piirremalli tarjoaa kattavan ohjeen järjestelmän ominaisuuksista ja eri konfiguraatioista. Griss et al. [22, sivu 3] toteavat, että vastaavaa kattavuutta ei saavuteta missään muussa sovellusaluemallissa.

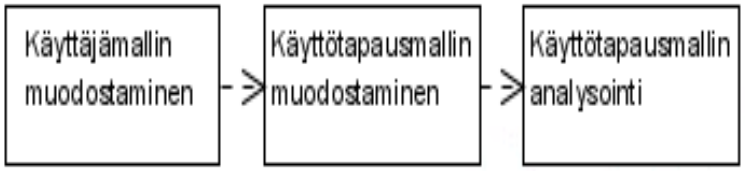
FeatuRSEB määrittelee, miten piirremalli ja käyttötapausmalli tulisi rakentaa rinnakkain. Näin mallit tarjoavat toisilleen materiaalia prosessin aikana. Piirremallin rakentaminen on nelivaiheinen prosessi. Ensimmäisessä vaiheessa yhdistetään näytesovellusten käyttötapausmallit yhdeksi sovellusalueen käyttötapausmalliksi. Lisäksi mallien variaatiopisteet dokumentoidaan. Toisessa vaiheessa syötteenä saadun sovellusaluemallin pohjalta muodostetaan toiminnallinen piirremalli. Kolmannessa vaiheessa muodostetaan RSEBin mukainen analyysimalli (engl. *analysis object model*). Tässä vaiheessa piirremallia laajennetaan analyysimallista saatavilla arkkitehtuurisilla rakennetta ja asetuksia kuvaavilla piirteillä. Neljännessä vaiheessa muodostetaan RSEBin suunnittelumalli (engl. *design model*). Tässä vaiheessa piirremalliin lisätään toteutukseen liittyvät piirteet.



Kuva 3.4: Piirremallin rakennusvaiheet

Piirremalli koostuu piirre-elementeistä ja niiden välisistä suhteista. Nämä elementit ja suhteet muodostavat puita tai verkkoja, joiden avulla saadaan nopeasti käsitys järjestelmän ominaisuuksista ja niiden välisistä suhteista. Piirremallista on olemassa korkean tason näkymä, jossa piirrekohtaista tietoa on vähän. Piirteiden osajoukoista voidaan muodostaa tarkempia UML-luokkakaavioita, joissa piirteet esitetään stereotyyppeinä ja suhteet esimerkiksi perintä- ja riippuvuusuhteilla. Käytetyt notaatiot esitetään tarkemmin FeatuRSEBia käsittelevässä artikkelissa *Integrating Feature Modeling with RSEB* [22, sivu 3].

3.3.2 Käyttötapausmalli Käyttötapausmalli rakentuu kolmen osion varaan. Ensin konstruoidaan sovellusalueen käyttäjämalli (engl. *domain actors model*). Tämä tapahtuu tunnistamalla aktorit tutkituilta sovellusalueilta. Toisiaan muistuttavat aktorit yhdistetään yhdeksi käyttäjäksi ja annetaan näille käyttäjille abstraktit roolinimet. On tärkeää jäljittää uudet aktorit alkuperäisiin järjestelmiinsä. Vastaava operaatio suoritetaan myös eri sovellusalueiden käyttötapausmalleille. Ensin korvataan vanhat aktorit uusilla. Tutkimalla samankaltaisia käyttötapauksia abstrakteille käyttäjille muodostetaan abstraktit käyttötapaukset. Sitten jäljitetään konkreetit ja abstraktit käyttötapaukset alkuperäisiin. Lopuksi analysoidaan muodostettu sovellusalueen käyttötapausmalli. Arvioidaan mallin yhtenäisyyttä, epäjohdonmukaisuuksia ja turhaa tietoa sekä muokataan sitä havaittujen epäkohtien korjaamiseksi. Tämän vaiheen jälkeen malli on valmis.

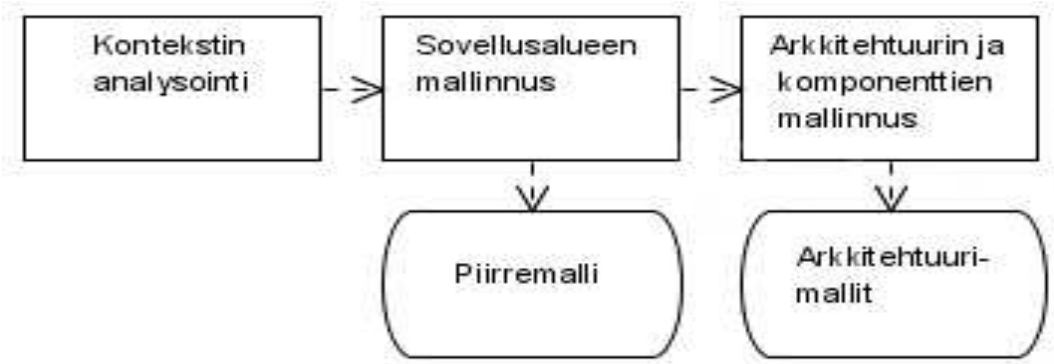


Kuva 3.5: Käyttötapausmallin rakennusvaiheet

3.4 FORM

FORM (*Feature-Oriented Reuse Method*) [32, sivu 143] on menetelmä sovellusalueen järjestelmien yhtenevyyksien ja eroavaisuuksien havaitsemiseen sekä analysointiin piirrekeskeisellä tavalla. Havaintojen pohjalta kehitetään sovellusaluearkkitehtuureja ja elementtejä. Keskeisessä roolissa FORMissa on monikerroksinen piirremalli, jolla kuvataan sovellusalue kattavasti. FORM on FODAn [31] laajennettu ja kehitetty versio, joka kattaa myös toteutusvaiheen. Sovellusaluetekniikka (engl. *domain engineering*) ja sovellustekniikka (engl. *application engineering*) ovat FORMin kaksi päävaihetta. Keskitymme tässä tutkimaan vain ensimmäistä vaihetta tarkemmin. FORMia ovat olleet kehittämässä puhelinverkko-organisaatiot ja sitä onkin sovellettu laajasti juuri televerkkosovellusalueella.

FORMin sovellusaluetekniikkavaiheessa on kolme alaosiota: konteksin analysointi, sovellusalueen mallinnus sekä arkkitehtuurin ja elementtien mallinnus. Kontekstin analysoinnilla sovellusalue, sen aiottu käyttötarkoitus, ulkoiset suhteet ja vaikuttajat pyritään määrittelemään sovellusalueanalyysia varten. Sovellusalueanalyysissa sovellusalueen eroavuudet ja yhtenevyydet pyritään kartoittamaan ja esittämään hyödynnettävässä muodossa.

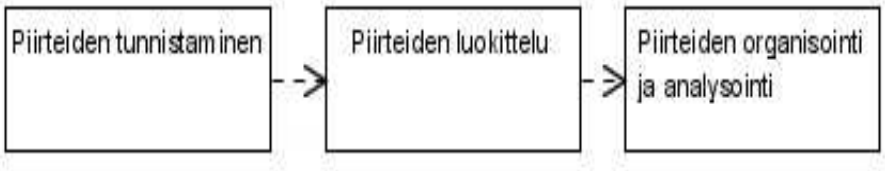


Kuva 3.6: FORMin sovellusaluetekniikkavaihe

3.4.1 Sovellusalueanalyysi Sovellusalueanalyysi muodostuu FORMissa kontekstin analysoinnin tuloksista ja sovellusalueen mallinnuksesta. Se on kolmivaiheinen prosessi, joka sisältää suunnittelun, piirreanalyysin ja tulosten validoinnin. Suunniteluvaiheessa tuoteperhe tunnistetaan ja ennakoivaa yhteneväisyyksien arviointia suoritetaan. Piirreanalyysissa tunnistetaan tuotteen piirteet, luokitellaan ne ja järjestetään niistä yhtenäinen malli. Piirremallin validoinnilla tarkoitetaan kaikkien sovellusalueanalyysissa käytettyjen sovellusten mallintamista piirremallin avulla ja mallin kuvaavuuden tarkastelua. Arkkitehtuurin ja elementtien mallinnus rakentuu useampien osamallien varaan, jotka kuvaavat järjestelmän rakennetta eri näkökulmista. FORMissa käytetyt arkkitehtuurimallit ovat alijärjestelmämalli, prosessimal-

li ja moduulimalli. Näitä käytetään viitteenä toteutettaessa varsinaista tuoteperheen arkkitehtuuria.

3.4.2 Piirremallin määrittely Tutustutaan FORMissa keskeisen piirremallin [32, sivu 150] rakentamiseen tarkemmin. Piirremalli rakentuu sovellusalueanalyysissa kolmen vaiheen kautta. Ensin tunnistetaan piirteet. Vaihe vaatii vahvaa sovellusalueen tuntemusta ja monipuolista osaamista. Kang et al. [32, sivu 151] mainitsevat myös lähdekoodit, käyttöohjeet, kirjallisuuden ja suunnitteludokumentit hyvinä piirteiden lähteinä. Piirteet tunnistetaan kyvykkyyden (engl. *capability*), toimintaympäristön, sovellusalueteknologian ja toteutustekniikan näkökulmasta.



Kuva 3.7: Piirremallin rakennusvaiheet

Seuraavaksi tunnistetut piirteet luokitellaan sisältämänsä informaation mukaan kyvykkyyksiin sekä toimintaympäristöllisiin, sovellusalueteknologisiin ja toteutusteknisiin piirteisiin. Kyvykkyyspiirteet määrittävät jonkin palvelun, operaation tai muun toiminnon, joka sovelluksella voi toimialueella olla. Kyvykkyyspiirteet voidaan jakaa toiminnallisiin ja ei-toiminnallisiin piirteisiin. Piirremallin tulisi sisältää mahdollisimman monesta tuotteesta kaikkia eri tyyppisiä piirteitä.

Piirteiden organisointi ja analysointi liittyy piirteiden välisiin suhteisiin ja näiden mallintamiseen. Piirteet järjestetään loogisista AND/OR-suhteista koostuvaan puuhun. Piirrepuu koostuu neljästä kerroksesta aiemmin tehdyn näkökulmiin perustuvan luokittelun mukaisesti. Tällä pyritään piirteiden monipuoliseen mallintamiseen. Piirteet itsessään voivat olla välttämättömiä (engl. *mandatory*), valinnaisia (engl. *optional*) tai vaihtoehtoisia (engl. *alternative*). Piirteiden välillä voi olla erilaisia suhteita kuten toteutussuhde (engl. *implemented-by*) ja koostamissuhde (engl. *composed-of*). Nämä suhteet voivat toteutua myös eri kerroksissa olevien piirteiden välillä. Erilaisiin suunnittelupäätöksiin ja valintoihin liittyvä tieto tallennetaan piirteistä tehtäviin sanallisiin kuvauksiin, jotka ovat perustana myös piirrepuulle. Kang et al. ovat havainneet, että ylemmällä tasolla sijaitsevat AND-suhteet ja alemmilla tasoilla sijaitsevat OR-suhteet luovat hyvän pohjan uudelleenkäytettäville elementeille. Suhteiden vallitessa toisinpäin uudelleenkäyttömahdollisuuksia on paremminkin matalan tason kirjastoissa.

3.4.3 Arkkitehtuurimallit ja niiden määrittely FORMissa piirremallin pohjalta kehitettävät arkkitehtuurimallit [32, sivu 155] ovat tärkeässä osassa toteutusta suun-

niteltaessa. FORMissa käytetään alijärjestelmämallia, prosessimallia ja moduulimallia. Tämän vaiheen tarkoituksena on liittää tunnistetut ja analysoidut piirteet arkkitehtuuriin ja sen elementteihin. Piirteiden perusteella mallinnetaan eri elementtiyhdistelmiä. Piirremallin loogisten rajoitusten tulee vastata arkkitehtuurimallien fyysisiä rajoituksia. Tämä vähentää ristiriitoja piirteiden välillä. Aikakriittiset alijärjestelmät voidaan mallintaa omiksi alijärjestelmikseen ja samaa tietoa käyttävät piirteet omaan alijärjestelmäänsä.

Alijärjestelmämalli kuvaa järjestelmän kokonaisrakennetta ja sen jakamista alijärjestelmiin. Tämä tapahtuu liittämällä kyvykkyydet (engl. *capabilities*) toimintoryhmiin (engl. *function block*). Tärkeää on myös mallintaa alijärjestelmien ja suorittavien resurssien yhteydet. Piirteitä voidaan joutua jalostamaan tätä mallia muodostettaessa. Malliin määritellään myös alijärjestelmien rajapinnat, niiden välinen tiedonkulku (esim. viestikäytävät, palvelupyynnöt) sekä suorittavat resurssit. Mallin tulee ottaa huomioon myös suorituskykyyn liittyvät tavoitteet.

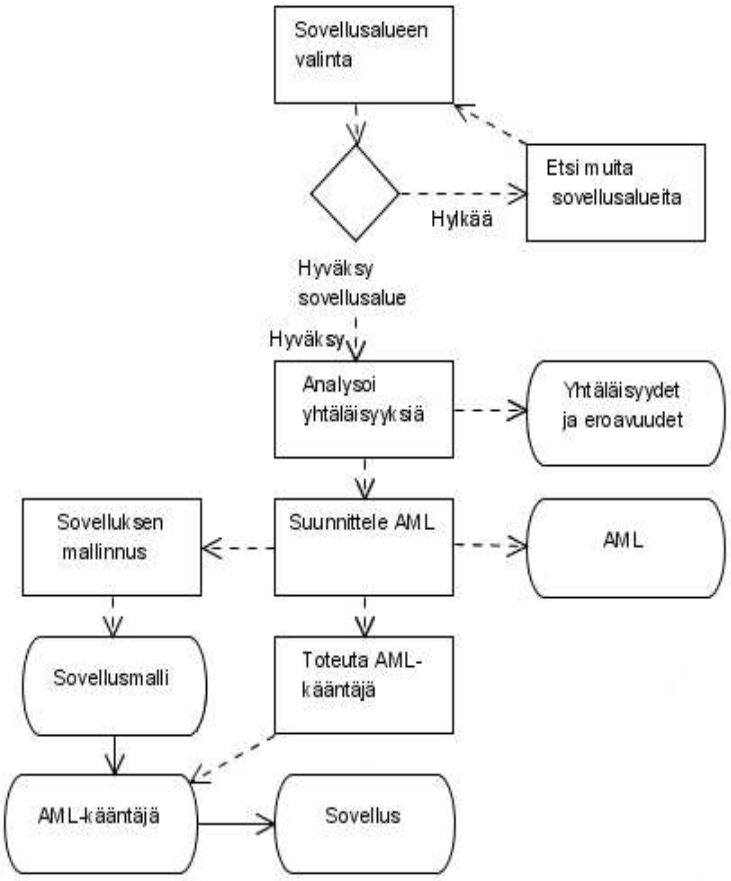
Prosessimalli kuvaa kunkin alijärjestelmän ajonaikaista käyttäytymistä. Mallissa kukin alijärjestelmä kuvataan prosessien avulla. Jako prosesseihin tulisi suorittaa ottaen huomioon muunmuassa prosessien yhtenäisyys, eriytettävyys ja suoritustiheys. Esimerkiksi luonteeltaan jatkuvat ja vaiheittaiset prosessit tulee eriyttää. Prosessit voidaan jakaa pysyviin tai vaiheittaisiin, ja yksittäisiin tai rinnakkaisiin. Aikakriittiset operaatiot voidaan mallintaa omana prosessinaan. Mallia voidaan yksinkertaistaa esittämällä samaan aikaan esiintyvät prosessit ja saman tapahtuman laukaisemat prosessit yhtenä.

Moduulimalli rakentuu kaikkien piirremallin kerrosten pohjalta. Moduulimallin tulisi liittää piirteet moduuleihin. Moduulihierarkia ja piirrehierarkia vastaavat paljolti toisiaan mutta moduuli voi sisältää myös useita vaihtoehtoisia piirteitä. Kukin moduuli määritellään abstraktisti. Tämä määrittely voidaan suorittaa käyttämällä esiohjelmoituja elementtejä, käyttäen parametrisoitavia malleja tai valitsemalla luurankokoodi ja täydentämällä sitä.

3.5 FAST

FAST [57, sivu 43] on malli tuoteperheiden määrittelylle ja tuottamiselle. Keskeisinä ajatuksina FASTissa ovat muutosten ennustaminen, käsitteiden erillisyys, abstraktiot ja tiedon piilotus. Näiden avulla pyritään tehokkaasti yhdistämään suunnittelu ja toteutus. FAST määrittelee organisaatioroolit käytettäväksi itse prosessissa. Kaikki FASTia noudattavat prosessit jakaantuvat kolmeen alaprosessiin: sovellusalueen hyväksyntä (engl. *qualifying the domain*), sovellusaluevaihe (engl. *domain engineering*) ja sovellusvaihe (engl. *applications engineering*). Ensimmäisessä vaiheessa suoritetaan kannattavuustutkimusta tuoteperheelle ja tutkitaan tuotteiden määrää sekä tuotantokustannuksia. Sovellusaluevaiheessa määritellään ja toteutetaan tuotantoprosessi

ja -ympäristö sovellusvaiheelle. Sovellusvaihe käyttää näitä tuotteiden toteuttamiseen. FAST on kehitetty *Lucent Technologies* -yrityksessä, jossa sitä myös sovelletaan [57, sivu 5]. FASTia on käytetty Lucentilla esimerkiksi konfiguraatiojärjestelmiin.



Kuva 3.8: FAST-prosessi

3.5.1 Organisaatioroolit FASTia toteuttavissa organisaatioissa on oma ryhmänsä sovellusaluevaiheelle ja toinen sovellusvaiheelle [57, sivu 59]. Sovellusalueryhmä on vastuussa kehityksen jatkumisesta. Sovellusryhmä vastaa tuotteiden toteutuksesta tehtyjen sopimusten ja määrittelyjen mukaan. Ryhmät ovat hierarkkisia kokonaisuuksia, ja ryhmän jäsenten tehtävät voidaan järkevästi jakaa eri vaiheissa. Tämä perustuu oletukseen, että myös vaiheet on järjestetty vastaaviin hierarkioihin.

3.5.2 Sovellusaluevaihe Sovellusvaiheen [57, sivu 53] tehtävänä on AML-kielen (*application modeling language*) kehittäminen, toteutusympäristön kehittäminen, toteutusprosessin määrittely ja näiden kautta tuoteperheen jäsenten toteutuksen mahdollistaminen. Sovellusaluevaihe on iteratiivinen prosessi, joka alkaa analysoimalla tuoteperhettä. Analyysin kohteena ovat tuoteperheen yhteneväisyydet ja eroavuu-

det. Tuloksena saavutetaan tieto, onko tuoteperheen jäsenten välillä riittävästi yhteisiä ominaisuuksia tuotelinjaan. Tuloksena syntyy myös määritys tuoteperheestä, jonka pohjalta kehitysympäristö laaditaan. FASTissa tätä määrittelyä kutsutaan sovellusaluemalliksi. Keskeinen osa sovellusaluemallia on AML, jonka avulla tuoteperheen jäseniä voidaan määritellä. Tuloksena sovellusaluevaiheesta saadaan taloudellinen malli tuoteperheestä, tuoteperheen määrittely, prosessin määrittely, AML:n määrittely, toteutusympäristö, mallikirjasto (engl. *template library*), koodigeneraattori ja dokumentointi tästä, analysointityökalut ja dokumentointi niistä, ohjeet ympäristön käytöstä sekä tuotteen kokoamisohjeet.

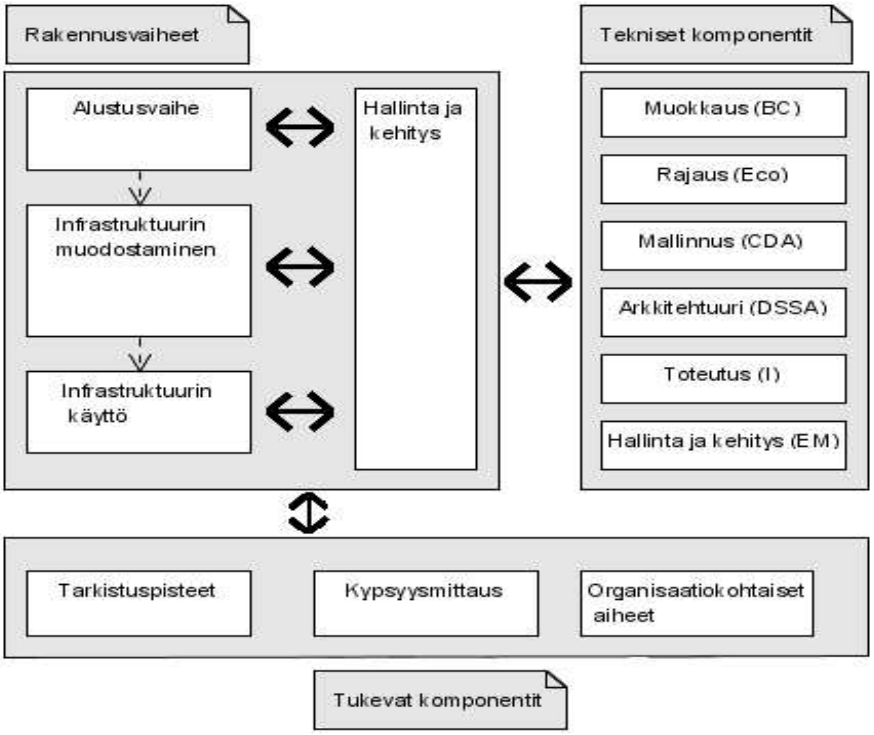
3.5.3 Sovellusvaihe Sovellusvaiheen [57, sivu 49] tarkoituksena on selvittää toteutettavan tuotteen vaatimukset nopeasti ja tuottaa sovellus. Sovellusryhmä käyttää valmiita resursseja toteuttaakseen asiakkaan vaatiman tuotteen. Sovellusvaihe voidaan tiivistää asiakkaiden vaatimusten analysointiin, tuotteen kasaamiseen ja sen dokumentoinnin toimittamiseen asiakkaalle. Kyseessä on kuitenkin iteratiivinen prosessi, jonka alkuvaiheessa voidaan tuottaa useita malleja ennen varsinaista tuotetta. Tärkeä kokonaisuus tässä vaiheessa on AML. Kieltä käytetään juuri mallien laatimiseen. Ympäristön täytyy luonnollisesti tarjota menetelmät analysoida näitä malleja. Toimivasta mallista generoidaan lopullisen asiakkaan hyväksymän instanssin lähdekoodi. Tuloksena sovellusvaiheesta syntyy AML-kielinen malli sekä mallista generoitu jaettava lähdekoodi ja dokumentointi.

3.6 PuLSE

PuLSE (*Product Line Software Engineering*) [5, sivu 3], [10, sivu 122] on IESEssä [29] 1990-luvun lopulla kehitetty metodi tuotelinjojen määrittelyyn ja toteuttamiseen. PuLSE on tuotekeskeinen metodi toisin kuin muut esitellyt menetelmät. Se ei pyri sovellusalueen vaan tarkemmin linjan tuotteiden määrittämiseen. PuLSE on myös muokattavissa projektikohtaisesti. PulSE ei ole kehitetty minkään tietyn sovellusalueen tarpeisiin, ja sitä on sovellettu ohjelmistokehityksessä esimerkiksi tavaratietojärjestelmien toteutuksessa ja simulointiytimen uudelleensuunnittelussa [49].

Menetelmä rakentuu kolmesta elementistä: sijoitusvaiheet (engl. *deployment phases*), tekniset komponentit (engl. *technical components*) ja tukevat komponentit (engl. *supporting components*) [2, sivu 3]. Tekniset ja tukevat komponentit ovat tässä yhteydessä PuLSE-menetelmän osia, eikä niitä pidä sekoittaa kehitettävien ohjelmistojen elementteihin tai komponentteihin. Sijoitusvaiheet kuvaavat PuLSE-prosessin, ja miten tukevia ja teknisiä komponentteja käytetään eri vaiheissa. Jokaiseen vaiheeseen liittyy teknisiä komponentteja ja myös sijoitusvaiheet voivat liittyä toisiinsa tarvittaessa. Sijoitusvaiheita on neljä: alustus, infrastruktuurin rakennus, infrastruktuurin käyttö sekä hallinta ja kehitys. Tekniset komponentit liittyvät kaikkiin sijoi-

tusvaiheisiin. Ne tarjoavat teknistä tietoa, jota komponenttiin liittyvä vaihe tarvitsee. Tukevat komponentit ovat tietoa tai ohjeistusta muiden elementtien käyttöön, jotka mahdollistavat PuLSEn tehokkaamman käytön.



Kuva 3.9: PuLSEn vaiheet ja komponentit

3.6.1 Tukevat komponentit Tukevat komponentit [10, sivu 122] jakaantuvat kolmeen ryhmään: tarkistuspisteet (engl. *project entry points*), kypsyysmittaus (engl. *maturity scale*) ja organisaatiokohtaiset aiheet (engl. *organizational issues*). Tarkistuspisteet liittyvät metodin muokkaukseen yleisimmille projektityypeille. Esimerkiksi useamman projektin integrointi tai vanhojen elementtien hyödyntäminen voidaan huomioida tarkistuspisteissä. Kypsyysmittaus tuottaa PuLSEn käyttöönotto-ohjeen yrityksille. Organisaatiokohtaiset aiheet sisältävät ohjeita oikean organisaatiorakenteen ja tuotelinjan ylläpitämiseen ja kehittämiseen.

3.6.2 Tekniset komponentit Tekniset komponentit [10, sivu 122] jaetaan kuuteen ryhmään: muokkaus (BC), rajaus (Eco), mallinnus (CDA), arkkitehtuuri (DSSA), toteutus (I) sekä hallinta ja kehitys (EM). Muokkaus liittyy PuLSEn muunteluun alustusvaiheessa. Mallinnus kuvaa, miten tuotekohtaiset piirteet mallinnetaan ja rajataan juuri tuoteperheen tuotteisiin. Arkkitehtuuri kuvaa, miten viitearkkitehtuuri rakennetaan säilyttäen jäljitettävyys alkuperäiseen sovellusaluemalliin. Toteu-

tus kuvaa infrastruktuurin käyttöönottovaihetta. Hallinta ja kehitys liittyvät käytön myötä löytyvien tuotelinjan epäkohtien korjaukseen ja linjan konfigurointiin.

3.6.3 Alustusvaihe Alustusvaiheessa [10, sivu 124] PuLSEsta räätälöidään käyttäjän tarkoitukseen sovellettava versio. Alustusvaihe jakaantuu kolmeen alaosioon: perusvaihe (engl. *baselining*), arviointi (engl. *evaluation*) ja räätälöinti (engl. *customization*). Vaiheiden tavoitteena on muokata BC-komponenttia käyttötarkoitukseen sovellettavaksi. Perusvaiheessa kerätään räätälöimiseen tarvittavaa tietoa. Tämän tiedon määrittelevät muuntelutekijät (engl. *customization factors*), jotka kuvaavat tuotelinjaa kuvaavia tekijöitä. Ensin valitaan tarvittavat muuntelutekijät käyttäen tukevia komponentteja. Perusvaihestrategioita käyttäen kerätään tarvittava tieto. Tuloksena on profiili nykyisestä tilasta. Arviointivaiheessa kerätty tieto ja tekijöiden väliset riippuvuudet analysoidaan ja muodostetaan päätöspuu. Tuloksena on toteutusprofiili, joka sisältää räätälöintiin liittyvät päätökset. Kustomointivaiheessa lopullinen prosessi määritellään valmiiksi käyttöönottoa varten.

3.6.4 Infrastruktuurin muodostaminen Infrastruktuurin muodostaminen [10, sivu 125] on vaiheena monimutkainen ja jakaantuukin kolmeen alatehtävään, joista jokaisen suorittaa tekninen komponentti. PuLSE-Eco rajaa tuotelinjan taloudellisen näkymän (engl. *scope*). PuLSE-CDA määrittelee tuotelinjan kokonaisuudet ja niiden väliset suhteet. PuLSE-DSSA määrittelee viitearkkitehtuurin tuotelinjalle.

PuLSE-Eco aloitetaan rajaamalla odotettavat tuoteperheen jäsenet. Perheenjäsenistä erotetaan tuotepiirteet ja validoidaan ne. Tuloksena syntyy tuotekartta. Taloudellinen analyysi suoritetaan käyttäen arviointifunktioita, jotka sisältävät piirrefunktioita (engl. *characterization function*) ja hyötyfunktioita (engl. *benefit function*). Funktioiden tuoma tieto lisätään tuotekarttaan. Tärkeä osuus PuLSE-Ecoa on hyötyanalyysi, jossa piirrefunktioiden arvoja käytetään hyötyfunktioissa ja määritellään niitä käyttäen taloudellinen näkymä.

PuLSE-CDA jalostaa PuLSE-Econ muodostamaa taloudellista näkymää. Tässä alitehtävässä määritellään tuotelinjan kokonaisuudet, niiden väliset suhteet ja rakenne. Vaiheen tehtävää määrittelee PuLSE-BC. Näkymän määrittelyä seuraa mallinnus. Mallinnukseen käytetään kuvaustauluja (engl. *storyboard*) ja muita vaihtoehtoisia malleja. Yksittäisten tuotteiden malleista muodostetaan geneerisiä kuvaustauluja ja lopulta muodostetaan päätösmalli (engl. *decision model*), joka kuvaa tuotelinjaa ja sen vaihtelevuutta.

PuLSE-DSSA sisältää sovellusaluekohtaisen arkkitehtuurin muodostamisen. Arkkitehtuuri muodostetaan skenaarioiden [28, sivu 93] avulla. Ne jaetaan geneerisiin skenaarioihin ja ominaisuuksiin liittyviin skenaarioihin. Skenaario on kuvaus järjestelmän toiminnosta, joka voi liittyä toiminnalliseen vaatimukseen tai laadulliseen vaatimukseen. Skenaariot perustuvat aiemmissa vaiheissa laadittuihin malleihin ja

ne luokitellaan arkkitehtuurisen merkityksensä mukaan. Joukko geneerisiä skenaarioita valitaan ja sovelletaan arkkitehtuurikandidaattiin. Tätä vaihetta toistetaan iteratiivisesti ja arkkitehtuuria jalostetaan saatujen tulosten mukaisesti. Jos useampi arkkitehtuuri on mahdollinen, ominaisuuksiin littyviä skenaarioita käytetään arkkitehtuurin valintaan. Arkkitehtuuria jalostettaessa kerätään tietoa ja suunnittelupäätöksiä konfiguraatiomalliksi laajentamaan päätösmallia.

3.6.5 Infrastruktuurin käyttövaihe Infrastruktuurin käyttövaihe [10, sivu 127] sisältää yhden tuotelinjan jäsenen määrittelyn, toteuttamisen ja validoinnin. Muutoksia olemassaoleviin malleihin ei tehdä, vaan muutostarpeet siirretään eteenpäin PuLSE-EM:lle. Ensimmäisessä alavaiheessa tuotelinjamalli instantioidaan ja validoidaan. Käyttäen päätösmallia uusi instanssi määritellään. Syntynyt määrittely validoidaan käyttäjän vaatimukset huomioon ottaen. Seuraavassa alavaiheessa DSSA instantioidaan ja validoidaan. Arkkitehtuuri määritellään käyttäen konfiguraatiomallia ja tuotemäärittelyä. Seuraavaksi arkkitehtuuri validoidaan käyttäen tuotemäärittelyä. Tämän jälkeen matalan tason suunnittelu ja ohjelmakoodi tuotetaan. Jokainen elementti kuuluu joko olemassaoleviin elementteihin, näkymään kuuluviin mutta vielä toteuttamattomiin elementteihin tai erityisiä instanssikohtaisia vaatimuksia toteuttaviin elementteihin. Suunnittelu ja ohjelmakoodi validoidaan arkkitehtuurin kannalta. Tuote viimeistellään, ja sen on läpäistävä hyväksymiskoe, jonka asiakas suorittaa.

3.6.6 Hallinta- ja kehitysvaihe Hallinta- ja kehitysvaiheen [10, sivu 128] tehtävänä on valvoa ja muokata toteutettua tuotelinjaa ajan kuluessa. Muokkaukset kohdistetaan PuLSE-EM-komponenttiin. PuLSE-EM koordinoi muiden teknisten komponenttien toimintaa. Se saa syötteeksi muuttuvia tai uusia vaatimuksia, tutkii niiden vaikutukset ja aloittaa tarpeelliset toimenpiteet tuotelinjan muokkaamiseksi.

3.7 Määriteltyjen piirteiden hallinta

Piirteiden ja niiden välisten suhteiden hallintaan liittyy ongelmakohtia. Piirteiden välillä voi vallita ristiriitoja ja erilaisia riippuvuussuhteita. Miten näitä tulisi käsitellä, jotta ristiriitaisuuksilta vältyttäisiin tai ne voitaisiin korjata? Tässä kappaleessa tutkitaan piirteiden hallintaan liittyviä ristiriitaisuusongelmia ja hahmotellaan niihin olemassaolevia ratkaisuja.

3.7.1 Ristiriitaongelma Piirteet muodostavat monimutkaisen verkon. Ne voivat olla esimerkiksi toisensa poissulkevia, riippuvaisia toisistaan muodostaen isompia kokonaisuuksia, tai toteutusriippuvaisia. Vaikka tätä kokonaisuutta pyritään hallitsemaan erilaisten metodien, mallien ja tekniikoiden avulla, muodostuu laajasta ja

monimutkaisesta kokonaisuudesta ajan myötä vaikeasti ylläpidettävä ja kehitettävä [23, sivu 5], [25, sivu 5]. Muutokset voivat aiheuttaa yllätävän laajoja muutoksia piirreverkossa. Ongelman ydin on, että piirteet eivät aina yksiselitteisesti toteudu yksittäisessä elementissä tai elementtiryppäässä. Tällöin tuotteen kokoaminen piirteet valitsemalla on monimutkainen operaatio, jossa täytyy valita ristiriidattomat elementit. Kun piirre muuttuu, sen vaikutuksia joudutaan jäljittämään useisiin eri lähdekoodeihin, jossa muutokset voivat aiheuttaa uusia muutoksia.

3.7.2 Ratkaisuja Esitettyyn ongelmaan ei ole olemassa täydellistä ratkaisua. Sen vaikutusta ja laajuutta voidaan kuitenkin tehokkaasti ehkäistä. Mitä ylemmällä tasolla asiaan kiinnitetään huomiota, sitä parempia tulokset ovat. Määritellyn prosessin käyttö ja huolellinen dokumentointi sekä muutoksien suorittaminen noudattaen ennalta määriteltyä metodia ovat järkeviä valintoja. Tehokkaat suunnittelutyökalut helpottavat muutosten vaikutusten havaitsemista. Griss [23, sivu 8], [25, sivu 6] kiinnittää huomiota erityisesti jäljitettävyyteen, arkkitehtuuriin ja elementtien kokoamiseen osasista.

Tuotelinjaa käyttäessä ja ylläpidettäessä on tärkeä säilyttää jäljitettävyys piirteiden ja elementtien välillä. Muutos piirteisiin aiheuttaa muutoksia myös toiseen osapuoleen. Muutosten tekoa helpottaa piirteiden väliset suhteet hallitsevien kehitystyökalujen käyttö muutosten hallitsemiseen. Jos piirteiden ja elementtien välisiä yhteyksiä ei voida jäljittää, on muutosten teko mahdotonta.

Arkkitehtuurin, arkkitehtuurityylien ja suunnittelumallien käyttö johtaa paremmin jaettuun ja ylläpidettävämpään järjestelmään. Ohjelmakoodi on tällöin paremmin ylläpidettävää ja selvemmin jaoteltavissa osakokonaisuuksiin. Näin pyritään minimoimaan vaikutusten laajuutta muutoksia tehtäessä. Oikean muutokset huomioivan rakenteen löytäminen ja toteuttaminen vaatii vahvaa osaamista.

Piirteistä voidaan muodostaa aspekteja (engl. *aspect*), jotka voidaan koota luokiksi, elementeiksi tai tuotteiksi. Näin kokonaisuudet muodostuvat jäjitettävistä palasista, jotka voidaan koota sopivaksi kokonaisuudeksi esimerkiksi konfiguraatiotiedoston avulla.

3.8 Yhteenveto

Tuoteperheen määrittelyyn on olemassa useita eri menetelmiä. Jokaisella on oma tapansa lähestyä asetettua ongelmaa mutta menetelmissä on nähtävissä myös paljon yhteneväisyyksiä. Menetelmät kuten PuLSE, FAST tai FORM ovat laajoja resursseja vaativia metodeja, joissa eri vaiheet on määritelty huomattavasti tarkemmin kuin esimerkiksi Arangon sovellusalueanalyysissa. Arangon malli on enemmän ohjaava kuin suoraan sovellettava. Eri menetelmien paremmuutta toisiinsa verrattuna on vaikea lähteä arvioimaan. Sovellusalue, tottumukset, resurssit ja mieltymykset vai-

kuttavat valintaan. Tuoteperheen määrittely on joka tapauksessa haastava tehtävä, jossa määritelty toimintapa on tarpeen. Arangon mallissa vapaus toimintatapojen ja notaatioiden suhteen on suuri, mikä mahdollistaa sen joustavan käytön tilanteen mukaan. Tämän joustavuuden vuoksi sitä sovelletaankin määriteltäessä rannekellotuotelinjaa luvussa 8.

4 Arkkitehtuurin suunnittelu

Ohjelmistoarkkitehtuuri vaikuttaa ohjelmiston laadullisiin ominaisuuksiin. Samoin ohjelmistoarkkitehtuuri ohjaa elementtipohjaiseen ajatteluun ja parantaa ymmärtämystä järjestelmän toiminnasta. Tuotelinja on rakenteeltaan monimutkainen kokonaisuus, jota arkkitehtuurisuunnittelulla pyritään hallitsemaan. Tärkeää on myös täyttää tuoteperheelle asetetut laadulliset vaatimukset. Arkkitehtuurin suunnittelu on monivaiheinen prosessi, johon osallistujilta vaaditaan vahvaa tuntemusta toteutettavasta järjestelmästä ja kokemusta arkkitehtuurien suunnittelusta ja niiden toteutuksesta. Tuotelinjan arkkitehtuuria suunniteltaessa pitää ottaa huomioon tuoteperheen aiheuttamat erityispiirteet arkkitehtuuriin. Tässä luvussa perehdytään tuotelinjan arkkitehtuuriin ja sen suunnitteluun tarkemmin sekä kuvataan siihen liittyvät vaiheet ja kokonaisuudet.

4.1 Ohjelmistoarkkitehtuuri

Ohjelmistoarkkitehtuuri [44, sivu 3], [19, sivu 1], [6, sivu 21] määrittää järjestelmän rakenteen tai rakenteet. Tämä rakenne määritellään elementtien, elementtien näkyvien ominaisuuksien ja niiden välisten suhteiden kautta. Näkyvät ominaisuudet ovat muille elementeille näkyviä piirteitä kuten tarjottavat palvelut. Arkkitehtuuri on tavallaan abstraktio järjestelmästä, josta on poistettu epäolennainen elementtien sisäinen tieto.

Arkkitehtuuri kuvaa järjestelmän rakenteen tai rakenteet. Usein suuret järjestelmät koostuvat useista alijärjestelmistä, jotka voivat olla rakenteeltaan monimutkaisia ja muodostavat kokonaan oman arkkitehtuurin. Mallinnettavan järjestelmän kannalta ei tällä tiedolla kuitenkaan ole väliä. Mikään näistä rakenteista ei yksinään ole arkkitehtuuri vaan se koostuu näiden rakenteiden välisistä suhteista. Kaikilla järjestelmillä on arkkitehtuuri, koska kaikki järjestelmät koostuvat elementeistä ja niiden välisistä suhteista. Arkkitehtuurin suunnittelulla ei ole siis vaikutusta arkkitehtuurin olemassaoloon, vaikka se vaikuttaa arkkitehtuurin sopivuuteen ja hyvyyteen. Yksinkertaisimmillaan ohjelma itse on arkkitehtuurin ainoa elementti.

Arkkitehtuurin elementit tai komponentit ovat järjestelmän kokonaisuuksia, jotka voidaan järkevästi modularisoida. Elementti voi olla monimutkainen kokonaisuus, mutta arkkitehtuurin kannalta tämä tieto on epäolennaista ja sen vuoksi sitä ei arkkitehtuuriin mallinneta. Voidaan ajatella, että elementillä on yksityinen ja julkinen puoli, joista julkinen puoli on arkkitehtuurin kannalta kiinnostava. Elementtiä ei ole määritelty tarkemmin, joten se voi olla melkein mikä tahansa järjestelmän ark-

kitehtuuriin liittyvä osa kuten luokka, kirjasto tai tietokanta. Elementtien käyttäytyminen on olennainen osa arkkitehtuuria. Kaikilla elementeillä on rajapinta, joiden kautta ne ovat yhteydessä toisiinsa.

4.2 Arkkitehtuurityylit

Arkkitehtuureille on kehitetty arkkitehtuurityylejä [20, sivu 5], [44, sivu 4], [17, sivu 1] tai arkkitehtuurimalleja. Arkkitehtuurityylit määrittävät käyttämänsä elementit ja niiden väliset liittimet, joita tyylin toteutuksessa voidaan käyttää. Tyyli määrittelee myös rajoituksia siitä, miten elementtejä voidaan toisiinsa liittää. Rajoitukset voivat liittyä esimerkiksi arkkitehtuurin topologiaan. Arkkitehtuurityyliin voidaan liittää tietoa sen yleisimmistä sovellusalueista, laatuvaatimusten soveltuvuudesta ja siihen liittyvistä kustannuksista. Yleisiä arkkitehtuurityylejä ovat piippu-suodin, oliomalli, tapahtumamalli, kerrosmalli, tietovarasto, tulkki ja hajautettu arkkitehtuuri. Eri tyylit soveltuvat paremmin tietyille sovellusalueille kuin toiset riippuen asetetuista laatuvaatimuksista.

4.3 Arkkitehtuurille asetetut odotukset

David Garlan [18, sivu 94] määrittelee useita ohjelmistokehitykseen liittyviä piirteitä, joihin arkkitehtuurin käyttö vaikuttaa: ymmärrys järjestelmästä, uudelleenkäyttö, toteutus, kehitys, analysointi ja hallinnointi. Arkkitehtuuri on korkean tason abstraktio, jonka avulla saavutetaan nopeasti ymmärrys järjestelmästä, sen rakenteesta ja sen kyvystä täyttää asetetut kriteerit. Arkkitehtuurin avulla ohjelmistosuunnittelijat voivat lisäksi kuvata ja keskustella järjestelmään liittyvistä suunnittelupäätöksistä jo aikaisessa kehitysvaiheessa. Arkkitehtuuri ohjaa elementtipohjaiseen ajatteluun ja edesauttaa näin lähdekoodin uudelleenkäytettävyyttä. Arkkitehtuurin upottaminen esimerkiksi sovelluskehykseen, ja näin valmiin luurankoarkkitehtuurin tarjoaminen, on tästä hyvä esimerkki samoin kuin tuotelinjatkin. Niissä arkkitehtuurin avulla mallinnetaan variaatiopisteet ja elementtirajapinnat elementeille. Toteutukseen osalta arkkitehtuuri toimii mallina määrittäen elementit, niiden näkyvät ominaisuudet ja suhteet. Tuotteen evoluutiota ajatellen arkkitehtuuri mahdollistaa muutosten aiheuttamien vaikutusten paremman ymmärtämisen, koska elementtien väliset suhteet ovat selkeästi havaittavissa. Tämä helpottaa kustannusten arviointia. Arkkitehtuuri tarjoaa uusia mahdollisuuksia järjestelmän analysoimiseksi. Arkkitehtuuria analysoimalla saavutetaan parempi käsitys järjestelmästä, sen vaatimuksista ja niiden täyttymisestä. Arkkitehtuurin evaluointiin onkin kehitetty erilaisia metodeja kuten ATAM [6, sivu 271].

Bosch [11, sivu 10] korostaa vielä arkkitehtuurin merkitystä laatuvaatimusten

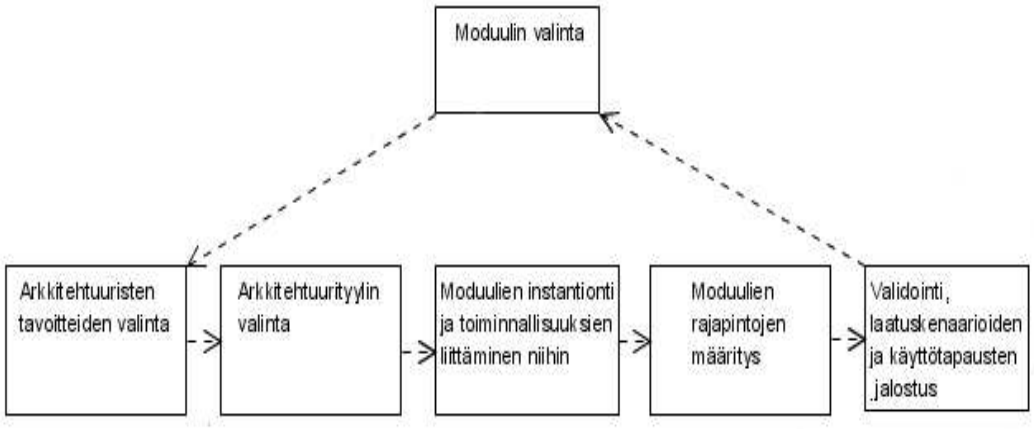
(esimerkiksi turvallisuus ja suorituskyky) kannalta. Arkkitehtuuri vaikuttaa vahvasti näiden vaatimusten toteutumiseen, koska ne täyttyvät arkkitehtuureissa eri tavalla. Nopeaan tietoliikennesovellukseen soveltuva arkkitehtuuri ei välttämättä täytä paperikoneen ohjausjärjestelmän laatuvaatimuksia. Täytyy kuitenkin muistaa, että arkkitehtuuri toimii vain perustana laatuvaatimuksille ja mahdollistaa niiden täyttymisen. Täyttyminen on aina useamman kokonaisuuden summa.

4.4 Arkkitehtuurin määrittäminen

ADD (*Attribute-Driven Design*) [6, sivu 155], [12, sivu 745] on toiminnalliset ja laatuvaatimukset huomioonottava arkkitehtuurien suunnittelumetodi. ADD on rekursiivinen prosessi, jossa järjestelmä jaetaan elementteihin laatuvaatimusten toteutuminen huomioiden. ADD voidaan käynnistää, kun järjestelmän arkkitehtuuriset tavoitteet ovat selvinneet vaatimuksia analysoimalla. ADD jakaantuu kahteen vaiheeseen: moduulin valinta ja sen jalostaminen tietyn metodin mukaisesti. Näitä vaiheita toistetaan jokaiselle moduulille rekursiivisesti, kunnes muodostunut arkkitehtuuri täyttää asetetut vaatimukset. Tuloksena ADD tuottaa elementteihin jaetun järjestelmärakenteen. Saavutettua mallia tarkennetaan toteutuksiin liittyvien yksityiskohtien avulla, jotta toteutusvaiheeseen soveltuva arkkitehtuuri saavutettaisiin.

Syötteenään ADD saa toiminnallisia vaatimuksia ja rajoitteita sekä laatuskenaarioita. Laatuksenaariot ovat ADD:n keino ottaa huomioon laatuvaatimusten täyttyminen. Ne ovat tiettyyn laatuattribuuttiin liittyviä vaatimuksia, joissa kuvataan järjestelmän kyseiseen laatuattribuuttin liittyvää toimintoa. ADD:n alkaessa järjestelmä on yksi iso moduuli, joka hajotetaan alijärjestelmiin. Moduulin pilkkominen on viisivaiheinen prosessi. Ensin valitaan arkkitehtuuriset tavoitteet laatuskenaarioiden ja toiminnallisten vaatimusten joukosta. Näihin perustuen päätetään, mitä arkkitehtuurityyliä sovelletaan sekä tunnistetaan lapsimoduulit. Tunnistettuihin moduuleihin liitetään toiminnallisuus käyttötapauksista ja muodostetaan tästä malli. Seuraavaksi moduuleille muodostetaan rajapinnat. Rajapintoihin liittyen määritellään moduulien liitoksille rajoituksia. Lopuksi validoidaan ja jalostetaan käyttötapaukset ja laatuksenaariot uutta kierrosta varten.

4.4.1 Arkkitehtuuristen tavoitteiden valinta Arkkitehtuuriset tavoitteet [6, sivu 157] ovat yhdistelmä toiminnallisia ja laadullisia vaatimuksia, jotka muokkaavat arkkitehtuuria tai muokattavaa moduulia. Arkkitehtuuriset tavoitteet ovat löydettävissä tärkeimpien vaatimusten joukosta. Moduulin pilkkominen perustuu juuri löydettyihin tavoitteisiin. Tavoitteena on, että tehtävä jako tyydyttäisi merkittävimmät vaatimukset. ADD ei siis käsittele kaikkia vaatimuksia tasa-arvoisina vaan vähemmän tärkeät vaatimukset huomioidaan tärkeimpiin liittyvillä rajoitteilla.



Kuva 4.1: ADD:n vaiheet

4.4.2 Arkkitehtuurityylin valinta Jokaista laatuattribuuttia kohden on olemassa suunnittelupäätöksiä ja strategioita, joilla sen saavuttamista voidaan tehostaa. Arkkitehtuurityylit ovat kokoelma tällaisia päätöksiä. Jokainen suunnittelupäätös vaikuttaa kuitenkin useaan laatuattribuuttiin, ja tavoitteena on löytää tasapaino useamman laatuvaatimuksen kesken. Tähän pyritään soveltamalla arkkitehtuurityyliä, joka tukee tehtyjä suunnittelupäätöksiä ja joka soveltuu toivottuihin arkkitehtuurisiin tavoitteisiin [6, sivu 158]. Arkkitehtuurityylin myötä määritellään myös sen käyttämät moduulityypit.

4.4.3 Moduulien instantiointi ja toiminnallisuuksien liittäminen niihin Edellisessä vaiheessa valitun arkkitehtuurityylin määrittelemistä moduulityypeistä pitää luoda instansseja, jotta nykyinen käsiteltävä moduuli jakaantuisi useampaan alimoduuliin [6, sivu 160]. Toiminnallisuus luokitellaan yläkäsitteiden mukaisiin ryhmiin ja jokaisella ryhmällä tulisi olla oma moduulinsa. Tuloksena on käsiteltävän moduulin jakaantuminen alimoduuleihin. Soveltamalla käsiteltävään moduuliin liittyviä käyttötapauksia ja muokkaamalla moduulijakoa tulisi lopulta jokaisen tarkastellun käyttötapauksen olla esitettävissä lapsimoduulien vastuualueiden avulla. Vastuualueiden määrittämisellä selvitetään myös moduulien väliset suhteet. Tässä vaiheessa ei oteta kantaa suhteen yksityiskohtiin vaan todetaan vain niiden olemassaolo ja tuottaja/kuluttaja-roolit. Tämän vaiheen pitäisi selvittää, pystyykö järjestelmä toteuttamaan halutun toiminnallisuuden ja laadun. Tähän ei kuitenkaan muodostettu vastuun jakaminen ole riittävä keino. Tarvitaan myös esimerkiksi ajonaikaista tietoa. Tätä varten ADD:ssä tulisi käyttää arkkitehtuurin kuvaamiseen ainakin yhtä näkymää kolmesta eri pääryhmästä: moduulijako-, rinnakkaisuus- ja sijoitusnäkymistä (engl. *deployment view*).

Rinnakkaisuusnäkymä mallintaa järjestelmän dynaamisia ominaisuuksia kuten samaan aikaan tapahtuvat toiminnot. Monimutkaiset järjestelmät käsittävät usein

säikeitä tai prosesseja. Nämä prosessit mallinnetaan tietovirtojen, tapahtumien ja jaettujen resurssien osalta. Näkymän avulla pyritään havaitsemaan esimerkiksi prosessien aiheuttamat resurssien jakamiseen ja tiedon eheyteen liityvät ongelmat. Näkymän avulla on mahdollista myös löytää uusia vastuualueita ja moduuleita, jotka liitetään moduulinäkymään.

Sijoitusnäkymä liittyy tietoliikenteeseen liittyvien toiminnallisuuksien tarkkailuun ja toimintojen sijoittamiseen prosessoreille tai erikoistuneiden laitteiden käyttöön. Sijoitusnäkymästä havaitaan eri kohteille kasaantuva kuormitus ja niiden välinen liikenne. Näkymän avulla voidaan havaita ruuhkautumiset esimerkiksi tietoliikenteen kulussa ja suunnitella laatuvaatimukset täyttävä jako kuormituksen suhteen. Esimerkiksi turvallisuuden takaamiseksi jostain moduulista täytyy olla useita ilmentymiä. Samoin jollakin moduulilla täytyy olla vastuu tietoliikenteen hallinnoinnista. Nämä tulee dokumentoida moduulinäkymään.

4.4.4 Moduulien rajapintojen määritys Moduulin rajapinta määrittää tarjottavat ja vaadittavat palvelut ja ominaisuudet [6, sivu 163]. Moduulijaon tarkastelu ja analysointi käyttäen muodostettuja näkymiä paljastaa vuorovaikutussuhteita, jotka tulee dokumentoida. Moduulinäkymään talletetaan kuluttaja/tuottaja-suhteet sekä vuorovaikutusmallit, jotka vaativat moduulia käyttämään ja tarjoamaan palveluja. Rinnakkaisnäkymään tallennetaan samanaikaisten tehtävien vuorovaikutus, joka vaatii moduuleilta palvelujen tarjoamista tai käyttöä. Samoin siihen voidaan liittää toimintoihin liittyvä elementtien aktiivisuusaika sekä synkronisointiin ja toimintosarjoihin osallistumiseen liittyvä tieto. Sijoitusnäkymään kuuluvat laitevaatimukset, aikavaatimukset ja tietoliikennevaatimukset. Näistä saatava tieto tulee koota rajapintamääritelmään.

4.4.5 Validointi, käyttötapausten ja laatuskenaarioiden jalostus sekä rajoitteiden määrittely lapsimoduuleille Aikaansaatu moduulijako tulee validoida ja lapsimoduulit tulee valmistaa omaan jakamisprosessiinsa [6, sivu 164]. Tähän vaiheeseen kuuluu toiminnallisten vaatimusten, rajoitteiden ja laatuksenaarioiden käsittely.

Jokaiselle syntyneelle lapsimoduulille tulee määritellä omat käyttötapauksensa. Koska jokaisella moduulilla on oma vastuualueensa, voidaan se muuttaa käyttötapauksiksi. Mahdollista on myös isämoduulin käyttötapausten pilkkominen ja muokkaus vastaamaan uutta moduulirakennetta. Esimerkiksi auton ovien avaaminen voitaisiin jakaa takaovien ja etuovien avaamiseen. Huomionarvoista on, että jälkimmäisessä menetelmässä säilytetään käyttötapausten jäljitettävyys.

Isämoduuliin voi liittyä rajoitteita, jotka lapsimoduulien tulee ottaa huomioon. Tämän toteuttamiseen on useampia tapoja. Moduulijako toteuttaa rajoitteen ehdot, jolloin muutoksia ei tarvitse tehdä. Rajoite voidaan myös toteuttaa lapsimoduuli-

na, jolloin se kapseloidaan lapsimoduulin sisään, ja vastuu rajoitteen toteutumisesta siirretään eteenpäin lapsimoduulin pilkkomiseen. Mahdollista on myös, että useampi lapsimoduuli toteuttaa rajoitteen, jolloin toteutuminen riippuu lapsimoduulien pilkkomisesta ja niiden lasten toiminnan tarkastelusta.

Myös laatuskenaariot vaativat jalostusta, kun ne siirretään lapsimoduuleille. Laatuskenaario voi täyttyä täysin muodostetun moduulijaon perusteella, jolloin se voidaan merkitä täytetyksi. Laatuskenaario voi täyttyä saavutetun moduulijaon ja annettujen rajoitteiden avulla. Tällöin rajoitus määrää myös lapsimoduuleja. On mahdollista, että moduulijako ei vaikuta lainkaan laatuskenaarion. Tällöin laatuskenaarion toteutuminen tulisi asettaa yhden lapsimoduulin vastuulle. Jos laatuskenaario ei täyty nykyisellä moduulijaolla, tulee harkita jaon muokkaamista vastaamaan vaatimuksia.

Näiden tarkastelujen tuloksena saavutetaan moduulijako, jossa jokaisella moduulilla on oma vastuualueensa, siihen liittyvät käyttötapaukset, rajapinta, laatuskenaariot ja joukko rajoitteita. Kaikki on valmiina uutta iteraatiota varten.

4.5 Tuotelinjan vaikutus arkkitehtuuriin

Arkkitehtuuri ilmaisee sen mitä järjestelmässä on pysyvää eli sen rakenteen. Muutoksia järjestelmän toiminnallisuuteen on mahdollista tehdä muokkaamalla ohjelmakoodia. Tuotelinja-arkkitehtuurin tulee ottaa kuitenkin huomioon jo arkkitehtuuritasolla sallitut muutokset. Suurienkin muutosten toteutuminen tuotetta toteutettaessa on mahdollista ja tavallista. Kazman [33, sivu 14] määrittelee tuotelinja-arkkitehtuurien suurimmiksi haasteiksi laatuattribuuttien toteutumisen eri instansseissa ja ja toiminnallisten muutosten hallinnan määritellyn arkkitehtuurin avulla. Näiden vaatimusten takia on arkkitehtuurin pystyttävä määrittelemään ja eristämään muuttuvat osiot itsessään ja näin rajoittamaan muutoksien vaikutus. Näiden variaatiopisteiden tunnistaminen ja tukeminen ovat tuotelinja-arkkitehtuurin tärkeimpiä tehtäviä [6, sivu 360].

4.5.1 Variaatiopisteiden tunnistaminen Variaatiopisteiden tunnistaminen on jatkuva toiminto, jota voi tapahtua koko kehitysprosessin ajan aina vaatimusmäärittelystä toteutukseen. Variaatiot liittyvät esimerkiksi käyttöliittymiin ja laatuvaatimuksiin, jotka voivat olla tuotelinjassa tuotekohtaisia. Variaatiopisteiden tarkoitus on määrittää arkkitehtuurissa muutospisteet, joissa tuotekohtaiset erot arkkitehtuuritasolla toteutetaan.

4.5.2 Variaatiopisteiden tukeminen Arkkitehtuuri voi tukea variaatiopisteitä eri tavoin. Yleinen tapaus variaatiopisteestä on elementtiliitos, jolla määritellään elementin ominaisuudet tuotekohtaiseksi. Tätä voidaan tukea ajonaikaisesti tai staat-

tisesti riippuen vaatimuksista. Ajonaikainen tehokkuus voi kärsiä joustavuuden lisääntyessä ja päinvastoin. Arkkitehtuuri voi myös mahdollistaa useamman kuin yhden elementin liittämisen variaatiopisteessä. Tällöin puhutaan replikoinnista. Esimerkiksi suorituskykyä voidaan mahdollisesti nostaa käyttämällä useampaa suorituselementtiä. Elementtien poistaminen ja mukaanottaminen ovat keinoja, joilla määritellään tuotekohtaisia ominaisuuksia eriävien ominaisuuksien sijaan.

4.6 Yhteenveto

Arkkitehtuurisuunnittelulla on erityistä merkitystä laatuvaatimusten toteutumisen kannalta. Tuotelinjojen ollessa kyseessä elementtipohjainen ajattelutapa ja ymmärrys järjestelmästä ovat myös keskeisiä. Vaikka suunnittelu voi olla prosessina vaativa ja raskas, mahdolliset hyödytkin ovat tavoittelemisen arvoisia. Tässä luvussa esiteltyä ADD-suunnittelumetodia ei suoraan käytetä määriteltäessä rannekellotuotelinjan arkkitehtuuria luvussa 8. ADD on kattava metodi, mutta täysmittaisesti sovellettuna tarpeettoman raskas rannekellotuotelinjan tapauksessa. ADD:tä kuitenkin käytetään ohjeena ja muun muassa arkkitehtuurimallien kohdalla noudatetaan ADD:n suosituksia.

5 Arkkitehtuurin arviointi

Aikaisemmin on jo todettu arkkitehtuurin merkitys tuotelinjoille. Arkkitehtuurisuunnittelun tulokset on pystyttävä jollain tasolla varmentamaan. Tuotelinjalle suunniteltu arkkitehtuuri onkin järkevää analysoida, jotta nähdään täyttääkö se sille asetetut tavoitteet. Arviointi on järkevää suorittaa mahdollisimman aikaisessa vaiheessa, jolloin sen vaikutukset on helpompaa ottaa huomioon. Toisaalta on myös mahdollista järjestää suunnittelemattomia arviointeja hätätilanteissa, joissa pyritään radikaaleihin ratkaisuihin. Järjestelmällisellä arkkitehtuurin arvioinnilla tavoitellaan hyötyjä, joita ei välttämättä muuten saavutettasi. Arkkitehtuurin arviointiin on olemassa erilaisia menetelmiä. Yksinkertaisimmillaan kyseessä on arkkitehtuurimäärityksen katselmointi mutta käytössä on myös järeämpia arviointiprosesseja. Menetelmät vaihtelevat kyselylomakepohjaisista prosesseista järjestelmällisiin monivaiheisiin prosesseihin. Arviointimenetelmät voidaan Jan Boschin mukaan luokitella skenaario-, simulaatio-, matemaattis- ja kokemusperustaisiin menetelmiin [11, sivu 91]. Menetelmää valittaessa on pohdittava, miten paljon resursseja ollaan valmiita laittamaan arviointiin, ja mikä menetelmä on sopiva. Tässä luvussa tutustutaan arkkitehtuurin arvioinnin tavoitteisiin ja skenaarioita käyttävään menetelmään nimeltä *ATAM*.

5.1 Arvioinnin tavoitteet

Bass et al. [6, sivu 263] listaavat kuusi arkkitehtuurin arvioinnilla tavoiteltavaa hyötyä: taloudellinen hyöty, valmistautuminen, perustelujen keräys, arkkitehtuurin ongelmakohtien selvitys, vaatimusten validointi ja parantunut arkkitehtuuri.

Taloudelliset hyödyt seuraavat ylläpitokustannusten ja projektikulujen vähentymisellä. Arviointi itsessään aiheuttaa runsaasti kustannuksia. Arvioinnin toivotaan ja uskotaan kuitenkin tuottavan enemmän hyötyjä kuin kuluja.

Selkiytyneet tavoitteet ja niiden tehokas saavuttaminen edistävät projektin kulkua ja lopputuotteen laatua. Pakollinen valmistautuminen arviointiin vaatii arkkitehdeiltä kattavan arkkitehtuurimallin ja sen esityksen. Aikaisempi dokumentaatio on mahdollisesti liian suppea tai liian laaja.

Selkeä dokumentaatio seuraa implisiittisesti valmistautumisesta. Arkkitehdeille esitettävistä keskeisille aihealueille osuvista kysymyksistä seuraa suunnittelupäätösten keräys. Nämä kysymykset ja vastaukset johtavat suunnittelupäätösten perusteluun ja dokumentointiin. Olemassaolevan arkkitehtuurin ongelmakohdat paljastuvat arvioinnin yhteydessä.

Tunnettua on, että mitä aikaisemmin ongelmat havaitaan, sitä edullisempaa niiden korjaus on. Arkkitehtuurin arvioinnin yhteydessä paljastuvat ongelmat liittyvät yleensä kannattamattomiin vaatimuksiin ja suorituskykyyn. Arvioinnin yhteydessä selvitetään sen rajat ja mahdollisuudet.

Havaituista ongelmista ja niiden käsittelystä seuraa parantunut arkkitehtuuri. Myös prosessista kertyvä kokemus auttaa ennustamaan ongelmakohtia ja vaadittavaa dokumentaatiota, mikä taas aiheuttaa suunnittelun tehostumista. Näin arvionti johtaa arkkitehtuurien kehittymiseen jo ennen varsinaista arviointia.

5.2 Esiehdot

Jotta arvioinnit on järkevää suorittaa, täytyy sille asetettujen esiehtojen olla täytettyinä. Seuraavat esiehdot voidaan määrittää arviointiprosessille [6, sivu 266]: selkeästi määritellyt tavoitteet ja vaatimukset arkkitehtuurille, hallittu rajaus (engl. *scope*), kustannustehokkuus, henkilöstön saatavuus, pätevä arviointiryhmä ja järkevät odotukset.

Selkeät tavoitteet ja vaatimukset arkkitehtuurille ovat tärkein esiehto arvioinnin suorittamiselle. Arkkitehtuurin avulla pyritään täyttämään tiettyjä laatuvaatimuksia. Jos nämä eivät ole määriteltyjä, on vaikeaa arvioida arkkitehtuurin soveltuvuutta laatuvaatimuksille.

Hallittu rajaus merkitsee arvioinnin kohdistamista oleellisiin asioihin. Tärkeää on pystyä määrittelemään kolmesta viiteen tavoitetta, jotka arvioinnilla pyritään kattamaan. Näin taataan hyödyllisten tulosten syntyminen.

Kustannustehokkuus on myös tärkeä tekijä arviointiin lähdettäessä. Hyötyjen on oltava suurempia kuin kustannusten, jotta arviointi on järkevää. Pienille projekteille ei ole järkevää suorittaa viikkoja kestävää arviointiprosessia. Arvioinnin laajuutta voidaan myös supistaa, ja minimaalinen arvionti on arkkitehtuurimäärityksen katselmointi.

Arviointihenkilöstön saatavuus on oleellisen tärkeää. Henkilön, joka tuntee arkkitehtuurin, on osallistuttava arviointiin, jotta paikalla on suunnittelupäätöksistä tietoinen henkilö. Suuremmissa järjestelmissä jokaisen elementin kohdalla on sama vaatimus. Onkin järkevää sopia arvioinnista alustavasti jo aikaisemmassa vaiheessa, jotta siihen osataan varautua.

Pätevä arviointiryhmä auttaa saavuttamaan hyödyllisiä tuloksia. Parhaassa tapauksessa yrityksellä on käytössään erillinen arviointiryhmä, joka omaa kokemusta ja rutiinia arviointiprosesseista. Vakuuttavan ryhmän tuoma uskottavuus arvioinnille ja sen hyödyille motivoi kaikkia arviointiin osallistuvia. Arviointiryhmän jäsenten tulee hallita ohjelmistoarkkitehtuurit ja sillä tulee olla kokemusta niiden soveltamisesta.

Järkevät odotukset arviointiprosessilta ovat ehdottoman tärkeitä. On tiedettävä

mitkä ovat arvioinnin tavoitteet, mille alueille se keskittyy ja mitä siitä syntyy tuloksena. Myös sovittu aikataulu on tärkeä esiehto arviointiprosessille.

5.3 Arviointimenetelmätyypit

Arviointimenetelmät voidaan luokitella Boschin mukaan skenaario-, simulaatio-, matemaattis- ja kokemusperustaisiin menetelmiin. Tämä ei kuitenkaan ole ainoa mahdollinen jakoperuste. Menetelmätyypeillä on omat etunsa ja haittansa. Toiset soveltuvat paremmin tietyille sovellusaluille ja toiset taas ovat kustannuksiltaan edullisempia. Myös yrityksessä vallitseva käytäntö ohjaa menetelmien valintaa.

5.3.1 Skenaarioperustaiset menetelmät Skenaarioperustaisissa analyysimenetelmissä laatuvaatimusten täyttymistä arvioidaan laadittujen skenaarioiden avulla [11, sivu 92]. Menetelmien tehokkuus on sidoksissa skenaarioiden kattavuuteen ja tarkkuuteen. Mitä kuvaavampia ja tarkempia skenaariot ovat, sen tarkempi tulos arvioinnilla saavutetaan. Skenaarioperustaiset menetelmät soveltuvat hyvin kahden arkkitehtuurin vertailuun. Skenaarioperustaiset menetelmät voidaan jakaa karkeasti kahteen vaiheeseen: vaikutusanalyysiin ja laatuattribuuttitarkasteluun. Vaikutusanalyysissa arkkitehtuurin vaikutusta skenaarioihin tutkitaan. Laatuattribuuttitarkastelussa tutkitaan ensimmäisessä vaiheessa kertynyttä tietoa ja sen vaikutusta tutkittavaan laatuattribuuttiin.

5.3.2 Simulaatioperustaiset menetelmät Simulaatioon perustuvissa menetelmissä [11, sivu 96] käytetään korkean tason toteutusta arkkitehtuurista. Sovelluksen ajoympäristöä pyritään mallintamaan, jotta sovellusta voidaan ajaa arviointimielessä. Koeajoja suoritetaan ja kerättyä tietoa analysoidaan. Simulaatioperustainen menetelmä edellyttää, että ajettava toteutus on olemassa, ja että ajoympäristöä on mahdollista mallintaa. Simulaatioperustaiset menetelmät voidaan jakaa useaan askeleeseen: ympäristön toteutus, elementtien toteutus, profiilin toteutus, simulointi ja laatuattribuutin tarkastelu.

Ympäristön toteutuksessa sovelluksen rajapinnat ulkomaailmaan tunnistetaan ja päätetään, miten niitä simuloidaan. On tärkeää valita oikea abstraktiotaso simuloinnin kannalta. Ympäristön toteutusta ei kannata viedä liian pitkälle jo pelkästään kustannussyistä. Kun simulaatioympäristö on valmis, arkkitehtuurin elementit toteutetaan simulointia varten. Ainakin elementtien rajapinnat ja niiden väliset yhteydet tulisi toteuttaa simulointia varten. Elementtien toteutusaste määräytyy arvioitavien laatuattribuuttien mukaisesti. Myös simulointiin liittyvät profiilit tulee toteuttaa. Simulaation yhteydessä tulee voida ajaa sekä yleisiä että profiilikohtaisia tapauksia. Seuraava vaihe on itse simulointi. Halutut testit suoritetaan ja kertyneet tiedot tallennetaan analyysia varten. Kerätyn tiedon avulla arvioidaan laatuattri-

buuttien täyttymistä.

5.3.3 Matemaattisperustaiset menetelmät Matematiikkaan perustuvat menetelmät [11, sivu 100] pohjautuvat matemaattisten mallien käyttöön. Toisin kuin simulaatioperustaiset menetelmät, matemaattiset arviointimenetelmät sopivat staattiseen arviointiin ja ne eivät vaadi pitkälle edennyttä toteutusta käyttöä varten. Matemaattiset menetelmät soveltuvat erityisesti toiminnallisten laatuvaatimusten tarkasteluun. Matemaattiset menetelmät jaetaan neljään askeleeseen: matemaattisen mallin valinta, arkkitehtuurin esitys mallin avulla, tarvittavan syöttötiedon arviointi ja laatuattribuutin tarkastelu.

Menetelmät alkavat valmiin matemaattisen mallin valinnalla, joka soveltuu tarkasteltavan laatuvaatimuksen arviointiin. Mallia täytyy usein muokata sopivalle abstraktiotasolle. Seuraavaksi arvioitava arkkitehtuuri esitetään mallin avulla. Mallin tarvitsema syöttötieto täytyy ensin muodostaa vaatimusmäärittelystä. Lopulta syöttiedon, arkkitehtuuriesityksen ja mallin avulla lasketaan miten laatuattribuutti täyttyy arkkitehtuurissa.

5.3.4 Kokemusperustaiset menetelmät Kokemusperustaiset menetelmät [11, sivu 103] perustuvat käytännön kautta kertyneeseen kokemukseen arkkitehtuureista ja suunnittelupäätöksistä. Kokemusperustaiset menetelmät voivat olla projektin omien ohjelmistoarkkitehtien suorittamaa arviointia tai ulkopuolisten arviointiryhmien suorittamaa arviointia. Vaikka kokemuksen tuoma osaaminen on arvokasta, tulisi arvioinnin tapahtua jotain muuta validoidumpaa arviointimenetelmää käyttäen.

5.4 ATAM

ATAM (*Architecture Tradeoff Analysis Method*) [6, sivu 271], [35, sivu 54], [37] on suurille ja keskikokoisille projekteille sopiva skenaarioita käyttävä arviointimenetelmä. ATAM paljastaa miten hyvin arkkitehtuuri täyttää sille asetetut laatuvaatimukset, laatuvaatimusten väliset suhteet ja riskit laatuvaatimusten täyttymiselle. ATAMissa keskitytään arvioimaan arkkitehtuurin ja järjestelmän keskeisiä osia. ATAM on pitkäkestoinen ja useaa osanottajaa vaativa prosessi.

ATAM jakaantuu neljään vaiheeseen: kumppanuus ja valmistautuminen, ensimmäinen arviointivaihe, toinen arviointivaihe ja jälkivaihe [6, sivu 276]. Eri vaiheet jakautuvat askeliin [35, sivu 55], joita pitkin edetään vaiheen sisällä ja vaiheesta toiseen. Ensimmäinen vaihe on valmistautumisvaihe. Toisessa ja kolmannessa vaiheessa tapahtuu varsinainen arviointi. Neljännessä vaiheessa kerätään arvioinnin tulokset yhteen.

Vaihe	Toiminto	Osalliset	Tyypillinen kesto
0	Kumppanuus ja valmistautuminen	Arviointiryhmän johto ja tärkeät päätöksentekijät	Tarpeen mukaan muutaman viikon verran.
1	Arviointi 1	Arviointiryhmä ja päätöksentekijät	Kahdesta kolmeen viikkoa.
2	Arviointi 2	Arviointiryhmä, päätöksentekijät ja arkkitehtuuriosakkaat	Kaksi päivää.
3	Jälkivaihe	Arviointiryhmä ja arvioinnin kohde	Yksi viikko.

Taulukko 5.1: ATAMin vaiheet

5.4.1 Osanottajat ATAMiin osallistuvat ihmiset jaetaan kolmeen ryhmään: arviointiryhmä, päätöksentekijät ja arkkitehtuuriosakkaat (engl. *architecture stakeholders*) [6, sivu 272].

Arviointiryhmä koostuu ihmisistä, jotka eivät liity arvioitavaan arkkitehtuuriin tai kyseiseen projektiin. Se koostuu kolmesta viiteen ihmisestä, joista kullakin on useita rooleja arvioinnissa. Kyseessä voi olla pysyvä ryhmä tai varta vasten koottu kokoonpano arkkitehtuuriasiantuntijoita. Jäsenet voivat olla yrityksen työntekijöitä tai ulkopuolisia konsultteja.

Rooli	Vastuualue	Kaivatut ominaisuudet
Ryhmänjohtaja	Järjestää evaluoinnin, on yhteydessä asiakkaaseen, laatii arviointisopimuksen ja varmistaa, että loppuraportti laaditaan ja toimitetaan perille.	Organisointikykyinen, hallinnollisesti kyvykäs, hyvä suhde asiakkaaseen, pysyy aikataulussa.
Arviointijohtaja	Suorittaa arvioinnin, edesauttaa skenaarioiden laatimista, johtaa skenaarioiden valintaa ja priorisointia, hoitaa skenaarioiden ja arkkitehtuurin vertaamisen sekä hallitsee keskustelua niistä.	Esiintymiskykyinen, aikaansaapa ja kannustava, ymmärtää arkkitehtuureja, osaa johtaa puhetta oikeaan suuntaan.
Skenaariokirjuri	Kirjaa skenaariot ylös taululle niitä luotaessa, kirjaa hyväksynnät ja johtaa puhetta.	Selkeä käsiala, pysyy aiheessa, ymmärtää olennaisen teknisestä keskustelusta.
Tuloskirjuri	Tallentaa sähköiseen muotoon skenaariot, niiden alkumuodot ja niiden arkkitehtuuriarviot. Laatii listan jaettavaksi käytettävistä skenaarioista.	Hyvä konekirjoittaja, nopea tiedonhakija, ymmärtää arkkitehtuureista, sulattaa nopeasti teknistä tietoa, ei pelkää keskeyttää keskustelua, jotta tärkeä tieto tallentuu.
Ajanmittaaja	Auttaa arviointijohtajaa pysymään aikataulussa ja auttaa kontrolloimaan kullekin skenaariolle omistettua aikaa.	Valmis keskeyttämään keskustelun varatun ajan loppuessa.
Prosessitarkkailija	Pitää kirjaa prosessin kehittämistarpeista, voi tehdä ehdotuksia ryhmänjohtajalle arvioinnin aikana ja raportoi arvioinnin kulusta ja opituista asioista.	Tarkkailijaluonne, tuntee arviointiprosessin hyvin, omaa aikaisempaa kokemusta prosessista.
Prosessivalvoja	Auttaa arviointijohtajaa muistamaan ja suorittamaan käytetyn metodin kaikki vaiheet.	Hallitsee sujuvasti menetelmän vaiheet ja on kyvykäs tarjoamaan hienovaraista ohjausta arviointijohtajalle.
Kyselijä	Herättää arkkitehtuuriin liittyen kysymyksiä aiheista, joita ei välttämättä ole havaittu.	Hyvä käsitys arkkitehtuurista, hyvä käsitys arkkitehtuuriosakkaiden tarpeista, kokemusta samankaltaisista järjestelmistä, ei pelkää tuoda aihetta esiin ja jatkaa sitä, perillä mahdollisista huolenaiheista.

Taulukko 5.2: Arviointiryhmän jäsenten roolit

Päätöksentekijät ovat henkilöitä, joilla on valta tehdä päätöksiä projektin ja muutosten suhteen. Tähän ryhmään kuuluvat yleensä ainakin projektipäällikkö ja asiak-

kaan edustaja. Myös arkkitehti on päätöksentekoryhmässä.

Arkkitehtuuriosakkaat ovat sekalainen ryhmä arvioitavaan arkkitehtuuriin sidoksissa olevia henkilöitä. Heidän työnsä on jollain lailla riippuvainen arkkitehtuurista. Tähän ryhmään kuuluu esimerkiksi testaajia, integroijia, loppukäyttäjiä, ylläpitäjiä ja kehittäjiä. Ryhmän tehtävä on tuoda esiin arkkitehtuurin ongelmakohtia ja vahvuuksia omaan alueeseensa liittyen. Tämän ryhmän suuruus on noin 10-15 henkeä.

5.4.2 Kumppanuus ja valmistautuminen Ensimmäisessä vaiheessa [6, sivu 275] arviointiryhmän johto ja merkittävät päätöksentekijät tapaavat epävirallisesti keskustellakseen arvioinnin yksityiskohdista. Projektin edustajat esittelevät lyhyesti projektin tavoitteet, jotta ryhmään osataan koota tarvittavat henkilöt. Osapuolet sopivat aikataulusta ja paikasta sekä muista yksityiskohdista. Alustava lista arkkitehtuuriosakkaista laaditaan. Loppuraporttiin liittyvistä asioista sovitaan. Arkkitehdille selvitetään, mitä hänen esitykseltään odotetaan arvioinnissa. Kyseessä on lähinnä asioista sopiminen ja osapuolten tutustuminen toisiinsa.

5.4.3 Ensimäinen arviointivaihe Ensimmäinen arviointivaihe [6, sivu 276] jakaantuu kuuteen askeleeseen. Ensimmäisessä askeleessa arviointijohtaja esittelee ATAMin projektin edustajille. ATAM ja sen tulokset selvitetään, ja aiheesta esitettyihin kysymyksiin vastataan.

Toinen askel liittyy liiketoiminnallisten tavoitteiden selvittämiseen kaikille osanottajille. Kaikkien täytyy ymmärtää järjestelmäkonteksti ja liikentoiminnalliset motivaattorit kehityksessä. Projektin vetäjä tai asiakkaan edustaja pitää yleiskatsauksen kehitettävään järjestelmään liiketoiminnallisesta näkökulmasta. Esityksessä kuvataan järjestelmän tärkeimmät toiminnot, merkittävät rajoitteet, liiketoiminnalliset tavoitteet, pääosakkaat ja arkkitehtuuriset tavoitteet.

Kolmannessa askeleessa esitellään arkkitehtuuri. Johtava arkkitehti tai arkkitehtiryhmä pitää esitelmän järjestelmän arkkitehtuurista. Esityksen tarkkuustaso voi vaihdella ajasta ja tarpeesta riippuen. Esitelmään sisällytetään ainakin tekniset rajoitteet ja käytetyt arkkitehtuurimallit. Tärkeää on myös selvittää, miten laatuvaatimukset täyttyvät. Esityksen tulee keskittyä oleellisiin osiin arkkitehtuurissa. Erilaisia malleja esittämällä ja etukäteen valmistautumalla esityksestä saadaan ytimekäs. Arviointiryhmä tekee tässä vaiheessa kysymyksiä täydentääkseen tietojaan. Ryhmä kirjaa myös käytetyt arkkitehtuurityylit ylös.

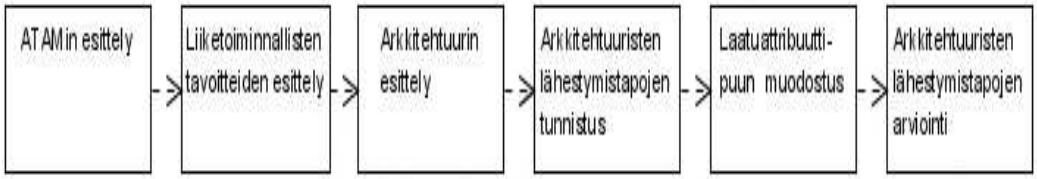
Neljännessä vaiheessa tunnistetaan arkkitehtuuriset lähestymistavat. ATAMissa keskitytään analysoimaan arkkitehtuuria ymmärtämällä sen arkkitehtuuriset lähestymistavat [36, sivu 87]. Tässä vaiheessa erityisesti tunnistetuista arkkitehtuurityyleistä on hyötyä. Arviointiryhmä on tähän mennessä tunnistanut ja selvittänyt arkkitehtuurin hyödyntämät arkkitehtuurityylit. Kuultuaan arkkitehdin esityksen

ryhmä tutustuu dokumentaatioon. Tässä vaiheessa luetteloidaan käytetyt tyylit ja lähestymistavat. Tämä luettelo on julkinen ja hyödyksi tulevissa vaiheissa.

Viidennessä vaiheessa muodostetaan laatuattribuuttipuu (engl. *quality attribute utility tree*). Tärkeät arkkitehtuuriset tavoitteet on nimetty askeleessa kaksi. Tässä vaiheessa laatuvaatimuksia jalostetaan käyttäen laatuattribuuttipuuta. Arviointiryhmä työskentelee päätöksentekijöiden kanssa tunnistaakseen, järjestääkseen ja jalostaakseen järjestelmän tärkeimmät laadulliset tavoitteet. Apuna vaatimusten esittämisessä käytetään skenaarioita. Laatuattribuuttipuu konkretisoi asetetut laatuvaatimukset ja pakottaa määrittämään tarkasti asetettavat vaatimukset. Laatuattribuuttipuun juurena on järjestelmän yleinen laadukkuus (engl. *utility*). Toisella tasolla ovat askeleessa kaksi määritellyt laatuvaatimukset. Yleisiä laatuvaatimuksia ovat suorituskyky [59, sivu 1], muokattavuus, turvallisuus, käytettävyys ja saatavuus. Jokaisen toisen tason laadun alle tulee jalostuneempia vaatimuksia. Lehtinä laatuattribuuttipuussa ovat laatuskenaariot [6, sivu 78]. ATAMissa skenaario koostuu stimuluksesta (mihin elementtiin liittyy, kuka loi), ympäristöstä (missä tilassa järjestelmä on) ja reaktiosta (engl. *response*) (järjestelmän reaktio skenaarioon). Skenaarioita voi kertyä runsaasti, joten ne täytyy asettaa tärkeysjärjestykseen. Päätöksentekijät asettavat kullekin skenaariolle prioriteetin asteikolla 1-10 tai korkea/keskiverto/matala. Arkkitehti arvioi jokaisen skenaarion toiseen kertaan valittua asteikkoa käyttäen. Arvioinnin jälkeen jokaisella skenaariolla on prioriteettiarvopari (X,X). Tuloksena vaiheesta viisi saadaan laatuattribuuttipuu ja järjestetyt skenaariot. Saavutetun järjestyksen perusteella ratkaistaan mihin skenaarioihin keskitytään arviointiprosessissa. Näin pyritään tärkeimpien laatuvaatimusten täyttymiseen.

Kuudes askel on arkkitehtuuristen tavoitteiden analysointi. Arviointiryhmä tutkii tärkeimpiä skenaarioita yksi kerrallaan. Arkkitehti selittää skenaariokohtaisesti, miten arkkitehtuuri tukee skenaariota. Ryhmän jäsenet tutkivat skenaarioon liittyviä arkkitehdin näkökulmia. Oleelliset arkkitehtuuriset päätökset kirjataan ylös. Niihin liittyvät riskit, epäriskit, kustannukset ja herkkyyspisteet [34, sivu 298] tunnistetaan ja dokumentoidaan. Havaittuja heikkouksia tutkitaan ja laatuvaatimusten täyttymistä pohditaan. Arviointiryhmän tulee vakuuttua, että valittu lähestymistapa on oikea laatuvaatimusten kannalta. Riskien, epäriskien ja herkkyyspisteiden punnitseminen johtaa arkkitehdin vastauksista riippuen syvempään analyysiin. Epäselvyyksiä yritetään selvittää. Tarkoituksena on yhdistää tehdyt suunnittelupäätökset laatuvaatimuksiin. Epäriskit, riskit, herkkyyspisteet ja kustannukset kerätään erillisiksi listoiksi. Askeleen lopussa arviointiryhmällä tulee olla selkeä kuva tärkeimmistä kohdista arkkitehtuurissa, käsitys tärkeimmistä suunnittelupäätöksistä sekä lista riskeistä, epäriskeistä, herkkyyspisteistä ja kustannuksista.

5.4.4 Toinen arviointivaihe Ensimmäisen arviointivaiheen loputtua arviointiryhmä vetäytyy pohtimaan hankittua tietoa ja keskustelee asioista arkkitehdin kanssa.



Kuva 5.1: Ensimmäisen arviointivaiheen askeleet

Tämä seisomavaihe kestää viikosta kahteen. Tarpeen vaatiessa analysoidaan lisää skenaarioita ja selvitetään heränneitä kysymyksiä. Kun päätöksentekijät ovat valmiita jatkamaan, siirrytään jälkimmäiseen arviointivaiheeseen [6, sivu 284]. Saavutetut tulokset esitetään kaikille osallistujille, jotta kaikki ovat ajan tasalla.

Vaihe seitsemän sisältää aivoriihen ja skenaarioiden priorisointia. Laatuattribuuttipuu selvensi, miten arkkitehti käsitti ja käsitteli laatuattribuutteja. Skenaarioaivoriihen tarkoituksena on selvittää laajemman kohderyhmän mielipiteet. Tilaisuuteen soveltuu isohko ryhmä, jonka kesken vallitsee vapaa ilmapiiri. Tarkoituksena on herättää uusia ajatuksia, jotka synnyttävät jälleen uusia ideoita. Laadittavien skenaarioiden tulee olla tarkoituksenmukaisia laatijoiden rooliin nähden. Myös analysoimattomia skenaarioita saa hyödyntää. Samankaltaiset skenaariot yhdistetään ja jäljelle jääneet priorisoidaan, jotta havaitaan tärkeimmät skenaariot. Jokaisella osallistujalla on äänimäärä, jonka hän saa jakaa haluamallaan tavalla skenaarioiden kesken. Äänestystilanne on julkinen. Tuloksena syntyvää järjestettyä skenaariolistaa verrataan laatuattribuuttipuun skenaarioihin. Jos näiden kesken vallitsee yhteisymmärrys, vastaa arkkitehtuuri hyvin tavoitteita. Eriävät skenaariot paljastavat ristiriitoja arkkitehdin ja arkkitehtuuriosakkaiden näkökulmissa. Järjestyksen perusteella valitaan haluttu määrä skenaarioita käsittelyyn.

Arviointiryhmä ottaa valitut skenaariot käsittelyyn. Arkkitehdille esitetään jokainen skenaario yksitellen. Arkkitehdin tehtävänä on selvittää arviointiryhmälle oleelliset arkkitehtuuriset päätökset ja niiden vaikutukset skenaarioihin liittyen. Kyseessä on kuudennen askeleen kaltainen tilanne.

5.4.5 Jälkivaihe Jälkivaiheeseen [6, sivu 285] liittyy yhdeksäs askel: tulosten esittäminen. Koottu tieto pitää koota ja esittää arkkitehtuuriosakkaille. Esitystä varten voidaan koota kalvoesitys ja tueksi voidaan laatia loppuraportti, joka sisältää ATAMin tulokset. ATAMin tuloksena [6, sivu 274], [34, sivu 298] saadaan:

- *Ytimekäs arkkitehtuurin esitys*. ATAM vaatii ytimekkään arkkitehtuurin esityksen, joka usein joudutaan sitä varten laatimaan.
- *Selkeät liiketoiminnalliset tavoitteet*. Liiketoiminnalliset tavoitteet lausutaan julki ja esitetään kaikille osanottajille.

- *Laatuvaatimukset skenaarioiden avulla esitettyinä*. Tärkeimmistä liiketoiminnallisista tavoitteista laaditaan skenaariot.
- *Arkkitehtuuriset päätökset liitettyinä laatuvaatimuksiin*. Jokaiseen käsiteltyyn skenaarioon liittyvät suunnittelupäätökset tunnistetaan ja dokumentoidaan.
- *Joukko tunnistettuja herkkyyspisteitä ja kustannuksia*. Arkkitehtuuria analysoitaessa havaitut herkkyyspisteet ja epäkohdat dokumentoidaan ja luokitellaan.
- *Joukko riskejä ja epäriskejä*. Riski on ATAMissa päätös, joka mahdollisesti johtaa ei-haluttuihin tuloksiin. Epäriski on riskitön suunnittelupäätös.
- *Riskiteemat*. Tunnistetuista riskeistä muodostetaan riskiteemoja, jotka kuvaavat järjestelmän heikkouksia.

ATAM on riskien tunnistamiseen ja arkkitehtuurin arviointiin tarkoitettu menetelmä. Se ei kuitenkaan kerro, miten riskejä kannattaa lähteä hallitsemaan ja millaisin kustannuksin tämä tapahtuu. Riskien ja kustannusten hallintaan on olemassa erillinen menetelmä CBAM [6, sivu 307], [34, sivu 299], joka jatkaa suoraan siitä mihin ATAM jäi.

5.5 Yhteenveto

Tuotelinjat asettavat omat vaatimuksensa ohjelmistoarkkitehtuurille, ja menestyksekäs arkkitehtuurisuunnittelu onkin merkittävä askel matkalla kohti onnistunutta tuotelinjaa. Tämän vuoksi arkkitehtuurin arviointi on tärkeä vaihe tuotelinjan kannalta. On olemassa erilaisia ja erityyppisiä arviointimenetelmiä, jotka soveltuvat eri tilanteissa. Esitelty arviointimenetelmä ATAM on laaja ja runsaasti osallistujia vaativa arviointimenetelmä. Sen raskauden ja pitkäkestoisuuden vuoksi sitä on todella vaikeaa soveltaa luvun 8 rannekellotuotelinjan arkkitehtuurille. Tämän vuoksi vaihe on jätetty pois luvusta 8. Jos rannekellotuotelinja oltaisiin toteuttamassa, olisi arkkitehtuurin arviointi kuitenkin järkevää.

6 Elementtien toteutus

Kun arkkitehtuuri ja siihen liittyvät elementit on määritelty, ne on toteutettava. Tuotelinja-arkkitehtuureissa tuotteiden välinen eroavaisuus saavutetaan juuri elementtien kautta. Valinnaisia elementtejä käyttämällä tai poisjättämällä säädellään tuotteiden ominaisuuksia. Ominaisuuksien eroja määritellään elementtejä konfiguroimalla. Ominaisuus voi muodostua useamman eri elementin kombinaationa ja juuri tietyin asetuksin.

Tuotelinja-arkkitehtuurin elementin täytyy täyttää useita vaatimuksia. Tätä varten elementin täytyy tarjota rajapintoja, joiden kautta se on kosketuksissa ulkomaailman kanssa. Rajapinnan takaa löytyy elementin toteutus. Elementit koostuvat perinteisesti ohjelmakoodista. Elementtien toteutus ja rakenne vaikuttavat arkkitehtuurin laatuun. Elementtien toteutuksessa tuleekin huomioida uudelleenkäytettävyys. Näiden vaatimusten täyttämiseksi voidaan elementtien toteutusmenetelminä käyttää esimerkiksi ohjelmointikieliä ja järjestelmägeneraattoreita (esim. GenVoca). Ohjelmointikielissä olioparadigma on varteenotettava lähestymistapa modulaarisen suunnittelun ja uudelleenkäytön kannalta. Tässä luvussa perehdytään elementtien rajapintoihin ja toteutukseen.

6.1 Elementtien rajapinnat

Bosch [11, sivu 221] määrittää elementille kolme rajapintatyyppiä, joiden kautta se on vuorovaikutuksessa ympäristöönsä: konfiguraatiorajapinnat, vaadittavat rajapinnat ja tarjottavat rajapinnat. Konfiguraatiorajapintojen kautta hallitaan elementin muunneltavuutta. Vaadittavat rajapinnat määrittävät rajapinnat, jotka vaaditaan elementtiin liitettäviltä elementeiltä. Tarjottavat rajapinnat kertovat, minkä tyyppiset elementit voivat liittyä elementtiin. Vaadittavat ja tarjottavat elementit voidaan ajatella liittimien instansseiksi arkkitehtuurissa. Niiden avulla voidaan toteuttaa arkkitehtuurityylin määrittämät rajoitukset elementtien liitoksille.

6.1.1 Konfigurointirajapinnat Elementille voidaan määrittää tiettyjä asetuksia, jotka liittyvät sen muunneltavuuteen. Nämä voivat vaikuttaa tietyn toteutuksen valintaan esimerkiksi käytetyn algoritmin osalta. Jos geneerisiä malleja on käytetty, voidaan niiden parametrit asettaa konfigurointitiedoston kautta. Yleisesti voidaan sanoa, että konfigurointirajapinnan kautta hallitaan elementin variaatiopisteitä. Konfigurointirajapinta voi tukea sekä ajonaikaista että staattista konfigurointia.

6.1.2 Tarjottavat rajapinnat Kaikki palvelut, joita elementti tarjoaa, määritellään tarjottavien rajapintojen kautta. Tarjottava rajapinta on liitos, johon tarjoavaa elementtiä käyttävä elementti voi liittyä. Elementti voi toteuttaa useamman kuin yhden tarjottavan rajapinnan. Jotta rajapinta olisi eheä, liittyy siihen rajoituksia. Rajapinta saa käyttää vain itse määrittelemiään tyyppejä ja olioita tai ohjelmointikielen tarjoamia natiivityyppejä [11, sivu 223].

6.1.3 Vaadittavat rajapinnat Jokainen tarjottava rajapinta on sidoksissa yhteen tai useampaan vaadittavaan rajapintaan. Vaadittava rajapinta on elementin keino käyttää tarjoavien rajapintojen palveluja. Se siis vaaditaan näiden palvelujen saamiseksi. Käytännössä vaadittava rajapinta on viite tarjottavaan rajapintaan.

6.2 Ohjelmointikieliin perustuvat variaatiomenetelmät

Ohjelmointikielten tarjoamien menetelmien avulla on mahdollista toteuttaa variaatiopisteitä tukeva elementti. Elementtien toteuttamiseen käytettäviä ohjelmointiparadigmoja ja tekniikoita on monia. Yhtä ainoaa oikeaa tapaa rakentaa elementti ei ole, vaan useampia menetelmiä voidaan hyödyntää erikseen tai rinnakkain [24, sivu 6]. Tarkoituksena on nyt esitellä näitä variaatiopisteiden toteuttamiseen käytettäviä yleisiä mentelmiä.

6.2.1 Perintä Perintä on olio-ohjelmoinnin perusmenetelmiä. Perinnän avulla saadaan käyttöön yliluokan julkaisemat toiminnot aliluokalle. Perintä onkin hyvin tehokas uudelleenkäytön väline. Voidaan ajatella olevan abstrakteja luokkia, joista tarvitaan erikoistuneempia luokkia, jotka voidaan instantioida. Tämä on looginen tapa ajatella erilaisia taksonomioita. Kaikkien luokkien määrittely erikseen on mahdollista mutta aikaa vievää ja tuottaa paljon toistuvaa ohjelmakoodia. Perinnän avulla yhteiset ominaisuudet ja yhteiset metodit määritellään abstraktiin yliluokkaan, jolloin ne ovat kirjoitettuina vain yhteen paikaan. Erikoistuneemmat aliluokat peritään tästä yliluokasta ja tarvittava erikoistuminen toteutetaan näihin luokkiin. Näin muodostuu *is a* -suhde yläluokan ja sen aliluokkien välille. Tämä on klassinen esimerkki perinnän käytöstä. Perintä toimii myös pohjana kehittyneemmille uudelleenkäytön välineille kuten suunnittelumalleille.

Perintä on tyypiltään *lasilaatikko*-uudelleenkäyttöä (engl. *white-box reuse*), mikä aiheuttaa myös ongelmia olio-ohjelmoinnin kannalta. Nimitys lasilaatikko tulee perinnän tavasta paljastaa toteutuksensa aliluokille. Tämä rikkoo olio-ohjelmoinnin kapselointiperiaatetta vastaan [52, sivu 39]. Ylä- ja aliluokat liittyvät tiiviisti toisiinsa, ja muutokset yläluokassa aiheuttavat usein muutoksia aliluokissa [16, sivu 19]. Tämä voi johtaa myös ongelmiin aliluokasta perittäessä. Toinen ongelma perintää käytettäessä liittyy sen staattisuuteen. Perintäsuhteita ei voi muuttaa ajonaikaisesti,

mikä vähentää joustavuutta.

Lisäkeluokat (engl. *mixin class*) ovat moniperintää käyttävä menetelmä, jossa luokka rakennetaan perimällä sitä useasta lisäkeluokasta. Jokainen lisäkeluokka tarjoaa tietyn toiminnallisuuden ja useammasta palasesta muodostuu laajempi kokonaisuus. Tuotelinjojen kannalta lisäkeluokkia voidaan ajatella koodifragmentteina, joista kokoamalla saadaan halutuin ominaisuuksin rakennettu elementti [24, sivu 7].

6.2.2 Koostaminen Koostamisessa uudelleenkäyttö tapahtuu kokoamalla olio toisista olioista. Muodostuvat suhteet ovat *has a* -luonteisia. Koostaminen toteuttaa perintää paremmin kapselointiperiaatetta, koska koostettu luokka näkee käyttämistään luokista vain niiden tarjoaman ulkoisen rajapinnan. Tämän vuoksi oliot pakotetaan kunnioittamaan tiedon kapselointia. Tästä johtuen koostamista kutsutaan myös *mustalaatikko*-uudelleenkäytöksi (engl. *black-box reuse*). Koostamisesta seuraa useita hyötyjä [16, sivu 19].

- Koostaminen johtaa huolelliseen rajapintojen suunnitteluun.
- Koostesuhteet ovat ajonaikaisesti muokattavissa.
- Koostaminen johtaa pienempiin riippuvuussuhteisiin luokkien välillä kuin perintä.
- Koostamisesta muodostuvat luokat ovat paremmin rajautuneita tehtäväänsä, ja muodostuvat luokat ja luokkahierarkiat ovat pienempiä.

6.2.3 Rajapinnat Rajapinnat [11, sivu 221] ovat menetelmä, jonka avulla voidaan määritellä metodijoukko, joka rajapinnan toteuttavien luokkien on toteutettava. Rajapintojen tehtävänä ei ole määrittää aliluokkien yhteisiä ominaisuuksia vaan taata, että toteuttamalla rajapinta toteutetaan myös sen esittelemät metodit. Rajapinnat eivät siis sisällä lainkaan logiikkaa. Rajapinnan kautta voidaan sen toteuttavia luokkia käyttää rajoitetusti vain määritellyn rajapinnan kautta.

6.2.4 Makrot Makrot kuuluvat alkeellisempiin keinoihin toteuttaa variaatiopisteitä. Koodiin määritellään tarkistuskohtia, joissa tutkitaan tietyn makron olemassaoloa. Tämä vaikuttaa siihen, huomioidaanko eristetty lähdekoodi käännettäessä vai ei. Svahnberg ja Bosch [55, sivu 9] määrittelevät kolme käyttötapaa makroille: tuotekohtaisen koodin sisällyttämisen elementtien toteutuksiin, elementin rajapinnan muuttamisen konfiguroinnin mukaan ja elementtien toteutuksien joukon valinnan tuotteeseen. Makrot ovat staattinen, vaikeasti hallittavissa oleva menetelmä etenkin tuoteperheen kasvaessa.

6.2.5 Suunnittelumallit Suunnittelumallit ovat ratkaisuja ohjelmoinnissa usein vastaan tuleviin ongelmiin. Ne voidaan esittää esimerkiksi olio-ohjelmoinnin tarjoamia välineitä, kuten koostamista, perintää ja polymorfismia käyttäen. Vaikka allaoleva määritys koskee rakennuksia ja kaupunkeja, pätee se myös suunnittelumalleihin [1, sivu X].

> Each pattern describes a problem which occurs over and over again in our enviroment, and then describes the core of the solutions to that problem, in such way that you can use this solution a million times over, without ever doing it the same way twice.

Gamma et al. määrittelevät suunnittelumallin koostumaan nimestä, ongelmasta, ratkaisusta ja seurauksista [16, sivu 3]. Nimi on viite tiettyyn malliin. Ongelma kuvailee suunnittelumallin sovellusalaa. Tähän voi liittyä ehtoja ja tilannekuvauksia, jotka liittyvät yleisiin sovelluskohteisiin. Perustaltaan ongelmat ovat kuitenkin yleisiä ja sovellettavissa laajasti. Ratkaisu kuvaa siihen liittyvät elementit ja niiden väliset suhteet. Se kuvataan yleisellä tasolla, jotta soveltuvuus ei kärsisi. Suunnittelumallin käytöstä seuraavat edut ja haitat kuvataan Seuraukset-osassa.

Suunnittelumallit jaetaan kahden kriteerin mukaan joukkoihin [16, sivu 10]. Toiminnan mukaan ne jaetaan luomismalleihin, rakenteellisiin malleihin ja käyttäytymismalleihin. Sovellusalueensa mukaan ne jaetaan luokka- ja oliotasolle. Luomismallit liittyvät olioiden luomisprosesseihin. Rakenteelliset mallit käsittelevät luokkien ja olioiden rakennetta ja koostumusta. Käyttäytymismallit määrittelevät, miten luokat tai oliot keskustelevat ja jakavat vastuuta. Sovellusalue määrittelee, vaikuttaako malli olio- vai luokkatasolla. Luokkamallit koskevat luokkia ja niiden välisiä staattisia suhteita. Oliomallit soveltuvat ajonaikaisiin tilanteisiin ja ovat vaikutukseltaan dynaamisempia.

Suunnittelumallin soveltaminen vaatii ongelman hahmottamisen. Pitää ajatella, minkä haluaa olevan vaihdeltavaa järjestelmässään, ilman että joudutaan suunnittelemaan koko rakenne uudelleen. Kun tämä on ymmärretty, voidaan tutkia suunnittelumalleja ja etsiä niistä sopivia käytettäväksi. Tuotelinja-arkkitehtuureissa ongelmana on variaatiopisteiden toteutus. Toki suunnittelumallien soveltaminen muihinkin toteutusongelmiin on suositeltavaa. Variaatiopisteitä ajatellen on etsittävä suunnittelumalleja, jotka lisäävät joustavuutta ja muunneltavuutta. Esimerkiksi *Strategia* (engl. *Strategy*) [16, sivu 315] mahdollistaa algoritmin ja elementin ongelmattoman erottamisen toisistaan.

6.2.6 Subjektiohjelmointi Perinteisesti luokka määrittää omat attribuuttinsa ja metodinsa. Tämä on kuitenkin melko rajattu näkymä. Asiat voidaan nähdä erilaisina tarkastelijasta ja hänen maailmankuvastaan riippuen. Subjektiohjelmoinnin avulla jokaisen tarkastelijan on mahdollista määrittää oma näkymänsä kohteesta.

Tätä näkökulmaa kutsutaan subjektiksi. Subjekti sisältää tietynlaisen näkymän sovellusalueen luokista, jotka ovat näkymälle tarpeellisia. Koska samasta kohteesta voi olla useampi kuin yksi näkymä, on niiden välillä mahdollista olla yhteistoimintaa. Ne voivat myös jakaa keskenään tilatietoa yhteisestä kohteestaan. Yhdestä tai useammasta subjektista muodostuu sovellus. Sovellus määrittää tilan ja käyttäytymisen, joka on ominaista sovelluksen sisältämille luokille. Harrison ja Ossher [26, sivu 412] asettavat seuraavat tavoitteet subjektiohjelmoinnille:

- Sovellusten erillinen kehittäminen ja niiden kokoaminen jälkeenpäin.
- Erikseen kehiteltyjen sovellusten riippumattomuus toisistaan.
- Sovellusten tulee pystyä toimimaan yhdessä heikosti sidottuina, vahvasti sidottuina tai hajautettuina.
- Uuden sovelluksen lisäys koosteeseen ilman muutoksia vanhoihin sovelluksiin ja pysyviin olioihin pitää olla mahdollista.
- Odottamattomia muutoksia ja laajennuksia pitää pystyä tukemaan.
- Olio-ohjelmoinnin tarjoamat mahdollisuudet tulee säilyttää.

Subjekti määrittää tietynlaisen tilan ja sen käyttäytymisen. Subjektilla itsellään ei ole tilaa. Subjektit eivät ole luokkia, mutta ne sisältävät aina niitä. Subjekti määrittää näkemillensä olioille toteutusluokat. Nämä luokat sisältävät operaatiot ja tilan, joka on oleellista subjektin kannalta. Jokaiseen subjektiin liittyy toteutusluokkien mukaisia olioita. Subjektit voivat määrittää ja käsitellä näitä olioita tietämättä mitään toisten subjektien määrityksistä. Nämä oliot voivat olla myös jaettuja subjektien kesken. On huomioitava, että subjekti voi muokata vain määrittelemiään attribuutteja ja käyttää vain määrittelemiään operaatioita. Subjektit voivat muodostaa kompositioita, jotka toimivat yhdessä. Yhteistoiminta määritellään sääntöjen avulla.

Subjektien yhteistoiminta on kohtuullisen monimutkaista, koska kaikki määrittävät omat operaatiot ja tilat olioille. Kuitenkin oliot ovat tavallaan yhteisiä subjekteille vain eri näkökulmasta nähtynä. Jotta olioiden tunnistaminen on mahdollista, annetaan jokaiselle syntyneelle oliolle tunnus *oid*. Samaa kohdetta ilmentävillä olioilla on siis erilliset tunnukset. Kun subjekti muuttaa olionsa tilaa tai suorittaa operaation, on sillä mahdollisia vaikutuksia muiden subjektien hallitsemien olioiden tiloihin. Suoritettuihin operaatioihin liittyy sääntöjä, jotka koskevat operaatioista ilmoittamista ja ilmoituksen vaikutuksia.

On useita tapoja suorittaa tunnistus. Usein tunnistaminen vaatii useamman menetelmän käyttöä. Yksinkertainen menetelmä luokan tunnistamiselle on määritellä käytettävät nimet etukäteen. Tämä poistaa koko ongelman mutta rajoittaa nimeämistä. Jos luokka täytyy tunnistaa vastaamaan omaa versiota, voidaan lähteä tut-

kimaan yliluokkia. Jos niiden välillä tapahtuu tunnistus, voidaan aliluokkakin tunnistaa. Samoin voidaan tutkia tunnistettavan luokan aliluokkia ja pyrkiä tunnistamiseen tätä kautta. Myös rajapintojen kautta tapahtuva tunnistaminen on mahdollista. Oletuksena on, että kyseessä on jaettu olio.

Elementtien suunnittelussa ja toteutuksessa subjektiohjelmoinnin soveltamisala on ilmeinen. Tuotelinja-arkkitehtuurin elementin on tarkoitus täyttää usean eri tuotteen vaatimukset. Jokainen tuoteperheen jäsen voi määrittää subjektin, joka sisältää sen kannalta oleelliset piirteet elementissä. Näin variaatiovaatimukset tulevat kaikkien tuotteiden osalta tuetuiksi.

6.2.7 Aspektiohjelmointi Toiminnallisuus ei aina rajaudu tarkalleen yhden elementin sisälle. Laatuvaatimukset, kuten suorituskyky, voivat olla koko järjestelmän elementtien toiminnasta kiinni. Tämän kokonaisuuden ymmärtäminen, toteuttaminen ja hallinta on kuitenkin vaikeaa [41, sivu 2] sekä ylläpito raskasta. Aspektiohjelmointi [39, sivu 1] pyrkii erottamaan nämä hajaantuneet ominaisuudet erilleen useista elementeistä. Ominaisuuksia, jotka eivät kapseloidu yhteen elementtiin, kutsutaan *aspekteiksi*. Tuloksena saadaan uudelleenkäytettävämpi ratkaisu ja kokonaisuuksista itsenäisempiä. Kehitettävä järjestelmä koostuu aspekteista ja elementeistä, jotka kootaan yhdeksi kokonaisuudeksi. Punoja (engl. *weaver*) yhdistää aspektit ja elementit toimivaksi kokonaisuudeksi.

AOP-toteutus sovelluksesta koostuu elementtikielestä, jolla toteutetaan elementit; aspektikielestä, jolla toteutetaan aspektit; punojasta, jolla aspektit ja elementit yhdistetään; elementtiohjelmasta sekä aspektiohjelmasta. Tavoitteena on, että elementtien sisältämä toiminta ei hajoa useamman elementin välille. Aspektikielen avulla toteutettavat hajanaiset ominaisuudet määritellään aspekteiksi. Aspektikieli on usein ohjelmointikielen laajennus. Esimerkiksi AspectJ [38] on Javan [4] laajennus, joka mahdollistaa aspektiohjelmoinnin. AspectJ käyttää olioita ja aspekteja käsitteiden erottamiseen ja tätä kautta modularisoidun rakenteen aikaansaamiseksi [53, sivu 176].

Aspektit ovat keino eristää hajautettujen ominaisuuksien toteutus omiin moduuleihinsa. Tuotelinjojen kohdalla juuri ristiriitaongelma liittyy aiheeseen. Piirreverkkoa on vaikea hallita komponttien välisten monimutkaisten suhteiden vuoksi. Aspektiohjelmointi tarjoaa keinon vähentää ja eristää näitä suhteita. Tämä johtaa selkeämpään piirreverkkoon ja yksinkertaisempaan toteutukseen.

6.2.8 Geneerinen ohjelmointi Geneerinen ohjelmointi [46, sivu 1] pyrkii tuottamaan algoritmeja ja tietorakenteita abstraktilla tasolla. Tavoitteena on tilanne, jossa algoritmit ovat riippumattomia tiedon esitystavasta. Data talletetaan tietorakenteisiin, jotka parametrisoidaan käyttöön otettaessa. Geneerinen ohjelmointi jakaa abstraktiot neljään luokkaan: datan, algoritmien, rakenteen ja esitystavan abstraktioi-

hin [46, sivu 2].

Data-abstraktiot ovat tietotyyppejä ja niille määriteltyjä operaatioita. Ne voidaan käsittää erillisinä omasta toteutuksestaan. Data-abstraktioihin voidaan liittää myös algoritmien abstraktioita. Säiliötyypeille on esimerkiksi olemassa useita lajittelualgoritmeja. Rakenteellinen abstraktio voidaan käsittää algoritmisten abstraktioiden leikkauksina, jonka tuloksena syntyy erikoistuneempia algoritmien ja rakenteiden kombinaatioita. Esitystapa on myös abstraktio. Allaoleva tietorakenne voidaan esittää toisena, kun määritellään uudet operaatiot, jotka käyttävät allaolevan rakenteen operaatioita.

Geneeristä ohjelmointia on sovellettu runsaasti tieteellisiin kirjastoihin, ja se on tehokas tapa toteuttaa elementtejä ja niiden rajapintoja algoritmeihin [15, sivu 1]. Geneerisen ohjelmoinnin tunnettu sovelluskohde on C++:n *Standard Template Library* [54]. Geneeristä ohjelmointia voidaan soveltaa uudelleenkäytön tehostamiseksi myös suunnittelumalleihin [16, sivu 113], [15, sivu 19].

6.3 Järjestelmägeneraattorit

Elementtejä voidaan toteuttaa ohjelmoimalla elementit ja kirjoittamalla ohjelma, joka niitä käyttää. Järjestelmägeneraattorit toimivat hieman eri tavalla. Elementit toteutetaan noudattaen abstrakteja kerroksia. Järjestelmägeneraattori kokoaa tai kääntää järjestelmän näistä erillisistä elementeistä. Tunnetuin järjestelmägeneraattori on *GenVoca*.

6.3.1 GenVoca GenVoca on järjestelmägenerointiin tarkoitettu menetelmä, joka pyrkii sovellusten rakentamiseen elementeistä *kytke ja käytä* -periaatteella. Mitään manuaalisesti kirjoitettavaa koodia kuin elementit itse ei tarvitse tuottaa. GenVoca määrittelee kerrokset, jotka kapseloivat luokkia sisäänsä. Saman rajapinnan toteuttavat kerrokset muodostavat alueita (engl. *realm*). Eri kerrosten sisältämien luokkien välillä voi tapahtua jalostumista, jossa kerroksen A luokka jalostaa tai perii kerroksen B sisältämän luokan [7, sivu 1]. Sovellus kootaan toisiinsa liittyneistä kerroksista käyttämällä esimerkiksi P2-generaattoria, joka kokoaa P++-kielellä toteutetut elementit sovellukseksi.

GenVoca määrittelee kerrokset, jotka vastaavat elementtejä. Kerroksella on rajapinta, jonka jokainen kerroksen implementaatio toteuttaa. Kerrokset liittyvät toisiinsa määritellyn kieliopin mukaisesti. Uusi kerros lisää aina uuden säännön kielioppiin. Kerrosten liittyessä toisiinsa puhutaan kollaboraatiosta. Kerrosten luokat jalostavat toisiaan, ja näiden jalostushierarkioiden terminaaliluokat instantioituvat sovelluksessa. Jalostus toteutetaan käyttäen lisäkekerroksia (engl. *mixin layer*) ja parametrisointia [8, sivu 7]. Kerrosten sisältämät luokat toteuttavat rajapintoja, jotka määritetään parametrillä. Lisäke on siis keino toteuttaa kerrostenvälinen kollabo-

raatio antamalla kerrokselle parametrinä toinen kerros.

Kerrosten tarjoamat liitännät määritellään kieliopin avulla. Jokainen alue ja sen elementit määrittävät säännön, mitkä elementit tai alueet voivat liittyä niihin. Esimerkkinä [9, sivu 5] määritellään alueet $S = \{a, b, c\}$, $T = \{d[S], e[S], f[S]\}$ ja $W = \{n[W], m[W], p, q[T, S]\}$. Näille määritellään kielioppi:

$$\begin{aligned} S &:= a \mid b \mid c; \\ T &:= dS \mid eS \mid fS; \\ W &:= nW \mid mW \mid p \mid qTS; \end{aligned}$$

Tuotelinjanäkökulmasta kerros vastaa piirrettä, joka on yhteinen tuoteperheelle. Tuote koostuu sopivalla tavalla yhteen liitetyistä kerroksista. Jokainen tuote valitsee sopivan kerroksen toteutuksen, joka vastaa asetettuja vaatimuksia. Tuoteperhe määritellään GenVocassa kaikkina kieliopin mahdollistamina kombinaatioina.

6.4 Yhteenveto

Tuotelinjan elementtien toteuttamiseen on useita vaihtoehtoja. Erityisesti on otettava huomioon uudelleenkäytettävyys ja tulevat muutokset. On tärkeää ymmärtää, miten muunneltavia elementtejä voidaan toteuttaa. Tässä luvussa esiteltyjä menetelmiä, kuten perintää, koostamista, rajapintoja ja suunnittelumalleja, käytetään luvussa 8 rannekellotuotelinjan toteutuksen hahmottelussa ja elementttien määrittelyssä.

7 Tuotelinjat ja sovelluskehykset

Sovelluskehykset ovat voimakas uudelleenkäytön väline. Niiden avulla pyritään suunnittelun, analyysin ja lähdekoodin uudelleenkäyttöön [30, sivu 13]. Samoin kuin tuotelinjat myös sovelluskehys määrittelee sovelluksen arkkitehtuurin [30, sivu 13]. Tuotelinjoilla ja sovelluskehyksillä on myös samankaltaisia tavoitteita. Lyhyt tuotantoaika uusille ominaisuuksille, kokonaisuuksien itsenäisyys ja laatuvaatimusten täyttyminen [58, sivu 50], [21, sivu 191] ovat sekä tuotelinjojen että sovelluskehysten pyrkimyksiä. Sovelluskehykset soveltuvatkin hyvin tuotelinjamaiseen elementtiajatteluun, ja ne voidaan nähdä mielenkiintoisena toteutusmenetelmänä tuotelinjalle. Kehys itsessään voi olla elementti tai tuoteperhe riippuen näkökulmasta. Yleinen määritys [11, sivu 242] sovelluskehyksille on:

> A framework is a set of classes that embodies an abstract design for solutions to a family of related problems.

Joissain tilanteissa sovelluskehyksen ja tuotelinjan eroa on vaikea havaita. Periaatteessa tuotelinja voi olla sovelluskehys [8, sivu 2]. Sovelluskehykset voidaan luokitella mustalaatikko- ja lasilaatikkokehyksiin niiden tarjoamien uudelleenkäyttötekniikoiden mukaisesti [58, sivu 52]. Mustalaatikkokehykset mahdollistavat sovellusten tai elementtien rakentamisen koostamalla parametrisoituja luokkia. Lasilaatikkokehykset perustuvat luokka-abstraktioihin ja perinnän kautta tapahtuvaan uudelleenkäyttöön.

7.1 Kehys tuotelinjana

Sovelluskehys voi toimia tuotelinjana. Kehys koostuu joukosta abstrakteja luokkia, jotka määrittävät sovelluksen elementit ja niiden väliset suhteet. Näistä luokista perimällä saavutetaan konkreetti toteutus sovelluskehyksen tarjoamasta abstraktiosta. Toinen toteutustapa perustuu siihen, että parametrisoitavat luokat koostetaan yhdeksi sovellukseksi. Molemmissa tapauksissa tuoteperheen jäsenten kesken on olemassa variaatioita. Vaikka tuoteperhe ei ole rajattu ja määritelty, on sen sovellusalue kuitenkin rajattu [11, sivu 242]. Sovelluskehykset tuotelinjoina muistuttavat läheisesti koodigeneraattoreihin perustuvia tuotelinjoja.

7.2 Kehykset elementteinä

Toinen ajattelutapa, jossa itse kehykset nähdään elementteinä, on mielenkiintoisempi ja tarjoaa monipuolisempia sovellusnäkökulmia. Nyt kehitettävän sovelluksen ei tarvitse rajoittua kehyksen määräämään arkkitehtuuriin tai sovellusalueeseen. Tässä mallissa kehykset toteuttavat rajapinnan, jota sovellusrunko käyttää. Tunnistettuja kehyselementtimalleja ovat: tuotekohtainen laajennusmalli, standardikohtainen laajennusmalli, hienojakoinen laajennusmalli ja generaattorimalli [11, sivu 250].

7.2.1 Tuotekohtainen laajennusmalli Tuotekohtaisessa laajennusmallissa [11, sivu 250] jokaiselle tuotteelle on olemassa oma tuotekehyksen tarjoaman elementin instanssi. Kehys kattaa vain kaikille yhteisen toiminnallisuuden. Tässä mallissa kehys onkin vain tietyn toiminnallisuuden tarjoava kokonaisuus, jonka muokkaus vaikuttaa kaikkiin tuotteisiin. Jokainen tuote lisää tuotekohtaiset ominaisuutensa itse. Tämä voi tapahtua esimerkiksi perimällä alkuperäinen elementti. Esimerkki tämänkaltaisesta uudelleenkäytöstä on Symbian-käyttöjärjestelmän [56] käyttöliittymärajapintalaajennus *Eikon* [14, sivu 24]. Jokainen laitevalmistaja toteuttaa itse Symbianin tarjoaman *Eikon*-rajapinnan, joka peritään *Uikon*-käyttöliittymäkehyksestä. Tuotekohtaisen laajennusmallin mallin vahvuus on sen yksinkertaisuus. Heikkouksiin lukeutuvat tuotekohtaisen lähdekoodin huono uudelleenkäytettävyys ja joustamattomuus.

7.2.2 Standardikohtainen laajennusmalli Standardikohtaisessa laajennusmallissa [11, sivu 251] muunneltavuutta on siirretty enemmän kehyksen vastuulle. Kehys itsessään sisältää standardeja rajapintoja toteutukseltaan vaihteleville osioille ja toteuttaa näitä variaatioita. Nyt kehyksen tarjoama elementti voidaan konfiguroida tuotteelle sopivaksi ilman, että toteutusta joudutaan tekemään kehyksen ulkopuolella. Tämän mallin hyviä puolia ovat yksinkertaisuus ja standardit rajapinnat variaatiopisteille ja muille elementeille. Huonoina puolina ovat uudelleenkäytön vähäisyys, tuotekohtaisten laajennusten puute ja heikentynyt ylläpidettävyys. Vaikka saman sovellusalueen eri standardit eroavat toisistaan, sisältävät ne huomattavasti myös yhtäläisyyksiä. Standardikohtaisessa laajennusmallissa ei tätä yhtenevyyttä hyödynnetä. Tämän mallin mukainen järjestelmä ei salli tuotekohtaisia laajennoksia elementtiin, koska variaatiopisteet ovat elementin sisäisiä ja niille on olemassa kaikille yhteiset toteutukset. Ylläpidettävyys kärsii rajapintamuutosten tapahtuessa, koska muutokset aiheuttavat muutoksia kaikkiin tuotteisiin.

7.2.3 Hienojakoinen laajennusmalli Hienojakoisessa laajennusmallissa [11, sivu 252] kehykseen liitetään pieniä itsenäisiä konfiguroitavia moduuleja, jotka sisältävät korkeintaan muutamia variaatioita. Kehys sisältää rajapinnan ja yleisen kaikil-

le yhteisen toteutuksen. Variaatiot hoidetaan konfiguroimalla kehykseen näitä pieniä moduuleja. Mallin mukaiset kehykset ovat hyvin joustavia ja uudelleenkäytettäviä. Tosin ongelmana on löytää kustannustehokas moduulikoko. Hienojakoisen laajennusmallin suurin heikkous on sen monimutkaisuus. Sen käyttö voi olla hyvin monimutkaista riippuen variaatiopisteiden määrästä.

7.2.4 Generaattorimalli Generaattorimalli [11, sivu 252] on hienojakoisen mallin laajennus. Siinä ajatusta konfiguroitavista variaatioista viedään askeleen pidemmälle. Kun variaatiopisteet ja hyödylliset laajennokset on tunnistettu, pyritään niiden hallintaan kehittämään työkaluja. Yleisimmät lähestymistavat tähän ovat joko graafisen konfigurointityökalun tai DSL:n (*domain specific language*) käyttö. Tämän mallin etuja ovat korkea uudelleenkäytettävyys ja joustavuus. Työkalujen avulla voidaan prosessiin lisätä myös elementin semantiikan tarkistus. Haittapuolina voidaan nähdä tuotekohtaisten laajennusten hankaluus ja kehyksen kehittämisen kalleus. Työkaluja on aina muokattava, kun kehystä muokataan. Generaattori myös rajoittaa kehyksen laajuutta, koska se harvoin tukee mullistavia muutoksia alkuperäiseen mielikuvaan nähden.

7.3 Yhteenveto

Tuotelinjoilla ja sovelluskehyksillä on selkeästi havaittavia yhtäläisyyksiä ja yhteisiä tavoitteita. Sovelluskehykset voidaankin nähdä mielenkiintoisena keinona toteuttaa tuotelinjoja. Sovelluskehyksiä ei kuitenkaan hyödynnetä luvun 8 rannekellotuotelinjassa, koska on valittu toinen toteutuslähestymistapa.

8 Rannekellotuotelinja

Tähän saakka on edetty pelkän teorian pohjalta. Tässä luvussa on tarkoituksena demonstroida esimerkin avulla tuotelinjan perustamisen vaiheita ja tuloksia. Tavoitteena on määritellä rannekelloja tuottava tuotelinja. Sovellusalueen valintaa on perusteltu kohdassa 8.1. Määrittely tehdään mukaillen Arangon sovellusanalyysimallia, joka on esitelty kohdassa 3.1. Valintaa on perusteltu kohdassa 3.8. Kun tuotelinjan laadulliset ja toiminnalliset tavoitteet on määritelty, suunnitellaan sovelluksen arkkitehtuuri mukaillen osittain kohdassa 4.4 esiteltyä ADD-prosessia. ADD-prosessin soveltamista on perusteltu kohdassa 4.6. Tämän jälkeen tutkitaan määriteltyä tuotelinjaa. Pyrkimyksenä on tarkastella tuotelinjan perustamisessa vastaantulevia ongelmia, saavutettuja tuloksia ja lähestymistavan kannattavuutta.

Määriteltävästä tuotelinjasta oletetaan muodostuvan monimutkainen kokonaisuus. Tämän vuoksi suunnitteluvaiheen oletetaan olevan raskaampi kuin yhtä ohjelmistoa suunniteltaessa. Tuoteperheelle asetettujen laatuvaatimusten oletetaan täyttyvän arkkitehtuurisuunnittelun myötä. Lähdekoodin hyödyntämisen uskotaan olevan tehokasta elementtipohjaisen suunnittelun takia. Tuoteperheen osalta odotetaan kustannusten jakautumisesta useamman tuotteen kesken seuraavan tuotannon tehostumista. Muodostettavat johtopäätökset ovat suuntaa antavia. Tulosten lopullinen varmentaminen vaatisi määritellyn tuotelinjan toteutusta, kehityksen seurausta ja vertailevan tutkimuksen suorittamista vastaavan yhden tuotteen projektin kanssa.

8.1 Sovellusalueen määritys

Tuotelinjan sovellusalueeksi on valittu rannekellot. Rannekello on tuote, joka voi ominaisuuksiltaan vaihdella paljon mutta peruselementeiltään pysyy silti samana. Rannekellon tapauksessa ominaisuuksien on hyödyllistä jakaa resursseja niiden vähyyden vuoksi. Rannekellojen kohdalla on myös tarve tuottaa erilaisia malleja käyttäjien tarpeisiin. Ihmiset tarvitsevat eri ominaisuuksilla varustettuja kelloja. Esimerkiksi urheilijat kaipaavat harrastukseen sopivaa kelloa ja tavallinen kuluttaja saattaa tyytyä riisutumpaan malliin. Samoin rannekello on tuote, jolla on hyvät myyntiodotukset, koska suurin osa hyvinvointivaltioissa asuvista ihmistä käyttää kelloa. Tuotteen menestymiseen vaikuttavat ainakin hinta, laatu ja ominaisuudet. Laadukas ja monipuolinen tuote on kiinnostava. Alhainen hinta vaikuttaa positiivisesti ostopäätökseen samoilla ominaisuuksilla varustettuun kalliimpaan tuotteeseen verrattuna. Monipuolisilla ominaisuuksilla varustettu laadukas rannekello alhaiseen hintaan on

hyvä yhdistelmä. Tässä tapauksessa ei ole järkevää tuottaa jokaista mallia erillään toisista saman sarjan tuotteista. Näin syntyisi runsaasti päällekkäistä toimintaa ja kustannusten kasvua, mikä vaikuttaisi hintaan ja ominaisuuksien määrään. Sovellusalueeltaan rannekellot sopivat ilmeisen hyvin tuotelinjalähestymistapaan.

8.1.1 Tiedon keräys Rannekellosovellusalue on määriteltävä tarkemmin. Tämä aloitetaan tutkimalla, mitä ominaisuuksia rannekellossa voi olla. Ensimmäiseksi sen on näytettävä kellonaika. Tämä on jokaisen kellon perustehtävä. Ajan lisäksi päivämäärä on tärkeä tieto käyttäjälle. Näiden ominaisuuksien varaan voidaan rakentaa jo monia valinnaisia ominaisuuksia. Rannekellolla voidaan mitata kokonaisaikaa ja kierrosaikoja. Nämä voisivat olla tärkeitä ominaisuuksia urheilijalle. Hälytys on perinteinen rannekellojen ominaisuus. Näiden lisäksi kellolla voitaisiin esimerkiksi mitata lämpötila ja pulssi, kuunnella radiota, nauhoittaa ääntä tai ottaa valokuvia. Näiden ominaisuuksien kohdalla olisi tärkeää tutkia markkinoita ja kustannuksia ennen päätöksentekoa.

8.2 Tiedon analysointi ja luokittelu

Tiedon keräyksen jälkeen materiaalia pitää analysoida ja luokitella. Tarkoituksena on tuottaa edullisia peruskelloja tavalliselle käyttäjälle. Tarvittavia ominaisuuksia ovat aika, päivämäärä, ajanotto ja muistutus. Nämä jaetaan pakollisiin ja valinnaisiin ominaisuuksiin. Tässä vaiheessa pitää myös selvittää tuotteiden väliset yhtenevyydet, eroavaisuudet, rajattavat kokonaisuudet ja niiden väliset suhteet sekä rajoitukset. Jokaiselle tuotteelle määritellään pakollisina aika ja päivämäärä. Ajanotto ja muistutus ovat valinnaisia ominaisuuksia. Rannekellon kohdalla kaikki piirteet jakavat ainakin käyttöliittymän. Tämä sisältää näytön ja painikkeet. Muisti ja virrankulutus vaikuttavat yhtälailla kaikkien piirteiden toimintaan. Tässä vaiheessa voidaan myös määrittää tuotteen koostavat piirteet: kellonaika, päivämäärä, ajanotto ja muistutus. Näiden alle luokitellaan erilaiset variaatiot.

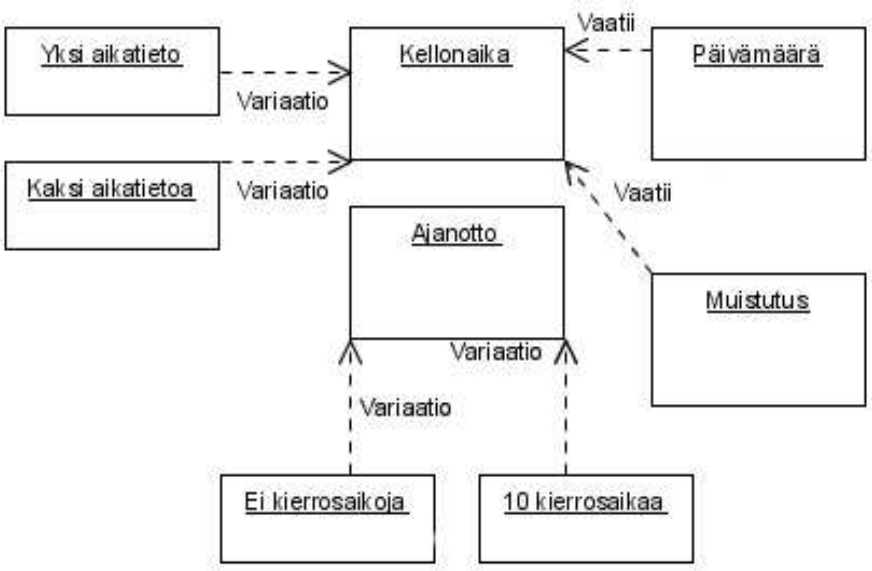
8.2.1 Kellonaika Kellonajan tehtävä on julkaista aikatieto. Se ei vaadi mitään muita piirteitä. Kellonaika esitetään digitaalisessa formaatissa. Kellonaikaa on voitava säätää. Kellonaika on pakollinen piirre, jota ilman tuotetta ei voi rakentaa. Muodostettavat variaatiot kellonaikapiirteestä ovat kahden eri aikatiedon tallettava ja näyttävä ja vain yhden aikatiedon näyttävä kellonaika.

8.2.2 Päivämäärä Päivämäärän tehtävä on näyttää päivämäärä. Se vaatii kellonajan toimiakseen. Se tarvitsee kellonaikaa voidakseen vaihtaa päivämäärää. Päivämäärä esitetään digitaalisessa formaatissa. Päivämäärä on voitava säätää. Päivämäärä on pakollinen piirre, jota ilman tuotetta ei voida rakentaa. Päivämäärästä ei

ole variaatioita.

8.2.3 Ajanotto Ajanoton tehtävä on mitata suoritusaikaa. Se ei vaadi muita piirteitä toimiakseen. Ajanotto esitetään digitaalisessa formaatissa. Ajanotto on valinnainen piirre. Ajanotto on voitava käynnistää, pysäyttää ja nollata. Muodostettavat variaatiot ajanottopiirteestä ovat ilman väliaikoja ja 10 väliajan ajanotto. Väliaikojen vuoksi vaaditaan myös väliajan merkintätoiminto.

8.2.4 Muistutus Muistutus aiheuttaa hälytyksen haluttuna kellonaikana. Se vaatii kellonaikaa toimiakseen. Muistutus esitetään digitaalisessa formaatissa. Muistutus pitää voida asettaa ja poistaa. Muistutus on valinnainen piirre. Muistutuksesta ei ole olemassa variaatioita.



Kuva 8.1: Piirrediagrammi

8.2.5 Laatuvaatimukset Tuoteperheeltä toivottavat laatuvaatimukset on määriteltävä. Pelkästään laajat ominaisuudet eivät tee kellosta hyvää vaan vaaditaan myös tarpeellista laadukkuutta. Tuotteiden halutaan olevan joustavia ja ylläpidettäviä. Rannekelloissa muistinkulutuksen ja virrankulutuksen pitää olla pieni. Tällöin tuotteiden pitää viedä vähän resursseja eli olla tehokkaita. Rannekellon halutaan lisäksi toimivan ilman ongelmia. Tuotteiden pitää siis olla luotettavia. Laatuvaatimuksissa keskitytään ylläpidettävyyteen, tehokkuuteen ja luotettavuteen.

8.2.6 Määritelty tuoteperhe Tuoteperhe on nyt määritelty. Neljää piirrettä ja niiden variaatioita käyttäen laajuudeksi muodostuu 12 tuotetta, jotka vaihtelevat ko-

koonpanoltaan ja ominaisuuksiltaan. Muutamalla piirtellä saadaan aikaan useita tuotteita. Tietysti voidaan pohtia, onko järkevää jättää yhdestäkään kellosta pois ajanottoa tai muistutusta ja mikä on riittävä eroavaisuus, jotta voidaan puhua eri tuotteista. Oikeastaan tuotteet vain sisältävät toisiaan. Teoriassa kuitenkin toisistaan poikkeavien tuotteiden lukumäärä on huomattavan suuri jo pienilläkin piirteiden lukumäärillä. Taulukossa 8.1 on esitelty tuoteperheen jäsenet eri ominaisuuksilla. Kyseisessä taulukossa *kellonaika 1* tarkoittaa yhden aikatiedon tallettavaa ja *kellonaika 2* kaksi aikatietoa tallettavaa kellonaikapiirrettä. Samoin *ajanotto 1* tarkoittaa ajanottopiirrettä ilman kierroasaikoja ja *ajanotto 2* ajanottopiirrettä 10 kierrosajalla.

Tuote	Kellonaika 1	Kellonaika 2	Päivämäärä	Ajanotto 1	Ajanotto 2	Muistutus
Tuote 1	X		X			
Tuote 2		X	X			
Tuote 3	X		X			X
Tuote 4		X	X			X
Tuote 5	X		X	X		
Tuote 6	X		X		X	
Tuote 7		X	X	X		
Tuote 8		X	X		X	
Tuote 9	X		X	X		X
Tuote 10	X		X		X	X
Tuote 11		X	X	X		X
Tuote 12		X	X		X	X

Taulukko 8.1: Määritelty tuoteperhe

8.3 Arkkitehtuurityyli

Kun tuoteperhe on määritelty, on aika siirtyä rakenteen ja toteutuksen määrittelyyn. Ensimmäiseksi on tunnistettava arkkitehtuuriset tavoitteet ja valittava niiden perusteella sovellettava arkkitehtuurityyli.

8.3.1 Arkkitehtuuriset tavoitteet Halutut arkkitehtuuriset tavoitteet heijastelevat pitkälti laatuvaatimuksia. Arkkitehtuurin määrittelyllä pyritään myötävaikuttamaan luotettavuuden, tehokkuuden, ylläpidettävyyden ja joustavuuden toteutumiseen. Arkkitehtuurin pitää luonnollisesti sopeutua tuotelinjan aiheuttamiin variaatioihin. Vaatimuksista tärkeimmät ovat ylläpidettävyys, luotettavuus ja joustavuus. Nämä valitaan sen takia, että rannekellon luotettava toiminta on tärkeää. Päivän tapahtumat kuten työ, koulu ja tapaamiset on määritelty kellonaikojen mukaan. Jos kello ei toimi luotettavasti, tapahtumarytmi kärsii. Ylläpidettävyys ja joustavuus ovat tärkeitä tulevan kehityksen kannalta. Tuotetta pitää pystyä kehittämään ja muokkaamaan, jotta se menestyisi myös tulevaisuudessa. Toki myös virrankulutus sulautetuissa järjestelmissä on aina otettava huomioon.

8.3.2 Arkkitehtuurityylin valinta Arkkitehtuurityylin valinta on ensimmäinen askel arkkitehtuurin määrittelyssä. Tyylin valinta ohjaa arkkitehtuurin kykyä täyttää laatuvaatimukset. Seuraavaksi pohditaan arkkitehtuuristen tyylien sopivuutta käsiteltävässä tapauksessa.

Eri tyylien heikkouksia ja vahvuuksia on tutkittu kirjallisuudessa [11, sivu 117], [20, sivu 5]. Piippu-suodin-tyylissä suorituskyky voi koitua ongelmaksi. Ylläpidettävyydeltään tyyli on hyvin joustava mutta muutokset vaikuttavat useampaan järjestelmän kokonaisuuteen. Myöskään luotettavuus ei ole piippu-suodin-tyylin vahvuus, eikä se sovellu interaktiivisiin järjestelmiin. Kerrosarkkitehtuurin suorituskyky ja luotettavuus eivät ole sen vahvoja ominaisuuksia mutta toisaalta se on hyvin ylläpidettävä. Tietovaraston heikkouksia ovat suorituskyky ja vahvuuksia ylläpidettävyys. Luotettavuudeltaan tyyli voi olla ongelmallinen. Implisiittinen kutsu on kohtuullisen luotettava ja hyvin ylläpidettävä. Tehokkuus ei kuitenkaan ole tietyin rajoituksin sen vahvuus. Olioarkkitehtuurityyli on neutraali luotettavuuden suhteen mutta suorituskyvyltään ja ylläpidettävyydeltään se on riittävä. Käsitellyistä tyyleistä ei mikään ole optimaalinen tavoitelluille laatuvaatimuksille. Useampi mahdollisuus kävisi tässä tilanteessa. Parhaiten soveltuvat kuitenkin implisiittinen kutsu ja oliotyyli.

Oliotyylissä [11, sivu 125] tehokkuuskysymys on myös paljon suunnittelijan harteilla. Optimaalinen tulos saavutetaan minimoimalla yleisimpien käyttöskenaarioiden resurssienkulutusta. Tämä voi tapahtua ylläpidettävyyden kustannuksella. Ylläpidettävyyden kannalta olisi tärkeää suunnitella järjestelmä todennäköisimmät muutokset huomioonottaen. Tehokkuus ja ylläpidettävyys on kuitenkin mahdollista yhdistää myös onnistuneesti. Luotettavuudeltaan oliotyyli on neutraali. Luotettavuuskin on tosin riippuvainen oliosuunnittelun painopisteistä. Haittapuolena luotettavuuden osalta voidaan mainita korkeamman tason vianhallinnan vaikea toteuttaminen.

Implisiittinen kutsu [11, sivu 127] mahdollistaa tehokkaan uudelleenkäytön ja helpottaa järjestelmän kehittämistä. Elementtien korvaus uusilla on löyhien suhteiden ansiosta helppoa. Implisiittisen kutsun käyttämä tapahtumankäsittelytekniikka voi tosin heikentää arkkitehtuurin suorituskyvyn laatua. Luotettavuudeltaan oliotyyli ja implisiittinen kutsu ovat samaa luokkaa. Implisiittisen kutsun tapauksessa tehokkaampi korkean tason vianhallinta on kuitenkin mahdollista.

Kahden parhaiten soveltuvan tyylin välillä ei voida selkeästi määrätä paremmin soveltuvaa tyyliä. Oliotyyli asettaa kovemmat vaatimukset suunnittelun suhteen kun taas implisiittinen tyyli on selkeämpi toteuttaa. Oliotyyli voi suunnittelun painopisteistä riippuen olla suorituskykyisempi mutta implisiittinen kutsu taas ylläpidettävämpi. Luotettavuudeltaan implisiittinen kutsu on lievästi parempi.

Sovellettavaksi valitaan tässä tapauksessa implisiittinen kutsu, koska se on hyvin ylläpidettävä ja joustava sekä luotettavuudeltaan kohtuullinen. Tuotelinjoissa

vaikeiden riippuvuuksien ja suhteiden hallinta voi implisiittisessä kutsussa olla helpompaa kuin oliotyylissä. Implisiittisen kutsun ongelmakohdat kuitenkin tunnistetaan. Implisiittisessä kutsussa elementit aiheuttavat tapahtumia, joita voidaan rekisteröityä kuuntelemaan. Kuuntelija sitten toteuttaa tapahtuman käsittelyn kutsumalla haluamiaan metodeja. Tapahtuman aiheuttajien ei tarvitse tietää, kuka kuuntelija itse asiassa on. Tämän ansiosta elementit ovat itsenäisiä ja järjestelmä erittäin joustava. Implisiittisen kutsun haittana pidetään luotettavuuden kannalta sitä, että tapahtuman käsittelyä ei suoraan taata. Myöskin kontrollin hallinta voi olla vaikeaa, koska tapahtumien suoritusjärjestystä on vaikea seurata.

8.4 Elementtien tunnistus

Seuraava askel on elementtien tunnistaminen. Kehitettävän järjestelmän tapauksessa piirteet jakautuvat omiksi elementeikseen. Tämä on järkevää, koska toiminnallisuus on jakaantunut selkeisiin kokonaisuuksiin ja tätä jakoa kannattaa luonnollisesti hyödyntää. Kukin elementti on myös sopivan kokoinen, eikä kata liian laajaa toiminnallisuutta sisäänsä. Piirteet kattavat kuitenkin vasta osan järjestelmän toiminnallisuudesta. Tunnistetaan tässä vaiheessa neljä piirre-elementtiä: kellonaika, päivämäärä, ajanotto ja muistutus.

Vaikka elementit ovat melko erillisiä, ne tarvitsevat toimiakseen myös ympäristön, joka ne yhdistää. Tätä varten tarvitaan lisää elementtejä. Näyttöelementti vastaa näytöstä ja käyttöliittymästä, joiden kautta elementtejä voidaan käyttää. Nimetään vielä kellomoottori, joka hoitaa järjestelmän logiikan. Tunnistetut kuusi elementtiä ovat nyt näyttö, kellomoottori, ajanotto, muistutus, päivämäärä ja kellonaika. Tunnistetut elementit täytyy tietysti määritellä tarkemmin toiminnaltaan, rajapinnaltaan ja vastuualueiltaan.

Tässä vaiheessa tarvitaan kuitenkin yksi moduuli lisää, joka kattaa kaikille elementeille yhteisiä kokonaisuuksia. Kyseessä voisi olla eräänlainen kirjasto, jonka toiminta ei näy suoraan järjestelmässä. Tässä moduulissa määritellään ainakin *Plugin*-luokka, jonka kaikki elementit perivät. Näin on mahdollista määritellä yleisiä elementtitason ominaisuuksia muuttamatta jokaista elementtiä suoraan. Myös järjestelmässä aiheutuvien tapahtumien yliluokka *Event* on järkevää määritellä tässä moduulissa. Näin siis järjestelmällisesti määriteltäisiin arkkitehtuurin elementit ja liittimet. Näiden luokkien toteutukseen ei kuitenkaan lähemmin perehdytä vaan ne mainitaan suunnittelupäätöksinä, jotka helpottavat suunnittelua ja toteutusta.

8.5 Näyttöelementti

Näyttöelementti on järjestelmän näkyvä osa ja käyttöliittymä. Sen kautta hallitaan digitaalinäyttöä. Digitaalinäyttö näyttää kellonajan, ajanoton, muistutuksen ja päivämäärän sekä symbolin asetetusta muistutuksesta ja näytettävästä aikamoodista. Käyttöliittymänä toimii neljä painiketta, joille kuormitetaan toimintoja tilasta riippuen. Näyttö voi olla enimmillään kolmessa tilassa, joissa kussakin tilassa voidaan käyttää tiettyä piirrettä. Tilojen määrä riippuu elementtien määrästä. Ensimmäinen tila näyttää kellon ja päivämäärän. Tässä tilassa niitä voidaan tarkastella ja asettaa sekä mahdollisesti vaihtaa aikavyöhykettä. Toinen tila on hälytystila. Siinä voidaan asettaa ja tarkastella hälytystä. Kolmannessa tilassa voidaan ottaa aikaa, väliaikoja ja tarkastella niitä. Näyttö on vuorovaikutuksessa suoraan moottorin kanssa. Tämän lisäksi on näytön saatava viite aktiiviseen elementtiin, jotta jatkuva päivitys olisi mahdollista.

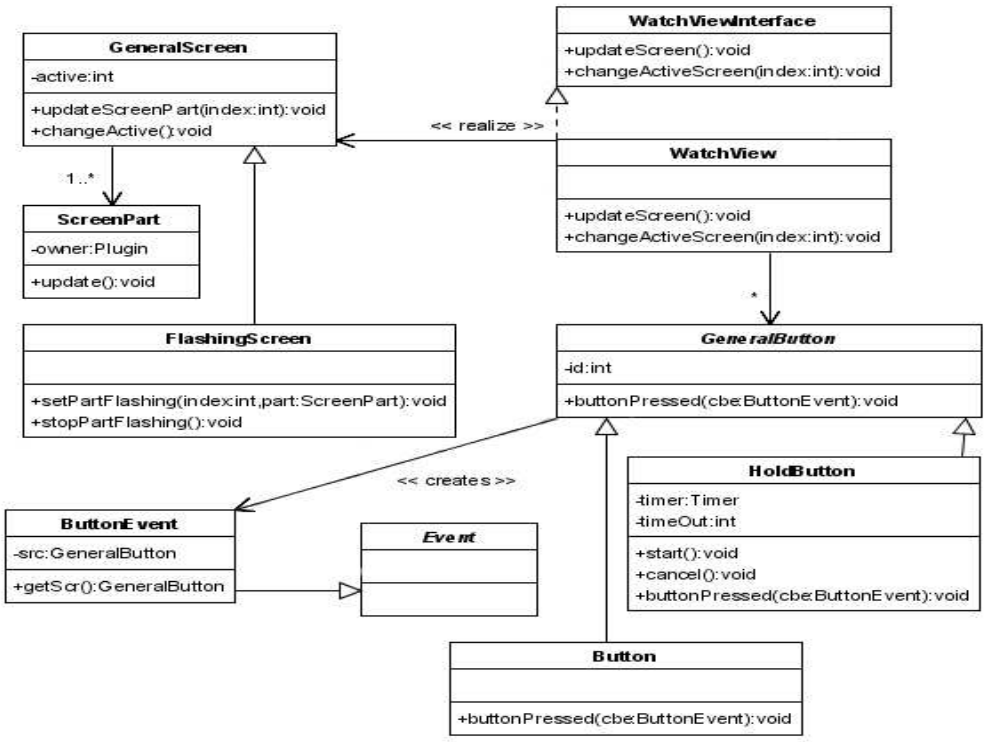
8.5.1 Vastuualueet Näytön tehtävä on tarjota käyttöliittymä rannekellon ominaisuuksille ja visualisoida käyttäjälle tapahtuvat toiminnot. Näyttö aiheuttaa tapahtumia, jotka moottori käsittelee, ja reagoi moottorilta tuleviin viesteihin.

8.5.2 Luokkarakenne Elementti määrittelee *GeneralButton-*, *Button-*, *HoldButton-*, *ButtonEvent-*, *WatchView-*, *ScreenPart-*, *GeneralScreen-* ja *FlashingScreen*-luokan. Näitä käytetään *WatchView*-luokan toteuttaman *WatchViewInterface*-rajapinnan kautta. Luokkien väliset suhteet on mallinnettu kuvassa 8.2. Yleisistä *GeneralScreen-* ja *GeneralButton*-luokista voidaan periä erikoistuneempia versioita tarpeen mukaan (esim. *FlashingScreen*, *HoldButton*). *GeneralButton* on abstrakti painikeluokka, jonka aliluokan instanssi voi aiheuttaa *ButtonEvent*-tapahtuman. *WatchView* sisältää viitteet käytettyihin painikkeisiin ja *GeneralScreen*-luokkaan. *GeneralScreen* on digitaalinäyttö, joka voi jakaantua useaan osaan. Jokaista näytön osaa vastaa *ScreenPart*-luokan ilmentymä. *ScreenPart* sisältää viitteen visualisoimaansa elementtiin, jolloin jatkuva päivitys on mahdollista. Esitetty luokkarakenne ei ole täydellinen mutta kuvaa oleelliset luokat ja niiden väliset suhteet.

8.5.3 Riippuvuudet Näyttö ei ole riippuvainen muista elementeistä.

8.5.4 Konfiguraatio Näyttö on irrallinen kokonaisuus, joka on vaihdettavissa, kunhan sen tarjoama rajapinta pysyy samana moottorille. Tuotelinjan rannekellot voivat vaihdella esimerkiksi näytön visuaaliselta rakenteelta ja käyttöliittymältä.

8.5.5 Rajapinnat Näyttö toteuttaa rajapinnan *WatchViewInterface*, jonka kautta sitä voidaan käsitellä.



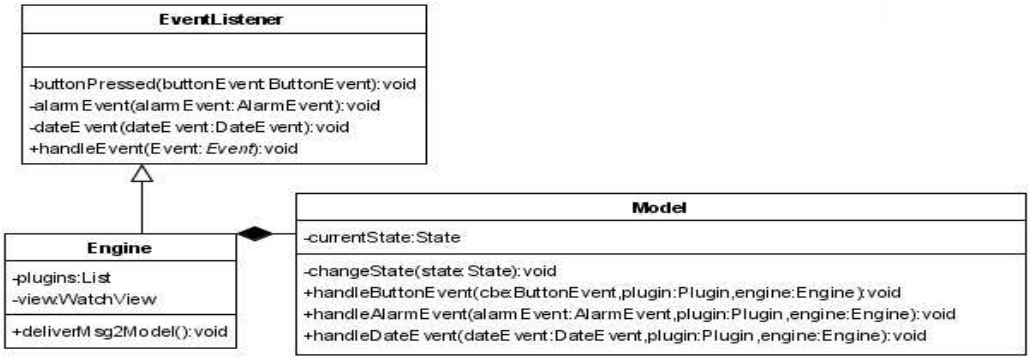
Kuva 8.2: Näyttöelementin luokkarakenne

8.6 Kellomoottorielementti

Kellomoottori on järjestelmän osien vuorovaikutusta ohjaava elementti. Se sisältää viitteet muihin elementteihin ja välittää viestejä elementeiltä näytölle sekä näytöltä elementeille. Moottorin on käsiteltävä järkevällä tavalla tuoteperheen varioituvuus.

8.6.1 Vastuualueet Kellomoottori välittää viestejä näytön ja elementtien välillä sekä toteuttaa variaatiopisteet elementtiliittymien osalta.

8.6.2 Luokkarakenne Elementti määrittelee luokat *Engine*, *EventListener* ja *Model*. Moottori on rakenteeltaan hyvin yksinkertainen. *Engine* on liitokset piirteet toteuttaviin elementteihin ja näyttöön määrittelevä luokka. Luokka *Engine* perii *EventListener*-luokan, joka kuuntelee kaikkia elementtien ja näytön aiheuttamia tapahtumia. *Model* on tilakone, jonka rakenne täytyy määritellä jokaiselle tuotteelle erikseen. Tämän avulla jokainen kokoonpano toimii oikein. *Engine* käyttää siis vaihdettavaa mallia logiikan toteukseen. Tilakone on toiminnaltaan äärimmäisen yksinkertainen ratkaisu. *Model*-luokan ja *Engine*-luokan liitoksessa on käytetty *Strategia*-suunnittelumallia (engl. *Strategy*) [16, sivu 315]. *Model*-luokassa on käytetty *Tila*-suunnittelumallia (engl. *State*) [16, sivu 305] kellon tilan toteutukseen. Luokkien väliset suhteet on mallinnettu kuvassa 8.3.



Kuva 8.3: Moottorielementin luokkarakenne

8.6.3 Riippuvuudet Moottori ei ole riippuvainen muista elementeistä.

8.6.4 Konfiguraatio Kellomoottori reagoi järjestelmän konfigurointiin vaihdettavan mallin avulla. Jokaiselle tuotteelle on määriteltävä erikseen malliluokka, joka toteuttaa tilasiirtymäautomaatin. Tässä automaatissa määritellään painikkeiden painalluksista ja muista tapahtumista aiheutuvat toiminnot ja tilasiirtymät.

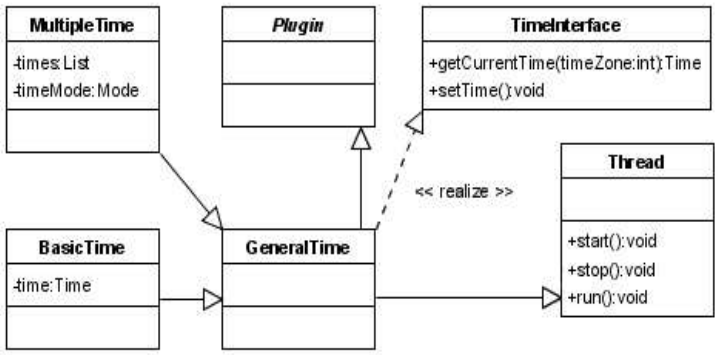
8.6.5 Rajapinnat Mikään elementti ei varsinaisesti käytä moottoria vaan moottori käyttää muita elementtien rajapintoja. Rannekellon tapauksessa moottoriin voisi liittyä matalan tason laiterajapintoja. Tietyllä tavalla rajapintana voidaan myös mieltää tapahtumien kuuntelu, jonka kautta moottori vastaanottaa viestejä elementeiltä.

8.7 Kellonaikaelementti

Kellonaikaelementti on kellon toteuttava elementti, joka julkaisee aikatietoa. Se on hyvin itsenäinen piirre. Kellonaikaelementti keskustelee kellomoottorin kanssa, joka käyttää sitä. Kellonajan näyttää lopulta näyttö, jota moottori ohjaa. Kellonaika pyörii omassa säikeessään, joka on jatkuvasti päällä.

8.7.1 Vastuualueet Kellonaikaelementin tehtävä on toteuttaa kellonaikapiirre.

8.7.2 Luokkarakenne Aikaelementti määrittelee *TimeInterface*-rajapinnan, jonka *GeneralTime*-luokka toteuttaa. *GeneralTime* peritään *Thread*-luokasta, jotta se voi suorittaa itseään ilman katkoksia omassa säikessään. *GeneralTime*-luokasta peritään *BasicTime*, joka toteuttaa yhden kellonajan mahdollistavan ajan ja *MultipleTime*, joka mahdollistaa useamman ajan tallennuksen. Luokkien väliset suhteet on mallinnettu kuvassa 8.4.



Kuva 8.4: Aikaelementin luokkarakenne

8.7.3 Riippuvuudet Kellonaikaelementti ei ole riippuvainen muista elementeistä.

8.7.4 Konfiguraatio Kellonaika voidaan konfiguroida julkaisemaan yksi aikatieto tai kaksi aikatietoa. Tämä tapahtuu instantioimalla *GeneralTime*-luokan oikea aliluokka.

8.7.5 Rajapinnat Kellonaikaelementti määrittää *TimeInterface*-rajapinnan. Tämän rajapinnan kautta aikatietoa voidaan kysyä ja määrittää.

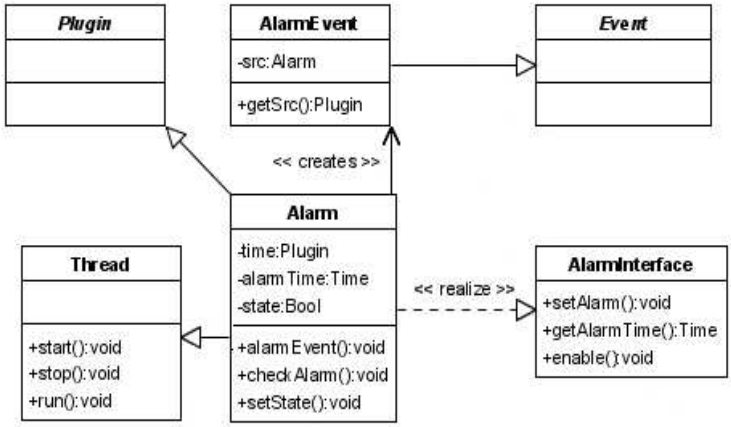
8.8 Muistutuselementti

Muistutuselementti sisältää muistutustoiminnon toteutuksen. Muistutus voidaan asettaa aiheuttamaan ilmoitus, kun kello seuraavan kerran saavuttaa asetetun muistutusajan. Muistutuksesta huolehtiva säie käynnistetään, kun muistutus aktivoidaan, ja pysäytetään, kun muistutus tapahtuu.

8.8.1 Vastuualueet Muistutuselementti vastaa muistutuspiirteen toteutuksesta.

8.8.2 Luokkarakenne Muistutuselementti määrittelee rajapinnan *AlarmInterface*, jonka toteuttaa luokka *Alarm*. *Alarm* peritään *Thread*-luokasta, jotta voidaan aktiivisesti tarkkailla asetetun hälytyksen tilannetta. *AlarmEvent* on tapahtuma, joka aiheutuu, kun muistutus aktivoituu. Luokkien väliset suhteet on mallinnettu kuvassa 8.5.

8.8.3 Riippuvuudet Muistutuselementti on riippuvainen kellonaikaelementistä, jota se tarvitsee kellonajan saamiseksi.



Kuva 8.5: Muistutuselementin luokkarakenne

8.8.4 Konfiguraatio Muistutuselementtiä ei voi konfiguroida.

8.8.5 Rajapinnat Muistutuselementti määrittää *AlarmInterface*-rajapinnan, jonka kautta sitä voidaan asettaa, poistaa ja laittaa päälle. Hälytyksestä elementti ilmoittaa *AlarmEvent*-tapahtumalla.

8.9 Päivämääräelementti

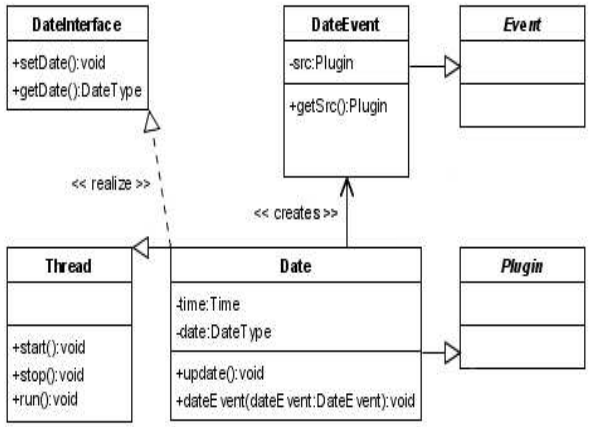
Päivämääräelementti toteuttaa päivämäärän toiminnallisuuden. Se julkaisee päivämäärätietoa. Päivämäärä pitää voida asettaa. Päivämäärästä huolehtiva säie on jatkuvasti päällä.

8.9.1 Vastuualueet Päivämääräelementti toteuttaa päivämääräpiirteen toiminnallisuuden.

8.9.2 Luokkarakenne Päivämääräelementti määrittää rajapinnan *DateInterface*, tapahtuman *DateEvent* ja luokan *Date*. *DateInterface* on elementin tarjoama rajapinta ulospäin, jonka *Date* toteuttaa. *Date* peritään myös yleisestä järjestelmän säieluokasta *Thread*, jotta se voi tarkkailla keskeytyksettä vuorokauden vaihtumista. *DateEvent* on tapahtuma, joka aiheutetaan, kun päivämäärä vaihtuu. Luokkien väliset suhteet on mallinnettu kuvassa 8.6.

8.9.3 Riippuvuudet Päivämääräelementti on riippuvainen aikaelementistä, jota se tarvitsee päivämäärän vaihtumisen havaitsemiseen.

8.9.4 Konfiguraatio Päivämääräelementtiä ei voi konfiguroida



Kuva 8.6: Päivämääräelementin luokkarakenne

8.9.5 Rajapinnat Päivämääräelementti määrittää *DateInteface*-rajapinnan. Tämän rajapinnan kautta päivämäärää voidaan tiedustella ja asettaa. Muuttumisestaan päivämääräelementti ilmoittaa *DateEvent*-tapahtumalla.

8.10 Ajanottoelementti

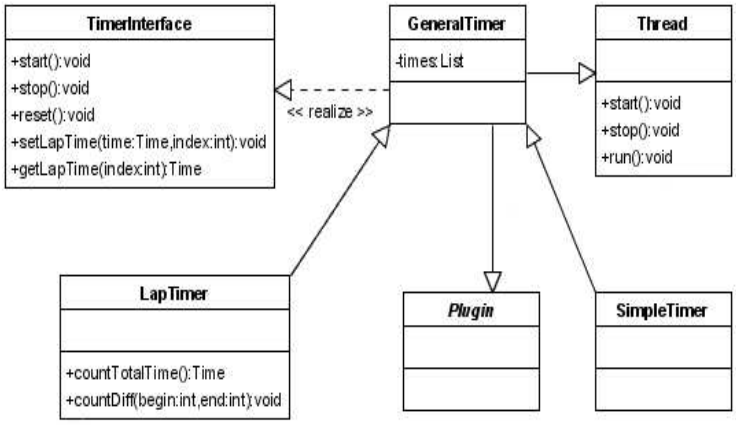
Ajanottoelementti toteuttaa ajanottotoiminnon. Elementillä voidaan mitata yhden suorituksen aikaa sekunnin sadasosan tarkkuudella. Mahdollisuus on myös 10 väliajan ottoon ja niiden tarkasteluun. Ajanotto pitää voida nollata. Ajanotosta huolehtiva säie käynnistetään, kun ajanotto aloitetaan, ja pysäytetään, kun ajanotto lopetetaan.

8.10.1 Vastuualueet Ajanotto vastaa ajanottopiirteen toteutuksesta.

8.10.2 Luokkarakenne Ajanottoelementissä määritellään rajapinta *TimerInterface*, jonka *GeneralTimer* toteuttaa. *GeneralTimer* peritään *Thread*-luokasta, jotta sen suoritus taustalla olisi mahdollista. *GeneralTimer*-luokasta peritään elementin variaatiot toteuttavat luokat *SimpleTimer* ja *LapTimer*. *SimpleTimer* on yksinkertaistettu ajanottoelementti yhden ajan mittaamiseen. *LapTimer*-luokka voi tallettaa 10 kierrosaikaa sekä tehdä laskutoimituksia niille. Luokkien väliset suhteet on mallinnettu kuvassa 8.7.

8.10.3 Riippuvuudet Ajanottoelementti ei ole riippuvainen muista elementeistä.

8.10.4 Konfiguraatio Ajanottoelementti voidaan konfiguroida toimimaan ilman väliaikoja tai 10 väliajan kanssa.



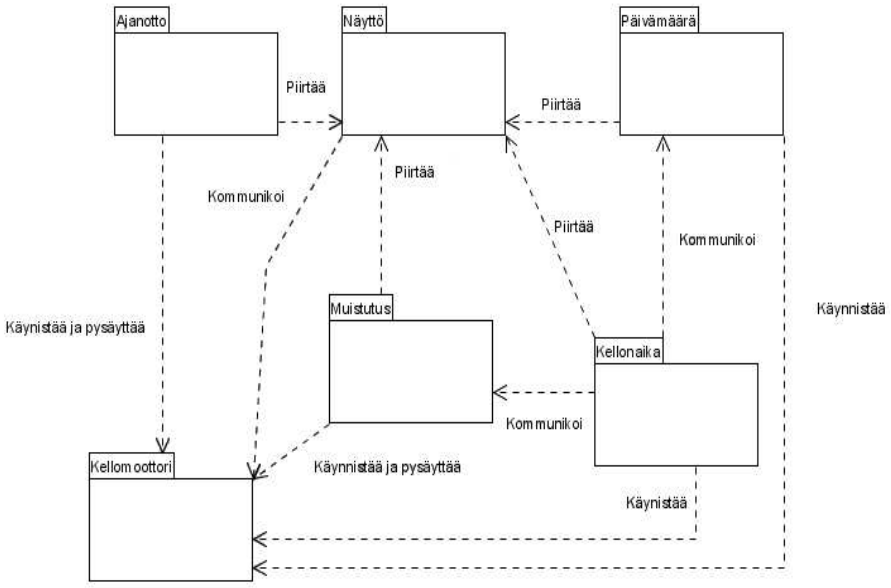
Kuva 8.7: Ajanottoelementin luokkarakenne

8.10.5 Rajapinnat Ajanottoelementti määrittelee *TimerInterface*-rajapinnan, jonka kautta sitä voidaan käyttää.

8.11 Yhteistoiminnan mallinnus

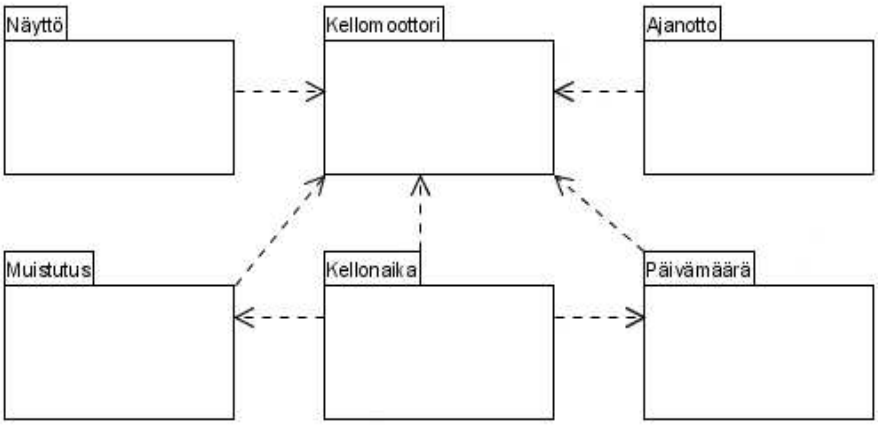
Elementit on nyt määritelty ja niiden yhteistoimintaa kuvattu. Perusajatuksena on elementtien käyttö rajapintojen kautta. Elementit ilmoittavat tapahtumien avulla toiminnoistaan. Kellomoottori kuuntelee kaikkia tapahtumia ja käsittelee ne asianmukaisesti. Elementtien välisistä riippuvuuksista johtuen myös piirteet toteuttavien elementtien välillä vallitsee suhteita. Elementtien yhteistoimintaa mallinnetaan ADD:n ohjeiden mukaisesti moduulinäkymällä, sijoitusnäkymällä ja rinnakkaisuusnäkymällä.

8.11.1 Rinnakkaisuusnäkymä Rannekello sisältää elementtejä, jotka suorittavat itseään omassa säikeessä. Tällaisia ovat päivämääräelementti, muistutuselementti ja kellonaikaelementti. Yhtäaikaiset toiminnot voivat aiheuttaa suoritusjärjestysongelmia, jonka vuoksi on järkevää mallintaa järjestelmän rinnakkaisia toimintoja. Tällä pyritään välttämään rinnakkaisuudesta aiheutuvia ongelmia. Rannekellosovelluksessa rinnakkaisia toimintoja on useita. Näytölle kertyy paljon piirtämistä mutta kuitenkin vain yhtä aktiivista näkymää päivitetään. Moottorin kuuntelemat tapahtumat aktivoivat toimintoja saapumisjärjestyksessä. Järkevää on, että elementtien tapahtumat ilmoittavat itsestään mutta eivät estä käyttöliittymän tapahtumia. Käyttäjän toimenpiteet ovat siis etusijalla suorituksessa. Rannekellotuotelinjan rinnakkaisuusnäkymä on mallinnettu kuvassa 8.8.



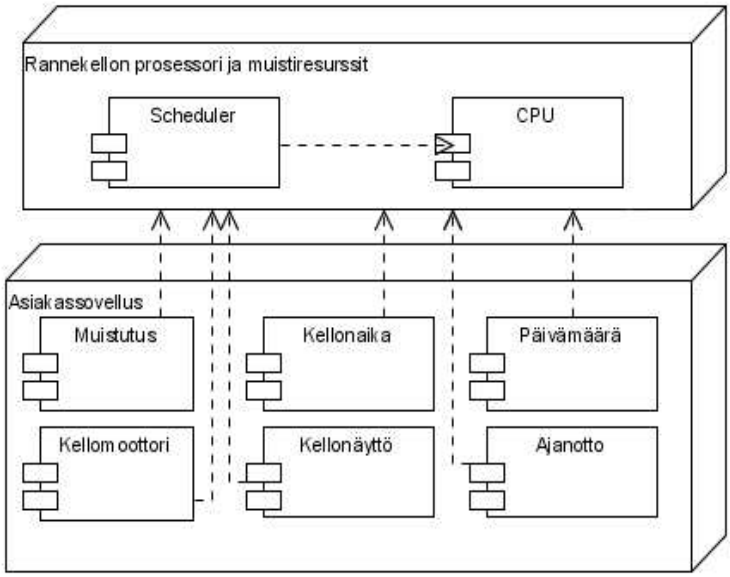
Kuva 8.8: Rannekelloarkkitehtuurin rinnakkaisuusnäkymä

8.11.2 Moduulinäkymä Moduulinäkymässä mallinnetaan rannekellotuotelinjan moduulit ja niiden väliset suhteet. Haluttuja suhteita ovat näytön ja moottorin välinen suhde sekä elementin ja moottorin väliset suhteet. Piirre-elementtien väliset suhteet eivät sinällään ole haitallisia vaan luonnollisia, mutta aiheuttavat monimutkaisuuden kasvua ja lisäävät suunnittelun aiheuttamaa kuormitusta. Lisäksi ne aiheuttavat joustavuuden vähentymistä sitomalla elementtejä toisiinsa. Rannekellotuotelinjan moduulit on mallinnettu kuvassa 8.9.



Kuva 8.9: Rannekelloarkkitehtuurin moduulinäkymä

8.11.3 Sijoitusnäkymä Sijoitusnäkymässä kuvataan rannekellon elementtien rautatason resurssikulutusta ja tietoliikennettä. Koska rannekellossa on yleensä vain yksi prosessori, on sijoitusnäkymä hyvin yksinkertainen. Kaikki säikeet saavat suoritusaikansa samalta prosessorilta, jonka täytyy huolehtia tärkeimpien prosessien katkeamattomuudesta. Tietoliikennettä rannekellossa ei ulkopuolelle tapahdu. Rannekellotuotelinjan sijoitusnäkymä on mallinnettu kuvassa 8.10.



Kuva 8.10: Rannekelloarkkitehtuurin sijoitusnäkymä

8.12 Suuntaviivoja toteutukselle

Toteutukseen liittyen voidaan kuvailla muutamia asioita, jotka tulee ottaa huomioon. Hahmotellaan yleisiä suunnittelupäätöksiä, jotka ovat järkeviä rannekellotuotelinjan kannalta.

Elementeistä pyritään luomaan yhtenäisiä. Tätä tavoitellaan perimällä elementit yhteisestä yliluokasta. Sama pätee tapahtumille. Tämä on järkevää valitulla sovellusalueella. Hajautetun arkkitehtuurin tapauksessa elementti voi olla paljon laajempi kokonaisuus kuten palvelin, jolla on oma arkkitehtuurinsa. Komponetteja on mahdollista vaihtaa dynaamisesti mutta tämä vaatii samalla moottorin käyttämän mallin vaihtoa. Turvallisempaa onkin määritellä sovellus jo käännösvaiheessa käyttäen Make-työkalua tai samantyyppistä ratkaisua (esim. Ant [27]). Näin saadaan määriteltyä toimiva sovellus. Lisäksi pitää olla säännöstö, jonka mukaan elementtejä voidaan luoda. Sen avulla on mahdollista tehdä staattisia tarkistuksia konfiguraation oikeellisuudesta.

Virran- ja muistinkulutus on sulautetuissa järjestelmissä merkittävä asia. Tämä voidaan huomioida esimerkiksi luomalla käytettävät elementit vasta ensimmäistä kertaa käytettäessä. Myös tuhoaminen, kun elementtiä ei käytetä, on mahdollista. Prosessien priorisointi virran ja muistin ollessa vähissä on järkevää. Jokaisen elementin tulee myös määrittää maksimaalinen muistin ja resurssien kulutus, jotta voidaan toimivalla tavalla ratkaista yhtäaikaisten toimintojen aiheuttamat ongelmat.

Toteutuksen osalta voidaan huomioida vielä sulautetuille järjestelmille soveltuvan reaaliaikakäyttöjärjestelmän olemassaolo. Esimerkiksi säikeet ovat käyttöjärjestelmän tukema ominaisuus, jonka oletetaan olevan käytössä. Käyttöjärjestelmän tarjoamia käytettyjä ominaisuuksia ovat myös grafiikkakirjastot, joiden avulla näytölle voidaan piirtää. Reaaliaikakäyttöjärjestelmän voidaan olettaa tarjoavan myös tukea muistin- ja virransäästölle. Rannekellotuotelinjan tuotteet tarvitsevat ehdottomasti tämän käyttöjärjestelmän tarjoaman kehyksen. Sitä ei kuitenkaan tarkemmin määritellä, koska sen merkitys erityisesti tuotelinja-arkkitehtuurin kannalta ei ole olennainen.

8.13 Huomioita rannekellotuotelinjasta

Kootaan lopuksi yhteen rannekellotuotelinjan määrittelyssä saavutettuja tuloksia. Määritellyn tuotelinjan perusteella tutkitaan ajattelutavan sovellusmahdollisuuksia, ongelmia ja hyötyjä.

8.13.1 Sovellusmahdollisuus Sovellusalueeltaan rannekello soveltuu hyvin tuotelinjalähestymistapaan. Rannekellot muistuttavat peruselementeiltään suuresti toisiaan. Tästä johtuen niiden toteutuksessa on paljon toisiaan muistuttavia kokonaisuuksia. Rannekellot myös eroavat toisistaan ominaisuuksiltaan, jolloin on tilaisuus hyödyntää tätä yhteneväisyyttä ja eroavaisuutta. Rannekellojen peruselementit ovat hyvin pitkälti samanlaiset. Tällöin varioituvuus saavutetaan tekemällä näistä elementeistä monipuolisesti konfiguroitavia. Näin saavutetaan ominaisuuksiltaan todella eroavia tuotteita. Jotta tämän hyödyntäminen olisi kannattavaa, on tuoteperheen koon oltava riittävän suuri ja markkinat riittävät. Rannekellojen tapauksessa eri ominaisuuksilla varustetuille tuotteille on olemassa kysyntää.

Abstraktimmalla tasolla tuotelinjan kannalta oleellisia piirteitä ovat riittävä samankaltaisuus ja eroavuus. Tarpeeksi yhtenevyyttä, jotta tuotetta voidaan käsitellä perheenä ja eroavaisuutta, jotta kyse ei ole vain yhdestä tuotteesta. Tuoteperheen optimaalinen koko on vaikeasti määriteltävissä. Mitä suurempi perhe on kyseessä, sitä enemmän ongelmia suunnittelussa ja toteutuksessa on odotettavissa. Liian pieni perhekoko taas johtaa mahdolliseen taloudelliseen kannattamattomuuteen, vaikka tuote olisi miten erinomainen. Tässä ongelmassa kokemus auttaa ratkaisun löytämisessä. Tuotelinjojen soveltuvuutta asiakasprojekteissa ei ole kiistetty, mutta

ajattelutapa tuntuu soveltuvan luonnostaan paremmin markkinoitaviin massatuotteisiin, joita ei erityisesti suunnitella tietyn organisaation toivomuksia huomioiden. Tällä tavoitellaan laajempia markkina-alueita ja vältetään samalla ohjelmakoodin oikeuksia koskevat ongelmat.

8.13.2 Useamman tuotteen monimutkaisuus Rannekellotuotelinjassa saavutettiin neljällä piirre-elementillä 12 erilaista tuotetta. Jotta tähän päästään, pitää elementtien välisiä suhteita tutkia ja ottaa ne myös huomioon. Rannekellotuotelinjassa ilmeni vain käyttö- ja riippuvuussuhteita, mutta jo näinkin pienen sovellusalueen kohdalla aiheuttivat konfiguraatiot ja riippuvuudet huomattavaa monimutkaisuutta ja suunnitteluprosessin kasvamista. Jos tuotelinjassa olisi ilmentynyt vielä poissulkevia suhteita, olisi monimutkaisuus lisääntynyt entisestään. Erityisesti piirre-elementtien väliset suhteet ovat ongelmallisia niiden aiheuttamien riippuvuuksien vuoksi. Ne heikentävät joustavuutta ja ylläpidettävyyttä, koska muutokset vaikuttavat myös toisten elementtien toimintaan. Tätä voidaan ehkäistä rajapintojen käytöllä mutta niiden muuttuessa seuraukset voivat levittäytyä ikävän laajalle. Toisaalta elementtien kokoaminen vielä pienemmistä palasista lisäisi edelleen uudelleenkäytettävyyttä. Näiden tekniikoiden käyttäminen kuitenkin lisää palasten välisiä suhteita ja monimutkaisuutta.

8.13.3 Arkkitehtuuri Rannekellotuotelinjalle suunniteltu arkkitehtuuri noudattaa tapahtumapohjaista arkkitehtuurityyliä. Tapahtumapohjainen arkkitehtuuri sopii vahvoilta ominaisuuksiltaan kohtuullisen hyvin yhteen valitun sovellusalueen kanssa. Erityisen hyvin valittu arkkitehtuurityyli soveltuu kellon toimintaperiaatteeseen ja joustavuuden toteutukseen. Koska tuotelinjat laajuudessaan ovat monimutkaisia kokonaisuuksia, on arkkitehtuuri erinomainen ylemmän tason abstraktio elementeistä ja niiden välisistä suhteista. Kun valittua tyyliä noudatetaan, johtaa se hyviin suunnittelupäätöksiin ja nopeuttaa itse asiassa suunnittelua. Elementit käyttäytyvät jossakin määrin samalla tavalla, ja sen vuoksi kertaalleen kehitettyjä ratkaisuja voidaan käyttää uudelleen. Tuloksena on johdonmukainen suunnitelma järjestelmästä. Eri asia on, miten helppoa yhtä arkkitehtuurityyliä on soveltaa koko järjestelmän suhteen. Kuitenkin jo tietynsuuntainen yhdenmukaisuus edesauttaa suunnitteluprosessia.

Arkkitehtuurin suunnittelussa joudutaan ottamaan huomioon usean tuotteen vaikutus. Tuoteperhe kuitenkin tavoittelee kokonaisuutena samankaltaisia laadullisia ominaisuuksia, jolloin arkkitehtuurin suunnittelu ei erityisesti monimutkaistu tuotelinjojen kohdalla. Tuotelinja-arkkitehtuurille variaatiopisteiden tunnistaminen on tosin eräänlainen lisätaakka. Arkkitehtuurityylin valinta on myös ongelmallinen. Vaikka tyylin ominaisuudet olisivatkin järjestelmälle optimaaliset, ei se yksin takaa valinnan onnistumista. Tietyntyyliset arkkitehtuurit eivät sovellu kaikkiin jär-

jestelmiin. Toiset ovat soveliaampia esimerkiksi interaktiivisiin sovelluksiin. Väärän tyylin soveltaminen voi johtaa toiminnallisuuden toteuttamiseen vaikeasti ja mahdolliseen arkkitehtuurin epäonnistumiseen. Kokemus luonnollisesti parantaa edellytyksiä päätyä oikeaan ratkaisuun.

8.13.4 Tulokset Rannekellon tuotelinjalla haettiin lähdekoodin uudelleenkäyttöä, ylläpidettävyyttä, tuotteiden laadukkuutta ja tuotannon tehostumista. Laadukkuus, lähdekoodin uudelleenkäyttö ja sen ylläpidettävyys otettiin huomioon arkkitehtuurin ja yhteisten elementtien avulla. Arkkitehtuuri suunniteltiin noudattaen valittua arkkitehtuurityyliä, joka oli implisiittinen kutsu. Valitun arkkitehtuurityylin avulla pyritään edesauttamaan haluttujen laatuattribuuttien toteutumista. Elementtien uudelleenkäytettävyys aikaansaatiin käyttämällä uudelleenkäyttöä tukevia menetelmiä kuten suunnittelumalleja ja olio-ohjelmointia. Yhteen elementtiin kapseloidaan näin useamman tuotteen käyttämiä ominaisuuksia, jotka jakavat toteutustaan. Tuotelinjalähestymistapaa soveltamalla muodostui kahdentoista sovelluksen perhe. Jo pienillä elementtimäärillä päästiin huomattavan suureen tuotemäärään. Ylläpitovaiheessa tuotelinjalla on hyvät edellytykset kehittyä, koska elementtejä kehittämällä kaikkien tuotteiden kehitys etenee. Arkkitehtuurilla tavoitellun joustavuuden ansiosta uusien piirteiden lisäys on mahdollista ilman suuria muutoksia ja uusia elementtejä varten on olemassa valmista ohjelmakoodia.

Vaikka tulokset ovat kannustavia, eivät ne ole ilmaisia. Suunnitteluvaihe tuntuu monimutkaistuvan ja kasvavan huomattavasti tuoteperheen jäsenten välisten suhteiden takia verrattuna perinteiseen yhden tuotteen ohjelmistoprojektiin. Useampi tuote aiheuttaa useampia riippuvuussuhteita. Myös dokumentoinnin määrän voidaan olettaa kasvavan tuotteiden lukumäärän ja monimutkaisuuden kasvaessa. Tuotteiden väliset suhteet on tallennettava huolellisesti. Tuotelinjaa perustettaessa on varmistuttava, että tuotelinjan elinkaari on riittävän pitkä, jotta perustamisen kustannukset saadaan muutettua hyödyksi. Jos tuotteet eivät pääse kehittymään ja syntyneitä elementtejä ja määrityksiä ei hyödynnetä täydessä mitassa, on tuotelinja melkoisen raskas operaatio. Tuotelinja-arkkitehtuuriin täytyy siis sitoutua pitkäksi aikaa, jotta sen hyödyt tulevat ilmi. Kehitysvaiheessa haasteiksi muodostunevat muutosten vaikutusten hallinta ja kehityksen järkevä suuntaus, mikä takaisi jatkuvuutta ja kasvattaisi näin hyötyjä.

9 Yhteenveto

Tuotelinjat ovat lupaava ja yleistyvä menetelmä tuottaa laadukkaita ja uudelleenkäytettäviä sovelluksia. Tuoteperheen yhteisten ominaisuuksien kautta avautuu uudelleenkäytön mahdollisuus. Tämän mahdollisuuden menestyksekäs hyödyntäminen johtaa parhaimmillaan koko tuotelinjan korkeaan laadukkuuteen. Tuoteperheen yhteinen arkkitehtuuri ja muunneltavat elementit ovat järkevä tapa hyödyntää tätä samankaltaisuutta. Arkkitehtuuri ohjaa kohti toivottuja laatuvaatimuksia sekä selkeyttää ja kasvattaa ymmärrystä järjestelmien rakenteesta ja toiminnasta. Elementit kapseloivat variaatiot sisäänsä ja tehostavat uudelleenkäytettävyyttä. Valmiin tuotelinjan laajentaminen uusilla tuotteilla on huomattavasti nopeampaa kuin vastavan rakentaminen tyhjältä pöydältä. Jatkossa tuotelinjoista kertyvä osaaminen on siirrettävissä uusiin perustettaviin tuotelinjoihin

Tuotelinjan perustaminen on raskas ja riskialtis operaatio. Vaikka lähestymistapaa on teollisuudessa sovellettu ja useita sovellusalueteknisiä menetelmiä kehitetty, on käytäntö vaihtelevaa. Monet prosessit on kehitetty alunperin sidoksissa tiettyyn sovellusalueeseen. Määritelty ja toimiva prosessi on kuitenkin elinehto tuotelinjan perustamisessa. Suuri piirteiden määrä aiheuttaa ongelmia. Kokonaisuus on liian laaja hallittavaksi ilman mallinnusta, ymmärrystä ja dokumentaatiota. Monen tuotteen aiheuttama monimutkaisuus johtaa suunnittelun raskauteen, joka kuluttaa aikaa ja resursseja. Tämän monimutkaisuuden käsittely on yksi ongelmista, jotka tuotelinjamenetelmien on hallittava tehokkaasti. Elementtien välisten suhteiden hallinta ja ylläpito vaativat järjestelmällistä työtä. Itsessään raskas suunnitteluprosessi aiheuttaa riskejä. Sovellusalueen valinta ja sen koko ovat vaikeita päätöksiä. Jos tuotelinja epäonnistuu tai tuoteperhe ei menesty, on taloudellinen menetys suuri. Tämän vuoksi tuotelinjalähestymistapaan on vaikea siirtyä ilman aiempaa kokemusta.

Yksikään ohjelmistoprojekti ei toisaalta ole riskitön. Vaikka menetelmä vaatii resursseja ja osaamista, ovat myös lähestymistavan tarjoamat mahdollisuudet houkuttelevia. Tuotelinjat vaativat sitoutumista pidemmälle aikavälille, jotta niiden mahdollistamat hyödyt voidaan lunastaa. Samalla sitoutuminen ohjaa kohti suunnitelmallisia toimintamalleja ja kestävää kehitystä. Tuotelinjat eivät ole kaikkiin projekteihin soveltuva lähestymistapa. Yrityksille, jotka toimivat tietyllä rajatulla toimialueella ja joiden tuotteet liittyvät vahvasti toisiinsa, on tuotelinjalähestymistapa järkevä vaihtoehto. Tällaisissa olosuhteissa saavutettavat hyödyt toteutuvat todennäköisimmin. Juuri sovellusalueen rajaus ja samankaltaisuuden ja erilaisuuden hyödyntäminen kestävällä ja tuottavalla tavalla ovatkin tuotelinjan avainkohdat.

10 Kirjallisuutta

[1] Christopher Alexander, Sara Ishikawa et al. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977. ISBN 0195019199.

[2] Michalis Anastasopoulos, Colin Atkinson and Dirk Muthig. "A Concrete Method for Developing and Applying Product Line Architectures". In "Net.Object Days", (pp. 294–312). 2002.
URL `http://citeseer.ist.psu.edu/551392.html`

[3] Guillermo Arango. "A brief introduction to domain analysis". In "Proceedings of the 1994 ACM symposium on Applied computing", (pp. 42–46). ACM Press, 1994. ISBN 0-89791-647-6.
URL `http://doi.acm.org/10.1145/326619.326656`

[4] Ken Arnold, James Gosling and David Holmes. *The Java Programming Language*. Addison-Wesley, 2003. ISBN 0-201-70433-1.

[5] Colin Atkinson, Joachim Bayer and Dirk Muthig. "Component-Based Product Line Development: The KobrA Approach". In P. Donohoe (ed.), "Proceedings of the First Software Product Line Conference", (pp. 289–309). 2000.
URL `http://citeseer.ist.psu.edu/409856.html`

[6] Len Bass, Paul Clements and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003. ISBN 0-321-15595-9.

[7] D. Batory. "Subjectivity and GenVoca Generators". In "4th International Conference on Software Reuse (ICSR '96)", (pp. 166–175). IEEE, 1996.
URL `http://citeseer.ist.psu.edu/batory96subjectivity.html`

[8] Don Batory, Rich Cardone and Yannis Smaragdakis. "Object-Oriented Frameworks and Product Lines". In P. Donohoe (ed.), "Proceedings of the First Software Product Line Conference", (pp. 227–247). 2000.
URL `http://citeseer.ist.psu.edu/batory00objectoriented.html`

[9] Don S. Batory. "Product-line architectures, aspects, and reuse (tutorial session)". In "International Conference on Software Engineering", (p. 832). 2000.
URL `http://citeseer.ist.psu.edu/142433.html`

[10] Joachim Bayer, Oliver Flege et al. "PuLSE: a methodology to develop software product lines". In "Proceedings of the 1999 symposium on Software reusability", (pp. 122–131). ACM Press, 1999. ISBN 1-58113-101-1.
URL http://doi.acm.org/10.1145/303008.303063

[11] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach.* Addison-Wesley, 2000. ISBN 0-201-67494-7.

[12] Felia Buchmann and Len Bass. "Introduction to the attribute driven design method". In "Proceedings of the 23rd international conference on Software engineering", (pp. 745–746). IEEE Computer Society, 2001. ISBN 0-7695-1050-7.
URL http://delivery.acm.org/10.1145/390000/381623/p745-bach%mann.pdf?key1=381623&key2=6609547701&coll=GUIDE&dl=GUIDE&CFID=13312950&CFTOKEN%=59360140

[13] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002. ISBN 0201703327.

[14] Digia (ed.). *Programming for the Series 60 Platform and Symbian OS*. Wiley, 2003. ISBN 0-470-84948-7.

[15] Alexandre Duret-Lutz and Thierry Géraud. *Improving Object-Oriented Generic Programming*. Tech. Rep. 0001, Laboratoire de Recherche et Développement de l'EPITA, Paris Sud, April 2000.
URL http://citeseer.ist.psu.edu/duret-lutz00improving.html

[16] Erich Gamma, Richard Hjelm, Ralph Johnson and John Vlissides. *Desig Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2001. ISBN 0201633612.

[17] David Garlan. "What is Style?" In "Proceedings of the Dagstuhl Workshop on Software Architecture", Saarbruecken, Germany, 1995.
URL http://www-2.cs.cmu.edu/publications/style-iwass95/

[18] —. "Software architecture: a roadmap". In "Proceedings of the conference on The future of Software engineering", (pp. 91–101). ACM Press, 2000. ISBN 1-58113-253-0.
URL http://doi.acm.org/10.1145/336512.336537

[19] —. "Software Architecture". In J. Marciniak (ed.), "Wiley Encyclopedia of Software Engineering", John Wiley & Sons, 2001.
URL http://www-2.cs.cmu.edu/~able/publications/encycSE2001/

[20] David Garlan and Mary Shaw. "An Introduction to Software Architecture". In V. Ambriola and G. Tortora (eds.), "Advances in Software Engineering and Knowledge Engineering", (pp. 1–39). Singapore: World Scientific Publishing Company, 1993.
URL http://www-2.cs.cmu.edu/~able/publications/intro_softar%ch/

[21] Jun Ginbayashi, Rieko Yamamoto and Keihi Hashimoto. "Business Component Framework and Modeling Method for Component-based Application Architecture". In "Proceedings of the fourth international Enterprise Distributed Object Computing Conference", (pp. 184–193). IEEE, 2000. ISBN 0-7695-0865.
URL http://ieeexplore.ieee.org/iel5/7075/19084/00882358.pdf%?tp=&arnumber=882358&isnumber=19084

[22] M. Griss, J. Favaro and M. d'Alessandro. "Integrating feature modeling with the RSEB". In "Proceedings of the Fifth International Conference on Software Reuse", (pp. 76–85). Vancouver, BC, Canada, 1998.
URL http://citeseer.ist.psu.edu/griss98integrating.html

[23] Martin L. Griss. "Implementing product-line features by composing component aspects". In P. Donohoe (ed.), "Proceedings of the First Software Product Line Conference", (pp. 271–288). 2000.
URL http://citeseer.ist.psu.edu/griss00implementing.html

[24] —. "Implementing product-line features with component reuse". In "International Conference on Software Reuse", (pp. 137–152). 2000.
URL http://citeseer.ist.psu.edu/328533.html

[25] —. "Product-Line Architectures". In T. Heineman and William Councill (eds.), "Component-Based Software Engineering", Addison-Wesley, 2001.
URL http://www.hpl.hp.com/reuse/papers/cbse-product-line.pd%f

[26] William Harrison and Harold Ossher. "Subject-oriented programming: a critique of pure objects". In "Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications", (pp. 411–428). ACM Press, 1993. ISBN 0-89791-587-9.
URL http://doi.acm.org/10.1145/165854.165932

[27] Erik Hatcher and Steve Loughran. *Java Development With Ant*. Manning Publications Company, 2002. ISBN 1930110588.

[28] Ann M. Hickey, Douglas L. Dean and Jay F. Nunamaker, Jr. "Establishing a foundation for collaborative scenario elicitation". *SIGMIS Database*, 30 (3-4),

pp. 92–110, 1999. ISSN 0095-0033.
URL http://doi.acm.org/10.1145/344241.344247

[29] "Institute for Experimental Software Engineering". http://www.iese.fhg.de/.

[30] Ralph E. Johnson. "Components, frameworks, patterns". In "Proceedings of the 1997 symposium on Software reusability", (pp. 10–17). ACM Press, 1997. ISBN 0-89791-945-9.
URL http://doi.acm.org/10.1145/258366.258378

[31] K.C. Kang, S.G. Cohen et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[32] Kyo C. Kang, Sajoong Kim et al. "FORM: A feature-oriented reuse method with domain-specific reference architectures". *Ann. Softw. Eng.*, 5, pp. 143–168, 1998. ISSN 1022-7091.
URL http://www.kluweronline.com/article.asp?PIPS=326181

[33] Rick Kazman. "Software Architecture". In S-K Chang (ed.), "Handbook of Software Engineering and Knowledge Engineering Volume 1", World Scientific Publishing, 2001.
URL http://citeseer.ist.psu.edu/466167.html

[34] Rick Kazman, Jai Asundi and Mark Klein. "Quantifying the costs and benefits of architectural decisions". In "Proceedings of the 23rd international conference on Software engineering", (pp. 297–306). IEEE Computer Society, 2001. ISBN 0-7695-1050-7.
URL http://portal.acm.org/ft_gateway.cfm?id=381504&type=pdf%&coll=GUIDE&dl=ACM&CFID=13312950&CFTOKEN=59360140

[35] Rick Kazman, Mario Barbacci et al. "Experience with performing architecture tradeoff analysis". In "Proceedings of the 21st international conference on Software engineering", (pp. 54–63). IEEE Computer Society Press, 1999. ISBN 1-58113-074-0.
URL http://portal.acm.org/ft_gateway.cfm?id=302452&type=pdf%&coll=GUIDE&dl=ACM&CFID=13312950&CFTOKEN=59360140

[36] Rick Kazman and Mark Klein. "Performing architecture tradeoff analysis". In "Proceedings of the third international workshop on Software architecture", (pp. 85–88). ACM Press, 1998. ISBN 1-58113-081-3.
URL http://doi.acm.org/10.1145/288408.288430

[37] Rick Kazman, Mark Klein and Paul Clements. *ATAM: Method for Architecture Evaluation*. Tech. Rep. CMU/SEI-2000-TR-004, Software Engineering Institute, Carnegie Mellon University, 2000.

[38] Gregor Kiczales, Erik Hilsdale et al. "Getting started with aspectj". *Commun. ACM*, 44 (10), pp. 59–65, 2001. ISSN 0001-0782.

[39] Gregor Kiczales, John Lamping et al. "Aspect-oriented programming". In Mehmet Akşit and Satoshi Matsuoka (eds.), "Proceedings European Conference on Object-Oriented Programming", vol. 1241, (pp. 220–242). Berlin, Heidelberg, and New York: Springer-Verlag, 1997.
URL http://citeseer.ist.psu.edu/kiczales97aspectoriented.ht%ml

[40] Philippe Kruchten. "The 4+1 View Model of Architecture". *IEEE Softw.*, 12 (6), pp. 42–50, 1995. ISSN 0740-7459.
URL http://www3.software.ibm.com/ibmdl/pub/software/rationa%l/web/whitepapers/2003/Pbk4p1.pdf

[41] C. Kuloor and A. Eberlein. "Aspect-oriented requirements engineering for software product lines". In "Engineering of Computer-Based Systems, 2003. Proceedings. 10th IEEE International Conference and Workshop.", (pp. 98–107). IEEE Computer Society Press, 2003.
URL http://ieeexplore.ieee.org/iel5/8501/26873/01194788.pdf%?tp=&arnumber=1194788&isnumber=26873

[42] Juha Kuusela and Juha Savolainen. "Requirements engineering for product families". In "Proceedings of the 22nd international conference on Software engineering", (pp. 61–69). ACM Press, 2000. ISBN 1-58113-206-9.
URL http://doi.acm.org/10.1145/337180.337189

[43] D. McIlroy. "Mass Producted Software Components". In "Software Engineering: Report on a Conference by the Nato Science Committee", (pp. 138–150). 1969.
URL http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato19%68.PDF

[44] Robert T. Monroe, Andrew Kompanek, Ralph Melton and David Garlan. "Architectural Styles, Design Patterns, and Objects". *IEEE Software*, 14 (1), pp. 43–52, January 1997.
URL http://www-2.cs.cmu.edu/~able/publications/ObjPatternsA%rch-ieee97/

[45] H. Muccini and A. van der Hoek. "Towards testing product line architectures". *Electronic Notes in Theoretical Computer Science*, 82 (6), 2003.
URL http://citeseer.ist.psu.edu/573837.html

[46] Musser and Stepanov. "Generic Programming". In "ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation (formerly SYMSAM, SYMSAC, EUROSAM, EUROCAL) (also sometimes in cooperation with the Symbolic and Algebraic Manipulation Groupe in Europe (SAME))", 1989.
URL http://citeseer.ist.psu.edu/musser88generic.html

[47] D. L. Parnas. "On the criteria to be used in decomposing systems into modules". *Commun. ACM*, 15 (12), pp. 1053–1058, 1972. ISSN 0001-0782.
URL http://doi.acm.org/10.1145/361598.361623

[48] —. "On the Design and Development of Program Families". *Transactions on Software Engineering*, 2 (16), pp. 1–9, 1976.

[49] "PuLSE flyer".
URL http://www.iese.fhg.de/PuLSE/pulse_e.pdf

[50] "Software Engineering Institute". http://www.sei.cmu.edu/sei-home.html.

[51] Mark A. Simos. "Organization domain modeling (ODM): formalizing the core domain modeling life cycle". In "Proceedings of the 1995 Symposium on Software reusability", (pp. 196–205). ACM Press, 1995. ISBN 0-89791-739-1.
URL http://doi.acm.org/10.1145/211782.211845

[52] Alan Snyder. "Encapsulation and inheritance in object-oriented programming languages". In "Conference proceedings on Object-oriented programming systems, languages and applications", (pp. 38–45). ACM Press, 1986. ISBN 0-89791-204-7.
URL http://doi.acm.org/10.1145/28697.28702

[53] Sergio Soares, Eduardo Laureano and Paulo Borba. "Implementing distribution and persistence aspects with AspectJ". In "Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications", (pp. 174–190). ACM Press, 2002. ISBN 1-58113-471-1.
URL http://doi.acm.org/10.1145/582419.582437

[54] A. A. Stepanov and M. Lee. *The Standard Template Library*. Tech. Rep. X3J16/94-0095, WG21/N0482, Hewlett-Packard Laboratories, 1994.
URL http://citeseer.ist.psu.edu/stepanov95standard.html

[55] Mikael Svahnberg and Jan Bosch. "Issues concerning variability in software product lines". *Lecture Notes in Computer Science*, 1951, pp. 146–??, 2001.
URL http://citeseer.ist.psu.edu/465223.html

[56] "Symbian". http://www.symbian.com/.

[57] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Approach*. Addison-Wesley, 1999. ISBN 0-201-69438-7.

[58] Jan Gerben Wijnstra. "Supporting diversity with component frameworks as architectural elements". In "Proceedings of the 22nd international conference on Software engineering", (pp. 51–60). ACM Press, 2000. ISBN 1-58113-206-9. URL http://doi.acm.org/10.1145/337180.337188

[59] Lloyd G. Williams and Connie U. Smith. "Performance evaluation of software architectures". In "Proceedings of the first international workshop on Software and performance", (pp. 164–177). ACM Press, 1998. ISBN 1-58113-060-0. URL http://doi.acm.org/10.1145/287318.287353