Milla Törhönen

Graafiset käyttöliittymät hajautetussa sovelluskehyksessä

Tietotekniikan
pro gradu -tutkielma
19. lokakuuta 2006

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Milla Törhönen
Yhteystiedot: milla.torhonen@patria.fi
Työn nimi: Graafiset käyttöliittymät hajautetussa sovelluskehyksessä
Title in English: Graphical User Interfaces in a Distributed Application Framework
Työ: Tietotekniikan pro gradu -tutkielma
Sivumäärä: 94
Tiivistelmä: Tutkielmassa käsitellään graafisten käyttöliittymien perusteita, joiden pohjalta etsitään eri ikkunointijärjestelmistä ja käytetyistä työkaluista riippumattomia yhteisiä piirteitä käyttöliittymien koostamisessa sekä kommunikoinnissa ohjelmalogiikan ja ikkunointijärjestelmän kanssa. Tavoitteena on kehittää joustava tapa yhdistää graafiset käyttöliittymät valmiiseen, hajautettuun sovelluskehykseen työkalusta ja ikkunointijärjestelmästä riippumattomalla tavalla. Sovelluskehyksessä, kuten myös siihen liitettävissä käyttöliittymissä tavoitellaan sovelluksen suunnittelun helppoutta ja ohjelmakoodin uudelleenkäytettävyyttä. Myös sovelluskehyksen hajautus heterogeeniselle alustalle on huomioitava siihen rakennettavia käyttöliittymiä suunniteltaessa. Ratkaisuna ongelmaan kehitetään kehykseen liitettävä, viestinvälityksen kautta järjestelmän ikkunapalvelimena toimiva komponentti.
English abstract: This thesis introduces basic principles of graphical user interfaces. Based on this discussion common features of composition and communication in graphical user interfaces are collected. This information is used to develop a way to combine graphical user interfaces to an existing, distributed application framework in a way that is independent of windowing systems and toolkits. The framework and its user interfaces aim to easiness of application development and re-use of previously developed components. Distribution of the framework in heterogenous environment must also be considered. As a solution to the problem a windowing component is developed to be added to the framework to serve other components of the application.
Avainsanat: Graafinen käyttöliittymä, GUI, hajautettu järjestelmä, sovelluskehys, Qt
Keywords: Graphical User Interface, GUI, distributed system, application framework, Qt

Sisältö

Sanasto

Dialogin riippumattomuus *(dialogue independence)* Käyttöliittymän ja ohjelmalogiikan irrotus toisistaan.

GUI *(graphical user interface)* Graafinen käyttöliittymä.

Hajautettu järjestelmä *(distributed system)* Kokoelma itsenäisiä tietokoneita, jotka näkyvät käyttäjälle yhtenä yhtenäisenä järjestelmänä.

Ikkuna *(window)* Sovelluksen graafisen käyttöliittymän sisältävä kehysikkuna.

Ikkunointijärjestelmä *(windowing system)* Käyttöjärjestelmän ja käyttöliittymien välissä oleva ohjelmisto, joka huolehtii prosessien ikkunoista, piirtää näytölle ja välittää käyttäjältä saapuneet viestit sovellukselle.

Komento Käyttöliittymäkomponentin päivitykseen käytettävä viesti.

Komponenttitapahtuma Käyttöliittymäkomponentin käyttäjän ja ikkunointijärjestelmän tapahtumista kokoama uusi viesti.

Käyttöliittymäkomponentti *(widget, control)* Graafisen käyttöliittymän komponentti, esimerkiksi painonappi (button). Sisältää komponentin piirron ja käyttäjän toimenpiteisiin reagoimisen.

Käyttöliittymästandardi *(look and feel)* Käyttöliittymän ulkoasu ja käyttäytyminen.

Ohjelmistokomponentti Sovelluksen rakenneyksikkö, jolla on yksiselitteisesti määritellyt rajapinnat, jotka se tarjoaa, vaatii ja joilla se konfiguroidaan.

Signaali Toteutuksessa käytetty termi käyttöliittymään liittyvälle, tapahtumia, komponenttitapahtumia ja komentoja sisältävälle viestille.

Sovelluskehys *(application framework)* Luokka-, komponentti- ja/tai rajapintakokoelma, joka toteuttaa jonkin sovellusjoukon yhteisen arkkitehtuurin ja perustoiminnallisuuden.

Tapahtuma *(event)* Käyttäjältä tai ikkunointijärjestelmältä saapunut viesti.

Työkalupakki *(toolkit)* Kirjasto tai sovelluskehys, joka sisältää käyttöliittymäkomponentteja sekä niiden toiminnallisuutta koostettuna yleiskäyttöiseen muotoon.

Viestijono *(event queue, message queue)* Ikkunointijärjestelmän ylläpitämä jono, johon talletetaan käyttäjältä ja järjestelmältä saapuneet viestit odottamaan käsittelyä.

Viestisilmukka *(event loop, message pump)* Graafisen käyttöliittymän perustana toimiva silmukka, joka poimii käyttäjältä saapuneita viestejä viestijonosta ja käsittelee ne yksi kerrallaan.

Visuaalinen formalismi *(visual formalism, VF)* Koostettu, laajempaa toiminnallisuutta sisältävä käyttöliittymäkomponentti.

1 Johdanto

Tutkimuksen tavoitteena on löytää hyvä tapa rakentaa graafisia käyttöliittymiä olemassa olevaan hajautettuun sovelluskehykseen. Koska kyseessä on sovelluskehys, on graafisten käyttöliittymien tekotavan sovelluttava useamman, ennalta määrittämättömän sovelluksen kehitykseen. Näin ollen toteutuksen on oltava joustava ja annettava varsinaisen sovelluksen kehittäjälle mahdollisuus vaikuttaa graafisen käyttöliittymän rakenteeseen ja toteutustapaan. Tutkimuksessa ei perehdytä graafisten käyttöliittymien suunnitteluun, vaan keskitytään tapaan, jolla ne saadaan liitettyä sovelluskehykseen siten, että käyttöliittymän suunnittelijan ei tarvitsisi keskittyä sovelluskehyksen asettamiin vaatimuksiin. Toteutuksen tulisi myös sallia erilaisten työkalujen käyttö graafisten käyttöliittymien suunnittelussa.

Jotta graafisista käyttöliittymistä saataisiin mahdollisimman kattava kuva, luvussa 2 perehdytään graafisten käyttöliittymien perusteisiin. Perusteita tutkittaessa pyritään löytämään kaikille oliopohjaisille graafisille käyttöliittymille yhteisiä piirteitä, sekä selvittämään kontrollin kulkua graafisen käyttöliittymän sisältävässä sovelluksessa. Löytyneiden yhteisten piirteiden myötä toteutuksessa voidaan keskittyä niihin ja erottaa työkalusta riippuvaiset osiot siten, että uusien työkalujen liittäminen sovelluskehykseen olisi mahdollista. Kontrollin kulun tunteminen puolestaan on tarpeellista yhdistettäessä toteutusta sovelluskehykseen.

Luvussa 3 käydään läpi hajautettuja järjestelmiä, sekä graafisiin käyttöliittymiin liittyviä ratkaisuja niissä. Näiden pohjalta pyritään löytämään ajatuksia omaan toteutukseen, jonka on toimittava sovelluskehyksen tarjoaman hajautuksen mukaisesti. Lisäksi tarkastellaan käyttöliittymän ja ohjelmalogiikan irrotuksen tuomia hyöty- ja haittapuolia, jotka hajautetussa järjestelmässä nousevat merkittävään rooliin eri osien välisten etäisyyksien kasvaessa.

Sekä graafisissa käyttöliittymissä että viestinvälitykseen perustuvissa hajautetuissa järjestelmissä tärkeään osaa nousevat viestit, joten luvussa 4 perehdytään graafisia käyttöliittymiä koskeviin viesteihin. Viestien käsittely ja nimeäminen riippuvat käytetystä työkalusta, joten ratkaisuissa on suuria eroja. Luvussa pyritäänkin löytämään työkalusta riippumaton, yleisen tason luokittelu ja terminologia erilaisille välitettäville viesteille.

Luvussa 5 perustellaan sovelluskehyksen käyttö ja esitellään kyseessä oleva hajautettu sovelluskehys. Esittelyn lisäksi pohditaan sovelluskehyksen graafisille käyttöliit-

tymille tuomia vaatimuksia.

Tutkimuksen pohjalta kehitetty toteutus esitellään luvussa 6. Toteutetun käyttöliittymäratkaisun lisäksi esitellään myös tapa, jolla käyttöliittymiä tullaan tulevaisuudessa rakentamaan sovelluskehystä erikoistettaessa. Varsinaisen sovelluksen kehitystä on havainnollistettu luvussa 7, jossa sovelluskehyksen pohjalta on toteutettu fiktiivinen autolaskurisovellus.

2 Graafiset käyttöliittymät

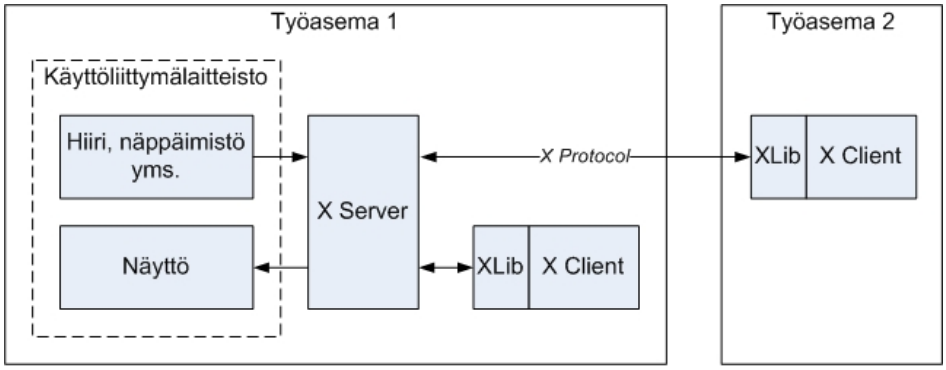
Graafinen käyttöliittymä (Graphical User Interface, GUI) on sovelluksen käyttöliittymä, jonka avulla käyttäjä kontrolloi sovelluksen toimintaa *osoitinlaitteiden*, kuten hiiren ja *syöttölaitteiden*, kuten näppäimistön avulla ohjeistaen sovellusta muuttamaan tilaansa. Graafinen käyttöliittymä puolestaan ilmaisee sovelluksen tilan käyttäjälle piirtämällä sen näytölle käyttäjän tarkasteltavaksi. [23, s. 89]

Tässä luvussa esitellään graafisiin käyttöliittymiin liittyviä peruskäsitteitä. Peruskäsitteet käydään läpi siten, että esille nostetaan kaikille käyttöliittymiä sisältäville oliopohjaisille järjestelmille yhteisiä piirteitä. Aluksi esitellään käyttöjärjestelmien yhteydessä käytetyt käsitteet: ikkunointijärjestelmät luvussa 2.1 ja käyttöliittymästandardit luvussa 2.2. Tämän jälkeen jatketaan graafisten käyttöliittymien ohjelmistonkehitykseen liittyvillä käsitteillä: käyttöliittymäkomponentit esitellään luvussa 2.3 ja niistä kootut työkalupakit luvussa 2.4. Lopussa luvussa 2.5 valaistaan graafisten käyttöliittymien viestien kulkua ikkunointijärjestelmästä työkalupakkien kautta sovellusohjelmaan ja takaisin.

2.1 Ikkunointijärjestelmät

Graafisen käyttöliittymän alimmalla tasolla toimii *ikkunointijärjestelmä (Windowing System)*. Ikkunointijärjestelmä huolehtii eri prosessien käyttöliittymäikkunoiden ylläpidosta, kuten siirtämisestä näytöllä, koon muuttamisesta ja päivittämisestä. Ikkunointijärjestelmä tarjoaa ohjelmistoille *ohjelmointirajapinnan (Application Programming Interface, API)*, jonka kautta voidaan esimerkiksi piirtää viiva pisteestä toiseen tai lisätä tekstiä tiettyyn kohtaan. Lisäksi ikkunointijärjestelmä huolehtii kommunikoinnista käyttäjän kanssa vastaanottamalla viestejä muun muassa näppäimistöltä ja hiireltä, sekä toimittamalla ne oikeille prosesseille. Oikea prosessi valitaan tapauskohtaisesti joko aktiivisena olevan ikkunan tai hiiren kursorin sijainnin perusteella. [22] [10, sivut 51-52]

Yhteisistä piirteistä huolimatta eri käyttöjärjestelmien ikkunointijärjestelmät on toteutettu eri tavoin ja ne tarjoavat eri tasoisia palveluita niitä käyttäville ohjelmistoille. Esimerkiksi *Microsoft Windows (MS Windows)* ja *Macintosh*-ikkunointijärjestelmät on toteutettu suoraan käyttöjärjestelmän yhteyteen ja ne tarjoavat rajapinnassaan hyvin korkeatasoisia piirto-ominaisuuksia. Tätä vastoin Unix-pohjaisissa käyttöjärjestelmissä



Kuva 2.1: X Server.

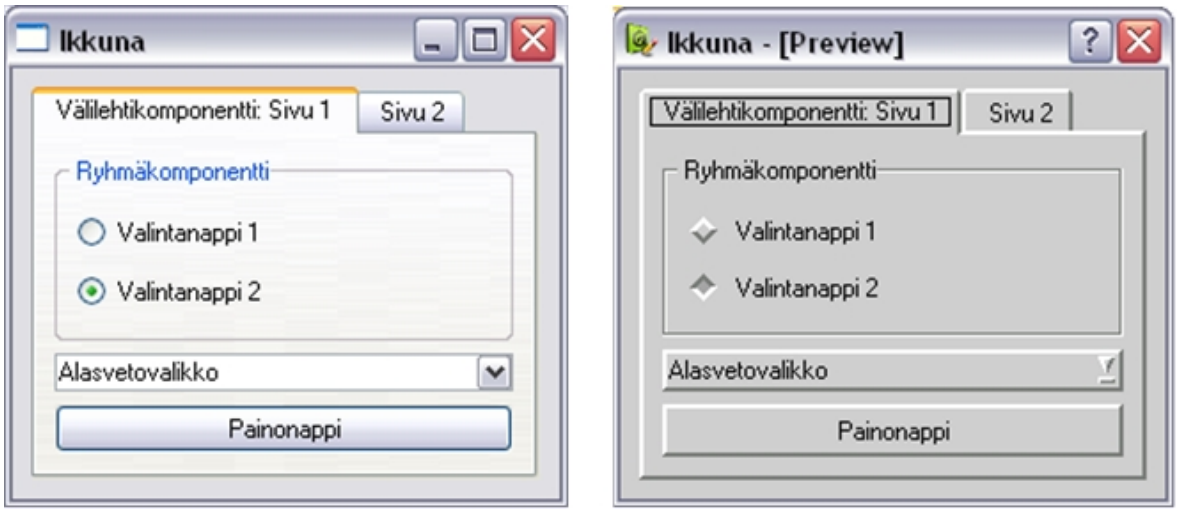
käytettävä *X-ikkunointijärjestelmä (X Window System)* [25] toimii käyttöjärjestelmästä irrallisena prosessina ja tarjoaa vain yksinkertaisimmat piirto-ominaisuudet. [22]

Seuraavassa on selvitetty tarkemmin X-ikkunointijärjestelmän toimintaa, sillä toteutus on poikkeuksellinen verrattuna käyttöjärjestelmiin liitettyihin ikkunointijärjestelmiin.

X-ikkunointijärjestelmän motivaationa on ollut Unix-pohjaisten käyttöjärjestelmien moninaisuus: erillisen prosessin avulla käyttöjärjestelmien rajapinnat saadaan piilotettua yhtenäisen ikkunointijärjestelmän taakse. X-ikkunointijärjestelmässä käyttöliittymää koskeva toiminnallisuus on jaettu lisäksi erillisille *ikkunamanagereille (Window Manager)*, mutta koska jako koskee vain X-ikkunointijärjestelmää, se on jätetty huomioimatta ja järjestelmästä käytetään kokonaisuuteen viittaavaa ikkunointijärjestelmätermiä. [22]

X-ikkunointijärjestelmässä ikkunapalvelimeen, eli *X Serveriin*, otetaan yhteys käyttäen *X-protokollaa* noudattavaa *XLib-kirjastoa* [16]. X Serverin tehtävänä on palvella varsinaisia sovelluksia, joista järjestelmässä käytetään nimitystä *X Client*. X Server sijoitetaan näytön, näppäimistön ja hiiren sisältävälle työasemalle, josta käsin se palvelee myös muilla työasemilla sijaitsevia X Client -sovelluksia tarjoamalla käyttöliittymälaitteistonsa näiden käyttöön. Se lähettää käyttäjältä tulleet komennot sovelluksille ja vastaanottaa niiltä näytön päivityskäskyt. X Serverin ja X Clientin rooleja on havainnollistettu kuvassa 2.1.

Erillisellä ikkunapalvelimella saavutetun joustavuuden haittapuolena myös samalla koneella sijaitsevien sovellusten näyttötoiminnot hidastuvat X-protokollan mukaisten viestien muodostuksen ja purkamisen sekä prosessien välisen kommunikaation takia.



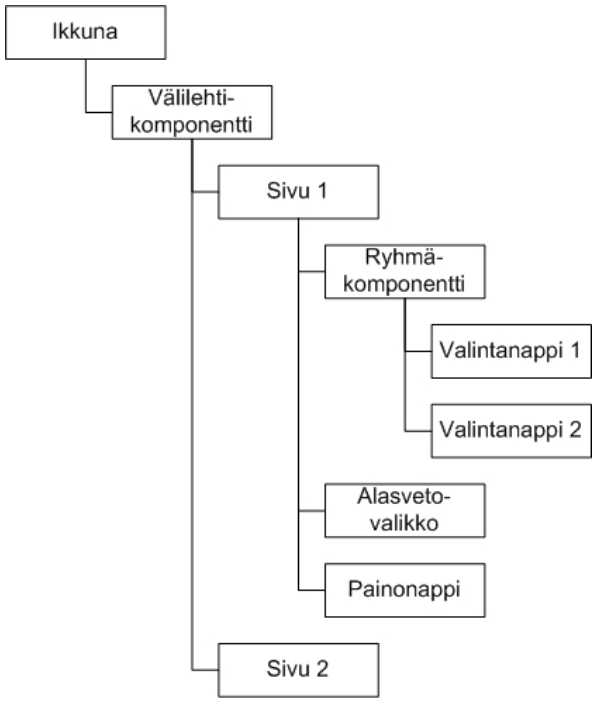
Kuva 2.2: Sama ikkuna esitettynä kahden käyttöliittymästandardin mukaan. Vasemmalla Windows XP, oikealla Motif.

2.2 Käyttöliittymästandardit

Käyttöliittymästandardit (look and feel) määrittelevät käyttöliittymän ulkonäön ja tyylin. MS Windows -ikkuna näyttää erilaiselta kuin Motif-käyttöliittymästandardin ikkuna X-ikkunointijärjestelmässä (kuva 2.2). Käyttöliittymästandardit määrittelevät myös tavan, jolla ikkuna kommunikoi käyttäjän kanssa. Esimerkiksi ikkunan aktivointiin riittää joissakin käyttöliittymästandardeissa pelkkä hiiren kursorin vienti ikkunan päälle, kun taas toisissa aktivointi vaatii hiiren painalluksen kyseisen ikkunan alueella. Käyttöliittymästandardit eivät ole sidottuja ikkunointijärjestelmiin, vaan käyttäjä voi esimerkiksi X-ikkunointijärjestelmässä valita useamman käyttöliittymästandardin välillä. [7, sivut 1-4] [22]

2.3 Käyttöliittymäkomponentit

Graafinen käyttöliittymä koostuu *käyttöliittymäkomponenteista (widget, control)*, joiden tehtävänä on yhdistää ikkunointijärjestelmiltä saadut viestit sovelluksen toimintoihin [22]. Käyttöliittymäkomponenttien avulla ikkuna jaetaan pienempiin osiin, joille annetaan vastuu kyseisen ikkunan osan käyttöliittymästandardin mukaisesta piirtämisestä sekä sitä koskevien viestien käsittelystä. Esimerkiksi hiiren painalluksesta sovellukselle saapuu viesti, jossa ilmoitetaan painalluksesta ja sen koordinaateista. Koordinaattien perusteella määritetään painalluksen vastaanottava käyttöliittymäkomponentti, jolle viesti annetaan käsiteltäväksi ja joka sen seurauksena saattaa esimerkiksi

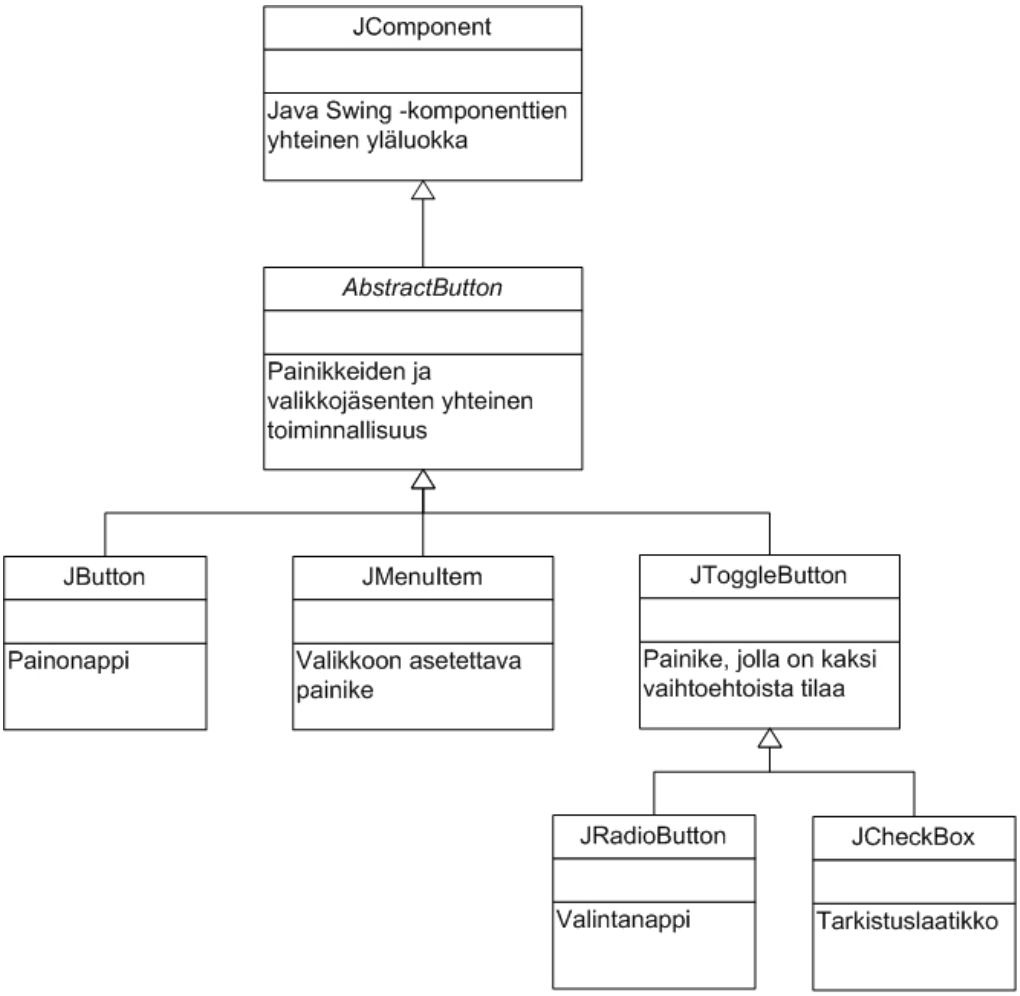


Kuva 2.3: Kuvan 2.2 käyttöliittymäkomponenttien muodostama puurakenne.

piirtää itsensä uudelleen. Esimerkkeinä yleisesti käytössä olevista käyttöliittymäkomponenteista voidaan mainita kuvassa 2.2 näkyvät *ryhmäkomponentti (group box)*, *valintanappi (radio button)*, *alasvetovalikko (combobox)* ja *painonappi (button)*.

Ikkunan osiin jakamisessa voidaan käyttää avuksi käyttöliittymäkomponentteja, joiden sisään voidaan asettaa muita käyttöliittymäkomponentteja. Näin jaettaessa käyttöliittymäkomponenteista syntyy puurakenne, jossa juurikomponenttina toimii ikkunakomponentti. Puurakenne jäsentää ikkunan pienempiin kokonaisuuksiin ja helpottaa siten monimutkaisten ikkunoiden hallintaa. Käyttöliittymäkomponentista, jonka sisään voidaan asetella muita käyttöliittymäkomponentteja, käytetään nimitystä *säiliökomponentti (container widget)*. Säiliökomponentti huolehtii lapsikomponenttiensa sijoittelusta ja piirtokäskyistä, esimerkiksi ikkunaa siirrettäessä siirtyvät myös kaikki sen sisällä olevat lapsikomponentit. Kuvan 2.2 ikkunassa säiliökomponentteja ovat mm. välilehtikomponentti ja ryhmäkomponentti. Ikkunan käyttöliittymäkomponenttien muodostama puurakenne on esitetty kokonaisuudessaan kuvassa 2.3.

Omia, uusia käyttöliittymäkomponentteja tekemällä pystytään tuottamaan sovelluskohtaista käyttöliittymän toiminnallisuutta. Tämä on sovelluskehityksen kannalta tärkeää, joten aihetta käsitellään tarkemmin luvussa 2.3.1. Omista komponenteista vielä pidemmälle viety ratkaisu ovat visuaaliset formalismit, joihin sisällytetään myös sovelluksen semantiikkaa. Visuaaliset formalismit esitellään luvussa 2.3.2.



Kuva 2.4: Painikkeiden muodostama luokkarakenne Java Swing -kirjastossa. [28]

2.3.1 Uudet käyttöliittymäkomponentit

Käyttöliittymäkomponentit on oliopohjaisissa työkaluissa usein rakennettu periyttämällä yhteisestä yläluokasta, jossa on toteutettuna kaikille käyttöliittymäkomponenteille yhteiset ominaisuudet, kuten sijainti vanhempaan nähden, koko ja siirto. Perintää on käytetty myös yhdistämään samantyyppisten käyttöliittymäkomponenttien yhteisiä ominaisuuksia; esimerkiksi erilaisten painikkeiden (painonappi, valintanappi) toteutukseen on yleensä käytetty yhteistä napin yläluokkaa. Näin ollen myös käyttöliittymäkomponenttien luokkien välinen hierarkia muodostaa puun, jonka juurena on kaikkien perimä yhteinen käyttöliittymäkomponentin yläluokka. Esimerkkinä kuvassa 2.4 on esitetty erilaisten painikkeiden muodostama puurakenne Java Swing -kirjastossa.

Uudet komponentit peritään työkalupakin valmiista komponenteista, jolloin voidaan keskittyä halutun uuden toiminnallisuuden lisäämiseen. On kuitenkin muistet-

tava, että käyttöliittymäkomponenttien yhteisen yläluokan takia moniperintä useammasta käyttöliittymäkomponentista on kielletty, joten useamman käyttöliittymäkomponentin yhdistäminen on aina tehtävä koostamalla.

Joissakin ohjelmointityökaluissa oman komponentin saa lisättyä työkalun komponenttivalikoimaan, jolloin käyttöliittymän suunnittelija voi käyttää sitä samaan tapaan kuin työkalupakin omia komponentteja. Monipuolisimmillaan komponenteille pystytään määrittelemään myös työkalussa suunnittelun aikana muokattavia ominaisuuksia.

2.3.2 Visuaaliset formalismit

Pidemmälle jalostettuja, toiminnallisuutta sisältäviä käyttöliittymäkomponenttikokonaisuuksia kutsutaan *visuaalisiksi formalismeiksi (Visual Formalisms, VFs)*. Esimerkkinä visuaalisista formalismeista esitellään Johnsonin [14] kehittämät älykkäät, semantiikasta tietoiset *valitsijat (Selectors)*.

Valitsijat ovat komponentteja, jotka ovat tietoisia käsittelemänsä datan tyypistä ja rajoituksista. Näiden tietojen perusteella ne valitsevat aina tilanteen mukaan käyttöön parhaiten soveltuvat käyttöliittymäkomponentit. Esimerkiksi se, käytetäänkö tekstimuuttujan arvon valitsemiseen valintanappeja vai alasvetovalikkoa, on riippuvainen mahdollisten vaihtoehtojen määrästä; kahden vaihtoehdon välillä valintanapit soveltuvat paremmin, kun taas kahdenkymmenen vaihtoehdon välillä alasvetovalikko on toimivampi ratkaisu. Valitsijat siirtävät käyttöliittymäkomponenttien valinnan ongelman pois käyttöliittymän suunnittelijalta, estävät tilanteeseen sopimattomien komponenttien käytön ja parantavat siten käyttöliittymien laatua. Ne myös ohjaavat suunnittelijaa keskittymään semantiikkaan eli siihen, *mitä* halutaan näyttää sen sijaan että keskityttäisiin siihen, *miten* se näytetään. [14]

Valitsijat ovat vain yksi esimerkki laajempaa toiminnallisuutta sisältävistä, yleiskäyttöisistä käyttöliittymäkomponenteista. Perinteisempänä esimerkkinä voidaan pitää joissakin työkalupakeissa (luku 2.4) valmiiksi toteutettua *tiedostonvalintadialogia (file selector dialog)*. Useamman käyttöliittymäkomponentin ja niiden semantiikan sisällyttäminen samaan komponenttiin on hyödyllistä tapauksissa, joissa toiminnan ja sitä kuvaavan näkymän halutaan aina olevan samanlaisia. Muutokset tällaisiin komponentteihin ovat kuitenkin työläitä ja heijastuvat kaikkiin käyttöliittymiin, joissa komponenttia on käytetty. Komponentin rajapinta muuttuu toiminnallisuuden lisääntyessä monimutkaisemmaksi ja sen ajattelematon muuttaminen rikkoo kaikki aiemmin tehdyt käyttöliittymät. Lisäksi useampien käyttöliittymästandardien tukeminen vaikeuttaa kompleksisemman komponentin kehitystä huomattavasti, esimerkkinä mainittakoon tiedostonvalintadialogin erilainen käyttäytyminen eri käyttöliittymästandardeissa.

2.4 Työkalupakit

Graafisten käyttöliittymien ohjelmistokehityksen avuksi on kehitetty useita erilaisia *työkalupakkeja (toolkit)*. Työkalupakit ovat käyttöliittymäkomponenteista koostuvia kirjastoja tai sovelluskehyksiä, jotka muodostavat kuorikerroksen ikkunointijärjestelmän yksinkertaisista piirtokäskyistä ja viestien vastaanotosta koostuvan ohjelmointirajapinnan päälle.

Nykyään graafisten käyttöliittymien ohjelmointiin käytetään lähes poikkeuksetta työkalupakkeja, jotka vähentävät ohjelmointiin tarvittavaa työmäärää huomattavasti. Esimerkkinä painonapin piirtäminen näytölle ilman työkalupakkia vaatisi ohjelmoijalta jokaisen yksittäisen pikselin värin määrittämisen kaikille painonapin eri tiloille. Piirtämisen lisäksi työkalupakkiin on sisällytetty myös toiminnallisuutta. Esimerkiksi painonapin tapauksessa työkalupakkiin on toteutettu painalluksen aiheuttama tilan ja ulkoasun muuttuminen normaalista alas painuneeksi ja takaisin normaaliksi.

2.4.1 Työkalupakit eri ikkunointijärjestelmissä

Eri ikkunointijärjestelmissä työkalupakki-käsite ymmärretään eri tavalla. Yksinkertaisemman rajapinnan omaavissa ikkunointijärjestelmissä työkalupakkeja käytetään ikkunointijärjestelmän päällä, kun taas monimutkaisemman ohjelmointirajapinnan tarjoavissa ikkunointijärjestelmissä työkalupakki tulee ikkunointijärjestelmän mukana. Jälkimmäisessä tapauksessa sovellukset ja myös itse ikkunointijärjestelmä voivat käyttää työkalupakin käyttöliittymäkomponentteja. Tarvittaessa sovellukset voidaan kuitenkin ohjelmoida käyttämään pelkästään alemman tason piirtokomentoja, joten myös tällaisten ikkunointijärjestelmien päälle voidaan ohjelmoida uusia työkalupakkeja. [22]

2.4.2 Ikkunointijärjestelmästä riippumattomat työkalupakit

Tätä nykyä on saatavilla myös ikkunointijärjestelmästä riippumattomia työkalupakkeja. Nämä voidaan jakaa kahteen ryhmään sen mukaan millä periaatteella ne toteuttavat riippumattomuuden:

1. Työkalupakit, jotka käyttävät piirtämiseen jotakin alustana toimivan ikkunointijärjestelmän omista, *natiiveista* työkalupakeista. Tätä lähestymistapaa käyttävät mm. *wxWidgets* [40] (ent. *wxWindows*), *XVT* [24] ja *Java AWT* [27]. Näiden työkalupakkien komponenttivalikoimat rajoittuvat kaikkien tuettujen alustojen natiivien työkalupakkien leikkaukseen. [22, 7]

2. Työkalupakit, jotka käyttävät suoraan alustana toimivan ikkunointijärjestelmän

alemman tason piirto-operaatioita, esimerkkinä *Qt* [37] ja *Java Swing* [29]. Näillä työkalupakeilla voidaan ikkuna näyttää minkä tahansa tuetun käyttöliittymästandardin mukaisena missä tahansa tuetussa ikkunointijärjestelmässä, esimerkiksi kuvassa 2.2 Motif-käyttöliittymästandardin mukainen vasemmanpuoleinen ikkuna on piirretty Windows XP -käyttöjärjestelmässä käyttäen Qt-työkalupakin esikatselua. [22, 7]

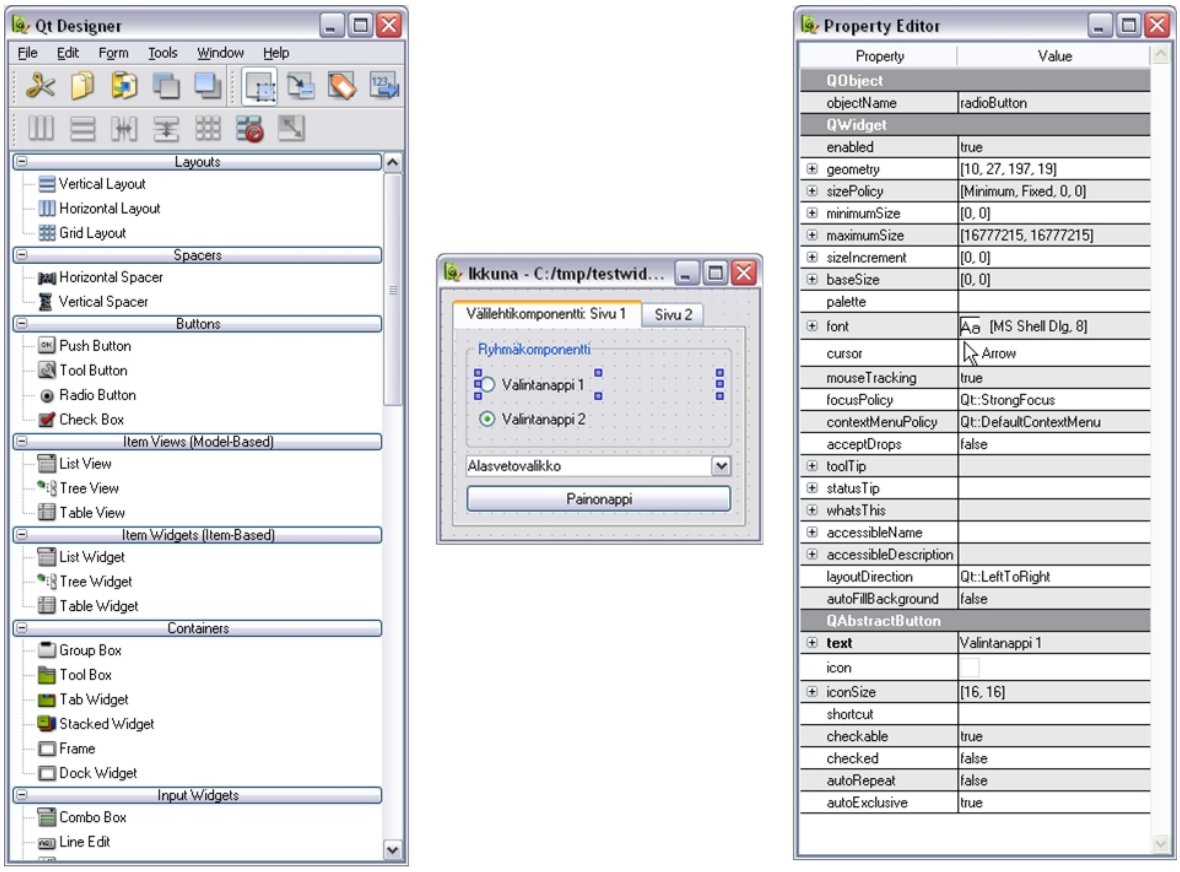
Eri toteutustapojen välinen vertailu on vaikeaa, sillä esimerkiksi nopeuteen vaikuttavat kulloinkin käytettävän natiivin työkalupakin ominaisuudet. Jos natiivi työkalupakki on integroitu syvälle ikkunointijärjestelmään ja optimoitu huolellisesti, sen käyttö välikerroksena nopeuttaa uuden työkalupakin toimintaa. Jos taas natiivi työkalupakki on erillään ikkunointijärjestelmästä, sen käyttö saattaa vaikuttaa hidastavasti. Uuden työkalupakin kehitykseen tarvittava työmäärä puolestaan riippuu siitä, kuinka hyvin sen toimintaperiaatteet käyvät yksiin tuettavien natiivien työkalupakkien kanssa. Jos esimerkiksi kontrollin käsittelyssä on suuria eroja, voi olla yksinkertaisempaa kehittää uusi työkalupakki siten, ettei se käytä natiivia työkalupakkia lainkaan.

2.4.3 Työkalupakkien käyttöä helpottavat ohjelmointityökalut

Erilaisten työkalupakkien käytön avuksi on kehitetty ohjelmointityökaluja. Työkalun toimittajasta riippuen ohjelmointityökaluista käytetään eri nimikkeitä, kuten *Rapid Prototyper*, *User Interface Builder*, *User Interface Management System (UIMS)*, *User Interface Development Environment (UIDE)* tai *Rapid Application Developer (RAD)*. Esimerkkeinä ohjelmointityökaluista voidaan mainita MFC-työkalupakkia käyttävä *Visual Studio* ja alustariippumattoman Qt-työkalupakin suunnitteluohjelma *Qt Designer* (kuva 2.5).

Ohjelmointityökalun avulla työkalupakin komponentteja voidaan asetella suunnitteilla olevaan ikkunaan sekä määritellä arvoja niiden *ominaisuuksille (properties)*, kuten esimerkiksi painonapissa näytettävälle tekstille. Käyttöliittymäkomponenttien asettelun helpottuminen nopeuttaa käyttöliittymän prototyyppien kehitystä ja testausta, ja parantaa siten käyttöliittymien laatua.

Ohjelmointityökaluissa on usein piilotettu tapa, jolla ohjelma reagoi käyttäjän toimenpiteisiin. Yleisin menetelmä perustuu *takaisinkutsuihin (callback)*, joissa reaktio määritellään funktio-osoittimen avulla. Takaisinkutsun luominen voidaan toteuttaa työkalussa esimerkiksi siten, että käyttöliittymäkomponenttia painamalla ohjelmoija saa täytettäväkseen syntyneen funktion rungon, josta työkalu luo automaattisesti takaisinkutsun. Työkalu helpottaa käyttöliittymän ohjelmointia, mutta viestien todellisesta käsittelytavasta tietämätön ohjelmoija voi pahimmassa tapauksessa luoda hitaan,



Kuva 2.5: Qt Designer.

helposti jumiutuvan tai jopa kaatuvan käyttöliittymän. Viestien käsittely esitellään tarkemmin luvussa 2.5.

2.5 Tapahtumapohjainen viestien käsittely

Komentorivisovelluksissa käyttöliittymän osuus on liitetty muun toiminnallisuuden lomaan siten, että sovellus kysyy käyttäjältä syötteen, käsittelee saamansa syötteen ja toimii sen mukaisesti, jonka jälkeen sovellus kysyy uuden syötteen. Koko tapahtumaketjun ajan kontrolli on sovelluksella, eli käyttäjä pystyy syöttämään komentoja sovellukseen vain silloin, kun sovellus kysyy niitä. [23, s. 89]

Graafisessa käyttöliittymässä roolit on vaihdettu: kontrollista vastaa käyttäjä, joka voi antaa valitsemiaan syötteitä sovelluksen tilasta riippumatta. Graafista käyttöliittymää kutsutaan *tapahtumapohjaiseksi (event based, event-driven)* järjestelmäksi, joka perustuu siihen, että sovellus on koko ajan valmiina ottamaan vastaan käyttäjän syötteitä. [23, s. 105-109]

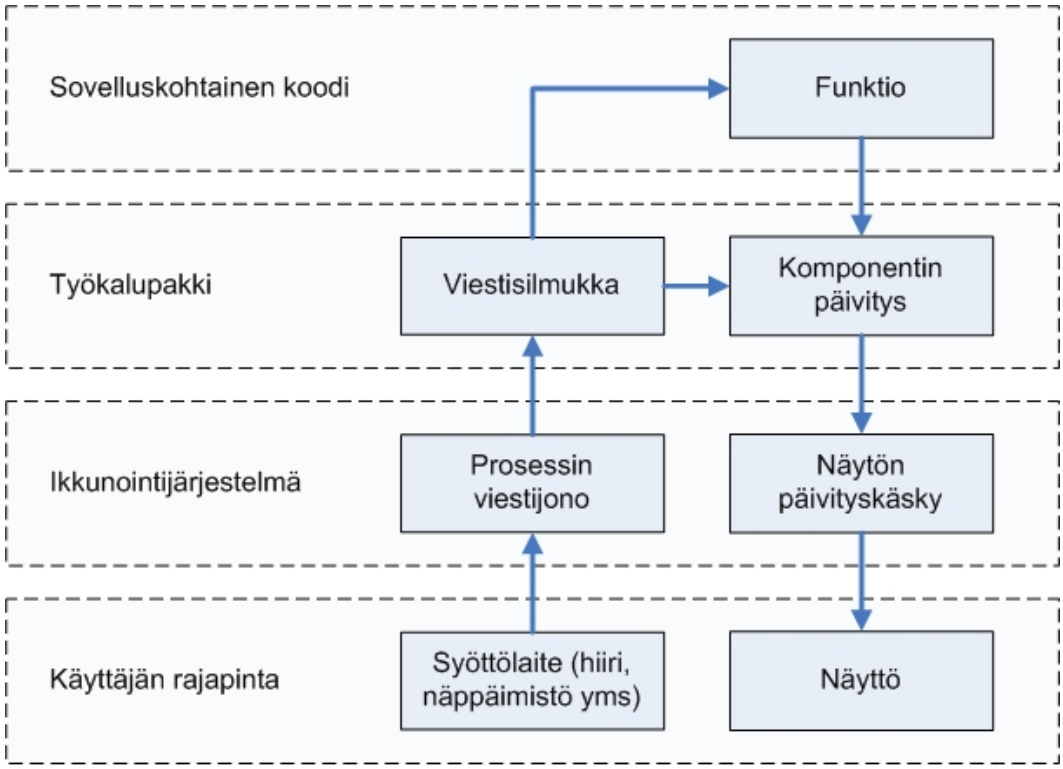
Käytännössä toiminnallisuus toteutetaan käyttämällä *viestisilmukkaa (event loop, message pump)*, jonka tehtävänä on odottaa käyttäjän syötettä ja käynnistää siihen soveltuva reaktio. Viestisilmukka ja sen myötä koko graafisen käyttöliittymän perustoiminnallisuus voidaan yksinkertaistaa seuraavalla tavalla:

```
alustus();
while (!lopetus) // Viestisilmukka
{
   Viesti v = haeSeuraavaViesti();
   kasitteleViesti(v);
}
lopetus();
```

Käyttäjän syötteet toimitetaan sovellukseen ikkunointijärjestelmän ylläpitämän *viestijonon (event/message queue)* avulla. Ikkunointijärjestelmä lisää viestejä viestijonoon, josta viestisilmukka hakee ne sovelluksen käsiteltäviksi. Viestijonon toiminnan periaate on sama kaikissa ikkunointijärjestelmissä, eroavaisuuksia on vain siinä, onko viestijono prosessikohtainen, kuten X-ikkunointijärjestelmässä, vai ikkunakohtainen, kuten MS Windowsissa. [23, s. 106]

Kuvassa 2.6 on esitetty viestijonon ja viestisilmukan roolit suhteessa ikkunointijärjestelmään, työkalupakkiin ja sovellusohjelmaan. Ilman työkalupakkia ohjelmoitaessa viestisilmukan ylläpito ja komponentin päivitys toteutettaisiin sovelluskohtaisessa koodissa. Käytännössä varsinaisen sovelluksen muodostavat kuvan kaksi ylintä kerrosta, sillä työkalupakki on sovelluksen käytössä joko kirjastona tai sovelluskehyksenä. Viestisilmukka sijaitsee näin ollen sovelluksessa. Sovelluksen kehittäjän harteille jää tällöin vastuu siitä, että viestisilmukan olemassaolo huomioidaan, eikä sitä lukita muulla toiminnalla. Seuraavassa on esitetty kaksi erilaista, varsin tyypillistä konfliktitilannetta, jotka aiheutuvat siitä ettei sovelluksen kehittäjä huomioi käyttöliittymän perustoiminnallisuuden sisältymistä sovellukseen:

1. Muun toiminnallisuuden toteuttaminen käyttöliittymäsäikeessä pysäyttää viestisilmukan läpikäynnin. Jos toiminta kestää kauan, käyttöliittymä jumiutuu ja lakkaa sekä piirtymästä näytölle että reagoimasta käyttäjän toimenpiteisiin. Esimerkkinä voidaan mainita tapaus, jossa täytettävä funktiorunko avautuu kehitystyökalussa ikkunaan siirrettyä painonappia painamalla. Kehittäjä ohjelmoi funktiorunkoon laskentasilmukan, joka päivittää arvoja käyttöliittymässä. Käytännössä funktio yhdistetään takaisinkutsuksi, johon käyttöliittymäsäikeen ajo siirtyy painonappia painettaessa. Näin ollen käyttöliittymäsäie suorittaa laskentasilmukan, jonka aikana viestisilmukka pysähtyy ja käyttöliittymän päivityspyynnöt



Kuva 2.6: Tapahtumapohjainen viestien käsittely graafisessa käyttöliittymässä.

jäävät käsittelemättä. Tässä tilanteessa esimerkiksi toisen ikkunan käyttäminen käyttöliittymän päällä jättää koko käyttöliittymän harmaaksi.

2. Käyttöliittymäkomponenttien päivittäminen muista säikeistä käsin saattaa aiheuttaa kilpailutilanteita, sillä käyttöliittymäkomponentit eivät yleensä ole säieturvallisia. Tällainen tilanne aiheutuu esimerkiksi sovelluksen laskentaa tekevän säikeen päivittäessä tekstikenttää samanaikaisesti käyttäjän tai toisen säikeen kanssa. Kilpailutilanteen tulos ei ole ennakoitavissa, ja tuloksena on arvaamattomasti toimiva sovellus. Kuvassa 2.6 tilanne on yksinkertaistuksen takia juuri tällainen. Jos kuva piirrettäiseen oikeaoppisesti, päivityspyyntö menisi eri käyttöliittymäsäikeen ulkopuolelta kutsuttaessa sovellusohjelman funktiosta ikkunointijärjestelmän viestijonoon, josta viesti siirtyisi viestisilmukan kautta komponentin päivitykseen.

3 GUI-arkkitehtuurit hajautetuissa järjestelmissä

Tässä luvussa tarkastellaan graafisten käyttöliittymien arkkitehtuureita hajautetuissa järjestelmissä. Hajautetut järjestelmät määritellään yleisellä tasolla luvussa 3.1, jossa esitellään myös hajautettujen järjestelmien perustavoitteet. Yleisintä hajautetuissa järjestelmissä käytetty arkkitehtuuria, eli Asiakas-Palvelin -arkkitehtuuria, sekä käyttöliittymien roolia siinä tarkastellaan lyhyesti luvussa 3.2.

Tutkimuksen kannalta oleellinen viestinvälitysarkkitehtuuri esitellään luvussa 3.3, jossa tarkastellaan lisäksi lähemmin kahta viestinvälitykseen perustuvaa hajautettua arkkitehtuuria, joissa käyttöliittymien käsittely on otettu erityisesti huomioon. Olemassaolevia toteutuksia tarkastelemalla pyritään löytämään ajatus- ja mahdollisesti myös toteutusmalleja, joita voidaan hyödyntää omaa toteutusta suunniteltaessa.

Lopulta luvussa 3.4 tutkitaan käyttöliittymän ja ohjelmalogiikan irrottamista toisistaan yleisellä tasolla sekä sen vaikutuksia käyttöliittymien suunnitteluun. Irrotuksen vaikutukset on otettava huomioon varsinkin hajautetussa järjestelmässä, jossa eri osien väliset etäisyydet ja kommunikointiin kuluva aika saattavat kasvaa huomattavasti.

3.1 Hajautetut järjestelmät

> *A distributed system is a collection on independent computers that appears to its users as a single coherent system.*
> *(Hajautettu järjestelmä on kokoelma itsenäisiä tietokoneita, jotka näkyvät käyttäjälle yhtenä yhtenäisenä järjestelmänä.)*

Määritelmä on peräisin Tanenbaumilta ja van Steeniltä [31]. Sen mukaan hajautettu järjestelmä jakaantuu teknisessä mielessä usealle erilliselle koneelle, mutta on ohjelmiston avulla toteutettu siten, että käyttäjä näkee vain yhden yhtenäisen järjestelmän.

Hajautetuilla järjestelmillä on yleisesti neljä tavoitetta [31, luku 1.2], jotka vaikuttavat niiden yhteisiin piirteisiin:

1. *Käyttäjien ja resurssien yhdistäminen (connecting users and resources)* on hajautettujen järjestelmien päätavoite. Verkon eri puolilla toimivien käyttäjien tulee kyetä jakamaan yhteisiä resursseja, kuten esimerkiksi tulostimia, tiedostoja tai tietokantoja.

2. *Läpinäkyvyys (transparency)* tarkoittaa sitä, että hajautettu järjestelmä piilottaa hajautuksen käyttäjiltä. Läpinäkyvässä järjestelmässä ohjelmistot voivat kommunikoida keskenään tietämättä sijaitseeko toinen osapuoli fyysisesti eri koneella vai ei. Tästä seuraa myös se, että käyttäjät voivat kommunikoida järjestelmän kanssa aina samalla tavalla sijainnista riippumatta. Laitteiston heterogeenisyyden piilotus hoidetaan useimmiten *väliohjelmiston (middleware)* avulla.

3. *Avoimuus (openness)* tarkoittaa sitä, että hajautetun järjestelmän eri osien rajapinnat on tarkoin määritelty kaikille osille yhteisellä tavalla sekä syntaktisessa että semanttisessa mielessä. Syntaktinen rajapinnan määritelmä luodaan usein jonkin rajapintakielen avulla, mutta semanttinen rajapinta eli se, *mitä* rajapintaa käytettäessä tapahtuu, määritellään useimmiten vain sanallisella kuvauksella.

4. *Skaalautuvuutta (scalability)* voidaan vaatia ainakin kolmessa eri dimensiossa: skaalautumista koon, etäisyyksien ja ylläpidon suhteen. Esimerkiksi koon suhteen skaalautuvaan järjestelmään voidaan lisätä uusia resursseja, kuten palvelimia, mahdollisimman pienillä muutoksilla. Valitettavasti skaalautuvuuden lisääminen huonontaa aina suorituskykyä jossain määrin.

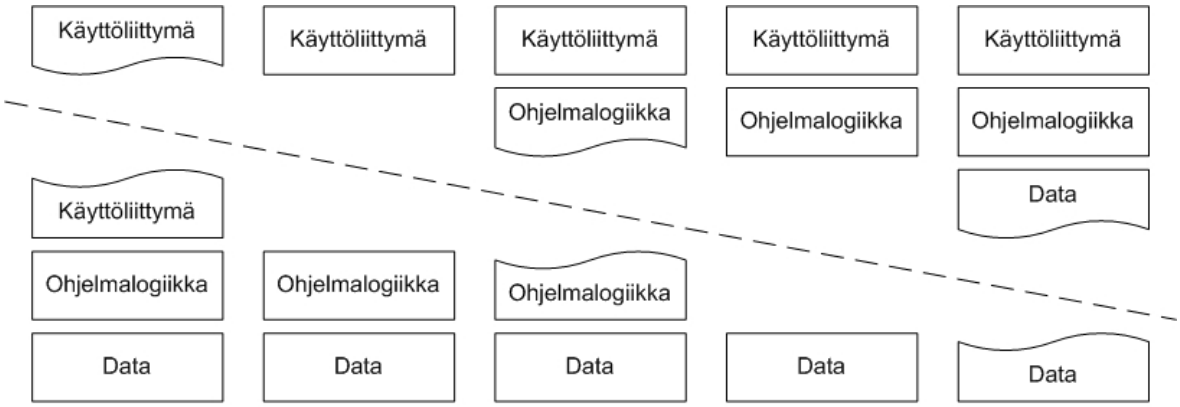
3.2 Asiakas-Palvelin -arkkitehtuuri

Asiakas-Palvelin (Client-Server) -arkkitehtuuri on yleisin hajautetuissa järjestelmissä käytetty arkkitehtuuri, joka perustuu siihen, että *asiakasohjelma (client)* pyytää operaatioita tai *palveluita (service)*, jotka *palvelinohjelma (server)* suorittaa. Vastaanotettuaan asiakkaan pyynnön palvelinohjelma suorittaa pyydetyn palvelun ja palauttaa tulokset asiakkaalle. Palvelinohjelman *asiakasrajapinta (client interface)* määrittää palvelinohjelman tarjoamat palvelut ja operaatiot, joihin asiakkaan pyyntöjen on rajoituttava. [1]

Asiakas-Palvelin -arkkitehtuurissa ohjelmiston hajautuksen taso on vapaasti valittavissa ja esimerkiksi useamman asiakkaan käyttö saman palvelimen yhteydessä on yleistä. Ongelmaksi koituu kuitenkin selkeä rajan veto asiakkaan ja palvelimen välille. Usein palvelin saattaakin toimia myös asiakkaana toiselle palvelimelle. Varsinkin tietokantoihin perustuvissa järjestelmissä käytetään yleisimmin 3-kerrosarkkitehtuuria, jossa ohjelmisto hajautetaan kolmeen loogiseen osaan:

1. *Käyttöliittymäkerrokseen* sijoitetaan kommunikointi käyttäjän kanssa.

2. *Ohjelmalogiikkakerros* sisältää varsinaisen sovelluksen logiikan.



Kuva 3.1: Loogiset kerrokset jaettuna katkoviivan yläpuolella sijaitsevan asiakkaan ja alapuolella sijaitsevan palvelimen kesken.

3. *Datakerros* käsittää yhteyden tietokantaan tai johonkin muuhun pysyvään datalähteeseen.

Loogisten kerrosten jako asiakas- ja palvelinohjelmien välillä voidaan tehdä usealla tavalla. Kuvassa 3.1 esitetään erilaiset vaihtoehdot kerrosten jakamiseksi. Katkoviivan yläpuolinen osa kuvaa asiakasta ja alapuolinen osa palvelinta. Vasemmanpuolimmaisessa versiossa osa käyttöliittymäkerroksesta on sijoitettu palvelimen puolelle, joten asiakas on pelkkä terminaali. Oikeanpuolimmaisessa versiossa jako on puolestaan viety toiseen ääripäähän sijoittamalla osa datan käsittelystä asiakkaaseen ja palvelimelle on sijoitettu pelkkä datavarasto. [31, luku 1.5]

Vastaavaan kerrosten jaon ongelmaan törmätään aina, kun ohjelmistoa hajautetaan useampaan eri komponenttiin tai useammalle eri koneelle. Sovelluskehyksen tapauksessa ongelma on vielä tavallisen sovelluksen tapausta monimutkaisempi, sillä ratkaisua ei voida sitoa yhteen sovellukseen vaan sen on sovelluttava kaikkiin tuoteperheen sovelluksiin.

3.3 Viestinvälitysarkkitehtuuri

Viestinvälitysarkkitehtuurissa (mm. message dispatcher architecture) joukko komponentteja kommunikoi asynkronisesti keskenään keskitetyn viestinvälittäjän/väylän kautta. Keskeisenä erona Asiakas-Palvelin -arkkitehtuuriin on se, ettei komponenttien rooleja ole kiinnitetty. Viestinvälitysarkkitehtuuriin voidaan lisätä uusia komponentteja järjestelmää muuttamatta, joten se tarjoaa erittäin joustavan tavan toteuttaa sovelluksia. [17, sivut 139-141]

Komponentin rakenteellinen rajapinta viestinvälitykseen on usein staattinen, sillä se

on sidottu viestinvälityksessä kuljetettavien viestien ennaltamäärättyyn rakenteeseen. Käytännössä rajapinta on kuitenkin hyvin dynaaminen, sillä viestin sisältö ratkaisee komponentin reaktion. Näin ollen sisällöltään uudentyyppinen viesti ei muuta järjestelmän staattista rakennetta, mutta uuden viestin käsittelyyn kykenevä komponentti voi lisätä järjestelmään uutta toiminnallisuutta. [17, sivut 139-141]

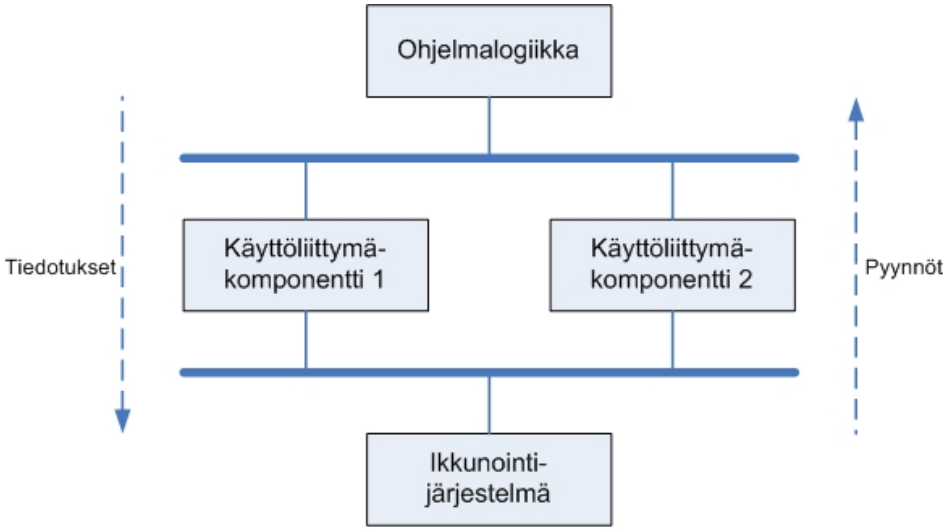
Komponenttien keskinäinen viestien välitys voidaan määrittää dynaamisesti käyttäen esimerkiksi *julkaisija-tilaaja (publish-subscribe)* -mallia. Siinä komponentit ilmoittavat datasta jonka ne haluavat *julkaista* julkaisijakomponentille tuntemattomien komponenttien käyttöön. Toiset komponentit voivat puolestaan rekisteröityä datan *tilaajiksi*, jolloin viestinvälitys huolehtii tilaajien kutsumisesta silloin kun niiden tilaamaa dataa on julkaistu. Tuloksena saadaan järjestelmä, jossa komponentit voivat kommunikoida keskenään tuntematta toisiaan. [4, s. 101-102]

Viestinvälitysarkkitehtuuri palvelee erityisen hyvin graafisia käyttöliittymiä, jotka perustuvat myös asynkroniseen viestinvälitykseen (luku 2.5). Luvuissa 3.3.1 ja 3.3.2 esitellään kaksi viestinvälitysarkkitehtuuriin perustuvaa käyttöliittymäratkaisua. Esimerkit käsitellään varsin perusteellisesti, sillä tutkimuksen kohteena oleva hajautettu sovelluskehys perustuu vastaavaan viestinvälitysarkkitehtuuriin.

3.3.1 Chiron-2, C2

Chiron-2, lyhyemmin *C2* on Taylorin ja kumppaneiden kehittämä komponentti- ja viestipohjainen arkkitehtuuri GUI-ohjelmistoille. Lähtökohtana kehittämisessä on ollut yhä lisääntyvä vaatimus komponenttipohjaiselle, uudelleenkäyttöä edistävälle tyylille myös graafisten käyttöliittymien kehityksessä. Arkkitehtuuri sallii muun muassa ohjelmointikielistä riippumattomien komponenttien välisen kommunikaation, yhteisen nimiavaruuden puuttumisen ja ajon hajautetuissa, heterogeenisissä järjestelmissä. [32]

C2-arkkitehtuuri koostuu komponenteista ja viestinvälityksestä huolehtivista *liittimistä (connector)*, sekä näiden keskinäisten yhteyksien konfiguroinnista. Komponenteille ja liittimille on määritelty ylä- ja alapuoliset rajapinnat. Rajapintojen yhdistäminen on sallittu siten, että komponentin yläpuolinen rajapinta voidaan yhdistää liittimen alapuoliseen ja vastaavasti komponentin alapuolinen liittimen yläpuoliseen. Täten konfiguroinnista syntyy vertikaalinen hierarkia, jossa komponentit on rajoitettu näkemään vain yläpuolellaan sijaitsevat komponentit. Kahden komponentin välinen kytkentä on kielletty, joten kaikki komponenttien välinen kommunikaatio kulkee asynkronisesti liittimien kautta. Komponenttien väliset viestit koostuvat nimiosiosta ja tyypitetyistä parametreista. Viestit jaetaan hierarkiassa ylöspäin kulkeviin *pyyntöihin (request)* ja alaspäin kulkeviin *tiedotuksiin (notification)*. [32]

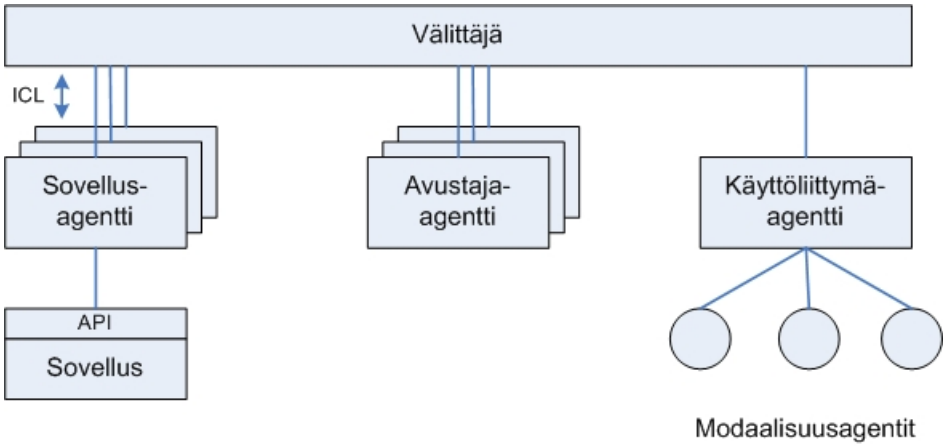


Kuva 3.2: C2 arkkitehtuuri. Kuvassa laatikot ovat komponentteja, paksut poikittaiset viivat liittimiä ja ohuemmat pystyviivat konfiguroituja yhteyksiä liittimiin.

Käyttöliittymäkomponentit sijoitetaan vertikaalisessa hierarkiassa keskelle (kts. kuva 3.2), jolloin ikkunointijärjestelmä sijaitsee niiden alapuolella ja ohjelmalogiikka yläpuolella. Käyttöliittymäkomponentti saa alapuoleltaan ikkunointijärjestelmältä pyynnön tapahtumasta, tulkitsee tämän ja lähettää pyynnön edelleen ylöspäin. Ohjelmalogiikka reagoi pyyntöön muuttamalla tilaansa ja lähettämällä alaspäin tiedotuksen tilan muutoksesta. Tämän perusteella käyttöliittymä päivittää tilan graafista esitystä. [32]

Koska C2-komponentit eivät ole tietoisia alapuolellaan sijaitsevista komponenteista, sama ohjelmalogiikka voidaan kytkeä uudelleen erilaisiin käyttöliittymiin. Uudelleenkytkentä vaatii kuitenkin viestien sisällön yhteneväisyyden sekä nimiosion että parametrien määrän, järjestyksen, tyypin ja yksikön suhteen. Esimerkkinä pinosta elementin poistavan viestin nimiosio `"stack_pop"` voi olla uudessa komponentissa määritettynä nimellä `"pop_stack"`, jolloin viestin tulkitseminen epäonnistuu vaikka viestit olisivatkin rakenteeltaan toisiaan vastaavat. Vastaava esimerkki parametrien tyypityksen osalta voisi olla `"alarm"`-viestin välittämä aikaviive, jota käsitellään toisessa komponentissa millisekunteina ja toisessa sekunteina. [32]

Nimeämisongelmiin Taylor ehdottaa ratkaisuksi kehitysympäristöön lisättävää mahdollisuutta muokata nimeämistä tapauskohtaisesti konfiguraation luonnin yhteydessä, mutta myöntää että monimutkaisemmissa tapauksissa, kuten parametrien lukumäärän muuttuessa ongelma on hoidettava erillisen tulkin avulla. [32]



Kuva 3.3: OAA arkkitehtuuri. [18]

3.3.2 Open Agent Architecture, OAA

Martin ja Moran kumppaneineen ovat julkaisuissaan [18] ja [19] esitelleet *avoimiin agentteihin perustuvan arkkitehtuurin (Open Agent Architecture, OAA)*. OAA-arkkitehtuurin rakenne ja siihen liittyvät agenttityypit on havainnollistettu kuvassa 3.3.

Tavoitteena työssä on kehittää agentteihin perustuva hajautettu järjestelmä, joka tukee useita eri käyttöliittymätyyppejä. OAA:ssa hyödynnetäänkin perinteisen graafisen käyttöliittymän lisäksi myös käsin kirjoitettua tekstiä ja puheen tunnistusta, joka mahdollistaa muun muassa puhelimen kautta ohjattavan käyttöliittymän. Hajautuksen ansiosta vain varsinaisen käyttöliittymän on sijaittava käyttäjän ulottuvilla olevalla koneella (esim. kevyet PDA-laitteet). Tilaa ja laskentatehoa vaativat komponentit, kuten puheentunnistusalgoritmit, voidaan sijoittaa tehokkaammille koneille. [19]

OAA koostuu *agenteista* eli itsenäisistä komponenteista, joilla on sisäinen tila ja jotka tarjoavat palveluja muille agenteille. Agenttien kommunikointi on OAA:ssa toteutettu korkean tason *ICL (Interagent Communication Language)* -kielellä. Keskeisessä roolissa toimiva *välittäjäagentti (Facilitator Agent)* rekisteröi kaikkien käytössä olevien agenttien palvelut, ja saadessaan pyynnön lähettää sen edelleen kyseiseen pyyntöön vastaaville agenteille. Pyyntöjen välitys toimii asynkronisesti. *Sovellusagentit (Application Agent)* huolehtivat varsinaisesta ohjelmalogiikasta ja *avustaja-agentit (Meta Agent)* toimivat välittäjäagentin ja sovellusagenttien avustajina. [19]

Käyttöliittymäagentti (User Interface Agent) liittyy järjestelmään muiden agenttien tavoin, mutta toimii keskeisessä roolissa kommunikoidessaan suoraan käyttäjän kanssa. Se lähettää käyttäjältä saatuja pyyntöjä suoraan muille agenteille ja siten koordinoi koko pyynnön käsittelyä. Monimutkaisemmissa tapauksissa, kuten useamman käskyn sisältävän lauseen purkamisessa, käyttöliittymäagentti lähettää pyynnön luonnollisen

kielen kääntäjän kautta välittäjäagentille. Arkkitehtuurin kuvauksessa käyttöliittymäagentti on jaettu useisiin *modaalisuusagentteihin (Modality Agents)*, jotka kuvaavat eri syöttövälineitä, kuten hiirtä, näppäimistöä, puheen nauhoitusta ja kaiuttimia. [19]

Älykäs, agentteihin perustuva käyttöliittymä mahdollistaa erittäin monipuolisen kommunikoinnin käyttäjän kanssa. Käyttöliittymä kykenee myös itsestään mukautumaan muuttuviin tilanteisiin, kuten näytön puuttuessa välittämään viestin puheen avulla. Kokonaan uuden käyttöliittymän rakentaminen vaatii vastaavasti uuden käyttöliittymäagentin tekemistä.

3.4 Ohjelmalogiikan irrotus käyttöliittymästä

Kaikissa tarkastelluissa arkkitehtuureissa graafinen käyttöliittymä on irrotettu ohjelmalogiikasta siten, että se voidaan sijoittaa hajautetussa järjestelmässä erilliselle koneelle. Yhtenäisen sovelluksen sisällä tapahtuvaa ohjelmalogiikan ja käyttöliittymän irrottamista on tutkittu graafisia käyttöliittymiä koskevassa kirjallisuudessa, mutta hajautetussa järjestelmässä ongelmat korostuvat järjestelmän osien väliseen kommunikaatioon kuluvan ajan pidentyessä. Tässä luvussa tarkastellaan irrotuksen hyviä ja huonoja puolia, joista varsinkin jälkimmäiset on otettava erityisesti huomioon graafisia käyttöliittymiä hajautettuun järjestelmään suunniteltaessa.

Ohjelmalogiikan irrotus käyttöliittymästä on ollut yksi kantavia periaatteita käyttöliittymien suunnittelussa jo 80-luvulta lähtien. Käyttöliittymämallit, kuten *Sisältö-Näkymä-Kontrolleri (Model-View-Controller, MVC)* [12] [10, s. 4-6] ja *Presentaatio-Abstraktio-Kontrolli (Presentation-Abstraction-Control, PAC)* [6], perustuvat siihen. Hartson käsittelee aihetta 1989 julkaistussa artikkelissaan [13] käyttäen termiä *dialogin riippumattomuus (Dialogue Independence)*.

Tarve dialogin riippumattomuudelle perustellaan sillä, että käyttöliittymän kehittäminen on riippuvainen käyttäjästä ja tämän tuomasta inhimillisestä näkökulmasta toisin kuin ohjelmalogiikan kehittäminen, joka on riippuvainen teknisistä arvoista. Hyvän käyttöliittymän suunnittelu vaatii kognitiivisia taitoja, joten usein käyttöliittymän suunnittelijana toimii eri henkilö kuin itse ohjelmiston suunnittelija. Eritasoisille käyttäjille halutaan käyttöön eritasoisia käyttöliittymiä, joiden liittäminen samaan ohjelmalogiikkaan vähentää uudelleen ohjelmointia. Lisäksi käyttöliittymän viimeistely vaatii useampia iteraatiokierroksia loppukäyttäjien testejä, joten käyttöliittymään tehdyt muutokset täytyy saada nopeasti testattavaksi. [13, luku 1]

Vaikka pääasiallinen motivaatio käyttöliittymän irrotuksessa onkin käyttäjän palveleminen ja käyttöliittymän laatu, saadaan irrotuksesta lisäetuja ohjelmiston lisääntyneen modularisoinnin myötä. Komponenttien uudelleenkäytettävyys helpottuu, jolloin

kustannukset muutosvaiheessa pienenevät. Modularisoinnin myötä myös alustariippumattomien versioiden tuottaminen nopeutuu.

Käyttöliittymän irrotus aiheuttaa myös ongelmia. Suurimmat niistä johtuvat *semanttisen palautteen (Semantic Feedback)* sijainnin määrittämisestä arkkitehtuurin sisällä. Semanttisella palautteella tarkoitetaan sitä, että käyttöliittymä avustaa käyttäjää antamalla vinkkejä kulloinkin toteutettavasta tehtävästä. Esimerkiksi käyttäjän syöttäessä hakukenttään merkkijonoa voidaan käyttäjälle esittää jokaisen syötteen mukaan rajoittuva lista jäljellä olevista vaihtoehdoista. Jos halutaan pitää käyttöliittymän irrotus täydellisenä, käyttöliittymän on haettava ohjelmalogiikalta lista vaihtoehdoista jokaisen syötteen jälkeen. Semanttisen palautteen lisäksi ongelmia aiheuttaa statustietojen esittäminen, kuten laskennan etenemisen ilmaiseminen. Tässä tapauksessa ratkaistavaksi jää, sijoitetaanko statustietojen lähetys ohjelmalogiikkaan, vaikka kyse onkin käyttöliittymään liittyvästä toiminnosta. [9]

Ongelmatapauksissa loppuratkaisu on tasapainoilua irrotusasteen pienenemisen ja lisääntyneen kommunikaation välillä [9]. Hajautetussa järjestelmässä molemmat korostuvat entisestään, sillä hajautus motivoi uudelleenkäytettävien komponenttien kapseloimista, mutta komponenttien välimatkan pidentyessä viestien välitys hidastuu.

4 Viestit

Sekä graafisissa käyttöliittymissä että viestinvälitysarkkitehtuuriin perustuvissa hajautetuissa järjestelmissä tärkeään osaan nousevat *viestit*. Graafisissa käyttöliittymissä viestit käsittävät luvussa 2.5 mainittujen ikkunointijärjestelmän viestien lisäksi myös käyttöliittymäkomponenttien generoimat viestit. Hajautetussa järjestelmässä viestit käsittävät ohjelmalogiikan ja käyttöliittymän välillä kuljetettavat viestit. Tässä luvussa pohditaan hajautetuissa järjestelmissä käytettyihin graafisiin käyttöliittymiin liittyviä viestejä, sekä pyritään löytämään niille yhtenäinen, järkevä jaottelu ja suomenkielinen terminologia.

Aihetta vaikeuttaa se, että graafisissa käyttöliittymissä viestien nimeäminen, jaottelu, sisältö ja käsittely riippuvat sekä ikkunointijärjestelmästä että käytettävästä työkalupakista. Jo itse viesti-termille on käytössä useita englanninkielisiä termejä, kuten *event*, *message* ja *action*. Viestejä generoivat sekä ikkunointijärjestelmä, käyttäjä että käyttöliittymäkomponentit ja niitä voidaan käsitellä käyttämällä erilaisia *suodattimia (filter)* tai *kuuntelijoita (listener)* sekä ylikirjoittamalla käyttöliittymäkomponenttien jäsenfunktioita.

Luvussa 4.1 esitellään joitakin valmiita tapoja luokitella käyttöliittymien viestejä. Yksikään esitellyistä luokitteluista ei kuitenkaan sovellu käyttöön sellaisenaan, joten niiden pohjalta suunnitellaan uusi luokittelu, joka esitellään luvussa 4.2. Viestejä läheisesti koskevia tilatietoja käsitellään luvussa 4.3.

Viestien käsittelyä eri työkalupakeissa sivutaan tapahtumien etenemistä käsittelevässä luvussa 4.4, jossa esimerkkinä esitellään XUL-tapahtumamalli. Viestien käsittelyyn ei perehdytä tätä esimerkkiä syvällisemmin, vaan sen sijaan luvussa 4.5 pyritään laajentamaan työkalupakkien yleisesti tarjoamaa viestien käsittelytyyliä esittelemällä kaksi tarkoitukseen soveltuvaa suunnittelumallia. Tavoitteena on lisätä viestien käsittelyyn joustavuutta ja ajonaikaista muunneltavuutta.

4.1 Valmiita luokittelutapoja

Hajautettujen järjestelmien käyttöliittymäviestien luokitteluun yleisellä, työkalusta ja ohjelmointikielestä riippumattomalla tasolla löytyy kirjallisuudesta varsin vähän lähtökohtia. Eri lähteistä yhdistelemällä voidaan kuitenkin löytää muutamia luokitteluperiaatteita, jotka on esitelty tässä luvussa. Termit on jätetty suomentamatta, jotta ne

eivät sekaantuisi myöhemmin luokitteluun valittavien suomenkielisten termien kanssa.

4.1.1 Status- ja event-viestit

Dix ja Abowd [8] jakavat käyttöliittymäviestit *status*- ja *event*-tietoja käsitteleviin viesteihin. Näistä status, esimerkiksi hiiren kursorin sijainti näytöllä, on jatkuvaa ja statuksen arvo voidaan saada selville millä tahansa hetkellä. Eventit puolestaan ovat kertaluontoisia, kuten hiiren napin painallus. Status ja eventit vaikuttavat toisiinsa, event saattaa muuttaa statusta, ja statuksen muutos saattaa laukaista eventin. Käsitettä monimutkaistaa se, että jaottelu muuttuu riippuen siitä, missä mittakaavassa tilannetta tarkkaillaan. Esimerkiksi hiiren napin painallus voidaan käsittää yhtenä painallus-eventtinä tai vaihtoehtoisesti alas- ja ylös-eventteinä, joiden välillä status muuttuu alas painuneeksi ja taas vapautetuksi.

Työkalupakit ja ikkunointijärjestelmät käsittelevät eventtejä ja statusta eri tavoilla, mikä täytyy ottaa huomioon, kun etsitään yhtenäistä viestien nimeämis- ja käsittelytapaa. Tilannetta voidaan havainnollistaa hiiren napin painalluksella: www-pohjaisia käyttöliittymiä javascriptillä tehtäessä ainoa käsittelyyn tuleva viesti on `clicked`, kun taas Qt:ta käytettäessä painallus-viestiä ei tule viestijonoon lainkaan, vaan se on koottava statusta seuraamalla `mouseDown` ja `mouseUp`-viesteistä.

4.1.2 Ikkunointijärjestelmän ja käyttöliittymäkomponenttien viestit

Alencar ja kumppanit jakavat käyttöliittymäohjelmistojen viestit järjestelmän ulkopuolelta tuleviin ja objektien itsensä generoimiin, toisiin objekteihin vaikuttaviin viesteihin. Objekteiksi määritellään sekä käyttöliittymäkomponentit että ohjelmalogiikkaan kuuluvat komponentit. Järjestelmän ulkopuolelta tulevista viesteistä käytetään nimitystä *Causal Action* ja objektin tuottamasta viestistä nimitystä *Effectual Action*. Tässä yhteydessä actionit määritellään funktioiksi ja tapahtumiksi, jotka kysyvät tai muuttavat käytettävän objektin sisäistä tilaa. [2]

Käyttöliittymäkomponentteihin rajattuna jako voidaan rinnastaa ikkunointijärjestelmältä tuleviin ja komponentin itsensä generoimiin viesteihin. Edellisessä luvussa esitetty hiiren painallus toimii esimerkkinä myös tässä tapauksessa, ikkunointijärjestelmältä saadaan erilliset ylös- ja alas-viestit, joista käyttöliittymäkomponentti kokoaa painallusviestin.

Jakoon vaikuttaa käytössä olevan työkalupakin tapa käsitellä viestejä. Joissakin työkalupakeissa, kuten esimerkiksi XUL:ssa [21], ikkunointijärjestelmän viestejä ei pääse käsittelemään, vaan on tyydyttävä valmiisiin käyttöliittymäkomponenttien viesteihin. Myöskään omien käyttöliittymäkomponenttien viestien laukaisemiseen ei XUL:ssa ole

kehitetty tekniikoita. Qt:ssa puolestaan näin luokitelluille, erityyppisille viesteille on kehitetty erilliset käsittelytavat, jotka esitellään tarkemmin luvussa 6.2.1.

4.1.3 Käyttöliittymän ja ohjelmalogiikan väliset viestit

Luvussa 3.3.1 esitellyssä hajautetussa C2-järjestelmässä käyttöliittymästä ohjelmalogiikalle kulkevat, käyttäjältä lähtöisin olevat viestit nimettiin *pyynnöiksi* ja ohjelmalogiikalta käyttöliittymälle kulkevat, näytön päivitystä vaativat viestit *tiedotuksiksi*. Tämä jako tulee käytännössä vastaan juuri hajautetuissa, asynkroniseen viestinvälitykseen perustuvissa järjestelmissä, joissa käyttöliittymä on irrotettu ohjelmalogiikasta. Muissa järjestelmissä käyttöliittymäkomponenttien päivitykseen käytetään pääasiassa suoria funktiokutsuja tai kutsujen toimittamista viestijonoon. Suorien kutsujen ongelmana on luvussa 2.5 mainittu säieturvallisuus siinä tapauksessa, että funktiota kutsutaan käyttöliittymäsäikeen ulkopuolelta.

4.2 Viestien luokittelu

Tarkastelun tuloksena todettiin, ettei hajautettujen järjestelmien graafisia käyttöliittymiä koskevien viestien luokitteluun ole olemassa yleistä, työkalusta riippumatonta jakoa, joten tällainen kehitettiin itse luvussa 4.1 esitettyjen lähtökohtien pohjalta.

Viestit voidaan jakaa yleisesti kolmeen luokkaan: *tapahtumiin*, *komponenttitapahtumiin* ja *komentoihin*. Näistä kaksi ensimmäistä on ilmoituksia siitä, että jotain *on tapahtunut*, kun taas viimeisestä seuraa että jotain *tulee tapahtumaan* käyttöliittymässä. Kaikkiin viesteihin voidaan tarvittaessa lisätä tietoja viestin lähettäjän *tilasta (status)*, jota on käsitelty tarkemmin luvussa 4.3.

- *Tapahtumat* ovat ikkunointijärjestelmältä tulevia, alemman tason viestejä. Tapahtumiin kuuluvat käyttäjältä lähtöisin olevat viestit, esimerkkeinä hiiren ja näppäimistön toimintojen laukaisemat viestit. Viesti tapahtumasta tulee viestijonoon välittömästi sen jälkeen, kun käyttäjä on laukaissut kyseisen tapahtuman. Lisäksi tapahtumia ovat ikkunointijärjestelmän viestit, joista esimerkkeinä voidaan mainita ilmoitus komponentin kuvan vioittumisesta näytöllä, johon komponentin on reagoitava piirtämällä itsensä uudelleen.

- *Komponenttitapahtumat* ovat käyttöliittymäkomponenttien laukaisemia viestejä, jotka saattavat koostua useammista tapahtumista. Jako tapahtuman ja komponenttitapahtuman välillä ei ole kaikissa työkalupakeissa selvä. Siitä huolimatta

jako on mielekäs etenkin tapauksissa, joissa halutaan luoda omia käyttöliittymäkomponentteja ja niille täysin uusia komponenttitapahtumia.

- *Komennot* ovat käyttöliittymäkomponenttien päivitykseen käytettäviä, ohjelmalogiikan generoimia viestejä. Esimerkiksi tekstikentän `setText`-funktion kutsu on komento. Hajautetussa järjestelmässä funktion kutsun aiheuttava, logiikalta saapuva viesti on komento.

Toteutuksen (luku 6) kannalta oleellinen jako on myös kaikkien viestityyppien jako *visuaalisiin* ja *funktionaalisiin* viesteihin, joista visuaalinen viesti vaikuttaa näytön ulkonäköön, kun taas funktionaalinen vaikuttaa toiminnallisuuteen. Esimerkkeinä mainittakoon painonapin painallus, jolla on funktionaalinen merkitys, mutta ei visuaalista ellei haluta välittää napin painumisen ja palautumisen aiheuttamaa muutosta. Vastaavasti ikkunan siirrolla on visuaalinen merkitys, mutta ei funktionaalista. Tätä jakoa käyttäen voidaan suunnata viestejä haluttuihin kohteisiin ja siten vähentää viestinvälityksen kuormitusta. Huomattavaa kuitenkin on, että viestien visuaalisuus ja funktionaalisuus riippuu käytössä olevan komponentin lisäksi myös kehitettävästä sovelluksesta, esimerkiksi muissa sovelluksissa vain visuaalisesti merkittävä ikkunan venytys saattaa karttasovelluksessa olla toiminnallinen viesti, joka aiheuttaa uuden karttapohjan haun.

4.3 Tilatiedot

Kuten luvussa 4.1.1 esitettiin, myös *tila (status)* vaikuttaa viestien merkitykseen. Esimerkiksi hiiren painallus kartalla symbolin päällä vaikuttaa siten, että symboli tulee valituksi, kun taas painallus vapaalla alueella lisää uuden symbolin. Tällaisissa tapauksissa viestiin tarvitaan mukaan tilatieto eli informaatio siitä, missä painallus tapahtui.

Käyttöliittymäkomponentteja, niille tarkoitettuja komentoja ja niiden generoimia komponenttiviestejä suunniteltaessa on suunniteltava myös se, mitä tilatietoja viesteihin täytyy laittaa mukaan. Lisätäänkö monimutkaisemman komponentin tapauksessa viestiin koko tila vai pelkästään muutettava osa? Myös erilliselle tilakyselylle voi tulla tarvetta varsinkin tapauksessa, jossa tila halutaan tallettaa uudelleenlatausta tai undo/redo-toimintoja varten.

4.4 Tapahtumien eteneminen käyttöliittymässä

Tapahtuman eteneminen (event propagation) kuvaa tapahtumien kuljettamista graafisen käyttöliittymän puurakenteen komponentilta toiselle, jonka avulla lapsikompo-

nentin tapahtumat saadaan vanhempi-komponenttien tietoon. Tapahtuman etenemisen toteutus riippuu työkalupakista: tapahtuma voidaan toimittaa ikkunalle, josta se kulkee puuta pitkin lapsikomponenteille tai se voidaan antaa suoraan komponentille ja kuljettaa sieltä ylöspäin ikkunalle. Joissakin työkalupakeissa viestien välitys komponenttien välillä on totetutettu kokonaan eri tyylillä, jolloin tapahtuman etenemistä ei ole toteutettu lainkaan. Esimerkiksi Qt:n tapahtumien käsittelyssä komponenttien välinen viestien välitys on toteutettu luvussa 6.2.1 esitellyllä tavalla, joten tapahtumien etenemistä ei ole tarvinnut toteuttaa.

Luvussa 4.4.1 esitellään XUL-käyttöliittymien toteuttama tapahtumamalli. Esimerkki on valittu sillä perusteella, että mallissa käytetään tapahtumien etenemistä mahdollisimman monipuolisella tavalla.

4.4.1 Esimerkki: XUL-tapahtumamalli

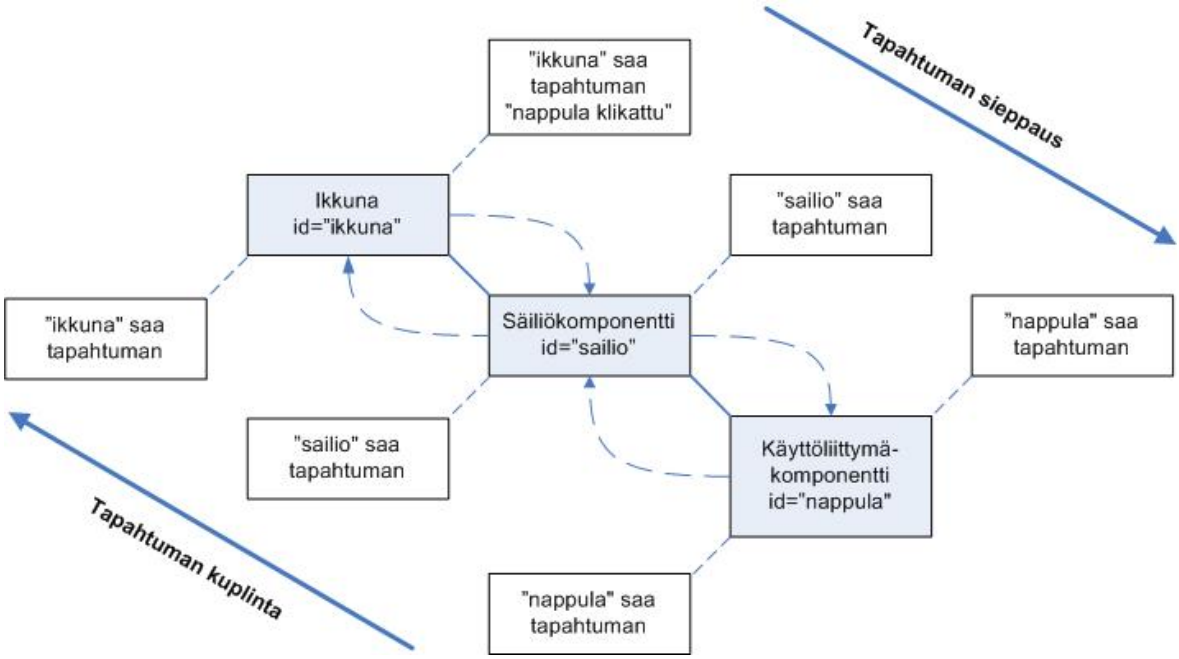
Esimerkkinä tapahtuman etenemisestä esitellään XUL:n tapahtumamalli ([41], luvut 6.1 ja 6.2), jossa on toteutettuna molempiin suuntiin kulkeva eteneminen. XUL on XML-pohjainen, ikkunointijärjestelmästä riippumaton käyttöliittymien kuvaukseen tarkoitettu kieli, jota sellaisenaan käyttävät mm. Mozilla Firefox ja Thunderbird. Kaikki toiminnallisuus XUL-käyttöliittymään toteutetaan javascript-kielellä.

XUL:ssa tapahtuman eteneminen koostuu *tapahtumien sieppauksesta (event capturing)* ja *tapahtumien kuplimisesta (event bubbling)*. Tapahtumien sieppaus alkaa puun juurikomponentin vastaanottaessa tapahtuman viestijonosta. Juurikomponentti lähettää tapahtuman käyttöliittymäkomponenttien muodostamaa puuta alaspäin. Tapahtumien kupliminen on sieppauksen vastakohta ja tarkoittaa tapahtuman etenemistä sen laukaisseelta käyttöliittymäkomponentilta takaisin puuta ylöspäin. Komponentit voivat käsitellä tapahtuman sekä sieppaus- että kuplimisvaiheessa. Käsittelyn jälkeen tapahtuma jatkaa etenemistään ja lopussa se käsitellään oletuskäsittelijällä. Tarvittaessa eteneminen voidaan pysäyttää.

Tapahtumamalli on esitetty kuvassa 4.1. Esimerkkinä toimivassa ikkunassa `ikkuna` on yksi säiliökomponentti `sailio`, joka sisältää painonappi-käyttöliittymäkomponentin `nappula`. Painonappia painettaessa tieto painalluksesta tulee ikkunalle, jolta tapahtuman sieppaus etenee painonapille. Tämän jälkeen tapahtuma kuplii takaisin ikkunalle.

Tapahtumiin reagoidaan lisäämällä komponenteille *tapahtumakuuntelijoita (event listener). Tapahtumakäsittelijäksi (event handler)* kutsutaan kuuntelijalle parametrina välitettävää funktiota. Seuraava koodirivi lisää `nappula`-komponentin `click`-tapahtumalle kuuntelijan, käsittelijäksi annetaan `widgetClicked`-funktio.

```
nappula.addEventListener('click', widgetClicked, false);
```



Kuva 4.1: XUL tapahtumamalli.

Alla olevassa koodissa luodaan kuvan 4.1 kaltainen XUL-käyttöliittymä. Säiliökomponentille ja käyttöliittymäkomponentille lisätään tapahtumakuuntelijat sekä sieppaus- että kuplintavaiheeseen. Tapahtumakäsittelijänä toimii `widgetClicked`-funktio, joka tulostaa sen komponentin tunnisteen jossa tapahtuma on etenemässä. Tulostuksia tulee neljä: `sailio`, `nappula`, `nappula`, `sailio`.

```
<!-- XUL ikkunan määritykset -->
<window id="ikkuna" title="Esimerkki"
   xmlns:html="http://www.w3.org/1999/xhtml"
   xmlns=
     "http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">

   <!-- XUL ikkunan sisältö -->
   <box id="sailio">
     <button id="nappula" label="Nappula"/>
   </box>

   <!-- Toiminnallisuus skriptissä -->
   <script>
     // Käsittelijäfunktio
     function widgetClicked(event) {
```

```
            alert(event.currentTarget.id);
        }

        // Haetaan osoittimet muuttujiin nimien perusteella
        var nappula = document.getElementById("nappula");
        var sailio = document.getElementById("sailio");

        // Lisätään sailiolle ja painonapille kuuntelijat
        // sekä sieppaus- että kuplintavaiheeseen
        // (viimeinen parametri = "capturing listener")
        nappula.addEventListener('click', widgetClicked, true);
        nappula.addEventListener('click', widgetClicked, false);
        sailio.addEventListener('click', widgetClicked, true);
        sailio.addEventListener('click', widgetClicked, false);
    </script>
</window>
```

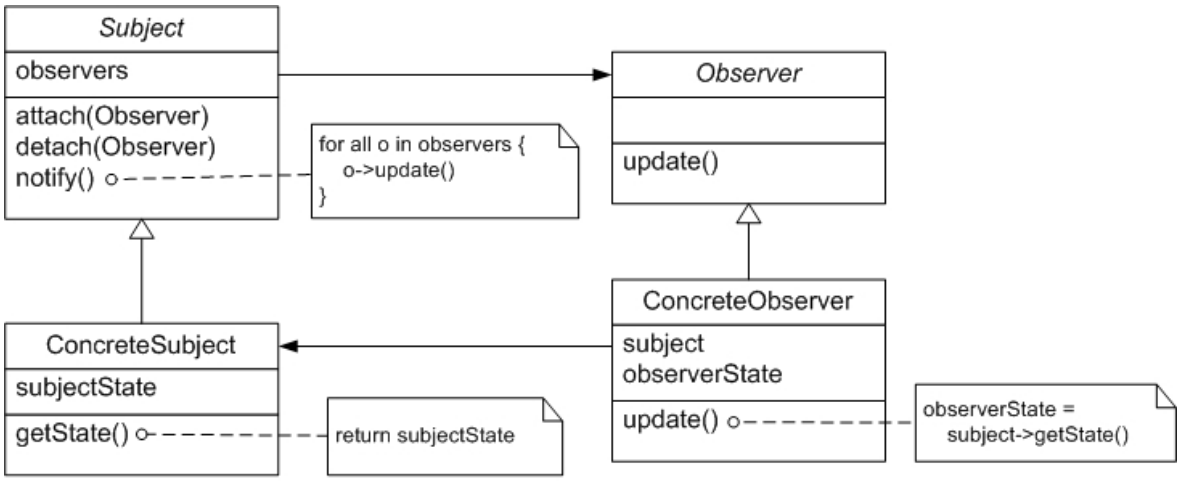
4.5 Suunnittelumallit viestien käsittelyssä

Viestien käsittely hoidetaan yleisesti työkalupakin tarjoamalla tavalla. Sen lisäksi tapahtumia voidaan käsitellä myös yleiskäyttöisemmillä keinoilla. Tässä luvussa on esitelty kaksi viestien käsittelyyn soveltuvaa *suunnittelumallia (design patterns)*: *tarkkailija (observer)* ja *komento (command)*, joiden avulla voidaan lisätä käsittelyyn joustavuutta ja ajonaikaista muunneltavuutta. Mallien avulla voidaan myös pienentää kommunikoinnin aiheuttamaa komponenttien välistä riippuvuutta.

4.5.1 Tarkkailija (Observer)

Tarkkailija (Observer) -suunnittelumallin [10, sivut 293-304] tavoitteena on luoda olioiden välille yksi-moneen riippuvuuksia siten, että yhden olion tilan muuttuessa siitä riippuvat oliot saavat muutoksesta ilmoituksen ja päivittyvät automaattisesti. Mallin tärkeimmät oliot ovat *subjekti (subject)* ja *tarkkailija (observer)*. Subjektiin, jonka tilaa seurataan, voi liittyä mielivaltaisen monta tarkkailijaa, joille ilmoitetaan kun subjektin tilassa tapahtuu muutos. Ilmoituksen seurauksena tarkkailija päivittää omaa tilaansa vastaavasti. Toteutuksesta riippuen tarkkailija voi myös kysyä subjektilta sen tilatietoja päivitystä varten.

Käytännön toteutus tehdään käyttämällä tarkkailija- ja subjektiluokkia. Subjek-



Kuva 4.2: Tarkkailija-suunnittelumallin luokkarakenne.

ti pitää yllä listaa tarkkailijoista ja käyttää ilmoituksen tekemiseen näiden yhteistä tarkkailija-rajapintaa. Konkreettisissa tarkkailijoissa toteutetaan varsinainen toiminnallisuus. Tarkkailijoita voidaan lisätä ja poistaa dynaamisesti. Kuvassa 4.2 on havainnollistettu tarkkailija-mallin toimintaa luokkakaavion avulla.

Joissakin työkalupakeissa tarkkailija-mallia on sovellettu jo käyttöliittymäkomponentteja suunniteltaessa. Esimerkiksi Javassa `Action` ja `ActionListener` [26] toteuttavat tarkkailija-mallin siten, että käyttöliittymäkomponentti, jolle kuuntelijoita lisätään toimii subjektina ja toteutettu `ActionListener` tarkkailijana. Tilatietojen kysymisen sijaan tiedotuksen parametrina lähetetään `Action`, joka selvittää muutoksen luonteen.

4.5.2 Komento (Command)

Komento (Command) suunnittelumalli [10, sivut 233-242] kapseloi pyynnön olioksi. Abstraktin `Command`-luokan avulla esitetään pyynnön rajapinta, joka on usein pelkistetty `execute`-funktioksi. Rajapintaa käyttäen muille komponenteille voidaan asettaa pyyntöjä tietämättä etukäteen mitä toteutuksessa tullaan tekemään. Varsinainen toiminnallisuus toteutetaan rajapinnan toteuttavissa luokissa.

Esimerkiksi graafisten käyttöliittymien tapauksessa komentoa voitaisiin käyttää valikon toteuttamiseen. Valikon `MenuItem`-jäseniin toteutettaisiin `setCommand`-funktio, jolla voitaisiin asettaa valinnan vaikutus. Valikon jäsenen painalluksen toteutus sisältäisi pelkän `command->execute()`-kutsun. Esimerkiksi aineiston tallennus toteutettaisiin `Command`-luokasta perityn `SaveCommand` luokan `execute`-metodiin, jonka jälkeen `SaveCommand` annettaisiin tallennuksesta vastaavalle valikon jäsenelle. Suunnittelumallia käyttäen tallennus voitaisiin helposti toteuttaa samaa `SaveCommand`-luokan ilmen-

tymää käyttäen myös muista valikon kohdista, sekä tarvittaessa myös muualta käyttöliittymästä.

Komento-suunnittelumalli helpottaa pyyntöjen seuraamista. Sen avulla pyynnöt voidaan kirjoittaa lokiin, tai tallentaa jonoon komentohistorian selaamista varten (undo-redo).

5 Sovelluskehys

Luvussa 4.5 esiteltyjen suunnittelumallien avulla pyrittiin lisäämään viestien käsittelyn joustavuutta ja pienentämään komponenttien välisiä riippuvuuksia. Joustavuuden lisäämiseen, komponenttien välisten riippuvuuksien pienentämiseen ja ohjelmakoodin uudelleenkäyttöön voidaan pyrkiä myös korkeammalla tasolla käyttämällä sovelluskehyksiä ja niihin liitettäviä ohjelmistokomponentteja.

Luvussa 5.1 käsitellään ohjelmistokehityksen yleisiä ongelmia, joiden ratkaisemista sovelluskehyksen avulla käsitellään luvussa 5.2. Luvussa 5.3 esitellään hajautettu sovelluskehys, jonka jatkoksi tutkielman käytännön osuuden graafisia käyttöliittymiä tullaan suunnittelemaan, sekä listataan kyseisen hajautetun sovelluskehyksen graafisille käyttöliittymille tuomia lisävaatimuksia.

5.1 Ohjelmistonkehityksen yleisimmät ongelmat

Perinteinen ohjelmistonkehitys on keskittynyt ohjelmistotuotteiden projektimuotoiseen kehitykseen eli tuottamaan yhden valmiin ohjelmiston kerrallaan ennalta sovittuna eräpäivänä. Keskittyminen eräpäivään on vienyt huomiota mahdollisesti tarvittavalta myöhemmältä kehitykseltä ja ylläpidolta, mutta silti vain murto-osa ohjelmistoista valmistuu eräpäivään mennessä ja sovitussa budjetissa. Äärimmäistä luotettavuutta vaativien ohjelmistojen kehityksessä laatuvaatimusten aiheuttamat kustannukset ovat osoittautuneet huomattaviksi. Ylläpidon kustannuksista pitkäikäisen ohjelmiston tapauksessa kertyy jopa 80% koko ohjelmiston kustannuksista, mistä johtuen lisääntyvä työvoiman tarve vanhempien ohjelmistojen ylläpidossa vähentää yrityksen kilpailukykyä uusia ohjelmistoja kehitettäessä. [5, sivut 4-5]

Nykyiseen ohjelmistonkehitykseen liittyvien ongelmien pohjalta voidaan määritellä tavoitteita, joihin ohjelmistokehitystä parannettaessa tulisi pyrkiä. Boschin [5, sivut 5-6] mukaan näitä on neljä:

- *Kehityskustannusten vähentäminen* on tärkeää, jotta ohjelmistojen kehitys säilyisi kannattavana.
- *Kehitettävien ohjelmistojen laadun parantaminen* kohtuullisilla kustannuksilla.
- *Kehitykseen ja ohjelmiston julkaisemiseen kuluvan ajan pienentäminen* lisää kilpailukykyä markkinoilla jopa enemmän kuin kehityskustannusten vähentäminen.

- *Ylläpitokustannusten pienentäminen*, joka on jopa tärkeämpää kuin kehityskustannusten pienentäminen ylläpitokustannusten ollessa moninkertaisia kehityskustannuksiin nähden.

5.2 Ohjelmiston osien uudelleenkäyttö sovelluskehyksen avulla

Olemassa olevien ohjelmiston osien uudelleenkäyttö on yleisesti tunnustettu keino luvussa 5.1 mainittuihin tavoitteisiin pääsemiseksi. Jos ohjelmisto voidaan kehittää jo olemassa olevista ohjelmistokomponenteista, kehitykseen kuluva aika ja kustannukset pienenevät. Laatu paranee, kun samoja komponentteja testataan useissa järjestelmissä, ja ylläpitoon kuluva aika pienenee komponenttien päivitysten hyödyttäessä useampia valmiita ohjelmistoja. Ohjelmistojen ja niiden osien onnistunutta uudelleenkäyttöä esiintyy jo huomattavissa määrin, perinteisiä esimerkkejä tällaisista ovat mm. käyttöjärjestelmät, tietokantojen hallintaohjelmistot ja kääntäjät. Uudempina esimerkkeinä voidaan mainita käyttöliittymien suunnitteluun luodut työkalupakit, komponenttien väliseen kommunikaatioon tarkoitetut standardoidut rajapinnat sekä www-palvelimet ja -selaimet. [5, sivut 6-8]

Ongelmalliseksi koituu kuitenkin ohjelmistoyrityksen sisäisten, liiketoiminta-alaan liittyvien ohjelmiston osien käyttö useissa tuotteissa. Näkökulma, jonka mukaan kerran tehtyä voidaan LEGO-palikan tapaan käyttää uudelleen seuraavassa tuotteessa, ei ole yksinkertaisuudestaan huolimatta realistinen. Kokemus on osoittanut, että uudelleenkäytön on toimiakseen oltava *suunnitelmallista* jo komponenttia kehitettäessä ja komponenttien kehityksen on tapahduttava suunnittelemalla ja sovittamalla komponentteja valmiiseen, korkeamman tason rakenteeseen. Tämä tarkoittaa yrityksen sisäisen *tuoterungon (product line)* arkkitehtuurin suunnittelua ja *sovelluskehyksen (application framework, suom. myös ohjelmistokehys)* toteuttamista tuoterunkoarkkitehtuurin pohjalta. [5, sivut 6-8]

Sovelluskehys on luokka-, komponentti- ja/tai rajapintakokoelma, joka toteuttaa jonkin sovellusjoukon yhteisen arkkitehtuurin ja perustoiminnallisuuden. Kehys voidaan ymmärtää ohjelmistorunkona, joka sisältää aukkoja, eli *laajennoskohtia*, eikä sellaisena ole välttämättä suorituskelpoinen ohjelma. Sovellukehyksestä *erikoistetaan* itsenäinen sovellus lisäämällä kehyksen laajennoskohtiin sovelluskohtaista koodia komponentin tai uuden luokan muodossa. [17, s. 187-188]

Yllä esitetyssä sovelluskehyksen määritelmässä termit luokka ja rajapinta ovat oliopohjaisten kielten tapauksessa selkeät. Komponentti-termiä tarkennetaan luvussa 5.2.1 ja lisäksi luvussa 5.2.2 tarkastellaan graafisten käyttöliittymien kannalta oleellista sovelluskehyksiin liittyvä seikkaa, eli kontrollin hallintaa sovelluskehyksessä.

5.2.1 Ohjelmistokomponentit

Bosch [5, sivu 13] määrittelee ohjelmistokomponentin rakenneyksiköksi, jolla on yksiselitteisesti määritellyt rajapinnat, jotka komponentti *tarjoaa (provided)*, joita se *vaatii (required)* ja joilla sen sisäinen tila *konfiguroidaan*. Lisäksi hän jakaa komponentit kolmelle eri tasolle niiden uudelleenkäytettävyyden mukaan:

- Ohjelmiston eri versioissa uudelleenkäytettävät komponentit, joiden kehityksessä on varauduttava vain tuleviin muutoksiin.
- Tuoterungon komponentit, joiden on sovelluttava kaikkien tuoteperheen ohjelmistojen käyttöön.
- Kolmannen osapuolen komponentit, joiden käytettävyyden on ulotuttava myös yrityksen rajojen ulkopuolelle. Näitä kutsutaan myös *COTS-komponenteiksi (commercial off the shelf)*.

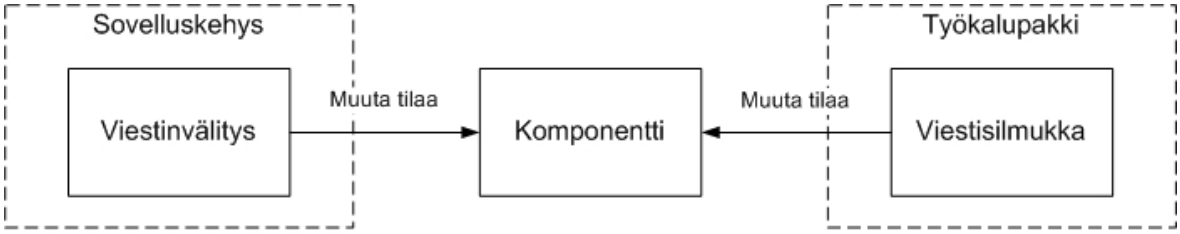
Komponenttikehys (component framework, framelet) [30, s. 280] tukee komponenttien kehityksen standardoimista ja yhtenäistä liittämistä sovelluskehykseen. Komponenttikehyksen avulla voidaan asettaa komponentin ympäristön ominaisuuksia, sekä säädellä komponentin kommunikointia kehyksen ja muiden komponenttien kanssa.

5.2.2 Kontrolli sovelluskehyksessä

Sovelluskehyksen tapauksessa sovelluksen kontrolli säilyy useimmiten kehyksellä, joka tarvittaessa kutsuu sovelluskohtaisia komponentteja. Tätä kontrollin siirtoa kutsutaan Hollywood-periaatteeksi, ajatuksen ollessa *'Älä soita meille, me soitamme sinulle' ('Don't call us, we'll call you')* [17, s. 191].

Sovelluskehyksen kontrollin hallinnasta syntyy ongelmia, kun pyritään yhdistämään graafisia käyttöliittymiä sovelluskehykseen, sillä myös työkalupakit olettavat normaalitapauksessa hallitsevansa sovelluksen pääkontrollisäiettä [5, sivut 244-245]. Tilannetta on havainnollistettu kuvassa 5.1, joka kuvaa graafisen käyttöliittymän omaavaa sovelluskehyksen komponenttia. Sovelluskehys saattaa kutsua komponenttia koska tahansa esimerkiksi toisen komponentin aloitteesta, kun taas graafinen käyttöliittymä kutsuu komponenttia käyttäjän aloitteesta. Näin ollen komponentille saapuu tilanpäivityspyyntöjä kahdelta suunnalta.

Kontrollia ei voida siirtää graafisilta käyttöliittymiltä sovelluskehykselle, koska graafisten käyttöliittymien toiminnan kannalta on tärkeää, että käyttäjä pystyy kutsumaan sovelluksen toiminnallisuutta haluamanaan ajankohtana. Sovelluskehyksen kontrollia ei



Kuva 5.1: Sovelluskehyksen ja työkalupakin välinen kontrolliristiriita.

myöskään voida siirtää graafisille käyttöliittymille, sillä se rikkoisi sovelluskehyksen toimintaperiaatteen. Toteutuksen (luku 6) yhteydessä pyritäänkin löytämään keino, jolla kahden kehyksen kontrollit saataisiin yhdistettyä ilman ristiriitoja.

5.3 Hajautettu sovelluskehys

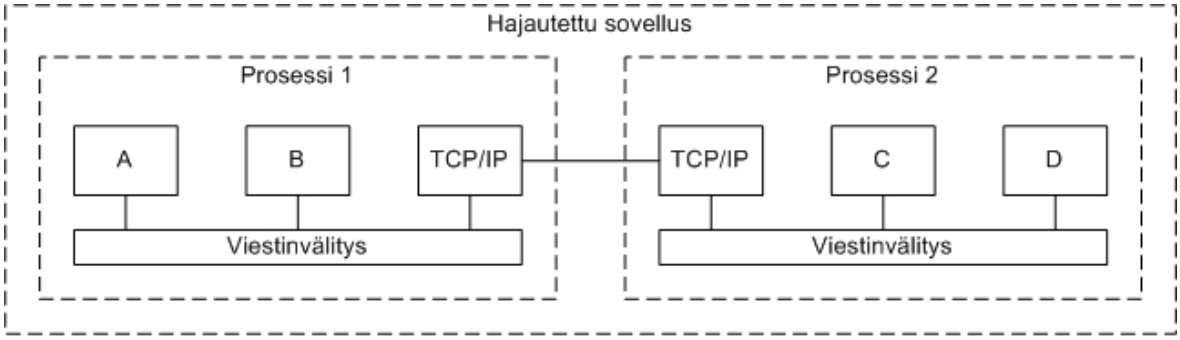
Tässä luvussa esitellään käytännön osuuden perustana toimiva, jo aiemmin kehitetty hajautettu sovelluskehys, sekä pohditaan sen hyviä ja huonoja puolia. Lisäksi pohditaan sovelluskehyksen graafisille käyttöliittymille asettamia vaatimuksia. Jatkossa tässä tutkielmassa termillä sovelluskehys tarkoitetaan tätä kyseistä sovelluskehystä, jonka yhteyteen käyttöliittymiä kehitetään tutkimuksen käytännön osuudessa.

5.3.1 Esittely

Kyseinen hajautettu sovelluskehys on tarkoitettu pohjaksi sovelluksille, joiden tehtävänä on datan haku erilaisilta sensoreilta, datan simulointi ja sen analysointi. Kehyksen rakenne perustuu luvussa 3.3 esiteltyyn viestinvälitysarkkitehtuuriin, jossa viestit välitetään julkaisija-tilaaja-mallin mukaisesti. Kehys sisältää siis pääasiassa komponenttien lataamisen sekä viestien välityksen näiden välillä. Sovelluskehyksen arkkitehtuuria on havainnollistettu kuvassa 5.2, jossa sovelluskehystä on erikoistettu neljällä komponentilla: A, B, C ja D.

Sovelluskehys toimii hajautettuna järjestelmänä siten, että eri koneille käynnistetyt sovellukset kommunikoivat keskenään tietoliikennekomponentteja apuna käyttäen. Tietoliikennekomponenttien avulla verkkoyhteydet saadaan läpinäkyviksi sovelluskehykselle ja tietovuoarkkitehtuurin mukaisia datayhteyksiä voidaan konfiguroida komponenttien välille niiden sijainnista välittämättä. Kuvan 5.2 sovellus on hajautettu eri koneilla toimiviin prosesseihin sovelluskehyksen avulla. Kommunikaatio koneiden välillä hoidetaan kehykseen kuuluvien TCP/IP-komponenttien avulla.

Komponentit toteutetaan sovelluskehystä varten kehitetyn komponenttikehyksen

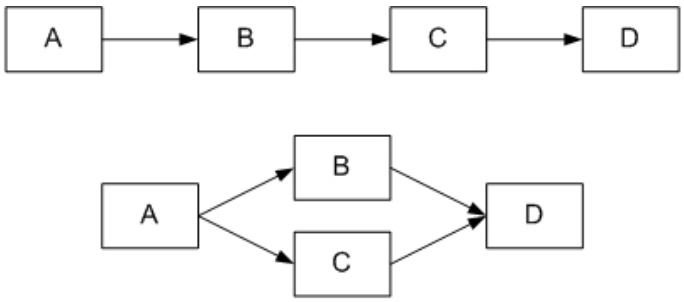


Kuva 5.2: Sovelluskehyksen viestinvälitysarkkitehtuuriin perustuva rakenne.

mukaisiksi. Komponenttien rajapinta viestinvälitykseen rajoittuu yhteen viestejä käsittelevään funktioon sekä viestien julkaisemisesta huolehtivaan toimintoon. Komponentit liitetään sovelluskehykseen ajonaikaisesti käyttäen *dynaamista sidontaa (dynamic binding)*. Sovelluksen kontrolli säilyy koko sovelluksen ajon ajan kehyksellä.

Varsinainen sovelluksen toiminnallisuus toteutetaan konfiguroimalla viestiyhteyksiä komponenttien välille. Näin ollen sovelluskehyksen toiminnallisuus noudattaa *tietovuoarkkitehtuuria (pipes-and-filters architecture)* [4, s. 96-97], jossa konfiguraatiota voidaan muuttaa ajonaikaisesti. Tietovuoarkkitehtuuri yksinkertaistaa kommunikaatiota, sillä siinä komponenttiin tarvitaan rajapinnat vain sisään ja ulos kulkeville viesteille. Interaktiivisten järjestelmien rakentaminen tietovuoarkkitehtuuria käyttäen on kuitenkin vaikeaa [4, s. 97]. Sovelluskehyksen toiminnan perustumista tietovuoarkkitehtuuriin on havainnollistettu kuvassa 5.3, jossa on esitetty kaksi mahdollista vaihtoehtoa kuvan 5.2 sovelluksen konfigurointiin.

Sovelluskehyksen toimintaa voidaan sanoa *datakeskeiseksi (data-centered, data centric)* [4, s. 95], sillä keskenään kommunikoivat komponentit eivät ole tietoisia toisistaan, vaan ainoastaan viestien sisällä kulkevasta datasta.



Kuva 5.3: Sovelluskehyksen ajonaikainen, toiminnallinen rakenne perustuu tietovuoarkkitehtuuriin.

Viestinvälityksen viestinä toimii binäärinen datapaketti. Tieto talletetaan pakettiin avain-arvo-pareina, joissa avaimena toimii aina merkkijono. Arvo voi olla datatyypiltään esimerkiksi string tai double. Arvo voi olla myös toinen datapaketti, joten puurakenteen luominen paketin sisälle on mahdollista.

Sovelluskehys on toteutettu C++-kielellä.

5.3.2 Kehyksen hyviä ja huonoja puolia

Loppukäyttäjää ajatellen sovelluskehyksen hyviin puoliin kuuluvat sen konfiguroitavuus, dynaaminen päivitettävyys sekä hajautuksen monipuoliset mahdollisuudet. Uuden tietoliikennetyypin lisäämiseen kahden verkon välille riittää vastaavien tietoliikennekomponenttien lisääminen sovelluskehykseen. Yhteydet voidaan konfiguroida yksisuuntaisiksi tapauksissa, joissa käsitellään turvaluokiteltuja erillisverkkoja. Äärimmäisessä tapauksessa yhteyden ylläpito onnistuu jopa massamuistin avulla, jolloin verkkojen välillä ei tarvita minkäänlaista pysyvää yhteyttä. Näin ollen järjestelmän hajautus saadaan toimimaan erittäin vaikeissakin olosuhteissa.

Loppukäyttäjän kannalta äärimmäisestä joustavuudesta seuraa myös huonoja puolia, sillä monimutkaisemmassa sovelluksessa tarvittavan konfiguroinnin määrä kasvaa niin suureksi, että sen hallitseminen vaatii koulutusta ja järjestelmän tuntemista. Myös järjestelmän reaaliaikaisuus kärsii, jos hajautukseen sisältyy myös hitaampia tietoliikennetyyppejä.

Sovelluskehittäjän kannalta huonona puolena voidaan pitää oppimisvaikeuksia sovelluskehyksen käytön alkupuolella. Uuden ohjelmoijan on ennen sovelluskehyksen käyttöä tutustuttava datapakettien käsittelyyn, komponenttikehykseen sekä sovelluskehyksen periaatteisiin ja toimintatapoihin. Komponentin ohjelmoinnin harjoitteluun tarvitaan koko sovelluskehyksen asennus ja konfigurointi kyseiselle koneelle. Komponenttien kehityksen lisäksi on opeteltava viestinvälityksen konfiguroinnin periaatteet, joita käyttäen uusi komponentti saadaan toiminnallisessa mielessä liitettyä jo olemassa oleviin komponentteihin. Vasta tämän jälkeen voidaan testata varsinaisen sovelluksen toimintaa.

Ensimmäisen komponentin kehityksen jälkeen kynnys kuitenkin laskee huomattavasti, joten sovelluskehyksen hyödyt tulevat ohjelmoijalle ilmi vasta tässä vaiheessa. Uuden sovelluksen kehitykseen tarvittava ohjelmakoodin määrä vähenee valmiita komponentteja käyttämällä, ja testattavaksi jää pelkästään oman komponentin osuus. Sovelluksen eri osia voidaan kehittää toisistaan riippumatta, sillä niitä voidaan testata itsenäisesti käyttämällä datan simulointiin tarkoitettuja komponentteja. Binäärinen datan käsittely mahdollistaa alusta- ja kääntäjäriippumattoman toteutuksen. Hajautus

onnistuu konfigurointia muuttamalla ja päivitysten asennus testikoneelle vaatii vain päivitettyjen dynaamisten kirjastojen siirtoa.

Uusia komponentteja suunniteltaessa on kuitenkin aina otettava huomioon niiden tuleva uudelleenkäyttö, joten sovelluskehityksen painopiste siirtyy automaattisesti ohjelmoinnista suunnitteluun, mikä puolestaan parantaa loppusovellusten laatua yleisellä tasolla. Sovelluskehyksen tapauksessa suunnitteluvaiheessa on huomioitava erityisesti viestiyhteyksien konfigurointi, joka sovelluksen kasvaessa muuttuu aina monimutkaisemmaksi. Konfiguroinnin monimutkaisuuden ja dynaamisuuden takia varsinaisen tuotantoversion *täydellinen* lopputestaus on lähes mahdotonta, sillä se vaatisi kaikkien mahdollisten konfiguraatiokombinaatioiden testausta.

5.3.3 Sovelluskehyksen vaikutukset graafisiin käyttöliittymiin

Sovelluskehyksen viestinvälitysarkkitehtuuri tukee hyvin graafisia käyttöliittymiä, joiden kommunikointi perustuu erilaisten, luvussa 4 luokiteltujen viestien asynkroniseen välitykseen.

Sovelluskehyksen toiminnallisuuden aikaan saava tietovuoarkkitehtuurin mukainen ratkaisu on jossain määrin ristiriidassa graafisten käyttöliittymien viesteihin nähden. Vastoin tietovuoarkkitehtuurin periaatetta ohjelmalogiikan ja sille kuuluvan käyttöliittymän välisen sidoksen on oltava varsin vahva. Esimerkiksi käyttöliittymäkomponentin päivitysviesti on suunnattava jollekin tietylle käyttöliittymäkomponentille eikä sitä voida käyttää tietovuossa datakeskeisenä, kenelle tahansa ohjattavana viestinä. Sidosta voidaan kuitenkin pyrkiä heikentämään suunnittelemalla käyttöliittymäkomponenttien rajapinnat mahdollisimman yleiskäyttöisiksi.

Tietovuoarkkitehtuuria voidaan joissain määrin myös hyödyntää graafisia käyttöliittymiä kehitettäessä: mahdollisuus logiikkakomponenttien jakamiseen ja ajonaikaiseen vaihtoon tarjoaa käyttöliittymille uudenlaista dynaamisuutta. Myös käyttöliittymiä koskevien viestien ohjaaminen edelleen tarjoaa uusia mahdollisuuksia, kuten esimerkiksi käyttöliittymän näkymän kopioimisen toiseen käyttöliittymään.

Sovelluskehys asettaa joitakin vaatimuksia käyttöliittymien toteutukselle:

- *Riippumattomuus ikkunointijärjestelmästä.* Koska sovelluksia on kyettävä käyttämään käyttöjärjestelmästä riippumatta, myös käyttöliittymät on rakennettava siten, että niitä voidaan käyttää eri ikkunointijärjestelmissä.

- *Dynaaminen sidonta.* Sovelluskehyksen luonteeseen kuuluu komponenttien dynaaminen lataaminen, joten dynaaminen ladattavuus on olennainen vaatimus myös käyttöliittymien ollessa kyseessä.

- *Datavirtojen ajonaikainen määrittely.* Sovelluskehyksen rakenne perustuu siihen, että pakettien välitystä komponenttien välillä voidaan muuttaa ajonaikaisesti. Tämä ominaisuus tulisi olla käytettävissä myös graafisissa käyttöliittymissä, esimerkiksi erilaisia monitorointinäyttöjä tulisi pystyä lisäämään ja vaihtamaan ajonaikaisesti.

6 Graafiset käyttöliittymät sovelluskehyksessä

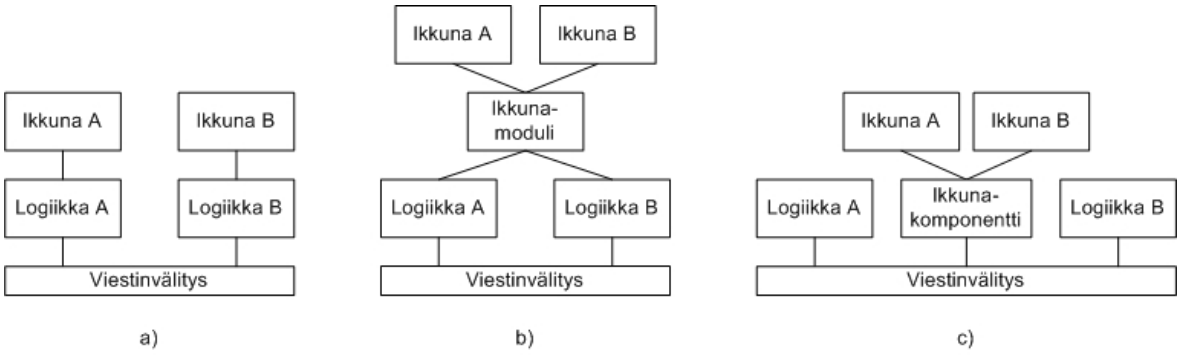
Tutkimustyön pohjalta toteutettiin käyttöliittymämalli luvussa 5.3 esiteltyyn hajautettuun sovelluskehykseen. Toteutus sisältää käyttöliittymien integroinnin kehykseen ja rajapinnan uusille käyttöliittymäkomponenteille. Varsinaisia uusia käyttöliittymäkomponentteja kehitettiin testaustarkoituksessa vain muutama.

Tässä luvussa esitellään toteutettu järjestelmä kokonaisuudessaan. Luvussa 6.1 pohditaan mahdollisia vaihtoehtoja toteutuksen rakenteelle, jonka jälkeen luvussa 6.2 esitellään pääasialliseksi työkaluksi valittu Qt. Viestinvälityksessä kuljetettavien viestien sisällön rakenne esitellään luvussa 6.3. Lopullinen toteutus, sekä käyttöliittymäkomponenttien tekotapa esitellään luvussa 6.4. Lopuksi luvussa 6.5 tarkastellaan valitulla tavalla toteutettujen käyttöliittymien kehityksen kannalta riskialtteinta osaa eli nimeämiskäytäntöjä ja niiden aiheuttamia ongelmia.

6.1 Toteutuksen rakennevaihtoehdot

Kyseisessä sovelluskehyksessä logiikkakomponentit käyttävät omia ikkunoitaan yhteisen prosessin sisällä. Tällä tavalla sekä ohjelmalogiikkakomponentteja että niiden ikkunoita voidaan muokata ja lisätä, ilman että siitä aiheutuu muutoksia muihin komponentteihin. Suoraviivainen tilanne, jossa kukin komponentti hallitsee omaa ikkunaansa on havainnollistettu kuvan 6.1 kohdassa a.

Tässä luvussa pohditaan mahdollisuuksia irrottaa ikkunointi logiikkakomponentista. Tällä pyritään pienentämään logiikkakomponentin ja käyttöliittymän välistä sidos-



Kuva 6.1: Käyttöliittymien hajautus viestinvälityksessä.

ta, jotta järjestelmään voitaisiin toteuttaa muun muassa työkalusta riippumattomia graafisia käyttöliittymiä. Ensimmäinen ratkaisuvaihtoehto on irrottaa ikkunointi omaksi modulikseen (luku 6.1.1), ajatusta jatketaan pohtimalla, voitaisiinko modulista tehdä kokonaan erillinen sovelluskehyksen komponentti (luku 6.1.2). Molemmat ratkaisut toteutettiin ja niiden välinen vertailu löytyy luvusta 6.4.4.

6.1.1 Ikkunamoduli

Ikkunoiden avauksen ja ikkunan viestien käsittelyn sisällyttäminen suoraan komponenttiin aiheuttaa saman toiminnallisuuden toistumista jokaisessa ikkunaa käyttävässä dynaamisesti ladattavassa komponentissa, joten se eristetään omaksi dynaamisesti ladattavaksi modulikseen. Erillisen ikkunoita käsittelevän modulin sisään voidaan kapseloida komponentilta tulleiden käskyjen siirto käyttöliittymän viestijonoon ja edelleen käyttöliittymäsäikeen viestisilmukkaan, jolloin logiikkakomponentin ei enää tarvitse vastata säieturvallisuuden säilyttämisestä käyttöliittymän käsittelyssä (kts. luku 2.5). Lisäksi käyttöliittymien modularisointi vähentää logiikkakomponentin ja käyttöliittymän keskinäistä sidosta, eikä logiikan tarvitse olla tietoinen muusta kuin modulin rajapinnasta.

Komponenttien ikkunat saattavat sijaita keskenään samassa prosessissa. Jos ikkunat on toteutettu samalla työkalulla tulee niiden käsittelyssä vastaan työkalukohtaisia vaatimuksia. Yksi näistä on alustariippumattomien työkalujen vaatimus viestisilmukan yksiselitteisyydestä [6], joka johtuu siitä, että X-ikkunapalvelin varaa jokaiselle prosessille vain yhden viestijonon. Tämä tarkoittaa sitä, että kaikki ikkunat saman prosessin sisällä jakavat yhteisen globaalin käyttöliittymäsäikeen ja viestisilmukan. Jos ikkunoiden avaus liitetään suoraan komponenttien yhteyteen, viestisilmukan olemassaolo on testattava ennen käyttöä ja siitä huolehtivan komponentin määrittäminen vaikeutuu. Yhteisen modulin käyttö korjaa nämä ongelmat vastaamalla työkalukohtaisten globaalien muuttujien ylläpidosta.

Globaalien käyttöliittymäsäikeiden tarve estää modulin kopioimisen erikseen jokaisen komponentin käyttöön, joten moduliin on lisättävä tuki useamman eri ikkunan käsittelylle. Yhteiskäyttöisen modulin sisälle voidaan haluttaessa lisätä tuki useamman erilaisen työkalun käyttöliittymäsäikeille siten, että rajapinta logiikkaan pysyy yhtenäisenä. Näin ollen samaa logiikkaa voidaan käyttää useiden eri työkaluilla kehitettyjen käyttöliittymien kanssa.

Ikkunamodulin käyttö voidaan toteuttaa siten, että komponentit lataavat sen tarvittaessa käyttöönsä. Kuvan 6.1 kohdassa b on esitettynä ratkaisua vastaava arkkitehtuuri. Suora yhteys ikkunamoduliin on riippumaton viestinvälityksen kuormituksesta.

6.1.2 Ikkunakomponentti

Luvussa 6.1.1 esitellystä ikkunamodulista voidaan myös tehdä erillinen hajautetun järjestelmän komponentti jolloin viestit logiikkakomponentin ja ikkunakomponentin välillä konfiguroidaan kulkemaan viestinvälityksen kautta. Tämä vaihtoehto on havainnollistettu kuvan 6.1 kohdassa c. Rakenteensa perusteella ikkunakomponentti vastaa luvussa 2.1 esiteltyä X Serveriä ja luvussa 3.3.2 esiteltyä OAA-järjestelmän käyttöliittymäagenttia. Ikkunakomponentin käyttö lisää joustavuutta, ja sillä saavutetaan mm. seuraavia etuja:

- Poistetaan ikkunan lataus logiikkakomponentista. Samalla päästään eroon kontrolliristiriidasta sovelluskehyksen kanssa (luku 5.2.2), sillä ikkunakomponenttia käytettäessä myös ristiriitatilanne on eristetty sinne. Logiikkakomponenteille kaikki viestit tulevat viestinvälityksen kautta, joten niiden kontrollointi tapahtuu kokonaisuudessaan sovelluskehyksestä käsin.
- Mahdollistetaan tilanne, jossa ohjelmalogiikka ja käyttöliittymä voidaan sijoittaa eri koneille.
- Mahdollistetaan yhden käyttöliittymän logiikan hajauttaminen useammalle erilliselle logiikkakomponentille.
- Mahdollistetaan viestinvälityksen hyödyntäminen muun muassa testauksen automatisoinnissa ja käyttöliittymämakrojen luonnissa, sillä viestinvälityksessä käyttöliittymän viestit voidaan nauhoittaa ohjaamalla ne lisäksi erilliselle tähän tarkoitukseen rakennetulle komponentille. Ajamalla nauhoitetut käyttöliittymäviestit uudelleen voidaan simuloida käyttäjän toimenpiteitä testausta varten.

Ikkunakomponentin käytöstä seuraa myös ongelmia, sillä käyttöliittymäviestien ajo viestinvälityksen kautta asettaa suuria vaatimuksia viestinvälitykselle. Käyttöliittymien vasteajat eivät saisi kärsiä huomattavasti suuressakaan kuormituksessa, joten ne tulisi priorisoida muuta viestinvälitystä korkeammalle tasolle.

6.2 Qt

Pääasialliseksi työkaluksi käyttöliittymien kehitykseen valittiin Qt. Qt on vuodesta 1994 toimineen Trolltechin [34] kehittämä ohjelmisto, josta on julkaistu sekä kaupallinen että GPL:n [11] alainen vapaa versio. Qt:ta käyttävät kaupallisella puolella muunmuassa Adobe, IBM ja NASA [35]. Avoimella puolella KDE [15] on Qt:lla kehitetty ja Trolltechin tukema.

Valintaperusteena käytettiin seuraavia seikkoja:

- Qt on ohjelmoitu C++-kielellä.
- Varustettu korkeatasoisella dokumentoinnilla.
- Oliokeskeinen ja selkeä, joten myös avoimen lähdekoodin seuraaminen onnistuu kohtuullisen helposti.
- Käyttöliittymien suunnitteluun saa asennuksen mukana kehitysympäristön (Qt Designer [36]).
- Työkalupakille on tarjolla myös kaupallinen tuki.

Tässä luvussa esitellään toteutuksen yhteydessä tarpeellisiksi havaittuja Qt:n ominaisuuksia. Qt:n tapahtumien käsittely esitellään luvussa 6.2.1, dynaamiseen latautukseen löytyvät keinot luvussa 6.2.2 ja lopuksi joitakin Qt:n erityispiirteitä luvussa 6.2.3.

6.2.1 Tapahtumien käsittely Qt:ssa

Käyttöliittymäkomponenttien välinen viestinvälitys Qt:ssa perustuu takaisinkutsujen sijaan *signaaleihin (signal)* ja *vastaanottimiin (slot)* [38]. Signaali vastaa tarkkailijamallin (luku 4.5.1) subjektin `notify`-funktiota ja vastaanotin tarkkailijan `update`-funktiota.

Signaalit ja vastaanottimet voidaan *kytkeä (connect)* tai irroittaa toisistaan ajonaikaisesti. Kytkentä on tyypiltään *löyhä kytkentä (loose coupling)*, jossa kumpikaan osapuoli ei tiedä toisesta. Kytkentää tehtäessä tarkastetaan parametrilistat eikä tyypiltään erilaisia funktioita kytketä. Näin ollen kytkentä on tyyppiturvallinen eikä tyyppikonversioita tarvitse vastaanottimissa tehdä. Signaaleja ja vastaanottimia voidaan tarvittaessa lisätä Qt:n käyttöliittymäkomponenteista perittyihin luokkiin. Harvemmin tarvittavaa signaaleja matalamman tason tapahtumien tarkkailua varten Qt:ssa voi lisätä käyttöliittymäkomponenteille *tapahtumasuodattimia (event filter)*.

Qt:n stardandin mukaista signaalin ja vastaanottimen yhdistämistä ei voida käyttää kun ohjelmalogiikka halutaan erottaa käyttöliittymästä hajautetun järjestelmän erillisiin moduleihin. Signaalit on lähetettävä ohjelmalogiikalle ja varsinaiset vastaanottimien kutsut on tehtävä dynaamisesti ohjelmalogiikalta saatujen viestien perusteella. Dynaamiseen vastaanottimen kutsuun voidaan käyttää Qt:n `QMetaObject`-luokkaa ja siitä löytyvää `invokeMethod`-metodia. Seuraava koodiesimerkki toteuttaa vastaanottimen kutsun viestijonon kautta. Se hakee käyttöliittymäkomponentista `widget` ensimmäisen `QTextEdit`-tyyppisen lapsen nimeltä `textEditViesti`, etsii tältä metodin

append ja laittaa kutsun tälle metodille QString-tyyppisellä parametrilla str odottamaan tapahtumajonoon. Seurauksena merkkijono str lisääntyy tekstikentän loppuun. Paluuarvo on tyyppiä bool; true jos metodin kutsu onnistui, muuten false.

```
QMetaObject::invokeMethod(
   widget->findChild<QTextEdit*>("textEditViesti"),
   "append",
   Qt::QueuedConnection,
   Q_ARG(QString, str));
```

6.2.2 Dynaaminen lataus

Dynaamiseen ikkunoiden lataukseen käytetään Qt:ssa QUiLoader-luokkaa. Tarvittava xxx.ui-tiedosto on XML-muotoinen ja sitä käytetään vain ikkunan luontivaiheessa, joten tiedosto on ajonaikaisesti päivitettävissä. Seuraava ohjelmakoodi lataa Qt:n kehitysympäristön avulla suunnitellun QWidget-luokasta perityn ikkunan muuttujaan widget ja kutsuu sen show-metodia, joka piirtää ikkunan näytölle:

```
QUiLoader loader;
QFile file("testwidget.ui");
file.open(QFile::ReadOnly);
QWidget *widget = loader.load(&file);
file.close();
widget->show();
```

Käyttöliittymäkomponenttien lisääminen kehitysympäristöön onnistuu perimällä rajapintana toimiva *latausluokka (plugin)* QDesignerCustomWidgetInterface-luokasta. Latausluokka ja uusi käyttöliittymäkomponentti käännetään dynaamisesti ladattavaksi kirjastoksi, joka sijoitetaan Qt:n plugins/designer-hakemistoon. Tällä menetelmällä uusia komponentteja voidaan käyttää kehitysympäristössä valmiiden komponenttien tapaan raahaamalla suoraan ikkunaan.

Komponenttien ajonaikainen päivitys ei suoraan onnistu, sillä dynaaminen kirjasto on käytön aikana lukittu. Tämä voidaan kuitenkin tarvittaessa kiertää nimeämällä kirjastot version mukaan ja päivittämällä ikkunan käyttämää versiota ennen uutta latausta.

6.2.3 Qt:n erityispiirteet

Qt:n erityispiirteiden, kuten signaalien, vastaanottimien ja QObject-pohjaisten luokkien metadata-tietojen luonti ohjelmoidaan käyttämällä Qt:n makroja. Makrojen kään-

tämisestä standardin mukaiseksi C++-koodiksi huolehtii *Meta-Object Compiler (moc)*, joka ajetaan ennen varsinaista kääntäjää. Koko projektin kääntämiseen tarvittava *Makefile*-tiedosto tehdään *qmake*-komentorivityökalulla `xxx.pro`-projektitiedostosta. Makefile sisältää myös tarvittavan moc-käännöksen.

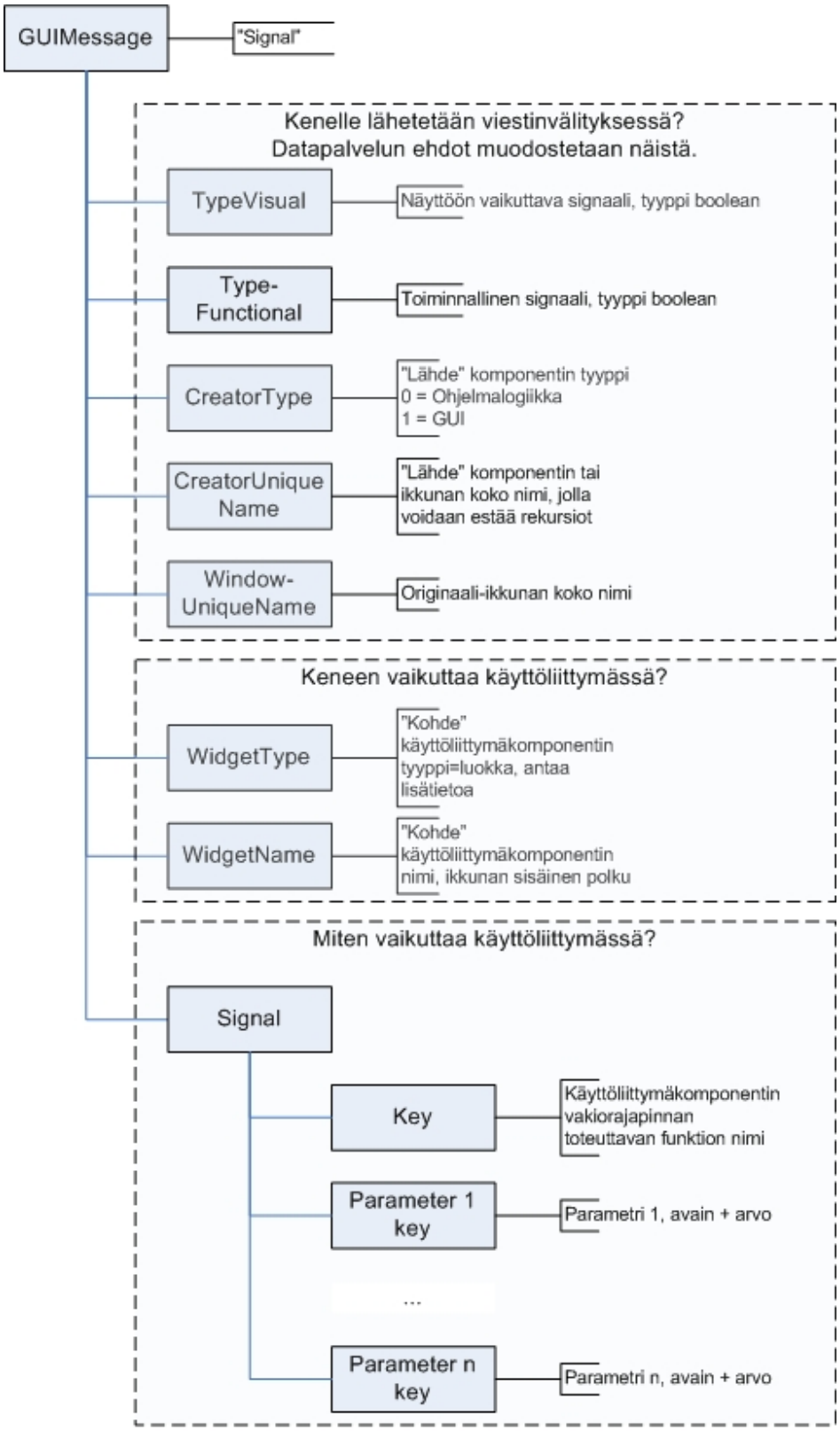
Qt:ssa yhdessä prosessissa voi käyttää vain yhtä globaalia käyttöliittymäsäiettä joka sisältää prosessin ainoan viestisilmukan. Kaikki käyttöliittymiä koskevat operaatiot on suoritettava tässä säikeessä. Käytännössä käyttöliittymäkomponentteihin liittyvät kutsut toisesta säikeestä hoidetaan asettamalla ne viestijonoon, josta käyttöliittymäsäie ehtiessään suorittaa ne. Viestisilmukan ollessa kiireinen eli esimerkiksi hiiren nappulan ollessa pohjassa toisarvoisten tapahtumien päivitys taukoaa. Qt:n tapauksessa tämä tarkoittaa kaikkien kyseisen prosessin Qt-ikkunoiden päivittymisen pysähtymistä.

6.3 Viestirakenne

Käyttöön valittu viestirakenne on esitetty graafisesti kuvassa 6.2. Kuva esittää viestin sisältämien avainkenttien muodostamaa puurakennetta, jossa jokainen laatikko kuvaa viestiin sisältyvää avainta. Laatikoihin liitetyt kommentit kertovat tarkemmin avaimeen liitettävän arvon sisällöstä. Käyttöön valittu viestirakenne jakaantuu kahteen osaan: *viesti-* ja *ohjausosaan*, jotka esitellään tässä luvussa. Molemmissa tärkeään osaan nousee nimeämiskäytäntö, jota on pohdittu tarkemmin luvussa 6.5.

1. Graafisessa esityksessä alimmainen katkoviivalla eristetty laatikko sisältää *viestiosan*. Viestiosa sisältää käyttöliittymän ja ohjelmalogiikan välillä kulkevat tapahtumat, komponenttitapahtumat ja komennot. Koska tapahtumien ja komponenttitapahtumien välinen raja vaihtelee työkalun mukaan, yhtenäisen viestiosan muodostamiseksi on käytetty signaali-käsitettä. Signaali koostuu avainmerkkijonosta ja parametreista. Avain ilmaisee signaalin merkityksen, parametrit sen välittämän tiedon. Samalla signaali-rakenteella voidaan välittää myös komennot. Valitsemalla toisiaan vastaavien signaalien ja komentojen avaimet samoiksi saadaan aikaiseksi yleispätevä viestiosa jota voidaan käyttää molempiin suuntiin. Suunnan selvitykseen käytetään tarvittaessa ohjausosaa.

2. Viestiosan lisäksi viestirakenteeseen kuuluu graafisen esityksen kaksi ylintä laatikkoa käsittävä *ohjausosa*. Se sisältää tiedot signaalin/komennon luojasta, tyypistä (visuaalinen/funktionaalinen), ikkunasta ja käyttöliittymäkomponentista. Ohjausosan avulla voidaan päätellä onko kyseessä signaali vai komento, ja mihin se on viestinvälityksessä lähetettävä. Esimerkiksi pelkästään visuaalisia, ikkunassa syntyneitä viestejä ei tarvitse lähettää ohjelmalogiikalle.



Kuva 6.2: Viestirakenne.

6.4 Käyttöliittymien toteutus

Käyttöliittymien rakenne toteutettiin alustana toimivan viestinvälitysjärjestelmän päälle luvussa 6.1 esiteltyä ikkunamodulin mallia ja laajennettuja käyttöliittymäkomponentteja käyttäen. Pääasiallisena tavoitteena pyrittiin modulin käyttöön ikkunakomponentin kautta sen tuomien, luvussa 6.1 esitettyjen lisämahdollisuuksien takia, joten molempia vaihtoehtoja testattiin toteutuksen valmistuttua (luku 6.4.4).

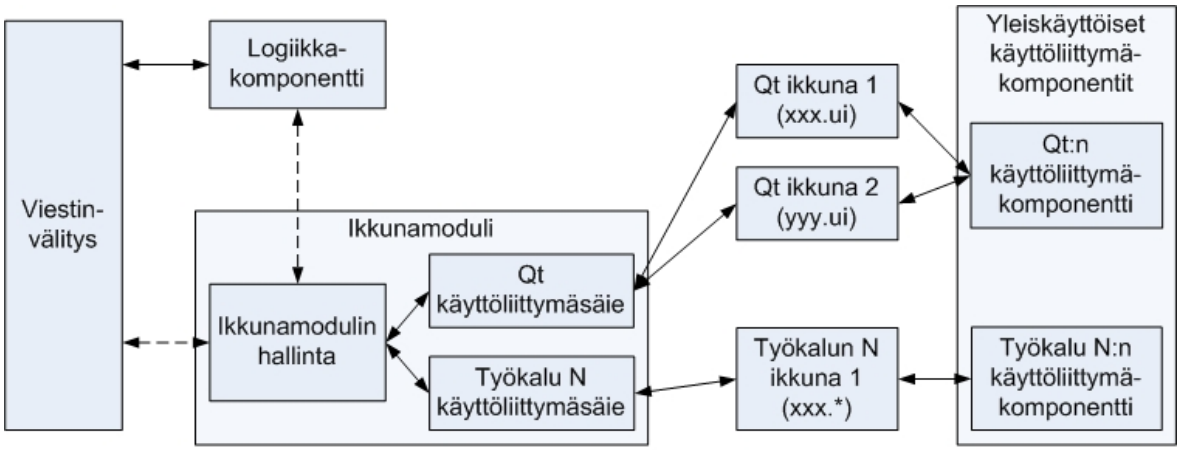
Ikkunamoduli kapseloi käyttöliittymäsäikeisiin liittyvän toiminnallisuuden ja käyttöliittymien liittämisen viestinvälitykseen. Sen tehtäviin kuuluvat näin ollen käyttöliittymäsäikeiden käynnistäminen ja pysäyttäminen, ikkunoiden avaaminen, viestien välitys oikealle ikkunalle ja ikkunoilta saapuvien viestien toimitus viestinvälitykseen. Moduliin toteutettiin mahdollisuus avata useita ikkunoita ja lähettää niille viestejä ikkunan yksilöllisen nimen perusteella. Rakenne suunniteltiin siten, että eri työkaluja varten toteutettuja käyttöliittymäsäikeitä voidaan lisätä muuttamatta olemassaolevaa rakennetta. Ikkunamodulin luokkarakenne esitellään tarkemmin luvussa 6.4.1.

Varsinainen käyttöliittymän toiminnallisuus tullaan toteuttamaan laajentamalla työkalupakin käyttöliittymäkomponenteista uudet, ikkunamodulin kanssa yhteensopivat käyttöliittymäkomponentit, joiden tekotapa esitellään tarkemmin luvussa 6.4.2. Varsinaiset käyttöliittymät tullaan kokoamaan näistä komponenteista ja ladataan käyttöön ikkunamodulin avulla. Näin saadaan aikaiseksi automaattisesti syntyvä yhteys ikkunan käyttöliittymäkomponenttien ja viestinvälityksen välille. Alkuun käyttöliittymien tekeminen tällä tavalla tulee olemaan varsin työlästä, sillä kaikki uudet komponentit on toteutettava ennen käyttöä. Tuoteperheen kasvaessa työmäärä kuitenkin vähenee huomattavasti, kun valmiita, automaattisesti järjestelmään liittyviä komponentteja voidaan hyödyntää useammassa sovelluksessa.

Ikkunamodulin suhde käyttöliittymäkomponentteihin ja ikkunatiedostoihin on havainnollistettu karkealla tasolla kuvassa 6.3. Myös eri työkaluille toteutettujen käyttöliittymäsäikeiden sijainti ja erilaiset mahdollisuudet liittää ikkunamoduli viestinvälitysjärjestelmään on sisällytetty kuvaan. Logiikkakomponentti on kuvassa merkitty vain yhdellä laatikolla, mutta koska mahdollisuudet ovat monipuolisemmat, mainitaan niistä joitakin luvussa 6.4.3.

6.4.1 Ikkunamodulin luokkarakenne

Tässä luvussa tarkastellaan toteutetun ikkunamodulin sisäistä rakennetta luokkatasolla. Tarkastelu aloitetaan kokoamalla modulin erilaiset, toiminnalliset osat ja niiden tehtävät, joiden perusteella käytetyt luokat ryhmitellään luokkakaavioon. Lisäksi luokkien suhteita havainnollistetaan esittämällä ikkunan avaus sekvenssikaaviolla.



Kuva 6.3: Ikkunamoduli hajautetussa järjestelmässä. Katkoviivat kuvaavat vaihtoehtoisia tapoja liittää moduli järjestelmään.

Ikkunamodulin toiminnalliset osat voidaan jakaa neljään rooliin: *modulin hallinta*, *käyttöliittymäsäikeet*, *ikkunat* ja *ikkunan viestien käsittely*. Toimintojen jakaantumista näiden osien kesken voidaan tarkentaa seuraavasti:

1. *Modulin hallinnasta* vastaa pääasiallinen ikkunamoduli `GUIModule`. Se huolehtii käytettävien luokkien luomisesta, kutsumisesta ja tuhoamisesta. Ikkunamoduli sisältää myös ulos tarjottavan rajapinnan, joka koostuu neljästä funktiosta:

   - `initialize` alustaa modulin.
   - `openWindow` avaa uuden ikkunan annetun tiedostonimen, halutun työkalupakin ja ikkunan nimen perusteella.
   - `handlePacket` vastaanottaa viestin ja ohjaa sen oikealle käyttöliittymäkomponentille.
   - `finalize` sulkee auki olevat ikkunat, pysäyttää käynnissä olevat säikeet ja vapauttaa varatun muistin.

2. *Käyttöliittymäsäikeet* huolehtivat käyttöliittymiä koskevista toimenpiteistä. Näitä ovat ikkunan avaus ja viestin toimittaminen oikealle käyttöliittymäkomponentille. Käyttöliittymäsäikeitä käsitellään modulissa `GUIProviderThread`-rajapinnan kautta, varsinaiset käyttöliittymäsäikeet toteutetaan työkalukohtaisesti. Tässä vaiheessa toteutettiin Qt-käyttöliittymäsäikeen sisältävä luokka `QtThread`, jossa hyödynnettiin luvussa 6.2 esiteltyjä dynaamiseen lataukseen ja komponenttien ajonaikaiseen hakuun tarkoitettuja mekanismeja. Qt-säie kovakoodattiin modulin

sisään, sillä uusien työkalujen lisääminen tulee olemaan harvinaista. Kehitysmahdollisuutena eri työkaluille toteutettujen säikeiden liittämisessä voitaisiin käyttää esimerkiksi *pistoke (plugin)* -mallia.

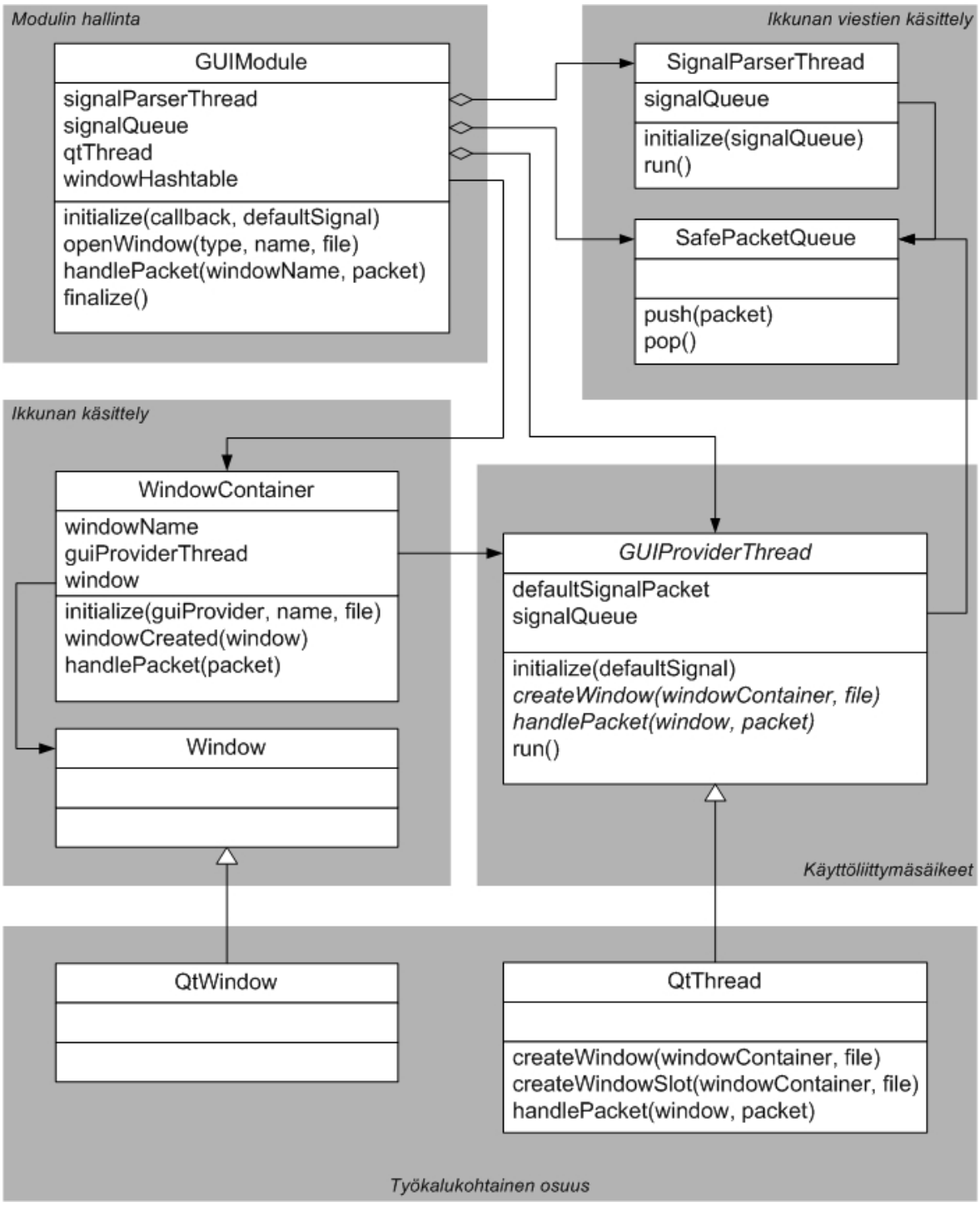
3. *Ikkunoista* huolehtivat `WindowContainer`-säilytysluokat, jotka tallentavat osoittimen ikkunan käyttöliittymäsäikeeseen sekä viitteen ikkunaan. Ikkunan viitettä varten käytetään `Window`-rajapintaa, jolloin kullekin työkalulle voidaan toteuttaa oma ikkunaluokkansa. Ikkunaluokkaan tallennetaan viittaus varsinaiseen ikkunaan, joka esimerkiksi Qt:ssa on tyyppiä `QWidget`. Ikkunamoduli tallentaa ikkunoiden säilytysluokat hajautustauluun, jossa avaimena käytetään ikkunan uniikkia nimeä.

4. *Ikkunan viestien käsittely* tarkoittaa käyttöliittymäkomponenteilta saatujen viestien toimittamista edelleen viestinvälitykseen modulin alustuksessa annettua takaisinkutsufunktiota käyttäen. Jotta useamman ikkunan asynkroniset viestit saataisiin käsiteltyä ilman viiveitä, modulin sisään toteutettiin säieturvallinen jonoluokka `SafePacketQueue` ikkunoilta saapuvia viestejä varten. Jonon ilmentymämuuttujalle annettiin nimi `signalQueue` ja sitä purkamaan toteutettiin erillinen säie `SignalParserThread`. Kaikille `GUIProviderThread`-käyttöliittymäsäikeille annetaan osoitin `signalQueue`-jonoon ja säikeiltä osoite annetaan edelleen käyttöliittymäkomponenteille ikkunan avauksen yhteydessä. Komponentit asettavat käyttäjältä vastaanotetut, viestinvälityksen vaatimaan pakettimuotoon muokatut viestit jonoon, josta ne päätyvät viestinvälitykseen `SignalParserThread`-säikeen purkaessa jonoa.

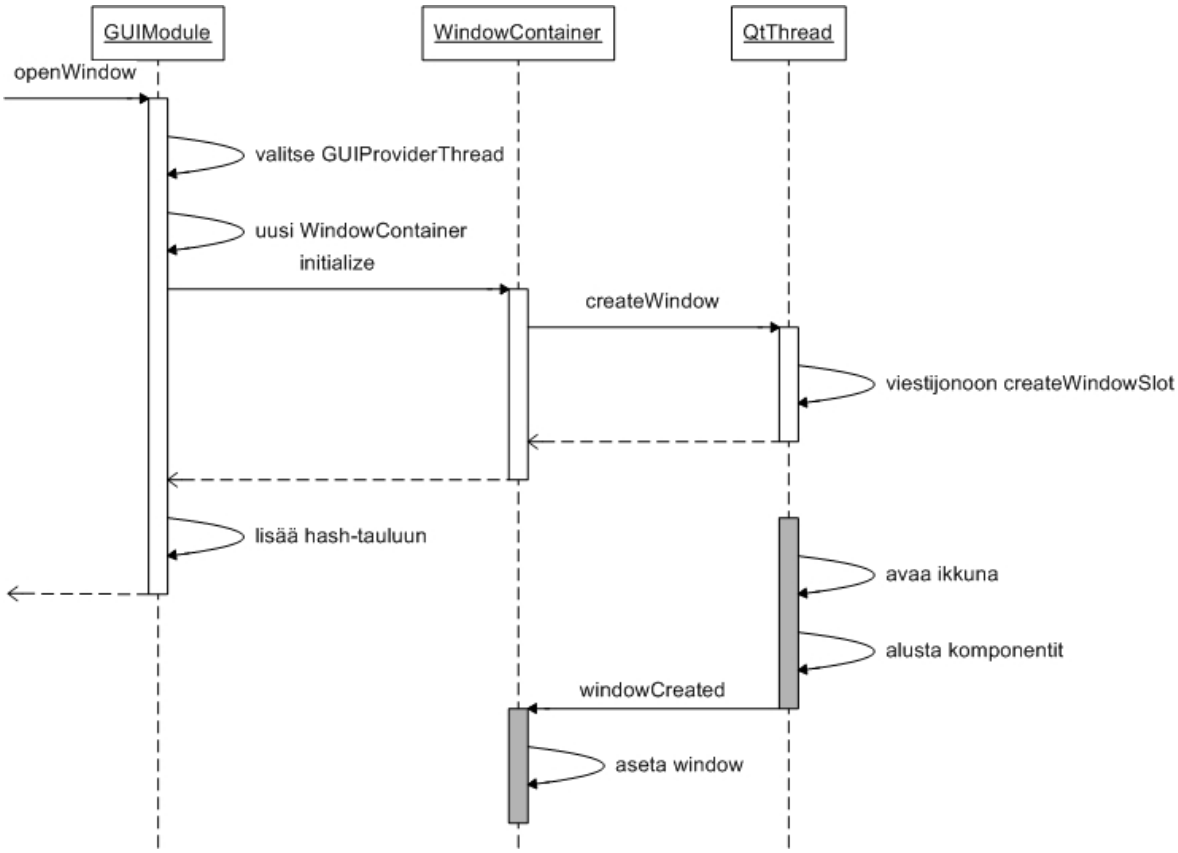
Ikkunamodulin luokkarakenne on esitetty kuvassa 6.4, jossa yllä luetellut roolit on eritelty harmailla taustalaatikoilla. Perusroolien lisäksi kuvaan on esimerkkinä lisätty työkalukohtainen osuus, jolla laajennetaan käyttöliittymäsäikeestä ja ikkunasta Qt:ta käyttävät versiot. Vastaavalla laajennuksella voidaan lisätä tuki myös muille työkaluille.

Luokkien välisiä suhteita voidaan havainnollistaa ikkunan avausta kuvaavalla sekvenssikaaviolla (kuva 6.5), joka perustuu Qt-käyttöliittymäsäikeen toteutukseen. Vastaanotettuaan avauspyynnön ikkunamoduli valitsee oikean käyttöliittymäsäikeen sekä luo ja alustaa uuden ikkunansäilytysluokan, jonka se lisää omaan hajautustauluunsa. Ikkunansäilytysluokka puolestaan kutsuu käyttöliittymäsäieluokkaansa avaamaan vastaavan ikkunan. Avauspyyntö siirretään viestijonoon, jotta ikkunan avaus toteutettaisiin käyttöliittymäsäikeessä. Avauksen jälkeen viite avattuun ikkunaan toimitetaan ikkunansäilytysluokalle.

Kuvassa eri säikeiden toteuttamat alueet on eroteltu eri väreillä: valkoinen kuvaa sovelluskehyksen säiettä ja harmaa käyttöliittymäsäiettä. Esimerkistä käy hyvin ilmi



Kuva 6.4: Ikkunamodulin luokkarakenne. Kuvassa tavallinen nuoli tarkoittaa osoitinta eli käyttösuhdetta ja salmiakkipäinen nuoli koostetta eli osasuhdetta.



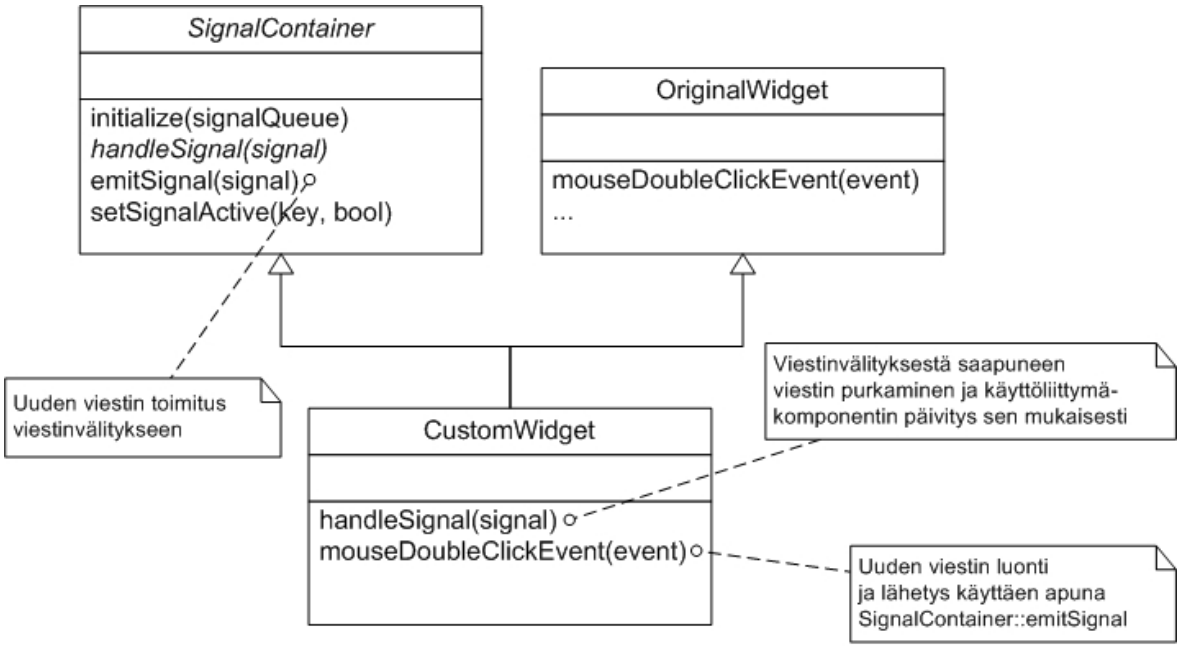
Kuva 6.5: Ikkunan avaus. Käyttöliittymäsäikeen toteuttamat osat on korostettu harmaalla.

kontrollien välinen ristiriita: ikkunaa ei voida avata sovelluskehyksen säikeestä, sillä se on saatava liitettyä osalliseksi käyttöliittymäsäikeen viestisilmukkaan.

6.4.2 Käyttöliittymäkomponentit

Tarvittavat käyttöliittymäkomponentit suunniteltiin toteutettavaksi siten, että ne liittyvät ikkunamoduliin ja sitä kautta viestinvälitysjärjestelmään yhteisen rajapinnan avulla. Käyttöliittymäsäikeet voivat välittää viestejä komponenteille suoraan rajapinnan kautta. Toiseen suuntaan viestit kuljetetaan ikkunamodulin sisältämän säieturvallisen jonon kautta, jonka osoitin annetaan käyttöliittymäkomponenteille ikkunan avauksen yhteydessä. Tässä luvussa esitellään tarkemmin komponenttien rajapinta ja omien komponenttien teko sen avulla. Lisäksi komponenttien toimintaa ikkunamodulin yhteydessä havainnollistetaan esittämällä paketin käsittely sekvenssikaavion avulla.

Käyttöliittymäkomponenttien yhteisenä rajapintana käytetään kuvassa 6.6 kaaviona esitettyä `SignalContainer`-sovitinluokkaa [10, s. 139-150]. Sovittimesta tehtiin tyy-



Kuva 6.6: Oman käyttöliittymäkomponentin tekoon käytettävä luokkasovitin.

piltään luokkasovitin, jotta alkuperäisten käyttöliittymäkomponenttien tapahtumien käsittelyyn tarkoitettuja ominaisuuksia voitaisiin hyödyntää.

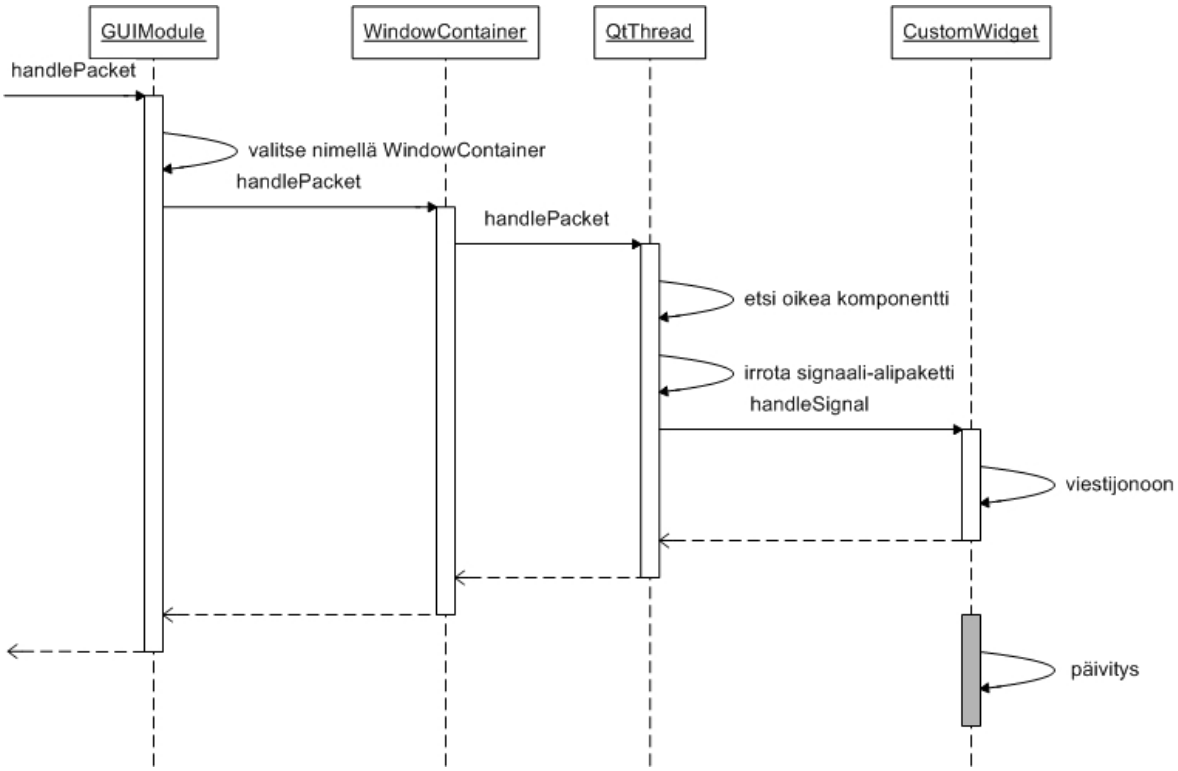
Sovittimen tärkein tehtävä on käyttöliittymäkomponentin liittäminen ikkunamoduliin eli kahteen suuntaan kulkevien viestien välittäminen. Nämä tehtävät toteutettiin `handleSignal`- ja `emitSignal`-funktioiden avulla:

- Abstrakti `handleSignal`-funktio sisällytettiin sovittimeen viestien vastaanottoa varten. Funktio on toteutettava käyttöliittymäkomponentissa, ja siinä puretaan vastaanotetuista viesteistä kaikki kyseisen komponentin ymmärtämät signaalit. Päivitystoiminnot siirretään viestijonon kautta käyttöliittymäsäikeeseen.

- Sovittimen `emitSignal`-funktiota käytetään viestien lähetykseen. Käytännössä funktion tehtävänä on laittaa viestit ikkunamodulin `signalQueue`-jonoon.

Lisäksi sovittimeen sisällytettiin lähetettävien viestien konfigurointi avaimen perusteella. Konfiguroinnilla voidaan määrätä komponentti muunmuassa jättämään osa signaaleista lähettämättä tilanteissa, joissa niitä ei tarvita. Konfiguroinnin yhteyteen tulisi lisätä myös signaalin visuaalisuus/funktionaalisuus tilanteen mukaan.

Omat käyttöliittymäkomponentit tullaan toteuttamaan moniperintää käyttäen: sovittimen perintä tuo komponenttiin rajapinnan ikkunamoduliin ja työkalupakin alkuperäisen käyttöliittymäkomponentin perintä varsinaisen toiminnallisuuden. Esimerkki



Kuva 6.7: Paketin käsittely.

omasta komponentista on kuvan 6.6 `CustomWidget`-luokka, joka perii sovittimen lisäksi työkalupakin tarjoaman `OriginalWidget`-käyttöliittymäkomponentin. Alkuperäisen käyttöliittymäkomponentin toiminnallisuudesta on mainittuna `mouseDoubleClickEvent`, joka ylikirjoittamalla voidaan reagoida tuplaklikkaukseen. Ainoa omassa komponentissa välttämättä toteutettava funktio on sovittimen abtrakti `handleSignal`.

Oman käyttöliittymäkomponentin ja ikkunamodulin välistä suhdetta on havainnollistettu sekvenssikaaviossa 6.7, joka kuvaa paketin käsittelykutsun kulkua ikkunamodulilta käyttöliittymäkomponentille. Ikkunamoduli delegoi pyynnön nimen perusteella valitsemalleen `WindowContainer`-luokalle, joka antaa sen ja ikkunareferenssin edelleen omalle `GUIProviderThread`-luokalleen. `GUIProviderThread` parsii paketista käyttöliittymäkomponentin nimen, etsii komponentin ikkunasta ja antaa paketista purkamansa signaalin tälle käsiteltäväksi `handleSignal`-rajapintafunktiota käyttäen. Käyttöliittymäkomponentissa päivitys siirretään viestijonon kautta käyttöliittymäsäikeelle, joka myös tässä kaaviossa on erotettu harmaalle pohjalle.

6.4.3 Logiikkakomponentit

Käyttöliittymien toteutukseen käytetty rakenne ei ota kantaa logiikkakomponenttien jakoon tai sijoitteluun. Rakenne sallii myös visuaalisten formalismien käytön eli käyttöliittymäkomponenttiin liittyvän logiikan ohjelmoinnin käyttöliittymäkomponentin yhteyteen.

Esimerkeissä logiikkakomponenttina on yksinkertaisuuden vuoksi käytetty yhtä koko ikkunaa hallitsevaa komponenttia. Tämä tapa kuitenkin aliarvioi järjestelmän tarjoamat resurssit siinä suhteessa, että viestinvälityksen myötä sovellus voidaan tarpeen mukaan hajauttaa tilanteen vaatimalla tavalla vaikka kuinka moneen osaan. Tämä vaihtoehto on otettu huomioon viestirakennetta ja varsinkin sen ohjausosaa suunniteltaessa. Hajautus useampaan komponenttiin monimutkaistaa kuitenkin viestinvälityksen konfigurointia ja kokonaisuuden hallintaa huomattavasti, joten hajautettuja käyttöliittymäkomponentteja toteutettaessa sille pitäisi myös olla hyvät perusteet.

6.4.4 Testaus

Ikkunamodulin toimintaa testattiin sekä sovelluskehyksen komponenttien ladattavana (6.1.1) että erillisenä ikkunakomponenttina (6.1.2), mistä johtuen myös itse modulin toteutus on sekoitus molempien käyttötapojen tarpeiden tyydyttämistä, joka sellaisenaan ei täysin palvele kumpaakaan:

- Komponenttien ladattavana modulina kyseinen toteutus toimii vain yhden sovelluskehyksen komponentin kanssa. Jos moduli haluttaisiin ladata useammasta komponentista, `signalQueue`-jono olisi siirrettävä ikkunakohtaiseksi jonoksi ikkunansäilytysluokkaan ja takaisinkutsufunktio olisi tuotava ikkunan avauksen yhteydessä modulin alustuksen sijaan.

- Ikkunakomponenttikäyttöä varten modulin päälle rakennettiin komponenttisovitin, joka huolehtii modulin lataamisesta ja ikkunan avauspyynnön purkamisesta. Tämä toteutus valittiin siksi, ettei sovelluskehys ollut ikkunamodulin kehityshetkellä kääntäjäriippumaton. Tästä johtuen komponenttisovitin täytyi toteuttaa kehyksen kääntäjällä, joka oli eri kuin ikkunamodulin kääntämiseen käytetty.

Tarkoituksena oli käyttää modulia testaukseen, jonka tuloksen mukaan voitaisiin karsia toista toteutusta varten kehitetyt piirteet pois. Ikkunakomponentin tapauksessa toteutuksen siistiminen jätetään kuitenkin odottamaan kääntäjäriippumattomuutta, jolloin koko komponentin toiminta voidaan kääntää samaan kirjastoon.

Taulukko 6.1: Testiajojen tulokset 30 mittauksen keskiarvoina.

Testi	Ei kuormaa (ms)	Kuorma 100% (ms)
Ikkunamoduli	87	287
Ikkunakomponentti	146	414
Ikkunakomponentti verkon yli	816	1110

Testiä varten rakennettiin logiikkakomponentti sovelluskehykseen sekä tekstikentän ja painonapin sisältävä ikkuna. Tarvittavat käyttöliittymäkomponentit toteutettiin luvussa 6.4.2 esitellyllä tavalla. Painonapin painallus lähettää logiikalle viestin, johon tämä reagoi lähettämällä päivitysviestin `append("kissa")` tekstikentälle. Ikkunaan lisättiin myös kuluneen ajan mittaus painonapin painalluksen ja tekstin lisääntymisen välille.

Testit tehtiin mittaamalla suoritusaikaa ensin 30 kertaa siten, että viestinvälityksen kuormitus oli normaalilla tasolla. Tämän jälkeen viestinvälityksen kuormitusta nostettiin siten, että prosessori joutui 100 prosentin kuormitukselle. Täydessä kuormituksessa mitattiin taas suoritusaikaa 30 kertaa. Testi suoritetiin kolmea erilaista komponenttijakoa käyttäen:

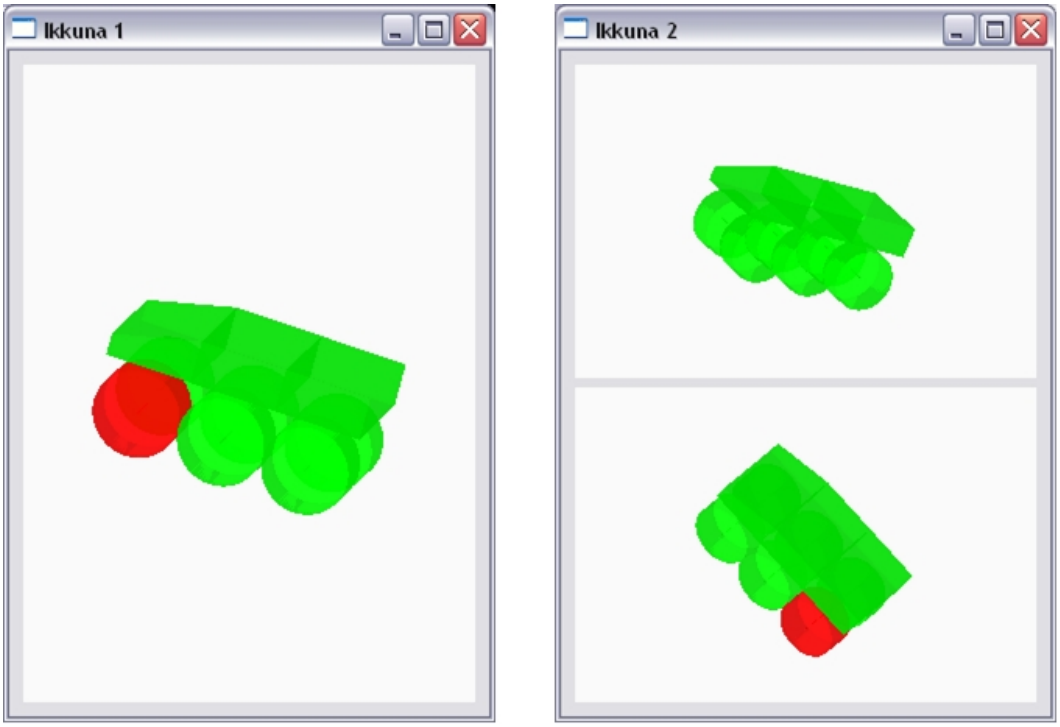
1. Ikkunamoduli ladattiin suoraan logiikkakomponentista.
2. Ikkunamoduli sijoitettiin erilliseksi komponentiksi viestinvälitykseen.
3. Logiikkakomponentti sijoitettiin eri koneelle, jolloin viestit kulkivat lisäksi erillisen TCP/IP-komponentin läpi.

Testit ajettiin Visual C++ -kehitysympäristön debuggerissa, mikä vaikuttaa tuloksiin pidentämällä suoritukseen kulunutta aikaa. Tulokset ovat kuitenkin keskenään vertailukelpoisia.

Taulukossa 6.1 esitellään testiajojen tulokset, joiden mukaan viestinvälityksen käyttö nostaa kuluneen ajan noin 1.5-kertaiseksi komponenttien lataamaan moduliin verrattuna. Ero on jopa oletettua pienempi, joten se ei aiheuttane ongelmia ikkunakomponentin käytössä. Logiikkakomponentin sijainti toisella koneella vaikuttaisi nostavan suoritusaikaa vakioarvolla verrattuna viestinvälityksen kautta ajettuun, samalla koneella sijaitsevaan logiikkakomponenttiin.

6.5 Nimeämiskäytäntö

Dynaamisen päivitettävyyden ja versioriippumattomuuden saavuttamiseksi käyttöliittymäkomponenttien nimeämisen on oltava yhtenäistä ja nimeämiskäytäntöä mietittäes-



Kuva 6.8: Käyttöliittymäkomponentin havainnollistaminen.

sä on otettava huomioon käyttöliittymäkomponenttien puumainen rakenne. Myös signaalien avainten ja parametrien nimeäminen on oltava yhtenäistä kaikissa käytetyissä työkaluissa. Havainnollistavana esimerkkinä esitetään ajoneuvokomponentin nimeäminen luvussa 6.5.1.

Nimeämiskäytännön joustavuus tuo myös sovellukseen joustavuutta, sillä sen ansiosta käyttöliittymäkomponenttia voidaan vaihtaa ilman muutoksia logiikkaan sillä ehdolla, että rajapinta ja nimi pysyvät samoina. Joustavuus tuo myös ongelmia, sillä koko sovelluksen toiminta loppuu yhtenäisyyden rikkoontuessa. Ongelma on sama kuin Taylorin C2-järjestelmässä (luku 3.3.1), ja sitä sekä mahdollista ratkaisua pohditaan luvussa 6.5.2.

6.5.1 Esimerkki nimeämiskäytännöstä

Esimerkkinä yhtenäisen nimeämiskäytännön tärkeydestä käytetään graafista komponenttia, joka piirtää kulkuneuvon (kuva 6.8). Kulkuneuvon eri osia voidaan käskeä vaihtamaan väriä ja kulkuneuvoa voidaan pyörittää hiirellä raahaamalla. Samanlaisia ikkunoita voi olla käynnissä useita yhtä aikaa, ja kulkuneuvokomponentteja voi lisätä yhteen ikkunaan useita.

Ajoneuvokomponentin tapauksessa käytetään kolmenlaisia komentoja: vaihda vä-

riä, muuta katselukulmaa ja päivitä kuvaa. Jos komponentin osat (renkaat ja rungon osat) ohjelmoidaan puurakennetta noudattaen, värinvaihtokomennot voidaan lähettää suoraan niille nimen perusteella. Katselukulman muutos ja kuvan päivitys puolestaan vaikuttavat koko näkymään, joten ne lähetetään ajoneuvokomponentille.

Komponentti lähettää tapahtumia aina, kun hiiri liikkuu sen alueella alas painettuna. Vastaavasti voitaisiin odottaa hiiren liikkeen loppumista ja lähettää yksi ainoa signaali, joka summaisi hiiren liikkeen ja pienentäisi viestiliikennettä. Yksittäinen rengas lähettää signaalin värin vaihtumisesta ja ajoneuvokomponentti katselukulman muutoksesta sekä päivittymisestä.

Ikkunoilla ja jokaisella ikkunan sisällä olevalla komponentilla on yksikäsitteiset nimet, mutta ajoneuvokomponenttien sisällä olevat renkaat ovat edelleen samannimisiä muiden ajoneuvokomponenttien renkaiden kanssa. Esimerkiksi kuvassa 6.8 on kaksi näkymää saman sovelluksen käyttöliittymästä. Käyttöliittymää on päivitetty ensimmäisen latauksen jälkeen lisäämällä siihen toinen kulkuneuvokomponentti. Ensimmäiselle ikkunalle riittää yksinkertainen komento, mutta toiselle tarvitaan pidempi nimi, jotta komento menisi oikein perille:

```
Ikkuna 1 -> Vasen eturengas -> Muuta punaiseksi
Ikkuna 2 -> Kulkuneuvo 2 -> Vasen takarengas -> Muuta punaiseksi
```

Signaalien nimeämistä tässä tapauksessa edustaa `Muuta punaiseksi`, jolle on määritettävä avain `changeColor` ja parametri `color`, jolla on attribuutit `red`, `green` ja `blue`. Signaalin nimeämisen on oltava yhdenmukaista sekä ohjelmalogiikkaa että käyttöliittymää kehitettäessä.

6.5.2 Nimeämiskäytännön yhdenmukaisuus

Toteutetussa ratkaisussa eniten ongelmia tuottaa nimeämiskäytännön pitäminen yhdenmukaisena, sillä ohjelmalogiikkaa tehdessä on tiedettävä käyttöliittymäkomponenteille annetut nimet. Ikkunan dynaaminen päivittäminen vaatii tällöin nimeämisen pitämisen yhdenmukaisena, jotta versioriippumattomuus säilyisi.

Nimeämiskäytännön aiheuttamat yhteensopivuusongelmat voitaisiin ratkaista kokonaan, jos sovelluskehyksen datapalvelimeen lisättäisiin mahdollisuus muuttaa kenttien arvoja liityntään konfiguroitujen ehtojen mukaan. Käyttöliittymäkomponenttien nimeä voitaisiin tällöin vaihtaa viestinvälityksessä esimerkiksi seuraavasti:

```
if (widgetName == "tekstikentta1")
   widgetName = "tekstikentta2"
```

Jos kenttien arvojen muutokset voitaisiin ulottaa myös alipakettien tasolle, myös signaalien avainten ja parametrien nimeäminen vapautuisi. Esimerkiksi värin asetuksen avainta voitaisiin muuttaa viestinvälityksessä seuraavasti:

```
if (Signal::Key == "setColor")
   Signal::Key = "changeColor"
```

Näin toimimalla logiikkakomponentin ja käyttöliittymän välisiä, versioiden myötä mahdollisesti syntyviä ristiriitoja voitaisiin korjata liityntöjen konfiguraatiota muuttamalla. Viestien rakentuminen avain-arvo-pareittain poistaa sellaisenaan Taylorin mainitsemat ongelmat parametrien järjestyksessä ja lukumäärissä, joten tällä keinolla nimeäminen saataisiin mahdollisimman joustavalle tasolle, jolla toimimisen estäisi vasta vaaditun parametrin puuttuminen viestistä.

7 Esimerkkisovellus - Autolaskuri

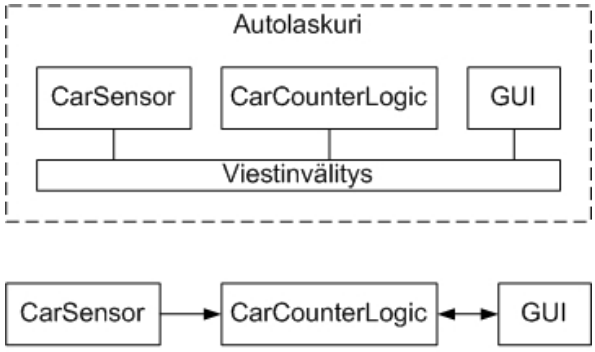
Fiktiivisenä esimerkkinä sovelluskehykseen luotiin yksinkertainen autolaskuri. Esimerkki havainnollistaa sovelluskehyksen ja siihen liitetyn graafisen käyttöliittymän toimintaa. Luvuissa 7.1 ja 7.2 esitellään sovellukseen tarvitut uudet sovelluskehyksen komponentit ja käyttöliittymäkomponentit, luvussa 7.3 komponenttien välinen konfigurointi ja luvussa 7.4 sovelluskehyksen mahdollistamat laajennukset.

7.1 Sovelluskehyksen komponentit

Sovelluskehykseen tarvittiin autolaskuria varten kaksi uutta komponenttia: `CarSensor`- ja `CarCounterLogic`. Tässä luvussa esitellään komponentit ja niille suunnitellut rajapinnat. `CarSensor`-komponentti jätettiin toistaiseksi toteuttamatta, mutta rajapinnan ollessa tiedossa komponenttia voidaan simuloida sovelluskehyksen tarjoamilla resursseilla. Komponentit ja niiden välinen tiedon kulku on havainnollistettu kuvassa 7.1.

7.1.1 `CarSensor`

Autolaskuri perustuu tien varressa olevaan sensoriin, joka ilmoittaa aina kun auto ajaa ohi. Sensoria varten kehitettiin `CarSensor`-komponentti, joka ottaa vastaan sensorin lähettämät viestit, muuntaa ne datapaketeiksi ja lähettää viestinvälitykseen. Lähetettävän datapaketin sisältö määriteltiin seuraavasti:



Kuva 7.1: Autolaskurin komponenttirakenne ja komponenttien välille konfiguroitavat yhteydet.

- Tunnisteeksi annetaan avain `"packetInformation"` arvolla `"CarSensorData"`.
- Varsinaisen datan lähetykseen annetaan avain `"addCounterBy"` ja arvon tyypiksi määritetään integer, oletusarvo on 1.
- Tulevaisuudessa sensoriteknologian kehittyessä itseään korjaavaksi voidaan pakettiin lisätä myös esimerkiksi kentät `"subtractCounterBy"` ja `"speed"` ilman, että ne vaikuttavat aikaisempien sovellusten toimintaan.

Esimerkkitapauksessa sensorikomponentti jätettiin kehittämättä tarvittavan laitteiston puuttuessa. Varsinaisen sensorin sijasta sensoria voidaan simuloida Matlab-skriptillä, joka lähettää tietyn ajan välein sensoriviestien kaltaisia viestejä logiikkakomponentille. Käytännön tilanteessa tällainen testidatan generointi ja lähetys järjestelmään mahdollistaa kunkin komponentin erillisen testauksen, esimerkiksi silloin kun tarvittavaa laitteistoa ei vielä ole saatavilla.

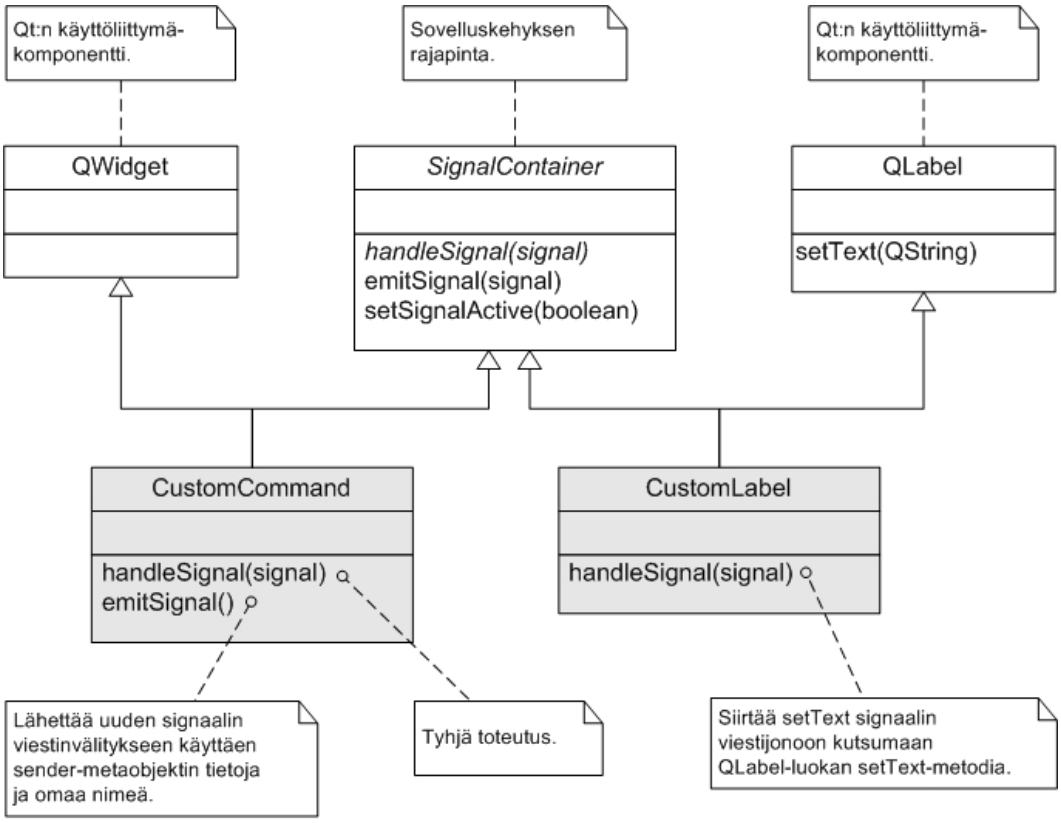
7.1.2 `CarCounterLogic`

Sovelluskehykseen kehitettiin sensorikomponentin lisäksi `CarCounterLogic`-komponentti, joka pitää yllä laskurin arvoa. Komponentti reagoi sensorilta saatuihin viesteihin ja käyttäjän painonappien painalluksiin muuttamalla laskurin arvoa. Muuttunut arvo päivitetään käyttöliittymään. Komponentin rajapintana sovelluskehykseen päin toimii `handlePacket`-metodi eli paketin vastaanotto ja käsittely. `CarCounterLogic`-komponentin `handlePacket`-metodin kaksi vaihtoehtoista toteutustapaa on esitetty liitteessä A, mutta niihin palataan vasta käyttöliittymäkomponenttien yhteydessä luvussa 7.2.4.

7.2 Käyttöliittymä

Autolaskurin graafisessa käyttöliittymässä on numeronäyttö, joka esittää autojen kokonaismäärän, sekä kaksi painonappia (lisää ja vähennä) manuaalisesti syötettäville havainnoille ja korjauksille.

Käyttöliittymäkomponenttien piirto-ominaisuudet saatiin perimällä ne Qt-työkalupakin luokista. Yhteys sovelluskehykseen puolestaan syntyi perimällä luvussa 6.4.2 esitelty `SignalContainer`-luokka. Jotta komponentit saatiin lisättyä Qt Designeriin, niille oli toteutettava myös Qt:n plugin-luokat. Selvyyden vuoksi nämä on jätetty huomioimatta kuvauksissa, sillä niiden ainoa tehtävä on toimia tehtaana, joka luo käyttöliittymäkomponentin Designerin pyynnöstä, joten ne eivät vaikuta varsinaisen komponentin toimintaan.



Kuva 7.2: Autolaskurin erillisten käyttöliittymäkomponenttien luokkakaavio, jossa uudet komponentit on korostettu harmaalla. Perittävien luokkien roolit ja uusien luokkien tärkeimmät toiminnallisuudet on selvennetty huomautuksilla.

Autolaskurin käyttöliittymäkomponentit voitiin toteuttaa erillisinä painonappeina ja tekstikenttänä. Tämä toteutus on esitelty luvussa 7.2.1. Vaihtoehtoisesti autolaskuria varten voitiin kehittää yksi kaikki ominaisuudet sisältävä käyttöliittymäkomponentti, joka on esitelty luvussa 7.2.2. Erilaisia toteutuksia vertaillaan luvussa 7.2.3.

7.2.1 Erilliset käyttöliittymäkomponentit

Ensimmäinen toteutus tehtiin siten, että kaikki käyttöliittymän tarvitsemat komponentit toteutettiin toisistaan erillisinä. Autolaskuria varten tarvittiin kaksi uutta käyttöliittymäkomponenttia: `CustomLabel` laskurin arvon näyttämiseen ja `CustomCommand` painonappien klikkausten välittämiseen. Komponenttien suhteet on havainnollistettu kuvassa 7.2 ja niiden lähdekoodit löytyvät liitteestä B. Selvyyden vuoksi liitteissä on esitettynä vain toiminnan kannalta oleellisten funktioiden toteutukset. Tätä käyttöliittymän toteutusta vastaava, yhdestä laskurista huolehtivan logiikkakomponentin toteutus on esitettynä liitteessä A.1.

`CustomLabel` perittiin `QLabel`-luokasta, ja sen ainoa funktio on `handleSignal`, joka asettaa tekstikentälle uuden arvon vastaanottaessaan `setText`-signaalin. Funktion toteutus on esitetty liitteessa B.1. `CustomCommand`-komponenttia voidaan pitää luvussa 4.5.2 esitellyn komento-suunnittelumallin johdannaisena, jolla hoidetaan painonappien painallusten siirtäminen viestinvälitykseen. `CustomCommand` on näkymätön komponentti, jolla on yksi vastaanotin (*slot*, kts. luku 6.2.1) `emitSignal`. `emitSignal` luo uuden signaalin viestinvälitykseen, asettaa sen avaimeksi oman nimensä ja luojatiedoiksi alkuperäisen signaalin lähettäneen komponentin tiedot. Vastaanottimen toteutus on esitelty liitteessä B.2.

Painonappien painalluksien synnyttämät `clicked`-signaalit yhdistettiin Qt Designerissa `CustomCommand`-komponenttien `emitSignal`-vastaanottimiin. Tämä on havainnollistettu kuvassa 7.3. Samaa `CustomCommand`-komponenttia voidaan käyttää myös valikkojen lähettämille signaaleille, joten käyttöliittymässä voidaan käyttää tavallisia Qt:n menuja ja painonappeja.

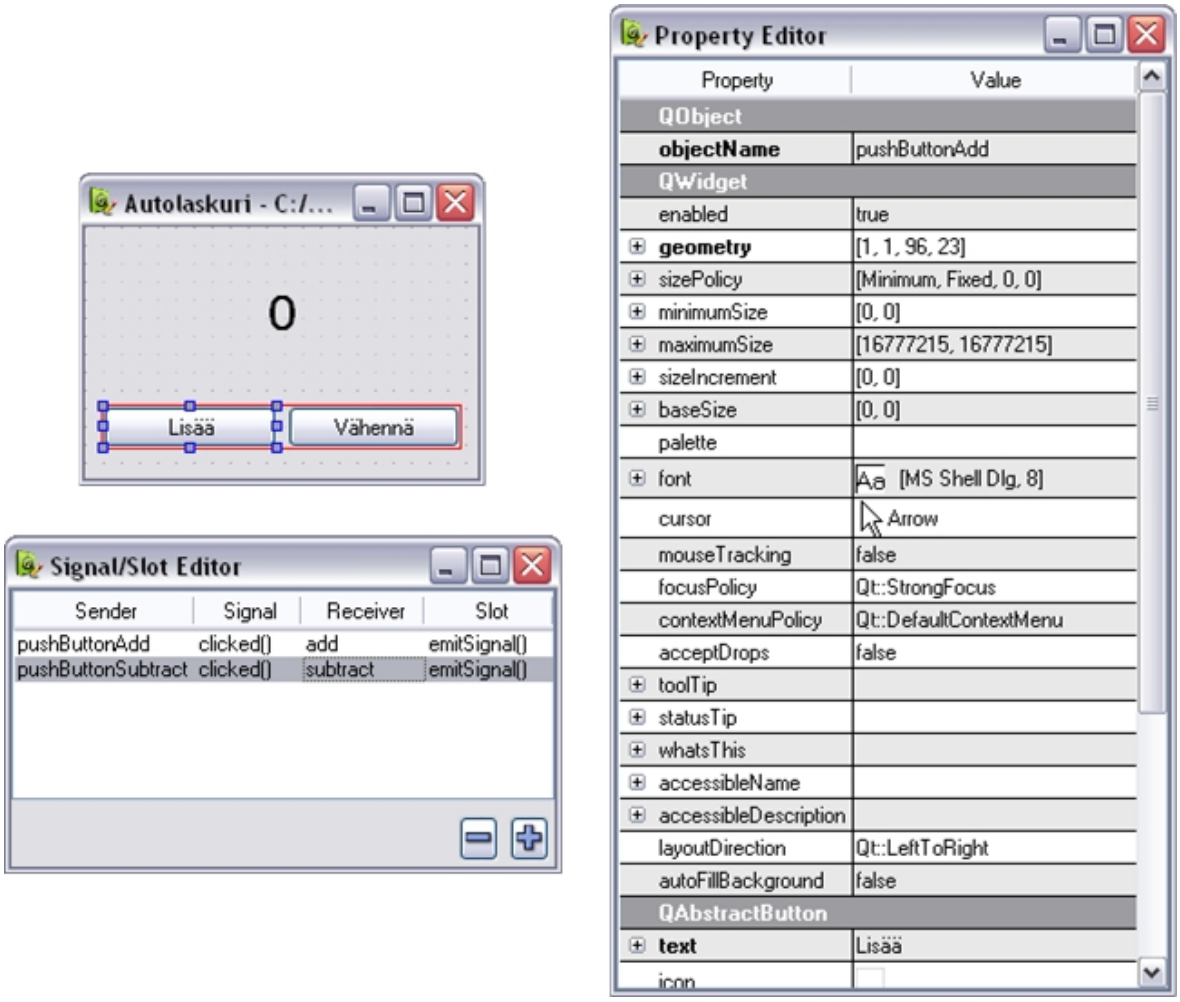
7.2.2 Yhdistetty laskuri-käyttöliittymäkomponentti

Autolaskurin toinen toteutusvaihtoehto perustui siihen, että rakennettiin kokonainen laskuri-käyttöliittymäkomponentti `Counter`. Laskurikomponenttiin sisällytettiin tarvittavat tekstikenttä ja kaksi painonappia. Koska kommunikaatio sovelluskehyksen kanssa hoidettiin `Counter`-komponentissa, sen sisällä voitiin käyttää tavallisia Qt:n komponentteja. Laskurikomponentin luokkakaavio on esitetty kuvassa 7.4.

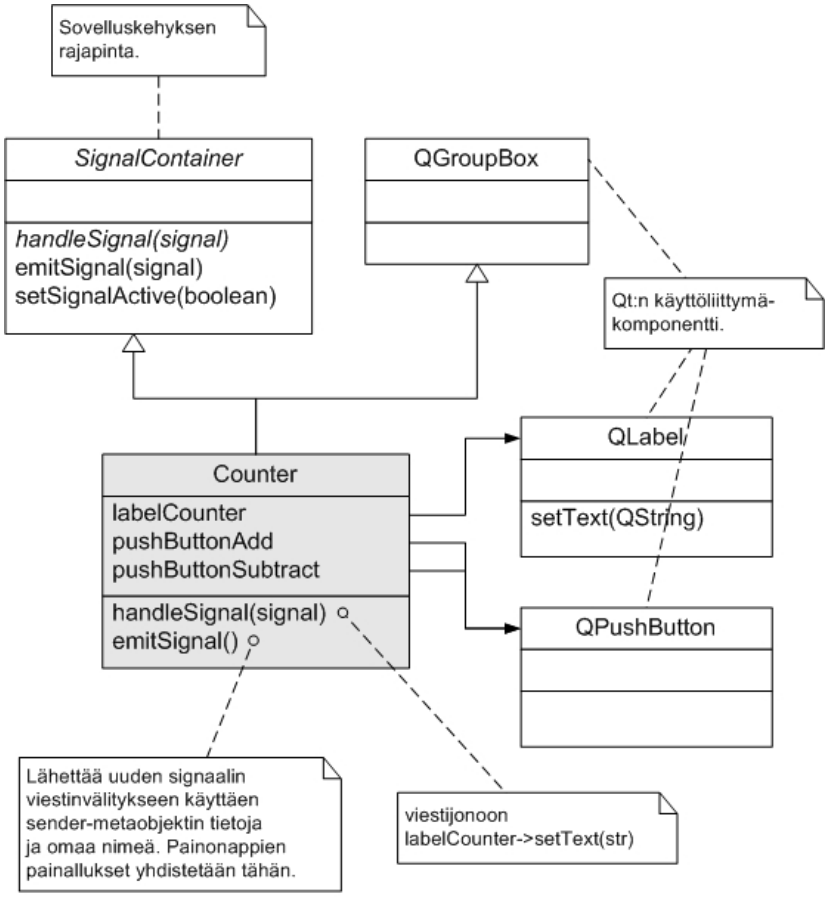
`Counter` perittiin `QGroupBox`-luokasta, jotta sille saatiin otsikoitava reunus. Jäsenmuuttujiksi sille määriteltiin tekstikenttä `labelCounter` sekä kaksi painonappia `pushButtonAdd` ja `pushButtonSubtract`. Painonapeille asetettiin Qt:n `ObjectName`-attribuuttien arvoiksi "add"ja "subtract", jotta näitä voitiin käyttää suoraan ulos lähtevien signaalien avaimina.

Luokassa toteutettiin kaksi metodia: viestinvälitykseltä sisään tulevia paketteja käsittelevä `handleSignal` ja paketteja viestinvälitykseen lähettävä `emitSignal`. Viestinvälityksestä saapunut `setText`-signaali siirrettiin `handleSignal`-metodissa viestijonoon `labelCounter`-jäsenmuuttujan `setText`-kutsuksi. `emitSignal`-metodissa luotiin uusi signaali viestinvälitykseen ja asetettiin siihen laskurikomponentin tiedot. Painonappien painallukset kytkettiin `emitSignal`-metodiin, joka on esitelty Qt:n vastaanottimena. Ulos lähetettävään signaaliin lisättiin avaimeksi alkuperäisen signaalin lähettäneen painonapin `ObjectName`. Laskuri-käyttöliittymäkomponentin lähdekoodi löytyy kokonaisuudessaan liitteestä C jaettuna esittelyyn (C.1) ja toteutukseen (C.2).

Lisäämällä `Counter` Qt Designeriin saavutettiin tilanne, jossa kokonaisia laskureita



Kuva 7.3: Autolaskuri Qt Designerissa. Signal/Slot Editorilla on tässä yhdistetty nappuloiden painallukset vastaavien `CustomCommand`-komponenttien `emitSignals`-vastaanottimiin. `CustomCommand`-komponentit on nimetty yksinkertaisesti nimillä `add` ja `subtract`, sillä nämä nimet menevät viestinvälitykseen lähteviin signaaleihin avaimiksi.



Kuva 7.4: Autolaskurikomponentin luokkakaavio.

voi lisätä raahaamalla käyttöliittymään. Tämä kuitenkin vaikuttaa logiikkakomponentin toimintaan, sillä liitteessä A.1 esitelty logiikkakomponentti pitää yllä ainoastaan yhtä laskuria. Liitteessä A.2 onkin esitelty logiikkakomponentti, joka toimii automaattisesti useamman laskurin kanssa säilyttäen laskureiden arvot käyttöliittymäkomponenttien nimien perusteella. Tätä logiikkaa käyttäen laskureita voidaan lisätä sovellukseen kääntäjää käyttämättä, pelkästään Qt Designerilla tehdyn ui-tiedoston kautta.

Laskureiden logiikka voidaan myös jakaa useampien komponenttien kesken. Erilaisia logiikan sijoittelumahdollisuuksia pohditaan tarkemmin luvussa 7.2.4.

7.2.3 Erilaisten toteutusten vertailu

Käyttötarkoituksesta riippuu, toteutetaanko jonkin käyttöliittymän osan toiminnallisuus erillisillä alemman tason komponenteilla vai isommalla, koostetulla komponentilla. Tässä luvussa esitetään laskuria apuna käyttäen näkökohtia, jotka tulee ottaa huomioon komponenttijakoa suunniteltaessa.

- Kokonaisuuden yleiskäyttöisyys on parempi koostetulla komponentilla. Uuden kokonaisen laskurin lisääminen onnistuu pelkästään raahaamalla uusi `Counter`-komponentti ikkunaan, toisin kuin erillisiä komponentteja käytettäessä.

- Alemman tason osien, kuten pelkän tekstikentän, uudelleenkäyttö ei onnistu koostetulla komponentilla, sillä tekstikenttä on liitettynä laskurikomponentin sisälle ilman, että siitä on kehitetty sovelluskehykseen integroitavaa versiota.

- Laskurin ulkonäön muuttaminen, esimerkiksi nappuloiden siirto plus- ja miinusmerkeiksi alapalkkiin, onnistuu erillisten komponenttien tapauksessa Designerilla. Koostetun komponentin tapauksessa se vaatii ohjelmointia ja komponentin kääntämistä.

- Laskurin arvoa näyttävän komponentin vaihto onnistuu erillisten komponenttien tapauksessa Designerilla, olettaen, että se käyttää samaa rajapintaa arvon asetukseen.

- Laskurin ulkonäön muunneltavuus käyttökohteiden välillä ei onnistu lainkaan koostettua komponenttia käytettäessä. Tuoteperheen sovellusten kesken tämä on hyvä ominaisuus, sillä se lisää sovellusten yhdenmukaisuutta. Jos komponenttia halutaan muokata erilaisten käyttäjäryhmien välillä, koostetusta komponentista olisi tehtävä useita erilaisia versioita.

- Uuden ohjaustavan, esimerkiksi menun kautta tapahtuvan ohjauksen lisäys käy helpommin erillisiä komponentteja ja `CustomCommand`-luokkaa käytettäessä.

7.2.4 Logiikan sijoittamisen vaihtoehdot

Logiikan sijoittelu aiheuttaa pohdintaa siinä tapauksessa, kun toisiaan vastaavia logiikkayksiköitä halutaan lisätä sovellukseen useampia. Seuraavassa on esitelty kolme vaihtoehtoa autolaskuria esimerkkinä käyttäen, sekä pohdittu hieman niiden soveltuvuutta sovelluskehyksen ajatusmaailmaan:

1. Logiikkakomponentin monistamisen on näistä vaihtoehdoista parhaiten sovelluskehyksen tietovuoarkkitehtuuria tukeva. Logiikkakomponenttia monistamalla saadaan yhden komponentin sisältämä työmäärä pysymään pienenä ja laskurin tapauksessa esimerkiksi sensorien konfigurointi pysymään yksinkertaisena. Monistaminen vaatii kuitenkin enemmän konfigurointia jokaista sovellukseen lisättyä laskuria kohti ja myös monimutkaistaa lopullisen sovelluksen rakennetta.

2. Logiikkakomponentti voidaan muokata sellaiseksi, että se pystyy käsittelemään useamman laskurin dataa kerralla. Esimerkkinä liitteessä A.2 on toteutettuna autolaskurin logiikkakomponentti, joka käsittelee useiden laskureiden arvoa käyttäen avaimena käyttöliittymäkomponentille annettua nimeä. Tämä vaihtoehto vaatii kuitenkin sen, että myös kaikkien sensorien data ohjataan samalle logiikkakomponentille. Näin ollen myös sensorin viestiin on lisättävä kenttä `counterID`, ja logiikkakomponenttiin selvitys siitä, mikä sensori yhdistetään mihinkin laskuriin. Ensimmäiseen vaihtoehtoon verrattuna joustavuus ja ajonaikainen konfiguroitavuus ovat näin ollen huonompia.

3. Kolmas vaihtoehto logiikan eli tässä tapauksessa laskurin arvon sijoittamiselle olisi käyttöliittymäkomponentti, jolloin komponenttia voitaisiin kutsua visuaaliseksi formalismiksi. Tämä kuitenkin hajottaisi tietovuoajatuksen perinpohjaisesti. Sensoreilta tulevat viestit täytyisi muokata suoraan käyttöliittymäkomponentille soveltuviksi, mikä poistaisi sensorin yleiskäyttöisyyden. Vaihtoehtoisesti välille täytyisi tehdä sensoriviestin käyttöliittymäviestiksi muuttava komponentti, joka puolestaan vastaa logiikkakomponenttia ja poistaa monimutkaisuudellaan visuaalisen formalismin aikaansaaman hyödyn.

7.3 Viestinvälityksen konfigurointi

Viestinvälitys täytyy konfiguroida siten, että kaikki tarpeellinen data saadaan liikkumaan komponenttien välillä. Seuraavassa listassa on esitetty tarvittavat konfiguroinnit tilanteessa, jossa käytetään yhtä logiikkakomponenttia.

- Sensorikomponentin data konfiguroidaan kulkemaan logiikkakomponentille. Lähetettävien viestien ehdoksi laitetaan, että `"packetInformation"`-avaimen arvon on oltava `"CarSensorData"`. Esimerkkitilanteessa tämä jätetään tekemättä ja sensoria simuloivat viestit lähetetään suoraan logiikkakomponentille.

- Käyttöliittymän päivitysviestit konfiguroidaan kulkemaan logiikkakomponentilta ikkunakomponentille. Lähetettävien viestien ehdoksi laitetaan, että `"GUIMessage"`-avaimen arvon on oltava `"Signal"` ja `"CreatorUniqueName"`-avaimen arvon on oltava kyseisen `AutolaskuriLogiikka`-komponentin instanssin uniikki nimi.

- Käyttäjältä tulevat viestit eli tässä tapauksessa painonappien painallukset konfiguroidaan kulkemaan ikkunakomponentilta logiikkakomponentille. Lähetettävien viestien ehdoksi laitetaan, että `"GUIMessage"`-avaimen arvon on oltava `"Signal"` ja `"CreatorUniqueName"`-avaimen arvon on oltava ikkunakomponentin instanssin uniikki nimi.

7.4 Sovelluskehyksen mahdollistamat laajennukset

Sovelluskehyksen tehokkuus ei autolaskurisovelluksessa pääse oikeuksiinsa sovelluksen yksinkertaisuuden takia. Esimerkin avulla voidaan kuitenkin havainnollistaa sovelluskehyksen tarjoamaa dynaamisuutta. Mahdollisista helposti toteutettavista muutoksista voidaan mainita seuraavat:

- Pelkän konfiguroinnin avulla voitaisiin lisätä useita sensoreita esimerkiksi moottoritien eri kaistoja varten.

- Vastaavanlainen näyttö voitaisiin ajonaikaisesti avata myös ylemmän tason autolaskennan valvojan koneelle. Tämä näyttö voitaisiin konfiguroida kopioimaan varsinaisen laskentatyöntekijän näytön laskurin arvoja, jolloin valvoja pystyisi seuraamaan tapahtumia vaikuttamatta niihin.

- Edelliseen kohtaan luonnollisena jatkona valvojan näytölle avattaisiin useita, eri puolella Suomea sijaitsevien laskentapisteiden tapahtumia seuraavia ikkunoita. Tämä onnistuisi konfiguraatiota muuttamalla.

- Uuden grafiikkaa esittävän käyttöliittymäkomponentin avulla laskurien data voitaisiin ohjata valvojan näytölle kootussa muodossa, esimerkiksi pylväsgraafina. Tämä vaatisi laskurilogiikkakomponentteja lisäämään lähettämiinsä viesteihin laskentapaikan tunnisteen.

- Datavirta voitaisiin ohjata tallennettavaksi tietokantakomponentin lisäyksellä.
- Analysointikomponentin lisäyksellä voitaisiin dataa analysoida esimerkiksi vertaamalla aiemmin talletettuihin arvoihin. Analysoinnin visualisoimiseksi voitaisiin käyttää vaikkapa valmista pylväsgraafikomponenttia.

8 Yhteenveto

Tutkielman tavoitteena oli sovittaa käyttöliittymien kehitys yhteen hajautettuna järjestelmänä toimivan sovelluskehyksen kanssa. Haluttiin suunnitella tapa, jolla graafisia käyttöliittymiä kehitettäisiin jatkossa kun sovelluskehyksestä erikoistettaisiin uusia sovelluksia. Vaatimusten kohdistuessa useamman, ennalta määrittelemättömän sovelluksen kehitykseen tutkimuksessa jätettiin pois varsinainen käyttöliittymien suunnittelu ja keskityttiin pohtimaan tapaa, jolla käyttöliittymät saataisiin liitettyä kehykseen siten, että toteutus jättäisi varsinaisen sovelluksen kehittäjälle mahdollisimman paljon liikkumavaraa.

Jotta graafisista käyttöliittymistä saataisiin riittävän kattava kuva, keskityttiin tutkielman alussa niitä koskeviin perusasioihin. Perusteita tutkittaessa pyrittiin löytämään kaikille oliopohjaisille graafisille käyttöliittymille yhteisiä piirteitä sekä selvittämään kontrollin kulkua graafisen käyttöliittymän sisältävässä sovelluksessa. Todettiin, että kaikki graafiset käyttöliittymät perustuvat viestisilmukkaan, joka käsittelee ikkunointijärjestelmältä vastaanotettuja viestejä asynkronisesti. Lisäksi todettiin että valmiita työkalupakkeja käyttämällä voidaan graafisten käyttöliittymien kehitystä helpottaa huomattavasti.

Kun graafisten käyttöliittymien perusteet oli saatu selvitettyä, paneuduttiin tutkimuksen kannalta toiseen merkittävään seikkaan, eli hajautettuihin järjestelmiin. Luvussa 3 määriteltiin hajautetut järjestelmät yleisesti, sekä tutustuttiin joihinkin graafisia käyttöliittymiä koskeviin arkkitehtuuriratkaisuihin hajautetuissa järjestelmissä. Erityisesti paneuduttiin graafisiin käyttöliittymiin viestinvälitysarkkitehtuurissa, jota myös tutkimuksen kannalta tärkeässä sovelluskehyksessä noudatetaan. Tärkeimmäksi haasteeksi graafisia järjestelmiä hajautettuun järjestelmään suunnitellessa todettiin tehtävien jakaantuminen käyttöliittymän ja ohjelmalogiikan kesken. Ongelma on sama myös tavallisia sovelluksia kehitettäessä, joten luvun lopussa tarkasteltiinkin *dialogin riippumattomuuteen* liittyviä etuja ja ongelmia graafisia käyttöliittymiä käsitteleviä lähteitä käyttäen.

Graafisten käyttöliittymien perusteita ja hajautettuja järjestelmiä tarkasteltaessa päädyttiin siihen, että molempien kannalta erittäin tärkeään osaan nousevat graafisten käyttöliittymien eri osien ja hajautuksen myötä syntyvien erillisten logiikkakomponenttien välillä välitettävät viestit. Tästä syystä viestien tarkastelulle omistettiin oman lukunsa, jossa pyrittiin löytämään yhtenäinen luokittelu tarvittaville viestityy-

peille. Ongelmaa hankaloitti yhtenäisyyden puute olemassaolevien työkalujen viestien nimeämisessä ja käyttötavoissa. Valmiin yhtenäisen ratkaisun puuttuessa viestit päätettiin jakaa niiden loogisten roolien mukaan ikkunointijärjestelmän *tapahtumiin*, käyttöliittymäkomponenttien *komponenttitapahtumiin* ja ohjelmalogiikan *komentoihin*.

Viestien jälkeen selvittämättä oli enää sovelluskehyksen tavoitteet, rakenne ja vaikutukset toteutettavaan ratkaisuun, joita selvitettiin luvussa 5. Sovelluskehysten käytön mielekkyys perusteltiin, jonka jälkeen esitettiin joitakin yleisiä, sovelluskehyksiä koskevia seikkoja kuten kontrollin siirron ongelma graafisten käyttöliittymien työpakkeja käytettäessä. Tämän jälkeen esiteltiin tutkielman pohjana toimiva sovelluskehys ja sen asettamat vaatimukset graafisille käyttöliittymille.

Varsinaisessa toteutuksessa päädyttiin kehittämään sovelluskehykseen ikkunapalvelimena toimiva komponentti. Toteutuksessa käytettiin Qt-työkalupakkia, mutta rakenne suunniteltiin siten, että siihen voitaisiin jatkossa lisätä tuki myös muille työkalupakeille. Ongelmia toteutuksessa aiheuttivat viestien rakenteen määrittäminen sekä kontrollin siirto sovelluskehyksen ja työkalupakin välillä. Viestit päätettiin lopulta yleistää *signaaleiksi*, joihin voitaisiin yhtenäistä nimeämiskäytäntöä noudattaen sisällyttää tapahtumat, komponenttitapahtumat ja komennot, ja joiden kontrollointi hoidettaisiin erillistä viestiin sisällytettävää ohjausosaa hyväksi käyttäen. Kontrollin siirron ongelmat ratkaistiin siten, että sovelluskehyksestä kontrolli siirrettiin työkalupakille sen viestijonoa käyttäen. Graafisista käyttöliittymistä alkunsa saaneet viestit puolestaan siirrettiin sovelluskehyksen kontrollin piiriin tarkoitusta varten kehitetyn jonon ja sitä purkavan erillisen säikeen avulla.

Lopuksi sovelluskehykseen toteutettiin fiktiivinen autolaskurisovellus ikkunakomponenttia hyväksi käyttäen. Autolaskurisovelluksen avulla havainnollistettiin sovelluskehyksen toimintaa ja käyttöliittymien roolia siinä. Autolaskurin käyttöliittymästä toteutettiin kaksi erilaista vaihtoehtoa käyttäen erillisiä käyttöliittymäkomponentteja ja yhdistettyä laskurikomponenttia. Erilaisia käyttöliittymien toteutuksia vertailtiin keskenään ja logiikan sijoitukselle esitettiin kolme mahdollista toteutusvaihtoehtoa.

9 Viitteet

[1] Richard M. Adler: *Distributed Coordination Models for Client/Server Computing*, IEEE, 1995.

[2] P.S.C. Alencar, D.D. Cowan, C.J.P. Lucena, L.C.M. Nova: *Formal Specification of Reusable Interface Objects*, ACM, 1995.

[3] Seán Baker: *CORBA Distributed Objects*, Addison-Wesley, 1997.

[4] Len Bass, Paul Clements, Rick Kazman: *Software Architecture in Practice*, Addison-Wesley, 1998.

[5] Jan Bosch: *Design & Use of Software Architectures - Adopting and evolving a product-line approach*, Addison-Wesley, 2000.

[6] Joëlle Coutaz: *PAC-ing the Architecture of Your User Interface*, DVS-IS'97, pp. 15-32, 1997.

[7] Matthias Kalle Dalheimer: *Programming with Qt*, O'Reilly, 1999.

[8] Alan Dix, Gragory Abowd: *Modelling status and event behaviour of interactive systems*, 1996.

[9] Marc Evers: *A Case Study on Adaptability Problems of the Separation of User Interface and Application Semantics*, 1999.

[10] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Olio-ohjelmointi Suunnittelumallit*, IT-Press, 2001.

[11] *GNU General Public License*,
<URL: `http://www.gnu.org/copyleft/gpl.html`>, viitattu 20.9.2006.

[12] Glenn E. Grasner and Stephen T. Pope: *A Description of the Model-View-Controller User Interface Paradigm int the Smalltalk-80 System*, 1988.

[13] H. Rex Hartson and Deborah Hix: *Human-Computer Interface Development: Concepts and Systems for Its Management*, ACM Computing Surveys, Vol. 21, No. 1, March 1989.

[14] Jeff Johnson: *Selectors: Going Beyond User Interface Widgets*, 1992.

[15] *KDE*, <URL: `http://www.kde.org/`>, viitattu 20.9.2006.

[16] Guy Keren, *Basic Graphics Programming With The Xlib Library*, <URL: `http://users.actcom.co.il/ choo/lupg/tutorials/ xlib-programming/xlib-programming.html#async_model`>, viitattu 20.9.2006.

[17] Kai Koskimies, Tommi Mikkonen: *Ohjelmistoarkkitehtuurit*, Gummerus Kirjapaino, 2005.

[18] David L. Martin, Adam J. Cheyer, Douglas B. Moran: *The Open Agent Architecture: A Framework for Building Distributed Software Systems.*

[19] Douglas B. Moran, Adam J. Cheyer, Luc E. Julia, David L. Martin: *Multimodal User Interfaces in the Open Agent Architecture.*

[20] Microsoft, *Microsoft Foundation Class Library*, <URL: `http://msdn.microsoft.com/library/default.asp?url=/library/ en-us/vcmfc98/html/mfchm.asp`>, viitattu 20.9.2006.

[21] Mozilla.org, *XML User Interface Language (XUL)*, <URL: `http://www.mozilla.org/projects/xul/`>, viitattu 20.9.2006.

[22] Brad A. Myers: *Graphical User Interface Programming*, Draft of: January 27, 2003 to appear in: CRC Handbook of Computer Science and Engineering - 2nd Edition, 2003.

[23] Dan R. Olsen, Jr.: *Developing User Interfaces*, Morgan Kaufmann Publishers, Inc., 1998.

[24] Providence Software Solutions inc., *XVT*, <URL: `http://www.xvt.com/`>, viitattu 20.9.2006.

[25] University of Strathclyde Computer Centre, *Introduction to X Windows*, <URL: `http://www.strath.ac.uk/CC/Courses/oldXC/xc.html`>, viitattu 20.9.2006.

[26] Sun Microsystems, *How to Write an Action Listener*, <URL: `http://java.sun.com/docs/books/tutorial/uiswing/events/ actionlistener.html`>, viitattu 20.9.2006.

[27] Sun Microsystems, *Package java.awt*,
<URL: http://java.sun.com/j2se/1.4.2/docs/api/java/awt/package-summary.html>, viitattu 20.9.2006.

[28] Sun Microsystems, *JavaTM 2 Platform, Standard Edition, v 1.4.2, API Specification*,
<URL: http://java.sun.com/j2se/1.4.2/docs/api/index.html>, viitattu 20.9.2006.

[29] Sun Microsystems, *Trail: Creating a GUI with JFC/Swing*,
<URL: http://java.sun.com/docs/books/tutorial/uiswing/>, viitattu 20.9.2006.

[30] Clemens Szyperski: *Component Software, Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[31] Andrew S. Tanenbaum, Maarten van Steen: *Distributed Systems - Principles and Paradigms*, Prentice-Hall, 2002.

[32] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., Jason E. Robbins, Kari A. Nies, Peyman Oreizy, Deborah L. Dubrow: *A Component- and Message-Based Architectural Style for GUI Software*, IEEE Transactions of Software Engineering, Vol. 22, No. 6, June 1996.

[33] Tcl/Tk, *Tcl/Tk*, <URL: http://tcl.sourceforge.net/>, viitattu 20.9.2006.

[34] Trolltech, *About Trolltech*,
<URL: http://www.trolltech.com/company/index.html>, viitattu 20.9.2006.

[35] Trolltech, *Qt Customers*,
<URL: http://www.trolltech.com/company/customers.html>, viitattu 20.9.2006.

[36] Trolltech, *Qt Designer*,
<URL: http://www.trolltech.com/products/qt/designer.html>, viitattu 20.9.2006.

[37] Trolltech, *Qt Overview*, <URL: http://www.trolltech.com/products/qt/>, viitattu 20.9.2006.

[38] Trolltech, *Signals and Slots*,
<URL: http://doc.trolltech.com/4.0/signalsandslots.html?cid=18>, viitattu 20.9.2006.

[39] W3C, *Extensible Markup Language (XML)*, <URL: http://www.w3.org/XML/>, viitattu 20.9.2006.

[40] wxWindows, *wxWindows*, <URL: http://wxwidgets.org/>, viitattu 20.9.2006.

[41] XUL Planet, *XUL Tutorial*, <URL: http://www.xulplanet.com/tutorials/xultu/>, viitattu 20.9.2006.

A Autolaskurin logiikkakomponentti

A.1 `CarCounterLogic::handlePacket()` yhdelle laskurille

```
// carcounterlogic.cpp
// Paketin käsittelyfunktio, eli komponentin rajapinta
// sovelluskehykseen.
int CarCounterLogic::handlePacket(Packet* p)
{
   int ret = false;
   try
   {
      // Tarkistetaan "GUIMessage"-tunniste
      std::string id = p->getIDStringItem("GUIMessage");
      if (id == "Signal")
      {
         // Käyttöliittymältä tulleen signaalin käsittely
         handleSignal(p);
         ret = true;
      }
      // Muussa tapauksessa ulkopuolelta (=sensorilta)
      // tullut viesti, johon reagoidaan vain jos oletussignaali
      // on alustettu.
      else if (defaultSignal)
      {
         std::string packetInfo =
            p->getIDStringItem("packetInformation");
         if (packetInfo == "CarSensorData")
         {
            int x = 0;
            // Getteri konvertoi samalla arvon tarvittaessa
            p->getIDStringItemAs("addCounterBy", &x);
            addCounter(x);
            ret = true;
```

```
         }
      }
   }
   catch(...)
   {
      log("CarCounterLogic::handlePacket failed, exception.");
   }
   return ret;
}

// Käyttöliittymältä tulleen signaalin käsittely.
int CarCounterLogic::handleSignal(Packet* p)
{
   int ret = false;
   // Signaali alipaketissa
   Packet* signal = 0;
   if (!p->getIDStringItemAs("Signal", signal))
   {
      log("CarCounterLogic::handleSignal signal missing.");
      return false;
   }

   std::string key = signal->getIDStringItem("Key");
   if (key == "add")
   {
      addCounter(1);
      ret = true;
   }
   else if (key == "subtract")
   {
      addCounter(-1);
      ret = true;
   }
   return ret;
}

// Laskurin muutokseen käytettävä funktio.
```

```
// Huolehtii siitä, että laskuria päivitettäessä
// päivitetään myös käyttöliittymää.
void CarCounterLogic::addCounter(int x)
{
   counter += x;
   try
   {
      Packet* newPacket = defaultSignal->duplicatePacket();
      if (!newPacket))
      {
         log("CarCounterLogic::addCounter, duplicate failed.");
         return false;
      }

      // Signaali alipaketissa
      Packet* newSignal = 0;
      if (!newPacket->getIDStringItemAs("Signal", newSignal))
      {
         // Signaalin puuttuessa luodaan uusi
         newSignal = new Packet();
         newPacket->setIDStringItem("Signal", newSignal);
      }

      // Komponentin tyyppi, konfiguroitavissa
      newPacket->setIDStringItem("WidgetType", widgetType);
      // Komponentin nimi, konfiguroitavissa
      newPacket->setIDStringItem("WidgetName", widgetName);

      // Signaalin avain ja arvo
      newSignal->setIDStringItem("Key", "setText");
      newSignal->setIDStringItem("str", counter);

      // Lähetetään paketti viestinvälitykseen
      sendPacketToDataServer(newPacket);
   }
   catch (...)
   {
```

```
      log("CarCounterLogic::addCounter exception.");
   }
}
```

A.2 CarCounterLogic::handlePacket() useammalle laskurille

```
// carcounterlogic.cpp
// Paketin käsittelyfunktio, eli komponentin rajapinta
// sovelluskehykseen.
int CarCounterLogic::handlePacket(Packet* p)
{
   int ret = false;
   try
   {
      // Tarkistetaan "GUIMessage"-tunniste
      std::string id = p->getIDStringItem("GUIMessage");
      if (id == "Signal")
      {
         // Käyttöliittymältä tulleen signaalin käsittely
         handleSignal(p);
         ret = true;
      }
      // Muussa tapauksessa ulkopuolelta (=sensorilta)
      // tullut viesti, johon reagoidaan vain jos oletussignaali
      // on alustettu.
      else if (defaultSignal)
      {
         std::string packetInfo =
            p->getIDStringItem("packetInformation");
         if (packetInfo == "CarSensorData")
         {
            int x = 0;
            // Getteri konvertoi samalla arvon tarvittaessa
            p->getIDStringItemAs("addCounterBy", &x);
            std::string counterID =
               p->getIDStringItem("counterID");
            addCounter(counterID, x);
```

```
            ret = true;
         }
      }
   }
   catch(...)
   {
      log("CarCounterLogic::handlePacket failed, exception.");
   }
   return ret;
}

// Käyttöliittymältä tulleen signaalin käsittely.
int CarCounterLogic::handleSignal(Packet* p)
{
   int ret = false;
   // Signaali alipaketissa
   Packet* signal = 0;
   if (!p->getIDStringItemAs("Signal", signal))
   {
      log("CarCounterLogic::handleSignal signal missing.");
      return false;
   }

   std::string key = signal->getIDStringItem("Key");
   std::string widgetName = signal->getIDStringItem("WidgetName");
   if (key == "add")
   {
      addCounter(widgetName, 1);
      ret = true;
   }
   else if (key == "subtract")
   {
      addCounter(widgetName, -1);
      ret = true;
   }
   return ret;
}
```

```
// Laskurin muutokseen käytettävä funktio.
// Huolehtii siitä, että laskuria päivitettäessä
// päivitetään myös käyttöliittymää.
void CarCounterLogic::addCounter(std::string& counterIndex, int x)
{
   // Jäsenmuuttujan tyypiksi vaihdetaan hashmap<std::string, int>
   // joten siinä voidaan ylläpitää useampaa laskuria kerralla
   counter[counterIndex] += x;
   try
   {
      Packet* newPacket = defaultSignal->duplicatePacket();
      if (!newPacket))
      {
         log("CarCounterLogic::addCounter, duplicate failed.");
         return false;
      }

      // Signaali alipaketissa
      Packet* newSignal = 0;
      if (!newPacket->getIDStringItemAs("Signal", newSignal))
      {
         // Signaalin puuttuessa luodaan uusi
         newSignal = new Packet();
         newPacket->setIDStringItem("Signal", newSignal);
      }

      // Komponentin tyyppi, konfiguroitavissa
      newPacket->setIDStringItem("WidgetType", widgetType);
      // Komponentin nimenä käytetään laskurin tunnistetta
      newPacket->setIDStringItem("WidgetName", counterIndex);

      // Signaalin avain ja arvo
      newSignal->setIDStringItem("Key", "setText");
      newSignal->setIDStringItem("str", counter[counterIndex]);

      // Lähetetään paketti viestinvälitykseen
```

```
        sendPacketToDataServer(newPacket);
    }
    catch (...)
    {
        log("CarCounterLogic::addCounter exception.");
    }
}
```

B Autolaskurin käyttöliittymäkomponentit erillisinä

B.1 CustomLabel-luokan handleSignal-metodin toteutus

```
// customlabel.cpp
// Viestinvälitykseltä saapuneen datapaketin käsittely.
// Datapaketti sisältää yhden signaalin.
int CustomLabel::handleSignal(Packet* p)
{
   int ret = false;
   if (p == 0)
   {
      return ret;
   }

   // Signaalin avain
   std::string key = p->getIDStringItem("Key");
   if (key == "setText")
   {
      // Signaalin parametri avaimella "str"
      std::string s = p->getIDStringItem("str");
      QString str = s.c_str();
      // Qt:n tapa siirtää päivityskäsky viestijonoon
      if (QMetaObject::invokeMethod(this, key.c_str(),
      Qt::QueuedConnection, Q_ARG(QString, str)))
      {
         ret = true;
      }
   }
   return ret;
}
```

B.2 CustomCommand-luokan emitSignal-vastaanottimen toteutus

```
// customcommand.cpp
// Uuden signaalin lähetys viestinvälitykseen.
// Pohjana käytetään alustuksen yhteydessä asetettua
// signaalipohjaa jäsenmuuttujasta defaultSignal.
void CustomCommand::emitSignal()
{
   if (defaultSignal == 0)
   {
      return;
   }

   Packet* newPacket = defaultSignal->duplicate();
   if (!newPacket)
   {
      log("CustomCommand::emitSignal duplicate failed");
      return;
   }

   // Signaali alipaketissa
   Packet* newSignal = 0;
   if (!newPacket->getIDStringItemAs("Signal", newSignal))
   {
      // Signaalin puuttuessa luodaan uusi
      newSignal = new Packet();
      newPacket->setIDStringItem("Signal", newSignal);
   }

   // Lähettäjä-käyttöliittymäkomponentin (esim pushButton)
   // tiedot lisätään ulos lähtevään, signaalin sisältävään
   // pakettiin käyttäen Qt:n metaObject-tietoja

   // Komponentin tyyppi
   newPacket->setIDStringItem("WidgetType",
      sender()->metaObject()->className());
   // Komponentin nimi
```

```
    newPacket->setIDStringItem("WidgetName",
       sender()->objectName().toAscii().data());

    // Oma nimi signaalin avaimeksi
    newSignal->setIDStringItem("Key",
       this->objectName().toAscii().data());

    // Yläluokan emitSignal lähettää paketin viestinvälitykseen
    SignalContainer::emitSignal(newPacket);
}
```

C Laskuri-käyttöliittymäkomponentti

C.1 `Counter`-luokan esittely `counter.h`

```
#ifndef COUNTER_H
#define COUNTER_H

#include <QGroupBox>
#include <QLabel>
#include <QPushButton>
#include <QHBoxLayout>
#include <QVBoxLayout>

#include "signalcontainer.h"
#include "packet.h"

// Counter-käyttöliittymäkomponentti, jota käytetään
// laskurin arvon ylläpitoon ja päivitykseen
class Counter : public QGroupBox, public SignalContainer
{
   // Qt:n makrot, joilla luodaan metadata ja sinne
   // tarvittavat rajapinnat
   Q_OBJECT
   Q_INTERFACES(SignalContainerInterface)

public:
    Counter(QWidget *parent = 0);
    virtual ~Counter();

   int handleSignal(Packet* p);

// Qt:n käyttämä vastaanottimen määritys
public slots:
   void emitSignal();
```

```
private:
   QLabel* labelCounter;
   QPushButton* pushButtonAdd;
   QPushButton* pushButtonSubtract;
};

#endif
```

C.2 Counter-luokan toteutus counter.cpp

```
#include "counter.h"

Counter::Counter(QWidget *parent)
   : QGroupBox(parent)
{
   // Luodaan ja alustetaan tekstikenttä
   labelCounter = new QLabel(this);
   labelCounter->setAlignment(Qt::AlignHCenter);
   QFont f("Helvetica", 18, QFont::Bold);
   labelCounter->setFont(f);
   labelCounter->setText("0");

   // Luodaan ja alustetaan "Lisää"-painonappi
   pushButtonAdd = new QPushButton(this);
   // Nimeä käytetään lähtevässä signaalissa avaimena
   pushButtonAdd->setObjectName("add");
   pushButtonAdd->setText("Lisää");
   // Yhdistetään painonapin klikkaus
   // emitSignal-vastaanottimeen
   // emitSignal-funktiota kutsutaan
   // näin ollen painonappia klikattaessa
   connect(pushButtonAdd, SIGNAL(clicked()),
      this, SLOT(emitSignal()));

   // Luodaan ja alustetaan "Vähennä"-painonappi
```

```
   pushButtonSubtract = new QPushButton(this);
   pushButtonSubtract->setObjectName("subtract");
   pushButtonSubtract->setText("Vähennä");
   // Yhdistetään painonapin klikkaus
   // emitSignal-vastaanottimeen
   connect(pushButtonSubtract, SIGNAL(clicked()),
      this, SLOT(emitSignal()));

   // Asettelu Qt:n layout-komponenttien avulla, ensin
   // painonapit rinnakkain uudelle tyhjälle komponentille...
   QWidget* w = new QWidget(this);
   QHBoxLayout* hLayout = new QHBoxLayout(w);
   hLayout->addWidget(pushButtonAdd);
   hLayout->addWidget(pushButtonSubtract);

   // ... ja sitten tekstikenttä ja painonappikomponentti
   // allekkain.
   QVBoxLayout* vLayout = new QVBoxLayout(this);
   vLayout->addWidget(labelCounter);
   vLayout->addWidget(w);

}

Counter::~Counter()
{
   // Luotujen komponenttien tuhoamisesta huolehtii
   // Qt:n puurakenne, jossa vanhempana toimivat komponentit
   // tuhoavat automaattisesti lapsensa.
}

// Viestinvälitykseltä saapuneen datapaketin käsittely.
// Datapaketti sisältää yhden signaalin.
int Counter::handleSignal(Packet* p)
{
   int ret = false;
   if (p == 0)
   {
```

```
      return ret;
   }

   // Signaalin avain
   std::string key = p->getIDStringItem("Key");
   if (key == "setText")
   {
      // Signaalin parametri avaimella "str"
      std::string s = p->getIDStringItem("str");
      QString str = s.c_str();
      // Kutsutaan labelCounterin setText-metodia viestijonon
      // kautta
      if (QMetaObject::invokeMethod(labelCounter, key.c_str(),
      Qt::QueuedConnection, Q_ARG(QString, str)))
      {
         ret = true;
      }
   }
   return ret;
}

// Uuden signaalin lähetys viestinvälitykseen.
// Pohjana käytetään SignalContainer-yläluokan
// alustuksen yhteydessä asetettua signaalipohjaa
// jäsenmuuttujasta defaultSignal.
void Counter::emitSignal()
{
   if (defaultSignal == 0)
   {
      return;
   }

   Packet* newPacket = defaultSignal->duplicate();
   if (!newPacket)
   {
      log("CustomCommand::emitSignal duplicate failed");
      return;
```

```
    }

    // Signaali alipaketissa
    Packet* newSignal = 0;
    if (!newPacket->getIDStringItemAs("Signal", newSignal))
    {
       // Signaalin puuttuessa luodaan uusi
       newSignal = new Packet();
       newPacket->setIDStringItem("Signal", newSignal);
    }

    // Lisätään omat tiedot lähettäjän paikalle
    // ulos lähtevään, signaalin sisältävään pakettiin käyttäen
    // Qt:n metaObject-tietoja

    // Komponentin tyyppi
    newPacket->setIDStringItem("WidgetType",
       this->metaObject()->className());
    // Komponentin nimi
    newPacket->setIDStringItem("WidgetName",
       this->objectName().toAscii().data());

    // Lähettäjä-käyttöliittymäkomponentin (tässä tapauksessa
    // pushButtonAdd tai pushButtonSubtract) nimi (add tai subtract)
    // lisätään signaaliin avaimeksi
    newSignal->setIDStringItem("Key",
       sender->objectName().toAscii().data());

    // Yläluokan emitSignal lähettää paketin viestinvälitykseen
    SignalContainer::emitSignal(newPacket);
}
```