STADIA
HELSINGIN AMMATTIKORKEAKOULU

TEKNIIKAN JA LIIKENTEEN TOIMIALA

Tietotekniikka

Ohjelmistotekniikka

INSINÖÖRITYÖ

SOVELLUSKEHYS VETOBOX-SOVELLUSPALVELIMEN MOBIILEILLE KÄYTTÖLIITTYMÄSOVELLUKSILLE

Työn tekijä: Marko Heiskanen
Työn valvoja: Simo Silander
Työn ohjaaja: Hannu Puhakka

Työ hyväksytty: __. __. 2004

Simo Silander
lehtori

ALKULAUSE

Tein insinöörityöni projektina Vetokonsultit Oy:lle. Projekti antoi minulle tilaisuuden tutustua Java 2 Micro Edition -ohjelmistotekniikkaan sekä käytännön että teorian tasolla. Työn lomassa opin paljon mobiililaite-ympäristössä tapahtuvasta haasteellisesta ohjelmistosuunnittelusta ja toteutuksesta. Uskon tästä olevan minulle hyötyä tulevaisuudessa mobiililaitteiden alati kasvavan ohjelmistotuotannon johdosta.

Haluan kiittää työni ohjaajaa, filosofian kandidaatti Hannu Puhakkaa käyttöliittymän suunnitteluun sekä käytettävyyteen liittyvistä ideoista ja neuvoista. Kiitokset myös diplomi-insinööri Aapo Puhakalle VetoBox-sovellusten ohjelmointiin liittyvistä neuvoista.

Erityiskiitokset perheelleni, joka on jaksanut tukea ja kannustaa koko opintojeni ajan, etenkin opinnäytetyön kriittisimmillä hetkillä. Kiitokset filosofian maisteri Tuula Rantaselle abstractin tarkistamisesta sekä työn valvojalle lehtori Simo Silanderille, kirjoitustyöhön liittyvistä neuvoista.

Keravalla 5.10.2004

Marko Heiskanen

STADIA
HELSINGIN AMMATTIKORKEAKOULU

TEKNIIKAN JA LIIKENTEEN TOIMIALA

INSINÖÖRITYÖN TIIVISTELMÄ

Tekijä:
Marko Heiskanen

Työn nimi:
Sovelluskehys VetoBox-sovelluspalvelimen mobiileille käyttöliittymäsovelluksille

Päivämäärä:	Sivumäärä:
5.10.2004	**84 + 17**

Koulutusohjelma:	Suuntautumisvaihtoehto:
Tietotekniikka	**Ohjelmistotekniikka**

Työn valvoja:
lehtori Simo Silander
Työn ohjaaja:
filosofian kandidaatti Hannu Puhakka

Tämä työ tehtiin Vetokonsultit Oy:lle. Työssä perehdyttiin Java 2 Micro Edition -teknologiaan, jonka pohjalta suunniteltiin sekä toteutettiin mobiililaitteille tarkoitettu käyttöliittymäsovelluskehys. Käyttöliittymäsovelluskehys suunniteltiin toimimaan Vetokonsulttien kehittämälle VetoBox-sovelluspalvelimelle.

Työ aloitettiin tutustumalla Java 2 Micro Edition -ympäristön sekä mobiiliohjelmoinnin erityispiirteisiin. Tämän jälkeen suunniteltiin sovelluskehyksessä käytettävä perus-arkkitehtuuri. Arkkitehtuurin valinnan ja suunnittelun jälkeen aloitettiin sovelluskehyksen toteuttaminen käyttämällä prototyyppimäistä lähestymistapaa. Sovelluskehyksen testaus suoritettiin käytännön tilanteissa eri projektien yhteydessä.

Toteutetun käyttöliittymäsovelluskehyksen avulla mallinnettiin VetoBox-sovelluspalvelimen normaalin käyttöliittymän toiminta ja ulkoasu mobiililaiteympäristöön sopivaksi. Sovelluskehyksen avulla saadaan yhdenmukaistettua mobiililaitteille tarkoitettujen sovellusten ulkoasua ja käyttäytymismallia sekä nopeutettua sovellusten kehitystyötä.

Avainsanat:
J2ME, sovelluskehys, MIDP, CLDC

STADIA
HELSINGIN AMMATTIKORKEAKOULU

HELSINKI POLYTECHNIC

ABSTRACT

Name:
Marko Heiskanen

Title:
J2ME based user interface framework for VetoBox-application server

Date:
October 5, 2004

Number of Pages:
84 + 17

Department:
Information Technology

Study Programme:
Software Engineering

Instructor:
Simo Silander, M. Sc.
Supervisor:
Hannu Puhakka, M. Sc.

In this graduate study, the Java 2 Micro Edition technology was studied for the purpose of developing a user interface framework for mobile devices. This framework was developed to enable the use of the VetoBox-application server made by Vetokonsultit Oy.

First, the basis of Java 2 Micro Edition was studied. Then, designing the framework was commenced by developing the architecture of the framework. After drawing up the architecture, the prototype phase followed. The prototype phase was started by making very simple software and continued by, after each new prototype, increasing its feature requirement. The prototype phase was completed when the developed software was able to communicate with the VetoBox-application server by using the VetoRPC-protocol. The following phase was to design the actual framework. The framework development, design and testing was carried out within several projects.

As a result of this graduate study, we have accomplished a user interface framework for Java 2 Micro Edition based mobile devices. Although the framework needs further developing, it speeds up the procedure of making mobile software for the VetoBox-application server.

Key Words:
J2ME, Application Framework, MIDP, CLDC

SISÄLLYS

SYMBOLILUETTELO

AMS	*Application Management System*; laitteistokohtainen ohjelmisto, joka hallitsee Java-ohjelmien asennusta, poistoa ja suoritusta
ASP	*Application Service Provider*; yritys joka vuokraa omissa tiloissa ja palvelimissa ylläpitämiään sovelluksia; asiakkaat ottavat yhteyden sovelluksiin joko yksityisillä yhteyksillä tai julkisen Internet-verkon kautta
AWT	*Abstract Window Toolkit*; Javan alkuperäinen käyttöliittymäkomponenttikirjasto, joka käyttää käyttöjärjestelmän natiiveja komponentteja
CDC	*Connected Device Configuration*; Java 2 Micro Edition -ympäristön konfiguraatio PDA- yms. laitteille
CLDC	*Connected Limited Device Configuration*; Java 2 Micro Edition -ympäristön konfiguraatio matkapuhelimille, hakulaitteille yms. laitteille
CORBA	*Common Object Request Broker Architecture*; arkkitehtuuri ja määritelmä hajautettujen objektien luontiin, jakeluun ja hallinnointiin
GCF	*Generic Connection Framework*; CLDC:n määrittelemä yleinen tietoliikennekehys
GOF	*The Gang of Four*; yleisesti käytetty nimitys tunnetun ohjelmistoalan kirjan ”Design Patterns: Elements of Reusable Object-Oriented Software” neljästä kirjoittajista
ITU-T	*International Telecommunications Union - Telecommunications standardization sector*; tietoliikennestandardeja sekä suosituksia tekevä järjestö

J2EE *Java 2 Enterprise Edition*; Sun Microsystemsin kehittämä Java-ohjelmointiympäristö hajautettujen sovellusten tuottamiseen

J2ME *Java 2 Micro Edition*; Sun Microsystemsin kehittämä Java-ohjelmointiympäristö pieniresurssisille kuluttajalaitteille ja sulautetuille järjestelmille

J2SE *Java 2 Standard Edition*; Sun Microsustemsin kehittämä Java -ohjelmointiympäristö.

JAD *Java Application Descriptor*; Java 2 Micro Edition MIDP -sovellusten kuvaustiedosto

JAR *Java Archive*; alustariippumaton tiedostomuoto, joka sallii monien tiedostojen yhteen kokoamisen; käytetään normaalisti Java-sovellusten jakeluun

JCP *Java Community Process*; prosessi Java-teknologia -määritelmien, referenssitoteutusten ja testiympäristöjen kehittämiseen sekä tarkistamiseen

JNI *Java Native Interface*; rajapinta, joka sallii Java-sovellusten käyttää Java VM:n ulkopuolista koodia; koodi voi olla kirjoitettu toisella ohjelmointikielellä, kuten C:llä tai C++:lla

JSR *Java Specification Request*; Java-teknologioita määritteleviä dokumentteja, jotka käyvät läpi JCP-prosessin

KVM *Kilobyte Virtual Machine*; J2ME CLDC:n referenssi Virtual Machine

MIDP *Mobile Information Device Profile*; J2ME-profiili, joka määrittelee käyttöliittymä-, tietoliikenne- ja tietojen tallennus -kirjastot CLDC-perusteisille laitteille

MVC	*Model-View-Contoller*; ohjelmistojen suunnittelussa sekä toteutuksessa käytettävä suunnittelumalli
OTA	*Over-The-Air*; MIDP-sovellusten Internetin yli tapahtuva asennusmenetelmä
PDA	*Personal Digital Assistant*; kämmentietokone, digitaalinen kalenteri, muistio, yms. laite; käytetään yleensä kuvaamaan tietokonetta, joka on tarpeeksi pieni mahtuakseen kämmenelle
QWERTY	standardi tietokoneen näppäimistö; nimi tulee näppäimistön ylimmän kirjainrivin kuuden ensimmäisen kirjainnäppäimen yhdistelmästä
RMI	*Remote Method Invocation*; Java-tekniikka kahden eri VM:n väliseen kommunikointiin
RMS	*Record Managemet System*; Java 2 Micro Edition MIDP:n määrittelemä tietojen tallennusmenetelmä
SOAP	*Simple Object Access Protocol*; hajautettujen järjestelmien tietojen välitykseen tarkoitettu protokolla
Swing	Javan käyttöliittymäkirjasto, joka tarjoaa AWT:tä laajemman valikoiman komponentteja; Swing on toteutettu Java-kielellä ja on käyttöjärjestelmäriippumaton
UML	*Unified Modeling Language*; standardi analysointi- ja suunnittelutapa, jota käytetään mallintamaan reaalimaailman objekteja tietokonesovellusten objekteiksi
VetoBox	Vetokonsultit Oy:n kehittämä Java-kielellä toteutettu sovelluspalvelin
VetoRPC	VetoBox-sovelluspalvelimen käyttämä etäproseduurien kutsuun tarkoitettu protokolla

1 JOHDANTO

Nykyään lähes jokaiselta löytyy jokin mobiililaite. Usein tämä on matkapuhelin, johon on pakattu mukaan mitä monimutkaisimpia ominaisuuksia, kuten digitaalikamerat, MP3-soittimet, FM-radiot sekä peli- ja ohjelmointiympäristö. Nämä ominaisuudet liittyvät joko suoraan tai epäsuorasti viihdesovelluksiin. Ohjelmointiympäristöt antavat viihdesovellusten lisäksi mahdollisuuden mobiililaitteiden hyödyntämiseen hyötysovellusten alueella.

Tähän asti mobiilisovellukset on lähinnä mielletty vapaa-ajan harrasteiksi ja peleiksi, mutta viime aikoina työelämässä on alkanut herätä kiinnostusta mobiililaitteiden hyötykäyttöön työnteon apuvälineenä. Tämä tarkoittaa uutta ja kasvavaa markkina-aluetta ohjelmistoja tuottaville yrityksille.

Mobiililaitteiden ohjelmointiympäristöiksi on markkinoilla monia vaihtoehtoja, joista suosituimpiin kuuluvat Java 2 Micro Editioniin (J2ME) perustuvat ympäristöt. J2ME:n suosio perustuu mm. sen avoimuuteen, alustariippumattomuuteen ja ohjelmistokehityksen kohtalaisen matalaan aloituskynnykseen. J2ME-alustaan perustuvien sovellusten pääasialliseksi kohteeksi on arvioitu erilaiset pelisovellutukset. Tämä ei kuitenkaan tarkoita, ettei se sopisi myös hyötysovellusten tuottamiseen.

Tässä opinnäytetyössä suunnitellaan ja toteutetaan J2ME MIDP -profiilin versioon 1.0 perustuva käyttöliittymäsovelluskehys. Sovelluskehys suunnitellaan hyödyntämään Vetokonsultit Oy:n kehittämän VetoBox-sovelluspalvelimen palveluita ja näin ollen tarjoamaan asiakkaille uusi käyttöliittymävaihtoehto perinteisten työasema- ja web-pohjaisten käyttöliittymien rinnalle. Sovelluskehyksen avulla on tarkoitus nopeuttaa ja yhtenäistää uusien sovellusten tuottamista mobiiliympäristössä. Käyttöliittymäsovelluskehyksen laitteistokohderyhmäksi on valittu matkapuhelimet niiden suosion johdosta.

J2ME:hen päädyttiin koska valitussa kohdelaitteistoissa se on ylivoimaisesti yleisin ohjelmointiympäristö. Yhtä tärkeä tekijä valintaan oli se, että VetoBox-sovelluspalvelin on toteutettu Java-kielellä. Näiden kahden seikan johdosta J2ME:n valinta mobiilisovelluksien alustaksi oli lähes itsestäänselvyys.

Aluksi työssä selvitetään J2ME-teknologian taustaa ja historiaa, jonka jälkeen käydään läpi sen perusarkkitehtuuri. Arkkitehtuurin jälkeen keskitytään työssä käytettyjen Connected Limited Device Configurationin (CLDC) ja Mobile Information Device Profilen (MIDP) ominaisuuksien tarkasteluun. Teknologia käydään läpi tarkasti, koska se poikkeaa merkittävästi J2SE- ja J2EE-tekniikoista. Tämän jälkeen selvitetään VetoBox-sovelluspalvelimen ja sitä käyttävien käyttöliittymien toimintaperiaatteita ja rakennetta työhön olennaisesti liittyvin osin.

Työssä käytettyjen teknologioiden selvityksen jälkeen käydään käyttöliittymäsovelluskehyksen arkkitehtuuri ja toteutus läpi. Toteutus käsitellään siten, että niihin liittyvä teoria ja työssä käytetyt menetelmät selostetaan toteutusta käsittelevän tekstin yhteydessä. Sovelluskehyksen rakenteen ja toteutuksen läpikäynnin jälkeen luodaan esimerkkisovellus vaihe-vaiheelta käyttäen työssä luotua sovelluskehystä. Lopuksi pohditaan työn onnistumista ja mahdollisia jatkokehitysmahdollisuuksia.

2 JAVA 2 MICRO EDITION -SOVELLUSKEHITYSYMPÄRISTÖ

Java ohjelmointikielen juuret juontavat 1990-luvulle, jolloin Sun Microsystems loi Oak-nimisen ohjelmointikielen. Oak oli tarkoitettu kulutuselektroniikkalaitteisiin ja sulautettuihin järjestelmiin kuten interaktiiviseen TV:hen ja kämmentietokoneisiin. Oakin suunnittelun lähtökohtina olivat luotettavuus, pienikokoisuus ja alustariippumattomuus. Tämäntyyppisten laitteiden ohjelmistojen tulee olla erittäin luotettavia sekä toimia hyvin vaatimattomilla muistimäärillä ja prosessoritehoilla verrattuna nykyisiin tietokoneisiin. Kämmentietokoneiden ja vastaavien laitteiden myynti ei kuitenkaan vielä lähtenyt käyntiin odotetulla tavalla, joten Sun Microsystems suuntasi katseensa suosiotaan kasvattavaan ja yleiseen tietoisuuteen tulleeseen Internetiin ja vaihtoi Oakin nimen Javaksi. Internetin suosion kasvun myötä myös Java tuli tunnetuksi erityisesti selainten sisällä toimivien Applettien johdosta. [1, s. 3 - 4; 2.]

Javan alustariippumattomuus ja mahdollisuus tehdä itsenäisiä sovelluksia Appletien lisäksi alkoi kiinnostaa sovelluskehittäjiä. Sun Microsystems laajensi nopeasti Java-alustan toiminnallisuutta lisäämällä siihen mm.

kattavamman käyttöliittymäkirjaston sekä hajautettujen järjestelmien ja parannellun tietoturvatuen. Kuten yleensä, ohjelmiston koko ja sen laitteistoille asettamat vaatimukset kasvoivat jokaisen uuden version myötä, eikä Java ollut tässä suhteessa mitenkään poikkeava. Kun ensimmäinen versio Java 2:sta esiteltiin, alusta oli kasvanut niin laajaksi, että se jaettiin kahteen osaan. Java 2 Standard Edition (J2SE) sisältää Javan ydintoiminnallisuuden sekä asiakassovelluksien vaatimat käyttöliittymäkirjastot. Java 2 Enterprise Edition (J2EE) puolestaan sisältää yritys- ja palvelintason sovellusten vaatimat kirjastot. [1, s. 3 - 4.]

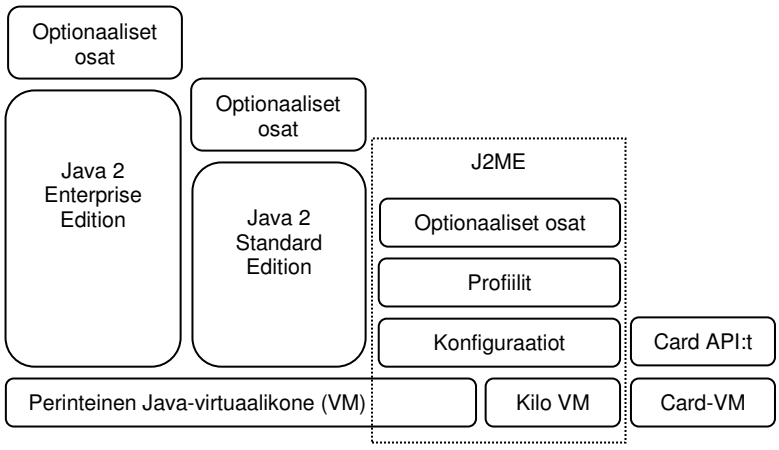
Samalla kun Javan kehitys ajautui yhä kauemmaksi sen alkuperäisestä kulutuselektroniikka ja sulautettujen järjestelmien juurista, alkoi kuluttajaelektroniikalle suunnatun Javan kaltaisen alustariippumattoman ohjelmointiympäristön tarve kasvaa. Tarve ilmaantui juuri samaisten laitteiden suosion kasvun myötä, joita varten Sun Microsystems aikoinaan Oak-ohjelmointikielen kehitti (kämmentietokoneet, interaktiiviset TV:t ja uutena sovellusalueena etenkin matkapuhelimet). Sun Microsystems reagoi tähän tarpeeseen luomalla erilaisia rajoitettuja Java-ympäristöjä, jotka kaikki pohjautuivat Java 2:n edeltäjään JDK 1.1:een. Jokainen näistä ympäristöistä ratkaisi omalla tavallaan ongelman pienentääkseen ympäristön vaatimukset käytettävissä oleviin resursseihin sopiviksi. [1, s. 4.]

Luodakseen yhtenäisen ympäristön kulutuselektroniikkalaitteiden sovelluskehitykseen Sun Microsystems kehitti kolmannen Java 2:een perustuvan ohjelmistoympäristön, jota kutsutaan nimellä Java 2 Micro Edition, eli lyhyesti J2ME. J2ME:n tarkoitus on yhtenäistää ja korvata aikaisemmat JDK 1.1:een perustuvat ratkaisut. Kulutuselektroniikkalaitteiden kirjo on kuitenkin valtava ja niiden tekniset ominaisuudet eroavat toisistaan hyvin paljon. Yksi yhtenäinen ohjelmistoalusta ei siis voi järkevästi kattaa kaikkia eikä edes suurinta osaa sulautetuista laitteista. J2ME onkin kokoelma spesifikaatioita, jotka määrittelevät joukon alustoja, joista kukin kattaa pienen osajoukon laitteista. [1, s. 4 - 5.]

J2ME:ssä nämä spesifikaatiot on jaettu kahteen tasoon: konfiguraatioihin ja profiileihin. Konfiguraatiot on kohdistettu laitteiden fyysisille ominaisuuksille ja ohjelmistoympäristölle. Profiilit taas on tarkoitettu täydentämään konfiguraatioita lisäämällä niihin laitteistoryhmä- tai markkinasegmentti-

kohtaisia ominaisuuksia. [1, s. 4 - 5.] Lisäksi on olemassa optionalisia paketteja, jotka eivät välttämättä ole riippuvaisia konfiguraatioista tai profiileista ja voivat periaatteessa toimia minkä tahansa näiden kombinaatioiden kanssa. Optionaliset paketit lisäävät ohjelmistoympäristöön tyypillisesti vain yhden lisäominaisuuden, kun profiilit puolestaan ovat laajempia kokonaisuuksia. J2ME-laitteet toteuttavat normaalisti ohjelmisto-alustan (software stack) pelkän yhden spesifikaation sijasta. Ohjelmisto-alusta koostuu normaalisti konfiguraatioista, profiilista ja mahdollisista optionalisista paketeista. [3.]

Kuva 1 esittää J2ME:n arkkitehtuurin ja sen suhteen muihin Java-ympäristöihin.



Kuva 1. J2ME-ympäristön arkkitehtuuri ja suhde muihin Java-standardeihin [4]

2.1 Java Community Process - Java-teknologioiden määritys

Kaikki J2ME-konfiguraatiot ja -profiilit kehitetään osana Java Community Processia (JCP). JCP on iteratiivinen prosessi, jonka avulla määritellään Java-teknologian spesifikaatiot. JCP on joukko saman alan yrityksiä, jotka yrittävät päästä yhteisymmärrykseen spesifikaatioista, joiden perusteella kaikki voivat tehdä tuotteensa. Jokainen spesifikaatio toteutetaan Java Specification Requestina (JSR). JCP itsessään on kuvattu JSR:nä, ja uusin JCP on kuvattu JSR 171 -dokumentissa. JSR 171 kuvaa spesifikaation läpikäymät vaiheet, jonka päävaiheet ovat aloitteen teko, aikainen vedos, julkinen vedos ja ylläpito. [5.]

Aloitteen teon tarkoitus on kuvata tuleva spesifikaatio, sen tarkoitus, laajuus, alustava aikataulu sekä asettaa sille reunaehdot, jotka sen tulee toteuttaa. Aloitteen tekijä valitsee alustavan asiantuntijaryhmän, joka suunnittelee ja toteuttaa spesifikaation. Aloitteen voi tehdä kuka tahansa JCP:n jäsen. Kun aloite on tehty, sille annetaan numero ja se laitetaan julkiseen tarkasteluun, minkä jälkeen pidetään hyväksyttämisäänestys. [5.]

Aikainen vedos tehdään, kun aloite on läpäissyt hyväksyttämisäänestyksen. Kun vedos on valmis, se laitetaan julkiseen tarkasteluun JCP:n Internet-sivuille, josta kuka tahansa voi lukea ja kommentoida sitä. Tämä julkinen tarkastelujakso kestää 30 - 90 päivää. Vedos voi sisältää ratkaisemattomia ongelmia ja muuttuu tämän jakson aikana useaan otteeseen. [1, s. 8; 5.]

Julkinen vedos spesifikaatiosta tehdään, kun aikaisen vedoksen tarkastelujakso on päättynyt. Julkista vedosta tehtäessä käytetään apuna kommentteja, joita edellisestä vaiheesta saatiin. Julkisen vedoksen valmistuttua se laitetaan Internetiin yleiseen tarkasteluun, joka on pituudeltaan 30 - 90 päivää. Spesifikaatio voi muuttua vielä tämänkin jakson aikana, mutta se jäädytetään kuitenkin 7 päivää ennen jakson loppumista. Julkisen vedoksen hyväksyttämisäänestyksen mennessä läpi asiantuntija-ryhmä valmistelee lopullisen vedoksen ehdotuksen tehden siihen edellisen vaiheen kommenttien perusteella aiheelliseksi katsomansa muutokset. [5.]

Ylläpitovaiheessa ylläpitojohdon (nimetty vastuuseen yksi henkilö; Maintenance Lead, ML) tehtävänä on käydä läpi spesifikaatiota koskevia kommentteja ja tunnistaa niistä yleisiä piirteitä sekä tehdä niiden perusteella muutosehdotuksia. [5.]

2.2 Konfiguraatiot

Konfiguraatio on spesifikaatio, joka määrittelee ohjelmistoympäristön laitteistoille, jotka täyttävät sen asettamat minimivaatimukset. Jotta laitteiston valmistaja voisi sanoa jonkin laitteen olevan J2ME:n määrittelemän konfiguraation mukainen, laitteen täytyy toteuttaa täysin kyseisen konfiguraation spesifikaatio. Tällöin ohjelmistonkehittäjä voi luottaa siihen, että ohjelmistoympäristö on yhdenmukainen saman konfiguraation toteuttavissa laitteissa. Näin ohjelmoijat voivat luoda laitteisto- ja valmistaja-riippumatonta koodia. [1, s. 5; 4, s. 3.]

Konfiguraatioiden tarkoitus on määritellä ohjelmistoympäristö mahdollisimman laajalle laitteistovalikoimalle. Tästä syystä konfiguraatio asettaa hyvin vähän vaatimuksia laitteistolle. Asioita, jotka konfiguraatio laitteistopuolella määrittelee, ovat muistin määrä ja prosessoriteho sekä käytettävät tietoliikenneyhteydet. Konfiguraation tehtävänä on myös määrittää ohjelmistoympäristöön liittyvät vaatimukset kuten itse Java-kieli ja virtuaalikoneen (Virtual Machine, VM) ominaisuudet sekä käytettävissä oleva luokkakirjaston minimikokoonpano. [1, s. 5; 6, s. 14.]

J2ME määrittelee tällä hetkellä kaksi konfiguraatiota, jotka ovat Connected Device Configuration (CDC) ja Connected Limited Device Configuration (CLDC). Näistä CLDC on tarkoitettu pieniin kuluttajaelektroniikkalaitteisiin, joissa on hyvin rajallinen muisti ja prosessoriteho. Tavallisimpia CLDC-laitteita ovat matkapuhelimet ja kämmentietokoneet (Personal Digital Assistant, PDA), joissa on noin 512 kt:n muisti ja vaatimaton prosessoriteho sekä jonkinasteinen tietoliikenneyhteys. CDC puolestaan on tarkoitettu laitteisiin, jotka sijoittuvat CLDC:n vaatimukset täyttävien laitteiden ja täyden J2SE-ympäristön väliin. Näissä laitteissa on enemmän muistia, tyypillisesti 2 Mt tai enemmän, sekä tehokkaammat prosessorit kuin CLDC-laitteissa. Tällaisia laitteita ovat esimerkiksi PDA:t, reitittimet, digitaali-TV:t ja älypuhelimet. [1, s. 5.]

Taulukossa 1 on esitettynä J2ME:ssä tällä hetkellä määriteltyjen konfiguraatioiden JSR-dokumentin numero, lyhenne sekä nimi.

Taulukko 1. J2ME:n kongfiguraatioiden JSR-dokumentien numerot, lyhenteet ja nimet

JSR numero	Lyhenne	Nimi
JSR 30	CLDC 1.0	Connected, Limited Device Configuration 1.0
JSR 139	CLDC 1.1	Connected, Limited Device Configuration 1.1
JSR 36	CDC	Connected Device Configuration

2.3 Profiilit

Kaksi laitetta voi täyttää ja toteuttaa saman konfiguraation määritykset, mutta silti ne voivat olla käyttötarkoitukseltaan tai laitteistoltaan hyvinkin erilaiset. Esimerkiksi pakastin ja matkapuhelin voivat molemmat toteuttaa CLDC:n, mutta niiden käyttö ja laitteisto poikkeavat ratkaisevasti toisistaan. Molemmat laitteet tarvitsevat täyden hyödyn saamiseksi lisäominaisuuksia, joita ei ole määritelty CLDC:ssä. Pakastin voisi esimerkiksi tarvita lämpötila- ja kosteusanturin lukumahdollisuuden ja matkapuhelin tekstinsyöttömahdollisuuden. Tällaisia laiteryhmä- tai markkinasegmenttikohtaisia ominaisuuksia varten on tehty lisämäärityksiä, joita kutsutaan profiileiksi. [4, s. 4; 6, s. 16.]

Profiilit täydentävät konfiguraatiota lisäämällä niihin erityispiirteitä. Käytännössä erityispiirteet toteutetaan siten, että profiilit lisäävät luokkia ja luokkakirjastoja konfiguraation päälle. Esimerkiksi pakastimelle ja muille kylmälaitteille voitaisiin määritellä kylmälaiteprofiili sekä matkapuhelimille matkapuhelinprofiili. Profiileille pätee sama kuten konfiguraatioillekin: jos laitevalmistaja ilmoittaa laitteensa toteuttavan jonkin J2ME-profiilin, hän sitoutuu toteuttamaan kaikki profiilin määrittelemät ominaisuudet sekä laitteisto- että ohjelmistotasolla. Profiilit voivat pohjautua myös johonkin toiseen profiiliin, jolloin luonnollisesti laitteen pitää toteuttaa kaikkien vaadittavien profiilien vaatimukset. [1, s. 7.]

Molemmissa, sekä CDC:ssä että CLDC:ssä, on yksi perusprofiili, jonka varaan muut profiilit tukeutuvat. CDC:ssä tämä profiili on Foundation Profile (FP), joka perustuu J2SE-sovellusohjelmointirajapintaan (Application Programming Interface, API). Vaikka FP sisältääkin suurimman osan J2SE:n ydinkirjastosta, se eroaa J2SE:stä siten, että se ei sisällä lainkaan graafiseen käyttöliittymään liittyviä luokkia. Luonnollisesti luokat on optimoitu mahdollisimman resursseja säästäviksi. [4, s. 7, 240.] Graafisen käyttöliittymän luokat lisätään vasta Personal Basis- ja Personal Profile -profiileissa. [4, s. 7.]

CLDC:ssä perusprofiili on nimeltään Mobile Information Device Profile (MIDP). Profiili lisää CLDC:hen kirjastot tietoliikennettä, paikallista tallennusta ja yksinkertaisia käyttöliittymiä varten. [1, s. 7.]

Taulukossa 2 on lueteltu nykyisten valmiiden profiilien JSR-dokumentaation numerot, lyhenteet, nimet sekä konfiguraatio, jonka päälle se on tarkoitettu.

Taulukko 2. J2ME:n nykyisien profiilien JSR-dokumentin numero, lyhenne, nimi ja konfiguraatio, jonka päälle profiili sijoittuu

JSR numero	Lyhenne	Nimi	Konfiguraatio
JSR 37	MIDP 1.0	Mobile Information Device Profile	CLDC 1.0
JSR 118	MIDP 2.0	Mobile Information Device Profile 2.0	CLDC 1.0
JSR 46	FP	Foundation Profile	CDC
JSR 129	PBP	Personal Basis Profile	CDC
JSR 62	PP	Personal Profile	CDC
JSR 195	IMP	Information Module Profile	CLDC 1.0

2.4 Optionaliset paketit

Konfiguraatioiden ja profiilien lisäksi J2ME:ssä on monia optionalisia paketteja. Nämä paketit eivät välttämättä liity mihinkään olemassa olevaan konfiguraatioon tai profiiliin, vaan ne voidaan liittää lähes mihin tahansa konfiguraatio- ja profiilikombinaatioon [3]. Optionaliset paketit lisäävät tyypillisesti jonkin ominaisuuden tai joukon samankaltaisia ominaisuuksia ohjelmistoalustaan ja laajentavat näin profiilia. Kun J2ME:hen tehdään uusi

API, se toteutetaan ensin optionalisena pakettina, ja jos tämä paketti vakiinnuttaa asemansa, siitä saattaa tulla osa seuraavaa profiilia. [2.]

Optionalisista paketeista mielenkiintoisimmat, osittain laajimpien laitteistotukiensa ansiosta, ovat Java API Bluetoothille, Wireless Messaging API, Mobile Media API, RMI Optional Package ja the JDBC Optional Package. Näistä JDBC Optional Package on tarkoitettu vain CDC- ja FP-yhdistelmälle. Uusien optionalisten pakettien käytettävyyttä ja mielenkiintoa vähentää niiden kohtalaisen hidas laitteistotuen kasvu.

Taulukossa 3 esitellään J2ME:n valmiiden optionalisten pakettien JSR-dokumenttien numerot, lyhenteet sekä nimet.

Taulukko 3. Valmiiden optionalisten pakettien JSR-dokumenttien numerot, lyhenteet ja nimet

JSR numero	Lyhenne	Nimi
JSR 66	RMI OP	RMI Optional Package Specification Version 1.0
JSR 75	PIM	PDA Optional Packages for the J2ME™ Platform
JSR 82	BTAPI	Java APIs for Bluetooth
JSR 120	WMA	Wireless Messaging API
JSR 135	MMAPI	Mobile Media API
JSR 169	-	JDBC Optional Package for CDC/Foundation Profile
JSR 172	-	J2ME™ Web Services Specification
JSR 177	-	Security and Trust Services API for J2ME™
JSR 179	-	Location API for J2ME™
JSR 180	SIPAPI	SIP API for J2ME™
JSR 184	3DAPI	Mobile 3D Graphics API for J2ME™
JSR 205	WMA 2.0	Wireless Messaging API 2.0

3 CONNECTED LIMITED DEVICE CONFIGURATION 1.0

Tässä luvussa käydään läpi niitä CLDC 1.0:n ominaisuuksia, jotka poikkeavat totutuista J2SE:n ominaisuuksista. Kuten jo konfiguraatioita esittelevässä kappaleessa on mainittu, CLDC sisältää vain minimijoukon J2SE:n paketeista ja luokista sekä virtuaalikoneen, jonka ominaisuuksia on supistettu. Siitä voidaankin sanoa, että se on pienin yhteinen tekijä

kohdelaitteistollensa. CLDC tekee hyvin vähän oletuksia laitteistosta, joille se on tarkoitettu. Muistivaatimuksena CLDC olettaa saavansa 160 kt pysyvää muistia virtuaalikonetta ja ydinluokkakirjastoja varten sekä 32 kt RAM-muistia ohjelmien dynaamisen muistin varaukseen. Dynaaminen muistin varaus käsittää mm. luokkien latauksen, ohjelman keon sekä pinon. CLDC olettaa nimensä mukaisesti, että laitteessa on jonkinasteinen tietoliikenneyhteys. Minimiprosessoritehoa ei määritellä spesifikaatiossa, mutta tyypillisesti CLDC-laitteissa prosessori on 16- tai 32-bittinen ja kellotaajuudeltaan noin 12 - 35 MHz. [1, s. 11 - 12; 2.]

Ohjelmistovaatimuksena CLDC olettaa, että laitteesta löytyy käyttöjärjestelmä, joka voi suorittaa virtuaalikonetta. Vaikka Java ja CLDC tukevat monisäikeisyyttä, ei CLDC oleta laitteiston käyttöjärjestelmältä monisäikeisyyden tukea. Sen sijaan itse virtuaalikoneen täytyy pystyä antamaan ainakin illuusio monisäikeisestä ympäristöstä, jotta säikeitä voitaisiin käyttää CLDC-sovelluksissa. [1, s. 12; 7.]

3.1 CLDC:n määrittelemä virtuaalikone ja Java-kieli

Ominaisuudet, jotka virtuaalikoneen tulee toteuttaa, kuvataan CLDC-spesifikaatiossa. Se määrittelee ne J2SE-virtuaalikoneen ominaisuudet, joita CLDC-virtuaalikoneen ei tarvitse toteuttaa. Sun Microsystems tarjoaa referenssitoteutuksen CLDC-spesifikaation vaatimukset täyttämästä virtuaalikoneesta, jota kutsutaan Kilobyte Virtual Machineksi (KVM). Vaikka KVM on kaikkien käytettävissä, se ei ole ainoa CLDC:n vaatimukset täyttävä virtuaalikone. Muun muassa IBM on tehnyt J9-virtuaalikoneen, joka täyttää CLDC:n vaatimukset. [1, s. 12 - 13.]

Yksi näkyvimmistä rajoituksista CLDC VM:ssä on liukulukuoperaatioiden puuttuminen. Liukulukuoperaatioiden puuttuminen selitetään sillä, että laitteet, jolle CLDC on suunnattu, eivät useinkaan sisällä laitteistotason liukulukuaritmetiikan tukea. Vaikka liukulukuoperaatioita voidaan emuloida ohjelmallisesti, sitäkään ei ole spesifikaatiossa määritelty pakolliseksi, koska sen resurssivaatimukset on koettu liian raskaiksi [6, s. 42]. Tästä seuraa välittömästi se, ettei ohjelmissa voida käyttää float- ja double-tyyppisiä

muuttujia, vakioita, metodin argumentteja ja metodin paluuarvoja eikä näiden tyyppien kääreluokkia *Float* ja *Double* [1, s. 13; 8].

Liukulukujen puutteen lisäksi on muita rajoituksia ja ominaisuuksia, jotka puuttuvat CLDC:n määrittelemästä Java-kielestä. Luokkien ominaisuuksien tutkimiseen tarkoitettu reflection-paketti (java.lang.reflect) puuttuu kokonaan tietoturvasyistä sekä osittain myös muistin säästämiseksi [1, s. 13 - 14]. Heikot viittaukset (Weak references) -paketti (java.lang.ref) puuttuu kokonaan CLDC 1.0:sta sen implementoimiseen vaaditun muistimäärän vuoksi [1, s. 14]. Object-luokka ei sisällä finalize()-metodia, koska objektien viimeistelyn toteuttaminen aiheuttaa huomattavasti monimutkaisuutta virtuaalikoneeseen ja siitä saatava hyöty jää verrattain pieneksi [1, s. 14; 8]. Sarjallistuvuutta (ts. *Serializable*-rajapintaa) eikä Remote Method Invocationia (RMI) vaadita CLDC:n määrityksissä [8].

Poikkeuksien (*Exception*) ja virheiden (*Error*) luokkien määrää on rajoitettu huomattavasti. Virhetilanteiden hallinta on siirretty pääasiassa isäntälaitteiston vastuulle sen sijaan, että niistä raportoitaisiin sovellukselle (VM:lle). Tämä ei tarkoita, että sovelluksia tehtäessä ei tarvitse varautua virheisiin ja poikkeustilanteisiin. Esimerkiksi muistin loppuminen ja siitä aiheutuva java.lang.*OutOfMemoryError*-virhetilanne on huomattavasti todennäköisempi kuin normaalia J2SE (tai J2EE) -sovellusta tehtäessä. Näin säästetään huomattavasti muistia, koska implementoitavien luokkien määrä pienenee oleellisesti. [1, s. 14; 8.]

Toisin kuin J2SE, CLDC ei tarjoa Java Native Interfacea (JNI), jonka avulla voidaan suorittaa kohdelaitteiston/käyttöjärjestelmän alkuperäistä (natiivia) koodia ajon aikana. Tämä estää käyttämästä järjestelmän tarjoamia ominaisuuksia, ellei CLCD tai jokin profiili tarjoa rajapintaa siihen. Laitteiston valmistajat voivat kuitenkin halutessaan laajentaa sovelluksille tarjottavaa API-rajapintaa esilinkittämällä natiivia koodia itse rakentamaansa virtuaalikoneeseen. Tämä vaihtoehto koskee vain laitteiston valmistajia, joten sovelluskehittäjällä ei ole mahdollista suorittaa laitteiston natiivia koodia. [1, s. 13 - 14, 16.] Myös JNI on poistettu vaatimuksista lähinnä sen toteuttamisen muistin tarpeen vuoksi sekä osittain tietoturvan takia [8].

Luokkien latausmekanismi poikkeaa normaalista J2SE:n mallista. CLDC:ssä määritellään, että implementaatioiden täytyy tarjota luokkien lataaja (Class loader), jota ei voida ylikirjoittaa tai laajentaa sovelluskohtaisella koodilla. Sovellukset eivät ts. saa vaikuttaa millään tavalla siihen prosessiin, jolla järjestelmän luokkienlataaja etsii ja lataa luokat käyttöönsä. Tällä estetään sovellusten yritykset korvata ydinluokat (java- ja javax.microedition-pakettien luokat) omilla toteutuksillaan. Ydinluokkien korvaaminen sovelluskohtaisilla luokilla olisi tietoturvariski ympäristössä, johon ei voida toteuttaa J2SE:n kaltaista hienorakenteista tietoturvamallia. Sen toteuttaminen vaatisi yksinkertaisesti liikaa prosessoritehoa ja muistia sopiakseen CLDC-spesifikaation kohdelaitteisiin. [1, s. 14 - 15.]

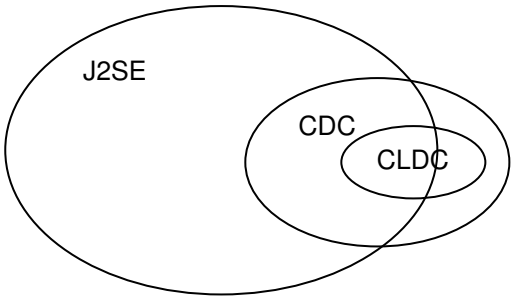
3.2 Luokkakirjastot

CLDC:n sisältämä luokkakirjasto on erittäin pieni. Se sisältää vain osan J2SE:n pakettien java.io-, java.lang- ja java.util-luokista. J2SE:n yhteisten pakettien lisäksi on määritelty uusi javax.microedition.io-paketti, joka on luokille, jotka eivät ole yhteisiä J2SE:n ja J2ME:n kanssa. Luokkien, jotka J2ME jakaa J2SE:n kanssa, täytyy noudattaa seuraavia sääntöjä:

- Luokkien ja pakettien nimien tulee olla samat.
- J2ME:n luokkien ja pakettien semantiikan täytyy olla sama kuin samannimisillä J2SE:n luokilla ja paketeilla.
- Luokkiin ei saa lisätä julkisia (public) tai suojattuja (protected) metodeita tai jäsenmuuttujia.
- J2ME-luokat eivät saa lisätä toiminnallisuutta luokkiin, jota ei J2SE:n vastaavissa luokissa tai paketeissa ole.

Nämä säännöt varmistavat sen, että J2ME:n luokka on aina joko osajoukko tai identtinen samannimisen J2SE-luokan kanssa. Tämä puolestaan varmistaa yhteensopivuuden J2ME:stä J2SE:hen samannimisten luokkien ja pakettien osalta. [1, s. 19 - 20.]

Kuvassa 2 esitetään CLDC:n ja J2SE:n suhdetta joukko-opillisen esitystavan avulla. Kuvaan on otettu myös mukaan CDC, jonka osajoukko CLDC on. Kuvasta havaitaan kuinka CLDC:sta osa on standardin J2SE:sen ulkopuolella. Tämä osuus kuvaa javax.microedition-pakettia ja sen luokkia.



Kuva 2. J2SE:n, CDC:n ja DLDC:n suhteet toisiinsa nähden joukko-opillisesti kuvattuna [6, s. 15]

3.2.1 java.lang

Noin puolet J2SE:n java.lang-paketin luokista sisältyy CLDC:n määrittelemään java.lang-pakettiin ja mukana olevista luokista kaikki eivät ole täysiä J2SE-toteutuksia. *Object*-luokka ei sisällä finalize()-metodia. *Cloneable*-rajapinta on poistettu, joten Object ei sisällä clone() metodia ja näin ollen ei ole yleistä tapaa kloonata olioita. Numeerisista luokista *Float*- ja *Double*-luokkien lisäksi on jätetty pois J2SE:ssä ollut numeroluokkien yhteinen kantaluokka *Number*, jolloin numeroluokat *Byte*, *Integer*, *Long* ja *Short* on peritty *Object*-luokasta. Kaikista CLDC:n luokista on poistettu kaikki liukulukuja tai *Number*-luokkaa käsittelevät metodit. Luokkien vertailuun tarkoitettu *Comparable*-rajapinta on jätetty pois CLDC:n määrityksistä. Luokasta java.lang.*Class* on poistettu kaikki reflection-pakettiin liittyvät metodit, koska CLDC ei tue reflection-ominaisuutta, kuten aikaisemmin on jo todettu. CLDC:n versiossa 1.0 ei ole java.lang.ref-pakettia ja sen luokkia lainkaan. [2; 7.]

J2SE:n *System*- ja *Runtime*-luokat sisältävät paljon sellaisia metodeja, jotka vaativat isäntäjärjestelmän natiivin koodin suorittamista. CLDC:n määrittelemän virtuaalikoneen rajoitusten vuoksi monet näistä metodeista on jätetty pois. Esimerkiksi input-, output- ja error- standardivirtojen asetus-

metodit puuttuvat sekä metodit, joilla saa viittauksen ja pystyy vaihtamaan aktiivisen *SecurityManager*-luokan. [1, s. 22.]

Säikeiden tuesta huolimatta CLDC:n säieominaisuuksia on rajoitettu. CLDC ei tue daemon säikeitä (daemon threads), eikä säieryhmiä (thread groups). CLDC:n *Thread*-luokka ei ole täysin samanlainen kuin J2SE:n vastaava luokka. Säieryhmien ja daemon-säikeisiin liittyvien metodien lisäksi on *Thread*-luokasta poistettu destroy()-, interrupt()- ja isInterrupted()-metodit sekä J2SE:ssä vanhentuneiksi merkityt resume()-, suspend()- ja stop()-metodit. Näiden metodien puuttumisen johdosta ainoa tapa keskeyttää säikeen suoritus on run()-metodin suorituksen keskeyttäminen. J2SE:ssä yleisesti säikeissä tapahtuvien virheiden lokiin kirjoittamiseen ja virhetilanteiden etsimiseen käytettyä dumpStack()-metodia ei myöskään ole määritelty CLDC:ssä. [1, s. 14, 22 - 23; 7.]

3.2.2 java.util

Päivämääräluokka *Date* on CLDC:ssä vain kääreluokka long-arvolle, joka esittää kulunutta aikaa millisekunteina hetkestä 1.1.1970 kello 00.00 GMT. *Date*-luokkaa onkin tarkoitettu käytettäväksi *Calendar*-luokan kanssa, joka on yksinkertaistettu versio vastaavasta J2SE:n luokasta. Luokan tehtävänä on muuntaa annettu aika (*Date*-luokan ilmentymä) sitä vastaavaksi vuoden, kuukauden, päivän, minuutin ja sekunnin arvoiksi sekä päinvastoin. Muunnoksen tulos on riippuvainen kahdesta tekijästä: aikavyöhykkeestä, jonka alaisena muunnos on tehty sekä kalenterin säännöistä, jotka määräytyvät ympäristöstä, jossa laitetta käytettään. Aikavyöhyke voidaan asettaa käyttämällä apuna *TimeZone*-luokkaa, joka edustaa siirtymää GMT-aikaan nähden (kokonaislukua). *TimerZone*-luokalta voidaan pyytää oikeaa aikavyöhykettä käyttämällä aikavyöhykkeen nimeä. *Calendar*-luokka itsessään on abstrakti luokka ja sitä toteuttavan luokan ilmentymä saadaan käyttöön getInstance()-metodilla. [1, s. 24 - 25.]

Kokoelmaluokista on util-pakettiin sisällytetty vain JDK 1.1:een kuuluvia luokkia ja Java 2 kokoelmakehyksen luokat ovat kokonaan poistettu. Tästä johtuen osa *Hashtable*- ja *Vector*-luokkien metodeista on poistettu. [1, s. 24.]

3.2.3 java.io

CLDC:n java.io-paketti sisältää erittäin rajoitetun osajoukon J2SE:n java.io-paketista. Paketissa määritellään *InputStream*- ja *OutputStream*-rajapinnat. Ainoat luokat, jotka oikeasti käsittelevät tietolähteitä ovat *ByteArrayInputStream* ja *ByteArrayOutputStream*. Nämä luokat voidaan kuorruttaa *DataInputStream*- ja *DataOutputStream*-luokilla sekä *InputStreamReader*- ja *OutputStreamReader*-luokilla. *ByteArray* kuorruttajat ovat perustietotyyppien, kuten int ja long, lukemista ja kirjoittamista varten. *InputStreamReader*- ja *OutputStreamReader*-luokat ovat merkkitiedon lukemista ja kirjoittamista varten. [1, s. 27.]

3.2.4 javax.microedition.io

javax.microedition.io on CLDC:n ainoa paketti jota ei löydy J2SE:n luokkakirjastosta. Paketti sisältää kokoelman rajapintoja, jotka luovat J2ME:lle varta vasten luodun tietoliikennekehyksen, jota kutsutaan Generic Connection Frameworkiksi (GCF). Se määrittelee yleisen tavan päästä käsiksi resursseihin, joita osoitetaan nimellä ja käytetään *InputStream*- tai *OutputStream*-rajapintoja toteuttavien luokkien avulla. Lähtökohtaisesti tämä on tarkoitettu tietoliikennettä varten, mutta mikään ei estä toteuttamasta myös muun tyyppistä resurssien käsittelyä sen avulla. CLDC vain määrittelee rajapinnat ja metodit kehykselle sekä ehdottaa kuinka niitä tulisi käyttää, mutta ei itsessään vaadi yhtään varsinaista toteutusta. [1, s. 27.]

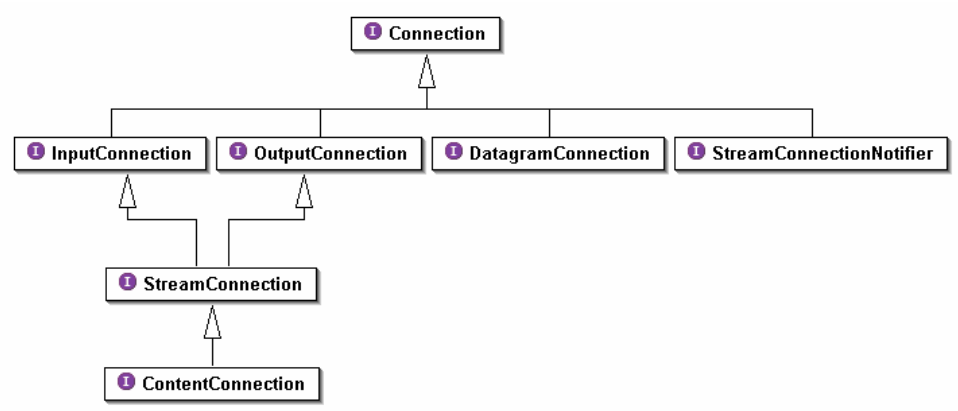
3.3 Generic Connection Framework -tietoliikennekehys

Sen sijaan, että CLDC:ssä määriteltäisiin käytettäväksi osaa tai kaikkia J2SE:n tietoliikenneluokkia java.net-paketista, määritellään siinä kokonaan uusi kehys. Uusi kehys luotiin, koska J2SE:n java.net-paketti koostuu kymmenistä luokista ja niissä käytetään java.io-paketin luokkia, jotka eivät välttämättä sisälly mihinkään profiiliin. Tällainen luokkamäärä vaatisi aivan liian paljon muistia toteutettavaksi CLDC:n kohdelaitteistossa. Toisaalta uuden kehyksen määrittelyyn oli syynä se, että J2SE:ssä jokaista yhteystyyppiä varten käytetään omaa ohjelmointimallia. Esimerkiksi

ohjelmat, jotka käyttävät socket-yhteyksiä, tarvitsevat *Socket*-luokan luontia varten IP-osoitteen sekä porttinumeron ja puolestaan http-yhteyttä käyttävät ohjelmat tarvitsevat URL-osoitteen voidakseen luoda http-yhteyteen pystyvän luokan. J2SE:ssä jokaista yhteystyyppiä varten on olemassa oma konkreettinen luokka, jota ohjelmoija käyttää ohjelmia tehdessään. [1, s. 182.]

Tällainen lähestymistapa tietoliikenneohjelmointiin ei ole käytännöllistä CLDC:ssä, missä laitteiston tietoliikenneyhteyksien toteutukset voivat vaihdella erittäin paljon toisistaan. Esimerkiksi http-yhteyden toteutus kämmentietokoneeseen, joka on suoraan kytketty verkkoon (verkkokaapeli, usb, yms.), on todennäköisesti hyvin erilainen kuin matkapuhelimen, joka käyttää yhteyden muodostamiseen GPRS-yhteyttä. GCF tarjoaa yleisen tavan erilaisten yhteyksien muodostamiseen. Yleinen ja johdonmukainen yhteyden muodostaminen on saavutettu siten, että siihen käytetään tehdasluokkaa ja yhteystyypit toteutetaan laajennettavana rajapinta-hierarkiana. Lisäksi käytetään standardin muotoista Uniform Resource Locatoria (URL) kertomaan mihin ja minkä tyyppinen yhteys luodaan. [9.]

Kuvassa 3 esitetään UML-luokkakaaviona CLDC:n javax.microedition.io-paketin määrittelemä rajapintahierarkia.



Kuva 3. Generic Connection Framework rajapintahierarkia UML-luokkakaaviona [6, s. 62]

Kuvasta 3 nähdään, että hierarkian ylimmän tason muodostaa *Connection*-rajapinta. Se on yksinkertaisin yhteystyyppi ja kaikki muut yhteystyypit periytyvät siitä. Itse asiassa *Connection*-rajapinta on niin yksinkertainen, että

se määrittelee vain metodin close(), joka tarkoitus on sulkea yhteys. Seuraavassa kerroksessa laajennetaan *Connection*-rajapintaa niin, että tuetaan molempia perusyhteystyyppejä, sekä paketti- että virtapohjaisia. [6, s. 60 - 62; 10.]

Pakettipohjaisille yhteyksille määritellään *DatagramConnection*-rajapinta ja virtapohjaisille *InputConnection*-, *OutputConnection*- sekä *StreamConnectionNotifier*-rajapinnat. *StreamConnectinNotifier*-rajapinta antaa mahdollisuuden sovellusten kuunnella ulkoapäin tulevia yhteydenottoja asynkronisesti. *StreamConnection* perii sekä *InputConnection*- että *OutputConnection*-rajapinnat, jolloin siitä saadaan kaksisuuntaiset virtapohjaiset yhteydet mahdollistava rajapinta. *ContentConnection*-rajapinta periytyy *StreamConnectionista* ja lisää välitettävään tietoon liittyviä metodeita. Esimerkiksi datan pituuden, tyyppin ja käytetyn koodiston kertovat metodit. [9.]

CLDC:ssä määritelty rajapintahierarkia sisältää vain hyvin yleisen tason rajapintoja, mutta sitä voidaan helposti laajentaa lisäämällä siihen uusia rajapintoja. Rajapintoja voidaan lisätä mihin tahansa hierarkian kohtaan ja ne ovat tarkoitettu tehtäviksi osana profiilia tai omana optionalisena pakettina. [9.]

Jotta laajennettavasta luokkahierarkiasta saataisiin hyötyä, tarvitaan yleinen tapa luoda yhteyksiä. GCF:ssä tämä on toteutettu *Connector*-luokkalla, joka toimii tehdasluokkana. *Connector*-luokkaa pyydetään luomaan uusi yhteys käyttämällä jotakin luokan kolmesta open()-metodista. Kaikki kolme open()-metodia palauttavat *Connection*-rajapintaa toteuttavan luokan ja ottavat parametrina URL:n. URL on merkkijono, joka ilmoittaa, minkä tyyppinen yhteys luodaan, mihin resurssiin sekä mitä mahdollisia lisäparametreja käytetään. URL-merkkijonon yleinen muoto on seuraavanlainen:

protokolla:osoite;parametrit

Edellisessä URL-merkkijonossa protokolla-osa kertoo *Connector*-luokalle, minkä tyyppistä yhteyttä halutaan muodostaa ja osoite, mihin yhteys muodostetaan. Parametrit ovat optionalisia ja niitä voi olla kuinka monta tahansa, mutta niiden täytyy olla puolipisteellä eroteltuna.

URL-merkkijono voisi näyttää esimerkiksi tällaiselta:

comm:0;baudrate=28800;parity=even

tai

http://www.stadia.fi/index.htm

Ensimmäisessä pyydetään *Connector*-luokkaa luomaan sarjaporttiyhteys porttiin 0 nopeudella 28 800 kb/s ja pariteetilla parillinen. Toinen puolestaan pyytää luomaan http-yhteyden osoitteeseen www.stadia.fi/index.htm. [9.]

4 MOBILE INFORMATION DEVICE PROFILE 1.0

4.1 Laitteisto- ja ohjelmistovaatimukset

Tietojen tallennuksen, tietoliikenteen ja syöttölaitteiston tuen puute rajoittaa CLDC:n käytettävyyttä merkittävästi. Mobile Information Device Profile (MIDP) on profiili, joka on tarkoitettu lisäämään juuri nämä puutteet ja käytettäväksi laitteissa, kuten matkapuhelimet, kaksisuuntaiset hakulaitteet tai kämmentietokoneet. MIDP toimii CLDC:n päällä, joten kaikki CLDC:n laitteistolle ja ohjelmistolle asettamat vaatimukset ovat voimassa, mutta niiden lisäksi MIDP asettaa kuitenkin lisävaatimuksia sekä laitteistolle että sen käyttöjärjestelmälle. Laitteistolle asetettavat vaatimukset ovat [10, s. 25 - 26; 11, s. 21 - 22]:

- Näyttö joka on vähintään kaksivärinen ja resoluutioltaan 96x54 pikseliä, joiden reunojen suhde noin 1:1.
- Yksi tai useampi seuraavista syöttölaitteista: QWERTY-näppäimistö (kahdenkäden), ITU-T puhelinnäppäimistö (yhdenkäden) tai kosketus-näyttö.
- 128 kt pysyvää muistia (ROM, Flash tms.) MIDP-komponenteille.
- 8 kt pysyvää muistia sovellusten luomien tietojen tallennusta varten.
- 32 kt RAM muistia dynaamista muistin varausta varten (ts. Java kekoa varten).
- Kaksisuuntainen langaton tietoliikenneyhteys, joka on mahdollisesti katkonainen sekä kapeakaistainen.

Laitteistovaatimusten lisäksi MIDP 1.0 tarkentaa ja asettaa uusia vaatimuksia käyttöjärjestelmälle, joka suorittaa Java-virtuaalikonetta. Vaatimukset käyttöjärjestelmälle ovat [10, s. 26 - 27; 11, s.22 - 23]:

- Pienimuotoinen kernel, joka pystyy huolehtimaan laitteistosta ja tarjoamaan alkeellisia palveluita Java-ympäristölle (esimerkiksi keskeytyksiä, poikkeuksia ja minimaalista vuorontamista).
- Mekanismi pysyvästä muistista lukemiseen ja siihen kirjoittamiseen.
- Mekanismi langattoman tietoliikenneyhteyden lukemiseen ja kirjoittamiseen.
- Mekanismi näytön bittikarttamaiseen käsittelyyn, jossa jokainen yksittäinen pikseli voidaan asettaa miksi tahansa näytön tukemista väreistä [1, s.48].
- Mekanismi vähintään yhden laitteistovaatimuksessa kuvatun syöttölaitteen syötteiden vastaanottamiseen ja välittämiseen Java VM:lle.

4.2 MIDlet

Sovelluksia, jotka toimivat MIDP:ssä, kutsutaan MIDleteiksi. Kaikki MIDP-sovellukset sisältävät yhden luokan, joka periytyy javax.microedition.midlet-paketissa määritellystä abstraktista *MIDlet*-luokasta. *MIDletin*-luokan tärkein tehtävä on toimia käynnistysalustana (käynnistyspisteenä, entry point) MIDP-sovellukselle sekä hallita sovelluksen elinkaarta. Jokaisen laitteiston, joka toteuttaa MIDP-määrityksen, on toteutettava toiminnot, joilla käyttäjä voi asentaa, valita, suorittaa ja poistaa *MIDlettejä*. Kuvaamaan edellä mainittuja toimintoja käytetään yleisesti termiä sovelluksenhallintaohjelmisto (Application Management Software, AMS). [1, s. 49 - 50; 11, s. 39 - 40.]

4.2.1 MIDlet-pakkaukset

Yksittäinen *MIDlet* tai ryhmä läheisesti toisiinsa liittyviä *MIDlettejä* pakataan Java Archive -tiedostoon (JAR) muiden sovelluksessa käyttävien luokka-tiedostojen ja resurssien kanssa. Tätä JAR-tiedostoa kutsutaan MIDlet-pakkaukseksi (MIDlet suite). MIDlet-pakkaus asennetaan sekä poistetaan aina kokonaisuutena, eikä yksittäisten *MIDlettien* ja resurssitiedostojen (kuvat, tekstitiedostot, yms. tiedostot) asennus tai poisto pakkauksesta ole

sallittua [1, s. 50]. Yksi sovelluksenhallintaohjelmiston tehtävistä on käynnistää sovellukset ja tarjota niiden käytettäväksi seuraavat ominaisuudet [11, s. 40]:

- CLDC:n toteuttavat luokat (CLDC-luokkakirjasto) sekä Java-virtuaalikoneen.
- MIDP:n toteuttavat luokat (MIDP-luokkakirjasto).
- Kaikki luokat sovelluksen MIDlet-pakkauksesta.
- Kaikki muut MIDlet-pakkauksen tiedostot käytettäväksi resurssi-tiedostoina.
- Kuvaustiedoston sisältö.

Laiteisto voi tukea usean samassa MIDlet-pakkauksessa olevan *MIDletin* yhtäaikaista suorittamista, mutta on harvinaista, että *MIDletejä* suoritetaan samanaikaisesti. Jos näin tehdään, suoritetaan aktiivisia *MIDlettejä* samassa Java virtuaalikoneessa. Tästä ja edellä luetelluista ominaisuuksista seuraa, että samassa MIDlet-pakkauksessa olevat *MIDletit* jakavat kaikki luokkien instanssit ja resurssit, jotka on ladattu virtuaalikoneeseen. Täten esimerkiksi datan jakaminen on mahdollista MIDlet-pakkauksen luokkien välillä. Pysyvien tietojen tallennus hallitaan myös MIDlet-pakkauksen tasolla, johon palataan RMS:stä kertovassa luvussa. [1, s. 50.]

4.2.2 MIDletin kuvaustiedostot

MIDlet täytyy paketoida asianmukaisesti ennen kuin se voidaan toimittaa ja asentaa laitteeseen. Aikaisemmin mainittujen luokka- ja resurssitiedostojen lisäksi täytyy JAR-pakettiin sisällyttää manifest-tiedosto. Manifest-tiedoston tehtävä on kertoa sovelluksenhallintaohjelmistolle MIDlet-pakkauksen nimi, versio sekä mitkä luokkatiedostoista ovat *MIDlettejä* eli suoritettavissa olevia sovelluksia. Manifest-tiedoston käyttöön liittyy kuitenkin ongelma, koska se pakataan sovelluksen JAR-tiedostoon. Ongelmana on, että laitteen, joka aikoo käyttää manifest-tiedostoa, täytyy ladata koko JAR-paketti ja purkaa manifest-tiedosto paketista päästäkseen käsiksi sen tietoihin. Tällainen menettelytapa ei ole toimiva laitteissa, joille MIDP on suunnattu, sillä JAR-tiedoston koko on usein verrattain suuri suhteutettuna laitteiden tiedonsiirto-kapasiteettiin. Tällöin tuhlataan paljon aikaa tiedon siirtämiseen tapauksissa, joissa MIDlet-pakkaus hylätään välittömästi esimerkiksi väärän sisällön tai

version vuoksi. Tämän lisäksi monet yhteydentarjoajat veloittavat tietoliikenteestä siirretyn tiedon määrän mukaan, jolloin ylimääräinen tiedonsiirto aiheuttaa käyttäjälle lisäkuluja. [1, s. 51 - 53.]

Edellä mainituista syistä johtuen manifest-tiedoston sisältö sekä joitakin lisätietoja on erotettu omaksi tiedostoksi, jota kutsutaan nimellä Java Application Descriptor (JAD). JAD sisältää vain MIDlet-pakkauksesta kertovia ominaisuuksia, joten sen koko on huomattavasti pienempi kuin koko JAR-paketin ja näin ollen sen lataaminen vie huomattavasti vähemmän aikaa kuin JAR-paketin lataaminen. Laite näyttää käyttäjälle MIDlet-pakkauksen tiedot lataamastaan JAD-tiedostosta. Näiden tietojen perusteella käyttäjä voi tehdä päätöksen, lataako ja asentaako MIDlet-pakkauksen laitteeseensa. Jotta kohdelaitteen sovelluksenhallintaohjelmisto pystyisi tunnistamaan tiedoston sovelluksen kuvaustiedostoksi täytyy tiedoston päätteen olla ”.JAD” ja MIME-tyypin ”text/vnd.sun.j2me.app-descriptor”. [1, s. 53; 11, s. 44.]

Sekä manifest- että JAD-tiedostot ovat tekstitiedostoja, jotka sisältävät ominaisuuksien nimi-arvo-pareja. Ominaisuuksien nimi-arvo-parien muoto on seuraavanlainen:

ominaisuuden nimi: ominaisuuden arvo

Kaikki ominaisuudet, jotka ovat merkityksellisiä MIDlet-pakkauksen asennuksen kannalta alkavat ”MIDlet-” etuliitteellä. MIDP määrittelee kolme pakollista ominaisuutta, joiden täytyy olla molemmissa sekä manifest- että JAD-tiedostossa. Ominaisuuksien arvojen pitää olla identtisiä, sillä jos ne eroavat toisistaan, ei MIDlet-pakkausta voida asentaa laitteeseen. Sovelluskehittäjät voivat lisätä manifest- ja JAD-tiedostoihin omia nimi-arvo-pareja, kunhan niiden nimen alku ei ole ”MIDlet-”. Näihin ominaisuuksiin päästään sovelluksesta käsiksi *MIDlet*-luokan getAppProperty()-metodilla. [1, s. 51 - 54.]

Taulukossa 4 on esitelty kaikki MIDP 1.0:ssä määritellyt kuvaustiedostojen ominaisuudet sekä kerrottu ovatko ne pakollisia (P), optionaalisia (O) vai merkityksettömiä (M). Merkityksetön tarkottaa, että ominaisuus saa olla tiedostossa, mutta sitä ei käytetä mihinkään. [1, s. 51 - 54; 11, s. 41 - 46.]

Taulukko 4. MIDP 1.0:n määrittelemät MIDlet-paketin kuvaustiedoston ominaisuuksien nimet, arvojen selitykset sekä pakollisuus kuvaustiedostossa. Pakollisuuksien kuvaukset: P pakollinen, O optionaalinen ja M merkityksetön. [1, s. 52 - 53]

Ominaisuuden nimi	Ominaisuuden arvo ja selitys	Manifest	JAD
MIDlet-Name	MIDlet-paketin nimi	P	P
MIDlet-Version	MIDlet-paketin versio numero. Numerointi on muotoa a.b.c, missä a, b ja c voivat olla numeroita väliltä 0 - 99.	P	P
MIDlet-Vendor	MIDlet-paketin toimittajan nimi, joka on vapaamuotoista tekstiä.	P	P
MIDlet-n	Jokainen MIDlet-paketissa oleva *MIDlet* kuvataan ”nimi,ikoni,luokka” -ominaisuudella. Nimi on *MIDletin* nimi, ikoni on täydellinen hakemisto kuvaan, joka esittää sovellusta valikoissa ja luokka on *MIDlet*-luokan täydellinen nimi, jossa pakettien erottimena käytetään ’/’-merkkiä pisteen sijasta. Ikoni kohdan saa jättää tyhjäksi. Ominaisuuden nimessä oleva n korvataan juoksevalla numerolla, joka yksilöi jokaisen MIDlet-paketin *MIDletin*.	P	M
MicroEdition-Profile	MIDP:n versiot, joilla sovellus toimii. Jos versioita on useampia kuin yksi, ne erotetaan toisistaan välilyönnillä.	P	M
MicroEdition-Configuration	*MIDletien* vaatima konfiguraation versio.	P	M
MIDlet-Description	Kuvaus MIDlet-paketista, joka on vapaamuotoista tekstiä.	O	O
MIDlet-Icon	Täydellinen hakemisto ikonille, joka näytetään asennuksen yhteydessä. Ikonin tulee olla PNG-muotoinen.	O	O
MIDlet-Data-Size	Minimimäärä pysyvää muistia, jonka MIDlet-paketti tarvitsee tietojen tallentamiseen. Ei siis itse MIDlet-paketin asennuksen vaatima tila.	O	O
MIDlet-Jar-URL	URL-osoite JAR-tiedostoon, joka sisältää ominaisuuksien kuvaaman MIDlet-paketin. Ohjelmaa asennettaessa laite etsii JAR-tiedoston tästä osoitteesta. Jos ominaisuuden arvona on pelkkä JAR-tiedoston nimi, oletetaan sen olevan samassa osoitteessa kuin JAD-tiedosto.	M	P
MIDlet-Jar-Size	MIDlet-paketin JAR-tiedoston koko tavuina. Verrataan asennuksen yhteydessä JAR-tiedoston kokoon ja jos ei täsmää, MIDlet-pakettia ei asenneta.	M	P

Seuraavassa on esimekki JAD-tiedoston sisällöstä. Esimerkissä kaksi viimeistä ominaisuutta on sovelluskehittäjän määrittelemiä.

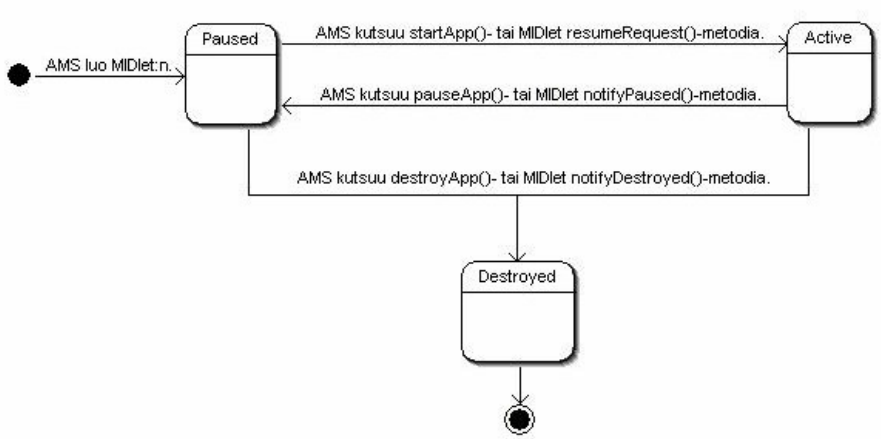
```
MIDlet-1: Asiakas, Asiakas.png, asiakas.AsiakasMIDlet
MIDlet-2: Parametrit, , com.veto.mobile.Parametrit
MIDlet-Jar-Size: 100
MIDlet-Jar-URL: Asiakas.jar
MIDlet-Name: Asiakas
MIDlet-Vendor: Vetokonsultit Oy
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-1.0
PORT: 12721
URL: kehitys.vetokonsultit.fi
```

4.2.3 MIDletin elinkaari

MIDletin elinkaari koostuu kolmesta tilasta, joissa *MIDlet* voi olla vain yhdessä kerrallaan. *MIDletin* mahdolliset tilat ovat Paused, Active ja Destroyed. Toisin kuten normaalilla J2SE-sovelluksilla *MIDletillä* ei ole julkista ja staattista void main() -metodia ja vaikka sellainen määriteltäisiinkin, sitä ei käytetä sovelluksen aloitusmetodina. *MIDletin* suoritus alkaa sovelluksenhallintaohjelman kutsuessa *MIDletin* abstraktia startApp()-metodia. *MIDletin* elinkaarta ja tilasta toiseen siirtymistä hallitsee laitteen sovelluksenhallintaohjelmisto, eikä sovelluskehittäjän tulisi luoda itse instansseja *MIDlet*-luokasta periytyvistä luokista. Sovelluksenhallintaohjelmisto käyttää *MIDletin* tilasta toiseen siirtämiseen *MIDlet*-luokassa määriteltyjä abstrakteja metodeja [6, s. 66 - 68; 11, s. 46 - 47.]:

- protected abstract void startApp()
- protected abstract void pauseApp()
- protected abstract void destroyApp(boolean unconditional)

Kuvassa 4 esitetään tilakaaviona *MIDletin* tilojen yhteydet ja mahdolliset siirtymät tilasta toiseen.



Kuva 4. *MIDletin* tilat ja niiden väliset tilasiirtymät. Tilasiirtymät voivat tapahtua joko sovelluksenhallintaohjelmiston (AMS) tai *MIDletin* toimesta. [1, s. 57]

Kun *MIDlet* ladataan, luodaan MIDletin ilmentymä (alustetaan instanssimuuttujat, kutsutaan parametritonta muodostinta) ja se siirretään Paused-tilaan. Jos *MIDletin* luonnin yhteydessä tapahtuu poikkeus, tuhotaan *MIDlet* ja sen suoritus lopetetaan. Mikäli poikkeusta ei tapahdu, sovelluksenhallintaohjelmisto siirtää *MIDletin* Paused-tilaan odottamaan sovelluksen aktivoimista. [1, s. 57 - 58; 11, s. 119 - 122.]

Sovelluksenhallintaohjelmiston päättäessä siirtää *MIDlet* Paused-tilasta Active-tilaan kutsuu se *MIDletin* startApp()-metodia. Jos startApp()-metodi kutsu suoritetaan onnistuneesti, siirtyy *MIDlet* Active-tilaan. Toisaalta jos metodin suorituksen aikana havaitaan virhetilanne, josta mahdollisesti voidaan toipua myöhemmin, voidaan *MIDletin* käynnistys keskeyttää ja siirtää myöhemmäksi aiheuttamalla *MIDletStateChangedException*. Tällöin sovelluksenhallintaohjelmiston tulee siirtää *MIDlet* Paused-tilaan odottamaan myöhempää ajankohtaa, jolloin sen käynnistämistä voidaan yrittää uudestaan. Havaittaessa virhetilanne, josta ei todennäköisesti voida toipua, tulisi *MIDletin* kutsua notifyDestroyed()-metodia. Käynnistyksen yhteydessä tapahtunut poikkeus, jota ei oteta kiinni, tulkitaan kriittiseksi virheeksi ja sovelluksen suoritus keskeytetään kutsumalla *MIDletin* destoyApp()-metodia. On syytä huomata, että startApp()-metodia kutsutaan aina

siirryttäessä Paused-tilasta Active-tilaan, mikä tarkoittaa, että sitä voidaan kutsua useaan kertaan sovelluksen elinkaaren aikana. [1, s. 57 - 58.]

MIDletin ollessa Active-tilassa, voi sovelluksenhallintaohjelmisto koska tahansa siirtää *MIDletin* Paused-tilaan. Tällainen tilanne voi tapahtua esimerkiksi matkapuhelimen ottaessa vastaan puhelun. Ennen tilan vaihtumista sovelluksenhallintaohjelmisto kutsuu *MIDletin* pauseApp()-metodia. Metodin tarkoitus on antaa sovelluskehittäjälle mahdollisuus vapauttaa sovelluksen käyttämiä resursseja ja mahdollisesti tallentaa sovelluksen tila pysyvään muistiin ennen Paused-tilaan siirtymistä. *MIDletin* siirtyminen Paused-tilaan tarkoittaa, että sovelluksella ei enää ole käytettävissään laitteen näyttöä. Kaikki säikeet ja ajastimet, jotka sovellus on luonut aikaisemmin jäävät kuitenkin aktiivisiksi. Sovelluskehittäjän pitää siis huolehtia myös niiden pysäyttämisestä ja vapauttamisesta tilasiirroksen yhteydessä, mikäli niitä ei tarkoituksella haluta jättää suoritettavaksi. [1, s. 58.]

Sovelluksenhallintaohjelmisto voi myös lopettaa *MIDletin* suorituksen, jolloin se kutsuu destroyApp(boolean unconditional)-metodia. Tässä metodissa on tarkoitus vapauttaa kaikki sovelluksen varaamat resurssit. Metodi saa parametrikseen boolean-arvon, joka ilmoittaa voidaanko sovelluksen tuhoaminen keskeyttää. Parametrin arvon ollessa looginen tosi, sovelluksen tuhoamista ei voida keskeyttää. Joissain tilanteissa voi kuitenkin olla hyödyllistä pystyä keskeyttämään sovelluksen tuhoaminen ja jatkaa sen suorittamista. Esimerkiksi, jos sovelluksella on tallennettavia tietoja, voidaan käyttäjää huomauttaa ja antaa mahdollisuus sovelluksen suorituksen jatkamiseen ja tietojen tallentamiseen. Sovelluksen tuhoaminen voidaan keskeyttää aiheuttamalla *MIDletStateChangedException*, joka on määritelty javax.microedition.midlet-paketissa. destroyApp()-metodin suorittamisen jälkeen *MIDletin* tila muuttuu Destroyed-tilaan, jossa *MIDlet* voi olla vain kerran elinkaarensa aikana. [1, s. 58 - 60; 11, s. 119 - 122.]

MIDlet-luokka tarjoaa sovelluskehittäjälle metodit oman tilansa vaihtamiseen. Nämä tilanvaihtometodit ovat:

- public final void notifyPaused()
- public final void resumeRequest()
- public final void notifyDestroyed()

notifyPaused()-metodilla sovelluskehittäjä pyytää sovelluksenhallinta-ohjelmistoa siirtämään *MIDlet* Paused-tilaan. resumeRequest()-metodilla voidaan taas pyytää palauttamaan *MIDlet* Active-tilaan. notifyDestroyed()-metodilla pyydetään sovelluksenhallintaohjelmistoa siirtämään *MIDlet* Destroyed-tilaan. notifyPaused()- ja notifyDestroyed()-metodeja kutsuttaessa sovelluksenhallintaohjelmisto ei kutsu pauseApp()- tai destroyApp()-metodeja vaan olettaa sovelluksen suorittaneen valmiiksi vaadittavat toimenpiteet. [1, s. 60.]

4.3 Record Management System

MIDP tarjoaa *MIDleteille* mekanismin tallentaa ja lukea tietoja pysyvästä muistista [11, s. 37]. Mekanismin nimi on Record Management System (RMS) ja sen toteuttavat luokat ja rajapinnat sijaitsevat javax.micro-editon.rms-paketissa. RMS on yksinkertainen järjestelmä, missä tiedot tallennetaan tietuekantoihin (RecordStore). Tietuekannat ovat tiedostoja, jotka sisältävät tietueita (Records), jotka koostuvat tietueen yksilöivästä kokonaisluvusta (id-numero) sekä itse tiedon sisältävästä byte-taulukosta. Tietuekantaa voidaan kuvata kaksiulotteisena taulukkona, jossa rivi kuvaa tietuetta ja rivissä olevasta kahdesta sarakkeesta toinen kuvaa tietueen id-numeroa ja toinen sen sisältämää dataa. Taulukko 5 esittää RMS:n tietuekannan ja sen tietueiden periaatteellista rakennetta. [10, s. 360.]

Taulukko 5. Tietuekannan periaatteellinen rakenne [10, s. 360]

Tietuekanta (RecordStore)	
Tietueen ID	**Tietueen data**
1	Tavutaulukko
2	Tavutaulukko
...	...

Jokainen tietuekanta sisältää myös versionumeron ja aikaleiman, joita muutetaan aina, kun tietuekantaan tehdään muutoksia. Versionumero on integer-tyyppinen arvo ja sitä kasvatetaan muutosten yhteydessä. MIDP:n määritykset eivät kuitenkaan määrittele, mistä numerosta versioinnin pitäisi

alkaa uuden tietuekannan luonnin yhteydessä, joten tämä saattaa vaihdella laitteistokohtaisesti. Aikaleima on long-tyyppinen arvo, joka on samaa muotoa kuin java.lang-paketin *System*.currentTimeMillis()-metodin paluu-arvo. [11, s. 38.]

Järjestelmä on pidetty yksinkertaisena, koska MIDP:n toteuttavat laitteet voivat sisältää hyvin erilaisia tiedostojärjestelmiä tai useissa tapauksissa ei tiedostojärjestelmää lainkaan. Tietuekantoja käsitellään MIDlet-paketin tasolla ja ainoa rajoittava tekijä niiden määrälle MIDlet-paketissa on, että tietuekannat ovat nimetty yksilöllisesti MIDlet-paketin laajuisesti. Tietuekannan nimi voi olla maksimissaan 32 unicode-merkkiä pitkä merkkijono, joka on merkkikokoriippuvainen. Turvallisuussyistä *MIDletit* eivät pääse käsiksi kuin itse ja samassa MIDlet-paketissa olevien *MIDlettien* luomiin tietuekantoihin. Tästä ominaisuudesta johtuen *MIDletit* eivät siis pääse käsiksi esimerkiksi puhelimen osoitekirjaan, kalenteriin tai muihin laitteen tiedostoihin. [11, s. 37 - 38.]

RMS sisältää yhden varsinaisen luokan. Se on *RecordStore* ja sillä voidaan luoda, avata, sulkea ja poistaa tietuekantoja sekä hallita yksittäisiä tietueita. *RecordStore* sisältää lisäksi metodeita, joilla saadaan tietoja tietuekannoista, esimerkiksi tietuekannan koko tavuina tai MIDlet-paketin kaikkien tietuekantojen nimet merkkijonotaulukkona. *RecordStore*-luokan lisäksi javax.microedition.rms-paketti sisältää neljä rajapintaa, jotka ovat *RecordComparator*, *RecordEnumeration*, *RecordFilter* ja *RecordListener*. *RecordComparator*- ja *RecordFilter*-rajapinnat on tarkoitettu käytettäviksi *RecordEnumeration*-rajapinnan kanssa. Tämä sisältää metodit tietueiden selaamiseen eteen- ja taaksepäin tietuekannassa. *RecordEnumeration*-rajapintaa toteuttava luokka saadaan *RecordStore*-luokalta metodilla [6, s.167 - 170]:

> *public RecordEnumeration enumerateRecords(RecordFilter filter, RecordComparator comparator, boolean keepUpdated)*

Metodin parametrina saatavan *RecordFilter*-rajapintaa toteuttavan olion avulla voidaan käydä läpi vain tietyt kriteerit täyttävät tietueet. Tietueille asetettavat kriteerit määräytyvät *RecordFilter*-rajapinnan ainoan metodin toteutuksessa, jonka sovelluskehittäjä itse kirjoittaa [1, s. 219 - 220]:

public boolean matches(byte[] data)

Metodin tulee palauttaa boolean-arvo true, jos tietue täyttää halutut kriteerit ja muussa tapauksessa arvo false. Kun *RecordEnumerator* luodaan, luetaan kaikki tietuekannan tietueet ja välitetään ne *RecordFilter*-rajapintaa toteuttavan luokan matches()-metodille. Lopulliseen palautettavaan enumeration-olioon sisällytetään vain ne tietueet, joilla matches()-metodi palautti arvon true. [1, s. 219 - 220.]

enumerateRecords()-metodin toinen parametri on vain yhden metodin määrittävä *RecordComparator*-rajapinta, jonka avulla läpikäytävät tietueet voidaan lajitella haluttuun järjestykseen. Rajapinta toimii samalla tavalla kuin *RecordFilter*-rajapinta eli enumeratoria luotaessa kaikki tietueet annetaan *RecordComparatorin* compare()-metodille. Tietueet lajitellaan metodin palauttaman arvon perusteella. Metodi muoto on seuraava [1, s. 220]:

public int compare(byte[] first, byte[] second)

Metodi saa parametrikseen kaksi tavutaulukkoa, joita verrataan keskenään. Metodin tulee palauttaa jokin kolmesta rajapinnassa määritellyistä arvosta [11, s. 93 - 94]:

- *RecordComparator*.EQUIVALENT, joka ilmoittaa arvojen olevan samanarvoisia.
- *RecordComparator*.PRECEDES, joka ilmoittaa, että ensimmäisen arvon tulisi olla ennen toista arvoa lajittelujärjestyksessä.
- *RecordComparator*.FOLLOWS, joka ilmoittaa, että ensimmäisen arvon tulisi olla toisen arvon jälkeen lajittelujärjestyksessä.

Sovelluskehittäjä voi itse määritellä perusteet, joilla tietueet lajitellaan, kirjoittamalla toteutuksen *RecordComparator*-rajapinnalle ja välittämällä sen instanssin enumerateRecords()-metodille. Sovelluskehittäjän tekemän *RecorComparatorin* toteutuksen tulisi palauttaa arvot seuraavien sääntöjen mukaan [1, s. 221]:

- Jos tietueet A ja B ovat samanarvoisia, tulee compare(A,B) ja compare(B,A) kutsujen kummankin palauttaa arvo *RecordComparator*.EQUIVALENT
- Lisäksi jos compare(A,B) ja compare(B,C) palauttavat arvon *RecordComparator*.EQUIVALENT, tulee myös metodi kutsun compare(A,C) palauttaa sama arvo
- Jos tietue A edeltää (on suurempi kuin) tietuetta B, tulee metodi kutsun compare(A,B) palauttaa arvo *RecordComparator*.PRECEDES ja compare(B,A) arvo *RecordComparator*.FOLLOWS

Metodille enumerateRecords() voidaan välittää *RecordFilter*-rajapintaa toteuttavan luokan instanssin sijasta arvon null, joka tarkoittaa, että tietueita ei suodateta ja kaikki tietueet tulevat mukaan palautettavaan enumerationiin. Sama pätee *RecordComparator*-rajapinnalle eli arvolla null tietuekannan tietoja ei lajitella. [11, s. 106.]

Kolmas enumerateRecords()-metodin saama parametri on boolean-arvo, joka ilmoittaa pidetäänkö enumerationin sisältämät tietueet ajan tasalla. Parametrin arvolla true luodaan dynaaminen enumeration, jonka arvot pidetään ajan tasalla vaikka tietuekannassa tapahtuisi muutoksia enumerationin läpikäynnin aikana. Arvolla false puolestaan luodaan staattinen enumeration, joka sisältää ne tiedot, jotka tietuekannassa on enumerationin luontihetkellä. Tällöin poistettaessa tietuekannasta enumerationissa mukana oleva tietue ja käsiteltäessä sitä enumerationilla, aiheutetaan *InvalidRecordIDException*. [1, s. 218 - 219.]

Neljäs rajapinta javax.microedition.rms-paketissa on *RecordListener*, joka määrittää kolme metodia [10, s.421]:

- public void recordAdded(*RecordStore* st, int recordID)
- public void recordChanged(*RecordStore* st, int recordID)
- public void recordDeleted(*RecordStore* st, int recordID)

RecordListener-rajapinta toimii kuten normaalit kuuntelijarajapinnat. Sovelluskehittäjä luo sitä toteuttavan luokan ja rekisteröi sen kuuntelemaan *RecodStorea* välittämällä sen *RecordStorelle* käyttämällä sen metodia [1, s. 215]:

public void addRecordListener(RecordListener listener)

Kun kuuntelija on rekisteröity kuuntelemaan tietuekantaa, kutsuu tietuekanta aina tilan muutoksen yhteydessä kuuntelijan oikeaa metodia. Esimerkiksi tietueen lisäyksen yhteydessä kutsuu tietuekanta kaikkien siihen rekisteröityneiden kuuntelijoiden recordAdded()-metodia ja välittää sille parametriksi itsensä ja lisätyn tietueen tunnisteen. Samalla tavalla tietuetta poistettaessa kutsutaan recordDeleted()- ja muutettaessa recordChanged()-metodia. [1, s. 215.]

4.4 Tietoliikenne

CLDC määrittelee yleisen tietoliikennekehyksen, mutta se ei sisällä ainuttakaan varsinasta toteutusta ja toteutukset on jätetty profiilien lisättäviksi. MIDP lisääkin CLDC:n javax.microedition.io-paketin määrittämään tietoliikennekehyksen rajapintahierarkiaan uuden *HTTPConnection*-rajapinnan, joka periytyy *ContentConnectionista*. Kuten rajapinnan nimestä voi jo päätellä, sitä toteuttavan luokan avulla on tarkoitus käsitellä HTTP-pohjaisia yhteyksiä. HTTP 1.1 on ainoa protokolla, jonka MIDP vaatii toteutettavaksi. Vaikka HTTP-protokolla yleensä toteutetaan IP-pohjaisilla yhteyksillä (esim. TCP/IP), se ei ole välttämätöntä ja se voidaan toteuttaa esimerkiksi käyttämällä WAP- tai i-mode-protokollia. MIDP jättääkin laitevalmistajan vastuulle toteuttaa HTTP-protokolla ja *HttpConnection*-rajapinnan luokka siten, että se on sovelluksille ja Internet-palvelimille täysin läpinäkyvä. Tällöin niiden ei tarvitse olla tietoisia millaista yhteyttä oikeasti käytetään. [11, s. 33 - 34.]

HttpConnection-rajapinta lisää *ContentConnection*-rajapintaan metodeita pyyntöotsikoiden asettamiseen, vastausotsikoiden jäsentämiseen (parserointiin) sekä muita HTTP:lle ominaisia toimintoja. Vaikka MIDP määrittelee HTTP:n ainoaksi pakolliseksi protokollaksi, ei se estä laitteistovalmistajaa

lisäämästä muitakin protokollia MIDP-toteutukseen, kunhan ne vain toteutetaan käyttämällä GCF:tä. Esimerkiksi socket-yhteydet ovat helposti lisättävissä ja kohtuullisen yleisiä laitteissa, jotka valmiiksi sisältävät TCP/IP-pinon (mm. älypuhelimet, kämmentietokoneet). [1, s. 181 - 185, 195.]

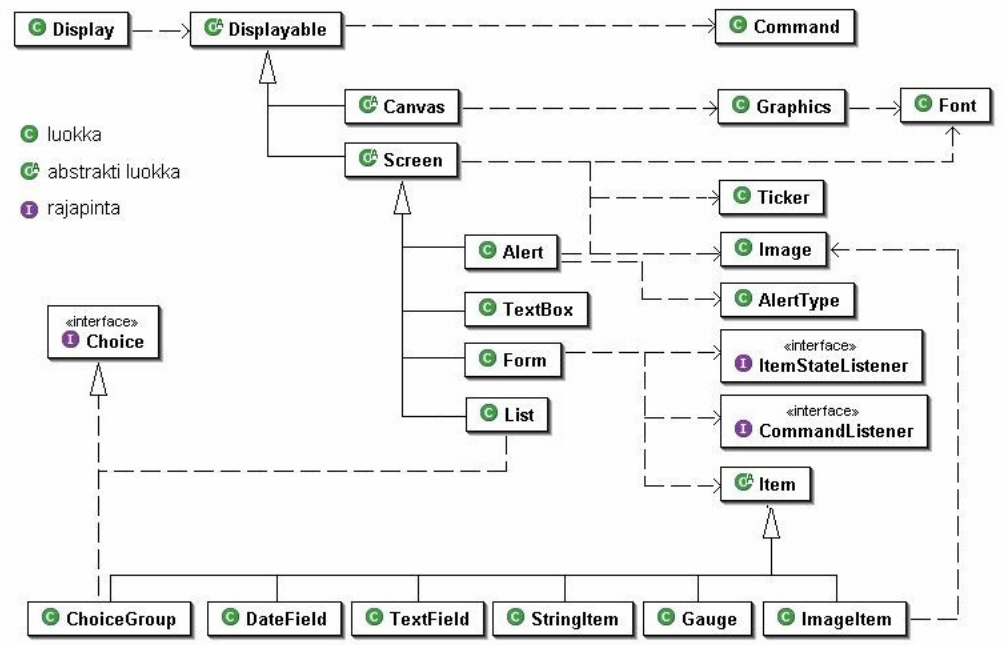
4.5 Käyttöliittymäkomponentit

MIDP:n suunnittelun yksi päätavoitteista on ollut pitäytyä Javan perusajatuksessa: alustariippumattomuudessa eli ohjelmien ja koodin siirrettävyydessä. MIDP:n toimintaympäristö poikkeaa monella tavalla J2SE- ja J2EE-ympäristöistä, mutta erityisesti vuorovaikutusmahdollisuudet käyttäjän kanssa (näyttö- ja syöttöresurssit) eroavat suuresti. MIDP:tä suunniteltaessa tämä on otettu huomioon ja *MIDletit* on suunniteltu siirrettäväksi laajalla laitealueella, jonka laitteiden syöttötavat ja näytön kapasiteetti vaihtelevat suuresti. [11, s. 49 - 50.]

Käyttöliittymäkomponenttien suunnitteleminen tällaiseen ympäristöön on erittäin vaativa tehtävä. MIDP:n käyttöliittymäkomponentteja suunniteltaessa yhtenä vaihtoehtona on ollut käyttää osajoukkoa J2SE:n Abstract Window Toolkit (AWT) tai Swing -komponenttikirjastoista. Nämä jouduttiin kuitenkin hylkäämään niiden monimutkaisuuden (muistin kulutus) sekä täysin erityyppisen käyttöympäristön takia. Molemmat, sekä AWT että Swing, on tarkoitettu antamaan sovelluskehittäjälle mahdollisimman vapaat kädet luoda ominaisuuksiltaan monipuolisia käyttöliittymiä moni-ikkunoituun ja osoitinlaitteella (hiiri, kosketusnäyttö, yms.) varustettuun ympäristöön. [1, s. 84 - 85; 11, s. 49 - 50.]

MIDletit eivät kuitenkaan voi olettaa näytölle mahtuvan kuin yhden ikkunan kerrallaan, eikä osoitinlaitettakaan ole kuin harvoissa erikoistapauksessa. Tästä syystä MIDP:n käyttöliittymä-API on suunniteltu näyttöpohjaisiksi (yksi kokoruudun lomake kerrallaan näytöllä) ja rajoitetulla näppäimistöllä käytettäväksi. Tämän ansiosta käyttöliittymäkomponentteihin ei tarvita mm. monimutkaista ikkunoinnin ja kohdistuksen hallintaa, jolloin kirjaston resurssivaatimuksen pienenevät merkittävästi. Lisäksi komponenteista tulee yksinkertaisempia ja helpompia käyttää sekä sovelluskehittäjälle että loppukäyttäjälle. [1, s. 84 - 85; 11, s. 49 - 50.]

MIDP:n käyttöliittymäkomponentit sijaitsevat javax.microedition.lcdui-paketissa. Kuva 5 esittää UML-luokkakaaviona MIDP 1.0:n käyttöliittymä-luokkien periytymis- sekä riippuvuussuhteet.



Kuva 5. MIDP 1.0 javax.microedition.lcdui-paketin UML-luokkakaavio [6, s. 78]

4.5.1 Display- ja Displayable-luokat

MIDP:n käyttöliittymä-API sijaitsee paketissa javax.microedition.lcdui ja se jakautuu kahteen tasoon sovelluskehittäjälle jaettavan näytölle piirtämisen ja tapahtumien käsittelyn perusteella. Komponentit jaetaan korkean- ja matalan tason komponentteihin. Vaikka korkean- ja matalan tason komponentit ovat luonteeltaan erilaisia, periytyvät ne molemman samasta abstraktista *Displayable*-luokasta. Yhteisen yläluokkansa ansiosta sekä korkean- että matalan tason komponentteja voidaan hallita yhdellä luokalla, joka on *Display*. Jokaiselle *MIDletille* luodaan yksi oma instanssinsa *Display*-luokasta. Instanssi on *MIDletin* käytössä sen startApp()-metodikutsusta aina destroyApp()-metodista palautumiseen asti ja se saadaan *MIDletin* käyttöön *Display*-luokan staattisella metodilla:

public static Display getDisplay(MIDlet midlet)

Metodi saa parametrikseen *MIDletin*, jonka *Display*-luokan ilmentymä palautetaan. *Display* voi näyttää vain yhtä *Displayable*-oliota kerrallaan laitteen näytöllä, joten asetettaessa näytölle uutta *Displayable*-oliota korvataan mahdollinen edeltävä *Displayable*-olio uudella. Uuden *Displayablen* asettaminen voidaan tehdä *Display*-luokan metodilla:

public void setCurrent(Displayable displayable)

MIDP:n määrityksissä ei kuitenkaan vaadita, että uusi näyttö olisi käyttäjälle näkyvissä heti metodikutsusta palattaessa ja normaalisti näytön vaihtaminen tapahtuu pienellä viiveellä. *MIDlet* voi päätellä näkyykö jokin *Displayable* laitteen näytöllä kutsumalla sen isShown()-metodia, joka palauttaa boolean-arvon tosi (true) mikäli se on piirrettynä laitteen näytölle. [11, s. 197 - 198.]

Display-luokka ei suoraan vastaa laitteen näyttöä vaan toimii ikään kuin virtuaalinäyttönä, jonka avulla *MIDlet* voi kontrolloida, mitä se haluaa näytettävän laitteen näytöllä. *MIDlettiä*, jonka *Display*-oliolla on kontrolli laitteen näytöstä, sanotaan olevan etualalla ja muiden *MIDlettien* olevan taka-alalla. Etualalla oleva *MIDlet* on aina aktiivisessa tilassa ja taka-alalla olevat Paused-tilassa. Vaikka *MIDlet* olisi taka-alalla, se voi suorittaa ja käyttää kaikkia *Display*-olion ominaisuuksia, mutta muutokset jotka tehdään tulevat näkyviin vasta *MIDletin* siirtyessä etualalle. [1, s. 85 - 88; 11, 197 - 198.]

4.5.2 Canvas-luokka

Matalan tason komponentteja käytettäessä sovelluskehittäjä luo alaluokan joka periytyy *Canvas*-luokasta. Tällöin sovelluskehittäjän täytyy itse huolehtia kaikesta näytölle piirtämisestä toteuttamalla *Canvas*-luokassa määritelty abstrakti-metodi:

protected abstract void paint(Graphics g)

Tämä on minimivaatimus käytettäessä matalan tason komponentteja, mutta yleensä sovelluskehittäjän tulee vielä käsitellä käyttäjän syötteet ylikirjoittamalla osa tai kaikki *Canvas*-luokan metodeista:

- protected void keyPressed(int keyCode)
- protected void keyReleased(int keyCode)
- protected void keyRepeated(int keyCode)
- protected void pointerDragged(int x, int y)
- protected void pointerPressed(int x, int y)
- protected void pointerReleased(int x, int y)

Matalan tason käyttöliittymäkomponenttien ei voida taata olevan siirrettäviä, koska niillä päästään käsiksi laitekohtaisiin ominaisuuksiin (esimerkiksi näppäimet). [6, s. 121 - 124, 138 - 139.]

Canvas-luokkaan liittyy läheisesti *Graphics*-luokka, jota käytetään yksinkertaisten geometristen kuvioiden piirtämiseen laitteen näytölle. Luokan avulla voidaan piirtää viivoja, suorakulmioita, kaaria, tekstiä ja kuvia. *Graphics*-luokalle voidaan kertoa, minkälaisella fontilla teksti tulisi kirjoittaa käyttämällä metodia:

public void setFont(Font font)

Metodi saa parametriksi *Font*-luokan ilmentymän. *Font* on luokka, joka sisältää tiedot fontin koosta, tyylistä ja ilmeestä. *Font*-luokasta ei luoda ilmentymiä itse, vaan niitä pyydetään käyttämällä *Font*-luokan metodia:

public static Font getFont(int face, int style, int size)

Metodi palauttaa mahdollisimman lähelle pyydettyjä arvoja vastaavan *Font*-luokan ilmentymän. *Font*-luokka tarjoaa palvelun, jolla voidaan selvittää merkkijonon tai yksittäisen merkin viemä pituus ja korkeus näytöllä. Näitä tietoja voidaan käyttää hyväksi esimerkiksi tekstin asettelussa näytölle. [6, s. 130, 137.]

4.5.3 Screen-luokka

Abstrakti *Screen*-luokka toimii korkean tason käyttöliittymäkomponenttien yliluokkana. *Screen*-luokan tehtävä on lisätä *Displayable*-luokkaan mahdollisuus näyttää otsikkorivi sekä ticker. *Ticker* on luokka, joka luo näytölle ”nauhan”, jonka sisällä kulkee vaakatasossa jatkuvasti vierivä teksti. Ticker-luokan vierittämän tekstin nopeus, kulkusunta sekä koko ovat laitekohtaisia eikä sovelluskehittäjä voi vaikuttaa niihin. [6, s. 84.]

Sovelluskehittäjä ei luo omia käyttöliittymäkomponentteja *Screen*-luokasta vaan käyttää valmiita komponentteja, jotka on peritty tästä luokasta. Korkean tason API ei anna sovelluskehittäjälle mahdollisuutta piirtää suoraan näytölle tai käsitellä käyttäjän syötteitä. Sen sijaan piirtäminen käsitellään sisäisesti ja syötteet muutetaan tarvittaessa korkean tason tapahtumiksi. Tämä antaa laitteistovalmistajalle mahdollisuuden toteuttaa komponenttien ulkonäön ja käyttäytymisen mahdollisimman samankaltaiseksi laitteen natiivin käyttöliittymän kanssa, joka puolestaan helpottaa sovellusten käyttämistä loppukäyttäjien kannalta. [1, s. 88 - 89.]

4.5.4 Command-luokka ja CommandListener-rajapinta

Komennot (*Command*-luokka) tarjoavat tavan toteuttaa vuorovaikutusta käyttäjän ja sovelluksen välillä. Komento voidaan mieltää normaalin tietokonesovelluksen nappiksi, jota painamalla suoritetaan siihen liittyvä toimenpide. Komennot näkyvät käyttäjälle pääsääntöisesti tekstinä laitteen näytön alareunassa, mutta komentojen ulkoasu ja sijainti näytöllä on laitteistokohtaista. Saman sovelluksen komennot voivat näyttää eri laitteissa erilaisilta. Esimerkiksi jokin laite näyttää komennon pelkkänä tekstinä ruudun alareunassa, kun taas toinen laite näyttää saman komennon graafisessa pudotusvalikossa näytön yläreunassa. Komento voidaan liittää *Displayable*-luokkasta perityvään olioon metodilla [10, s. 95 - 115]:

public void addCommand(Command command)

Poikkeuksena on *Alert*-luokka, jossa tämän metodin kutsuminen aiheuttaa *IllegalStateExceptionin*.

Komennot koostuvat kolmesta tiedosta, jonka perusteella laite voi sijoittaa ja näyttää ne parhaakseen katsomalla tavalla. Sijoittamisella tarjoitetaan sitä, että komento asetetaan aktivoitumaan jostakin teitystä laitteen napista ja näyttämisellä puolestaan miten ja missä kohtaa näyttöä komento esitetään. Tiedot joista komennot koostuvat ovat nimi, tyyppi ja prioriteetti. Nimi on merkkijono, joka kuvaa komennon aktivoimaa toimintoa. Nimi pyritään näyttämään sellaisenaan käyttäjälle. Tyyppi kertoo laitteelle, millaisesta komennosta on kyse. Kaikki MIDP 1.0:n määrittelemät *Command*-luokan tyypit ja lyhyet selitykset niistä on luoteltu taulukossa 6. Prioriteetti antaa laitteelle vinkin siitä kuinka tärkeä komento on. [6, s. 80 - 81.]

Taulukko 6. *Command*-luokan määrittelemät komentojen tyypit ja niiden lyhyet kuvaukset [6, s. 80 - 81]

Tyyppi	**Selite**
BACK	Pyyntö siirtyä edelliseen ruutuun.
CANCEL	Pyyntö keskeyttää toiminto.
EXIT	Pyyntö lopettaa *MIDletin* suoritus.
HELP	Pyyntö näyttää ohje tietoja.
ITEM	Pyyntö ohjata komento jollekin tietylle komponentille näytöllä. Esimerkiksi *List*-komponentti voi jäljitellä sisältöriippuvaista toiminnallisuutta ohjaamalla komennon tietyille listan elementeille.
OK	Määrittelee positiivisen myöntymyksen käyttäjältä.
SCREEN	Komennoille, joille ei ole suoraan kartoitettu näppäintä tai ne eivät liity mihinkään näytön elementteihin. Esimerkiksi komento ”Download” on tällainen.
STOP	Pyyntö keskeyttää toiminto.

Laitteet voivat käyttää komennon tyyppiä ja prioriteettia hyväkseen päättäessään mihin laitteen näppäimeen ja mihin kohtaa näyttöä komento sijoitettaan. Esimerkiksi, jos laitteessa on erillinen näppäin avustusten pyytämiseen (help-näppäin), voi laite sijoittaa kaikki *Command*.HELP-tyyppiset komennot aktivoitumaan tästä näppäimestä. Jos *Command*.HELP-tyyppisiä komentoja on useita samalla näytöllä, voi laite muodostaa nappia painettaessa valikon, jossa kaikki komennot on järjestetty prioriteetin mukaan. *MIDlet* ei siis anna sovelluskehittäjälle mahdollisuutta itse päättää missä ja miten komennot näytetään käyttäjälle, vaan jättää sen

laitevalmistajan päätettäväksi. Sovelluskehittäjä voi vain antaa vinkkejä laitteelle, miten komennot tulisi sijoittaa. [1, s. 96 - 102; 6, s. 80 - 82.]

Komennot eivät itsessään sisällä toiminnallisuutta. Toiminnallisuus määritellään *CommandListener*-rajapintaa toteuttavan luokan metodissa [6, s. 80.]:

public void commandAction(Command c, Displayable d)

Metodi saa parametrina tapahtuman aiheuttaneen komennon sekä näytön, jossa komento herätettiin (aktivoitiin). Oliota joka toteuttaa *CommandListener*-rajapinnan kutsutaan tapahtumankäsittelijäksi ja se voidaan asettaa kuuntelemaan *Displayable*-luokan tapahtumia käyttämällä luokassa määriteltyä metodia [6, s. 84.]:

public void setCommandListener(CommandListener listener)

4.6 Korkean tason käyttöliittymäkomponentit

4.6.1 Alert-komponentti

Alert on MIDP:n vastine J2SE:n dialogeille ja niiden avulla voidaan käyttäjälle näyttää varoituksia sekä virheilmoituksia. *Alert* voi toimia sekä ajastettuna että modaalisena. *Alertille* voidaan asettaa aika, jonka se näkyy käyttäjälle metodilla:

public void setTimeout(int time)

Parametri time ilmoittaa millisekunteina ajan, jonka *Alert* näkyy käyttäjälle. Modaalinen *Alert* puolestaan näkyy käyttäjälle, kunnes jotakin laiteen näppäintä painetaan. *Alertista* saadaan modaalinen antamalla setTimeout()-metodille parametriksi *Alert*-luokan FOREVER-jäsenmuuttuja. *Alert*-luokalle voidaan antaa teksti ja kuva, jotka näytetään käyttäjälle. *Alert*-luokkaan liittyy läheisesti *AlertType*-luokka. Luokan tärkein ominaisuus on sen sisältämät julkiset jäsenmuuttujat, joilla *Alert*-luokalle voidaan kertoa varoituksen tyyppi. Jäsenmuuttujat ovat INFO (tiedoksianto), WARNING (varoitus), ERROR (virhe), ALARM (hälytys) ja CONFIRMATION (varmistus).

Varoituksen tyyppi voidaan kertoa *Alert*-luokalle joko olion luonnin yhteydessä muodostimen parametrina tai käyttämällä *Alertin* metodia [6, s. 90 - 93.]:

public void setType(int type)

4.6.2 List-komponentti

List-luokalla voidaan toteuttaa erilaisia listoja. Luokka toteuttaa *Choice*-rajapinnan, joka määrittää listojen käsittelyissä tarvittavat metodit ja listatyypit. *Choice*-rajapinta määrittää kolme listatyyppiä, jotka on lueteltu taulukossa 7 lyhyen selityksen kanssa. [6, s. 87.]

Taulukko 7. *Choice*-rajapinnan määrittelemät listatyypit sekä niiden lyhyet selitykset [6, s. 118 - 119]

Tyyppi	Selite
IMPLICITE	Listaelementin valitseminen aiheuttaa välittömästi tapahtuman lomakkeella. Voidaan käyttää esimerkiksi valikoiden luontiin.
EXCLUSIVE	Listaelementin valitseminen ei aiheuta tapahtumaa vaan listalla olevan valitun elementin vaihtumisen. Listalla voi olla valittuna vain yksi elementti kerrallaan.
MULTIPLE	Samanlainen kuin EXCLUSIVE, mutta listasta voi olla valittuna useita elementtejä yhtä aikaa.

List-luokalla olevat elementit voivat koostua kuvasta sekä tekstistä tai pelkästä tekstistä. IMPLICITE ja EXCLUSIVE -tyyppisten listojen valittuna olevan elementin indeksin saadaan kysyttyä *Choice*-rajapinnassa määritellyllä metodilla:

public int getSelectedIndex()

Listaelementtien indeksointi aloitetaan nollasta, mutta MULTIPLE-tyyppisellä listalla metodi palauttaa aina -1. Tämän tyyppisen listan valitut elementit saadaan selvitettyä metodilla:

public void getSelectedFlags(boolean[] flags)

Metodi asettaa parametrina saamansa boolean-taulukon arvot vastaamaan listalla valittujen elementtien arvoja (valitut elementit saavat arvon tosi (true) ja ei valitut arvon epätosi (false)). [6, s. 88 - 89.]

4.6.3 TextBox-komponentti

TextBox on koko näytön käyttävä muokattava tekstikenttä ja sille annetaan muodostimen yhteydessä maksimipituus, jota enempää merkkejä siihen ei voida syöttää. Tekstiä voidaan vierittää, mikäli se ei mahdu kerralla näytölle sekä kentälle voidaan asettaa rajoite, joka rajoittaa syötettävän tiedon muotoa. Rajoitteet on määritelty *TextField*-luokassa sekä niiden käyttö ja merkitys ovat samat kuin kyseisessä luokassa (ks. taulukko 8). Rajoite annetaan olion muodostuksen yhteydessä parametrina tai käyttämällä *TextBox*-luokan metodia [6, s. 85.]:

public void setConstraints(int constraints)

4.6.4 Form-komponentti

Form on käyttöliittymäkomponenteista monipuolisin ja sen avulla luodaan erilaisia lomakkeita. Lomakkeet koostuvat käyttöliittymäkomponenteista eli ns. elementeistä, jotka kaikki periytyvät *Item*-luokasta (kuva 5). Elementit poistetaan ja lisätään lomakkeelle sovelluksen suorituksen aikana, joten käyttöliittymää voidaan muokata tarvittaessa esimerkiksi käyttäjän tekemien valintojen perusteella. Elementteihin viitataan niiden lomakejärjestyksen eli indeksin mukaan. *Form*-luokka sisältää metodit, joilla elementtejä lisätään, poistetaan ja korvataan lomakkeelta. [6, s. 96.]

Lomakkeella olevat komponentit aiheuttavat tapahtumia, joihin päästään käsiksi asettamalla lomakkeelle *ItemStateListener*-rajapintaa toteuttava kuuntelija. Idea on samanlainen kuin *CommandListener*-rajapinnan ja *Command*-luokan tapauksessa. Lomakkeella olevan elementin sisäisen tilan muutos voi aiheuttaa tapahtuman, josta lomake ilmoittaa tapahtuman kuuntelijalle kutsumalla sen *ItemStateListener*-rajapinnassa määriteltyä metodia:

public void itemStateChanged(Item item)

Metodi saa parametrina viittauksen *Item*-luokan olioon, jonka tila muuttui. MIDP:n määritys ei vaadi, että jokainen komponentin tilanmuutos aiheuttaisi itemStateChanced()-metodi kutsun. Se, mitkä tilanmuutokset aiheuttavat tapahtuman, jätetään laitevalmistajan vastuulle. Määrityksissä suositellaan,

että tapahtuma aiheutettaisiin viimeistään, kun kohdistus on siirtymässä komponentista toiseen ja vain jos komponentin tila on oikeasti muuttunut. [6, s. 120; 11, s. 247.]

Item on yksinkertainen abstrakti-luokka ja sovelluskehittäjälle sen ainoa näkyvä ominaisuus on nimiöteksti (otsikko) sekä sen kyselyyn ja asettamiseen vaadittavat metodit. Itemistä periytyvät luokat ovat *ChoiceGroup*, *TextField*, *DateField*, *Gauge*, *ImageItem* ja *StringItem*. Seuraavaksi kerrotaan lyhyesti jokaisen tärkeimmät ominaisuudet. [6, s. 99 - 100]

ChoiceGroup on lomakkeelle tarkoitettu versio *List*-komponentista. *ChoiceGroup* toteuttaa *Listin* tapaan *Choice*-rajapinnan, mutta erona *List* -komponenttiin on, että *ChoiceGroup* ei voi olla IMPLICITE-tyyppinen. IMPLICITE-tyyppinen *ChoiceGroup* olisi käytännössä pudotusvalikko, eikä sellaista ole katsottu tarpeelliseksi MIDP:hen. [6, s. 100.]

TextField-komponentti on samantapainen komponentti kuin *TextBox*. Oikeastaan ainoa ero on, että *TextField*-komponenttia voidaan käyttää lomakkeella. *TextField* määrittelee seitsemän erilaista rajoitetta, jotka ovat sen julkisia jäsenmuuttujia. Taulukossa 8 luetellaan ja kuvataan rajoitteiden kentille asettamat säännöt. [6, s. 103 - 106.]

Taulukko 8. *TextField*- ja *TextBox*-komponenteissa käytettävät rajoitukset sekä lyhyet kuvaukset niiden tarkoituksesta [6, s. 103 - 104]

Rajoite	Selite
ANY	Sisältö voi olla minkä muotoista tahansa.
CONSTRAINT_MASK	Ei ole varsinainen rajoite. Tämän ja getConstraints()-metodin avulla voidaan selvittää tekstikentän rajoitteet.
EMAILADDR	Kenttään voidaan syöttää sähköpostiosoite.
NUMERIC	Kenttään voidaan syöttää vain kokonaislukuja.
PASSWORD	Kenttään syötetyt merkit korvataan jollakin toisella merkillä (usein '*'-merkillä). Tämä rajoite voidaan yhdistää muiden rajoitteiden kanssa käyttämällä bittitason loogista OR-operaatiota.
PHONENUMBER	Kenttään voidaan syöttää puhelinnumero.
URL	Kenttään voidaan syöttää URL-osoite.

DateField mahdollistaa päivämäärän esittämisen ja muokkaamisen. Se sisältää päivämäärän oikeellisuuden tarkistuksen, jolloin sovelluskehittäjän ei tarvitse itse huolehtia siitä. *DateField* toimii kolmessa eri tilassa, jotka ovat DATE, DATE_TIME ja TIME. Tilat ilmoittavat minkälaista tietoa kentällä näytetään ja muokataan. DATE-tilassa ainoastaan päivämäärää voidaan käsitellä, DATE_TIME-tilassa molempia, sekä päivämäärää että aikaa, voidaan käsitellä ja TIME-tilassa voidaan käsitellä ainoastaan aikaa. Luettaessa *DateFieldin* arvoa ei sovelluskehittäjä saa suoraan kentän arvoa merkkijonona tai kokonaislukuina, vaan arvo palautetaan ainoastaan java.util.*Date*-luokan ilmentymänä. Kentälle voidaan asettaa *TimeZone*-luokan olio, joka kertoo kentälle käytettävän aikavyöhykkeen. [6, s. 107 - 109.]

Gauge-komponentin avulla voidaan kuvata graafisesti kokonaislukuarvoa nollan ja sovelluskehittäjän *Gaugelle* määrittelemän maksimiarvon välillä. Arvo esitetään usein joko pysty tai vaaka pylväskaaviona. *Gauge* toimii kahdessa eri tilassa: interaktiivisessa, jossa käyttäjä voi muokata sen arvoa tai ei-interaktiivisessa, jossa arvoa voidaan muokata vain ohjelmallisesti. Tärkeimmät metodit ovat esitettävän- ja maksimiarvon asettamiseen sekä lukemiseen tarkoitetut metodit. [6, s. 110.]

ImageItem-luokan avulla voidaan asettaa kuvia lomakkeelle. *ImageItem*-luokka sisältää kuusi sääntöä, joilla voidaan vaikuttaa kuvien asemointiin näytöllä (ks. Taulukko 9). Osaa säännöistä voidaan yhdistellä loogisen bittitason OR-operaation avulla. Näytettävän kuvan tulee olla muuttumattoman *Image*-luokan ilmentymä ja se asetetaan *ImageItemille* joko muodostimessa tai setImage(*Image* image)-metodilla. *Form*-luokka sisältää append(*Image* image)-metodin, jolla voidaan lisätä lomakkeelle kuva. Metodi ei lisää kuvaa suoraan lomakkeelle vaan luo siitä ensin ImageItemin ja lisää sen lomakkeelle. [1, s. 119 - 121.]

Taulukossa 9 luetellaan *ImageItemissä* käytettävät säännöt ja selitetään, miten niiden tulee vaikuttaa kuvan asemointiin laitteen näytöllä, mikäli laitteistovalmistaja on toteuttanut asemoinnin.

Taulukko 9. *ImageItem*-komponentin asemointiin käytettävät säännöt ja niiden selitykset [11]

Sääntö	Selite
LAYOUT_CENTER	Kuva tulisi sijoittaa horisontaalisesti keskelle näyttöä. Ei tarkoitettu yhdistettäväksi LAYOUT_LEFT tai LAYOUT_RIGHT:n kanssa.
LAYOUT_DEFAULT	Käytetään laitteen oletus layoutia. Kaikki muut säännöt ylikirjoittavat tämän säännön mikäli niitä yritetään yhdistää (arvo on 0).
LAYOUT_LEFT	Kuva tulisi sijoittaa horisontaalisesti näytön vasempaan reunaan. Ei tarkoitettu yhdistettäväksi LAYOUT_CENTER tai LAYOUT_RIGHT:n kanssa.
LAYOUT_NEWLINE_AFTER	Uusi rivi tulisi aloittaa kuvan piirtämisen jälkeen.
LAYOUT_NEWLINE_BEFORE	Uusi rivi tulisi aloittaa ennen kuvan piirtämistä.
LAYOUT_RIGHT	Kuva tulisi sijoittaa horisontaalisesti näytön oikeaan reunaan. Ei ole tarkoitettu yhdistettäväksi LAYOUT_CENTER tai LAYOUT_LEFT:n kanssa.

Image on luokka, joka kapseloi näytölle piirrettävän kuvadatan. *Image* voi olla muuttuva (mutable) tai muuttumaton (immutable). *Image*-luokalla ei ole julkista muodostinta, joten siitä ei voida luoda ilmentymiä new-operaattorilla. Sen sijaan käytetään jotakin luokan neljästä createImage()-metodista. *Image*-luokan ilmentymä voidaan siis luoda neljällä eri tavalla, jotka ovat:

- byte-taulukosta, joka sisältää kuvan datan
- toisesta *Image*-oliosta kopioimalla
- resurssitiedostosta, antamalla resurssitiedoston nimi
- antamalla kuvan leveys ja korkeus pikseleinä, jolloin luodaan uusi tyhjä kuva (leveys x korkeus pikselin kokoinen täysin valkoinen kuva).

StringItem kapseloi merkkijonon lomakkeelle asetettavaan muotoon. Käyttäjä ei voi muokata merkkijonon arvoa, mutta luokka sisältää metodit, joilla sovellus voi asettaa ja lukea merkkijonon arvon. *Form*-luokalla on append(*String* str)-metodi, joka toimii samalla tavalla kuin *Imagen* lisäävä

append(*Image*)-metodi eli luo *Stringistä StringItemin* ennen lomakkeelle sijoittamista. [1, s. 108 - 109.]

4.7 Muut lisäykset

MIDP lisää CLDC:ssä määriteltyihin järjestelmäominaisuuksiin (voidaan lukea java.lang.*System*.getProperty()-metodilla) kaksi uutta pakollista arvoa, jotka ovat:

- microedition.locale
- microedition.profiles

Ominaisuus microedition.locale sisältää kieli-maakoodi-parin, jonka tulee olla muotoa ”kielikoodi-MAAKOODI” (esimerkiksi ”fi-FI”). Toinen ominaisuus microedition.profiles ilmoittaa laitteen tukemat profiilit välilyönnillä eroteltuna ja sen arvon täytyy sisältää ainakin arvo ”MIDP-1.0”. [1, s. 49.]

MIDP lisää java.lang.util-pakettiin *Timer*- ja *TimerTask*-luokat, jotka ovat suoraan J2SE:n APIsta. Luokkien avulla voidaan siirtää tehtäviä suoritettavaksi myöhemmäksi tai toistuvin väliajoin. *Timer* on luokka, joka suorittaa jaksottaisesti (ajastetusti) yhtä tai useampaa tehtävää omissa taustasäikeissään. *TimerTask* on puolestaan tehtävä, jonka *Timer* ajastaa suoritettavaksi. Pakettiin java.lang on lisätty ainoastaan yksi luokka, joka on poikkeus *IllegalStateException*. Sitä käytetään laittomien tilasiirtymien ilmoituksessa eli silloin, kun sovellukset eivät ole sopivassa tilassa pyydetylle operaatiolle. [11, s.71, 75.]

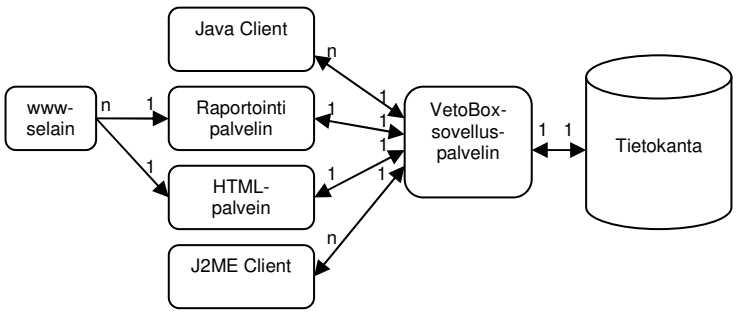
5 VETOBOX-SOVELLUSPALVELIN

Työssä tehty käyttöliittymäsovelluskehys on kiinteästi sidoksissa Vetokonsultit Oy:n toteuttamaan sovelluspalvelimeen. Tässä luvussa käydään läpi sovelluspalvelimen arkkitehtuuria, sen yhteydessä käytettäviä käsitteitä sekä normaalien asiakassovellusten rakennetta työn vaatimalta osalta. Luvussa on käytetty lähteenä Aapo Puhakan Diplomityötä [12], joka käsittelee sovelluspalvelimen suunnittelua ja toteutusta.

Sovelluspalvelin on toteutettu Java-kielellä ja siitä käytetään nimitystä VetoBox-sovelluspalvelin. Sovelluspalvelin on suunniteltu siten, että se tukee ns. Application Service Provider (ASP) -toimintaa. Yksinkertaistettuna ASP-toiminta tarkoittaa sovellusten vuokrausta asiakkaille siten, ettei asiakkaiden tarvitse omistaa palvelimia, vaan ne ovat sovellusta vuokraavan yrityksen hallinnassa. Tällöin asiakkaat siis käyttävät sovelluksia julkisen verkon yli ja maksavat käytöstä esimerkiksi kuukausipohjaisesti. Kustannusten minimoimiseksi ASP-mallia käytettäessä on käytännöllistä palvella useita asiakkaita samalla palvelimella ja tietokannalla. Tämä ei saa näkyä asiakkaille mitenkään, lisäksi asiakkaiden tulee voida luottaa siihen etteivät muut pääse käsiksi heidän tietoihinsa. Muun muassa näistä syistä turvallisuudelle ja luotettavuudelle on asetettava korkeat vaatimukset.

5.1 Arkkitehtuuri

VetoBox-sovelluspalvelin on toteutettu ns. kolmitasoarkkitehtuurin mukaisesti. Sovelluspalvelin sijaitsee asiakassovelluksen (Client) ja tietokannan välissä, josta syystä sitä kutsutaan myös sovelluspalvelin-kerrokseksi tai middlewareksi. Sovelluspalvelinkerros sisältää sovellusten toiminnallisen logiikan eli ns. business-logiikan ja sen tehtäviin kuuluvat asiakassovellusten synkronointi, yksikäsitteisten avainten jakaminen, lukitusten hallinnointi jne. Järjestelmään voidaan tarvittaessa lisätä muita sovelluspalvelukerroksen palveluita käyttäviä elementtejä, kuten esimerkiksi HTML- tai mobiilikäyttöliittymä. Kuva 6 esittää periaatekuvan VetoBox-sovelluspalvelimen suhteesta asiakassovelluksiin ja tietokantaan.



Kuva 6. VetoBox-sovelluspalvelimen kolmitasoarkkitehtuuri ja suhde muihin järjestelmän komponentteihin [12]

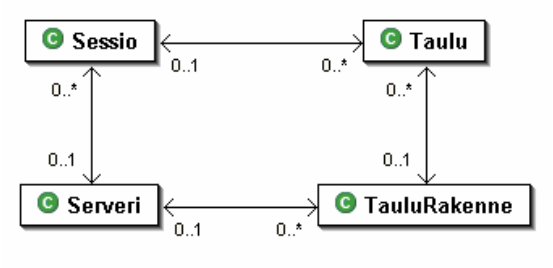
Normaalisti VetoBox-sovelluspalvelimissa sovellus- ja HTML-palvelin, yms. elementit sijaitsevat saman prosessin sisällä, jolloin vältytään tietoliikenneprotokollan aiheuttamalta lisärasitukselta. Usein kannattaa myös tietokanta asentaa samaan koneeseen sovelluspalvelimen kanssa, jolloin vältytään turhalta verkkoliikenteeltä. Isoissa järjestelmissä on mahdollista kohtuullisella työmäärällä toteuttaa järjestelmä, jossa eri palvelimet ovat omilla koneillaan.

5.2 Sovelluspalvelimen toteutus

Kuvassa 7 esitetään VetoBox-sovelluspalvelimen neljä keskeistä luokkaa sekä niiden väliset suhteet. *Serveri*-luokka toimii keskusluokkana, jonka avulla päästään käsiksi muihin luokkiin. Jokaista asiakassovellusta varten luodaan oma *Sessio*-olio, joka kuvaa palveltavaa sovellusta. Molemmista luokista periytetään järjestelmäkohtainen versio. Järjestelmässä käytettävien tietokantojen taulut kuvataan *Taulu*- ja *TauluRakenne*-luokkien avulla. Jokaisesta taulusta luodaan sekä *Taulu*- että *TauluRakenne*-luokka, joista *Taulu*-luokka kuvaa tietokannan taulun käyttöä tietyssä tarkoituksessa ja *TauluRakenne*-luokka kuvaa käytettävän taulun rakenteen. *Taulu*-luokasta on useita instansseja, kun taas *TauluRakenne*-luokasta yleensä vain yksi instanssi järjestelmää kohden.

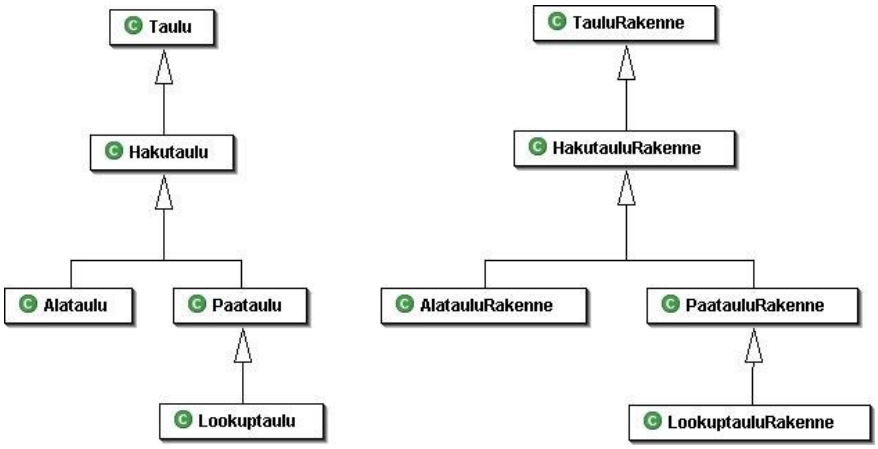
MIDP-käyttöliittymän kannalta oleellisimmat luokat ovat *Taulu* ja *TauluRakenne*. *TauluRakenne*-luokka koostuu kentistä, jotka ovat myös olioita ja sisältävät tietoja kuten ohjelmoijan kentästä käyttämä nimi, kentän tietokantanimi, kentän tyyppi, yms. tietoja. Kentän oliot siis muodostavat ns. metadatan taulun rakenteesta. *TauluRakenne*-luokka sisältää myös taulujen väliset yhteydet toisiinsa. *Taulu*-luokan instanssilla voi olla toisia *Taulu*-luokan olioita komponentteinaan määriteltyjen yhteyksien mukaisesti.

Kuvassa 7 esitetään UML-luokkakaaviona *Taulu*- ja *TauluRakenne*-luokkien väliset yhteydet.



Kuva 7. VetoBox-sovelluspalvelimen neljä keskeistä luokkaa *Taulu*, *TauluRakenne*, *Sessio* ja *Serveri* [12]

Kuvassa 8 puolestaan esitetään *Taulu*- ja *TauluRakenne*-luokkien periytymishierarkia UML-luokkakaaviona. Alimpana hierarkiassa olevat *Paataulu*-, *Lookuptaulu*- ja *Alataulu*-luokat vastaavat tauluista käyttölliittymissä käytettäviä rooleja. *Paataulu*-luokalle voidaan käyttöliittymässä tehdä suoraan haku-, lisäys- sekä tallennusoperaatioita ja se voi sisältää ala- ja lookup-tauluja. *Alataulu* on taulu, jonka operaatiot liittyvät aina päätauluun. Sitä käytetään tilanteissa, joissa halutaan näyttää monta tietuetta, jotka kaikki liittyvät päätaulun yhteen tietueeseen. Esimerkiksi tilanteessa, jossa päätaulun tietue sisältää henkilön tiedot ja henkilöllä voi olla monta puhelinnumeroa, puhelinnumerosta tehdään alataulu, joka liittyy henkilö-päätauluun.



Kuva 8. VetoBox-sovelluspalvelimen *Taulu*- ja *TauluRakenne*-luokkahierarkiat [12]

Lookuptaulua käytetään, kun käyttöliittymässä on kenttä, jonka arvo voi olla vain jokin ennalta määritelty. Esimerkkinä tämäntyyppisestä kentästä voisi olla henkilön osoitteen postinumero. Tällainen kenttä voidaan määritellä lookup-kentäksi, jolloin sen arvo tarkistetaan siihen liittyvästä *Lookuptaulusta*. Lisäksi lookup-kentälle voi määritellä kenttiä, joiden arvon se asettaa automaattisesti vastaamaan lookup-kentän arvoa. Edellisessä esimerkissä tällainen kenttä voisi olla kunnan nimi, joka liittyy postinumeroon. Arvojen tarkistuksen ja asettamisen lisäksi *Lookuptaululta* voidaan hakea lista kaikista sallituista arvoista, jotka lookup-kenttään voidaan sijoittaa.

5.3 Tietoliikenne

VetoBox-sovelluspalvelimen ja asiakassovellusten välistä kommunikaatiota varten on luotu oma protokolla, jota kutsutaan VetoRPCksi. Vaihtoehtoina olisivat olleet mm. Java RMI, CORBA ja SOAP. Oman protokollan toteuttamiseen vaikuttavia tekijöitä olivat:

- RMI:hin salauksen toteuttaminen on mahdollista vain tietyillä JDK:n versioilla ja kommunikaatio haluttiin versiosta riippumattomaksi.
- RMI sitoo sovellukset Javaan, eikä toisten ohjelmointikielten käyttöä sovelluspalvelinkerroksen palveluja käyttävänä osapuolena voida toteuttaa.
- CORBA on turhan raskas ja monimutkainen.
- SOAP oli projektin alkaessa uudehko protokolla ja vielä ”lapsen kengissä”.

VetoRPC toimii suoraan TCP/IP:n päällä ja on HTTP:n tapaan yhteydetön eli jokaista palvelupyyntöä varten avataan uusi yhteys. Kuten HTTP:ssä myös VetoRPCssä asiakassovellus on aktiivinen osapuoli, joka avaa yhteydet palvelimen vain kuunnellessa palvelupyyntöjä. Menettelyn haittapuolena on, ettei sovelluspalvelin pysty lähettämään oma-aloitteisesti viestejä asiakassovelluksille. Ongelma on ratkaistu siten, että asiakassovellus avaa käynnistyksen yhteydessä oman TCP/IP-yhteyden, joka on tarkoitettu palvelimelta asiakkaalle lähetettäville viesteille.

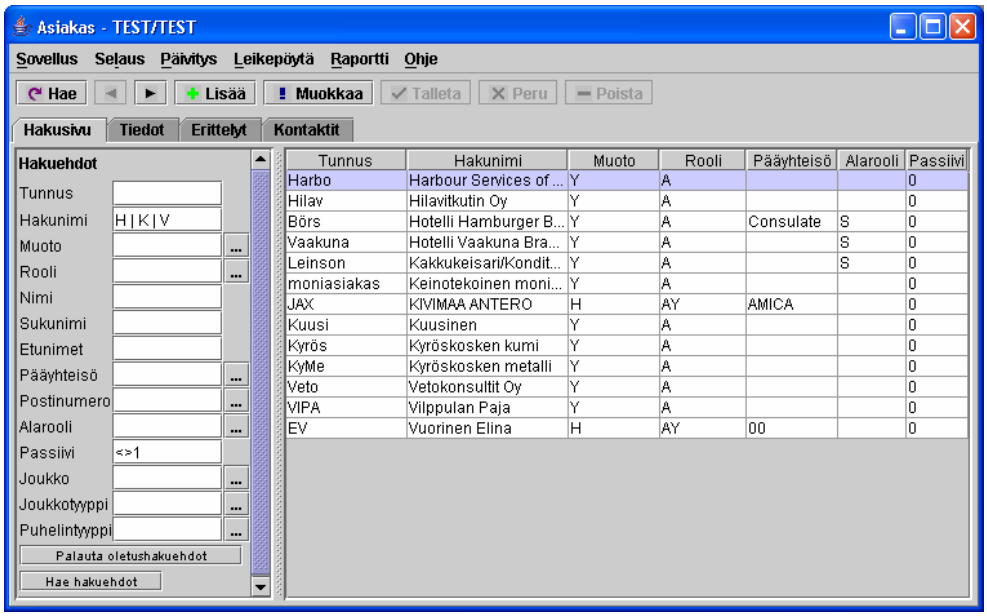
VetoBox-sovelluspalvelin ja asiakassovellukset käyttävät instanssi-viittausten, metodien ja parametrien muuttamiseen VetoRPC-muotoon yhteisiä luokkia. Samoin salauksen toteuttamiseen käytetään yhteisiä luokkia. Salaus on samantapainen kuin SSH-protokollan julkisen avaimen (RSA) ja symmetrisen salauksen (Blowfish) yhdistelmä.

5.4 VetoBox-sovelluspalvelimen yleinen käyttöliittymämalli

VetoBox-sovelluspalvelimelle tehtävät käyttöliittymät sisältävät usein paljon yhteisiä piirteitä. Käyttöliittymät, joita käytetään, ovat pääasiassa Java Swing-kirjastoon pohjautuvia tai HTML-käyttöliittymiä. Tässä kappaleessa esitellään Swing-kirjastolla tehdyn käyttöliittymän ulkoasua ja toimintatapaa yleisellä tasolla. Muut käyttöliittymät on pyritty tekemään toiminnaltaan ja ulkoasultaan samankaltaisiksi, mikäli tämä vain on teknisesti ja käytettävyydeltään järkevää.

Eri järjestelmiin tehtävät käyttöliittymät muistuttavat toisiaan, koska ne generoidaan puoliautomaattisesti palvelimelta saatavien *TauluRakenne*-olioiden rakennetietojen perusteella. Tämä yhdenmukaistaa käyttöliittymiä, mikä parantaa järjestelmän käytettävyyttä, koska kun on oppinut yhden sovelluksen, on uuden oppiminen luontevaa samankaltaisen käyttäytymis-mallin ja ulkoasun johdosta. Yleisen käyttöliittymämallin ikkuna koostuu menusta, nappirivistä ja välilehdistä (tabsivuista).

Kuvassa 9 on esimerkkinä käyttöliittymän ikkuna Prosla-järjestelmän (projektien hallinta-, seuranta-, budjetointi ja laskutusjärjestelmä) Asiakas-sovelluksen käyttöliittymästä.

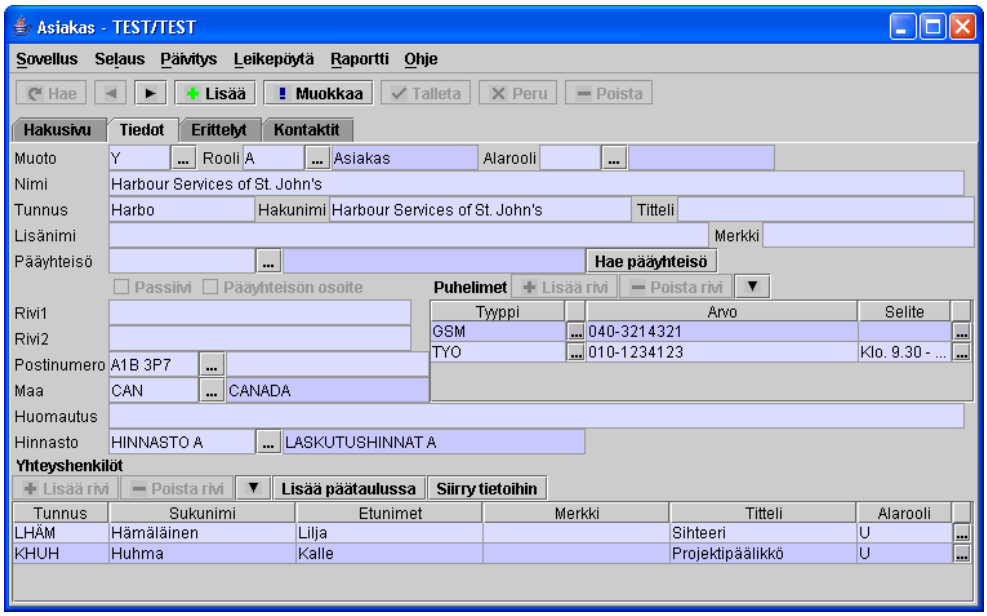


Kuva 9. Prosla-järjestelmän Asiakas-sovelluksen käyttöliittymä Hakusivu-tilassa

Kuvassa 9 on valittuna Hakusivu-välilehti, jolla käyttäjä suorittaa hakuja. Hakusivu on sovelluksissa aina ensimmäinen välilehti ja se generoidaan lähes poikkeuksetta automaattisesti sovelluspalvelimelta saatavan *TauluRakenne*-olion tietojen perusteella. Ikkunan vasempaan laitaan generoidaan hakukentät, joihin käyttäjä syöttää hakuehdot, jotka hakua suoritettaessa lähetetään sovelluspalvelimelle. Palvelimen tehtävä on muuttaa ehdot sopivaksi SQL-lauseen WHERE-osaksi. Kun haku on suoritettu, tulevat haun tulokset ikkunassa oikealla olevaan taulukkoon, joka myös on generoitu *TauluRakenne*-olion tiedoista.

Hakusivu-välilehden lisäksi käyttöliittymässä on yksi tai useampi välilehti, joita kutsutaan tietosivuiksi. Tietosivu luodaan joko täysin automaattisesti tai komponenttien sijoittelu voidaan määrätä itse. Se kumpaa tapaa käytetään, valitaan luokalla, josta näytettävä tietosivu periytetään. Tietosivun tiedot haetaan sovelluspalvelimelta vasta siirryttäessä Hakusivulta tietosivulle. Tällä tavoin voidaan Hakusivun hakukentistä generoitava SQL-hakulause, joka kohdistuu useisiin tietokannan riveihin, pitää mahdollisimman yksinkertaisena ja nopeana. Tietosivulle siirryttäessä on käyttäjä valinnut

hakutuloksista jonkin tietyn rivin käsittelyyn, jolloin tietokantahaku kohdistuu enää yhteen riviin. Tällöin haku voi sisältää huomattavasti monimutkaisempia taulujen välisiä liitoksia hidastamatta sovelluksen toimintaa kuitenkaan merkittävästi. Kuvassa 10 on Prosla-järjestelmän Asiakas-sovelluksen Tiedot-tietosivu.

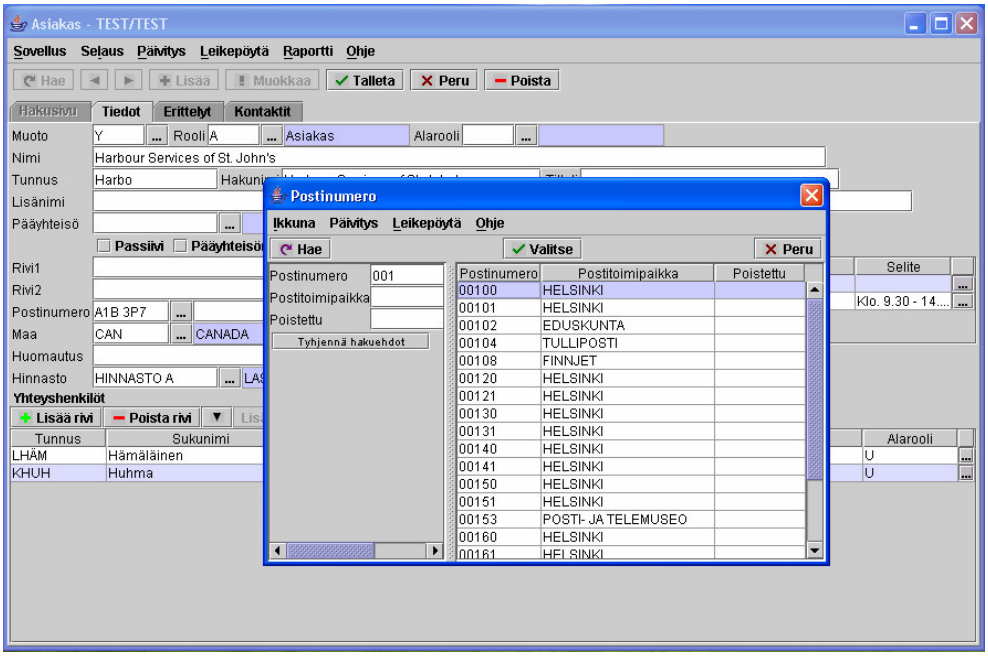


Kuva 10. Prosla-järjestelmän Asiakas-sovelluksen käyttöliittymän Tietosivu

Kuvassa 10 ikkunan alaosassa Yhteyshenkilöt-taulukon sekä oikealla keskellä oleva Puhelimet-taulukon tiedot ovat peräisin *Alataulusta*. *Alataulujen* tiedot näytetään käyttöliittymissä normaalisti taulukkoina.

Kuvan 11 aktiivinen ikkuna esittää ns. lookup-ikkunan, joka saadaan esille painamalla lomakkeella näkyvistä '…'-napeista. Lookup-ikkunasta valittu arvo sijoitetaan '...'-napin viereiseen kenttään. Kuvan 11 tapauksessa on kyseessä lookup-ikkuna, jossa voidaan suorittaa hakuja samaan tapaan kuin Hakusivu-välilehdellä. Jos mahdollisia arvoja on vain muutama, ei hakukenttiä näytetä lookup-ikkunan vasemmassa reunassa ja kaikki mahdolliset arvot ovat haettuina suoraan avautuvaan lookup-ikkunaan.

Kuva 11 esittää VetoBox-sovelluspalvelimen Swing-käyttöliittymäsovelluksissa käytettävää lookup-ikkunaa.



Kuva 11. Postinumeron ja -toimipaikan haku lookup-ikkunan avulla Prosla-järjestelmän Asiakas-sovelluksessa

6 MOBIILI KÄYTTÖLIITTYMÄSOVELLUSKEHYS

6.1 Työn kulku

Työn aiheesta keskusteltiin tammikuun 2004 aikana Vetokonsultit Oy:n kanssa ja aihe valittiin tammikuun puolessavälissä, jonka jälkeen työn toteuttaminen aloitettiin. Työ lähti liikkeelle perehtymällä uuteen J2ME-kehitysympäristöön sekä tutkimalla sen tarjoamia mahdollisuuksia. Perehtymisvaiheen aikana testattiin J2ME:n MIDP-profiilin ominaisuuksia toteuttamalla erinäisiä testisovelluksia. Testisovelluksissa lähdettiin liikkeelle aivan yksinkertaisesta palvelimen ”pingaus”-sovelluksesta. Sovellusten vaativuutta kasvatettiin vähitellen ja päämääränä oli luoda sovellus, joka käyttää VetoBox-sovelluspalvelimen VetoRPC-protokollaa kommunikoinnissa palvelimen kanssa.

Tämä välietappi saavutettiin maaliskuun puolivälissä, jolloin aloitettiin käyttöliittymäsovelluskehyksen arkkitehtuurin suunnittelu. Kun arkkitehtuurin

pääpiirteet oli päätetty, aloitettiin sovelluskehyksen toteuttaminen. Sovelluskehyksen yksityiskohtien suunnittelu ja toteutus tapahtuivat rinnakkain toisiaan tukien. Kehyksen ensimmäiset käyttökelpoiset testi-versiot valmistuivat huhtikuun lopussa, jonka jälkeen käyttöliittymä-sovelluskehystä on testattu sekä paranneltu muutaman eri projektin yhteydessä. Sovelluskehyksen kehitys jatkuu edelleen muiden projektien ohella.

6.2 Kehitysvälineet

Ohjelmointityössä käytettiin Sun Microsystemsin J2ME Wireless Toolkit -työkalua projektin kääntämiseen, testaamiseen ja jakelupakettien tuottamiseen. Wireless Toolkit on yksinkertaisella käyttöliittymällä varustettu J2ME-projektien hallintatyökalu, jonka Sun Microsystems tarjoaa kaikkien käyttöön ilmaiseksi. Sillä voidaan kääntää ja luoda projekteista sovellusten jakeluun vaadittavat JAD- ja JAR-tiedostot. Se sisältää valmiiksi muutaman J2ME MIDP-emulaattorin, jolloin sovellusten testaus ilman oikeaa laitetta on mahdollista. Emulaattoreissa on sovellusten virheiden etsintää ja optimointia helpottavia ominaisuuksia kuten tietoliikenteen ja muistin käytön seuranta, tietoliikennenopeuden rajoittaminen, Java virtuaalikoneen käskyjen suoritus nopeuden säätö, yms. Näitä ominaisuuksia säätelemällä testataan, miten sovellukset käyttäytyvät eritasoisilla laitteilla. Itse ohjelmakoodi kirjoitettiin käyttäen eri tekstieditoreita, kuitenkin pääosa koodista kirjoitettiin emacsilla.

Testaus- ja kehitysympäristöksi luotiin olemassa olevan Prosla-järjestelmän kehityspalvelimeen uusi Asiakas-sovellus, joka on tarkoitettu erityisesti mobiilikäyttöliittymää varten. Prosla-järjestelmässä on olemassa myös normaalille käyttöliittymälle tarkoitettu Asiakas-sovellus, mutta tässä tapauksessa katsottiin parhaaksi luoda oma versio mobiilikäyttöliittymää varten, jotta näytettävän ja siirrettävän tiedon määrää saatiin pienennettyä. Tällä ns. mobiililla Asiakas-sovelluksella haetaan, muokataan ja lisätään asiakastietoja matkapuhelinten avulla.

6.3 Arkkitehtuuri

Eräs määritelmä oliopohjaiselle sovelluskehykselle on annettu Erich Gamman ja kumppaneiden (GOF) tunnetussa teoksessa Design Patterns: Elements of Reusable Object-Oriented Software [13]:

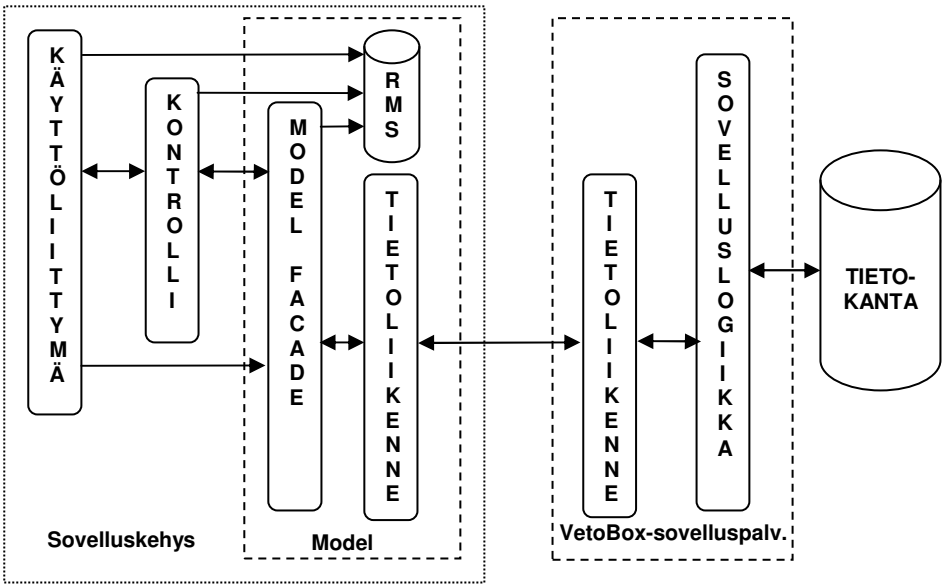
> *Sovelluskehys on kokoelma yhteistyötä tekeviä luokkia, jotka muodostavat uudelleen käytettävän mallin tietylle ohjelmistoalueelle. Sovelluskehys luo arkkitehtuaalisen ohjeistuksen jakamalla mallin abstrakteiksi luokiksi ja määrittelemällä niiden vastuut ja yhteistyösuhteet. Sovelluskehittäjä yksilöi sovelluskehyksen sovelluskohtaiseksi periyttämällä ja koostamalla instansseja sovelluskehyksen luokista.*

Sovelluskehys siis määrittää sovellusten arkkitehtuurin, luokkien yhteistyön ja kontrollin kulun. Sovelluskehys kokoaa ne suunnittelumallit ja sovellusten piirteet yhteen, jotka ovat yhteisiä kyseisen sovelluskehyksen sovellusalueella. Se korostaa mallin uudelleen käyttöä koodin uudelleenkäytön asemasta, vaikka sovelluskehykset usein sisältävätkin konkreettisia luokkia, joita sovelluskehittäjät voivat suoraan käyttää sovelluksia luodessaan. [14.]

Työssä on toteutettu näiden määritysten mukainen mobiililaitteilla toimiva käyttöliittymäsovelluskehys VetoBox-sovelluspalvelimen sovelluksille. Sovelluskehystä toteutettaessa lähdettiin liikkeelle perusarkkitehtuurin valitsemisella. Arkkitehtuurin pohjaksi valittiin Model-View-Controller (MVC) -suunnittelumalli, koska se soveltui luontevasti tarkoitukseen. Mallissa erotellaan käyttöliittymä (View) ja sovelluslogiikka (Model) toisistaan lisäämällä niiden väliin kontrollikerros (Controller). Kontrollikerroksen tehtävä on kääntää käyttöliittymän tapahtumat sovelluslogiikkakerroksen tapahtumiksi. Kontrollikerros päättää myös, mitä käyttöliittymäkerros näyttää käyttäjälle valitsemalla sopivan käyttöliittymäelementin sovelluslogiikkakerroksen tilan ja käyttäjän toimien perusteella. Kontrollikerros luo siis sovelluksen toiminnallisuuden. [15.]

Vaikka työssä suunniteltiin käyttöliittymäsovelluskehystä, jakaantui se varsin luontevasti MVC-mallin mukaisiin osiin. MVC-mallin View-taso luonnollisesti muodostettiin käyttöliittymäkomponenteista ja Controller-taso puolestaan luokasta, joka käsittelee kaikki käyttöliittymäkomponenttien tapahtumat sekä ohjaa sovelluksen kulkua. Koska suurin osa sovelluslogiikasta sijaitsee VetoBox-sovelluspalvelimella, kehyksen Model-tason tehtäväksi jäi

pääasiassa tietoliikenteen hallitseminen sekä tietojen puskurointi ja mahdollinen tallentaminen (tai lukeminen) laitteiston pysyvään muistiin. Kuvassa 12 esitetään periaatekuva sovelluskehyksen eri kerroksista sekä niiden suhteesta VetoBox-sovelluspalvelimeen.



Kuva 12. Käyttöliittymäsovelluskehyksen arkkitehtuuri ja suhde VetoBox-sovelluspalvelimeen

Sovelluskehykset voidaan jakaa karkeasti kahteen ryhmään, ns. White Box ja Black Box -tyyppisiin sovelluskehyksiin. Nämä edustavat sovelluskehysten ääripäitä ja oikeat sovelluskehykset ovatkin jotain näiden kahden väliltä (Grey Box). Sovelluskehysten kehitysprosessi on vahvasti iteratiivinen ja yleensä ensimmäiset versiot kuuluvat lähemmäksi White Box -tyyppistä kuin Black Box -tyyppistä sovelluskehystä, jota kohti jokaisella iteraatio-kierroksella pyritään. [16.]

White Box -tyyppinen sovelluskehys olettaa, että sovelluskehyksen käyttäjällä on joko pääsy kehyksen koodiin tai käytetään periyttämistä uusien sovelluksien luomiseen sovelluskehyksen abstrakteista luokista. Tällainen oletus vaatii, että sovelluskehyksen käyttäjä tuntee kehyksen toteutuksen hyvin, mikä kasvattaa dokumentoinnin tärkeyttä. White Box -sovelluskehystä käytettäessä luotetaan siis periytymiseen tai staattiseen koostamiseen sekä mahdollisuuteen ylikirjoittaa yleiset ominaisuudet sovelluskohtaisilla ominaisuuksilla. [16.]

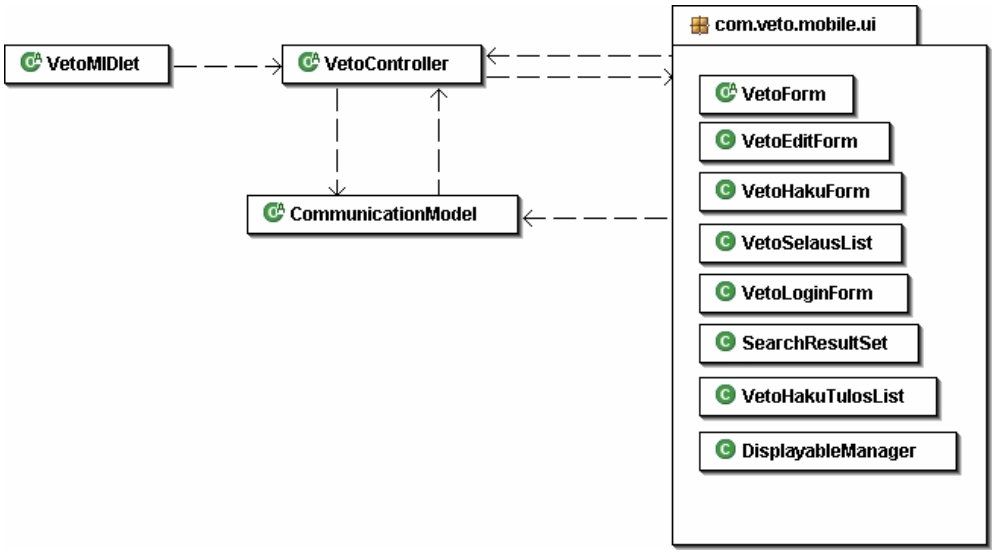
Käytettäessä periytymistä törmätään sovelluskehysten kannalta ikävään ongelmaan nimittäin tiukkoihin sidoksiin. Periyttämisessä periytyvät luokat (aliluokat) ovat aina tiukasti sidottuja yliluokkaan, josta ne periytyvät. Tiukat sidokset ovat yksi rajoittavimmista tekijöistä pyrittäessä joustaviin ja laajennettaviin järjestelmiin, mikä on eräs sovelluskehysten päätavoitteista. [16.]

Black Box -sovelluskehykset käyttävätkin periyttämisen sijasta polymorfista koostamista. Siinä sovelluskehyksessä olevat kohdat, jotka vaihtelevat sovelluskohtaisesti (ns. hot spotit) on tunnistettu ja niissä kohtaa sovelluskehys käyttää rajapintoja, jotka toteuttamalla sovelluskehyksen käyttäjä ”konfiguroi” sovelluskehyksestä uuden sovelluksen. Tällöin itse sovelluskehyksestä tulee staattinen, eikä periyttämistä tarvitse käyttää. Tällaisiin Black Box -tyyppisiin sovelluskehyksiin päästään yleensä vasta monien kehyksen toteutus- ja suunnitteluiteraatioiden jälkeen. [16.]

Työssä toteutettu sovelluskehys ei poikkea normaalista sovelluskehyksen kehityksestä ja se perustuukin pitkälti periyttämiseen. Periytyminen J2ME-ympäristössä, missä resursseista on pulaa, on kuitenkin ymmärrettävämpää kuin normaalissa Java-ympäristössä. Käyttämällä abstrakteja luokkia ja periyttämällä niistä sovelluskohtaisia versioita, saadaan luokat hyöty-käyttöön. Abstraktit luokat itsessään sisältävät toiminnallisuutta, toisin kuin rajapinnat, jotka vain määrittelevät toiminnallisuuden. MIDP-ympäristössä jopa pelkällä luokkien määrällä on merkitystä, koska esimerkiksi jo muutaman metodin sisältämä rajapinta saattaa kasvattaa MIDlet-pakkauksen tarvitseman tilan määrää noin 300 - 1000 tavua. Tästä syystä rajapintoja tulisi käyttää harkiten. Rajapintojen ja yhteen asiaan keskittyvien luokkien välttelystä seuraa usein, etteivät MIDP-sovellukset ole niin oliosuuntautuneesti ohjelmoituja kuin J2SE- tai J2EE-sovellukset. Tämän opinnäytetyön kannalta tällä on erityistä merkitystä, sillä kohdelaitteisto, jolla sovelluskehystä on tarkoitettu käytettäväksi, koostuu pääasiassa matka-puhelimista, joiden malleista suurimmalla osalla on vielä 64 kt:n jakelu-tiedoston maksimikoon rajoitus. [17, s. 11 - 12.]

Kuten aikaisemmin jo mainittiin, perustuu sovelluskehyksen arkkitehtuuri MVC-malliin. Mallin Controller-taso on kehyksessä toteutettu abstraktilla *VetoController*-luokalla ja Model-taso abstraktilla *CommunicationModel-*

luokalla. View-taso koostuu monista käyttöliittymäkomponenteista ja niitä avustavista luokista. Käyttöliittymäsovelluskehyksen MVC-mallia toteuttavat luokat esitellään kuvassa 13, jossa ei ole esitetty kaikkia com.veto.mobile.ui-paketin luokkia piirustusteknisistä syistä. Siitä puuttuvat muutamat käyttöliittymäkomponenttien käyttämät apuluokat.



Kuva 13. Käyttöliittymäsovelluskehyksen MVC-mallia toteuttavat luokat UML-luokkakaaviona

Työssä toteutetut luokat on jaettu kolmeen pakettiin. Paketti com.veto.mobile sisältää *VetoControllerin* ja muita sovelluskehyksen yleisesti käyttämiä luokkia. Paketin UML-luokkakaavio, jossa esitetään paketin luokkien periytymis- ja riippuvuussuhteet, esitetään liitteessä 1. Paketti com.veto.mobile.ui sisältää kaikki käyttöliittymäkomponentteihin liittyvät luokat ja sen UML-luokkakaavio on esitetty liitteessä 2. Sovelluskehyksen Modelin muodostava *CommunicationModel*-luokka sijaitsee paketissa com.veto.mobile.communication. *CommunicationModelin* UML-luokkakaavio on liitteessä 3.

6.4 Model-taso

Model-tason tehtävä on kapseloida sisäänsä sovelluksen tila sisältämällä sovelluksen käyttämä data ja sen käsittelyyn tarvittava toiminnallisuus. Työssä tehdyn sovelluskehyksen Model-kerroksen toiminnallisuus on pääasiassa tietoliikenteen kontrollointia, koska varsinainen sovelluslogiikka sijaitsee VetoBox-sovelluspalvelimella. Sovelluskehyksessä Model-kerros on koottu yhden luokan taakse, jota kehyksen muut osat käyttävät. Tämä luokka on *CommunicationModel* ja se toimii J2EE:ssä käytettävän Business Delegate -suunnittelumallin [18] tapaan. *CommunicationModel* (liite 3) piilottaa taakseen sovelluspalvelimen palveluiden ja tietoliikenteen toteutuksen. Tämä vähentää View- ja Controller-kerrosten kytköstä Modelia toteuttaviin luokkiin, jolloin mahdollisuus View- ja Controller-tasolla tarvittaviin muutoksiin vähenee vaikka Model-luokkiin tulisikin muutoksia. Toinen merkittävä etu varsinkin J2ME-sovellusten kannalta, joissa tietoliikenne on hidasta, on mahdollisuus tietojen varastoimiseen, joka on yksi *CommunicationModelin* tärkeimmistä tehtävistä.

Jotta sovellusten ylläpito olisi helpompaa, tulee Model-kerroksen käyttää samoja luokkia tietoliikenne- ja sovelluspalveluiden kutsuihin, kuin muutkin VetoBox-sovelluspalvelinta käyttävät käyttöliittymäsovellukset. Näiden luokkien käyttö ei kuitenkaan aivan suoraan ollut mahdollista kolmesta syystä:

- Luokat käyttivät sellaisia Java-kielen ominaisuuksia, joita ei ole toteutettu MIDP:ssä. Tällaisia ovat esimerkiksi liukuluvut ja sarjallistuminen.
- Luokkakirjasto vei itsessään liikaa tilaa ja näin ollen rajoitti käytettävien laitteiden joukkoa.
- Luokkakirjasto käytti yhteyden muodostamiseen socket-yhteyksiä, joita ei ole määritelty MIDP:ssä. Tämä myös rajoitti käytettävien laitteiden joukkoa.

Kaksi ensimmäistä ongelmaa ratkaistiin jakamalla luokkakirjasto kahteen tasoon. Ylempi taso sisältää vain pienimmän mahdollisen joukon luokkia ja toiminnallisuudet, joita tarvitaan asiakassovelluksen ja sovelluspalvelimen väliseen kommunikointiin. Ylemmän tason luokat sisältävät vain ominaisuuksia, joita MIDP 1.0 tukee. Alemman tason luokat ovat periytetty ylemmän tason luokista ja ne lisäävät toiminnallisuuden, jotka alkuperäinen

luokkakirjasto sisälsi ja jota ei ylemmän tason kirjastoon voitu toteuttaa. Kun jakamisen yhteydessä alemman tason luokkien ja metodien nimet pidetään samoina kuin alkuperäisessä luokkakirjastossa, ei olemassa oleviin sovelluksiin tarvitse tehdä mitään muutoksia. Itse asiassa kukaan muu kuin mobiilikäyttöliittymäsovelluskehys ei ole tietoinen kirjaston jakamisesta.

Sovelluskehyksessä ei ole tietoliikenteen salausta sen toteuttamiseen tarvittavan muistimäärän ja prosessoritehon vuoksi. Näin ollen tietoliikenneluokkia muokattiin siten, että salaus toteutuu vasta alemman tason luokissa. Palvelimelta asiakassovellukseen päin tapahtuvalle tietoliikenteelle ei nähty tarvetta, joten myös sen toteutus jätettiin alemman tason luokille. Näin säästettiin hieman luokkakirjaston viemää tilaa. Toisaalta sen toteuttamiseen vaadittavia luokkia ei ole määritelty pakollisiksi MIDP 1.0:ssa, joten toteuttamalla se tingittäisiin yhteensopivuudesta.

Kolmas ongelma ratkaistiin muokkaamalla VetoRPCn toteuttamia luokkia siten, että niillä voidaan käyttää sekä http- että socket-yhteyksillä. Tämä tehtiin määrittelemällä uusi *IYhteys*-rajapinta, jota VetoRPC-protokollan luokat käyttävät normaalin J2SE:n *Socket*-luokan sijasta. Rajapinnassa määritellään metodit *Output*- ja *InputStreamien* pyytämiseen sekä yhteyden avaamiseen ja sulkemiseen tarvittavat metodit. MIDP-sovelluksissa käytetään *IYhteys*-rajapintaa toteuttavaa luokkaa *MIDPYhteys*. VetoBox-sovelluspalvelimissa sekä Swing-käyttöliittymissä käytetään *VetoYhteys*-luokkaa, joka myös toteuttaa *IYhteys*-rajapintaa. *MIDPYhteys*-luokka käyttää luonnollisesti yhteyden muodostamiseen CLDC:ssä määriteltyä GCF:ää ja *VetoYhteys* J2SE:n *Socketia*.

MVC-mallissa Model-kerros yleensä ilmoittaa tilansa muutoksista View-kerrokselle. Tilan muutosten ilmoittamiseen on työssä käytetty Observer-suunnittelumallin [13, s. 293 - 303] hieman muunneltua versiota, jota varten on luotu *Subject*- ja *Observer*-rajapinnat (liite 1). *Subject*-rajapinta määrittelee metodit, jolla *Observer*-rajapintaa toteuttavat luokat (tarkkailijat) voivat lisätä tai poistaa itsensä *Subject*-rajapintaa toteuttavan luokan (tarkkailtava) huomautettavien olioiden listaan. *Subject*-rajapinta määrittää perinteisestä mallista poiketen tarkkailtavan tilan kyselyn mahdollistavan metodin update(). *Subject* ei määritä tarkkailijoiden huomauttamiseen tarkoitettua metodia, joka perinteisessä mallissa on usein notify()-niminen.

Observer-rajapinta sisältää puolestaan vain metodin notify(*Subject*), jonka avulla tarkkailtava voi huomauttaa tilansa muutoksesta kaikkia huomautettavien olioiden listaan rekisteröityneitä olioita. *Observer*-rajapinnan notify()-metodi saa parametrikseen viitteen tarkkailtavaan, jonka avulla tarkkailija voi halutessaan pyytää itseään kiinnostavan osan tarkkailtavan sisäisestä tilasta. Tila kysytään käyttämällä joko *Subject*-rajapinnassa määriteltyä update(*Observer*)-metodia tai suoraan tarkkailtavan omia tilakyselymetodeita, mikäli tarkkailtavan tyyppi tunnetaan. *CommunicationModel* toteuttaa *Subject*-rajapinnan, mutta sen käyttö sovelluskehyksen tasolla on jäänyt vähäiseksi.

6.5 Controller-taso

Controller-kerroksen tehtävä on käsitellä käyttöliittymäkomponenteilta vastaanottamansa tapahtumat ja muuttaa ne Modelin tilan muutoksiksi. Controller-kerros siis luo sovelluksen käyttäytymisen. [14.]

Työssä tehdyssä sovelluskehyksessä Controller-kerrosta vastaa abstrakti *VetoController*-luokka (kuva 13, s. 56). Luokan vastuulla on toteuttaa oletus-käyttäytyminen perustapahtumille, kuten edelliseen ruutuun palaaminen, lookup-kenttien ja alataulukomentojen käsitteleminen, yms. tapahtumat. Sovelluksen kulun kontrolli siirretään *VetoController*-luokalle käyttäjän valittua käynnistettävä sovellus. Sovelluksen valinta suoritetaan *VetoMIDlet*-luokassa, joka periytyy *MIDlet*-luokasta.

VetoMIDlet-luokan tehtävänä on toimia sovelluksen käynnistysalustana ja se sisältää toiminnot aloitusruudun näyttämiseen. Aloitusruutuna toimii lista, jossa on lueteltu kaikki valittavissa olevat sovellukset (yksi jakelupaketti voi sisältää useamman sovelluksen). Lista muodostetaan parametrien avulla, jotka luetaan joko jakelupaketin mukana tulevista parametreista tai sovelluksen omasta parametritiedostosta. *VetoMIDlet* on abstrakti luokka ja sisältää yhden metodin, jonka toteutus jätetään järjestelmäkohtaiseksi. Tämä metodi saa parametrina listalta valitun sovelluksen nimen ja sen tulee palauttaa parametrin perusteella oikean tyyppinen *VetoController*-luokka. Tämän jälkeen sovelluksen kontrolli siirretään *VetoController*-luokalle, mutta

MIDlet-luokka kuitenkin vastaanottaa kaikki *MIDlettien* tilanmuutokset kuten esim. Paused-tilaan siirtymisen. *MIDlet*-luokka voidaan siis katsoa kuuluvaksi Controller-kerrokseen.

VetoController-luokka vastaanottaa kontrollin abstraktilla alusta()-metodilla, johon sovelluskehyksen käyttäjän tulee kirjoittaa koodi, joka näyttää sovelluksen ensimmäisen ruudun. Tästä eteenpäin paketin com.veto.mobile.ui käyttöliittymäkomponenttien tapahtumien käsittely suoritetaan oletusarvoisesti *VetoController*-luokan commandAction()- tai itemStateChanged()-metodeissa, joista ne välitetään toteuttavalle luokalle kasitteleKomennot()- tai kasitteleItemMuutokset()-metodien avulla. Näihin metodeihin tulee sovelluskehittäjän kirjoittaa sovelluksen käyttäytyminen ja tapahtumienkäsittely.

Sovelluskehys käsittelee seuraavat tapahtumat:

- Sovelluksenlaajuinen poistu-komento. *VetoController*-luokka sisältää poistu-komennon, jota kaikki kehystä käyttävät sovellukset voivat käyttää. Komento lopettaa sovelluksen suorittamisen.
- Sovelluksenlaajuinen takaisin-komento. *VetoController*-luokka sisältää myös sovelluksenlaajuisen takaisin-komennon. Komento palaa edelliseen näyttöön, mikäli sellainen on ollut.
- Lookup-komennot, jotka avaavat lookup-näytön. Lookup-komennot on käsittely tarkemmin luvussa 6.6.
- Alataulu-komennot, jonka avulla näytetään, muokataan ja lisätään alataulun rivejä. Tämä on käsitelty tarkemmin luvussa 6.6.

6.6 View-taso

View-kerroksen tehtävä on näyttää sovelluksen käyttäjälle Modelin sisäinen tila. Se päättää miten Modelin sisältämä tila esitetään ja on vastuussa itsensä päivittämisestä, kun Modelin tila muuttuu. Tämä voidaan toteuttaa ns. ”push”-mallilla, missä View-komponentit rekisteröivät itsensä Modelin tilan kuuntelijoiksi, jolloin Modelin tilan muuttuessa tämä huomauttaa View-komponentteja siitä. Toinen tapa toteuttaa päivitys on käyttää ns. ”pull”-mallia, missä View-komponentit itse päättävät, koska on tarpeen hakea ja näyttää Modelin nykyinen tila. [15.]

Työn sovelluskehyksen View-tason muodostavat MIDP:n käyttöliittymä-komponenttien lisäksi useat niistä perityt käyttöliittymä-komponentit, jotka kaikki sijaitsevat paketissa com.veto.mobile.ui (liite 2). Osa näistä luokista on abstrakteja luoden toiminnallisuutta sovelluskehystasolla ja osa on käyttö-valmiita komponentteja. Lisäksi paketti sisältää muutamia käyttöliittymä-komponenttien käyttämiä apuluokkia. Paketin com.veto.mobile.ui luokat käyttävät pääasiassa ”push”-mallia tilansa päivittämiseen.

VetoForm, VetoFormListener ja VetoLookupCommand

VetoForm-luokka on abstrakti luokka, joka periytyy *Form*-luokasta. Luokka lisää *Formiin* toiminnallisuuden, jolla VetoBox-sovelluspalvelimelta saatavien *TauluRakenne* tietojen perusteella voidaan luoda oikean tyyppisiä *Item*-olioita. Luokka sisältää myös lookup-kenttien sovelluskehystasolla tapahtu-vaan käsittelyyn tarkoitettuja metodeja. Metodilla appendLookupItem() voidaan lisätä lomakkeelle lookup-kenttä. Metodi tarvitsee parametriksi *VetoLookupCommand*-tyyppisen komennon sekä Item-olion, johon lookup-komento liittyy. Jokainen lookup-kenttä lisää laitteen valikkoon uuden komennon ja tällä komennolla käyttäjä saa avattua lookup-arvojen haku- ja valintaruudut.

VetoLookupCommand periytyy *Command*-luokasta (liite 2) ja lisää siihen vain tietoja lookup-komennosta, ei toiminnallisuutta. Näiden tietojen avulla *VetoController*-luokka osaa luoda edellä mainitut lookup-arvojen haku- ja valintaruudut sekä asettaa valitun arvon oikeaan kenttään.

VetoForm-luokka aiheuttaa tapahtuman ennen ja jälkeen normaalin sekä lookup-kentän (Itemin) lisäystä. Näillä tapahtumilla annetaan sovelluskehyksen käyttäjälle mahdollisuus muuttaa kenttien arvoja ja ominaisuuksia ennen tai jälkeen niiden lomakkeelle asettamista. Tapahtumiin päästään käsiksi toteuttamalla *VetoFormListener*-rajapinta ja rekisteröimällä tämä toteuttava luokka kuuntelemaan *VetoFormin* tapahtumia käyttämällä sen asetaVetoFormListener()-metodia.

VetoEditForm ja VetoAlatauluCommand

VetoEditForm on tarkoitettu mallintamaan Swing-kirjastolla toteutetun käyttöliittymän Tietosivuja (kuva 10, s.50). Tietosivuilla mahdollisesti olevat alataulut ja lookup-kentät joudutaan MIDP-sovelluksissa näyttämään eri tavalla kuin Swing-sovelluksissa. Alataulut näytetään omina ruutuinaan, koska MIDP ei tarjoa valmista taulukkokomponenttia ja sellaisen luominen *Canvas*-luokalla olisi ollut työn kannalta liian aikaa vievää. Lisäksi *Canvasilla* tehtäessä taulukkokomponentti olisi joka tapauksessa omana ruutunaan, koska sitä ei voida liittää osaksi *Form*-komponenttia. Lookup-kenttien lisääminen ja käsittely tapahtuvat edellisessä luvussa kuvatulla tavalla.

VetoEditForm periytyy *VetoForm*-luokasta (liite 2). Luokka sisältää toiminnallisuudet, joilla se luo muokattavat kentät *Taulu-* ja *TauluRakenne*-luokkien sekä rivin datan perusteella, mikäli käyttäjällä on muokkaukseen vaadittavat oikeudet. Jos taululla, jonka riviä muokataan, on alatauluja ja niitä halutaan näyttää tai muokata ne annetaan *VetoEditFormille*, jolloin se luo niiden käsittelyyn tarvittavat alataulu-komennot. Jokaista alataulua kohden luodaan muokkaa-, lisää- tai näytä-komennot riippuen käyttäjän omaamasta käyttöoikeustasosta. Lisättävät komennot ovat *VetoAlatauluCommand*-tyyppisiä ja ne toimivat samalla tavalla kuin lookup-komennot eli lisäävät *VetoController*-luokkaa varten *Command*-luokkaan lisätietoja alataulusta, johon komento liittyy.

Muokkaus- ja näytä alataulu-komennot käyttäytyvät siten, että kun käyttäjä valitsee alataulu-komennon, luodaan ensin lista, joka näyttää lyhyet tiedot alataulun riveistä. Näistä käyttäjä sitten valitsee jonkin rivin lähempään tarkasteluun, jolloin valitusta rivistä luodaan lomake, joka sisältää joko muokattavat kentät täytettyinä rivin arvoilla (muokkaa-komento) tai pelkkiä *StringItemejä*, jota ei voida muokata (näytä-komento). Lisää komento luo suoraan lomakkeen, joka sisältää alataulun riviä vastaavat tyhjät kentät, joihin käyttäjä voi syöttää uudet arvot.

VetoHakuForm ja VetoHakuTulosList

VetoHakuFormilla mallinnetaan Swing-käyttöliittymien hakusivua. Hakusivun mallinnuksessa on ongelmana taulukko, jossa hakutulokset näytetään (kuva 9). Ongelma on ratkaistu näyttämällä ensin vain hakuehto-kentät, joihin käyttäjä kirjoittaa hakuehdot. Haun suorituksen jälkeen siirrytään automaattisesti ruutuun, jossa näytetään haun tulokset. *VetoHakuForm* koostuu hakukentät sisältävästä lomakkeesta sekä hakutulokset näyttävästä *VetoHakuTulosList*-komponentista (liite 2). Näiden kahden luokan suhde on rakennettu siten, että ne näkyvät ja käyttäytyvät sovelluskehyksen käyttäjälle yhtenä komponenttina.

VetoHakuForm luo *TauluRakenne*-tietojen perusteella hakukentät ja näyttää ne käyttäjälle. Käyttäjä syöttää hakuehdot ja suorittaa haun, jolloin haun tulokset näytetään *VetoHakuTulosListillä*, jonka *VetoHakuForm* luo ja tuhoaa tarvittaessa. Kun käyttäjä valitsee jonkin rivin *VetoHakuTulosListiltä*, ei tapahtumaa käsitellä *VetoControllerissa* kuten normaalisti, vaan *VetoHakuTulosList* käsittelee sen itse. Rivin valintakomennon vastaanottaessaan *VetoHakuTulosList* asettaa valitusta rivistä tarvittavat tiedot *VetoHakuFormille*, joka välittää *VetoHakuTulosListiltä* saamansa haku-komennon *VetoController*-luokalle. Näin valintakomento näyttää tulevan *VetoHakuFormilta* ja sovellus voi jatkaa toimintaansa tietämättä *VetoHakuTulosListin* olemassa olosta mitään.

VetoSelausList ja ResultSet

VetoSelausList-luokkaa käytetään esittämään listoja, joiden tiedot haetaan Modelilta ja näytettään n-kappaleen paloissa. Luokka käyttää hyväkseen com.veto.mobile-paketissa määriteltyä abstraktia *ResultSet*-luokkaa. *ResultSet*-luokka kapseloi tiedon haun Modelilta (VetoBox-sovelluspalvelimelta) n-riviä kerrallaan eteen- tai taaksepäin selattavaan muotoon. *VetoSelausList*-luokka toteuttaa myös *Subject*-rajapinnan, jolloin siinä tapahtuvia tilan muutoksia voidaan tarvittaessa kuunnella. *VetoController*-luokka rekisteröikin itsensä kuuntelemaan muutoksia käyttäessään *VetoSelausListiä* alataulujen käsittelyssä.

VetoLoginForm

VetoLoginForm periytyy *Form*-luokasta ja sisältää kolme tekstikenttää kirjautumistietojen syöttämistä varten. Luonnin yhteydessä *VetoLoginForm* etsii parametritiedostosta oletusarvoja tekstikenttiin, jotta käyttäjän syötteiden antamisen määrää voitaisiin vähentää. Luokka sisältää metodit kenttien arvojen lukemiseksi.

6.7 Parametrit

Käyttöliittymäsovelluskehys käyttää parametreja erilaisten sovelluksista riippuvaisten muuttujien arvojen asettamiseen. Tällaisia ovat mm. sovelluspalvelimen url-osoite, portti tai käytettävän tietoliikenneyhteyden tyyppi. Nämä parametrit luetaan ensisijaisesti tietuekannasta (recordstoresta), jonka nimi määritellään jakelupaketin mukana tulevassa JAD- tai manifest-tiedostossa ominaisuuden nimellä ”INI_TIEDOSTO”. Tämä on ainoa pakollinen ominaisuus CLDC:n ja MIDP:n määrittämien pakollisten ominaisuuksien lisäksi, jotka täytyy asettaa jakelupakettia luotaessa.

Parametrien lukeminen sovelluksissa onnistuu *RecordHandler*-luokan avulla, jossa on määritelty staattiset metodit parametrien lukemiseen ja kirjoittamiseen RMS:ään. Kirjoitettavat parametrit ovat merkkijonomuotoisia nimi-arvo-pareja, kuten *MIDlet*-pakkauksen jakelupakettien ominaisuudetkin. Parametrien muoto on hieman erilainen kuin *MIDlet*-pakkauksessa. Parametrit kirjoitettaan RMS:ään muodossa:

parametrin nimi=parametrin arvo

Parametrien lukeminen tapahtuu siten, että ensisijaisesti parametri luetaan sovelluksen parametritiedostosta. Jos parametrin nimellä ei löydy parametria tiedostosta, etsitään samalla nimellä arvoa sovelluksen jakelupaketin ominaisuuksista. Arvon löytyessä se palautetaan, mutta jos arvoa ei löydy, niin palautetaan oletusarvo, joka voidaan antaa parametrin lukemisen yhteydessä lisätietona parametreja lukevalle lueInista()-metodille. Jos lueInista()-metodille annettu oletusparametri poikkeaa null-arvosta ja parametrin nimellä ei löytynyt arvoa, kirjoitetaan oletusarvona saatu parametrin arvo sovelluksen parametritiedostoon.

Koska parametrit kuuluvat olennaisesti sovelluskehykseen, on niiden hallintaa varten tehty oma sovellus, joka liitetään aina mukaan järjestelmän jakelupakettiin. Tämä sovellus on yksinkertainen ja se on kirjoitettu kokonaan yhteen luokkaan nimeltä *Parametrit*. *Parametrit*-luokka periytyy *MIDlet*-luokasta ja siinä käytetään hyväksi MIDP:ssä määriteltyä ominaisuutta, missä saman *MIDlet*-paketin *MIDletit* voivat lukea ja muokata toisten *MIDlettien* tekemiä tietuekokoelmia eli tiedostoja. Sovelluksella lisätään, poistetaan ja luetaan parametreja. Lisäksi sovelluksella luetaan ja poistetaan virhelokeja, joita toiset *MIDletit* ovat mahdollisesti kirjoittaneet. Sovelluksella tarkistetaan myös käytettävän laitteen sisältämät J2ME-profiilit.

6.8 Muita ominaisuuksia

Paketissa com.veto.mobile esitellään *Pauseable*-rajapinta, jonka avulla hallitaan *MIDletin* tilan muutokset aktiivisesta passiiviseen ja päinvastoin. Rajapinnan avulla toteutetaan Composite-suunnittelumallia [13]. Rajapinta määrittelee pause()- ja resume()-metodit, jotka molemmat saavat parametrina long-tyyppisen kokonaisluvun. *Pauseable*-rajapintaa toteuttavaan luokkaan on tarkoitus kirjoittaa pause()-metodiin koodi, joka tallentaa olion tilan siten, että se myöhemmin voidaan palauttaa resume()-metodissa. Käytännössä tämä tarkoittaa olion primitiivi-muuttujien kirjoittamista RMS:ään ja kutsumalla jäsen olioiden pause()-metodia samalla parametrilla, jonka olio sai oman pause()-metodin kutsun yhteydessä. Parametrina saatavan long-tyyppisen muuttujan avulla luokkien on pause()-metodissa tarkoitus nimetä yksilöllisesti tiedosto, johon ne tietonsa tallentavat. Parametrin arvo on käytännössä Paused-tilaan siirtymisen ilmoitushetken kellonaika. Tämä arvo tallennetaan *VetoMIDlet*-luokassa jäsenmuuttujaan ja arvo annetaan resume()-metodille kun Paused-tilasta palataan takaisin Active-tilaan. Kun Paused-tilasta palataan Active-tilaan, kutsutaan resume()-metodia, jossa luokka alustaa tilansa samaksi kuin se oli ennen Paused-tilaan siirtymistä. Tämä taas tarkoittaa käytännössä primitiiviarvojen lukemista RMS:stä ja uusien instanssien luontia jäsenolioista, jonka jälkeen kutsutaan niiden resume()-metodia oman resume()-metodin parametrilla.

DisplayableManager-luokka sijaitsee paketissa com.veto.mobile.ui ja sen avulla hallitaan eri *Displayable*-olioiden näyttämistä laitteen näytöllä.

DisplayableManagerin toiminta perustuu pinoon, joten sen avulla on helppo palata taaksepäin edellisiin näyttöihin. Kun sovelluskehystä käytettäessä halutaan näyttää uusi *Displayable*-olio, käytetään *VetoController*-luokassa esitellyn *DisplayableManager*-luokan instanssin metodia pushDisplayable(). Metodi työntää edellisen näytön pinon päällimmäiseksi ja asettaa parametrina saamansa *Displayablen* näytölle. Jos halutaan palata edelliseen näyttöön, kutsutaan *DisplayableManagerin* popDisplayable()-metodia, jolloin pinon päällimmäinen näyttö asetetaan näytölle. Luokka sisältää metodin, jolla korvataan nykyinen näyttö uudella ilman, että nykyinen näyttö työnnettäisiin pinoon.

Paketissa com.veto.mobile oleva *RecordHandler*-luokka sisältää kokoelman staattisia metodeita, joilla käsitellään MIDP:n tiedostoja eli RMS:ää. Samassa paketissa on *ErrorHandler*-luokka, joka sisältää kokoelman staattisia metodeita. *ErrorHandler*-luokan metodit on tarkoitettu erilaisten virheilmoitusten eli *Alertien* näyttämisen sekä virheilmoitusten lokitiedostoihin kirjaamisen helpottamiseksi.

7 ESIMERKKISOVELLUS

Tässä luvussa tehdään vaihe vaiheelta esimerkkisovellus käyttäen työssä toteutettua käyttöliittymäsovelluskehystä. Esimerkkisovelluksena käytetään Asiakas-sovelluksen mobiiliversiota. Sovelluksella voidaan hakea, muokata ja lisätä asiakasrekisterin tietoja. Sovelluksen kaikkien luokkien ohjelmakoodi listaukset ovat liitteenä 4. Koodiesimerkeissä olevat kolme pistettä ’...’, ilmoittavat, että tässä kohtaa on poistettu koodia, joka ei ole esimerkin kannalta tärkeää.

7.1 AsiakasMIDlet

Sovelluksen kirjoittaminen aloitetaan luomalla *VetoMIDlet*-luokasta periytyvä sovelluskohtainen *MIDlet*. Se toimii sovelluksen käynnistävänä luokkana ja nimetään *AsiakasMIDletiksi*. Luokkaan kirjoitetaan yksi metodi ja se on *VetoMIDletissä* määritelty abstrakti kaynnistaSovellus()-metodi. Metodissa valitaan käynnistettävä sovellus perustuen parametrina saatavaan

sovelluksen nimeen ja palautetaan sitä vastaava *VetoController*-luokan ilmentymä. Metodi näyttää tältä:

```
public VetoController kaynnistaSovellus(String sSovellus,
String sPaataulu) {
  if(VetoCore.sama(sSovellus, "MIDPASIAKAS"))
    return new AsiakasController(sSovellus, sPaataulu);
  return null;
}
```

Kun laite käynnistää *VetoMIDletin*, se luo valintalistan käynnistettävistä sovelluksista parametreista lukemiensa tietojen perusteella. Käyttäjän valittua sovelluksen, kutsutaan kaynnistaSovellus()-metodia. Metodista palattuaan *VetoMIDlet*-luokka suorittaa erinäisiä alustustoimintoja, jonka jälkeen se siirtää kontrollin palautetulle *VetoController*-luokalle.

Kuvassa 14 esitetään *VetoMIDlet*-luokan luoma sovelluksen valintaruudun eräs esitystapa. Esimerkissä ei ole kuin yksi valittava sovellus MIDPASIAKAS.

Kuva 14. *VetoMIDlet*-luokan luoma käynnistettävän sovelluksen valintaruutu

7.2 AsiakasController

AsiakasMIDlet-luokan kirjoittamisen jälkeen luodaan tarvittavat *VetoController*-luokat. AsiakasController-luokka periytetään *VetoController*-luokasta ja siinä toteutetaan metodit alusta(), annaCommunicationModel(), destroyApp(), kasitteleItemMuutokset(), kasitteleKomennot(), pause() ja resume().

7.2.1 Kirjautuminen

Kontrollerin muodostimeen ei tarvitse kirjoittaa sovelluskohtaista koodia, yläluokan muodostimen kutsu riittää. *AsiakasMIDlet* siirtää kontrollin *AsiakasController*-luokalle kutsumalla sen alusta()-metodia. Tänne kirjoitetaan varsinaisen sovelluksen käynnistävä koodi. Metodiin kirjoitetaan lähes poikkeuksetta login-näytön luova ja näyttävä sekä *CommunicationModelin* alustava koodi. Esimerkkisovelluksen alusta()-metodi:

```
public void alusta() {
   luoLoginLomake(false);

   try {
     cModel = new AsiakasCommunicationModel(this);
     // Nyt voidaan kirjautua, koska model on luotu ja voidaan
     // ottaa yhteys sovelluspalvelimeen.
     login.kirjautuminenSallittu(true);
   }
   catch(SysteemiException se) {
     naytaJaKirjaaException("Tietoliikennevirhe",
                            "AsiakasController.alusta(): "
                            +se.getMessage());
   }
 }
```

Kirjautumislomakkeen luonti suoritetaan omassa metodissa, jotta se voidaan helposti luoda myöhemmin uudestaan. Kirjautumislomakkeen luonnin ja näyttämisen jälkeen muodostetaan yhteys sovelluspalvelimeen luomalla sovelluksen *CommunicationModel*. *CommunicationModel*-luokka pyytää heti luonnin yhteydessä VetoBox-sovelluspalvelinta luomaan uuden istunnon (sessio), mistä aiheutuu tietoliikennettä. Normaalisti tietoliikennettä vaativia operaatioita ennen näytölle tulee asettaa lomake (kuva 16, s. 71), joka ilmoittaa käyttäjälle pitkäkestoisen toiminnon käynnistämisestä *VetoController*-luokan metodilla startCommunication(String). Parametrina annetaan lomakkeella käyttäjälle näytettävä ilmoitus ja lomake tulee poistaa näytöltä käyttämällä stopCommunication()-metodia pitkäkestoisen operaation suorituksen jälkeen. Kirjautumislomakkeen tapauksessa ei startCommunication()-metodia kuitenkaan pidä käyttää. Tämä mahdollistaa, että käyttäjä voi syöttää käyttäjätunnusta ja salasanaa samalla, kun laite luo taustalla uutta istuntoa.

Kuva 15 esittää sovelluksen kirjautumisruutua. Kirjautumisruudun ilmestyessä näytölle ei käyttäjä heti voi valita ”Kirjaudu”-komentoa, vaan se ilmestyy valittavaksi vasta kun Model on luonut uuden istunnon.

Kuva 15. *VetoLoginForm*-luokan muodostama kirjautumisruutu

7.2.2 Tapahtumien käsittely

Tapahtumien käsittely kirjoitetaan *VetoController*-luokan määrittelemään kasitteleKomennot()-metodiin. Metodi saa parametrina viitteen komentoon ja *Displayable*-luokkaan, johon komento liittyy. Tapahtumien käsittely kannattaa suorittaa siten, että jokaista *Displayable*-luokkaa varten luodaan oma tapahtumienkäsittelymetodi, jolle välitetään tapahtuman laukaissut komento. *Displayable*-kohtaisia luokkia ei tapahtumien käsittelyyn kannata käyttää resurssien kulutuksen vuoksi (ks. Luku 6.3). Esimerkkinä *AsiakasController*-luokan kasitteleKomennot()-metodin alkuosa, jossa komento välitetään kasitteleLoginKomennot()-metodille, jos komento on peräisin login-lomakkeelta:

```
public void kasitteleKomennot(Command c, Displayable d) {
   try {
     if(d == login)
       kasitteleLoginKomennot(c);
     else if(d == hakulomake)
       kasitteleHakulomakeKomennot(c);
     ...
}
```

kasitteleLoginKomennot()-metodi ei tarvitse parametriksi kuin suoritetun komennon, koska tiedetään, että käsitellään vain login-lomakkeeseen liittyviä komentoja. Tässä metodissa tutkitaan, mikä komento vastaanotettiin

ja toimitaan sen mukaan. *VetoLoginForm*-luokka luo oletusarvoisesti ”Kirjaudu”- ja ”Poistu”-komennot. Näistä ”Poistu”-komento käsitellään sovelluskehyksen tasolla, mutta ”Kirjaudu”-komento täytyy käsitellä itse. Seuraavassa on esimerkkinä lomakkeiden käsittelystä kasitteleLogin-Komennot()-metodi, joka on varsin normaali tapahtumien käsittelymetodi:

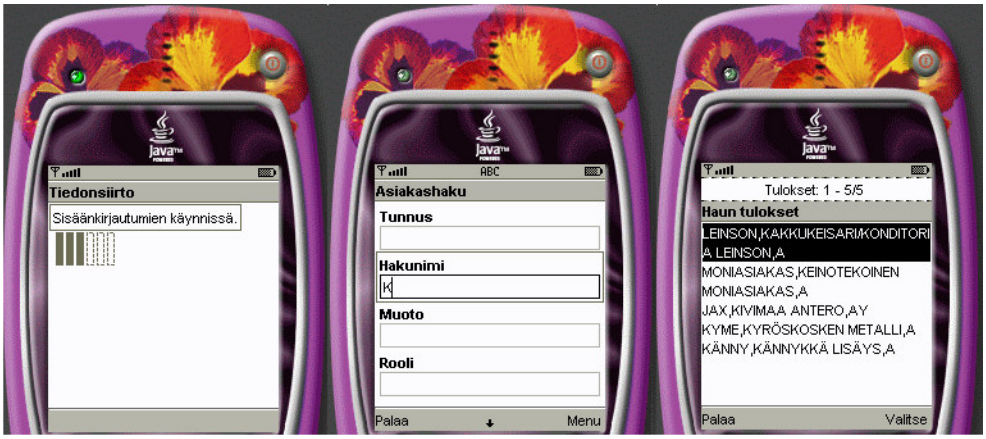
```
private void kasitteleLoginKomennot(Command c)
throws VetoException {
  if(VetoCore.sama("Kirjaudu",c.getLabel())) {
    try {
      startCommunication("Sisäänkirjautumien käynnissä.");
      cModel.kirjaudu(login.annaYritys(),
                      login.annaKayttaja(),
                      login.annaSalasana());

     // Kirjautuminen onnistui, luodaan hakulomake.
      luoHakulomake();
    }
    catch(SysteemiException se) {
      // Ei onnistunut, joten näytetään virheilmoitus ja palataan
      // takaisin Login-ruutuun.
      naytaException("Kirjautuminen epäonnistui.",
                  "Tarkista yritystunnus/käyttäjänimi/salasana ja "
                  +"yritä uudelleen.");
      kirjaaException("AsiakasController.kasitteleLoginKomennot():"
                   +"Kirjautumen epäonnistui. Virhe: "
                   +se.getMessage());
      return;
    }
    finally { stopCommunication(); }
  }
}
```

Mikäli käyttäjä on valinnut kirjautumiskomennon, pyydetään *CommunicationModelia* kirjaamaan käyttäjä sisään login-lomakkeelta saatavilla parametreilla. Kun kirjautuminen on suoritettu luodaan normaalia Swing-käyttöliittymää vastaava ”hakusivu”. Tämä tehdään metodissa luoHakuLomake(), jossa luodaan ja näytetään sovelluksen seuraava lomake. Kirjautuminen ja hakulomakkeen luonti käyttävät molemmat tietoliikenneyhteyttä, mikä on kohtalaisen hidasta MIDP-ympäristössä lähinnä yhteyden muodostamiseen kuluvan ajan vuoksi. Tästä syystä ennen operaatioiden kutsumista on näytölle asetettu kirjautumisesta kertova lomake startCommunication()-metodilla. Lomake poistetaan näytöltä try-lohkon finally-osassa, jolla varmistetaan, ettei tämä lomake jää näytölle vaikka kirjautuminen epäonnistuisi.

Kuva 16 esittää kirjautumisruudun jälkeen ilmestyvän ruudun, joka ilmoittaa tietoliikenteen ja pitkäkestoisen operaation suorittamisesta. Kirjautumisen

onnistuttua siirrytään hakulomakkeelle, jolle voidaan syöttää hakuehtoja. Kuvassa 16 haetaan kaikki asiakkaat, joiden hakunimi alkaa K-kirjaimella. Viimeinen ruutu on hakusivun hakuehtojen tulokset. Tähän ruutuun tullaan hakusivulta valitsemalla ”hae”-komento, joka löytyy valikosta ”Menu”.



Kuva 16. Kuvassa esitetään esimerkkisovelluksen kirjautumisoperaation suorituksesta ilmoittava ruutu, hakusivu-ruutu sekä hakutulokset esittävä ruutu

7.2.3 Hakusivu-lomake

Swing-pohjaisen käyttöliittymän hakusivua (kuva 9, s. 49) mallinnetaan mobiilisovelluksissa *VetoHakuForm*-luokalla (kuva 16). *VetoHakuForm*-luokan luominen on yksinkertaista, koska se hakee tarvitsemansa tiedot Modelilta itse. Ainoa muistettava asia on asettaa lomake näytölle. *VetoHakuFormin* luonti tapahtuu seuraavasti:

```
...
hakulomake = new VetoHakuForm(this, "Asiakastietojen haku");
dManager.korvaaDisplayable(hakulomake);
...
```

Hakusivu aiheuttaa ”Takaisin”-, ”Hae”- ja ”Valitse”-komentoja. ”Hae”-komento ilmoittaa, että käyttäjä on kirjoittanut hakuehdot ja haluaa suorittaa haun. Komento ”Valitse” puolestaan ilmoittaa, että käyttäjä on valinnut jonkin rivin haun tuloksista ja haluaa sen tarkemmat tiedot. ”Takaisin”-komento käsitellään sovelluskehystasolla. Esimerkkisovelluksessa hakulomakkeen tapahtumat käsitellään metodissa kasitteleHakulomakeKomennot(). Metodissa käsitellään ”Takaisin”-komentoa normaalista poikkeavalla tavalla. Vaikka ”Takaisin”-komento käsitelläänkin sovelluskehyksen tasolla

(*VetoController*-luokassa), annetaan myös sovelluskohtaisessa kontrolle-rissa mahdollisuus sen ”jatkokäsittelyyn”. Esimerkki *VetoHakuFormin* komentojen käsittelystä:

```
private void kasitteleHakulomakeKomennot(Command c)
throws VetoException {
   // Palataanko takaisin, kirjautumisruutuun?
   if(c == annaBackCommand()) {
     // Palataan, joten suljetaan sessio.
     cModel.suljeSessio();

     // Luodaan ja näytetään login lomake.
     luoLoginLomake(true);
     return;
   }

   // Tutkitaan muut komento mahdollisuudet.
   if(VetoCore.sama("Hae",c.getLabel())) {
     try {
       startCommunication("Haku käynnissä.");

       // Tehdään haku...
       cModel.suoritaHaku(hakulomake.annaHakuEhdot());

       // Sitten voidaan vasta näyttää haun tulokset...
       hakulomake.naytaHaunTulokset();
     }
     finally { stopCommunication(); }
   }
   else if(VetoCore.sama("Valitse",c.getLabel())) {
     luoTuloslomake();
   }
 }
```

7.2.4 Tietosivu-lomake

VetoEditForm-luokka on tarkoitettu mahdollisimman yleiskäyttöiseksi. Siksi sen käyttäminen on monimutkaisempaa kuin muiden käyttöliittymä-komponenttien. Luokan käyttö perustuu siihen, että sille annetaan eri metodien avulla tietoja, joiden perusteella se luo itsensä. Lomake luo itsensä vasta, kun sovelluskehittäjä kutsuu nayta()-metodia. *VetoEditForm*-luokka tarvitsee kaksi pakollista tietoa ennen kuin se voidaan luoda nayta()-metodilla. Nämä pakolliset tiedot ovat näytettävän (tai muokattavan) rivin data sekä riviin liittyvä metadata, kaikki muut tiedot ovat valinnaisia. Normaalisti luokan avulla mallinnetaan Swing-käyttöliittymien tietosivuja (kuva 10, s. 50).

VetoEditForm lisää lomakkeelle tarvittaessa lookup- ja alataulu-komennot, mutta muita komentoja se ei lisää. Tämä tarkoittaa, että sovelluskehittäjän tulee itse lisätä haluamansa komennot lomakkeelle, kuten esimerkiksi ”Takaisin”-komento. Luokan tapahtumien käsittely tapahtuu normaalin käytännön mukaan.

Esimerkkisovelluksessa halutaan asiakkaan perustietojen muokkaamisen ja lisäämisen lisäksi antaa käyttäjälle mahdollisuus lisätä, muokata sekä katsella riviin liittyvien alataulujen tietoja. Tällöin *VetoEditFormille* tulee antaa lisätietona viitteet alatauluihin taulukkomuodossa ennen nayta()-metodin kutsua.

Esimerkkisovelluksen asiakastietojen tietosivun luontiin tarvittava koodi on kirjoitettu luoTuloslomake()-metodiin. Metodissa asetetaan lomakkeelle *VetoFormListener*, jolla päästään muokkaamaan *Item*-komponentteja ennen ja jälkeen niiden lisäämistä lomakkeelle. Metodin koodi on seuraavanlainen:

```
private void luoTuloslomake() throws VetoException {
   try {
     startCommunication("Haetaan asiakkaan tiedot.");
     tuloslomake = new VetoEditForm(this, "Asiakas tiedot");
     //Kysytään rivin ID, jonka perusteella haetaan rivin tiedot.
     Object[] oRivi = cModel.annaRivi(hakulomake.annaRivinID());

     // annetaan rivin tiedot tuloslomakkeelle.
     tuloslomake.asetaData(oRivi);

     // annetaan rivin metadata tuloslomakkeelle.
     tuloslomake.asetaClientKentta(cModel.annaClientKentat());

     // Koska halutaan näyttää/muokata/lisätä alataulujen rivejä,
     // annetaan viitteet alatauluihin tuloslomakkeelle.
     tuloslomake.asetaAlataulut(cModel.annaAlataulut());

     // Asetetaan "takaisin"-komento.
     tuloslomake.addCommand(cmdBack);

     // Asetetaan item:ien lisäyksen kuuntelija.
     tuloslomake.asetaVetoFormListener(this);

     // Kun kaikki tarvittavat tiedot on annettu, voidaan käskeä
     // lomaketta "muodostamaan" itsensä.
     tuloslomake.nayta();

     // Näytetään lomake.
     dManager.pushDisplayable(tuloslomake);
   }
   finally { stopCommunication(); }
}
```

Esimerkkisovelluksessa asiakastietojen muokkaus/lisäys-lomakkeelle asetetaan vain ”Takaisin”-komento. Koska ”Takaisin”-komento käsitellään automaattisesti sovelluskehyksessä, lomakkeelle ei tarvitse kirjoittaa omaa tapahtumankäsittelijää. Kuva 17 esittää esimerkkisovelluksen tieto-sivuruudun.

Kuva 17. Esimerkkisovelluksen tietosivuruutu

7.2.5 Muut metodit

AsiakasController-luokka toteuttaa *Pauseable*-rajapinnan metodit resume() ja pause(). Metodeissa tallennetaan tarvittavien muuttujien ja luokkien tilat RMS:ään sekä palautetaan ne sieltä. annaCommunicationModel()-metodi ainoastaan palauttaa AsiakasCommunicationModel-luokan ilmentymän ja destroyApp()-metodi valmistelee Modelin tuhoamista varten. Esimerkki-sovelluksessa ei tarvita *Item*-komponenttien tilojen muutoksen kuuntelua, joten metodin kasitteleItemMuutokset() toteutus jätetään tyhjäksi.

AsiakasController-luokka toteuttaa *VetoFormListener*-rajapinnan. Rajapinta määrittelee neljä metodia, joista yhteen kirjoitetaan esimerkin vuoksi koodia. Tämän koodi katkaisee asiakkaan nimen 10 merkin mittaiseksi. Katkaisu tehdään metodissa ennenItemLisaysta(), jota kutsutaan ennen kentän lomakkeelle asettamista. Katkaisun suorittava koodi:

```
public void ennenItemLisaysta(Item item) {
   if( VetoCore.sama(item.getLabel(), "Nimi")) {
     String nimi = ((TextField) item).getString();
     if(nimi.length() > 10)
       ((TextField)item).setString(nimi.substring(0,9));
   } }
```

7.3 AsiakasCommunicationModel

Viimeinen luokka, joka täytyy kirjoittaa ennen sovelluksen kääntämistä, on *CommunicationModelista* periytyvä *AsiakasCommunicationModel*. *AsiakasCommunicationModelissa* on kolme pakollista metodia, jotka pitää kirjoittaa. Nämä ovat *CommunicationModelissa* määritellyt abstraktit metodit annaHakuResultSet(), onkoLisausOikeus() sekä onkoMuokkausOikeus(). Oikeustasojen kyselymetodit toimivat siten, että ensimmäisellä kerralla kumpaa tahansa kutsuttaessa, haetaan palvelimelta vaadittava oikeustaso ja verrataan sitä käyttäjän oikeustasoon. Vertauksen tulos tallennetaan *AsiakasCommunicationModelin* jäsenmuuttujiin, jolloin seuraavat käyttöoikeuskyselyt ovat nopeampia suorittaa.

Metodi annaHakuResultSet() tarvitaan *VetoHakuForm*-luokkaa varten. Mikäli sovelluksessa aiotaan käyttää *VetoHakuForm*-luokkaa tulee annaHakuResultSet()-metodin palauttaa halutun tyyppinen *ResultSet*. Jos haun tulosjoukkoa halutaan jostain syystä muokata tai suodattaa, voidaan se tehdä toteuttamalla oma *ResultSet*-luokka. Normaalisti hakutulosten selaukseen voidaan käyttää com.veto.mobile.ui-paketista löytyvää *SearchResultSet*-luokkaa. Esimerkkisovelluksen annaHakuResultSet() tekee juuri tällä tavalla:

```
public ResultSet annaHakuResultSet(boolean buffered)
throws VetoException {
   // onko hakuehtoja muutettu. Jos ei ole, voidaan palauttaa
   // edellisellä kerralla luotu searchresultset. (vältytään mahdollisesti
   // turhalta tietoliikenteeltä, mikä on hidasta ja maksaa).
   if(bHakuMuuttunut) {
     if(paataulu == null)
       avaaPataulu();
     if(paaKentat == null)
       haeClientKentta();

     hakuRS = new SearchResultSet(paataulu, paaKentat);
     // Asetetaan lippu, joka kertoo, että ollaan luotu SearchResultSet
     // uusimilla hakuehdoilla.
     bHakuMuuttunut = false;
   }
   return hakuRS;
 }
```

Buffered-parametrilla voi metodinkäyttäjä ilmoittaa, haluaako hän käyttöönsä puskuroidun luokan, joka tallentaa hakemansa tuloksen muistiin. Esimerkki-sovelluksessa parametri jätetään huomioimatta, koska *SearchResultSet*-luokka on aina puskuroiva. Metodissa käytetään *CommunicationModelissa*

määriteltyä muuttujaa bHakuMuutunut, joka ilmoittaa, onko suoritettu uusi haku edellisen annaHakuResultSet()-metodikutsun jälkeen. Muuttujan arvoksi asetetaan tosi (true) aina suoritaHaku()-metodia kutsuttaessa.

7.4 Jakelupaketin luominen

VetoMIDlet-, *VetoController*- ja *CommunicationModel*-luokkien sovelluskohtaisten totutusten luonnin jälkeen luodaan niistä uusi projekti käyttäen Sun Microsystemsin WTK-työkalua. Sen avulla voidaan sovellusta testata ja luoda jakeluun vaadittavat JAD- ja JAR-tiedostot. Esimerkkisovellus on käännetty ja paketoitu käyttämällä WTK:n versiota 2.1.

Jakelupaketin tekeminen aloitetaan luomalla uusi projekti valitsemalla WTK:n valikosta ”File – new project ... ”. WTK avaa ikkunan, jossa kysytään projektin nimeä ja *MIDlet*-luokkaa (kuva 18). Nimeksi annetaan ”Asiakas” ja *MIDletiksi* juuri luotu asiakas.*AsiakasMIDlet*.

Kuva 18. Uuden projektin luonti. Projektin nimen ja MIDlet-luokan kysely-ikkuna.

Tämän jälkeen kysytään projektin asetuksia, jossa asetetaan mm. JAD-tiedostoon tulevat ominaisuudet (ks. kappale 4.2.2). Seuraavassa on lueteltu kaikki tarvittavat muutokset, jotka täytyy tehdä oletusasetuksiin:

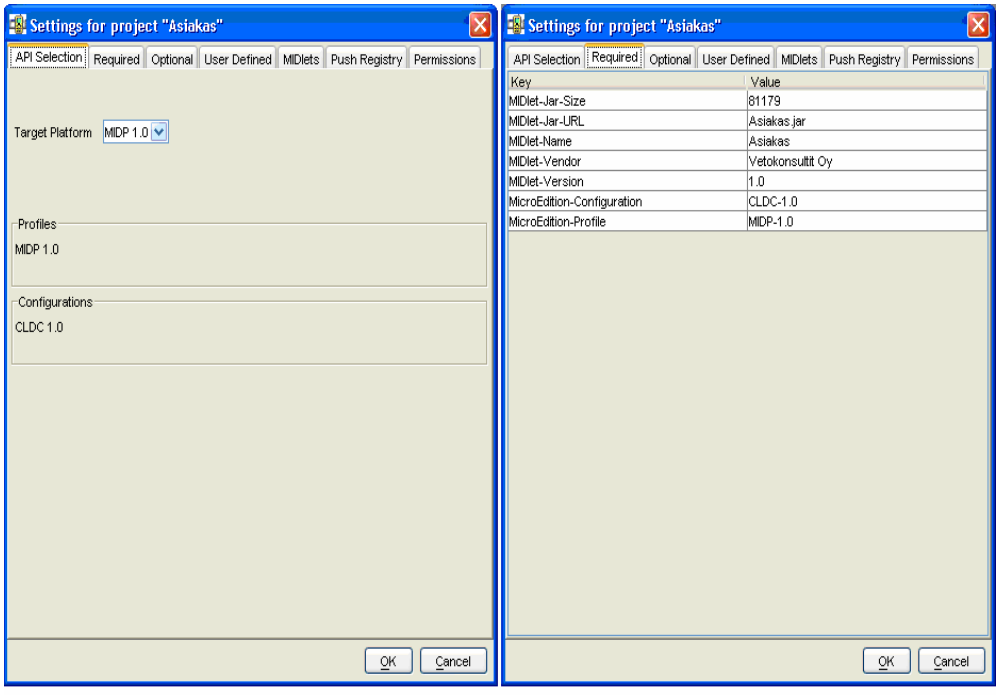
- API selection -välilehdellä Target Platform vaihdetaan MIDP 1.0:ksi
- Required-välilehdellä MIDlet-Vendoriksi kirjoitettaan ”Vetokonsultit Oy”
- User Defined -välilehdellä (kuva 20a) lisätään ominaisuuksia painamalla add-nappia ja antamalla ominaisuuden nimi. Tämän jälkeen muokataan ominaisuuden arvoa. Seuraavat ominaisuudet ja arvot lisätään
    - nimi ”INI_TIEDOSTO”, arvo ”asiakas.ini”
    - nimi ”SOVELLUS1”, arvo ”MIDPASIAKAS”
    - nimi ”PAATAULU1”, arvo ”MidpAsiakas”
    - nimi ”URL”, arvo ”kehitys.vetokonsultit.fi”

- nimi ”PORT”, arvo ”12721”
- nimi ”YHTEYS”, arvo ”SOCKET”
- nimi ”JARJESTELMA”, arvo ”Prosla”

- MIDlets-välilehdellä (kuva 20b) lisätään uusi *MIDlet* painamalla add-nappia. Annetaan sille nimeksi ”Parametrit”. Ikoni-kohta jätetään tyhjäksi ja luokaksi annetaan ”com.veto.mobile.Parametrit”. Tämä lisää MIDlet-pakettiin Parametrit-sovelluksen.

User Defined -välilehdellä voidaan lisätä muitakin parametreja, joita sovelluskehys käyttää. Kaikki mahdolliset parametrit on lueteltu liitteessä 5.

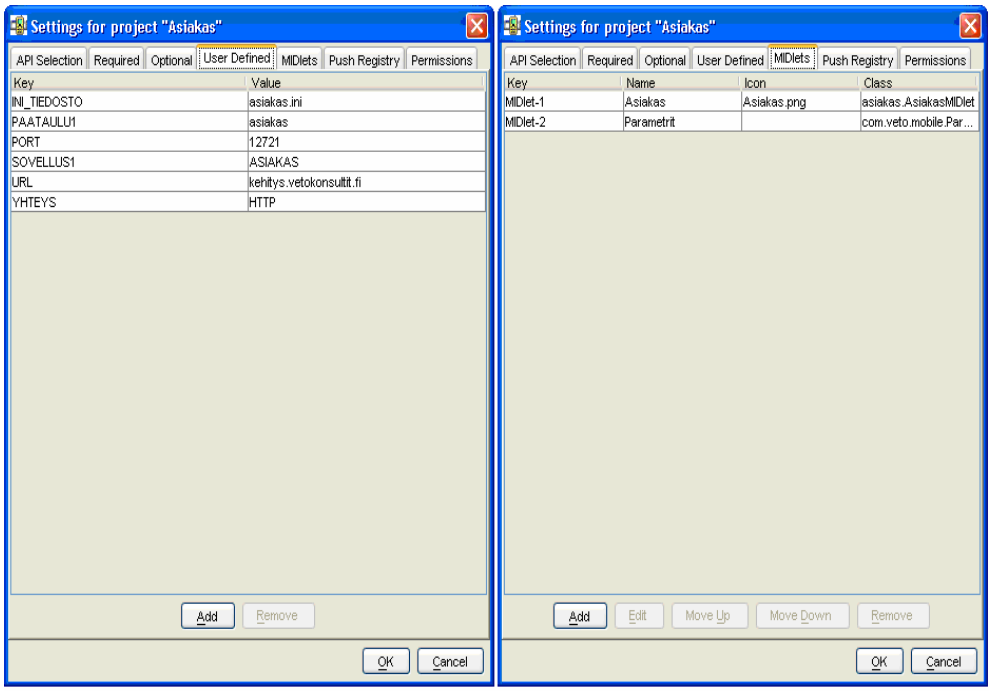
Kuva 19a esittää projektin API Selection -välilehden oikeat arvot esimerkkiprojektia luotaessa. Kuvassa 19b puolestaan esitetään oikeat asetukset Required-välilehdeltä. MIDlet-Jar-Size-ominaisuus asetetaan automaattisesti jakelupakettia luotaessa, joten sen arvoa ei tarvitse asettaa.



19a 19b

Kuva 19a ja 19b. 19a) Esimerkkiprojektin API Selection -välilehden asetukset. 19b) Esimerkkiprojektin Required-välilehden asetukset.

User Defined -välilehden arvot on esitetty kuvassa 20a. Tälle välilehdelle asetetut arvot kirjoitetaan jakelupaketin luonnin yhteydessä MIDlet-paketin JAD-tiedostoon. Kuvan 20b MIDlets-välilehdellä annetaan tiedot kaikista projektiin tulevista *MIDlet*-luokista, joiden halutaan olevan käynnistettävissä kohdelaitteessa. Mikäli projekti sisältää *MIDlet*-luokan, mutta sitä ei ole määritelty täällä, ei se ole suoritettavissa MIDlet-paketin asennuksen jälkeen laitteistossa.



Kuva 20a ja 20b. 20a) Esimerkkiprojektin User Defined -välilehden asetukset. 20b) Esimerkki projektin MIDlets-välilehden asetukset.

Edellä User Defined -välilehdellä (kuva 20a) lisätyistä ominaisuuksista ainoastaan INI_TIEDOSTO on pakollinen, mutta jos muut jätetään pois, ne joudutaan kirjoittamaan sovelluksen asennuksen jälkeen Parametrit-sovelluksella. Kirjoittamalla ominaisuudet saadaan sovellukselle oletusarvot ja säästytään parametrien kirjoittamiselta, mikä koetaan normaalisti matkapuhelimilla hankalaksi verrattuna tietokoneella kirjoittamiseen.

Näiden vaiheiden jälkeen on uusi projekti luotu. Projekti ei tosin sisällä vielä tarvittavia lähdekoodi tiedostoja. WTK luo projektin luonnin yhteydessä alihakemistoonsa apps uuden hakemiston Asiakas. Tämä hakemisto sisältää hakemiston src, jonne tulee kopioida kaikki projektin vaatimat lähde-

koodit. Esimerkissä nämä ovat AsiakasMIDlet.java, AsiakasController.java, AsiakasCommunicationModel.java sekä mobiilikirjaston luokat.

Kun kooditiedostot on kopioitu projektin lähdekoodihakemistoon, se käänetään valitsemalla valikosta ”Project – Build”. Projektin kääntyessä läpi ilman virheilmoituksia, voidaan sovellusta testata valitsemalla ”Project – Run”.

Testauksen jälkeen projektista luodaan jakelupaketti ja kuvaustiedosto. Jakelupaketti tehdään valitsemalla WTK:n valikosta ”Project – Package – Create Package”. Tämä kääntää ja luo tarvittavat JAD- ja JAR-tiedostot projektin bin-hakemistoon. Esimerkki sovelluksen JAR-tiedoston koko on noin 80 kt. Tämä on selvästi enemmän kuin tavoiteltu 64 kt:a, joka on kohdelaitteistossa eli matkapuhelimissa yleinen yläraja tiedoston koolle.

Paketin kokoa saadaan pienennettyä käyttämällä ns. Obfuscatoria. Obfuscator on ohjelma, joka optimoi koodia poistamalla siitä ylimääräistä tietoa kuten esimerkiksi lyhentämällä metodien ja muuttujien nimiä [17, s. 11]. WTK tukee Obfuscatoreita, mutta sellaisen käyttö vaatii kuitenkin sen asentamisen. Ilmaisen Obfuscatorin voi hakea osoitteesta:

http://proguard.sourceforge.net/

Obfuscatorin asennuksen jälkeen voidaan luoda ”Obfuscoitu”-jakelupaketti valitsemalla WTK:n valikosta ”Project – Package – Create Obfuscated Package”. Tämän jälkeen paketin koko on noin 63 kt, eli paketti pieneni noin 20 prosenttia, jolloin päästään alle tavoitellun 64 kt:n.

Kun JAD- ja JAR-tiedostot on luotu, ne laitetaan jakeluun. Tämä tapahtuu kopioimalla ne WWW-palvelimelle hakemistoon, jonne on julkinen pääsy. WWW-palvelin pitää muistaa konfiguroida siten, että se tunnistaa JAD-tiedoston oikean mime-tyypin. Apache-palvelimella linux-ympäristössä tämä tapahtuu lisäämällä tiedostoon /etc/mime.types rivi

text/vnd.sun.j2me.app-descriptor jad

Ilman tätä lisäystä eivät kaikki laitteet välttämättä tunnista JAD-tiedostoa MIDlet-paketin kuvaustiedostoksi, jolloin MIDlet-paketin asennus ei onnistu. Näiden vaiheiden jälkeen kaiken pitäisi olla kunnossa sovelluksen

lataamiseen Over-The-Air (OTA) -menetelmällä, jossa MIDlet-pakkaus asennetaan mobiililaitteeseen Internetin yli.

8 YHTEENVETO

Tässä opinnäytetyössä suunniteltiin ja toteutettiin käyttöliittymä-sovelluskehys J2ME MIDP -ympäristöön. Tavoitteena oli luoda kehys, jota käyttämällä saadaan helpotettua, nopeutettua ja yhdenmukaistettua mobiililaitteille tarkoitettujen VetoBox-sovelluspalvelimen käyttöliittymä-sovellusten tekemistä.

Työssä käytetty teknologia oli uutta sekä työntekijälle että yritykselle, jolle työtä tehtiin. Tästä johtuen työn alku oli pitkälti uuden ympäristön ja sen ominaisuuksien opettelemista. Työtä tehtäessä esille nousi sekä positiivisia että negatiivisia asioita. Positiivisena pidettiin teknologian kohtuullisen helppoa käyttämistä ja oppimista, esimerkiksi verrattuna C++-kielellä tehtäviin Symbian-sovelluksiin. Negatiivisena asiana pidettiin käytetyn MIDP-profiilin rajoitteet laitteistojen ominaisuuksien suhteen kuten puhelin-muistioiden ja laitteen muiden tiedostojen käyttömahdollisuuden puuttuminen.

Sovelluskehyksen arkkitehtuurin perustaksi valittu MVC-malli osoittautui varsin toimivaksi ratkaisuksi. Sen avulla koodi saatiin jaettua selkeisiin osiin, joilla jokaisella on omat vastuualueensa. Toteutuksessa olisi voitu käyttää toisenlaista lähestymistapaa View-kompomenttien luonnissa ja esittämisessä. Nyt View-komponenttien luonti ja oikean datan asettaminen niihin on jätetty kokonaan *VetoController*-luokkaa toteuttavan luokan vastuulle, mikä lisää sovelluskehystä käyttävän kehittäjän tarvitsemaa tietämystä itse sovelluskehyksen toiminnasta.

Toisenlainen ratkaisu olisi voinut olla esimerkiksi sellainen, missä sovelluskehittäjä pyytäisi Controller-tasoa luomaan ja näyttämään uuden käyttöliittymäkomponentin, joka itse hakisi tarvitsemansa datan Model-tasolta. Käyttöliittymäkomponenttien luonti olisi voitu hoitaa tehdasluokalla, joka palauttaisi kontrollerille oikean tyyppisen komponentin, jonka se sitten asettaisi itse ruudulle sovelluskehittäjän tarvitsematta käyttää *DisplayableManager*-luokkaa. Tämä vähentäisi kirjoitettavan koodin määrää

toteuttavassa Controller-luokassa ja tekisi sovelluksen kulun seuraamisen helpommaksi koodista.

Observer-mallin käyttö jäi suunniteltua vähäisemmäksi, mikä puolestaan näkyy Controller-tason vastuun kasvamisena käyttöliittymäkomponenttien päivityksessä. Observer-mallin lisäksi *Pauseablen*-rajapinnan toteutukset eivät ole vielä täysin valmiit kaikissa luokissa, joten aktiivisesta tilasta siirryttäessä passiiviseen tilaan tai päinvastoin sovelluskehyksen toiminta on määrittelemätöntä.

Esimerkkisovelluksen kääntämisen yhteydessä nähty Obfuscatorin käytön tarpeellisuus osoittaa, että luokkakirjastoa pitäisi vielä pienentää. Tällaisenaan se vie liikaa tilaa jättäen sovelluskohtaiselle koodille vähän mahdollisuuksia käyttää kuvia tai muita mahdollisia MIDlet-pakkaukseen tulevia resursseja. Esimerkkisovellus osoitti kuitenkin, että uusien perussovellusten luonti sovelluskehystä ja sen komponentteja käyttäen on suhteellisen nopea prosessi. Esimerkkisovelluksen tuottamiseen meni kokonaisuudessaan aikaa noin 4 tuntia. Tämä sisältää koodauksen, kääntämisen, Obfuscatorin asentamisen ja MIDlet-pakkauksen jakelukuntoon asettamisen. Tässä täytyy kuitenkin huomioida, että sovelluspalvelin oli valmiiksi käyttökunnossa, eikä sitä tarvinnut koodata tai konfiguroida erikseen.

Sovelluskehystä on käytetty esimerkkisovelluksen lisäksi myös metsäaluetietojen etähakuohjelmassa. Sovellusta on kehitetty yhteistyössä Kaakkois-Suomen Metsäkeskuksen kanssa. Sovelluksen on tarkoitus toimia metsäsuunnittelijan apuvälineenä kenttätyötä tehtäessä. Sovelluksella metsäsuunnittelija voi matkapuhelimen avulla hakea erityistietoja läheisistä metsäalueista samalla, kun laatii metsäsuunnitelmaa. Järjestelmä on vielä kehitysasteella, eikä varsinaista kenttätestausta loppukäyttäjillä ole tehty. Metsäkeskuksen tietohallinnon suorittamien alustavien testien tulokset ovat kuitenkin olleet lupaavia.

Kokemusten perusteella näyttäisi siltä, että mobiilisovellukset sisältävät paljon potentiaalia työnteon apuvälineinä. Tulevaisuudessa mobiilisovellukset tulevat olemaan todennäköisesti olennainen osa yritysten tietojärjestelmiä.

VIITELUETTELO

[1] Topley Kim, *J2ME in a Nutshell*. Sebastopol: O'Reilly & Assosiates Inc. 2002.

[2] Ortiz C. Enrique, *A Survey of J2ME Today* [verkkodokumentti]. Sun Microsystems Inc. November 2002. [viitattu 1.6.2004]. Saatavissa: http://developers.sun.com/techtopics/mobility/getstart/articles/survey/

[3] Sun Microsystems Inc. *Introduction to Mobility Java Technology* [verkkodokumentti]. 22.3.2003. [viitattu 23.6.2004]. Saatavissa: http://developers.sun.com/techtopics/mobility/getstart/

[4] Day Bill, *Developing Wireless Applications using the java 2 Platform, Micro Edition* [PDF-dokumentti]. 2001. [viitattu 21.6.2004]. Saatavissa: http://developers.sun.com/techtopics/mobility/getstart/articles/wirelessdev/wirelessdev.pdf

[5] Sun Microsystems Inc. *The Java Community Process(SM) Program - JCP Procedures - JCP 2: Process Document* [verkkodokumentti]. Version 2.6, 9.3.2004. [viitattu 23.6.2004]. Saatavissa: http://jcp.org/en/procedures/jcp2

[6] Kontio Mikko, *Inside Mobiili Java J2ME.* Helsinki: Edita Publishing Oy. 2002.

[7] Sun Microsystems Inc. *J2ME(TM) Connected Limited Device Configuration (CLDC) ("Specification")* [pakattu PDF-dokumentti]. 19.5.2000. [viitattu 11.7.2004]. Saatavissa: http://jcp.org/aboutJava/communityprocess/final/jsr030/index.html

[8] Mahmoud Qusay. *J2ME APIs: Which APIs come from the J2SE Platform?* [verkkodokumentti]. January 2001. [viitattu 11.7.2004] Saatavissa: http://developers.sun.com/techtopics/mobility/midp/articles/api/

[9] Ortiz C. Enrique, *The Generic Connection Framework* [verkkodokumentti]. Sun Microsystems Inc. August 2003. [viitattu 15.7.2004] Saatavissa: http://developers.sun.com/techtopics/mobility/midp/articles/genericframework/

[10] Muchow John W., *Core J2ME Technology & MIDP*. Upper Saddle River: Prentice Hall PTR. 2002.

[11] Sun Microsystems Inc. *Mobile Information Device Profile (JSR-037)* [pakattu PDF-dokumentti]. 15.12.2004. [viitattu 20.7.2004]. Saatavissa: http://jcp.org/aboutJava/communityprocess/final/jsr037/index.html

[12] Puhakka Aapo, *Erilaisten käyttöliittymien yhteisen toiminnallisuuden keskittäminen sovelluspalvelimeen*. Diplomityö. Teknillinen korkeakoulu. Tietotekniikan osasto. 2001. Saatavissa: http://www.iki.fi/aapo/dityo

[13] Erich Gamma – [et al.]. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman Inc. 17th Printing June 1999 (1995).

[14] Brad Appleton, *Patterns and Software: Essential Concepts and Terminology* [verkkodokumentti]. 14.2.2000. [viitattu 4.9.2004] Saatavissa: http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html

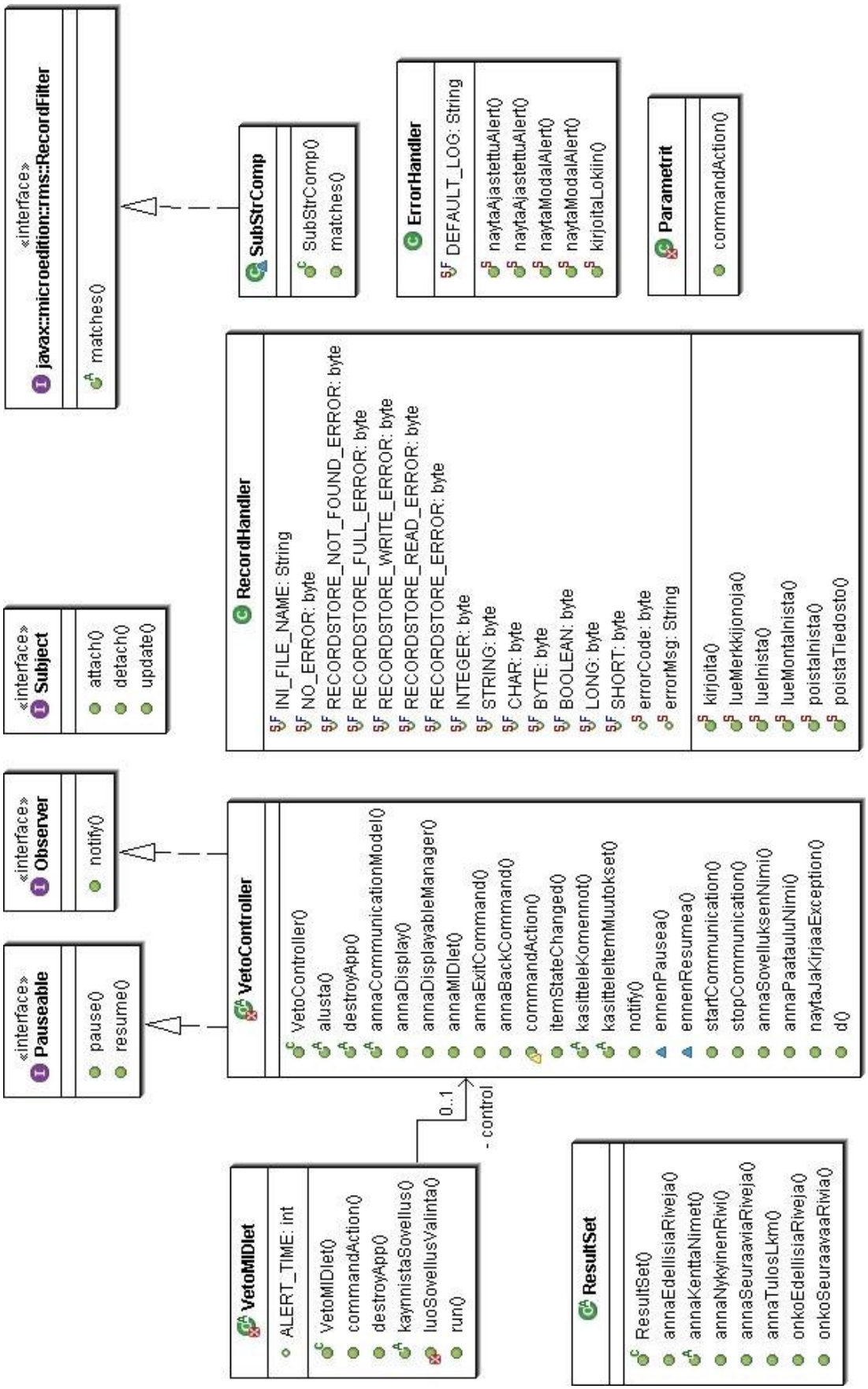
[15] Sun Microsystems Inc. *Java BluePrints Model-View-Controller* [verkkodokumentti]. 2002. [viitattu 6.9.2004] Saatavissa: http://java.sun.com/blueprints/patterns/MVC-detailed.html

[16] John Earles, *Framework Evolution! One Box, Two Box, White Box, Black Box* [verkkodokumentti]. Castek Software Factory Inc. 2000. [viitattu 6.9.2004]. Saatavissa: http://www.cbd-q.com/articles/2000/000401je_frameworks2.asp

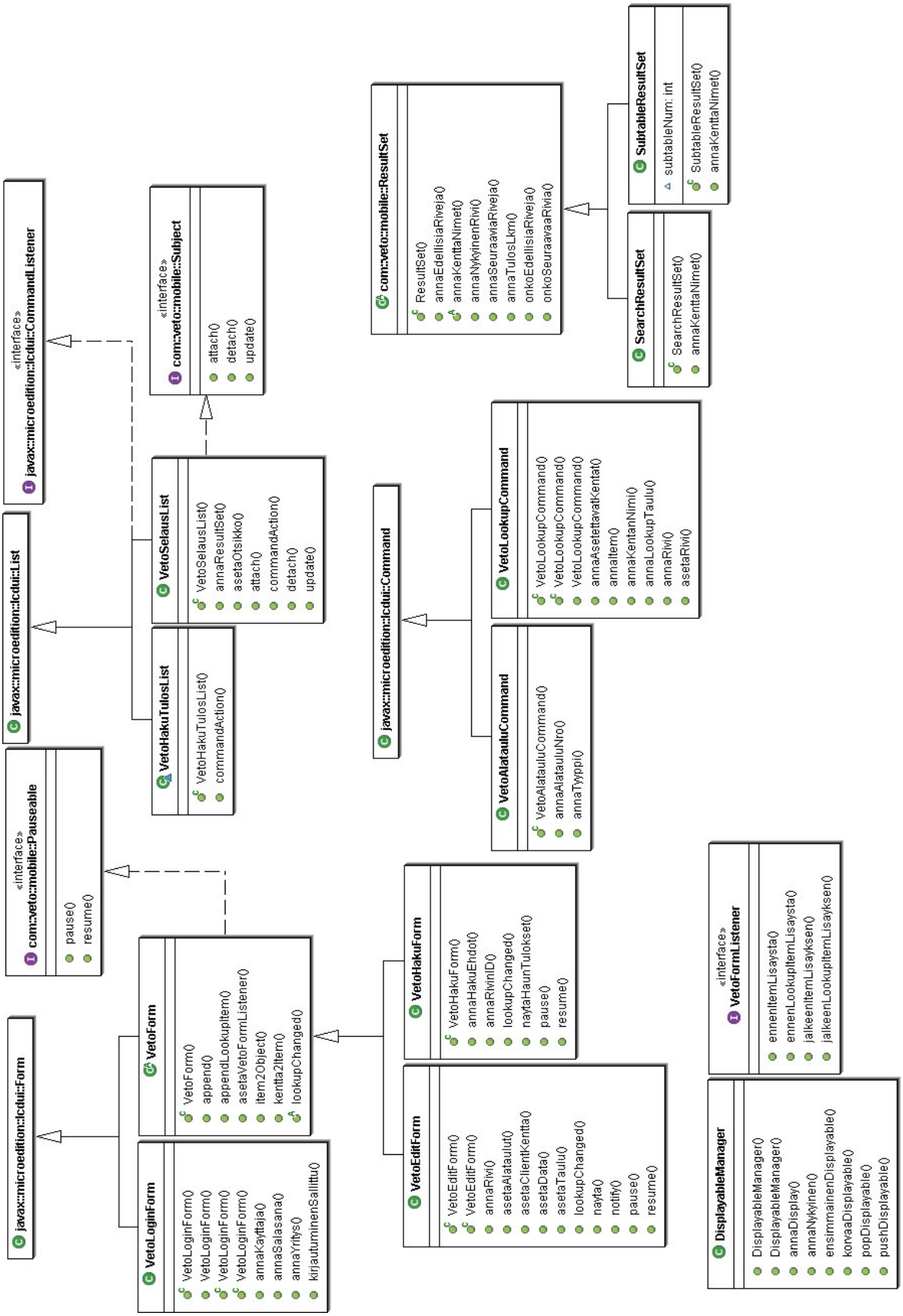
[17] Nokia Corporation. *Efficient MIDP Programming version 1.1* [PDF-dokumentti]. 19.3.2004. [viitattu 27.8.2004]. Saatavissa: http://www.forum.nokia.com/main/1,,21_10,00.html#java

[18] Sun Microsystems Inc. *Core J2EE Patterns - Business Delegate* [verkkodokumentti]. 2002. [viitattu 8.9.2004] Saatavissa: http://java.sun.com/blueprints/corej2eepatterns/Patterns/BusinessDelegate.html

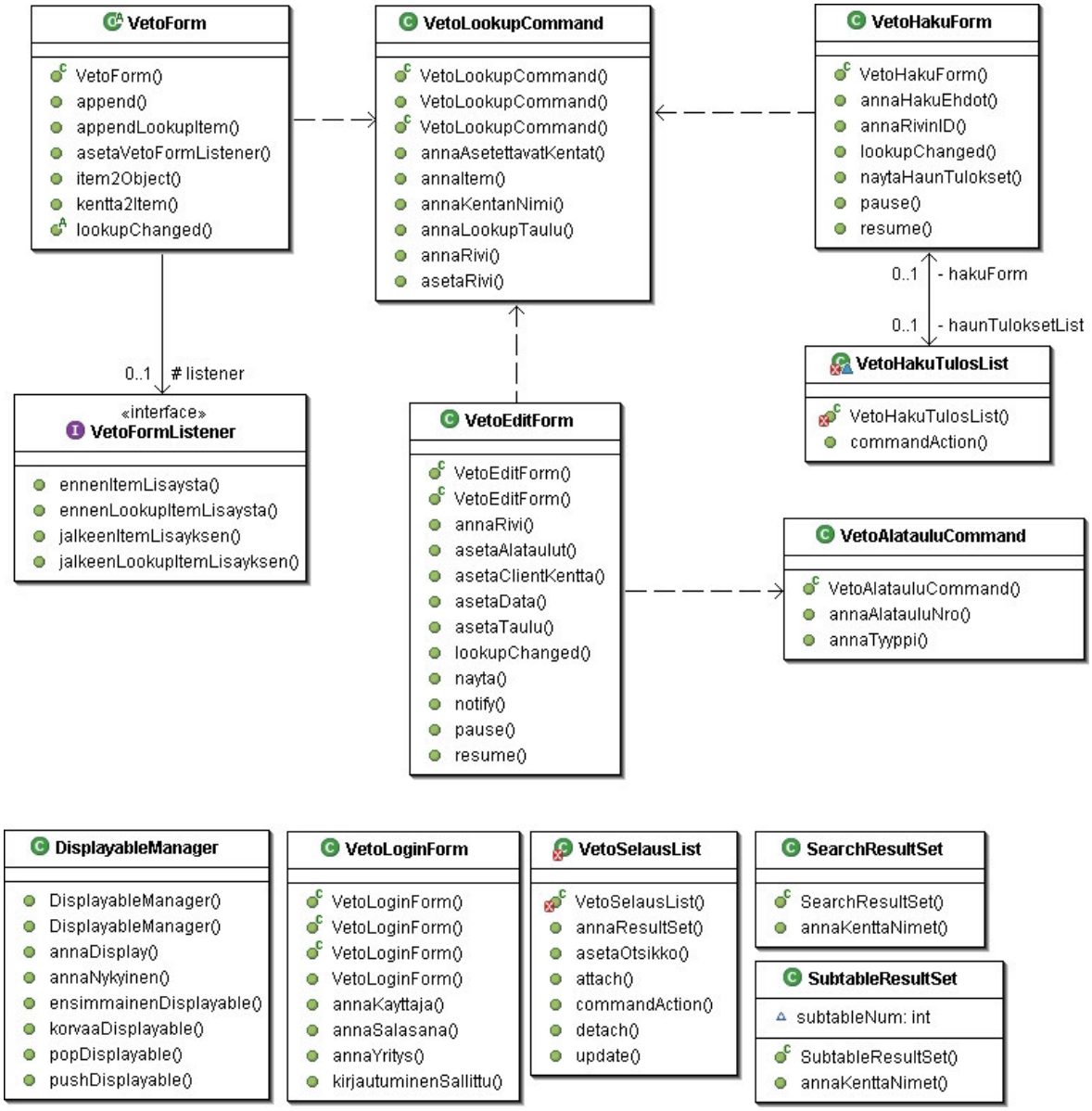
com.veto.mobile-paketin UML-luokkakaavio



com.veto.mobile.ui-paketin periytymishierarkia UML-luokkakaaviona

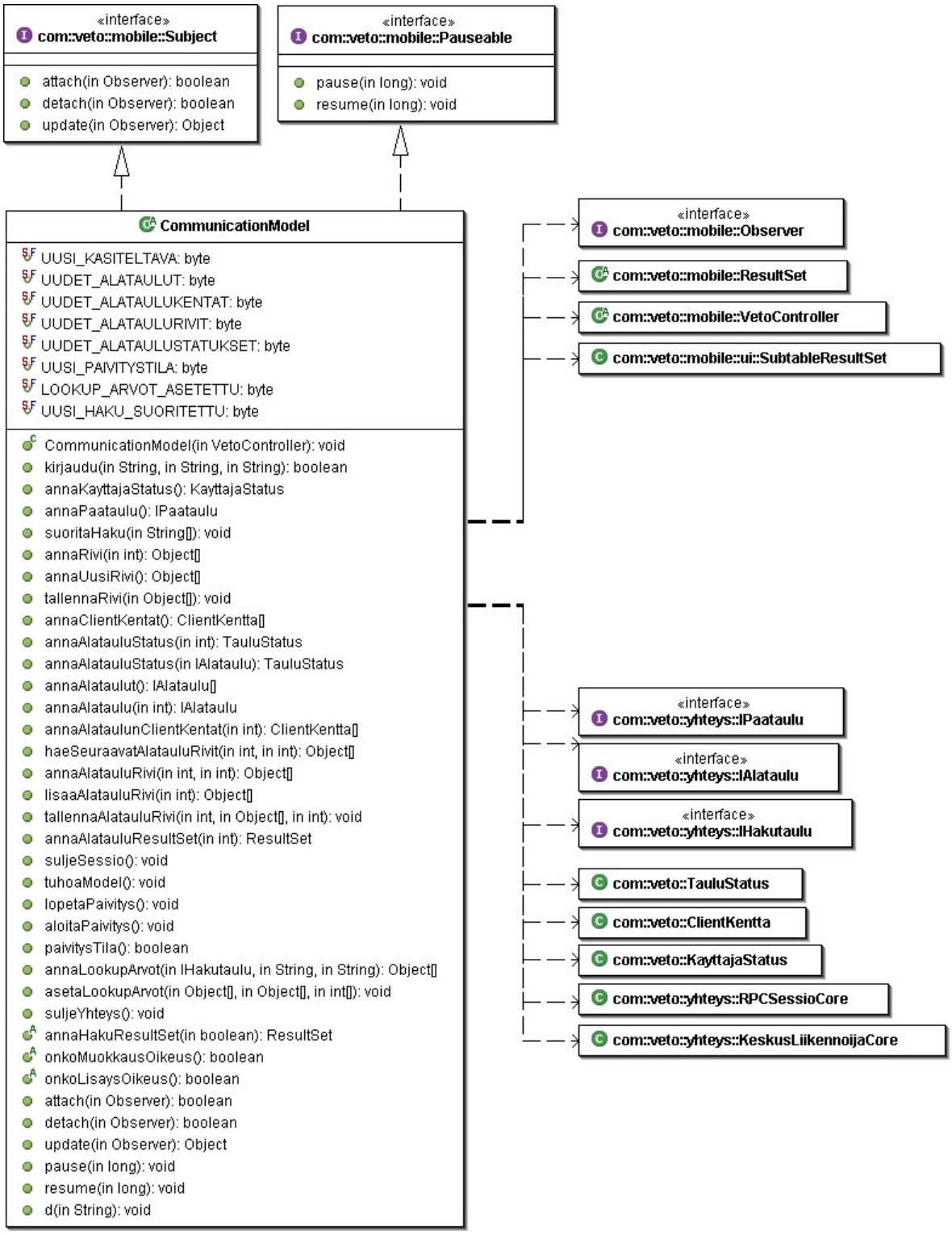


com.veto.mobile.ui-paketin riippuvuussuhteet UML-luokkakaaviona



com.veto.mobile.communication-paketin UML-luokkakaavio

com.veto.mobile.communication-paketin ainoa luokka CommunicationModel ja sen periytymis- ja riippuvuussuhteet UML-kaaviona.



Esimerkkisovelluksen AsiakasMIDlet-luokan koodi

```
package asiakas;

import com.veto.mobile.VetoMIDlet;
import com.veto.mobile.VetoController;
import com.veto.VetoCore;

public class AsiakasMIDlet extends VetoMIDlet {
  /**
   * Tämä metodi palauttaa sovellusvalintalistalta saatavan merkkijono
   * tietojen perusteella oikean tyyppisen VetoController-luokan.
   * Sovelluksen valintalista luodaan MIDlet-paketin parametrien avulla,
   * joten niiden tulee olla asetettuna oikein, jotta tämä metodi
   * "toimisi oikein".
   * @param sSovellus String, käynnistettävän sovelluksen nimi.
   * @param sPaataulu String, käynnistettävän sovelluksen päätaulun nimi.
   * @return VetoController sovelluksen nimeä vastaava controller.
   */
  public VetoController kaynnistaSovellus(String sSovellus,
  String sPaataulu) {
    if(VetoCore.sama(sSovellus, "MIDPASIAKAS"))
      return new AsiakasController(sSovellus, sPaataulu);

    return null;
  }
}
```

Esimerkkisovelluksen AsiakasController-luokan koodi, s. 1 (7)

```
package asiakas;

import com.veto.mobile.VetoController;
import com.veto.mobile.ui.*;
import com.veto.mobile.communication.CommunicationModel;
import com.veto.VetoCore;
import com.veto.SysteemiException;
import javax.microedition.lcdui.Item;
import javax.microedition.lcdui.TextField;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.Displayable;
import com.veto.VetoException;
import javax.microedition.rms.*;
import java.io.*;

public class AsiakasController extends VetoController
  implements VetoFormListener {

  /** login lomake. */
  private VetoLoginForm login;

  /** hakulomake */
  private VetoHakuForm hakulomake;

  /**
   * tuloslomake, jolla näytetään yksittäisen rivin tiedot
   */
  private VetoEditForm tuloslomake;

  /**
   * MIDletin communication model.
   */
  private AsiakasCommunicationModel cModel;

  public AsiakasController(String sSovellusNimi, String sPaatauluNimi) {
    super(sSovellusNimi, sPaatauluNimi);
  }

  /* ############################################## */
  /* #### VetoControllerin abstraktit metodit #### */
  /* ############################################## */

  // ----------------------- a l u s t a -------------------------------
  public void alusta() {
    // luodaan ja näytetään kirjautumislomake, siten ettei kirjaudu-
    // komento ole vielä käytettävissä.
    luoLoginLomake(false);
    try {
      cModel = new AsiakasCommunicationModel(this);
      // Nyt voidaan kirjautua, koska model on luotu ja voidaan ottaa
      // yhteys sovelluspalvelimeen.
      login.kirjautuminenSallittu(true);
    }
    catch(SysteemiException se) {
      naytaJaKirjaaException("Tietoliikenne virhe",
                             "AsiakasController.alusta(): "
                             +se.getMessage());
    }
  }
```

Esimerkkisovelluksen AsiakasController-luokan koodi, s. 2 (7)

```
// ---------- a n n a C o m m u n i c a t i o n M o d e l ------------
/**
 * annaCommunicationModel, palauttaa sovelluksen CommunicationModelin.
 *
 * @return CommunicationModel
 */
public CommunicationModel annaCommunicationModel() {
  return cModel;
}

// -------------------- d e s t r o y A p p ----------------------
/**
 * destroyApp, valmistelee VetoControllerin ja kaikki muut luokat
 * tuhoamista varten.
 *
 * @param unconditional boolean
 */
public void destroyApp(boolean unconditional) {
  try {
    cModel.tuhoaModel();
  }
  catch(SysteemiException se) { /* Ei tehdä mitään. */ }
}

// ---------- k a s i t t e l e I t e m M u u t o k s e t ------------
/**
 * kasitteleItemMuutokset, ei ole käytössä (metodissa ei tehdä mitään).
 *
 * @param item Item
 */
public void kasitteleItemMuutokset(Item item) {
  // Ei käytetä.
}

// ---------- k a s i t t e l e K o m e n n o t ---------------
/**
 * kasitteleKomennot, käsitellään sovelluksen komennot.
 *
 * @param c Command komento, joka “laukaistiin”.
 * @param d Displayable, näyttö jolla komento “laukaistiin”.
 */
public void kasitteleKomennot(Command c, Displayable d) {
  try {
    if(d == login)
      kasitteleLoginKomennot(c);
    else if(d == hakulomake)
      kasitteleHakulomakeKomennot(c);
  }
  catch(VetoException ve) {
    stopCommunication();
    naytaJaKirjaaException("", "AsiakasController.kasitteleKomennot(: "
                           +ve.getMessage());
  }
}
```

Esimerkkisovelluksen AsiakasController-luokan koodi, s. 3 (7)

```
// ---------------------- p a u s e ---------------------------
/**
 * pause, valmistelee sovelluksen paused-tilaan siirtymiseen.
 *
 * @param pauseID long, aika jolloin MIDlet ilmoitti paused-tilaan
 * siirtymisestä.
 */
public void pause(long pauseID) {
  try {
    RecordStore rs =
      RecordStore.openRecordStore(this.getClass().getName()+ pauseID,
                                  true);
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    DataOutputStream dos = new DataOutputStream(baos);

    // tallennetaan tiedot luokista, jotka
    // täytyy palauttaa.
    // login lomaketta ei tarvitse tallentaa...
    // tallennetaan tieto cModelista, jotta se osataan palauttaa.
    if (cModel != null) {
      dos.write("cModel".getBytes());
      dos.flush();
      rs.addRecord(baos.toByteArray(), 0, baos.size());
      baos.reset();
      // Myös sModelin sisäinentila pitää tallentaa.
      cModel.pause(pauseID);
    }

    if (hakulomake != null) {
      dos.write("hakulomake".getBytes());
      dos.flush();
      rs.addRecord(baos.toByteArray(), 0, baos.size());
      baos.reset();
      // Myös hakulomakkeen sisäinen tila pitää tallentaa.
      hakulomake.pause(pauseID);
    }

    baos.close();
    dos.close();
    rs.closeRecordStore();
  }
  catch(RecordStoreException re) {

  }
  catch(IOException ioe) {

  }
}

// ---------------------- r e s u m e ---------------------------
/**
 * resume, Palautetaan Controller luokan tila ennen active-tilaan
 * siirtymistä.
 * @param pauseID long, Sama arvo, kuin edellisellä paused-tilaan
 * siirtymisen yhteydessä saatu long-arvo.
 */
public void resume(long pauseID) {
  try {
    DataInputStream dis;
```

Esimerkkisovelluksen AsiakasController-luokan koodi, s. 4 (7)

```
      RecordStore rs =
        RecordStore.openRecordStore(this.getClass().getName()+pauseID,
                                    true);

      byte[] data;

      // Luetaan loput recordit. Jokainen record kertoo luokan nimen,
      // joka pitää palauttaa.
      for(int i=1; i<rs.getSize(); i++) {
        data = rs.getRecord(i);
        dis = new DataInputStream(new ByteArrayInputStream(data));
        String className = dis.readUTF();

        if("cModel".equals(className)) {
          try {
            cModel = new AsiakasCommunicationModel(this);
            cModel.resume(pauseID);
          }
          catch(SysteemiException se) {
            naytaJaKirjaaException("",
              "AsiakasController.resume("+pauseID+"):"
              +"CommunicationModelin  palautus epäonnistui. "
              +se.getMessage());
          }
        }
        else if("hakulomake".equals(className)) {
          try {
            luoHakulomake();
            hakulomake.resume(pauseID);
          }
          catch(VetoException ve) {
            naytaJaKirjaaException("",
              "AsiakasController.resume("+pauseID+"):"
              +"Hakulomakkeen palautus epäonnistui. "
              +ve.getMessage());
          }
        }

        dis.close();
        dis = null;
        data = null;
        System.gc();
      }
      rs.closeRecordStore();

    // Lopuksi tuhotaan tarpeettomaksi käynyt recodstore.
    RecordStore.deleteRecordStore(this.getClass().getName()+pauseID);
    }
    catch(RecordStoreException re) {
      naytaJaKirjaaException("","AsiakasController.resume("+pauseID+"):"
                             +"Palautus epäonnistui. "
                             +re.getMessage());

    }
    catch(IOException ioe) {
      naytaJaKirjaaException("","AsiakasController.resume("+pauseID+"):"
                             +"Palautus epäonnistui. "
                             +ioe.getMessage()); }
    }
```

Esimerkkisovelluksen AsiakasController-luokan koodi, s. 5 (7)

```
/* ##################################### */
/* #### Komentojen käsittely metodit #### */
/* ##################################### */

// --------- k a s i t t e l e L o g i n K o m e n n o t ------------
/** Käsitellään login-lomakkeen komennot. */
private void kasitteleLoginKomennot(Command c) throws VetoException {
  if(VetoCore.sama("Kirjaudu",c.getLabel())) {
    try {
      startCommunication("Sisäänkirjautumien käynnissä.");
      cModel.kirjaudu(login.annaYritys(),
                  login.annaKayttaja(),
                  login.annaSalasana());
      // Kirjautuminen onnistui, luodaan hakulomake.
      luoHakulomake();
    }
    catch(SysteemiException se) {
      // Ei onnistunut, joten näytetään virheilmoitus ja palataan
      // takaisin login-ruutuun.
      naytaException("Kirjautuminen epäonnistui.",
                     "Tarkista yritystunnus/käyttäjänimi/salasana ja "
                     +"yritä uudelleen.");
      kirjaaException("AsiakasController.kasitteleLoginKomennot():"
                      +"Kirjautumen epäonnistui. Virhe: "
                      +se.getMessage());
      return;
    }
    finally { stopCommunication(); }
  }
}

// ----- k a s i t t e l e H a k u l o m a k e K o m e n n o t ------
/** Käsitellään hakusivun komennot. */
private void kasitteleHakulomakeKomennot(Command c)
throws VetoException {
  // Palataanko takaisin, kirjautumisruutuun?
  if(c == annaBackCommand()) {
    // Palataan, joten suljetaan sessio.
    cModel.suljeSessio();

    // Luodaan ja näytetään login lomake.
    luoLoginLomake(true);
    return; // ei syytä jatkaa pidemmälle tämän metodin suoritusta.
  }

  // Tutkitaan muut komento mahdollisuudet.
  if(VetoCore.sama("Hae",c.getLabel())) {
    try {
      startCommunication("Haku käynnissä.");
      // Tehdään haku...
      cModel.suoritaHaku(hakulomake.annaHakuEhdot());
      // Sitten voidaan vasta näyttää haun tulokset...
      hakulomake.naytaHaunTulokset();
    }
    finally { stopCommunication(); }
  }
  else if(VetoCore.sama("Valitse",c.getLabel()))
    luoTuloslomake();
}
```

Esimerkkisovelluksen AsiakasController-luokan koodi, s. 6 (7)

```
/* ################################### */
/* #### Lomakkeiden luonti metodit #### */
/* ################################### */

// ---------------- l u o L o g i n L o m a k e --------------------
/**
 * luodaan ja näytetään kirjautumis lomake.
 */
private void luoLoginLomake(boolean bKirjSallittu) {
  // luodaan kirjautumislomake ja asetetaan se näytölle, käyttämällä
  // sovelluskehyksen DisplayableManageria.
  login = new VetoLoginForm(this, "Kirjautuminen", bKirjSallittu);
  dManager.pushDisplayable(login);
}

// ----------------- l u o H a k u l o m a k e --------------------
/**
 * Luo ja näytää hakulomakkeen.
 */
private void luoHakulomake() throws VetoException {
  hakulomake = new VetoHakuForm(this, "Asiakashaku");
  // Asetetaan hakulomake login lomakkeen tilalle.
  dManager.korvaaDisplayable(hakulomake);
  // "Poistetaan" login lomake muistista.
  login = null;
}

// ---------------- l u o T u l o s l om a k e ----------------------
/**
 * Luo ja näytää "Tietosivun".
 */
private void luoTuloslomake() throws VetoException {
  try {
    startCommunication("Haetaan asiakkaan tiedot.");
    tuloslomake = new VetoEditForm(this, "Asiakas tiedot");
    // Kysytään rivin ID, jonka perusteella heataan rivin tarkemmat
    // tiedot.
    Object[] oRivi = cModel.annaRivi(hakulomake.annaRivinID());
    // annetaan rivin tiedot tuloslomakkeelle.
    tuloslomake.asetaData(oRivi);
    // annetaan rivin metadata tuloslomakkeelle.
    tuloslomake.asetaClientKentta(cModel.annaClientKentat());
    // Koska halutaan näyttää/muokata/lisätä alataulujen rivejä,
    // annetaan viitteet alatauluihin tuloslomakkeelle.
    tuloslomake.asetaAlataulut(cModel.annaAlataulut());
    // Asetetaan "takaisin"-komento.
    tuloslomake.addCommand(cmdBack);

    tuloslomake.asetaVetoFormListener(this);
    // Kun kaikki tarvittavat tiedot on annettu, voidaan käskeä
    // lomaketta "muodostamaan" itsensä.
    tuloslomake.nayta();
    // Näytetään lomake.
    dManager.pushDisplayable(tuloslomake);
  }
  finally { stopCommunication(); }
}
```

Esimerkkisovelluksen AsiakasController-luokan koodi, s. 7 (7)

```
  /* ############################################## */
  /* #### VetoFormListener-rajapinnan metodit #### */
  /* ############################################## */

  // -------------  e n n e n I t e m L i s a y s t a  ----------------
  /**
   * ennenItemLisaysta, Katkaistaan "Nimi"-niminen kenttä 10 merkin
   * mittaisesksi, aina kun sellainen asetetaan tuloslomakkeelle.
   *
   * @param item Item, lomakkeelle asetettava kenttä.
   */
  public void ennenItemLisaysta(Item item) {
    if( VetoCore.sama(item.getLabel(), "Nimi")) {
      String nimi = ((TextField) item).getString();
      if(nimi.length() > 10)
        ((TextField)item).setString(nimi.substring(0,9));
    }
  }

  // ------------ e n n e n L o o k u p I t e m L i s a y s t a ---------
  /** Ei toteuteta */
  public void ennenLookupItemLisaysta(VetoLookupCommand c, Item item) {
  }

  // ------------- j a l k e e n I t e m L i s a y k s e n --------------
  /** Ei toteuteta */
  public void jalkeenItemLisayksen(Item item, int indeksi) {
  }

  // ------- j a l k e e n L o o k u p I t e m L i s a y k s e n --------
  /** Ei toteuteta */
  public void jalkeenLookupItemLisayksen(VetoLookupCommand c, Item item,
                                         int indeksi) {
  }
}
```

Esimerkkisovelluksen AsiakasCommunicationModel-luokan koodi, s. 1 (3)

```
package asiakas;

import com.veto.mobile.communication.CommunicationModel;
import com.veto.mobile.ResultSet;
import com.veto.mobile.VetoController;
import com.veto.SysteemiException;
import com.veto.TauluStatus;
import com.veto.KayttajaStatus;
import com.veto.VetoException;
import com.veto.mobile.ui.SearchResultSet;

public class AsiakasCommunicationModel extends CommunicationModel {

  /** lippu, joka kertoo onko käyttöoikeudet haettu */
  private boolean bOikeudetHaettu;
  /** Kirjautuneen käyttäjän editointi oikeus. */
  private boolean bMuokkausOikeus;
  /** Kirjautuneen käyttäjän lisäys oikeus. */
  private boolean bLisaysOikeus;
  /** Haun tulokset */
  private ResultSet hakuRS;

  public AsiakasCommunicationModel(VetoController control)
    throws SysteemiException {
    super(control);
    bOikeudetHaettu = false;
  }

  /* ################################################ */
  /* #### CommunicatioModel:n abstraktit metodit #### */
  /* ################################################ */

  // --------------- a n n a H a k u R e s u l t S e t ------------------
  /**
   * annaHakuResultSet luo ja palauttaa SearchResultSet:n.
   *
   * @param buffered Boolean parametrin arvoa ei käytetä.
   * @return ResultSet, SearchResultSet-luokan ilmentymä.
   */
  public ResultSet annaHakuResultSet(boolean buffered)
  throws VetoException {
    // onko hakuehtoja muutettu. Jos ei ole, voidaan palauttaa
    // edellisellä kerralla luotu searchresultset. (vältytään
    // mahdollisesti turhalta tietoliikenteeltä, mikä on hidasta ja
    // maksaa).
    if(bHakuMuuttunut) {
      if(paataulu == null)
        avaaPaataulu();
      if(paaKentat == null)
        haeClientKentta();

      hakuRS = new SearchResultSet(paataulu, paaKentat);
      // Asetetaan lippu, joka kertoo, että ollaan luotu searchResultSet
      // uusimilla hakuehdoilla.
      bHakuMuuttunut = false;
    }
    return hakuRS;
  }
```

Esimerkkisovelluksen AsiakasCommunicationModel-luokan koodi, s. 2 (3)

```
// -------------- o n k o L i s a y s O i k e u s -------------------
/**
 * onkoLisaysOikeus ilmoittaa onko käyttäjällä oikeus lisätä tietueita
 * sovelluksella.
 * @return boolean, true mikäli on tietueiden lisäysoikeus, muuten
 * false
 */
public boolean onkoLisaysOikeus() {
  if(!bOikeudetHaettu)
    haeOikeudet();
  return bLisaysOikeus;
}

// --------------- o n k o M u o k k a u s O i k e u s --------------
/**
 * onkoMuokkausOikeus ilmoittaa onko käyttäjällä oikeus muokata
 * tietueita.
 *
 * @return boolean true mikäli käyttäjä saa muokata tietueita, muuten
 * false.
 */
public boolean onkoMuokkausOikeus() {
  if(!bOikeudetHaettu)
    haeOikeudet();
  return bMuokkausOikeus;
}

/* ######################################## */
/* #### Pauseable -rajapinnan metodit #### */
/* ######################################## */

// ----------------------- p a u s e ----------------------------
/** valmistelee modelin paused-tilaan siirtymiseen. */
public void pause(long pauseID) {
  super.pause(pauseID);
}

// --------------------- r e s u m e --------------------------
/** valmistelee modelin active-tilaan siirtymiseen. */
public void resume(long pauseID) {
  super.resume(pauseID);
}


/* ################################## */
/* #### yksityiset apu metodit #### */
/* ################################## */

// ---------------- h a e O i k e u d e t ----------------------
/**
 * Hakee palvelimelta käyttöoikeustasot ja päättelee lisäys- ja
 * muokkaus-oikeudet niiden avulla.
 */
private void haeOikeudet() {
  try {
    // Käyttöoikeus liput oikeiksi.
    TauluStatus ts = paataulu.annaStatus();
    KayttajaStatus ks = annaKayttajaStatus();
```

Esimerkkisovelluksen AsiakasCommunicationModel-luokan koodi, s. 3 (3)

```
      bMuokkausOikeus = (ts.iPaivitysoikeus >= ks.iOikeus);
      bLisaysOikeus = (ts.iLisaysoikeus >= ks.iOikeus);
      ts = null;
    }
    catch(SysteemiException se) {
      bMuokkausOikeus = false;
      bLisaysOikeus = false;
    }
    finally {
      bOikeudetHaettu = true;
    }
  }
}
```

SOVELLUSKEHYKSEN PARAMETRIT

Käyttöliittymäsovelluskehystä käyttävät sovellukset (MIDlet:it) tarvitsevat yhden pakollisen parametrin, joka asetetaan MIDlet-jakelupakettiin. Tämä parametri kertoo sovelluksille, minkä nimisestä tiedostosta muita parametreja yritetään lukea. Parametrin nimen täytyy olla INI_TIEDOSTO ja arvo voi olla mitä tahansa tekstiä. Parametri asetetaan joko sovelluksen JAD-tiedostoon tai sovelluksen MANIFEST.MF tiedostoon, joka pakataan JAR-tiedoston ”sisään”. Parametri on molemmissa tiedostoissa vain tekstirivi, joka on muotoa:
PARAMETRIN_NIMI: PARAMETRIN_ARVO

Parametrien käsittely (lukeminen) tapahtuu sovelluksissa normaalisti seuraavasti:

1. Yritetään lukea parametrin arvo ini-tiedostosta (oikeastaan MIDP:ssä ei ole tiedostoja vaan ns. RecordStoreja, mutta puhun tiedostoista, koska se on helpompaa).
2. Siinä tapauksessa, jos ini-tiedostosta ei löydy annetulla parametrin nimellä arvoa, yritetään samalla parametrin nimellä hakea arvoa MIDletin propertyistä (ts. JAD- tai MANIFEST-tiedostoista).
3. Jos parametria ei löydy MIDletinkään propertyistä, palautetaan oletusarvo, joka on annettu parametrinlukumetodille. Jos oletus arvo on erisuuri kuin null, parametrin nimi- ja oletusarvo tallennetaan ini-tiedostoon. (Ideana on se, että käyttäjän ei tarvitse kirjoittaa puhelimella niin paljoa, koska parametrin nimi tallentuu automaattisesti ini-tiedostoon. Käyttäjän tarvitsee siis vain muuttaa parametrin arvo sopivaksi.)

Käyttöliittymäsovelluskehyksessä käytettävät pakolliset parametrit sekä niiden selitykset:

Parametri	**Esimerkki**	**Selite**
URL	194.29.195.202	palvelimen url-osoite ilman protokolla osaa (siis ei http:// tai socket:// -alkua) eikä portti-numeroa.
PORT	2244	VetoBox-palvelimen VetoRPC-portti.
PAATAULUn	Kuvio	Sovelluksen päätaulun nimi (Ohjelmoija-nimi). n-kirjain korvataan juoksevalla kokonaisluvulla 1, 2, 3, jne. ja sen tulee vastata sovelluksen nimen numeroa.
JARJESTELMA	Metsa	Järjestelmän nimi, johon sovellukset kuuluvat.
SOVELLUSn	KUVIO	Sovelluksen nimi. Jokaista järjestelmässä olevaa (MIDP) sovellusta varten luodaan yksi. (nimistä muodostetaan sovellusten valintalista). n-kirjain korvataan juoksevalla kokonaisluvulla 1, 2 , 3, jne. ja sen tulee vastata sovelluksen päätaulun numeroa.

Optionaalisia parametreja ovat:

Parametri	**Esimerkki**	**Selite**
LOGINNAME	TEST	Login-lomakkeelle haettava oletus käyttäjänimi.
LOGINCOMPANY	TEST	Login-lomakkeelle haettava oletus yritystunnus. Jos tämä parametri asetetaan, login-lomakkeella ei näytettä yritystunnus kenttää.
YHTEYS	HTTP	Arvot voivat olla joko HTTP tai SOCKET. Mikäli parametria ei aseteta, oletetaan yhteyden olevan HTTP.
ALERTTIME	4000	Aika millisekunteina, kuinka kauan ajastetut varoitus lomakkeet (Alertit) näkyvät käyttäjälle.
MAXLOOKUP	10	Montako riviä haetaan ja näytettään maksimissaan lookup-hakua tehdessä.