LAPPEENRANNAN TEKNILLINEN KORKEAKOULU **21.9.1999**
TIETOTEKNIIKAN OSASTO / TIETOLIIKENNETEKNIIKKA

1715 TIETOTEKNIIKAN ERIKOISTYÖT

Loppuraportti

KOODAUSFUNKTIOIDEN OLIOPOHJAINEN TOTEUTUS ATM-SIGNALOINNISSA

Timo Pärnänen 4859 TiTe N

Työn tarkastaja: prof. Jorma Jormakka
Työn ohjaaja: Vesa-Matti Puro

SISÄLLYSLUETTELO

1 JOHDANTO 3
2 TYÖN MÄÄRITTELY 4
2.1 TOVE-projekti 4
2.2 Tavoite 5
3 KOODAUSFUNKTIOT 6
3.1 OSI-malli 6
3.2 ATM-signalointi 7
4 OLIOTEKNIIKAT 9
4.1 Suunnittelumallit 9
4.2 UML 10
5 TOTEUTUS 12
5.1 Bittien käsittely 14
5.2 Sanomat 15
6 TYÖN TULOKSET 17
6.1 Jatkokehitys 18
7 YHTEENVETO 20

Liite 1. BitHandler-luokan C++-otsikkotiedosto.

LYHENTEET

ATM	Asynchronous Transfer Mode
BCD	Binary Coded Decimal
BISUP	Broadband ISDN User Part
CVOPS	C-based Virtual Operations System
DECT	Digital Enhanced Cordless Telecommunications
DSS2	Digital Subsciber Signaling System No. 2
FSR	Frame Synchronized Ring
ISDN	Integrated Services Digital Network
ITU-T	International Telecommunication Union – Telecommunication standardization sector
OMG	Object Management Group
OMT	Object Modeling Technique
OOSE	Object-Oriented Software Engineering
OSI	Open Systems Interconnect
OVOPS	Object-oriented Virtual Operations System
PCI	Protocol Control Information
PDU	Protocol Data Unit
SAP	Service Access Point
SDU	Service Data Unit
TEKES	Teknologian kehittämiskeskus
TKK	Teknillinen korkeakoulu
TOVE	Toimialaverkot
UML	Unified Modeling Language
UNI	User-Network Interface
VTT	Valtion teknillinen tutkimuskeskus

1 JOHDANTO

Tämä erikoistyö on tehty Teknillisen korkeakoulun (TKK) TOVE-projektissa (Toimialaverkot). Projektissa tutkittiin uusia oliomenetelmiä ja niitä käyttäen toteutettiin ATM-kytkimen (Asynchronous Transfer Mode) ohjausohjelmisto. Ohjausohjelmiston osana toteutettiin ITU-T:n määritysten mukaisia ATM-signalointiprotokollia.

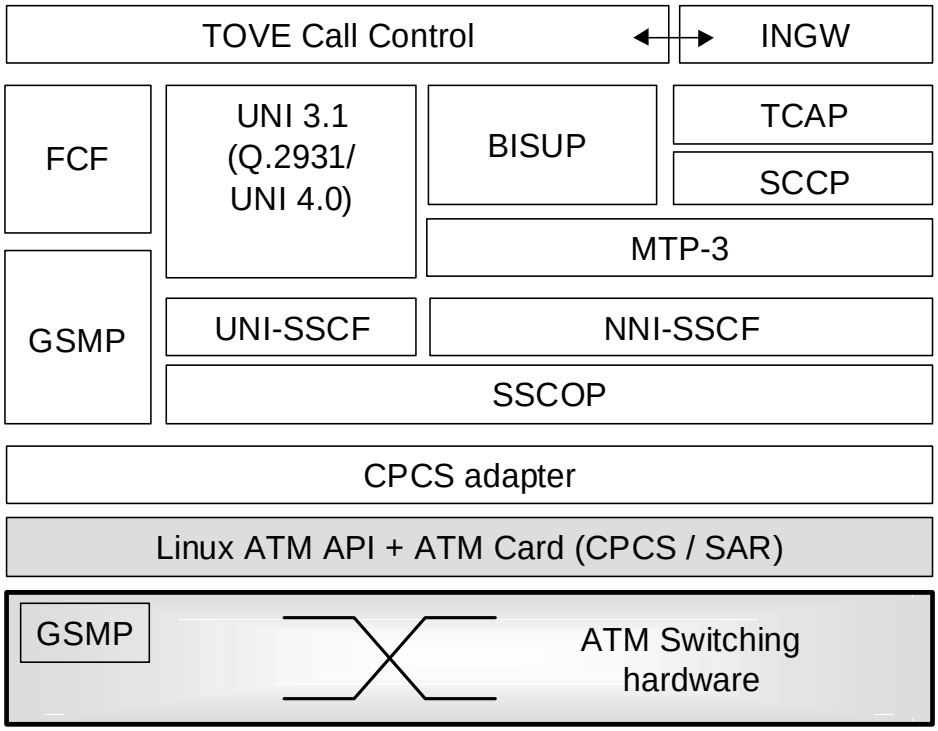
Oliopohjaisten suunnittelumenetelmien ja ohjelmointikielten käyttö on yleistynyt 90-luvulla. Oliomenetelmien avulla pyritään parantamaan koodin uudelleenkäytettävyyttä. Tietoliikenneprotokollien toteutukseen Suomessa on käytetty vahvasti CVOPS-ympäristöä (C-based Virtual Operations System), mutta oliomenetelmät ovat syrjäyttämässä perinteiset proseduraaliset ohjelmointikielet.

Koodausfunktiot ovat yksi keskeisimpiä ongelma-alueita tietoliikenne-protokollien toteutuksessa. Tässä työssä tutkittiin oliomenetelmiä ja niitä hyödyntäen toteutettiin ATM-signalointisanomien koodausfunktiota. Toteutuksessa haluttiin piilottaa virhealttiit bittioperaatiot selkeän ja yksinkertaisen rajapinnan taakse. Toteutuksesta haluttiin helposti luettavaa, jotta sitä pystyttäisiin hyödyntämään projektin edetessä uusien protokollien toteutuksessa myös mahdollisten tulevien työntekijöiden toimesta.

2 TYÖN MÄÄRITTELY

2.1 TOVE-projekti

TOVE-projekti on vuosina 1996-1998 toteutettu TEKES:n (Teknologian kehittämiskeskus) ja yrityspartnereiden rahoittama tutkimushanke, jossa kehitettiin tietoliikenneratkaisuja multimediaan pohjautuvaa yritysviestintää varten. Projektissa suunniteltiin ja pilotoitiin ATM-kytkimen ohjausohjelmistoa. Ohjelmisto rakentuu OVOPS++-sovelluskehyksen (Object-oriented Virtual Operations System) päälle rakennetuista ITU-T:n määrittämistä laajakaista-ISDN-verkon (Integrated Services Digital Network) merkinantoprotokollista, kytkentäkentän ohjauksesta sekä älyverkko-osasta. Kytkentälaitteistona käytettiin VTT:n (Valtion teknillinen tutkimuskeskus) tarjoamaa FSR-kytkintä (Frame Synchronized Ring).



Kuva 1. TOVE-projektin protokollapinot.

2.2 Tavoite

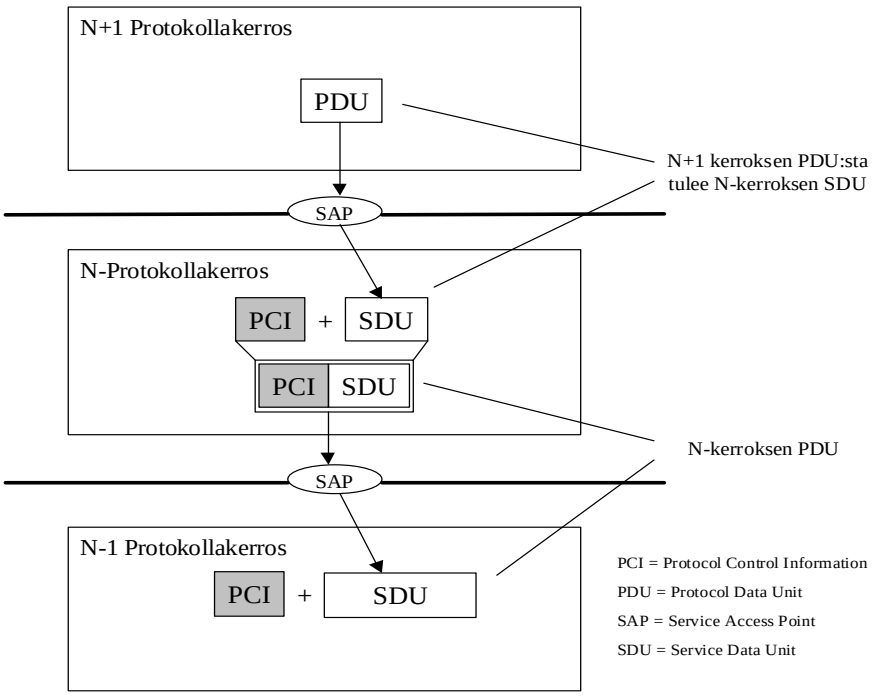
Tämän erikoistyön tarkoituksena oli tietoliikennesanomien koodaus- ja dekoodausfunktioiden oliopohjaisen toteutuksen tutkiminen ja syntyvän ratkaisun soveltaminen puhelunohjausprotokollien toteutuksessa. Tavoitteena oli löytää yleinen ja tehokas tapa toteuttaa koodausfunktiot oliopohjaisesti. Käyttäjä- ja verkkosignaloinnissa esiintyviä yhtäläisyyksiä etsimällä pyrittiin koodin uudelleenkäytettävyyteen eri signalointiprotokollien koodausfunktioiden toteutuksessa. Tehtävänä oli syntyvää koodaus-sovelluskehystä hyväksi käyttäen toteuttaa aluksi Q.2931-koodausfunktiot sekä OVOPS++-sovelluskehykseen pohjautuen Q.2931-signalointiprotokollan toiminnallisuudesta prototyyppi.

3 KOODAUSFUNKTIOT

Tietoliikennesanomien koodausfunktiot ovat protokollan tilakoneen ohella yksi työläimmistä tehtävistä protokollatoteutuksessa. Dekoodauksen tehtävänä on purkaa bittivirrasta protokollamääritysten mukaiset erityyppiset kentät protokollan käsiteltäväksi. Koodaus on käänteinen operaatio dekoodaukselle.

3.1 OSI-malli

Protokollapinon jokainen kerros koodaa lähtevään sanomaan oman otsikkotietonsa eli protokollan ohjaustiedon (PCI, Protocol Control Information). Tulevalle sanomalle suoritetaan vastaavasti otsikon dekoodaus. Ylemmän kerroksen palvelutietoyksikkö eli SDU (Service Data Unit) lisättynä protokollan otsikkotiedolla muodostaa kyseisen kerroksen protokollatietoyksikön eli PDU:n (Protocol Data Unit).



Kuva 2. Sanoman muodostaminen OSI-mallin mukaisesti.

Protokollakerrosten välillä tietoyksiköt siirretään primitiivien kuljettamana palvelurajapinnan kautta. Käyttäjien välinen (vastinoliot) informaatio pakataan protokollatietoyksiköihin eli PDU:hin. Kuva 2 havainnollistaa sanoman muodostumista ja termistöä OSI-mallin (Open Systems Interconnect) mukaisesti /1/.

3.2 ATM-signalointi

ATM-signaloinnissa signalointipinon alin kerros toimii laitesovittimena (adapter), jonka tehtävänä on lukea ATM-sokettia. Sokettirajapinnasta luetusta bittivirrasta muodostetaan kehys (frame). Se on merkkijono, jonka alkioita käsitellään oktetin eli kahdeksan bitin (merkin) kokoisina palasina. Oktettiin voidaan koodata esimerkiksi kokonaisluku tai merkki.

Laiterajapinnassa muodostettu sanoma kulkee protokollapinoa ylöspäin, jokaisen kerroksen purkaessa (dekoodatessa) oman informaationsa sanomasta. Protokollapinon alimpien kerrosten signalointisanomat ovat rakenteeltaan yksinkertaisia, sillä sanomien kentät ovat pituudeltaan oktetin monikertoja. Näiden kerrosten koodausfunktiot ovat yksinkertaisia, ja niiden toteutus on hyvin suoraviivaista.

Protokollapinon ylimmällä kerroksella tilanne on hankalampi. Ylin kerros käsittelee käyttäjätietoa ja purkaa kehyksen viimeisetkin tiedot. ATM-signaloinnissa (Q.2931, UNI 3.x/4.0, BISUP) sanomat muodostuvat informaatioelementeistä, jotka puolestaan sisältävät useita alle oktetin mittaisista kenttiä. Informaatioelementeissä on kokonaisluku- ja totuusarvokenttien (boolean) lisäksi myös monimutkaisempia rakenteita, kuten taulukkomuotoisia BCD-koodattuja (Binary Coded Decimal) numeroita (osoitteita) sekä optionaalisia osia varten mahdollisuus informaatioelementtien laajennukseen (extension).

Koodausfunktioiden tehtäviin kuuluu oleellisesti myös koodausvirheiden havaitseminen ja niiden informointi protokollalle. ATM-signaloinnissa vastaanottaja informoi lähettäjää STATUS-sanoman avulla virheistä. Koodausfunktioiden havaitsemia virheitä ovat mm. pakollisen informaatioelementin puuttuminen ja informaatioelementin virheellinen sisältö (eli ei noudata protokollan koodaussääntöjä) /2/.

4 OLIOTEKNIIKAT

Perinteisten proseduraalisten ohjelmointikielten (esim. Cobol, C) rinnalle kehitettiin uuden ajattelutavan mukaiset oliotekniikat, jossa ongelmanratkaisu perustuu reaalimaailman hahmottamiseen olioina. Tutkimuslaboratorioissa kehiteltiin olioperusteisia ohjelmointikieliä, ja 80-luvulla syntyivät mm. Smalltalk ja C++.

Oliopohjaisten ohjelmointikielten yleistyessä ohjelmistosuunnittelun ja koko prosessin eri vaiheisiin on alettu kiinnittää entistä enemmän huomiota. Huolellisen suunnittelun tärkeys on huomioitu ja sen apuvälineiksi on kehitetty erilaisia menetelmiä ja tekniikoita. Tämän erikoistyön kannalta oleellisia suunnittelun apuvälineitä ovat suunnittelumallit (design patterns) ja UML-mallinnuskieli (Unified Modeling Language).

4.1 Suunnittelumallit

Suunnittelumallit ovat yksinkertaisia ja elegantteja ratkaisuja spesifisiin ongelmiin olioperusteisessa ohjelmistosuunnittelussa. Ne ovat alun perin neljän herran (Gamma, Helm, Johnson ja Vlissides) listaamia ja vuonna 1995 kirjassa Design Pattern julkaistuja hyvin dokumentoituja yleisiä ratkaisuja, joita on helppo soveltaa ongelmien ratkaisuihin ohjelmistosuunnittelussa.

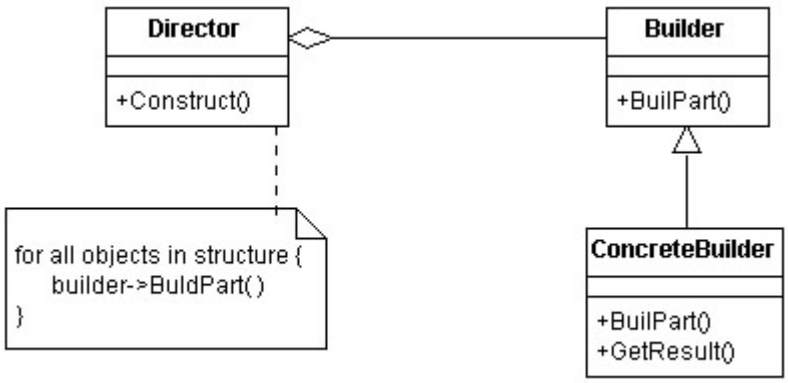
Yleisesti suunnittelumallilla on neljä rakenneosaa: nimi, ongelma, ratkaisu ja tulos. Suunnittelumallin nimi kuvaa ongelman, ratkaisun ja tuloksen sanalla tai kahdella. Se helpottaa dokumentoinnissa ja ryhmätyöskentelyssä kertomaan, mistä suunnittelumallista on kyse. Ongelma kertoo, milloin ko. suunnittelumallia voi käyttää. Ratkaisu kuvaa mallin elementit (luokat tai oliot), niiden väliset suhteet ja yhteistoiminnan. Tulokset kertovat suunnittelumallin käytön seuraamukset, kuten vaikutukset systeemin joustavuuteen, laajennettavuuteen ja siirrettävyyteen /3/.

4.2 UML

UML syntyi, kun kolme olioperusteisen mallintamisen suurta vaikuttajaa yhdistivät menetelmänsä: Jim Rumbaughin OMT (Object Modeling Technique), Ivar Jakobssonin OOSE (Object-Oriented Software Engineering) ja Grady Boochin BOOCH. Tavoitteena heillä oli luoda yleinen standardoitu mallinnuskieli, joka rohkaisee ja helpottaa suunnittelijoita mallintamaan ohjelmistonsa kunnolla ennen toteutusta /4/. OMG (Object Management Group) on standardoinnut UML-kielen.

UML on nimenomaan mallinnuskieli, joka määrittelee mallinnukseen (pääasiassa) graafiset esitysmuodot. UML:ää käytetään ohjelmistojen ja liiketoimintaprosessien visualisointiin, määrittelyyn, toteutukseen ja dokumentointiin. UML tarjoaa tuen kaikkiin kehitysvaiheisiin, jotka kuvataan erilaisilla kaavioilla /4/.

Tässä erikoistyössä UML:ää käytettiin ainoastaan dokumentointiin, ja sen osalta keskeisimmät kaaviotyypit olivat luokkakaavio (class diagram) ja sekvenssikaavio (sequence diagram).

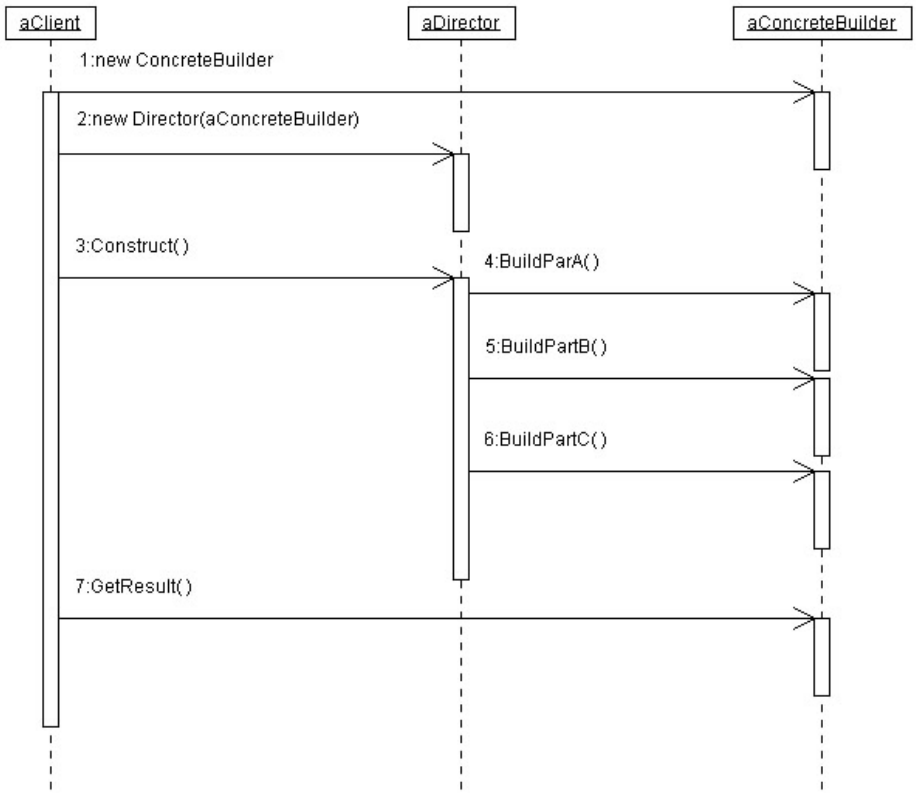


Kuva 3. Luokkakaavio /3/.

Luokkakaavio on olioperusteisen suunnittelun keskeisin tekniikka. Sillä kuvataan systeemin staattinen rakenne. Luokkakaaviolla esitetään systeemin olioiden tyypit ja niiden väliset staattiset suhteet. Kaaviosta selviää luokkien

perintähierarkia ja luokkien väliset kytkentätyypit. Luokkakaaviossa voidaan esittää myös luokan attribuutit, sekä operaatiot (metodit) /4/.

Sekvenssikaavio (sequence diagram) kuvaa olioiden väliset operaatiot ja niiden vastaukset aikajärjestyksessä /4/.

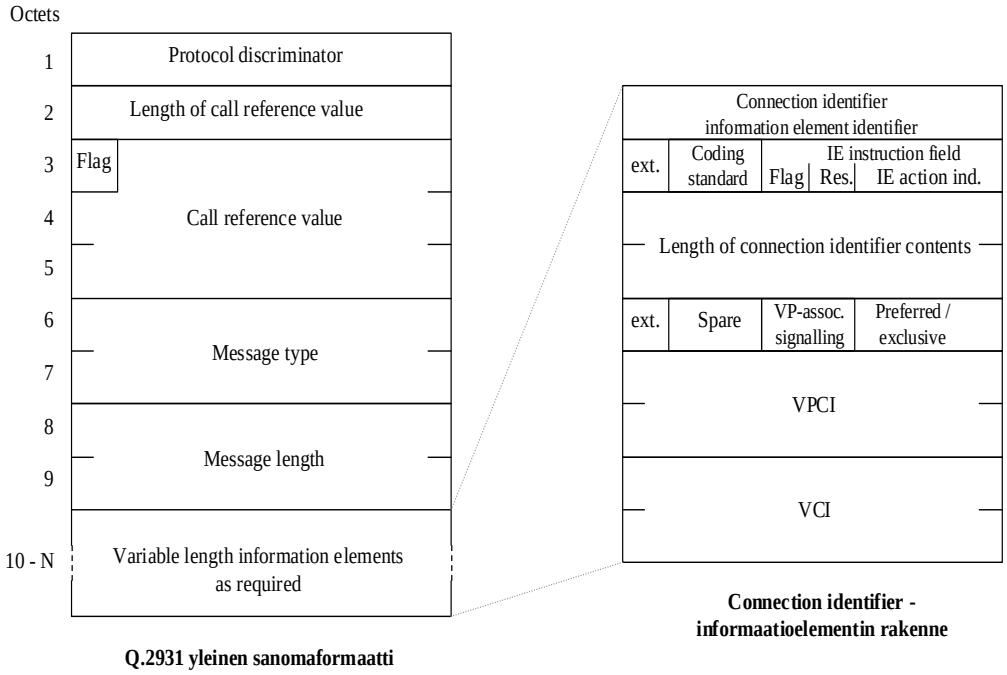


Kuva 4. Sekvenssikaavio /3/.

5 TOTEUTUS

Koodausfunktioiden toteutukseen haluttiin löytää siis yleinen ja elegantti tapa, koska samalla menetelmällä tultaisiin toteuttamaan useita sekä ITU-T:n että ATM-Forumin määritysten mukaisten signalointiprotokollien koodereita.

Ensimmäisessä vaiheessa verrattiin sekä Q.2931, UNI3.1 että B-ISUP standardeja ja havaittiin näiden sanomarakenteiden olevan hyvin samanlaisia. Sanoman otsikosta löytyy sanoman tunnistekenttä ja sanoman pituuskenttä. Otsikon jälkeen on vaihteleva määrä informaatioelementtejä, joista osa on pakollisia ja osa valinnaisia. Informaatioelementeissä on kaikille yhteinen osa sekä elementtispesifiset osat. Informaatioelementit koostuvat useista alle tavun mittaisista kentistä.



Kuva 5. Q.2931 yleinen sanomarakenne ja esimerkki informaatio-elementistä /2/.

Ideoita lähdettiin hakemaan suunnittelumalleista (design pattern) ja tavoitteesta että syntyvä koodi olisi selkeää ja helposti tulkittavaa, jotta toteutusta voisi helposti jatkaa kuka tahansa projektin jäsenistä. Vastakohta tavoitellulle ratkaisulle on oheisen kuvan koodiesimerkki (DECT:n liikkuvuudenhallintakerroksen koodereista), joka on erään CVOPS:lla toteutetun informaatioelementin koodausfunktio. Koodi on tyypillinen esimerkki kooderien toteutuksesta C-kielellä. Koodausfunktiot sisältävät runsaasti bittioperaatioita sekä pointterien epäselvää käsittelyä. Tätä kuvaakin hyvin sanonta C-kielestä: ”write only language”. Jos tämän tyyppistä koodia annettaisiin toiselle ohjelmoijalle jatkokehitystä varten, veisi se häneltä valtavasti aikaa sisäistää ja selvittää koodin toiminta.

```
/* mm_auth_type encode */
void mm_enc_auth_type(uint8 **v_addr, MM_AUTH_TYPE_STR
*mm_auth_type)
{
    uint8 len;
    uint8 apu;

    **v_addr = MM_AUTH_TYPE_TAG;
    (*v_addr)++;

    if(mm_auth_type->auth_alg_id == AAID_PROPR) len=3; else len=4;
    **v_addr = len;
    (*v_addr)++;

    **v_addr = mm_auth_type->auth_alg_id;
    (*v_addr)++;

    if(mm_auth_type->auth_alg_id == AAID_PROPR) {
        **v_addr = mm_auth_type->propr_alg_id;
       (*v_addr)++;
    }

    apu = ((mm_auth_type->auth_key_type & 0xf) << 4) |
       (mm_auth_type->auth_key_number & 0xf);
    **v_addr = apu;
    (*v_addr)++;

    apu = ((mm_auth_type->INC & 1) << 7) |
       ((mm_auth_type->TXC & 1) << 5) |
           ((mm_auth_type->UPC & 1) << 4) |
              (mm_auth_type->cipher_key_number &0xf);
    **v_addr = apu;
    (*v_addr)++;
}
```

Kuva 6. Tyypillinen C-kielinen koodausfunktio /5/.

5.1 Bittien käsittely

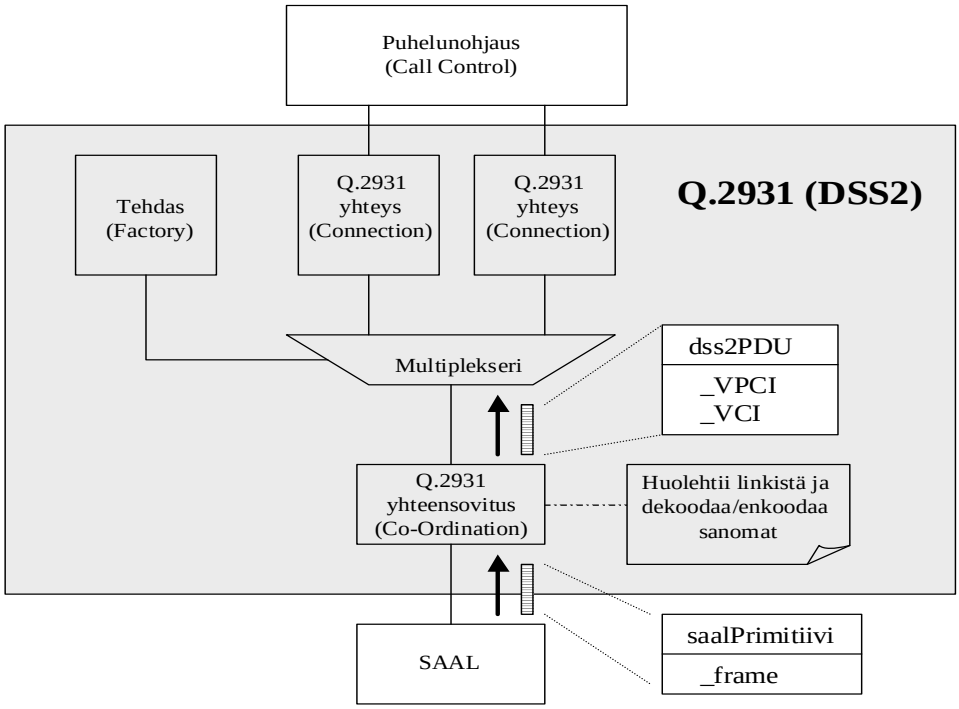
Projektin toteuttama ja käyttämä OVOPS++-sovelluskehys sisältää frame-luokan informaatiokehykselle, josta voidaan lukea oktetti tai sen monikertoja kerrallaan. Vastaavat metodit löytyvät kirjoitusoperaatiota varten. ATM-pinon alempien kerrosten koodausfunktiot toteutettiin yksinkertaisuutensa vuoksi hyödyntäen suoraan frame-luokan rajapintaa. Ylempien kerrosten monimutkaisia koodereita varten haluttiin siis frame-luokkaa monipuolisempi rajapinta. Koska sovelluskehyksen frame-luokka haluttiin säilyttää yksinkertaisena, ja pinon läpi kulkevan kehyksen kopiointia haluttiin välttää, uuden luokan periminen frame-luokasta ei tullut kyseeseen.

Toteuttiin täysin erillinen yleinen apuluokka (BitHandler) ylempien kerrosten koodereita varten. BitHandler-luokkaan asetetaan käsiteltävä kehys dekoodattaessa. Siltä voidaan pyytää halutun mittaisia kenttiä (1-N bittiä), hypätä (poistaa kehyksestä) halutun bittimäärän yli tai käyttää extension-mekanismia valinnaisten kenttien dekoodaukseen. Myös frame-luokan tapaan BitHandler-luokasta voidaan lukea halutusta kohdasta tavu tai kysyä jäljellä olevan kehyksen pituus. BitHandler-luokka käyttää frame-luokan metodeja ottaakseen kehyksestä tavuja dekoodaukseen. Koska BitHandler mahdollistaa alle tavun mittaisten kenttien käsittelyn, sen tehtävänä on säilyttää käsiteltävä tavu ja ylläpitää sen dekoodauskohtaa (bitin järjestysnumeroa).

Koodaukselle BitHandler-luokka sisältää käänteiset operaatiot: asetetaan halutun mittaisia kenttiä, koodataan tyhjiä bittejä (spare) sekä käytetään mainittua extension-mekanismia.

5.2 Sanomat

Q.2931-protokollan toteuttaminen käyttäen OVOPS++-sovelluskehystä antoi tietyt raamit myös sanomien toteuttamiselle. Kuva 7 havainnollistaa Q.2931-protokollan rakennetta sekä sanomien kulkua siinä.

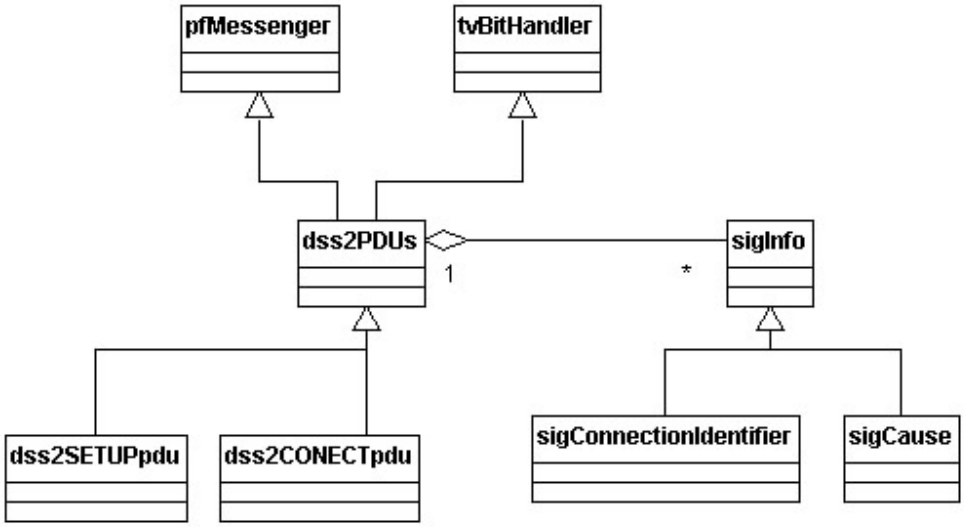


Kuva 7. Q.2931-protokolla ja sanoman muodostuminen.

Kerrosten välillä kulkevat primitiivit ovat OVOPS++-sanomia eli ne on peritty pfMessenger-luokasta. PfMessenger on command-suunnittelumallin mukainen. Se erottaa ja parametrisoi erilaiset pyynnöt omiksi olioikseen. Q.2931-protokollalle (yhteensovitusosalle) tuleva primitiivi sisältää OVOPS++-kehyksen (frame-luokka). Yhteensovitusosa dekoodaa kehyksen ja muodostaa siitä PDU:n. Myös PDU on peritty pfMessenger-luokasta, jotta se voi kulkea multiplekserin läpi yhteysosalle. PDU:n datajäseninä ovat kehyksestä puretut informaatioelementit.

Kappaleen 4.2 (UML) kaavioesimerkeissä näkyy Builder-suunnittelumallin periaate. Builder-suunnittelumallin idea näytti sopivan hyvin koodausfunktioiden toteutukseen. Siinä sanoma rakennetaan osa (informaatioelementti) kerrallaan ja lopuksi voidaan pyytää valmis sanoma kooderilta. Sanoman ja informaatioelementtien sekä kooderien välisistä suhteista syntyi lukuisia ideoita ja luokkakaavioita. Koska pääpaino oli koodausfunktioiden toteutuksella valittiin yksinkertainen ratkaisu, ja jätettiin sanoman/kooderien hienommat rakenteet odottamaan olio-ohjelmointikokemuksen karttumista.

Päädyttiin ratkaisuun, jossa informaatioelementeillä on oma oliohierarkiansa ja PDU:lla omansa. Q.2931-sanomille määriteltiin kantaluokka, jonka attribuuteiksi tuli kaikki mahdolliset informaatioelementit ja metodeiksi näiden informaatioelementtien koodausfunktiot. Tämä yksinkertaisti rakennetta huomattavasti. Ratkaisu ei välttämättä ollut kovin tyylikäs, mutta olio-ohjelmoinnissa rakenteen muuttaminen myöhemminkin on verraten helppoa. Suunnittelumalleja hyödynnettäisiin jatkossa enemmän. Metodeina toteutettavat informaatioelementtien koodausfunktiot olisi myöhemmin helppo sijoittaa mihin tahansa rakenteeseen.



Kuva 8. Q.2931-sanomarakenne.

6 TYÖN TULOKSET

Työn tärkein tulos oli yleinen luokka (BitHandler) koodausfunktioiden toteutuksen apuvälineeksi. Tämä luokka piilottaa hankalat ja virhealttiit bittioperaatiot selkeän rajapinnan taakse. Lisäksi se tarjoaa automaattisen extension-bitin käsittelyn liittyen valinnaisten kenttien koodaukseen ja dekoodaukseen. Tätä luokkaa voidaan käyttää varsinaiset koodausfunktiot sisältävien luokkien kantaluokkana. Luokan attribuutiksi asetetaan käsiteltävä kehys. Luokan tarjoamien metodien avulla tietoliikennesanomien ja informaatioelementtien kenttien käsittely selkeytyy ja virheet koodausfunktioiden toteutuksessa vähenee. Liitteessä 1 on BitHandler-luokan C++-otsikkotiedosto (header file).

Koodin luettavuuteen vaikutti olennaisesti myös projektin alussa sovitut tyylisäännöt. Kun jokainen projektiryhmän jäsen käytti koodatessaan sovittuja tyylisääntöjä, tuli koko ohjelmistosta yhdenmukainen ja helposti tulkittavaa.

Taulukko 1. Esimerkkejä tyylisäännöistä.

_attribuutti	Luokan datajäsenet alkavat alaviivalla.
parametri_	Metodien parametrit loppuvat alaviivaan.
muuttuja	Metodien paikallisissa muuttujissa ei alaviivaa.
moduliLuokanNimi	Luokan nimen alussa modulin nimi (prefix).
moniosainenMetodinNimi moniosainenMuuttujanNimi	Moniosaiset metodien ja muuttujien nimet yhteen kirjoitettuna, uuden sanan alkaessa aina isolla kirjaimella.

Koodin laatu (oikeellisuus) tarkastettiin koodikatselmuksissa koko ryhmän kesken. Koodikatselmuksia varten jokainen ryhmän jäsen tutustui koodiin ja merkitsi löydökset (mahdolliset ohjelma- ja tyylisääntövirheet) ylös, jotka sitten esitettiin koodikatselmuksissa koodin tekijälle.

Edellä mainittua BitHandler-luokkaa käyttäen toteutettiin Q.2931:sen määritysten mukaiset signalointisanomien koodausfunktiot ja prototyyppi Q.2931-protokollasta. Protokollan toteutuksessa käytettiin OVOPS++-sovelluskehystä. Oheinen koodiesimerkki on Q.2931:n (DSS2, Digital Subsciber Signaling System No. 2) Connection Identifier –informaatio-elementin koodausfunktiosta.

```
void dss2Pdu :: encodeConnectionIdentifier(void)
{
    if (_connectionId.isIEpresent() != 0)
    {
        long startLength = frameLength();
        putBits(8, _connectionId.getVCI());
        putBits(8, _connectionId.getVPCI());
        putBits(3, _connectionId.getPreferredExclusive());
        putBits(2, _connectionId.getVPassociatedSignalling());
        putSpare(2);
        putExtension(1);
        encodeCommonFields(_connectionId, startLength, B0101_1010);
    }
    return;
}
```

Kuva 9. Esimerkki koodausfunktiosta.

Tärkeitä työn tuloksia olivat myös oppiminen, ohjelmointikokemuksen kertyminen, protokollasuunnitteluun tutustuminen ja yleisesti olioajattelun sisäistäminen projektin muita tehtäviä varten.

6.1 Jatkokehitys

Koska tämän työn loppuraportti on viivästynyt parilla vuodella, on tähän raporttiin ollut mahdollisuus kertoa myös työn tulosten todellisesta jatkokehityksestä.

BitHandler-luokka on säilynyt loppujen lopuksi lähes ennallaan ja sitä käytetään edelleen signalointiprotokollien koodausfunktioiden toteutuksessa. Luokka on nykyisin nimeltään (pfIE, protocol framework –

Information Element) ja se on liitetty osaksi OVOPS++-sovelluskehystä. PfIE-luokkaan kannattaisi toteuttaa valmiiksi muitakin usein esiintyviä rakenteita, kuin extension-mekanismi, virheiden minimoimiseksi ja toistuvan koodin välttämiseksi. Tällaisia olisivat mm. erilaiset osoitteiden koodaukset, kuten BCD-luvun (taulukkomuotoinen) koodaus.

Dekoodattujen informaatioelementtien kenttien tallettamiseen suunniteltiin myös yleinen merkkijonoa (string) avaimenaan käyttävä hierarkinen tietorakenne. Mutta tämä hylättiin myöhemmin merkkijonojen käsittelystä aiheutuneen huonon suorituskyvyn takia.

Koodausfunktioiden ja informaatioelementtien suhteesta toisiinsa tutkittiin useita vaihtoehtoja, hyödynnettiin suunnittelumallien ideoita ja toteutettiinkin erilaisia protoja. Nykyisin kooderit (toiminta) ja data (informaatioelementit) on erotettu state-suunnittelumallin mukaisesti omiksi hierarkioikseen.

7 YHTEENVETO

Tässä työssä tutkittiin tietoliikennesanomien koodausfunktioiden toteutusta ATM-signaloinnissa. Koodausfunktioiden virhealttiit bittioperaatiot eristettiin selkeän ja helppokäyttöisen rajapinnan taakse. Syntynyttä yleiskäyttöistä apuluokkaa varten toteutettiin Q.2931-sanomien kooderit sekä OVOPS++-sovelluskehyksen päälle prototyyppi Q.2931-protokollasta.

Kooderien ja sanomarakenteen luokkahierarkioista syntyi useita ideoita, mutta suunnittelumallien laajempi hyödyntäminen siirrettiin myöhemmän tutkimuksen kohteeksi. Suunnittelumalleista ja olioajattelusta yleensä saatiin kuitenkin arvokasta kokemusta jatkoa ajatellen.

Tässä työssä UML-kieltä hyödynnettiin ainoastaan dokumentoinnissa. Jatkossa UML kannattaa ottaa käyttöön heti projektin alusta alkaen suunnittelun kokonaisvaltaiseksi apuvälineeksi.

LÄHTEET

1. M. Piscitello, A. Chapin, Open Systems Networking TCP/IP and OSI, Addison-Wesley, 1993.

2. ITU-T Recommendation Q.2931, Broadband Integrated Services Digital Network (B-ISDN) – Digital Subscriber Signaling System No. 2 (DSS2) – User-Network Interface (UNI) Layer 3 Specification for Basic Call/Connection Control, ITU-T, 1995.

3. E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995, ISBN 0-201-63361-2.

4. M. Fowler, K. Scott, UML Distilled: Applying the Standard Object Modeling Language, Addison-Wesley, 1998, ISBN 0-201-32563-1.

5. Kurssin 1651 Siirtyvä tietoliikenne -harjoitustyö, LTKK 1994.

LIITE 1. BitHandler-luokan C++-otsikkotiedosto.

```
//Editor-Info: -*- C++ -*-
//
//Subject: TOVE project / CODEC
//
//File: bithandler.h
//
//Version: $Revision: 1.7 $
//
//Date: $Date: 1997/12/05 07:27:39 $
//
//Organisation:
//      Helsinki University of Technology
//      Laboratory of Telecommunications and Multimedia
//
//Author:
//      Timo Pärnänen
//
//Description:
//      Provides methods used in encoding and decoding
//      functions. Interface has methods to set/get bits
//      to/from frame and it provides also some methods to
//      handle more complex structures like extension
//      mechanism.
//

#ifndef __TV_BITHANDLER_H__
#define __TV_BITHANDLER_H__

#include "pf/types.h"
#include "pf/frame.h"

//
//Class: tvBitHandler
//

class tvBitHandler
{
    public:
        ~tvBitHandler(void);
        tvBitHandler(void);

        void start(pfFrame &frame_);
        pfFrame getFrame(void);
        pfLong frameLength(void);

        pfByte read(int offSet_);

        pfLong getBits(int amount_);
        void putBits(int amount_, pfLong field_);

        void skip(int amount_);
        void putSpare(int amount_);

        void getExtension(void);
        void putExtension(pfLong value_);
        pfLong getBitsInExtension(int amount_);
        void skipInExtension(int amount_);
        pfBoolean isInExtension(void);
        void endExtension(void);
```

```
        void getEvenOdd(void);
        void putEvenOdd(int value_);

        pfBoolean isValid(pfLong value_, pfLong invalid_);
        void setDefault(pfLong &value_,
                        pfLong invalid_,
                        pfLong default_);


    protected:
        pfFrame _frame;

    private:
        void putBitsToByte(int amount_, pfLong field_);
        void getBitsFromByte(int amount_, pfLong &field_);

        static const int MAXBIT = 8;

        pfByte _currentByte;
        int _encPosition;
        int _decPosition;
        pfLong _extBit;
        pfLong _currentExt;
        pfLong _evenOdd;
};

#endif // __TV_BITHANDLER_H__
```