Spagettia – Raviolia – Lasagnea

Anne Eerola*

Kuopion yliopisto, Tietojenkäsittelytieteen laitos

Avainsanat

Tietojärjestelmän suunnittelu, ohjelmiston mallinnus, tiedonhallinta, ohjelmiston rakenne, oliokeskeisyys, liiketoimintaprosessit, ohjelmistotekniikka, modulit, komponentit, arkkitehtuuri, integraatio.

1. JOHDANTO

Tarkastelen tässä artikkelissa tietojärjestelmien määrittelyn ja ohjelmistotekniikan kehittymistä Suomessa omien kokemusteni pohjalta. Esitän niitä asioita ja näkemyksiä, jotka ovat tulleet vastaan ja jotka tuntuvat edelleen tärkeiltä. Olen toiminut teollisuuden ohjelmistoammattilaisena, yliopiston opettajana ja tutkijana 1970-luvulta alkaen. Esimerkiksi Lyytinen ja Ivari ovat tarkastelleet tietojärjestelmien kehitystä ja kehitykseen vaikuttaneita suuntauksia Skandinaviassa [8].

Esitettävien asioiden ymmärtämisen helpottamiseksi määritellään muutama keskeinen käsite: *Systeemi* on joukko komponentteja, jotka vaikuttavat toisiinsa ja joilla on tietty yhteinen tarkoitus. Systeemi koostuu ihmisistä, koneista sovelluksista, apuvälineistä, alisysteemeistä jne. *Liiketoimintasysteemi* tai *toimialasysteemi* on perustettu toteuttamaan tietty liikeidea tai yleishyödyllinen tehtävä. *Tietojärjestelmä* (*Information System*) ohjaa tietoa, jota tarvitaan systeemissä. Tietojärjestelmä on sosiaalinen järjestelmä, jossa ihmiset käyttävät tietotekniikkaa, eli *ohjelmistoja* (*software*), *tietokoneita* (*hardware*) ja tietoliikenneyhteyksiä.

* Kirjoittaja työskentelee Ohjelmistotekniikan professorina Kuopion yliopiston Tietojenkäsittelytieteen laitoksella
email: Anne.Eerola@cs.uku.fi

2. 1970-LUKU

2.1 Temppulevy ja ISAC

Opiskelin tietojenkäsittelyoppia 1970 luvun alussa Turun yliopistossa. Ohjelmointia harjoiteltiin Fortran-, Cobol- ja Assmbler-kieliä käyttäen. Ohjelmat lävistettiin *reikäkorteille*. Korttipinot laitettiin laatikkoon, josta operaattorit ajoivat ne. Parhaimmillaan ohjelman sai päivän aikana ajoon pari kertaa. Tästä oli se etu, että ohjelman logiikkaa ehti miettiä rauhassa ja *lohkokaavioita (flow chart)* ohjelman toiminnoista ja haarautumisista kannatti piirtää. Tähän käytettiin temppulevyä. Ohjelmoinnin lisäksi opetuksessa keskeisellä sijalla olivat tietorakenteet, kuten pinot ja linkitetyt listat, algoritmit, käyttöjärjestelmät ja kääntäjät. Codd'in (1970) esittämät *relaatiotietokantojen normaalimuodot* [4] ja *kyselyjen optimointi* tulivat tutkimuksessa tutuiksi. Sivuaineena 1970-luvun alussa opiskeltiin mm. kauppatiedettä (kauppakorkeakoulusta), matematiikka ja fysiikkaa.

Systemointia opeteltiin Kerolan ja Järvisen laatiman luentomonisteen avulla. Ensimmäinen ja antoisa kontakti käytännön työelämään tuli suunnitteluprojektin muodossa. Mieleen jäi yritysedustajan neuvo:

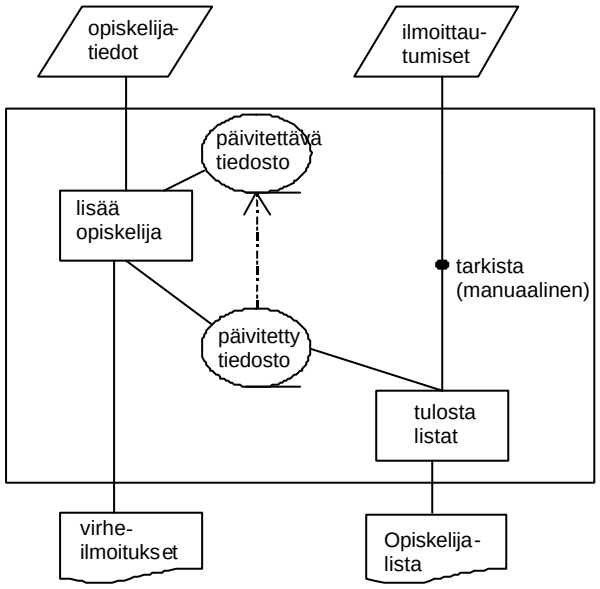
Jos tehdään kaikki, mitä käyttäjä haluaa, saadaan aikaiseksi ns. halusysteemi, jota ei halua enää kukaan.

Systemointitaitoja syvennettiin tutustumalla ISAC menetelmään (Information Systems work and Analysis of Change). Menetelmä oli Suomessa suosittu 70 ja 80-luvulla. Menetelmä oli toimintopainotteinen. Mallintaminen aloitettiin kuvaamalla nykytila. Mukaan tarkasteluun otettiin sekä *manuaaliset että automaattiset toiminnot*.

Menetelmässä kuvattiin (kts. Kuva 1):

- toimintoihin tulevat syöttötiedot tai materiaalijoukot,
- tuloksena saatavat tulostiedot ja materiaalijoukot ja
- ylläpidetyt tiedostot.

Paljon huolta ja vaivaa aiheuttivat magneettinauhatiedostojen päivitykset, jotka piti suunnitella tarkasti, jotta päivitys ei kestänyt liian kauan. Nykytilan mallintamisen jälkeen suunniteltiin kehittämistavoitteet ja johdettiin tavoitetila, joka kuvattiin samalla notaatiolla kuin nykytila.



Kuva 1. ISAC KAAVIO

2.2 Yrityksen sisäinen atk-osasto

1970-luvulla oli tyypillistä, että isoilla yrityksillä oli omat atk-osastot, jotka ylläpitivät ja kehittivät *räätälöintiperiaatteella* ohjelmia juuri kyseisen yrityksen tarpeisiin. Työskentelin tällaisessa organisaatiossa. Kehittämishankkeet organisoitiin projekteiksi, mutta *projektiseuranta oli löyhää*. Projektiin saatettiin lisätä tehtäviä ja työntekijöitä arvioimatta kannattavuutta uudelleen. Projektin aikataulua ja päättymistä ei seurattu, vaan projektit joko kasvoivat tai loppuivat pikkuhiljaa. Yleisesti atk-projekteja pidettiin kannattamattomina. Elinkaarimalleista ei puhuttu, mutta *noudatettiin vesiputous mallia* hyvin ankaralla tavalla:

– Ensin määriteltiin asiakkaiden kanssa yhteistyössä lähinnä keskustellen ja haastattelujen avulla, mitä rakennettavan järjestelmän tulee tehdä ja mitä tietoja käsitellään. Tämä vaihe saattoi kestää 3-9 kuukautta.

– Kun tarpeet oli määritelty, ohjelmistoammattilaiset alkoivat toteuttaa sovellusta. Toteutusvaihe saattoi kestää 1-3 vuotta. Tänä aikana oltiin hyvin vähän yhteyksissä asiakkaisiin.

Kun ohjelma saatiin valmiiksi, se ei yleensä ollut ihan sellainen, mitä käyttäjä olisi tarvinnut.

Elettiin aikaa, jolloin atk-ammattilaisen pöydällä ei ollut tietokonetta. Asiakasvaatimuksista kirjoitettiin tekstidokumentteja, joita sihteerit kirjoittivat puhtaaksi. Lisäksi ohjelmia dokumentoitiin lohkokaavioiden ja *kommenttirivien* avulla. Kaikki koneet olivat konehuoneessa, jonne piti mennä kääntämään ja ajamaan ohjelmansa. Tietokoneiden muisti ja suorituskapasiteetti oli murto-osa nykyisestä. Niinpä yhtenä isona työvaiheena oli suunnitella *ohjelmien segmentointi* siten, että linkitetyn ohjelman osien *heittoa levylle (swapping*) tapahtui mahdollisimman vähän. Tavoitteena oli, että suoritusaika ei kasvaa kohtuuttomaksi ja keskusmuistiin mahtuu samanaikaisesti tarvittavat segmentit.

Tuotannossa tapahtumat kirjattiin ylös *syöttölomakkeille*, joilta tiedot lävistettiin keskitetyssä *lävistyksessä reikäkorteille*, jotka syötettiin konehuoneessa *eräajo-ohjelmille*.

Asiakkaiden kommenteista ja ohjeista tärkeimpänä jäi mieleen:

Tärkeintä ei ole tehdä kaikkia asioita oikein vaan tehdä oikeita asioita.

3. 1980-LUKU

3.1 Osituskäyttö ja päätteet yleistyvät

Äitiyslomalle jäädessäni 1980 käytettiin yleisesti reikäkortteja. Lomalta palatessani niitä ei ollut enää missään. Yrityksessä siirryttiin vähitellen *osituskäyttöön*

ja tapahtumia alettiin syöttää järjestelmään suoraan *päätteiden* avulla. Paljon keskusteltiin siitä, onko kyseinen menettelytapa riittävän luotettava, kun tapahtumista ei jäänyt mitään syöttölomaketta tositteeksi. Tästä syystä kehitettiin *tapahtumien varmistukseen* automatisoituja menetelmiä.

3.2 Jokainen tekee ohjelmansa itse

Käyttäjät ja asiakkaat alkoivat olla tyytymättömiä pitkiin toimitusaikoihin, ohjelmistotuotannon tuottavuuteen ja siihen, että sovellus ei ollut valmistuessaan sellainen kuin käyttäjä oli ajatellut. Samanaikaisesti yleistyivät *mikrotietokoneet ja* ns. *neljännen sukupolven välineet*. Ensimmäistä kertaa käyttäjät pystyivät rakentamaan itse sovelluksiaan. Välineet olivat helppoja oppia ja niiden avulla saattoi *ratkaista spontaaneja tietotarpeita* riittävän edullisesti ja nopeasti.

Atk- ammattilaisena tehtäväkseni jäi opettaa välineen käyttöä, suunnitella ratkaisu karkealla tasolla ja kirjoittaa ohjelma, joka poimi yrityksen tiedostoista käyttäjän tarvitsemat tiedot. Näitä käyttäjä sitten muokkasi itsenäisesti:

- valitsemalla tietojoukosta spontaanisti määritellyt ehdot täyttävät tiedot,
- lajittelemalla, ryhmittelemällä ja suorittamalla laskutoimituksia.

Tällainen väline oli esim. Mapper. Myös taulukkolaskentaa hyödynnettiin.

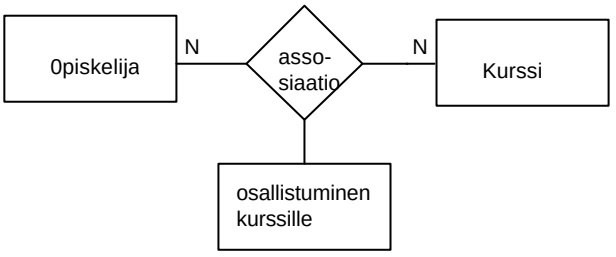
Menettelytapa mahdollisti ”mitä jos” analyysien suorittamisen ja *päätöksentekoa* tukeviin tietojärjestelmiin (esim. *asiantuntijajärjestelmät*), alettiin kiinnittää huomiota järeiden *tuotannon suunnittelua ja tuotantoa* tukevien järjestelmien (esim. CAD/CAM) rinnalla.

Yleisesti arveltiin, että kehitys jatkuu samaan suuntaan. Vähän kerrassaan jokainen tekee ohjelmansa itse ja atk-ammattilaiset käyvät tarpeettomiksi. Asia ei kuitenkaan ollut aivan näin yksinkertainen. Mikrot ja neljännen sukupolven välineet johtivat siihen, että *tietojen redundanssi* kasvoi hallitsemattomasti. Tiedot eivät olleet enää *yhdenmukaisia* ja *ristiriidattomia*. Esimerkiksi varaston rahallinen arvo saattoi olla eri riippuen siitä laskettiinko se materiaalihallinnan mikrolla vai talousosaston tietojärjestelmällä. Lisäksi suurten tietomäärien hallinnassa tuli tehokkuusongelmia. Analyysiä varten tarvittiin useita satoja tuhansia rivejä ja niitä ei pystytty analysoimaan ja lajittelemaan riittävän nopeasti.

1980-luvulla yleistyi ohjelmien prototyyppien rakentaminen. Markkinoilla oli välineitä, joiden avulla voitiin nopeasti rakentaa sovelluksesta prototyyppi, joka annettiin asiakkaalle kokeiltavaksi. Prototyyppien tärkein tehtävä oli varmistaa vaatimusten oikeellisuus ja antaa käyttäjälle mahdollisuus kokeilla sovellusta etukäteen. *Prototyypit tulivat jäädäkseen*. Niitä käytetään edelleen, joskin tehokkuussyistä lopullinen koodi joudutaan usein ohjelmoimaan tehokkaammalla ohjelmointiympäristöllä ja -kielellä.

3.3 Tieto yrityksen keskeinen resurssi

Vähitellen tieto alettiin ymmärtää yrityksen keskeiseksi voimavaraksi. Käsitteitä ja tietoja määriteltiin ja mallinnettiin ER-kaavioiden avulla (kts. Kuva 2) [3]. Tiedostopohjaiset järjestelmät korvautuivat vähitellen *tiedonhallintaohjelmistoilla*, joissa oli tuettuna *tietojen samanaikainen päivitys*.



Kuva 2. Yksinkertainen ER-malli

3.4 Ohjelmistokehitys asiakasryhmissä

1980-luvun puolessa välissä muutin Kuopioon ja alkoi ohjelmistotalo-vaihe.

Ohjelmistoa ei enää räätälöity yhdelle asiakkaalle, vaan sovelluksia kehitettiin useiden asiakkaiden toiveiden ja *prioriteettien* mukaisesti asiakkaista kootun *ohjausryhmän* tuella. Apuna käytettiin muun muassa *seinätaulutekniikkaa,* jossa seinälle tai taululle

hahmoteltiin kuvaavien symbolien avulla nykytilaa tai tavoitetilaa. Näitä seinätauluja päivitettiin yhdessä asiakkaiden kanssa *aivoriihi*menettelyä käyttäen. Luovuutta eristävät keinot ja toimintatavat olivat kahvipöytäkeskustelujen aiheena. Tyypillistä oli, että asiakasryhmät kokoontuivat puolenvuoden välein yhden sovelluksen ympärille. Tuotteistuksesta ei vielä puhuttu. Isompiin kehitysponnistuksiin joku asiakas lähti yleensä *pilottiyritykseksi* osallistuen ohjelmiston määrittelyyn ja testaukseen tiiviimmin kuin muut yritykset. Yritys sai näin äänensä paremmin kuuluville ja ohjelmistot ensimmäisenä käyttöönsä.

Parametriohjatut ohjelmistot mahdollistivat sen, että asiakkaalle ei rakennettu sovellusta alusta alkaen, vaan parametrien ja skripti-kielen avulla sovitettiin ohjelmisto asiakkaan käyttöön sopivasi.

Projektityömenetelmiin alettiin kiinnittää entistä enemmän huomiota. Projektisuunnitelmat tehtiin tekstinkäsittelyohjelmalla ja niissä myös yritettiin pysyä. Taulukkolaskennalla hoidettiin projektiseuranta. Uusinta uutta olivat *työasemat*.

Ohjelmien rakenteeseen, dokumentointiin ja testauksen ei kiinnitetty paljoa huomiota. Sovellukset olivat edelleen *monoliittisia* ja koodi oli yleensä ainoa dokumentti. *Uudelleenkäytössä yleisin menettely oli "copy and paste"* Tämä aiheutti sen, että ohjelmarivien määrä kasvoi nopeasti. Koska kopioitua ohjelmakoodia ei yleensä kokonaan ymmärretty, siitä ei uskallettu ottaa mitään pois. Niinpä tehtiin lisää if-lauseita, joiden taakse koodattiin tarvittava lisätoiminnallisuus. Yleinen vitsi oli:

Koskaan ei ole aikaa tehdä ohjelmaa kunnolla, mutta aina on aikaa tehdä se uudelleen.

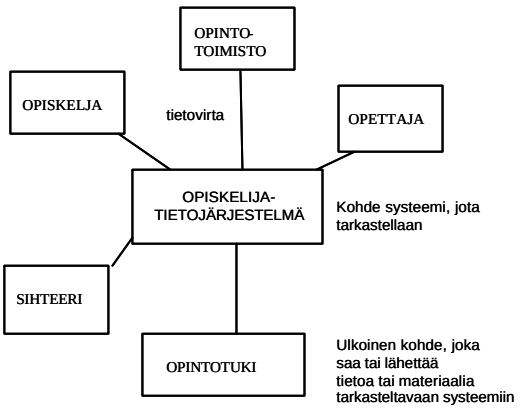
Tyypillistä oli, että ohjelmiston uudistamisen yhteydessä piti uusia kaikki laitteistosta ja tietokantaohjelmistosta lähtien. Jos yrityksellä oli aiemmin ollut käytössä hierarkkinen tietokanta (esim. IMS) tai verkostopohjainen ratkaisu, harkittiin olisiko syytä siirtyä *relaatiotietokantaan*. Alettiin myös keskustella *tietokantojen vaihdettavuudesta*. Nähtiin, että olisi etu, jos asiakkaalle voitaisiin myydä sovellus siten, että tietokantaratkaisu olisi asiakkaan valittavissa. Tähän ajanjaksoon liittyy yksi työurani moka. Hankin tiedonhallintaohjelmiston, jossa ei ollut samanaikaisen päivittämisen tukea. Ei tullut mieleeni varmistaa selvältä tuntuvaa asiaa.

3.5 Rakenteellinen analyysi

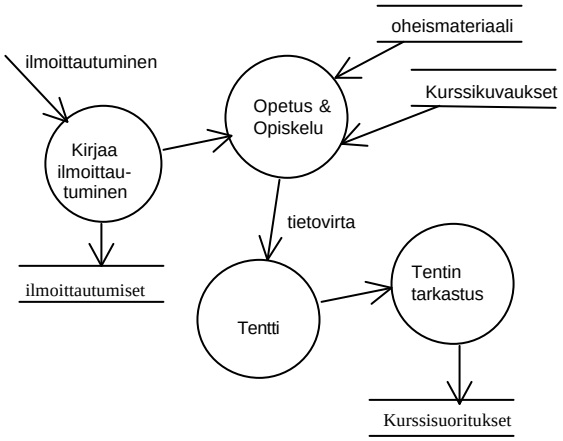
1989 siirryin opettajaksi ja tutkijaksi Kuopion yliopistoon tietojenkäsittelyopin ja matematiikan laitokselle. Käsiteanalyysin rinnalla alettiin kiinnittää huomiota ohjelmiston toiminnallisuuden ja rakenteen kehittämiseen:

Ohjelma ei saa olla spagetti-koodia, jossa "go-to" -käskyjä käyttäen hypitään paikasta toiseen.

Rakenteellisessa analyysissä hahmoteltiin tietojärjestelmän rajaus (Kuva 3) ja yleiskuva tietojärjestelmän toiminnasta (Kuva 4) [9]. Myös ihmisten työskentelyä tarkasteltiin (manuaaliset toiminnot).



Kuva 3. Yhteys kaavio (context diagram)



Kuva 4. Yleiskaavio (overflow diagram)

Jälkiviisaasti ajatellen rakenteellinen analyysi on käyttökelpoinen organisaatioyksiköiden välisen *tiedonkulun (data flow)* kuvaamisessa ja järjestelmän yleiskuvan muodostamisessa. Myös ohjelmiston rajaus ja ulkoisten yhteyksien määrittely ovat hyödyllisiä. Alemman tason kaaviot menivät helposti liian yksityiskohtaisiksi ja ne kannatti usein korvata *päätöstaululla, -puulla tai rakenteellisella kielellä*. Rakenteellinen analyysi sopi huonosti tietokantapohjaisten ohjelmistojen suunnitteluun.

4. 1990-LUKU

4.1 Suutarin lapselle kengät

1990-luvulla huomattiin, että tietojenkäsittelyn ammattilaisetkin tarvitsevat avukseen tietotekniikkaa. CASE (Computer-Aided Software Engineering) välineet alkoivat yleistyä. Niiden avulla pystyttiin mallintamaan rakennettavaa sovellusta tietokonetta hyväksikäyttäen. Alettiin suunnitella, mitä *notaatioita* tarvitaan ja minkälaista *prosessia* tulisi käyttää ohjelmiston suunnittelussa. Tavoitteena oli, että *ohjelmistojen dokumentaatio* kehittyy ja että ohjelmia pystytään *generoimaan automaattisesti* laadittujen mallien ja kaavioiden perusteella. CASE-välineen olennaisena osana oli ja on edelleen *tietohakemisto (repository)*, jonne projektin alusta lähtien kerätään *tietoa tiedosta*. Markkinoilla oli saatavana useita CASE-välineitä - myös suomalaisia (esim. Prosa).

Huomiota kiinnitettiin myös *laatujärjestelmiin*. Yritykset ja ohjelmistotalot kuvasivat toimintansa laatukäsikirjoihin, toimintaa optimoitiin ja toiminnalle hankittiin *sertifikaatteja*. Organisaation prosesseja määriteltiin, kehitettiin ja uudistettiin käyttäen prosessikaavioita. *Liiketoimintaprosessien uudistamisessa (reengineering)* edellytyksenä on että kaikki toiminta tuottaa lisäarvoa. Kehittäminen fokusoitui avainprosesseihin ja organisaatiot madaltuivat (*baskerimalli*, eli kaikki hikinauhassa) [2]. Ohjelmien *tarkastusmenettelyt, parityöskentely ohjelmoinnissa ja ohjelmien testaus* alkoivat saada huomiota ohjelmistoteollisuudessa.

4.2 Objects are there, just pick them up

1990 luvun alkupuolella keskeiseksi systeemityön ongelmaksi tunnistettiin se että:

- Käsiteanalyysi painotti tiedonhallintaa unohtaen toiminnot.
- Rakenteellinen analyysi keskittyi toimintoihin unohtaen tiedot

Ei oltu enää tyytyväisiä siihen, että tiedot ja toiminnot olivat erikseen. Alettiin puhua *vastuista (responsibility)* ja *oliokeskeiset (object oriented)* menetelmät alkoivat yleistyä ja niiden tutkimus ja käyttö teollisuudessa lisääntyi:

Spagetti mallista siirryttiin ravioliin [7].

Olio kapseloi tiedot (raviolissa liha) ja toiminnot (raviolissa taikina), siten, että tietoihin päästiin käsiksi vain toimintojen avulla. Alun perin 1970- luvulla kehitetty Smalltalk -ohjelmointikieli pääsi uudelleen tarkastelun ja sen rinnalla C++ yleistyi C-kielen laajennuksena. Perusideana oliokeskeisessä näkökulmassa oli reaalimaailman käsitteiden ymmärtäminen olioiksi, joilla oli ominaisuutena tiedot ja näihin liittyvät toiminnot. Järjestelmän toiminnallisuus saatiin aikaan *olioiden* välisen *viestinvälityksen* avulla. Oliot kuvattiin *luokkahierarkiassa*, jossa alaluokka peri yläluokan ominaisuudet.

Rakenteellinen analyysi ei tietenkään soveltunut oliokeskeisen järjestelmän suunnittelumenetelmäksi. Paremman lähtökohdan antoi käsiteanalyysi, jonka pohjalta syntyi *useita oliokeskeisiä suunnittelumenetelmiä* 1990 luvun alussa. Olioehdokkaita olivat substantiivit, kuten käsitteet, tiedot, tapahtumat, henkilöt, organisaatiot, systeemit ja koneet. Verbit olivat metodiehdokkaita ja ne pyrittiin sijoittamaan oikeaan olioon. Tämä ei ollut aina helppoa.

Monet oliot esiintyivät sekä reaalimaailmassa (esim. kirjaston asiakas) ja ohjelmistossa (asiakastiedot). Jälkiviisaasti ajatellen tuntuu siltä, että "object oriented" -sana suomennettiin harhaanjohtavasti: Olio sanan käyttö (asiakas olio, potilas olio jne.) tuntui höl-möltä ihmisistä puhuttaessa. Olio viittasi johonkin elä-

vään. Parempia käännöksiä sanalle olio olisivat ehkä olleet objekti, entiteetti, kohde tai subjekti.

Moniperinnöllisyys (luokalla enemmän kuin yksi yläluokka) aiheutti tutkijoille erityisen paljon päänvaivaa. Oliokeskeisyydellä tavoiteltiin parempaa modulaarisuutta [12]:

- vähemmän riippuvuuksia ohjelman osien välillä (*coupling*) ja
- parempi *koossapysyvyys (cohesion)*, eli olio muodostaa tiiviin kokonaisuuden.

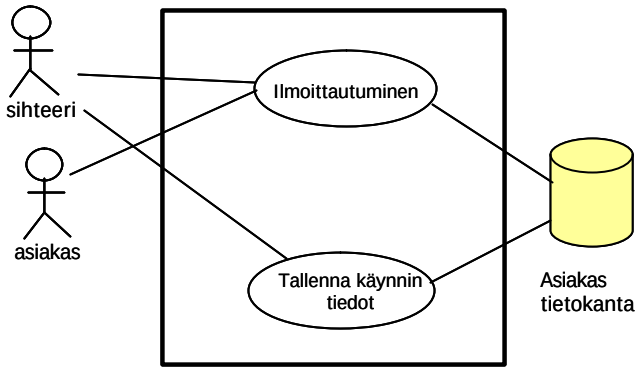
Luokkahierarkian avulla tavoiteltiin *uudelleenkäytön* lisääntymistä. Käytännössä asiat eivät edenneet ihan tavoitteiden mukaisesti, Wilde ja Huit kirjoittivat artikkelissaan [16]: ”Oliokeskeisessä ohjelmassa on perinteiseen ohjelmaan verrattuna huomattavasti enemmän ja erityyppisiä riippuvuuksia. Lisäksi olion toiminnan ymmärtämiseksi joutuu tutkimaan yläluokkia ja alaluokkia vuorotellen *(jojo-ilmiö)*”.

Onkohan niin, että kun ohjelmistoalan ammattilaiset alkoivat 1990-luvulla puhua olioista, samalla etäisyys ohjelmistojen hyödyntäjiin kasvoi. Asiaa ei auttanut se, että tietojärjestelmätutkimus ja tietojenkäsittelytiede etääntyivät toisistaan.

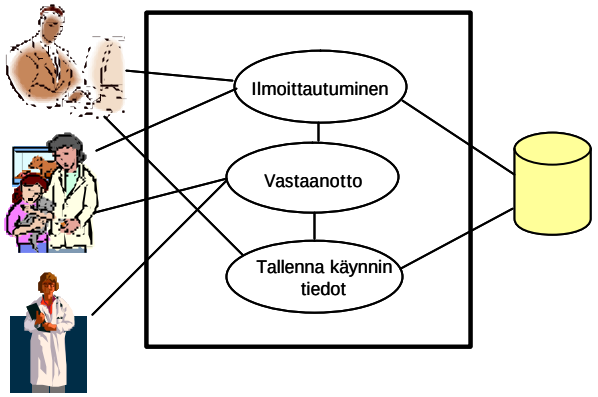
4.3 Unified Modelling Language

UML (Unified Modelling Language) kielen käyttö ohjelmien mallinnuksessa yleistyi. Näin ohjelmistoalan ammattilaiset eri puolilla maailmaa pystyvät ymmärtämään toisiaan. UML:ää tukemaan kehitettiin RUP – prosessikuvaus (Rational Unified Process) [10]. Pienien ja keskisuurten yritysten käyttöön RUP on liian raskas ja laaja. Sitä pitää supistaa. Tähän on tuki olemassa.

Käyttötapauskaaviot skenaarioineen tulivat ohjelmistoteollisuudessa suosituksi kuvaustavaksi. Määritelmän mukaan käyttötapaus on tuokiokuva ihmisen ja tietokoneen vuorovaikutuksesta (kts. Kuva 5). Tässä on heikkoutena se, että ihmisten toimintaan ilman tietokonetta ei kiinnitetä huomiota. Kaavioista ei selviä tarkasteltava toimintokokonaisuus eikä sitä näin ollen voida kehittää. On hyödyllistä kuvata käyttötapauskaavioon myös ihmisten toiminta, eli *toimintatapaukset (action case)*, jotta voidaan ottaa huomioon ihmisten työ (kts. Kuva 6). Samalla ohjelmistojen *käytettävyys paranee*, kuvauksista tulee ymmärrettävämpiä ja niiden järkevyyden pystyy tarkistamaan.



Kuva 5. Käyttötapaus ilman ihmisten aktiviteetteja



Kuva 6. Käyttötapaus, jossa on mukana ihmisten työ

UML:ssä käytetään prosessien kuvaukseen *aktiviteettikaaviota (activity diagram)*, jonka yhdennäköisyys 1970-luvun lohkokaavion kanssa on ilmeinen. Uutena asiana on organisaatioyksikköjä kuvaavat kanavat ja prosessien tahdistuksen kuvaaminen. Näiden avulla reaalimaailmassa luonnollinen rinnakkaisuus pystytään mallintamaan. *Luokkakaaviot (class diagrams)* ovat ohjelmistoammattilaiselle hyödyllinen väline. Ongelmana on, että kaavioon tulee helposti liikaa asiaa. Tilannetta voi helpottaa kuvaamalla erik-

seen määrittely-, käyttöliittymä- ja toteutustason luokkakaaviot [5].

Kokemusten mukaan ohjelmistojen hyödyntäjät eivät kovin hyvin ymmärrä luokkakaavioita ja niissä esitettyjä asioita. Asiakkaiden kanssa kommunikointi sujuu prosessikaavioita, käyttötapauskaavioita ja skenaariokaavioita käyttäen. Kannattaa muistaa, että ennen kuin aletaan määritellä ja suunnitella ohjelmistoa pitää tarkastella työkokonaisuutta, jota ollaan tukemassa.

UML:n heikkoutena on, että se antaa kovin vähän tukea arkkitehtuurin ja komponenttien kuvaamiseen. Tämä on ikävä asia, koska arkkitehtuurien kehittäminen on tänä päivänä keskeistä niin käytännön tietojärjestelmäsuunnittelijoiden kuin tutkijoidenkin keskuudessa.

4.4 Komponentit ovat eri kokoisia

1990-luvulla yritykset alkoivat muodostaa yhteistoimintaverkkoja. Tänä päivänä on yleistä, että yritys toimii hajautetusti, tietojärjestelmät ovat hajautettuja ja jopa tiimit toimivat hajautetusti. Ohjelmistoyrityksen kilpailutekijäksi on noussut tehokkuus, toimitusaika, laatu ja luotettavuus. Uudelleenkäytön pitäisi kehittyä ja monimutkaisuus pitää hallita. Samanaikaisesti sovelluskehitys on vaikeutunut. Ei ole helppo kehittää ohjelmistoja, jotka mahdollistavat uuden teknologian, kuten hajautus, internet/intranet, tietoliikenne, mobiliteknologia. Ohjelmistotuotanto on siis, toisin kuin 1980-luvulla uskottiin, tullut entistä vaativammaksi. Kukaan alan ammattilainenkaan ei hallitse kaikkia tekniikoita. Tarvitaan erikoistumista, mutta pitää myös muistaa yhteisen kielen muodostuminen ja säilyttäminen.

Komponenteissa on modulaarisuus saatu ensi kertaa toteutettua kohtuullisen hyvin: Komponentilla on *rajapinta (interface)*, jonka kautta sen tarjoamia palveluja kutsutaan. Komponentti tarvitsee toimiakseen yleensä toisia komponentteja ja infrastruktuuria. Myös *nämä riippuvuudet (dependency)* määritellään selkeiden rajapintojen avulla. Komponentit *ovat eri kokoisia (granularity)* lueteltuna pienimmästä suurimpaan [7]:

- *Hajautettu komponentti (Distributed Component*) toteuttaa ja kätkee hajautuksen. Sillä on yksi tai useampi hyvin määritelty rajapinta, johon voidaan viitata verkosta käsin. Nimi hajautettu komponentti on hiukan harhaanjohtava, koska komponentin ei tarvitse olla hajautettu. Tyypillisesti tänä päivänä hajautettu komponentti voi olla EJB, COM, CORBA jne. teknologialla toteutettu.
- *Toimialakomponentti (Business Component)* koostuu hajautetuista komponenteista. Toimialakomponentin rajapinta muodostuu sen hajautettujen komponenttien rajapinnoista. Toimialakomponentti noudattaa *kerrosarkkitehtuuria* ja sisältää tarpeesta riippuen käyttöliittymä-, työtila-, toimintalogiikka- ja resurssikerroksen. Näin Italian keittiön *lasagne korvaa raviolin*.
- *Komponenttisysteemi (Business Component System)* koostuu toimialakomponenteista, jotka ovat tyypiltään esim. prosessi-, käsite-, apu- tai rajapintakomponentteja (kts. Kuva 7). Komponenttisysteemin rajapinta muodostuu sen toimialakomponenttien rajapinnoista.

Ohjelmien oikeellisuudelle asetetaan entistä kovempia vaatimuksia, joten testausmenetelmiä on kehitettävä [17].

4.5 Ohjelmistotekniikan koulutusta

1990-luvun lopulla useat tahot havahtuivat siihen, että ohjelmistoteollisuus tarvitsee palvelukseensa lisää tietojenkäsittelytieteen maistereita. Tähän liittyen Kuopion kaupunki lahjoitti Kuopion yliopistolle ohjelmistotekniikan professorin viran (5 vuotta). Alkoi *ohjelmistotekniikan opetuksen ja tutkimuksen kehittäminen*. Tietojenkäsittelyn opetusta laajennettiin käytännönläheiseen suuntaan huomioiden suurten, usein hajautettujen, tietojärjestelmien rakentamisen vaikeus. Opetusta lisättiin seuraavilla aihealueilla:

- Ohjelmistotuotanto, tuotteenhallinta.
- Mallinnusmenetelmät, ml. liiketoimintaprosessien ja ihmisten toimintakokonaisuuksien mallintaminen ja kehittäminen.

– Ohjelmiston rakenne: komponentit, arkkitehtuurit, suunnittelumallit.
– Ohjelmiston testaus.
– Käytettävyyden suunnittelu.
– Hajautetut järjestelmät.

Näkökulmana ovat olleet ohjelmiston hyväksikäyttäjän tarpeet, ohjelmistoyrityksen toiminta, ja ohjelmistoammattilaisen työ. Ohjelmistotekniikan tutkimuksessa tarkastellaan samoja asioita kuin opetuksessa.

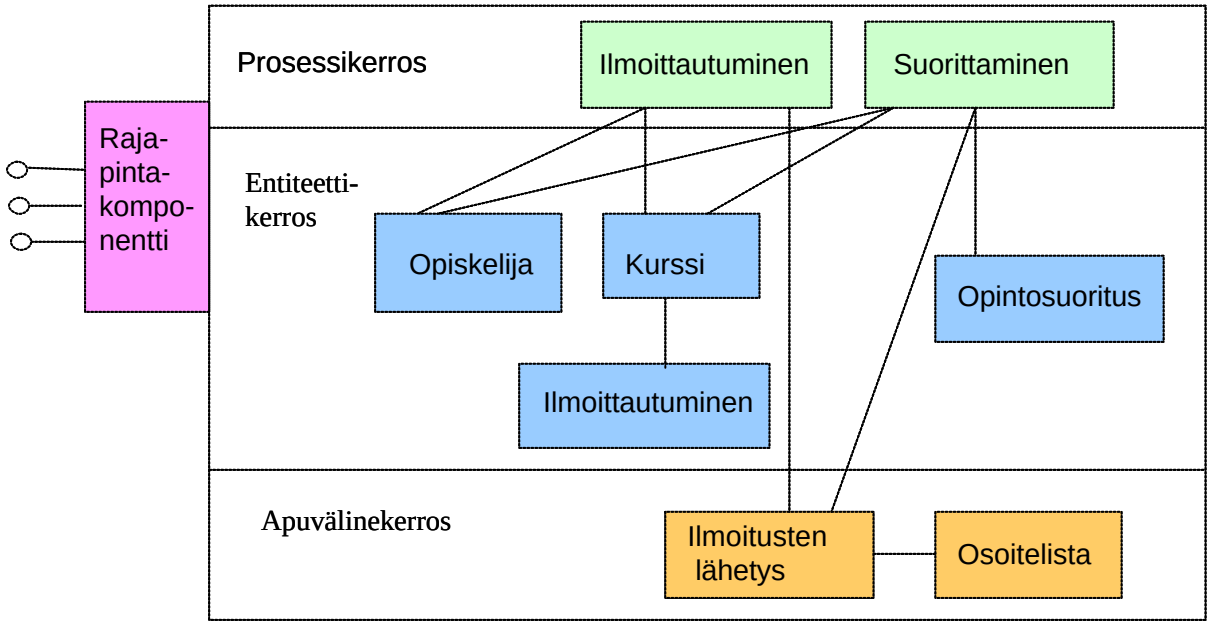
5. 2000-LUKU

Komponenttien kehittymisen *rinnalla arkkitehtuurin suunnittelu* on nousut keskeiseksi kehittämisasiaksi niin ohjelmistoteollisuudessa kuin tutkimuksessakin. Arkkitehtuurin suunnittelu, kuvaaminen ja arviointi ovat tärkeitä, koska näin voidaan etukäteen (ennen ohjelman koodaamista) arvioida sovelluksen toiminnallisten ja laadullisten ominaisuuksien toteutumismahdollisuuksia [1]. Tämä säästää kalliilta virheinvestoinneilta. Lisäksi arkkitehtuurikuvaukset mahdollistavat ajatustenvaihdon, neuvottelun ja yhteisymmärryksen muodostumisen sovelluksen ominaisuuksista komponentin toimittajan, komponentin integroijan ja sovelluksen hyväksikäyttäjän välillä [15].

Ilahduttavaa on, että alunperin Gamm:an julkistamat *suunnittelumallit (design pattern)* [6] kiinnostavat ohjelmistotuottajia, opiskelijoita ja tutkijoita. Suunnittelumallien avulla tehdään *piilossa oleva suunnittelutietämys näkyväksi*. Omaa tietotaitoa voi dokumentoida tai voi käyttää hyödyksi muiden kehittämiä malleja. Suunnittelumallien avulla helpotetaan hajautetun sovelluksen rakentamista (esim. välitin ja edustaja/korvike), sovitetaan ohjelmistoja (adapters) tai kääritään perinnejärjestelmiä (wrappers).

Tänä päivänä sovelluksilta edellytetään *yhteistoiminnallisuutta (interoperability)*. Integraatiota toteutetaan työpöytä-, toimintalogiikka- ja tiedostotasolla. Tärkeäksi mahdollisuudeksi yritysten ja yritystenvälisen toiminnan tehokkuuden ja laadun nostajana on noussut prosessi-integraatio [11]. Integrointitekniikoiden lisäksi integrointiprosessi on tarkastelun ja kehittämisen kohteena [13],[14].

Terveydenhuollon sovellusintegraatiota tutkitaan ja kehitetään Mikko Korpelan vetämässä PlugIT-hankkeessa, jonka rahoittajina ovat TEKES sekä useat ohjelmistotalot ja sairaanhoitopiirit. PlugIT



Kuva 7. Kompponenttisysteemi

tuottaa avoimia ohjelmistorajapintojen määrityksiä sekä niihin liittyviä menetelmiä ja osaamista terveydenhuollon ohjelmistoyrityksille ja niiden asiakkaille. Hankkeen tavoitteena on tukea terveydenhuollon palvelutoimintaa ohjelmistotuotannon palveluketjun kautta, paremmin integroituvien ohjelmistokokonaisuuksien avulla. Hankkeessa on eri ammattiryhmien aitoa yhteistyötä. Mukana on kolme Kuopion yliopiston ja yksi Pohjois-Savon Ammattikorkeakoulun tutkimusyksikkö (kts. http://www.uku.fi/atkk/plugit).

Uusimpana tutkimusaiheena on *käytettävyyden kehittäminen*. Ohjelmiston arkkitehtuuria ja rakennetta parannetaan huomioiden käyttäjien toimintaprosessit, ongelmanratkaisutavat ja oppimisen. Jos olet havainnut käytettävyysongelman, lähetä siitä lyhyt kuvaus Mirja Immoselle (Mirja.Immonen@cs.uku.fi). Kiitos!

6. Loppuyhteenveto

Tätä historiaa kirjoittaessa tuli mukava olo. Paljon me ohjelmistoammattilaiset olemme vuosien aikana oppineet. Olkoonkin, että pyörä on keksitty moneen kertaan, on silti päästy eteenpäin. Tekniikkaa osataan. Entistä parempiin tuloksiin varmasti päästään, jos ohjelmiston toimittajan, integroijan ja asiakkaan yhteistoimintaa kehitetään entisestään. Suomen kaltaisessa pienessä maassa ei kannata sortua vääränlaiseen kilpailuun. Yhteistyö ja korkeatasoinen osaaminen mahdollistavat kansainvälisen ohjelmistoliiketoiminnan lisääntymisen.

Ihmiset tekevät ohjelmistot, ihmiset hyödyntävät ohjelmistoja - ei unohdeta ihmistä.

LÄHDELUETTELO

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

[2] Berztiss A.: *Software Methods for Business Reengineering*, Springer-Verlag, 1996.

[3] Chen P. P.: *The Entity-Relationship Model – Toward a Unified View of Data*, ACM Transactions on Database Systems, Vol. 1, No. 1, March 1976, pp. 9-36.

[4] Codd E.: *A Relational Model of Data for Large Shared Data Banks*, Communications of the ACM, Vol. 13, No. 6, 1970, 377-387.

[5] Fowler M., Scott K.*: UML Distilled, Applying the Standard Object Modelling Language*, Addison-Wesley, 1997.

[6] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[7] Herzum P., Sims O.: *Business Component Factory*, Wiley Computer Publishing, New York, 2000.

[8] Iivari J., Lyytinen K.: *Research on information systems development in Scandinavia – unity in plurality*, Scandinavian Journal of Information Systems 1998: 10 (1&2):135-185.

[9] Kendall A.: *Introduction to Systems Analysis and Design, A Structured Approach*, Elsevier Science Publishers B.V. (North-Holland), ERI 1989.

[10] Kruchten, P.: *The Rational Unified process, an introduction*, Addison-Wesley, 2001.

[11] Linthicum D. S.: *B2B Application Integration e-Business-Enable Your Enterprise*. Addison-Wesley Information Technology Series, USA, 2001

[12] Meyer B.: *Object-oriented Software Construction*, Prentice Hall, 1988

[13] Mykkänen J., Porrasmaa J., Korpela M.: A process for specifying integration for multi-tier applications in healthcare. In: Health Data in the Information Age. Proceedings of the 17th International Congress of the European Federation of Medical Informatics MIE 2002 (25-29 August 2002, Budapest). Surján G, Engelbrecht R, McNair P, eds. p. 691-696. IOS Press, Amsterdam 2002.

[14] Mykkänen J., Tikkanen T., Rannanheimo J., Eerola A., Korpela M.: *Integration of Health Information Systems – Specification Levels and Collaborative Definition*, MIE 2003.

[15] Smolander K.: *On the Role of Architecture in Systems Development*, Academic dissertation, Lappeenrannan teknillinen yliopisto, 2003.

[16] Wilde N., Huitt R., 1992: *Maintenance Support for Object-Oriented Programs*, IEEE Transactions on Software Engineering, Vol. 18, No. 12, Dec 1992, pp. 1038-1044.

[17] Toroi, T., Eerola, A., Mykkänen, J.: *Testing Business Component Systems*, raportti, Kuopion yliopisto, 2000.