Tietojenkäsittelytieteen laitos
Julkaisusarja D
Luentomoniste D-2007-1

Johdatus tietojenkäsittelytieteeseen: Tarinoita tietojenkäsittelytieteen osa-alueilta

Kimmo Raatikainen

Luentomoniste

Helsingin yliopisto

ISBN 978-952-10-4450-2 (PDF)

ACM Computing Reviews (1998) luokitus: A.1

Helsinki, joulukuu 2007 (120 sivua)

Johdatus tietojenkäsittelytieteeseen: Tarinoita tietojenkäsittelytieteen osa-alueilta

Kimmo Raatikainen
Tietojenkäsittelytieteen laitos
Pl 68, 00014 Helsingin yliopisto
Kimmo.Raatikainen@cs.helsinki.fi
http://www.cs.helsinki.fi/kimmo.raatikainen/

Esipuhe

Helsingin yliopistossa tietojenkäsittelytieteen opinnot alkavat kurssilla *Johdatus tietojenkäsittelytieteeseen*, jonka luentomonisteen esipuhetta olet juuri lukemassa. Kurssin tavoitteena on antaa opiskelijoille yleiskuva tietojenkäsittelytieteestä, sen perusmenetelmistä ja käytännöistä sekä perehdyttää heidät alamme keskeisimpään käsitteistöön. Kurssin sisällön ja rakenteen lähtökohtana on ollut Peter J. Denningin kolumni *Great Principles of Computing* (Communications of the ACM, 46, 11, marraskuu 2003, sivut 15–20). Luentomonistetta laatiessani olen käyttänyt Denningin *The Profession of IT* -kolumnien (julkaistu Communications of the ACM -lehdessä vuosina 2001–2005) lisäksi avoimen yhteisön jatkuvasti laajentamaa vapaata WikipediA tietosanakirjaa (`http://en.wikipedia.org/wiki/Main_page`) sekä tietojenkäsittelytieteen eri osa-alueilta pääasiassa ACM:n ja IEEE:n lehdissä julkaistuja artikkeleita.

Denning jakaa tietojenkäsittelytieteen kolmeen pääosaan: 1) tietojenkäsittelyn perusperiaatteet (engl. *great principles of computing*), 2) tietojenkäsittelyn käytännöt (engl. *computing practices*) ja 3) keskeiset teknologiat (engl. *core technologies*). Kurssin painopiste on perusperiaatteissa ja käytännöissä. Tietojenkäsittelytieteen keskeisiin teknologioihin perehdytään koulutusohjelmamme muilla kursseilla.

Omasta kokemuksestani voin kertoa, että monet opiskeluaikanani oppimani teknologiat ovat jo kauan kuuluneet vain tietojenkäsittelyn historiaan. Sen sijaan perusperiaatteet ja -menetelmät ovat mahdollistaneet uusien tietojen ja taitojen oppimisen, mikä on välttämätöntä ammattitaidon ylläpitämiseksi tietojenkäsittelyssä. Vaikka teknologiat tulevat ja menevät, ei niiden opetteleminen kuitenkaan ole turhaa. Ne konkretisoivat perusperiaatteita ja -menetelmiä. Siten ne edesauttavat olennaisten tietojen ja taitojen oppimista.

Ohjeita opiskeluun

Ainakin tietojenkäsittelytieteen pääaineopiskelijoille tämä kurssi lienee ensimmäinen tuntuma yliopisto-opetukseen, joka eroaa olennaisesti lukio-opetuksesta. Kurssilaisten on syytä pitää jatkuvasti mielessä yliopisto-opetuksen yksi keskeisimmistä periaatteista:

> *Professorin tehtävä ei ole opettaa, vaan saada opiskelijat oppimaan.*

Tämän mukaisesti luennoilla käsitellään vain joitakin kiinnostavimpia kysymyksiä. Kaikkea kurssien vaatimuksiin kuuluvaa ei käsitellä luennoilla, vaan jätetään opiskelijoille omatoimisesti opittavaksi.

Luentomonisteessa alaotsikkojen 'Taustamateriaalia' alla on opiskelijoiden omatoimisesti luettavaksi oletetut (luentomonistetta täydentävät) artikkelit ja muu materiaali.

Tutustuminen tietojenkäsittelyn keskeisten termien etymologisiin taustoihin ei kuuluu kurssin muodollisiin vaatimuksiin. Alan ammattilaisten oletetaan kuitenkin tietävän tällaiset asiat. Nämä tarinat kuuluvat alan yleissivistykseen. Tutustu WikipediA:n (`http://en.wikipedia.org/wiki/List_of_computer_term_etymologies`) tarjoamaan aineistoon.

Kiitokset

Kurssin sisällön suunnittelun ja tämän luentomonisteen valmistelussa sain rakentavia ehdotuksia ja arvokasta palautetta useilta kolleegoiltani. Erityisesti haluan kiittää Timo Alankoa, Hannu Erkiötä, Patrik Floréenia, Teemu Kerolaa, Jyrki Kivistä, Markku Kojoa, Tiina Niklanderia, Matti Nykästä, Seppo Sippua, Juha Tainaa, Hannu Toivosta ja edesmennyttä Inkeri Verkamoa.

Syksyn 2005 kurssin ohjaajilta ja opiskelijoilta saatu palaute muokkasi luentomonisteen sen nykyiseen muotoon.

Lopuksi haluan kiittää kurssin seuraavaa luennoijaa, Heikki Lokkia, luentomonisteen huolellisesta läpikäynnistä.

Helsingissä, 5. joulukuuta, 2007

Kimmo Raatikainen

ACM Computing Reviews (1998) luokitus ja aihekuvaukset:
A.1 General Literature: Introductory and Survey

Yleiset termit: tietojenkäsittelytiede

Avainsanat: tietojenkäsittelyn perusperiaatteet, tietojenkäsittelyn menetelmät, tietojenkäsittelyn käytännöt

Sisällysluettelo

LUKU 1

Käsityksiä tietojenkäsittelytieteestä

Tieteenalana tietojenkäsittely on noin 60-vuotias. Tänä aikana tietojenkäsittelytieteen olemuksesta ja sisällöstä on esitetty monenlaisia käsityksiä. Tässä luvussa tarkastelemme, miten käsitys tietojenkäsittelytieteestä on aikojen saatossa muuttunut.

Tietokoneiden, algoritmien, tietorakenteiden vai monimutkaisuuden tutkimista?

Vuosien varrella tietojenkäsittelytiedettä on pidetty muun muassa

- tietokoneiden [Newell, 1967],
- algoritmien [Knuth, 1974],
- tietorakenteiden [Wegner, 1971] ja
- monimutkaisuuden [Dijskstra, 1972]

tutkimisena. Useimmat muutkin 1960- ja 1970-luvuilla esitetyt määritelmät olivat lyhyitä ja selkeitä, mutta puolinaisia. Yleensä esitetty määritelmä korosti sitä tietojenkäsittelyn osa-aluetta, jota määritelmän esittäjä tutki.

1980-luvulle tultaessa määritelmät pidentyivät ja monimutkaistuivat. Alkuaikojen väittely siitä, onko tietojenkäsittelytiede tiedettä vai insinööritaitoa, oli ainakin hetkellisesti päättynyt kompromissiin: tietojenkäsitte-

lytieteessä yhdistyvät sekä tiede että insinööritaito. Näkemykset, että tietojenkäsittely olisi vain matematiikan osa-alue, olivat lähes täysin kadonneet. Tuolloin esitettyjen määritelmien perusongelma on niiden monimutkaisuus. ACM:n komitea [Denning, 1989] määritteli 1980-luvun lopussa tietojenkäsittelytieteen seuraavasti:

> *Tieteenalana tietojenkäsittely tutkii systemaattisesti informaatiota kuvaavia ja muuntavia algoritmisia prosesseja; niiden teoriaa, analysointia, suunnittelua, tehokkuutta, toteuttamista ja soveltamista.*

Määritelmä sisältää vielä 2000-luvun slussa kaiken oleellisen, mutta tietojenkäsittelytieteilijäkin tarvitsee ensilukemalla jonkun tovin määritelmän sulattelussa.

Automatisointi on tietojenkäsittelytieteen perusta

Tietojenkäsittelytiedettä on määritelty myös kuvailemalla tietojenkäsittelytieteen osa-alueita. ACM:n komitea päätyi 1980-luvun lopulla yhdeksään[1] osa-alueeseen: algoritmit ja tietorakenteet, ohjelmointikielet, tietokonearkkitehtuurit, numeerinen ja symbolinen laskenta, käyttöjärjestelmät, ohjelmistotekniikka, tietokannat ja tiedonhaku, tekoäly ja robotiikka sekä ihmisen ja tietokoneen vuorovaikutus. Useissa muissakin lähteissä on vastaavia, edellisestä enemmän tai vähemmän poikkeavia luetteloita. Esimerkiksi rinnakkaisuus ja hajautus, tietoturva, luotettavuus sekä suorituskyvyn arviointi ovat esiintyneet tietojenkäsittelytieteen osa-alueina.

Tietojenkäsittelytieteestä saadaan muodostettua huomattavasti selkeämpi kuva, kun tarkastellaan niitä kaikkein keskeisimpiä kysymyksiä, joita tietojenkäsittelytiede yrittää ratkaista. Stanfordin yliopiston edesmennyt professori George Forsythe [Forsythe, 1968] esitti vuonna 1968, että tietojenkäsittelytieteen peruskysymys on:

> *Mitä voidaan automatisoida?*

[1] Vuoteen 2003 tultaessa lukumäärä oli noussut kolmeenkymmeneen. [Denning, 2003].

Tämä kysymys esiintyy myös vuonna 1980 ilmestyneen yli 900-sivuisen Computer Science and Engineering Research Study raportin [Arden, 1980] otsikkona. Tietojenkäsittelytieteen olemusta pohtinut ACM:n komitea [Denning, 1989] täydensi peruskysymyksen muotoon:

Mitä voidaan (tehokkaasti) automatisoida?

Onko tietojenkäsittelytiede tiedettä?

Vaikka tietojenkäsittelytieteilijät päätyivät jo 1980-luvun lopulla käsitykseen, että tietojenkäsittelytieteessä yhdistyvät tiede ja insinööritaito, niin maallikoiden ja muiden tieteenalojen harjoittajien keskuudessa esiintyy vielä käsityksiä, että tietojenkäsittelytiede ei olisi tiedettä. Tätä käsitystä vastaan Peter J. Denning argumentoi kolumnissaan *Is Computer Science Science?* (Communications of the ACM, 48, 4, huhtikuu 2005, sivut 27–31), johon alla oleva teksti pääasiassa perustuu.

Tietojenkäsittelytieteessä yhdistyvät tiede, insinööritaito ja matematiikka ainutlaatuisella tavalla. Jotkut tietojenkäsittelytieteen osa-alueet ovat ensisijaisesti tiedettä, kuten esimerkiksi kokeelliset algoritmit, kokeellinen tietojenkäsittelytiede ja laskennallinen tiede. Toiset alueet ovat pääasiassa insinööritaitoa, kuten esimerkiksi suunnittelu, ohjelmistotekniikka ja tietokoneiden suunnittelu. Jotkut alueet ovat matemaattispainotteisia, kuten esimerkiksi laskennallinen vaativuus, matemaattiset ohjelmistot ja numeerinen analyysi. Kuitenkin useimmilla osa-alueilla tiede, insinööritaito ja matematiikka ovat tasa-arvoisia elementtejä. Siten tietojenkäsittelytiede on kuin urheilumaailman triatlon: kaikkea kolmea on hallittava vähintään kohtuullisen hyvin.

Nykyinen tieteen paradigma on peräisin englantilaiselta Francis Baconilta (1561–1626). Siinä tiede on prosessi, jossa muodostetaan väittämiä (hypoteesejä), joiden paikkansapitävyyttä koetellaan kokeilla tai havainnoilla. Hypoteesistä, joka läpäisee koettelemukset, muodostuu ilmiötä kuvaava malli, joka sekä selittää havainnot että ennustaa ilmiön tulevaa käyttäytymistä. Tietojenkäsittelytiede käyttää tätä lähestymistapaa tutkiessaan informaatiota systemaattisesti kuvaavia ja käsitteleviä prosesseja.

Tieteestä keskusteltaessa usein vielä erotellaan perustutkimus (engl. *pure science, basic research*) ja soveltava tutkimus. Soveltavassa tutkimuksessa keskitytään tietämykseen, josta on välitöntä hyötyä. Toisessa

Taulu 1.1: Tieteen ja taitamisen tunnusmerkkejä

Tiede	**Taitaminen**
periaatteet	käytäntö
keskeiset yhtäläisyydet	taidokas suoritus
selitys	toimenpide
löytö	keksintö
analyysi	synteesi
erittely	konstruktio

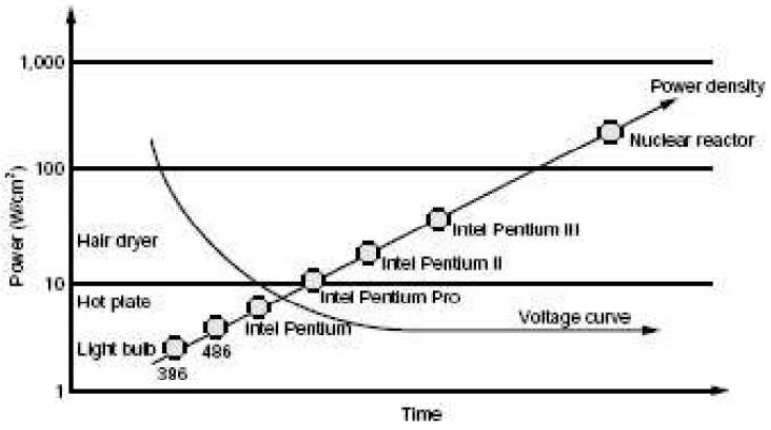
tieteen jaottelussa puhutaan eksakteista ja kuvailevista tieteistä. Eksaktit tieteet käyttävät havaintoja, mittauksia ja kokeita ennustamiseen ja verifiointiin.

Taulukko 1.1 vertaa tieteen ja taitamisen tunnusmerkkejä. Ohjelmointi, suunnittelu, ohjelmistotekniikka, mallintaminen ja käyttöliittymien tekeminen ovat tietojenkäsittelyn taitolajeja. Esteettisyyden mukaanottaminen laajentaa taitolajien joukkoa mm. tietokonegrafiikalla, animoinnilla ja peleillä.

Missä ovat tieteenrajat?

Kirjassaan *The End of Science* (Addision-Wesley, 1996) John Horgan väitti, että uudet tieteelliset löydöt vaativat yhä suuremman monimutkaisuuden hallintaa. Horganin pääasiallisen kritiikin kohteena olivat eksaktit luonnontieteet eli fysiikka, kemia, tähtitiede, jne. Horganin perusväittämä oli, että yhä suuremmilla panostuksilla eksakteihin tieteisiin saadaan yhä pienempiä läpimurtoja. Tammikuussa 2004 Horgan julkaisi IEEE:n Computer-lehdessä [Horgan, 2004] päivityksen *The End of Science Revisited.* Tässä kirjoituksessa Horgan käsitteli myös tietojenkäsittelytiedettä, erityisesti kompleksisuutta ja tekoälyä.

Horgan nosti esille myös nk. Mooren lain [Schaller, 1997], jonka mukaan prosessorien käsittelynopeus kaksinkertaistuu joka 18 kuukausi. Tämä laki on pitänyt paikkansa jo useita vuosikymmeniä, mutta rajat saattavat olla piakkoin vastassa. Kuvassa 1.1 [Herring, 2000] on kuvattu Intelin prosessorien energiatiheyden kehitys. Mikäli kehitys jatkuu samanlaisena, niin Intelin prosessorien energiatiheys olisi melko pian



Kuva 1.1: Prosessorien energiatiheys [Herring, 2000]

sama kuin ydinvoimalan reaktorin ytimessä.

Kolumnissaan *Is Computer Science Science?* Peter J. Denning [Denning, 2005] kyseenalaistaa Horganin johtopäätöset tietojenkäsittelytieteen osalta. Denningin mukaan tietojenkäsittelytiede etenee eri polkua. Erityisesti laskennallinen tiede, eli muiden tieteiden tietojenkäsittelyltä tarvitsema tuki pitää tietojenkäsittelytieteen kehityksen hengissä. Kolumnissaan Denning viittaa Rosenloomin artikkelin *A New Framework for Computer Science and Engineering* [Rosenbloom, 2004] johtopäätökseen: Tietojenkäsittelytieteen uudet ja jatkuvasti laajentuvat yhteydet muiden tieteenalojen ongelmien ratkaisuihin takaavat, ainakin toistaiseksi, tietojenkäsittelytieteen jatkuvan kehityksen.

Taustamateriaalia

Denning, P. J. (2003) Great Principles of Computing. Communications of the ACM, 46, 11: 15–20.

Denning, P. J. (2005) Is Computer Science Science? Communications of the ACM, 48, 4: 27–31.

Horgan, J. (2004) The End of Science Revisited. IEEE Computer, 37, 1: 37–43.

Lähteitä

Arden, B. (toim.) (1980) What Can Be Automated? Report of the NSF Computer Science and Engineering Research Study (COSERS). MIT Press, Cambridge, Mass.

Denning, P. J. et al. (1989) Computing as a discipline. Communications of the ACM, 32, 1: 9–23.

Dijkstra, E. W. (1972) Notes on Structured Programming. Teoksessa Structured Programming; O.-J. Dahl, E. W. Dijkstra ja C. A. R. Hoare (toim.). New York: Academic Press.

Forsythe, G. E. (1968) Computer science and education. Proceedings of IFIP Congress 68, osa 2: 1025–1039. Amsterdam: North-Holland/IFIP.

Herring, C. (2000) Microprocessors, Microcontrollers, and Systems in the New Millennium. IEEE Micro, 20, 6: 45–51.

Horgan, J. (1996) The End of Science. Addison-Wesley.

Knuth, D. E. (1974) Computer science and its relation to mathematics. American Mathematical Monthly, 81, 4: 323–343

Newell, A., Perlis, A. J. ja Simon, H. (1967) What is computer science. Science, 157, 3711: 1373–1374.

Rosenbloom, P. (2004) A New Framework for Computer Science and Engineering. IEEE Computer, 37, 11: 23–28.

Schaller, R. R. (1997) Moore's law: past, present and future. IEEE Spectrum, 34, 6:52–59.

Wegner, P. (1971) A view of computer science education. Teoksessa Proceedings of IFIP Congress 71, osa 2: 1515–1522. Amsterdam: North-Holland/IFIP.

LUKU 2

Tietojenkäsittelytieteen kokovartalokuva

Olemassaolonsa aikana tietojenkäsittely on murtautunut merkittäväksi tekijäksi (länsimaissa) lähes kaikilla inhimillisen elämän, tieteen ja tutkimuksen saroilla. Erityisesti viimeksi kuluneiden 10–15 vuoden aikana tietojenkäsittelytiede on arkipäiväistynyt, tunkeutunut myös koteihin ja ihmisten vapaa-aikaan. Kaukana on aika, jolloin (ehkä) tunnetuin tietojenkäsittelyn virhe-ennuste lausuttiin:

> *Viisi tietokonetta riittää rakaisemaan maailman kaikki laskennalliset ongelmat.* -Thomas Watson Senior, IBM:n pääjohtaja.

Edellisessä luvussa kerrottiin lyhyesti, että tietojenkäsittelytieteen kuva on vuosien saatossa olennaisesti muuttunut. Uusin yritys hahmottaa tietojenkäsittelytieteen olemusta on Peter J. Denningin (2003) perusperiaatteisiin perustuva tietojenkäsittelyn ”muotokuva”.

Denningin mukaan tietojenkäsittelyn perusperiaatteet voidaan jakaa kahteen pääryhmään:

1. tietojenkäsittelyn rakenteiden ja käyttäytymisen periaatteet[1] sekä

2. suunnittelun periaatteet.

Lähestymistapansa Denning on lainannut fysiikasta [Feynman, 1970], biologiasta [Hazen & Trefil, 1991] ja tähtitieteestä [Sagan, 2002]. Nämä

[1] Näitä Denning kutsuu fysikaalisia tieteitä lainaten tietojenkäsittelyn mekaniikoiksi.

vakiintuneet tieteenalat perustavat rakenteensa muutamaan perusperiaatteeseen.

Denningin rakennelmassa periaatteet, käytännöt ja ydinteknologiat kattavat käyttötilanteet ja niiden taustat. Minkä tahansa periaatteen ymmärtäminen on mahdotonta (tai ainakin hyvin hankalaa) tuntematta taustoja:

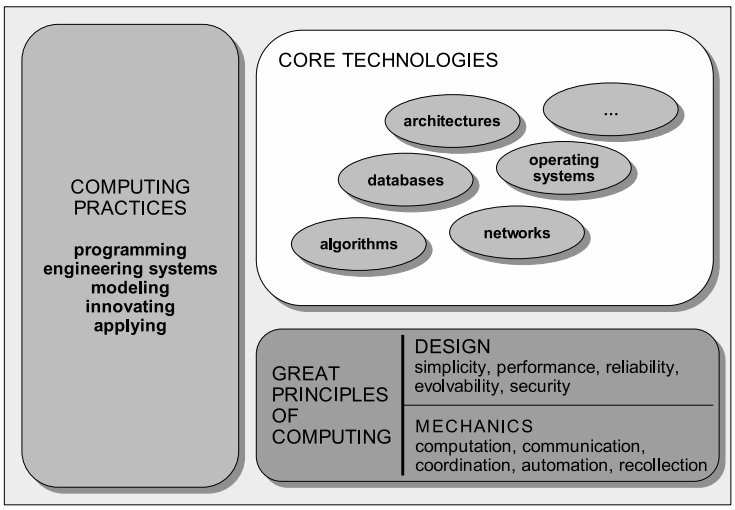
- mistä se (periaate) on peräisin,
- miksi se on tärkeä,
- miksi se toistuu eri yhteyksissä,
- miksi se on yleispätevä,
- miksi se on välttämätön.

Useimmat tietojenkäsittelytieteen ydinteknologiat on aikoinaan kehitetty eri sovellusalueiden tarpeisiin. Siksi meillä on lukuisia ohjelmointikieliä, kuten Ada, Algol, C, Cobol, C++, Fortran, Java, Lisp, Pascal, Perl, Prolog. Kukin niistä on aikoinaan kehitetty tietyn sovellusalueen tarpeisiin. Siksi kukin niistä edelleen peittoaa soveltuvuudessaan kaikki muut omalla erikoisalueellaan.

Samalla tavalla tietojenkäsittelyssä on erilaisia ratkaisuja lähes kaikilla tietojenkäsittelytieteen osa-alueilla. Alan kirjallisuudesta löytyy lukuisia esimerkkejä väittelyistä eri lähestymistapojen paremmuudesta. Tuntematta taustoja ja käyttötarkoituksia väittelyissä käytetyt argumentit voivat tuntua käsittämättömiltä. Jotakin tiettyä ratkaisua tai lähestymistapaa puolustettaessa on syytä muistaa, että

> *one size does not fit all* tai
> *jos vasara on ainut työkalu, niin kaikki ongelmat näyttävät nauloilta.*

Tässä kohdassa on syytä huomauttaa suomenkielisestä terminologiasta. Termillä *laskenta* on useita vastineita englannissa. Termi *computing* käännetään useimmiten termillä *laskenta*, vaikka *computer* kääntyy tietokoneeksi ja *computer science* tietojenkäsittelytieteeksi. Tietojenkäsittelytieteen laskentaa ei tule sekoittaa matematiikan laskentaan (engl. *calculus*) tai kauppatieteiden laskentaan (engl. *accounting*). Termi *accounting* esiintyy myös tietojenkäsittelyssä tarkoittaen resurssien käyttötietojen keräämistä mahdollista laskutusta (engl. *billing*) varten.



Kuva 2.1: Tietojenkäsittelytieteen kokovartalokuva [Denning, 2003]

Tietojenkäsittelyn käytännöt (*Computing practices*): ohjelmointi (*programming*), järjestelmien rakentaminen (*engineering systems*), mallintaminen (*modeling*), innovointi (*innovating*) ja soveltaminen (*applying*).

Ydinteknologiat (*Core technologies*): algoritmit (*algorithms*), tietokannat (*databases*), arkkitehtuurit (*architectures*), tietoverkot (*networks*), käyttöjärjestelmät *operating systems*), jne.

Tietojenkäsittelyn perusperiaatteet (*Great principles of computing*):

Suunnittelu (*Design*): yksinkertaisuus (*simplicity*), suorituskyky (*performance*), luotettavuus (*reliability*), kehitettävyys (*evolvability*) ja tietoturva (*security*).

Tietojenkäsittelyn mekaniikat (*Mechanics*): laskenta (*computation*), kommunikointi (*communication*), koordinointi (*coordination*), automatisointi (*automation*) ja muistaminen (*recollection*).

Denningin rakentama tietojenkäsittelytieteen kokovartalokuva on hahmoteltu kuvassa 2.1. Seuraavissa luvuissa tutustumme tarkemmin kaavion osiin:

tietojenkäsittelyn mekaniikat: lait ja yleisesti toistuvat ilmiöt, jotka kattavat tietojenkäsittelyn toiminnot,

suunnittelun periaatteet: tietojenkäsittelyn suunnittelun vakiintuneet tavat ja

tietojenkäsittelyn käytännöt: tietojenkäsittelyjärjestelmien vakiintuneet rakentamis- ja käyttöönottotavat.

Kuvan 2.1 kaaviosta puuttuu sovellusalueiden, ydinteknologioiden, suunnittelun periaatteiden ja tietojenkäsittelyn mekaniikkojen vaikutukset tietojenkäsittelyn käytäntöihin. Näitä vaikutuksia voidaan luonnehtia seuraavilla peruskysymyksillä:

Sovellusalueet: Miten toimia yhdessä sovellusalueiden edustajien kanssa suunniteltaessa eri sovellusalueiden tietojenkäsittelyä?

Ydinteknologiat: Miten suunnitella käsittely, joka tukee eri sovellusten yhteisiä tarpeita?

Suunnittelun periaatteet: Miten organisoida toimiva tietojenkäsittelyn toteutus?

Tietojenkäsittelyn mekaniikat: Miten tietojenkäsittely toimii?

Denning väittää, että hänen muotoilema perusperiaatteisiin ja käytäntöihin perustuva kuva tietojenkäsittelystä edistää informaatioteknologian taustalla olevan tieteen ja insinööritaidon syvällisempää ymmärtämistä. Tosin väitteensä tueksi Denningin esittää vain samankaltaisen lähestymistavan kuin muissa luonnontieteissä. Esitetyn kuvan väitetään parantavan huomattavasti tietojenkäsittelijöiden kykyä keskustella alansa riskeistä, hyödyistä, mahdollisuuksista ja rajoitteista maallikoiden kanssa.

Kuva 2.1 korostaa tietojenkäsittelyn toimintaan suuntautumista (engl. *action-orientation*). Tietojenkäsittelyllä on lukuisia asiakkaita. Tietojenkäsittelyn käyttökohteet ovat yhtä tärkeitä kuin tietojenkäsittelyn menetelmät (tietojenkäsittelyn mekaniikat). Denningin esittämä rakenne tuo esille tietojenkäsittelyn ammattilaiselta edellytettäviä kykyjä ja taitoja (engl. *competences*), jotka koostuvat tietojenkäsittelyn mekaniikkojen, suunnittelun, tietojenkäsittelyn käytäntöjen, ydinteknologioiden ja sovellusten hallinnasta.

Taustamateriaalia

Denning, P. J. (2003) Great Principles of Computing. Communications of the ACM, 46, 11, marraskuu 2003, sivut 15–20.

Lähteitä

Feynman, R. (1970) *Lectures in Physics.* Addison-Wesley.

Hazen, R. & Trefil, J. (1991) *Science Matters.* Anchor.

Sagan, C. (2002) *Cosmos.* Random House.

LUKU 3

Tietojenkäsittelijän ammattietiikka

Johdantoluvussa ’Käsityksiä tietojenkäsittelytieteestä’ tietojenkäsittelytieteen peruskysymykseksi nostettiin *Mitä voidaan (tehokkaasti) automatisoida.* Omissa kolumneissani [Raatikainen, 1991, 2002] olen esittänyt, että tehokkuuden lisäksi luotettavuus on nostettava vähintään yhtä tärkeäksi automatisoinnin attribuutiksi kuin tehokkuus. Lisäksi vaadin, että tietojenkäsittelytieteilijän on myös kysyttävä, mitä saa automatisoida. Näiden pohdintojen lisäksi tässä luvussa käydään läpi ACM:n ja IEEE:n ohjelmistotekniikan eettiset ohjeet [Gotterbarn, Miller & Rogerson, 1999] sekä Tietotekniikan liiton julkaisema tietotekniikan etiikan ohjeisto (versio 3) [Tietotekniikan liitto, 2002].

Tietojenkäsittelytieteessä on myös kysyttävä, mitä saa automatisoida

Kysymykset *Mitä voidaan*, *Miten voidaan* ja *Miten hyvin osataan* heijastavat länsimaisen kulttuuriperinnön mukaista asennoitumista tieteen ja tekniikan kehitykseen, jonka oikeutusta ei ole ollut tapana pohtia. Yleensä pohdinta on kuitattu korkeintaan ylimalkaisella toteamuksella, että kansakunnan on välttämättä pysyttävä mukana teknis-tieteellisessä kehityksessä.

Kun kysymystä, onko jokin asia oikein tai oikeutettu, ei selkeästi esitetä, jäävät perimmäiset päämäärät yleensä hyvin hämäriksi. Tällöin teknin en kehitys muodostuu helposti itsetarkoitukseksi. Ilmiö tunnetaan myös

nimellä teknologinen imperatiivi.

Teknologisen imperatiivin oikeutusta on selitetty juutalais-kristillisellä maailmankuvalla, jossa ihminen on luomakunnan kruunu. Tietojenkäsittelytieteenkin on uskottu osaltaan toteuttavan Francis Baconin unelmaa, jossa tieteen avulla tapahtuva luonnon alistaminen ihmisen herruuteen johtaa ihmiskunnan (kadotettuna olleeseen) paratiisiin. Kuitenkaan erityisesti toisen maailmansodan jälkeen elinympäristössämme kiihtyneet muutokset eivät ole kovin vakuuttava näyttö baconilaisen kehitysoptimismin oikeutuksesta. Luonnon alistaminen onkin ehkä muuttunut luonnon peruuttamattomaksi tuhoamiseksi.

Tieteellisen totuuden etsintää on perinteisesti pidetty tieteen oikeutuksen riittävänä perusteena. Tällöin lähtökohtana on näkemys, että tietämys koostuu löydetyistä totuuksista. Siten voidaan kognitiivisin perustein pitää tieteellisen tiedon etsintää sinänsä arvokkaana ja mahdollisesti myös oikeutettuna.

Tieteenalana tietojenkäsittelytiede tutkii systemaattisesti informaatiota kuvaavia ja muuntavia algoritmisia prosesseja, niiden teoriaa, analysointia, suunnittelua, tehokkuutta, toteuttamista ja soveltamista. Siten tietojenkäsittelytieteen tutkimuskohde ei ole olemassa ihmiskunnasta riippumatta. Systemaattisesti informaatiota kuvaavat ja muuntavat prosessit ovat ihmiskunnan tuote. Ne vaikuttavat ihmiskunnan kehitykseen, mutta myös ihmiskunta voi vaikuttaa niiden kehitykseen. Tässä suhteessa tietojenkäsittelytiede on lähempänä yhteiskunta- ja insinööritieteitä kuin luonnontieteitä. Siten tietojenkäsittelytieteen on haettava oikeutuksensa samanlaisista lähtökohdista kuin yhteiskunta- ja insinööritieteiden.

Oikeutuksen perustelut edellyttävät tietämyksen olemuksen erittelyä. Perinteisesti tieto on jaettu empiiriseen (kokemusperäiseen) ja rationaaliseen (pääteltyyn) tietoon. Tietämyksen tuottamisen ja hyväksikäytön etiikan kannalta tiedon syntytapaan perustuva jaottelu ei ole riittävä. On tarkasteltava myös tietämyksen omistusoikeutta, julkistamisvelvollisuutta, jne.

Kun tarkastellaan ihmisen ja ihmiskunnan henkisen työn materialisoituneita tuloksia, niin tietämys koostuu sekä tekijän tiedosta että Durkheimin sosiaalisten tosiasioiden kaltaisesta yliyksilöllisestä tiedosta. Periaatteessa tiedon subjektilla on täysi valta ja vastuu tekijän tiedosta. Yliyksilöllinen tieto, josta kellään yksittäisellä tutkijalla ei voi olla tyhjentävää tekijän tietoa ja johon on suhtauduttava kuten tosiasioihin, on vallan ja vastuun osalta ongelmallinen. Kaikilla on valta ja vastuu, mutta sen osista jollakin on joskus ollut täysi valta ja vastuu. Julkistaessaan tekijän

tietonsa tutkija luopuu vallastaan. Kuitenkaan ei ole itsestään selvää, että hän samalla vapautuu vastuustaan.

Tietojenkäsittelytiede on voimakkaasti käytäntöön suuntautunut tieteenala, sillä tieteenalan peruskysymykseen sisältyvä automatisointi liittyy aina käytäntöön. Näin ollen tietojenkäsittelytieteen legitimaatio ei voi perustua vain tosiasioiden löytämiseen. Legitimaatiopohjaksi ei myöskään riitä pelkkä epämääräinen teknis-tieteellinen edistys. Oikeutus edellyttää myös mahdollisimman perusteellista tietoisuutta tietojenkäsittelyn vaikutuksista. Siksi tietojenkäsittelytieteilijän olisi tarkasteltava myös kysymyksiä:

> *Mitä saa automatisoida?* ja *Mitä pitäisi automatisoida?*

Nämä kysymykset ilmentävät tietojenkäsittelytieteen tarkoituksenhakuisuutta. Ongelma on, kenen tarkoituksia ja tavoitteita palvellaan.

Yhteiskunnallisten vaikutusten arviointi

Suomessa havaitsemani keskustelu tietojenkäsittelyn vastuusta on ollut vähäistä. Huomaamilleni puheenvuoroille on ollut tyypillistä joko epämääräiset viittaukset teknis-tieteelliseen edistykseen tai yltiöoptimistiset utopiat tulevasta paratiisista. Minuunkin tosin saattaa päteä Grace Hertleinin [Hertlein, 1990] huomio:

> *Yes, there are other significant papers and selections*[1] *on this topic—but we have not read nor heeded them. Perhaps we only read scientific journals!*

Tietojenkäsittelytieteilijöistä yhteiskunnallisten vaikutusten tutkiminen saattaa osittain tuntua epätieteelliseltä politikoinnilta. Tietojenkäsittelytieteilijät, jotka pitäytyvät usein matematiikan ja luonnontieteiden tutkimustraditioissa, eivät ole tottuneet yhteiskuntatieteille välttämättömään lähtökohtien erittelyyn ja problematisointiin.

Lähtökohtien analysoinnin puute vaivaakin osaa tietojenkäsittelyn yhteiskunnallisia vaikutuksia käsittelevää tutkimusta. Abbe Mowshowitzin

[1] Hertlein käsitteli tässä yhteydessä tietojenkäsittelytieteilijöitä ja yhteiskunnallista vastuuta. Esikuvakseen Hertlein mainitsee Edmund C. Berkeleyn ja hänen kirjansa The Computer Revolution [Berkeley, 1962].

[Mowshowitz, 1981] mukaan lähtökohtien itsestäänselvyys vaivaa erityisesti teknisismiä, jossa tietokonetta pidetään oletusarvoisesti edistyksen instrumenttina. Hän erittelee viisi erilaista lähestymistapaa: teknisismi, edistyksellinen individualismi, elitismi, pluralismi ja radikaali kritisismi. Tämä jaottelu on tuskin kaikenkattava ja yksikäsitteinen, mutta se havainnollistaa hyvin lähtökohtia, jotka usein ovat vain implisiittisiä.

Lähtökohtien erilaisuutta voidaan selittää myös sosiologian erilaisten metodologisten virtausten perusteella. Erojen taustalla on kuitenkin se vääjäämätön tosiasia, että useimmiten spekulaatiot ovat ainoa mahdollinen lähestymistapa, koska uutta teknologiaa, jonka vaikutuksia halutaan arvioida, ei vielä ole rakennettu tai otettu käyttöön. Tällöin joudutaan ainakin implisiittisesti turvautumaan teoreettisiin malleihin, jotka kuvaavat tietojenkäsittelyn mahdollisia vaikutuksia [Kling, 1980]. Rob Klingin artikkelin ohella erinomaisia katsauksia ovat mm. Attewell ja Rule [Attewell & Rule, 1984] sekä kokoomateokset *Perspectives on the Computer Revolution* [Pylyshyn & Bannoin, 1989] ja *Computerization and Controversy* [Dunlop & Kling, 1991].

Automatisointi on sekä riski että mahdollisuus

Tieteentekemiseen liittyy aina jonkinasteinen eettinen vastuu tulosten käytöstä. Tietojenkäsittelytieteen, missä automatisointi useimmiten nivoo perustutkimuksen, soveltavan tutkimuksen ja tuotekehittelyn kiinteäksi kokonaisuudeksi, tulisi avoimesti arvioida omia vaikutuksiaan. Automatisointi sisältää aina sekä hyväksikäytön mahdollisuuden että väärinkäytön riskin. Tieteentekijän omantunnon kannalta on ensiarvoisen tärkeää, että hän erittelee toistuvasti toimintansa seurauksia mahdollisimman pitkälle.

Jotta tietojenkäsittelytieteilijä pystyisi välttämään Oppenheimerin tragedian [Oppenheimer, 1955] on tietojenkäsittelytieteilijän eriteltävä tieteentekemisen seurauksia. Vaikka tietojenkäsittelytieteen sovellusalue olisi ei-sotilaallinen, voivat seuraukset automatisoidusta päätöksenteosta olla tuhoisat. Siten Lewis Mumfordin [Mumford, 1971] maalaama kauhukuva

> *The most disastrous result of automation, then, is that its final product is Automated or Organization Man: he who takes all his orders from the system, and who, as scientist, engineer, expert, administrator, or, finally, as consumer and subject, cannot conceive of any departure from the system.*

on tietojenkäsittelytieteen saavutusten seurauksena entistä ajankohtaisempi, kun Mumfordin Automated Man korvataan asiantuntijajärjestelmäksi kutsulla ohjelmalla. Myös Bertolt Brecht on näytelmässään Galilein elämä konkretisoinut tieteentekijän asemaa:

> Galilei: *... Einige Jahre lang war ich ebenso stark wie die Obrigkeit. Und ich überlieferte mein Wissen den Machthabern, es zu gebrauchen, es nicht zu gebrauchen, es zu mißbrauchen, ganz wie es ihren Zwecken diente.*

ACM:n ja IEEE:n ohjelmistotekniikan eettiset ohjeet

ACM ja IEEE:n Computer Society ovat jo pitkään toimineet ohjelmistotekniikan ammattimaisuuden edistämiseksi. Vuoden 1999 lopulla valmistunut ohjelmistotekniikan eettiset ohjeet ja ammatilliset käytännöt (*Software Engineering Code of Ethics and Professional Practise*) [Gotterbarn, Miller & Rogerson, 1999] on yksi ammattimaisuuden kulmakivi. Eettisissä ohjeissa kuvataan ohjelmistotekniikan opetuksen ja harjoittamisen eettiset ja ammatilliset velvoitteet. Ohjeet heijastavat niitä odotuksia, jotka yhteiskunta, kolleegat ja suuri yleisö odottavat ohjelmistoammattilaisten täyttävän.

Ohjelmistojärjestelmien kehittäjinä ohjelmistoammattilaisilla on erinomaiset madollisuudet aikaansaada hyvää tai aiheuttaa harmia, edistää muiden madollisuuksia aikaansaada hyvää tai aiheuttaa harmia, tai vaikuttaa muiden madollisuuksiin aikaansaada hyvää tai aiheuttaa harmia. Jotta ohjelmistoammattilaiset pystyisivät mahdollisimman hyvin vakuuttumaan, että heidän työpanoksensa käytetään hyvän aikaansaamiseksi, niin heidän on sitouduttava edistämään ohjelmistojen laatimisen muodostumista hyödylliseksi ja arvostetuksi ammatiksi.

ACM:n ja IEEE:n eettiset ohjeet sisältävät kahdeksan periaatetta, jotka liittyvät ohjelmistoammattilaisen käyttäytymiseen ja päätöksentekoon ammatissaan. Eettiset ohjeet on luettava kokonaisuutena. Yksittäisiä osia ei pidä käyttää selittämään tehtyjä virheitä. Ohjeet eivät suinkaan kata kaikkia mahdollisia tilanteita, joissa ohjelmistoammattilainen joutuu arvioimaan oman toimintansa oikeututusta. Siksi eettiset ohjeet eivät ole algoritmi, joka generoisi eettisesti hyväksyttävät päätökset ja valinnat.

Eettisiä jännitteitä on parempi tarkastella kokonaisvaltaisesti perusperiaatteiden kautta kuin sokeasti luottaa yksityiskohtaisiin säädöksiin. ACM:n ja IEEE:n eettisten ohjeiden kahdeksan periaatteen tarkoituksena on saada ohjelmistoammattilaiset

- ajattelemaan oman työnsä laaja-alaisia vaikutuksia,
- tarkastelemaan, kohtelevatko he ja heidän kolleegansa muita ihmisiä riittävällä kunnioituksella,
- arvioimaan, miten suuri yleisö, jos se olisi tarpeeksi hyvin informoitu, suhtautuisi heidän päätöksiinsä,
- analysoimaan, miten heidän päätöksensä vaikuttavat vähempiosaisiin ja
- arvioimaan omien toimenpiteidensä hyväksyttävyyttä.

Kaikissa näissä tilanteissa yleinen terveys, turvallisuus ja hyvinvointi ovat ensisijaisia.

Alla on periaatteiden lyhyet kuvaukset. Kaikkien ohjelmistoammattilaisten on syytä tutustua myös periaatteiden yksityiskohtaisiin kuvauksiin, jotka erittelevät kutakin periaatetta monipuolisesti. Tässä luentomonisteessa käydään läpi vain *yleisen edun periaate.*

Yleinen etu. Ohjelmistoammattilaiset toimivat aina yleisen edun mukaisesti.

Asiakas ja työnantaja. Ohjelmistoammattilaiset toimivat tavalla, joka vastaa asiakkaan ja työnantajan etuja, mutta on sopusoinnussa yleisen edun kanssa.

Tuote. Ohjelmistoammattilainen varmistaa, että hänen tuotteensa ja niihin liittyvät muutokset täyttävät parhaalla mahdollisella tavalla ammattimaisen toiminnan vaatimukset.

Harkinta. Ohjelmistoammattilaiset ovat johdonmukaisia ja säilyttävät riippumattomuutensa ammatillisissa mielipiteissään.

Johto. Ohjelmistotekniikan johtajat ja esimiehet sitoutuvat ohjelmistojen kehityksen ja ylläpidon eettisesti kestävään johtamiseen ja edistävät sitä.

Ammatti. Ohjelmistoammattilaiset edistävät ammattinsa rehellisyyttä ja mainetta yleisen edun mukaisesti.

Työtoverit. Ohjelmistoammattilaiset ovat työtovereilleen rehtejä ja kannustavia.

Oma toiminta. Ohjelmistoammattilaiset ovat sitoutuneet oman ammattitaitonsa elinikäiseen kehittämiseen ja edistävät eettisesti kestäviä ammatillisia käytäntöjä.

Yleisen edun periaate

Yleisen edun periaatetta ACM:n ja IEEE:n eettiset ohjeet tarkentavat seuraavilla kahdeksalla kohdalla.

1. Ohjelmistoammattilaiset ottavat täyden vastuun omasta työstään.
2. Ohjelmistoammattilaiset sovittavat yhteen oman, työnantajan, asiakkaan ja käyttäjien edut yleisen edun kanssa.
3. Ohjelmistoammattilaiset hyväksyvät ohjelmiston luovutuksen vain, jos heillä on hyvin perusteltu uskomus, että ohjelmisto on turvallinen, täyttää määritykset ja läpäisee asianmukaiset testit, ei vähennä elämän laatua tai yksityisyyttä eikä vahingoita ympäristöä.
   Työn perimmäisten vaikutusten on oltava yleisen edun mukaisia.
4. Ohjelmistoammattilaiset paljastavat asianmukaisille henkilöille tai viranomaisille minkä tahansa todellisen tai mahdollisen käyttäjiin, suureen yleisöön tai ympäristöön kohdistuvan uhan, jonka he kohtuudella uskovat liittyvän ohjelmistoon tai sen dokumentaatioon.
5. Ohjelmistoammattilaiset osallistuvat aktiivisesti ohjelmistojen, niiden installoinnin, ylläpidon tai dokumentaation aiheuttamien vakavien julkisten huolenaiheiden selvittelyyn.
6. Ohjelmistoammattilaiset ovat rehtejä ja välttävät harhakuvia kaikissa, erityisesti julkisissa, lausunnoissaan, jotka koskevat ohjelmistoja sekä niihin liittyviä dokumentteja, menetelmiä tai työvälineitä.
7. Ohjelmistoammattilaiset ottavat huomioon fyysisestä rajoittuneisuudesta, käytettävistä voimavaroista, taloudellisesta eriarvoisuudesta ja muista syistä johtuvat tekijät, jotka voivat vähentää ohjelmistojen saatavuuden hyötyjä.
8. Ohjelmistoammattilaiset käyttävät ammatillisia taitojaan vapaaehtoistyössä ja osallistuvat alan julkiseen kouluttamiseen.

Tietotekniikan liiton tietotekniikan etiikan ohjeisto

Tietotekniikan liiton etiikan työryhmä[2] julkisti 2002 tietotekniikan ammattilaisen eettiset ohjeet [Tietotekniikan liitto, 2002] tukemaan tietotekniikan ammattilaisia heidän työssään kohtaamiensa eettisten ongelmien ratkaisemisessa. Tavoitteena on tuoda esille eettisiä toimintatapoja ja auttaa käsittelemään moraalisia ongelmia.

Ohjeiston tarkoituksena on syventää tietotekniikan ammattilaisuuden eettistä ulottuvuutta. Aivan samoin kuin ACM:n ja IEEE:n eettiset ohjeet TTL:n ohjeisto ei yritä olla täydellinen. Koska eettisiin ongelmiin ei voida ennalta antaa täydellisiä ohjeita, TTL:n ohjeiston kohtia ei tule lukea ehdottomina totuuksina vaan suunnannäyttäjinä. Vastuu päätöksistä on jokaisella itsellään.

Alla oleva teksti on suora kopio TTL:n ohjeistosta.

Valta ja vastuu. Tietotekniikan ammattilainen ei saa käyttää asemaansa väärin. Hänen on kannettava vastuunsa, joka näkyy tekoina ja toimina. Tieto on valtaa ja tiedon käyttäminen vaatii viisautta kuten muukin vallankäyttö.

Tieto ja kokemus. Ammattilaisen on tunnettava rajansa: tiedettävä mitä osaa ja myös mitä ei osaa. Kehittyvällä alalla ammattilaisen on ylläpidettävä osaamistaan. Hänen on tunnettava työtään koskeva, esimerkiksi tietosuojaan liittyvä, lainsäädäntö. Ammattilainen ei panttaa tietoa, vaan pyrkii lisäämään omaa ja muiden osaamista ja jakaa omat kokemuksensa muulle yhteisölle. Ammattilainen suojaa kuitenkin asiakkaan omat asiat ja muut suojaamista vaativat tiedot.

Saadessaan kritiikkiä työstään, aiheellisesti tai aiheetta, ammattilainen osaa ottaa sen vastaan ja ottaa tapahtuneesta oppia.

Asenne. Ammattilainen ei toimi vain itseään vaan myös muita varten. Hän ottaa huomioon toimintansa kohteiden näkökannan. Hän ei anna valtaa ahneudelle ja piittaamattomuudelle. Hän ymmärtää myös, että hänen työllään on merkitystä vain muiden ihmisten kautta.

[2] `www.tt-tori.fi/Etiikka`: työryhmälle voi lähettää palautetta osoitteeseen `IT_etiikan.tyoryhma@ttlry.fi`.

Viestintä. Ammattilainen ymmärtää viestinnän merkityksen. Hän kommunikoi asiakkaan kanssa, dokumentoi tekemisensä ja tiedottaa toimistaan kaikille asianomaisille.

Ammattilaisen on pyrittävä viestimään selväkielisesti ja määrittelemään tarvittaessa käyttämänsä käsitteet. Viestinnän tavoitteena on yhteisen näkemyksen ja ymmärryksen luominen toiminnan pohjaksi.

Asioidessaan asiakkaan kanssa ammattilaisen on kerrottava myös niistä seikoista, joita asiakas ei osaa itse kysyä. Ammattilaisen on kerrottava myös huonot uutiset.

Työn vaikutukset. Tietoteknisen työn tulokset saavat usein arvonsa vasta kun niitä hyödynnetään. Tietotekniikan ammattilaisen on pyrittävä ymmärtämään oman työnsä vaikutus usein pitkävaiheiselle ketjulle, jonka päässä on lopullinen hyödyntäjä. Ammattilaisen on myös otettava huomioon kuluttajan, laskun maksajan ja työnantajan vaatimukset.

Toimiessaan ammattilainen pyrkii katsomaan työnsä laajempaa merkitystä koko sille yhteisölle, jolle työ tehdään, eikä rajoitu vain hänen kanssaan asioivien edustajien näkemyksiin.

Muut ihmiset. Tietotekniikan ammattilainen kunnioittaa toisten työtä ja ottaa huomioon muiden ihmisten oikeuden luomaansa ja tekemäänsä.

Tietotekniikan ammattilaisen työ koskee sidosryhmien kautta yhteiskuntaa laajemmin. Ammattilaisen on käsitettävä työnsä seuraukset ja otettava huomioon esimerkiksi ihmisoikeudet, ympäristön suojelu, lainsäädäntö ja tekijänoikeudet.

Eettisyyden kasvu. Tietotekniikan ammattilaisen tulee edistää eettisesti kestävien toimintatapojen yleistymistä tietotekniikka-alalla.

Toimiminen eettisesti on valinta, jonka jokainen yksilö voi tehdä tai olla tekemättä. Eettisyys ei ole mustavalkoinen asia, vaan ihminen voi kehittyä koko ajan ottamalla ympäristöään enemmän huomioon. Nämä ohjeet pyrkivät esittämään tietotekniikan ammattilaiselle eettisen toimintamallin, joka tukee sekä hänen itsensä että ympäristönsä eettistä kasvua.

Taustamateriaalia

Gotterbarn, D., Miller, K. & Rogerson, S. (1999) Computer Society and ACM Approve Software Engineering Code of Ethics. IEEE Computer, 32, 10, lokakuu 1999, sivut 84–88.

Raatikainen, K. (1991) Tietojenkäsittelytiedettä etsimässä. Tietojenkäsittelytiede, marraskuu 1991.[3]

Raatikainen, K. (2002) Issues in Essence of Computer Science.[4]

Tietotekniikan liitto (2002) Etiikan ohjeet, versio 3.[5]

Lähteitä

Attewell, P. & Rule, P. (1984) Computing and Organizations: What We Know and What We Don't Know. Communications of the ACM 27, 12, joulukuu 1984, sivut 1184–1192.

Berkeley, E. C. (1962) The Computer Revolution. Doubleday, Garden City.

Dunlop, C. & Kling, R. (toim.) (1991) Computerization and Controversy: Value Conflicts and Social Choices. Academic Press, Boston.

Hertlein, G. C. (1990) Computers and the Quality of Life? ACM SIGCAS Computers & Society 20, 3, lokakuu 1990, sivut 60–66.

Kling, R. (1980). Social Analyses of Computing: Theoretical Perspectives in Recent Empirical Research. ACM Computing Surveys 12, 1, maaliskuu 1980, sivut 61–110.

Mowshowitz, A. (1981) On Approaches to the Study of Social Issues in Computing. Communications of the ACM 24, 3, maaliskuu 1981, sivut 146–155.

Mumford, L. (1971) The Pentagon of Power. Secker & Warburg, London.

Oppenheimer, J. R. (1955) The Open Mind. Simon and Schuster, New York.

[3] `http://www.cs.helsinki.fi/u/kraatika/Papers/tktiede.pdf`.

[4] `http://www.cs.helsinki.fi/u/kraatika/Papers/IssuesInEssenceOfComputerScience.pdf`.

[5] `http://www.tt-tori.fi/pls/ttl/docs/F148570701/Eettisetohjeet3.htm`.

Pylyshyn, Z. W. & Bannoin, L. J. (toim.) (1989) Perspectives on the Computer Revolution, 2nd Ed. Ablex Publishing Corp., Norwood.

LUKU 4

Silmäys tietojenkäsittelyn ydinteknologioihin

1950-luvulla tietojenkäsittelytieteen ydinteknologioihin laskettiin kuuluvaksi: algoritmit (*algorithms*), numeeriset menetelmät (*numerical methods*), laskennan mallit (*computation models*), kääntäjät (*compilers*), ohjelmointikielet (*languages*) ja logiikkapiirit (*logic circuits*).

1980-luvun loppuun mennessä joukkoon oli lisätty: käyttöjärjestelmät (*operating systems*), tiedonhaku (*information retrieval*), tietokannat (*databases*), tietoverkot (*networks*), tekoäly (*artificial intelligence*), ihmisen ja tietokoneen vuorovaikutus (*human-computer interaction, HCI*) ja ohjelmistotekniikka (*software engineering*).

Viimeisten 15 vuoden aikana eli ACM:n ja IEEE:n *Computing as a Discipline* -raportin [Denning, 1989] julkaisemisen jälkeen uusien ydinteknologioiden määrä on yli kolminkertaistunut. Denningin viimeisin luettelo [Denning, 2003] sisältää kolmekymmentä ydinteknologiaa. Denningin luettelossa esiintyvät laskennallinen tiede ja tieteellinen laskenta erillisinä ydinteknologioina. Itse pidän Denningin tieteellistä laskentaa osana laskennallista tiedettä. Siksi seuraavat lyhyet ydinteknologioiden luonnehdinnat, jotka perustuvat pääasiassa WikipediA:ssa [Wikipedia, 2005] oleviin artikkeleihin, käsittävät ”vain” 29 ydinteknologiaa.

algoritmit (*algorithms*): Termi algoritmi on johdettu 800-luvulla eläneen persialaisen matemaatikon al-Khrwarizmin nimestä. Sillä tarkoitetaan äärellistä joukkoa hyvin määriteltyjä ohjeita jonkin tehtävän suorittamiseksi. Algoritmitutkimuksessa kehitetään algoritmeja sekä analysoidaan niiden ominaisuuksia.

hajautettu tietojenkäsittely (*distributed computation*): Hajautetussa tietojenkäsittelyssä tutkitaan fyysisesti eri paikoissa olevien tietokoneiden yhteistoimintaa. Perimmäisenä tavoitteena on tarjota käyttäjille tietojenkäsittelyresursseja läpinäkyvästi, avoimesti ja laajentuvasti eli skaalautuvasti.[1] Hajautetulla tietojenkäsittelyllä tavoitellaan parempaa saatavuutta, vikasietoisuutta (*fault-tolerance*) ja suoritustehoa.

ihmisen ja tietokoneen vuorovaikutus (*human-computer interaction, HCI*): Ihmisen ja koneen välisessä vuorovaikutuksessa tutkitaan käyttäjien ja tietokoneiden välistä vuorovaikutusta. Se on monitieteinen tutkimusalue. Tietojenkäsittelytieteen osalta tutkimus keskittyy käyttöliittymään (*user interface, UI*), joka kattaa sekä laitteiston (tietokoneen oheislaitteet) että ohjelmiston. Teknologian perustavoitteena on tehdä tietokoneista ja tietojärjestelmistä käyttäjäystävällisiä (*user-friedly*) ja helppokäyttöisiä.

johdon tietojärjestelmät (*management information systems, MIS*): Johdon tietojärjestelmä on (yleensä tietokonepohjainen) järjestelmä, joka kerää, muokkaa ja tallettaa tietoa sekä tarjoaa sen organisaation hallinnon (*management*) käyttöön päätöksentekoa, suunnittelua, toteutusta ja seurantaa varten.

konenäkö (*vision*): Konenäön tavoitteena on saada tietokone “ymmärtämään” kuvien sisältöä. Tässä yhteydessä “ymmärtäminen” on hyvin rajallista. Konenäössä kuvien sisällöstä etsitään tarkkaan määriteltyä, tiettyä tarkoitusta palvelevaa informaatiota. Tämä informaatio välitetään joko ihmiselle (lääkäri saa röntgenkuvan, jossa epäilyttävät alueet on korostettu) tai jotakin prosessia ohjaavalle järjestelmälle (automaattisen varastotrukin liikkumisen ohjaus tai virheellisten tuotteiden poistaminen pakkauslinjalta). Jotkut pitävät konenäköä osana tekoälyä, jossa järjestelmän toiminnanohjaus saa syötteen kuva-aineistona ja oppii tunnistamaan haluttuja, yleensä poikkeavia tilanteita. Koska kameraa voidaan pitää valotunnistimena (*light sensor*), niin konenäön monet menetelmät perustuvat valon fysikaalisten ominaisuuksien ja kuvien välisiin vastaa-

[1] Hajautetun tietojenkäsittelyn alkuaikoina koiranleuat tosin määrittelivät hajautetun tietojenkäsittelyjärjestelmän järjestelmäksi, jossa käyttäjälle täysin tuntemattoman tietokoneen virheellinen toiminta estää käyttäjää käyttämästä omaa työasemaansa.

vuuksiin. Kolmas konenäköön keskeisesti vaikuttava tieteenala on biologia, erityisesti erilaisten näkemisjärjestelmien rakenne ja toiminta (silmän fysiologia). Myös signaalinkäsittely (*signal processing*) liittyy konenäköön. Useat yksiulotteisen signaalin käsittelyyn kehitetyt menetelmät voidaan melko suoraviivaisesti yleistää konenäössä tarvittavaan kaksi- tai moniulotteisen signaalin käsittelyyn. Toisaalta kuvainformaation erityisluonteen vuoksi konenäössä on kehitetty menetelmiä, joilla ei ole vastinetta yksiulotteisen signaalin käsittelyssä. Konenäön keskeisiä osa-alueita ovat mm. esineiden tunnistaminen (*object recognition*), kohteen seuraaminen (*tracking*), näkymän tulkitseminen (*scene interpretation*) ja itsepaikannus (*ego positioning*).

käyttöjärjestelmät (*operating systems*): Käyttöjärjestelmä on se osa tietokonejärjestelmän ohjelmistoa, joka huolehtii laitteiston ja järjestelmän perustoiminnallisuuden valvonnasta ja hallinnasta. Käyttöjärjestelmän tarjoamien järjestelmäkutsujen (*system call*) avulla ohjelmat pääsevät käyttämään oheislaitteita (*peripherials*), tiedostoja ja keskusmuistia. Käyttöjärjestelmä huolehtii myös keskeytyksistä (*interrupts*), ajastimista (*timers*), prosesseista (*processes*) ja säikeistä (*threads*) sekä niiden vuorottamisesta (*scheduling*). Käyttöjärjestelmä huolehtii myös samanaikaisuuden hallinnasta (*concurrency control*), samanaikaisesti suorituksessa olevien ohjelmien eristämisestä ja prosessien välisestä kommunikoinnista (*interprocess communication, ipc*). Toisinaan käyttöjärjestelmiin lasketaan kuuluvaksi myös kirjastot (*libraries*), jotka helpottavat järjestelmäkutsujen käyttöä eri ohjelmointikielissä. Käyttöjärjestelmätutkimuksen keskeisiä alueita ovat mm. muistinhallinta (*memory management*), tiedostojärjestelmät (*file systems*), samanaikaisuuden hallinta, vikasietoisuus (*fault-tolerance*) ja virrankulutuksen hallinta (*power management*).

kääntäjät (*compilers*): Kääntäjä on tietokoneohjelma, joka muuntaa lähdekieliset (*source code*) lauseet tuloskielisiksi (*object code*) lauseiksi. Tyypillisesti kääntäjän tuottama objektikoodi on konekieltä, johon on lisätty tietoa nimistä ja niiden sijainneista sekä ulkoisista funktioista. Suorituskelpoinen koodi (*executable*) saadaan linkittämällä yksi tai useampi objektikoodi sekä kirjastoja. Useimmat nykyiset kääntäjät on suunniteltu kaksivaiheisiksi (*two stage*). Ensimmäisessä vaiheessa käännetään lähdekoodi välimuotoon (*in-*

termediate representation). Tämä vaihe sisältää sanastollisen eli leksikaalisen (*lexical*), muotosääntöjen eli syntaktisen (*syntax*) ja merkitystä koskevan eli semanttisen (*semantic*) analysoinnin. Toisessa vaiheessa välimuotoinen koodi muunnetaan objektikoodiksi. Tämän vaiheen asioita ovat kääntäjäanalyysi (*compiler analysis*), optimointi (*optimization*) ja koodin generointi (*code generation*).

laskennallinen tiede (*computational science*): Peter Denningin ydinteknologioiden listassa laskennallinen tiede ja tieteellinen laskenta (*scientific computation*) on erillisinä ydinteknologioina. Näiden ero on hyvin epämääräinen, siksi pidän tieteellistä laskentaa osana laskennallista tiedettä. Laskennallinen tiede on muiden tieteenalojen tutkimusongelmia kuvaavien mallien ratkaisemista tietokoneen avulla. Tieteellisessä laskennassa tarkastellaan eri tieteenaloilla esiintyvien matemaattisten mallien numeerisia ratkaisumenetelmiä sekä niiden tietokonetoteutuksia. Historiallisesti tieteellinen laskenta jatkaa numeeristen menetelmien perinteitä tietojenkäsittelytieteessä. Laskennallisesta tieteestä on alettu puhua, kun tietokoneiden ja laskentamenetelmien käyttö on laajentunut eksakteista luonnontieteistä ja teknisistä tieteistä biotieteisiin ja lääketieteeseen.

luonnollisten kielten käsittely (*natural language processing, NLP*): Luonnollisten kielten käsittely on tekoälyn ja kielitieteen (*linguistics*) osa-alue. Tällä tutkimusalueella tarkastellaan luonnollisen kielen käsittelyyn ja ymmärtämiseen liittyviä ongelmia. Tavoitteena on saada tietokone “ymmärtämään” ihmisten käyttämiä kieliä.

ohjelmistotekniikka (*software engineering*): Ohjelmistotekniikka tarkastelee ohjelmistojen suunnitteluun, toteuttamiseen ja ylläpitoon liittyviä teknologioita ja käytäntöjä. Siinä yhdistyvät tietojenkäsittelytieteen lukuisat muut ydinteknologiat, projektinhallinta (*project management*), insinööritaito (*engineering*) ja kulloisenkin sovellusalueen tietämys. Ohjelmistotekniikassa kustannukset ja luotettavuus (*reliability*) ovat yhtä keskeisesti esillä kuin perinteisillä insinööritaidon alueilla. IEEE:n standardi 610.12 [IEEE, 1990] määrittelee ohjelmistotekniikan olevan systemaattisen, kurinalaisen, kvantifioitavissa olevan lähestymistavan käyttämistä ohjelmiston kehittämisessä, käytössä ja ylläpidossa sekä tällaisten lähestymistapojen tutkimista.

ohjelmointikielet (*programming languages*): Ohjelmointikieli on täs-

mällisesti määritelty tapa antaa tietokoneelle toimintaohjeet. Se koostuu syntaktisista ja semanttisista säännöistä. Ohjelmointikieli määrittelee ohjelmoijan käytettävissä olevat tietotyypit (*data types*), tietorakenteet (*data structures*) ja lauseet (*statements*). Ohjelmointikielten tutkimuksessa keskeisiä alueita ovat ohjelmointikielten ominaisuudet ja ohjelmointimallit.

päätöksenteontukijärjestelmät (*decision support systems, DSS*): Päätöksenteontukijärjestelmät ovat tietokoneistettuja informaatiojärjestelmiä, jotka tukevat organisaatioiden päätöksentekoa. Käsitteenä DSS on hyvin laaja. Eri kirjoittajat antavat sille hyvinkin erilaisia määritelmiä. Kirjon toista ääripäätä kuvaa hyvin Finlayn antama määritelmä: "Päätöksenteontukijärjestelmä on tietokonepohjainen järjestelmä, joka avustaa päätöksentekoprosessia" [Finlay, 1994]. Turbanin määritelmä edustaa sitä suppeampaa ääripäätä: "Päätöksenteontukijärjestelmä on vuorovaikutteinen, joustava ja mukautuva tietokonepohjainen informaatiojärjestelmä, joka on päätöksenteon tehostamiseksi kehitetty tukemaan strukturoimattoman hallinnointiongelman ratkaisua. Se hyödyntää dataa, tarjoaa helppokäyttöisen käyttöliittymän ja toimii päätöksentekijän oman näkemyksen mukaisesti" [Turban, 1995]. Powerin katsaus [Power, 2005] antaa hyvän kuvan päätöksenteontukijärjestelmien kehittymisestä.

realiaikajärjestelmät (*real-time systems*): Realiaikaisessa tietojenkäsittelyssä tarkastellaan järjestelmiä (sekä laitteistoja että ohjelmistoja), joiden on täytettävä aikavaatimuksia. Realiaikaisen järjestelmän ei välttämättä tarvitse olla nopea, mutta tulokset on saatava valmiiksi ennalta määrättyyn aikarajaan (*deadline*) mennessä. Lisäksi nämä aikarajat eivät riipu järjestelmän kuormituksesta. Realiaikajärjestelmät luokitellaan koviin (*hard*) ja pehmeisiin (*soft*) aikarajojen ehdottomuuden perusteella. Kovan realiaikajärjestelmän on aina ja kaikissa mahdollisissa tilanteissa saatava jokaikinen tehtävä suoritettua määräaikoihin mennessä. Tällaisia järjestelmiä ovat mm. ydinvoimaloiden ohjaus- ja hallintajärjestelmät, lennonjohtojärjestelmät sekä monet erilaisia laitteita ohjaavat järjestelmät.

rinnakkaislaskenta (*parallel computation*): Rinnakkaislaskennassa yksi tehtävä jaetaan osatehtäviin, joita suoritetaan samanaikaisesti usealla prosessorilla (vrt. hajautettu tietojenkäsittely). Rinnakkaislaskennan tavoitteena on nopeuttaa tehtävän suorittamista. Termillä

rinnakkaisprosessori (*parallel processor*) tarkoitetaan tietokonetta, jossa on useita suorittimia (*processor, central processor unit, CPU*) yhden käyttöjärjestelmän hallinnassa. Kun järjestelmässä on tuhansia prosessoreita, niin puhutaan massiivisesta rinnakkaisuudesta (*massively parallel*). Moniprosessorikoneessa (*multiptrocessor*) on tyypillisesti muutamia suorittimia. Rinnakkaislaskennan tutkimuskohteita ovat muun muassa laitteistoarkkitehtuurit—erityisesti prosessorien välinen ja prosessorien ja muistien välinen kytkentä (*interconnection*)—sekä rinnakkaislaskentaan soveltuvat algoritmit ja säikeiden välinen kommunikointi.

robotiikka (*robotics*): Robotiikka[2] tutkii robottien suunnitteluun, rakentamiseen ja käyttöön liittyviä kysymyksiä. Robotiikan kehittäminen edellyttää elektroniikan, mekaniikan ja ohjelmistotekniikan hallintaa. Tyypillisesti tiettyyn tehtävään soveltuvan robotin kehittämiseen tarvitaan sopivia havaintoja tekevät tunnistimet (*sensors*), ohjausalgoritmi(t) ja robotin fyysistä toimintaa ohjaavat säätimet (*actuators*).

supertietokoneet (*supercomputers*): Supertietokoneiksi kutsutaan tietokoneita, jotka aikoinaan olivat laskentateholtaan maailman parhaita. Supertietokoneet ovat perinteisesti saavuttaneet laskentatehon kulloisenkin huipun käyttämällä innovatiivisia ratkaisuja rinnakkaisuuden lisäämiseksi käskyjen käsittelyssä, muistin käytössä ja operaatioiden suorituksessa. Supertietokoneet on lähes poikkeuksetta suunniteltu tietyn tyyppiseen tietojenkäsittelyyn, useimmiten numeeriseen laskentaan eli numeronmurskaukseen. Supertietokoneiden muistihierarkia on suunniteltu huolellisesti, jotta suorittimet eivät joutuisi odottamaan käskyjen ja datan saantia muistista.

sähköinen kaupankäynti (*e-commerce*): Sähköinen kaupankäynti koostuu tuotteiden tai palveluiden jakelusta, ostamisesta, myynnistä, markkinoinnista ja tarjonnasta tietoverkkojen, ennenkaikkea Internetin, välityksellä. Sähköiseen kaupankäyntiin liittyviä osatoimintoja ovat mm. sähköinen varainsiirto, tuotantoketjun hallinta (*supply chain management*), sähköinen markkinointi (*e-marketing*), välitön tapahtumankäsittely (*online transaction processing*), sähköi-

[2]Termi *robotics* on peräisin tieteiskirjallisuudesta, Isaac Asimovin vuonna 1941 julkaistusta tarinasta *Liar!*.

nen tiedonvaihto (*electronic data interchange, EDI*), automatisoidut varastokirjanpitojärjestelmät ja automatisoidut tiedonkeruujärjestelmät. Sähköinen kaupankäyntijärjestelmä on monitieteellinen haaste. Teknologian toimivuus on perusedellytys, mutta sähköisen kaupankäynnin menestyminen edellyttää myös sopivia liiketoimintamalleja ja riittävää, tietoturvaan ja vahvaan tunnistamiseen pohjautuvaa, luottamusta. Viimekädessä osapuolten käyttökokemukset ratkaisevat sähköisten palvelujen henkiinjäämisen.

tekoäly (*artificial intelligence, AI*): Tekoäly määritellään keinotekoisen luomuksen, yleensä tietokoneohjelman, osoittamaksi älykkyydeksi. Tekoälytutkimuksessa tarkastellaan järjestelmiä, jotka automatisoivat älykästä käyttäytymistä edellyttäviä tehtäviä. Tällaisia tehtäviä ovat mm. ohjaus (*control*), suunnittelu ja ajoitus (*planning and scheduling*) ja puheentunnistus (*speech recognition*). Tekoälyyn perustuvat järjestelmät ovat nykyisin laajalti käytössä mm. taloustieteissä ja lääketieteessä sekä videopeleissä.

tiedonhaku (*information retrieval*): Tiedonhaussa keskitytään informaation etsimiseen (*searching*) dokumenteista, dokumenttien etsimiseen, dokumentteja kuvaavan metatiedon (*metadata*) etsimiseen sekä etsintään tietokannoista ja erilaisista tietoverkoista. Alunperin automatisoituja tiedonhakujärjestelmiä käytettiin hallitsemaan räjähdysmäisesti kasvavaa tieteellisten julkaisujen sisältämää informaatiota. Nykyisin painopiste on Webin hakukoneissa.

tiedon louhinta (*data mining*): Tiedon louhinnassa etsitään laajoista tietomassoista kaavaimia (*pattern*) käyttäen tilastotieteen ja hahmon tunnistuksen (*pattern recognition*) laskennallisia menetelmiä. Teknologiasta käytetään englanninkielessä myös termiä *knowledge-discovery in databases* (*KDD*). Tavoitteena on löytää tietomassasta (*data*) aiemmin tunnistamatonta ja mahdollisesti hyödyllistä informaatiota.

tietokannat (*databases*): Tietokanta on organisoitu kokoelma tietoa. Tietokannassa tieto on järjestetty tietueisiin (*record*), jotka koostuvat tietoalkioista (*data elements*). Tietokannan hallintajärjestelmäksi (*database management system, DBMS)* kutsutaan ohjelmaa, jonka avulla hallitaan tietokannassa olevaa tietoa ja voidaan kohdistaa kyselyjä (*query*) tietokannan tietosisältöön. Tietokannan

käyttämä tietomalli (*data model*) määrää kyselykielet (*query language*), joiden avulla tietokantaa käytetään. Tietokannan käsittelyn nopeuttamiseksi voidaan käyttää indeksointia (*indexing*). Transaktioiden[3] avulla hallitaan samanaikaisuutta (*concurrency*) eli tietokannan samanaikaista käyttöä. Transaktioilla voidaan taata nk. happo-ominaisuudet (*ACID*) eli atomisuus (*atomicity*), eheys (*consistency*), eristettävyys (*isolation*) ja pysyvyys (*durability*). Toisintamalla (*replication*) voidaan parantaa suorituskykyä (*performance*) ja saatavuutta (*availability*).

tietokonearkkitehtuuri (*computer architecture*): Tietokonearkkitehtuuri on tietokoneiden rakenteen suunnittelun taustalla oleva teoria. Tähän kuuluu laitteiston suunnittelu siten, että laitteisto toimii ohjelmoijien olettamalla tavalla, ja toteutusteknologioiden, kuten puolijohteiden, käyttäminen siten, että laitteisto on paras mahdollinen. Paras mahdollinen riippuu suunnittelun tavoitteista. Tavallisimmin paras on kustannusten ja nopeuden välinen kompromissi. Muita keskeisiä tavoitteita voivat olla laitteen koko ja paino sekä virrankulutus.

tietokonegrafiikka (*graphics*): Tietokonegrafiikka kattaa visuaalisen tietojenkäsittelyn. Siihen kuuluu sekä kuvamateriaalin synteettinen tuottaminen että todellisuudesta peräisin olevan visuaalisen informaation ja paikkatiedon (*spatial information*) muokkaaminen. Tietokonegrafiikassa on lukuisia osa-alueita, kuten tosiaikainen kolmiulotteinen kuvien esittäminen (*3-D rendering*), animointi, videosignaalin käsittely, visuaalisten tehosteiden luonti ja muokkaaminen, kuvan (*image*) muokkaaminen ja mallintaminen.

tietorakenteet (*data structures*): Tietorakenne on tiedon talletustapa tietokoneessa. Tietorakenteen valinta vaikuttaa olennaisesti tiedon käsittelyn tehokkuuteen. Hyvin suunniteltu tietorakenne mahdollistaa tärkeimpien toimenpiteiden suorituksen mahdollisimman vähäisin resurssein (suoritusaika ja muistintarve). Koska tietoraken-

[3]Käsitteestä *transaction processing* käytetään suomenkielisiä termejä transktioiden käsittely ja tapahtumien- tai tapahtumankäsittely. Jälkimmäisen ongelma on, että englanninkielen käsite *event handling* on suomeksi käännettynä myös tapahtumankäsittely. Nykyisin suomenkielistä termiä tapahtumankäsittely käytetään useimmiten vastaamaan englanninkielistä termiä *event handling.* Englanninkielen käsite *transaction processing* on nykyään suomeksi useimmiten transaktioiden käsittely.

teet ovat ohjelmissa erittäin keskeisiä, useat ohjelmointikielet ja -ympäristöt tarjoavat tietorakenteiden käsittelyyn optimoituja kirjastorutiineja.

tietoturva (*data security*): Tietoturva (englanniksi myös *information security*) käsittelee tietoon ja informaatioon liittyviä luottamuksen (*trust*) eri aspekteja. Tietoturvaan liittyy mm. pääsynvalvonta (*access control*), luottamuksellisuus (*confidentiality*), tiedon eheys (*integrity*), saatavuus (*availability*), vastuullisuus (*accountability*), kiistämättömyys (*non-repudiation*), varmennettavuus (*assurance*) ja tunnistus (*authentication*). Tietoturvan keskeisin periaate on yksinkertainen ilmaista: *"oikea informaatio oikeille ihmisille oikeaan aikaan"*.

tietoverkot (*computer networks*[4]): Tietoverkko on erilaisilla fyysisillä tietoliikenneyhteyksillä ja tietoliikenneprotokollilla yhteenkytkettyjen tietokoneiden muodostama järjestelmä. Viimeisen kymmenen vuoden aikana tietoverkot ovat nousseet lähes kaikkien tietojärjestelmien keskeiseksi osaksi. Tietoverkkojen tutkimusalueita ovat muun muassa tietoliikennelaitteet, tiedon esitysmuodot, tietoturva, tietoliikenneprotokollat, verkonhallinta (*network management*), langaton tiedonsiirto (*wireless communication*) ja liikkuva tietojenkäsittely (*mobile computing*).

työnkulku (*workflow*): Tietojenkäsittelyssä työnkulku liittyy organisaatioiden työtehtävien tekemisen järjestämiseen ja miten tietokonejärjestelmiä voidaan käyttää työn organisoimisen apuna. Keskeisiä kysymyksiä ovat: miten työtehtävät järjestetään, kuka suorittaa minkäkin tehtävän, missä järjestyksessä työtehtävät on suoritettava, mitkä ovat tehtävän aloittamisen edellytykset, miten informaatiovirrat (*information flows*) tukevat tehtäviä ja miten tehtävien etenemistä seurataan. Työnkulun tukijärjestelmissä (*workflow systems*) on yleensä kaksi melko erillistä osaa: työkulun mallintaminen (*workflow modeling component*) ja työnkulun seuranta (*workflow execution component, workflow run-time system*). Työnkulun mallintamisessa hallintohenkilöstö (*administration*) ja analyytikot

[4] WikipediA:n tietoverkkoja käsittelevä luku (`http://en.wikipedia.org/wiki/computer_network`) on ainoastaan linkkikokoelma tietoverkkojen osa-alueisiin.

voivat määritellä prosessit ja aktiviteetit, analysoida ja simuloida niitä sekä kohdennetaan ne eri henkilöille. Seurantajärjestelmän keskeisin osa on työnkulkukone (*workflow engine*), joka auttaa prosessien ja aktiviteettien koordinointia ja suorittamista.

virtuaalitodellisuus (*virtual reality, VR*): Virtuaalitodellisuus on tietokoneella toteutettu simuloitu ympäristö. Useimmat virtuaaliympäristöt ovat ensisijaisesti visuaalisia kokemuksia, jotka näytetään tietokoneen näytöllä tai erityisellä stereoskooppisella näytöllä. Yhä useammin ympäristöt sisältävät myös kuvan kanssa synkronoidun äänen. Simuloitu ympäristö voi olla todellisuuden kaltainen, kuten lentäjien koulutuksessa käytettävät opetussimulaattorit. Toisaalta useiden videopelien simuloidulla ympäristöllä ei ole mitään tekemistä oikean todellisuuden kanssa. Yksityiskohtaisesti tarkan (*high-fidelity*) virtuaalitodellisuuden toteuttaminen on edelleen hyvin haastavaa. Keskeiset ongelmat liittyvät tarvittavaan laskentatehoon ja näytön resoluutioon. Tästä ydinteknologiasta on käytetty myös englanninkielisiä termejä *artificial reality*, *augmented reality* ja *cyberspace*.

visualisointi (*visualization*): Visualisoinnin alueelle kuuluvat menetelmät, joilla luodaan kuvia, kaavioita tai animaatioita. Tavoitteena on parantaa tiedon välittymistä ja saada haluttu sanoma esitetyksi paremmin. Visualisointia käytetään yhä laajemmin tieteissä, tekniikassa, tuotekehityksessä ja tuotannossa, opetuksessa ja lääketieteessä. Tietokonegrafiikka on visualisoinnin tärkein apuväline. Visualisoinnin ja tietokonegrafiikan eroa voisi pelkistää: Visualisoinnissa keskitytään kysymykseen, *mitä halutaan näyttää*, tietokonegrafiikassa kysymykseen, *miten haluttu visuaalinen ilme saadaan aikaan.*

Taustamateriaalia

Denning, P. J. (2003) Great Principles of Computing. Communications of the ACM, 46, 11, marraskuu 2003, sivut 15–20.

Lähteitä

Denning, P. J. et al. (1989) Computing as a discipline. Communications of the ACM 32, 1, tammikuu[5] 1989, sivut 9–23.

Finlay, P. N. (1994) Introducing decision support systems. Oxford, UK Cambridge, Mass., NCC Blackwell; Blackwell Publishers.

IEEE (1990) IEEE standard glossary of software engineering terminology. IEEE Standard 610.12-1990

Power, D. J. (2005) A Brief History of Decision Support Systems, version 2.8. `http://dssresources.com/history/dsshistory.html`.

Turban, E. (1995) Decision support and expert systems: management support systems. Englewood Cliffs, N.J., Prentice Hall.

WikipediA (2005) `http://en.wikipedia.org/wiki/`: Algorithm, Artificial Intelligence, Compiler, Computer Architecture, Computer Graphics, Computer Vision, Database, Data Mining, Data Structure, Decision Support System, Distributed Computing, Electronic Commerce, Human-Computer Interaction, Information Retrieval, Information Security, Natural Language Processing, Operating System, Parallel Computing, Programming Language, Real-time Computing, Robot, Scientific Computing, Software Engineering, Supercomputer, Virtual Reality, Visualization, Workflow.

[5] ACM Digital Libraryssä on CACMin tämän numeron kuukaudeksi mainittu myös helmikuu.

LUKU 5

Tietojenkäsittelyn mekaniikat

Tietojenkäsittelyn ydinteknologioiden perusperiaatteiden analysoinnin perusteella Denning nimesi viisi perustoiminnallisuutta:

laskenta, jonka peruskysymykset liittyvät laskennan rajoihin: mitä voidaan laskea äärellisessä ajassa;

kommunikointi, jonka peruskysymykset liittyvät sanoman välittämiseen paikasta toiseen;

koordinointi, jonka peruskysymykset liittyvät yhteistyöhön: miten vähintään kaksi toimijaa työskentelee yhteisen päämäärän hyväksi;

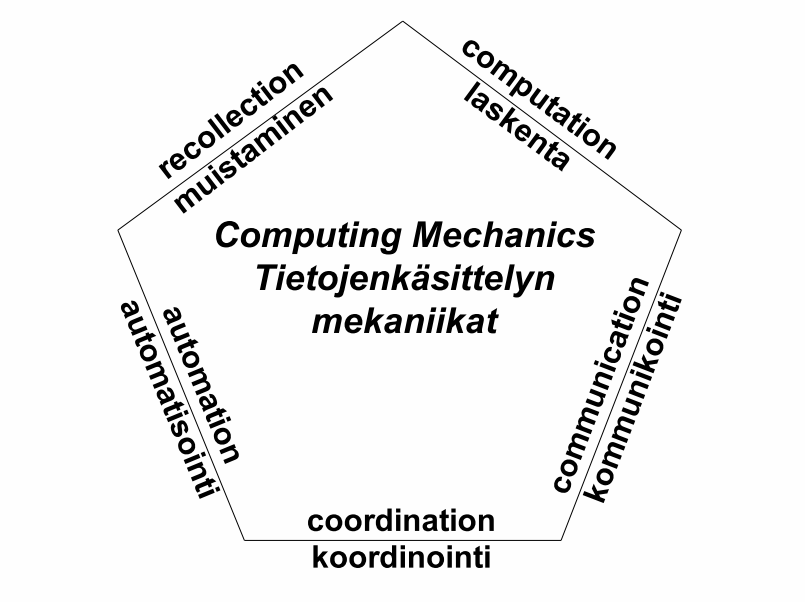
automatisointi, jonka peruskysymykset liittyvät tietokoneella suoritettaviin kognitiivisiin (ihmiselle tyypillisiin tietojenkäsittelyn) tehtäviin;

muistaminen, jonka peruskysymykset liittyvät informaation tallentamiseen ja hakemiseen.

Kullekin edellä kuvatuille toiminnallisuuksille laitoksemme opetusohjelmassa on useita kursseja. Lisäksi on huomattava, että perustoiminnallisuudet eivät ole toisiaan poissulkevia. Esimerkiksi Internetin protokollapino liittyy sekä kommunikaatioon että koordinointiin. Tästä syystä Denningin kutsuukin perustoiminnallisuuksia tietojenkäsittelyn mekaniikkojen näkymiksi (engl. *windows into computing mechanics*)–katso kuva 5.1

Tässä luvussa kerrotaan kustakin perustoiminnallisuudesta yksi tarina:

– **laskenta:** Turingin kone,
– **kommunikointi:** protokollapino,

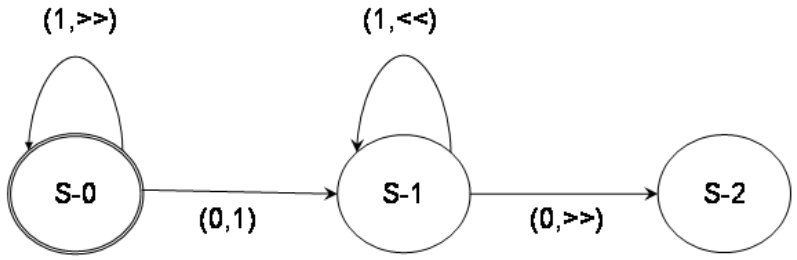


Kuva 5.1: Tietojenkäsittelyn mekaniikkojen viisi näkymää [Denning, 2003]

- **koordinointi:** synkronointi,
- **automatisointi:** Turingin testi,
- **muistaminen:** välimuistit.

5.1 Laskenta

Laskennan (engl. *computing*) peruskysymykset liittyvät laskennan rajoihin: *Mitä voidaan laskea äärellisessä ajassa?* Laskentaan liittyviä keskeisiä tarinoita ovat algoritmit (*algorithm*), ohjausrakenteet (*control structures*), tietorakenteet (*data structures*), automaatit (*automata*), kielet (*languages*), Turingin koneet (*Turing machines*), universaalit koneet (*universal computers*), Turingin kompleksisuus (*Turing complexity*), Kolmogorov-kompleksisuus (*Kolmogorov-Chaitin complexity*), predikaattilogiikka (*predicate logic*), likimääräismenetelmät (*approximations*), heuristiikat (*heuristics*) ja satunnaisalgoritmit (*probabilistic algorithms*), laskennallinen ratkeamattomuus (*non-computability*), muunnokset (*translations*).



Kuva 5.2: Esimerkki tilakoneesta: Turingin kone ”Lisää yksi”

Turingin kone

Turingin koneet ovat yksinkertaisia abstrakteja laskentalaitteita. Ne ovat peräisin ajalta ennen varsinaisia tietokoneita, englantilaisen matemaatikon Alan Turingin julkaisuista vuosilta 1936 ja 1937. Turing kehitteli ”koneitaan” laskettavuuden rajojen tutkimuksessa: *Mitä voidaan algoritmisesti ratkaista?* Samalla Turing määritteli matemaattisen tarkasti algoritmin.

Turingin koneen ideana oli antaa ihmiselle riittävän tarkat ohjeet laskentatehtävän suorittamiseksi—*siis riittävän tarkat ohjeet ihmiselle, ei tietokoneelle, jota ei silloin vielä ollut.*

Turingin kone on tilakone, joka voidaan esittää kaaviopiirroksena kuten kuvassa 5.2. Jokaisella hetkellä se on jossakin tietyssä tilassa eli koneen tilamuuttujalla on jokin äärellinen arvo. Turingin koneella on ääretön yksiulotteinen peräkkäismuisti. Jokaiseen muistipaikkaan voidaan tallettaa yksi arvo. Muistiin liittyy *luku- ja kirjoituspää*, jota voidaan siirtää edelliseen tai seuraavaan muistipaikkaan ja jonka avulla voidaan lukea ja kirjoittaa nykyisen muistipaikan arvo. Ohjelman suorituksen aikana tilasiirtymät määräytyvät muistipaikoista luettujen arvojen perusteella.

Turingin koneen toiminnan täydelliseen määrittämiseen tarvitaan koneen nykyinen tila, nykyisen muistipaikan arvo, ja tilasiirtymät eli koneen ”ohjelma”. Jokainen tilassssiirtymä on nelikko:

$$< nykytila, arvo, uusitila, toimenpide >$$

jonka tulkinta on: Jos koneen tila on $nykytila$ ja nykyisen muistipaikan arvo on $arvo$, niin muuta koneen tilaksi $uusitila$ ja suorita $toimenpide$. Toimenpide on joko symbolin tallentaminen nykyiseen muistipaikkaan

(merkintänä muistipaikan uusi arvo), tai luku- ja kirjoituspään siirtäminen joko eteenpäin ($\gg$) tai taaksepäin ($\ll$).

Jos kone päätyy tilanteeseen, johon ei ole määritelty siirtymäsääntöä tai johon on määritelty useampi kuin yksi sääntö, niin kone pysähtyy.

Turingin koneet näyttävät hyvin yksinkertaisilta. Kuitenkin niiden analysointi on osoittanut, että ne ovat laskennallisesti yhtä voimakkaita kuin nykyiset tietokoneet. Itse asiassa Churchin-Turningin teesiä, että ei ole olemassa ongelmaa, joka voitaisiin ratkaista nykyisillä tietokoneilla mutta ei Turingin koneella, voidaan pitää matemaattisena tosiasiana. Tässä yhteydessä on syytä korostaa, että *laskennallinen voimakkuus* tai laskentavoima tarkoittaa kykyä ratkaista jokin tehtävä, eikä se liity mitenkään tehtävän ratkaisemiseen kuluvaan aikaan eli laskentatehoon.

Turingin koneen yksinkertaisuus on tehnyt siitä tehokkaan työvälineen laskettavuuden teoriassa. Koneellaan Turing pystyi osoittamaan, että tietyt ongelmat ovat laskennallisesti ratkeamattomia. Merkittävin aikaansaannos lienee nk. *pysähtymisongelma (engl. halting problem)*, jonka Turing osoitti olevan laskennallisesti ratkeamaton. Ei ole olemassa ohjelmaa, joka pystyisi päättelemään päättyykö vai ei mikä tahansa toisen ohjelman suoritus millä tahansa syötteellä.

Turingin todistus perustui vastaesimerkkiin. Olettamalla, että pysähtymisongelman ratkaiseva ohjelma olisi olemassa, päädytään loogisesti mahdottomaan tilanteeseen.

Esimerkki

Alla on kenties yleisimmin käytetty Turingin koneen esimerkki (sama kuin kuvassa 5.2):

$$< s_0, 1, s_0, \gg >$$
$$< s_0, 0, s_1, 1 >$$
$$< s_1, 1, s_1, \ll >$$
$$< s_1, 0, s_2, \gg >$$

Jotta kyseisen Turingin koneen toimintaa voidaan selittää, on muistipaikkojen symboleille ’0’ ja ’1’ annettava tulkinta. Jos tulkitaan, että ’0’ on erotin ja n peräkkäistä ’1’-symbolia esittää positiivista kokonaislukua $n-1$, niin yllä oleva Turingin kone kasvattaa syötteenä annettua (luku- ja kirjoispään osoittaman muistialueen esittämää) lukua yhdellä.

Kaksi lukua yhteenlaskeva Turingin kone on:

$$
\begin{aligned}
&< s_0, 1, s_0, \gg > \\
&< s_0, 0, s_1, 1 > \\
&< s_1, 1, s_1, \ll > \\
&< s_1, 0, s_2, \gg > \\
&< s_2, 1, s_2, 0 > \\
&< s_2, 0, s_3, \gg > \\
&< s_3, 1, s_3, 0 > \\
&< s_3, 0, s_4, \gg >
\end{aligned}
$$

Harjoitus

Käy läpi Turingin kone, joka laskee $2 + 1$, toiminta. Koneen alkutila on s_0 ja muistin sisältö '$\ldots 0\mathbf{1}110110\ldots$'. (Luku- ja kirjoituspään sijainti vahvennettu.) Lopetustilanteen pitäisi olla '$\ldots 000\mathbf{1}1110\ldots$'.

Universaalit Turingin koneet

Yksi Turingin kone laskee vain yhden laskettavissa olevan funktion. Tässä mielessä se on kuin tietokone, johon on asennuttu yksi kiinteä ja muuttumaton ohjelma. Toisin sanoen jokainen ongelma tarvitsee oman Turingin koneensa. Aikoinaan Alan Turing osoitti, että on olemassa *universaali Turing kone*, joka pystyy jäljittelemään (simuloimaan) minkä tahansa Turingin koneen toimintaa.

Tähän asti olemme puhuneet vain Turingin koneista, jotka tunnistavat symbolit '0' ja '1' eli binäärisen aakkoston. Laskennan kannalta aakkoston kokoa, eli symbolien lukumäärää, voidaan kasvattaa. Laajentamalla aakkostoa Turingin koneen tarvitsema tilojen lukumäärä yleensä pienenee. Itseasiassa universaali Turingin kone on melko yksinkertainen. Pienimmät tunnetut koneet ovat 2×18 (kaksi tilaa ja 18 symbolia), 3×10, 4×6, 5×5, 7×4, 10×3, 22×2.

Tietojenkäsittelyn alkuaikoina tietokoneet rakennettiin yhtä tehtävää varten. Toisen maailmansodan aikana englantilaisten rakentama COLOSSUS mursi saksalaisten salakirjoitusta. Amerikkalaisten vuonna 1946 rakentama ENIAC laski ballistisia taulukoita. Universaalin Turingin koneen olemassaolo osoitti, että yleiskäyttöinen ohjelmoitava tietokone on

mahdollinen. Siitä, vaikuttiko tämä tulos mitenkään tietokoneiden kehitykseen, voidaan olla montaa eri mieltä.

Laskennallisen vaativuuden merkityksestä

Laskennan teorissa (engl. *theory of computation*) käytetään Turingin koneen lisäksi muitakin laskennan malleja, kuten esimerkiksi rekursiivisia funktioita ja λ-kalkyyliä. Laskentamallien avulla tutkitaan laskettavuuden rajoja:

- Mitkä ongelmat ovat todistettavasti algoritmisesti ratkeamattomia.

  Tällaisia ongelmia on löytynyt muodollisen päättelyn alueelta, mikä on vaikuttanut mm. tekoälyn kehitykseen.

- Mitkä ongelmat voidaan kyllä ratkaista tietokoneella, mutta ratkaisun tuottaminen kestää niin kauan, että ratkaisu on valmistumishetkellään käytännössä käyttökelvoton.

  Tällaisia hankalia eli NP-kovia (engl. *NP-hard, intractable*) on runsaasti, erityisesti tilanteissa, kun halutaan löytää paras mahdollinen vaihtoehto. Tyypillisiä esiintymisalueita ovat mm. skedulointi, pakkausongelmat, pelit, verkkoteoria.

Lisäksi laskennan vaativuudessa analysoidaan eri algoritmien suorittamiseen tarvittavien operaatioiden lukumäärää ja tilan tarvetta.

Algoritmin aikavaativuus eli operaatioiden määrä voi olla esimerkiksi $O(n \log n)$. Tulos luetaan muodossa: Jos algoritmille annetaan syötteenä n alkiota, niin ratkaisuun tarvittavien operaatioiden määrä on kertaluokkaa $n \log n$. Jos algoritmin vaativuus on $O(\exp(n))$, niin kyseisen algoritmin sanotaan olevan skaalautumaton. Sen vaativuus kasvaa eksponentiaalisesti. Suomeksi sanottuna kysyinen algoritmi on käyttökelpoinen vain hyvin pienillä syötteillä.

Vaikka vain aniharvat tietojenkäsittelijät analysoivat algoritmeja päätyökseen, niin kaikkien tietojenkäsittelijöiden tulisi tuntea laskennan teorian perusteet ja perustulokset. Jos tulee luvanneeksi kirjoittaa pysähtymisongelman ratkaisevan ohjelman, niin jossakin vaiheessa joutuu tunnustamaan, ettei osaa ratkaista annettua tehtävää. Alan ammattilainen olisi heti osannut kertoa, että kyseinen tehtävä on todistetusti ratkeamaton.

5.2 Kommunikointi

Kommunikaation (engl. *communication*) peruskysymykset liittyvät sanoman välittämiseen paikasta toiseen. Kommunikaation keskeisiä tarinoita ovat tiedonsiirto (*data transmission*), tiedon fyysinen esittäminen (*encoding to medium*), kanavan kapasiteetti (*channel capacity*), kohinanpoisto (*noise suppression*), tiedon tiivistäminen (*file compression*), salakirjoitus (*cryptography*), pakettiverkko (*reconfigurable packet networks*), virheiden havaitseminen ja korjaaminen (*error detection and correction*).

Protokollapino

Protokollapino[1] (engl. *protocol stack*) on tietoliikenneohjelmisto, jolla toteutetaan tiedonsiirtoa ja siihen liittyviä toimintoja. Siinä ilmentyy yksi tietojenkäsittelyn keskeisimmistä lähestymistavoista: haasteiden eriyttäminen (engl. *separation of concerns*).

Protokollapino koostuu kerroksista. Kukin kerros toteuttaa sille määrätyt asiat ja tarjoaa ne palveluina seuraavaksi ylemmälle kerrokselle. Samalla kerros ”piilottaa” itseään alemman kerroksen toiminnan ylemmältä kerrokselta.

Nykyisin käytetään lähes yksinomaan IETF:n eli Internet Engineering Task Forcen määrittelemää TCP/IP-pinoa (engl. *TCP/IP Protocol Suite*). Käsitteellisenä kehikkona toimii useimmiten ISO:n (International Standardization Organization) OSI (Open System Interconnect) malli. OSI-mallissa (kuva 5.3) on seitsemän kerrosta.

Sovelluskerroksella keskitytään kunkin sovelluksen vuoropuheluun eli dialogin määrittelyyn; määritellään viestit; niiden rakenne ja merkitys. Yleisimpiä sovellustason protokollia ovat sähköposti, uutisryhmät, Web.

Esitystapakerroksella keskitytään sanoman sisällön esitystapaan. Perinteisessä Internet-maailmassa tämä kerros on ollut lähes olematon. Se on jätetty sovelluksen sisäiseksi asiaksi tai sitten on käytetty yksinkertaista ”TLV” eli tyyppi–pituus–arvo (engl. *type–length–value*) esitysmuotoa.

[1] Suomenkielisessä kirjallisuudessa käytetään protokollasta myös – nykyään enää harvemmin – termiä yhteyskäytäntö.

Kuva 5.3: ISO:n OSI-malli

Aivan viime vuosina Web konsortion eli W3C:n (World Wide Web Consortium) määrittelemä XML eli eXtensible Markup Language on yleistynyt sanoman sisällön esitystapana.

Istuntokerros mahdollistaa istunnon (engl. *session*) muodostamisen. Tätä kerrosta ei Internet-maailmassa ole juuri tarvittu, sillä sovellukset ovat useimmiten perustuneet tilattomaan pyyntö/vastaus vuoropuheluun, jossa jokainen pyyntö on itsenäinen, aiemmista pyynnöistä riippumaton.

Tilattoman vuoropuhelun etuna on, että keskustelijoiden ei tarvitse muistaa mitään menneestä. Tämä on suunnaton etu palvelimelle, joka keskustelee samanaikaisesti satojen tai tuhansien asiakkaiden kanssa. Haittana on, että jokaisessa pyynnössä ja vastauksessa on kerrottava kaikki tarvittava.

Session Initiation Protocol (SIP) on melko uusi tulokas Internet-maailmassa. Sen avulla muodostetaan kahden tai useamman osapuolen välinen keskustelu, muokataan keskusteluun liittyviä hallintatietoja ja aikanaan lopetetaan keskustelu. SIP vastaa telemaailman puhelunohjausta (engl. *call control*).

Istuntoon voidaan liittää tilatietoa, jota ei tarvitse lähettää jokaisessa sanomassa uudelleen. Jos istuntokerrosta ei ole käytössä, kuten nykyisessä Webissä, niin istunto joudutaan toteuttamaan sovellustasolla. Webissä tämä näkyy 'piparien' (*cookie*) käyttönä.

Kuljetuskerros keskittyy sanoman siirtämiseen päätepisteiden välillä. Internet-maailman keskeiset kuljetusprotokollat ovat TCP ja UDP. TCP on luotettavuuden takaava tietovuo (engl. *data stream*). UDP on epäluotettava tietosähke (engl. *datagram*).

Verkkokerroksella keskitytään pääasiassa sanomien reititykseen – mitä reittiä pitkin sanoma kulkee lähettäjältä vastaanottajalle. Verkkokerroksella huolehditaan myös ruuhkanhallinnasta. Internet-maailmassa verkkokerroksen keskeisin protokolla on IP eli Internet Protocol. IP:n ruuhkanhallinta on suoraviivaista: Jos sanomia on liikaa, jotkut sanomat yksinkertaisesti tuhotaan.

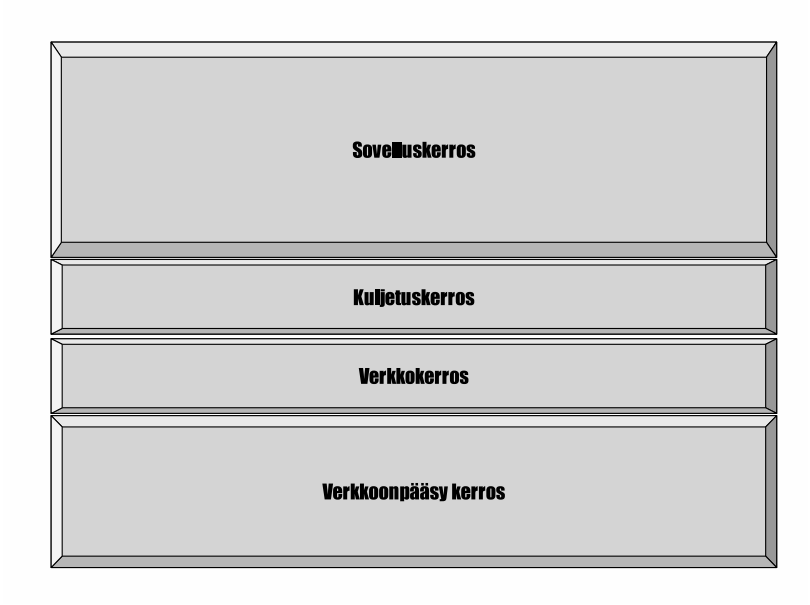
Linkkikerroksen tärkein tehtävä on muuntaa fyysinen tiedonsiirtokanava tiedonsiirtolinjaksi siten, että fyysisessä tiedon siirrossa tapahtuvat virheet eivät näy verkkokerrokselle. Tavanomaisin tapa on ryhmitellä bitit muutaman sadan tai tuhannen tavun kehyksiksi. Kehykseen liitetään joitakin ylimääräisiä bittejä, joiden avulla vastaanottaja pystyy päättelemään onko kehyksen sisältö muuttunut matkan varrella. Muuttumattomat kehykset kuitataan vastaanotetuiksi, rikkoutuneet pyydetään lähettämään uudelleen.

Fyysisellä kerroksella keskitytään bittien lähettämiseen tiedonsiirtokanavaa pitkin. Tyypillisesti määritellään kuinka monta volttia käytetään esittämään arvoa 1 ja kuinka monta arvoa 0. Lisäksi määritellään bitin kesto. Muita määriteltäviä asioita ovat mm. miten fyysinen tiedonsiirtoyhteys muodostetaan ja miten se lopetetaan.

OSI-mallin ja TCP/IP-pinon, joka perustuu USA:n puolustusministeriön viitekehykseen eli nk. DoD[2]-malliin (kuva 5.4), keskeisimmät erot ovat:

1. DoD-mallista puuttuvat esitystapa- ja istuntokerrokset. Näiden kerrosten toiminnallisuus on toteutettava jokaisessa sovellustason protokollassa erikseen.

[2]Department of Defense



Kuva 5.4: USA:n puolustusministeriön DoD-malli, johon TCP/IP-pino perustuu

2. DoD-mallissa fyysinen kerros ja linkkikerros on yhdistetty verkkoonpääsykerrokseksi (*network access layer*). Tällä erolla ei ole suurta merkitystä, sillä käytännössä tarvittava toiminnallisuus on useimmiten toteutettu verkkokortilla.

Protokollapino lisää järjestelmien joustavuutta ja monipuolisuutta. IP-kerroksen päällä voidaan samanaikaisesti käyttää useita eri kuljetuskerroksia. Useat sovellukset voivat toimia samanaikaisesti, kukin käyttäen omaa sovellustason protokollaansa. Toisaalta uusi (fyysinen) tiedonsiirtokanava saadaan käyttöön lisäämällä linkkikerrokselle kyseisen kanavan tuki.

Protokollapinosta on myös kustannuksensa. Jokainen protokollakerros yleensä tarvitsee oman otsakkeen (engl. *header*), joka ohjaa kyseisen protokollakerroksen toimintaa. Esimerkiksi verkkokerroksen otsakkeessa yleensä kerrotaan mitä verkkoprotokollaa käytetään, mistä sanoma on lähtöisin, ja minne sanoma pitäisi toimittaa. Jos sovellus haluaa kertoa kaverilleen ’42’, niin fyysisellä tiedonsiirtokanavalla kulkee useimmiten muutamia satoja tai joitakin tuhansia tavuja dataa.

Protokollapino on yksinkertaistanut sovellusten välistä tiedonsiirtoa.

Pistoke-rajapinnan (engl. *socket*) avulla sovellusohjelmoijan on helppo kirjoittaa verkkosovelluksia.

Esimerkki:

```
// tarvittavat Javan määrittelyt
try {
  theSocket = new Socket(hostname, 13);
  theTimeStream = new
           DataInputStream(theSocket.getInputStream());
  String theTime = theTimeStream.readLine();
  ...
} // end try
// tarvittavat poikkeusten käsittelyt
```

5.3 Koordinointi

Koordinoinnin (engl. *coordination*) peruskysymykset liittyvät yhteistyöhön. *Miten vähintään kaksi toimijaa työskentelee yhteisen päämäärän hyväksi?* Denning jakaa nämä kysymykset kolmeen ryhmään: ihmisten välisen, ihmisen ja tietokoneen välisen, sekä tietokoneiden välisen yhteistyön kysymyksiin. Ihmisten välisen yhteistyön keskeisiä tarinoita ovat toimintaketjut (*action loops*) ja tietokoneiden tukema työnkulku (*workflow as supported by communicating computers*). Ihmisen ja tietokoneen välisen yhteistyön keskeisiä tarinoita ovat rajapinta (*interface*), syöte (*input*), tuloste (*output*), vasteaika (*response time*). Tietokoneiden välisen yhteistyön keskeisiä tarinoita ovat synkronointi (*synchronization*), kilpatilanteet (*race conditions*), lukkiutuminen (*deadlock*), sarjallistuvuus (*serializability*), atomiset toimenpiteet (*atomic actions*).

Synkronointi

Synkronointi eli tahdistaminen on aikaan liittyvää koordinointia. Esimerkiksi monikanavaviestinnässä eri kanavien tietovirtojen tulee edetä samaa tahtia. Toisissa tilanteissa synkronointilla varmistetaan, että itsenäisten toimijoiden aikaansaannokset valmistuvat ”oikeassa” järjestyksessä.

Hajautetuissa järjestelmissä sanomiin liitetään usein lähettäjän aikaleima. Useissa tilanteissa vastaanottaja saa jonkinasteisen ”hikan”, jos sen mielestä sanomia alkaa saapua tulevaisuudesta. Sekä GPS (Global Positioning System) että NTP (Network Time Protocol) tarjoavat melko

hyvän globaalin ajan (UTC – Coordinated Universal Time) likiarvon. GPS perustuu satelliitteihin ja signaalien siirtoaikojen kolmiolaskentaan. NTP on Internetissä käytössä oleva protokolla, jonka avulla eri koneiden kellot saadaan synkronoitua siirtoviiveistä riippuen muutaman kymmenen millisekunnin (avoin Internet) tai muutaman sadan mikrosekunnin (nopea lähiverkko) tarkkuudella.

Erilaisten synkronointiongelmien ratkaisut kuuluvat tietojenkäsittelytieteen ydinainekseen. Ongelmat liittyvät usein yhteiskäyttöisten resurssien hallintaan. Ratkaisujen on estettävä sekä nälkiintyminen että lukkiutuminen.

Nälkiintyminen (*starvation*): Joku joutuu odottamaan vuoroaan ikuisesti.

Lukkiutuminen (*deadlock*): Kaikki odottavat, että joku toinen tekisi jotain.

Synkronointiongelmat esitetään pelkistettyinä tyyppiongelmina. Näistä tunnetuimpia ovat:

- **Tuottaja-kuluttaja -ongelma.** Kuluttaja ei voi kuluttaa ennenkuin tuottaja on tuottanut. Toisaalta välivaraston täyttymisen jälkeen tuottajan on odotettava, että kuluttaja ehtii kuluttamaan. Kun tuottajia ja kuluttajia on useita, niin on myös huolehdittava poissulkemisesta. Tuottajien tuotokset on saatava välivarastossa eri paikkohin. Kaksi kuluttajaa ei saa saada samaa tuotosta. Lisäksi samanaikaiset lisäykset ja poistot eivät saa sotkea varastokirjanpitoa.
- **Lukija-kirjoittaja -ongelma.** Kaikki voivat lukea samanaikaisesti, mutta vain yksi kerrallaan voi kirjoittaa. Kirjoittajan nälkiintymisen estäminen on erityisen tärkeää. Jos kirjoittaja ei saa kirjoitusvuoroa, niin kaikki lukijat lukevat vanhentunutta tietoa.
- **Aterioivien filosofien ongelma.**

Kilpatilanne

Käsite kilpatilanne (engl. *race hazard (or race condition)*) tarkoittaa järjestelmässä olevaa suunnitteluvirhettä, jonka seurauksena järjestelmän toiminta riippuu ennalta-arvaamattomalla tavalla tapahtumien käsittelyjärjestyksestä. Käsite juontaa juurensa tilanteesta, missä kaksi signaalia kilpailee siitä, kumpi pääsee ensin vaikuttamaan tulokseen.

Tietojenkäsittelyssä kilpatilanne syntyy, jos kaksi tai useampi ohjelma pääsee kontrolloimattomasti käsiksi yhteiskäyttöiseen resurssiin samanaikaisesti. Kuvittele, miltä pienryhmäsi oppimispäiväkirjan luku näyttäisi, jos Moodle ei pitäisi huolta, että jäsenten mahdollisesti samanaikaiset editoinnit tulevat hallitusti yhteiseen tuotokseen.

Oheinen WikipediAsta lainattu esimerkki yrittää havainnollistaa kilpatilannetta:

```
global integer A = 0;

task Received()
{    A = A + 1; }

task Timeout() // Print only the even numbers
{    if (A is divisible by 2)
    {        print A;    }
}
```

`Received` aktivoituu aina, kun sarjaportti aiheuttaa keskeytyksen eli vastaanottaa dataa. `Timeout` aktivoituu kerran sekunnissa (kellolaitekeskeytys).

Ennemmin tai myöhemmin seuraava tapahtumaketju pääsee tapahtumaan:

1. `Timeout` aktivoituu.
2. `Timeout` toteaa, että `A` on parillinen.
3. Sarjaportti vastaanottaa dataa, joten `Received` aktivoituu.
4. `Received` suoritetaan loppuun eli `A` kasvaa parittomaksi.
5. `Timeout` jatkaa siitä, mihin se jäi.
6. `Timeout` tulostaa `A`:n arvon, joka nyt on pariton.

Valitettavasti kaikki kilpatilanteet eivät ole näin yksinkertaisia ja helposti havaittavia. Sädehoitolaitetta ohjannut Therac-25 (`http://en.wikipedia.org/wiki/Therac-25`) on traaginen esimerkki ohjelmistosta, jossa synkronoinnilla ei oltu esitetty kilpatilannetta. Vuosilta 1985–1987 on kuusi todistettua tapausta potilaan massiivisesta ylisäteilytyksestä. Pahimmillaan yliannostus oli yli $10,000$-kertainen. Ainakin viiden potilaan kuolema pystyttiin osoittamaan kyseisen sädehoitolaitteen ohjelmistovirheestä johtuvaksi.

Aterioivien filosofien ongelma

Edsger Dijkstra käytti ryhmää aterioivia filosofeja esimerkkinä synkronointiongelman ratkaisemisessa.

> Viisi filosofia istuu pyöreän pöydän ympärillä. Jokaisen filosofin edessä on spagettilautanen. Jokaisen lautasen välissä on yksi haarukka. Siis viisi filosofia, viisi lautasta ja viisi haarukkaa.

Filosofien elämässä on kaksi vuorottelevaa tilaa: syö, ajattele. Jokainen filosofi tarvitsee kaksi haarukkaa syödäkseen, mutta haarukat otetaan yksitellen. Saatuaan kaksi haarukkaa filosofi syö jonkin aikaa, minkä jälkeen hän vapauttaa molemmat haarukat ja jatkaa ajatteluaan.

Jos jokainen filosofi saisi samanaikaisesti päähänsä tarttua oikeanpuoleiseen haarukkaan, niin syntyisi lukkiutuminen. Kullakin filosofilla olisi yksi haarukka ja jokainen odottaisi ikuisesti toisen haarukan vapautumista.

Vaihtoehtoisesti filosofit voisivat olla kohteliaita: Jos toinen haarukka ei ole vapaa, niin filosofi vapauttaa varaamansa haarukan, odottaa hetken ja yrittää varata haarukoita uudelleen. Nyt järjestelmä ei lukkiudu, mutta eivät ne filosofit pääse silti syömään ellei odotusaika ole satunnainen. Vaikka odotusaika olisikin satunnainen, niin ei ole mitään takeita, että jokainen filosofi pääsee joskus syömään.

Aterioivien filosofien ongelman ratkaisuun tarvitaan algoritmi, joka estää sekä edellä kuvatun lukkiutumisen että nälkiintymisen. Tällaisia ratkaisuja on useita.

5.4 Automatisointi

Automatisoinnin (engl. *automation*) peruskysymykset liittyvät tietokoneella suoritettaviin kognitiivisiin tehtäviin. Automatisoinnin keskeisiä tarinoita ovat kognitiivisten tehtävien simulointi (*simulation of cognitive tasks*), automatisoinnin filosofia (*philosophical distinctions about automation*), asiantuntemus ja asiantuntijajärjestelmät (*expertise and expert systems*), älykkyyden lisääminen (*enhancement of intelligence*), Turingin testit (*Turing tests*), koneoppiminen ja tunnistaminen (*machine learning and recognition*).

Turingin testi

Turingin testi on peräisin Alan Turingilta. Testin taustalla on englantilainen seurapiirileikki *Imitation Game*. Turing tarjosi testiään älykkyyden määritelmäksi. Vuonna 1950, jolloin Turing esitteli testinsä, keskustelu tietokoneen kyvyistä oli sangen värikästä. Tuohon aikaan tietokoneista, joita maailmassa oli muutama, puhuttiin lehdistössä myös ’ajattelevina koneina’ (engl. *thinking machine*). Suomessa puhuttiin vielä vuosia myöhemmin sähköaivoista.

> *Turingin testissä ihminen kirjoittaa kahdelle testattavalle kysymyksiä. Saamiensa kirjallisten vastausten perusteella hän yrittää erottaa, kumpi testattavista on tietokone.*

Tekoälyn maailmassa Turing testi edustaa tekoälyn ’toimii kuin ihminen’ määritelmää. Muita tekoälyn määritelmiä ovat ’ajattelee kuin ihminen’, ’ajattelee rationaalisesti’, ja ’toimii rationaalisesti’.

Turingin testin mielekkyydestä on vuosien saatossa käyty kiivastakin sananvaihtoa. Oltiin Turingin testin mielekkyydestä mitä mieltä tahansa, niin Loebnerin palkinnon saa vuosittain parhaiten Turingin testissä menetyvä järjestelmä. Lisäksi tarjolla on toistaiseksi (vuonna 2005) jakamaton palkinto ensimmäiselle Turingin testin läpäisevälle järjestelmälle.

Tutkimusyhteisössä Turingin testin läpäiseminen on jäänyt taka-alalle. Perusteluna on käytetty muun muassa lentämistä: Lentokoneen ilmassapysyminen perustuu aerodynamiikkaan, ei riittävän hyvään lintumaiseen (muut linnut hämääväänn) käyttäytymiseen.

Turingin testin kritiikkiä

Turingin testin käyttökelpoisuutta koneälyn määritelmänä on arvosteltu voimakkaasti. Kone, joka läpäisisi Turingin testin, kyllä jäljittelisi ihmisen käyttäytymistä keskustelussa, mutta tällainen jäljittely on vielä kaukana todellisesta älykkyydestä. Kone vain noudattaisi jotain nokkelasti laadittua säännöstöä. Toisaalta voidaan myös kysyä, mistä tiedämme, etteivät ihmisetkin vain noudata joitain nokkelasti laadittua säännöstöä. Kaksi tällaiseen argumentointiin perustuvaa tunnettua vastaesimerkkiä ovat John Searlen *kiinalainen huone* ja Ned Blockin *Blockhead*.

Lisäksi on huomattava, että Turingin testi ei edellytä tietoisuutta (engl. *consciousness*) tai tavoitteellisuutta (engl. *intentionality*). Esimerkiksi tieteen popularisoija Larry Gonick ei pidä Turingin testiä älykkyyden

määritelmänä, koska hänestä jäljittelyllä ei ole mitään tekemistä todellisen ajattelun kanssa.

Turingin testin opetuksia

Turingin testin kritiikin kärki on kohdistunut testin tavoiteasetteluun: *Järjestelmän tulisi käyttäytyä kuin ihminen.* Useimpiin tietojenkäsittelyn sovelluksiin tavoite on virheellinen. Tunnetusti ihminen tekee virheitä, joita muut ihmiset eivät välttämättä huomaa. Siten Turingin testin läpäisevä palkanlaskujärjestelmä voisi tehdä virheitä.

Täten Turingin testin läpäisevä järjestelmä olisi ”oikea vastaus väärään kysymykseen”. Useasti tällainen vastus on hyödytön. Toisaalta Turingin testin läpäisevän järjestelmän olisi ratkaistava ainakin neljä tietojenkäsittelyn keskeistä ongelmaa:

luonnollisen kielen käsittely: Järjestelmän on ymmärrettävä sille esitetyt kysymykset ja muotoiltava vastaukset. Luonnollisen kielen käsittelyssä erityisenä ongelmana on, että usealla sanalla on useita asiayhteydestä riippuvia merkityksiä.

tietämyksen esittämisen: Järjestelmän on talletettava tietämänsä ja kuulemansa. Tietämyksen laajentuessa talletuksen ja etsinnän tehokkuus nousevat keskeiseen asemaan.

automaattinen päättely: Järjestelmän on pystyttävä talletetun informaation perusteella tekemään päätelmiä, joiden perusteella vastaus voidaan muotoilla. Koska loogisen päättelyn jotkut ongelmat ovat todistetusti ratkeamattomia, niin järjestelmän on myös ”tiedettävä” mitä se ei pysty päättelemään.

koneoppiminen: Järjestelmän on sopeuduttava uusin tilanteisiin. Koneoppimisessa järjestelmän maailmankuva (eli malli järjestelmää kiinnostavista asioista) muuttuu järjestelmän saaman datan perusteella. Siten koneoppiminen on osittain päällekkäistä tilastotieteen kanssa, mutta koneoppimisessa oppimisen laskennallinen vaativuus on keskeisesti esillä.

Siten tässä tapauksessa ”oikea vastaus väärään kysymykseen” olisi erittäin hyödyllinen. Tämä hyödyllisyys johtuu ratkaisun välttämättömien ominaisuuksien samankaltaisuudesta muiden mielekkäiden ongelmien ratkaisujen ominaisuuksiin:

– Järjestelmän käyttäminen olisi ihmiselle helpompaa, jos vuorovaikutus perustuisi luonnolliseen kieleen ja järjestelmä todella ymmärtäisi luonnollista kieltä.
– Järjestelmä käyttäytyisi käyttäjän kannalta juohevasti, jos se pystyisi päättelemään mitä käyttäjä milloinkin haluaa.

Itseasiassa nykyisen käyttäjäkeskeisen suunnittelun tavoite *"Järjestelmän tulee käyttäytyä siten kuin ihminen haluaa"* on hyvin lähellä Turing testin tavoiteasettelua.

5.5 Muistaminen

Muistamisen (engl. *recollection*) peruskysymykset liittyvät informaation tallentamiseen ja hakemiseen. Muistamisen keskeisiä tarinoita ovat tallennusvälineiden hierarkiat (*hierarchies of storage*), viittausten paikallisuus (*locality of reference*), välimuistit (*caching*), osoiteavaruudet ja niiden väliset kuvaukset (*address space and mapping*), nimentä (*naming*), yhteiskäyttö (*sharing*), etsintä (*searching*), haku nimen perusteella (*retrieval by name*), haku sisällön perusteella (*retrieval by content*).

Välimuisti

Nykyisten tietojenkäsittelylaitteiden yksi suurimmista suorituskyvyn haasteista on saada käsiteltävä data riittävän nopeasti käsiteltäväksi. Prosessorit ovat huomattavasti nopeampia kuin keskusmuistit, keskusmuistit huomattavasti nopeampia kuin levymuistit. Web-sivujen nouto kaukaiselta palvelimelta vie aikaa. Erilaisia välimuisteja (engl. *cache*) käytetään nopeuttamaan tulosten valmistumista. Tieto talletetaan tilapäisesti välimuistiin lähemmäksi käsittelypaikkaa.

Välimuistit ovat osoittautuneet erittäin tehokkaiksi useilla tietojenkäsittelyn alueilla, koska datan käyttö on tyypillisesti paikallista. Paikallisuus (*locality*) esiintyy tietojenkäsittelyssä useissa eri muodoissa. Tavanomaisimmin käsite tarkoittaa, että samoja tietoalkioita käsitellään useita kertoja ajallisesti lähekkäin tai että lähellä toisiaan sijaitsevia tietoalkioita käsitellään ajallisesti lähekkäin.

Välimuisti on kokoelma (tietoalkio,tunniste) -pareja. Tietoalkio on kopio varsinaisessa muistissa olevasta tiedosta. Tunniste (*tag*) kertoo tietoalkion identiteetin eli mitä varsinaisessa muistissa olevaa tietoa välimuistissa oleva kopio vastaa.

Kun välimuistin käyttäjä (prosessori, Web-selain, käyttöjärjestelmä) haluaa käyttää varsinaisessa muistissa olevaa tietoa, niin se ensimmäiseksi tarkistaa, löytyykö kyseinen tieto välimuistista. Jos tarvittavan tiedon kopio on välimuistissa, niin käytetään kopiota. Jos välimuistissa ei ole kopiota, niin tietoalkio noudetaan varsinaisesta muistista ja useimmiten talletetaan myös välimuistiin.

Välimuisti on useimmiten huomattavasti pienempi kuin varsinainen muisti. Tällöin välimuistista joudutaan poistamaan alkioita uusien tietoalkioiden tieltä. Heuristiikkaa, jolla poistettava alkio valitaan, kutsutaan poistopolitiikaksi (*replacement policy*). Poistopolitiikka perustuu aikaisemmin mainittuun paikallisuuteen. Ohjelmallisesti toteutetuissa välimuisteissa yleisin poistopolitiikka on LRU (*least recently used*) eli poistetaan alkio, joka on ollut pisimpään käyttämättä.

Kun välimuistissa olevaa tietoa muutetaan, niin jossakin vaiheessa muuttunut tieto on kirjoitettava myös varsinaiseen muistiin. Tämän kirjoituksen ajoitusta ohjaa kirjoituspolitiikka (*write policy*). Läpikirjoittavassa (*write-through*) välimuistissa jokainen välimuistiin kirjoitus aiheuttaa välittömän kirjoituksen varsinaiseen muistiin. Vaihtoehtoisesti kirjoitusta voidaan viivästää (*write-back cache*). Tällöin välimuisti pitää kirjaa niistä tietoalkioista, joita on muutettu. Nämä kirjoitetaan varsinaiseen muistiin, kun alkio poistetaan välimuistista. Läpikirjoittaminen on tavallista välimuistin laitteistototeutuksessa (prosessorien välimuistit), viivästetty kirjoittaminen ohjelmallisesti toteutetuissa välimuisteissa (tiedostovälimuistit). Läpikirjoittavaa välimuistia käytetään myös tilanteissa, joissa välimuistin ja varsinaisen muistin välinen tietoliikenneyhteys on epäluotettava.

Välimuistissa oleva tietoalkio voi vanhentua (*stale*) myös siten, että varsinaisen muistin sisältö muuttuu. Tällöin välimuistin toteutuksessa tarvitaan eheysprotokolla (*coherency protocol*), jonka avulla välimuistien sisällöt pidetään yhteinäisinä. Välimuistin ajantasaisuus on Web-välimuistien ja hajautettujen tiedostovälimuistien erityinen haaste.

Prosessorin välimuisti

1980-luvun alkupuolelta lähtien prosessoreilla on ollut pieni välimuisti, joka on varsinaista keskusmuistia huomattavasti nopeampi. Nykyisissä tietokoneissa on useita, jopa puolisen tusinaa, erikoistuneita välimuisteja, joilla tehostetaan ohjelman suoritusta. Näitä välimuisteja prosessori käyttää pienentämään muistinsaantiviivettä (*access latency*).

Tiedon saantiviive (*read latency*) on yksi keskeisimmistä nykyisten prosessorien suorituskyvyn pullonkauloista (*bottleneck*). Hyvin usein prosessori ajautuu tilanteeseen, jossa se ei voi tehdä muuta kuin odottaa, että käsitellyt tietoalkiot saadaan talletettua keskusmuistiin tai että käsiteltävät tietoalkiot saadaan keskusmuistista prosessorin käsiteltäväksi. Vuosien saatossa tämä ongelma on tullut yhä polttavammaksi, koska prosessorit ovat nopeutuneet huomattavasti enemmän kuin muistipiirit ja väylät.

1970-luvulla keskusmuistin hitaus oli supertietokoneiden suunnittelijoiden ongelma. 1980-luvulla se tuli graafisten työasemien suunnittelijoiden ongelmaksi. Viimeistään 1990-luvun puolivälissä keskusmuistin hitaus alkoi olla jo tavallisten henkilökohtaisten tietokoneiden suunnittelijoiden ongelma. Prosessorien välimuisteilla on yritetty tasoittaa prosessorin ja keskusmuistin nopeuseroa. Nykyiset (2000-luvun alkupuolen) prosessorit suorittavat satoja konekäskyjä siinä ajassa, kun keskusmuistista saadaan noudettua yksi tietoalkio.

Prosessorien välimuistien toteutuksessa assosiatiivisuus (*associativity*) on keskeinen käsite. Se kuvaa, mihin paikkaan välimuistia keskusmuistin tietoalkiot sijoitetaan. Täysin assosiatiivisessa (*fully associative*) välimuistissa tietoalkiot voidaan sijoittaa minne tahansa välimuistissa. Toinen äärimmäisyys on suorasijoitus (*direct mapped*), jolloin sijainti lasketaan tietoalkion osoitteen perusteella. Useat nykyiset prosessorien välimuistit käyttävät näiden kahden ääripään välimuotoa, joka kulkee nimellä ryhmäassosiatiivisuus (*set associative*). Useimmin käytetyt ryhmäassosiatiivisuuden asteet ovat 2 tai 4 (*2-way, 4-way*). Ryhmäassosiatiivisuuden aste n tarkoittaa, että välimuistissa on n mahdollista paikkaa, johon tietoalkio voidaan kopioida.

Assosiatiivisuus on aina kompromissi. Mitä useampaan paikkaa tietoalkio voidaan sijoittaa, sitä useammasta paikasta sitä on myös etsittävä. Koska nyt puhumme prosessorien välimuisteista, etsintä on toteutetteva laitteistoratkaisuna, jotta välimuistioperaatiot pysyvät prosessorin vauhdissa. Mitä useampia paikkoja on etsittävänä, sitä enemmän tarvitaan virtaa, tilaa piireille ja aikaa. Toisaalta assosiatiivisuuden asteen nostaminen parantaa osumistodennäköisyyttä (*hit rate*), koska sijoituskonfliktit vähenevät. Käytännössä on havaittu, että assosiatiivisuuden asteen nostaminen korkeammaksi kuin 4 tai 8 ei olennaisesti paranna osumistodennäköisyyttä.

Useimmissa nykyaikaisissa tietokonejärjestelmissä on toteutettu virtuaalimuisti (*virtual memory*). Virtuaalimuisti luo jokaiselle suoritettavalle ohjelmalle illuusion, että se on yksin tietokoneessa ja voi käyttää omaa

osoiteavaruutta (*address space*). Järjestelmä sijoittelee eri ohjelmien muistiavaruudet eri paikkoihin fyysistä muistia. Siten prosessorin on muutettava nämä virtuaaliosoitteet todellisiksi muistiosoitteiksi. Koska lähes kaikki prosessorin käskyt viittaavat tavalla tai toisella muistiin, osoitteen muunnoksen on oltava tehokasta. Siksi lähes kaikissa prosessoreissa on erityinen osoitteenmuunnoksista huolehtiva laitteisto, jota kutsutaan muistinhallintayksiköksi (*memory management unit, MMU*). Muistinhallintayksiköllä on yleensä käytössä oma välimuistinsa, jota kutsutaan nimellä osoitteenmuunnospuskuri (*Translation Lookaside Buffer, TLB*).

Prosessorien välimuisteissa on päädytty useaan erikoistuneeseen välimuistiin, koska nykyisten liukuhihnoitettujen (*pipeline*) prosessorien on päästävä käsiksi muistiin käskynsuorituksen eri vaiheissa. Liukuhihnoituksella on nopeutettu prosessorin toimintaa lisäämällä prosessorin sisäistä rinnakkaisuutta. Prosessori käsittelee samanaikaisesti useaa eri käskyä, joiden suoritukset ovat käskynkäsittelyn eri vaiheissa.

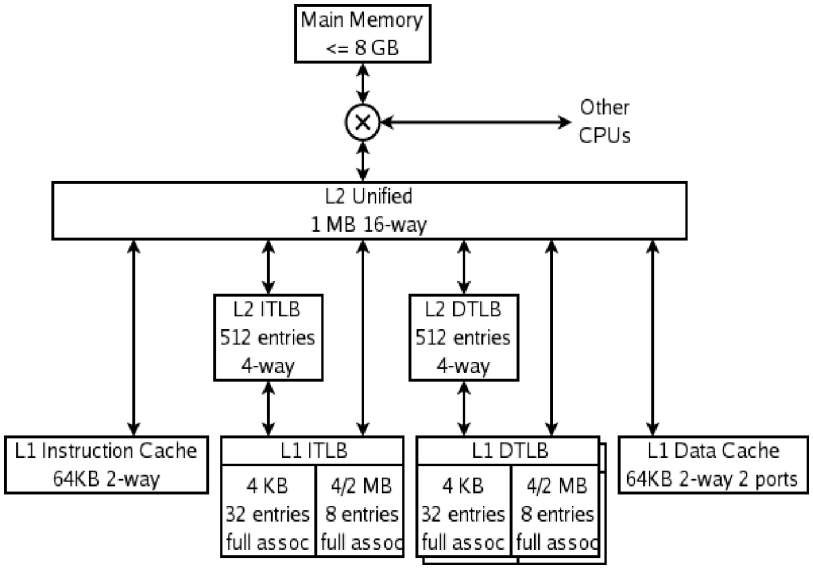
Yksinkertaisimmillaan liukuhihnoitettu prosessori jakaa käskynsuorituksen kolmeen vaiheeseen: käskyn nouto (*instruction fetch*), osoitteenmuunnos (*address translation*), tietoalkion nouto (*data fetch*). Luonnollinen ja helpoin tapa suunnitella välimuistin fyysinen toteutus on tehdä kullekin vaiheelle oma fyysinen välimuisti. Tällöin laitteistotasolla ei tarvitse suunnitella kilpatilanteen hallintaa. Tämän vuoksi lähes kaikissa nykyisissä prosessoreissa on ainakin kolme erikoistunutta välimuistia.

Useimmissa nykyaikaisissa prosessoreissa on vielä lisäksi hierarkinen välimuistien järjestelmä. Tämäkin on suunnittelun kompromissiratkaisu (*tradeoff*). Mitä laajempi välimuisti on, sitä hitaampi se on, mutta sitä paremman osumistodennäköisyyden se myös antaa. Useimmissa nykyaikaisissa prosessoreissa on ainakin kaksi välimuistien tasoa. Kuvassa 5.5 on kuvattu AMD Athlon 64 prosessorin välimuistien hierarkia. Tällä kurssilla ei tarvitse välittää kuvan yksityiskohdista, mutta lisätietoa löytyy URLista `http://www.sandpile.org/impl/k8.htm`.

Taustamateriaalia

Stanford Encyclopedia of Philosophy (2005) The Turing Test. `http://plato.stanford.edu/entries/turing_test`.

Stanford Encyclopedia of Philosophy (2005) Turing Machines. `http://plato.stanford.edu/entries/turing_machine`.



Kuva 5.5: AMD Athlon 64 prosessorin välimuistien hierarkia [WikipediA, 2005, CPU cache]

WikipediA (2005) Cache. `http://en.wikipedia.org/wiki/Caching`.

WikipediA (2005) CPU cache. `http://en.wikipedia.org/wiki/CPU_cache`.

WikipediA (2005) Dining philosophers problem. `http://en.wikipedia.org/wiki/Dining_philosophers_problem`.

WikipediA (2005) DoD model. `http://en.wikipedia.org/wiki/DoD_model`.

WikipediA (2005) Internet Protocol Suite. `http://en.wikipedia.org/wiki/Internet_protocol_suite`.

WikipediA (2005) Open Systems Interconnection. `http://en.wikipedia.org/wiki/Open_Systems_Interconnect`.

WikipediA (2005) OSI model. `http://en.wikipedia.org/wiki/OSI_reference_model`.

WikipediA (2005) Race hazard. `http://en.wikipedia.org/wiki/Race_condition`.

WikipediA (2005) Turing machine. `http://en.wikipedia.org/wiki/Turing_machine`.

WikipediA (2005) Turing test. http://en.wikipedia.org/wiki/Turing_test.

Lähteitä

WikipeadiA (2005) Blockhead. http://en.wikipedia.org/wiki/Blockhead.

WikipeadiA (2005) Chinese room. http://en.wikipedia.org/wiki/Chinese_room.

WikipediA (2005) Coordinated Universal Time. http://en.wikipedia.org/wiki/UTC.

WikipediA (2005) Global Positioning System. http://en.wikipedia.org/wiki/GPS.

WikipediA (2005) Network Time Protocol. http://en.wikipedia.org/wiki/Network_Time_Protocol.

WikipediA (2005) Therac-25. http://en.wikipedia.org/wiki/Therac-25.

LUKU 6

Suunnittelu

Tietojenkäsittelyn mekaniikka ei kuvaa täysin tietojenkäsittelyn kaikkia periaatteita. Tietojenkäsittelyn ammattilaiset tukeutuvat suunnittelussa periaatteisiin, jotka soveltavat tietojenkäsittelyn mekaniikkaa.

Vuosien saatossa hyväksi havaittuja suunnittelun käytäntöjä ovat mm.:

- abstrahointi (engl. *abstraction*) – epäolennaisten yksityiskohtien häivyttäminen,
- informaation piilottaminen (engl. *information hiding*) – ohjelmiston osan (modulin) sisäisten tietojen ja tietorakenteiden piilottaminen muilta osilta (moduleilta),
- modulit (engl. *modules*) – ohjelmiston jakaminen osiin siten, että osien väliset vuorovaikutukset ja rajapinnat ovat hyvin määriteltyjä,
- erikseen kääntäminen (engl. *separate compilation*) – ohjelman osien kääntäminen erikseen ja linkittäminen myöhemmin suorituskelpoiseksi kokonaisuudeksi,
- pakkaukset (engl. *packages*) – jakelu- ja asennusyksikkö, johon on koottu ohjelmisto(je)n osat ja dokumentaatio(t),
- versionhallinta (engl. *version control*) – menetelmät, joilla hallitaan ohjelmiston kehitystyötä ja sen aikana syntyviä ohjelmaversioita,
- hajoita ja hallitse (engl. *divide-and-conquer*) – periaate, jonka mukaisesti suuret kokonaisuudet jaetaan paremmin ja helpommin hallittaviksi osakokonaisuuksiksi,

- toimintatasot (engl. *functional levels*) – toiminnallisuuden ryhmitteleminen eri tasoisiksi toimenpiteiksi,
- kerrosajattelu (engl. *layering*) – toimintakokonaisuuden jakaminen kerroksiin siten, että kukin kerros tarjoaa joitakin (muutamia) palveluja ylemmille kerroksille käyttäen hyväksi alempien kerrosten tarjoamia palveluja,
- hierarkiat (engl. *hierarchy*) – suunnittelussa hierarkiat liittyvät yleensä olio-ohjelmoinnin luokkarakenteeseen; kukin luokka on jonkun luokan aliluokka perien sen ominaisuudet ja joidenkin luokkien yläluokka, jonka ominaisuudet periytyvät,

  (Tietojenkäsittelyssä esiintyy myös käsite muistihierarkia; tällöin on kyse eri muistivälineiden tarjoamasta talletuskapasiteetista ja saantinopeudesta: suuri talletuskapasiteetti implikoi pitkän saantiajan.)
- ongelmien eriyttäminen (engl. *separation of concerns*) – periaate, jonka mukaan ratkaisussa keskitytään yhteen, mahdollisimman täsmällisesti määriteltyyn tehtävään,
- uudelleenkäyttö (engl. *reuse*) – olemassa olevien modulien, määritysten ja suunnitelmien käyttäminen sen sijaan, että tehtäisiin uusia,
- kapselointi (engl. *encapsulation*) – ohjelmaelementtien sisällyttäminen laajempaan ja abstraktimpaan kokonaisuuteen; käsite on hyvin läheistä sukua informaation piilottamiselle ja ongelmien eriyttämiselle; kapseloinnilla yleensä piilotetaan toteutuksen sisäiset yksityiskohdat ja tarjotaan rajapinta kapselin sisältämien tietojen manipulointiin,
- rajapinta (engl. *interface*) – näiden välityksellä ohjelmisto- tai laitteistokomponentti tarjoaa toiminnallisuuksiaan muiden käyttöön,
- virtuaalikone (engl. *virtual machine*) – ohjelmisto, joka toteuttaa määritellyn abstraktin koneen toiminnallisuuden.

Näillä suunnittelussa vakiintuneilla toimintatavoilla pyritään saavuttamaan:

yksinkertaisuus (*simplicity*): erilaiset abstraktiot ja rakenteet, joilla pyritään hallitsemaan sovellusten luontaista monimutkaisuutta,

suorituskyky (*performance*): suoritustehon (engl. *throughput*) ja vasteajan (engl. *response time*) ennustaminen, pullonkaulojen (engl. *bottlenecks*) paikallistaminen, kapasiteetin suunnittelu (engl. *capacity planning*),

luotettavuus (*reliability*): toisteisuus (engl. *redundancy*), toipuminen (engl. *recovery*), varmistaminen (engl. *checkpointing*), eheys (engl. *integrity*), luottamus (engl. *trust*),

kehitettävyys (*evolvability*): varautuminen toiminnallisuuden ja käytönlaajuuden muutoksiin, ja

tietoturva (*security*): pääsynvalvonta (engl. *access control*), salassapito (engl. *secerecy*), yksityisyys (engl. *privacy*), tunnistus (engl. *authentication*), eheys (engl. *integrity*), turvallisuus (engl. *safety*).

Edellä lueteltujen tavoitteiden lisäksi suunnitteluun vaikuttavat useat rajoitteet, kuten kustannukset, aikataulut, yhteensopivuus (engl. *compatibility*) ja käytettävyys (engl. *usability*).

6.1 Yksinkertaisuus

Yksinkertaisuus on ollut kauttaaikojen keskeinen tavoite tietojenkäsittelyjärjestelmien suunnittelussa. Seaymor Crayn, joka oli 1970- ja 1980-luvuilla etevin supertietokoneiden suunnittelija, toinen laitteistosuunnittelun periaate oli, että monimutkainen on hidasta. (Se toinen oli, että kauas on pitkä matka, eli kaikki komponentit on saatava mahdollisimman lähelle toisiaan.) IBM:n ohjelmistosuunnittelijoiden huoneentauluna kerrotaan olleen *"KISS!"*, tarkoittaen joko

Keep It Simple and Small!

tai

Keep It Simple and Stupid!

Kumpikin heijastaa sitä tosiasiaa, että ohjelmien tai järjestelmien monimutkaistuminen tuo mukanaan ongelmia. Samaa asennetta heijastaa myös Edsger W. Dijkstran vuodelta 1972 oleva tietojenkäsittelytieteen määritelmä [Dijkstra, 1972]: *Tietojenkäsittelytiede on monimutkaisuuden hallitsemisen tutkimista.*

Aikoinaan tätä Dijkstran tietojenkäsittelytieteen määritelmää vastaan hyökättiin voimakkaasti väittämällä, että useat muutkin tieteenalat yrittävät hallita monimutkaisuutta. Tänään, yli kolmekymmentä vuotta myöhemmin, useimmat tietojenkäsittelytieteen ammattilaiset näkevät Dijkstran naulankannan. Tietojenkäsittelyjärjestelmät alkavat olla monimutkaisimpia ihmisten tekemiä konstruktioita, joiden toimintaa olisi hyvä ymmärtää, hallita ja säädellä.

Yksinkertaisuuden tarinan pohjaksi olen valinnut David Lorge Parnasin helmikuussa 1996 IEEE Computer -lehdessä julkaistun artikkelin *"Why Software Jewels Are Rare"* [Parnas, 1996]. Sen kovin *'kilpakumppani'* oli Pascal-ohjelmointikielen kehittäjän Niklaus Wirthin artikkeli *"A Plea for Lean Software"* [Wirth, 1995].

Molemmat artikkelit ovat jokaiselle tietojenkäsittelytieteen opiskelijalle lukemisen arvoisia. Meidän tietojenkäsittelytieteilijöiden pitää suhteuttaa lyhyt historiamme. Tietojenkäsittelytieteen da Vincit, Rembrandtit ja Rubensit ovat 50-, 60-, ja 70-luvuilta; van Goghit ja Manetit 80-luvulta, Picassot ja Munchit 90-luvulta, toisen maailmansodanjälkeiset taiteilijat tältä vuosikymmeneltä.

Parnas aloittaa artikkelinsa toteamalla olevansa ohjelmistomatkaaja, joka innokkaasti kurkkii toisten ohjelmoijien aikaansaamaan koodiin. Toisinaan hän väittää törmänneensä helmiin eli rakenteeltaan erinomaisiin ohjelmiin, joissa ohjelmointityyli on johdonmukainen, joista ohjelmointitrikit puuttuvat, joissa jokainen komponentti on yksinkertainen ja selkeärakenteinen. Lisäksi tällaiset ohjelmat tai ohjelmoinnin helmet on suunniteltu siten, että myöhemmin tehtävät muutokset ovat helppoja eivätkä riko ohjelman (ohjelmiston) selkeää rakennetta. Artikkelissaan Parnas ihmettelee, miksi tällaiset helmet ovat harvinaisuuksia.

Vuosien varrella ohjelmoinnista ja ohjelmistojen rakentamisesta on kirjoitettu runsaasti. Saatavilla on sekä hyviä oppikirjoja sekä alan taiturien, kuten Niklaus Wirth [Wirth, 1995] ja Edsger Dijkstra [Dijkstra, 1968], kertomuksia siitä, miten he tuottivat ohjelmistohelmensä. Tästä huolimatta suurimman osan näkemistään ohjelmista Parnas katsoo kuuluvan luokkaan *"karsea–epäluotettava–vaikeasti muutettava"*. Aiheellisesti Parnas kysyykin, miksi ohjelmistohelmiä on niin vähän, jos keskeiset kirjoitukset tarjoavat käyttökelpoisia ratkaisuja ohjelmoinnin ja ohjelmistojen tekemisen ongelmiin. Päällimmäiseksi selitykseksi Parnas tarjoaa sitä tosiasiaa, että helmet on tehty olosuhteissa, jotka harvoin toteutuvat ohjelmistoteollisuudessa. Useimpien helmien tekijöiden ei ole tarvinnut välittää tuotoksensa kaupallisesta menestyksestä.

Usein ohjelmiston laajentumisen yhteydessä sen rakenne on hämärtynyt. Ulkoiset paineet ovat vaatineet, että ohjelmistoon lisätään uusia piirteitä. Lisäksi vaatimukset, että ohjelmiston on oltava yhteensopiva joidenkien vanhojen ohjelmistojen kanssa, ovat lisänneet ohjelmiston monimutkaisuutta. Jatkuva uusien ominaisuuksien lisääminen tuhoaa alkuperäisen rakenteen, ellei ohjelmistoa ole alunperin suunniteltu laajennettavaksi. Tähän teemaan palaamme luvussa 6.4 *Kehitettävyys*.

Wirthin lääke ohjelmien ”ylilihomista” (*fat software*) vastaan on keskittyminen olennaiseen ja ’kilkuttimien’ unohtaminen. Tämä on kaunis periaate, mutta ongelmat alkavat, kun yritämme päättää, mikä on olennaista ja mikä on turhaa ylellisyyttä. Jos asiakkaalle tarjotaan vaihtoehdoiksi olennaiseen keskittyvä helmi ja hieman käyttökelpoisempi työkalu, niin Parnaksen mukaan useimmat valitsevat työkalun. Jotta ohjelmatuote pärjäisi markkinoilla, sen on tarjottava yleisesti hyödyllisiksi katsotut ominaisuudet. Karrikoidusti Parnas toteaa, että joidenkin ominaisuuksien poisjättäminen ohjelmien solakoimiseksi olisi sama kuin jalan amputoiminen ihmisen liikapainon hoitokeinona.

Ohjelmat paisuvat useista eri syistä. Wirth pitää tärkeimpänä syynä huonoa suunnittelua. Parnas puhuu muistakin syistä. Huomattavan usein paisuminen johtuu piirteistä ja rajapinnoista, jotka ovat välttämättömiä yhteensopivuuden aikaansaamiseksi vanhojen ohjelmien kanssa. Kaupallisilla markkinoilla suunnittelijat joutuvat lisäämään järjestelmiinsä toiminnallisuutta, jota he eivät sisällyttäisi järjestelmiinsä, jos yhteensopivuusvaatimusta ei olisi. Järjestelmät, jotka tarjoavat yhteensopivuuden toisten ohjelmatuotteiden ja käytössäolevien järjestelmien kanssa, eivät Parnaksen mukaan ole koskaan helmiä, vaikkakin ne ovat hyvin käyttökelpoisia.

Suorituskykytavoitteet ja laitteistorajoitteet ovat usein ristiriidassa siistin rakenteen kanssa. Artikkelissaan Parnas kertoo johtamansa ryhmän kokemuksista USA:n laivaston A-7E lentokoneen ohjausjärjestelmän toteutuksesta 1970-luvulla. Heille oli asetettu kaksi reunaehtoa: laitteistoa ja käyttöliittymää ei saanut muuttaa. Projektin aikana Parnasin ryhmä kehitti ja julkaisi monia hyödyllisiä ohjelmistosuunnittelun välineitä. Kuitenkin Parnas toteaa ryhmänsä epäonnistuneen yrityksessään toteuttaa toimiva helmi. Keskeisimmäksi epäonnistumisen syyksi Parnas nostaa laitteistorajoitteet. Jotta suorituskykyvaatimukset olisivat täyttyneet, niin annetulla laitteistolla olisi tarvittu lähes optimaalinen rekisterimuistin käyttö. Parnasin ryhmä oli kuitenkin asettanut tavoitteekseen laitteistoriippumattomuuden. Heidän olisi pitänyt kehittää yleispätevä tehokas rekiste-

rimuistin allokointialgoritmi. Tätä he eivät onnistuneet tekemään. Tähän päivään mennessä (syyskuu 2005) kukaan muukaan ole patentoinut tällaista ratkaisua.

Nykyisin useimmat käytössä olevat ohjelmistot ovat vuosien saatossa laajentuneet. Niitä ei ole laadittu ns. *"puhtaalta pöydältä"*. Jos suunnittelijat saisivat aloittaa alusta, useimmat ohjelmistotuotteet näyttäisivät hyvin erilaisilta. Parnas mainitsee esimerkin tietoliikenneohjelmistosta, jonka koosta arvioilta 75 % on historian painolastia. Jos vanha ohjelmistokanta ja vanhat työtavat voitaisiin unohtaa, niin useimmat ammattitaitoiset suunnittelijat pystyisivät Parnasin mukaan tuottamaan yksinkertaisia ja tehokkaita ohjelmia ja ohjelmistoja.

Teknologiaa ihannoivassa yhteiskunnassa kunnioitetaan originaalisuutta ehkä aivan liikaa. Luovuus ja originaalisuus ovat arvokkaita, kun tarvitaan parannuksia. Ne ovat välttämättömiä, kun yritämme ratkaista ongelmia, joihin ei vielä ole kelvollista ratkaisua.

Ohjelmistohelmien suunnittelijat ovat usein pystyneet oppimaan aiemmin tehdyistä virheistä. Käyttämällä aikaa aiempien yritysten tutkimiseen ja analysointiin he ovat ymmärtäneet aiempien lähestymistapojen olennaiset heikkoudet. Liian monet ohjelmistotuotteet tekevät samat virheet kuin aiemmat. Historiasta ei ole opittu mitään. Toisaalta hyviäkin aiempia ratkaisuja jätetään hyödyntämättä.

Usein väitetään, että pyörää ei kannata keksiä uudelleen. Parnasin väittämä on, että pyörä on keksitty niin moneen kertaan, koska se on erinomainen idea. Parnas kertookin vuosien varrella oppineensa, että ajatuksiin, jotka on keksitty vain kerran, on syytä suhtautua hyvin epäluuloisesti.

Usein ohjelmistohelmi on aikaansaatu uuden ohjelmointikielen avulla. Jotkut suunnittelijat katsovat ratkaisunsa menestyksen johtuvan pääasissa tuosta uudesta ohjelmointikielestä. Parnas kertoo olevansa hyvin skeptinen tällaisiin väitteisiin. Lukuisten uusien ohjelmointikielien on väitetty ratkaisevan ohjelmointiin ja ohjelmistojen tekemiseen liittyvät ongelmat. Kuitenkin nämä ongelmat toistuvat jatkuvasti.

Parnas väittääkin, että ongelman ydin on suunnittelussa, ei ohjelmointikielessä. Parnas mainitsee nähneensä kauniita ja selkeitä ohjelmia, jotka on kirjoitettu symbolisella konekielellä (*assembler*), Fortranilla tai C:llä. Heti perään hän kuitenkin toteaa nähneensä näillä kielillä tehtyjä huonoja ohjelmia. Ohjelmoinnista puhutaan lisää luvussa 7.1.

Uuden ohjelmointikielen suunnittelu jokaista uutta ohjelmistoa varten on ylellisyyttä, joka vain hyvin harvoin on mahdollista. Meidän on useimmiten tultava toimeen nykyisillä ohjelmointikielillä. Parnas kritisoi myös

asennetta, että ohjelmointikielen olisi estettävä ohjelmointivirheet. Tätä hän vertaa veitseen, joka leikkaisi lihaa mutta ei sormia. Parnasin mukaan työn tekeminen hyvin edellyttää teräviä työkaluja. Jos työkalu estää virheiden tekemisen, se myös estää oikeiden ja tehokkaiden ratkaisujen tekemisen. Tästä hyvänä esimerkkinä on nykyinen Java. Ohjelmointivirheiden tekeminen on melko hankalaa, mutta samalla tehokkaiden oikein toimivien ratkaisujen tekeminen on usein lähes mahdotonta.

Ohjelmistohelmiin tutustumisen tärkeimmäksi opetukseksi Parnas nostaa sen tosiasian, että ohjelmistohelmien tekijät näyttävät käyttäneen runsaasti aikaa järjestelmiensä rakenteen ajatteluun ennen kuin he ovat aloittaneet ohjelmien kirjoittamisen. Lukion matematiikan opettajani Risto Saarela kuvasi tätä deduktiivista lähestymistapaa oivallisesti fraasilla *ensin tutkitaan, sitten hutkitaan.*

Yksinkertaisuus ei synny itsestään. Se vaatii runsaasti ajattelua ja suunnittelua. Hyviä ohjelmia ei vain kirjoiteta. Ne on suunniteltu huolellisesti.

Kirjoituksensa Parnas päättää omien opettajiensa antamiin perussääntöihin:

1. Suunnittele ennen toteutusta.
2. Dokumentoi suunnittelu.
3. Katselmoi ja analysoi dokumentoitu suunnittelu.
4. Katselmoi, että toteutus vastaa suunnittelua.

Nämä pätevät niin ohjelmistoissa kuin laitteistoissa.

6.2 Suorituskyky

Suorituskykyä (*performance*) kuvaavia keskeisiä suureita ovat

suoritusteho (*throughput*) kertoo kuinka paljon hyödyllistä työtä tehdään aikayksikössä;

vasteaika (*response time*) kertoo kauanko tietyn tehtävän suorittamiseen kuluu;

käyttöaste (*utilization*) kertoo osuuden ajasta, jonka laite on aktiivisena.

Suorituskykyä tutkitaan mittaamalla järjestelmän toimintaa, matkimalla eli simuloimalla järjestelmän keskeisten osien toimintaa sopivalla tarkkuudella sekä laatimalla järjestelmästä matemaattisia malleja, joista keskeiset suorituskykysuureet lasketaan.

Suorityskykyanalyysin (*performance analysis*) yksi osa-alue on ohjelmistojen suorituskyvyn suunnittelu (*software performance engineering*), joka on myös osa ohjelmistotekniikkaa. Tavoitteena on jo ohjelmiston suunnitteluvaiheessa arvioida tulevan järjestelmän tehokkuutta (*efficiency*) ja tarvitsemaa kapasiteettia (*capacity*) eli laitteistoresursseja. Suorituskyvyn tarinaksi olen valinnut Daniel A. Menascén kesäkuussa 2002 pitämän esitelmän ”Software, Performance, or Engineering?” ACM:n WOSP’02 (Workshop on Software and Performance) -konferenssissa.

Menascén mukaan useimmat kiinnostavat ohjelmistojärjestelmät ovat monimutkaisia suunnitella ja toteuttaa. Ohjelmistojen vaatimukset (*requirements*) jaetaan toiminnallisiin (*functional*) ja ei-toiminnallisiin (*non-functional*) vaatimuksiin. Niin kutsutut ei-toiminnalliset vaatimukset kattavat ominaisuuksia, kuten tietoturva (*security*), saatavuus (*availability*), luotettavuus (*reliability*), ja suorituskyky. Vasteajan ja suoritustehon lisäksi suoritettujen (tai suorittamatta jääneiden) pyyntöjen osuus on keskeinen suorituskykysuure.

Ohjelmistotekniikan useimmat menestystarinat ovat liittyneet menetelmiin ja työkaluihin, joilla hallitaan ohjelmistojen laatimisen monimutkaisuutta. Nämä kattavat suunnitelun, vaatimusten, testitapausten, konfiguraatioiden, versioiden ja kehittämisen hallinnoinnin.

Menascén mukaan on valitettava tosiasia, että realiaikasovelluksia[1] (*real-time appilcation*) lukuunottamatta suorituskykyvaatimukset vain hyvin harvoin otetaan huomioon suunnitteluvaiheessa Yhdeksi syyksi Menascé mainitsee, että suorituskykyvaatimuksia kutsutaan ohjelmistotekniikassa ei-toiminnallisiksi vaatimuksiksi. Aiheellisesti Menascé kysyykin, miten ohjelmisto voi toimia kelvollisesti, jos jotkut—esimerkiksi suorituskykyvaatimukset—jäävät täyttämättä. Toisin sanoen, ohjelmistotekniikassa pitäisi lopettaa puhuminen ei-toiminnallisista vaatimuksista: kaikkien vaatimusten pitäisi olla toiminnallisia.

Ohjelmistojen suorituskyvyn suunnittelusta puhumisen aloitti Connie Smith vuonna 1981 Computer Measurement Groupin konferessissa. Esittämässään paperissa Smith kehoitti kiinnittämään huomiota siihen tosi-

[1] Reaikajärjestelmä takaa, että kaikki tai useimmat tehtävät tulevat tehtyä määräaikaan mennessä.

asiaan, että ohjelmistotekniikassa suorituskyvystä ruvetaan huolestumaan vasta ohjelmiston kehitysprosessin viime metreillä. Tämä ”korjaa-myöhemmin” (*fix-it-later*) -asenne heijasti sitä tosiasiaa, että suorituskyvyn ei katsottu kuuluvan suunnitteluun vaan ainoastaan viimeistelyyn.

Perinteisessä insinööritaidossa tällainen asenne on käsittämätön. Näillä aloilla järjestelmät suunnitellaan alusta alkaen tehokkaiksi, täyttämään suorituskykyvaatimukset. Tietojenkäsittelyssä Mooren laki (prosessorien nopeus kaksinkertaisuu n. 18 kuukauden välein) on mahdollistanut tehottomien ohjelmistojen tekemisen.

Menascé hämmästelee, kuinka ohjelmistoteknikka kehtaa kutsua itseään insinööritaidoksi, kun ohjelmistojen suorituskyvyn suunnittelu ei ole tullut osaksi jokapäiväistä ohjelmistotuotantoa. Selvitykseksi Menascé tarjoaa:

1. tieteellisten periaatteiden ja mallien puute,
2. koulutus,
3. tietojenkäsittelyn ”ammattilaiset”,
4. yhden käyttäjän asenne ja
5. pienen tietokannan asenne.

Tieteellisten periaatteiden ja mallien puute

Perinteisten insinööritaitojen alueilla käytetään matematiikkaan, fysiikkaan ja laskennallisiin tieteisiin perustuvia tieteellisiä periaatteita ja malleja. Sen sijaan ohjelmistotekniikassa ei systemaattisesti käytetä formaaleja ja kvantitatiivia malleja osana ohjelmiston elinkaarta.

Ohjelmistotekniikassa on kehitetty erilaisia formalismeja tukemaan ohjelmiston elinkaarta. Suurin osa tästä työstä on keskittynyt menetelmiin, joilla hallittaisiin ohjelmistoprosessien, testauksen, ylläpidon (*maintenance*) ja kehitettävyyden monimutkaisuutta. Ohjelmistotekniikan yhteisön julkaisuista yli 80 % käsittelee toiminnallisiin vaatimuksiin liittyviä formalismeja ja menetelmiä.

Ohjelmistojen suorituskyvyn suunnittelu ei ole myöskään ollut suorituskykyanalyysin yhteisön suurimman mielenkiinnon kohteena. Suorituskykyanalyysissa on keskitytty suorituskyvyn ja resurssitarpeiden välisiin suhteisiin. Jos sovelluksia ei eksplisiittisesti malliteta, niin suorituskykymalleja ei voida helposti käyttää ohjelmistojen suunnitteluvaihtoehtojen suorituskykyvaikutusten analysoimiseen.

Koulutus

Menascén mukaan tietojenkäsittelystä valmistuvat kandidaatit eivät ole usein valmiita teollisuudessa vastaantuleviin ohjelmistotekniikan haasteisiin. ACM:n ja IEEE:n tietojenkäsittelytieteen opetussuosituksissa ei ole tietokonejärjestelmien suorituskyvyn analysoinnin pakollista kurssia. Suosituksessa on vain muutama luentotunti suorituskyvystä, yleensä käyttöjärjestelmä- ja tietoliikennekursseilla. Perusteeksi on usein mainittu, että kaikki hyödyllinen aines ei mahdu tutkintoon.

Tietojenkäsittelyn ”ammattilaiset”

Vuosituhannen vaihteen it-buumin seurauksena tietojenkäsittelyn suunnittelu- ja ohjelmointitehtävissä toimii runsaasti työntekijöitä, joilla ei ole minkäänlaista alan muodollista koulutusta. Siksi ei ole yllättävää, että monet nykyiset ohjelmistot kärsivät vakavista suorituskykyongelmista.

Yhden käyttäjän asenne

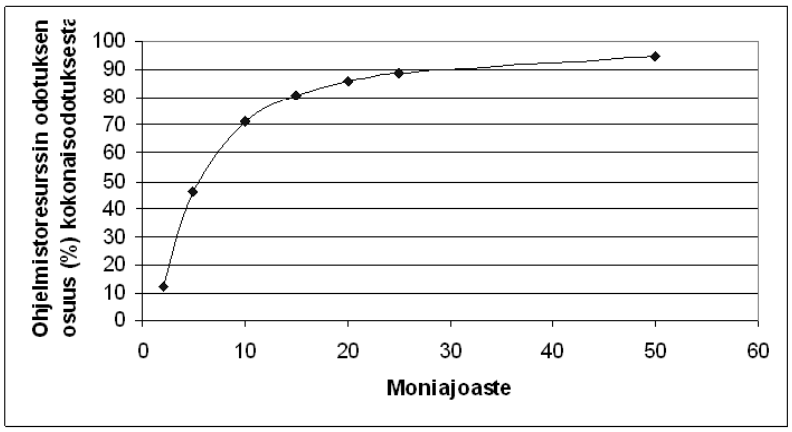
Menascén mukaan useimmat suunnittelijat ja ohjelmoijat valitettavasti kehittävät järjestelmiään yhden käyttäjän asenteella. Toisin sanoen ohjelmistojen kehittäjät eivät ota huomioon sitä, että useimmissa järjestelmissä on useita samanaikaisia käyttäjiä. Samanaikaisuus aiheuttaa kilpailua sekä laitteisto- (prosessorit, muisti, talletusvälineet, tietoliikenneyhteydet) että ohjelmistoresursseista (tietokantalukot, kriittiset alueet, ohjelmistosäikeet).

Kuvassa 6.1 on esimerkki. Oletetaan, että pyynnön käsittelyyn tarvitaan 0,3 sekuntia prosessoriaikaa, josta 0,1 sekuntia tarvitaan kriittisellä alueella[2]. Kuvan käyrä näyttää, että moniajoasteen lisääntyessä kilpailu ohjelmistoresurssista eli kriittisestä alueesta alkaa dominoida vasteaikaa.

Pienen tietokannan asenne

Pienen tietokannan asenteella Menascé tarkoittaa, että tietokannan käyttöä toteutettaessa ei kovinkaan usein ole selkeästi otettu huomioon käsiteltävän tietokannan kokoa. Tietokannan koko, johon SQL-kysely kohdistuu, vaikuttaa olennaisesti kyselyn suorituskykyyn. Jos tietokannassa on miljoonia rivejä, niin tehokas tietokannan käyttö saattaa edellyttää

[2]Krittisellä alueella voi olla vain yksi käyttäjä kerralla.



Kuva 6.1: Ohjelmistoresurssin kilpailuajan osuus [Menascé, 2002]

esimerkiksi aputaulujen käyttöä. Useimmiten tehokkain kyselystrategia $10,000$ rivin tietokantaan on erilainen kuin miljoonan rivin tietokantaan.

Menascén johtopäätökset

Artikkelinsa johtopäätökset Menascé aloittaa toteamalla, että ohjelmiston monimutkaisuus aiheuttaa usein tehottomuutta. Vuosien varrella on myös havaittu, että paras tapa hyökätä monimutkaisuutta vastaan on parantaa ohjelmoijien ammattitaitoa, eikä kehittää heidän käyttämiään työkaluja ja tekniikoita [Glass, 2001]. Siksi tehokkain tapa tuottaa tehokasta ohjelmistoa on kouluttaa ohjelmistojen suunnittelijat ja ohjelmoijat kunnollisesti suorituskykyyn liittyvissä asioissa. Ohjelmiston tehokkuus riippuu useimmiten enemmän hyvästä suunnittelusta kuin suunnitelman hyvästä toteutuksesta.

6.3 Luotettavuus

Otsikon luetettavuus (*reliability*) alle Peter Denning on sijoittanut osa-alueet toisteisuus (*redundancy*), toipuminen (*recovery*), varmistaminen

(*checkpointing*), eheys (*integrity*) ja luottamus (*trust*). Tarinaksi olen vallinnut George Candean ja Armando Foxin pääasiassa toipumista käsittelevän USENIX yhteisön (`http://www.usenix.org`) Hot Topics in Operating Systems työpajassa toukokuussa 2003 esitetyn artikkelin *Crash-Only Software* [Candea & Fox, 2003]. Candean ja Foxin kirjoitus eroaa olennaisesti muusta tämän kurssin taustamateriaalista. Artikkelissa esitetään radikaalisti uudenlaista lähestymistapaa, joka ei ainakaan vielä ole tietojenkäsittelytieteen vakiintunutta käytäntöä.

Lähestymistavan perusajatus on, että ohjelmien kaatuminen (*crash*) on tehtävä turvalliseksi ja siitä toipuminen (*recovery*) nopeaksi. Artikkelin kirjoittajat ehdottavat, että ainoa tapa päättää ohjelman suoritus olisi kaataminen, ja että ainoa tapa käynnistää ohjelma olisi käynnistää toipuminen. Käsitteenä 'crash-only' siten kuin Candea ja Fox sen esittävät on kohtuullisen hyvin määritelty. Valitettavasti en ole pystynyt muodostamaan tälle käsitteelle siedettävää suomenkielistä vastinetta. Siksi tässä tarinassa esiintyy termi 'crash-only' kääntämättömänä. Candean ja Foxin käsitteelle 'crash-safe' käytän (huonohkoa) suomenkielistä vastinetta *turvallinen kaatuminen* ja sen erilaisia johdannaisia.

Candea ja Fox aloittavat artikkelinsa mielenkiintoisella toteamuksella. Usein järjestelmän hallittu alasajo ja uudelleen käynnistäminen on hitaampaa kuin järjestelmän kaataminen ja käynnistäminen toipumisen kautta. Alla on heidän esittämät luvut:

järjestelmä	uudelleen-käynnistämis-aika	kaatamis-toipumis-aika
RedHat 8 (ext3fs)	104 s	75 s
JBoss 3.0 sovelluspalvelin	47 s	39 s
Windows XP	61 s	48 s

Useissa tilanteissa on epäkäytännöllistä yrittää rakentaa tietojenkäsittelyjärjestelmää, joka ei missään tilanteessa kaadu. Siksi järjestelmien on varauduttuva kaatumiseen. Aiheellisesti Candea ja Fox kysyvätkin, miksi järjestelmään on rakennettava sekä hallittu alasajo että kaatumiseen varautuminen.

Perinteisillä insinööritaidon alueilla käytetään fysiikan lakeja fyysisten järjestelmien rakentamisessa ja niiden käyttäytymisen ymmärtämisessä. Ohjelmisto on kuitenkin abstraktio, jolla ei ole fyysistä ruumiillistumaa. Siten ohjelmisto ei noudata fyysisiä lakeja. Tietojenkäsittelytieteilijät ovat yrittäneet käyttää erilaisia formaaleja menetelmiä, joiden avulla

voitaisiin päätellä ohjelmistojen käyttäytymistä. Nämä menetelmät perustuvat kuitenkin usein ohjelmiston abstraktiin malliin, joka ei täydellisesti kuvaa ajoaikaisen järjestelmän toimintaa, mikä sisältää myös laitteiston, käyttöjärjestelmän ja kirjastojen vaikutukset. Siksi formaaleilla menetelmillä ei juuri koskaan pystytä kuvaamaan toteutuksen todellista käyttäytymistä riittävän tarkasti, koska koko järjestelmän kaikilla fyysisesti mahdollisilla tiloilla ei ole vastinetta abstraktissa mallissa.

Järjestelmän toipumisesta huolehtiva ohjelmakoodi käsittelee poikkeustilanteita ja siksi sen olisi toimittava virheettömästi. Valitettavasti poikkeukselliset tilanteet ovat vaikeasti käsiteltäviä. Ne esiintyvät harvoin, eikä niitä ole helppo saada aikaan kehitysvaiheessa, jotta toipumista päästäisiin testaamaan. Siksi toipumisesta huolehtiva ohjelmakoodi on usein epäluotettavaa.

Yhdeksi oman lähestymistapansa eduksi Candea ja Fox nostavat, että ’crah-only’ -järjestelmässä toipumismekasmia käytetään aina, kun järjestelmä käynnistetään. Siksi tämän tulisi parantaa toipumiskäsittelyn luotettavuutta. Empiirisesti on havaittu [Chou et al., 2001], että järjestelmien monimutkaistuessa ohjelmointi- ja suunnitteluvirheiden löytäminen hidastuu. Enemmän virheitä tarkoittaa useampia häiriöitä (*failure*), joten järjestelmien on toivuttava useammin.

’Crash-only’ järjestelmien useat edut on aiemmin havaittu tiedon talletukseen ja tiedonhakuun erikoistuneissa järjestelmissä. Siellä käytetään käsitettä transaktio (*transaction*) kuvaamaan toimenpidesarjaa, joka suoritetaan joko kokonaan tai ei ollenkaan. Transaktionaalisuuden takaavat järjestelmät pitävät huolen siitä, että toimenpidesarjan vaikutukset näkyvät muille vasta sitten, kun kaikki transaktioon kuuluviat toimenpiteet suoritukset on saatu päätökseen. Candea ja Fox tavoittelevat samantapaista lopputulosta Internet-järjestelmien häiriötilanteisiin. He kuitenkin toteavat, että transaktioiden niin kutsutut ACID-ominaisuudet—atomisuus, eheys, eristettävyys (*isolation*) ja pysyvyys (*durability*)—eivät ole aina välttämättömiä Internet-maailmassa.

’Crash-only’ -järjestelmä mahdollistaa yksinkertaisen vikamallin. Koska toipuminen on nopeaa, jokainen häiriö voidaan käsitellä ohjelmistokomponentin kaatumisena. Komponenttien on toivuttava vain yhdestä häiriöstä eli kaatumisesta. Esimerkkinä tällaisesta järjestelmästä on klusteroitu Web-palvelin [Nagaraja et al., 2002], jossa kaikki tunnistamattomat viat aiheuttivat solmun kaatumisen. Tämä ratkaisu paransi huomattavasti kyseisen Web-palvelimen saatavuutta (*availability*).

Järjestelmissä, joissa kaatuminen ja toipuminen on nopeaa, voidaan

käyttää ennakoivaa uudelleenkäynnistystä. Itsediagnosoinnin avulla järjestelmä voi uudelleenkäynnistää epäilyttävät komponentit.

'Crash-only' ohjelmiston ominaisuuksia

Jotta ohjelmistokomponentti olisi 'crash-only', sen on talletettava kaikki ei-tilapäinen tilatieto erityiseen tilatietovarastoon (*state store*). Tämän tietovaraston on tarjottava sopivat abstraktiot ja sen on oltava 'crash-only'. Jotta erillisistä komponenteista rakennettava järjestelmä olisi 'crash-only', se on suunniteltava siten, että komponentit varautuvat muiden komponenttien kaatumisiin ja tilapäisiin käytöstä poissaoloihin (*unavailability*). Tämä puolestaan edellyttää järjestelmän modulaarisuutta, suhteellisen läpäisemättömiä rajoja, ajastimiin perustuvaa kommunikointia, laina-aikaan (*lease*) perustuvaa resurssien varausta, sekä pyyntöjä, jotka kuvaavat itsensä riittävän täydellisesti. Monet Internet-järjestelmät sisältävät jo nykyisin joitakin näitä piirteitä. Candean ja Foxin mukaan mikään heidän tuntemansa järjestelmä ei vielä sisältänyt kaikkia 'crash-only' -järjestelmän tarvitsemia ominaisuuksia.

Nykyisissä Internet-sovelluksissa on vain muutamaa perustyyppiä olevaa tilatietoa: transaktionaalinen pysyvä tilatieto, yhden lukijan ja kirjoittajan pysyvä tilatieto (esimerkiksi käyttäjän profiili), istunnon tilatieto (esimerkiksi edellisen haun tulos myöhempää tarkentavaa hakua varten), ei-kriittinen tilatieto (*soft state* eli tilatieto, joka voidaan koska tahansa koota uudelleen muista lähteistä) ja tilapäinen tilatieto. Näiden tilatietotyyppien keskeisin ero liittyy taattuun elinaikaan. Talletusjärjestelmälle ne asettavat kuitenkin hyvin erilaiset vaatimukset. Siten myös tarvittavat toteutustavat ovat erilaisia.

Kun erikoistuneet tilatietovarastot hallitsevat kaiken tärkeän ja ei-tilapäisen tilatiedon, niin sovellusohjelmien huoleksi jää vain ohjelmalogiikan toteuttaminen. Erikoistuneet tietovarastot, kuten relaatio- tai oliotietokannat, tiedostojärjestelmät, hajautetut tietorakenteet, ei-transaktionaaliset hajautustaulut (*hashtable*) tai istunnon tilatietovarastot, soveltuvat paljon paremmin tilatiedon hallintaan kuin sovellusohjelmoijien kirjoittamat ohjelmat. Tavallisilla ohjelmoijilla on useimmiten vain minimaalinen kokemus järjestelmätason ohjelmoinnista. Sovellukset ovat tietovarastojen tilattomia asiakkaita, mikä mahdollistaa yksinkertaisemmat ja nopeammat toipumisrutiinit.

Tilatietovarastojen on tietenkin oltava 'crash-only'. Mikäli näin ei ole, niin ongelma on siirretty vain kerrosta alemmaksi. Useimmat nykyisin

saatavilla olevat tietovarastot takaavat turvallisen kaatumisen. Toisin sanoen ne osaavat kaatua siten, että tietoa ei huku. Useimmat tietokannat ja verkkojen talletuslaitteet eivät ole kuitenkaan ’crash-only’, koska toipuminen on yleensä melko hidasta.

Tietovarastojen on tarjottava sovellusten vaatimuksiin sopivat abstraktiot ja takuut, ei liian voimakkaita eikä liian heikkoja. Mikäli varaston tarjoama abstraktio on liian heikko, niin sovellus joutuu itse tekemään osan tilatiedon hallinnasta. Liian voimakkaat abstraktiot puolestaan ovat turhan hitaita ja kuluttavat turhan paljon tietojenkäsittelyresursseja.

’Crash-only’ -järjestelmien on varauduttava alijärjestelmien tilapäisiin käytöstä poissaoloihin. Komponenttien välillä on oltava rajat, jotka estävät häiriöiden heijastumisen muiden komponenttien toimintaan. Virtuaalikoneet, mikroytimet ja käyttöjärjestelmien prosessit ovat menetelmiä, joilla voidaan toteuttaa halutun tasoinen eristettävyys.

Kaikilla komponenttien välisillä vuorovaikutuksilla on oltava aikaraja. Jos vastaus ei tule aikarajaan mennessä, niin pyynnön lähettäjä olettaa, että vastaanottaja on vikaantunut, ja raportoi häiriötilanteen toipumismekanismille. Uudelleenkäynnistyksellä (kaataminen-topuminen) varmistetaan, että kutsuttu komponentti on tunnetussa tilassa.

Kaikilla resursseilla on varausaika. Tämä koskee myös prosessorin resursseja. Jos komponentti ei saa uusittua varaustaan, niin sen suoritus päätetään.

Kaikkien pyyntöjen on oltava itsensä kuvaavia, kerrottava eksplisiittisesti tarvitsemansa käsittelyn tilatieto ja konteksti. Tämän avulla uudelleenkäynnistetty komponentti voi suorittaa pyynnön uudelleen. Erityisesti pyyntöjen on kerrottava, ovatko ne idempotentteja, ja mikä on niiden aikaraja. Idemponentti operaatio on sellainen, jonka tulos ei muutu, vaikka se suoritettaisiin useamman kuin yhden kerran. Siten toivuttaessa idempotentti pyyntö voidaan yksinkertaisesti suorittaa pyynnön uudestaan. Jos uudelleen suoritettava pyyntö ei ole idempotentti, niin järjestelmän on palautettava keskeytettyä operaatiota edeltänyt tila, tehtävä komponsoivia toimenpiteitä, tai siedettävä uudelleensuorituksen aiheuttama epäkonsistentti tila.

Aikarajojen lisäksi häiriöiden havaitsemisessa voidaan käyttää perinteistä sydämenlyöntimekanismia (*heartbeats*). Tällöin komponentit lähettävät aika-ajoin erityisiä ’hengissä ollaan’ viestejä. Lisäksi voidaan käyttää edistymistä seuraavia laskureita. Erityisesti tilatietovarastot ja sanomanvälitysjärjestelmät voivat tehokkaasti seurata laskurien avulla komponenttien edistymistä ja aktiivisuutta.

6.4 Kehitettävyys

Ohjelmistotekniikan kriisistä on puhuttu 1960-luvun lopulta, jolloin NATO:n tiedekomitea järjesti kaksi ohjelmistotekniikan konferenssia[3]. David Parnas toteaa artikkelissaan *Software Aging* [Parnas, 1994], että kyseessä ei voi olla kriisi, koska siitä oli puhuttu ainakin viimeiset 25 vuotta. Kriisi on äkillinen, lyhytaikainen ja vakava hätätila. Toisaalta hän toteaa, että ohjelmistokriisiksi kutsuttu ilmiö on vakava. Koska se ei ole yllättävä eikä lyhytaikainen, sitä ei pidä käsitellä kriisinä, vaan pitkäaikaista hoitoa vaativana kroonisena tautina.

Perusongelmaksi Parnas nostaa ohjelmistojen ikääntymisen. Kriisiapu johtaa lyhytjänteiseen ajatteluun ja nopeasti ikääntyviin ohjelmistoihin. Tämä ohjelmistojen ikääntyminen on keskeinen tekijä ohjelmistojen kehitettävyydessä (*software evolution*) ja ylläpidettävyydessä (*maintenance*). Kehitettävyyden tarinan pohjaksi olen valinnut Meir Lehmanin tammi-helmikuussa 1998 IEEE Software -lehdessä julkaistun artikkelin *Software's Future: Managing Evolution.*

Tilannekatsaus 1998

Artikkelinsa Lehman aloittaa tilannekatsauksella. Keskeinen trendi on jatkuva tietokoneistuminen. Koska tietokoneet tunkeutuvat kaikkialle sekä yhteiskunnan ohjelmistoriippuvuus että käyttäjien riippuvuus toisistaan kasvaa ylilineaarisesti. Tämä luo sekä ulkoisia että sisäisiä paineita muuttaa ohjelmistokehityksen arviointia, suunnittelua ja hallinnointia.

Kun organisaatio lisää tietotekniikan hyväksikäyttöään, eri tietojenkäsittelyjärjestelmien integrointi nousee keskeiseksi vaatimukseksi. Alunperin irralliset järjestelmät on saatava keskustelemaan keskenään. Siitä huolimatta niiden on edelleen pystyttävä toimimaan organisaation aiempien prosessien ja käytäntöjen kanssa. Alkuaan hyvinkin löyhä kytkös (*loose coupling*) muuttuu ajan myötä yhä kiinteämmäksi (*tight coupling*). Määriteltäessä uusia sovelluksia, etsittäessä yhdistämisiä ja integroimalla jo toiminnassa olevia järjestelmiä, kehittäjien on rajattava sovellusten käyttöaluetta, laajuutta ja yksityiskohtia (kilkuttimia—*bells and whistles*). Lopputuloksena on rajoituksia, jotka saattavat muodostua pullonkauloiksi ja

[3]Näiden konferenssien raportit (*Software Engineering* ja *Software Engineering Techniques*) löytyvät URLista `http://homepages.cs.ncl.ac.uk/brian.randell/NATO/`.

ärtymyksen aiheuttajiksi. Käyttäjien turhautuminen, kasvava vaatimustaso ja toimintaympäristöjen lisääntyvä riippuvuus toisistaan synnyttää jatkuvia muutospaineita ja loputtomia yhtenäistämisvaatimuksia.

Järjestelmiin kohdistuu myös ulkoisia käyttöalueista johtuvia paineita. Vuosituhannen vaihtumisen (Y2K ongelma) aiheuttamat muutospaineet olisi ollut vältettävissä, mikäli 1900-luvulla tehdyt järjestelmät olisi alunperin suunniteltu toimiviksi myös 2000-luvulla. Jotkut ulkoiset paineet johtuvat poliittisista päätöksistä, jotka eivät ota huomioon niiden vaikutuksia tietojenkäsittelyyn. Tällaisia ovat olleet esimerkiksi euroon siirtyminen ja työeläkejärjestelmän maksuperusteiden muutokset. Myös teknologiset muutokset ovat vaatineet ohjelmistojen muuttamista. Esimerkiksi puhelinnumerot ovat useaan kertaan muuttuneet laajentuneen puhelinkannan vuoksi.

Riippumatta siitä mikä on ollut ongelman alkuperä, niin tulos on ollut sama: turhautuneisuutta, kustannuksia, pienempiä tai suurempia onnettomuuksia, jotka ovat vaatineet välittömiä muutoksia ohjelmistoihin. Lisäksi käyttäjäkunnan vaatimustaso kasvaa jatkuvasti. Käyttöalueet muuttuvat, teknologia muuttuu, lainsäädäntö ja talous muuttuu. Suunnittelun ja toteutuksen tarjoamat mahdollisuudet ovat yhtä rajattomat. Elämän tosiasiat johtavat loputtomaan ohjelmistojen muutoksiin, parannuksiin ja kehitykseen sekä jatkuvaan uusien ohjelmistoversioiden ilmaantumiseen.

Huomioita ohjelmistoista

Rakenteeltaan, sisällöltään ja toiminnallisuudeltaan ohjelmistot ovat monimutkaisimpia ihmisen luomia järjestelmiä. Ohjelmisto itsessään on malli sovellutuksesta, osallistujista (ihmiset, organisaatiot, laitteet, koneet), käyttöalueesta ja kyseisen alueen toiminnoista. Periaatteessa koko maailmankaikkeus muodostaa käyttöalueen, vaikkakin useimmat maailmankaikkeuden osat ja ominaisuudet ovat sovellutuksen kannalta epäolennaisia.

Tässä luvussa olen tehnut eron sovelluksen ja sovellutuksen välillä, vaikka niitä useimmiten pidetään synonymeina. *Sovelluksella* tarkoitan tehtävän ratkaisevaa tietokoneohjelmaa ja *sovellutuksella* tehtävän ratkaisuprosessia. Sovellutus siis käsittää sovelluksen ja tehtävän ratkaisemiseen tarvittavat muut toimenpiteet.

Lehman nostaa CERN:in erään hiukkaskiihdyttimen käyttöönotossa kohdatut ongelmat esimerkiksi siitä, miten aiemmin epäolennaisesta yksityiskohdasta tuli merkittävä. Uusi hiukkaskiihdytin oli halkaisijaltaan

kaksinkertainen edeltäjäänsä verrattuna. Sitä kuitenkin ohjattiin samalla ohjausohjelmistolla kuin edeltäjäänsä. Samaten tulosten tulkinta-ohjelmisto oli sama. Tulokset olivat katastrofaalisia, ne vaihtelivat päivästä toiseen. Edes edellisellä kiihdyttimellä saatuja tuloksia ei pystytty toistamaan. Lopulta joku huomasi, että tulokset vaihtelivat kuun vaiheen mukaisesti. Aiempi perusteltu oletus, että kuun vetovoimaa ei tarvitse ottaa huomioon, ei pitänytkään enää paikkaansa. Ainoa mahdollisuus oli kirjoittaa kaikki ohjelmat uudestaan ottaen huomioon uuden kiihdyttimen ominaisuudet.

E-ohjelmistolla Lehman tarkoittaa ohjelmistoja, joita käytetään todellisen maailman sovellutuksissa, ongelmissa tai toiminnoissa. Ohjelmisto on aina äärellinen. Siten e-ohjelmisto on äärellinen ja epätäydellinen malli rajoittamattoman käyttöalueen rajoittamattomasta sovellutuksesta. Jotta asiat eivät olisi liian yksinkertaisia, niin ohjelmansuorituksesta tulee osa sovellutusta ja sen käyttöaluetta. Siten ohjelmalla olisi oltava malli itsestään ja toiminnastaan. Ohjelman rajallisuudesta seuraa, että ohjelma pystyy ottamaan oman käyttäytymisensä toimintaympäristössään huomioon vain epätarkasti ja epätäydellisesti.

Rajallisen järjestelmän ja rajoittamattoman käyttöalueeen rajoittamattoman sovellutuksen välillä on aina välttämättä kuilu. Tämä kuilu paikataan oletuksilla, kuten algoritmien valinnoilla ja parametrien arvoilla. Järjestelmän valinta, määrittely, suunnittelu ja toteutus tuovat mukaan lukuisia oletuksia. Jotkut oletukset ovat eksplisiittisiä, kuten vaatimusmäärittelyn aikana tehdyt valinnat. Toiset ovat implisiittisiä juontuen valitusta teoriasta, algoritmin suunnittelusta, proseduurin valinnasta, rajapinnan määrittelystä tai raja-arvojen valinnoista.

Lehman arvioi, että tyypillisessä e-ohjelmistossa on jokaista kymmentä ohjelmariviä kohden yksi todellisuudesta tehty oletus. Jotkut näistä oletuksista pitävät paikkansa ohjelmiston koko elinkaaren ajan. Toisten paikkansapitävyys päättyy sovellutuksen tai käyttöalueen muutoksiin. Hankalimpia on väliinputoajat eli oletukset, jotka muutosten vuoksi pitävät toisinaan paikkansa, mutta toisinaan saavat aikaan virheitä. Esimerkkinä Lehman mainitsee Euroopan avaruusjärjestön ESA:n Ariane 5 -raketin ensilennon kesäkuulta 1996. Ohjausjärjestelmässä oletettiin, että 64-bitin liukuluvut voitiin muuntaa 16-bitin kokonaisluvuiksi. Tehokkuussyistä tarkistukset puuttuivat ja kokonaislukujen ylivuodon virhekäsittelijä oli otettu pois päältä. Raketti tuhosi itsensä 40 sekunttia laukaisun jälkeen, koska kantorakentin suuttimia ohjattiin väärillä arvoilla.

Tarkempi kuvaus löytyy WikipediA:sta[4].

Ohjelmistot ovat staattisia niin kauan kun ihmiset eivät muuta niitä. Itsestään ohjelmisto ei voi mukautua ulkoisiin muutoksiin. Ainoastaan siinä tapauksessa, että ohjelmiston laatijat varautuvat muutoksiin ja toteuttavat tarvittavat mukautumismekanismit, ohjelmisto voi olla itsestään mukautuva. Mukautuvuus ja joustavuus voidaan saada ohjelmakoodiin vain siinä määrin kuin epävarmuustekijät ja mahdolliset muutostarpeet on tunnistettu sekä ohjelmaan on toteutettu muutostarpeen päättelysäännöstö ja toimintalogiikan vaihtomekanismit.

Lehmanin suositukset

Lehman muotoili 1990-luvun puolivälissä nk. FEAST (*feedback, evolution, and software technology*) hypoteesin:

> *Ohjelmistoprosessi tulee edelleenkin olemaan monitasoinen (multilevel), monisilmukkainen (multiloop) takaisinkytkentäjärjestelmä (feedback system) ja sitä on käsiteltävä sellaisena, jos haluamme saada huomattavia parannuksia ohjelmistoprosessin suunnitteluun, ohjaukseen ja tehostamiseen.*

Tässä yhteydessä Lehman korostaa, että hän käyttää termiä prosessi kuvaamaan kaikkia niitä toimintoja, jotka vaikuttavat tulokseen. Lehmanin prosessi käsittää myös hallinto-, markkinointi- ja myyntihenkilöstön sekä käyttäjien tukihenkilöstön ja käyttäjät. Yleensä ohjelmistotekniikassa ohjelmistoprosessi rajataan käsittämään ohjelmiston kehitykseen ja ylläpitoon liittyvät toiminnot.

Lehman päättää artikkelinsa pitkään listaan toimintatapoja, joilla ohjelmistojen kehitettävyyttä voidaan parantaa.

- Kun tietokonejärjestelmä otetaan käyttöön tai sen käyttöä laajennetaan, vaikutuksia on tarkasteltava organisaation koko toimintaan. Ei riitä, että tarkastellaan vain sovellutuksen tehokkuutta tai taloudellisia hyötyjä.
- Sovellutuksen ja käyttöalueen rajat on identifioitava heti alussa. Nämä päätökset on kirjattava siten, että myös riippuvuudet ja keskinäiset suhteet tulevat selvästi esille.

[4] `http://en.wikipedia.org/wiki/Ariane_5_Flight_501`

- On tunnistettava ja tunnustettava, että rajat muuttuvat ajan ja käyttökokemusten myötä. Muutettavuus on ohjelmistoarkkitehtuurien, suunnittelujen ja toteutusten välttämätön ominaisuus.
- Ohjelmistokehityksen edetessä rajoja on päivitettävä vastaaman ymmärryksen väistämätöntä lisääntymistä, tarkentuvan suunnittelun ja toteutuksen vaikutuksia sekä ennakoitua vaikutusta käyttäjiin ja käyttäjien vaikutusta järjestelmään.
- Rajojen määrityksiä on katselmoitava säännöllisesti sekä ohjelmistokehityksen aikana että sen jälkeen, jotta niiden paikkansapitävyys voidaan varmistaa olosuhteiden muuttuessa.
- Määrittelyn, suunnittelun ja kehitystyön kaikissa vaiheissa on pyrittävä tunnistamaan ja kirjaamaan sekä eksplisiittiset että implisiittiset oletukset, jotka tehdään suunnittelun ja toteutuksen valinnoissa. Nämä oletukset eivät liity vain teknisiin ja hallinnollisiin seikkoihin, vaan myös käyttäjien reaktioihin, ohjelmien vaikutuksiinn käyttöalueella sekä taloudellisiin ja sosiaalisiin tekijöihin.
- Oletukset on kirjattava—mieluiten tietokoneella käsiteltävässä muodossa—systemaattisesti, jotta oletuksien paikkansapitävyyttä voidaan helposti tutkia.
- Sekä rajojen että oletusten dokumentaation tulisi sisältää arviot sovellutuksen ja käyttöalueen tulevien muutosten todennäköisyyksistä, muutosten lähteistä sekä ohjata ja helpottaa katselmointiprosessia.
- Oletukset on säännöllisesti katselmoitava sekä toteutuksen että järjestelmän käytön aikana, jotta niiden paikkansapitävyys voidaan varmistaa.
- Aivan samoin kuin tehtyjä oletuksia ja ohjelmakoodia on päivitettävä, niin myös sovellutuksen mallit, käyttöalue ja ratkaisun määritys on päivitettävä.
- Aina kun rajat muuttuvat, oletukset on katselmoitava.
- Kun erillisten sovellutuksien tai järjestelmän osien yhteistoimintaa suunnitellaan—erityisesti kun harkitaan käytetäänkö löyhää vai tiukkaa kytköstä (*loose coupling vs. tight coupling*)—sovellutukset tai järjestelmän osat on käsiteltävä yhtenä kokonaisuutena tarkasteltaessa rajoja ja oletuksia.
- Ohjelmistoon tehtävien muutosten vaikutukset rajoihin ja oletuksiin on käytävä läpi ennen muutosten tekemistä, jotta yhteensopimattomuudet tai muut ei toivotut sivuvaikutukset voidaan välttää.
- Esitettyjen muutosten kaikki vaikutukset, niin globaalit kuin paikalliset, on analysoitava. Tämä edellyttää muun muassa sitä, että kaik-

ki muutoksiin liittyvät takaisinkytkentäketjut tunnetaan ja ymmärretään.
- Kun ohjauksen hallinta otetaan käyttöön tai sitä muutetaan, muutosten globaalien vaikutusten arvioinnissa on otettava huomioon takaisinkytkentämekanismit ja näiden globaalit vaikutukset.
- Ohjelmistoarkkitehtuurien on minimoitava osien (modulit, komponentit, alijärjestelmät) väliset riippuvuudet.
- Aina kun mahdollista, järjestelmän jokaisen osan tulisi palvella suoraan sen käyttäjää eikä tuottaa tietoa muiden osien käsiteltäväksi. Tavoitteena tulisi olla ohjelmiston rakentaminen osista, jotka ovat toiminnallisesti riippumattomia toisistaan.

Loppukaneettina Lehman toteaa, että jotkut hänen suosituksistaan ovat heti sovellettavissa, mutta toiset edellyttävät tutkimusta tai kehittelyä. Joka tapauksessa paljon työtä on vielä edessä, jotta esitetyt toimintatavat saadaan otettua käyttöön systemaattisesti, taloudellisesti ja ennen kaikkea luotettavasti. Kirjoituksessaan Lehman viittaa usein Ed Yourdonin esseeseen [Yourdon, 1998]. Se on erittäin suositeltavaa luettavaa kaikille tietojenkäsittelytieteen opiskelijoille.

6.5 Tietoturva

Otsikon tietoturva (*security*) alle Denning on ryhmitellyt pääsynvalvonnan, salassapidon, yksityisyyden, tunnistamisen, eheyden ja turvallisuuden. Näistä olen valinnut tarinan aiheeksi turvallisuuden.

Therac-25 mainittiin jo luvussa 5.3 traagisena esimerkkinä ohjelmistosta, jossa kilpatilanteen jääminen ehdotonta turvallisuutta vaatineeseen (*safety-critical*) järjestelmään aiheutti tuhoisat seuraukset. Tarina perustuu Nancy G. Levesonin ja Clark S. Turnerin heinäkuussa 1993 IEEE Computer -lehdessä julkaistuun artikkeliin [Leveson ja Turner, 1993], jossa analysoidaan Therac-25 onnettomuuksia. Artikkelissa on perusteellinen kuvaus kuudesta kyseisen laitteiston aiheuttamasta vakavasta onnettomuudesta. Tarina keskittyy Levesonin ja Turnerin johtopäätöksiin.

Leveson ja Turner aloittavat opetusten yhteenvedon toteamalla, että onnettomuudet ovat harvoin yksinkertaisia. Useimmiten onnettomuuden takana on monimutkainen vyyhti toisiinsa liittyviä tapahtumia, jotka johtuvat teknisistä, inhimillisistä ja työyhteisöön liittyvistä tekijöistä. Therac-25:n tapauksessa yksi vakavista virheistä oli uskomus, että onnettomuuden syyt oli poistettu yhden löydeteyn virheen korjaamisella,

vaikka tälle johtopäätökselle ei ollut kestäviä perusteita. Erityisesti vaihtoehtoisia onnettomuuden syitä ei selvitetty kuin ylimalkaisesti. Toinen vakava virhe oli olettaa, että yhden ohjelmistovirheen korjaaminen estäisi uudet onnettomuudet. Monimutkaisissa järjestelmissä on lähes aina seuraavaksi löytyvä ohjelmistovirhe.

Onnettomuuksien katsotaan usein joutuneen yhdestä syystä, esimerkiksi inhimillisestä erehdyksestä. Toisaalta lähes kaikkien syiden voidaan sanoa olevan inhimillisiä erehdyksiä. Jopa laitteiston kulumisesta johtuvien vikojen voidaan väittää olevan inhimillisiä erehdyksiä, koska laitteistoon ei oltu suunniteltu riittävää redundanssia tai koska järjestelmän ylläpitäjät eivät olleet kunnollisesti huoltaneet laitteistoa tai vaihtaneet kuluneita osia ajoissa. Siksi onnettomuuden syynä inhimillinen erehdys ei ole kovin hyödyllinen ellei sitä riittävästi tarkenneta.

Lähes yhtä hyödytöntä on käyttää tietokoneen laitteistovikaa tai ohjelmistovirhettä onnettomuuden syynä. Tämä ei kuitenkaan tarkoita, etteivät ohjelmistovirheet olisi vaikuttaneet Therac-25:n aiheuttamiin onnettomuuksiin. Ne vaikuttivat, mutta eivät olleet ainoa syy.

Jos johtopäätöksemme olisi, että Therac-25 onnettomuudet johtuivat vain ohjelmistovirheistä, niin meidän pitäisi tulla johtopäätökseen, että ainoa tapa estää tällaiset onnettomuudet olisi toteuttaa täydellisiä ohjelmistoja. Toisin sanoen ohjelmistomme ei saisi koskaan käyttäytyä odottamattomasti tai epätoivotusti missään olosuhteissa. Tällaista vaatimusta on mahdoton täyttää. Toinen vaihtoehto olisi, että ohjelmistoa ei koskaan käytettäisi ehdottomaa turvallisuutta vaativissa järjestelmissä. Kumpikin johtopäätöstä on ylipessimistinen.

Monimutkaisten järjestelmien aiheuttamien onnettomuuksien analysoinnin on perustuttava järjestelmien rakentamisen (*system-engineering*) lähestymistapaan. Erityisesti kaikkia mahdollisia onnettomuuteen mahdollisesti vaikuttaneita tekijöitä on tarkasteltava. Therac-25;n onnettomuuksiin vaikuttaneita tekijöitä olivat:

- valmistajan hallinnoinnissa olleet epätarkoituksenmukaisuudet ja raportoitujen tapausten käsittelytapojen puutteellisuudet,

- kohtuuton luottaminen ohjelmistoon ja laitteistovarmistusten jättäminen pois, minkä seurauksena ohjelmistosta tuli varmistamaton virhelähde (*single point of failure*),

- oletettavasti ohjelmistotekniikan käytäntöjen puutteet ja

- epärealistinen riskien arviointi ja ylenpalttinen luottamus arvioinnin antamiin tuloksiin.

Järjestelmien rakentaminen

Hyvin yleinen virhe, niin Therac-25:n tapauksessa kuin monissa muissa, on luottaa liikaa ohjelmistoon. Vaikka ohjelmistossa ei esiinny laitteistojen tavoin satunnaisia kulumisvirheitä, ohjelmiston suunnitteluvirheiden löytäminen ja estäminen on huomattavasti hankalampaa kuin kulumisvirheiden. Lisäksi laitteiston virhekäyttäytymisen muotoja on yleensä vain muutama. Siksi niitä vastaan suojautuminen on useimmiten olennaisesti helpompaa kuin ohjelmistovirheitä vastaan suojautuminen. Tapauksen Therac-25 ehkä tärkein opetus oli, että laitteistovarmistuksia ei ole syytä poistaa, kun ohjelmistoa ruvetaan käyttämään järjestelmän ohjaamiseen.

Viimeaikainen trendi on ollut vähentää laitteistovarmistuksia. Niissäkin tapauksissa, joissa laitteistovarmistuksia käytetään, niitä yhä useammin kontrolloidaan ohjelmistolla. Ehdotonta turvallisuutta vaativissa järjestelmissä ei saa olla varmistamattomia virhelähteitä. Tässä mielessä ohjelmistoa on kohdeltava järjestelmän yhtenä komponenttina. Järjestelmiä ei saisi suunnitella siten, että yksittäinen ohjelmistovirhe voi aiheuttaa katastrofin.

Therac-25:n ensimmäinen turvallisuusanalyysi ei kattanut ohjelmistoa, vaikka järjestelmän turvallisuus riippui lähes yksinomaan ohjelmistosta. Kun ongelmia alkoi esiintyä, niiden selvittelyssä oletettiin ongelmien johtuvan laitteistosta. Ohjelmiston epäileminen mahdollisena virhelähteenä ei saisi olla viimeinen vaihtoehto.

Ohjausjärjestelmissä ohjelmistovirheen voidaan ajatella olevan tilapäinen laitteistovirhe. Ohjausohjelmisto lukee arvoja tunnistimista (*sensors*) ja lähettää komentoja säätimille (*actuators*). Usein virhetilanteiden selvittelyssä on hyvin työlästä, ellei mahdotonta, päätellä, antoiko tunnistin väärää tietoa, lähettikö ohjelmisto väärän komennon, vai toimiko säädin tilapäisen laitevirheen vuoksi väärin.

Therac-25:n tapauksessa potilaiden oireet olivat ainoat todelliset indikaattorit järjestelmän vakavista ongelmista. Järjestelmässä ei ollut riippumattomia tarkistuksia ohjelmiston toiminnan oikeellisuudesta. Therac-25 ei pystynyt havaitsemaan antamaansa säteilyn yliannostusta. Keskeinen opetus on, että ohjausjärjestelmät on suunniteltava pahimman tilanteen varalle. Ehdotonta turvallisuutta vaativiin järjestelmiin on rakennet-

tava jäljitysmekanismit (*audit trails*) sekä poikkeavien tilanteiden analysointi. Näitä on käytettävä aina, kun järjestelmä havaitsee jotain, mikä saattaisi johtaa onnettomuuteen. Toisin sanoen järjestelmällä on oltava malli omasta oikeasta toiminnasta.

Turvallisuusanalyysin antamiin numeerisiin tuloksiin luotettiin Therac-25:n tapauksessa aivan liikaa. Väite, että mikrokytkimen lisääminen olisi lisännyt järjestelmän turvallisuutta viisi kertaluokkaa eli virhetoiminnan todennäköisyys olisi pienentynyt sadastuhannesosaan alkuperäisestä, oli täysin perusteeton. Turvallisuusanalyysien yksi keskeinen ongelma on, että niissä usein jätetään ottamatta huomioon ongelman osia, joita on vaikea kvantifioida. Usein tällaisilla tekijöillä on suurempi vaikutus turvallisuuteen kuin tekijöillä, jotka pystytään kvantifioimaan.

Ohjelmistotekniikka

Therac-25:n onnettomuudet olivat siinä mielessä melko harvinaisia, että järjestelmään oli jäänyt suoranaisia ohjelmointivirheitä. Useimmiten tietokoneperäisissä onnettomuuksissa ei ole kyse ohjelmointivirheistä, vaan ohjelmiston suunnitteluvirheistä. Tavanomaisimmat suunnitteluvirheet liittyvät vaatimusmäärittelyyn. Kaikkia ulkoisia olosuhteita ei oteta huomioon tai joitakin niistä käsitellään virheellisesti. Toinen merkittävä virhelähde on joidenkin järjestelmän tilojen virheellinen käsittely.

Vaikka ohjelmistotekniikan peruskäytäntöjen oikeaoppinen soveltaminen ei estä kaikkia ohjelmistovirheitä, niin niiden noudattaminen on minimivaatimus. Therac-25:n tapauksessa ainakin seuraavia ohjelmistotekniikan peruskäytäntöjä ei noudatettu:

- Dokumentointi ei saa olla jälkijättöistä.
- Ohjelmiston laadunvalvonnan vakiintuneita käytäntöjä ja standardeja on noudatettava.
- Suunnitelmat on pidettävä mahdollisimman selkeinä ja yksinkertaisina.
- Virhetilanteista tiedonkeruun suunnittelu on alusta alkaen oltava osa järjestelmän suunnittelua.
- Ohjelmiston testauksen on katettava niin yksittäiset modulit kuin koko ohjelmisto. Pelkkä järjestelmätestaus ei riitä.

Ehdotonta turvallisuutta vaativissa ohjelmistoissa turvallisuus on rakennettava ohjelmistoon. Lisäksi tällaisten ohjelmistojen turvallisuus on analysoitava. Turvallisuus on pystyttävä takaamaan järjestelmätasolla mahdollisista ohjelmistovirheistä huolimatta. Therac-25:n edeltäjässä oli sama ohjelmistovirhe, mutta laitteistovarmistus esti onnettomuudet. Ohjelmistovirheitä vastaan voidaan suojautua myös itse ohjelmistossa.

Therac-25 antoi opetuksen myös ohjelmien uudelleen käyttämisestä (*software reuse*). Usein naivisti oletetaan, että ohjelman uudelleen käyttäminen tai kaupallisen valmisohjelman (*off-the-shelf software*) käyttäminen lisää turvallisuutta, koska ohjelma on jo pitkään ollut käytössä. Ohjelmamodulin uudelleen käyttäminen ei takaa uuden järjestelmän turvallisuutta. Uudelleen käyttäminen saattaa jopa kasvattaa suunnitelman monimutkaisuutta, mistä usein seuraa turvallisuusriskien lisääntyminen. Viime kädessä turvallisuus on järjestelmän, jossa ohjelmistoa käytetään, ominaisuus, ei varsinaisesti ohjelmiston ominaisuus.

Therac-25:n käyttöliittymästä puhuttiin hyvin runsaasti. Kuitenkaan sillä ei ollut olennaista vaikutusta itse onnettomuuksiin. Järjestelmän käyttäjät vaativat aikoinaan muutoksia käyttöliittymään. Tehdyt käytettävyyden parannukset tuli kuitenkin tehtyä vahingossa turvallisuuden kustannuksella.

Taustamateriaalia

Candea, G. & Fox, A. (2003) Crash-Only Software. Proceesings of the 9^{th} Workshop on Hot Topics in Operating Systems, USENIX Association, toukokuu 2003, sivut 67–72. `http://www.usenix.org/events/hotos03/tech/full_papers/candea/candea.pdf`.

Lehman, M. M. (1998) Software's Future: Managing Evolution. IEEE Software, 15, 1, tammi-helmikku 1998, sivut 40–44.

Leveson, N. G. & Turner, C. S. (1993) An Investigation of the Therac-25 Accidents. IEEE Computer, 26, 7, heinäkuu 1993, sivut 18–41.

Menascé, D. A. (2002) Software, Performance, or Engineering? Proceedings of ACM Workshop on Software and Performance (WOSP'02), heinäkuu 2002, sivut 239–242.

Parnas, D. L. (1996) Why Software Jewels Are Rare. IEEE Computer, 29, 2, helmikuu 1996, sivut 57–60.

Lähteitä

Chou, A., Yanf, J.-F., Chelf, B., Hallem, S. & Engler, D. (2001) An Empirical Study of Operationg Systems Errors. Proceedings of 18^{th} ACM Symposium on Operating Systems Principles, sivut 73–88.

Dijkstra, E. W. (1968) The Structure of the THE Multiprogramming System. Communications of the ACM, 11, 5, toukokuu 1968, sivut 341–346.

Dijkstra, E. W. (1972) Notes on Structured Programming. Teoksessa Structured Programming; O.-J. Dahl, E. W. Dijkstra ja C. A. R. Hoare (toim.). New York: Academic Press.

Glass, R. L. (2001) Frequently Forgotten Fundamental Facts about Software Engineering. IEEE Software, 18, 3, touko/kesäkuu 2001, sivut 110–112.

Nagaraja, K., Bianchini, R., Martin, R. P. & Nguyen, T. D. (2002) Using Fault Model Enforcement to Improve Availability. Proceeding of 2^{nd} Workshop on Evaluating and Architecting System Dependability.

Parnas, D. L. (1994) Software Aging. Proceedings of the 16^{th} International Cpnference on Software Engineering (ICSE-16), sivut 279–287.

WikipediA (2005) Ariane 5 Flight 501. `http://en.wikipedia.org/wiki/Ariane_5_Flight_501`.

Wirth, N. (1995) A Plea for Lean Software. IEEE Computer, 28, 2, helmikuu 1995, sivut 64–68.

Yourdon, E. (1998) A Tale of Two Futures. IEEE Software, 15, 1, tammihelmikuu 1998, sivut 23–29.

LUKU 7

Tietojenkäsittelyn käytännöt

Tietojenkäsittelyn käytännöt täydentävät kuvaa tietojenkäsittelytieteestä. Tietojenkäsittelyn käytännöissä ilmentyvät tietojenkäsittelijöiden ammatilliset taidot. Viime kädessä tietojenkäsittelijän taidot ja osaaminen arvioidaan työn tulosten laadun perusteella.

Tietojenkäsittelyn käytännöt voidaan jakaa viiteen pääalueeseen:

Ohjelmointi: (engl. *programming*) Järjestelmän käyttäjien kanssa määritellyn ohjelmiston toteuttaminen ohjelmointikieliä käyttäen.

Tietojenkäsittelyn ammattilaisten on hallittava useita erilaisia ohjelmointikieliä ja osattava valita tarkoituksenmukaisin kuhunkin ongelmanratkaisutilanteeseen.

Järjestelmien rakentaminen: (engl. *engineering systems*) Tietoverkossa toimivien hajautettujen järjestelmien suunnitteleminen ja toteuttaminen ohjelmisto- ja laitteistokomponenteista. Tämän alueen käytäntöihin kuuluuvat[1] **suunnittelu**, jossa keskitytään järjestelmän organisointiin siten, että järjestelmä hyödyttää käyttäjiä, **toteuttaminen**, jossa keskitytään järjestelmän moduleihin, abstraktioihin, uudistuksiin (engl. *revisions*), toteutustavan valintoihin (engl. *design decisions*) ja riskeihin, sekä **käyttö**, jossa keskitytään konfiguraatioon (engl. *configuration*), hallintaan (engl. *management*) ja ylläpitoon (engl. *maintenance*).

[1]Olen kääntänyt Denningin käyttämät termit *design component, engineering component,* ja *operations component* termeillä suunnittelu, toteuttaminen ja käyttö.

Tietojenkäsittelyn ammattilaisella on oltava taidot osallistua laajojen (tuhansia moduleja ja miljoonia ohjelmarivejä käsittävien) järjestelmien toteuttamiseen.

Mallintaminen ja validointi: (engl. *modeling and validation*) Järjestelmien mallintaminen niiden käyttäytymisen ennustamiseksi erilaisissa tilanteissa ja olosuhteissa. Kokeiden (engl. *experiment*) suunnittelu algoritmien ja järjestelmien validoimiseksi.

Innovointi: (engl. *innovating*) Johtajuuden (engl. *leadership*) käyttäminen pysyvien muutosten aikaansaamiseksi ryhmien ja yhteisöjen toimintatavoissa.

Innovaattorit etsivät ja analysoivat mahdollisuuksia. He kuuntelevat asiakkaitaan ja muotoilevat heille hyödyllisiä ehdotuksia. *Innovaattoreiden on myös huolehdittava, että luvatut tulokset saavutetaan.*

Soveltaminen: (engl. *applying*) Työskentely sovellusalueiden ammattilaisten kanssa näitä palvelevien tietojenkäsittelyjärjestelmien toteuttamiseksi. Työskentely muiden tietojenkäsittelyn ammattilaisten kanssa useita erilaisia sovelluksia tukevien ydinteknologioiden kehittämiseksi.

7.1 Ohjelmointi

Mitä tahansa ohjelmointikielitä käytätkin,
niin tehtäväsi ohjelmoijana on käyttää kyseistä kieltä
ja ohjelmointivälinettä niin hyvin kuin osaat.
Hyvä ohjelmoija pystyy ylittämään
huonon ohjelmointikielen tai
kömpelön käyttöjärjestelmän asettamat esteet,
mutta hyväkään ohjelmointiympäristö
ei pelasta huonoa ohjelmoijaa.
–Kernighan ja Pike,
Unix-järjestelmän ja C-ohjelmointikielen kehittäjät.

Kautta aikojen ohjelmoijat ovat väitelleet eri ohjelmointikielten eduista. Jokaisella ohjelmoijalla on oma suosikkinsa. Ohjelmoinnin tarinan

perustuu Donn Seeleyn artikkeliin *How Not to Write Fortran in Any Language* (ACM Queue, joulukuu 2004/tammikuu 2005, sivut 58–65). Seeleyn mukaan maailmassa on nähty niin monia huonoja Fortran-ohjelmia, että Fortranista on tullut huonon ohjelmakoodin synonyymi.

Syyt lienevät historiallisia. Fortran kehitettiin 1950-luvulla numeerisen laskennan tarpeisiin. Useimmat Fortran-ohjelmat eivät olleet tietojenkäsittelyn ammattilaisten, vaan fyysikoiden ja muiden luonnontieteilijöiden kirjoittamia. Seeley toteaakin, että Fortranilla voidaan kirjoittaa hyödyllisiä ja ylläpidettäviä ohjelmia huolimatta kielen monista rajoitteista. Itse asiassa useimmat huonot Fortran-ohjelmat johtuvat ammattitaidottomista ohjelmoijista, jotka eivät olisi saaneet aikaan kelvollista ohjelmakoodia millään ohjelmointikielellä. Vielä tänäkin päivänä useimmissa ohjelmistoja tuottavissa yrityksissä melkoinen osa henkilöstöstä ei ole saanut kuin korkeintaan tietojenkäsittelyn alkeet kattavan koulutuksen.

Hyvän ohjelmakoodin ominaisuudet ovat jokseenkin ohjelmointikielestä riippumattomia. Hyvin suunniteltu algoritmi voidaan toteuttaa lähes millä tahansa ohjelmointikielellä siten, että ohjelmakoodi on selkeärakenteinen ja helposti ymmärrettävä. Mikään käyttökelpoinen ohjelmointikieli ei pysty estämään huonon ohjelmakoodin kirjoittamista.

Aikoinaan Edsger Dijkstra [Dijkstra, 1968] väitti goto-lauseen olevan haitallinen[2]. Tätä dogmia vastaan esitettiin lukuisia vasta-agrumetteja. Perusteellisin vasta-argumenttien kokoelma lienee Donald E. Knothin artikkeli *Structured Programming with go to Staments* [Knuth, 1974]. Myös Steeley toteaa, että täysin käsittämätön ja ylläpitämätön ohjelma voidaan kirjoittaa ilman ainuttakaan hyppykäskyä ja ohjelman rivinumeroa käyttäen vain poikkeusten käsittelyä, abstrakteja tietotyyppejä ja jätteiden keruuta (*garbage collection*). Riippumatta käyttämästään ohjelmointikielestä ohjelmoija voi kirjoittaa hyvää koodia (kuten pitäisi) tai huonoa koodia.

Käytetyn ohjelmointikielen vaikutuksia on yliarvioitu

Käytetyllä ohjelmointikielellä on vaikutuksensa ohjelmakoodin laatuun. Joillakin kielilillä on helpompi tai vaikeampi kirjoittaa käsittämätöntä koodia. Yleensä kielen vaikutusta on kuitenkin yliarvioitu. Itse opin

[2]Tämä Dijkstran artikkeli on yksi tietojenkäsittelytieteen virstanpylväitä. Hyvin monet kirjoittajat ovat imitoineet sen otsikkoa muodossa *'XYZ' Considered Harmful.*

tämän aikoinani Helsingin yliopiston tilastotieteen laitoksella. Professori Seppo Mustosen johdolla tehty tilastollisten aineistojen analysointijärjestelmä Survo oli kirjoitettu vanhalla tulkintaan perustuvalla Wang-järjestelmän Basic-kielellä. Viikossa opin lukemaan ryhmän tuottamaa koodia, koska hyppykäskyjä ja muita kyseisen kielen antiikkisia (Fortrania huomattavasti kamalampia) piirteitä oli käytetty systemaattisesti. Jossakin mielessä kaipaan vanhoja ”hyviä” aikoja. Kun ohjelmointikielet olivat kamalia, tiimin piti sopia yhteisistä käyttötavoista, jotta yhteistuotos olisi jollakin tavoin hallittava.

Nykyiset kehittyneet ohjelmointiympäristöt sallivat ajattelemattomat kokeilut. Vanhat kamalat järjestelmät vaativat ohjelmoijia ajattelemaan sitä, mitä he toteuttavat. En kaipaa vanhaa reikäkorttiaikaa ja neljää yritystä vuorokaudessa kääntää ja suorittaa ohjelmani eräajona, mutta nykyiset kehitysympäristöt eivät näytä pakottavan kehittäjiä ajattelemaan vaan yrittämään ensimmäistä mieleentulevaa vaihtoehtoa.

Seeley toteaa, että nykyisin samankaltaisia ohjelmointiongelmia kohdataan niin C:tä, Perl:iä, Scheme:ä, Smalltalk:ia tai vastaavia käytettäessä. Siksi ohjelmointikielestä riippumattomat ohjelmointiratkaisut ovat keskeisiä. Taustalla on se tosiasia, että ongelmat eivät johdu ohjelman toteutuksesta jollakin ohjelmointikielellä vaan ohjelman suunnittelusta. Hyvä suunnittelu voidaan toteuttaa kelvollisesti millä tahansa ohjelmointikielellä. Huono suunnittelu johtaa lähes aina huonoon toteutukseen ohjelmointikielestä riippumatta. Tosin jotkut ratkaisut on helpompi toteuttaa siedettävästi jollakin tietyllä ohjelmointikielellä. Useimpien ohjelmointikielten perusprimitiivit ovat kuitenkin niin samankaltaisia, että ohjelmointikielen valinta ei olennaisesti vaikuta suunnitelman toteutukseen.

Usein väitetään, että primitiivinen C-ohjelmointikieli ei tue olio-ohjelmointia. C ei tue perintää (*inheritance*), pakkauksia (*package*) eikä paikallisia jäseniä (*private members*). Toisaalta useat C:n tietorakenteet ja kirjastot ovat hyvin olio-ohjelmoinnin kaltaisia. Koiranleuat ovatkin väittäneet, että C:llä saat tehtyä kaiken minkä oliokielilläkin, mutta et pysty vahingossa tekemään aikaavieviä tyhmyyksiä.

Seeleyn mukaan hyvät ohjelmointikäytännöt ovat ohjelmointikielestä riippumattomia, koska ohjelmat kirjoitetaan ongelmien ratkaisujen suunnitelmiksi ja näiden suunnitelmien välittämiseksi muille ihmisille. Siksi hyvä ohjelmakoodi sisältää hyviä suunnitteluratkaisuja. Koska useimmat ongelmat eivät ole ainutkertaisia, hyviä suunnitteluratkaisuja voidaan uusiokäyttää. Ohjelmoinnissa hyvin suunniteltu on enemmän kuin puoliksi tehty.

Ihmisille muoto ja tyyli ovat erittäin tärkeitä sisällön välittymisessä. Hyvin muotoiltu ja sujuvasti kerrottu tarina on helposti omaksuttavissa. Hyvä muoto on huomaamaton: Lukija näkee sisällön eikä muotoa tai tyyliä. Hyvä muoto on myös elegantti. Se ei sotke sivua tai ajatustasi. Ohjelmakoodisi lukijat arvostaisivat, että sitä olisi helppo lukea eikä se päästäisi ajatusta karkaamaan.

Hyvän ohjelmakoodin ominaisuuksia

Hyvä ohjelmoija on täysin tietoinen oman päänuppinsa rajoitteista.
Siksi hän asennoituu ohjelmointitehtävään nöyrällä mielellä
ja karttaa ohjelmointitrikkejä kuin ruttoa.
–Edsger Dijkstra

Artikkelissaan Seeley toteaa, että 30 vuoden kokemus ohjelmien ylläpitämisestä on muovannut hänen käsitystään hyvästä ohjelmakoodista. Ohjelmakoodia pitäisi olla yhtä helppo lukea kuin sanomalehteä. Ohjelmakoodin on tuotava esille taustalla oleva suunnittelu niin ilmeisenä ja helposti tajuttavana, että hän pystyy kertomaan välittömästi äidinkielellään kyseisen ohjelman toimintalogiikan.

Artikkelissaan Seeley painottaa, että hyviä suunnitteluratkaisuja on runsaasti olemassa. Siksi hän keskittyy artikkelissaan ohjelmoinnin muoto- ja tyyliseikkoihin.

Hyvä ohjelmakoodi on helposti luettavaa. Vaikka kääntäjät eivät välitä välilyönneistä, niin ohjelmaa myöhemmin ylläpitävät ihmiset joutuvat lukemaan sitä. Seeley tarjoaa kolme esimerkkiä:

```
                    This

                                        =

   is

+

                                  very

              *

annoying
```

```
                                ;

This=is+almost(as-annoying);

This = is + much * (more - readable);
```

Sopivasti ryhmittelemällä saat ohjelmakoodistasi luettavampaa. Tyhjät rivit, sisennykset ja välilyönnit ovat kääntäjälle merkityksettömiä, mutta ohjelmaasi ylläpitävälle erittäin keskeisiä. Erityisesti tyhjä rivi niputtaa yhteenkuuluvat lauseet kokonaisuudeksi. Ne toimivat ohjelmakoodin lukijalle ohjelmasi kappalejakona. Romaani, jossa ei olisi lukujen sisällä kappalejakoa, jäisi minulta todennäköisesti kesken. En olisi valmis uhraamaan tolkuttomasti aikaa ymmärtämisen vaatiman jäsentelyn muodostamiseksi. Ohjelmien kirjoittamisessa tämä aspekti usein unohdetaan. Jos ylläpitäjä joutuu muodostamaan uudelleen ohjelman alkuperäisen kirjoittajan valitseman ohjelman rakenteen, on se hyvin vaivalloista ja ajan haaskausta.

Tietokoneet eivät välitä siitä, kuinka pitkiä yhteenkootut lauseiden ryhmät ovat. Ihmiset ovat rajoittuneita. Seeley mainitsee rajan seitsemän, joka perustunee tutkimuksiin. Esimerkkinä toiminee hyvin puhelinnumerot. Puhelinnumeron muistaminen perustunee ryhmittelyyn. Muutaman numeron ryhmittelyllä saadaan osakokonaisuuksia, jotka helpottavat muistamista.

Tietokoneet eivät välitä kuinka isoja ryhmät ovat. Toisaalta ihmisen kyky hahmottaa ja ymmärtää isoja ryhmiä on rajallinen. Jos ohjelmarivi on täynnä operaatioita, niin sen lukeminen ja ymmärtäminen on hidasta. Toisinaan käytettävissä oleva rajapinta asettaa omat rajoitteensa. Seeleyn antama esimerkki on

```
Can(you, tell, at + a, glance, which * of, these,
parameters, is(the, eight), one ? yeah : sure);
```

Tyhjien rivien lisäksi sisennyksiä voidaan käyttää ryhmittelyyn. Nykyisten ohjelmointikielten tyylioppaat painottavat voimakkaasti sisennyksen merkitystä. Lisäksi sisennyksen väärinkäyttöä esiintyy nykyisin melko vähän.

Luonnollinen ryhmittely on luettavuuden kannalta niin tärkeää, että toisinaan olisi hyödyllistä muuttaa ohjelman rakenne käyttämään ryhmiteltyjä osia. Joskus hyvin syvälle upotetut ohjausrakenteet voivat rikkoa ryhmittelyn. Tällaisissa tilanteissa vaihtoehtoisten ohjausrakenteiden tai aliohjelmien käyttäminen parantaisi ohjelman luettavuutta.

Kommentointi

Nykyisissä ohjelmointikielissä kommentit eivät vaikuta mitenkään ohjelman käyttäytymiseen. Kääntäjät ohittavat kommentit, mutta ohjelmakoodia lukevat ihmiset eivät ohita niitä. Jokaisella ohjelmointia opettavalla on omat suosikkinsa huonoista kommenteista. Seeleyn suosikki on:

```
/* add one to i */
i = i + 2;
```

Seeley puhuu voimakkaasti sen puolesta, että kommentit olisivat kokonaisia lauseita eikä vain sähkösanomakieltä tai pseudokoodia. Jotkut pitävät tätä työläänä, mutta Seeley väittää, että kun olet ruvennut kirjoittamaan hyvää ohjelmakoodia ja välttämään huonoja kommentteja, niin huomaat, että muutama hyvä kommentti on huomattavasti hyödyllisempi kuin lukuisat huonot kommentit.

Ihmiset arvostavat luonnollisen kielen kokonaisia lauseita kommenteissa, koska heidän ei tarvitse painiskella vielä kommentointiin käytetyn kielen kanssa. Ohjelmakoodin lukeminen on jo riittävän hankalaa, siksi lukijan työtaakan lisääminen kommenttien ymmärtämiseksi ei ole perusteltua.

Hyvässä ohjelmakoodissa käytetään *pienimmän hämmennyksen periaatetta*. Lukijat eivät usein huomaa ohjelmointitrikkejä, jos niistä ei ole varoitettu. Hyvän ohjelmakoodin tulisi välttää kaikenlaisia ohjelmointitrikkejä niin paljon kuin mahdollista. Jos niitä ei voida välttää, ne tulisi merkitä selvästi. Trikin selittävät kommentit on erinomainen käytäntö.

Nimet

Hyvät nimet hyvän ohjelmakoodin välttämätön edellytys. Kuten kommentit, nimet eivät ole tietokoneelle kuin merkkijonoja. Sen sijaan ihmisille ne ovat olennaisia. Viisas nimien valinta on luettavan ohjelmakoodin perusedellytys.

Tuttuus vähentää älyllistä kuormitusta. Siksi tutut nimet tutussa yhteydessä on helppo ymmärtää. Ohjelmoijat ovat jo ammoisista ajoista käyttäneet nimeä `i` indeksimuuttujana. Siksi `i`:n käyttäminen muussa merkityksessä on harhaanjohtavaa.

Nimiä tulee käyttää systemaattisesti eli sama nimi aina samassa merkityksessä. Tuttujen ja kuvaavien nimien käyttäminen helpottaa ohjelmakoodin lukemista. Viisas nimien valinta lienee tärkein ohjelmakoodin

luettavuuteen vaikuttava tekijä. Hyvin pitkät nimet vähentävät osaltaan ohjelmakoodin luettavuutta. Toisaalta myös lyhenteiden runsas käyttö vähentää luettavuutta.

Johdonmukaisuus

Huonon ohjelmakoodin keskeinen tunnusmerkki on epäjohdonmukaisuus. Jos kolmekymmentä ohjelmoijaa työskentelee saman lähdekoodin kanssa, niin heidän on noudatettava yhteistä ohjelmointityyliä ja nimentätapoja. Muussa tapauksessa epäjohdonmukaisuus tekee ohjelmakoodista huonon. Ohjelmoijien on oltava nöyriä ja hyväksyttävä ohjelmakoodin luettavuus niin keskeiseksi tavoitteeksi, että jokaisen on luovuttava omasta suosikkityylistään ja käytettävä yhteistä ohjelmointityyliä.

Yhteenvetona Seeley toteaa, että ohjelmointikielestä riippumatta hyvä ohjelmakoodi:

- välttää epäjärjestystä,
- käyttää ryhmittelyä,
- käyttää hyväksi tuttuutta,
- ei aiheuta yllätyksiä ja
- on johdonmukainen.

Hyvän ohjelmakoodin tarkkoja vaatimuksia on hankala määritellä täsmällisesti. Kuitenkin useimmat meistä tunnistavat nähdessään huonon ohjelmakoodin.

> *Hyvän ohjelmointityylin opettaminen on käytännössä mahdotonta opiskelijoille, jotka ovat käyttäneet Basic:iä. Potentiaalisina ohjelmoijina he ovat älyllisesti parantumattomasti vaurioituneita. –Edsger Dijkstra*

Artikkelinsa Seeley päättää toteamukseen, että huonot ohjelmat eivät tunnu loppuvan. Yleisimmät selitykset ovat:

- Ohjelma oli kirjoitettava hätäisesti.
- Ohjelma oli tekijänsä ensimmäinen iso ohjelmointiprojekti.
- Ohjelma oli tarkoitettu vain prototyypiksi.
- Se alkoi henkilökohtaisena projektina.

Seeleyn mukaan hyvän ohjelmakoodin kirjoittaminen ei ole kovinkaan paljon vaativampaa kuin huonon. Hyvän ohjelmakoodin hyödyt tulevat selkeästi esille ohjelman ylläpidon aikana. Ei ole mitään järkeä kirjoittaa muuta kuin hyvää ohjelmakoodia.

7.2 Järjestelmien rakentaminen

Järjestelmien rakentaminen on ohjelmistotekniikan keskeisin haaste. Kuten tämän luvun johdanto-osassa todettiin, niin *tietojenkäsittelyn ammattilaisella on oltava taidot osallistua laajojen järjestelmien toteuttamiseen.* Denningin mukaan järjestelmien rakentamiseen kuuluvat suunnittelun, toteutuksen ja käytön komponentit.

Tarina perustuu Richard E. Fairleyn ja Mary Jane Willshiren IEEE Software -lehdessä maalis-huhtikuussa 2003 julkaistun artikkelin *Why the Vasa Sank: 10 Problems and Some Antidotes for Software Projects.* Varteenotettavina vaihtoehtoina harkitsin Frederick P. Brooks Jr:n artikkelia *No Silver Bullet — Essence and Accidents of Software Engineering* [Brooks, 1987] ja sen päivitettyä versiota *"No Silver Bullet" Refired* [Brooks, 1995]. Näiden ongelmana on, että ne eivät ole julkaistussa muodossa saatavilla verkosta.

Vasa laivan tarina

Elokuun kymmenes päivä 1628 Ruotsin kuninkaallisen laivaston uusin laiva, Vasa, lähti neitsytmatkalleen Tukholman satamasta. Hieman yli kilometrin purjehduksen jälkeen pieni tuulenpuuska kaatoi ja upotti laivan. Vuonna 1961 eli 333 vuotta myöhemmin alus nostettiin. Suojainen uppoamispaikka ja Itämeren alhainen suolapitoisuus olivat säilyttäneet puisen aluksen erittäin hyvässä kunnossa. Nykyisin alukseen ja sen tarinaan pääsee tutustumaan sekä Tukholman Vasa museossa että Internetissä[3].

Artikkelissaan Fairley ja Willshire kertovat melko laajasti Vasa-laivan rakentamisen vaiheista. Minä tulen keskittymään artikkelin loppuosaan, jossa peilataan järjestelmän suunnittelun ja toteuttamisen yleisiä ongelmia Vasan tarinaan. Aluksi kuitenkin lyhyt yhteenveto Vasan rakentamisen vaiheista.

Tammikuussa 1625 Ruotsin kuningas Kustaa II Adolf määräsi amiraali Flemingin tilaamaan Henrik ja Arend Hybertssonilta neljä sota-alusta, kaksi kölimitaltaan 108-jalkaista (n. 33 metriä) ja kaksi 135-jalkaista (n. 41 metriä). Nämä alihankkivat aluksen varsinaisen rakentamisen laivanrakentaja Johan Isbrandssonilta.

Vasa-laivan rakentaminen alkoi vuoden 1626 alussa tavanomaisena

[3] `http://www.vasamuseet.se`

laivanrakennustehtävänä. Aluksen valmistuessa kaksi ja puoli vuotta myöhemmin se ei ollut enää tavanomainen vaan ensimmäinen lajissaan.

Marraskuussa Kustaa II Adolf kasvatti tilattujen pienempien laivojen kölimitaksi 120 jalkaa, jotta ne pystyisivät kantamaan kolmekymmentäkaksi 24-paunan (ammuksen paino) kanuunaa. Yksi tällainen kanuuna painoi n. 1,4 tonnia. Henrik Hybertssonilla oli kuitenkin käytettävissään raakapuuta vain kahteen alukseen, yhteen 111-jalkaiseen ja yhteen 135-jalkaiseen.

Kun 111-jalkaisen aluksen kölirakenne oli tehty, Kustaa II Adolf vaati, että alus suurennetaan 135-jalkaiseksi ja että alukseen tulee kaksi kanuunakantta. Kukaan Ruotsissa ei ollut koskaan rakentanut kahden kanuunakannen alusta. Aikataulupaineista johtuen aluksen rakentajat olettivat, että 111-jalkainen alus voidaan kasvattaa 135-jalkaiseksi kölirakennetta muuttamatta.

Vasa-laivan rakenteesta, niin 111-jalkaisesta kuin 135-jalkaisesta, ei ole mitään suunnitteludokumentaatiota, ei edes karkeita luonnoksia. Tällaiseen materiaaliin ei löydy vittauksia missään tuon ajan asiakirjoissa. Siten on erittäin todennäköistä, että missään vaiheessa Vasa-laivan rakentamisesta ei ole tehty rakentamissuunnitelmaa.

Jälkikäteen Hybertssonien lähin avustaja, Hein Jacobsson, kertoi, että Vasa-laivan leveyttä kasvatettiin noin puolella metrillä, jotta alukseen saataisiin kaksi kanuunakantta. Koska köli oli jo rakennettu, levennys voitiin tehdä vain aluksen ylärakenteisiin. Tämä kohotti aluksen painopistettä ja lisäsi Vasa-laivan kiikkeryyttä. Purjehtijat tietävät, että muutaman senttimetrin muutos painopisteessä vaikuttaa olennaisesti aluksen purjehdusominaisuuksiin. Kaiken kukkuraksi kölirakenteen kapeus ei antanut riittävästi tilaa tasapainoisuutta vakauttaville lisäpainoille.

Myös Vasa-laivan aseistukseen tehtiin muutoksia. Aluksen 111-jalkaiseen versioon oli alunperin ajateltu kolmekymmentäkaksi 23-paunan kanuunaa, joiden yhteispaino olisi ollut lähes 50 tonnia. Lopullisessa 135-jalkaisessa aluksessa aseistuksen määrä ja yhteispaino oli 50 %:a suurempi. Koska puolet kanuunoista oli ylemmällä kannella, niin aluksen painopiste nousi yhä ylemmäksi.

Kustaa II Adolf vaati alukseen satoja ornamentteja ja puuveistoksia. Vasa-laivan oli oltava myös ulkonäöltään vaikuttava. Raskaat tammipuiset veistokset nostivat osaltaan aluksen painopistettä ja lisäsivät kiikkeryyttä.

Henrik Hybertsson sairastui vakavasti vuonna 1626 ja kuoli seuraavana vuonna, vuotta ennen aluksen valmistumista. Sairastelunsa aika-

na Henrik jakoi rakentamisen johtamisen Jacobssonin ja Ibsbrandssonin kanssa. Historialliset dokumentit kuitenkin osoittavat, että rakentamisen hallinnointi oli lähes olematonta. Vastuunjako kolmen keskeisen henkilön osalta oli epäselvää, ja heidän välinen yhteydenpito lähes olematonta.

Koska aluksen rakentamisesta ei ollut minkäänlaisia suunnitelmia eikä edistymisen virstanpylväitä (*milestone*), Jacobssonin oli lähes mahdotonta ymmärtää ja toteuttaa Hybertssonin dokumentoimattomia suunnitelmia. Kolmen avainhenkilön välinen huono kommunikointi vain lisäsi ongelmia ja viivästytti aluksen valmistumista.

Henrik Hybertssonin kuoleman jälkeen amiraali Fleming nimitti Jacobssonin hankkeen vastuulliseksi johtajaksi. Vuodesta 1627 hankkeessa työskenteli noin 400 ihmistä viidessä eri ryhmässä. Asiakirjat eivät osoita, että ryhmät—laivan runko, veistokset, takila, aseistus, painolasti—olisivat olleet vuorovaikutuksessa toistensa kanssa.

1600-luvulla ei ollut käytettävissä menetelmiä, joilla olisi osattu laskea aluksen painopiste, kallistumisominaisuuksia tai tasapainoisuutta. Siten aluksen kapteenin oli opittava aluksen käyttäytyminen kantapään (yrityksen ja erehdyksen) kautta. 1960-luvulla tehtyjen laskelmien mukaan Vasa-laiva oli niin kiikkerä, että se olisi kaatunut 10 asteen kallistumasta. Viimeisempien laskelmien mukaan neljän solmun (2 m/s) tuuli olisi kaatanut aluksen.

Vasa-laivan kapteeni Hannson ja miehistön ydinjoukko tekivät kesällä 1628 aluksen vakaustestin amiraali Flemingin valvonnassa. Kolmekymmentä miestä juoksi ryhmänä aluksen laidalta toiselle. Kolmen kannenylityksen jälkeen testi oli keskeytettävä, sillä alus uhkasi kaatua. Tästä huolimatta Vasa-laiva päätettiin lähettää matkaan.

Kymmenen ohjelmistoprojektien tautia ja joitakin lääkityssuosituksia

Monet Vasa-laivan ongelmat ovat vielä tänään nähtävissä useissa ohjelmistoprojekteissa. Fairley ja Willshire ovat koonneet kymmenen ongelma-alueen listan ja ehdottaneet joitakin parannuskeinoja kuhunkin ”tautiin”. Taulussa 7.1 on lyhyt yhteenveto.

1. Kiireinen aikataulu

Monien ohjelmistoprojektien aikataulu on epärealistisen tiukka. Fred Brooksin mukaan epärealistinen aikataulu on yleisin ohjelmistoprojek-

Taulu 7.1: Kymmenen ohjelmistoprojektien tautia ja joitakin lääkityssuosituksia [Fairley & Willshire, 20003]

Tauti:	lääkitys
Aikataulupaine:	objektiiviset arviot, resurssien lisäämien, resurssien parantamien, vaatimusten priorisointi, vaatimusten uudelleenkohdentaminen, tuotoksen vaiheistaminen
Tavoitteiden muuttuminen:	iteratiivinen ohjelmistokehitys, perusratkaisun hallinta
Teknisten määritysten puuttuminen:	alustavien määritysten tekeminen, määritysten päivittäminen, määritysten hallinnointi
Dokumentoidun projektisuunnitelman puuttuminen:	alustavan suunnitelman tekeminen, suunnitelman toistuva päivittäminen, projektisuunnitelman hallinnointi, nimetty projektipäällikkö
Yletön ja toissijainen innovointi:	perusdokumenttien hallinnointi, vaikutusten analysointi, jatkuva riskien hallinta, nimetty ohjelmistoarkkitehti
Vaatimusten luisuminen:	alustava vaatimusten perusversio, versioiden hallinnointi, riskien hallinta, nimetty ohjelmistoarkkitehti
Tieteellisten menetelmien puuttuminen:	prototyyppien tekeminen, vaiheittainen kehittäminen, suorituskykymittaukset
Olennaisen unohtaminen:	karkeat (*back-of-the-envelope*) laskelmat, opittujen opetusten sulauttaminen
Epäeettinen käyttäytyminen:	eettinen työympäristö ja työtavat, henkilökohtainen eettisten säännösten noudattaminen

tien epäonnistumisen syy. Hän väittää, että se on yleisempi kuin kaikki muut syyt yhteensä. Kun aikatauluarviot eivät perustu objektiiviseen dataan, ohjelmistojen kehittäjillä ei ole perusteita puolustaa omia aikaarvoitaan eikä vastustaa esitettyjä (epärealistisia) aika-arvioita.

Aikataulussa pysymiseksi voidaan:

- lisätä projektin resursseja,
- käyttää parempia resursseja,
- kohdentaa uudelleen (priorisoituja) vaatimuksia ja/tai
- vaiheistaa ohjelmiston valmistuminen.

2. Tarpeiden muuttuminen

Tavanomaisimpia syitä ohjelmistojen vaatimusten muuttumiseen ovat ulkoiset kilpailutekijät, käyttäjien tarpeiden muuttuminen, suoritusympä-

ristön muuttuminen ja projektin aikana tarkentunut näkemys. Vaatimusten muuttumisen vaikutuksia voidaan hallita joko iteratiivisilla ohjelmistotuotannon menetelmillä tai perusratkaisun hallinnoinnilla (*baseline management*).

Iteratiivisia menetelmiä käytettäessä vaatimusten muuttumiset voidaan helpohkosti priorisoida ottaen huomioon aikataulun, resurssien ja teknologioiden asettamat rajoitukset. Perusratkaisujen hallinnassa vaatimukset ja niiden muutokset käsitellään versioiden hallinnan menetelmillä. Hyväksytyt muutokset synnyttävät vaatimusten uuden version (ja uuden perusversion) siten, että aikataulut, resurssit, teknologiat ja muut tekijät on asianmukaisesti päivitetty.

3. Teknisten määritysten puuttuminen

Ohjelmistoprojekti voi alkaa pienenä ja tavanomaisena hankkeena, jossa tekniset määritykset tuntuvat tarpeettomilta. Joskus tällainen projekti muuttuu matkan varrella laajaksi ja innovatiiviseksi. Näin kävi Vasalaivan tapauksessa. Suullinen tiedonvaihto saattoi jossakin määrin korvata kirjallisten määritysten ja projektisuunnitelman puuttumista. Ohjelmistoprojektissa, oli se kuinka pieni tahansa, alustavien ja perusratkaisun vaatimusten kirjaaminen on paljon helpompaa ja vähemmän riskialtista kuin niiden myöhempi esiinkaivaminen ohjelmakoodista.

4. Projektisuunnitelman puuttuminen

Projektisuunnitelman tekemiseen pätevät samat huomiot kuin teknisten määritysten tekemiseen. Pienessäkin hankkeessa kannattaa tehdä projektisuunnitelma heti alussa. Myöhemmin projektisuunnitelman konstruointi on huomattavan hankalaa.

Ohjelmistoprojektin projektisuunnitelmassa on oltava:

- tehtävän työn jaottelu osatehtäviksi,
- vaatimusten sijoittaminen osatehtäviin,
- aikataulu tarkastuspisteineen ja virstanpylväineen,
- kunkin osatehtävän tuotokset määräaikoineen,
- tarvittavien ohjelmistojen hankintasuunnitelma,
- alihankinnan hallinnointisuunnitelma ja
- vastuiden selkeä kirjaaminen.

5. Yletön innovointi

Usein konstruointiprojektit epäonnistuvat, koska yritetään innovoida pitkälti yli sen hetkisen tietotaidon. Ohjelmistoprojektit ovat aina jossakin määrin innovatiivisia, sillä ohjelmiston monistaminen on triviaalia verrattuna fyysisten esineiden monistamiseen. Ohjelmistoprojekteissa luodaan joko aivan uusi järjestelmä tai olemassa olevasta järjestelmästä uusi versio.

Ohjelmistoprojektien yletöntä innovointia voidaan hallita seuraavilla menetelmillä:

- työdokumenttien (vaatimukset, projektisuunnitelma, suunnitteludokumentit, testaussuunnitelmat, lähdekoodi) versioiden hallinta,
- ehdotettujen muutosten vaikutusten analysointi ja
- jatkuva riskien hallinta.

6. Toissijaiset innovaatiot

Toissijaisiksi innovaatioiden syiksi Fairley ja Willshire mainitsevat käytettyjen teknologioiden asettamien rajoitusten kiertämisen ja kehittäjien luovat ratkaisut. Aivan kuin Vasa-laivan tapauksessa, ohjelmistoprojekteissakin toissijaiset vaatimukset voivat muodostua dominoiviksi. Siksi ohjelmistoprojekteissa olisi oltava vastuullinen ohjelmistoarkkitehti, joka huolehtii ohjelmatuotteen eheydestä.

7. Vaatimusten luisuminen

Näyttää siltä, että Vasa-laivan rakentamisen aikana kenelläkään ei ollut kokonaiskuvaa kaikkien tehtyjen muutosten vaikutuksista. Sama tilanne pääsee helposti syntymään myös ohjelmistoprojekteissa.

Keskeiset tekniikat, joilla ohjelmistoprojektit pidetään hallinnassa, ovat alustavan dokumentaation luominen sekä vaatimusten ja suunnitelmien toistuva päivittäminen, jotta vaatimusten, aikataulun ja resurssien välillä säilyy hyväksyttävissä oleva tasapaino. Usein kuultuja tekosyitä alustavan projektidokumentaation tekemättä jättämiselle ovat riittävän tiedon puuttuminen sekä uskomus, että suunnittelu on hyödytöntä, koska kaikki kuitenkin muuttuu.

Alustavat vaatimukset ja suunnitelmat tulee tehdä, vaikka kaikkea tarvittavaa tietoa ei olisikaan käytettävissä. Alustavissa projektidokumenteissa on varauduttava muutoksiin sekä esitettävä muutosten hallintame-

netelmät. Alustavien vaatimusten ja suunnitelmien toistuvan päivittämisen epäonnistuminen johtuu usein asenteesta, että päivittämiselle ei ole riittävästi aikaa. Tämä asenne puolestaan johtuu vaatimusten ja suunnitelmien jatkuvan hallinnoinnin kehittymättömistä menetelmistä sekä tähän liittyvien riskien aliarvioimisesta.

8. Tieteellisten menetelmien puuttuminen

Koska ohjelmistolla ei ole fyysisiä ominaisuuksia, niin monet perinteisen insinööritaidon menetelmät eivät sovellu ohjelmistotekniikkaan. Päinvastoin kuin Vasa-laiva ja monet muut fyysiset esineet, ohjelmisto voidaan rakentaa vaiheittain siten, että ohjelmiston ominaisuuksia, kuten muistintarvetta, suorituskykyä, turvallisuutta, tietoturvaa ja luotettavuutta, voidaan seurata ohjelmistoprojektin aikana.

9. Olennaisen unohtaminen

Vasa-laivan tapauksessa kallistustesti osoitti, että alus on vaarallisen epävakaa. Tasapainottaville lisäpainoille ei ollut tilaa, ja vaikka tilaa olisikin ollut, niin lisäpainoja ei olisi voitu käyttää, koska alemmat ampuma-aukot olisivat jääneet vesirajan alapuolelle. Vaikka monilla ohjelmistotekniikan alueilla ei ole tieteellisiä menetelmiä, niin karkeilla (*back of the envelope*) laskelmilla voidaan usein tunnistaa ratkaisevia suunnitteluvirheitä ja ohjelmistojen toiminnallisia puutteita.

10. Epäeettinen käyttäytyminen

Nykyisessä länsimaisessa yhteiskunnassa tekniikka tuottaa järjestelmiä ja tuotteita, jotka edistävät ihmiselämän aineellista puolta tehden elämän helpommaksi, turvallisemmaksi ja nautittavammaksi. Teknologisiin innovaatioihin liittyy usein eettisiä näkökohtia. Vasa-laivan tapauksessa aivan viime hetkillä oli tullut ilmeiseksi, että alus ei ollut merikelpoinen. Kuitenkin ne, joilla olisi ollut valtuudet estää neitsytmatka, sallivat aluksen lähdön satamasta. ACM:n ja IEEE:n eettisten säännösten [Gotterbarn, Miller & Rogerson, 1999] mukaan

> *Ohjelmistoammattilaisten on otettava täysi vastuu työstään ja hyväksyttävä ohjelmisto vain siinä tapauksessa, että heillä on perusteltu usko, että se on turvallinen, täyttää määritykset,*

> *läpäisee tarkoituksenmukaiset testit, ei vähennä elämän laatua tai yksityisyyttä eikä vahingoita ympäristöä.*

Tietojenkäsittelyn eettisiä kysymyksiä käsiteltiin luvussa 3.

7.3 Mallintaminen ja validointi

Mallintamisen ja validoinnin tarinaksi olen valinnut Charles E. Knadler Juniorin artikkelin *The Robustness of Separable Queueing Network Models* [Knadler, 1991]. Artikkelissaan Knadler analysoi suorituskykyanalyysissa käytettyjen jonoverkkomallien herkkyyttä niiden taustalla olevien matemaattisten mallien tarvitsemien oletusten paikkansapitävyydelle.

Jonoverkkomallit (*queueing network models*) nousivat tietokonejärjestelmien suorituskyvyn analysoinnin keskeiseksi työvälineeksi 1970-luvulla. Buzen osoitti, että klassisen jonoteorian analyyttiset tulokset pätevät myös löyhemmillä nk. operationaalisilla oletuksilla [Denning & Buzen, 1978]. Markovin jonoverkkojen tulokset pätevät myös separoituvissa jonoverkoissa. Keskeisin tulos on, että järjestelmän keskeisten suorituskykysuureiden keskiarvot voidaan laskea yksinkertaisella keskiarvoanalyysilla (*mean value anlysis, MVA*).

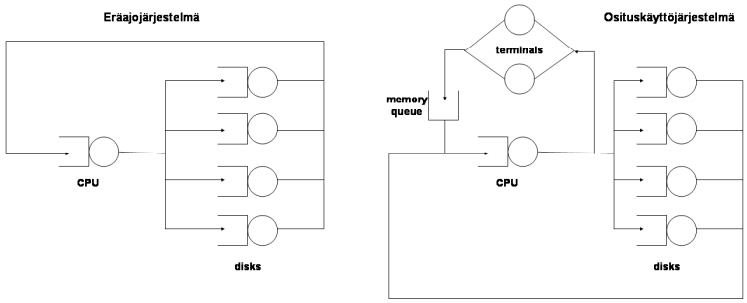
Jotta jonoverkko olisi separoituva, sen on täytettävä seuraavat ehdot:

- Jokainen palvelupiste on vuotasapainossa (eli kaikki palvelupyynnöt saadaan täytettyä äärellisessä ajassa).
- Minkään kahden työn tila ei muutu samanaikaisesti.
- Töiden kulku verkossa ei riipu palvelupisteiden jononpituuksista.
- Töiden palveluaika ei riipu järjestelmän kuormituksesta.
- Töiden saapumisajat eivät riipu järjestelmän kuormituksesta.

Joissakin tilanteissa vaaditaan lisäksi, että palveluajat ovat kuormituksesta riippumattomasti eksponenttiaalisesti jakautuneita. [Lazowska et al, 1984].

Tietokonejärjestelmät eivät yleensä täytä näitä operaationaalisia vaatimuksia:

1. Resurssien samanaikainen hallinta rikkoo oletusta, että *kahden työn tila ei muutu samanaikaisesti.* Nykyisissä järjestelmissä, jos muisti on rajoittava tekijä, keskusyksikön ja muistin varaukset kulkevat käsi kädessä. Nykyiset massamuistit eivät ole yksittäisiä resursseja,



Kuva 7.1: Eräajo- ja osituskäyttöjärjestelmää kuvaavat jonoverkkomallit [Knadler, 1991]

vaan väylä, ohjain, ja talletusalusta ovat samanaikaisesti käytössä luku- ja kirjoitusoperaatioiden aikana.

2. Levymuistien käyttämä kiertokulmantunnistin (*rotational position sensing, rps*) rikkoo oletusta, että *palveluaika ei riipu järjestelmän kuormituksesta.*

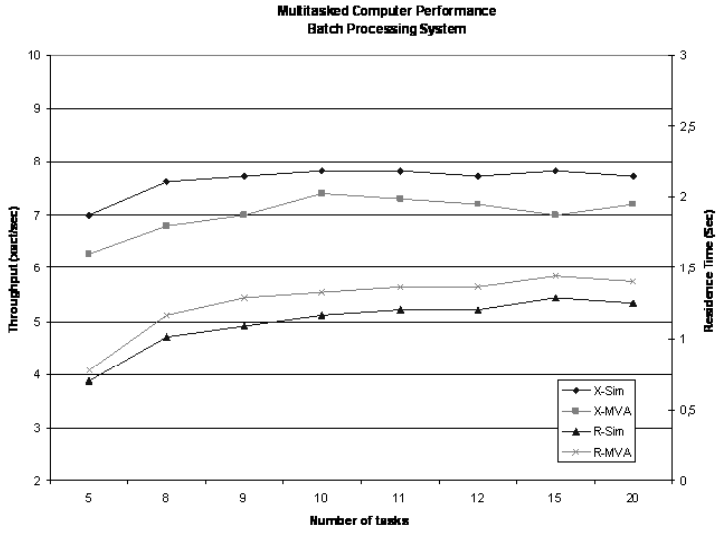
3. Lisäksi sekä keskusyksikkö että massamuisti voi rikkoa oletusta, että *palveluaika ei riipu järjestelmän kuormituksesta ja on eksponentiaalisesti jakautunut.*

Artikkelissaan Knadler tutkii, miten herkkä keskiarvoanalyysi on edellä mainituille kolmelle yleisesti rikotulle oletukselle. Kuvassa 7.1 on vanhoja eräajo- ja osituskäyttöjärjestelmiä kuvaavat jonoverkkomallit. Mallinnettu järjestelmä on yhden prosessorin (cpu) ja yhden väylän (*channel*) järjestelmä. Väylällä on neljä kiertokulmantunnistimella varustettua levy-yksikköä. Levy-i/o:n käyttö perustuu Knadlerin ja Mayn [Knadler & May, 1990] malliin ja sen suorituskyky Olsonin [Olson, 1989] mittauksiin.

Jonoverkkomalli

Tutkimuksessaan Knadler vertaa tulomuotoisen jonoverkkomallin ja simuloinnin antamia tuloksia. Jonoverkkomallin keskeiset suorituskykysuureet saadaan laskettua nk. MVA-algoritmilla [Lazowska et al, 1984]:

$$R_k(n) \quad = \quad D_k[1 + Q_k(n-1)]$$



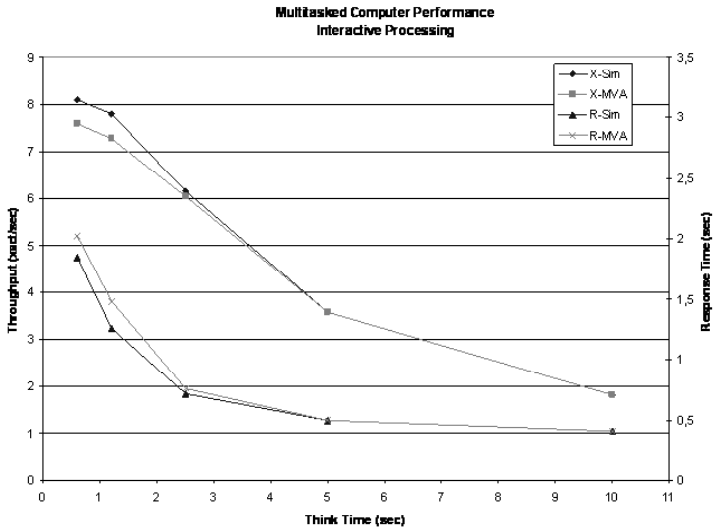
Kuva 7.2: Eräajojärjestelmää kuvaavan matemaattisen mallin ja simulointikokeiden antamia suorituskykysuureita [Knadler, 1991]

$$\begin{aligned} X(n) &= n/\left(Z + \sum_{k=1}^{K} R_k(n)\right) \\ Q_k(n) &= X(n)R_k(n) \end{aligned}$$

missä $R_k(n)$ on keskimääräinen viipymisaika palvelupisteessä k, kun järjestelmässä on n asiakasta, D_k keskimääräinen palveluaika palvelupisteessä k, $Q_k(n)$ on palvelupisteen k keskimääräinen jonopituus, kun järjestelmässä on n asiakasta, $X(n)$ on järjestelmän suoritusteho, kun järjestelmässä on n asiakasta, K on palvelupisteiden lukumäärä ja Z on asiakkaan keskimääräinen miettimisaika päätteellä.

Kuvassa 7.2 on jonoverkkomallin ja simuloinnin antamat eräajojärjestelmän suoritusteho ja läpimenoaika joillakin asiakasmäärillä. Jonoverkkomallin antama vasteaika eroaa alle 15 % simulointituloksesta ja suoritusteho alle 11 %. Tämän perusteella Knadler päättelee, että jonoverkkomallit ovat riittävän tarkkoja järjestelmän säätämiseen ja erilaisiin herkkyysanalyyseihin, vaikka todellinen levyjärjestelmä ei täytä jonoverkkomallin tulomuotoisen ratkaisun edellyttämiä oletuksia.

Kuvassa 7.3 on jonoverkkomallin ja simuloinnin antamat osituskäyt-



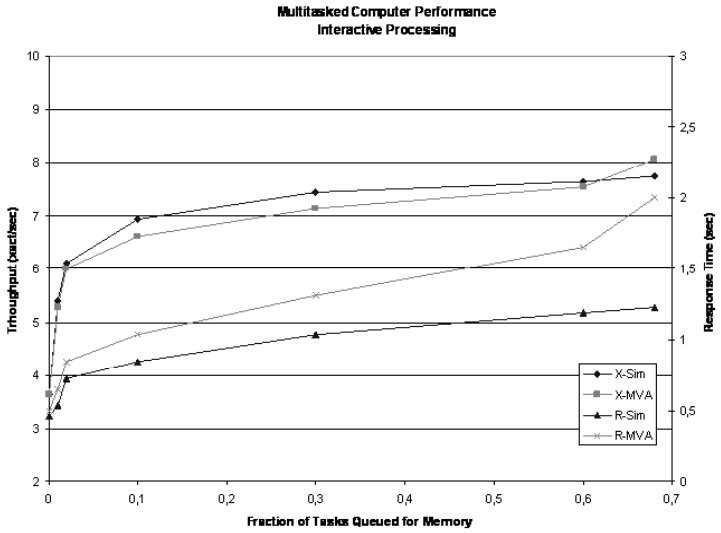
Kuva 7.3: Osituskäyttöjärjestelmää kuvaavan matemaattisen mallin ja simulointikokeiden antamia suorituskykysuureita [Knadler, 1991]

töjärjestelmän suoritusteho ja vasteaika muutamilla keskimääräisillä miettimisajoilla. Tässä tapauksessa jonoverkkomalli antaa lähes samat tulokset kuin simulointimalli varsinkin suuremmilla miettimisajoilla.

Resurssien samanaikainen käyttö

Jonoverkkomallissa oletetaan, että asiakas siirtyy palvelupisteestä toiselle käyttäen vain yhtä resurssia kerrallaan. Esimerkiksi järjestelmissä, joissa on kilpailua muistista tämä oletus ei päde. Hyvin monet tapahtumankäsittelyjärjestelmät rajoittavat samanaikaisesti käsiteltävien transaktioiden määrää, koska vähentynyt kilpailu järjestelmän sisällä parantaa suorituskykyä.

Knadler tutki jonoverkkomallin antamien tulosten tarkkuutta tällaisessa tilanteessa. Hän vaihteli miettimisaikaa siten, että töiden todennäköisyys joutua muistijonoon ennen suoritusta kasvoi 0 %:sta 68 %:in. Tulokset ovat kuvassa 7.4. Jonoverkkomallin antama suoritusteho on kaikissa tapauksissa hyvin lähellä (ero alle 4.3 %) simulointituloksia. Sen sijaan vasteaikojen ero kasvaa asteittain muistin jonotuksen todennäköisyyden



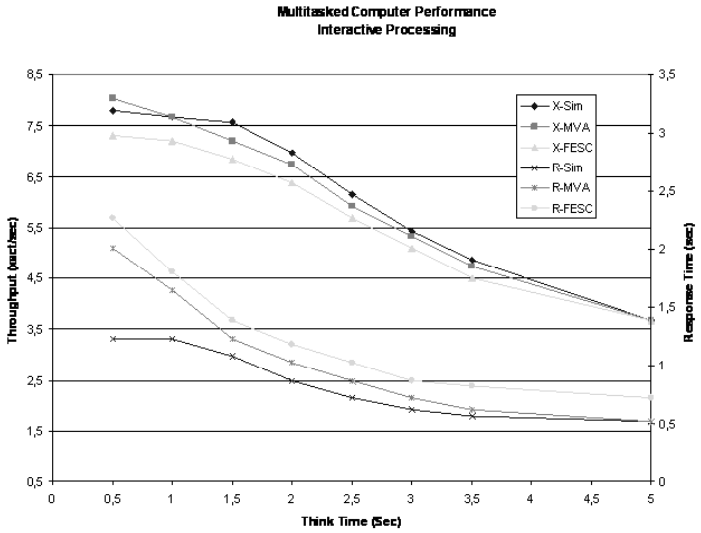
Kuva 7.4: Muistirajoitteista osituskäyttöjärjestelmää kuvaavan matemaattisen mallin ja simulointikokeiden antamia suorituskykysuureita [Knadler, 1991]

kasvaessa. Suurimmillaan ero on 62.9 %, kun muistijonotuksen todennäköisyys oli 68 %. Tämä johtuu siitä tosiasiasta, että tulomuotoinen jonoverkko heijastaa suurta tietokonejärjestelmän käyttäjäkuntaa, mutta muistirajoitus rajoittaa samanaikasten käyttäjien määrää. Kuitenkin Knadler pitää jonoverkkomalleja käyttökelpoisina, koska vasteajan ero oli alle 21 % niin kauan kuin muistijonotuksen todennäköisyys oli alle 30 %.

Järjestelmän ulkopuolella tapahtuvaa odotusta voidaan mallittaa nk. vuoekvivalentilla palvelupisteellä (*flow equivalent service center, FESC*). Siinä rajoitetun käyttäjäkunnan alijärjestelmä korvataan yhdellä palvelupisteellä, jonka palveluaika mallitetaan siten, että sen ja alijärjestelmän suoritusteho on sama.

Kuvassa 7.5 on perinteisen jonoverkkomallin, FESC-mallin ja simuloinnin antama suoritusteho ja vasteaika muutamilla miettimisajoilla. Kuten kuvasta nähdään FESCin tulokset ovat kauempana simulointituloksista kuin MVA-algoritmin tulokset. FESCin antama vasteaika kasvaa voimakkaasti miettimisajan lyhentyessä.

Artikkelissaan Knadler muistuttaa, että yhden esimerkkitapauksen pe-



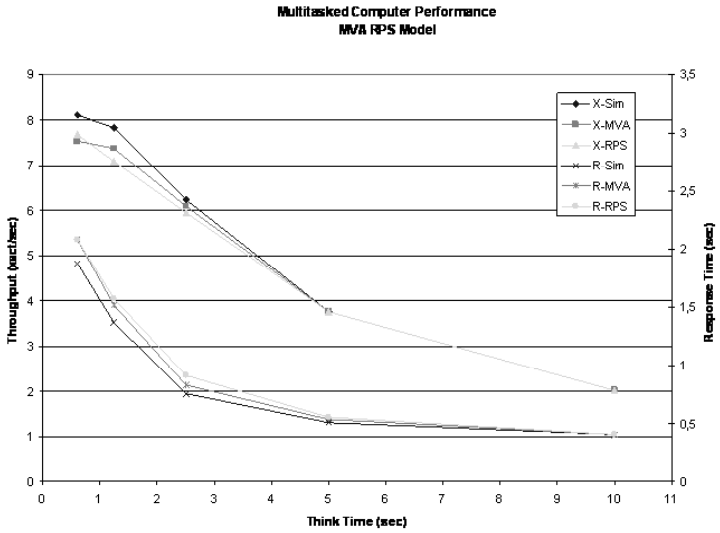
Kuva 7.5: Muistirajoitteista osituskäyttöjärjestelmää kuvaavan matemaattisen mallin ja simulointikokeiden antamia suorituskykysuureita [Knadler, 1991]

rusteella FESC-menetelmää ei pidä pitää käyttökelvottomana. Jonkinasteiseen varovaisuuteen on kuitenkin varauduttava.

Levyjärjestelmän analysointi

Lazowskan oppikirjassa [Lazowska, 1984] on kehitetty levyjärjestelmälle iteratiivinen malli, joka ottaa huomioon kiertokulmantunnistuksen sekä kanavan, ohjaimen ja levy-yksikön eri pituiset varausajat levy-i/o:ssa. Tässä mallissa ei tarvitse mitata levy-operaatioiden palveluaikoja, vaan ne lasketaan iteratiivisesti kahden mallin avulla, kunnes kummankin mallin antama suoritusteho on riittävän lähellä toisiaan.

Kuvassa 7.6 on perinteisen jonoverkkomallin, iteratiivisen levymallin (RPS) ja simuloinnin antama suoritusteho ja vasteaika muutamilla miettimisajoilla. RPS-mallin antamat tulokset ovat hyvin lähellä perinteisen jonoverkkomallin ja simuloinnin antamia tuloksia.



Kuva 7.6: Osituskäyttöjärjestelmää kuvaavan matemaattisen mallin ja simulointikokeiden antamia suorituskykysuureita [Knadler, 1991]

Knadlerin johtopäätökset

Artikkelinsa johtopäätöksissä Knadler toteaa osoittaneensa, että tulomuotoiset jonoverkkomallit ovat hyvin robusteja. Ne antavat hyviä tuloksia myös tilanteissa, joissa mallitettu järjestelmä ei täytä matemaattisen ratkaisun tarvitsemia oletuksia.

7.4 Innovaatiot

Everything that can be invented has been invented.
- Charles Duell[4]*, 1899.*

Muutosten aikaansaaminen käytäntöihin on paljon vaikeampaa kuin uusien teknologioiden keksiminen. Innovaatiot ovat uusia tapoja tehdä asioita. Innovaatioiden tarinan pohjaksi olen valinnut Peter J. Denningin

[4]USAn patenttiviraston johtaja erottuaan tehtävästään

huhtikuussa 2004 Communications of the ACM -lehdessä julkaistun kolumnin *"The Social Life of Innovation"* [Denning, 2004]. Kolumnissaan Denning väittää, että innovaatioiden käytännön voi oppia, kunhan tietää, mitä innovaatio on.

Innovaatio on yksi teknologian arvostetuimmista alueista. Elinkeinoelämän johtajat pitävät sitä yhtenä olennaisimpana osaamisalueena—ainoana tapana taata markkinajohtajuus. Teknologiaa popularisoivat lehdet julkaisevat vuosittain artikkeleja ja henkilökuvia merkittävimmistä innovaattoreista. Innovoinnista on julkaistu lukuisia kirjoja. Denning mainitsee kolme esimerkkiä: The Innovator's Dilemma [Christenson, 1997], Creative Destruction [Foster, 2001] ja Value Migration [Slywotzky, 1995]. Itse lisäisin listaan vielä teoksen Open Innovation [Chesbrough, 2003].

Monet organisaatiojohtajat puhuvat välttämättömyydestä luoda innovaatiokulttuuri. Johtajuutta on kannustettava, koska muuten organisaatio keskittyy vain ratkaisemaan tämän päivän kiireisiä ongelmia. Innovaatiokulttuuria ei saada aikaan sivistämättä organisaatioita innovaatioiden henkilökohtaisista käytännöistä. Mitä nämä käytännöt ovat? Miten opitaan taitavaksi innovoijaksi? Miten innovatiivisuutta voidaan opettaa? Denningin kolumni yrittää vastata näihin kysymyksiin.

Innovaatio ja keksintö

Termiä innovaatio on käytetty tarkoittamaan sekä uusia ajatuksia että uusia käytäntöjä. Koska uusilla ajatuksilla ei ole käytännön vaikutusta ellei niitä sovelleta käytäntöön, Denning tarkoittaa innovaatiolla uuden käytännön soveltamista. Siten innovaatio on sosiaalinen muutos yhteisössä.

Denning tekee selkeän eron innovaation ja keksinnön välillä. Keksintö on uuden luomista: ajatuksen, esineen, laitteen tai toimintatavan. Riippumatta siitä, miten erinomainen keksintö on, se ei välttämättä muodostu innovaatioksi. Innovaation ja keksinnön erottaminen eri käsitteiksi on olennaista, koska innovoinnin käytännöt eivät ole keksimisen käytäntöjä. Keksinnöissä voidaan keskittyä teknologioihin. Innovaatioissa on otettava huomioon sosiaalinen yhteisö, mitä muut ihmiset arvostavat ja hyväksyvät otettavaksi käyttöön.

Lehtihaastattelussa vuodelta 1999 Ethernetin keksijä Bob Metcalfe kuvaa osuvasti tätä eroa. Haastattelijan toteamukseen, että Ethernetin keksiminen mahdollisti hänen ostaa asunto Bostonin rantakaistaleelta, Metcalfe vastasi, että ei keksiminen, vaan Ethernetin myyminen kymme-

nen vuoden ajan.

Vaikka liikemaailman innovaatiot saavat suurimman osan julkisuudesta, innovaatioita tapahtuu kaikenlaisissa organisaatioissa ja yhteisöissä. Myös innovaatioiden vaikutusten laajuudet vaihtelevat.

Innovointi on innovaattorien työtä, jolla on omat lainalaisuudet ja käytännöt. Innovointia voidaan tehdä systemaattisesti ja sen perusperiaatteita voidaan opettaa. Onnistuminen ei vaadi karismaa, inspiraatiota, erityistä lahjakkuutta tai älynväläystä. Denningin mukaan innovoinnin menestys on koulutuskysymys. Innovoinnin ydin on ajatusten myyminen tai niiden saattaminen käytännön toteutuksiksi.

Innovoinnin systematiikka

Yksi innovoinnin merkkiteoksia on Peter Druckerin vuonna 1985 julkaisema kirja *Innovation and Entrepreneurship* [Drucker, 1985]. Tässä kirjassa keskitytään kahteen pääasiaan:

innovaatioiden käytäntö: Innovaattori etsii mahdollisuuksia ja muuttaa ne markkinoiden uusiksi käytännöiksi.

yrittäjyyden käytäntö (*entrepreneurship*): Organisaation tavat omaksua innovaatiot.

Taulussa 7.2 on Druckerin laajaan analyysin perusteella esille nostamat innovaatioprosessin viisi kulmakiveä. Denningin mukaan 1990-luvun ”dotcom” kupla olisi ehkä voitu välttää, jos uusien yritysten johtajat olisivat kaikki lukeneet Drukerin teoksen.

Druckerin kirjasta yli puolet käsittelee mahdollisuuksien etsimistä, jotka oli jaettu seitsemään ryhmään (katso taulu 7.3). Kaikkein olennaisinta on etsiä mahdollisuuksia erilaisissa tavanomaisesta poikkeavissa tilanteissa. Druckerin neljä ensimmäistä lähdettä ovat tavallisesti organisaation toiminnan sisäisiä haasteita, joita voidaan kehittää ilman ulkoisen kilpailun tuottamaa painetta. Kolme viimeistä liittyvät yrityksen ulkoiseen toimintaympäristöön. Kahdeksanneksi lähteeksi Denning on nostanut *marginaaliset käytännöt* [Spinoza et al., 1997]. Nämä ovat käytössä toisilla aloilla ja näyttävät epärelevanteilta. Esimerkiksi hyperteksti (*hypertext*) oli tietojenkäsittelyssä marginaali-ilmiö, kunnes Tim Berners-Lee teki siitä Webin keskeisen osan.

Drucker varoittaa, että uuteen tietämykseen perustuva innovointi on erityisen haastavaa. Se on kaikkein riskialtteinta ja sillä on pisin kypsymisaika. Tällainen innovointi vaatii tarkan ajoituksen ja etsikkoaika on

Taulu 7.2: Innovaatioprosessin peruselementit [Denning, 2004] (Denningin lähteenä [Drucker, 1985])

mahdollisuuksien etsiminen: mahdollisuuden tunnistaminen eri lähteistä (katso taulu 7.3)
analysointi: projekti- tai liiketoimintasuunnitelman laatiminen, kustannusten, resurssien ja ihmisten identifiointi
kuunteleminen: yhteisöjen huolenaiheiden kuunteleminen, ymmärtäminen mihin ihmiset ovat valmiita, ehdotusten muokkaaminen yhteisöä palveleviksi
keskittyminen: keskeisen idean pelkistäminen yksinkertaiseksi sanomaksi ja siinä pitäytyminen, vaikka mieli tekisi koristella tai laajentaa sitä ennenaikaisesti
johtajuus: kehittää teknologia lajinsa parhaaksi, saada ihmiset ja yhteisöt käyttämään sitä, muokata sille markkinarako

yleensä lyhyt. Kaiken lisäksi tällaiset alueet ovat kaikkein kilpailluimpia, sillä uusi tietämys saa suurimman huomion.

Yltiöpäisiä riskejä ottavat yrittäjät Drucker leimaa myyteiksi. Menestyneet yrittäjät käyttävät systemaattisia toimintatapoja, jotka vähentävät riskejä.

Innovoinnin henkilökohtaiset peruskäytännöt

Analysoidessaan laajasti innovaatioita ja innovointia Denning on tunnistanut kahdeksan menestyksekkään innovoinnin olennaista elementtiä. Denning väittää, että innovointi tuskin tulee onnistumaan, jos näitä käytäntöjä ei noudateta.

tietoisuus (*awareness*): kyky oivaltaa mahdollisuuksia ja kiinnostavia asioita, tunnistaa omien tavoitteiden kannalta kiinnostavat, kyky ylittää älyllinen (*cognitive*) sokeus

keskittyminen ja pitkäjänteisyys (*focus and persistence*): kyky keskit-

Taulu 7.3: Innovaatioden mahdollisuudet [Denning, 2004]
(seitsemän ensimmäistä [Drucker, 1985], viimeinen [Spinoza et al., 1997])

1. **odottamattomat tapahtumat:** odottamattomat onnistumiset tai epäonnistumiset, ulkoiset tapahtumat
2. **epäsuhdat:** todellisuuden ja yleisen uskomuksen välinen kuilu, asianosat, jotka eivät sovi yhteen
3. **prosessin tarpeet:** kriittisessä prosessissa oleva pullonkaula
4. **liike-elämän rakennemuutos:** uudet liiketoimintamallit, jakelukanavat, toimintatavat
5. **demografia:** ikärakenteessa, politiikassa, uskonnossa, elintasossa, jne. tapahtuvat muutokset
6. **ilmapiirin tai asenteiden muuttuminen:** ihmisen maailmankuvan (esim. 9/11 jälkeinen terrorismi), muodin, tapojen, jne. muuttuminen
7. **uusi tietämys:** uuden tietämyksen hyödyntäminen, johon usein liittyy tieteen tai tekniikan läpimurto ja aiemmin erillisten toiminta-alueiden yhdentyminen
8. **marginaaliset käytännöt:** reuna-alueiden käytännöt, jotka voivat ratkaista nykyisissä keskeisissä käytännöissä esiintyviä jatkuvia häiriöitä

tyä perustehtävään (*mission*) ja välttää syrjähypyt (*distractions*), perustehtävässä pitäytyminen kaaoksen, haasteiden tai vastustuksen keskellä, perustehtävän jatkaminen esteistä ja haasteista huolimatta

kuunteleminen ja sovitteleminen (*listening and blending*): keskeisten huolenaiheiden ja mielenkiinnon kohteiden kuunteleminen, toimenpiteiden muokkaaminen näihin sopivaksi (*"finding the win-win"*)

julistukset (*declarations*): kyky tehdä yksinkertaisia, voimakkaita, koskettavia, puhuttelevia julistuksia, jotka luovat muille mahdollisuuksia ja kiinnostavia uusia näköaloja

laajakatseisuus (*destiny*): toimiminen laajempien kuin vain itseä koskevien päämäärien hyväksi

tarjoukset (*offers*): tarjousten, jotka luovat lisäarvoa asiakkaillesi palveluina, käytäntöinä tai esineinä, tekeminen ja täyttäminen, ryhmien organisoiminen ja sitoutumisen (*commitment*) ylläpitäminen tulosten aikaansaamiseksi

verkottuminen ja instituutiot (*networks and institutions*): liittolaisten hankkiminen, vastustajia vastaan puolustautuminen, instituutioiden perustaminen innovaation edistämiseksi, yhteisen standardin luominen innovaation hyväksyttävyyden lisäämiseksi

oppiminen (*learning*): ajan varaaminen uusien taitojen oppimiseksi, uuden tietämyksen hankkiminen, hyvin perusteltujen arvioiden tekeminen oppimisen ja uusien toimenpiteiden valmistelussa

Denning huomauttaa, että julistukset ja laajakatseisuus ovat lähellä toisiaan. Tarjoukset, verkottuminen ja instituutiot kuuluvat johtajuuden saavuttamiseen. Kaikki kahdeksan peruskäytäntöä tukevat ”markkinointia”, jossa kaikki menestyksekkäimmät innovaattorit ovat mestareita.

Denningin esittämät käytännöt tukevat Druckerin esittämää toimintatapaa. Esimerkiksi mahdollisuuksien etsinnässä innovaattori tarvitsee laajaa tietoisuutta, jotta hän pystyy tunnistamaan mahdollisuudet. Keskeinen haaste on ylittää älyllinen sokeus, tila, jossa emme näe jotakin emmekä näe, että emme näe sitä. Yhteistyö ihmisten kanssa, jotka ovat merkittävästi eri mieltä tai ovat toisista yhteisöistä, voi olla erittäin hyödyllistä älyllisen sokeuden ylittämiseksi. Toinen keskeinen taito on keskittyä innovaation ytimeen (*core*). Tällainen keskittyminen—pysyminen asian ytimessä—vaatii harjaantumista ja kurinalaisuutta välttää rönsyilyt ja koristelut (*embellishments*).

Innovoinnin perustavaa laatua olevat henkilökohtaiset käytännöt ovat ensisijaisesti sosiaalisia, ei teknisiä. Innovaation keskeinen tavoite on tunnistaa jotain muilta ihmisiltä puuttuvaa, aikaansaada uusi toimintatapa. Tekniset innovaatiot edellyttävät innovaattorilta sekä sosiaalisia että teknisiä taitoja, joko-tai ei riitä.

Denningin kolumnissa on esimerkkinä World Wide Web ja sen keksijä Tim Berners-Lee. Pääasiassa Berners-Leen kirjan *Weaving the Web* [Berners-Lee, 2000] perusteella Denning kuvaa, miten tietoisuus, kuunteleminen ja sovittelu, keskittyminen ja pitkäjänteisyys, oppiminen ja johtajuuden saavuttaminen ilmenivät Berners-Leen toiminnassa. Denningin mukaan Berners-Leen kirja on yksi harvoista mahdollisuuksista tutustua yksityiskohtaisesti tietojenkäsittelyn innovaatioon siten kuin sen tekijä asian näki ja koki. Ilkka Tuomi on kirjassaan *Networks of Innovation* [Tuomi, 2003] analysoinut kolmea tietojenkäsittelyn innovaatiota: Web, Internet ja Linux. Yhteisenä piirteenä on se tosiasia, että tekijät tiedostivat pyrkivänsä sosiaaliseen muutokseen, eivät vain uuden teknologian keksimiseen.

Denningin suositukset

Innovaattoriksi kehittyminen edellyttää, että ymmärtää innovaatioprosessin mahdollisuudet ja keskeiset käytännöt. Lukemalla voi oppia ymmärtämään innovaatioprosessia. Denningin suosituslistan kärjessä ovat Drucker, Berners-Lee ja Tuomi.

Denning suosittaa muistiinpanojen jatkuvaa tekemistä. Tämä kehittää taitoa tulla mahdollisuuksien havaitsijaksi ja huolenaiheiden kuuntelijaksi. Kirjaa myös asiat, joita muut tuntuvat pitävän tärkeinä. Yritykset ymmärtää eri mieltä olevia harjaannuttavat älyllisen sokeuden ylittämistä.

Itseopiskelulla on vaikea parantaa tietoisuutta, laatia ja tiivistää julistuksia, ylläpitää keskittymistä, laatia toimintasuunnitelmia, tehdä tarjouksia ja viedä toimenpiteet päätökseen. Nämä ovat ensisijaisesti sosiaalisia taitoja, joiden oppimista valmentaja tai opettaja olennaisesti helpottavat.

Lopuksi Denning luettelee neljä väärinkäsitystä, jotka helposti lannistavat aloittelijan:

1. *Innovaatioiden on oltava isoja.* Useimmat menestyksekkäät innovaattorit oppivat taitonsa pienistä innovaatioista.

2. *Innovaatiot ovat vain muutamien lahjakkuuksien työsarka.* Runsaasti julkisuutta saaneita innovaattoreita on vähän. Useimmat innovaatiot tehdään työpaikoilla ja organisaatioiden sisällä. Kuka tahansa voi oppia parantamaan työyhteisönsä työtapoja.

3. *Innovaatiot perustuvat uusiin ajatuksiin.* Uusi tietämys on vain yksi innovaatioiden lähteistä—se kaikkein riskialttein. Muut seitsemän lähdettä ovat myös mahdollisuuksien sampoja.

4. *Innovaatioita tapahtuu vain elinkeinoelämässä.* Innovaatioita tehdään myös kaiken kokoisissa yhteisöissä, hallinnon ja koulutuksen aloilla sekä voittoa tavoittelemattomissaa organisaatioissa ja yhteisöissä.

7.5 Soveltaminen

Peter Denningin mukaan soveltamisessa on kyse yhteistyöstä. Työskennellään yhdessä tietojenkäsittelyn sovellusalueiden ammattilaisten kanssa näitä palvelevien tietojenkäsittelyjärjestelmien toteuttamiseksi. Toinen alue on työskentely muiden tietojenkäsittelyn ammattilaisten kanssa useita erilaisia sovelluksia tukevien ydinteknologioiden kehittämiseksi.

Soveltamisen tarinaksi olen valinnut Ahmed Seffahin *ACM Interactions* -lehdessä syys-lokakuussa 2003 julkaistun artikkelin *Learning the Ropes: Human-Centered Design Skills and Patterns for Software Engineers' Education* [Seffah, 2003]. Se käsittelee niitä taitoja, joita ohjelmistoammattilaiset tarvitsevat ihmiskeskeisessä suunnittelussa (*Human-Centered Design*). Ihmiskeskeinen suunnittelu on noussut viimeisten vuosien aikana dominoivaksi teemaksi suunnittelussa. Poleemisessa kolumnissaan *Human-Centered Design Considered Harmful* [Norman, 2005] Donald Norman varoittelee, että siitä on tullut ehkä liiankin dominoiva, koska se hyväksytään automaattisesti suunnittelun lähtökohdaksi ajattelematta ja kritiikittömästi. Norman ei kuitenkaan kiellä, etteikö ihmiskeskeinen suunnittelu olisi parantanut tuotteita. Hänen mukaansa ihmiskeskeisellä suunnittelulla saadaan aikaan hyviä tuotteita, mutta käyttäjillä ei useinkaan ole loistavan suunnittelun tarvitsemaa näkemystä.

Ihmiskeskeisen suunnittelun yksi keskeisistä periaatteista on kuunnella käyttäjiä. Tämä ei yksistään riitä. Ohjelmistoammattilaisten tulee tietää, miten käytettävyyttä (*usability*) mitataan ja miten havaintoaineiston perusteella tehdään päätöksiä. Artikkelissaan Seffah esittää 19 taitoa, joita tarvitaan menestyksekkäässä ihmiskeskeisessä suunnittelussa.

Ihmiskeskeisen suunnittelun taidot

Taito (*skill*) on sarja koordinoituja toimenpiteitä, jotka ovat yleensä tehokkaita tavoitteen saavuttamiseksi. Nykyisin termillä taito on ruvettu tarkoittamaan mitä tahansa kykyä, joka on saatu kokemuksen ja asiaankuuluvan tietämyksen omaksumisen myötä. Se muodostuu tunnustetusta

asiantuntemuksesta tai pätevyydestä, jonka avulla henkilö pystyy ymmärtämään ja ratkaisemaan tietyn alan ominaisia ongelmia.

Identifioidessaan ihmiskeskeisen suunnittelun tarvitsemia taitoja Seffahin johtama Concordia yliopiston (Montreal, Kanada) ihmiskeskeisen ohjelmistotekniikan ryhmä tukeutui sekä ihmisen ja koneen välisen vuorovaikutuksen (*human-computer interaction, HCI*) että ohjelmistotekniikan koulutusyhteisöihin. He käyttivät pitämiensä kurssien—niin yliopistoissa kuin teollisuudelle suunnattujen koulutustilaisuuksien—kirjallista palautetta sekä ACM-SIGCHI:n suosituksia ihmisen ja koneen välisen vuorovaikutuksen opetussisällöistä ja verkosta löytyvien alueen kurssikuvausten analysointia. Lisäksi he analysoivat keskeisiä oppikirjoja. Erityisen merkittävänä tietolähteenä Seffah mainitsee IEEE:n Computer Society:n koordinoiman *Guide to the Software Engineering Body of Knowledge* -projektin (SWEBOK[5]). Kyseisen projektin tuottaman dokumentaation avulla saa kattavan kuvan tämän hetken ohjelmistotekniikan käytännöistä.

Kuten jo aiemmin kerroin, Seffah identifioi yhdeksäntoista taitoa, joita ohjelmistoammattilaiset tarvitsevat ihmiskeskeisessä suunnittelussa. Nämä taidot ryhmiteltiin kolmeen ryhmään: välttämättömät edellytykset (*prerequisite skills*), erityistaidot (*specific skills*) ja yleistaidot (*generic skills*).

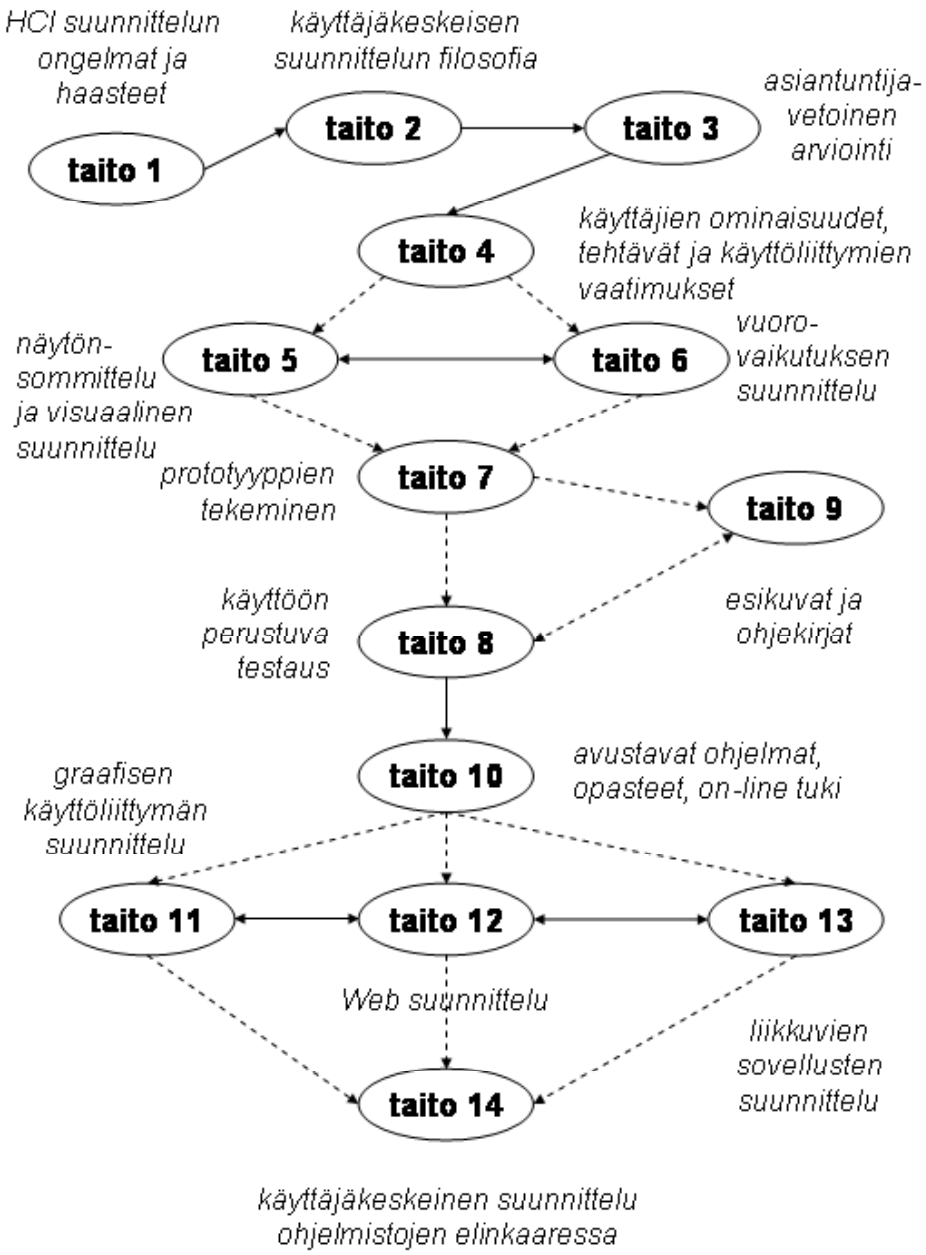
Välttämättömät edellytykset

Välttämättömiin edellityksiin Seffah katsoo kuuluvan seuraavat kaksi taitoa:

1) ohjelmistojen kehitysmenetelmien perusperiaatteiden, perusteiden ja standardien sekä käyttöliittymien kehitystyövälineiden ja oliopohjaisten suunnittelumenetelmien perusteiden hallitseminen

2) tietokoneiden käyttökokemus sekä ohjelmointikokemus käyttäen nopean kehittämisen työvälineitä ja graafisten käyttöliittymien kehitysympäristöjä

[5] `http://www.swebok.org`



Kuva 7.7: Ihmiskeskeisen suunnittelun erityistaitojen kartta. [Seffah, 2003]

Erityistaidot

Tähän ryhmään Seffah on sijoittanut kuvassa 7.7 olevat neljätoista taitoa. Jostakin syystä Seffah numeroi taidot tekstissään ja kuvassa erilailla. Kuvan 7.7 '**taito n**' on tekstin '*taito n+2*'. Lisäksi hän käytti kuvassa sekä käyttäjäkeskeistä suunnittelua tarkoittavaa lyhennettä UCD että ihmiskeskeistä suunnittelua tarkoittavaa lyhennettä HCD. Käytännössä näillä käsitteillä ei ole mitään eroa.

3) *Ihmisen ja koneen vuorovaikutuksen suunnittelun vaikeudet ja haasteet:* Vuorovaikutteisen järjestelmän rakentamisen ymmärtäminen monitieteisenä suunnittelualana, jolla on sen omat erityispiirteet ja vaikeudet.

4) *Ihmiskeskeisen suunnittelun*[6] *filosofia:* ihmiskeskeisen suunnittelun filosofian, sen perusteiden, terminologian, faktojen, periaatteiden ja prosessien hallitseminen

5) *Asiantuntijavetoinen arviointi:* varhaisten suunnittelukonseptien ja alhaisen totuudenmukaisuuden (*fidelity*) prototyyppien (mukaanlukien tehtävä- ja skenaario-perustaiset prototyypit) ilman käyttäjiä tapahtuva katselmointi ja arviointi

6) *Käyttäjien ominaisuudet, tehtävät ja käyttöliittymien vaatimukset:* käyttäjien ominaisuuksien, heidän tehtäviensä ja työympäristön sekä käytettävyyden ja käyttöliittymän vaatimusten kokoaminen, analysointi ja määrittely

7) *Näytön sommittelu, visuaalinen ja tietosisällön suunnittelu:* Näytön ulkoasun ja tietosisällön sekä visuaalisten kielikuvien, ulkoasun ja värienkäytön valinta ja suunnittelu

8) *Vuorovaikutuksen suunnittelu:* ihmisen ja koneen välisen keskustelun sekä järjestelmän antaman palautteen, erityisesti virheilmoitusten ja opasteiden mallintaminen ja määrittäminen sekä tarkoituksenmukaisen vuorovaikutuksen tyylien ja laitteiden valinta

9) *Prototyyppien tekeminen:* alhaisen, keskitason ja suuren totuudenmukaisuuden prototyyppien (kuten laitemallit (*mockups*), ta-

[6] Seffahin kuvassa oli UCD eli käyttäjäkeskeinen suunnittelu.

rinataulut (*story boards*), ohjelmat, videot, paperiprototyypit) kehittäminen

10) *Käyttöön perustuva testaus:* täysin toiminnallisiin järjestelmiin ja suuren totuudenmukaisuuden prototyyppeihin perustuvien käytettävyystestien suunnittelu ja toteutus kehityksen ja käyttöönoton eri vaiheissa

11) *Esikuvat ja ohjekirjat:* Suunnittelun esikuvien (*design patterns*), ohjekirjojen (*guidelines*) ja tyylioppaiden (*style guides*) avulla saatujen käyttäjäkokemusten ja parhaiden suunnittelukäytäntöjen kokoaminen, levittäminen ja hyväksikäyttäminen

12) *Käyttöohjeet, suoritusaikaiset opasteet ja tukijärjestelmät:* Käyttäjätuen toteuttaminen sisältäen käyttöohjeet, avustavat ohjelmat (*wizards*), suoritusaikaiset opasteet (*online help systems*), käyttäjien kouluttamisen ja palautejärjestelmät

13) *Graafisen käyttöliittymän suunnittelu:* ihmiskeskeisen suunnittelun menetelmien ja työvälineiden käyttäminen graafisten käyttöliittymien ja toimiston pöytäkoneille tarkoitettujen sovellusten toteuttamisessa

14) *Web-sovellusten suunnittelu:* ihmiskeskeisen suunnittelun menetelmien ja työvälineiden käyttö Web-sovellusten (Web-sivut, sähköisen kaupankäynnin sovellukset ja virtuaaliyhteisöt) toteuttamisessa

15) *Liikkuvien sovellusten suunnittelu:* ihmiskeskeisen suunnittelun menetelmien ja työvälineiden käyttäminen liikkuvien (*mobile*) ja langattomien (*wireless*) sovellusten toteuttamisessa kämmenlaitteille (*personal digital assistant, PDA*) ja matkapuhelimille

16) *Ihmiskeskeinen suunnittelu*[7] *ohjelmiston elinkaaressa:* käyttöliittymien suunnittelun käytettävyystekniikoiden (*usability engineering*) menetelmien yhdistäminen ohjelmistokehityksen elinkaareen ja menetelmiin

[7] Seffahin kuvassa oli UCD eli käyttäjäkeskeinen suunnittelu.

Yleistaidot

Kolmanteen ryhmään Seffah on koonnut joitakin yleistaitoja tai geneerisiä taitoja, joita tarvitaan lukuisissa ihmiskeskeisen suunnittelun tehtävissä. Toisissa yhteyksissä näitä taitoja on kutsuttu *”pehmeiksi taidoiksi”* (*“soft skills”*). Näiden taitojen tärkeydestä vallitsee laaja yksimielisyys niin käytettävyyden kuin ohjelmistotekniikan asiantuntijoiden keskuudessa. Kanadan Software Productivity Centerin[8] tekemän selvityksen mukaan tämä näkemys on vallalla myös alan teollisuudessa.

Seffahin mukaan ihmiskeskeisen suunnittelun koulutuksessa on opetettava myös näitä *”pehmeitä taitoja”*. Ei riitä, että opetetaan käyttöliittymien suunnittelua. Ohjelmistoammattilaisille on myös opetettava, miten käyttäjät ja muut asianosaiset (*stageholder*) otetaan mukaan ja miten heidän kanssa ollaan yhteistoiminnassa ohjelmiston koko elinkaaren aikana.

Seffahin mukaan kolme keskeisintä yleistaitoa ovat:

17) suurelle yleisölle tarkoitettujen, vaatimuksiin, suunnitteluun ja testaukseen liittyvien käyttäjäkeskeisten dokumenttien kirjoittaminen ja esittely

18) ohjelmistojen ja käytettävyyden työvälineiden ja menetelmien tutkiminen empiirisesti ja kustannustehokkaasti

19) monialaisessa ryhmässä kommunikointi ja työskentely, erityisesti asiakkaiden, käyttäjien ja käyttäytymistutkijoiden (*human factors engineers*) kanssa

Taustamateriaalia

Denning, P. J. (2004) The Social Life of Innovation. Communications of the ACM, 47, 4, huhtikuu 2004, sivut 15–19.

Fairley, R. E. & Willshire, M. J. (2003) Why the Vasa Sank: 10 Problems and Some Antidotes for Software Projects. IEEE Software, 20, 2, maalis-huhtikuu 2003, sivut 18–25.

Knadler Jr., C. E. (1991) The Robustness of Separable Queueing Network Models. Proceedings of the 1991 Winier Simulation Conference, sivut 661–668. Available from ACM Digital Library.

[8] `http://www.spc.ca/`

Seeley, D. (2004) How Not to Write FORTRAN in Any Language. ACM Queue, 2, 9, joulukuu 2004/tammikuu 2005, sivut 58–65.

Seffah, A. (2003) Learning the Ropes: Human-Centered Design Skills and Patterns for Software Engineers's Education. ACM Interactions, 10, 5, syys-lokakuu 2003, sivut 36–45.

Lähteitä

Berners-Lee, T. (2000) Weaving the Web. Harper Business.

Brooks Jr., F. P. (1987) No Silver Bullet — Essence and Accidents of Software Engineering. IEEE Computer, 20, 4, huhtikuu 1987, sivut 10–19.

Brooks Jr., F. P. (1995) "No Silver Bullet" Refired. Luku 17 teoksessa F. P. Brooks Jr. Mythical Man-Month, 20^{th} Anniversary Edition, Addison-Wesley.

Chesbrough, H. (2003) Open Innovations. Harvard Business School Press.

Christenson, C. (1997) The Innovator's Dilemma. Harvard Business School Press.

Denning, P. J. ja Buzen, J. P. (1978) The Operational Analysis of Queueing Network Models. ACM Computing Surveys, 10, 3, syyskuu 1978, sivut 225–261.

Dijkstra, E. D. (1968) Go To Statement Considered Harmful (letters to editors). Communications of the ACM, 11, 3, maaliskuu 1968, sivut 147–1148.

Drucker, P. (1985) Innovation and Entrepreneurship. Harper.

Foster, R. (2001) Creative Destruction. Currency.

Gotterbarn, D., Miller, K. & Rogerson, S. (1999) Computer Society and ACM Approve Software Engineering Code of Ethics. IEEE Computer, 32, 10, lokakuu 1999, sivut 84–88.

Knadler, C. E. ja May, R. (1990) Disk I/O, A Study in Shifting Bottlenecks. Proceedings of the 1990 Winter Simulation Conference, sivut 826–830. (ACM Digital Library)

Knuth, D. E. (1975) Structured Programming with go to Statements. ACM Computing Surveys, 6, 4, joulukuu 1974, sivut 261–301.

Lazowska, E. D., Zahorjan, J., Graham, G. S. & Sevcik, K. C. (1984)

Quantitative System Performance Computer System Analysis Using Queueing Network Models. Prentice-Hall. (`http://www.cs.washington.edu/homes/lazowska/qsp/`.)

Norman, D. A. (2005) Human-Centered Design Considered Harmful. ACM Interactions, 12, 4, heinä-elokuu 2005, sivut 14–19.

Olson, T. M. (1989) Disk Array Performance in a Random IO Environment. ACM Computer Architecture News, 17, 5, syyskuu 1989, sivut 71–77.

Slywotzky, A. (1995) Value Migration. Harvard Business School Press.

Spinoza, C., Dreyfus, H. ja Flores, F. (1997) Disclosing New Worlds. MIT Press.

Tuomi, I. (2003) Networks of Innovation. Oxford Press.