Osa C, teine diskett.

Sissejuhatus reaalajatarkvaratehnikasse
LAP 5711

Abimaterjal loengute koduseks analüüsiks.
koostas L.Mõtus

Peamised raamatud:

1. H.Kopetz “Real-time systems; Design principles for distributed embedded applications”, Kluwer Academic Publishers, Boston/Dordrecht/London, 1997, 338 pp., ISBN 0-7923-9894-7
2. P.T.Ward, S.J.Mellor “Structured development for Real-time Systems” vol.1, Prentice-Hall, 1985, 156 pp, ISBN 0-13-854787-4 025
3. S.R. Schach “Classical and Object-Oriented Software Engineering”, IRWIN (a Times Mirror Higher Education Group, Inc), 1996, 603, ISBN 0-256-18298-1
4. J.Rumbaugh, et al “Object-oriented modeling and design”, Prentice-Hall, 1991, 500 pp., ISBN 0-13-629841-9
5. I.Sommerville “Software Engineering”, Addison-Wesley Publishing Company, 1992, 649 pp.,ISBN 0-201-56529-3
6. A.Behforooz and F.J.Hudson “ Software Engineering Fundamentals”, Oxford University Press, 1996, 661 pp., ISBN 0-19-510539-7
7. L.Motus, M.G.Rodd “ Timing analysis of real-time software”, Elsevier Science Publishing/Pergamon, Oxford, 1994, 212 pp., ISBN 0 08 0420265 (hardcover), ISBN 0 08 0420257 (flexicover)

Viited täiendavatele allikatele on antud tekstis.

Õppematerjaliga saab tutvuda URL aadressil
http://www.dcc.ttu.ee/Automaatika/LAP/LAP5711/,
samas on kopeeritavad WORD.7 failid.

Viimati täiendatud

August 1999

Sisukord

I osa
Süsteemi- ja tarkvaratehnika

1.1 Tarkvaratehnika üldine kontekst 6
1.2 Süsteemitehnika – süsteemide loomise meetodid ja vahendid 7
1.3 Süsteem 8
1.3.1 Süsteemile iseloomulikud omadused 9
1.4 Tarkvarale erinevaid nõudeid esitavad süsteemide klassid 10
1.5 Eelülevaade tarkvarasüsteemide loomise protsessist 11
1.6 Loengukursuse pragmaatiline eesmärk ja selle saavutamiseks kasutatud teed 13
1.7 Esimese osa kordamisküsimused 15
1.8 Esimeses osas kasutatud kirjanduse viited 15

II osa
Sissejuhatus reaalajasüsteemidesse

2.1 Reaalajasüsteemid 16
2.2 Reaalajas töötav arvutisüsteem 17
2.3 Funktsionaalsed nõuded reaalajasüsteemile 20
2.3.1 Andmehõive 21
2.3.2 Juhtimine 24
2.3.3 Inimliides 26
2.4 Alarmiseire ja eriolukordade töötlus 27
2.5 Ajakitsendused reaalajasüsteemis 29
2.5.1 Näide ajakitsenduste mõnedest allikatest 30
2.6 Mittefunktsionaalsed nõuded reaalajasüsteemidele 33
2.6.1 Töökindlus 34
2.6.2 Ohutus 34
2.6.3 Hooldatavus 35
2.6.4 Valmisolek 36
2.6.5 Turvalisus 36
2.7 Reaalajasüsteemide klassifikatsioon 37
2.7.1 Ranges või nõrgas reaalajas töötavad süsteemid 37
2.7.2 Ohutult riknevad või töövõimet säilitavalt riknevad süsteemid 40
2.7.3 Garanteeritud ajakitsendustega või parimate võimalike ajakitsendustega süsteemid 40
2.7.4 Piisavate ressurssidega või mittepiisavate ressurssidega süsteemid 41
2.7.5 Sündmuste poolt või aja poolt juhitud süsteemid 41
2.8 Reaalajasüsteemide levik ja maksumuse hindamine 42

2.8.1 Sardsüsteemid kitsamas mõttes 43
2.8.2 Protsessijuhtimissüsteemid 45
2.8.3 Multimeedia süsteemid 46
2.9 Teise osa kordamisküsimused 46
2.10 Teises osas kasutatud kirjanduse viited 47
2.11 Teise osa terminite selgitav sõnastik 48

III osa
Tarkvara projektide planeerimine ja haldamine

3.1 Meenutusi tarkvaratehnikast 50
3.2 Süsteemi (ja tarkvara) loomise üldine kirjeldus 52
3.2.1 Suurte süsteemide loomisel sageli esinevad probleemid 52
3.2.2 Süsteemi analüüs ja projekteerimine 55
3.2.3 Mudeli ja tegelikkuse vahekord tarkvaratootes 56
3.3 Süsteemi loomise planeerimine 57
3.3.1 Kommentaarid süsteemi loomise plaanile 59
3.3.2 Projekti jagamine töödeks 59
3.3.3 Ressursside vajadus 62
3.3.4 Projekti perioodilised ülevaatused 64
3.3.5 Projekti muudatuste haldamine 66
3.3.6 Tarkvara loomise plaan 66
3.4 Süsteemi loomise elutsükkel 67
3.4.1 Tarkvara elutsükkel 68
3.4.2 Tarkvara elutsükli mudelid 71
3.4.2.1 Üldine kaskaadmudel 72
3.4.2.2 DOD mudel 74
3.4.2.3 NASA mudel ja kokkuvõte kaskaadmudelitest 75
3.4.2.4 Spiraalne mudel 76
3.5 Tarkvara protsessi kirjeldamise mudelid 78
3.6 Tarkvaraprotsessi puudutavad kordamisküsimused 80
3.7 Tarkvaratehnikas kasutatavad tööriistad 81
3.7.1 Üldised abivahendid 83
3.7.1.1 Andmesõnastik 83
3.7.1.2 Otsustamine ja kompromisside tegemine 85
3.7.1.3 Stsenaariumid 86
3.7.2 Praktikas enamlevinud mudelid tarkvara töö ja/või struktuuri kirjeldamiseks 87
3.7.3 Kordamisküsimused 89
3.8 Tarkvara meetrika 90
3.8.1 Näiteid tarkvaratoote kvaliteedimeetrikas kasutatavatest mõistetest 90
3.8.2 Tarkvara toodet iseloomustavad parameetrid 91
3.8.3 Tarkvara projekteerimist ja realiseerimist kirjeldavad koefitsiendid 92
3.9 Tarkvaratootega seotud riski haldamine 93

3.9.1 Tüüpilised riskifaktorid tarkvaraprojektides 93
3.9.2 Riski mudel 96
3.9.3 Riski haldamine 98
3.10 Tarkvara meetrika ja riski haldamise kordamisküsimused 99
3.11 Kolmandas osas kasutatud kirjanduse loetelu 99

III osa
Tarkvaraprojektide planeerimine ja haldamine

3.1 Meenutusi tarkvaratehnikast

Uued inimesed, kes satuvad tarkvaratoodet tootvasse meeskonda ei kujuta sageli ette, et tarkvaratoode ei ole mitte ainult tükike silutud programmi, vaid on osa keerulisest riistvara, inimeste, süsteem- ja rakendustarkvara ning bürokraatlike protseduuride kompleksist. Eriti kehtib see arvutiteaduse värskete lõpetajate kohta, kes kipuvad oma arvuti- ja tarkvaraalaseid teadmisi ülehindama.

Praktiliselt kõigil värsketel tarkvarainseneridel on tarkvara projekti ajalise ja rahalise külje planeerimine üsna suur müsteerium. Programmistidel on reeglina sisemine vastumeelsus igasuguse planeerimise vastu (eriti rahade ja aja planeerimise vastu). Kui planeerimine on lisaks veel seotud kohustusega lubadusi plaani kohaselt täita, muutub kogu asi veelgi vastumeelsemaks. Kõik tarkvara insenerid loodavad, et planeerimine – eriti aja ja raha planeerimine – toimub kusagil väljaspool tarkvara projekti meeskonda. Ja nad kõik eksivad rängalt. Isegi siis kui keegi administraator planeerib tarkvara projekti, ei ole projekti edu reaalne ilma meeskonnasisese alternatiivse planeerimiseta (või vähemalt ilma väljaspool tehtud plaani korrigeerimiseta).

Tarkvaratehnika kui omaette distsipliini nimetus tekkis aastatel 1967 – 1968, seoses tarkvaraprojektide katastroofilise hilinemise ja rahade ülekulutamisega. Tarkvaratehnika nimetus hakkas pikkamööda sisuga täituma 1970-ndatel aastatel, edaspidi hakkasid uurimistulemused ja soovitused tulema järjest suurema kiirusega. Näiteks mõned verstapostid:

- Programmeerimise “hea tava” põhiprintsiipide väljatöötamine (1968-1971) – langev projekteerimismetoodika (*top-down development*), järk-järguline arendamine (*stepwise refinement*), modulaarse tarkvara algus;
- Programmeerimise stiili mõiste sissetoomine (1972-1973) – struktuurne programmeerimine (*structured programming*), tarkvaratehnika tööriistade (*CASE tools*) loomise algus;
- Töökindluse ja kvaliteedinõuete ilmutatud esitamine (1974-1975) – testimine (*testing procedures*), koodi formaalne verifitseerimine (*formal verification of code*), programmi töökindluse mudelid (*software reliability models*) vahel nimetatud ka jääkvea ennustamise mudeliteks, hakati tõsisemalt tegelema tarkvara hinna ennustamisega (*software cost analysis*);
- Kasutaja nõuete ilmutatud esitamine (1976-1977) – tarkvara elutsükkel, nõuete spetsifitseerimise (*requirements specification*) etapp, projekteerimisetapi tükeldamine kaheks (abstraktseks ja detailseks) osaks (*high-level design or architectural design, low level design or detailed design*); esimesed publikatsioonid tarkvaratehnika tööriistadest;
- Tööstus avaldas esmakordset huvi tarkvaratehnika vastu (1978-1980) – esimesed tarkvaratehnika õppekursused, esimesed katsed tarkvara projekti alusel automaatseks koodi genereerimiseks;
- Reaalajatarkvaratehnika (*real-time software engineering*) kui omaette mõiste ilmumine (1981-1984) – tarkvaratehnika tööriistad hakkavad tegelema hajus- ja paralleeltarkvaraga (*distributed and concurrent software*);

- Tarkvaratehnika märgatav kasutamine praktikas (1989-1990), objekt-orienteeritud tarkvaratehnika meetodite edu (1990-1992)
- Uute ideede hulgaline tulek (1993-1996) – tehisintellekti meetodid tarkvaratehnikas, UML (*Unified Modelling Language*) koondab paljud objekt-orienteeritud meetodid ühtseks tervikuks, CORBA tekkimine ja mõju laienemine (*Common Object Request Broker Architecture*) hajusrakendustes ja suurtes süsteemides.

Kõigele vaatamata on tarkvara loomine pidevas kriisiseisus, nii nagu see oli 1960-ndatel – 1990-ndate aastate USA-s on ligi 30% alustatud tarkvaraprojektidest täielikult ebaõnnestunud, ligi 90% lõpuni viidud projektidest on olnud tunduvalt maas ajagraafikust ja tunduvalt ületanud planeeritud maksumuse.

Lisaks tarkvaratehnika meetodite suhtelisele algelisusele on sellisele seisundile kaasa aidanud kolm järgmist põhjust:
- olemasolevad tarkvaratehnika meetodid ja vahendid ei ole leidnud laialdast praktilist kasutamist (ilmne põhjus on vahendite kõrge hind, vähem ilmsed on potentsiaalsete kasutajate puudulik haritus ja pakutavatest meetoditest tulenev liiga vähene kasu (s.t. turul olevates tööriistades kasutatavate meetodite triviaalsus – paljud tööriistad täidavad pliiatsi ja paberi funktsioone)
- lihtsate, üldtunnustatud ja omaksvõetud standardite puudumine tarkvara tootmise protsessi ja tarkvara kui toote nõutavate omaduste kohta -- on sadu ametkondlikke ja riiklikke standardeid, kogumahuga tuhandetes lehekülgedes; üldine arvamus on, et need on praktiliseks kasutamiseks liiga abstraktsed ja keerulised
- tarkvara loomise protsess on käesoleval ajal üks keerulisemaid inimtegevuse valdkondi, puuduvad korralikud formaalsed, tarkvara kvantitatiivseid omadusi kirjeldavad matemaatilised mudelid, mis võimaldaksid varakult ennustada tulevase süsteemi omadusi.

Tänu tarkvara kui rakendusvaldkonna keerukusele ei saa loota kiirele edule, kuid olemasolevaid teadmisi mõistlikult kasutades on võimalik üsnagi edukalt vältida paljusid tavalisi vigu ning potentsiaalseid ohtusid.

Tarkvaratehnika kursuste eesmärk on:
- põhjendada õppuritele tarkvaratehnika (*software engineering*) kui distsipliini vajadust; see kõlab naeruväärselt, aga paljud tarkvaratehnika põhitõed tunduvad esmapilgul sedavõrd triviaalsed, et programmistid ei võta neid tõsiselt (analoogia halduse ja haldusjuhtimisega)
- selgitada, et tarkvara loomine on lahutamatu osa süsteemide loomisest, ning tarkvara inseneri tegevuspiirkond ja vastutus laieneb tegelikkuses tunduvalt väljapoole tarkvara kui omaette toote piire
- selgitada, miks tarkvara inseneri tegevus on lähedasem süsteemi-inseneri tegevusele kui arvuti-inseneri või arvutiteadlase tegevusele
- tutvustada ja õpetada kasutama lihtsaid, asjakohaseid ja praktikule arusaadavaid süsteemi ja tarkvara loomise protsessi erinevaid etappe kirjeldavaid mudeleid
- firmade empiirilise kogemuse üldistamine ja edastamine mudelite, eeskujude, dokumendivormide, tabelite ja eeskirjade ning soovituste näol, koos selgitustega, miks need on head ja vajalikud

- illustreerida eduka tarkvara projekti tegemise võimalust näidetega ja demonstreerida projekti realiseerimiseks loodud keskkondade kasulikkust
- rõhutada ja selgitada süsteemi ja tarkvara loomisprotsessi planeerimise tähtsust ja tarkvara inseneri rolli planeerimisel.

Paljudes ülikoolides on tarkvaratehnika kursus viidud üsna stuudiumi algusse – shokiteraapia abil tehakse õppuritele selgeks, et ilma tarkvaratehnika põhitõdedeta ei ole ettenähtud (kokkulepitud) ajaga, rohkem kui ühest inimesest koosnevas grupis, võimalik luua mõistlikult töötavat süsteemi.. Selline lähenemine eeldab, et tudengid saavad arvuti ja programmeerimise algtõed kätte juba keskkoolis – Eesti ei ole veel nii kaugele jõudnud, sellepärast ka soovitavalt stuudiumi viies semester.

Käesolevas osas selgitab mõningaid olulisemaid tarkvaratehnika alustõdesid – projektide eelnev planeerimine, testimine, realiseerimine, tarkvara omaduste mõõtmise küsimused (tarkvara meetrika), riski hindamine ja selle vähendamise teed, jne.

3.2 Süsteemi (ja tarkvara) loomise üldine kirjeldus

Süsteemi loomine on iteratiivne rühmategevus, mis algab süsteemi eesmärkide, nõuete ja kitsenduste defineerimisega ja lõpeb tootega, mis rahuldab süsteemi spetsifikatsioonis loetletud eesmärke, nõudeid ja kitsendusi eelnevalt fikseeritud täpsusega ja nõuetes ettenähtud aja jooksul. Loomulikult vaatame ainult selliseid süsteeme, mis sisaldavad arvutit ja rakendustarkvara alamsüsteemi. Süsteemi projekti õnnestumiseks on oluline, et süsteemi-insenerid ja tarkvara insenerid oleksid koostöös kogu süsteemi loomise kestel. Tegelikkuses ei ole võimalik rangelt eraldada etappe, kus süsteemi-inseneri töö lõpeb ja tarkvara inseneri töö algab.

Viimane väide on eriti asjakohane reaalajasüsteemide loomisel – arvuti kaudu tekib suletud juhtimisahel, mis võib oluliselt muuta esialgselt hinnatud (ilma arvutita eksisteerinud) süsteemi omadusi ja temale esitatud nõudeid ning kitsendusi. Teoreetilistel ja praktilistel põhjustel võivad esialgselt hinnatud omadused ja nõuded muutuda projekti realiseerimise käigus (võrdle avatud ja suletud ahelas töötavates kontrollerites kasutatavate algoritmide erinevat iseloomu juhtimisteoorias) ja niiviisi suurendada lahendamist vajavate probleemide keerukust suurusjärgu võrra (vt. ka Giddings 1984).

Süsteemitehnikast on pisut räägitud õppematerjali esimese osa alajaotuses 1.2, süsteemi täpne määratlus on toodud alajaotustes 1.3.

3.2.1 Suurte süsteemide loomisel sageli esinevad probleemid

Reaalselt on iga tarkvara toode vaadeldav mingi suurema süsteemi alamsüsteemina. Üha suurem osa süsteemide poolt pakutavast teenusest realiseeritakse arvutisüsteemi abil, üha sagedamini tagatakse enamus funktsioone tarkvara alamsüsteemi poolt – näiteks, telefonikeskjaamad, arvutiga otseselt juhitavad lennukid (*fly-by-wire airplanes*) jne. Selline olukord suurendab

tarkvara inseneride vastutust – tarkvara inseneride vastutus on äärmuseni viidud ohutuskriitiliste reaalajasüsteemide (*safety-critical real-time systems*) tegemisel ja kasutamisel.

Järgnevalt on loetletud mõned tüüpilised probleemid, mis tekivad kõigi küllalt suurte süsteemide tegemisel -- eriti huvitavad meid suured tarkvarasüsteemid. On loomulik, et tarkvaratehnika püüab leida meetodeid, mis vähendaksid nende sageliesinevate probleemide poolt põhjustatud riski tarkvaraprojekti õnnestumisele.

Süsteemi loomise kestus (*Development time problem*) Süsteemi loomise kestuseks loetakse aega tulevase süsteemi esimese kontseptsiooni fikseerimise hetkest kuni esimese töötava eksemplari andmiseni lõppkasutaja kätte. Suurte süsteemide loomise käigus võib tegevuse algul aluseks võetud tehnoloogia (arvutid, tarkvara, tehnoloogilised seadmed, jne) põhjalikult muutuda. Paljude projektide alustamisel ei ole süsteemi teisi osi veel olemas (isegi mitte projektina), eksisteerivad vaid nõuded tulevase süsteemi osadele. Taoliste projektide näideteks võiksid olla:

- võimsa elementaarosakeste kiirendaja tegemine – töö alustamisel on peamiseks piiravaks lüliks hajutatud arvutisüsteemi võime erinevates arvutites töötavaid kelli piisava täpsusega sünkroniseerida, piisava kiirusega infot koguda, töödelda ja teiste arvutitega vahetada, ning kiiresti ja piisavalt hästi sünkroniseeritult juhtida hajusalt paiknevaid tehnilisi seadmeid– pärast nende küsimuste põhimõttelist lahendamist saab tõsiselt hakata projekteerima ja katsetama ka muid süsteemi komponente;
- järgmise sajandi tanki projekt – milles on 15-20 erinevat kohtvõrku ja ligi 100 protsessorit, hajutatud reaalaja-andmebaas, satelliitside; projekti eesmärgiks on süsteemi maksimaalne vastupanu riketele ja kahjustustele (*viability* = visadus); peamiseks probleemiks on sobiva võrgu topoloogia ja tarkvara arhitektuuri väljatöötamine, mis tagaks piisava kiirusega küllaldase ning usaldatava info kättesaadavuse kõigile süsteemi osadele võimalikult suurte riistvaraliste ja tarkvaraliste kahjustuste korral; pärast selle probleemi lahendust saab kogu süsteemi projektiga edasi minna – s.t. defineerida ka teiste tarkvaraprojekti osade kohta täpsemad funktsionaalsed ja mittefunktsionaalsed nõuded.

Süsteemi loomise käigus võib üsna sageli kaduda vajadus süsteemi kui terviku, või mõne tema osa järele. Samuti võib olukord muutuda sedavõrd, et süsteemi esialgne spetsifikatsioon kaotab oma mõtte. Ideaalne projekt näeb sellised variandid ette -- kindlates kontrollpunktides hinnatakse projekti jätkamise otstarbekust, samades punktides võidakse otsustada projekti lähtetingimusi muuta, kooskõlas värskelt tekkinud uute arusaamadega.

Nimetatud probleemi mõju nõrgendamiseks kasutatavad vastuabinõud aitavad vaid osaliselt, vahel aga võivad teha asja isegi hullemaks. Efektiivsed universaalsed vastuabinõud seni (ja vist põhimõtteliselt) puuduvad, igal konkreetsel juhul sõltub edu projektijuhi ja meeskonna ettenägemisvõimest ja töökogemusest. Väga pikaajalise projektiga seotud riskide vähendamiseks kasutatakse näiteks järgmisi vastumeetmeid:

- puuduva tehnoloogiaga tugevalt seotud projektotsused lükatakse ajas nii kaugele kui vähegi võimalik; selle otsuse vastu võitleb “bürokraatia traditsioon” -- enne projektile rahade eraldamist nõutakse kasutatavate seadmete ja tehnoloogiate detailset kirjeldust; seega tuleb loota ka haritud bürokraatide olemasolule;

- valitakse piisavalt vanad ja järeleproovitud seadmed ja/või tehnoloogia; tulemuseks võib olla kasutu süsteem, kuna riistvara ja kasutatud seadmed ei vasta süsteemi valmimisajal esitatavatele nõuetele (või pole enam tootmises, või ei ole ühildatavad uuemate seadmetega)
- süsteem projekteeritakse ja ehitatakse etappide kaupa, muutes riistavara ja tehnoloogilist baasi vastavalt selle arengule; paljudes rakendustes on etapiviisiline süsteemi loomine andnud häid tulemusi -- tõeline oht selle meetodi kasutamisel on vahepeal (töö ajal) muutuda võivad liideste standardid ja sellest tekkivad interaktsiooni raskused erinevatel süsteemi arengu etappidel loodud osade vahel.

Süsteemi inimliidese probleem (*System's user interface problem*). See probleem tekib siis kui projekti algstaadiumis ei pöörata piisavalt tähelepanu inimliidese funktsioonidele ja neid funktsioone toetavate operatsioonide mõjule süsteemis liikuvatele andmetele. Näiteks, pärast süsteemi üleandmist kasutajale selgub vajadus anda inimesele juurdepääs täiendavatele andmeelementidele (mõõtmistele, juhttoimetele, alarmisõnumitele). Teiseks näiteks on süsteemi võimetus garanteerida andmete ajalist kooskõla alarmilaviini ajal. Kehva projekteerimise korral selguvad sellised hädad alles siis kui süsteem on juba kasutusele võetud (vt. Bransby (1998)). Paljud reaalselt juhtunud avariid on tingitud seda laadi möödalaskmistest inimliidese funktsionaalsete nõuete täitmist toetavate abifunktsioonide (ja mittefunktsionaalsete nõuete) mittepiisaval läbimõtlemisel, või teoreetiliselt mittesobiva (mittepiisavalt formaalse) arvutusmudeli valimine tarkvara projekti ja realisatsiooni aluseks.

Soovitused inimliidesega seotud probleemide vähendamiseks on järgmised:
- elutsükli algetappidel pöörata rohkem tähelepanu mittefunktsionaalsetel nõuetele
- tulevased kasutajad tuleb kaasata süsteemi projekteerimisse juba nõuete defineerimise etapil;
- aegsasti teha kasutaja tegevuse täpsed stsenaariumid, kus kasutaja tegevuste järjekord ja ajakitsendused on kirjas (nii normaalse töö kui ka avariiolukorras, erilist tähelepanu tuleb pöörata harva esinevates situatsioonides tegutsemise stsenaariumitele (näiteks alarmilaviin);
- võimaluse korral valida süsteemi loomise paradigmaks spiraalne elutsükli mudel, mis võimaldab süsteemi valmistamise prototüüpide jadana; igal prototüübil saab kasutaja läbi mängida teda huvitavaid stsenaariume ja tulemuste põhjal soovitada muutuseid edasistesse prototüüpidesse.

Katsetamise ja integreerimise probleem (*Test and integration problem*). On üsna tavaline, et palavikulises projekteerimise ja realiseerimise hoos ununeb läbi mõtelda tarkvarasüsteemi modulaarsus (süsteemi arhitektuur, süsteemi sisemine struktuur) katsetuste, diagnostika, vastuvõtu- ja üleandmiskatsetuste ning osi ühendavate liideste vaatepunktist. Projekti algetappide kiirendamiseks lükatakse need tööd tavaliselt edasi (hilisemaks täpsustamiseks), aga edaspidi läheb elu veelgi kiiremaks.

Ainus mõistlik lahendus on määrata katsetamisega ja tervikuks integreerimisega seotud tegevuste ja dokumentatsiooni eest vastutavaks parajalt suurte kogemustega isik. Erilist tähelepanu tuleb pöörata vastuvõtu- ja üleandmiskatsetustele, mis peavad demonstreerima tellijale, et süsteem rahuldab kõiki esitatud nõudeid. Olukord on veelgi tõsisem kui toode peab läbima sertifitseerimise. Katsetuste programm ja sisu tuleb tingimata kooskõlastada tellija ja kasutajaga ning, vajaduse korral, ka sertifitseerijaga. Sertifitseerimise nõue toob sageli kaasa toote omaduste

formaalse verifitseerimise vajaduse. Igal juhul peab tarkvara struktuur võimaldama vajaduse korral üksikute osade mugava väljavahetamise või modifitseerimise.

Toote hoolduse probleem (*maintenance problem*) lükatakse projekteerimise ajaks tihti kõrvale -- see on kaugem probleem, hädalisemad probleemid püütakse lahendada enne. Sellise praktika tulemusena moodustab hoolduse ja kasutuse faasi maksumus (isegi 1990-ndatel) kuni 60%. süsteemi kogu elutsükli maksumusest (paljudel juhtudel isegi kuni 80% kogu maksumusest). Tarkvaratehnika seisukohalt on 60% lubamatult suur osa kuludest. Probleem ise on tõsine -- paljude süsteemide eluiga on 15-30 aastat. Nii pika eluea jooksul peab süsteem kohanema nii tehnoloogia muutustega, süsteemi funktsionaalsuse ja jõudlusnõuete kardinaalsete muudatustega, kui ka mitme generatsiooni kasutajate nõudmistega inimliidesele.

Ainus võimalus hooldust lihtsustada on süsteemi arhitektuur hästi läbimõeldult projekteerida ja kasutada tarkvaratehnika keskkonna abi süsteemi modifitseerimiseks. Praktiline raskus on selles, et tarkvaratehnika keskkonnad, mis suudaksid spetsifikatsioonis tehtud muudatuse sobivust kontrollida ja teisendada modifitseeritud nõuded ja funktsioonid (pool)automaatselt objektkoodi, on alles katsetusjärgus. Sellistest keskkondadest räägitakse palju, on ilmunud üksikuid katsevariante, mis ei ole veel tööstuslikuks kasutamiseks sobivad.

Ühise eesmärgi puudumine (*Lack of common purpose*) tähendab, et tegijal, finantseerijal ja kasutajal puudub ühine arusaam süsteemi eesmärkidest. Arusaamade erinevused ei pea tingimata olema ilmutatult defineeritud või vastukäivad, erinevad tõlgendused võivad selguda alles süsteemi üleandmiskatsetustel. Sisuliselt viitab see mitteprofessionaalsele projekti juhtimisele – kolmepoolsed arutelud on olnud üldsõnalised või üldse puudunud, mõisted on olnud täpselt defineerimata, potentsiaalsed lahkarvamused on koosoleku protokollides silutud (selle asemel, et neid võimendada ja püüda järgmiseks koosolekuks lahendada).

Võib-olla aitab siin ammune (aastast 1961) IBM spetsialistide antud soovitus:

- süsteemi-insener peab süsteemi tegija kontoris olema tellija/kasutaja advokaat ja tellija/kasutaja kontoris peab ta olema süsteemi tegija advokaat.

Teiste sõnadega, süsteemi-insener peab looma vastastikuse arusaamise tellija/kasutaja ja tarkvara loova grupi vahel. Kui tulevase süsteemi omaduste kohta tekivad erinevad lootused, siis on selles peasüü süsteemi-inseneril. Palun ärge unustage, et süsteemi-inseneri ja tarkvara-inseneri rollide vahele süsteemide loomise protsessis on suhteliselt raske selget piiri tõmmata.

Kontseptuaalne segadus mudeli ja tegelikkuse vahekorras – on üsna sageli tarkvaratoote mittesobivuse põhjuseks. Eriti reaalajasüsteemide loomise praktikas on tugevalt tuntav arvutiteaduse pikaajaline traditsioon vaadelda programmi kui mudeli mudelit. Mudeli ja tegelikkuse vahekorrale on pühendatud alajaotus 3.2.3, probleemi üldine kontekst on selgitatud käesoleva abimaterjali esimeses osas.

3.2.2 Süsteemi analüüs ja projekteerimine

***Analysis** = the breaking up of any whole so as to find out its nature, functions, etc*

***Design** = to make preliminary sketches of; to sketch a pattern or outline for a plan; to plan and to carry out especially by artistic arrangements or in a skillful way.*

Süsteemi analüüs seisneb rakenduse uurimises – lähtudes tulevasele süsteemile seatud eesmärgist – selleks, et saada võimalikult hästi aru lahendust vajavast probleemist. Selline uurimine sisaldab vestlusi (*interviews*), vaatlusi (*observations*), katsetusi (*hands-on experiments*), konsultatsioone (*consultations*), ja palju muid andmete ja teadmiste kogumise vorme. Edaspidi, spetsifitseerimise ja projekteerimise etapil kasutatakse terminit "analüüs" ka veidi teises tähenduses – seal on tegemist spetsifikatsiooni ja/või projekti juba saavutatud omaduste hindamiseks läbiviidud uuringutega.

Süsteemi-inseneri ja tarkvara-inseneri ülesanded süsteemi analüüsil on praktiliselt kattuvad. Nende tööülesannete vahel vahe tegemine sõltub konkreetse firma konkreetsest olukorrast. On soovitav, et tarkvara insener lülituks töösse juba süsteemi analüüsi faasis – mida varem seda parem. Eriti reaalajasüsteemide tegemisel on oluline, et tarkvara insener saaks aru juhitava/jälgitava kobara olemusest ja teaks selle kobara kvantitatiivseid omadusi – paljud tarkvara mittefunktsionaalsed nõuded tulenevad otseselt juhitava/jälgitava kobara omadustest.

Süsteemi analüüsi tulemuste alusel peaks ideaalis tekkima realisatsioonist sõltumatu projekt (*implementation-free design*). Päris puhtal kujul ei ole selline asi reaalselt võimalik kuna projekteerijad ja tellijad peavad, vähemalt alateadvuses, alati silmas reaalseid kitsendusi (arvutite tüübile, rahasummale, jne), mis ei jäta mõju avaldamata projektile. Ideaalse eesmärgina on püüd teha realisatsioonist sõltumatu projekt täiesti oma kohal.

Süsteemi analüüs kasvab sujuvalt üle projekteerimiseks (*design*), projekteerimisetegevuse alguseks loetakse hetke süsteemianalüüsi etapil, kus küsimusele "mida on vaja ehitada?" lisaks hakatakse esmakordselt mõtlema ka küsimusele "kuidas seda teha?". Nagu juba öeldud, on need küsimused alateadvuses tihti omavahel seotud – oluline hetk on see, kui ka väljaspool alateadvust tekib kaks eraldi küsimust.

3.2.3 Mudeli ja tegelikkuse vahekord tarkvaratootes

Igasuguse toote tegemisel on tähtis teada, milleks seda toodet on vaja (eesmärk), millistes tingimustes toode töötab (ümbritsev keskkond), ja millised on kitsendused toote tegemisel. Ümbritseva keskkonna omadusi saab reeglina uurida vaid keskkonna mudelite abil, sama lugu on tulevase toote enda omadustega. Filosoofiliselt huvitav küsimus on millised uued tooted on tegelikkuse osad, millised tegelikkuse mudelid. Traditsioonilises teaduses on olukord aegade jooksul selgeks vaieldud – teooriad ja mudelid baseeruvad tegelikust maailmast saadud vaatluste ja nende analüüsil tekkinud hüpoteeside baasil. Mudel on tegeliku maailma eeskujul tehtud passiivne seade (abivahend), mis võimaldab (tavaliselt ligikaudu) uurida tegelikkuses toimuvaid protsesse vastavalt inimese kui eksperimentaatori ja vaatleja soovile. Tegelikkus on see, mis toimib koostöös loodusliku keskkonnaga vastavalt loodusseadustele ka ilma inimese sekkumiseta.

Rangelt võttes on suur osa tehnikateaduste ja inseneritöö tulemustest mudelid, mis toimivad vaid siis kui inimene-eksperimentaator suvatseb neid kasutada. Mudelid on tehtud looduses toimivate protsesside eeskujul, lähtudes neid juhtivatest seaduspärasustest. Mudelid on rangelt inimese kontrolli all. Aja jooksul on mudeli (algselt selge) filosoofiline kontseptsioon hägustunud – ka paljud ilmsed mudelid (näiteks tehisveehoidla, auto, jne) ei toimi ainult mudelina. Veehoidla mõjutab, lisaks oma mudelina antud ülesannetele, ka ümbritseva paikkonna kliimat ja elukeskkonda. Auto, lisaks oma mudelina antud ülesannetele, saastab loodust heitgaasidega – heitgaaside kvantiteet kasvab üle kvaliteediks ja võib hakata oluliselt mõjutama loodust. Selgub, et terve rida kontseptuaalselt selgeid mudeleid asub tegelikult hallis tsoonis – justnagu mudelid, aga ka mitte päriselt.

On inimtegevuse valdkondi, kus inimese tehtud toode väljub (võib väljuda) inimese kontrolli alt ja hakkab elama oma elu (reaalse maailma osana). Näiteks ravimitööstuse produktid on mõjutanud baktereid ja viiruseid nii, et on tekkinud täiesti uute omadustega eluvormid. Geneetiliselt modifitseeritud taimed ja loomad saavad põhimõtteliselt elada oma elu (kuigi juriidiliselt on see paljudes maades keelatud).

Traditsiooniliselt on arvutiteadus käsitlenud programmi kui mudeli mudelit (Meyer (1996)). Esmalt tehakse traditsiooniline tegelikkuse mudel (näiteks matemaatiline), mis seejärel realiseeritakse arvutis programmina (ehitatakse mudeli mudel). Metodoloogiline alus on toiminud hästi traditsioonilistes arvutikasutusvaldkondades -- seal kus arvutid on täielikult inimese kontrolli all. Kahjuks peab kogu see mõttekonstruktsioon paika seni, kuni arvutisüsteem jääb termodünaamiliselt suletuks (vt. ka alajaotus 2.2, käesolev dokument). See on ka üsna loogiline – termodünaamiliselt suletud süsteemi sees on võimalik teha asju, mis ümbritsevas keskkonnas ei oleks mõeldavad.

Reaalajatarkvara toimib termodünaamiliselt avatud süsteemis ja juba selletõttu ei saa olla käsitletav kui mudeli mudel. Lähtudes Simon (1996) sissetoodud tehismaailma teaduse mõistest, on mõistlik reaalajatarkvara (koos teda kandva arvutisüsteemiga) vaadelda tegeliku maailma objektina. Sellel filosoofilise kontseptsiooni muutusel on oluline mõju reaalajatarkvara projekteerimisele, ja eriti tarkvaratoodete verifitseerimisele. On ju selge, et mudeli mudel on ohutu -– kogu tema võimalik mõju välismaailmale kulgeb läbi inimese. Lõppotsuseid tegev inimene võtab vastutuse endale.

Enamus reaalajatarkvara tehtud otsuseid rakendatakse vahetult välismaailmale – seega ka vastutus otsuste tagajärgede eest langeb süsteemi projekteerinud ja realiseerinud meeskonnale. Arvestades ohutuskriitilisi rakendusvaldkondi (lennukite, laevade ja rongide juhtimine, keemiatehaste juhtimine, jne) võivad kehva otsuse tagajärjed olla üsna rasked. Tarkvarainseneri eetika on esialgu ainus tõsine tugi reaalajatarkvara kvaliteedi garanteerimiseks – enamus tarkvaratehnika tööriistu võimaldab üksikuid tarkvara omadusi kontrollida, kuid peaaegu ükski kontroll ei ole kohustuslik juriidilises mõttes, aga ka tööriist ei sunni formaalset verifitseerimist peale.

3.3 Süsteemi loomise planeerimine *(System-level project planning)*

Iga uue süsteemi loomise idee esimene konkretiseering tekib tulevase süsteemi projekti planeerimisel. Plaani koostamiseks on palju erinevaid meetodeid – üks viimasel ajal populaarsemaid meetodeid on nn. planeerimisseminari (*workshop planning approach*) kokkukutsumine, mis on tihedalt seotud paralleelse tootearenduse (*concurrent engineering*) ideedega. Seminaril peaksid osalema kõik uue süsteemi arendamisega seotud, või sellest huvitatud organisatsioonid. Oluline on, et kõigi projektis sisalduvate erialade spetsialistid oleksid esindatud. Näiteks, rakendusvaldkond, süsteemitehnika, tarkvara tehnika, inimfaktori asjatundjad (psühholoogid), töökindlus, katsetamine, rahastamine, projekti juhtkond.

Arutatavad küsimused algavad üldistest ja eksistentsiaalsetest – näiteks, millist toodet on kasutajale vaja? Mida selleks on vaja teha? Millal tuleb tegevusega alustada, millal peab kõik valmis olema? Mis järjekorras tuleks tegevused täita? Mida selgem ettekujutus tulevasest tootest tekib, seda konkreetsemaks lähevad ka arutatavad küsimused. Näiteks, millised on toote üleandmise/vastuvõtu kriteeriumid? Kes täidab milliseid toote tegemisega seotud ülesandeid? Kuidas vajalikke ülesandeid täita?

Seminari töö tulemusena koostatakse, sõltuvalt projekti suurusest, kas mitu eraldi plaani või üks plaan. Mõlemal juhul tuleb sisuliselt katta projekti tegemise plaanis, projekti haldusplaanis (*management plan*), riistavara plaanis, tarkvara plaanis, ohutusküsimuste plaanis, jne nõutavad küsimused. Uue süsteemi tegemise üldplaani tüüpiline sisukord näeb välja järgmine:

1. Kokkuvõte (*Document scope*)
2. Süsteemi kirjeldus (*System description*)
   Sisaldab süsteemi (uue toote), teda ümbritsevate süsteemide ja nendevaheliste liideste kirjeldused; loetleb kõik valmistamisele tulevad toote osad;
3. Lepingu kirjeldus (*Contract description*)
   Kõikide tellija, kasutaja ja tootja vaheliste sõlmitavate kokkuleppete loetelu, soovitavalt ka erimeelsuste loetelu; lepingu peamiste punktide kirjeldus.
4. Projekti täitvad asutused, grupid ja nende struktuur (*Project organisation*)
   Tellimust täitvad asutused, grupid, nende peale jäävad ülesanded, nende vaheline koostöö vorm, tööde koordineerimise ja juhtimise struktuur
5. Tööde jagunemine (*Work breakdown structure*)
   Tööde jagunemine esitatakse kahes vaates: vastavalt projektis osalevatele asutustele, gruppidele, jne; ja vastavalt tootele ning valmistatavatele osadele
6. Ressursside olemasolu ja vajadus (*Resource profile*)
   Töös osalevate asutuste, gruppide jne kaupa
7. Plaanid (*Schedules*)
   Plaan igale toote osale
   Plaanid peamistele toote osade gruppidele (*tasks*)
   Plaan kogu süsteemile (tootele)
8. Tehnilised ülevaatused (*Technical reviews (walk-throughs, audits*))
   Plaanid kõigi toodetava süsteemi osade ülevaatuseks
   Sobivuse/tagasilükkamise kriteeriumid (*pass/fail criteria*)
9. Riski haldamine (*Risk management*)

Jõudlust määravate parameetrite fikseerimine ja iseloomustus
Planeeritud ülevaated, planeeritud jõudluse hindamise ülevaated

10. Tegevuste eest vastutajad, tegevuste haldamine (*Action assignment and management*)
    Loetelu vastutajatest, tööde alustamise tingimused ja protseduurid, tööde kulgemise kontrolli protseduurid, tööde lõpetamise tingimused ja protseduurid
11. Haldamine (*Management*)
    Haldusstruktuuri kirjeldus; haldusstruktuuri toimimised reeglid ja protseduurid
12. Töö aluseks ja abiks olevad standardid, protseduurid, tööriistad, mida kasutatakse selles projektis.

Ei ole raske taibata, et peaaegu iga sisukorra alajaotuse koostamiseks läheb vajab tarkvara-inseneri abi, või vastav alajaotus sisaldab hinnalist infot tarkvara projekti koostamiseks. Sellest ka vajadus kaasata tarkvara inseneri süsteemi loomise planeerimisse päris algusest peale.

Pärast üldplaani koostamist ja heakskiitmist, sõlmitakse partnerite vahelised lepingud, koostatakse üksikute osade kohta detailsed plaanid ja hakatakse tööle.

3.3.1 Kommentaarid süsteemi loomise plaanile

Keskpäraste tarkvara inseneride üks enimlevinud küsimusi on ”Miks on vaja plaani kui midagi ei ole täpselt teada ja kõik asjad muutuvad pidevalt?” Pidevalt muutuvas ümbruses edukaks toimetulemiseks on vaja strateegilist plaani (mis muutub harva) ja taktikalist plaani (mida korrigeeritakse sageli). Vastavalt vajadusele korrigeeritavad taktikalised plaanid aitavad muudatustele õigeaegselt (s.t. minimaalsete kulutustega) reageerida.

Igal projekti juhatuse koosolekul peaks plaani ülevaatamine olema üks päevakorrapunktidest. Eelmisest koosolekust toimunud muudatuste ilmutatud analüüs, edasiliikumise analüüs, tekkinud või seni lahendamata probleemide (riskifaktorite) fikseerimine ja lahendusvariantide arutamine eelpoolnimetatud analüüsi valguses -- aitavad dünaamiliselt reageerida muudatustele ja valida häid edasiliikumise variante. Korralik planeerimine eeldab kannatlikkust, enesedistsipliini ja rasket tööd -- ükski nendest omadustest ei kuulu keskmise tarkvara inseneri tugevate omaduste hulka.

Esialgsete plaanide koostamisel peaks osalema kogu projekti meeskond, ainult nii saavutatakse töö käigus vajalik vastastikune mõistmine. Hilisem plaanide täideviimise kontroll ja vajaduse korral plaanide operatiivne muutmine on puhtalt projektijuhi ja haldusmeeskonna ülesanne.

3.3.2 Projekti jagamine töödeks (*Work breakdown structure*)

Kogu projekti realiseerimisega seotud tegevus tuleb jagada paraja suurusega töödeks, või tööde gruppideks. Paras suurus tähendab ennekõike juhtimise ja kontrolli mõttes parajat suurust. Tööde grupid tekivad suuremates projektides, kus kogu toote struktuur on hierarhiline –gruppide arv (ehk siis hierarhia ülemisel tasemel olevate tööde arv) on jällegi määratud inimese võimega hallata mitte üle seitsme erineva tööde grupi korraga.

Projekti tükeldamisel töödeks tuleks silmas pidada ka varem tehtud tarkvara osade taaskasutamise vajadust – kui on loota, et mingi töö tulemus kujuneb omaette kasutatavaks tooteks, tuleb see eraldada teistest. Ka vastupidi, kui on andmebaasis olemas toode, mida õnnestub kasutada käesolevas projektis, tuleb vastav tegevus eraldada ülejäänutest.

Igale kirjeldatud tööle (tööde grupile) eraldatakse oma ressursid (inimesed, arvutid, raha, jms), ning ta moodustab projekti omaette administreeritava osa. Sõltuvalt süsteemi funktsionaalsetest nõuetest ja muudest kitsendustes võib tööde tegemise järjekord olla osaliselt või täielikult ette määratud. Sularaha vood, inimeste ja arvutite vajadus sõltuvad oluliselt tööde järjekorrast. Seetõttu võib töödeks jagamine, peale terve mõistuse, pisut sõltuda ka kasutaja nõuetega ning rahastustingimustega fikseeritud kitsendustest. Ei ole üldse võimatu, et raamatupidamislikest kitsendustest tingituna on ühel projektil kaks erinevat töödeks jagamise struktuuri – üks raamatupidamisele, teine tegeliku töö ning projekti juhtimise jaoks (et tagada objektiivse info saamist projekti hetkeseisu kohta).

Korraliku projektijuhi eesmärk on saavutada projekti selline töödeks jagamine, et igal hetkel oleks kõigi projekti osade kohta teada:

- nende peale kulutatud raha;
- nende olek võrreldes plaaniga;
- iga üksiku hallatava programmistide grupi integraalne jõudlus (veel parem oleks iga üksiku inimese jõudlus).

Samal ajal peab minimiseerima tööde kulusid, mis on seotud projekti aruandlusega, kuluarvestusega ja üldjuhtimisega.

Tööde kirjeldused (*task descriptions*) on kasulik standardiseerida. Tööde kirjelduse tüüpsisukord võiks välja näha järgmiselt:

- töö nimetus (*task identifier*)
- üldkirjeldus (*general description*)
- töö käivitamiseks vajalikud lähteandmed ja muud sisendandmed, näiteks kuupäev ja kellaeg, muutujate nimed koos lubatud väärtuste muutumise piirkondadega, sisendmuutujate väärtuste saamise kohad
- töö tulemused, nende tootmise tegelik ja planeeritud aeg, tulemuste tarbijad
- töö arengut hindavate ülevaatuste kava (aeg, koht, osavõtjad, kontrollitavad omadused, vastuvõtmise ja/või tagasilükkamise kriteeriumid)
- töö lõpetamise tingimused
- vajaliku ressursid (inimesed, riist- ja tarkvara, muud)
- Gantt’i diagramm ja/või PERT’i diagramm

Gantt’i ja/või PERT’i diagramm on kasutatav kahel tasemel – projekti töödeks jagunemise kirjeldamiseks, samuti ka iga töö sisese tükelduse kirjeldamiseks. Projekti juhtimise tasemel tegeldakse siiski kogu tükeldusega projekti üksikuteks töödeks.

Kommentaar:

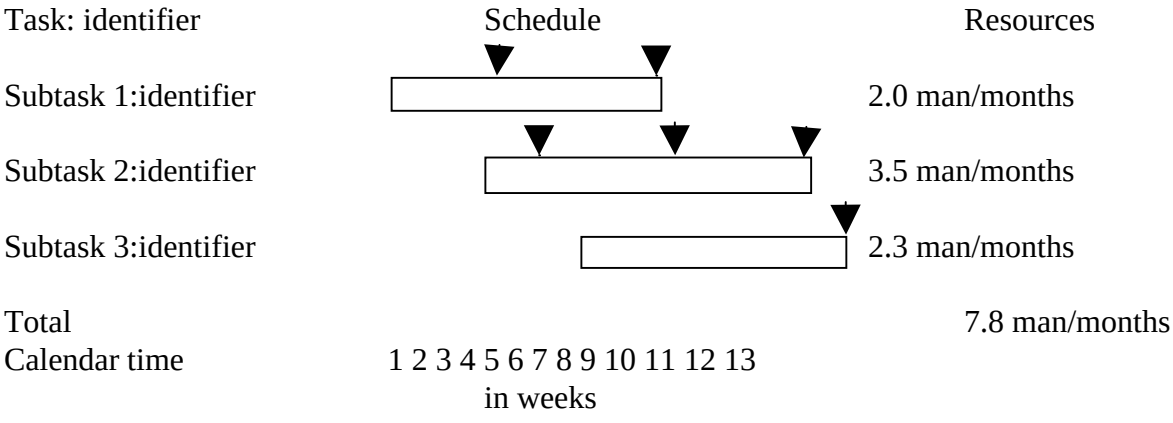
Nii Gantt'i kui ka PERT'i diagrammide koostamist toetab standardtarkvara – näiteks, Microsoft Office'i eraldiostetav programm *Microsoft Project.*

Gantti diagramm (vt. näidist joonisel 3.1) koondab andmed üksikute tööde alguse, lõpu, ja vahepealsete ülevaatuste kohta, ning fikseerib ka tegijad ja töömahud. Tihti koostatakse eraldi Gantti diagrammid tööde, ressursside kasutamise ja inimeste jagunemise kohta.

PERT'i diagramm sarnaneb andmevoo mudelile ja/või tihti kasutatavale tegevuste diagrammile, mis loetlevad vajalikud tegevused ja näitavad nendevahelised põhjuslikud seosed. Sageli lisatakse andmevoo mudelit või tegevusi kujutavale diagrammile ka tegevuse alguse ajad ning arvestuslikud (ja tegelikud) kestused, mis võimaldab erilise kontrolli alla võtta kriitilised teed. PERT'i diagrammis lisatakse iga tegevuse kestuse kohta pessimistlik, tõepärane ja optimistlik hinnang. Selle tagajärjel tekib mitte üks kriitiline tee, vaid palju kriitilisi teid (sõltuvalt üksikute tegevuste jaoks konkreetselt valitud kestuse hinnangu tüübist).

Detailsem kirjeldus koos näidetega on toodud raamatus **I.Sommerville "Software Engineering" Addison-Wesley, 1992, 649 pp, page 503-507** (Sommerville uses the "term" *activity network* for PERT, and *activity bar* for Gantt diagram).

Vt. ka K.A.Cori (1985) "Fundamentals of Master Scheduling for the Project Manager" Project Management Journal, June, 78-89. Reprinted in IEEE Tutorial "Software Engineering Management" Ed. R.H.Thayer, TTÜ Raamatukogu kataloog VB-63387.



Black arrows at the top of the timelines indicate either deliveries, milestones, or checkpoints

Joonis 3.1 Ülesande tükeldamine ja kirjeldamine Gantti diagrammina
(© Behforooz & Hudson, 1996)

3.3.3 Ressursside vajadus

Ressurssideks loetakse vajalik riistavara, tarkvaratehnika tööriistad, inimesed ja paljudel juhtudel ka ruumide üür ning firma arenduskulud. Esimeses lähenduses saab ressursside vajaduse hinnangu taandada kogu projekti maksumuse hindamiseks. Üks põhjalikemaid metoodikaid on välja töötatud Boehm (1984) poolt (B.Boehm (1984) "Software Engineering Economics", IEEE Transactions on Software Engineering, vol. SE-10, no.1, 4-21). Tarkvaratoote maksumuse arvutamine eeldab varasemat kogemust ja tarkvara meetrika (*software metrics*, tutvustatakse edaspidi käesolevas tekstis) vahendite kasutamise kogemust, ning võimalust. (vt. ka Võimete Küpsuse Mudel (CMM), käesoleva dokumendi alajaotus 3.5).

Projekti maksumuse hinnangute täpsus sõltub projekti kohta teadaoleva informatsiooni kogusest ja usaldusväärsusest (arenguetapist). Näiteks projekti mõistlikkuse (feasibility) hindamise etapil on loomulik kuni neljakordne allahindamine, või neljakordne ülehindamine. Spetsifitseerimisetapil hinnates peaks maksumuse täpsus jääma piiridesse 0.5-1.5 tegelikku maksumust, detailse projekteerimise etapil saab projekti maksumust hinnata täpsusega vahemikus 0.75 – 1.25 tegelikust maksumusest.

Behforooz and Hudson (1996) järgi jagunevad kulutused süsteemi arengu etappide vahel järgmiselt:

- 20% -- tehniline ettevalmistus (*engineering*), s.t. planeerimine, nõuete ja funktsioonide spetsifitseerimine, jõudluse nõuded ja jõudluse hindamise kriteeriumid, jne kuni kasutaja nõuete spetsifikatsiooni koostamiseni
- 20% -- projekteerimine (*design*), kõrgtasemega projekt, detailne projekt, nende analüüs
- 17% -- kodeerimine ja moodulite katsetamine (*code and module test*)
- 43% -- süsteemsed katsetused ja integreerimine (*system test and integration*)

Lisaks soovitava Behforooz ja Hudson (1996) projekti maksumuse hinnang tellida kahelt grupilt (inimeselt) , kes kasutavad erinevaid metoodikaid. Ka soovitatakse tarkvara meeskonnal endale aru anda, et maksumuse hinnang ja läbirääkimistel saavutatud rahaline kate on sageli kaks eri asja. Kui need kaks erinevad väga järsult, jääb alati äärmine võimalus – s.o. loobuda projekti tegemisest.

Peamised tarkvara maksumuse hindamise meetodid on Boehm (1984) järgi:

- Algoritmilised mudelid (*algorithmic models*) – töötatakse välja algoritmid, mis tarkvara meetrika tulemuste alusel arvutavad, näiteks muutujate arvu järgi või funktsioonipunktide alusel, eeldatava üldkulude summa
- Ekspert hinnangud (*expert judgement*) – konsulteeritakse ühe või enama eksperdiga, tulemus saadakse näiteks Delphi tehnikat kasutades (üks ekspertide arvamuste järgi koondhinnangu saamise meetoditest, vt. näiteks, Hicks and Gullet (1981)).

- Analoogsete projektidega võrreldes (*analogy*) – ühe või kahe analoogse projekti tegelikku maksumust ja tarkvara meetrika alusel saadud karakteristikuid võrreldes tuletatakse hinnang käesolevale projektile
- Parkinson'i meetod (*Parkinson*) – rakendatakse Parkinsoni printsiipi "tööks kuluvad ära kõik olemasolevad ressursid", s.t. projekti maksumus püütakse suruda tellija poolt etteantud summa sisse
- Võidupakkumise hind (*Price-to-win*) – projekti maksumuseks pannakse hind, mis loodetavasti toob konkursil võidu (sageli muudetakse mitte hinda, vaid projekti plaani nii, et konkurssi võita)
- Langev meetod (*Top-down*) – projekti maksumust hinnatakse lõpptoote üldiste parameetrite alusel, seejärel määratakse projekti üksikosade maksumus
- Tõusev meetod (*Bottom-up*) – iga projekti kuuluva tarkvara komponendi maksumus hinnatakse eraldi, kogu projekti maksumus saadakse summeerimisel ja üldkulude lisamisel.

Ülaltoodud meetodeid iseloomustab Boehm järgmiselt:
- Ükski alternatiividest ei ole ühtlaselt parem teistest
- Parkinsoni meetod ja võidupakkumise hind ei anna rahuldavat projekti maksumust – neid ei soovitada kasutada tegeliku eelarve tegemisel
- Praktikas tuleks kasutada mitut erinevat meetodit, võrrelda neid ja hinnata projekti maksumust korduvalt (erinevatel elutsükli etappidel).

Inimressursside vajadus (*Staffing*)

Inimressursside mõjust tarkvaratoodete tegemisele on oma klassikalises raamatus (F.P.Brooks, (1975)) väitnud:
- Tänu tarkvara tootmise keerukusele ja tegijate vahelise infovahetuse olulisusele võib programmistide arvu suurendamine pikendada toote valmimise tähtaegu; igal projektil on oma optimaalne arv tegijaid;
- Tarkvaraprojekti keerukuse (koodimahu, funktsiooni punktide, jne) lineaarse kasvu korral kasvab projekti valmimiseks vajalik inimkuude arv eksponentsiaalselt.

F.P.Brooks, Jr (1975) "The Mythical Man-Month", Addison-Wesley Publishing Co., Massachusetts, katkendid raamatust on trükitud IEEE Tutorial on Software Engineering Project Management, Ed. R.H.Thayer, TTÜ Raamatukogu kataloogi no. VB-63387.

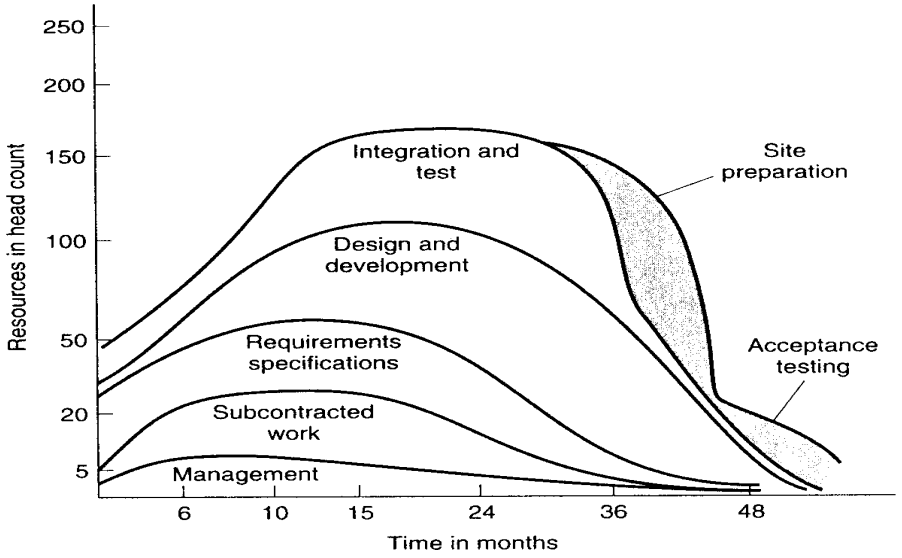
Inimkuude eksponentsiaalse kasvu kohta puuduvad 1990-ndatel katselised kinnitused – vahepeal on üsna laialdaselt kasutusele võetud tarkvaratehnika tööriistu ja märgatavalt automatiseeritud grupiviisilist tarkvaraarendust, mis peaks pisut muutma keerukuse ja inimkuude sõltuvust.

Huvitav, ja ikka veel kaasaegne fakt samast raamatust – korralikus organisatsioonis avastati, et kõik äärmise hoolega tehtud projekti plaanid võtavad tegelikkuses enamvähem kaks korda rohkem aega. Uuringud näitasid, et *projekti meeskonna liikmed kulutavad päevas vaid 50% tööajast projektiga vahetult seotud tööde jaoks*. Ülejäänud aeg kulus arvutite ja tarkvara probleemide kõrvaldamiseks, kõrge prioriteediga projektiga, kuid mitteseotud tööde tegemiseks,

koosolekuteks, mitmesuguseks bürokraatiaks, haigusteks, isiklike asjade ajamiseks, jne. **Kontrollige oma meeskonna tööajakasutamist ja arvestage tulemusi planeerimisel !**

Programmistid on isiksused – erinevate programmistide töö tootlikkus võib erineda rohkem kui 10 korda. Selletõttu on oluline hinnata iga projekti meeskonna liikme isiklikku tootlikkust. Lisaks sellele on igal programmistil oma isikupärane stiil, mida mitte kõik ei ole nõus meeskonnatöö nõuetele vastavaks muutma. Õnneks pehmendavad tarkvaratehnika tööriistad pisut individualismi probleemi.

Väiksemates firmades teeb projekti juhtimise raskeks see, et erinevatel elutsükli etappidel on vaja erinevat arvu ja erineva oskustega inimesi. Sageli tuleb hea spetsialist sisse osta nädalaks (või mõneks päevaks). Inimressursside vajaduse plaanis tuuakse ära integraalne inimeste vajadus kuude kaupa (vt. näiteks joonis 3.2)



Joonis 3.2 Integraalne inimeste vajadus kogu projekti vältel (© Behforooz & Hudson, 1996)

Integraalsele inimressursside plaanile lisandub detailne inimressursside plaan, mis näitab ära inimeste jagunemise organisatsioonis (grupid, osakonnad, firmad) koos iga inimese (antud projekti jaoks) töötamise ajaga (kuupäevast kuupäevani), vajalike oskuste loeteluga, töö alustamise ja lõpetamise tingimustega jne.

Inimressursside juures on oluline nende töö korraldamine, koostöö organiseerimine, kontroll ja juhtimine. Inimeste juhtimine ja kontroll on alati keeruline, mõned tarkvaraprojektides kasutatud võtted on kirjeldatud Sommerville (1992), lk.31-38.

3.3.4 Projekti perioodilised ülevaatused (*Technical reviews*)

Projekti ülevaatused on tavaliselt seotud projekti plaanis fikseeritud verstapostidega (*milestones*), kuid üsna traditsioonilised on:

- Tarkvara nõuete spetsifikatsiooni ülevaatus (*software requirements review*)
- Kõrgtasemega projekti ülevaatus (*preliminary design review, high-level design review*)
- Detailprojekti ülevaatus (*critical design review, low-level design review*)
- Lõppkatsetusteks valmisoleku ülevaatus (system test readiness review)

Täiendavaid ülevaatuseid korraldatakse vastavalt vajadusele ja projekti täitmisel kujunenud olukorrale. Projekti tehniline ülevaatus eeldab põhjalikku ettevalmistust (arvestuslik töökulu on 100-150 inimtundi ülevaatuse kohta). Ülevaatuseks valmistumisel tuleb ettekanded aegsasti kirjutada, kiled valmistada, läbi viia prooviesinemised (ajastamise ja sisu kontrollimiseks ning kooskõlastamiseks). Kui projekt võimaldab juba katsetusi, tuleb need eelnevalt läbi teha ja kontrollida, et katsetustest saadud sõnum on tõesti selline mida eeldab tehtav toode ja mida projekti juht tahab ülevaatajatele edastada. Kõik vajalikud dokumendid tuleb eelnevalt trükkida ja paljundada.

Esitatud dokumentide, ettekannete ja demode alusel tellitakse tavaliselt kahelt kolmelt inimeselt kriitiline ülevaade projekti kulgemisest ja soovitused töö jätkamiseks. Retsensentideks ja/või nõuandjateks kutsutakse erapooletud inimesed väljastpoolt projekti meeskonda (ja asutust). Ülevaatuste eesmärk on hinnata tehtud töö kvaliteeti, kontrollida selle vastavust kasutaja nõuetele ja kokkulepitud standarditele, kontrollida projekti ressursside kasutamist, projekti meeskonna poolt tehtud riskihinnangute paikapidavust ja riski vähendamise meetmete asjakohasust. Ülevaatusel antakse soovitusi projekti jätkamise teede kohta (vahel soovitatakse ka projekt pooleli jätta).

Tehnilistesse ülevaatustesse ja sellega seonduvasse suhtutakse Eesti võrdlemisi üleolevalt -- poole sajandi vanune potjomkinluse sissekasvatamine on andnud häid tulemusi. Isegi firmad, kes neid enamvähem regulaarselt korraldavad, püüavad oma inimestega läbi ajada. Finantse jagavad ametnikud, aga ka tegelikud süsteemi loojad ei suuda uskuda, et erapooletud eksperdid keda välismaal sageli kutsutakse projekti ülevaatustele, võivad olla paremad hindajad kui konkreetse töö tegijad. Erapooletu kõrvaltvaataja tunneb hooletult tehtud, või tegemata jäänud töö kohe ära ja oskab sageli anda head nõu projekti paremaks jätkamiseks. Eestis ei ole kombeks -- praegu veel -- projekti tehnilistele ülevaatustele erapooletuid eksperte kutsuda (ilmselt lootuses raha kokku hoida). Mitme välisriigi kogemus näitab siiski, et erapooletute ekspertide kasutamisel saadav kasu ületab mitme suurusjärgu võrra nende palkamiseks tehtud kulutused.

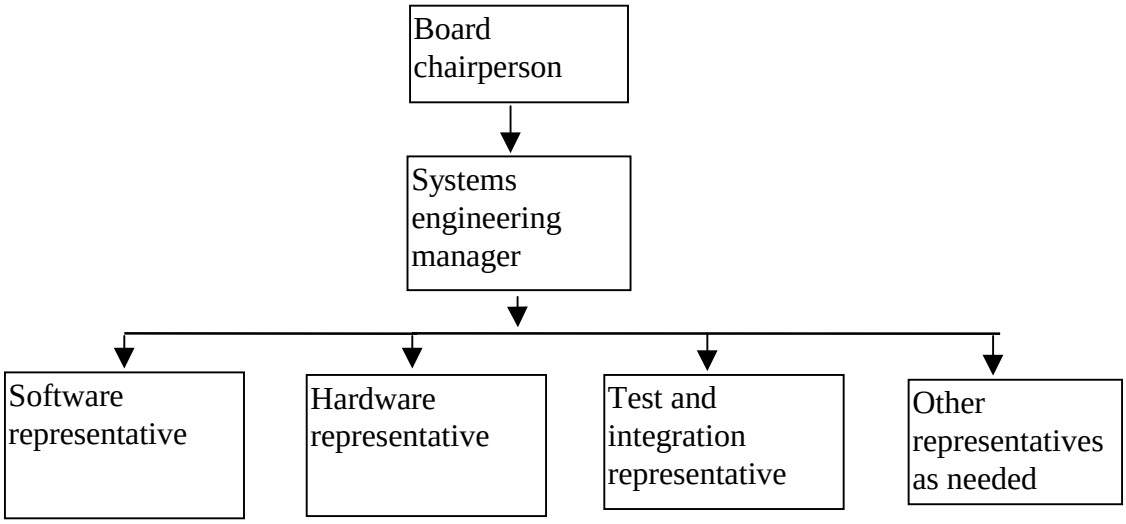
Teine põhjus erapooletute ekspertide mittekasutamiseks võib olla Eestis veel tugevalt juurdunud sotsialistlik/nõukogulik varguse sündroom. Kardetakse ärisaladuste ilmsiks tulemist, ei usuta ausate inimeste olemasollu, ei usuta vaikimislepingute pidavust. Kahjuks on nii ühiskonna kui ka õigusmõistmisorganite eetikaga lood kehvavõitu -- varastatakse tõepoolest kõike huvipakkuvat, aga karistatakse sümboolselt ja väga valikuliselt. Näiteks ei ole altkäemaksu, tööalaste saladuste, lepingulistest lubadustest mittekinnipidamise eest veel kedagi tõeliselt karistatud. Keskmise inimese ja ühiskonna eetika pärineb ikka veel nõukogu impeeriumi aegadest. Ei ole saladus, et paljud nõukogude impeeriumis toiminud salajased ettevõtted kasutasid salastatust ainult selleks, et varjata oma äärmiselt väheefektiivset tööd ja lootusetult vananenud tehnoloogiat -- sellised

nähtused on paratamatud suletud riikliku majandussüsteemi juures. Vaatamata Eesti mõningatele edusammudele avatud majandussüsteemi poole liikumises, ei ole suur osa ametnike korpusest muutunud ja nõukogudeaegne suhtumine on korralikus elujõus. Seetõttu soovitan välismaa ettevõtetega tööalasel suhtlemisel olla tähelepanelik – mitte kõigis maades ei ole eetikaga lood nii kehvad.

3.3.5 Projekti muudatuste haldamine (*Change control*)

Muudatused on iga projekti käigus paratamatud. Muudatustest tingitud erinevate projektiversioonide segunemine võib praktikas põhjustada äärmiselt palju pahandust. Selliste segaduste vältimiseks on tähtis sisse viia muudatuste tegemise lubade andmise, muudatuste tegemise, tehtud muudatuste kohta info levitamise ja muudatustele eelnenud versioonide arhiveerimise täpne kord -- ning sellest korrast rangelt kinni pidada.

Vahel on muudatuse tegemine kasulikum edasi lükata kuni süsteemi järgmise versiooni tegemiseni, paljudel juhtudel on edasilükkamine lubamatu. Iga üksiku muudatuse vajaduse peab hindama spetsiaalne selleks määratud isik (tavaliselt projekti juht), kuid iga üksiku muudatuse kõigi kõrval-mõjude hindamiseks soovitatakse luua spetsiaalne administratiivne struktuur (vt. joon. 3.3). Kõik muudatused tuleb kindlasti dokumenteerida. Igas dokumendis peaksid muudatused, mis on tehtud võrreldes eelmiste selle dokumendi versioonidega, olema dokumendi kokkuvõttes ilmutatult loetletud.



Joonis 3.3 Süsteemi konfiguratsiooni haldamise komitee struktuur
Formal configuration control board organisation(© Behforooz & Hudson, 1996)

3.3.6 Tarkvara loomise plaan (*software development plan*)

Tarkvara loomise plaani saab lõplikult kokku panna alles pärast süsteemi nõuete spetsifitseerimist ja sellest lähtuva tarkvara nõuete spetsifikatsiooni koostamist. Seejärel, kasutades käesoleva teksti kolmanda osa teise peatüki eelnevates alajaotustes kirjeldatud tulemusi koostatakse lõplik tarkvara loomise plaan. Plaan peab sisaldama realistliku tarkvara toote maksumuse (ei tohi olla tellijale liiga kallis, ei tohi olla tegijale liiga odav), realistliku valmimistähtaja (mitte liiga pikk tellija jaoks ja mitte liiga lühike tegija jaoks).

Tarkvara loomise plaani peamisteks komponentideks üldiste haldusosade ja töökorralduslike osade kõrval on:

- projektiga valmivad tooted (mis lähevad tellijale),
- ülevaatuste ja etapi lõppude kuupäevad (verstapostid), millal tellija näeb tulevast toodet või saab selle kätte, ja
- eelarve

Tarkvara loomise plaani sisukord on sarnane süsteemi loomis plaani sisukorrale, tähelepanu on fokuseeritud tarkvaraga seotud probleemidele. Tüüpiline tarkvara loomise plaani sisu peab katma järgmised teemad:

- Sissejuhatus
- Ressursside vajaduse ja realiseerimise ajakava hinnangud
- Tarkvaraprojekti täitjate grupid ja asutused, nendevahelised seosed ja vajalik inimeste arv ning nende kvalifikatsiooni nõuded
- Tööde jagunemine inimeste vahel, tööde grupid, kuluhinnangud
- Projekti tehniline juhtimine ja kontrolliprotseduurid
- Standardid ja soovituslikud (või lubatavad) protseduurid
- Ülevaated, inspektsioonid, arutelud -- osavõtjad, ajakava, tarkvara osad
- Arenduskeskkonnad, mida soovitatakse (lubatakse) kasutada
- Dokumentatsiooni loetelu ja vormistamise nõuded, muutuste sisseviimise protseduurid
- Verifitseerimine ja valideerimine
- Toote hooldus ja kasutamine
- Inimfaktorid (kasutajad, projekti täitjad)
- Toote kasutajale toimetamine, installeerimine, vastuvõtu-üleandmise protseduur
- Lisad ja viited täiendavatele allikatele.

3.4 Süsteemi loomise elutsükkel

Süsteemi loomise elutsükkel (*system development life cycle, SDLC*) on süsteemi loomisel läbikäidavate etappide ja tegevuste süstemaatiline esitus, tavaliselt nn. elutsükli mudeli kujul. Elutsükli mudel tekkis ennekõike projektijuhtide abistamiseks projekti haldamisega seotud tegevustest objektiivse pildi saamisel, projekti haldamise otsuste tegemisel, tehniliste projektotsuste tegemisel (alternatiivide valikul), riskide minimiseerimiseks jne.

Süsteemide elutsükli mudel osutus heaks abivahendiks, ta võeti üle tarkvaratehnikasse, tarkvara projektijuhtide töövahendina. Hiljem muutus elutsükli mudel oluliseks tarkvaratehnika meetodite väljatöötamise vajaduste generaatoriks ja süstematiseerijaks.

Seni puudub üks ja üldtunnustatud elutsükli mudel -- Behforooz & Hudson (1996) toovad näiteid suuremates firmades kasutatavatest elutsükli mudelitest. Näiteks, *US Department of Defence (DOD*), *National Aeronautics and Space Administration (NASA*), ja teised. Sageli tehakse vahet haldusinfosüsteemi ja reaalajasüsteemi (viimast nimetatakse vahel ka lihtsalt tehnilisele rakendusele orienteeritud infosüsteemiks) elutsüklimudelitel. Administraatori vaatepunktist on haldus- ja tehnilise süsteemi elutsükli mudelid üsna sarnased, lähemal sisulisel vaatlusel selgub siiski terve rida põhimõttelisi erinevusi (näiteks, haldussüsteemid on mudeli mudelid, paljud tehnilised süsteemid on tegelikkuse osa). Kui minna sügavamale süsteemi loomise detailidesse, siis on haldusinfosüsteemides ja tehnikale orienteeritud süsteemides suhteliselt tugevalt erinevad riskifaktorid, keerukuse allikad, keerukuse tase ja erinevatel elutsükli etappidel tarvilik verifitseerimise detailsus.

Süsteemide loomise elutsükli detailidega saab tutvuda iseseisvalt, kasutades näiteks Behforooz & Hudson (1996) raamatut (ja selles antud viiteid detailsematele allikatele). Selles loengusarjas tuleb veidi rohkem juttu süsteemi elutsükli ühest osast -- tarkvara elutsüklist (*software development life-cycle*).

3.4.1 Tarkvara elutsükkel (software development lifecycle)

Tarkvara elutsükkel on osa süsteemi elutsüklist ja, abstraktselt vaadates, koosneb põhimõtteliselt samadest osadest, mis süsteemi elutsükkelgi. Erinevad tarkvara elutsükli interpretatsioonid on esitatavad erinevate mudelite kujul.

Sõltumata konkreetses firmas kasutatavast elutsükli mudelist, läbib tarkvara oma loomise käigus kuus peamist etappi. Need kuus etappi moodustavad nn. **generatiivse elutsükli** (*generic software development life cycle*).

Esimene etapp – nõuete formuleerimine ja analüüs (*requirements specification and analysis*).

Lähteinfo. Selle etapi lähtematerjaliks on kirjalikult vormistatud **süsteemi nõuete spetsifikatsioon** (*systems requirements specification*), koos tarkvaras realiseeritava osa esiletõstmisega ja kogutud täiendavate andmetega (mis ei pruugi sisalduda nõuete spetsifikatsioonis) tarkvarasüsteemiga lahendatava probleemi, rakendusobjekti, tema töökorralduse ja töökeskkonna kohta. Analüüs toimub eraldi riist- ja tarkvarale esitatavate nõuete kohta.

Tulemus. Nõuete formuleerimise ja analüüsi etapi tulemuseks on verifitseeritud ja valideeritud tarkvara nõuete spetsifikatsioon, süsteemi ja tarkvara-alamsüsteemi arhitektuuri esialgne ülevaade. Ka tarkvara loomise plaani lõplik versioon peab valmima selle etapi lõpuks.

Tarkvara nõuete spetsifikatsioon tekib süsteemi analüüsi käigus ja on sageli vaadeldav kui süsteemi-inseneri ja tarkvara-inseneri vaheline leping. Reaalsuses tuleb tarkvara nõuete spetsifikatsioon koostada süsteemi- ja tarkvara-inseneride tihedas koostöös.

Tarkvara nõuete spetsifikatsiooni sisukord, mis rahuldab IEEE (*Institute of Electrical and Electronic Engineers*) ja ANSI (*American National Standards Institute*) standardeid peaks sisaldama järgmisi andmeid:

1. Sissejuhatus
    - 1.1 Eesmärk (*Purpose*)
    - 1.2 Tööpiirkond (*Scope*)
    - 1.3 Definitsioonid ja lühendite selgitused (*Definitions, Acronyms, Abbreviations*)
    - 1.4 Viited teistele projektiga seotud dokumentidele (*References*)
    - 1.5 Ülevaade dokumendist (*Overview*)
2. Üldkirjeldus (*General Description*)
    - 2.1 Produkti loomine ja kasutamine (*Product perspective*)
    - 2.2 Funktsioonid (*Product functions*)
    - 2.3 Kasutaja iseloomustus (*User characteristics*)
    - 2.4 Üldised kitsendused (*General constraints*)
    - 2.5 Tootele rakenduvad eeldused ja süsteemi osade omavahelised sõltuvused (*Assumptions and dependencies*)
3. Konkreetsed täidetavad funktsioonid (*Special Functions*)
    - 3.1 Funktsionaalsed nõuded (*Functional requirements*)
        - 3.1.1 Funktsionaalne nõue nr. 1
            - 3.1.1.1 Sissejuhatus (*Introduction*)
            - 3.1.1.2 Sisendid (*Inputs*)
            - 3.1.1.3 Töötlus (*Processing*)
            - 3.1.1.4 Väljundid (*Outputs)*
        - 3.1.2 Funktsionaalne nõue nr.2
            - jne....
            - ....
    - 3.2 Nõuded liidestele (*Interface requirements*)
        - 3.2.1 Kasutajaliidesed (*User Interfaces*)
        - 3.2.2 Riistvara liidesed (*Hardware Interfaces*)
        - 3.2.3 Tarkvara liidesed (*Software Interfaces*)
        - 3.2.4 Võrguliidesed (*Communication Interfaces*)
        - 3.2.5 Protsessiliidesed (*Instrumentation interface*) -- ainult reaalajasüsteemides
    - 3.3 Nõuded jõudlusele (*Performance requirements*)
    - 3.4 Projekti kitsendused
        - 3.4.1 Vastavus standarditele (*Standard Compliance*)
        - 3.4.2 Piirangud riistvarale (*Hardware limitations*)
            - ..jne.....
            - .....
    - 3.5 Nõutavad tarkvara omadused (*Attributes)*
        - 3.5.1 Turvalisus (*Security*)

3.5.2 Hooldatavus (*Maintainability*)
...jne.....
.....
3.6 Muud nõuded (Other requirements)
3.6.1 Andmebaas (database)
3.6.2 Võimalikud teenused (Operations)
3.6.3 Kasutajale kohandamine (Site adaptation)
...jne.....
.....
3.7 Katsetamine (Testing)
3.7.1 Osade testimine
3.7.2 Integreerimistestid
3.7.3 Vastuvõtu /üleandmise katsetused.

Teine etapp – Projekteerimine *(design specification)*

Lähteinfo: Tarkvara loomise plaan, ilmutatud kujul antud tarkvara nõuete spetsifikatsioon, millele on lisatud ilmutamata kujul antud nõuded, esialgselt kokkulepitud tarkvara arhitektuur.

Tulemus: Projekteerimise faasi tulemuseks on verifitseeritud ja kõigi osapoolte poolt vastuvõetud tarkvara arhitektuur, verifitseeritud ja katsetatud tarkvara detailne projekt (tavaliselt pseudokoodi tasemel). Lisaks eelpoolnimetatutele peab tulemus sisaldama ka algoritmide kirjeldusi, andmestruktuure, andmebaasi projekti, andmebaasi pöördumisprotseduure, kasutajaliidest, tarkvara osade katsetamise protseduure, jne.

Tarkvara detailse projekti koostamine on täielikult tarkvara inseneri vastutusel. Projekti sisust peaks selguma:

- hierarhiline tarkvara arhitektuur – kihiline struktuur, mis jagab kogu tarkvara loogilisteks osadeks; madalamad kihid on enamasti ülemistel kihtidel olevate komponentide detailsed kirjeldused; hajussüsteemide korral on tihti vaja eristada loogilist arhitektuuri ja füüsilist arhitektuuri – viimane kirjeldab loogilise struktuuri teisendust arvutikobara ja juhitava objekti konkreetsele konfiguratsioonile; sageli võib üks loogiline osa olla füüsiliselt dubleeritud mitmes erinevas ruumiosas paiknevas seadmes;
- andmestruktuuride, juhtimisstruktuuride, algoritmide ja nendevaheliste liideste täpsed ja detailsed kirjeldused;
- selgitus ja põhjendus, kuidas on detailses projektis realiseeritud tarkvara nõuete spetsifikatsioonis loetletud eesmärgid ja ülesanded -- ilmutatult iga üksiku nõude kohta
- kõik ülalöeldu on esitatud sellisel kujul, et asjast huvitatud isikud (kasutajad, tellijad, kontrollijad) oleksid võimelised kirjeldustest aru saama; detailne projekt peab olema läbinud tehnilise ülevaatuse koos detailse analüüsi ja võimaluse/vajaduse korral ka verifitseerimisega.

Kolmas etapp – Kodeerimine ja üksikosade katsetamine *(coding and unit testing)*

Lähteinfo – tarkvara projekt, tarkvara arhitektuur, üksikosade katsetamise protseduurid ja osade vastuvõtu nõuded, kogu süsteemi katsetamise protseduurid ja süsteemi vastuvõtu nõuded

Tulemus – Katsetatud, dokumenteeritud ja formaalselt vastuvõetud programmi moodulid, koos vajaliku dokumentatsiooniga (kaasa arvatud katsetuste tulemused).

Neljas etapp – Katsetamine ja osade ühendamine tervikuks (*software integration)*

Lähteinfo – Eelmisel etapil kodeeritud, katsetatud ja formaalselt vastuvõetud moodulid, moodulite vastuvõtukatsetuste protseduurid ja kriteeriumid, süsteemi vastuvõtukatsetuste protseduurid ja kriteeriumid.
Tulemus – Formaalselt vastu võetud (sertifitseeritud) koostoimiv moodulite kogum, millest neljanda etapi lõpuks tekib vastuvõtu katsetusteks valmis tarkvarasüsteem koos vajalike dokumentidega.

Viies etapp – Vastuvõtu katsetused *(acceptance test)*

Lähteinfo – Katsetatud ja süsteemiks integreeritud tarkvara koos dokumentatsiooniga, toote loomise käigus lõpliku kuju saanud vastuvõtu katsetuste kirjeldused (protseduurid, vastuvõtu kriteeriumid).
Tulemus – Otsus, kas võtta süsteem vastu, või nõuda täiendavaid töid. Dokument, kus on iga üksik vastuvõtuga seotud katse ja katse tulemused põhjalikult dokumenteeritud.

Vastuvõtu katsed tuleb korrektselt dokumenteerida, kuna katsete tulemusi kasutatakse tihti kohtuvaidluste lahendamisel (eriti negatiivse vastuvõtuotsuse korral), aga ka järgmisel etapil juhtuda võivate avariide uurimisel.

Kuues etapp - Kasutamine ja hooldus (*maintenance*)

Mitmed autorid käsitlevad hooldusetappi eraldi süsteemi loomise elutsüklist. Suurem osa autoreid peab hooldusetappi siiski tarkvara elutsükli oluliseks osaks. Kuigi sellel etapil tarkvarale tehtud kulutuste osa väheneb (vastavalt tarkvaratehnika meetodite ja tööriistade kasutuselevõtule), on see etapp siiski kõige kulukam -- keskmiselt peaaegu 60% kõigist elutsükli jooksul tarkvarale tehtud kulutustest. Tarkvaratehnika seisukohast olulised tegevused sellel etapil on seotud tarkvarasüsteemi adapteerimisega töö käigus muutunud ümbritseva keskkonna tingimustele ja kasutaja nõuetele. Uuringud on näidanud, et suurem osa tarkvarasüsteemidest visatakse ära mitte moraalse vananemise tõttu, vaid nende modifitseerimise (adapteerimise) järsu kallinemise tõttu – aja jooksul koguneb muutuseid väga palju, tarkvara struktuur ähmastub ja ei vasta enam dokumentides toodule. Tarkvara muutmise hinna järsk tõus on põhjustatud täiendavast tööst tegeliku tarkvara struktuuri väljaselgitamisel.

3.4.2 Tarkvara elutsükli mudelid

Elutsükli mudel on eelmises punktis kirjeldatud generatiivse protsessi poolformaalne (graafiline) kirjeldus, millele tavaliselt lisanduvad mudeli osade kirjeldamise automatiseeritud vahendid. Suur osa kasutusel olevatest elutsükli mudelitest on nn. kaskaadmudeli (waterfall model) variatsioonid, mida sageli on laiendatud prototüpiseerimise ideedega. Järgnevalt antakse lühiülevaade mõnedest

enamlevinud mudelitest, täpsem info on saadaval Behforooz & Hudson (1996) ja Schach (1996) raamatutest.

Ajalooliselt esimeseks *de facto* tarkvara elutsükli mudeliks peetakse nn “ehita-ja-paranda“ tüüpi mudelit (Schach nimetab seda “build-and-fix” mudeliks, Boehm aga “code-and-fix” mudeliks). Selle kohaselt ehitati programm ilma spetsifikatsiooni ja projektita, pärast valmimist hakatakse avastatud vigu kõrvaldama. Sisuliselt taandub tarkvara tegemine terve seeria prototüüpide loomisele, kuid ilma süstemaatilise arendus- ja mõttetööta – parandatakse vaid juhuslikult avastatud vigu ja puudujääke.

Kogemus näitab, et tarkvara muudatuste hind kasvab suhteliselt lineaarselt alates spetsifitseerimise etapist kuni projekteerimisetapi lõpuni. Kohe pärast kodeerimist toimub muudatuste hinnas hüpe – iga muudatus, mis viiakse sisse juba kodeeritud programmi on seitse korda kallim kui muutmine projekteerimise etapil (vt. Schach (1996), lk. 13). “Ehita-ja-paranda” tüüpi mudel on ajalooline relikt, mis kuulub tarkvaratehnika eelsesse aega, nn. metsprogrammeerijate ajastusse. 1970-ndast kuni 1980-ndate alguseni oli praktiliselt ainuvalitsev tarkvara elutsükli kaskaadmudel.

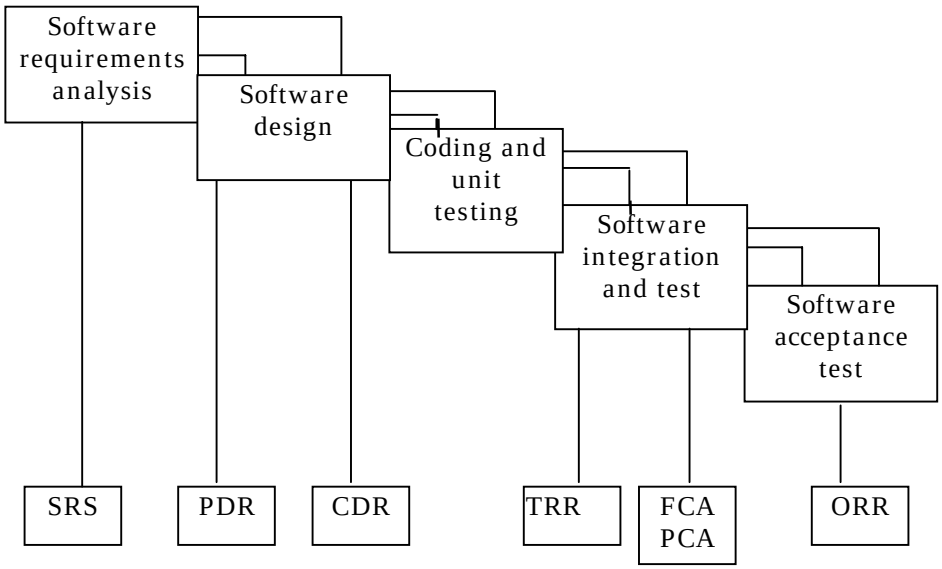
3.4.2.1 Üldine kaskaadmudel (*generic waterfall model*)

Üldine kaskaadmudel publitseeriti esmakordselt artiklis Royce (1970) – see artikkel jäi erilise tähelepanuta ligi viieteistkümneks aastaks (kuigi kaskaadmudeli ideed levisid praktikute seas). Esimese mudeli kohta oli Royce ettepanek võrdlemisi radikaalne – ta laiendas tarkvara projekti huvideringi spetsifitseerimisele, tõi tarkvara loomise varajastesse etappidesse ilmutatult sisse vetoõigusega tellija, ja formuleeris ilmutatult tagasiside/iteratsioonide vajaduse tarkvara loomisel. Enamus neid ideid olid juba kuuekümnendatel aastatel rahvaluule tasemel levinud, aga keegi ei olnud neid selgete juhtideedena formuleerinud. Kaskaadmudel on levinud veel kaasajalgi, temast on tehtud palju modifikatsioone. Tema baasil tekkis ka esimene prototüüpide jadal baseeruv elutsükli mudel. Tegelikult sisaldus prototüübi idee juba algses Royce poolt soovitatud mudelis – nn. “build it twice” nõue, mis eeldas, et paralleelselt nõuete analüüsiga ja projekteerimisega toimuks ka nende etappide tulemuste kontrollimine prototüübil.

Hilisemates, laiemalt levinud kaskaadmudeli versioonides (vt. joonis 3.4) jäi tarkvara algetappidel prototüübi kasutamise idee pikaks ajaks varjusurma. Tõenäoliseks põhjuseks oli spetsifikatsioonil ja projektil baseeruvate prototüüpide automaatse genereerimise meetodite ja vahendite puudumine. Prototüübid tulid uuesti päevakorda seoses spiraalse elutsükli mudeliga.

Iga elutsükli etapi lõpus peab valmima dokument selle etapi töö tulemustest koos hinnangutega ja tarkvara kvaliteedi hindamise grupi (SQA – Software Quality Assessment Group) heakskiiduga. Näiteks tarkvara nõuete analüüsi väljunddokumendiks on **tarkvara nõuete spetsifikatsioon** (*Software Requirement Specification, SRS*) ja selle analüüsi tulemused ning hinnangud koos eelprojektiga (kõrgtaseme projektiga) esitatakse aruandes nimega **eelprojekti ülevaade** (*Preliminary Design Review (PDR),* vahel ka *Primary Design Review*). Peale **eelprojekti ülevaate** valmimist loetakse nõuete analüüsi etapp lõppenuks ja algab projekteerimise etapp.

Royce (1970) tõi esmakordselt sisse tarkvara kvaliteeti hindava grupi (SQA – Software Quality Assessment group) mõiste ja tema sisuliselt vetoõigusega rolli projekti arengu juhtimisel. See viimane ettepanek on tippfirmade tavalisse praktikasse jõudnud alles 1980-ndatel.



Joonis 3.4 Simple (generic) waterfall model for software life-cycle (© Behforooz & Hudson 1996)

Joonisel kasutatud lühendid:

SRS - Software Requirement Specification
PDR - Preliminary design review
CDR - Critical design review
TRR - Test readiness review
FCA - Functional configuration audit
PCA - Physical configuration audit
ORR - Operation readiness review

Kaskaadmudeli puudused tulenevad osaliselt tema tugevast orientatsioonist dokumentatsioonile – elutsükli etapp sai lõppeda alles siis kui nõutav dokument oli koostatud ja kõigi vajalike inimeste poolt allkirjastatud. Tellija ei pruugi programmistide erialases zhargoonis koostatud dokumentidest täielikult aru saada – ta annab oma allkirja saamata lõpuni aru, mis dokumendist võib järelduda. Tulemuseks on tuntud lugu, kus tellija ütleb pärast toote kättesaamist – jah, ma tean, ma andsin nõusoleku nendes dokumentides toodud kirjeldustele, aga ma ei uskunud, et need nõuded viivad sellise süsteemini. Vaid prototüüp, kus tellija saab ise mängida, saab sellise häda vastu aidata. Lisaks on massiivse dokumentatsiooni produtseerimine väikeste tarkvaraprojektide puhul üsna kulukas tegevus.

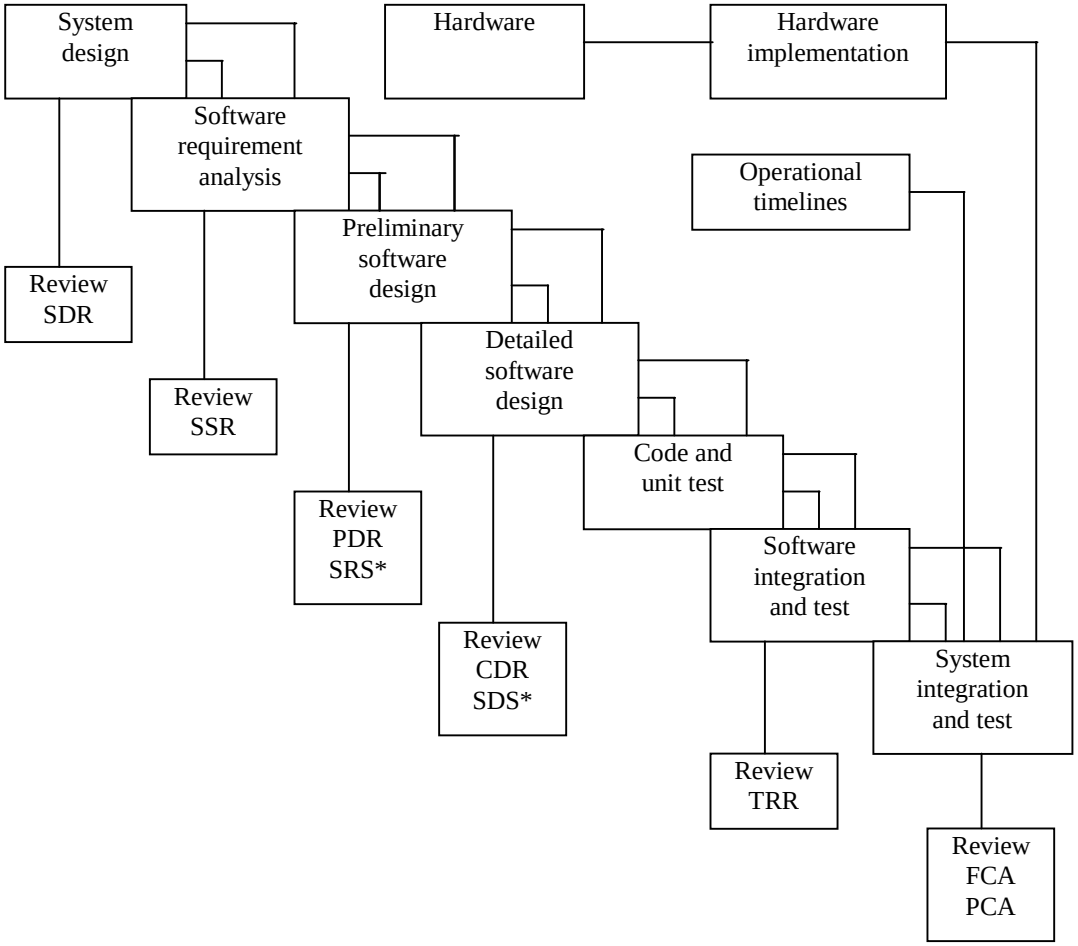
Teine kaskaadmudeli oluline puudus on, et ta ei peegelda tarkvara loomisprotsessi paralleelset ja iteratiivset olemust ning soodustab töö planeerimist rangelt järjestikuliste etappide kaupa. Kuigi tagasiside on kaskaadmudelisse sisse projekteeritud, on projekti progressi, paljude võimalike

pooleliolevate iteratsioonide tõttu kaskaadmudeli alusel raske hinnata. See nõuab projektijuhilt erilist oskust ja pingutust. Praktikas võib juhtuda, et mõned osad esimesest etapist saavad valmis enne teisi. Aja kokkuhoiu mõttes saab sellisel juhul alustada mõnede osade projekteerimisega enne teisi. Ka tuleb, formaalsetest reeglitest kinnipidamiseks, koostada dokumentidest mitu versiooni (s.t. esimeses versioonis puuduvate osade nõuete spetsifikatsioon ja selle analüüsi tulemused tohib lisada alles siis kui antakse välja dokumendi uus versioon). Analoogne olukord esineb iga etapi lõpetamisel. Teisalt, vigade ja mittekokkusobivate osade tekkimise tõenäosus projektis väheneb kui iga etapp saab korralikult lõpetatud enne järgmise juurde asumist.

Kaskaadmudel, nagu ka teised elutsükli mudelid eeldavad hästi töötavat dokumenteerimise ja versioonide kontrolli süsteemi, kuigi kaskaadmudeli tavalised kirjeldused ei rõhuta seda vajadust.

3.4.2.2 DOD mudel (*Software lifecycle model used by US Department of Defence*)

Erinevalt lihtsast (generatiivsest) kaskaadmudelist, kus tarkvara nõuded tekivad näiliselt tühjale kohale, saab DOD'i mudeli kohaselt tarkvara nõuete analüüsi etapp oma lähtematerjalid vahetult süsteemi (mille osaks saab loodav tarkvara alamsüsteem) väljatöötamisel valminud dokumentidest. (vt. joonis 3.5).



Joonis 3.5. DOD mudel (© Behforooz & Hudson, 1996)

SDR – System design review
SSR – Software system review
PDR – Preliminary design review
SRS – Software requirements specification
CDR – Critical design review
SDS – Software design specification
TRR – Test readiness review
FCA – Functional configuration audit
PCA – Physical configuration audit

Süsteemi projekti ülevaate ja analüüsi aruanne (*System Design Review, SDR*) on aluseks tarkvara alamsüsteemile esitatavate nõuete fikseerimisel. Tarkvara nõuete analüüsi tulemusena tekib tulevase tarkvara alamsüsteemi esialgne ülevaade ja analüüs (*Software System Review, SSR*), mis on tarkvara-inseneride poolt kontrollitud, kooskõlastatud ja heaks kiidetud versioon süsteemiprojekteerijate esitatud nõuetest tarkvara alamsüsteemile.

Teine erinevus lihtsast kaskaadmudelist on tarkvara projekteerimisetapi jagamine kaheks – eelprojekteerimiseks ja detailseks projekteerimiseks. Eelprojekteerimisetapil valmib tarkvara nõuete spetsifikatsiooni (*SRS document, formally approved Software Requirements Specification*) põhjal tarkvara kõrgtaseme projekt (*high-level design*). Selle projekti analüüsi tulemused koondatakse dokumenti PDR (*Preliminary Design Review*, eelprojekti ülevaade), mis ongi selle etapi lõppdokumendiks. Detailprojekteerimise etapp lõpeb SDS (Software Design Specification, tarkvara detailne projekt) dokumendi formaalse vastuvõtmisega. Lõplik detailprojekti heaks-kiitmine toimub ülevaate koosolekul, pärast arutelu. Arutelu käik ja tulemused vormistatakse CDR (Critical Design Review, projekti tehnilise ülevaatuse aruanne) dokumendina.

Kolmandaks erinevuseks lihtsast (generatiivsest) kaskaadmudelist on katsetamise ja integreerimise etapi jagunemine kaheks osaks – tarkvara katsetamise ja integreerimise ning süsteemi katsetamise ja integreerimise etapiks. Tarkvara katsetamise ja integreerimise etapil tekib dokument TRR (Test Readiness Review, katsetusteks valmisoleku aruanne). Selle dokumendi ettevalmistamise käigus kontrollitakse kõigi tarkvaraosade vastavust katsetusplaanis toodud protseduuridele ja kriteeriumitele. Tarkvara katsetamise etapi eesmärgiks on veenduda, et tarkvarasüsteem vastab esitatud nõuetele ja on valmis katsetusteks koos süsteemi ülejäänud komponentidega.

3.4.2.3 NASA mudel ja kokkuvõte kaskaadmudelitest

NASA *(National Aeronautics and Space Administration)* tarkvara elutsükli mudel on sisuliselt identne DOD mudeliga (nagu võikski oodata). Peamine erinevus on terminoloogiline. Näiteks, tarkvara loomise elutsükkel ise (*software development lifecycle*) on NASA terminoloogias tarkvara hankimise elutsükkel (*software acquisition life-cycle*), kodeerimise etapp (*coding phase*) on realiseerimise etapp (*implementation phase*), funktsioonide kokkusobivuse kontroll (*function configuration audit*) toimub veidi varem ja on tuntud nime vastuvõtu katsetused all (*acceptance testing*).

Kaskaadmudeli ja tema otseste järglaste omaduste kokkuvõte

On veel mitmeid generatiivsele kaskaadmudelile sisuliselt väga lähedasi elutsükli mudeleid, mida huvilised võivad ise raamatutest tundma õppida. Järgnevalt on loetletud mõned kaskaadmudelite klassi eelised ja puudused.

Kaskaadmudelite eelised:
- tarkvaraprotsess on hästi dokumenteeritud
- dokumentide sisu, ülevaatuste põhimõtted ja etapilt etapile kriteeriumid on hästi defineeritud
- tarkvaraprotsess, kaasa-arvatud ülevaatused ja dokumendid, on põhimõtteliselt kohaldatav iga üksiku toote (või projekti) unikaalsete nõuetega
- kaskaadmudelite klassi kasutamise kohta on palju kogemusi
- kõik olemasolevad standardid rakenduvad kaskaadmudelitele.

Kaskaadmudelite puudused:
- kaskaadmudelid ei suuda piisavalt kajastada tarkvaraprotsessi paralleelset ja iteratiivset olemust
- tarkvaraprotsess ei ole piisava efektiivsusega kohandatav erinevate rakenduste vajadustega
- paljud väidavad, et kaskaadmudel ei ole ühilduv ADA keele poolt eeldatud elutsükliga
- paljud väidavad, et kaskaadmudel ei ole ühilduv ekspertsüsteemide tehnoloogiaga
- kaskaadmudel ei sunni projekti juhtkonda kaasama tellijat ja kasutajat tarkvara loomise protsessi
- kaskaadmudeli rakendamise protsess on kulukas (dokumentatsioonimahukas, raskelt automatiseeritav)

3.4.2.4 Spiraalne mudel (*Spiral life-cycle model*)

Spiraalne elutsükli mudel on loodud 1980-ndatel aastatel (esimene publikatsioon ilmus 1986 aastal). Vaata B.W.Boehm (1986) “A spiral model of software development and enhancement”. Software Engineering Notes, vol.11. no.4, 14-24. Kättesaadavamal kujul on sama artikkel avaldatud B.W.Boehm (1988) “A spiral model of Software development and enhancement”, IEEE Computer, vol. 21, no.5, 61-72. (vt. ka III osa lõpus toodud kasutatud kirjanduse loetelu).

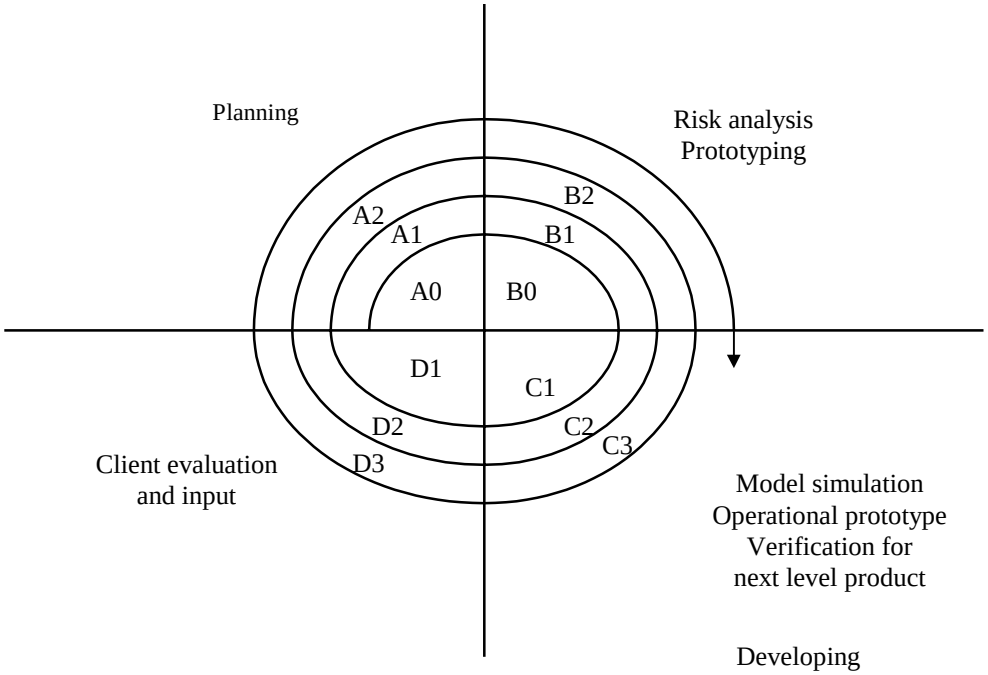
Uue mudeli loomise ajendiks oli kriitika mis väitis, et kaskaadmudel ei julgusta piisavalt prototüüpimise tehnika kasutamist ja tarkvara taaskasutamist (*software reuse*). Spiraalne elutsükli mudel baseerub toote loomise riski osatähtsuse arvestamise märgataval suurendamisel ning kombineerib klassikalise elutsükli mudeli parimad omadused prototüüpide loomisega ja varem valmistatud tarkvara osade taaskasutamisega.

Vaatame näiteks üsna tavalist arvutiteaduse , programmistide ja tarkvaratehnika kriitikat – pärast suure hulga raha ja aja kulutamist saab kasutaja toote, mis formaalselt vastab lepingus fikseeritud kasutaja nõuetele, kuid sisuliselt ei vasta kasutaja vajadustele. On lihtne süüdistada kasutajat rumaluses, kuid sama lihtne on tarkvara toote tegijaid süüdistada tellija tahtlikus petmises. Üks võimalus vähendada vastastikuseid süüdistusi on lihtsustada tellija suhtlemist tarkvara tegijatega.

Ainus seni leitud hästi töötav võimalus on valmistada toode prototüüpide jadana, kus iga prototüüp on kättesaadav ka kasutajale. Iga järgmise prototüübi tegemisel võetakse arvesse kasutaja märkuseid, soovitusi ja hirmusid, mis tulenevad eelmise prototüübi kasutamise kogemustest. Spiraalse elutsükli mudeli järgimine sunnib tarkvara projekti just selliselt arendama.

Schach (1996) võrdleb spiraalmudelit ja kaskaadmudelit järgmiselt – spiraalmudel on nagu kaskaadmudel, kus iga etapi alguses on kohustuslik riskianalüüs ja iga etapi lõpus on kohustuslik testimine, valideerimine ja/või verifitseerimine (kaasa arvatud prototüübil ja tellija osavõtul). Tuleb siiski arvestada, et mitte igasuguseid projektiga seotud riske ei saa avastada ja lahendada tänu spiraalsele elutsükli mudelile ja prototüüpide jadale. Näiteks, võimalus, et ei õnnestu palgata vajalikku tarkvara spetsialisti. Veel sarnaseid näited – meeskond ei saa projektiga hakkama (oskuste ja kogemuste puudumise tõttu); allhankijate töö kvaliteet ja ajastamine selgub alles vahetult enne nende kasutuselevõttu. Sellegipärast juhib spiraalmudel tähelepanu ka nende riskide perioodilise uurimise ja hindamise vajadusele.

Joonisel 3.6 on kujutatud toote arendus spiraalse elutsükli kohaselt. Kvadrant **A** esitab erineva tasemega plaanide kogumit. Plaanid tulenevad ajaliselt varem eksisteerinud plaaniversioonist, mida on korrigeeritud eelmise plaani põhjal valminud prototüübi katsetamisel saadud kogemuste ning järgnenud analüüsi ja verifitseerimise tulemuste põhjal. Kvadrant **B** esitab erinevatel plaaniversioonidel baseeruva riskianalüüsi, iga versiooni jaoks läbiviidud riskianalüüsi saab katseliselt kontrollida selleks ehitatud prototüübil. Prototüübi katsetamisel osalevad (reeglina) ka tulevased kasutajad.



Joonis 3.6 Spiraalne tarkvara elutsükli mudel (© Behforooz & Hudson, 1996)

Kvadrandis **C** viiakse läbi vajalikud tarkvara arendusega seotud protseduurid. Näiteks, ülesandepüstituse mõistlikkuse analüüs (*feasibility study*), projekteerimise kontseptsioonide valik (*design concepts*), luuakse vajaliku tasemega spetsifikatsioonid ja projektid (*specifications and designs*), toimub kõigi loodud toote osade poolformaalne analüüs, ja/või verifitseerimine, ning järgmisel spiraalikeerul lahendamist vajavate probleemide fikseerimine (uued stsenaariumid, verifitseerimist vajavad osad, detailiseerimist vajavad osad, juba tehtud kulutuste hindamine, jne).

Kvadrandis **D** hindavad tellija ja projekti juhtkond tehtud tööd, tehtud kulutusi, eelseisvaid kulutusi ja annavad loa minna järgmisele spiraalikeerule, või teevad otsuse töö lõpetada.

Spiraalse elutsükli kohaselt valmib toode kui prototüüpide jada piirväärtus, reaalselt katkestatakse toote arendus enne piirväärtuse (s.t. ilma vigadeta versiooni) saavutamist. Paljudel juhtudel antakse tooted ekspluatatsiooni teades, et 2-5% kõigist tootes olevatest vigadest on veel avastamata.

Spiraalse elutsükli eelised:

- see sobib hästi nii täiesti uue süsteemi loomiseks kui ka olemasoleva süsteemi modifitseerimiseks
- enamus eksisteerivaid elutsükli mudeleid on vaadeldavad spiraalse elutsükli mudeli erijuhtudena
- elutsüklisse sisseehitatud kohustuslik riskianalüüs võimaldab aegsasti avastada palju ohtusid ja vahetevahel neid ka vältida

Spiraalse elutsükli mudeli efektiivsust uuriti 25 projekti peal – kõige väiksem tootlikkuse tõus oli 50% (võrreldes tootlikkusega sama projekti tegemisel mõne teise elutsükli mudeli järgi).Enamuses uuritavast 25 projektist täheldati kahekordset (100%) tootlikkuse tõusu.

Mõned järgnevalt esitatud spiraalse elutsükli mudeli kasutamisel saadud kogemused on pikemalt kirjeldatud Schach (1996) lk. 69, näiteks:

- Katsetamise, valideerimise, verifitseerimise piisavuse hindamine iga etapi lõpus on alati raske – liiga palju katsetusi kulutab raha ja aega, liiga vähe katsetusi viib probleemideni toote üleandmisel, või täiendavate modifitseerimiskuludeni ekspluatatsioonis. Spiraalse mudeli kohaselt ei tehta vahet tarkvara arendamise ja tarkvara kasutamise etappide vahel, riski analüüs annab hinnangu toote omadustele ja nendega seotud kuludele; kui risk arvatakse olema liiga kõrge, võib teha veel ühe spiraali keeru.
- Spiraalne elutsükli mudel on sobiv firmasisese tootearendus jaoks. Kui riskianalüüs soovitab projekti lõpetada, on see ühe firma sees suhteliselt lihtne. Kui tarkvara arendamiseks on sõlmitud leping teise firmaga ja tarkvara tegija tahab lepingut ühepoolselt katsetada, tuleb maksta lepingus sätestatud trahve. Ei ole haruldane ka kohtuprotsess. Seda spiraalse mudeli omapära saab kompenseerida sobiva lepingu sõlmimisega – sageli on nurjumisele määratud projekti lõpetamine kasulik ka tellijale – kuid mitte alati pole see võimalik.
- Spiraalse tarkvara elutsükli mudeli kasutamist piirav hoiatus on, et riski analüüsiks tehtavad kulutused võivad olla samas suurusjärgus kui projekti enda maksumus. See tähendab, et riski analüüs omab mõtet vaid piisavalt suurte (kalliste) projektide puhul. Teiste sõnadega –

projekti juhtkond peab alati otsustama, kui palju võib ilma riski analüüsita kaotada ja selle põhjal eraldama rahad riski analüüsiks.

3.5 Tarkvara protsessi kirjeldamise mudelid

Tarkvara protsessi kirjeldamise mudelid on firmade küpsuse, professionaalsuse ja töövõime hindamise vahendid – tarkvara elutsükli mudeliga võrreldes metamudelid. Üks esimesi taolisi mudeleid oli Capability Maturity Model (CMM) – välja töötatud 1986 aastal Carnegie-Melloni ülikooli Tarkvaratehnika Instituudi poolt (Software Engineering Institute (SEI) of Carnegie-Mellon University). Kõige värskem teadaolev taoline toode pärineb samast instituudist -- Personal Software Process (PSP). Selle mudeli alusel on võimalik tarkvara inseneril parandada oma individuaalseid erialaseid oskuseid ja tööstiili – vastandina CMM mudelile, mis keskendub organisatsiooni võimete ja tööstiili parandamisele. Eestikeelsed vasted võiksid olla:

Capability Maturity Model (CMM) = Võimete Küpsuse mudel
Personal Software Process (PSP) = Isikupärane tarkvaraprotsess

Tarkvara protsess haarab endasse nii tarkvara loomisega seotud tehnilised kui ka haldusaspektid. CMM strateegia kohaselt tuleb parandada haldusosa kuna see toob (loodetavasti) automaatselt kaasa ka tehniliste aspektide paranemise. Organisatsioonid muutuvad aeglaselt, seetõttu jagab CMM ettevõtte arengu tasemeteks. Liikumine ühelt tasemelt teisele toimub evolutsiooniliselt, tavaliselt mitme väikese sammuga.

Eristatakse viit ettevõtte arengutaset:

- **Algtase (maturity level 1, initial level)** – ettevõttes puudub süstemaatiline tarkvaratehnika meetodite ja vahendite rakendamine. Kõik tehakse kuidas parajasti juhtub. Üksikud projektid võivad olla tõelised õnnestumised, keskelt läbi ületatakse nii projekti eelarveid kui ka tähtaegu. Juhtkond tegeleb peamiselt kriisiolukordade likvideerimisega, ei jää aega tulevikuarengute planeerimisele. Kahjuks on suur enamus tarkvara firmasid sellel tasemel.
- **Korratavuse tase (maturity level 2, repeatable level)** – ettevõttes toimivad elementaarsed tarkvaratehnika haldusstruktuurid ja meetodid. Planeerimine ja projektijuhtimine toimub edukalt kordaläinud projektide kogemuste kordamise teel (siit ka taseme nimi). Esimesed tarkvaraprotsessi mõõtmised on sisse viidud sarnaste projektide väljaselgitamiseks, peetakse suhteliselt täpset arvestust ka kuludest ja tähtaegadest kinnipidamise kohta. Kriisi likvideerimise asemel püütakse kriise ennetada, kuid mitte alati ei jätku selleks infot (pole piisavalt mõõtmisi).
- **Arusaamise tase (maturity level 3, defined level)** -- ettevõtte tarkvaraprotsess on täielikult dokumenteeritud; tarkvaraprotsessi tehnilised ja haldustegevused on selgelt kirjeldatud, kuid ei toimi alati nii nagu ette nähtud. Tehakse pidevaid pingutusi seatud eesmärgi poole liikumiseks. Toimuvad regulaarsed projektide ülevaatused, tulemusi üldistatakse kogu ettevõtte tarkvaraprotsessi parandamist silmas pidades. Sellele tasemele jõudnud firmades on mõtet kasutusele võtta tarkvaratehnika tööriistad (*CASE tools*) selleks, et parandada toote kvaliteeti ja töötajate tootlikkust. Tarkvaratehnika tööriistade kasutuselevõtt algtasemel olevas ettevõttes (kus peamiselt püütakse lahendada juba tekkinud kriisiolukordi) suurendab reeglina

seal juba niigi valitsevat kaost. Kolmanda taseme on saavutanud mitmed ettevõtted, kuid kahel järgmisel tasemel ei ole (1995 aasta seisuga) veel ühtegi ettevõtet, vaid üksikud töörühmad on saavutanud kolmandast tasemest kõrgema taseme.

- **Haldamise tase (maturity level 4, managed level)** – sellel tasemel seatakse igale projektile toote kvaliteedi ja tootlikkuse nõuded. Nende nõuete täitmist mõõdetakse pidevalt ja püütakse eesmärgist kõrvalekallet hoida nii väiksena kui võimalik. Lisaks rakendatakse statistilist kvaliteedi kontrolli, et eristada juhuslikku kõrvalekallet ning süstemaatilist kvaliteedi ja tootlikkuse nõuete rikkumist.
- **Optimeerimise tase (maturity level 5, optimizing level)** – eelmiste projektide mõõtmiste tulemuste kohaselt muudetakse tarkvaraprotsessi, eesmärgiga parandada toote garanteeritud kvaliteeti ja suurendada tootlikkust. See tase eeldab täielikke mõõtmisi ja võimet juhtkonna otsuseid kontrollitult ellu viia.

CMM mudeli rakendamise kogemus näitab, et esimeselt tasemelt teisele jõudmine nõuab 3 kuni 5 aastat pidevat tarkvaraprotsessi täiustamist. CMM kasutatakse USA-s ja ka mujal tarkvaratootjate hindamiseks ja tõsisemate konkursside korral on ametlikult omistatud CMM tase oluliseks otsustamise aluseks – näiteks US Air Force lepingupartneril peab olema vähemalt CMM kolmas tase (arusaamise tase).

Lisaks CMM mudelile on veel mitmeid analoogilisi mudeleid (näiteks SPICE). Ülevaade tarkvaraprotsessi taseme hindamise mudelitest ja nende rakendamisest on Janek Metsalliku magistritöös “Tarkvara võimete küpsuse mudeli teise taseme võtmepraktikate rakendamine Eestis”, 1998, 89 lk., mida saab lugeda Tallinna Tehnikaülikooli raamatukogus.

Alternatiivne katse parandada tarkvaraprotsessi baseerub *International Standards Organisation* ISO 9000-seeria standarditel. Selles seerias on 5 standardit, mis on rakendatavad paljude tööstusliku tootmisega seotud tegevuste (muuhulgas ka projekteerimine, toote arendamine, tootmine, installeerimine tellija juures, ja hooldamine) kvaliteedi hindamiseks.

ISO 9000 ideoloogia kohaselt ei garanteeri standardi nõuete rahuldamine kvaliteeti, vaid ainult vähendab kehva kvaliteediga toote väljastamise tõenäosust. Paralleelselt standardi nõuete täitmisega on oluline, et toimuks töötajate regulaarne täiendõpe ja eksisteeriks administratsiooni pidev tähelepanu kvaliteedi parandamise abinõudele.

Kuigi ISO 9000, CMM, SPICE ja teised järgivad sama eesmärki – toote kvaliteedi tõstmist ja töö tootlikkuse suurendamist – ei ole nende nõuded täielikult kattuvad, ega ka mitte eriti kokkulangevad. Nii on leitud vähemalt kaks organisatsiooni, kes on CMM mõttes algtasemel (maturity level 1), kuid on sertifitseeritud ISO 9000 kohaselt. On samuti teada ettevõtteid, kus on CMM kolmas tase, kuid keda ei ole sertifitseeritud ISO 9000 järgi.

Maailma areng tundub siiski olema kvaliteedi sertifitseerimise poole. Et saada korralikku lepingut, peab firmal olema tema küpsust näitav tunnistus – esialgu ei ole veel ühte üldtunnustatud sertifitseerijat, aga nii CMM kui ka ISO, või mõne teise poolt antud tunnistus on parem kui metsfirmaks olemine.

3.6 Tarkvaraprotsessi puudutavad kordamisküsimused

1. Mille poolest on tarkvaratehnika (software engineering) erinev ja raskemini hallatav kui muud tehnokäsitluse (engineering) valdkonnad?
2. Miks põhjustab mudeli ja tegelikkuse vahekorra ähmasus raskusi tarkvaratehnikas? Kas üldse põhjustab?
3. Miks on tarkvara-inseneri töö lähedasem süsteemi-inseneri tööle kui arvutiteadlase või arvuti-inseneri tööle? Loetlege mõned põhjused?
4. Kirjeldage uue süsteemi tegemise algust. Mis on esimene tõsisem tegevus? Mis on selle tegevuse tulemus?
5. Mida tuleks arvestada projekti loomisega seotud tegevuste konkreetseteks töödeks jagamisel?
6. Miks on vaja Gantt'i ja PERT'i diagramme? Milliseid andmeid on vaja nende koostamiseks?
7. Kuidas te hindate tulevase projekti üldmaksumust? Kas teate konkreetseid meetodeid?
8. Millistest komponentidest koosneb projekti üldmaksumus? Loetlege võimalikult paljusid komponente?
9. Kas suurema arvu programmistidega saab projekti alati kiiremini valmis? Miks? Miks mitte?
10. Mille peal kulub tegelikult programmisti tööaeg? Andke ühe tüüpilise tööpäeva ligikaudne ajajaotus protsentides.
11. Kui sageli ja milliste sündmuste puhul te korraldaksite oma juhitud projekti tehnilisi ülevaatusi?
12. Millised küsimused tulevad arutusele projekti tehnilistel ülevaatustel? Kes nendest osa võtavad?
13. Kuidas korraldatakse muudatuste sisseviimine tarkvaraprojekti? Kellega tuleb muudatused kooskõlastada?
14. Miks on vaja tarkvara elutsükli mudelit? Milliseid elutsükli mudeleid te teate?
15. Loetlege kaskaadmudeli head ja halvad omadused.
16. Mille poolest DOD mudel erineb üldisest kaskaadmudelist?
17. Võrrelge spiraalmudelit ja kaskaadmudelit.
18. Kas spiraalne elutsükkel on kasutatav kõikides tarkvaraprojektides? Miks? Miks mitte?
19. Miks soovitatakse moodulite ja integreeritud süsteemi katsetamine anda mitte projekteerijatele, vaid eraldi katsetusgrupile? Loetlege sellise otsuse häid ja halbu külgi.
20. Loetlege tarkvara-inseneri ülesandeid, kes peab projekteerima ja läbi viima katsetused. Milliseid tarkvara projekti dokumente läheb tal selleks vaja?
21. Miks on vaja uurida tarkvaraprotsessi (tarkvara loomise protsessi)? Kuidas seda tehakse?
22. Milliseid tarkvaraprotsessi uurimise metoodikaid te teate? Kas saab neis mõnda eelistada? Miks?
23. Milliseid probleeme Te näete 5-liikmelise tudengite meeskonna semestri pikkuse projekti tegemisel? Kas on õiglane panna kogu meeskonnale sama hinne? Miks?

3.7 Tarkvaratehnikas kasutatavad tööriistad.

Tarkvara loomiseks on kasutusel uskumatult suur hulk erinevaid meetodeid ja neid toetavaid tööriistu. Sageli ei sobi ühe projekti sees kasutatavad meetodid ja tööriistad omavahel kokku, mistõttu saadav integraalne kasu on küsitav.

Kogu käesolevas dokumendid toodud kirjeldus sisaldab meetodeid ja tööriistu, mida kasutatakse peamiselt spetsifitseerimiseks ja projekteerimiseks. Kodeerimise, silumise ja testimise vahendid jäävad käesolevast kursusest välja. Loetelus sisalduvad meetodid ja tööriistad on valitud peamiselt nende leviku ulatuse põhjal, mitte nende potentsiaalse kasulikkuse alusel, ka ei ole loetelu koostatud lähtudes ainult (peamiselt) reaalajasüsteemide vajadustest.

Suur osa turul olevatest tööriistadest on ikka veel seotud triviaalsete, kuid tüütute tegevuste automatiseerimisega – näiteks tüüpdokumentatsiooni tegemise abivahendid, diagrammide joonistamise ja nende elementaarse süntaktilise korrektsuse kontroll, projekti plaanide koostamine jne. Schach (1996) rõhutab kurba tõde, et *CASE* tähendab *computer assisted (aided) software engineering*, mitte *computer automated software engineering*. 1999 aastal võib siiski olla veidi optimistlikum, kui Schach 1996-ndal. Üsna mitmed tarkvaratehnikakeskkonnad on, lisaks tüütutes triviaalsustes abistamisele, hakanud automaatselt koodi genereerima lähtudes koostatud ja kontrollitud spetsifikatsioonist ja/või projektist – esialgu prototüüpideks, aga vahel sobib see ka lõppkoodiks. Siiski, mitte kõik kasutajad ei ole automaatselt genereeritud koodi stiiliga ja kvaliteediga rahul. Kahjuks on spetsifikatsiooni ja projekti korrektsuse formaalse analüüsi toetamise võime suurenenud tööriistades minimaalselt.

Mõned autorid (näiteks, Sommerville (1992)) teevad vahet tarkvaratehnika tööriistade (*CASE tools*), ja tarkvara tegemise keskkondade (*software development environments*, *software engineering environments*) vahenditel – vahetegemisel on suur mõte ajaloolises plaanis. Kaasaegse elu aspektist vaadates tundub küll, et ei ole enam mõtet rääkida eraldiseisvatest tööriistadest – sedavõrd valdavaks on muutunud keskkondade kasutamine (eriti objekt-orienteeritud keskkondade populaarsuse tõttu). Sellest tingituna kasutatakse edaspidi nimetust tarkvaratehnika tööriistad nii üksikute tööriistade kui ka keskkondade tähenduses.

Näiteks kasutab Sommerville (1992) järgmist tarkvaratehnika tööriistade liigitust:

- **Üksiku töö tegemiseks mõeldud tööriistad**:
  - planeerimis- ja hindamistööriistad
  - teksti redigeerimise vahendid
  - dokumentide genereerimise vahendid
  - prototüübi genereerimise vahendid
  - diagrammide redigeerimise vahendid
  - andmesõnastiku tegemise ja kasutamise vahendid
  - kasutajaliideste tegemise vahendid
  - keeletöötlusvahendid
  - interaktiivsed silumisvahendid
  - modelleerimise ja simuleerimisvahendid
  - jne
- **Tööde kompleksi tegemist toetavad tööriistad**:

- programmeerimiskeskkonnad – *programming environments* (ajalooliselt esimesed tarkvaratehnika keskkonnad) jagunevad omakorda üldotstarbelisteks ja konkreetsele keelele orienteeritud keskkondadeks;
- tarkvaratehnika töökohad – *CASE workbenches*, mis peamiselt toetavad tarkvaraprotsessi analüüsi ja projekteerimise alaseid tegevusi; tarkvaratehnika töökohad on saadud eraldiseisvate tööriistade integreerimisel (lisades vajaduse korral puuduvad liidesed);
- tarkvaratehnika keskkonnad – *software engineering environments (SEE)*, mis Sommerville väitel on loomulikult tekkinud programmeerimiskeskkondade arengu tulemusena; tema antud definitsioon tarkvaratehnikakeskkonnale on : *an SEE is a collection of hardware and software tools which can act in combination in an integrated way*; peab siiski tunnistama, et oma olemuselt ei erine tarkvaratehnika keskkonnad ja tarkvaratehnika töökohad oluliselt, seetõttu tundub Sommerville range töökohtade ja keskkonna eristamine veidi kunstlik.

Täpsem info tarkvaratehnika keskkondade ja muude tööriistade, ning seal kasutatavate meetodite kohta antakse kursuses LAP 8714 Tarkvaratehnika keskkonnad.

3.7.1 Üldised abivahendid.

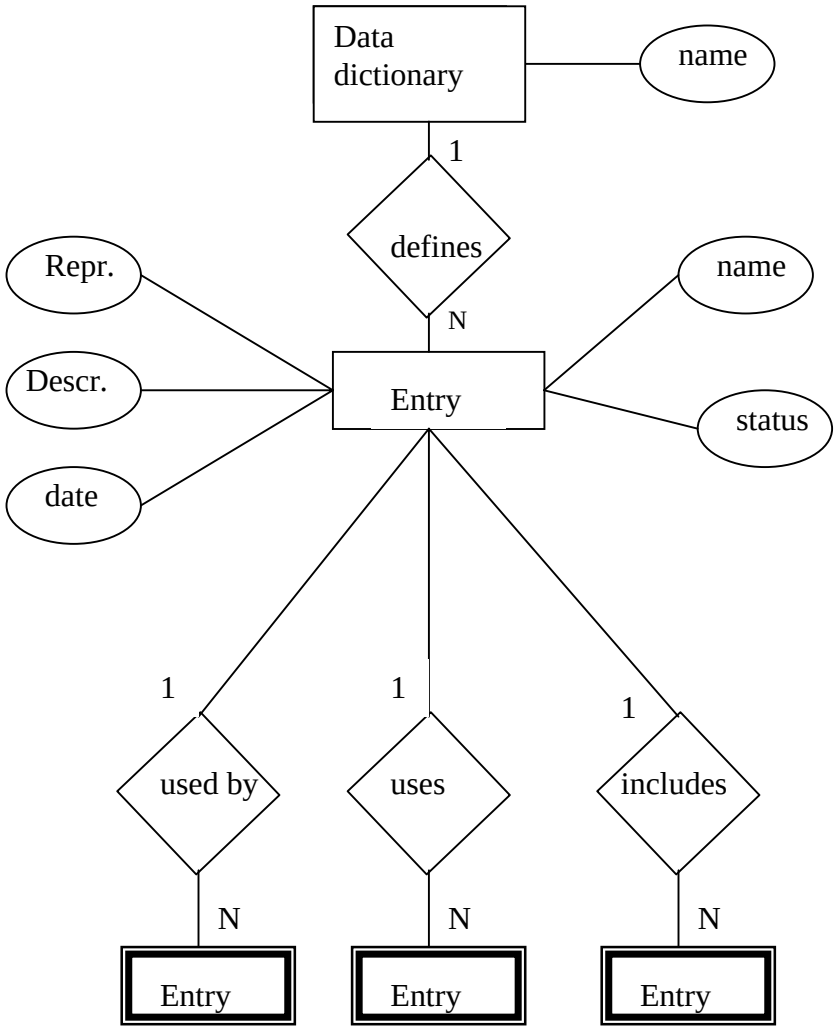
3.7.1.1 Andmesõnastik.

Andmesõnastiku peamine eesmärk on projektis kasutatavate nimede ja mõistete üheselt mõistetav ning täpne defineerimine. Sellega luuakse reaalne võimalus, et projektimeeskonna liikmed räägivad samadest asjadest kasutades samu mõisteid. Lisaks lihtsustub ka nimede ning mõistete unikaalsuse (automaatne) kontroll.

Lihtsustatult võib andmesõnastikku vaadata kui nimekirja tarkvarasüsteemi osade, muutujate, seoste, teisenduste, sündmuste, jne tähestikuliselt järjestatud nimedest. Iga nimega käib kaasas seda nime kandva objekti kirjeldus; kui objekt koosneb osadest, siis ka tema osade nimed ja nende kirjeldused. Tugitarkvara, mis võimaldab luua, hooldada ja kasutada andmesõnastikku on oluline osa andmesõnastikust. Enamasti on andmesõnastik ehitatud tarkvaratehnika tööriista (keskkonna) sisse. Joonisel 3.7 on illustreeritud andmesõnastiku loogilist struktuuri kasutades olemdiagrammide (entity-relationship diagram) tüüpi tähistust.

Igal andmesõnastiku elemendil on nimi, esitusvorm (*representation*), kirjeldus (*description*), loomise kuupäev (*date*) ja olek (*status*), mis võimaldab ajutiselt säilitada ka hetkel mittekasutatavaid andmesõnastiku elemente (*obsolate entries*). Iga andmesõnastiku element osaleb kolmes seoses – millised teised andmesõnastiku elemendid kasutavad teda, milliseid teisi andmesõnastiku elemente kasutab ta ise ja milliseid teisi andmesõnastiku elemente sisaldab vaadeldav element (näiteks operatsioonid abstraktses andmetüübis). Igal konkreetsel juhul tuleb mõistetele “kasutab” (uses), “kasutatakse” (used by), ja “sisaldab” (includes) anda konkreetne ja ühemõtteline sisu.

Väga paljudes tööriistades on andmesõnastiku (vahel nimetatud ka projektisõnastikuks) kasutamine kohustuslik. Andmesõnastik omandab sageli alusdokumendi rolli -- andmesõnastikku salvestatud terminite alusel valitakse diskussioonide ja arutluste käigus välja edasise töö jaoks olulised mõisted -- näiteks objektid ja nende vahelised seosed ning relatsioonid, operatsioonid ja muu selline; objekt-orienteeritud meetodites, andmeteisendused, juhtimisteisendused, andmevood, sündmused ja muu selline andmemudelitel baseeruvates meetodites, kasutajanõuded, katsetuste kriteeriumid ja eesmärgid. On üsna tavaline, et andmesõnastikku hakatakse koostama kohe pärast tulevase süsteemi (tarkvaratoote) esimese kirjelduse saamist tellijalt/kasutajalt. Kogu projekti jooksul täiendatakse ja täpsustatakse sõnastikku, vastuvõtu katsetustel (või neile järgnevatel kohtuprotsessidel) on andmesõnastikus defineeritud mõistetel sageli määrav tähtsus otsuse kujundamisel.



Joonis 3.7 Data dictionary logical model (© Sommerville (1992))

Erinevates töökeskkondades on kasutusel erinevad nõuded andmesõnastiku elementide, nende struktuuri ja parameetrite kohta. Tüüpiliselt on vaja teada sisend- ja väljund muutujate loetelu, nende muutumispiirkonda, lubatud väärtuse muutumise kiirust (ja/või kiirendust), vajalikku arvutamise/mõõtmise täpsust, muutuja väärtuse kehtivusintervalli, jne. Neile võivad lisanduda sideprotokollide võimalused, krüpteerimise nõuded, nõutavad andmeedastuse kiirused (maksimum, keskmine, minimaalne), eeldatav töötlemiskiirus jne.

Eestis antav tarkvara-alane haridus ei pööra (minu andmetel) andmesõnastikele vajalikku tähelepanu. Kuigi paljudes tarkvaratehnika keskkondades on andmesõnastike kasutamine kohustuslik ja nende loomine/täiendamine toimub paralleelselt muu projekti arendamisega, oskavad laisad tarkvara-insenerid leida alati võimaluse kohustustest mööda hiilida – varem või hiljem kannatab selle all töö tootlikkus ja tarkvaratoote kvaliteet.

Täpsem info andmesõnastike ja nendes kasutatava Backus-Nauri poolt loodud (ja teiste poolt täiendatud) formalismi kohta on toodud andmevoo mudeleid käsitlevas kursuse osas. (veelgi detailsem info on saadaval raamatus **R. Narayan "Data Dictionary Implementation, Use and Maintenance", Prentice-Hall Mainframe Software series, Upper Saddle River, N.J. 1988)**

3.7.1.2 Otsustamine ja kompromisside tegemine *(Decision support)*

Iga tarkvaraprojekti käigus on vaja valida alternatiivsete variantide vahel. Valiku aluseks olevate otsustuste ja järelduste objektiivsusest ja põhjendatusest sõltub tihti kogu projekti õnnestumine.

Otsustamine on enamasti raskendatud olemasoleva informatsiooni mittetäielikkuse, või mittepüsiva iseloomu tõttu. Seetõttu on loomulik üritada lõplikku valikut/otsustamist edasi lükata täieliku info saamiseni, või nii kaua kui võimalik -- otsustamise edasilükkamisest tekkiv võimalik kasu ei kasva siiski lineaarselt, vaid omab selget maksimumi.

On oluline, et esimese valikuvajaduse ilmnemisel (või esimese valiku tegemisel) fikseeritakse valiku aluseks olevad kriteeriumid ja vajalikud lähteandmed kirjalikult (veelgi parem oleks valiku kriteeriumid fikseerida kasutaja nõuete spetsifikatsioonis, kahjuks ei ole see alati võimalik). Projekti arengu käigus võivad valiku kriteeriumid muutuda (nii kvantitatiivselt kui ka kvalitatiivselt), on oluline et ka kõik kriteeriumite muutused oleksid kirjalikult fikseeritud.

Otsustusprotsess ise peab selgelt näitama:

- miks üks alternatiiv on parem kui teised
- kui usaldatav on saadud otsus (võrreldes peamiste alternatiividega)
- kuidas saaks otsuse usaldatavust parandada (mida oleks veel vaja arvesse võtta?).

Behforooz & Hudson (1996) soovitavad otsustamise abivahendina kasutada kompromissi maatriksil (trade-off matrix) baseeruvat meetodit (vt. detaile nimetatud raamatust lk.151 - 153). Behforooz ja Hudson rõhutavad, et mida kauem saab (administreerimisega seotud) otsustamist edasi lükata (ilma edasilükkamise negatiivse mõjuta – näiteks, meeskonna moraalile, projekti

maksumusele või ajagraafikule), seda täielikumaks muutub otsustamisel kasutatav info, ja seda parema otsuse saab teha. Siit tuleneb esimene kompromiss.

Otsustusprotsess, lisaks otsusele, peab tegema nähtavaks, miks on kasulik ühte alternatiivi teisele eelistada. Siit tuleneb teine vajalik kompromiss – otsustuskriteeriumid on subjektiivselt valitud, iga kriteeriumi väärtused erinevate alternatiivide korral on reeglina subjektiivselt hinnatud.

Kompromissi maatriksi read moodustatakse valikut mõjutavatest kriteeriumitest, veerud moodustuvad igale alternatiivile vastavast kriteeriumite väärtustest, ja iga kriteeriumi suhtelisest tähtsusest otsuse tegemisel. Koostatud kompromissi maatriks annab (antud lähtehinnangute jaoks) objektiivse aluse otsuste tegemiseks ja näitab ühtlasi iga alternatiivi nõrgad ja tugevad kohad.

Lisaks kompromissi maatriksile on veel palju teisi otsustusmeetodeid, paljud nendest on realiseeritud ka programmidena. Näiteks, Expert Choice (http://www.expertchoice.com) mis kasutab Thomas Saaty meetodit.

Kui on tegemist eksperthinnangutega ja otsustajal puudub korralik teoreetiline ettevalmistus, on soovitav kasutada Delphi meetodit (Hicks and Gullet (1981), lk. 250), mis töötab järgmiselt:

- Sõltumatutelt, üksteisega mitte kokkupuutuvatelt ekspertidelt kogutakse vastused küsimustikule.
- Anonüümsetest vastustest tehakse tabel ja saadetakse kõigile vastanud ekspertidele.
- Seejärel palutakse ekspertidelt uuesti vastuseid samale küsimustikule, paljud eksperdid korrigeerivad oma eelmisi vastuseid (kasutades teiste ekspertide vastuseid täiendava infona)
- Saadud vastustest koostatakse uus tabel ja saadetakse vastanud ekspertidele, protsessi korratakse kuni saavutatakse ekspertide piisav üksmeel.

Delphi meetodi põhieelis on, et ekspertide arvamusi ei saa mõjutada mitte kuulsad nimed ja suured isiksused (kuna need ei ole otsustajatele teada), vaid ainult tegelikud vastused.

Oluline on leida objektiivsed otsustuskriteeriumid. Reaalses elus ei tehta otsust peaaegu kunagi ainult ühe kriteeriumi alusel. Näiteks tarkvara arhitektuuri valikul tuleks arvestada vähemalt nelja kriteeriumiga (mis võivad omavahel olla vastuolus):

- realiseerimise kulutusi
- nõuded protsessori võimsusele ja mälumahule
- tarkvara struktuuri ülesehituse mugavust ja tema katsetatavust
- hooldatavust ja modifitseeritavust.

Lisaks mitme kriteeriumi olemasolu tunnistamisele ja arvestamisele, on oluline tunnistada kriteeriumite erinevat kaalu parima otsuse tegemisel, kaalud paneb iga projekti meeskond lähtudes oma konkreetse projekti vajadustest.

Mõned kontrollküsimused, mida projektijuht peaks endalt küsima enne otsustamise juurde asumist:

- Kas meil on kõik vajalikud kriteeriumid arvesse võetud?

- Millised sageli kasutatud (või antud juhul vajalikud) kriteeriumid on jäänud välja?
- Kas erinevate kriteeriumite kaalud on meie rakenduse jaoks sobivad?
- Milliste reeglite järgi kvantifitseeritakse kriteeriumites sisalduvaid kvantitatiivseid parameetreid?

3.7.1.3 Stsenaariumid.*(use cases, operational timelines, scenarios, operational procedures, event flows*)

Stsenaariumid on paljudel juhtudel asendamatud tarkvara katsetuste planeerimisel ja tarkvara valideerimisel. Eriti olulised on harvaesinevaid situatsioone kirjeldavad stsenaariumid süsteemi töö- ja veakindluse testimisel. Stsenaariumite kirjeldamiseks ei ole enamuses tarkvaratehnika keskkondades erilisi abivahendeid ja nad esitatakse suhteliselt vabas vormis. Erandiks on mõned prototüpiseerimisel baseeruvad metoodikad, kus stsenaariumi kasutatakse simulatsioonisessioonide planeerimiseks prototüübil.

Stsenaariumite veidi ootamatu, kuid väga efektiivne kasutusala on nende rakendamine tarkvara nõuete spetsifikatsiooni analüüsimisel (ja ka projekti analüüsimisel). Eriti kasulikud on stsenaariumid kasutaja nõuete valideerimisel.

Paljudes meetodites baseerub poolformaalsete mudelite käsitsi läbivaatamine (ülevaatus, *walk-through, inspection*) tulevase kasutaja poolt eelnevalt valmistatud stsenaariumitel (näiteks metoodikad, mis baseeruvad andmevoomudelitel ja objektmudelitel).

Mõned soovitused stsenaariumite komplekti koostamiseks on toodud Behforooz & Hudson 1996 raamatus (lk. 180-182):

- Eristada süsteemi töötamisel tekkivad erinevad rezhiimid ja fikseerida ühest rezhiimist teise ülemineku tingimused. Näiteks, süsteemi algkäivitus, külmstardiga seotud protseduurid, vigade poolt põhjustatud eriolukorrad, süsteemi normaalne sulgemine, süsteemi sulgemine avarii korral, taastumine vea korral (sõltuvalt vigadest) jne, jne.
- Täpsustada ühest rezhiimist teise ülemineku võimalusi ja tingimusi
- Valmistada detailsed stsenaariumid iga üksiku töörezhiimi jaoks, jne.

3.7.2 Praktikas enamlevinud mudelid tarkvara töö ja/või struktuuri kirjeldamiseks

Andmevoomudelid *(Data flow diagrams, DFD)*
Andmevoomudel on laialt kasutatav poolformaalsete meetodite klassi (vahel nimetatud ka struktuursete meetodite klassiks, ingliskeeles *semiformal, or structural methods)* kuuluv meetod. Andmevoomudeleid kasutatakse nii nõuete kirjeldamiseks, spetsifitseerimiseks kui ka projekteerimiseks. Sageli seostatakse andmevoomudelid funktsionaalset kirjeldust esmatähtsaks pidava koolkonnaga -- enamasti järeldub sellest ka funktsionaalse dekompositsiooni tähtsaks pidamine. Tuleb siiski rõhutada, et meetodi olemus ei eelda kitsalt funktsionaalsest dekompositsioonist lähtumist (vt. Ward & Mellor, 1985). Peamiseks mudeli eesmärgiks on

süsteemis, läbi andmeteisenduste (protsesside) ja andmeladude (data transformations and data stores), liikuvate andmevoogude ja nende juhtimise kujutamine.

Andmevoomudelitest ja nendega koostöötavatest mudelitest tuleb pikemalt juttu selle kursuse neljandas osas.

Objektmudelid *(Object-oriented modelling)* on vaieldamatult aluseks viimase aastakümne kõige populaarsematele tarkvara projekteerimisvahenditele. Nad kuuluvad, samuti nagu andmevoomudelid, poolformaalsete ehk struktuursete mudelite klassi. Peamised mudeli ehitamise elemendid on objektid ja nende vahelised assotsiatsioonid. Igal objektil on oma sisemine ehitus -- nad sisaldavad operatsioone ja atribuute. Kuigi paljud autorid püüavad jätta mulje, et objektmudelid erinevad andmevoomudelitest põhimõtteliselt, ei ole see tegelikult päris nii – sarnasus on palju tugevam kui paljud objektmudelite omapära propageerijad seda tunnistada tahaksid.

Objektmudelitest ja teistest nendega koos kasutatavatest mudelitest tuleb pikemalt juttu selle kursuse neljandas osas.

Lõplikud automaadid *(Finite state machines)* on ajalooliselt üks esimesi formaalseid mudeleid, mida programmide loomisel kasutati. Neid kasutatakse süsteemi käitumise kirjeldamiseks. Kahjuks kirjeldavad lõplikud automaadid ainult kvalitatiivset käitumist, s.t. tegevuste järjestamist objektiivsete põhjuslike seoste (või inimese poolt pealesurutud järjestusprintsiipide) alusel. Käitumise kvantitatiivsete omaduste kirjeldamine, kvantitatiivsete kitsenduste arvestamine, ja eriti nende garanteerimine, on lõplike automaatide abil üsna lootusetu -- vaatamata sellele, et erinevad autorid lubavad selliste tegevuste eest hoolitseda.

Lisaks lõplike automaatide kasutamisele iseseisvate mudelitena (vt. A.M.Davis “Software Requirements Analysis and Specification” Prentice-Hall, Upper Saddle River,N.J. 1990) kasutatakse neid sageli ka nn. ausa olekumudeli (*Fair Transition System)* kirjeldamiseks. Aus olekumudel on aluseks ajaloogikate kasutamist võimaldavate mudelite klassile. Mõned autorid nimetavad ajaloogikatel baseeruvaid mudeleid ka sekundaarseteks mudeliteks, kuna uurimisobjektideks on ausa olekumudeli poolt genereeritud sündmuste jada omadused. Sellistest mudelitest tuleb pisut rohkem juttu kursuses LAP 8714 “Tarkvaratehnika keskkonnad”.

Suhteliselt laia kasutuse on praktikas leidnud lõplike automaatide mitmesugused üldistused ja laiendused. Näiteks meetod, mis on tuntud *StateCharts* nime all ja võimaldab kirjeldada mitme samaaegselt ja paralleelselt toimiva lõpliku automaadi koostööd. Seda meetodit kasutatakse nii iseseisvalt kui ka mitmetes objekt-orienteeritud metoodikates.

Teine, vanem ja laiemalt tuntud lõplike automaatide üldistus on Petri võrgud. Petri võrke on nii palju erinevaid versioone, et nimetus ilma spetsiaalsete selgitavate ja täpsustavate atribuutideta omab väga vähe konkreetset sisu. Lihtsate Petri võrkude detailsem kirjeldus on toodud raamatus J.L.Peterson “Petri Net Theory and the Modeling of Systems” Prentice-Hall, N.J., 1981. Peamised hilisemad uuendused (kõrgtasemega Petri võrgud, värvilised võrgud, mittebinaarsete märgistega võrgud) koos viidetega vastavate programmidena realiseeritud tööriistadele on toodud raamatus

K.Jensen, G.Rozenberg (Eds) High Level Petri nets. Theory and applications, Springer Verlag, 1991, 724pp.

Matemaatiline loogika on viimased 20 aastat aeglaselt kuid kindlalt tunginud tarkvara spetsifitseerimise ja projekteerimise valdkonda. Praktikud peavad matemaatilisel loogikal baseeruvaid vahendeid siiski veel liiga teoreetiliseks (kuna enamusel tarkvarainseneridel puudub matemaatilisest loogikast arusaamiseks vajalik haridus). Reaalajasüsteemide üha laiema levimisega on esile kerkinud vajadus formaalselt verifitseeritud tarkvara ja süsteemide järele -- osaliselt sellest tingituna on valminud palju metoodikaid ja tööriistu, mis suures ulatuses baseeruvad matemaatilisel loogikal.

Üsna laialt on levinud katsed juurutada praktikasse süsteemide spetsifitseerimise ja projekteerimise keel VDM. Formaalsel mudelil baseeruv spetsifitseerimiskeel VDM muutus kasutatavaks 1970-ndate lõpus. VDM-st on oluliselt mõjutatud spetsifitseerimiskeel Z, mis valmis Oxfordis 1980-ndate lõpus (Sommerville (1992)). Z baseerub hulgateoorial, mis tänu heale formaliseeritusele võimaldab kasutada esimest järku predikaatarvutust spetsifikatsioonide ja projektide kirjeldamiseks ja verifitseerimiseks (vt. näiteks, J.B. Wordsworth “Software Development with Z: A Practical Approach to Formal Methods in Software Engineering” Addison-Wesley, Reading, Mass, 1994).

Sageli laiendatakse esimest järku predikaatarvutust ajakitsenduste arvestamise vahenditega. Enamasti kasutatakse ajakitsendustega probleemide korral siiski ühte modaal-loogikate erijuhtu -- nn. ajaloogikaid.

Algebralised meetodid on viimane siin nimetatav formaalsete meetodite klass. Esimeseks märgiks algebraliste meetodite tungimisest tarkvara tööriistade ja neis kasutatavate meetodite hulka oli abstraktsete andmetüüpide (*abstract data types*) levik. See oli ka üks esimesi formalisme, mida kasutati programmi spetsifitseerimiseks. Abstraktsed andmetüübid on oluliselt mõjutanud ka objekt-orienteeritud meetodite teoreetilisemat käsitlust.

Protsessialgebratel (*process algebra*) baseeruvad meetodid on, analoogiliselt matemaatilise loogikaga, tunginud programmeerimise teooriasse ja kohati ka praktikasse. Ükski ülalloetletud formaalsetest meetoditest ei tule selles kursuses (LAP5711) detailsemale käsitlemisele. Mingil määral antakse formaalsete meetodite olemuse ja nende kasutamise kohta lisainformatsiooni järgnevates kursustes (Tarkvara dünaamika LAP 5712 ja Tarkvaratehnika keskkonnad LAP 8714).

3.7.3 Kordamisküsimused

1. Kuidas peaks suhtuma paljudesse tarkvaratehnika tööriistade, keskkondade, jne liigitustesse? Milliseid liigitusi te teate? Milliseid eelistate?
2. Kuidas abistab andmesõnastik projekti meeskonna liikmeid? Tooge vähemalt neli näidet.
3. Miks on andmesõnastikus vaja fikseerida nõuded andmete täpsuse kohta?

4. Kirjeldage tüüpilise andmesõnastiku elemendi struktuuri ja tema seoseid teiste andmesõnastiku elementidega.
5. Mis on kompromissi maatriks? Kuidas koostada ja kasutada kompromissi maatriksit?
6. Kuidas toimib Delphi otsustusmeetod? Mis on selle meetodi head küljed?
7. Millistest otsustusmeetoditest olete veel kuulnud? Millist meetodit kasutaksite Teie?
8. Kuidas mõjutavad stsenaariumid tarkvara projekti arengut?
9. Kas stsenaariumite abil saab **hinnata** süsteemi veakindlust? Modifitseeritavust? Selgitage kuidas.
10. Kuidas on seotud lõplikud automaadid ja nn. *statecharts*?

3.8 Tarkvara meetrika *(Software metrics)*

Selle osa eesmärgiks on juhtida tudengite tähelepanu vajadusele mõõta tarkvara protsessi ja toodet nii nagu igat teist tehnilist toodet. Kogu müstika tarkvara, selle tegemise ja kasutamise ümber, samuti ka projektide tähtajast ja eelarvest mitte kinnipidamine tuleneb otseselt raskustest tarkvara parameetrite ja tarkvara loomise protsessi parameetrite mõõtmisel ja sageli ka mõõdetavate parameetrite valiku vaieldavusest. Paljude uurijate väitel on peamine erinevus käsitöö ja tööstusliku tootmise vahel selles, et käsitöö baseerub kvalitatiivsetel meetoditel (inspiratsioonil, sisetundel, meeleolul, intuitsioonil, jne) samas kui tööstuslik tootmine kasutab olulisel määral kvantitatiivseid mõõtmis- ja projekteerimismeetodeid.

Tarkvara meetrikaga on tegeldud ligi 30 aastat. Üht-teist on saavutatud, kuid üldiselt aktsepteeritud tulemusi ei ole. Juhindume siiski hiina vanasõnast -- parem süüdata üks küünal kui kiruda pimedust tuhat korda -- ja järgnevalt on toodud lühike loetelu tarkvara meetrika peamistest tulemustest koos viidetega nende detailsema kirjelduse allikatele.

Tarkvara meetrika jagatakse sageli kahte klassi (Sommerville (1992)):

- **Haldusmeetrika**, mille tulemusi kasutatakse tarkvaraprotsessi juhtimiseks; näiteks selleks, et täpsemalt koostada tarkvara plaani. Haldusmeetrika näideteks on tarkvaraprotsessis kulutatud aeg, kasutatud inimpäevad, toodetud koodi maht, kulutatud raha, jne.
- **Kvaliteedimeetrika**, mille tulemusena saab hinnata tarkvara toote kvaliteeti ja omadusi. Näiteks, toote kasutajamanuaali loetavust hinnatakse nn. Fog'i indeksiga, tarkvaratoote hooldatavust hinnatakse toote struktuuri tsüklomaatilise keerukuse alusel (nn. McCabe keerukus).

Uurijad on üksmeelel, et enamasti ei õnnestu mõõta neid parameetreid, mida tõesti oleks vaja. Selletõttu loodetakse (oletatakse), et eksisteerib meile arusaadav seos mõõdetava ja vajaliku parameetri väärtuste vahel. Kõige enam vaidlusi tekitabki oletatava (katseliselt näidatava seose) olemasolu ja tema olemuse mõistmine. Näiteks, McCabe keerukus on ilmselt seotud toote keerukusega; samas on kindlalt teada, et McCabe keerukus ei hõlma kõiki toote keerukust mõjutavaid faktoreid. Enamus tarkvara meetrika näitajaid omab statistiliselt piisavat usaldust, kui

mõõtmiseks vajalike andmete kogumine on automaatne (mis tagab andmete objektiivsuse) ja kasutatakse vaid käesolevat projekti iseloomustavaid andmeid.

3.8.1 Näiteid tarkvaratoote kvaliteedimeetrikas kasutatavatest mõistetest.

Tarkvaratoote iseloomustamiseks kasutatakse ebamääraseid, täpselt defineerimata mõisteid – näiteks hea projekt, adapteeruv tarkvarasüsteem, jne. Intuitiivselt tähendab hea projekt hästi arusaadavalt (nii kasutajale kui ka programmistile) esitatud projekti, millele on vajaduse korral kerge muudatusi sisse viia (ilma, et tekiks palju kõrvalmõjusid (*side-effects*)).

Tarkvaratoote arusaadavus on pisut seotud tema struktuuri keerukusega (või lihtsusega). Järelikult on vaja mingeid keerukuse hinnanguid. Tüüpiline mõõdetav keerukuse hinnang on McCabe keerukus (vt. täpsemalt Sommerville (1992) lk. 434-437). McCabe keerukus on tegelikult programmi kirjeldava voograafi läbivate sõltumatute teede arv. Testide arv, mis on vajalik kõikide programmis sisalduvate tingimuslike üleminekute kontrollimiseks on võrdne nn. tsüklomaatilise arvuga, ehk McCabe keerukusega. Keerukuse näitajana on McCabe keerukusmõõdul kaks puudust:

- Ta mõõdab keerukust , mis on määratud tsüklites olevate predikaatide ja tingimuslike üleminekutega. Kui programm on andmete poolt juhitud, võib tal olla väga madal McCabe keerukuse mõõt, aga sellegipoolest on ta raskesti arusaadav
- Sama kaaluga arvestatakse väliseid ja sisemisi tsükleid (*nested and non-nested loops*). On üsna kindel, et sisemised tingimuslikud struktuurid on palju raskemini mõistetavad kui välimised.

Toote muudetavus (adapteeruvus) on oluliselt seotud omadustega, millest tuleb juttu loengusarja viiendas osas – arvutiteaduse ja tarkvaratehnika vahekord. Need on moodulite (süsteemi osade) kokkukuuluvus (ingl.k. *cohesion*), ja ühendatus (ingl.k. *coupling*). Nõrk ühendatus ja tugev kokkukuuluvus on hästi adapteeruva süsteemi olulised omadused.

Kokkukuuluvust (*cohesion*) ei ole võimalik vahetult mõõta, ühendatust (*coupling*) saab mõõta automaatselt, kui selline funktsioon ehitada sisse tarkvaratehnika tööriista. Constantine and Yourdan (1979) on soovitanud ühendatuse (*coupling*) mõõduks võtta süsteemi komponenti sissetulevate ja väljaminevate voogude (ühenduste) arvu tarkvaratoote struktuuri kirjeldaval mudelil. Sisenevate voogude arv (*fan-in value of a component*) on struktuursel mudelil seda komponenti väljakutsuvate (kasutavate) teiste komponentide arv (sellesse komponenti sisenevate voogude arv). Väljuvate voogude arv (*fan-out value of the component*) on defineeritud analoogiliselt. Hiljem on Constantine & Yourdani ühendatuse (coupling) mõõdu definitsiooni pisut laiendatud – nn. informatsioonilise ühendatuse mõõduks on komponendist väljuvate lokaalsete andmevoogude arv pluss globaalsete andmestruktuuride arv, mille väärtuseid komponent muudab.

3.8.2 Tarkvara toodet iseloomustavad parameetrid

Ajalooliselt algas tarkvara meetrika valmis toodete analüüsist, selletõttu on see osa ka kõige paremini läbitöötatud. Kahjuks on valmis toote analüüsi tulemused suurel määral tagantjärgi tarkus ning seda laadi parameetrid ja soovitused on tarkvara elutsükli esimestel etappidel suhteliselt kasutud. Selletõttu piirdume soovitusega lugeda Behforooz & Hudsoni (1996) raamatut (lk. 401 - 416) täpsema ajaloolise info ja täpsemat infot andvate allikate leidmiseks.

Tarkvara toodet iseloomustavad peamised parameetrid on järgmised.

Tarkvara identifikaator määrab (ideaalis) üheselt nime, funktsioonid, liidesed jms, mis on vajalik tarkvaratoote taaskasutamiseks. Praktikas taandub identifikaator sageli inimese koostatud verbaalsele kirjeldusele, mis reeglina taaskasutamist ei lihtsusta.
Tarkvara ridade arv (*source lines of code*) on tarkvara mahtu iseloomustavaks parameetriks. Programmi ridade arv antakse kas loogiliste ridade arvuna (s.t. lähtekeele avaldiste arv) või füüsiliste ridade arvuna (füüsiliste ridade arv, mis kulub lähtekeele avaldise kirjutamiseks).
Funktsiooni punktide arv *(Function points),* mis arvestab sisendite, väljundite, päringute, failide ja välisliideste hulka vaadeldavas tarkvaratootes. Funktsiooni punktide arvu saab kogemuslikke koefitsiente kasutades teisendada arvestuslikuks tarkvara ridade arvuks.
Omaduste punktide arv *(Feature points)* on sarnane funktsiooni punktide arvuga, millele on lisatud algoritmi omaduste eksperthinnangust ja kogu toote keerukuse eksperthinnangust tulenevad punktid. Näiteks on andmetöötlustarkvara ja reaalajatarkvara oluliselt erineva keerukusega.

3.8.3 Tarkvara projekteerimist ja realiseerimist kirjeldavad koefitsiendid.

Selles alajaotuses kirjeldatud koefitsiendid võimaldavad tarkvaraplaani tegemisel arvestada toote, realiseerimise ja tootearenduskeskkonna tegelikke omadusi – koefitsiendid näitavad kui palju võivad realistlikud kulutused olla suuremad elementaarse tarkvaratootega, ideaalse realiseerimisplatvormiga ja ideaalse tootearendus-keskkonnaga võrreldes. Koefitsiendid jagunevad kolme suurde gruppi:

- toodet iseloomustavad koefitsiendid
- realisatsiooni iseloomustavad koefitsiendid
- toote arendus- ja realiseerimiskeskkonda iseloomustavad koefitsiendid

Siin toodud koefitsientide hindamist on detailsemalt käsitletud raamatus B.W.Boehm “ Software Engineering Economics” Prentice-Hall, Upper Saddle River, N.J., 1981

Toodet iseloomustavad koefitsiendid:

- nõuded tarkvaratoote töökindlusele, veakindlusele, jne; hinnatakse kvantitatiivselt; iga esitatud nõude mõju tarkvara loomise maksumusele ja ajakavale kirjeldatakse koefitsiendina, millega korrutatakse läbi tavalise tarkvaratoote maksumus ja ajakava
- nõuded andmebaasile ja andmebaasi iseloomustus; antakse koefitsiendid sõltuvalt andmebaasi suurusest ja nõuetest andmebaasile
- tarkvaratoote keerukuse hinnang: koefitsient vastab toote keskmisele keerukusele (kõige lihtsam on andmetöötlus tarkvara, kõige keerulisem reaalajasüsteemi tarkvara).

Realisatsiooni iseloomustavad koefitsiendid

- realiseerimiseks soovitatud protsessori(te) poolt esitatavad piirangud; koefitsient määratakse vastavalt tootele esitatud ajakitsenduste arvule, algoritmide keerukusele, protsessori(te) võimsusele ja muudele omadustele
- mälujaotussüsteemist tulenevate piirangute, ja/või mälumahu poolt seatud kitsenduste mõju
- arvutikonfiguratsiooni ja süsteemtarkvara (operatsioonisüsteemi, DBMS'i *(database management system*), taaskasutatavate moodulite teegi, programmeerimise vahendite) töökindluse ja muude omaduste poolt põhjustatud täiendava riski koefitsient

Arenduskeskkonda ja projekti meeskonna kogemust arvestavad koefitsiendid:

- arvutite kättesaadavust meeskonna liikmetele arvestav koefitsient
- arenduskeskkonna töö stabiilsust (vigade puudumist/olemasolu) arvestav koefitsient
- meeskonna liikmete eelnevat töökogemust ja üldist töövõimet arvestav koefitsient
- meeskonna eelnev kogemus antud projektis kasutatavate projekteerimis- ja programmeerimis-vahenditega
- kasutatavate tarkvaratehnikavahendite efektiivsust (kasulike funktsioonide hulka) arvestav koefitsient
- toote loomise plaani pingelisust arvestav koefitsient

Koefitsientide väärtuste määramise kohta saab lisadetaile ka Behforooz&Hudsoni (1996) raamatust lk 416-435 ja edasi.

3.9 Tarkvaratootega seotud riski haldamine

Sommerville (1992), lk. 409 annab terve komplekti mõisteid, mille abil selgitatakse, mis on risk. Nende definitsioonide kokkuvõttest järgneb, et:

Risk (*risk*) – kontseptsioon, mis on seotud ohu (*hazard*) tõsiduse astmega, ohu tõenäosusega ja tõenäosusega, et oht realiseerub õnnetusena (*mishap, accident*); tavaliselt kasutatakse sõna "risk" ka mõõduna, mis näitab millise tõenäosusega süsteemi käitumine võib ohustada ümbritsevat keskkonda.

Õnnetus (*mishap or accident*) – Planeerimata sündmus, või sündmuste jada, mis tekitab kahju ümbritsevale keskkonnale (inimesed, tehnorajatised, tehis- või looduskeskkond).

Tarkvara projektijuht peaks (ideaaljuhul) perioodiliselt hindama järgmisi riske – lepingute formuleeringutest tulenev risk, tehnoloogiline risk (riist- ja süsteemtarkvara ning tööriistad), projekteeritava toote keerukusest ja suurusest tulenev risk, tööjõuga seotud risk, kasutajaga suhtlemisest tulenev risk. Igal riski liigil on terve rida riski suurust mõjutavaid faktoreid, mis erinevates riski liikides võivad osaliselt kattuda.

Riski haldamise eesmärgiks on välja selgitada tüüpilised riski põhjustavad faktorid ja võimalused nende mõju vähendamiseks. Enamuse nurjunud tarkvaraprojektide puhul näitab *post mortem* analüüs mittepiisavat või hilinenud tähelepanu riski põhjustanud faktorite hindamisele ja vastumeetmete rakendamisele.

Riski haldamist käsitlevaid aruandeid nõuavad tavaliselt tellijad ja oma firma kõrgemad juhid -- mitte ainult sellepärast, et nad ei oleks arvestanud tõenäolise ülekuluga ja hilinemisega. Enamasti on riskianalüüsi põhjal võimalikult midagi olukorra parandamiseks ette võtta, või minimiseerida kaotust jättes lootusetus olukorras olev projekt õigeaegselt pooleli. Tellija ja tootja koostegevus on projekti tulemuslikuks lõpetamiseks äärmiselt oluline, kuid üpris raske saavutada.

3.9.1 Tüüpilised riskifaktorid tarkvaraprojektides

Konkreetsed riskifaktorid, nende tähtsus ja potentsiaalne mõju projekti tulemusele, tuleb iga konkreetse projekti puhul spetsiaalsete uuringute ja analüüsi abil välja selgitada. Enamlevinud 10 tarkvaraprojektide riskifaktorit on järgmised:

1. Tööjõu vähesus

Abinõud selle faktori neutraliseerimiseks (või vähendamiseks) on:

piisava arvu töötajate palkamine, tööde liitmine gruppideks, võtmeisikutega väga täpse lepingu sõlmimine, meeskonna liikmete ettevalmistuse parandamine (*cross-training*), võtmeisikute tööplaani muutmine intensiivsemaks, allettevõtu lepingute võimaluse ettenägemine.

2. Mitterealistlik tootmisplaan ja eelarve

Vahel õnnestub olukorda leevendada kui:

eelnevalt olukorda hoolega analüüsida ja katsuda leida täiendavaid finantseerimisvõimalusi, projekteerida toode ümber vastavalt plaani ja eelarve võimalustele, realiseerida toode osaliselt, maksimaalselt kasutada varem loodud tarkvaramooduleid, lihtsustada tootele esitatavaid nõudeid, pidada kliendiga täiendavaid läbirääkimisi alternatiivsete lahenduste sobivuse üle.

3. Valede tarkvara funktsioonide loomine

Õigeaegsele avastamisele aitab kaasa:

korralik süsteemianalüüs, regulaarsed projekti tehnilised ülevaatused (eriti koos tulevase kasutajaga), toote prototüübi õigeaegne loomine ja katsetamine kasutaja koostatud stsenaariumite alusel, toote kasutusjuhendi varajane koostamine, üleandmis-vastuvõtu kriteeriumite ja testide õigeaegne koostamine ja tellijaga kooskõlastamine.

4. Vale kasutajaliidese loomine

Riski saab vähendada sarnaselt eelmises punktid toodule, pluss:

kasutaja koostatud stsenaariumite läbimängimine prototüübil, rakendusülesannete analüüs, kasutaja iseloomustuse koostamine (funktsioonid, tööstiil, koormus) ja selle arvestamine projekteerimisel.

5. Liigsed lubadused ja liigne enesekiitus (*Gold plating*)

Esineb sageli, on tavaliselt põhjustatud meeleheitlikust vajadusest toodet müüa; enamasti (eriti pikemas perspektiivis) annab vastupidise efekti. Selle faktori mõju vähendamiseks on kasulik:

jälgida täpselt kasutaja nõudeid ja neid lihvida (koos kasutajaga); varane prototüpiseerimine ja prototüübil mängimine (toob sageli tagasi maapeale); põhjalik tulude-kulude analüüs; toote reklaamitud omaduste vastavusse viimine toote hinnaga (või vastupidi).

6. Pidev nõuete muutmine

On sagedane nähtus, eriti täiesti uue toote tegemisel; selle vastu saab kui:

muudatuste kontrolli ja muudatuste sisseviimise lubade süsteem muuta väga rangeks, projekteeritavad moodulid tugevalt kapseldada (ja liides ülejäänud süsteemiga teha kergelt

muudetavaks), kasutada toote osade kaupa projekteerimist, lükata enamus muudatusi edasi järgmisse tooteversiooni, kooskõlastada nõuete spetsifikatsioon õigeaegselt lepingupartneritega ja lubada muudatusi vaid äärmisel vajadusel.

7. Probleemid väljastpoolt ostetud komponentidega

On eriti ohtlikud, kuna nende tekkimist on raske ette näha. Veidi aitab kui:

eelnevalt selgitada partnerite tausta ja tutvuda nende varasemate töödega, täpsete nõuete fikseerimine mujalt hangitavatele komponentidele, eriti nende vastuvõtu katsetustele (katseülesannete loomine, *benchmarking*), sobivusanalüüs muude süsteemi osadega enne lõppsumma maksmist.

8. Probleemid allettevõtjatega

Allettevõtjad ei pruugi alati täita kõiki konkreetses lepingus fikseeritud kitsendusi ja kohustusi. Selle riskifaktori vähendamiseks on vaja:

kontrollida allettevõtjate tausta (eelnevate lepingute tulemusi), enne töö eest tasumist teha toote põhjalik analüüs ja katsetamine; lepingud sõlmida selliselt, et suur osa maksmist toimuks pärast toote kõlbulikuks tunnistamist; fikseerida täpsed vastuvõtu-üleandmise nõuded; konkureerivate pakkumiste kogumine mitmelt allettevõtjalt (vahel ka konkureerivate toodete tellimine); regulaarne koostöö allettevõtjaga lepingu täitmise ajal.

9. Probleemid jõudlusnõuete rahuldamisega

Selle faktori alla on ühendatud nii tavalised jõudlusnõuded (töödeldava info hulk, tulemuste saamise kiirus) kui ka reaalajasüsteemide ajalise korrektsusega seotud nõuded (mis sageli ei ole vahetult seotud jõudlusega tavalises mõttes). Ajalist korrektsust käsitletakse põhjalikumalt kursuses LAP 5712 “Tarkvara dünaamika”. Jõudlus- ja ajastamisnõuetega seotud riskifaktori vähendamiseks on kasulik:

kasutada palju simuleerimist, kontrollida toote töövõimet tüüpülesannetel (*benchmarks*), regulaarsed katsetused prototüüpidel sobivalt valitud stsenaariumite järgi, sobiva riistvara platvormi valik ja loomulikult ajalise käitumise verifitseerimist võimaldavate tööriistade kasutamine.

10. Töötamine teoreetilise ja tehnoloogilise piiri peal

Selline olukord esineb põhimõtteliselt uute toodete tegemisel, või traditsioonilise toote kasutamisel uues kontekstis (kus täpsed nõuded ei ole teada, kuid on märksa rangemad kui toote traditsioonilistes rakendustes). Võimaluse korral tuleks:

läbi viia kasutatavate teoreetiliste ja tehnoloogiliste aluste põhjendatuse uurimine; fikseerida mittepiisavalt läbitöötatud teemad ja teha spetsiaalne analüüs nende teemade arenguperspektiivide selgitamiseks; võimaluse korral vältida lõpetamata teoreetiliste ja tegelikkuses katsetamata tehnoloogiliste tulemuste kasutamist; erilist tähelepanu pöörata alternatiivsetele lahendusvariantidele (kui uus teooria/tehnoloogia osutub mittesobivaks); teha alternatiivsete projektivariantide kulude-tulude ja riskifaktorite põhjalik analüüs; sage prototüüpide tegemine; perioodilised taustauuringud (ehk on midagi uut selgunud) ja sagedased tulevase süsteemi omaduste analüüs-ülevaatused; tõenäoliselt enam-muutuvate osade tugevam kapseldamine.

Põhjalikumad kommentaarid riskifaktorite kohta võib leida artiklist B.W.Boehm “Software Risk Management: Principles and Practice” IEEE Software, vol.8, no.1, 1991.

Riskihaldamisel on esimeseks ülesandeks leida peamised riski põhjustavad faktorid. Järgmine samm on vastava valdkonna detailne analüüs ja analüüsile tuginedes sobivate vastumeetmete väljatöötamine ja kasutusele võtmine.

Näide. Oletame, et kasutaja nõuete analüüsi tulemusena tundub peamiseks riskifaktoriks kujunevat no.6 (nõuete pidev muutmine). Sellisele järeldusele jõuame kui kasutaja nõuete spetsifikatsioon sisaldab ühte või enamat elementi järgmisest loetelust:

- palju lõplikult fikseerimata (detailiseerimata, kooskõlastamata) nõudeid
- mittepiisavad liideste kirjeldused (inimesega, teiste süsteemidega, väliskeskkonnaga)
- ebamäärased ja mitteüheselt tõlgendatavad nõuete formuleeringud
- mittekontrollitavad nõuded (s.t. mille kontrollimiseks puudub meetod)
- nõuded, mille täitmise kontrollimiseks ei ole täpse testi kirjeldust
- puuduvad või mitmeti tõlgendatavad nõuded jõudlusele
- kasutaja ja/või kasutaja liidese puuduliku määratluse ja kirjelduse

Loomulikud vastumeetmed selle riskifaktori mõju vähendamiseks oleks loetelus nimetatud puuduste kõrvaldamine. Lisaks on vaja kõik riskitegurid kvantifitseerida, omavahel võrrelda, ja leida strateegia nende mõju vähendamiseks (ning toote loomise plaanis võtta arvesse võimalikke lisaraskuseid, s.t. korrutada planeeritud aeg ja kulutused sobivalt valitud koefitsiendiga).

3.9.2 Riski mudel

Riski mudel peab võimaldama hinnata üksikutest faktoritest põhjustatud riski ja ka koguriski suurust. Järgnevalt on lühidalt kirjeldatud ühte võimalikest riski mudelitest, kirjeldus baseerub pidevast nõuete muutmisest (riskifaktor nr.6 eelmises alajaotuses) tingitud riski näitel (vt. ka Behforooz & Hudson 1996, lk.470 ja edasi).

Riski mudelleeritakse kahe muutuja koosmõjuna:

- **Ebaõnnestumise "tõenäosus"** (P_F, *probability of failure*), mis tegelikult on suhtelistes ühikutes (intervallis [0,1]) hinnatud igast üksikust riskifaktorist tekkiva vea "tõenäosuste" summa jagatud riskifaktorite koguarvuga;
- **Ebaõnnestumise mõju suhteline suurus** (C_F, *consequences of failure*), intervallis [0,1] ja kujutab endast summat igast üksikust faktorist põjustatud vea mõjust projektile, jagatud riskifaktorite koguarvuga.

Termin "tõenäosus" on siin kasutusel nõrgas mõttes – tegemist võib olla tõenäosusega, uskumuse mõõduga (*belief measure*), osalusfunktsiooniga (*membership function, as in fuzzy logic*), empiiriline tõenäosus (*empirical probability, subjective probability*), jne. Ainus kindel nõue on see, et "tõenäosus" peab olema vahemikus [0,1].

Iga riskifaktori mõju võib vaadelda tekitatuna mitme erineva põhjuse poolt. Kõigi põhjuste osatähtsus omakorda kvantifitseeritakse vastavalt konkreetsele vea "tõenäosusele" ja vea mõju suhtelisele suurusele. Vaatame näidet riskifaktori nr.6 (sagedane kasutaja nõuete muutmine) jaoks

koostatud riski mudelist. Selle faktori põhjustena on välja toodud nõuete kvaliteet (*requirements quality*) muudatuste kontrolli mehhanism (*change control system*), ja tellija osalus projektis (*client concurrence*). Riskifaktori põhjuste relatiivsed osakaalud määratakse eksperthinnangute põhjal.

Value	Requirements quality	Change control	Client concurrence
0.1	Minor corrections required	Hierarchy of control boards	Has agreed to acceptance criteria
0.3	Requirements need significant work to resolve. Issues being addressed, plan in place	Control boards in place. Tendency to a relaxed discipline.	Understands the need to establish acceptance criteria. working to close out agreements
0.5	Significant problems remain. Some ongoing activity to address major issues.	Change activity not following established procedures. Informal procedures are working.	Operators often change scenarios and surface new requirements
0.7	Very significant requirements problems. Insufficient resources to address and fix.	Informal procedure not working well. Formal procedure ignored.	Operators not concerned about costs and schedule.
0.9	Major problem in requirements. No plan to fix it.	No formal change control process in place	Client lacks will to control user/operator change requests.

Toodud tabeli põhjal saab välja arvutada riski arvutamise ühe komponendi (P_F, ebaõnnestumise tõenäosuse) vastavalt tegelikele põhjustele, kasutades valemit ja kvantifitseeritud riski põhjuseid.

Valemi üldkuju on

P_F = sum of values assigned to each factors/number of factors,

Konkreetne rakendusnäide võiks olla (vt.eelnevat tabelit, kus põhjuste osatähtsused on kaalutud vastavalt 0.5, 0.3, 05)

P_F = (0.5 x requirements quality value) + (0.3 x change control value) + (0.5 x client concurrence value).

Analoogiliselt tuleb toimida, kui arvutatakse kogu projekti üldriski – igale faktorile antakse oma subjektiivne osakaal, ja iga faktoriga seotud riski väärtus hinnatakse nagu eelnevas näites.

Kogu hindamisprotsessi juures tuleb rakendada loomingulist mõtlemist, tervet mõistust, analüüsivõimet, ja intuitsiooni. Tabelis toodud kolm põhjust on ainult üks võimalik näide, neid põhjuseid võib olla rohkem, samuti võivad nende osakaalud ja väärtused oluliselt sõltuda konkreetsest olukorrast ja projekti iseloomust.

Riskifaktori no.6 poolt potentsiaalselt põhjustatud ebaõnnestumise suhteline suurus võib avalduda erinevates ilmingutes, nende osakaal kogu projekti ebaõnnestumises võib jällegi olla erinev (tabelis on kasutatud osakaalusid cost = 0.5, budget = 0.2, performance = 0.3 ja ilmingute endi väärtuseid).

Ebaõnnestumise tagajärgede tõsidust (C_F) saab hinnata analoogiliselt ebaõnnestumise tekkimise tõenäosusele, kasutades üldvalemit

C_F = sum of values for consequences of failure factors/number of factors,

mille rakendus konkreetsel juhul oleks

C_F = (0.5 x value for cost) + (0.2 x value for budget) + (0.3 x value of performance)

Value	Cost	Budget	Performance
0.1	Within budget	Within plan	Minimum consequences, manageable
0.3	1-5% increase	Minor slip: 1 month	Small reduction in product performance and function
0.5	5-20% increase	1-3 month slip	Some reduction in product performance and function
0.7	20-40% increase	Greater than 3 months	Significant reduction in product performance and function
0.9	Greater than 40%	Greater than 6 month	Product is unusable, rejected by client

Kogurisk, mis on põhjustatud riskifaktori no. 6 poolt arvutatakse valemiga

$R_F = P_F + C_F - P_F \times C_F$

Kui vähegi võimalik tuleb riskianalüüsi tulemuste kohaselt muuta eelarvet, toote loomise plaani kestust, või midagi muud (vastavalt koguriskile). Kahjuks ei ole paljudel juhtudel sellised korrektsioonid võimalikud. Nendel juhtudel saab riskianalüüsi tulemusi kasutada vaid oma ametiau päästmiseks.

Riskianalüüs on suhteliselt kallis ja eeldab heade spetsialistide olemasolu – sellest tingituna on riskianalüüsi kasutamine väiksemates firmades ja väiksemate projektide korral üsna tagasihoidlik. Puudulike kogemuste ja mittekvaliteetsete eksperthinnangute korral ei pruugi ka saadud riskihinnangute usaldatavuse tase olla piisav.

3.9.3 Riski haldamine (*risk containment and risk management*)

Pärast seda, kui riskianalüüs on selgitanud erinevate riskifaktorite poolt põhjustatud riski suuruse, tuleb valida kõige olulisemad riskifaktorid ja püüda nendest põhjustatud riski vähendada. Riski haldamiseks on Behforooz and Hudson (1996) soovitanud viit erinevat moodust:

- Riski vältimine (*avoidance*)
- Riski tõkestamine (*prevention*)
- Riskiga leppimine (*assumption*)
- Riski ülekanne (*transfer*)
- Riski vastu valmistumine (*knowledge through research*)

Riski vältimine tähendab ebaõnnestumise tagajärgede vältimist (või vähendamist). Kui risk on liiga suur, võib loobuda projektist (pakkumise ja lepingu sõlmimise staadiumis), või katkestada projekt (kui tööd on juba alanud). Kahjuks ei ole see alati võimalik, või on seotud suurte trahvidega.

Riski tõkestamine (juhtimine) eeldab, et peamised projekti arengut iseloomustavad tegurid on pideva kontrolli all ja regulaarselt mõõdetakse (hinnatakse) tarkvaraprotsessi ja projekti parameetreid. Tehnilised ülevaatused toimuvad regulaarselt ja nendel arutatakse tegelikke riski suuruseid ning planeeritakse meetmeid nende vähendamiseks. Tavaliselt piisab plaani ja kulutuste regulaarsest jälgimisest – eelarvest ja plaanist mitte kinnipidamisel tuleb hakata asja detailsemalt uurima.

Riskiga leppimine seisneb riski olemasolu konstanteerimises ja valmistumises ebaõnnestumisega seotud täiendavate kulude katmiseks. Samast liigist abinõud on ka eelarve suurendamine, tööde tähtaja pikendamine (nii, et vähemalt osa riskiga seotud potentsiaalseid kulusid oleksid kaetud). Seda tüüpi olukord tekib tavaliselt lõplikult katsetamata tööriista kasutamisel, uue tarkvara platvormi kasutamisel (meeskonnal puudub sellega kogemus), jne.

Riski ülekanne tähendab paljudel juhtudel kindlustusfirmaga kindlustuslepingu sõlmimist – teatud sarnasus riskiga leppimise strateegiaga, ainult eelarve suurendatud osa makstakse kindlustusfirmale.

Riski vastu valmistumine tähendab tarkvaraprotsessi muutmist arvestades riski võimalusega. Näiteks, suurt riski lubavad tegevused on tarkvara plaanis erilise tähelepanu all; eriti riskialtide toote osade jaoks tehakse prototüübid juba projekti algstaadiumis (eeldusel, et kasutatav tarkvaraprotsess ei baseeru spiraalselt elutsükli mudelil); kriitiliste funktsioonide projekteerimiseks, kodeerimiseks ja katsetamiseks moodustatakse paralleelselt töötavad grupid; rakendatakse verifitseerimist ja toote omaduste hindamist simuleerimise abil.

Rusikareegli kohaselt, kui riski “tõenäosus” on 0.3 või väiksem, loetakse seda minimaalseks riskiks. Rutiinsed vahendid (regulaarne tarkvaraprotsessi jälgimine ja mõõtmine ning regulaarsed tehnilised ülevaatused) on piisavad nii madala riski haldamiseks.

Risk, mille “tõenäosus” on vahemikus [0.3 , 0.7], on keskmine. Sellise riski korral tuleb tehnilisteks ülevaatusteks koostada riskianalüüsi aruanded ja esitada ka riskihaldamise plaan. Riskihaldamise plaan peab fikseerima täiendavad tarkvaraprotsessi mõõtmised, toote omaduste hindamised/mõõtmised, jne, mis võimaldaksid varakult hoiatada riski tegeliku realiseerumise eest.

Kui riski “tõenäosus “ on suurem kui 0.7, tuleb lisaks eelmises lõigus öeldule teavitada ka firma kõrgemat juhtkonda. Kõrgem juhtkond peaks nimetama ekspertidest inspektsioonigrupi, arutama olukorda tellijaga, vajadusel eraldama projektile täiendavaid ressursse, või katkestama projekti.

3.10 Tarkvara meetrika ja riski haldamise kordamisküsimused

1. Millega tegeleb tarkvara meetrika?
2. Milles avaldub tarkvara meetrika kasulikkus tarkvara projekti ettevalmistamisel ja tegemisel? Tarkavara firma töö korraldamisel?
3. Mille poolest erinevad haldusmeetrika ja kvaliteedimeetrika?
4. Tooge üks näide, kuidas tarkvara meetrikas hinnatakse tarkvaratoote keerukust?

5. Loetlege parameetreid mille abil kirjeldab tarkvara meetrika tarkvaratoodet?
6. Millised koefitsiendid iseloomustavad tarkvara meetrikas arenduskeskkonda ja projekti meeskonda ?
7. Millised on Teie arvates peamised tarkvara projekti riskifaktorid? Põhjendage oma valikut.
8. Mis on riski mudel ja milleks teda kasutatakse?
9. Milles seisneb riski haldamine? Milliseid riskihaldamise mooduseid Te teate?

3.11 Kolmandas osas kasutatud kirjanduse loetelu

Behforooz A. and F.J.Hudson (1996) " Software Engineering Fundamentals", Oxford University Press, 661 pp., ISBN 0-19-510539-7

Boehm B.W. (1981) " Software Engineering Economics" Prentice-Hall, Upper Saddle River, N.J.

Boehm B.W. (1984) "Software Engineering Economics", IEEE Transactions on Software Engineering, vol. SE-10, no.1, 4-21

Boehm B.W. (1987) " A Spiral Model of Software Development and Enhancement", in IEEE Tutorial on Software Engineering Project Management, Ed. R.H.Thayer, 1987, 128-142, TTÜ Raamatukogu kataloog, VB-63387

B.W.Boehm (1991) "Software Risk Management: Principles and Practice" IEEE Software, vol.8, no.1.

Bransby M. (1998) "Explosive lessons", IEE Computing and Control Engineering Journal, vol.9, no.2, 57-60

Brooks, F.P. Jr (1975) "The Mythical Man-Month", Addison-Wesley Publishing Co., Massachusetts, katkendid trükitud IEEE Tutorial on Software Engineering Project Management, Ed. R.H.Thayer, TTÜ Raamatukogu kataloogi no. VB-63387

Constantine L.L. and E.Yourdan (1979) "Structured Design", Englewood Cliffs NJ, Prentice Hall

K.A.Cori (1985) "Fundamentals of Master Scheduling for the Project Manager" Project Management Journal, June, 78-89. Reprinted in IEEE Tutorial "Software Engineering Management" Ed. R.H.Thayer, TTÜ Raamatukogu kataloog VB-63387.

Davis A.M. "Software Requirements Analysis and Specification" Prentice-Hall, Upper Saddle River,N.J. 1990

Giddings R.V. (1984) " Accommodating uncertainty in software design", Communications of the ACM, vol.27, no.5, 428-434

Hicks H.G. and C.R.Gullet (1981) "Management", McCraw-Hill Book Co, 668pp.

Jensen K., G.Rozenberg (1991) (Eds) High Level Petri nets. Theory and applications, Springer Verlag, 724pp.

Meyer B. (1996) “Reality: A cousin twice removed” IEEE Computer, vol.29, no.7, 96-97

Narayan R. (1988) “Data Dictionary Implementation, Use and Maintenance”, Prentice-Hall Mainframe Software series, Upper Saddle River, N.J.

Peterson J.L. (1981) “Petri net theory and the Modeling of Systems” Prentice-Hall, N.J.

Royce W.W. (1970) “Managing the Development of Large Software Systems: Concepts and Techniques”, reprinted in Proc. 11th International Conference on Software Engineering, Pittsburgh, 1989, 328-338, also reprinted in IEEE Tutorial on Software Engineering Project Management, ed. R.H.Thayer, 1987, 118-127, TTÜ Raamatukogu kataloogi nr. VB-63387

I.Sommerville (1992) “Software Engineering”, Addison-Wesley, 649 pp

Wordsworth J.B. (1994) “Software Development with Z: A Practical Approach to Formal Methods in Software Engineering” Addison-Wesley, Reading, Mass.