Tarkvara loomise etapid

Tarkvara loomise etappe on loetletud erineval viisil eelmistes materjalides. Kordamiseks:
- **Nõudete väljaselgitamine** *(requirements)*
- **Kavandamine** *(design)*
- **Teostamine** *(implementation)*
- **Valideerimine** *(validation)*
- **Tugi** *(support)*

Süsteemitehnika

Tarkvara loomise esimeste sammude kirjeldamiseks kasutatakse inglisekeelset väljendit *System Engineering*. Enne tarkvara arendamist on vaja **mõista "süsteemi"** - ümbrust, milles tarkvara funktsioneerima hakkab. Tegevus hõlmab nii **üldisemat nõuete väljaselgitamist** kui tarkvara **kavandamist kõrgemal tasemel.** Tuleb määrata riistvara, tarkvara, inimesed, andmestikud, protseduurid ja teised süsteemi elemendid. **NB! Osa süsteemi paikneb väljaspool arvutit!** Tuleb tuvastada tegevused (operatsioonid), mida süsteemis teha tuleb, analüüsida, spetsifitseerida, modeleerida, valideerida ja hooldada kõiki nõudeid. Sellega tegeleb süsteemiinsener, kes töötab koos kliendi, tulevaste kasutajate ja teiste võtmeisikutega. Enne ei saa hakata mingit tehnoloogiat üles ehitama, kui pole selge, millises ümbruses kõik funktsioneerima peab.

Süsteem ja tema ümbrus on omavahel väga tihedalt seotud. Kaks olulist põhjust, miks ümbrusega arvestada tuleb:
1. Tihti põhjustab süsteem muudatusi keskkonnas.
2. Samal ajal mõjutavad muudatused keskkonnas süsteemi funktsioneerimist.

Mõjud võivad olla nii füüsikalised kui ka organisatsioonilised. Tuues välja organisatsioonilised mõjud, saame järgmise nimekirja:
1. **Protsessi muutus.** Kas süsteem põhjustab muudatusi tööprotsessis? Kas on vaja koolitust? Kas keegi võib töökohast ilma jääda? Võimalikud on inimeste vastuseisud uuele süsteemile.
2. **Töö muutus.** Kas töötajate senised oskused muutuvad tarbetuks? Kas nad peavad muutma oma tööharjumusi? Nad võivad vastu hakata.
3. **Organisatsiooni muutus**. Kas muutub "poliitiline jõud" organisatsioonis? Kellest sõltub süsteem? Kes opereerib süsteemiga, millest sõltub kogu organisatsiooni elu?

Süsteemi nõuete kirjeldamine

Süsteeminõuedeid kirjeldades tuuakse välja:

Abstraktsed funktsionaalsed nõuded – väga üldisel tasemel mida süsteem teeb, detailsem kirjeldus tehakse alamsüsteemide tasemel ja sõltuvalt arendusmetoodikast hilisematel etappidel.
Süsteemi omadused – peamiselt mittefunktsionaalsed omadused, mis puudutavad kättesaadavust, jõudlust, turvalisust jms
Funktsionaalsused, mida süsteem ei pea tagama – ka seda on vahest kasulik välja tuua probleemide vältimiseks.

Süsteemi kavandamine

Peamised tegevused:

1. **Nõuete jaotamine osadeks** – seda saab kindlasti mitmel moel teha peale analüüsimist.
2. **Alamsüsteemide määramine** – üksikud allsüsteemid saavad üksi või koos teiste allsüsteemidega vastata kasutajate nõuetele. Jaotus võib sõltuda ka organisatsiooni olukorrast või keskkonna mõjudest.
3. **Nõuete jaotamine alamsüsteemide vahel.** Hea, kui saab kasutada esialgu tehtud loogilist jaotust, aga nt kui alamsüsteemi ostma hakata võivad nõudmised hoopis muutuda.
4. **Alamsüsteemide funktsionaalsuse määramine** – iga alamsüsteemi spetsiifilised funktsioonid. Samuti seosed alamsüsteemide vahel.
5. **Alamsüsteemide liideste defineerimine** - st võimalusi alamsüsteemide seostamiseks. Kui see on selge ja kokkulepitud, saab alamsüsteeme eraldi arendada.

Alamsüsteemide arendamine
Tuleb teha valmis või osta valmis komponendid. Saab osta osi ja neid kokku integreerida (COTS). Arendus toimub enamasti paralleelselt.

Süsteemi integratsioon
Eraldiseisvad alamsüsteemid ühendatakse kokku. See võib toimuda korraga, kuid mõistlikum on inkrementaalne lähenemine. Et vähendada tähtaegade riski ja lokaliseerida vigade teket.

Süsteemitehnika tulemuseks on reaalse süsteemi esitus: prototüüp, spetsifikatsioon, mudel. Oluline on, et ta iseloomustaks süsteemis toimuvaid protsesse ja läbiviidavaid protseduure, süsteemi funktsionaalsust ja süsteemi käitumist, samuti süsteemi arhitektuuri.

Nõuded tarkvarale (Software requirements)

Tarkvarale esitatavate nõuete väljaselgitamine on kahtlemata üks olulisemaid protseduure süsteemi arendamise käigus. Ilma selleta ei ole võimalik järgmisi samme astuda.

Tihti klassifitseeritakse nõuded tarkvarale:
funktsionaalseteks – teenused, mida süsteem osutab, kuidas reageerib sisendile või käitub teatud situatsioonis, vahest ka see, mida süsteem ei tee.
mittefunktsionaalseteks – piirangud teenustes ja funktsionaalsuses (aeg, arendus, standardid, ...)
valdkonnanõueteks - peegeldavad valdkonna iseloomu ja võivad olla nii funktsionaalsed kui ka mittefunktsionaalsed.

Nõuete üleskirjutamiseks võib leida väga erinevaid eeskujusid. Erinevad arendusmetoodikad pakuvad omi lahendusi. Aga enda/firma jaoks oleks tark kehtestada kohalik standard (numeratsioon, kujundus jms, et oleks lihtsam ja ülevaatlikum). Kasutada selget keelt. Eristada tuleks selgelt vajalikke ja soovitatavaid nõudeid. Teksti kujundada mugavamaks lugemiseks. Vältida võimaluste piires žargooni – nii arvutiteemalist kui ka valdkonnaspetsiifilist. **Saab kirjutada** (igal variandil on oma head ja vead, oluline on, et klient ka aru saaks):

- loomulikus keeles
- erilises kirjelduse keeles (nagu programmeerimiskeel, aga abstraktsem)

- graafilises keeles
- matemaatiliste spetsifikatsioonide abil.

Tarkvara nõuete kindlaks tegemine ei ole ühekordne tegevus, vaid protsess, mis järkub terve tarkvara elutsükli vältel, kindlasti algab ta projekti alguses.

Mis on tulemuseks?
Tulemused sõltuvad taas metoodikast. Tehakse andmemudel, infoliikumise ja juhtimise mudel ning käitumise mudel. Määratakse tarkvara liides teiste süsteemi osadega ning piirangud, millele tarkvara peab vastama. Tarkvara kavandajale antakse ette info, mille saab muuta andmekavandiks, arhitektuuri kavandiks, liides(t)e kavandiks ja komponentide kavandiks. Tekkinud mudeli(te) alusel peab olema võimalik hinnata valmis tarkvara kvaliteeti ennekõike tagatava funktsionaalsuse aspektist.

Kuidas?
Kõigi nende tegevuste jaoks kasutatakse analüüsi ja kavandamise käigus järeleproovitud printsiipe, tehnikaid, keeli ja tööriistu. Seda tööd võib teha edasi tarkvarainsener, kuid ka mõni uus tegelane, süsteemi analüütik. Korraliku analüüsita võib olla tulemuseks tarkvara, mis on küll väga hea ja ilus (töötab väga hästi), kuid lahendab valet probleemi. Klientide abil täpsustatakse andmed, funktsionaalsus ja käitumine. Peale tarkvara mudeli loomist teda valideeritakse nii tarkvarainseneri kui kliendi poolt. Mudeli saab esitada kasutades prototüüpi, spetsifikatsiooni ja/või sümbolkeelt.

Soovide/nõuete väljatoomise tehnika pakub konkreetseid mehanisme, et soovid selgeks ja realiseeritavaks saaksid:
- saada aru, mida klient tahab;
- analüüsida kliendi soove;
- hinnata soovide teostatavust;
- rääkida läbi mõistliku lahenduse leidmiseks;
- spetsifitseerida lahendus ühetähenduslikult;
- valideerida spetsifikatsioon;
- hallata nõudeid ka sel ajal, kui nad saavad tegelikuks tarkvarasüsteemiks.

Nõuete esmane väljaselgitamine. Peamised probleemid, mis tekivad nõuete uurimisel:
- **Piirkonna probleem**. Süsteemi piirid on segaselt kirjeldatud, kliendid külvavad sind üle mõtetute detailidega ja oluline läheb kaotsi.
- **Arusaamise probleem**. Klient ei ole päris kindel selles, mida ta tahab. Ta ei mõista riistvarast tingitud piiranguid ja võimalusi. Ta ei tunne täpselt probleemi valdkonda või on raskustes selle selgitamisega küsijale. Ta jätab ütlemate selle info, mis on tema arvates ilmselge. Ta esitab soove, mis on vastuolus teiste klientide soovidega või on hoopis mitmeti mõistetavad.
- **Muutlikkuse probleem**. Kust tuul, sealt meel. Soovid muutuvad aja jooksul. Alati ei saa küll selles klienti süüdistada – on ka objektiivseid põhjuseid.

Soovitused probleemide lahendamiseks:
- Hinda kirjeldatud süsteemi majanduslikku ja tehnilist läbiviidavust
- Leia õiged inimesed, kes aitavad süsteemi vajadusi spetsifitseerida ja aru saada nende mõjust organisatsioonis.

Tarkvara loomise etapid

- Kirjelda tehniline ümbrus (arvutid, op-süsteemid, kommunikatsioonivajadused, ...), kuhu süsteem pannakse.
- Määra valdkonna piirangud (süsteemi ümbritsevate toimingute iseloomustus), mis limiteerivad loodava süsteemi funktsionaalsust või jõudlust.
- Vali üks või mitu nõuete väljameelitamise meetodit.
- Nõua mitmete inimeste osalemist, et nõuded oleksid kirjeldatud erinevate vaatenurkade alt ja veendu, et on leitud loogiline põhjus igale kirjapandud nõudele.
- Määratle vasturääkivad nõuded, kus oleks ilmselt vaja kasutada prototüüpimist.
- Loo kasutaja stsenaariumid, et aidata kliendil olulisemaid nõudeid identifitseerida.

Tarkvarale esitatavate nõuete kindlakstegemiseks on väljapakutud mitmeid tehnikaid. Sellest räägitakse kindlasti midagi ka infosüsteemide kursuses.

Nõuete uurimise tehnikad (Pressmanni järgi)

- **Intevjuu** – On erinevaid soovitusi, kuidas seda teha. Näiteks alustada kontekstivabade küsimustega. Nende abil püütakse süsteemist üldiselt aru saada. Kes on selle tö tellimise taga? Kes hakkavad lahendust kasutama? Milline on lahenduse kasutamise majanduslik kasu? Kas võiks olla ka mõni teine lahendus? Nende küsimuste abil leitakse võtmeisikud.
  Milline oleks hea väljund, mille lahendus genereerima peaks? Millises keskkonnas lahendust kasutama hakatakse? Kas kasutamist mõjutavad mingid piirangud või jõudluse probleemid? Need küsimused peavad andma parema arusaamise ja panema klienti oma arvamust lahenduse kohta välja ütlema.
  Lõpuks kokkuvõte koosolekust, nö "metaküsimused" Kas oled õige inimene nendele küsimustele vastama? Kas Sinu vastused on ametlikud? Kas esitatud küsimused on Sinu probleemile vastavad? Kas küsimusi oli liiga palju? Kas keegi võiks veel jagada täiendavat infot? Keda tuleks veel küsitleda?
  Kindlasti tuleks küsimusi esitada paljudele inimestele, kes lahenduse kasutamisega seotud saavad olema. Kirjeldatud küsimused on alles protsessi algus. Ja ainult küsimuste läbi lõpuni ei jõua. Edasi tuleb hakata kasutama ka teisi elemente.
- **Stsenaariumid** – need on pikemad protseduuride kirjeldused (kuidas organisatsioonis seda või teist toimingut läbi viiakse), nad tekitavad konteksti. Stsenaariume saab hakata looma siis, kui midagi hakkab selgemaks saama, tarkvarainsener saab uurida, millisel viisil midagi tehakse, mis juhtub siis, kui ...; püütakse siduda OOAnalüüsiga ja kasutuslugudega.
- **Prototüüpimine** – luuakse tarkvara mudel, millest saab edasi arendada projekti. Peale ajurünnakut jms luuakse tarkvara prototüüp. See antakse kliendile ja arendajale hinnata. Erinevad prototüüpimise meetodid on *throwaway prototyping* ja *evolutionary prototyping.* Esimene on lihtsalt nõuete ning nõuetest arusaamise demonstreerimiseks ja see visatakse ära ning tarkvara hakatakse arendama nö otsast peale, teisi tehnikaid ja vahendeid kasutades. Teine variant kasutab prototüüpi kui ühte osa analüüsi tegevuses, mis areneb edasi läbi kavandamise ja realisatsiooni. Iga tarkvara ei sobi arendada proptotüüpimise läbi (ei sobi kumbki variant). Sobib reeglina selline tarkvara, mis on dünaamiline, interaktiivne, suhtleb palju kasutajaga. Samas ei tohiks prototüübi tegemine liiga keeruline olla ja palju aega võtta, ta ei saa olla eesmärk omaette. Sellega viiakse kasutaja rohkem sarnasesse olukorda, kus ta tegelikult töötama hakkab ja kasutaja saab ehk ka aru, millist infot temalt oodatakse – tarkvara väljanägemise joonised paberil, TV beeta-versioonid. Prototüüpe saab kasutada järgneva vestluse käigus. Näiteks peale intervjuusid küsimaks kliendilt, kas ta nii mõtleski?

Tarkvara loomise etapid

- **Suuremad kohtumised** – ka *(Facilitated Application Specification Techniques, FAST)* sellesse haaratakse rohkem inimesi lootuses saada suuremat summarset efekti kui isiklikel kohtumistel. Püütakse teha klient ja tarkvaraarendaja "lähedasemateks", et ei oleks suhtumist "meie ja nemad". Seda meetodit kasutab tihti infosüstemide seltskond, kuid sobib ka muu tarkvara jaoks.
  Soovitused kohtumiste läbiviimiseks: Kohtumine toimub nö neutraalsel pinnal ja kohal on mõlemad pooled. Määratakse ettevalmistamise ja osalemise reeglid. On ka esialgne tegevuskava, kuid see võib töö käigus muutuda. Keegi peab juhtima kohtumist (pool pole oluline). Eesmärgiks on identifitseerida probleem. Pakkuda välja lahenduse osi, rääkida läbi erinevate võimaluste osas ja määrata esimene nõuete hulk.
  Huvitava tegevusena selle juures võib ära märkida – enne kokkusaamist saavad kõik osalejad ülesandeks teha nimekirjad objektidest, mis süsteemi ümbritsevad, mida süsteemi poolt toodetakse ja mida süsteem kasutab toimimiseks. Lisaks palutakse teha nimekiri teenustest, mis manipuleerivad või suhtlevad nende objektidega. Lõpuks nimekiri piirangutest (maksumus, suurus, reeglid) ja jõudlusest (kiirus, korrektsus). Nimekirjad pole loomulikult lõplikud. Need pannakse kokku, jaotatakse minimeeskondade vahel edasi arutamiseks. Või hakatakse hoopis nendne pealt kasutuslugusid tegema. Ka järgnevad sammud on määratud ja nendest tasub lugeda juba konkreetsetest materjalidest. Ajurünnakud ja ideede täpsustamised. Koosoleks aitab varem leida konfliktseid nõudeid, siiski nõuab oskust neid konflikte ära tunda. Selliseid koosolekuid tuleb aga osata väga hoolikalt juhtida, et kriitilisemad nõudmised ei jääks grupi lojaalsuse tõttu esitamata või et sõna ei võtaks ainult tähtsamad/vanemad tegelased.
- **Jälgimine** – vajalik, et paremini tajuda konteksti, kus süsteem peab tööle hakkama. TV insener jälgib töökohal, kuidas inimesed suhtlevad üksteisega ja süsteemiga. Selline meetod on kallis, kuid samal ajal väga vajalik, sest mõnede keerulisemate toimingute puhul ei oska ka asjaosalised ise ära seletada, mis toimub.

Keeruline on ka nõuete analüüs (inseneri kodutöö) ja selle alusel läbirääkimised, et lahendada konfliktid. Võib leida mitmeid küsimustikke, mille alusel nõudeid töötlema hakata.

Nõuete spetsifikatsiooni **üleskirjutamiseks** pakutakse erinevaid tehnikaid. Siin on ülekaalus kirjalikud ülestähendused. Kasutatakse selle valdkonna mõisteid, mille jaoks süsteemi ehitama hakatakse, sest sellest kirjatükist peavad ka süsteemi tellijad aru saama. Võivad kaasneda erinevad tehised: mõistemudel valdkonna paremaks kirjeldamiseks, stsenaariumid, ... Nõudeid tuleb eelnevalt väga hoolikalt hinnata ja uurida, sest see paber on ka aluseks süsteemi mahu ja maksumuse määramisel. Dokumendi struktuuri määramiseks on mitmed standardid, millele süsteemi loomisel toetuda saab.

Spetsifikatsiooni väljanägemise osas ollakse erineval arvamusel – osa autoreid pooldab standardmallide kasutamist, et nõuded oleksid esitatud ühtses ja ehk ka arusaadavamas vormis. Teised pooldavad vabamat lähenemist. Igal juhul on tulemuseks süsteemi spetsifikatsioon.

Spetsifikatsiooni loomiseks on omad **printsiibid**. Sest sellest sõltub ka toote kvaliteet.
1. Eralda funktsionaalsus ja realisatsioon
2. Tee mudel, mis käitub nii nagu vaja ja mis näitab, kuidas süsteem erinevatele välistele stiimulitele reageerib
3. Tee selgeks, milline on keskkond, kus tarkvara töötama peab, kuidas teised "komponendid" suhtlevad tarkvaraga.

4. Loo ”tunnetuslik” mudel, mitte realisatsiooni mudel. Kirjelda, kuidas süsteemi näevad kasutajad.
5. Spetsifikatsioon on alati mudel – st abstraktsioon palju keerulisemale tegelikkusele. Seepärast ei ole ta täisulik ja võib olla erineval detailsuse astmel.
6. Organiseeri spetsifikatsioon nii, et seda oleks mugav muuta.

Kuidas nõudmisi esitada?
1. Esituse formaat ja sisu peavad vastama probleemile. Süsteemi saab enamasti mingis sümbolkeeles ka üles joonistada, kuid erinevate süsteemide jaoks on erinevad sümbolid.
2. Informatioon peab spets-s olema korrastatud erinevaid tasemeid arvestades, sellele vastaku ka mingi numeratsioon (peatükid, joonised, skeemid, tabelid, ...) jms.
3. Diagrammid ja muud üleskirjutused peavad olema reglementeeritud ja piiratud. Ei saa joonistada nii nagu parasjagu juhtub.
4. Spetsifikatsioon peab olema muudetav. Ta muutub. CASE-vahendid lubavad tavaliselt teha igasuguseid muudatusi.

Nõuete dokumendi valideerimine on väga oluline. Grupp “seltsimehi” loeb dokumendi läbi (sh keegi tellijate poolelt) ja püüab leida nõrku kohti, sh ka sõnastusest. Vajalik on leida mitmeti mõistetavad asjad, kahe silma vahele jäänud vajadused, vead ja nende parandused. Teine võimalus kliendiga suhtlemiseks on prototüüpimine. See võtab rohkem aega ja võib tõmmata tähelepanu valedele asjadele (näiteks sisu asemel tarkvara väljanägemisele).

Kasutusmallid *(use-cases)*
Kuhugi nõuete leidmise ja tarkvara kavandamise piirimaile jääb kasutusmallide kokkupanemine. Samuti stsenaariumide koostamine, kuidas süsteemi töötegemisel kasutatakse. Kõigepealt leitakse erinevat tüüpi inimesed (rollid) või seadmed, mis süsteemiga suhtlema hakkavad. Et aga kasutuslugudest seoses vastava skeemiga taas juttu tuleb, siis ei kirjutaks sellest siin rohkem. Samas ei pea kasutuslood olema mingi skeemiga seotud.

Tarkvara kavandamine (Software Design)

Tarkvara disain (kavandamine) on vastavalt IEEE definitsioonile *“süsteemi või komponentide arhitektuuri, osade, liideste ja teiste omaduste määramine”* Ingliskeelne sõna *design* kajastab ka selle tegevuse produkti, eestikeeli peaks siis tulemuseks olema **kavand** või **projekt.** Tarkvara elutsüklis on kavandmine protsess, kus nõudmisi analüüsitakse, et luua tarkvara sisemise struktuuri ja organisatsiooni kirjeldus. Loodud kirjeldus on omakorda realisatsiooni aluseks. Tarkvara projekt peab kirjeldama süsteemi arhitektuuri, st kuidas süsteem on jaotatud osadeks (komponentideks) ning millised on komponentide liidesed. Komponendid peavad olema kirjeldatud sellise täpsusega, mis lubaks hakata neid realiseerima.

Klassikalises tarkvara elutsüklis vastavalt standardile ISO/IEC 12207 *Software life cycle processes*, on kavandamise osa jaotatud **kahte etapp**i:
- **arhitektuuri kavandamine**, millega määratakse nö kõrgemal tasemel kindlaks komponendid, seosed suuremate ja üldisemate tarkvara komponentide vahel; siin saab kasutada kavandamise mustreid *(design pattern)*, arvestada tuleb ka piirangutega
- **detailsem** kavandamine, millega täpsustatakse komponentide ülesehitus (protseduurid, objektid,

jms.
Nendele kahele võivad lisanduda veel:
andmete kavandamine, millega valdkonna infomudel muudetakse kasutatavateks andmestruktuurideks. Andmeobjektide ja nende vaheliste seoste esitamiseks saab kasutada nt ERD-diagramme.
liidese kavandamine – kuidas tarkvara sees osad üksteisega suhtlevad (kuidas tarkvara iseendaga kommunikeerub); see sisaldab ka info liikumise kirjeldust.
Tekkinud kavandit on reeglina võimalik analüüsida, et kas ta täidab tarkvarale esitatud nõudeid. Võimalik ka, et tekib mitu kavandit, mida siis võrrelda saab.
Korralik kavand peab tagama tekkiva tarkvara kvaliteedi. Ka kavandi järgi peab saama hinnata tarkvara kvaliteeti. Kavand on aluseks kõigile järgmisteel sammudele.

Kavandamise protsess

Tavaliselt on tegemist iteratiivse protsessiga, kus nõudmistest saab täpne projekt. Hea kavandi iseloomustamiseks on järgmised mõtted:
- Kavand peab realiseerima kõik analüüsi mudelis olevad nõudeid ja hõlmama kõiki kasutaja soove.
- Kavand peab olema loetav ja arusaadav kodeerijatele, testijatele ja tehnilisele toele.
- Kavand peab andma tarkvarast tervikliku pildi: nii andmetest, funktsionaalsusest kui käitumisest realisatsiooni vaatest lähtudes.

Lähema suunised räägivad kvaliteedi näitajad rohkem lahti.

Meetodid

Ajalooliselt kujunenud mitmed meetodid, kus igaüks annab suuniseid selleks,
- kuidas analüüsi mudelist saada kavand,
- kuidas esitada kompnente ja nende liideseid,
- kuidas täpsustada ja tükeldada,
- kuidas hinnata kvaliteeti.

Igal meetodil on omad printsiibid ja kasutatav mõistete baas.

Tuntumad meetodid
- **Funktsionaalne/struktuurne kavandamine** *(Function-oriented e structured design)* – tarkvaraarenduse klassika. Dekompositsioon keskendub süsteemi oluliste funktsioonide, nende väljatöötamisele ja täpsustamisele ülalt-alla *(top-down)* meetodil. Struktuurne kavandamine järgneb tavaliselt struktuursele analüüsile, mille käigus on joonistatud andmevoo diagramme ja sellega seotud protseduuride kirjeldusi. Andmevoodiagrammid muudetakse tarkvara-arhitektuuriks kasutades struktuuriskeeme *(structure chart)*.
- **Objekt-orienteeritud kavandamine** – objektidel põhinevaid meetodeid on lõpmata palju, neid kasutatakse 1980-te keskelt alates. Võtmesõnadeks on pärimine ja polümorfism, sellest on jõutud komponent-põhise kavandamiseni, kus metainformatiooni saab talletada ja sellele ligi pääseda. Siiski OOD juured on andmeabstraktioonis
- **Andmestruktuurikeskne kavandamine** – omapärane variant, kus kõigepealt joonistatakse valmis andmestruktuurid, kasutades nt Jacksoni skeeme ning programmi juhtstruktuurid luuakse nendele skeemidele tuginedes.

Tarkvara kavandamise printsiibid

Printsiip on põhitõde või üldine seadus, mida saab kasutada arutluste baasiks või tegutsemisjuhiseks.

- Kavandamise protsess ei saa toimuda tunnelis. Disainer peab oskama näha alternatiive
- Kavandist peaks saama tagasi jõuda analüüsi mudelini, sest nõudmiste ja kavandi osade vahel ei ole üksühest vastavust.
- Ei tohi leiutada jalgratast. Vaata kavandamise mustreid või seda, mida oled juba teinud. Aega on vähe, juba tehtu kasutamine võib tagada parema kvaliteedi.
- Kavand peaks väljanägema ühtlane, nagu oleks selle teinud üks inimene.
- Kavand peaks olema struktureeritud nii, et teda on hea muuta.
- Kavandamine ei ole kodeerimine ja vastupidi! Isegi protseduure kavandades ei tohi minna liiga detailseks.

Kavandit tuleb üle vaadata, et vähendada semantika (tähenduse) vigu.

Kavandamise põhimõisted

Viimaste aastakümnete jooksul on tekkinud mitmed tarkvara kavandamise põhimõisted (kontseptid).

- **Abstraktsioon** on üks peamisi teid, kuidas inimene saab hakkama keerukusega (G. Booch) On protseduurne abstraktsioon ja andmeabstraktsioon.
- **Eraldatus** ja **sidusus**
  sidusus (*cohesion)* – üks moodul teeb ühte tööd ning ta vajab võimalikult vähe suhtlemist teiste protseduuridega.See on mõõt, kui tugevalt on omavahel seotud ühes moodulis olevad komponendid. Sidusus peab olema suur *(high cohesion)* – see on hea.
  Sõltuvus (*coupling)* – erinevate moodulite vahelise seose mõõt, kui keerulised on moodulite vahelised liidesed, kui palju infot moodulite vahel "sõidab"; mida väiksem sõltuvus *(low coupling)*, seda parem. Väike on sõltuvus siis, kui kaks moodulit suhtlevad vaid selleks ettenähtud liidese kaudu.
- **Dekompositsioon** ja **modulariseerimine** – dekopositsioon on suure süsteemi jaotamine mingiks arvuks väiksemateks sõltumatuteks süsteemideks. Eesmärgiks paigutada erinevad funktsionaalsused erinevatesse komponentidesse.
- **Kapseldamine** ja **info peitmine** – moodulid/objektid luuakse nii, et mooduli sisu pole ligipääsetav teistele moodulitele; eriti suur kasu on sellest testimisel ja hilisemal hooldusetapil, samuti osade korduskasutamisel.
- **Liidese ja realisatsiooni eraldamine** – komponent kirjeldatakse oma liidese kaudu, eraldatult realisatsiooni detailidest, komponenti tuleb ja saab kasutada vaid läbi liidese.
- **Piisavus, täiuslikkus ja lihtsus** – komponent tehku ainult seda, mida ta peab tegema, ei ole vaja liiga palju erinevaid funktsionaalsusi ühte komponenti koondada.

Peenendamine – *(Refinement)* on ülalt-alla kavandamise strateegia. Algselt on temast räägitud protseduurses võtmes. Sama ideed kasutab ka nõuete peenendamine ja see on tegelikult väljatöötamise protsess, kus kõrgemal tasemel kirjeldatud funktsionaalsuse abstraktsioonist saadakse lõpuks täpsemad laused. Esialgu pannakse kirja üldiselt "Õpetaja haldab hindeid." Edasi täpsustatakse, mis on hinnete haldamine: Hinde lisamine, muutmine ja kustutamine jms. Lõpuks pannakse stsenaariumina kirja, kuidas toimub hinde panemine.
Modulaarsus – *(Modularity) "Modulaarsus on tarkvara omadus, mis laseb programmil olla intellektuaalselt hallatav."*

`C(p1+p2) > C(p1) + C(p2)` – kahe probleemi keerukus *(Complexity)* koos (`C(p1+p2)`) on suurem, kui ühe probleemi keerukus + teise probleemi keerukus.
See on teadmine inimese oskusest ja võimest probleeme lahendada. Mis viib meid omakorda *"Divide et impera"* **(jaga ja valitse)** printsiibi juurde, mida juba vanad roomlased edukalt kasutasid. Mitte küll tarkvaraarenduses ;) Teisalt ei tohi üle modulariseerida, sest siis muutub liiga keeruliseks integreerimine.

Kavandi / projekti ülesmärkimine e dokumentatsioon

Mis on kavadis kirjas? Erinevad arendusmetoodikad esitavad omai nägemusi alates konkreetsetest dokumentide mallidest ja lõpetades sellega, et polegi vaja kõike dokumendiks kirjutada. Enamasti on seal järgmised osad.
Üldosa: mille jaoks kavand tehti?
Andmed: andmebaasi struktuur, failide struktuurid, seosed andmeobjektide ja failide vahel.
Arhitektuur: kuidas saadi analüüsi mudelist programmi arhitektuur, moodulite jms hierarhia
Liidesed: süsteemi sisesed liidesed ning liides inimesega (nt prototüübi abil)
Komponendid: eraldi seisvad protseduurid ja funktsioonid koos inimkeelse protseduuri kirjeldusega.
Leitavad peavad olema seosed nõudmistega (nt maatriksi kujul). Saame teada, kas kõik nõudmised on rahuldatud ja millised komponendid on kõigepealt vaja realiseerida.
Esimesed suunised testimiseks (testimisdokumentatsioon).
Piirangud mälu, kiiruse, välise liidese jne osas.

Tarkvara kavandi üleskirjutamiseks on palju ülesmärkimisviise ja keeli. Mõnda kasutatakse pigem struktuurse organisatsiooni kirjeldamiseks, mõnda käitumise kirjeldmiseks. Kuna tarkvarast luuakse mitu erinevat vaadet (loogiline vaade, protsessivaade, füüsiline vaade, arendusvaade), on tarvilikud ka erinevad üleskirjutamise võimalused.
Võimalikke vahendeid **struktuurse** (staatilise) ja **käitumisliku** (dünaamilise) vaate kirjeldamiseks:
Struktuurne kirjeldus – enamasti graafiline, kireldamaks struktuurset aspekti, peamisi komponente ja seoseid nende vahel.

- *Architecture Description Language* – keel komponentide ja nende vaheliste seoste kirjeldamiseks.
- **Klassi- ja objektiskeemid** - kasutatakse iseloomustamaks klasse ja nende vahelisi seoseid
- **CRC-kaardid** - klasside nimed, kohustused ja nendega koostööd tegevad klassid
- **Evitusskeem** – enamasti süsteemi füüsiliste sõlmede ja nende vaheliste seoste näitamiseks.
- **Olem-seos diagrammid** (ERD) – andmete kontsptuaalse mudeli esitamiseks infosüsteemis
- *Interface description Language* - keel, mille abil saab kirjeldada liidest (andmed, tüübid, operatsioonid)
- **Jacksoni skeemid** - kirjeldamaks andmete struktuure kasutades jada, valikut ja kordust.
- **Struktuurne diagramm** – kirjeldamaks programmis olevat väljakutsete struktuuri (millised protseduurid ja moodulid kutsuvad teisi välja ja kelle poolt neid ennnast välja kutsutakse).

Käitumise kirjeldus e dünaamiline vaade. Kirjeldatakse süsteemi ja komponentide dünaamilist käitumist

- **Tegevusskeemid** – näidata tegevuste kulgemist
- **Koostööskeemid** – näitamaks mingi grupi objektide vahel toimuvaid interaktsioone, kus

põhirõhk on seostel ja mööda seoseid saadetavatel teadetel
- **Andmevoodiagrammid** – näitamaks andmete liikumist protsesside vahel
- **Plokkskeemid** – näitamaks juhtimise käiku ja toimuvaid tegevusi
- **Formaalsed spetsifikatsioonikeeled** – keel, mis kasutab matemaatilisi tähistusi, et kirjeldada liideseid ja seoseid tarkvara komponentide vahel (eel- ja järeltingimused)
- **Järgnevusskeemid** – näitab grupi objektide omavahelisi seoseid rõhuga vahetatavate teadete vahelistele ajalisele järjestusele.

Realisatsiooni faas (Software construction)

Realisatsioonifaas on ilmselt kõige olulisem, sest sellega luuakse tarkvara ise. Faas järgneb kavandamisele ja tihti on raske nende kahe vahele jämedat joont tõmmata. Kui kavandamine ja peenendamine on jõudnud nii kaugele, et üksik arendaja võib ühe osa kallal tööle asuda, saab öelda, et on jõutud realisatsiooni faasi. Väiksem tarkvara, kus mõõduks ongi paras korraga realiseeritav tükk, ei pruugi nõuda eraldiseisvat kavandamisetappi. Samal ajal suured projektid võivad nõuda mitmeid iteratsioone kavandamise ja realisatsiooni vahel, kui luuakse rohkem prototüüpe ja neid kasutatakse või kõrvale heidetakse.
Realiseerimisvahendid on nii tarkvaralised kui ka riistvaralised. Hea vahend peaks parandama nii tootmise efektiivsust kui ka kvaliteeti. Tarkvara arendusvahendid on ise tarkvara ja kasutatavad realisatsiooni faasis:
- kompilaatorid,
- versioonide haldamine,
- silujad,
- koodigeneraatorid.
- spetsiaalsed tekstiredaktorid,
- testimist ja dokumenteerimist abistavad vahendid.

Realisatsioonifaasis tegeldakse ja on olulised ka järgmised tegevused ja mõisted:
Tarkvara evalveerimine: koodide läbivaatused, testimisplaanid, testimised, kvaliteedikindlustus ja mõõtmine. St realisatsiooni etapi sees peavad toimuma süstemaatilised kontrollid, et ikka olemasolev kood töötaks. Lihtsaim ja tuntuim evalveerimise vorm on mooduli testimine pärast mooduli valmimist. Kompilaatoripoolne kontroll, koodi ülevaatused, jms
Standardite vajalikkus
Edukaks suhtlemiseks on vaja kasutada ühist keelt. Standardi valik võib otseselt mõjutada realisatsiooni. Standardid võivad ka mõjutada programmeerimiskeele valikut.
Standardeid on mitmeks juhuks:
- programmeerimiskeelte ülesehituse jaoks
- koodikirjutamise jaoks
- andmete kirjeldamiseks ja kätte saamiseks (XML, SQL)
- kooditabelid
- dokumentatsioon
- protsesside vaheline kommunikatsioon
- andmeside

Realisatsioonitehnikad jagatakse manuaalseteks ja automaatseteks.
- **Manuaalne** tehnika on lihtsaimal juhul koodi kirjutamine programmeerija poolt. Igal juhul peab tulemuseks olema töötav kood.

- **Automaatne** realisatsioon – päris automaatselt koodi tekitada ei saa. Arvuti ei saa selle eest vastutada. Küll saab arvuti kaela veeretada realisatsiooni protsessi juhtimise ja jätta inimesele otsene kodeerimistöö. Ka suudab masin paremini leida ja kasutusele võtta korduvkasutatavaid komponente, mis tal laos olemas on. Automaatne realisatsioon ei ole tingimata odavam, sest kogu masinavärgi ülessättimine ja ka tarkvara maksumus, on üsna kõrge.

Suurem automatiseeritus on TT arengu üks eesmärkidest.

Tarkvara testimine (Software testing)

Kui lähtekood on valmis, tuleb tarkvara testida, et avastada ja kõrvaldada nii palju vigu kui vähegi võimalik enne kliendile üle andmist. Tuleb luua mitmeid *testcase'e*, mis võimalikult suure tõenäosusega vigu avastaksid. Vigu on vaja leida sisemises loogikas ja sisendis/väljundis (käitumises, jõudluses, funktsionaalsuses).
Testivad tarkvarainsenerid, kuid parem on eraldi testijad. Siiski esmase tarkvara osade testimise peavad tegema programmeerijad ise.
Vajalik on testimise teooria selleks, et leida õiged ja vajalikud testjuhtumid, mis süstemaatiliselt vigu otsiksid (ja leiaksid).
Toimub tarkvara kvaliteedi hindamine, vigade ja probleemide identifitseerimine. Peamine, mida ei tohi unustada, et testimisega ei saa tõestada, et tarkvaras vigu ei ole. Seetõttu on testimisel tihe seos hooldamisetapiga. Testimine tänapäevases mõistes peab saatma kogu arendusprotsessi.
Erinevad arendusmeetodid näevad testimiste erineval viisil. Näiteks XP lubab testid üldse enne kirjutada kui programmi ja vastavalt testidele toimub programmeerimine..

Testimise eesmärgid või reeglid:
1. Testimine on protsess, kus programmi käivitatakse eesmärgiga leida vigu.
2. Hea test on see, mis suure tõenäosusega leiab veel leidmata vea.
3. Edukas test on see, mis leiab veel leidmata vea.

Seega on vale arvata, et hästi läks siis, kui viga ei leitud.

Testimise printsiibid
1. Kõik testid peavad olema järgitavad kuni kasutaja nõudmisteni – need on vead, kui programmil ei õnnestu teha seda, mis oli kliendi sooviks.
2. Testid tuleb planeerida enne kui testimine algab (nt juba nõudmiste mudel on valmis) ja saada valmis enne kodeerimist.
3. Kehtib Pareto printsiip. 80% programmi vigadest paikneb 20%-s programmi komponentidest. Aga kuidas neid komponente üles leida?
4. Alusta väikesest testimisest ja mine edasi suurele testimisele: esialgu testitakse komponente, hiljem aga kogu süsteemi.
5. Ammendav testimine ei ole võimalik. Võimalikke programmi läbimise teid on hirmus palju ja me ei suuda programmi nii palju käivitada, et kõiki teede kombinatsioone läbida. Isegi keskmise suurusega programmi juures.
6. Parima ja efektiivsema testimisega saab hakkama keegi kolmas. St suurima tõenäosusega leiab võõras vigu. Eriti aga kasutaja. Sellest ka seos hooldamisetapiga.

Testimine jaguneb staatilisteks ja dünaamilisteks tehnikateks. Rohkem peetakse testimiseks siiski dünaamilist poolt – st testimist koos käivitamisega.

Dünaamilise testimise omadused
Dünaamiline – programmi käivitamine koos sisendiga. Tihti ainult sisendist ei aita, sest sõltuvalt programmi seisundist võib ta reageerida samale sisendile erinevalt. Seega on vajalik ka sisendi kirjeldus.
Lõplik – võimalike sisendite hulk on teoreetiliselt lõpmatu, nendest tuleb välja valida lõplik kogus teste. Testimise etapp on enamasti see, mille käigus püütakse tähtaegadele ja eelarvele mitte jalgu jääda (ja asi läheb kiireks). Testimine peab tagama piisava kindlustunde.
Valitud – testid valitakse vastavalt mingitele kriteeriumidele, sest erinevad valikud tagavad erineva efektiivsusega testimise
Oodatud – kuidagi peab saama määrata, kas testi tulemus on see mida oodati või mitte. Alati ei olegi seda nii lihtne määrata. Tarkvara vastus saab olla tarkvara nö pikem käitumine, mida tuleks võrrelda kasutajate ootustega ja teisalt spetsifikatsiooniga. Lõplik otsus kipub olema tihti subjektiivne või ka stiilis – rahuldab osaliselt.
Testimine on väga lai teema.
Testimise arengu oluline suund on automatiseerimine – see annab võimaluse kiiremaks ja põhjalikumaks testimiseks.

Tarkvara hooldamine (Software maintenance)

Tarkvara hooldamiseks nimetatakse tarkvara muutmist peale kliendile üleandmist, et parandada vigu, parandada jõudlust või muid omadusi, või kohandada toode muutunud keskkonnale.

Hoolduse eesmärk hoida tarkvara töös nii kaua kui võimalik. Ajalooliselt on sellele osale vähem tähelepanu pööratud. Üsna teravalt kerkis hooldamise probleem esile mitte väga ammu. Seoses uue aastatuhande tulekuga. Hooldamise juures on oluliseks aspektiks teiste programmeerijate poolt kirjutatud tarkvaraga töötamine (parandamine, täiendamine, muutmine). Hooldusele võib saada tuge avatud lähtekoodiga tarkvara maailmast, sest siingi tuleb tegelda palju teiste kirjutatud tarkvaraga.

Tarkvara tootmine lõppeb kliendile tarkvara üleandmisega. Valminud tarkvara peab olema selline, mida klient tahtis. Kuid tarkvara peab edasi arenema. Töötamise käigus leitakse anomaaliaid, muutub töö keskkond, tulevad uued nõuded.
Muudatuste vajadused logitakse, määratakse muudatuste mõju, kood muudetakse, tehakse testid, antakse välja tarkvara uus versioon ja vajadusel korraldatakse ka õpetus.