Tarkvarasüsteemide arhitektuur (LAI9710)

Ülevaade

Mis on tarkvara arhitektuur. Tarkvara arhitektuuri stiilid. Konkreetsete arhitektuuride näited. Arhitektuuri kirjeldamine. Arhitektuuri kirjeldamise keeled. Arhitektuuri hindamine. Arhitektuurile põhinev tarkvara tootmine ja tootepered.

Sisukord

1. Ülevaade kirjandusest
    1. Mary Shaw, David Garlan, Software Architecture. Perspectives on an Emerging Discipline
    2. Len Bass, Paul Clements, Rick Kazman, Software Architecture in Practice
2. Tarkvara arhitektuur.
    1. Arhitektuuri kirjeldamine
        1. Arhitektuuri alaste teadmiste esitamine
        2. Arhitektuuri kirjeldamise keeled
    2. Arhitektuurid konkreetsetele valdkondadele
    3. Arhitektuuri formaalsed alused
3. Arhitektuuristiilid.
4. Formaalsed mudelid ja spetsifikatsioonid
5. Arhitektuurile orienteeritud tarkvara tootmistsükkel

Ülevaade kirjandusest

Mary Shaw, David Garlan, **Software Architecture.** Perspectives on an Emerging Discipline, Prentice-Hall, 1996

1. Sissejuhatus

Tarkvara arhitektuuri mõiste määratletakse kui tarkvarasüsteemi komponentide, struktuuri, nende vaheliste vastasmõjude, nende ühendamise mallide ja piirangute hulk. Üldisemalt on tarkvarasüsteemi arhitektuur antud komponentide ja nende vaheliste vastasmõjude hulgaga. Komponentideks on kliendid, serverid, andmebaasid, filtrid ja hierarhilise süsteemi kihid. Vastasmõjud on lihtsad nagu alamprogrammi väljakutsed, andmete kasutamised jne. ning keerulised nagu klient-server protokollid jms.

Arhitektuurilised struktuurid on baasiks süsteemi kui terviku analüüsil. Lisaks süsteemi struktuuri ja topoloogia määratlemisele näitab arhitektuur ka sidet süsteemile esitatud nõuete (maht, läbilaskevõime, terviklikus jne.) ja süsteemi elementide vahel, olles aluseks disainiotsustele.

Tarkvara arhitektuuri alal võib eristada nelja tegevussuunda:

- arhitektuuri kirjeldamise keeled (arhitektuuri esitamine selle väljendamiseks ja analüüsiks),
- arhitektuuri alaste teadmiste kodifitseerimine (disainimallide katalogiseerimine),
- arhitektuurid/raamistikud konkreetsetele valdkondadele ja
- arhitektuuri formaalsed alused (uued arhitektuuri esitusviisid, formalismid arhitektuuride võrdluseks ja arutlusteks (*reasoning*) nende üle, ...).

Parima arhitektuuri valik antud probleemi (või valdkonna jaoks) jääb ikka veel avatud küsimuseks. Kuna erinevates stiilides pakendatakse komponente erinevalt pole need tavaliselt kasutatavad teise stiiliga arhitektuuris.

2. Arhitektuuri stiilid

Arhitektuuri disaini tunnuseks on, ühtse stiili moodustavate, süsteemi organisatsiooni mallide kasutamine. Et neid stiile võrrelda, käsitletakse arhitektuuri kui komponentide ja nende vastasmõjude e. ühenduste hulka. Komponentide näiteks on kliedid, serverid, filtrid, kihid, andmebaasid jms. Ühenduste näiteks on alamprogrammide väljakutsed, sündmused, sideprotokollid jms.

On hakanud kujunema arhitektuuristiilide liigitus. Levinumad arhitektuuristiilid on:

- andmevooarhitektuur (*dataflow*): pakett-jada arhitektuur (*batch sequential*), toru-filter arhitektuur (*pipes and filters*)
- väljakutsearhitektuur: peaprogramm ja alamprogrammid, objekt-orienteeritud süsteemid, hierarhilised kihid
- sõltumatute komponentidega arhitektuur: suhtlevad protsessid, sündmussüsteemid
- virtuaalsed masinad: interpretaatorid, reeglitel põhinevad süsteemid
- andmekeskne arhitektuur: andmebaasid, hüpertekstisüsteemid, tahvlisüsteemid (*blackboards*)

Muud arhitektuuristiilid on:

- hajussüsteemid, mille arhitektuure saab kategoriseerida kas topoloogia (nagu täht, ring või siin) või komponentide vaheliste protokollide (näit. "*heartbeat*") alusel -- üks sagedamaid vorme on klient-server;
- peaprogramm/alamprogrammid (paljude tarkvarasüsteemide arhitektuur peegeldab programmeerimiskeelt, milles süsteem on realiseeritud);
- valdkonnakohased arhitektuurid; viimasel ajal on hakanud tekkima huvi ja vajadus eri valdkondade (avioonika, juhtimissüsteemid jms.) jaoks nn. viitearhitektuuride (*reference architecture*) loomiseks kuna kitsendades rakendatavust teatud valdkonnaga on võimalik suurendada struktuuride väljendusliku võimsust; paljudel juhtudel saab täidetava süsteemi genereerida täis- või pool-automaatselt otse arhitektuuri kirjeldusest;
- olekumasinad on paljude reaktiivsete süsteemide organisatsiooni aluseks; need süsteemid on määratud olekute ja nende vaheliste üleminekute hulkadega.

Arhitektuuristiil koosneb komponentide ja ühenduste tüüpide sõnastikust ja nende kombineerimisele seatud piirangute hulgast. Paljudel stiilidel on olemas ka semantiline mudel, mis võimaldab süsteemi kui terviku omaduste tuletamist tema osade omadustest. Seega saab uurida arhitektuuri stiile vastates järgnevatele küsimustele:

- milline on komponentide ja ühenduste sõnastik?
- millised struktuurimallid on lubatud?
- milline on arvutusmudel?
- millised on stiili invariantsed omadused?
- millised on stiili tuntud kasutusnäited?
- millised on stiili head ja halvad küljed?

- millised on üldised erijuhud?

Heterogeensed arhitektuurid saadakse:

- erinevate arhitektuuristiilide kombineerimise (komponentide ja/või ühenduste hierarhilise dekompositsiooni) tulemusena,
- sama komponendi ühendamisel eri stiilidesse kuuluvate ühendustega (Unix'i filtrid võivad kasutada lisaks torudele ka failisüsteemi andmehoidlana) või
- kasutades erinevaid arhitektuuristiile erinevatel arhitektuuri kihtidel.

3. Arhitektuuride näited

Erinevate arhitektuurilahenduste omadused.

Parnas'e poolt 1972 süsteemide dekomponeerimise kriteeriumite uurimiseks püstitatud näiteülesanne "võtmesõna kontekstis" (*Key Word in Context*) kujutab endast süsteemi, mis järjestatud ridadest loob sorteeritud indeksi, milles iga rida on ringnihkega teisendatud iga reas esineva sõna jaoks. See ülesanne sobib ka illustreerima muudatuste (algoritmi ja andmete esituste) mõju erinevatele arhitektuuridele. Garlan, Kaiser ja Notkin kasutasid antud ülesannet ka kaudväljakutsetele põhineva arhitektuuri uurimisel vaadeldes süsteemi funktsioonide laiendamist, ajalist ja ruumilist effektiivsust ning korduvkasutamist.

Lahenduste võrdlus:

	Jagatud andmed	Abstraktsed andmetüübid	Kaud-väljakutse	Andmevoog (torud ja filtrid)
Muudatus algoritmis	-	-	+	+
Muudatus andmete esituses	-	+	-	-
Muudatus funktsionaalsuses	+	-	+	+
Effektiivsus	+	+	-	-
Korduvkasutamine	-	+	-	+

Mõõtetarkvara.

Vaadeldakse Tektronix'is välja töötatud ostsilloskoopide tarkvara arhitektuuri, mis on tüüpiline valdkonnakohane arhitektuur. Algselt töötati välja valdkonna objekt-orienteeritud mudel (see aga ei aidanud funktsionaalsuse dekomponeerimisel), järgmises faasis töötati välja ostsilloskoobi kihiline mudel (see oli aga liigselt kitsendav kuna tegelikuses peab kasutaja ostsilloskoobiga töötamisel omama juurdepääsu erinevatele funktsionaalsetele tasemetele). Kolmanda katse tulemuseks oli andmevoo mudel, mis vaatles ostsilloskoobi funktsioone filtritena (selle mudeli heaks küljeks oli funktsionaalne paindlikus ja vastavus kasutaja intuitsioonile, probleemiks oli aga kasutaja liides). Neljas lahendus lisas igale filtrile juhtliidese, mille kaudu väline osapool võis filtri tööd juhtida (see lahendas suures osas kasutajaliidese probleemi ja ühendas signaalitöötluse

kasutajaliidesest lahti). Andmevoo arhitektuuri suurimaks puuduseks jäi halb effektiivsus, millest ülesaamiseks tüpiseeriti torud luues eri "värvi" ühendused, millest osa lubas andmeid töödelda kopeerimata või ignoreerida andmeid mida filter ei suutnud töödelda.
Antud näide kirjeldab probleeme, mis valdkonnakohase arhitektuuri väljatöötamisel üles kerkivad ja seda kuidas eri arhitektuuristiilid samal probleemide hulgal käituvad. Samuti nähtub reaalsed arhitektuurid pole tavaliselt puhtalt ühte stiili.

Iseliikuvad robotid (Marco Schumacher)

Nõuded iseliikuva roboti tarkvara arhitektuurile:

- peab võimaldama tahtlike ja põhjustatud (reaktiivseid) tegevusi (et saavutada sihti)
- peab võimaldama määramatust (ennustamatud olukorrad ja ebatäielikk info)
- peab arvestama ohte ja riske (töökindlus, ohutus jms.)
- peab olema väljatöötluses paindlik (mobiilse roboti ehitamine on sageli seotud katsetamisega)

Lahenduste võrdlus:

	Juhtimistsükkel	Kihiline arhitektuur	Kaud-väljakutse	Tahvliarhitektuur
Tegevuste koordineerimine	+-	-	++	+
Määramatus	-	+-	+-	+
Töökindlus	+-	+-	++	+
Ohutus	+-	+-	++	+
Effektiivsus	+-	+-	+-	+
Paindlikus	+-	-	-	+

Sageli kasutatakse hübriidseid arhitektuure, näiteks NASA/NBS telerobotite viitemudel (*reference model*) NASREM kasutab juhtimistsükli ja kihtide kombinatsiooni.

Sõiduki kiiruse säilitamine

Sõiduki kiiruse säilitamine on lahendatav tüüpilise protsessijuhtimise tsükliga. Algülesanne ise on saada sisendväärtustest (rataste kiirus, kell, soovitav kiiruse muutus, jms.) väljundisse väärtus, mis juhib liikumist põhjustavat mootorit.
Booch on vaadelnud kiiruse säilitamise süsteemi objekt-orienteeritud dekompositsiooni lähtudes ülesande kirjeldusest, meie vaatleme kiiruse säilitamise ülesannet protsessijuhtimise seisukohalt.

Protsessijuhtimise arhitektuuri osadeks on:

- arvutuslikud elemendid
    - protsessi määrang -- protsess võtab soovitava kiiruse sisendiks ja juhib sõiduki kiirust

    - juhtimisalgoritm -- algoritm modelleerib jooksvat kiirust vastavalt anduritele, võrdleb seda soovitava kiirusega ja juhib liikumist põhjustavat mootorit; algoritm peab säilitama jooksvaid juhtimisparameetreid
- andmeelemendid (protsessi muutujad/parameetrid
    - juhitavad muutujad -- sõiduki jooksev kiirus
    - väljundmuutujad -- mootori kiiruse seade
    - seadepunkt -- soovitav kiirus
    - andurid -- rataste kiiruse andur
- juhtimistsükkel

Sobiv disainistrateegia on kasutada protsessijuhtimise arhitektuuristiili süsteemi kui terviku jaoks ja teisi arhitektuure nagu objektarhitektuuri ning olekumasinaid juhtimistsükli osade täpsustamisel.

Mingi arhitektuuri valik põhjustab probleemi vaatlemise teatava nurga alt. Iga abstraktsioon rõhutab probleemi mingeid aspekte ja surub teised maha. Näiteks objekt-orienteeritud arhitektuur vaatleb süsteemi dekomponeerimist objektideks, mida iseloomustab nende käitumine suhtes teiste objektidega, aga funktsionaalsed meetodid dekomponeerivad süsteemi mooduliteks, mis kujutavad endast kogu protsessi alametappe.

Tähtis on otsustada milline abstraktsioon on antud probleemi jaoks kõige kasulikum. Antud juhtumil lihtsustab protsessijuhtimise arhitektuuri kasutamine probleemi mitmel moel:

- juhib meid väljundit ümberspetsifitseerima kui sõiduki tegeliku kiirust
- juhtimise eristamine protsessist toob ilmutatud kujul välja tegeliku kiiruse mudeli probleemid
- juhtimisalgoritmi eristamine viib küsimusele, millist juhtimisalgoritmi kasutada
- ülesande vaatlemine juhtimisülesandena teeb selgemaks sisendite ja väljundite rollid ning tagasiside
- juhtimissüsteemis on selgelt eristatud manuaalne juhtimine automaatsest
- seadepunkti leidmine on lihtsustatud, kui see on juhtimisest eristatud

Metodoloogia peaks aitama disaineril otsustada, milline arhitektuur on sobiv, leida disaini elemendid ja nende vahelised seosed ning leida kriitilised disainiotsused.
Nii nagu objekt-orienteeritud arhitektuure toetab vastav metodoloogia, nii ka Åström ja Wittenmark kirjeldavad tüüpilisi protsessijuhtimise süsteemide lahendusi ja soovitavad:

- valida juhtimispõhimõtte
- valida juhtmuutujad
- valida mõõdetavad suurused
- luua alamsüsteemid

Metodoloogia peaks samuti sisaldama juhendit süsteemi muutmiseks.

Kolm näidet segatud arhitektuuristiilidest

1. Kihiline arhitektuur, mille eri kihtides kasutatakse eri arhitektuuristiile -- hajutatud protsessijuhtimine keemiatööstuses. Eri kihtidesse kuuluvad: protsessi mõõtmine ja juhtimine (*control*) on objekt-orienteeritud stiilis, protsessi jälgimine on objekt-orienteeritud protsessijuhtimise stiilis, protsessi majandamine (*management*) on objekt-orienteeritud

stiilis ning tehase ja ettevõtte majandamine on andmekeskses stiilis (klassikalised andmetöötlussüsteemid).
2. Interpretaator, mille komponentides kasutatkse eri arhitektuuristiile -- klassikaline reeglipõhine süsteem, milles täidetavaks pseudokoodiks on reeglite baasi sisu, interpretaator on reeglite interpretaator või tuletusmasin (*inference engine*), interpretaatori juhtimisolek on reeglite ja andmeelementide selektor ning programmi olek on töömälu.
3. Tahvliarhitektuuri realiseerimine interpretaatorina -- kõnetuvastussüsteem HEARSAY-II, kus pseudokoodiks on teadmisteallikate hulk, interpretaatori juhtimisolek on tähelepanu andmebaas (*focus of control database*) ja tahvli jälgija ning programmi olek on tahvli sisu.

4. Andmekesksed süsteemid

Vaadeldakse kuidas kolme andmekeskse arhitektuuriga süsteemi arhitektuurid on tehnoloogia ja kasutaja nõudmiste arenedes arenenud. Nende kõigi arengu juures ilmneb üldine mall -- *andmekeskse süsteemi arengumall*.

Ärilise andmetöötlustarkvara arenguetapid:

- pakett-jada süsteemid (andmevoo süsteemi erijuhtum; probleemseks osutub interaktiivse töö toetus) -- iseseisvad programmid, järjestikfailid -- pakett-jada stiil (andmevoo stiili erijuhtum)
- lihtne hoidla (järjestikfailide asemel asusid tehingutöötlusele orienteeritud andmebaasid; informatsioon on hajutatud mitmete andmebaaside vahel) -- interaktiivne töötlus, paralleelsed tehingud, iseseisvad andmebaasid
- virtuaalne hoidla (organisatsiooni andmebaaside ühendamine üheks ühtse skeemiga andmebaasiks; ühendatud andmebaasiskeemi halvaks omaduseks on tema staatilisus)
- hierahilised kihid (kasutaja ja andmebaaside vahele lisati vahekihid -- vahendajad, mis võimaldasid andmebaasidel sõltumatult areneda) -- multi-andmebaasid, aktiivsed vahendajad, klient-server organisatsioon

Tarkvara arenduskeskkondade arenguetapid:

- pakett-jada süsteemid (andmevoo süsteemi erijuhtum; iseseisvad tööriistad ja järjestikfailid)
- lihtne hoidla (üksikutel tööriistadel või tööriistade gruppidel oma pandam -- palju sõltumatuid/sobimatuid pandameid; puudub tihe side erinevate tööriistade vahel; korduvkasutamine võimatu)
- tsentraalne pandam (tihe side erinevate tööriistade vahel; korduvkasutamine ja avatus; võimalik keerukus)
- hierahilised kihid (näit. NIST/ECMA viitemudel)

Tarkvara arenduskeskkondade erinevuseks ärilisest andmetöötlusest on suurem hulk erinevaid andmetüüpe, väiksem andmete maht ja päringute hulk.

Andmekesksete süsteemide arengut juhtivad jõud tarkvara arenduskeskkondades:

üleminek pakett-töötluselt interaktiivsele tööle
effektiivsuse nõuede täitmiseks vajadus tarkvara inkrementaalselt arendada
vajadus juhtida tarkvara tegemise protsessi

Integreerimisastme suurenemisega kujunevad juhtimine ja planeerimine üha suurematekс probleemideks.

Andmekesksete süsteemide arengut juhtivad jõud tarkvara arenduskeskkondades:

- üleminek pakett-töötluselt interaktiivsele tööle
- effektiivsuse nõuede täitmiseks vajadus tarkvara inkrementaalselt arendada
- vajadus juhtida tarkvara tegemise protsessi

Integreerimisastme suurenemisega kujunevad juhtimine ja planeerimine üha suurematekс probleemideks.

Andmekesksete süsteemide arengus korduvad pidevalt andmevoo süsteemide variandid:

- pakett-jada arhitektuur (see on väga "suure-teraline"; ei võimalda tagasisidet, ei sobi paralleeltööks; ei sobi töötama interaktiivses reziimis)
- toru-filter arhitektuur (see on "peene-teraline"; võimeline andma väljundit enne kogu sisendi lugemist; võimaldab tagasisidet; sobiv inetraktiivseks tööks)

Teiseks korduvaks malliks olid hoidlate variandid, kus isesesivad protsessid suhtlevad vaid läbi jagatud andmete.

Kokkuvõttena esinevad andmekesksete süsteemide arengus järgmised etapid:

- eraldiseisvad mitte-interaktiivsed rakendused
- pakett-jadatöötlus
- integreerimine jagatud andmete kaudu -- pandamid
- dünaamiline integreerimine -- kihilised hierarhiad
- aktiivne juhtimine -- intelligentsed agendid

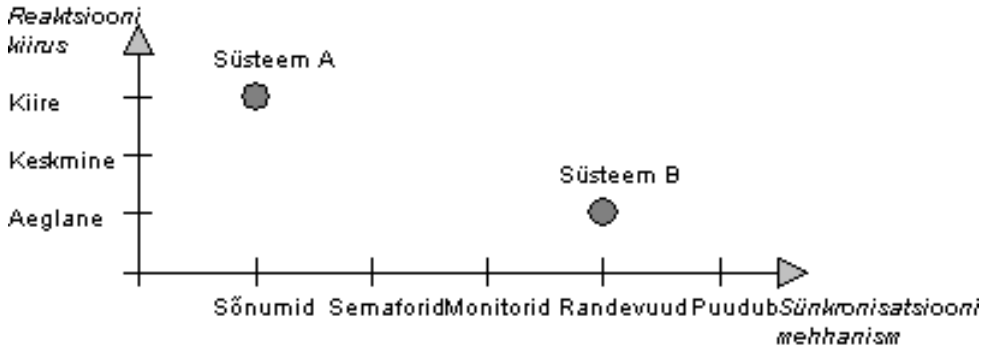
5. Juhendid arhitektuuri disainiks

Juhendid kasutajaliidese arhitektuuri disainiks

Alternatiivseid arhitektuure võib käsitleda disainiruumina.Disainiruumis võime formuleerida reeglid, mis lubavad tehtud valikuid hinnata ja juhivad disaini. Lisaks on disainiruum kasulik süsteemide kirjeldamisel ja mõistmisel kasutatava ühise keelena. Edasi vaadeldakse disainiruumi ja selle kasutamist kasutajaliidese disaini näitel.
Esiteks vaadeldakse teadmiste kodifitseerimist (teadmiste organiseerimist ja esitamist kasutataval kujul). Üheks programmeerimisalaste teadmiste kodifitseerimise näiteks on programmi juhtstruktuuri esitamine vähese hulga standartsete mõistete (hargnemised, tsüklid, alamprogrammi väljakutsed) abil. Kuna on tekkinud vajadus luua üha suuremaid tarkvarasüsteeme on järgnevaks loogiline kodifitseerida teadmised suuremate struktuursete konstruktsioonide osas. Tarkvara-alaste kodifitseeritud teadmiste koondamine käsiraamatutesse vähendaks tendentsi leiutada/luua iga uus süsteem tühjale kohale ja tendentsi kasutada ühte ja sama disaini, hoolimata selle sobivusest, suvalise probleemi korral.

Disainiruumi kujutatakse mitmemõõtmelise ruumina, mille iga mõõde kirjeldab mingit ühte

süsteemi omadust või disainivalikut. Väärtused ühes mõõtmes vastavad alternatiivsetele nõuetele süsteemi omaduse suhtes või disainivalikutele.
Näiteks:



Erinevad mõõtmed pole alati sõltumatud ja disainireeglite loomiseks osutub tähtsaks leida mõõtmete vaheline sõltuvus.
Eriti kasulikuks osutuvad sõltuvused nõudeid ja disainiotsuseid esitavate mõõdete vahel kuna nende abil saab disainiotsuste vahel valida. Üheks võimaluseks on uurida edukaid süsteeme ja vaadata, kus need disainiruumis paiknevad.
Mõõtmed, mis kirjeldavad nõudeid süsteemi funktsionaalsusele ja effektiivsusele moodustavad funktsionaalse disainiruumi (tarkvara elutsüklis kujutab see endast nõuete analüüsi tulemust). Struktuurseid disainivalikuid esitavad mõõtmed moodustavad aga struktuurse disainiruumi (tarkvara elutsüklis kujutab see endast algset süsteemi dekompositsiooni). Mõõtmed ei pruugi olla pidevad ega väärtused nendel järjestatavad.

Et kirjeldada kasutajaliidese arhitektuuri disainiruumi puhul disainiotsuseid kasutajaliidese struktuuri juures, jagatakse kasutajaliideste komponendid kolme kategooriasse:

- rakendusekohane komponent (eriomane konkreetsele rakendusele)
- jagatud kasutajaliidese komponent (toetab paljude rakenduste kasutajaliideseid)
- seadmest sõltuv komponent (eriomane konkreetsele sisend/väljund seadmete klassile)

Kusjuures rakendusekohase ja kasutajaliidese komponendi vahel on rakendusliides ning kasutajaliidese ja seadmest sõltuva komponendi vahel on seadmeliides.

Kasutajaliidese arhitektuuri disainiruumi funktsionaalsed mõõtmed langevad kolme rühma:

- välised nõudmised (rakendusest, kasutajatest ja seadmetest tulenevad)
    - väliste sündmuste töötlemine: (väliseid sündmusi pole, sündmuste töötlemine ooteajal, sündmused katkestavad kasutajat)
    - sätitavus (*customizability*): (kõrge, keskmine, madal)
    - kohandatavus (*adaptability*): (puudub, lokaalne käitumine muudetav, globaalne käitumine muudetav, semantika muudetav)
    - arvutussüsteemi organisatsioon: (üheprotssesiline töötlus, mitmeprotsessiline töötlus, hajustöötlus)
- interaktiivne käitumine (põhiotsused interaktsiooni iseloomust)
    - menüüd (valikute grupid)
    - vormid (antud muutujate väärtuste sisetamine)
    - käsukeel (tehiskeel)
    - loomulik keel (loomuliku keele alamhulk)
    - otsene manipuleerimine (andmete otsene graafiline kujutis)
- praktilised nõudmised (arenduskulud, muudatuste lihtsus jms.)

        - porditavus: (kõrge, keskmine, madal)

Kasutajaliidese arhitektuuri disainiruumi struktuursed mõõtmed võib samuti jagada kolme rühma:

- dekompositisoon (funktsioonide ja teadmiste jagamine moodulite vahel, moodulite liidesed)
    - rakendusliides: (monoliitne programm, abstraktne seade (abstraktne draiver), tööriistakomplekt (kooditeek), fikseeritud andmetüüpidega interaktsioonihaldur, laiendatavate andmetüüpidega interaktsioonihaldur, laiendatav interaktsioonihaldur)
    - seadme liides: (ideaalne seade, paramteriseeritud seade, muutuvate operatsioonidega seade, juhuslik seade)
- esitusviisid (andmete ja meta-andmete esitusviisid)
    - kasutajaliidese esitusviisid
        - varjatud: (kasutajaliidese koodis, rakenduse koodis)
        - väline: (deklaratiivne esitus (näiteks grammatika), protseduraalne esitus (näiteks spetsiaalne programmeerimiskeel)
        - sisemine: (deklaratiivne esitus, protseduraalne esitus)
- juhtimisvoog, side ja sünkroniseerimine (dünaamiline käitumine)
    - side: (sündmused, puhas jagatud olek, jagatud olek või vihjed (sündmused vaid täiendavad), jagatud olek ja sündmused (sündmused on olulised))
    - juhtimine (paralleelsus): (puudub, tavalised protsessid (lahus aadressiruumid), kerged protsessid (lõngad jagatud aadressiruumil), kooperatiivsed multiprotsessid, sündmuste haldurid, katkestuste töötlus)

Katse käigus leiti 622 madala taseme disainireeglit, millest abstraheeriti välja 170.
Näited leitud reeglitest, mis seovad funktsionaalseid ja struktuurseid mõõtmeid:

- kui on vajalik kasutaja katkestamine, tuleks kasutada juhtimises kas tavalisi protsesse, lõngu või katkestuste töötlust
- kui on vajalik kõrge sätitavus, tuleks kasutada väliseid kasutajaliidese esitusviise
- mida suuremad nõudmised on kohandatavusele, seda kõrgem peab olema rakendusliidese abstraktsus ja rakenduse sõltumatus kasutajaliidesest
- kui arvutussüsteem on hajus, tuleks kasutada sideks sündmusi; olekupõhine side vajab jagatud mälu
- interaktiivne käitumine määrab rakendusliidese abstraktsiooni taseme
- porditavus kasutajaliidese stiilide vahel nõuab rakendusliidese kõrgemat abstraktsust

Näited reeglitest, mis seovad struktuurseid mõõtmeid omavahel:

- rakendusliidese abstraktsioonitase mõjutab kasutajaliidese esitusviisi
- monoliitse programmi ja abstraktse seadmeliidese puhul piisab kasutajaliidese kaudsest esitusest
- tööriistakomplekt sisaldab kaudset ja sisemist deklaratiivset kasutajaliidese esitust
- interaktsioonihaldurid kasutavad välisi ja sisemisi deklaratiivseid esitusi
- laiendatavad interaktsioonihaldurid kasutavad protseduraalset esitust

Kaalutud disainiruum (*quantified design space -- QDS*) (*Toru Asada, Roy F. Swonger, Nadine Bounds, Paul Duerig*) on välja töötatud Carnegie Mellon'i ülikoolis eesmärgiga analüüsida ja võrrelda konkreetses rakendusvaldkonnas tarkvara arhitektuurilisi lahendusi. See põhineb kvaliteedifunktsiooni juurutamise (*quality function deployment -- QFD*) ja disainiruumi mõistetel.

Kvaliteedifunktsiooni juurutamine on kvaliteedi tagamise tehnika, mis aitab teisendada kasutaja nõuded süsteemile tehnilisteks nõueteks (see tehnika on toetatud graafilise esituse ja hästimääratletud protsessi poolt, kus kasutaja nõetele ja nende seostele disainilahenditega omistatakse kaalud). Seda kasutatakse disainiruumi mõõdete määramisel. Kogu protsess on hästi väljendatav ja automatiseeritav tabelarvutusprogrammi abil.
Kaalutud disainiruumide puhul on kasulikud statistilise analüüsi meetodid, mis võimaldavad hinnata disainilahendite komplekti üldist "headust" mingi baasi suhtes, selgitada disainilahendite komplekti halva hinnagu põhjusi (selgitada välja kohad, kus andtud disaini on võimalik parandada), selgitada, millises osas antud süsteemis olevad lahendused on parimad ja võrrelda kahte disainilahendite komplekti.

6. Formaalsed mudelid ja spetsifikatsioonid

Arhitektuuri formaliseerimise väärtus

Formalismid on tavaliselt väljakujunenud inseneriteaduse omaduseks ja neid võib kasutada täpsete mudelite esitamiseks ning nende analüüsimiseks. Tarkvara arhitektuuri alal on mitmeid asju, mida saab formaliseerida:

1. *Konkreetse süsteemi arhitektuur.* Sellised formalismid lubaks tarkvaraarhitektil planeerida konkreetset süsteemi ja olla osaks süsteemi spetsifikatsioonist, lubades süsteemi omaduste analüüsi.
2. *Arhitektuuristiil.* Selliseid formalisme saaks kasutada süsteemiperede arhitektuuriliste abstraktsioonide kirjeldamiseks. Nende abil saaks täpselt esitada praegu mitteformaalsel kujul olevad üldised idioomid, mallid ja viitearhitektuurid ja näidata, kuidas erinevaid arhitektuure saab käsitleda üldiste abstraktsioonide laiendustena.
3. *Tarkvara arhitektuuri teooria.* Sellised formalismid selgitaksid üldiste arhitektuurimõistete mõtet ja annaks deduktiivse baasi süsteemide arhitektuuri analüüsiks, näiteks andes arhitektuurikirjelduse hästivormistatuse reeglid (*wellformedness rules*).
4. *Arhitektuuri kirjeldamise keelte formaalne semantika.* Sellised formalismid käsitleksid arhitektuuri kirjeldust keeleliselt seisukohalt, kasutades traditsioonilisi keele semantika esitamise tehnikaid.

Konkreetse süsteemi arhitektuuri formaliseerimine

Sageli on arhitektuuri disain esitatud mitteformaalselt, kuna arhitektuuri taseme abstraktsioonide otseseks esitamiseks pole olemas vahendeid.
Vaatleme torud-filtrid arhitektuuristiilile vastavat arhitektuuri (ostsilloskoop), mille komponentideks on sidestamine (*couple*), eraldamine (*acquire*), lainevormi jälg (*waveform trace*) ja piiramine (*clip*):

1. Formaliseerimist alustame andmete iseloomustamisest
2. Iga komponendi jaoks anname formaalse kirjelduse:
   Sidestaja eraldab DC taseme signaalist ja teda võib parametriseerida 3 valikuga DC, AC ja GND. Valides DC, jääb signaal muutmata, AC lahutab vajaliku DC taseme ja GND tekitab 0 signaali. Funktsiooni *Couple* tulemuseks on sisendparameetrist *Coupling* sõltuvalt, funktsioon *Signal ---> Signal*.
   Lainevorm saadakse signaalist ajavahemiku pikkusega *dur* eristamisega alates mingist sündmusest *trig* peale viidet.
3. Arhitektuuris esinevaid komponentide ühendusi interpreteerime kui komponentide vahelisi sisend/väljund suhteid:
   Komponentide sisendparameetrid pakitakse ühte andmestruktuuri.

Alamsüsteem kirjeldatakse kui individuaalsete komponentide funktsionaalne kompositsioon.
4. Moodustame komponentidest alamsüsteemid.

Tulemuseks on ehitatava süsteemi täpne kirjeldus ja teiseks on süsteemi arhitektuur väljendatud sisendite ja väljundite kaudu ühendatud komponentide funktsionaalse kompositsioonina.

Arhitektuuristiili formaliseerimine: filtrid, torud, andmevoosüsteemi näitel

Eelneval kirjeldusel on ka mõned vead ja kõige suurem nendest on see, et süsteemi arhitektuuri stiil pole ilmutatud kujul esitatud. Komponendid on ühendatud funktsionaalse kompositsiooni abil ja seega ei saa vaadelda süsteemi topoloogilisi omadusi süsteemi spetsifikatsioonist sõltumatult. Püüame neid probleeme vältida kirjeldades ilmutatud kujul torude ja filtrite arhitektuuristiili.

Filtrid on defineeritud oma nime, portide ja oma programmiga. Filtri pordid on defineeritud nimede hulgana ja et porte modelleerida ühesuunalistena on portide hulk jaotatud sisend- ja väljundportideks. Iga pordiga on seotud tüüp, mis kirjeldab andmeid mida filter antud pordi kaudu on valmis töötlema.

Filtri programm on defineeritud võimalike programmi olekutega, algolekuga ja kujutusega sisenditest väljunditesse koos võimaliku oleku muutusega. See kirjeldab filtrit kui olekumasinat.

Torud on lihtsalt tüpiseeritud ühendused kahe pordi vahel.

Torude ja filtrite süsteemi kirjeldatakse kui torude ja filtrite hulka. Ja selle süsteemi terviklikuse tagavad nõudmised, et kõikidel filtritel on unikaalne nimi, et pole olemas mitte-ühendatud torusid ja toru on defineeritud portidega millede külge ta on ühendatud ning ühegi pordi külge pole ühendatud üle ühe toru. Mudel ei nõua aga, et filtri kõik pordid oleksid ühendatud mingi toruga.

Samuti nagu torude ja filtrite süsteem on defineeritud oma komponentidega on torude ja filtrite süsteemi olek defineeritud tema komponentide olekutega.

Selline formalism lubab meil vaadelda antud arhitektuuristiili kitsendusi. Näiteks konveierarhitektuur on kirjeldatud seades algsele süsteemile lihtsa kitsenduse.

Arhitektuuri disainiruumi formaliseerimine

Üks tarkvara arhitektuuride juures esinev probleem on see, et erinevad disainerid interpreteerivad arhitektuure erinevalt. See probleem väljendub selles, et nime poolest sama arhitektuuristiili kasutavad süsteemid võivad olla väga erinevad ja samas ei pruugi kahe samase arhitektuuriga süsteemi puhul disainerid seda samasust märgata. Arhitektuuri formaalne kirjeldamine muudab arhitektuuride vahelised suhted täpseiks.

Arhitektuuri teooria ja selle areng

Tähtsaks eesmärgiks tarkvara arhitektuuri alasele uurimistööle on selgitada tarkvara arhitektuuri olemust. Vastata küsimustele, mis on komponent ja mis on ühendus. Milline on korrektne (*well-formed*) arhitektuur. Millised on reeglid arhitektuuriliseks dekompositsiooniks.

7. Keele küsimused

Nõuded arhitektuuri kirjeldamise keelele

Arhitektuuri kirjelduse keeleline iseloom:

- Üldiselt kasutatavate arhitektuuride analüüs paljastab üldised mallid või idiomaatilised konstruktsioonid
- Need konstruktsioonid toetuvad üldiste elementide liikidele ja üldistele ühendusstrateegiatele
- Keeled sobivad hästi priitiivsete elementide vaheliste keeruliste seoste ja nende kombinatsioonide kirjeldamiseks
- Kui on võimalik leida vajalikud semantilised konstruktsioonid, on mõistlik defineerida keel.

Tarkvara organisatsiooni üldised mallid:

- Üldised komponendid ja seosed
    - (puhtale) arvutuslik -- lihtsad sisend/väljund seosed, puudub püsiv olek
    - mälu -- jagatud hulk püsivaid (*persistent*) struktureeritud andmeid
    - haldur (*manager*) -- olek ja sellega tihedalt seotud operatsioonid
    - juht (*controller*) -- juhib teiste sündmuste ajalist järgnevust
    - seos -- edastab informatsiooni elementide vahel
- Komponentide vahelised vastasmõjud
    - protseduuride väljakutsed -- üksikut juhtlõnga siirdatakse definitsioonide vahel
    - andmevoog -- iseseisvad protsessid suhtlevad andmevoogude kaudu; andmete olemasolek toob kaasa juhtimise
    - kaudväljakutse -- arvutused kutsutakse välja sündmuse toimumisel; protsesside vahel pole ilmutatud vastasmõjusid
    - sõnumiedastus -- iseseisvad protsessid suhtlevad andmete ilmutatud ja diskreetsel kujul edastamisega; võib olla sünkroonne või asünkroonne
    - jagatud andmed -- komponendid töötavad paralleelselt samadel andmetel
    - isendistamine (*instantiation*) -- isendistaja kasutab isendistatud definitsiooni võimeid (*capabilities*)

Keele kriitilised elemendid:

- komponendid -- primitiivsed semantilised elemendid ja nende väärtused
- operaatorid -- funktsioonid, mis kombineerivad komponente
- abstraktsioonid -- reeglid komponentidest ja operaatoritest koostatud avaldiste nimetamiseks
- katted (*closure*) -- reeglid määramiseks, milliseid abstraktsioone võib lisada primitiivsete komponentide ja operaatorite klassidele
- spetsifikatsioonid -- semantika seostamine süntaktiliste vormidega

Nõuded arhitektuuri kirjeldamise keeltele:

- *kompositsioon* -- peab olema võimalik kirjeldada süsteemi kui iseseisvate komponentide ja ühenduste kompositsiooni
    - arhitektuuri kirjeldamise keel peab võimaldama jagada keerukat süsteemi hierarhiliselt väiksemateks osadeks
    - elemendid peavad olema piisavalt iseseisvad, et neid saaks mõista isoleerituna süsteemist, kus neid kasutatakse

    - peab olema võimalik realisatsiooni puudutavaid küsimusi (nagu algoritmide ja andmestruktuuride valik) eraldada arhitektuuri puudutavatest küsimustest
- *abstraktsioon* -- peab olema võimalik kirjeldada komponente ja nende vahelisi ühendusi selliselt, et sellest selgelt ja ilmutatud kujul järeldub nende abstraktne roll süsteemis
    - abstraktsioonidena peab olema võimalik esitada uusi arhitektuurilisi malle ja uusi arhitektuurielementide vastamõju viise
- *korduvkasutus* -- peab olema võimalik komponente, ühendusi ja malle korduvalt kasutada erinevates arhitektuuri kirjeldustes, isegi kui nad on välja töötatud teises kontekstis (süsteemid jagavad sageli mitmeid arhitektuurilisi omadusi)
    - kui on võimalik kirjeldada üldisi komponentide ja ühenduste malle, on võimalik kirjeldada süsteemide arhitektuuride peresid, kui arhitektuurielementide lahtisi hulki koos nende struktuurile ja semantikale seatud piirangutega
    - arhitektuuriline korduvkasutamine erineb lõpliku hulga parametriseeritavate ja lõplikus süsteemis eristatavate komponentide teekide kasutamisest (keeled tavaliselt ei võimalda kirjeldada moodulite parametriseeritud hulki või struktuuri malle)
- *konfiguratsioon* -- arhitektuuriline kirjeldus peab eristama süsteemi struktuuri kirjelduse struktuuri elementide kirjeldustest ja toetama dünaamilist ümberkonfigureerimist (viimast on vaja süsteemi arhitektuuri arendamiseks süsteemi töö ajal)
- *ebaühtlus (heterogeneity)* -- peab olema võimalik kombineerida mitmeid erinevaid arhitektuuride kirjeldusi
    - esiteks peab olema võimalik kombineerida erinevaid arhitektuuri malle üheks süsteemiks ja kasutada arhitektuuri kirjelduse erinevatel tasemetel erinevaid arhitektuurilisi idioome
    - teiseks peab olema võimalik kombineerida erinevates keeltes kirjeldet komponente; kuna arhitektuuri kirjeldus on kõrgemal abstraktsiooni tasemel, kui algoritmide või andmestruktuuride kirjeldused, pole loogilist põhjendust sellele, et need peaksid kasutama sama notatsiooni
- *analüüs* -- peab olema võimalik sooritada arhitektuuri kirjelduste mitmesuguseid analüüse (arhitektuuri kirjelduse keeled peavad toetama automatiseeritud ja mitte-automatiseeritud arutlusi arhitektuuri kirjelduste üle)
    - sageli kasutab disainer mingit arhitektuurielementide hulka, kuna see võimaldab analüüsida antud süsteemi mingeid konkreetseid omadusi (näiteks torude ja filtrite kasutamisel on lihte analüüsida süsteemi läbilaskevõimet; samas ei võimalda olemasolevad moodulite ühendamise keeled muud, kui tüübikontrolli moodulite piiril, võimaldamata esitada või analüüsida semantilisi omadusi)
    - paljud arhitektuurilised omadused on dünaamilised, sellepärast on arenenud analüüsivormid arhitektuuriliste formalismide jaoks eriti tähtsad (näiteks kui ühendus on seotud mingi protokolliga, peab olema võimalik uurida, kas sellise ühenduse kasutamine antud kontekstis on korrektne ja samuti uurida näiteks ajastamist, effektiivsust ning ressursside kasutamist, mis näitavad, kas antud arhitektuuriline lahendus on sobiv)

Suurem osa olemasolevaid notatsiooneon liigitatavad viide laia kategooriasse:

1. Mitteformaalsed diagrammid
2. Programmeerimiskeelte moduleerimisvahendid
3. Moodulite ühendamise keeled
4. Alternatiivseid vastasmõju liike toetavad (*support for alternative kinds of interaction*) keeled
5. Spetsiaalsed notatsioonid teatud arhitektuuristiilide jaoks

Probleemid olemasolevate keeltega

- Mitteformaalsed diagrammid -- võivad olla kõrge abstraktsiooni tasemega aga nende mitteformaalsus takistab neid kasutada analüüsis ja nende side realisatsiooniga on väga nõrk
- Programmeerimiskeelte moduleerimisvahendid -- keeled nagu Simula, CLU, Alphard ja Ada põhinevad sellel, et moodul defineerib liidese, mis kirjeldab, milliseid teenuseid moodul süsteemile annab (mida ta ekspordib) ja milliseid teenuseid ta süsteemilt vajab (mida ta impordib); selles kontekstis on teenused antud programmeerimiskeele madala taseme konstruktsioonid, nagu muutujad, funktsioonid, tüübid jms.
    - kompositsioon -- suurem osa moduleerimisvahendeid võimaldab hierarhilist dekomponeerimist aga programmeerimiskeele moodulid ei anna head tuge arhitektuuriliste elementide kompositsioonile (nimede kasutamine elementide seostamisel, süsteemi struktuuri peitumine moodulite definitsioonidesse ja algoritmilise ning arhitektuurilise kirjelduse segamine)
    - abstraktsioon -- tavaliselt on programmeerimiskeeltes moodulite ühenduste valik kesine piirdudes protseduuride väljakutsete ja jagatud andmetega ja selle tulemuseks on suur osa süsteemi disainist ilmutamata kujul ning raskesti muudetav; programmeerimiskeeles ongi tavaliselt soovitav omada ühte lihtsat arvutusmudelit suure valiku mehhanismide asemel aga disaini arhitektuuriline tase nõuab komponentide ühenduste esitamiseks oluliselt laiemat abstraktsioonide hulka
    - korduvkasutus -- korduvkasutus on vaid moodulite tasemel, mitte kompositisoonimallide tasemel; moodulite kirjelduste juures pole kirjeldatud nende semantikat
    - ebaühtlus -- tavaliselt mooduleid, mis on kirjutatud erinevates programmeerimiskeeltes ei saa kombineerida
    - analüüs -- programmeerimiskeele tekstid ei sobi arhitektuuriliseks analüüsiks
- Moodulite ühendamise keeled -- programmeerimiskeele moduleerimisvahendite asemel võib süsteemi osade ühenduste kirjeldamiseks kasutada spetsiaalseid keeli (nagu MIL75 ja Intercol); moodulite kirjeldamise keeled eraldavad süsteemi konfiguratsiooni süsteemi osadest, mida komponeeritakse; tegelikuses ei kasuta nad rohkemaid abstraktsioone, kui programmeerimiskeelte moduleerimsvahendid ja ei võimalda kompositsioonimallide korduvkasutamist
- Alternatiivseid vastasmõju liike toetavad keeled -- on olemas keeli, milles on tehtud mõnede protseduuride väljakutsetele ja andmete jagatud kasutamisele alternatiivsete vastasmõjude kirjeldamine lihtsamaks (nagu Unix'i kest, mis toetab otseselt torusid ja filtreid ja Ada, mis toetab randevuusid (*rendezvous*)); kahjuks ei luba sellised keeled uute vastasmõju abstraktsioonide kirjeldamist
- Spetsiaalsed notatsioonid teatud arhitektuuristiilide jaoks -- toetavad vaid kindlat arhitektuuristiili või paradigmat ja ei võimalda kirjeldada ebaühtlasi arhitektuure

On ilmunud rida uusi arhitektuuri kirjeldamise keeli, mis püüavad eelpoolmainit nõudeid täita:

- Rapide -- arhitektuuri kirjeldamise keel, mis põhineb sündmuste mallidele
- ArTek -- arhitektuuri kirjeldamise keel, mis keskendub arhitektuuri disaini struktuurile
-

Ühendused

Arhitektuurilised kirjeldused käsitlevad tarkvarasüsteeme, kui komponentide kompositsiooni.

Keskendudes komponentidele, jäetakse ühenduste kirjeldused ilmutamata kujule ja hajutatuna. Järgnevalt pakutakse välja süsteemide komponeerimise mudel, milles ühendused on komponentidega võrdsed.

Praeguste lahenduste probleemid:

- Vastasmõjude kirjeldusel pole kohta.
  Suurem osa praeguseid moodulite ühendamise tehnikaid sõltuvad moodulisse kodeeritud *import/export* käskudest, mis tekitab kolm probleemi:
  - Nimede kokkulangemise nõue
  - Süsteemi struktuuri kirjelduse hajutamine moodulitesse
  - Vastasmõju asümeetrilisus
- Kehvad abstraktsioonid.
  See, et ühenduste abstarktsioonid pole otseselt kirjeldatavad, tekitab omakorda kolm probleemi:
  - Võimatus siduda mooduli liideses teatud elemente ühte gruppi (võib vaid komentaaridega märgistada)
  - Võimatus kirjeldada seotud elementide vahelisi suhteid (nagu järjestus)
  - Võimatus spetsifitseerida elemntide hulga (liit)omadusi
- Struktuuri puudumine liideste kirjeldustes.
  Puudub kaks abstraktsioonitaset:
  - Ühenduste abstraktsioonid (primitiivsete *import/export* käskude liitmine ja soovitud abstraktse funktsiooni kirjeldamine)
  - Liideste segmenteerimine (liideste dekompositsioon)
- Programmeerimiskeelte erinevad eesmärgid.
  Programmeerimiskeelte eesmärgiks on algoritmiliste operatsioonide kirjeldamine ja nad on selleks väga sobivad aga nad ei sobi eriti hästi mittefunktsionaalsete omaduste nagu töökindlus, effektiivsus, turvalisus jms. kirjeldamiseks. Arhitektuurne disain aga keskendub mitte algoritmidele ja andmestruktuuridele vaid süsteemi topoloogiale, komponentide omadustele, komponentide vahelistele vastasmõjudele ja süsteemi mittefunktsionaalsetele omadustele. Sellepärast pole mõtet kasutada ega püüda laiendada tavalisi programmeerimiskeeli arhitektuurse disaini jaoks.
- Komponentide halb pakendamine.
  Komponentide pakendamine eeldab sageli liiga palju komponentide keskkonnast ja kasutatavast arhitektuuristiilist.
- Halb tugi mitmekeelsusele ja mitmeparadigmalisusele.
  Tööriistad, mis on loodud toetama teatud arhitektuurilist paradigmat ei toeta sageli selliste süsteemide tegemist, mis segavad arhitektuuri idioome.
- Halb tugi pärandsüsteemidele.
  Puuduvad tööriistad, mis võimaldaksid analüüsida, milline oli süsteemi disaneri eesmärk.

Uus vaade tarkvarasüsteemide komponeerimisele

Süsteeme koostatakse erinevat tüüpi komponentidest. Komponendid võivad olla erinevates vastasmõjudes. Komponendid vastavad sageli kompileerimisühikuile.
Ühendused vahendavad komponentide vahelisi vastasmõjusid, s.t. nad seavad reeglid komponentide vastasmõjudele ja määravad vajalikud mehhanismid. Ühendused ei vasta kompileerimisühikuile vaid vastavad linkuri käskudele, protseduuride väljakutsete jadadele, süsteemi teenuste kasutamisele jms. Ühendusi võib vaadelda defineerivana hulka rolle, mida konkreetsed komponentide osad võivad täita.
Seega koosneb tarkvarasüsteem komponentidest ja ühendusist.

Komponendid on arvutuste ja oleku paigad (*loci*). Igal komponendil on liides, mis defineerib tema omadused (s.h. tema ressursside signatuurid ja funktsionaalsus, globaalsed seosed, effektiivsuse näitajad jms.). Iga komponent on mingit tüüpi. Komponendid võivad olla kas primitiivsed või liitkomponendid.

Ühendused on komponentide vaheliste seoste paigad. Ühendused vahendavad vastasmõjusid, igal ühendusel on protokolli kirjeldus, mis defineerib tema omadused (s.h. liideste tüübid, mida ta on võimeline vahendama, vastasmõju omadused, reeglid sündmuste järjekorra kohta jms.). Iga ühendus on mingit tüüpi. Ühenduse protokollis nimetatud komponendid on rollid, mis tuleb täita. Ühendused võivad olla kas primitiivsed või liitühendused. Primitiivsed ühendused võivad olla realiseeritud kui: programmeerimiskeele mehhanismid, operatsioonisüsteemi funktsioonid, jagatud andmed, kommunikatsiooniteenused, andmevahetusformaadid, parameetrid, jne.

Ühendusi on parem käsitleda komponentidest eraldi kuna:

- Ühendused võivad olla üpris keerukad. Sageli ei saa näidata ühte kindlat komponenti, milles sisaldub ühenduse kirjeldus.
- Ühenduste kirjeldused tuleb koondada. Samuti, nagu on nõutav komponendi kirjelduse koondamine ühte kohta, on nõutav ka ühenduse kirjelduse koondamine ühte kohta. See toetab nii disaini kui ka arendust ja hooldust.
- Ühendused on potentsiaalselt abstraktsed. Nad on paramteriseeritavad ja kasutajad võivad soovida oma ühenduste kirjeldamist. Süsteemis võib olla mitmeid ühendusi, mis kõik kuuluvad samasse ühenduste klassi.
- Ühendused nõuavad hajusat süsteemi toetust. Ühenduste poolt nõutavad mehhanismid pole alati koondatud ühte kohta.
- Komponendid peavad olema sõltumatud. Komponendi liidese kirjeldus peab andma komponenid võimete (*capabilities*) täieliku määrangu aga mitte seadma kitsendusi sellele, kuidas komonenti tegelikult kasutatakse.
- Ühendused peavad olema sõltumatud. Üksik kõrge-taseme ühendus võib vahendada dünaamiliselt muutuvate komponentide hulga vastasmõjusid.
- Komponentide vahelised suhted pole fikseeritud. Komponente võib erinevate ühenduste kaudu erinevalt kasutada ja süsteemi konfiguratsioon võib dünaamiliselt muutuda.
- Süsteemid kasutavalt sageli korduvalt samu kompositsioonimalle. Mõned nendest mallidest nagu torud ja filtrid, klient-server, kihilised süsteemid ja tahvlisüsteemid on üldiselt vähemalt intuitiivselt tuntud. Sellised idioomid saab defineerida kui üldised mallid, mis piiravad komponentide ja ühenduste tüüpe ning vastasmõju topoloogiat.

Ühendustega arhitektuuri kirjeldamise keel

Arhitektuuri kirjeldamise keelel peab:

- defineerima ühenduste ja nende kompositsioonide semantika
- üldistama import/export reeglitest asümeetria, mitmesuse (*multiplicity*), lokaalsuse, abstraktsiooni ja nimetamise
- looma süsteemi organisatsioonide, komponentide, ühenduste ja nende elementide liitekohtade (*association*) tüübistruktuurid ja tüübihierarhiad
- sisaldama reegleid arhitektuurilisteks abstraktsioonideks

Kui keele elemndil on olemas:

- Spetsifikatsioon
- Tüüp
- Liitekoht
- Realisatsioon

siis elementideks on:

- Element: Komponent
  - Spetsifikatsioon: Liides
  - Tüüp: Komponendi tüüp
  - Liitekoht: Võimalik roll (*player*)
  - Realisatsioon: Realisatsioon
- Element: Ühendus
  - Spetsifikatsioon: Protokoll
  - Tüüp: Ühenduse tüüp
  - Liitekoht: Roll (*role*)
  - Realisatsioon: Realisatsioon

Lihtelemente vaadeldakse realiseerituna programmeerimiskeeles. Liitelementide realisatsioon aga koosneb osade loetelust, komponeerimisjuhistest ja vastavatest spetsifikatsioonidest.

Ühendused ja nende semantika on spetsifitseeritud protokolliga. Keel peab võimaldama kirjeldada:

- Pakettide kohalejõudmise garanteeritust sidesüsteemis
- Kitsendusi sündmuste järjestusele kasutades jälgi (*traces*) või teekirjeldusi (*path expressions*)
- Inkrementaalse tootmise/tarbimise reegleid konveierites
- Klientide ja serverite rollide eristamist
- Parameetrite vastavuse ja sidumise reegleid tavalistes programmeerimiskeelte protseduuride väljakutsetes
- Kitsendusi parameetrite tüüpidele, mida võib protseduuride kaugkutsetes kasutada

Võimalik on kas kasutada formalismi, mis toetab protokollide kirjeldamist või kasutada omaduste liste. Primitiivsed ühendused on need, mis on otse toetatud programmeerimiskeele või operatsioonisüsteemi poolt. Ühendused on sageli realiseeritud protseduuride hulga poolt ja selle protseduuride hulgaga on seotud reeglite hulk, mis määrab, kuidas neid protseduure võib kasutada. Selliseid kitsendusi võib määrata kas teekirjeldustena või täitmise järjekorra jälgedena. Kuna abstraktsete andmetüüpide operatsioonide hulgal on sageli samuti kitsendused (näiteks tuleb initsialiseerimine kutsuda välja enne teisi operatsioone jne.), võib neid vaadelda protokolli omavatena.

Järgmised protokollid võiksid eksisteerida arvutuskeskkonnas, kui sõltumatud definitsioonid:

- Konveieri protokoll
- Side (*communication*) protokoll
- Kasutaja interaktsiooni protokoll
- Andmebaasiprotokoll
- Abstraktsete andmetüüpide protokollid

Tüübistruktuurid arhitektuuri kirjeldamise keeles

Tüübikontrollile sarnane probleem kerkib üles komponentide ja ühenduste tüüpide juures, ja nagu iga tüübisüsteemi korral väljendavad need disaineri kavatsusi nende elementide kasutamise kohta. Kolmandaks paigaks, kus tüübi kontrolli võib rakendada on komponentide liidese elementide ühendamine ühenduse rollide külge.
Komponentide ja ühenduste korduvkasutamise vajadus olukordades, mis pole täpselt ette määratud nõuab, et oleks võimalik teha ühendusi liitekohtadega, mis täpselt teineteisele ei vasta (näiteks võib olla vajadus siduda filter Unix'is toru asemel failiga). Sellisel juhul on olemas mitmeid võimalusi:

- Ühendada igal juhul ilma mingi lisapingutuseta (mingid alamtüübi seosed);
- Informatsiooni ümbervormindada (*reformat*) ja ümberpaigutada (andmete ümbervormindajad ja parameetrite vastavusse seadjad);
- Pakkida (*wrap*) komponent teisendajasse (näiteks protseduur, mis pakib sisse filtri, söödab sisandparameetrid filtri sisendtorusse ja võtab filtri väljundtorust resultaadi);
- Teisendada andmed läbi ühise jagatud vormingu (andmevahetusvorming);
- Teisendada andmed ühest komponendist vormingusse, mida teine komponent ootab (paarikaupa sobivus; ühine sõnumivorming, kuid andmed nõuavad interpreteerimist);
- Lisada teisendusmoodul (puhver);
- Lihtsalt keelduda.

Kirjeldatud lahenduse omadused:

- Ühenduste kohta käiv informatsioon on kogutud ühte kohta. Mitteprimitiivsed komponendid võivad kasutada keerukaid seoseid ja kontsentreerida struktuurset informatsiooni
- On võimalikud vastasmõjude abstraktsioonid
- Liidese elementide rollide kasutamisega on osaliselt on lahendatud liidese struktuuri probleem
- Arhitektuuriline kirjeldus on eristatud programmeerimisest erinevate erineva semantikaga keelte kasutamisega
- Võimaldab tüübikontrolli süsteemi, mis võib korduvkasutamise lihtsustamiseks toime tulla teatavate ebakõladega
- Selgitab tingimusi, mille puhul saab programmeerimiskeeli segamini kasutada
- Teeb võimalikuks pärandsüsteemide kasutamise tehes süsteemi arhitektuuri kirjelduse ilmutatuks

Kaudväljakutse lisamine traditsioonilistele programmeerimiskeeltele

Selle asemel, et kutsuda mingi teise moodulist protseduure otse välja kuulutab moodul välja ühe või mitu sündmust. Teised süsteemi kuuluvad moodulid võivad registreerida oma huvi mingite sündmuste vastu sidudes selle sündmusega mingi protseduuri. Kui sündmus toimub, kutsub süsteem välja kõik antud sündmusega seotud protseduurid.

Selline kaudväljakutse suurendab korduvkasutamise võimalusi -- mingi hulga moodulite ühendamine süsteemiks on võimalik vaid nende huvi registreerimisega süsteemi sündmuste vastu. Samuti on süsteemi arendamine lihtsustatud, kuna on võimalik teisi mooduleid puudutamata asendada mingi moodul süsteemis.

Üks variant sellistest süsteemidest on tööriistade komplektid (näiteks

programmeerimiskeskkonnad) teine aga on andmekesksed süsteemid, mis on varustatud päästikutega/triklitega (*trigger*)

Sellise arhitektuuriga süsteemi korral on vaja teha 6 kategooriasse jagatud disainiotsuseid:

1. Sündmuse defineerimine
    1. Fikseeritud sündmuste sõnastik (sündmuste hulk süsteemis on lõplik ja pole laiendatav; näiteks Smalltalk80 ja aktiivsed andmebaasid)
    2. Sündmuste staatiline kirjeldamine (sündmuste hulk on laiendatav kompileerimisel; võimaldab kompileerimisel tüübikontrolli) *
        1. Sündmuste keskne kirjeldamine (sündmused kirjeldatakse kogu süsteemi jaoks ühes kohas) *
        2. Sündmuste hajutatud kirjeldamine (sündmused kirjeldatakse iga mooduli poolt ja iga moodul kirjeldab sündmused, mida ta ootab toimuvat) *
    3. Sündmuste dünaamiline kirjeldamine (sündmuste hulk on laiendatav töö ajal)
    4. Sündmusi ei kirjeldata (komponendid võivad kuulutada suvalisi sündmusi; näiteks tööriistade integreerimis keskkonnad Field ja Softbench, kus küll tavaliselt iga tööriist defineerib oma sündmuste sõnasiku)
2. Sündmuse struktuur
    1. Lihtsad nimed (sündmus koosneb lihtsalt nimes, parameetrid puuduvad)
    2. Fikseeritud parameetrite listid (kõikidel sündmustel on nimed ja sama parameetrite list)
    3. Parameetrid sündmuste tüüpide kaupa (igal sündmusel on fikseeritud parameetrite list aga erinevatel sündmustel võib ta olla erinev vastavalt sündmuse tüübile) *
    4. Parameetrid sündmuse kuulutamise alusel (iga kord kui komponent kuulutab välja sündmuse, võib ta kirjeldada parameetrite listi)
3. Sündmuse sidumine protseduuridega
    1. Staatiline sündmuste sidumine (sündmused seotakse protseduuridega kompileerimisel) *

        Sündmuste parameetrite teisendamine protseduuri parameetriteks
        1. Kõik parameetrid (protseduuri parameetriteks on sündmuse parameetrid)
        2. Valitud parameetrid (sündmuse sidumisel võib realiseerija näidata, millised sündmuse parameetrid edastatakse protseduurile ja millises järjekorras) *
        3. Parameetrite avaldised (protseduurile edastatakse sündmuse parameetrite avaldiste väärtused)
    2. Dünaamiline sündmuset sidumine (sündmusi seotakse protseduurideg dünaamiliselt; komponendid registreerivad oma huvi sündmuste vastu, kui nad soovivad neid saada ja deregistreerivad oma huvi sündmuste vastu, kui nad enam ei soovi neid saada)
4. Sündmuse kuulutamine
    1. Üks sündmuste kuulutamise protseduur (suvaline sündmus kuulutatakse välja ühe ja sama protseduuriga) *
    2. Mitu sündmuste kuulutamise protseduuri (iga sündmuse tüübi kohta oma sündmuse kuulutamise protseduur)
    3. Keele laiendamine (uue sündmusi kuulutava keele-elemendi lisamine)
    4. Kaudne kuulutamine (sündmusi kuulutatakse mingi protseduuri kõrvalefektina)
5. Paralleelsus
    1. Pakk (*package*) (komponent on pakk ja väljakutse on paki liideses oleva protseduuri kutse) -- viib mitteparalleelsele ühelõngalisele süsteemile *
    2. Pakitud tööd (*packaged tasks*) (komponent on töö (mille liides on pakki kirjeldus) ja väljakutse on töö liidese sisendpunkti (*entry*) kutse)
    3. Vabad tööd (*free tasks*) (komponent on töö; väljakutse on töö liidese sisendpunkti kutse aga töö on sündmustehalduri pakis)
6. Kättetoimetamise poliitika

1. Täielik kättetoimetamine (sündmuse kuulutamine põhjustab kõikide temaga seotud protseduuride väljakutsumise) *
2. Üksik kättetoimetamine (sündmuse kuulutamine põhjustab vaid ühe temaga seotud protseduuri väljakutsumise)
3. Parameetritel põhinev valik (sündmuse parameetreid kasutatakse, et otsustada milline seotud protseduur välja kutsuda -- mallimise (*pattern matching*) variant -- sama sündmuse korral võidakse välja kutsuda erinev hulk protseduure)
4. Olekupõhine poliitika (mõned süsteemid liidavad mingi poliitika, mis määrab sündmuse tegeliku mõju (sündmuse ignoreerimine, uute sündmuste kuulutamine, protseduuride kutsumine, jne.), iga seosega)

8. Tööriistad arhitektuuri disainiks

UniCon -- universaalne ühenduste keel

UniCon on arhitektuuri kirjeldamise keel, mis:

- toetab disainerite poolt kasutatavaid abstraktsiooniidioome
- spetsifitseerib komponentide pakkimis omadused ja funktsionaalsed omadused
- kirjeldab komponentide vahelisi vastasmõjusid (protokolle, andmevahetusesitusi jms.) ühes kohas -- ühendustes
- sisaldab abstraktsioonifunktsiooni (sarnane abstraktsetele andmetüüpidele), mis kujutab kõrgema taseme konstruktsioone koodiks
- võimaldab lisada väliseid konstrueerimis- ja analüüsivahendeid

Liitelemendi realisatsioon koosneb:

- osade listist (komponentidest ja ühendustest)
- kompositsioonikirjeldusest (assotsiatsioonid liidese ja ühenduse rollide vahel)
- abstraktsioonide kujutus (seos sisemiste liidese rollide ja liitelemendi liidese rollide vahel)
- teised spetsifikatsioonid

Funktsionaalsete spetsifikatsioonide analüüsis kasutatakse eel- ja järeltingimusi ning välist teoreemide tõestuse süsteemi. Süsteemi reaalaja omaduste analüüs põhineb monotoonse kiirusega analüüsi (*rate-monotonic analysis*) tehnikal ja seda tehakse samuti välise süsteemiga.

Stiili kasutamine arhitektuuri disaini keskkondades

Näidatakse, kuidas genereerida konkreetsele arhitektuuristiilile orienteeritud tarkvara arenduskeskkondi arhitektuuristiili kirjeldusest. Järgnevalt kirjeldatakse süsteemi *Aesop*, mis on ette nähtud stiilipõhiste arhitektuuri disainikeskkondade ehitamiseks.

Arhitektuuri stiili moodustavad:

1. Idioomid ja mallid (globaalsed organisatsiooni struktuurid nagu kihilised süsteemid ja lokaalsed mallid nagu mudel-vaade-kontroller)
2. Viitemudelid (*reference models*) (süsteemi organisatsioonid, mis näevad ette konkreetsed (sageli paramteriseeritavad) komponentide kofiguratsioonid ja vastasmõjud konkreetse rakendusvaldkonna jaoks; tüüpiliseks näiteks on kompilaatori jagamine leksika

analüsaatoriks, süntaksi analüsaatoriks, optimisaatoriks ja koodi generaatoriks)

Arhitektuuri stiilid määravad tavaliselt järgmised omadused:

1. Nad annavad disainielementide sõnastiku -- komponentide ja ühenduste tüübid
2. Nad defineerivad rea konfigureerimisreegleid -- või topoloogilisi kitsendusi, mis määravad antud elementide lubatud konfiguratsioonid
3. Nad defineerivad semantilise interpretatsiooni, millega disainielementide kompositsioonid, mis on sobivalt kitsendatud konfigureerimisreeglitega omavad hästi-defineeritud mõtet
4. Nad defineerivad analüüsid, mida saab vastavat stiili järgides ehitatud süsteemidele teha (analüüsi erijuhtumiks on koodi genereerimine)

Arhitektuuristiilide kasutamise head küljed:

- edendab disainide korduvkasutamist
- võimaldab koodi korduvkasutamist (arhitektuuristiili invariandid võimaldavad jagatud ühiskoodi)
- lihtsustab süsteemi organisatsioonist aru saamist
- soodustab süsteemide ühilduvust (*interoperability*)
- kitsendades disainiruumi, võimaldab konkreetseid stiilile omaseid analüüse
- tavaliselt on võimalik visualiseerida arhitektuuri stiilile omaselt

Aesop'ist genereeritud disainikeskkonnad (nim. *Fable*) koosnevad:

- stiili sõnastikule vastavast disainielementide tüüpide paletist
- kontrollidest, mis tagavad disainielementide kompositsioonide vastavust stiili topoloogilistele kitsendustele
- elementide semantilistest spetsifikatsioonidest
- liidesest, mis lubab välistel tööriistadel arhitektuuri kirjeldusi analüüsida ja neid manipuleerida
- mitmetest stiilile omastest arhitektuurilise informatsiooni visualisatsioonidest koos graafilise editoriga nendega manipuleerimiseks

Arhitektuuri disain põhineb seitsmest olemist koosneval üldisel ontoloogial:

- komponendid (*components*) -- esitavad arvutusi
- ühendused (*connectors*) -- esitavad komponentide vahelisi vastasmõjusid
- konfiguratsioonid (*configurations*) -- esitavad komponentide ja ühenduste topoloogiaid
- pordid (*ports*)
- rollid (*roles*)
- esitused (*representations*)
- seosed (*bindings*)

millest põhilised on komponendid, ühendused ja konfiguratsioonid. Nii komponentidel kui ka ühendustel on liidesed. Komponentide liidesed koosnevad portide hulkadest ja ühenduste liidesed koosnevad rollide hulkadest. Komponendi või ühenduse sisu kirjeldamiseks on olemas esitus. Iga esituse jaoks on olemas seosed sisemise konfiguratsiooni elementide ja välise elemendi liidese osade vahel (iga seos seostab sisemise pordi välimise pordiga või sisemise rolli välimise rolliga). Ühendused seovad alati pordi ja rolli aga seosed seovad pordi pordiga ja rolli rolliga. Esitus võib

olla ka väline, millel võib olla omakorda alamtüüpe nagu tekstiline esitus jms. Väliseid esitusi interpreteerivad teised tööriistad.

Mõned *Aesop*'iga seotud probleemid:

- Stiilide kirjeldamise juures oleks soovitav:
  - Stiilipiirangute ilmutatud esitus (praegu on need ilmutamata kujul peidetud stiili kuuluvate tüüpide realisatsioonides)
  - Supertüüpide nähtavuse kontrollimine (praegu rakenduvad kitsendused nende kasutamise suhtes alles komponeerimisel)
- Stiilipõhise disainikeskkonna käitumises oleks soovitav:
  - Stiilikirjelduste dünaamiline lisamine (praegu on vaja uue stiilikirjelduse lisamiseks süsteem ümber kompileerida)
  - Tüüpide migreerimine (praegu nagu, nagu igas tugevalt tüpiseeritud süsteemis määratakse objekti tüüp tema loomisel, oleks aga vaja objektii tüüpi töö ajal muuta, kui ta omadused vastavad mingi teise tüübi omadustele ja teda oleks tarvis kasutada selle teise tüübina (mehhanismiks võiks olla predikaattüüpide mehhanism))

Arhitektuuriline ühendus

Tavalised moodulite ühendamise keeled ei tee vahet realisatsiooni (*implements*) ja vastasmõju (*interacts*) seoste vahel. Nende vahel vahet tegemine on tähtis kolmel põhjusel:

- ühenduse tüübist sõltuvad arutlused, mida süsteemi kohta saab teha (realisatsiooni seoste puhul arutluskäik on hierarhiline -- mooduli korrektsus sõltub nende moodulite korrektsusest, mida ta kasutab aga vastasmõju seoste puhul sõltub see lisaks nende moodulite korrektsusele, millega vastasmõjus ollakse, ka veel vastamõju enda korrektsusest)
- erinevat tüüpi ühendused nõuavad erinevaid abstraktsioone (realisatsiooniseoste jaoks sobivad tavaliselt programmeerimiskeele vahendid)
- erinevat tüüpi ühenduste puhul on ka vastavuse kontroll erinev

Praegused moodulite ühendamise keeled on väga sobivad kirjeldamaks realisatsiooni seoseid.

Arhitektuuri kirjeldamine *Wright*'i mudelis põhineb ideel, et ühendused võib defineerida protokollidena, mis kirjeldavad oodatud suhtlusmalle moodulite vahel:

-

Wright'i näide:

```
System Capitalize
        Component Split
                port In [input protocol]
                port Left, Right [output protocol]
                comp spec [Split specification]
        Component Upper
                port In [input protocol]
                port Out [output protocol]
                comp spec [Upper specification]
```

```
        ...
        Connector Pipe
                role Writer
                role Reader
                glue spec [Pipe specification]
        Instances
                split: Split;
                upper: Upper;
                lower: Lower;
                merge: Merge;
                p1, p2, p3, p4: Pipe;
        Attachments
                split.Left as p1.Writer;
                upper.In as p1.Reader;
                split.Right as p2.Writer;
                lower.In as p2.Reader;
        ...
End Capitalize
```

Komponentide ja ühenduste spetsifikatsioonid kirjeldatakse CSP alamhulka kasutades, näiteks (kus komponentide iga port kirjeldatakse protokollina ja ühenduse rollid kirjeldavad vastasmõjus olevate osapoolte oodatud käitumist ning *glue* spetsifikatsioon koosneb kahest osast: lõplikute olekutega protokoll ja predikaat selle protokolli poolt genereeritud jälgedel):

Component Split =
        port In = read?x --> In [] read-eof --> close --> √
        port Left, Right = write!x --> Out | close --> √
        comp spec =
                let Close = In.close --> Left.close --> Right.close --> √
                in Close []
                        In.read?x --> Left.write!x -->
                        (Close[] In.read?x --> Right.write!x --> **computation**)

Connector Pipe =
        role Writer = write!x --> Writer | close --> √
        role Reader =
                let ExitOnly = close --> √
        in let DoRead = (read?x --> Reader [] read-eof --> ExitOnly)
        in DoRead ExitOnly
                glue =
                        let ReadOnly = Reader.read!y --> ReadOnly [] Reader.read-eof -->
Reader.close --> √
        [] Reader.close --> √
        in let WriteOnly = Write.write?x --> WriteOnly [] Writer.close --> √
        in Writer.write?x --> glue
        [] Reader.read!y --> glue
        [] Writer.close --> ReadOnly
        [] Reader.close --> WriteOnly
        spec ∀ Reader.read_i!y • ∃ Writer.write_j?x • i = j & x = y & Reader.read-eof ==>

(Writer.close & #Reader.read = #Writer.write)

Et tagada mudeli terviklikust ja korrektsust, peaks omavahel ühendatud portide ja rollide kirjeldused vastavuses olema. Täpne vastavusoleks liiga kitsendav, seega soovima vastavust, mille korral serveri porti võib kasutada rollis, mis nõuab vähem teenuseid, kui serveri port võib anda. Kokkuvõttes tahame tagada, et port arvestaks rolli käitumist. CSP terminites tähendab see, et kui pordi protsessis on determineeritud valik ([]), peaks olema determineeritud valik ka rolli protsessis. Tuleb välja, et see nõue on kaetud CSP täpsustuse mõistega, mida saame kasutada rollide ja portide vahelise vastavuse definitsioonis. Lisaks on võimalik näidata, et kui ühendus on tupikuvaba, on tupikuvaba ka iga antud ühenduse kasutus, kus ühenduse rollid on vastavuses komponendi portidega.

Et oleks võimalik protokollide automaatne kontroll, on *Wright*'is ühenduse kirjeldus jagatud kahte ossa, millest protokollide osas kasutatakse CSP kitsendatud (lõplikku) varianti. Selletõttu on võimalik *Wright*'i mudelite täielikust kontrollida olemasoleva vahendiga (kasutuses on süsteem FDR).

Len Bass, Paul Clements, Rick Kazman, **Software Architecture in Practice**, Addison Wesley, 1998

Arhitektuuri visioon

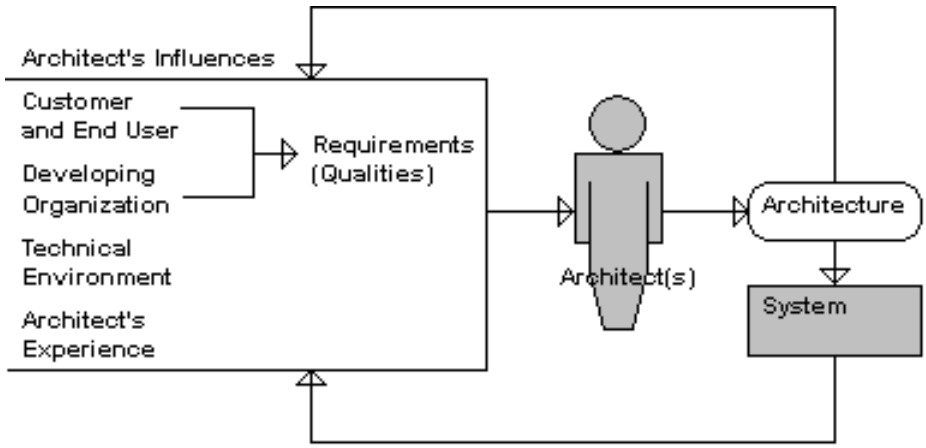
Arhitektuur on süsteemi keskne tegem (*artifact*), mille mõju säilib süsteemist kauem. Kui disainiprotsessi kujutletakse sageli tegevuste/sammudena, saavutamaks süsteemi oodatav funktsionaalsus, mõjutavad arhitektuuri väljatöötamist kaugemad eesmärgid (millest hämmastavalt vähesed puudutavad süsteemi korrektset toimimist).

Arhitektuur ja selle loonud organisatsioon on keerukas vastasmõjus, mida raamatus nimetatakse Arhitektuuri Äritsükliks (*Architecture Business Cycle -- ABC*).

Arhitektuuri äritsükkel

Arhitektuurid ei eksisteeri iseeneses, vaid on tsükli osad. Arhitektuur on vaid eesmärgi saavutamise vahend. Seda mõjutavad nii kliendi kui ka arendaja funktsionaalsed ja kvaliteedisihid. Samuti on see mõjutatud arhitekti taustast ja kogemustest ning tehnilisest keskkonnast. Arhitekuur omakorda mõjutab väljatöötatavat süsteemi ja on väljatöötava organisatsiooni põhiline vara ning mõjutaja. Väljatöötatav süsteem omakorda mõjutab väljatöötavat organisatsiooni, arhitektuuri ja võimalik et ka tehnilist keskkonda. See mõjutab süsteemile ja organisatsioonile seatavaid tulevasi sihte.

Arhitektuuri äritsükkel (ABC):



Arhitektuuripõhise protsessi sammud:

- Äripõhjenduse (*business case*) loomine.
- Nõudmiste mõistmine (kasutusjuhtude analüüs, valdkonna analüüs, prototüüpimine).
- Arhitektuuri loomine või valimine
- Arhitektuuri esitamine ja vahendamine huvitatud osapooltele (*stakeholders*)
- Arhitektuuri analüüs ja hindamine
- Süsteemi realiseerimine vastavalt arhitektuurile
- Realisatsiooni arhitektuurile vastavuse tagamine

Pole olemas arhitektuure, mis on head või halvad iseenesest, arhitektuuri saab vaid hinnata tema sobivuse järgi mingi konkreetse eesmärgi jaoks.

Soovitused protsessile:

- arhitektuur peaks olema ühe arhitekti või väikese, selge juhiga arhitektide grupi töö tulemus.
- arhitektil (või arhitektuurirühmal) peaks olema kasutada tehnilised nõudmised süsteemile ning prioritiseeritud omaduste nimekiri, mida arhitektuur peab rahuldama.
- arhitektuur peab olema hästidokumenteeritud, kasutades kokkulepitud esitust, mida huvitatud osapooled mõistavad vähese pingutusega.
- arhitektuur peab ringlema huvitatud osapoolte käes, kes peaks aktiivselt osalema selle läbivaatustel.
- arhitektuur peaks olema analüüsitud rakendatavate kvantitatiivsete omaduste suhtes (nagu maksimaalne läbilaskevõime) ja formaalselt läbi vaadatud kvalitatiivsete omaduste suhtes (nagu muudetavus) enne kui on liiga hilja seda muuta.
- arhitektuur peaks lubama luua endast algseid realisatsioone, milles kõik osad eksisteerivad kuid on minimaalse funktsionaalsusega. Sellist algset süsteemi peaks hiljem saama inkrementaalselt "kasvatada".
- arhitektuur peaks andma ressursside kasutamise kriitilised punktid, millede lahendused on selgelt määratletud ja mida järgitakse (näiteks võrgu koormus või töökiirus).

Soovitused struktuurile:

- arhitektuur peaks sisaldama hästimääratletud mooduleid, mille funktsionaalsed ülesanded on jagatud info peitmise ja funktsioonide lahususe põhimõtteid järgides. Igal moodulil peab

olema hästi määratletud liides, mis kapseldab/peidab muutuvad aspektid ülejäänud tarkvara eest.
- moodulid peaks peegeldama funktsiooniide lahusust, mis lubab neid välja töötavatel meeskondadel töötada üksteisest sõltumatult.
- informatsiooni peitvate moodulite hulgas peaks olema arvutuskeskkonna eripärasid kapseldavad moodulid, mis isoleerivad suurema osa tarkvarast platvormi muutustest.
- arhitektuur ei tohiks kunagi sõltuda mingist olemasolevast tarkvarast/tööriistast või selle konkreetsest versioonist. Kui see sõltub mingist konkreetsest tootest, peab ta olema nii struktureeritud, et selle toote vahetamine on lihtne ja odav.
- moodulid, mis loovad andmeid peavad olema lahus andmeid tarbivatest moodulitest. See suurendab muudetavust kuna muutused on sageli koondunud kas andmete loomise poolele või andmete tarbimise poolele.
- paralleeltöötluse süsteemides peaks arhitektuuri kuuluma hästi määratletud protsessid või tööd, mis ei peegelda alati moodulite struktuuri. See tähendab, et protsessid võivad hõlmata mitmeid mooduleid ja moodul võib sisaldada protseduure, mis kuuluvad mitmesse protsessi.
- iga töö või protsess peab olema kirjutatud nii, et tema paiknemist konkreetsel protsessoril saab kergelt muuta (võibolla isegi töö ajal).
- arhitektuur peab sisaldama vähest hulka lihtsaid suhtlemismalle. See tähendab, et samat asja peab igal pool üle kogu süsteemi tegema samal moel. See parandab arusaadavust, vähendab arendusaega, tõstab töökindlust ja parandab muudetavust. See väljendab samuti arhitektuuri mõistelist ühtsust, mis lihtsustab arendustööd.

Kokkuvõttes on arhitektuur enam kui tehniliste nõudmiste tulemus. Arhitektuur mõjutab omakorda teda sünnitanud keskkonda.

Mis on tarkvara arhitektuur?

Arhitektuur on süsteemi struktuuride, mida on mitu (andmevood, moodulid, protsessid, jne.), kirjeldus. Arhitektuur on esimene asi tulevasest süsteemist, mida võib analüüsida seatud kvalieetide saavutamise seisukohalt ja mis on ka tulevase süsteemi projekti kavandiks (*blueprint*). Arhitektuur on samas ka süsteemi komponentide ja ühenduste vaheliste suhete kirjeldus.

Programmi või arvutisüsteemi tarkvara arhitektuur on süsteemi struktuuride struktuur, mis koosneb tarkvara komponentidest, nende väliselt vaadeldavatest omadustest ja nendevahelistest suhetest. Väliselt vaadeldavad omadused on eeldused, mida teised komponendid võivad antud komponendi kohta teha. Arhitektuur peab abstraheeruma osast informatsioonist (vastasel juhul vaatleksime me tervet süsteemi) ja samas sisaldama küllalt informatsiooni analüüsiks, otsustusteks ja riskide vähendamiseks.

Esiteks arhitektuur määratleb komponendid. Arhitektuur sisaldab informatsiooni komponentide vastasmõjust, jätttes kõrvale informatsiooni, mis ei mõjuta komponendi käitumist vastasmõjus. Informatsiooni peitmist kasutavates süsteemides suhtlevad komponendid liideste kaudu, mis jagavad komponendi kirjelduse avalikuks ja privaatseks osaks. Arhitektuur on avalikus osas, privaatse osa detailid, mis puudutavad vaid komponendi sisemist realisatsiooni pole arhitektuurilised.

Teiseks võib süsteem koosneda rohkem kui ühest struktuurist, millest ükski pole üksiti võttes arhitektuur. Selletõttu on arhitektuuris ka mitut tüüpi komponente ning nende vahel mitut tüüpi vastasmõjusid ja suhteid. Samuti on olemas erinevad kontekstid, milles neid struktuure vaadelda. Komponendid: moodulid, protsessid, ... Suhted: sisaldumine, kasutamine, sünkronisatsioon, ...

Kontekstid: arendus, töö, ...

Kolmandaks on igal tarkvarasüsteemil arhitektuur. Lihtsaimal juhul moodustab kogu süsteem ühe komponendi. See, et igal süsteemil on arhitektuur, ei tähenda, et see arhitektuur ka kellelegi teada oleks. On olemas vahe süsteemi arhitektuuri kirjelduse või määratluse vahel. Arhitektuur on sõltumatu tema kirjeldustest, sellest tuleneb arhitektuuri esituse tähtsus.

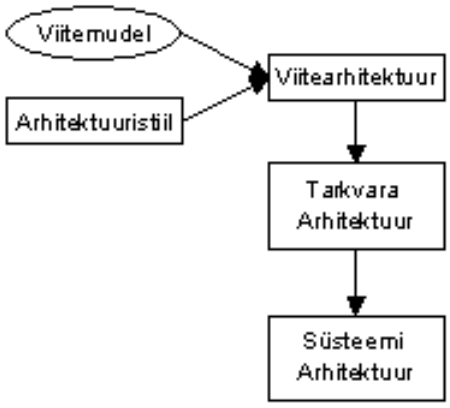
Neljandaks on iga komponendi käitumine arhitektuuri osa (vähemalt sellel määral, millisel seda saab vaadelda teiste komponentide poolt).

Arhitektuuri definitsioon võimaldab süsteemi hinnata lähtudes sellest, kas arhitektuur lubab või takistab süsteemil täita esitatud nõudmisi (funktsionaalseid, effektiivsus, jms.).

Arhitektuuri stiilid, viitemudelid ja viitearhitektuurid.

1. Arhitektuuristiilid on komponentide tüüpide ja nende tööaegse juhtimise ning andmesiirde mallide kirjeldus. Stiili võib kujutleda arhitektuurile seatud piirangute/kitsenduste hulgana, mis määratleb arhitektuuride pere, mille liikmed rahuldavad antud piiranguid/kitsendusi.
2. Viitemudel on funktsioonaalsuse tükeldus koos saadud tükkide vaheliste andmevoogudega. Viitemudel on teatud probleemi dekompositsioon osadeks, mis probleemi koostöös lahendavad. Viitemudelid on omased küpsele valdkonnale ja nad töötatakse välja valdkonna analüüsi käigus (näiteks kompilaatori või andmebaasi juhtimissüsteemi standartsed osad ja nende vahelised suhted).
3. Viitearhitektuur on tarkvara komponentidele (mis koostöös realiseerivad viitemudelis kirjeldatud funktsionaalsuse) kujutatud viitemudel ja komponentide vahelised andmevood.

Kui viitemudel jagab funktsionaalsuse, siis viitearhitektuur on selle funktsionaalsuse kujutus süsteemi dekompositsioonile. (Viitemudel on süsteemi funktsionaalsuse dekompositsioon ja viitearhitektuur on selle dekompositsiooni kujutus süsteemi dekompositsioonile)



On olemas mitmeid definitsioone tarkvara arhitektuurist:

- arhitektuur on kõrge-taseme disain (see on õige, kuid disaini hulka kuulub enam, kui arhitektuur)
- arhitektuur on süsteemi üldine struktuur (see definitsioon viitab vääralt sellele, et süsteemil on vaid üks struktuur)
- arhitektuur on programmi või süsteemi komponentide struktuur, nendevahelised suhted

ning põhimõtted ning juhised süsteemi disainimiseks ning arendamiseks (selles definitsioonis on liigsed tarkvaraprotsesse puudutavad osad, igal süsteemil on arhitektuur, mida võib jälgida ning analüüsida sõltumatult protsessist, mille abil see arhitektuur disainiti)
- arhitektuur on komponendid ja ühendused (selline definitsioon keskendub süsteemi tööaegsele konfiguratsioonile ega väljenda teisi struktuure nagu näiteks moodulite struktuur)
- arhitektuur on komponendid, ühendused ja piirangud/kitsendused (kui interpreteerida piiranguid/kitsendusi kui süsteemi komponentide käitumise määratlust puudub siiski veel väliste omaduste mõiste, mis on vajalik arhitektuuri kirjelduse kasutamiseks).

Miks on tarkvara arhitektuur tähtis

1. Huvitatud osapoolte vaheline suhtlus. Tarkvara arhitektuur kujutab endast ühist kõrge-tasemega abstraktsiooni, mida huvitatud osapooled saavad ühise arusaamise ning kokkulepete saavutamiseks kasutada.
2. Varajased disainiotsustused. Tarkvara arhitektuur väljendab kõige varajasemaid disainiotsustusi. Samuti on see kõige esimeseks võimaluseks ehitatavat süsteemi analüüsida.
3. Uuestikasutatav süsteemi abstraktsioon. Tarkvara arhitektuur sisaldab suhteliselt vähikest, mõistetavat mudelit sellest, milline on süsteemi struktuur ning kuidas tema komponendid koos töötavad. Seda saab kasutada teiste samasuguste nõudmistega süsteemide korral.

Igale süsteemist huvitatud osapoolele pakuvad huvi süsteemi erinevad omadused, mida arhitektuur mõjutab. Arhitektuur annab ühise keele, milles võib väljendada probleeme ning vajadusi ja lahendada neid piisavalt kõrgel tasemel.

Arhitektuur väljendab kõige varajasemaid disainiotsustusi:

- arhitektuur määrab realisatsiooni kitsendused
- arhitektuur määrab organisatsiooni struktuuri
- arhitektuur määrab süsteemi kvalitatiivsed omadused
- arhitektuur võimaldab ennustada süsteemi omadusi
- arhitektuur lihtsustab muudatuste planeerimist ja tegemist
- arhitektuur aitab evolutsioonilises prototüüpimises

Arhitektuur kui vahendatav ja korduvalt kasutatav mudel:

- tootepered jagavad ühist arhitektuuri (programmipere -- Parnas'76); arhitektuur määrab, mis on tootepere liikmetel jääv ja mis on muutuv. Sarnaselt kapitalimahutustega (investeeringutega) saab tootepere arhitektuurist väljatöötava organisatsiooni põhivara (*core asset*).
- süsteeme võib ehitada kasutades suuri, organisatsioonivälised arendatud/realiseeritud komponente. Kui vanad tarkvaraparadigmad on keskendunud programmeerimisele (mõõtes tulemusi koodiridades), keskenduvad arhitektuurile tuginevad tarkvara arenduse paradigmad iseseisvate komponentide kokkupanekule. Arhitektuuri üheks võtmeaspektiks on komponentide struktuuri, liideste ja tegevuspõhimõtete organiseerimine, milles kõige tähtsamaks põhimõtteks on vahetatavus. Et lahendada komponentide sobimatusest tulenevaid integreerimisprobleeme on soovitav (Garlan'i järgi) ilmutatud kujul määratleda komponendi arhitektuuriline kontekst.
- vähem on rohkem: maksab piirata disanivariantide sõnastikku. Eeliseks on ehitatavate

süsteemide disaini lihtsustumine, paranenud korduvkasutus, disaini regulaarsus, analüüsi lihtsustumine jms.
- arhitektuur võimaldab mallidel põhinevat komponentide arendust.
- arhitektuur võib olla väljaõppe aluseks.

Arhitektuurilised struktuurid

Kuigi sageli mõistetakse süsteemi struktuuri funktsionaalsuse terminites, on olemas mitmeid süsteemi omadusi lisaks funktsionaalsusele, nagu füüsiline hajutatus, protsesside vaheline side ja sünkronisatsioon, mida tuleb uurida arhitektuuri tasemel. Iga struktuur pakub meetodit mingite süsteemi omaduste analüüsiks. Samuti tuleb teatud omaduste saavutamiseks vastavaid struktuure teadlikult välja töötada/kujundada. Näiteks kasutusstruktuuri tuleb kujundada, et saavutada lihtne laiendatavus. Väljakutsestruktuuri tuleb kujundada, et vähendada pudelikaelu. Moodulite struktuuri tuleb kujundada, et saavutada modifitseeritavus/muudetavus. Mõned autorid kasutavad termineid vaade (*view*) ja struktuur (*structure*) sünonüümidena, raamatus eelistatakse terminit struktuur. Iga struktuur on abstraktsioon erineva kriteeriumi järgi.
Iga struktuur võib kasutada oma notatsiooni ja väljendada oma arhitektuurilist stiili, defineerida oma komponendid/elemendid, suhted/seosed, põhimõtted ja juhised.

Mõned kõige sagedasemad ja kasulikumad tarkvarastruktuurid:

- moodulite struktuur, mille elementideks on moodulid ja ühendusteks/seosteks alammooduli suhe; seda kasutatakse töö planeerimisel.
- kontseptuaalne e. loogiline struktuur, mille elementideks on süsteemile esitatud funktsionaalsete nõuete abstraktsioonid ja ühendusteks/seosteks on elementide vahel andmete jagamise suhe; üheks näiteks on viitemudel ja seda kasutatakse probleemivaldkonna olemite vastasmõjude uurimiseks/esitamiseks.
- protsessi struktuur või koordinatsiooni struktuur, mis on ortogonaalne moodulite ja loogilise struktuuriga, esitab süsteemi dünaamilisi aspekte, elementideks on protsessid ja lõngad ning ühendusteks/seosteks sünkroniseerimise, katkestamise, teise elemendi olemasolu nõudmise ja välistamise suhted (või muud suhted, mis tegelevad protsesside sünkroniseerimise või paralleelse tööga).
- füüsiline struktuur, näitab tarkvara kujutust riistvarale olles eriti vajalik hajussüsteemides, selle elementideks on riistvara olemid ja ühendusteks/seosteks on sidekanalid; see struktuur lubab analüüsida süsteemi effektiivsust, töökindlust, turvalisust jms.
- kasutusstruktuur, mille elementideks on protseduurid ja moodulid ning ühendusteks/seosteks on "kasutab" suhe; seda struktuuri kasutatakse alamsüsteemide eraldamiseks ja süsteemi laiendamiseks.
- väljakutsestruktuur, mille elementideks on alamprogrammid/protseduurid ja ühendusteks väljakutse suhe; seda kasutatakse programmi täitmise jälgimisel.
- andmevoo struktuur, mille elementideks on programmid ja moodulid ning ühendusteks/seosteks "saadab andmeid" suhe e. andmevoog (need suhted on märgistatud saadetavate andmete nimega); seda struktuuri kasutatakse süsteemile esitatavate nõuete jälgimisel.
- juhtimisvoo struktuur, mille elementideks on programmid, moodulid või süsteemi olekud ja ühendusteks/seosteks on "aktiviseerub peale" suhe e. juhtimisvoog; see struktuur on kasulik süsteemi funktsionaalse käitumise verifitseerimisel (kui ainus viis juhtimist üle anda on alamprogrammi väljakutse kaudu, on juhtimisvoo struktuur sarnane väljakutsestruktuuriga).
- klassistruktuur, mille elementideks on objektid ja ühendusteks on pärivus või klassi kuuluvuse suhted; see struktuur toetab sarnaste käitumiste hulkade analüüsi.

Hoolimata sellest, et struktuurid esitavad erinevaid vaateid süsteemile, pole need sõltumatud. Ühe struktuuri elemendid on ühendet teiste struktuuride elementidega ja ka neid ühendusi tuleb uurida. Väiksemate süsteemide korral võib eri struktuure kombineerida. Sageli on üks struktuuridest dominantne ja teised struktuurid sobitatakse sellega.

Tarkvara arhitektuuride alane uurimistöö on saanud alguse üle 25 aasta tagasi Fred Brooks'i Edsger Dijkstra, David Parnas'e ja teiste poolt. 1969 nimetasid Fred Brooks ja Ken Iverson arhitektuuri: "arvuti kontseptuaalne struktuur ... programmeerija poolt nähtuna". 1975 määratles Brooks arhitektuuri, kui: "kasutajaliidese täielik ja detailne kirjeldus" -- kui arhitektuur ütleb, mis juhtub, ütleb realisatsioon, kuidas see juhtub. Põhimõtteline vahe "mis" ja "kuidas" vahel on tänagi kehtiv. 1968 näitas Dijkstra, et lisaks korrektsele funktsionaalsusele maksab huvituda, kuidas tarkvara on tükeldatud ja struktureeritud. Dijkstra esitas kihilise struktuuri idee operatsioonisüsteemist kirjutades. Programmid olid grupeeritud kihtidesse ja ühe kihi programmid võisid suhelda vaid külgnevate kihtide programmidega. 1972 arendas Parnas seda liini, esitades fundamentaalsed põhimõtted:

- informatsiooni peitmine -- põhimõte süsteemi tükeldamisel komponentideks;
- komponendi kasutamine vaid tema liidese kaudu;
- mitmete üksteisest erinevate struktuuride olemasolu tarkvarasüsteemides;
- kasutusstruktuur;
- vigade töötluse põhimõtted komponentsüsteemides;
- iga programmi vaatlemine mingi programmipere liikmena (kus teatud disainiotsused on lahendatud ühtemoodi);
- süsteemi struktuuri mõju süsteemi omadustele

Parnas: Tarkvara komponendi liides on hulk eeldusi, mida võib teha antud komponendi kohta sõltuvalt kontekstist, milles komponenti kasutatakse.

Kruchten: 4+1 Vaate Mudel kirjeldab tarkvara, kasutades viite paralleelset vaadet, igaüks millest puudutab süsteemi teatud omadusi. Arhitektid väljendavad oma disainiotsuseid neljas vaates ja kasutavad viiendat nende illustreerimiseks ja valideerimiseks.

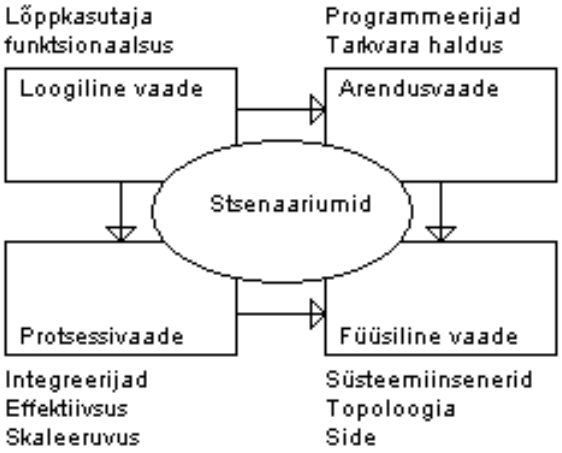
Süsteemi arhitektuuri kirjeldavad:

- loogiline vaade (objekt(i)mudel)
- protsessi vaade (paralleelsus ja sünkronisatsioon)
- füüsiline vaade (tarkvara kujutus riistvarale ja hajutamine)
- arendusvaade (tarkvara staatiline organisatsioon ja arenduskeskkond)
- stsenaariumite (kasutusjuhtude) vaade

Tarkvara arhitektuur Dewayne Perry ja Alexander Wolf'i järgi:

Tarkvara Arhitektuur = { Elemendid, Vormid, Põhimõtted/Kitsendused }

See kehtib iga vaate jaoks. Samuti võib iga vaate jaoks valida teatud arhitektuuristiili.



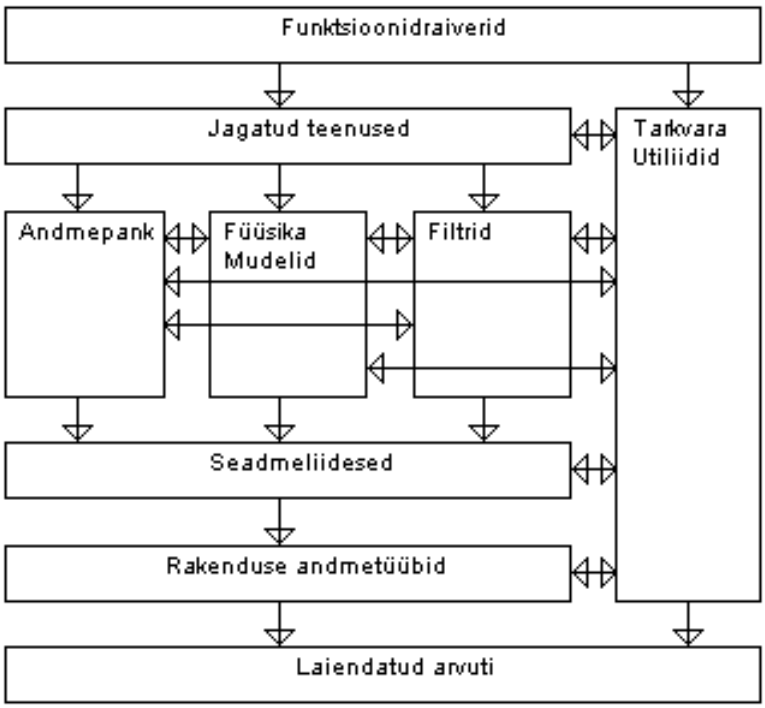
A-7E Arhitektuuristruktuuride kasutusanalüüs

A-7E avioonika süsteem oli projekt, kus pöörati erilist tähelepanu kolme arhitektuurilise struktuuri väljatöötamisele ja spetsifitseerimisele, et saavutada arendustöö ja muudetavuse lihtsus. Peatükis näidatakse, kuidas need struktuurid olid disainitud ja dokumenteeritud.

Kasutati kolme arhitektuuristruktuuri: moodulite struktuuri, kasutusstruktuuri, protsessistruktuuri:

- moodulite struktuur kirjeldab disainiaegseid suhteid süsteemi komponentide (mis on meeskondadele tööülesandeiks) vahel
- kasutusstruktuur kirjeldab tööaegseid kasutussuhteid süsteemi komponentide (moodulites paiknevate protseduuride) vahel
- protsessistruktuur kirjeldab süsteemi paralleelsust ja kujutust füüsilisele riistvarale

Kasutusstruktuur kirjeldab süsteemi tükeldust kihtideks.



Protsessistruktuur töötati välja peale kahe eelneva struktuuri kujunemist. Protsessistruktuur määratles, millised protseduurid igasse protsessi kuulusid (näitas, millised protseduurid tuli kodeerida taassisenetavatena (*reentrant*) -- võimelistena töötama paralleelselt mitmes lõngas, kasutades kaitstud andmeid või vastastikust välistust ja milliseid protseduure kõige sagedamini välja kutsutakse (näitas, kus optimiseerimine ennast ära tasub) ja millised protsessid ei või toimuda paralleelselt. See informatsioon võimaldas mõista vallasplaanurile (*off-line scheduler*) esitatavaid nõudmisi.

Andmevoostruktuure ei kasutatud, kuna puudusid kvaliteediomadused, mida see struktuur oleks ja teised kasutatud struktuurid poleks aidanud saavutada.

Arhitektuuri analüüs ja loomine

Kvaliteedi atribuudid

Kõikide arhitektuuride eesmärgiks on saavutada mingid kindlad tarkvara kvaliteedid. Peatükis kirjeldatakse tarkvara kvaliteete ja nende mõju tarkvara arhitektuuri seisukohalt. Kvaliteedid, võivad täielikult või osaliselt sõltuda arhitektuurist või olla arhitektuurist sõltumatud.

Süsteemi kvaliteetide (kvaliteediomaduste) saavutamine on alati kompromiss. Mistahes omaduse maksimeerimine võib toimuda ainult teiste omaduste vähenemise arvelt.

Paljud süsteemi kvaliteediomadused ei sõltu arhitektuurist, nagu kasutatavus (kasutaja liidese välimus ja kasutatavus pole arhitektuuri asi). Muudetavus on samas arhitektuurist väga sõltuv. Süsteemi effektiivsus aga sõltub nii arhitektuurist kui ka muudest tingimustest ja disainiotsustest (nagu algoritmide valik ja kodeerimistehnika).

1. Arhitektuur on kriitiline mitmete süsteemi kvaliteediomaduste realiseerimisel ja neid omadusi tuleb kavandada ning hinnata arhitektuuri tasemel.
2. Osa süsteemi kvaliteediomadusi ei sõltu arhitektuurist ja püüd analüüsida neid arhitektuuri kaudu ei vii sihile.

Süsteemi kvaliteedi omadused:

1. Süsteemi töös jälgitavad kvaliteediomadused:
    - effektiivsus/jõudlus (*performance*) -- süsteemi reaktsiooniaeg (*responsiveness*), töödeldavate sündmuste/tehingute arv ajaühikus, süsteemi läbilaskevõime; koos riistvara hinna langemisega effektiivsuse tähtsus väheneb.
    - turvalisus -- süsteemi võime vastu seista lubamatutele kasutuskatsetele, pakkudes oma teenuseid õigetele kasutajatele; lahenduseks on autentimisserverid, võrgumonitorid, sündmuste registreerimine, tulemüürid ja usaldatavad tuumad.
    - kättesaadavus -- aeg, mille kestel süsteem on töövõimeline (tõrgete vaheline aeg ja tõrkest taastumise kiirus), sellega on tihedalt seotud töökindlus (mõõdetakse keskmise tõrgete vahelise ajaga); keskmist tõrgete vahelist aega saab pikendada, kui teha süsteemi arhitektuur tõrkeid sallivaks.
    - funktsionaalsus (ainus arhitektuurist sõltumatu kvaliteediomadus) -- süsteemi võime teha tööd, milleks ta on ette nähtud; funktsionaalsus on struktuurist sõltumatu (struktuuriga ortogonaalne), s.t. et antud funktsionaalsust võib realiseerida mitmesuguste struktuuridega, seega on funktsionaalsus arhitektuurist sõltumatu.
    - kasutatavus -- õpitavus, kasutuseffektiivsus (*efficiency*), meeldejäävus (*memorability*), vigade välistamine (*error avoidance*), vigade töötlus, rahulolu.

2. Süsteemi töö ajal mitteavalduvad kvaliteediomadused:
    - muudetavus -- üks kõige rohkem arhitektuuriga seotud süsteemi omadus; muudatusi võib jagada kolme kategooriasse: ühe komponendi muudatus, mitme komponendi muudatus ja arhitektuuri(stiili) muudatus; muudatused tulenevad muudatustest ärivajadustes, mida võib jagada järgmiselt: võimete (*capabilities*) laiendamine või muutmine, mittevajalike võimete eemaldamine, uute tegevustingimustega kohanemine, ümberstruktureerimine.
    - porditavus (*portability*) -- süsteemi võime töötada erinevates arvutuskeskkondades (süsteemi võime saada lihtsalt realiseeritud erinevates arvutuskeskkondades).
    - uuestikasutatavus (korduvkasutatavus) -- tegelikult muudetavuse erijuhtum (hästi muudetav süsteem on tavaliselt ka lihtsalt uuestikasutatav).
    - integreeritavus/ühendatavus (*integrability*) -- sõltuv komponentide välisest keerukusest; integreeritavuse erijuhuks on koostöötavus (*interoperability*), mis väljendub süsteemi osade võimet töötada koos teiste süsteemidega.
    - testitavus -- seotud vaadeldavuse ja juhitavusega; et süsteem oleks testitav, peab olema võimalik juhtida iga komponendi sisemist olekut, sisendeid ja seejärel vaadelda tema väljundeid.
3. Süsteemi ärilised kvaliteediomadused:
    - turule jõudmise aeg.
    - maksumus.
    - planeeritud eluiga.
    - sihtgrupp/turg.
    - väljalaskegraafik.
    - olemasolevate (pärand)süsteemide laialdane kasutamine.
4. Arhitektuuri kvaliteediomadused:
    - mõisteline terviklikus (*conceptual integrity*) on arhitektuuri omadustest kõige tähtsam (Brooks)
    - korrektsus (*correctness*) ja lõplikus (*completeness*)
    - ehitatavus (*buildability*)

Arhitektuurilised vahendid kvaliteediomaduste saavutamiseks:

1. Milline struktuur aitab koostada ülesannete tükeldusskeemi (planeerida projekti ressursse), kasutada olemasolevaid komponente ja planeerida muudatusi? -- süsteemi moodulite struktuur.
2. Milline struktuur aitab tööajal saavutada käitumuslike ja kvaliteedinõudeid? -- arhitektuuristiilid annavad valmislahendused teatud olukordades.

Arhitektuuri väljatöötamisel tuleks hinnata:

- oodatud muudatusi (või muudatusi, mida samalaadsed süsteemid on vajanud)
- oodatud muudatusi, mis tulenevad ebatäpsetest ja mitmetähenduslikest nõudete spetsifikatsioonidest
- funktsionaalsuse kitsenemist tulenevalt ennetähtaegse väljalaske otsusest

Liikumine kvaliteedilt arhitektuurile: Arhitektuuristiilid

Kui süsteemi soovitud kvaliteedid on teada, säilib probleem kuidas disainida arhitektuur nende kvaliteetide saavutamiseks. Peatükk kirjeldab organisatsiooni sihtide, süsteemi kvaliteedi

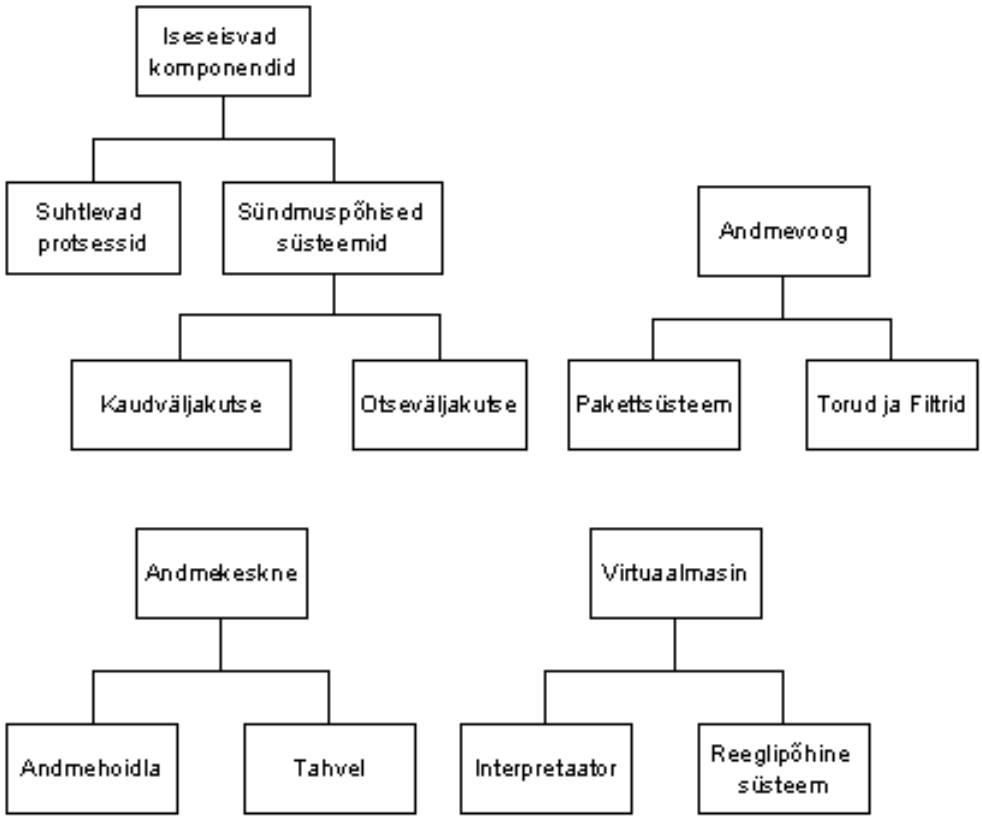
atribuutide ja arhitektuuri struktuuride vahelist seost. Samuti kirjeldab peatükk mitmeid tehnikaid, mida kasutatakse tööaegsete ja arendusaegsete kvaliteetide saavutamiseks.

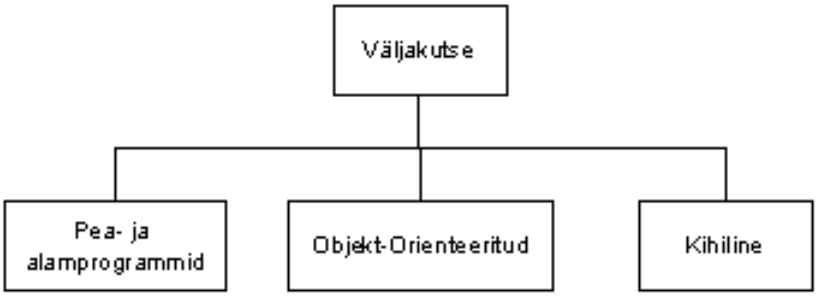
Raamatus jaotatakse mallid kahte kategooriasse: süsteemi mallid (arhitektuuristiilid) ja disainimallid. Kolmandat mallide kategooriat -- koodimalle raamatus ei käsitleta, kuna need pole iseloomult arhitektuurilised. Iga mall esindab hulka disaniotsuseid (disainiotsustusi), mida saab korduvalt kasutada.

Arhitektuuristiil koosneb mõnedest olulistest omadustest ja reeglitest nende kombineerimiseks selliselt, et arhitektuurne terviklikus säiliks. Tarkvara arhitektuuri stiili määravad:

- komponentide tüüpide hulk
- komponentide topoloogiline laotus/paigutus, mis väljendab nende vahelisi suhteid süsteemi töö ajal
- semantiliste kitsenduste hulk
- ühenduste hulk, mis vahendavad komponentide vahelist sidet, koordinatsiooni ja koostööd

Arhitektuuristiil defineerib arhitektuuride klassi (on abstraktsiooniks teatud arhitektuuride hulgale). Mary Shaw ja David Garlan on koostanud arhitektuuristiilide kataloogi:





Arhitektuuristiilide omaduste kategooriad:

- Juhtimise jagamine ja edasiandmine (topoloogia, sünkroonsus, seostamise aeg)
- Andmete liikumine läbi süsteemi (topoloogia, jatkuvus, juurdepääsu viis, seostamise aeg)
- Andmete ja juhtimise vastasmõju (kuju, suund)
- Võimalikud analüüsid
- Kasutatud komponentide ja ühenduste liigid

Arhitektuuristiilide omadustele vastav klassifitseerimine:

	Constituent parts		**Control issues**			**Data issues**			**Control/da**	
Style	**Compo-nents**	**Connec-tors**	**Topo-logy**	**Synchro-nicity**	**Binding time**	**Topo-logy**	**Conti-nuity**	**Mode**	**Binding time**	**Isomo shape**
Data flow: *dominated by motion of data through the system*									***Type of reasoni*** *Functional compo*	
Batch sequential	Stand-alone programs	Batch data	Linear	Seq.	r	Linear	Spor. hvol.	Passed, shared	r	Yes
Data-flow network	Transducers	Data stream	Arb.	Asynch.	i, r	Arb.	Cont. lvol. or hvol.	Passed	i, r	Yes
Pipes and filters										
Other data-flow substyles										
Call-and-return: *dominated by order of computation*									***Type of reasoni*** *Hierarchy (local*	
Main program/ subroutines	Procedures, data	Procedure calls	Hier.	Seq.	w, c	Arb.	Spor. lvol.	Passed, shared	w, c, r	No
Abstract data types	Managers	Static calls	Arb.	Seq.	w, c	Arb.	Spor. lvol.	Passed	w, c, r	Yes
Objects	Managers	Dynamic	Arb.	Seq.	w, c, r	Arb.	Spor.	Passed	w, c, r	Yes

	(objects)	calls					lvol.			
Call-based client-server	Programs	Calls or RPC	Star	Sync.	w, c, r	Star	Spor. lvol.	Passed	w, c, r	Yes
Layered	Various	Various	Hier.	Any	any	Hier.	Spor. lvol., cont.	Any	w, c, i, r	Often
Independent components: *dominated by communication patterns among independent, usually concurrent, processes*									***Type of reasoni*** *Nondeterministic*	
Event systems	Processes	Signals	Arb.	Asynch.	w, c, r	Arb.	Spor. lvol.	Multicast	w, c, r	Yes
Communicating processes	Processes	Message protocols	Arb.	Any but seq.	w, c, r	Arb.	Spor. lvol.	Any	w, c, r	Possib
Communicating processes substyles	Processes									
Data-centered: *dominated by a complex central data store, manipulated by independent computations*									***Type of reasoni*** *Data integrity*	
Repository	Memory, computa-tions	Queries	Star	Asynch. opp.	w	Star	Spor. lvol.	Shared passed	w	Possib
Blackboard	Memory, computa-tions	Direct access	Star	Asynch. opp.	w	Star	Spor. lvol.	Shared mcast	w	No
Virtual machine: *characterized by translation of one instruction set into another*									***Type of reasoni*** *Levels of service*	
Interpreter	Memory, state machine	Direct data access	Fixed hier.	Seq.	w, c	Hier.	Cont.	Shared	w, c	No

Topology: Hierarchical (hier.) | Arbitrary (arb.) | Star (star) | One-way (linear) | Determined by style (fixed)
Syncronicity: Sequential, one thread of control (seq.) | Synchronous (synch.) | Asynchronous (asynch.) | - Opportunistic (opp.)
Binding time: Write-time -- in source code (w) | Compile-time (c) | Invocation-time (i) | Runtime (r)
Continuity: Sporadic (spor) | Continuous (cont.) | High-volume (hvol.) | Low-volume (lvol.)
Mode: Multicast (mcast) | ...

Ühikoperatsioonid

Ühikoperatsioonid on hulk disainioperatsioone, mida arhitekt võib kasutada ühe arhitektuuri teiseks teisendamisel. Kõigepealt esitatakse ühikoperatsioonid ja siis näidatakse, kuidas neid on kasutatud kasutajaliideste viitemudelite ajaloolisel arengul alates 1980. Seda arengut jälgitakse, et seletada seda ühikoperatsioonides.

Näited:

1. WWW: Ühilduvuse (*interoperability*) näide.
   WWW loodi ühe organisatsiooni soovist vahetada informatsiooni oma uurijate vahel aga on oma esialgsetest sihtidest kaugele välja kasvanud. Peatükk kirjeldab WWW tarkvara arhitektuuri, selle muutumist WWW kasvu lubamiseks ja selle kasvu mõju organisatsioonidele kes seda kasutavad.
2. CORBA: Arvutusinfrastruktuuri tööstusstandardi näide.
   OMG loodi, et saavutada erinevate iseseisvate tarkvaratootjate toodete ühilduvust. CORBA on arhitektuur selle sihi saavutamiseks. Peatükk kirjeldab kuidas OMG loodi ja kuidas nende poolt loodud arhitektuurid, standardid ja viitemudelid nende sihte peegeldavad.

Arenduskvaliteetide analüüs arhitektuuri tasemel: Tarkvara arhitektuuri analüüsi meetod

On võimalik hinnata arhitektuuri, et näha kas ta lubab saavutada süsteemi teatud kvaliteete. Peatükk esitab meetodi selliseks hindamiseks muudetavuse seisukohalt. Meetodit illustreeritakse mitme näitega.

Arhitektuuri ülevaatused

Süsteemi ülevaatus arhitektuuri tasemel on enam kui lihtne süsteemi vaatlus. Ülevaatuse ajastamine, kokkulepped, sisendite ja väljundite määratlemine ja õige protsessi kasutamine on kõik tähtsad. Peatükk esitab "parimaid tööstuspraktikaid" arhitektuuri ülevaatuste korraldamiseks.

Näide:

Lennujuhtimise süsteemi väljatöötamisel oli sihiks pidev ilma tõrgeteta töö. See siht põhjustas mitmeid disainiotsuseid, mida peatükk kirjeldab. Lisaks arhitektuursetele valikutele, mis on põhjustatud töökindluse ja effektiivsuse soovist kirjeldatatkse ka arhitektuuri hindamist.

Arhitektuuridest süsteemideni

Arhitektuuri kirjeldamise keeled

Tavaliselt esitatakse arhitektuure mitteformaalselt. Viimase paari aasta jooksul on tekkinud arhitektuuri kirjeldamise keeled kui viis arhitektuure formaalselt esitada. Paljud sellised keeled on töötatud välja uurimisorganisatsioonide (teadusorganisatsioonide) poolt ja mõned on ka tootena kättesaadavad aga nad saavad tulevikus üha tähtsamateks. Peatükk kirjaldab kuidas hinnata arhitektuuri kirjeldamise keelte sobivust ja esitab näiteid nende kasutamise kasulikusest.

Arhitektuurile tuginev arendus

Arhitektuur on küll süsteemi kavandiks aga selle kavandi järgi on vaja veel süsteem ehitada. Protsess, millega liigutakse arhitektuurist töötava süsteemini sisaldab tööde jaotuse loomist suhtes arhitektuuri, arhitektuuri kasutamist süsteemi piiratud versiooni loomiseks, mallide kasutamist arhitektuuri struktuurides, korduvalt kasutatavate komponentide ühendamist süsteemi

ja lõpuks valmis süsteemi arhitekturile vastavuse kontrolli.

Lennusimulaator: Koostatavuse (integrability) näide

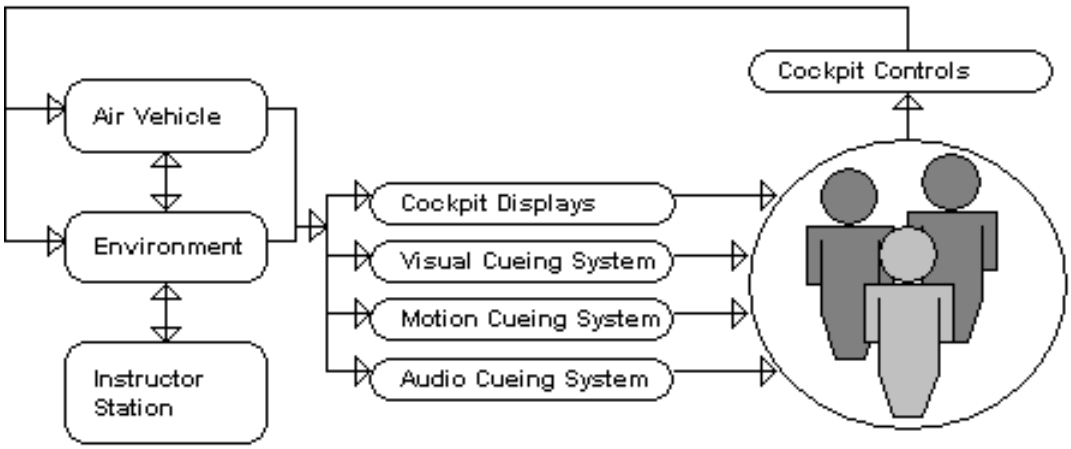
Peatükk kirjeldab lennusimulaatori arhitektuuri, näidates kuidas tähelepanu arhitektuurile võimaldas keerukas valdkonnas funktsionaalsetele ja töökindlusele esitatud nõuetele vastavate, paljude tarkvarainseneride poolt mõistetavate, kergelt koostatavate ja muudetavate suurte süsteemide pere ehitamist.

Moodsad lennusimulaatorid on peaaegu kõige keerulisemad tarkvarasüsteemid. Nad on tugevalt hajutatud, neil on ranged reaalaja nõudmised ja nad peavad olema kergelt muudetavad, et säilitada nende poolt simuleeritavate sõidukite ja keskkonna kõrget tõepärasust. Nende süsteemide loomine on keerukas nõudmiste tõttu effektiivsusele, koostatavusele (*integrability*), muudetavusele, laiendatavusele (muudetavuse erijuht, mida on vaja, et need süsteemid saaks modelleerida üha täpseminin ja üha suuremat osa reaalsest maailmast).

Peatükk kirjeldab lennusimulaatorite loomisel ette tulevaid raskusi ja uut arhitektuuristiili mis neid lahendab -- struktuurset mudelit, mis rõhutab:

- süsteemi alamstruktuuride lihtsust ja samasust nii et arhitektuur on lihtsalt mõistetav ja seega lihtsalt muudetav
- andmete ja juhtimise lahusust arvutusstrateegiate etteandmisega (arvutuste eraldamine otsustest, kuidas andmeid edastatakse ja kuidas toimub juhtimine; alamsüsteemide sõltumatus -- andmete andmine alamsüsteemist teise toimub läbi vahekomponentide, mis lihtsustab süsteemi koostamist ja muudetavust)
- komponentide erinevate tüüpide arvu minimiseerimist
- väikest hulka süsteemi koordineerimisstrateegiaid (mis lihtsustab keerukaid effektiivsuse ja ajajaotusega seotud küsimusi)
- disaini läbipaistvust (*transparency*) (lennusimulaatori osade tegemine simuleeritava lennumasina osade analoogideks)

1. Suhe arhitektuuri äritsüklisse
2. Nõuded ja kvaliteedid
    1. Reaalaja piirangud effektiivsusele
    2. Pidev arendus ja muutmine
    3. Suur maht ja kõrge keerukus
    4. Väljatöötajate paiknemine geograafiliselt hajutatud paikades
    5. Väga kallis silumine, testimine ja muutmine
    6. Ebaselge tarkvara struktuuri kujutus lennuki struktuurile
3. Arhitektuuriline lähenemine
4. Arhitektuuriline lahendus



1. Aja käsitlemine lennusimulaatoris
    1. Perioodiline aja haldamine
    2. Sündmustepõhine aja haldamine
    3. Sega-ajaga süsteemid
2. Arhitektuuristiili struktuurne mudel
3. Lennumasina täidetava mudeli komponentide konfiguratsioonid
    1. Aja sünkronisaator
    2. Perioodiline järjestaja
    3. Sündmuste haldur
    4. Surrogaat
4. Lennumasina mudeli rakenduse komponentide konfiguratsioonid
    1. Alamsüsteemide kontroller
    2. Komponent
    3. Skelet ja mallidel põhinev lihtsus
5. Viitearhitektuur
6. Rühmadeks dekomponeerimine
    1. N-ruudu kaart
7. Rühmade süsteemideks dekomponeerimine
    1. Kineetika rühma süsteemid
5. Sihtide saavutamine
    1. Effektiivsus
    2. Koostatavus
    3. Muudetavus
6. Kokkuvõte

Arhitektuuride korduvkasutamine

Tootepered: Arhitektuuriliste varade (assets) korduvkasutamine organisatsioonis

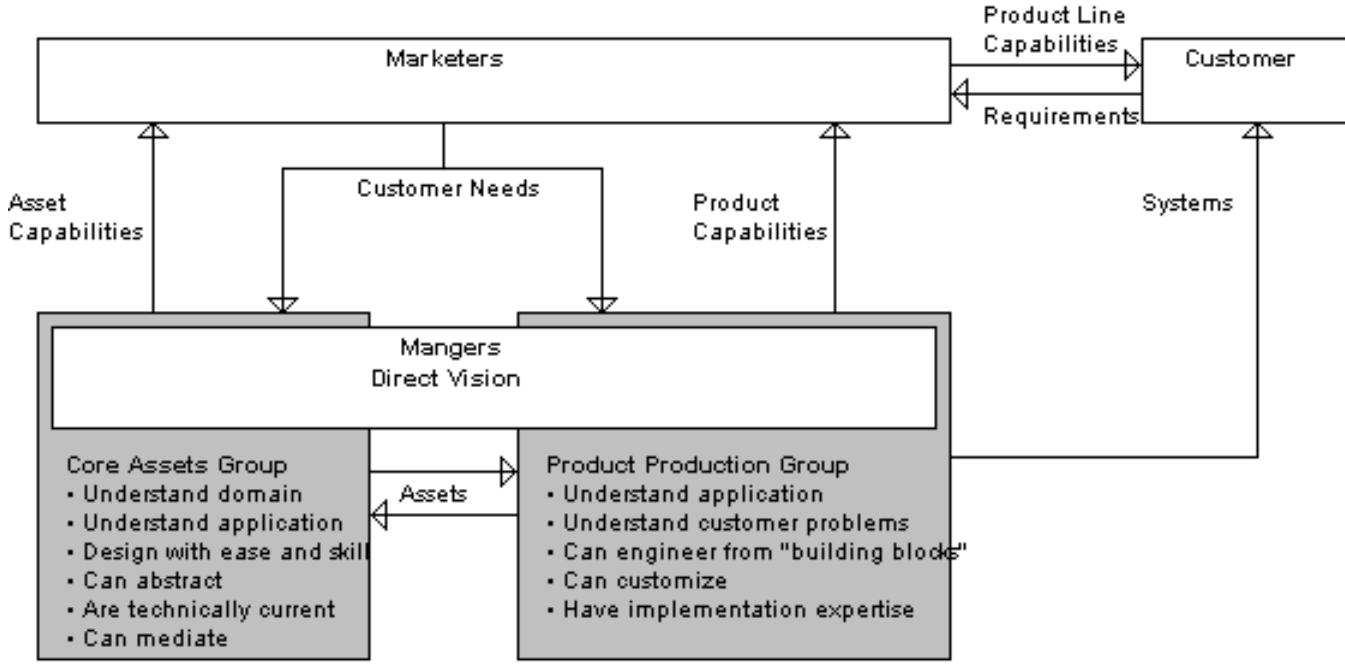
Arhitektuurid on eri tüüpi korduvkasutamise aluseks. Komponendid, mis on töötatud välja vastavalt mingile arhitektuurile on korduvalt kasutatavad, nagu arhitektuur isegi. Peatükk vaatleb komponentidele põhinevat arendustööd ja tooteperede arendust.

Siiani oleme rääkinud individuaalsetest süsteemidest aga organisatsioonile on arhitektuur suur aja ja *know-how* investeering. On loomulik püüda tulusid sellelt investeeringult suurendada kasutades samat arhitektuuri mitme süsteemi korral. See viitab tootepere, ühistest komponentidest koostatud toodete hulga, vajadusele.

Järgnevad varad lisaks arhitektuurile on tooteperes korduvalt kasutatavad:

- Komponendid -- komponente saab kasutada paljude toodete juures. Koodi korduvakasutusest erinevalt sisaldab komponentide korduvkasutamine ka disaini korduvkasutamist
- Personal -- personali saab siirdada vabamalt projektide vahel, kuna tooted on sarnased. Töötajate oskused on rakendatatavad kõikide tooteperesse kuuluvate toodete juures.
- Defektide eemaldamine -- ühes tootes komponendist eemaldatud defektid tõstavad ka teiste sama komponenti kasutatavate toodete kvaliteeti.
- Projekti plaan -- projekti plaan ja eelarve on ennustatavad ning tööde jaotust pole tarvis uuesti leiutada. Meeskonna suurus ja koosseis on ette teada.
- Effektiivsus -- effektiivsust mõjutavad mudelid ja analüüsid kehtivad kõikidele toodetele.
- Protsessid, meetodid, tööriistad -- konfiguratsiooni juhtimine, dokumentatsiooni plaan, tööprotsessid, tööriistad, arenduskeskkond, süsteemi genereerimise ja jaotamise protseduurid, kodeerimisstandardid, jpm. on samad kõikide toodete korral.
- Näitesüsteemid -- valmistooted on kasutatavad näitesüsteemide ja prototüüpidena. Projekti võimalikus (*feasibility*) pole enam tundmatu suurus.

1. Toodete loomine ja tootepere arendamine
   Organisatsioon, millel on tootepere, on arhitektuur ja hulk komponente, mis on tooteperega seotud. Aeg-ajalt organisatsioon loob uue tooteperesse kuuluva toote. Sellel tootel on omadusi, mis on ühised teiste tooteperesse kuuluvate toodetega ja omadusi, mis on erinevad.
   Üks tooteperega seotud probleemidest on selle areng. Aja jooksul peab tootepere või täpsemalt tuumvarad (*core assets*), millest tooteid ehitatakse, arenema.
   Tootepere arengut mõjutavad:
   - Tootepere komponentide uute versioonide väljastamine ja uute toodete loomise vajadus.
   - Uute väliselt loodud komponentide lisamine tooteperele. Uute toodete vajadus kasutada väliselt loodud komponentides sisalduvaid uusi tehnoloogiaid.
   - Uute omaduste/võimaluste lisamine tooteperele et vastata kasutajate vajadustele ja võstlejatele.
   Teine probleem on üksikute toodete areng. Oletame, et tootesse on vaja lisada uut funktsionaalsust. Tuleb otsustada, kas see funktsionaalsus kuulub tooteperesse. Kui jah, võib toote lihtsalt uuesti ehitada kasutades tooteperet, kui aga ei, tuleb otsustada kas toode eraldada tooteperest või laiendada tootepere varasid. Viimane on õige, kui on oodata samalaadse funktsionaalsuse vajadust tulevastes toodetes, aga see suurendab muudatuse töömahukust.
   Kolmas probleem on mida teha tootepere arenedes olemasolevate toodetega. Toodete hoidmine tooteperega sünkroonis vajab aega ja tööd aga seda mitte tehes muutub tulevikus toodete uuendamine töömahukamaks.
   See kõik teeb tootepere omaniku ja sellesse kuuluvate loodud toodete omanike suhte sarnaseks tarkvara tarnija ja tema klientide suhtega.
2. Organisatsiooni mõju tooteperele
   Tootepere mõjutab organisatsiooni, selle suhet klientidega ja liikmete välja õpet. See mõju tuleneb sellest, et tootepere korral peab organisatsioon vaatlema arhitektuuri, komponente, teste ja kõike seda mis alguses üles loetud kui organisatsiooni varasid.
   1. Struktuur
      See kuidas tootepere mõjutab organisatsiooni struktuuri sõltub sellest kui palju tooteid organisatsioon toodab ja kuidas on otsustatud hoida tooteid ja tooteperet sünkroonis. Hoolimata kõigest kui organisatsioonil on tootepere ja see peab arenema, peab olema organisatsioonis üksus, mis vastutab arhitektuuri ja teiste tootepere tuumvarade eest. See omakorda tähendab, et nii tuumvarade rühma kui ka tootearendusrühma(de) juhid peavad aru saama, mida tähendab tootepere ja kuidas see mõjutab toodete ja klientide tuleviku.



2. Suhe klientidega
   Tuumvarade (eriti arhitektuuri ja komponentide) omaniku ja arendajate suhe meenutab kaubandusliku tarkvara müüja ja selle klientide vahelist suhet. Toodete väljatöötajad on klientide agentideks, võideldes et nende klientide nõudmised rahuldataks tuumvarade omaniku poolt.
   Tuleb luua meetodid tagasisideks arhitektuuri omanikule, kes peab aru saama arhitektuuri muudatuste mõjust ja need heaks kiitma. Arhitektuuri muudatused tuleb grupeerida väljalasetesse (*releases*).
   Müügimehed võivad pakkuda paremaid tingimusi klientidele, kes nõustuvad kitsendustega, mida seab nende süsteemi tooteperesse kuulumine.
   Näide:
   Klient: (osutades paberikuhjale) "Siin on minu nõudmised. Kas te saate sellega hakkama?"
   Müügimees: (kurvalt meenutades möödunud häid aegu, kus kõik mis talt nõuti oli naeratus, noogutus ja lepinguga tagasitulek) "Ma olen kindel, et saame. Aga ma pean ütlema, et kui te olete nõus seda nõudmist siin lõdvendama ja ka seda siin ning seda siin vaid väheke muutma, võiksime teie nõudmised rahuldada meie tooteperet kasutades."
   Klient: "See on teile kasulik, aga mida see minule annab?"
   Müügimees: "Kuna teie nõudmistele vastava süsteemi tegemine läheb maksma $6 miljonit, oodake kaks aastat ja saate süsteemi, mida pole kellegi teisel. Kui te aga muudate oma nõudmisi vähe, maksaks see $1 miljon ja kahe kuu pärast oleks teil süsteem,, mis kasutab samu komponente, mida paljud kliendid on juba kuus aastat kasutanud. Millise variandi te valite, on täiesti teie teha."
   Klient: (kellele kunagi sellist valikut pole esitatud või kes kunagi pole õieti mõistnud, kui palju "erilised" nõudmised tegelikult maksavad) "Tõesti? Me võtame muudetud variandi. Näeme kahe kuu pärast."
3. Komponentidel põhinevad süsteemid
   Järgnevalt koondame oma tähelepanu mitte tooteperele vaid tootepere arhitektuuri "täitvatele" komponentidele. Üha suurem osa tarkvarast ehitatakse olemasolevatest komponentidest -- "osta, ära ehita ise" on tarkvara tegijate loosungiks. Aga ostmine tähendab väiksemat kontrolli süsteemi loomise eri aspektide üle. Kuidas sellist kontrolli kaotust lepitada sooviga saavutada paremat kvaliteeti? Vastus esimene osa peitub meie eelduses, et suurte süsteemide kvaliteet peitub peamiselt nende arhitektuuris. Teine osa aga on komponentide hoolikas integreerimine nii, et nad ei rikuks arhitektuuri ja selle poolt

väljendatavaid kvaliteete.
Peatükis 18. (📄) kirjeldatakse süsteemi, mis on ehitatud CORBA't kasutades peaaegu täielikult ostetud komponentidest.
Süsteemide ehitamine valmiskomponentidel põhinevate tooteperedena pakub järgnevaid eeliseid:
- võimalus kiiremini ära kasutada uute toodete ja uue tehnoloogia eeliseid
- oluliselt vähenenud turule tuleku aeg
- töötajate suurenenud tööviljakus, kuna raskuspunkt on kodeerimise asemel korduval kasutamisel ja integreerimisel
- väljatöötajate spetsialiseerumine organisatsiooni tegevusala(de)le
- töökindlamad komponendid, kuna süsteemis kasutatavad komponendid on välja töötatud terve klassi süsteemide jaoks ja eelnevad kasutajad on leidnud rohkem vigu kui selguks testimisel
- muudetavamad süsteemid, kuna vanade mitterahuldavate komponentide asemele võib süsteemi liita asenduskomponente
- laiendatavamad süsteemid, kuna süsteemi võib liita uut tüüpi komponente
Kui komponendid vastavad laialt levinud standarditele on võimalikud veelgi enamad eelised (vaata peatükk 17. 📄).
Muidaugi pole kõik see lihtsalt saavutatav. Mitte kõik komponendid ei ole ühilduvad isegi kui nii tootja poolt väidetakse -- näitena olgu toodud ühe andmebaasi juhtimissüsteemi teisega vahetamisel ette tulevad raskused. Komponendid on sageli peaaegu ühilduvad. Halvemad on juhud, kus komponendid näivad ühilduvat -- koostatud kood kompileerub ja isegi töötab aga süsteem annab vääraid resultaate, kuna komponendid ei tööta nagu oodatud. Sellised vead võivad olla raskesti avastatavad, eriti reaalaja süsteemides, kus komponenid võivad toetuda näiliselt ohututele oletustele ajastamise või sündmuste järjekorra kohta.
Komponendid, mis pole valmistatud spetsiaalselt teie süsteemi jaoks, ei pruugi vastata teie nõudmistele. Veelgi halvem on see, et te ei tea, kas kom ponendid sobiva, enne kui te pole neid ostnud ja proovinud, kuna liidesed ei kajasta kvaliteediatribuute.
1995 aasta konverentsil *International Conference on Software Engineering* võtsid Garlan, Allen ja Ockerbloom kasutusele termini arhitektuuriline sobimatus et kirjeldada komponentidel põhinevate süsteemide koostamisraskusi. Nad kirjeldasid probleemi, kui erinevates komponentides sisalduvate oletuste ebakõla.
Arhitektuuriline sobimatus ilmneb tavaliselt süsteemi integreerimisel -- süsteem ei kompileeru, ei linku või ei tööta. Arhitektuuriline sobimatus on liideste sobimatuse erijuht, kui liidest mõista samuti kui Parnas seda on defineerinud: oletused, mida komponendid üksteise kohta võivad teha. See liidese definitsioon läheb kaugemale, kui kahjuks praktikas üldiselt kasutatav: komponendi rakendusprogrammeerija liides (API) -- programmid/funktsioonid ja nende parameetrid ja võibolla mingi käitumise kirjeldus, mis on vaid väike osa komponendi korrektseks kasutamiseks vajalikust informatsioonist.
Liidese oletusi on kahesuguseid. Pakub-oletused kirjeldavad teenuseid, mida komponent pakub oma kasutajatele ja klientidele. Nõuab-oletused kirjeldavad teenuseid ja ressursse, mida on vaja, et komponent korralikult töötaks. Sobimatus tekivab siis, kui kahe komponendi pakub- ja nõuab-oletused ei vasta teineteisele.
Võimalikud väljapääsud liideste sobimatusest:
- vältige seda -- spetsifitseeriga ja kontrollige komponente hoolikalt (tooteperes kasutamiseks loodava komponendi väljatöötaja seisukohalt)
- selgitage komponente hinnates (*qualification*) välja tekkinud sobimatus
- parandage väljaselgitatud sobimatused komponente adapteerides (võõraste komponentide tooteperesse integreerija seisukohalt)

1. Liideste sobimatuse kõrvaldamise tehnikad
   Ilmne lahendus on sobimatu komponendi koodi muutmine, see aga pole alati võimalik.
   1. Pakendid (*wrappers*) -- peitmine, kus mingi komponent on ümbritsetud alternatiivse

abstraktsiooniga. Klient kasutab pakendatud komponendi teenuseid ainult läbi pakendi poolt pakutud alternatiivse liidese. Komponendi pakkimist võib vaadelda talle alternatiivse liidese andmisena. Võib vaadelda järgnevaid liidese teisendusi:
- komponendi liidese elemendi teisendamine teiseks elemendiks
- komponendi liidese elemendi peitmine
- komponendi liidese elemendi säilitamine muutusteta

2. Sillad (*bridges*) -- sild teisendab mingi komponendi mingi nõuab-oletuse mingi teise komponendi mingiks pakub-oletuseks. Põhiline pakendi ja silla erinevus on selles, et sild pole seotud mingi konkreetse komponendiga. Samuti peab mingi väline agent (võimalik et üks silla abil ühendatud komponentidest) silla ilmutatud kujul käivitama. Sillad on tavaliselt mööduvad (*transient*) protsessid. Sillad keskenduvad tüüpiliselt kitsamale liidese teisenduste hulgale kui pakendid, seda sellepärast, et sild on seotud teatud oletustega. Mida enam oletusi on sillaga seotud, seda väiksem on võimalike komponentide hulk, mida sild võib ühendada.
3. Vahendajad (*mediators*) --vahendajatel on nii pakendite kui ka sildade omadused. Põhiline vahendajate ja sildade erinevus on see, et vahendajad sisaldavad planeerimisfunktsiooni, mis töö ajal otsustab vajaliku liidese teisenduse (silla puhul otsustatakse see silla loomisel). Vahendajad omades piisavatsemantilist keerukust ja töö aegset autonoomiat, on sarnaselt pakenditele tarkvara arhitektuuri osad.

2. Liideste sobimatuse vältimise tehnikad
   Vaadeldes probleemi komponentide väljatöötaja seisukohalt, on võimalik ja kasulik spetsifitseerida disaini algetapil, niipalju kui võimalik või kõik oletused, mida komponent oma keskkonna kohta teeb. Hoolikas liideste kirjeldamine kõrvaldab integreerimise kui omaette sammu süsteemi ehitamisel.
   On olemas teatud eelised ühele komponendile ühe üleüldise asemel mitme liidese kirjeldamises. Täpsem komponentide vaheliste sõltuvuste juhtimine teeb teatud süsteemi arengu ennustatavamaks. Samuti võib arhitektuuristiil eeldada liideste kanoonilisi kujusid.
   Parametriseeritav liides on liides, mille pakub- ja nõuab-oletused on muudetavad mingi muutuja väärtuse muutmisega enne komponendi poolt pakutava teenuse kasutamist. Parametriseeritava liidese korral on adapteeriv kood jagatud kaheks: komponendist väljaspool, kus parameetrite väärtused seatakse ja komponendi sees, kus parameetrite väärtustele vastavalt muudetakse komponendi liidest.
   Nagu vahendaja on sild, millele on lisatud planeerimisfunktsionaalsus, on häälestuv liides (*negotiated interface*) parametriseeritav liides, millele on lisatud vigu parandav funktsionaalsus. Häälestuv liides võib ennast ise parametriseerida või olla parametriseeritud väljast. Isekonfigureeruv tarkvara peab sisaldama häälestuvaid liideseid.
   Nagu pakendite abil võib peita liideste sobimatust, võib vahendajaid kasutada häälestuvate liideste lisamiseks mittehäälestuvatele komponentidele.
4. Kokkuvõte
   Tootepere lubab tuntavat kasu nii toodete loomise hinnas kui ka kiiruses. Et seda saavutada peab organisatsioon välja töötama strateegia tootepere ja sellesse kuuluvate toodete sünkroonis hoidmiseks. Peab välja arendama strateegia tootepere arhitektuuri uuendamiseks mujal väljatöötatud komponentide kasutamiseks ja muutma oma ehitust tooteperel põhinevale tootmisele sobivaks.

CelsiusTech: Tootepere arenduse näide

CelsiusTech on organisatsioon, mis edukalt realiseeris arhitektuurile toetuva tootepere. Peatükk kirjeldab seda arhitektuuri ja näitab miks see oli eduks oluline. Ilma sellise lähenemiseta poleks

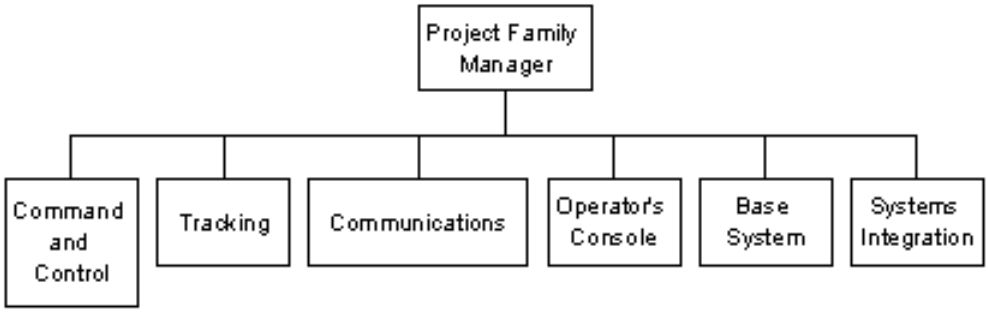
kirjeldatud süsteeme olnud võimalik ehitada -- CelsiusTech'il lihtsalt polnud piisavalt inimesi. Tootepere lähenemine muutis vastavalt organisatsiooni struktuuri ja äriprotsessi.

1. Suhe arhitektuuri äritsüklisse
   CelsiusTech Ab (kuulub koos Bofors'i Kockums'i, FFV Aerotech'i ja Telub'iga samasse kaitsetööstuse gruppi ja selles töötab ligikaudu 2000 töötajat) on Rootsi laevastiku tarnija on edukalt võtnud kasutusse tootepere lähenemise keerukate tarkvararikaste süsteemide loomisel. Nende tootepere nimi on Ship System 2000 (SS2000) ja see koosneb ligikaudu tosinast komando- ja juhtimissüsteemist (C3 -- command, control and communication systems) skandinaavia, kesk-ida ja lõuna Vaikse Ookeani laevastikele.
   1. Firma taust:
      1. Phillips Elektronikindustrier AB
      2. 1989 -- Bofors Electronics AB
      3. 1991 -- NobelTech AB
      4. 1993 -- CelsiusTech AB
   2. Ship Systems 2000 tootepere
      See tootepere (sisemise nimega Mk3) kujutab endast integreeritud süsteemi, mis ühendab sõjalaeva kõiki relvasüsteeme, komando- ja juhtimisfunktsioone ning sidesüsteeme. Tüüpiline süsteemi konfiguratsioon sisaldab 1 kuni 1.5 miljonit rida Ada koodi, mis töötab hajutatult 30 kuni 40 mikroprotsessoriga lokaalvõrgul.
      Ehitatud on lai valik laevastikusüsteeme pealvee- ja alveelaevadele:
      - Rootsi Göteborg klassi 380 tonnised rannakaitse korvetid (KKV)
      - Taani 300 tonnised SF300 klassi patrulllaevad
      - Soome 200 tonnised Rauma klassi kiired ründelaevad (FAC)
      - Austraalia ja Uus-Meremaa 3225 tonnised ANZAC klassi fregatid
      - Taani 2700 tonnised Thetis klassi ookeani valvelaevad
      - Rootsi 1330 tonnised Gotland A19 klassi allveelaevad
      - Pakistani Type 21 klassi fregatid
      - Omaani Vabariigi patrulllaevad
      - Taani Niels Juel klassi korvetid
      Kokku on müüdud erinevatesse maailma maadesse 55 Mk3 süsteemi. Süsteemid, mis sellesse tooteperesse kuuluvad on väga erinevad, iga maa vajab erisuguseid viise informatsiooni esitamiseks, andurid ja relvasüsteemid ning nende liidesed on erinevad, allveelaevadel on teised nõudmised kui pealveelaevadel. Kasutatavad arvutusplatvormid on 68020, 68040, RS/6000 ja DEC Alpha ning operatsioonisüsteemidest on kasutuses OS2000 (CelsiusTech'i toode), AIX, Ultrix, jt.
   3. Ökonoomia suurusest: CelsiusTech'i tulemused
      1. Lühenenud ajakavad
         Esimene süsteem valmis ligikaudu 9 aastaga ja kuna kaks süsteemi oli tellitud peaaegu samal ajal võeti vastu otsus kasutada tootepere lähenemist (tootepere pidi põhinema esimesel süsteemil). Teine süsteem valmis ligikaudu 7 aastaga. Teine süsteem ehitati paralleelselt esimesega valideerides tooteperet. Kuigi kumbagi süsteemi tegemine eraldi ei näidanud tööviljakuse kasvu valmis kaks süsteemi (ja lisaks tootepere) umbes sama tööjõu hulgaga kui tavaliselt üks süsteem. Järgnevad süsteemid vajasid juba oluliselt lühemat valmimisaega ja viimased kaks süsteemi seitsmest on püsinud kindlalt ja ennustatavalt graafikus.
      2. Koodi korduvkasutamine
         Keskmiselt 70-80% süsteemidest koosneb komponentidest, mida kasutatakse muutmatul kujul.
      3. Kesksete varade kasutamine ärivaldkonna laiendamiseks
         STRIC on uus Rootsi Õhujõudude õhukaitsesüsteem. Kasutates abstraktsiooni laevast, mille asukoht ei muutu sageli ja mille kreen ning different on püsivalt null ja

tänu SS2000 arhitektuuri paindlikusele võis CelsiusTech kiirelt ehitada STRIC arhitektuuri, kasutades 40% oma põhivaradest.

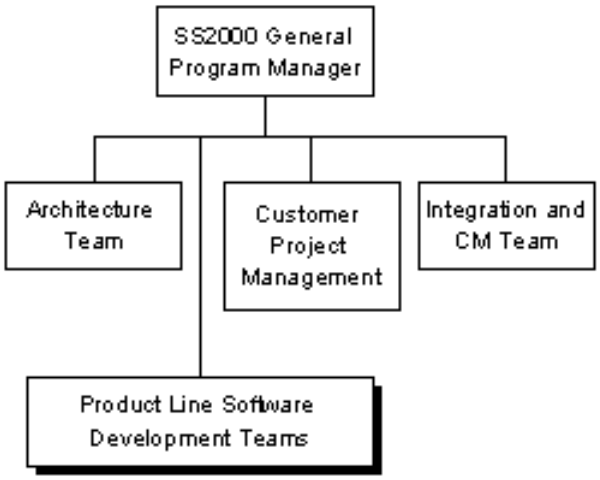
4. Mis motiveeris CelsiusTech'i
   Et mõista mis sundis CelsiusTech'i võtma vastu otsust luua tootepere, vaatame kust nad algasid. Enne 1986 oli firma välja töötanud üle 100 tulejuhtimissüsteemi 25's erinevas konfiguratsioonis, mahuga alates 30000 kuni 700000 koodireani.
   Alates 1975 kuni 1980 mindi üle 16 bitiste reaalaja digitaalsüsteemide peale (Mk2 süsteemid). Alates 1980 kuni 1985 tekisid klientidel nõudmised integreerida tulejuhtimissüsteemid komando- ja juhtimissüsteemidega, kujunes välja hajutatud süsteemide võrk (Mk2.5 süsteemid) mis olid oluliselt suuremad (kuni 700000 koodirida). Nende süsteemide korral püüti kasutada samu väljatöötamismeetodeid kui eelnevate väikeste süsteemide korral aga tekkisid raskused projekti ajakavas püsimisega ning eelarve ületamised.
   Aastal 1985 võitis firma kaks tellimust mis nõudsid oluliselt suuremat ja keerukamat süsteemi kui Mk2.5 süsteemid mis olid juba olnud aja- ja eelarveraskustes. Oli selge, et jätkates vanaviisi pole võimalik uusi süsteeme luua, ainuüksi vajaliku tööjõu hulk oleks seda takistanud.
   See situatsioon põhjustas uuele äristrateegiale ülemineku -- eraldiseisvate süsteemide asemel süsteemide seeriate või perede müük. Nii saigi alguse SS2000 tootepere. Teiseks teguriks sai süsteemide paindlikuse nõue, mis tulenes mereväe süsteemide 20 kuni 30 aastasest elueast, mille kestel tuli arvestada tehnoloogia ja ohtude arenguga .
   Sihiks sai paindlike ja töökindlate ehituskivide loomine, milledest saaks suhteliselt hõlpsalt luua tooteperesse kuuluvaid uusi süsteeme. Uute nõudmiste ilmnemisel saaks luua uusi ehituskive säilitades nii lähenemise elujõulisuse. Võeti vastu strateegiline otsus luua uus süsteemide sugupõlv (Mk3), uue tark- ja riistvaraga, mis saaks järgnevaks kümneks, kahekümneks aastaks uute süsteemide infrastruktuuriks.
5. Kõik oli uus
   VAX/VMS --> Motorola 68000
   vähene hulk protsessoreid --> suur (liiane) hulk tugevalt hajutatud mikroprotsessoreid
   RTL/2 --> Ada83
   struktuurne disain --> objekt-orienteeritud disain
   kose-mudel --> prototüüpiv, iteratiivne mudel
6. Ärikonteksti analüüs
   1. Omanike vahetumine
      Kõrgem juhtkond oli tegevuses muude asjadega ja keskastme juhtkond sai vabalt arendada tootepuret. Vastasel juhul oleksid vajalikud suured investeeringud äratanud kõrgema juhtkonna tähelepanu ja põhjustanud vahelesegamise.
   2. Vajadus on leiutiste ema
      Kahe suure tellimuse võitmine oli CelsiusTech'is kriisiks -- firma ellujäämine oli kaalul.
   3. Tehnoloogia muutustega kaasaskäimine
      Järgnevad tehnoloogiad olid veel küpsemata:
      - Ada ja objekt-tehnoloogia
      - võrgud ja hajussüsteemid
      - suurte projektide tarkvara arenduskeskkonnad
      - mikroarvutid
      - porditavad operatsioonisüsteemid
      - avatud süsteemide standardid
      Ligi kolmandik CelsiusTech'i algsest investeeringutest läks selliste varade ehitamiseks, mida nüüd on võimalik osta.
7. Organisatsiooni struktuur
   1. Eelnev projektorganisatsioon
      Projekti juhtis projektijuht, kes kasutas põhiliste funktsionaalsete alade teenuseid.

Igat funktsionaalse ala juhtis omakorda projektijuht.



Selline osakonnastunud struktuur põhjustas:
- süsteemianalüüsi jaotamist funktsionaalsete alade kaupa
- piiratud suhtlemine funktsionaalsete alade vahel, mis põhjustas erinevaid interpretatsioone
- liideste sobimatus avastati alles integreerimisel
- funktsionaalsete alade juhid ei mõistnud teisi süsteemi osi
- funktsionaalsete alade juhtide motivatsioon tegutseda meeskonnana oli väike

2. SS2000 organisatsioon 1986 - 1991
Tootepere arendust juhtis SS2000 programmi üldjuht, kes oli vastutav nii tootepere kui ka sellele toetuvate kliendisüsteemide arenduse eest. Temale allusid otseselt funktsionaalsete alade juhid. Loodi väike tehnoloogiale suunatud arhitektuurirühm, millele kuulus täielikult arhitektuur ja mis allus otseselt programmi üldjuhile. Kuna mitmete versioonide koordineeritud valmimine oli tähtis loodi integreerimis- ja konfigureerimisrühm, mis samuti allus otse programmi üldjuhile

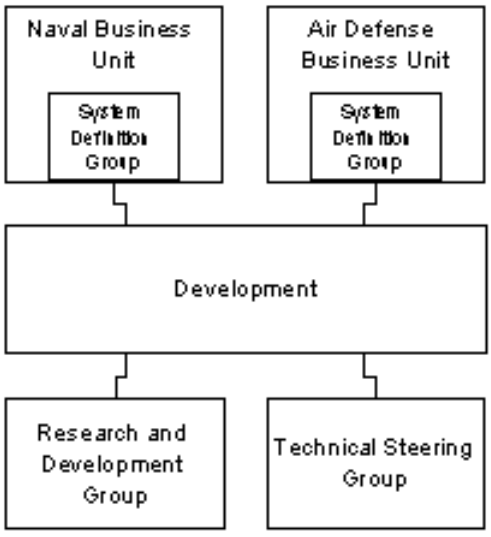


Arhitektuurirühm oli vastutav arhitektuuri väljatöötamise ja arendamise eest säilitatdes nii ühtse interpretatsiooni kõikides funktsionaalsetes alades. Eriliselt kuulus arhitektuurirühma kompetentsi:
- tootepere mõistete ja põhimõtete väljatöötamine
- kihtide ja nende vaheliste liideste identifitseerimine
- liideste määratlemine, nende tervikliikuse ja juhitud arengu tagamine
- süsteemi funktsioonide jagamine kihtidesse
- üldiste mehhanismide ja teenuste identifitseerimine
- üldiste mehhanismide, nagu veatöötlus ja protsesside vaheline side, defineerimine, prototüüpimine ja pealesundimine
- tootepere mõistete ja põhimõtete õpetamine projekti tööjõule

Algne arhitektuur loodi kahe vanema inseneri poolt kahe nädalaga, see on siiani tooteliini raamistikuks. Esimene iteratsioon koosnes mõistetest, kihtide määrangutest ja ligikaudu 125 süsteemse funktsiooni identifitseerimisest, nende

jaotamisest kihtide vahel ja põhilistest hajutamise ja sidemehhanismidest. Peale seda laiendati arhitektuurirühma kõikide funktsionaalsete alade peadisaineritega ja täielik arhitektuurirühm koosnedes kümnest insenerist jätkas arhitektuuri laiendamist ja arendamist. See on oluliselt erinev minevikust, kus funktsionaalsete alade juhid olid iseseisvad.
Integreerimis- ja konfigureerimisrühm vastutas:
- testimisstrateegiate, testimisplaanide ja testide väljatöötamise eest
- testide koordineerimise eest
- integreerimisajakavade eest
- testitud ja integreeritud alamsüsteemide väljalaske eest
- konfiguratsioonide juhtimise ja teekide loomise eest
- installatsioonimeedia loomise eest

3. SS2000 organisatsioon 1992 - 1994
   Raskuspunkt nihkus üha enam arhitektuuri ja tootepere komponentide väljatöötamiselt uute kliendisüsteemide koostamisele. Lisandusid komponentide projektid, mis lisasid tooteperele komponente ja kliendiprojektide juhtimisgrupist arenesid kliendiprojektide grupid, mis olid vastutavad konkreetse kliendi süsteemi ehitamise eest.
4. SS2000 organisatsioon alates 1994
   Igast rakendusalast (merevägi ja õhukaitse) sai osakond oma juhiga. Igas osakonnas on turundusgrupp, analüütikute grupp, kliendiprojektide grupid ja süsteemianalüüsi grupp. Iga osakond juhindub oma töös turundusplaanist, tooteplaanist ja tehnilisest arhitektuurist. Komponente toodab arendusgrupp. Kõik kliendispetsiifilised muudatused tehakse osakonna juhtimisel kasutades arendusgrupi ressursse. Osakonna süsteemianalüüsi grupp on vastutav arhitektuuri eest. On tekkinud SS2000 tootepere kasutajate grupp, mis sisaldab SS2000 klientide esindajaid ja annab juhiseid tootepere arendamiseks. Arendusgruppi ressursse kasutatakse maatriksis osakondadega.
   Hiljuti loodi SS2000 baaskonfiguratsiooni projekt, mille eesmärgiks on koostada valmisintegreeritud süsteemi tuum koos testide ja dokumentatsiooniga, mis oleks uute kliendisüsteemide tuumaks.
   Uue tehnoloogia hindamist ja kasutuselevõttu juhib tehnilise juhtimise grupp.

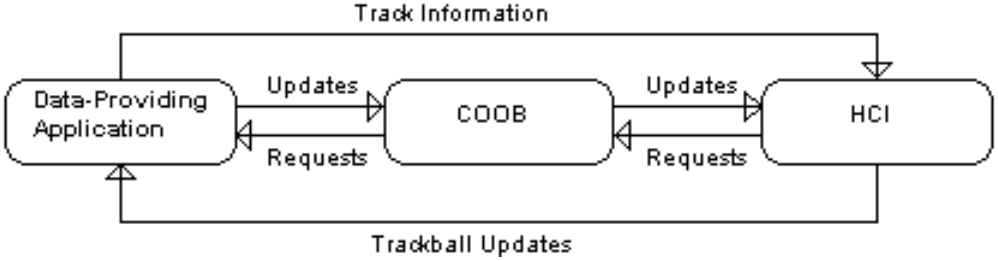


5. Tööjõud
   Arhitektuuri väljatöötamise algstaadiumis oli tööjõu hulk liiga suur -- see põhjustas segadust väljatöötajate hulgas.
   Hilisemas faasis vähenes disainerite vajadus ja suurenes integraatorite vajadus (keskmiselt 5 integraatorit kliendisüsteemi kohta).

Tootepere küpsemisel vähenes veelgi disainierite vajadus ja samuti integraatorite vajadus (1-2 integraatorit kliendisüsteemi kohta).

6. Oskused
   Arhitektuurirühma liikmed vajasid tugevat valdkonna ja klientide tundmist ja insenerioskusi. Väljendusoskus oli samuti nõutav. Väljatöötajad -- arendusrühma liikmed vajasid Ada, objekt-orienteeritud disaini ja tarkvara väljatöötamise keskkonna alaseid oskusi.
   Hilisemas faasis, kliendiprojektide lisandumisel muutis väljendusoskus üha tähtsamaks. Samuti ioli arhitektuurirühma liikmete jaoks tähtis oskus tasakaalustada uute nõuete rahuldamise ja arhitektuuri terviklikuse säilitamise vahel.
   Arhitektuurirühm peab säilitama ülevaate klientide arenevatest vajadustest ning uuest tehnoloogiast.
   Valmiskomponentide kasutamise faasis vajasid väljatöötajad rohkem rakendusvaldkonna alaseid teadmisi ja oskusi. Samuti suureneb vajadus tunda SS2000 tooteperet.
7. Juhtimisoskused
   Põhiliseks ülesandeks oli meeskonnale ärivajaduste ja soovitava tulevase oleku selgekstegemine. Juhtkond pidi olema orienteeritud lahenduste, mitte vigade leidmisele. Juhtkond pidi eksperimenteerimist julgustama.
   Suure hulga klientide puhul osutus vajalikuks oskus ühitada klientide nõudmisi ja seega muutusid eriti tähtsaks oskused pidada läbirääkimisi. Säilinud on ka nõue teada tooteperet ning selle arendusplaane.

2. Nõuded ja kvaliteedid
   Uute toodete effektiivseks loomiseks organisatsiooni hoidla (*repository*) baasil peavad tooted olema sarnase struktuuriga, nii et nad saaksid komponente jagada.
   Arhitektuuri ülesandeks on võimaldada nõuetele vastava süsteemi ehitamist. Tootepere arhitektuur peab aga võimaldama tereve rea erinevatele nõuetele vastavate süsteemide ehitamist. Seega peab olema võimalik asendada komponente ilma arhitektuuri rikkumata.
   1. Töökeskkond ja füüsiline arhitektuur
      Tüüpiline füüsiline arhitektuur koosneb liiasest lokaalvõrgust (topelt Ethernet) mis ühendab 30 - 70 erinevat protsessorit. Võrgu sõlm (mis võib sisaldada mitut protsessorit) võib olla meeskonna töökoht, relv, andur vms., mis kõik on haujutatud laeva erinevatesse paikadesse.
3. Arhitektuuriline lähenemine
   Kirjeldame arhitektuuri, kasutades mitmeid struktuure. Protsessistruktuuri kasutame näitamaks kuidas toimub hajutamine, moodulite struktuuri kasutame kihtide näitamiseks.
   1. Hajutamis ja tootepere nõudmiste rahuldamine
      Iga protsessor füüsilises arhitektuuris täidab hulka Ada programme. Iga Ada programm jookseb vaid ühel protsessoril ja võib koosneda mitmest Ada tööst (*task*).
      Hajutatud süsteemi puhul tekivad küsimused tupiku vältimisest, sideprotokollidest, tõrkekindlusest, võrgu haldamisest ja küllastumise (*saturation*) vältimisest.
      Tööde ja komponentide vahelised kokkulepped sisaldavad:
      - Side komponentide vahel toimub tugevalt tüpiseeritud teadete vahetamisega. Tugev tüpiseerimine aitab kompileerimisel selgitada terve rea vigu. Kuna teated on peamine liides komponentide vahel on võimalik komponente sõltumatult realiseerida ja neid vahetada.
      - Protsesside vaheline sideprotokoll on andmevahetusprotokolliks Ada rakenduste vahel tagades nii asukohast sõltumatuse. See lubab protsesse vabalt protsessoritele jagada.
      - Ada tööde-mehhanismi kasutatakse paralleelsuse realiseerimiseks (lõngamudelina).
      Andmete tekitaja on sõltumatu andmete kasutajast. Andmete haldamine ja uuendamine on andmete kasutamisest lahus. Disainierid kasutasid selle saavutamiseks tahvli-stiili (*blackboard style*). Põhiliseks andmete kasutajaks on kasutajaliides.



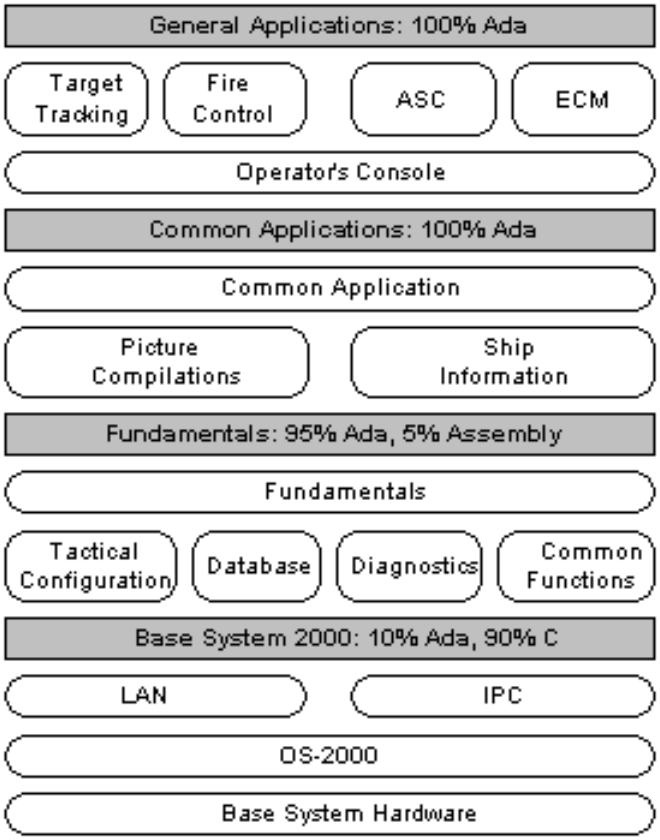
Kokkulepped andmete tekitamisel:
- Andmeid saadetakse vaid siis, kui need on muutunud. See väldib liigset võrgu koormust.
- Andmed esitatakse objekt-orienteeritud abstraktsioonidega et isoleerida programme realisatsioonide muutumisest. Tugev tüpiseerimine lubab kompileerimisel avastada andmete väärkasutamise vigu.
- Komponendid omavad nende poolt muudetavaid andmeid ja pakuvad päringuprotseduure mis toimivad monitoridena. See väldib sünkroniseerimisvigu kuna andmeid kasutab otseselt vaid andmeid omav komponent.
- Andmed on kasutatavad kõikidele huvitatud osapooltele hoolimata andmete tekitaja ja kasutaja paigutusest füüsilisel võrgul.
- Andmed on hajutatud nii et vastuse aeg oleks minimaalne.
- Pikema aja jooksul on andmed süsteemis kooskõlas. Lühikese aja kestel võivad ebakõlad eksisterida.

Võrguga seotud kokkulepped:
- Võrgu koormus püütakse disainida väikesena -- süsteem disainitakse nii, et vaid oluline informatsioon liigub võrgus.
- Andmekanalid on veakindlad. Rakendused püüavad vigu sisemiselt parandada.
- Rakendus võib mõnikord mõne andmete uuenduse "mööda lasta". Sellisel juhul pidevalt muutuvate väärtuste korral on võimalik puuduv väärtus interpoleerimisega taastada.

Lisakokkulepped:
- Laialdane Ada üldistatud konstruktsioonide (*generics*) kasutamine.
- Ada standartsete erandiprotokollide (*exception protocols*) kasutamine.

2. Moodulite struktuur

   Moodulite struktuur näitab, et SS2000 on kihiline:
   - Komponentide grupeerimine põhineb nende poolt kapseldatud informatsiooni tüübil. kõik komponendid, mis peavad muutuma, kui riistvara platvorm, kohtvõrk või sideprotokollid muutuvad moodustavad kihi. Komponendid, mis realiseerivad tootepere kõikidelel liikmetele ühist funktsionaalsust moodustavad samuti kihi. Lõpuks ka konkreetse kliendisüsteemi erilist funktsionaalsust realiseerivad komponendid moodustavad kihi.



- Kihid on järjestatud alates riistvarast sõltuvate kihtidega ja lõpetades rakendusest sõltuvate kihtidega.
- Järjestus kihitide vahel on range -- mingis kihis asuv komponent võib kasutada vaid teisi samas kihis olevaid või vahetult alumises kihis olevaid komponente.

Kõige alumse kihi nimeks on Base System 2000 ja see sidestab operatsioonisüsteemi, riistvara ja võrgu rakendusega. See kiht kujutab endast rakendusprogrammeerija jaoks virtuaalset masinat. Peamine põhimõte kihilise süsteemi juues on, et madalamad kihid on tõenäolisemalt ilma muudatusteta kasutatavad tootepere liikmete juures.

3. Süsteemi funktsioonid ja süsteemi funktsioonide rühmad
   SS2000 koosneb mitut tüüpi komponentidest. Funktsionaalsed nõudmised sisalduvad süsteemi funktsioonides. Süsteemi funktsioon on tarkvara hulk, mis realiseerib loogiliselt seotud nõudmisi ja koosneb reast Ada koodi ühikutest. Süsteemi funktsioonide rühm koosneb hulgast süsteemi funktsioonidest ja sellel põhineb arendusrühmale antud tööülesanne. SS2000 koosneb ligikaudu 30'st süsteemi funktsioonide rühmast, igas ligikaudu 20 süsteemi funktsiooni, mis on koondunud põhiliste funktsionaalsete alade ümber:
   - juhtimine ja side
   - relvade juhtimine
   - põhifunktsioonid (liides arvutussüsteemiga ja süsteemisisene side)
   - kasutajaliides

   Süsteemi funktsioonide rühmad võivad sisaldada rohkem kui ühte kihti kuuluvaid funktsioone ja neid töötab välja suurem arendajate rühm.
   Süsteemi funktsioonid ja süsteemi funktsioonide rühmad on testimise ja integreerimise aluseks -- see väldib tuhandete väikeste osade integreerimist ja testimist iga muutuse järel. Süsteemide koostamine suurtest testitud komponentidest on tähtis korduvkasutamise saavutamiseks.
4. Arhitektuur oli vundamendiks
   SS2000 arhitektuur võimaldas tootepere lähenemise ellu viia. Selle juures olid olulised abstraktsioonid ja kihiline ülesehitus. Abstraktsioonid lubasid välja töötada komponendid, mis kapseldasid oma liideste taha võimalikud muudatused/variatsioonid.

Et välja töötada tootepereret, mille komponendid:
- on sõltumatult väljatöötatavad
- on sobivad tooteperele, mille liikmed on väga erinevad
- lubavad lihtsalt tulevasi muutusi
on vajalik süsgav rakendusvaldkonna tundmine.
Seega on vajalik formaalne rakendusvaldkonna analüüs.

5. Komponentide säilitamine uute süsteemide loomisel
   CelsiusTech'i säilitatavaks tooteks pole mitte konkreetse kliendi süsteem või isegi mitte siiani loodud süsteemide hulk vaid tootepere ise. Tootepere säilitamine tähendab seda, et korduvalt kasutatavad komponendid on sellised, et suvalist tootepere liiget on võimalik uuesti genereerida (nad arenevad nõudmiste muutudes) ja uusi tootepere liikmeid on võimalik luua.
   Kokkuvõttes tähendab tootepere säilitamine võime säilitamist olemasolevast varast uusi tooteid luua. See tähendab üldiste komponentide hoidmist üldistena ja aja tasemel. Ühtegi toodet ei tohi lasta areneda isolatsioonis tooteperest. Iga komponent pole kasutuses igas süsteemis, võib olla vajadus ehitada väiksemaid tooteperesid üldise tootepere sisse, mõnda komponenti kasutatakse vaid üks kord aga isegi neid käsitletakse tootepere osana ja disanitakse paindlikuna ning konfigureeritavana juhuks kui tulevikus mõni uus toode peaks neid vajama.
   Väliselt ehitab CelsiusTech laevastiku süsteeme aga sisemiselt arendab ta üldisi varade baasi, mis tagab tema võime toota laevastikusüsteeme. See vahe võib tunduda tühisena aga see annab kõikidele tegevustele teise aspekti.
6. Suurte eelintegreeritud tükkide säilitamine
   Klassikalises tarkvara korduvkasutamise käsitluses on korduvalt kasutatav ühik tavaliselt väike komponent (Ada pakett, alamprogramm või objekt) või iseseisev alamsüsteem (tööriist või iseseisev toode). Esimesel juhul tuleb väikesed komponendid integreerida, konfigureerida ja testida, teisel juhul pole alamsüsteemid eriti konfigureeritavad ega paindlikud.
   CelsiusTech'i lähenemine on kesktee -- korduvalt kasutatav ühik on süsteemi funktsioon -- seotud funktsionaalsuse lõik mis läbib mitut süsteemi kihti. Süsteemi funktsioonid on eelintegreeritud -- nad on koostatud oma komponentidest, integreeritud ja koos testitud. Kui süsteemi funktsioon varade hoidlast välja registreeritakse on ta kohe valmis kasutamiseks. Seega kasutatakse korduvalt mitte üksnes komponente vaid ka integreerimis ja testimis tööd.
7. Parametriseeritavad komponendid
   Parameetrid on lihtsad, effektiivsed ja aja jooksul järgi proovitud vahendid komponentide korduvkasutatavaks tegemisel. Praktikas aga lisanduvad nad ohtlikult ruttu -- peaaegu igat komponenti on võimalik parametriseerides üldistada. SS2000 komponentidel on kokku 3000 - 5000 parameetrit, mida iga kliendi jaoks tuleb seada. Praegu puudub metodoloogiline lähenemine parameetrite vaheliste konfliktide vältimiseks. Praktiliselt kaotab parameetrite olemasolu mõned eeltestitud ja eelintegreeritud komponentide eelised sest parameetrite seadmisega on võimalik saavutada süsteem, mida pole kunagi testitud.
   Parameetrite tohutu hulk tõstatab vajaduse parameetrite vastasmõju(de) formaalsete mudelite järgi. Praktikas aga põhjustab selline risk soovimatust parameetreid muuta ja sageli kasutatakse komponente samasuguste parameetrite väärtustega , mis eelnevates kliendiprojektides.

4. Kokkuvõte
   CelsiusTech leidis, et effektiivse tootepere saavutamine ja säilitamine pole ainult õige tarkvara, süsteemiarhitektuuri, arenduskeskkonna, riistvara ja võrgu küsimus vaid ka organisatsiooni struktuur, juhtimispraktikad ja tööjõu struktuur muutusid tugevasti. Arhitektuur oli lähenemise aluseks nii tehniliselt kui ka kultuuriliselt. Arhitektuur muutus

asjaks, mille loomine oli kogu töö eesmärgiks. Arhitektuuri eest vastutas väike meeskond, mille tagajärjel arhitektuur oli mõisteliselt terviklik. Arhitektuuri definieerimine oli vaid pikaajalise arendustöö algus. Arhitektuuri rühma ülesandeks oli vältida arhitektuuri moondumist ja risustamist kliendiprojektides.

Arhitektuuriliste varade korduvkasutamine ühiskonnas

Peatükk vaatleb tellija perspektiivist süsteemide vastavust standarditele ja standardiseerimisprotsessi ennast.

Meteoroloogilise info kogumise ja analüüsi süsteem: Ostetud valmiskomponentide kasutamise näide

Peatükk kirjeldab süsteemi, mis ehitati kiirelt ja odavalt kasutades WWW ja CORBA tehnoloogiaid. Põhiliseks mureks süsteemi juurutamisel oli milline funktsionaalsus juurutatakse järgnevaks ja kuidas see väljendub kasutajale.

METOC Anchor Desk on võrgupõhine grupitöövahend informatsiooni kogumiseks ja otsuste toetamiseks Ühendriikide sõjaväele.

Kasutaja seisukohalt on METOC Anchor Desk'i eesmärk:

1. Toetada planeerijate ja meeskonnakomandöride keskkonnast sõltuvate otsuste tegemist
2. Koguda ja ühendada keskkonna alast informatsiooni
3. Interpreteerida andmeid komandöride jaoks

Planeerimimsülesannete lahendamisel on kasutada on järgnevad ressursid :

1. Tugi hajutatud kooperatiivsele planeerimisele
2. Hüpermeedia võrk keskkonna informatsioonist ja kriisihaldamise protseduuridest
3. METOC'i otsustusvahendid
4. Pärandtööriistad eelnevaist METOC'i süsteemidest

METOC Anchor Desk kasutas laialdaselt kolme uut tehnoloogiat:

1. WWW
2. CORBA
3. Grupivara

Evolutsiooniline arendusmeetod:

1. Nõuete ja süsteemi areng stsenaariumite kaudu
   Kohe kui arendajate ja kasutajate vaheline jagatud sõnastik on välja töötatud kirjeldasid kasutajad stsenaariume olemasoleva tehnoloogia seisukohalt ja hiljem peale prototüüpide kasutamist ka uue tehnoloogia seisukohalt. Süsteemi arenedes muutuvad stsenaariumid detailirikkamaks.
   METOC Anchor Desk süsteemi sihtide saavutamine:

Siht	Kuidas saavutati
Lühike aeg algselt demonstreeritavate võimalusteni	Laialdane valmiskomponentide kasutamine
Väljatöötajate kõrge tööviljakus	Vaid häälestuskoodi ja skriptide kirjutamine
Vahetatavad osad ja ühilduvus	Lõtv sidestus -- CORBA ja WWW/HTML kasutamine
Porditavus	Porditavate valmistoodete ja emulaatorite kasutamine

2. Pidev lõppkasutaja osalus ja pidev järelhäälestamine
   Suurte süsteemide väljatöötamisel on üheks probleemiks nõuete defineerimine süsteemi ehitamise alguses. Kuna süsteemi ehitamise aeg on pikk juhtub vältimatult, et kasutajate vajadused vahepeal muutuvad ja süsteemi juurutamisel selgub, et kasutajate arvatavad vajadused polnudki samad, mis tegelikud.
   Tihe tagasiside kasutajatelt prototüüpide kasutamisel võimaldab sellega järelhäälestada iga uut arendustsükklit.
3. Koostatavusele põhinev arhitektuur
   Objekt-tehnoloogia ja klient-server arhitektuurid pakuvad arhitektuurilist infrastruktuuri, millesse saab lülitada iseseisvalt väljatöötatud komponente.
4. Mõistlik ja hallatav dokumentatsioon
   Suurte süsteemide hinna määrab sageli dokumentatsioon. Dokumentatsiooni loomine on kallis ja samuti tema haldamine on kallis. Väike muudatus tarkvaras võib põhjustada suure hulga muudatusi nõuete-, disaini-, testimis-, haldamis- ja õppedokumentatsioonis. Evolutsiooniline lähenemine vähendab ka dokumentatsiooni muudatusi, kuna dokumentatsioon luuakse inkrementaalselt.
5. Ümbertegemine

Arhitektuuriline lahendus

Põhilised arhitektuurilised mõjutavad omadused tulenevad kliendi ja kasutaja sihtidest:

1. Lühike aeg algselt demonstreeritavate võimalusteni (kindel nõue)
2. Arendatavus
3. Kasutaja tööviljakus (kindel nõue)
4. Arendaja tööviljakus
5. Platvormide mitmesus
6. Osade vahetatavus (nii tarkvara kui ka riistvara)
7. Geograafiline hajutatus (kindel nõue)
8. Ühilduvus pärandtööriistadega, ja teiste samalaadsete süsteemidega (kindel nõue)

Samuti võib loetleda mitmeid arhitektuuri mitte mõjutavaid omadusi:

1. Effektiivsus
2. Töökindlus
3. Turvalisus

4. Kättesaadavus

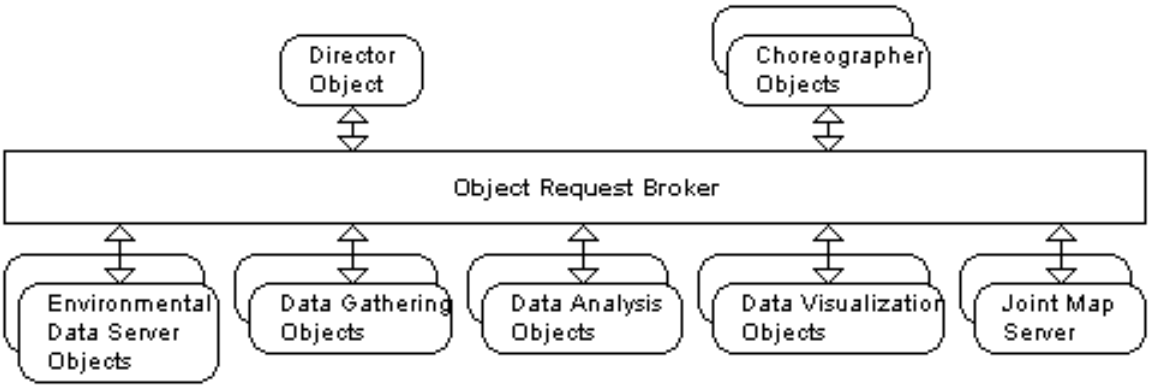
Need ei mõjutanud arhitektuuri mitte sellepärast, et nad oleksid vähetähtsad vaid sellepärast, et nad polnud väljatöötaja otsese kontrolli all. Muidugi valiti valmiskomponendid hoolikalt lähtudes ka nendest omadustest kuid need ei mõjutanud arhitektuuri.

Arhitektuuri komponendid:

1. Arvutid -- UNIX tööjaamad ja PC'd
2. Arvutivõrgud
3. Video-konverentsitarkvara
4. Emulaatorid
5. WWW lehitsejad
6. Grupivara
7. Utiliidid

Põhiline METOC Anchor Desk süsteemi disainiotsus oli mitte segada olemasolevate ja hästi töötavate pärandsüsteemide tegevust. Idee oli parandada mitte asendada.

METOC arhitektuur:



Kliendid on ülal ja serverid all

Director -- esitab lõppkasutajale otsustusabivahendite paleti (haldab tööriistade registrit).
Choreographer -- isoleerib Director'i konkreetsetest elementidest -- iga Choreographer omab teadmisi mingi tööriista teenedamiseks vajalike objektide hulga kohta.
Selline organisatsioon võimaldab töö ajal laiendatavust (uute tööriistade registreerimisega) ja teadmiste lokaliseerimine Choreographer'itesse vähendab süsteemi keerukust.
Uue tööriista lisamisel tuleb lisada uus Choreographer ja kui vaja uued teenused (analüüs või visualiseerimine).

Tarkvara arhitektuur tulevikus

Programmeerimise ajalugu võib vaadelda kui lisanduvaid vahendeid keeruka funktsionaalsuse esitamiseks. Kuuekümnendad olid alamprogrammide dekaad.
Seitsmekümnendatel algas mure struktuurse programmeerimise pärast, et saavutada enamaid kvaliteete kui korrektne funktsionaalsus. Andmevoo analüüs, olem-suhe diagrammid, informatsiooni peitmine ja muud tehnikad olid hulga arendusmeetodite baasiks mis kõik viisid mingitele kvaliteetidele vastavate alamprogrammide hulga loomiseni. Neid komponente nimetati tavaliselt mooduliteks. Ilmusid moodulitel põhinevad programmeerimiskeeled ja esimest korda osutus võimalikuks pakkida komponente nii et nende sisemist ehitust oli teoreetiliselt võimalik ignoreerida. Seitsmekümnendad olid moodulite dekaad.

Kaheksakümnendatel kristalliseerusid moodulitel põhinevad keeled, informatsiooni peitmine ja nendega seotud metodoloogiad objekti mõisteks.

Trend suuremate ja võimsamate abstraktsioonide poole jätkub. On olemas süsteemide generaatorid teatud valdkondadele ja räägitakse süsteemide süsteemidest mis rõhutavad süsteemide ühilduvust.

1. Arhitektuuri äritsükkel.
    - Üksik organisatsioon loob üksiku arhitektuuri ühe süsteemi jaoks
    - Üksik organisatsioon loob üksiku arhitektuuri tootepere jaoks
    - Mitu organisatsiooni loob standartse arhitektuuri ja viitemudeli suure hulga süsteemide jaoks
    - Arhitektuuri muutumine nii valitsevaks, et arendusorganisatsiooniks saab kogu maailm (näiteks WWW puhul)
2. Arhitektuur ja pärandsüsteemid.
   Olemasolevate süsteemidega toimetulek on suureks probleemiks. Neid on alates hästi disainituist ja hästi dokumenteerituist kuni halvasti disainituteni ja dokumenteerimatuteni välja. Algsed arhitektid ja väljatöötajad võivad olla organisatsioonist lahkunud. Seega on selliste süsteemide arhitektuuri (taas)avastamise tehnikad ja vahendid esimeseks selgelt väljenduvaks uurimisteemaks.
   Pärandsüsteemide arhitektuurilised probleemid on seotud muudatuste tegemisega süsteemidesse, millesse organisatsioon on palju insvesteerinud. Selle probleemi alamosadeks on pärandsüsteemi arhitektuuri kindlaks tegemine, muudetud arhitektuuri sihtide määramine ja süsteemi uuele arhitektuurile migreerimise strateegia väljatöötamine. Selline migreerimine võib toimuda osade ümberehitamisega, olemasolevate osade sissepakkimisega ja vana süsteemi uue süsteemiga asendamisega.
    - Arhitektuuri arheoloogia ja arhitektuurile vastavus
      Arhitektuurist pole kasu kui ei saa hinnata süsteemi vastavust arhitektuurile. Kuidas teha kindlaks, et süsteemi arendamise käigus ei murendata tema arhitektuuri.
        - Paljudel süsteemidel pole üldse dokumenteeritud arhitektuuri (kõikidel süsteemidel on arhitektuur aga sageli pole see ilmutatud kujul ja arendajate poolt teadvustatud vaid areneb juhuslikult)
        - Süsteemides, millel on teatav hulk arhitektuuri dokumentatsiooni, on arhitektuur esitatud sellisel viisil, et suhe dokumentatsiooni ja tegeliku süsteemi vahel on ebaselge
        - Süsteemides, millel on korralikult dokumenteeritud arhitektuur on arhitektuuri kirjelduse areng süsteemi tegeliku arhitektuuri arengust maas kuna süsteemi arendusele eraldatakse suurem tööjõud kui süsteemi kirjelduse arendusele
    - Arhitektuuri muutmine
      Kui olemasoleva süsteemi arhitektuur on väljaselgitatud, tuleb otsustada kas ja kuidas arhitektuur peab arenema: milliseid uusi tehnoloogiaid tuleb kaasata, millis(t)ele uu(t)ele stiili(de)le arhitektuur peab vastama, milliseid uusi malle tuleb kasutada ja kui regulaarne arhitektuur peab olema.
      Uute tehnoloogiate kaasamisel peab arhitekt arvestama valmistarkvara saadavust, standardite olemasolu ja uute tehnoloogiate levikut ning stabiilsust. Kui arhitektuuri keerukust kogu aeg ei hallata ja taltsutata muutub see takistuseks tulevasele arengule.
      Uute stiilide ja mallide kasutuselevõtul peab arhitekt olema teadlik olemasolevatest stiilidest ja mallidest, nende headest ja halbadest külgedest ja nende vastasmõjust. Arhitektuuri keerukust saab vähendada muutes seda regulaarsemaks. Et arhitektuuri regulariseerida peaks arhitektill olema vahendid mis aitavad leida struktuurseid sarnasusi arhitektuuris.

- Arhitektuuri migreerimise tehnoloogia
  Lähtudes muudatustest ja antud arhitektuurist on olemas muudatusi, mida antud arhitektuur toetab ja muudatusi, mis on antud arhitektuuriga võimatud. Viimaste jaoks on vaja tehnoloogiat, mis võimaldab organiseeritult migreerida süsteemi ühelt arhitektuurilt teisele. Selline muudatus võib nõuda ka väljatöötava või toetava organisatsiooni muutmist.

3. Arhitektuuri saavutamine
   Kõik näited raamatus on keskendunud mingi kvaliteediatribuudi saavutamisele. Konkreetsed kvaliteedid olid erinevates näidetes erinevad aga iga süsteemi korral tuleb alustada soovitud kvaliteetide väljaselgitamisest.
   Arhitektuuri saamine korratavalt ja kindlalt täpsetest kvaliteedinõuetest on lahtine uurimisala. Antud probleemil on mitmed osad:
   - Kvaliteetide mõõdetavad ja sisukad definitsioonid
   - Arhitektuuri loomine või valik vastavalt funktsionaalsetele ja kvaliteedinõuetele
   - Arhitektuuriotsuste juures tehtavate kompromisside mõistmine ja mõõtmine

   - Kvaliteedi atribuutide mõistmine
     Kvaliteedi mõiste tarkvara juures on segane. Algselt võiksime seda defineerida kui kasutuskõlblikust aga selle juures tekib probleeme mitmete atribuutidega. Kvaliteedi atribuutide sisu mõistmine on eriti terav arenduskvaliteetide puhul.
     Põhiliseks probleemiks arenduskvaliteetide sisu mõistmisel on sobivate mudelite puudumine. Effektiivsus on kvaliteet, mille jaoks on väljakujunenud mudel(id) olemas ja mille suhtes arhitektuuri varajane hindamine on seega võimalik. Effektiivsust saab analüüsida ja mõõta ressursside kättesaadavuse ja ressursside kulu terminites toetudes arvutussüsteemi mudelile esitatuna hulga ressurssidega ja ühendustega nende ressursside vahel.
     Arenduskvaliteetide jaoks sobivate mudelite küsimus on veel lahtine. Staatilised mudelid nagu COCOMO (*constructive cost modeling*) või funktsioonipunkti meetod püüavad hinnata väljatöötamiseks vajaliku aega aga nad toetuvad väliste nähtuste mõõtmisele mitte arendusprotsessi olemuse mõistmisele ja modelleerimisele.
     Kui kvaliteete saab siduda abstraktsete mudelitega, võib süsteeme analüüsida ja nende kavaliteedi atribuutide väärtusi välja selgitada. Sellised tehnikad nagu SAAM põhinevad arhitektuuri hindamisele tema sobivusele vastavalt teatud punktidele kvaliteediruumis ja eeldusele, et arhitektuur on sobiv ka nende punktide ümbruses. Sellised punktid on esitatud stsenaariumitega.
     Praegu on SEI tähelepanu koondunud kvaliteedi atribuutide abstraktsete mudelite ja nende tugipunktide (*fulcrum points*) defineerimisele. Tugipunktid on süsteemi mõõdetavad omadused mis mõjutavad kahte või enamat kvaliteedi atribuuti. Näiteks sõnumiviite aja variatsioonid süsteemi sidekanalites mõjutavad süsteemi effektiivsust, turvalisust (mida väiksemad on sõnumiviite variatsioonid seda vähem aega on sissetungijal muuta sõnumit ilma et seda avastataks) ja töökindlust (väga suured sõnumiviite variatsioonid raskendavad tõrgete avastamist). Seega on sõnumiviite variatsioon nende kvaliteedi atribuutide tugipunktiks.
   - Mallide mõju tarkvara arhitektuurile
     Võime analüüsida ja ehitada süsteeme kasutades korrastatud ehitusplokke aitab inimesi keerukate süsteemide mõistmisel ja seega toetab nii arendustööd kui ka haldamist. Korrastatud ehitusplokkide hulk esitab kasutusmalle. Mallid aitavad mõista keerukaid süsteeme suuremate kontseptuaalsete tükkidena vähendades tunnetusliku koormust.
     Mallide kasutamise alane uuriistöö edeneb kahel lsuunal: kirjeldamine ja toetavad tööristad. Kirjeldamise alal toimub aktiivne tegevus objekt-orienteeritud disainimallide kogukonnas (*community*). Tööriistade osas tegutsetakse interaktiivse

arhitektuurimallide äratundmise alal ja mallide teekide kasutamise toetamisel tarkvarasüsteemi komponeerimisel. Esimesi tööriistu võib kasutada ka süsteemide diagnostikaks -- mõõtes arhitektuuri kaetust mallidega (palju malle on kasutuses ja kui suur osa arhitektuurist jääb mallidega katmata). Need mõõdud sobivad süsteemi arhitektuurilise keerukuse määramiseks.

- Arhitektuuri loomine või valik
  Tehnoloogiad, mis toetavad arhitektuuri loomist või valikut moodustavad järgneva rea lihtsamalt/madalamalt keerukamale/kõrgemale:
  - vaba/juhuslik (*ad hoc*)
  - stiilide komplekt
  - korduvalt kasutatavad arhitektuurid
  - korduvalt kasutatavad arhitektuurid koos komponentide teekidega
  - korduvalt kasutatavad arhitektuurid koos parametriseeritavate komponentidega
  - süsteemiraamide generaatorid
  - rakendusegeneraatorid

  Alustades vabadest/juhuslikest tehnikatest, kus kogenud ja andekad disainerid loovad kordumatu arhitektuuri, jõuame arhitektuuride korduvkasutamise, arhitektuuri viitemudeliteni ja nende kasutamist toetava tarkvarani. Rakendusgeneraatorid on programmid, mis sisaldavad teadmisi konkreetse rakendusvaldkonna kohta ja genereerivad tarkvara vastavalt antud nõudmistele. Puhas rakendusgeneraator loob kasutusvalmis süsteemi. Süsteemiraamide või -osade generaatorite poolt loodud komponendid tuleb valmissüsteemi integreerida (näiteks YACC'iga loodud süntaksi analüsaator).
  Põhilised katmata uurimisalad arhitektuuride loomise ja valiku juures on arhitektuuride kvantitatiivne hindamine (praegu kasutatavate kvalitatiivsete tehnikate asemel) ja disaini retseptid.
  Kui on olemas kvaliteedi atribuutide mudelid, võib lisada arhitektuuristiilidele nende oodatud käitumise kirjelduse antud kvaliteedi atribuutide suhtes. Me võime rääkida effektiivsusele orienteeritud stiilidest (nagu prioriteedil põhinev sunnitud jaotamine (*preemptive scheduling*)) või muudetavusele orienteeritud stiilidest (nagu kihilised arhitektuurid) või töökindlusele orienteeritud stiilidest (nagu liiasus) ja siis vaadelda kuidas neid stiile saaks kombineerida. Selliselt saaks algse arhitektuuri uuele süsteemile genereerida atribuutidele orienteeritud stiilidest. Et seda teha tuleb kogude teadmisi korduvkasutatavate arhitektuuride ja eriti stiiliperede osas.
  Keskseks selles töös on mõista nõuete ja arhitektuuri vahelist seost -- kuidas süsteem sattub teatud stiiliperesse, ja mis rolli mängivad kvaliteedinõuded, effektiivsuse nõuded, organisatsiooni ajalugu või kitsendused.
  Eesmärgiks on kujundada välja disaini käsiraamatud nagu teistes inseneriteadustes. Järgnevaid tulemusi võib oodata näidete analüüsidest -- kujunevad välja järgnevad ühtsed liigitused:
  - Süsteemide probleemiruumide liigitus (*taxonomy*)
    Kas on tegu reaalaja süsteemiga, kas süsteem on hajussüsteem, kui tihedalt võib esineda tõrkeid jms. iseloomustavad probleemiruume.
  - Süsteemide kontekstiruumide liigitus
    Milline on organisatsiooni mõju arhitektuurile, milline on väljatöötajate eelnev kogemus, millises suunas areneb turg, millised tehnoloogiad "võidavad" jms.
  - Süsteemide lahendiruumide liigitus
    Tööd arhitektuuri stiilide alal ja disainimallide leidmine, sõnastamine, formaliseerimine ning kasutamine
- Rakendusgeneraatorite tehnoloogia
  Et luua puhast rakendusgeneraatorit mingi rakendusvaldkonna jaoks peab looma primitiivsete kombineeritavate komponentide sõnastiku. Põhilisteks probleemideks

on komponentide leidmine, komponeerimine ja antud füüsilisele arhitektuurile kujutamine.
Komponentide leidmine on täna juhuslik (*ad hoc*), selle süstematiseerimine rakendusvaldkonna analüüsi meetodite liitmisega arhitektuuri komponentide leidmisega peaks võimaldama generaatorite ehitamist uutele rakendusvaldkondadele. Näiteks analüsaatorite generaatorite, optimiseerijate generaatorite jms. olemasolu muudab tänapäeval mõeldamatuks kompilaatorite käsitsi ehitamise. Samuti võib tulevikus juhtuda näiteks andmebaasijuhtimissüsteemidega, tarkvara arenduskeskkondadega või muu tarkvaraga, kuna on olemas generaatorid mis loovad rakenduse-standardseid komponente, mis on ühendatavad rakenduse-standartsetesse arhitektuuriraamidesse. Areng generaatorite alal võib viia generaatorite generaatoriteni, graafilisi spetsifitseerimiskeeli lubavate kasutajaliidesteni ja soovitud tulemuse deklaratiivse spetsifitseerimiseni.

4. Arhitektuurist süsteemini
   Kui arhitektuur on spetsifitseeritud on veel vaja kontrollitud ja dokumenteeritud viisi arhitektuuri väljendamiseks ja süsteemi arhitektuurile vastavuse tagamiseks. Süsteemi arhitektuur teenib mitmeid huvitatuid ja peab olema väljendatav neile. Näiteks saab arhitektuur aluseks tööde jaotusele, testimise plaanile, projekti ajakavale jms. Arhitektuur on aluseks meeskondade vahelisele koostööle, süsteemi omaduste modelleerimisele, ennustamisele ja hindamisele. Arhitektuuri esitamisel huvitatule peab esitama üheselt ja arusaadavalt antud huvitatule vajaliku informatsiooni.
   - Infrastruktuurid, mis toetavad arhitektuuri kirjeldamise/defineerimise keelte (ADL) arendamist
     Suurem osa arhitektuuri kirjeldamise keeli jagab ühiseid mõisteid. Arhitektuuri kirjeldamise keeli toetava keskkonna ehitamine eeldab ühise probleemidehulga lahendamist.
   - ADL informatsiooni ühitamine teiste elutsükli toodetega
     - millised on vajalikud testid?
     - millist täidetavat koodi saab automaatselt genereerida?
     - kuidas tagada arhitektuuri ilmutatud seost nõuetega?
     - kuidas lisada arhitektuuri malle?
     Arhitektuuri kirjeldus tuleks täielikult integreerida tarkvara väljatöötamiskeskkonda. Tulemuseks võiks olla ekspertsüsteemi taoline tööriist, mis lubab arhitektuure visandada ja valideerida kujutades neid nõudmistele, uurida nende mõju analüüsi ning kiire prototüüpimise kaudu ja genereerida arendustööks vajalikud projekti infrastruktuurid.
   - Praktilised verifitseerimise strateegiad
     On olemas rida keskkondi nagu ObjectTime, kus arhitektuuri kasutatakse süsteemi simulatsiooni genereerimiseks. Simulatsioon nagu ka testimine võib vaid näidata vigade olemasolu mitte nende puudumist. Verifitseerijad ja teoreemi tõestajad on võimsamad -- antud arhitektuuri (komponendid, ühendused, komponentide funktsioonide kirjeldused, töökindluse ja effektiivsuse nõuded ja ühenduste semantika) alusel võib verifitseerija kinnitada, et effektiivsuse, töökindluse ja muud nõuded on saavutatavad või näidata ära kohad arhitektuuris, mis takistavad nende saavutamist.
5. Kokkuvõte
   Lisaks võimsamatele abstraktsioonidele ja keerukamatele komponentidele on kaks tulevast arengut, mis mõlemad väljendavad arhitektuuride standardiseerimist -- üks ettevõtte sisest ja teine kogukonna laiust:
   - Süveneb suund ettevõttesiseste tooteperede tekkimisele, paindlikus on tarkvara edu võtmeks, toode peab vastama ühe kliendi või väikese klientide rühma vajadustele ja

sellise paindlikuse saavutamise aluseks on ettevõtte standartne arhitektuur.
- Arhitektuuri alastest uurimustest tekib väike (alla 50) arhitektuuristiilide kogu, mida kasutatakse järgmise põlvkonna süsteemide ehitamisel. Disainierid valivad nende standartsete arhitektuuride hulgast ja seda toetavad arenenud tööriistad. Uusi arhitektuure tekib juurde arvutite uute kasutusvõimaluste avastamisega aga see on suhteliselt harv.

P. Kruchten, **The 4+1 View Model of Architecture**, IEEE Software 12(6), 42-50, 1995.

The 4+1 View Model organizes a description of a software architecture using five concurrent views, each of which addresses a specific set of concerns. Architects capture their design decisions in four views and use the fifth view to illustrate and validate them.

Four views:

- The *logical view* describes the design's object model when an OO design method is used (in Rational simplified Booch notation is used). To design an application that is very data-driven, you can use an alternative approach to develop some other form of logical view, such as an entity-relationship diagram.
  - supports the functional requirements (key abstractions (*objects* or *object classes*) are taken mainly from the problem domain)
  - a *class diagram* shows a set of classes and their logical relationships (association, usage, composition, inheritance, ...)
  - sets of related classes are grouped into *class categories*
  -
- The *process view* describes the design's concurrency and synchronization aspects (in Rational Booch's notation for Ada is used).
  - takes into account some nonfunctional requirements -- performance and system availability
  - addresses concurrency and distribution, system integrity, and fault-tolerance
  - specifies which thread of control executes each operation of each class identified in the logical view
  - highest level of abstraction is a set of independently executing logical networks of communicating programs ("processes") that are distributed across a set of hardware resources
  - a *process* is a group of tasks that form an executable unit
  - to develop the process view, designers partition the software into a set of independent *tasks*
  - process view can be used to estimate message flow and process loads
  -
- The *physical view* describes the mapping of the software onto the hardware and reflects its distributed aspect.
  - takes into account system's nonfunctional requirements such as system availability, reliability (fault-tolerance), performance(throughput), and scalability
  - various elements identified in the logical, process, and development views (networks, processes, tasks, and objects) must be mapped onto various nodes of network of computers
  - several different physical configurations will be used (for development, for testing, for deployment, ...) -- mapping of the software to the nodes must be highly flexible
  -

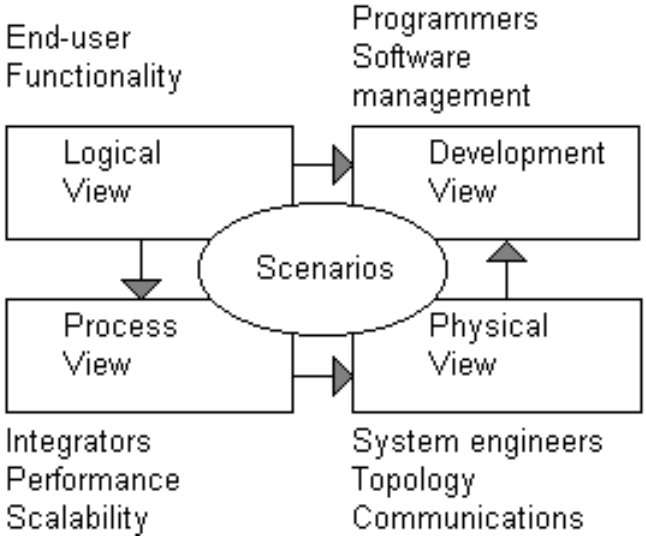
- The *development view* describes the software's static organization in its development environment.
  - focuses on the organization of the actual software modules in the software-development environment
  - the software is packaged in program libraries or *subsystems* that are organized in a hierarchy of layers, each layer providing a narrow and well-defined interface to the layers above it
  - development view is represented by module and subsystem diagrams that show the system's export and import relationships
  - subsystem can only depend on subsystems in the same or lower layers
  -

Software designers can organize the description of their architectural decisions around these four views, and then illustrate them with a few selected use cases, or *scenarios*, which constitute a fifth view.

Small subset of important scenarios (instances of use cases) is used to show that the elements of the four views work together seamlessly. For each corresponding *scripts* (sequence of interactions between objects and between processes) are described. This view is redundant with the others, but it:

- acts as a driver to help designers discover architectural elements during the architecture design
- validates and illustrates the architecture design, both on paper and as the starting point for the tests of an architectural prototype

The architecture is partially evolved from these scenarios.



At Rational, Perry and Wolf's formula [D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture", *ACM Software Eng. Notes*, Oct. 1992, pp. 40-52]:

Software = { Elements, Forms, Rationale/Constraints}

is applied independently on each view.

For each view a set of elemnts is defined (components, containers, and connectors). The architects can also pick a certain *architectural style* for each view. For the logical view Rational uses OO style. For process view styles from Garlan and Shaw's taxonomy (pipes and filters, client/server, ...) are used. For development view four to six layers of subsystemsis recommended (e.g. basic elements/virtual machine/domain framework/domain components/customer specific components).

Several important characteristics of the logical view:

- *Autonomy* identifies whether objects are *active* (invoke other objects' or its own operations, has full control over other objects invoking its operations), *passive* (never invokes any operations, has no control over other objects invoking its operations), or *protected* (never invokes any operations but arbitrates the invokation of its own operations).
- *Persistence* identifies whether objects are *transient* or *permanent*.
- *Subordination* determines if the existence or persistence of an object depends upon another object
- *Distribution* determines if the object's state or operations are accessible from many nodes in the physical view.

Logical view could consider each object as active and potentially concurrent -- logical view takes into account only the requirements' functional aspects. In process view it is not practical to implement each object with its own thread of control.

Threads of control are needed to:

- react rapidly to certain classes of external stimuli (inc. time related events)
- take advantage of multiple CPUs in a node or multiple nodes in a distributed system
- increase CPU utilization by allocating CPUs to other activities when a thread of control is suspended during another activity
- prioritize activities (and thus potentially improve responsiveness)
- support system scalability (by having additional processes sharing the load)
- separate concerns between different areas of the software
- achieve higher system availability (with backup processes)

Determining concurrency:

- *Inside out*. Starting from the logical view, agent tasks are defined that multiplex a single thread of control across multiple active objects of a given class. Subordinate objects are executed on the same agent as their parent.
- *Outside in*. Starting with the physical view, external stimuli (requests) to the system are identified and client processes to handle them are defined.

In development view a class is usually implemented as a module and large classes are decomposed into multiple packages; class categories are grouped into subsystems. Additional constraints are:

- team organization
- expected magnitude of code (typically 5000 to 20000 lines per subsystem)
- degree of expected reuse and commonality

- strict layering principles (visibility)
- release policy
- configuration management

An iterative process is advocated, in which the architecture is actually prototyped, tested, measured, and analyzed, and then refined in subsequent iterations (this mitigates the risks associated with the architecture, helps to build teams and improves training, architecture familiarity, tool acquisition, ...).

1. To begin, select a few scenarios on the basis of risk and criticality. Then create strawman architecture and script the scenarios, identifying major abstractions (classes, mechanisms, processes, subsystems) and decomposing them into sequences of pairs (object, operation).
2. Organize the architectural elements into the four views, implement the architecture, test it, and measure it.
3. Reassess the risks, extend the scenarios, and select few additional scenarios on the basis of risk or extending architecture coverage. Try to script those in the preliminary architecture and discover additional architectural elements or significant architectural changes. Update four views and revise the existing scenarios on the basis of these changes.
4. Upgrade the implementation (architectural prototype) to support the new extended set of scenarios.
5. Test architecture under load (in target environment, if possible) and review all five views.
6. Iterate.

Kirjandus

1. M. Shaw, D. Garlan, **Software Architecture**: Perspectives on an Emerging Discipline, 1996, pp. 242.
2. M. Shaw, P. Clements, **A Field Guide to Boxology**: Preliminary Classification of Architectural Styles for Software Systems, 1996, pp. 15.
3. L. Bass, P. Clements, R. Kazman, **Software Architecture in Practice**, 1998, pp. 452.
4. D. F. D'Souza, A. C. Wills, **Objects, Components, and Frameworks with UML**: The Catalysis Approach, 1999, pp. 785.
5. P. Kruchten, **The 4+1 View Model of Architecture**, IEEE Software 12(6), 42-50, 1995.