TARTU ÜLIKOOL

MATEMAATIKA-INFORMAATIKATEADUSKOND

Arvutiteaduse instituut

Infotehnoloogia eriala

Taavi Kotka

Spetsifitseerimine vs prototüüpimine: lahendus kollase kassi probleemile

Bakalaureusetöö (6 EAP)

Juhendaja: Vladimir Šor

Autor…….“…..“ aprill 2013

Juhendaja…….“…..“ aprill 2013

Lubada kaitsmisele:

Professor …… …….“…..“ aprill 2013

Tartu 2013

Sisukord

1. Sissejuhatus

Aastal 2000 loodi Tartus tudengiettevõtte Webmedia (tänane Nortal), mille eesmärgiks pärast esimeste kuude eneseotsinguid sai kasvada suurimaks tarkvaraarenduse rätseplahendusi (*custom made software*) pakkuvaks ettevõtteks Eestis. Eesmärk sai täidetud 2004. Aastal ning ka Äripäev IT TOP tunnustas Webmediat I kohaga. Aastaks 2012 oli toonane Webmedia, tänane Nortal, kasvanud suurimaks IT teenuseid pakkuvaks ning eksportivaks ettevõtteks Baltikumis (Joonis 1) .

Joonis 1. Baltikumi suurimad IT teenusettevõtted [Inve]

Prime Baltic IT services companies in 2012 H1, Euro '000

Country	Company	IT services revenue			% of total	Total revenue		
		2012H1	2011H1	Growth, %	2012H1	2012H1	2011H1	Growth, %
EE	Nortal (fmr.Webmedia Group)	21.317	12.204	75%	98%	21.731	12.336	76%
LV	Exigen Services	9.530	8.712	9%	94%	10.186	9.116	12%
LT	Baltic Data Center	7.675	6.982	10%	72%	10.651	7.543	41%
LV	Lattelecom Technology	7.416	5.054	47%	94%	7.893	5.686	39%
EE	Santa Monica Networks Group	7.282	5.225	39%	27%	26.963	25.155	7%
LT	Alna Group	6.607	6.486	2%	80%	8.234	10.330	-20,3%

Kümne aastaga regiooni suurima käibega IT teenuste ettevõtteks kasvamine baseerus mitmel eduteguril, alates nooruslikust entusiasmist ja agressiivsetest müügitaktikatest kuni õnnefaktorini välja. Kuid õnnest olulisema panuse andis kindlasti tolle aja kohta innovaatiline lähenemine tarkvara arendamise protsessile.

See vääriks omaette raamatut, et kirjeldada, kuidas noored vähese kogemusega tudengid tarkvara tootmise protsessile lähenesid, mismoodi kogu seda lähenemist aastate jooksul edasi arendati, milles läbi kukuti ning milliseid toetavaid töövahendeid ning metoodikaid loodi. Bakalaureusetöö mahupiire arvestades on antud töös keskendutud ainult ühele suuremale arendusprotsessi muutusele, milleks oli **kohustuslik kasutajaliidese prototüüpimine** Tellija vajaduste selgitamise ja loodava tarkvara analüüsi etappides.

Peamised küsimused, millele käesolev töö vastuseid pakub:

- Milliseid probleeme tarkvara komponentide visuaalne prototüüpimine aitab lahendada ning miks peab see samm olema analüüsiprotsessis kohustuslik?
- Millised on erinevad visuaalse prototüüpimise viisid ning miks on vaja jõuda kokkuvõttes täisfunktsionaalse prototüübi mootori ehitamiseni?
- Millised on visuaalse prototüüpimisega kaasnevad muud positiivsed mõjud tarkvara arendamise protsessile?

Olulise kitsendusena tuleb märkida, et Webmedia keskendus 90% ulatuses oma toodangus töömahukate (vähemalt 40 kasutuslooga) veebirakenduste realiseerimisele, nagu suurfirmade iseteenindused ja tagatoa (*back-office*) töövahendid, kus lõppkasutaja kasutajaliides oli projektide õnnestumise seisukohalt ülikriitilise tähtsusega.

Samuti tasub lugemisel silmas pidada, et *software mockuping, wireframing, stroyboarding* ja muud tänapäevase tarkvaraarenduse analüüsiga seotud metoodikad olid 12 aastat tagasi veel tundmatud ning on ilmunud koos agiilse tarkvaraarendusega massilisse kasutusse alles viimase 5-6 aasta jooksul.

Bakalaureusetöö autor oli perioodil 2000-2005 Webmedia arendusjuht (2006-2011 juhatuse esimees samas ettevõttes) ning vastutav arendusprotsessi ning eelkõige analüüsiprotsessi parendamise eest. Töös käsitletu refereerib tema otsest kogemust kohustusliku kasutajaliidese prototüüpimise kui metoodika väljatöötamisel ning integreerimisel üldisesse tarkvaraarenduse protsessi.

Ka veel täna kehtiva Nortali arendusprotsessi põhiideoloogiani jõudmine ei olnud ühe mehe panus, vaid meeskonnatöö, mistõttu eraldi tänusõnad pikaaegse kaasamõtlemise ja koostöö eest uuenduste elluviimisel kuuluvad Urmo Pärgile, Karel Kravikule, Imre Lumistele, Pirjo Vanemale, Andrus Tamboomile, Ivo Mägile, Vladimir Šorile, Ago Meistrile, Alar Antonsile, Lauri Heinale, Allan Kikkasele ning paljudele teistele Webmedia (Nortali) toonastele ja praegustele töötajatele.

2. Kuidas arendada tarkvara efektiivselt?

2.1.Arendusefektiivsus on kõige alus

Sarnaselt iga teise tootmisprotsessiga jaguneb ka spetsiaaltarkvara (*Custom made software*) tootmisprotsessis kaheks suureks maailmaks: tellija maailm ja täitja maailm. Ei ole vahet, kas ehitatakse maja, treitakse detaili või küpsetatakse leiba. Ikka on üks pool, kes tellib/tarbib ning maksab toote eest, ning teine pool, kes tellimuse täidab.

Valdkonnaspetsiifiliselt detailides need maailmad küll erinevad, kuid suurtes piirides on motivatsioonid samad kõikides teemades:

tellija motivatsioon – saada püstitatud eesmärkidele/ülesandele vastav parim võimalik tulemus kokkulepitud vea, aja ning eelarve piirides[1].

täitja motivatsioon – tarnida/toota kliendile maksimaalse tööefektiivsusega (võimalikult vähe Tellija poolt mittetasustatud töötunde) soovitud tulem ning teenida võimalikult suurt kasumit.

Ehk siis, kui tellija tahab poest värsket ning maitsvat leiba, teeb pagariettevõte kõik selleks, et tootmisprotsessist maksimaalne välja pigistada alates toorainehangedest kuni logistikani, et konkurentsis püsida ja mõistlikku kasumit teenida.

Teenida maksimaalselt suurt kasumit on kindlasti ka spetsiaaltarkvara tootjate üheks peamiseks eesmärgiks. Selliste ettevõtete puhul peetakse heaks näitajaks, kui kasumlikkus on rohkem kui 10 protsenti. Väiksematel kuni 100 inimesega kollektiividel võib see näitaja olla ka üle 20 või isegi üle 30 protsendi, kuid suuremates ettevõtetes peetakse stabiilset (st aastast aastasse korduvat) üle 10-protsendilist kasumimarginaali juba suurepäraseks näitajaks[2] (Joonis 2).

Joonis 2 Äripäeva IT TOP 2011 parimad spetsiaaltarkvaraga tegelevad ettevõtted [ÄP11]

	Käive 2010	Ärikasum 2010	Kasumimarginaal
Proekspert	5 975 747	1 408 454	23,6%
Icefire	2 756 691	852 980	30,9%
AS WebMedia	10 375 788	1 689 504	16,3%
Mindware	992 089	116 435	11,7%
Cybernetica	4 468 456	485 216	10,9%
Logica Eesti	3 089 425	216 788	7,0%

Analüüsides Eesti spetsiaaltarkvara tootvate ettevõtete majandusaasta aruannetes olevaid finantsnäitajaid [ÄP13], on kasumlikkus või õigemini selle väiksus absoluutnumbrina Eesti

[1] Kaasaegsed projektijuhtimismudelid ei pea mõistlikuks aja ja eelarve parameetri ranget jälgimist, kuna eesmärgi saavutamise nimel peavad olema need suurused paindlikult muudetavad, kuid antud töös on lähtutud, et enamikes tootmisprojektides ikkagi on oma kriitiliste vigade hulga piirang ning aja ja eelarve piirang

[2] Siinkohal ei loeta spetsiaaltarkvara tootvateks ettevõteteks arenduskeskusi, näiteks Playtech Eesti või Skype Eesti, kuna nad ei pea muretsema kõikide töötajate pideva väljamüügi eest ning kasumlikkus sõltub piltlikult emafirmale kirjutatud arve(te)st.

tarkvaraettevõtetel tõsiseks probleemiks. Kui piisavalt kasumimassi ei ole, puuduvad ka vahendid uutesse toodetesse/teadmistesse investeerimiseks või laienemiseks[3].

Tekib küsimus, miks siis ikkagi on täitjate kasuminäitajad niivõrd madalad? Ettevõtete juhtide arvates on põhjused erinevad, alates müügiprotsessi faasis tehtavatest vigadest ning oskamatusest oma teadmiste eest õiget tunnihinda küsida Tellija „rumaluseni“ välja. Kuid läbiva joonena tuuakse esile madalat arendusefektiivsust.

$$\textbf{Arenduse efektiivsus} = \frac{\text{Kasulike tundide arv (Tellija poolt tasutud tunnid)}}{\text{Tundide arv kokku (kõik projektile kulutatud tunnid)}}$$

Kui keskmiselt on kalendrikuus spetsialisti kohta 160 töötundi ja Tellija hindab neist kasulikuks (tasumist väärivaks) 90, siis on arendusefektiivsus 90/160 = 0,56 ehk 56%. See näites toodud 56% tundub tagasihoidlik number, kuid näiteks Webmedial läks aega rohkem kui 7 aastat, enne kui keskmine arendusefektiivsuse näitaja kõikide projektide peale kokku ületas 60 protsenti.

Spetsiaaltarkvara tootvad ettevõtted saavad oma käibe ja kasumi tulemusi parendada erinevate võtete kaudu. Alates sellest, et projektile kirjutatakse õhku juurde, sunnitakse töötajaid ületunde tegema, küsitakse inimressursi tasemest olulisemat kõrgemat tunnihinda jne, kuid kõik need meetodid on lühiajalise efektiga ja ei toimi pikemas ajahorisondis.

Sellest tulenevalt on oluline vähem tegeleda raamatupidamislike trikitamistega ning pöörata rohkem tähelepanu oma tootmisprotsessi süstemaatilisele parendamisele. Vähem tähtis ei ole siin ka Täitja arendusmeeskonna motivatsioon, keda samuti huvitab, et võimalikult vähe nende tehtud loomingust läheks ümbertegemisele või raisku.

Kasulike tundide suurendamine ja seeläbi arendusprotsessi efektiivsemaks muutmine oli peamine eesmärk (ülesanne), millele lahendust otsides töö autor **kohustusliku kasutajaliidese prototüüpimise** meetodini välja jõudis. Samuti kirjeldab käesolev töö, kuidas selle meetodi abil Webmedia arendusefektiivsust oluliselt tõsteti.

[3] Uue toote väljatöötamiseks on vaja investeerida inimeste aega. Võttes keskmiseks tunni omahinna kogukuluks 25 EUR-i, on näiteks viieliikmelise meeskonna kogukulu aastas 240 000 EUR-i.

2.2.Arendusefektiivsust mõjutavad tegurid ning nende juurpõhjused

Erinevad nüansid omavad tootmisprotsessile erinevat mõju, kuid lõpuks saab ebaefektiivsuse tilkadest kokku jõgi, mis kasumimassi olematuks sulatab. Tarkvaratootmisega tegelevate ettevõtete arendusjuhtide ainuülesandeks ongi nende tootmisprotsessi negatiivselt mõjutavate pisiasjade leidmine ning parendamine. Parendamiseks aga on vaja alati jõuda juurpõhjuseni, mida peab ja mida üldse saab mõjutada.

Igal tarkvaraarendusega tegeleval ettevõttel peaks olema pidevalt ajakohane nimekiri probleemidest, mis tema tootmisprotsessi efektiivsust langetavad, ning tegevuskava, kuidas neid probleeme kontrolli all hoida või kõrvaldada.

Olgu siinkohal ühe näitena välja toodud IT PRO portaali (http://www.itproportal.com) poolt koostatud nimekiri 10 enimlevinud tarkvara tootmisprotsessi riskidest ja nende võimalikest lahenditest [ITPR]:

1. **Ajahinnangute andmine ja planeerimine**. Individuaalsete tarkvaraprojektide unikaalne olemus tekitab probleeme arendajatele ja juhtidele arenduseks vajamineva aja hindamisel ja planeerimisel. Läbivalt tuleb mõõta ja jälgida olemasolevaid projekte, et neist saadud õppetunde saaks tulevikus rakendada.
2. **Nõuete hulga hüppeline kasv**. Projekti kulgemise käigus võivad varem mittetuvastatud nõuded takistada soovitud tähtaegade saavutamist. Proovi mõelda suurelt juba projekti varajastes faasides ja näe ette halvimat või kõige suurema koormusega stsenaariumit.
3. **Töötajate voolavus**. Iga projekti kallal töötab teatud arv arendajaid. Kui üks arendaja lahkub, võib temaga kaasa minna kriitilise tähtsusega info. See võib kogu projekti takerduma panna või vahel lausa nurjata. Kindlusta ressursside olemasolu, mille abil tiimiliikmed saaksid koostööd teha ja infot jagada.
4. **Algspetsifikatsiooni nõuete omavaheline konfliktsus**. Integratsiooni ja koodikirjutamise algfaasides võivad tekkida konfliktid nõuete vahel. Enamgi veel - arendajad võivad leida, et isegi spetsifikatsioon ise on ebaselge või mittetäielik.
5. **Mured töö efektiivsuse osas**. Pikaajalistes projektides kipuvad arendajad alguses asja vabalt võtma. Selle tulemusena läheb kaotsi palju aega projekti valmimisel. Koosta realistlik ajakava ja püsi selle juures.
6. **Kompromissid projekteerimises**. Tahe hakata tegelema järgmiste „reaalsete“ tööülesannetega paneb arendajad kiirustama projekteerimise protsessis. See on programmeerimistundide raiskamine, kuna projekteerimine on tarkvaraarenduse kõige kriitilisem osa.
7. **Kullaga katmine**. Vahel meeldib arendajatele eputada oma oskustega, lisades mittevajalikke funktsioone. Näiteks võib arendaja lisada lihtsale sisselogimise moodulile Flashi, et see näeks „stiilsem“ välja.
8. **Protseduurilised riskid**. Igapäevane äritegevus võib takerduda ebaõigete protsesside realiseerimise, prioriteetide konflikti või ebaselge vastutuse tõttu.
9. **Tehnilised riskid**. Vahel vähendavad tarkvaraarendusfirmad toodete funktsionaalsust, et kompenseerida eelarve ja ajakava ületamist. Alati on olemas konflikt tarkvara maksimaalse funktsionaalsuse ja parima jõudluse saavutamise vahel.

10. **Vältimatud riskid**. Need sisaldavad muudatusi seadusandluses, tarkvara vananemist ja teisi riske, mida ei saa maandada ega ennustada.

Sarnaseid TOP-e ja artikleid tarkvara arendusprotsessi riskidest on palju ning raske on väita ilma iga konkreetse ettevõtte konteksti teadmata, mis on tähtsuselt number üks ja mis number 10. Samuti on vaieldavad eeltoodud punktides kirjeldatud probleemide parendusettepanekud, kuid näite mõte oli meelde tuletada, milliste probleemidega enamus spetsiaaltarkvara tootmisega tegelevaid ettevõtteid vaeva näevad. Sarnane nimekiri oli ka Webmedias ning vähemalt algusaastatel troonis seal suurima riskina teema: „Iga projekti jaoks ei ole vaja arendada välja selle projekti põhist arendusraamistikku“.

Need nimekirjades väljatoodud riskid taanduvad üldjuhul aga kindlatele juurpõhjustele, miks üks või teine probleem realiseerub. Pretendeerimata ülima tõeni ja tuginedes ainult autori enda 15 aastasele kogemusele tarkvara tootmise valdkonnas, olgu siinkohal välja pakutud tema enda TOP 5 juurpõhjustest, miks tarkvaraprojektid ebaõnnestuvad (st ei vasta lõppkasutaja ootustele või ei saa valmis eeldatud aja, funktsionaalsuse ning eelarve piires) ning seda tähtsuse järjekorras:

1. Täitja projektimeeskonna ebakompetentsus ning kogenematus oma vastutusalas.
2. Tellija projektimeeskonna ebakompetentsus ning kogenematus oma vastutusalas.
3. Tellija kui täitja meeskonnal üheskoos puudub tervikvaade ning arusaam loodavast lõpptulemusest. Üheks juurpõhjuseks kindlasti IT PRO edetabelis olevatele riskidele nr. 1, 2, 4, 5, 6, 7, 8.
4. Nii tellija kui täitja C-taseme ehk otsustajate (tippjuhtkonna) taseme ebakompetentsus – võimetus õigeaegselt otsustada või sekkuda projekti võtmeküsimuste lahendamisse, hoolimata projektimeeskonna poolt korrektselt eskaleeritud probleemidest.
5. Force majeure – põhjused, mille esinemist ka kõige professionaalsem meeskond poleks suutnud ette näha või millelaadset lahendust maailmas varem katsetatud pole.

Selle nimekirja punktide 1,2, 4 ja 5 parendamiseks lihtsad lahendused puuduvad. Enamiku juurpõhjuste kõrvaldamine on raske ja käib läbi pideva õppimise ning kogemuspagasi korjamise ning muudatused võtavad aastaid aega. Küll aga saab oluliselt parandada tarkvaraprojektide ebaõnnestumiste juurpõhjust number 3 ehk siis otsida lahendust, kuidas Tellija ja Täitja saavutaksid ühise tervikvaate plaanitavast lõpptulemusest.

Tellija ja Täitja vahelist konflikti tarkvara arendamise protsessis, mille põhjuseks on erinev arusaam lõpptulemusest, hakkas autor kutsuma **kollase kassi probleemiks**.

Järgnev peatükk keskendubki Kollase kassi probleemi kirjeldamisele ning ülejäänud töö tutvustab lahendusi, kuidas seda suure mõjuga efektiivsuskao juurpõhjust tarkvaraarenduse protsessis kontrolli all hoida või üldse kõrvaldada.

2.3.Kollase kassi probleem

Tellija ja Täitja mõttemaailmad on oma olemuselt konfliktis. Lihtsustatult: üks pool tahab saada võimalikult palju ja maksta võimalikult vähe, kui samal ajal teine pool tahab toota soovitud tulemuse võimalikult väikese ressursikuluga (arendusefektiivselt). Ehk siis igas tootmisprotsessis on oluline leida tasakaalupunkt Tellija soovide suuruse ja reaalse vajalikkuse vahel ning seejärel toota nõuetele vastav lahendus. Vaja on kokkulepet Tellija ja Täitja vahel, millest mõlemad pooled ühtemoodi aru saavad.

Kahjuks isegi siis, kui mõlemad pooled annaksid endast maksimumi parima lahenduse saavutamiseks, segab neid eesmärgini jõudmisel **kollase kassi probleem**. See probleem ei ole tarkvaraarenduse-spetsiifiline, vaid sobib igasse valdkonda, kus on tegemist tellija soovide täitmisega.

Iseloomustamaks probleemi olemust, hakkas autor enda poolt läbiviidud tarkvara arendusprotsessi parendamise koolitustel läbi viima järgnevat katset.

Koolitatavad (üldjuhul 12-16 inimest) jagati kaheks rühmaks. Ühed olid tellijad ja teised täitjad. Paremaks ilmestamiseks anti neile ka detailsem roll: näiteks täitjate rühm koosnes ehitajatest (programmeerijatest), arhitektidest (analüütikutest), testijatest ja projektijuhist; tellijate rühm projektijuhist, äriarhitektidest, lõppkasutajatest jne. Seejärel anti kõikide inimestele valge paber ja paluti kirjutada lause:

„Üks kass läheb üle silla“

Süsteemianalüüsi kohaselt on tegemist situatsiooniga, kus tuleb luua lahendus kassile üle silla käimiseks.

Seejärel paluti igal osalejal seda situatsiooni („kass läheb“) ja objekti („sild“) ette kujutada ning kirjutada enda ees olevale paberile vastused alljärgnevatele küsimustele (aeg- ajalt anti ka valikvastuseid ette, aga katse mõtet ja tulemust see ei mõjutanud):

- Mis värvi kass? ……. (valge/must/oranzh/kollane…)
- Millest on tehtud sild? ……. (kivist/puust/rauast/…)
- Mis tüüpi sild? ……. (kaarjas/sirge/rippsild/…)
- Liikumise suund? ……(vasakule/paremale)
- Kui kaua kass silda ületas? ……(hetkega/minuti/10 minutit/…)

Vastuseid analüüsides oli tulemus alati ühesugune (korratud rohkem kui kahekümnel juhul): kahte ühesuguste vastustega paberit ei leidunud.

Seega - kuidas saavad Tellija ja Täitja töötada ühise eesmärgi nimel (realiseerida sild), kui projekti kõigil liikmetel on erinev arusaam sellest, kuidas peaks lõpptulemus välja nägema? Loomulikult on olemas tõenäosus, et mõnedel katses osalenutel vastused klapivad, kuid silla realiseerimiseks on vaja kõikidele projekti liikmetele tekitada ühine arusaam, mida tahetakse ehitada. Sellest ei piisa, et kaks projektiliiget ühist nägemust omavad.

Ilma ühise arusaamise tekitamiseta on Tellija projektiliikmete nägemus kassi jaoks vaja minevast sillast tõenäoliselt selline nagu on kujutatud Joonisel 3.

Joonis 3 Tellija nägemus sillast

Täitja tõenäoliselt tahaks realiseerida midagi lihtsamat (Joonis 4)

Joonis 4 Täitja nägemus sillast

Kokkuvõtlikult seisneb kollase kassi probleem selles, et meil võib olla lihtne lause: „üks kass läheb üle silla“, mille sisu kõik koosoleku laua ümber viibijad tarkade inimestena tundusid mõistvat, kuid reaalsuses on ilma visuaalse pildita meil võimatu saavutada ühist arusaama Tellija ja Täitja vahel vajalikust lõpptulemusest. Selles situatsioonis lõpeb silda realiseerima minek pideva vajaduste täpsustamise, möödarääkimise ja ümbertegemisega. Loogiline järeldus eelnevast:

Et kõik osapooled saaksid aru, millest jutt käib, on üks pilt parem kui 1000 sõna.

Maja ei hakata ehitama ilma detailsete joonisteta, samas tarkvara tootmine pelgalt spetsifikatsiooni alusel on tarkvaraarenduse projektides jätkuvalt valdav. Seda hoolimata riiulitetäitest raamatutest, mis kirjeldavad, kui oluline on lõppkasutaja kaasamine tarkvara arendamise protsessi kõikides faasides.

Kui eelnev pärisellu tõlkida, siis kipuvad tarkvaraanalüütikud liiga palju projekti algusfaasis kirjutama tekstilist dokumenti selle asemel, et visualiseerida lõpptulemust. Aga just visualiseerimine annab ühise arusaama, milline lõpptulemus välja hakkab nägema.

Autori isiklik kogemus on, et ükski klient ei viitsi mõttega lugeda 500 lehekülge spetsifikatsiooni uuest planeeritavast IT lahendusest. Tekib küsimus, mis sellest lugemisest kasu on, kui projekti iga osaline omab ka kõige lihtsamast lausest oma ettekujutust.

Kui projekti liikmetel ei ole ühist arusaama eesmärgist ja lahendusest, ei saa olla ka olemas Tellija ja Täitja vahelist kokkulepet. Kokkuleppe puudumine aga viib konfliktideni ning efektiivsus langeb.

Täitja peamine motivatsioon on tarnida kliendi soovitud tulem võimalikult efektiivselt. Iga soov, nõue, muudatus või muu tegevus, mis seda efektiivsust ohustab, klassifitseerib täitja silmis kohe hirmuks kulude kasvamise ees. Ebapiisav tagasiside analüüsifaasis ning suutmatus adekvaatselt lõppkasutaja teste juba enne programmeerimise faasi läbi viia toovad aga kaasa Täitja motivatsiooni languse ning seega kogu projekti kvaliteedi languse.

Klassikaliseks õudusunenäoks on need kliendid, kes spetsiaaltarkvara tellimise käigus hakkavad alles oma äri ja sellega seotud protsesse välja mõtlema ning soovivad seejuures jääda algselt kokkulepitud aja ja eelarve raamidesse.

Isiklik kogemus näitab, et täitja suurim stressiallikas ning motivatsioonilangetaja on see, kui tellija hakkab oma äri olemust analüüsima ning seda parendama alles rakenduse realisatsiooni käigus, mitte eelnevalt. IT rakendus peab olema lahend vajalikele muutustele, mitte põhjus muutusteks.

Samas on ju hea, kui Tellija ise ka lõpuks aru saab, mida ta tahab. Täitja silmis oleks kõige ideaalsem lahendus, kui Tellija saaks lahenduse sobivust katsetada enne seda, kui Täitja seda lahendust üldse programmeerima on hakanud.

Tuleb välja, et esialgu võimatuna tunduv ülesanne kogu rakendus enne tema valmimist valmis ehitada ei olegi nii võimatu ning sellele lahenduse kirjeldamisele keskenduvad järgnevad peatükid.

Lisakommentaar „Kollase kassi probleemi“ peatükile. Arusaadavalt võis lugejal tekkida eelnevat katset läbi tehes küsimus: kui eesmärgiks on ehitada sild, siis mis tähtsus on kassi värvil või sellel, kas kass liigub vasakult paremale või vastupidi.

Kassi värvil antud ülesandes ei olegi tähtsust, seetõttu on ka „Kollase kassi probleem“ saanud oma nimetuse pigem turunduslikel eesmärkidel, et sellele nähtusele rohkem tähelepanu pöörata. Kassi liikumise suund on tähtis, sest see tekitab uusi küsimusi: à la kas sillale peab mahtuma samaaegselt liikuma kaks kassi või isegi rohkem. Kui Tellija vastus on, et sillal peab saama liigelda rohkem kui üks kass, siis vaadeldes Joonist 4 (Täitja nägemus sillast), oleksime juba probleemi ees (muidugi juhul, kui kassid alati kokkuleppeliselt ühes suunas ei liigu hetkedel, kui neid on silda ületamas mitu).

2.4.Täiuslik lähteülesanne

Enne, kui kirjeldatakse lahendust Kollase Kassi probleemile ehk siis sellele, kuidas tekitada ühine tervikvaade Tellija ja Täitja meeskondade vahel lõpptulemusele, vajab selgitamist mõiste „täiuslik lähteülesanne“.

Hoolimata eelkirjeldatud hirmudest on Tellija ja Täitja motivatsioon lõpptulemuse osas ühtne. Ideaaltingimustel[4] on mõlemal poolel tarkvaraarendamise protsessis sama huvi:

- Maksimaalselt hea lõpptulemus
- Teha realisatsiooni käigus võimalikult vähe juba tehtud tööd ümber, st saavutada maksimaalselt hea lõpptulemus minimaalselt vajalike töötundide hulgaga
    - Tellija jaoks oluline: rahaline ja ajaline kulu
    - Täitja (kui ei ole tegemist pahatahtlikkusega): mida efektiivsem on töötaja, seda rohkem tema pealt teenib.

Seega, et võimalikult vähe tööd ümber teha, peavad nii Tellija kui Täitja oma koostöös püüdlema nõnda nimetatud **täiusliku lähteülesande** poole. Definitsioon:

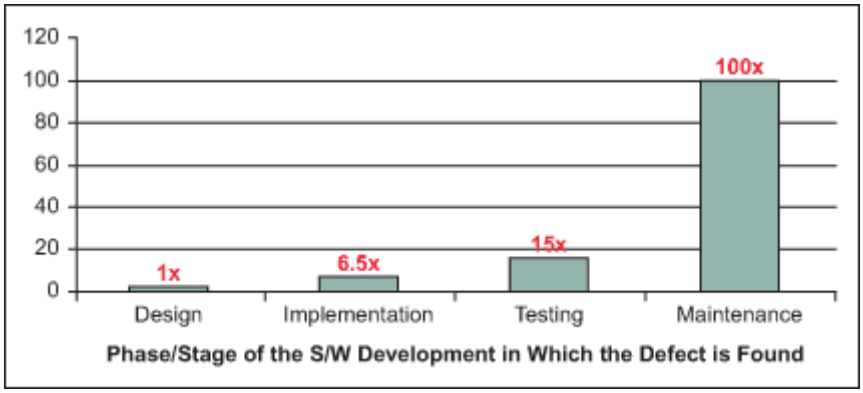
„Täiuslik lähteülesanne vastab Tellija poolt defineeritud vajadustele ning on Täitja poolt realiseeritav ja testitav ühtegi lisaküsimust Tellijale esitamata“.

Tarkvaraprojekti lõpptulemusena tahab Tellija kasutada töötavat ja vajadustele vastavat tarkvara. Lõppeesmärgi täitmise seisukohalt on ainus füüsilist väärtust omav töö programmeerimine ning analüüs ja kvaliteedikontroll (testimine) on abitegevused.

Paremaks arusaamiseks võrdlus ehitussektoriga. Kui Tellija sooviks on saada maja, siis füüsilist väärtust loob ehitaja. Arhitekti ja järelevaataja rollid on olulised, kuid reaalse väärtuse loomisel abitegevused. On oluline, et programmeerija kirjutaks kohe õige koodi nii, et seda ei peaks hiljem minema viskama või oluliselt parandama.

Seetõttu muutub eriti oluliseks tarkvara analüüsi faas, sest vea leidmine ja parandamine selles faasis on tootmiskulu seisukohalt odavaim (vt Joonis 5):

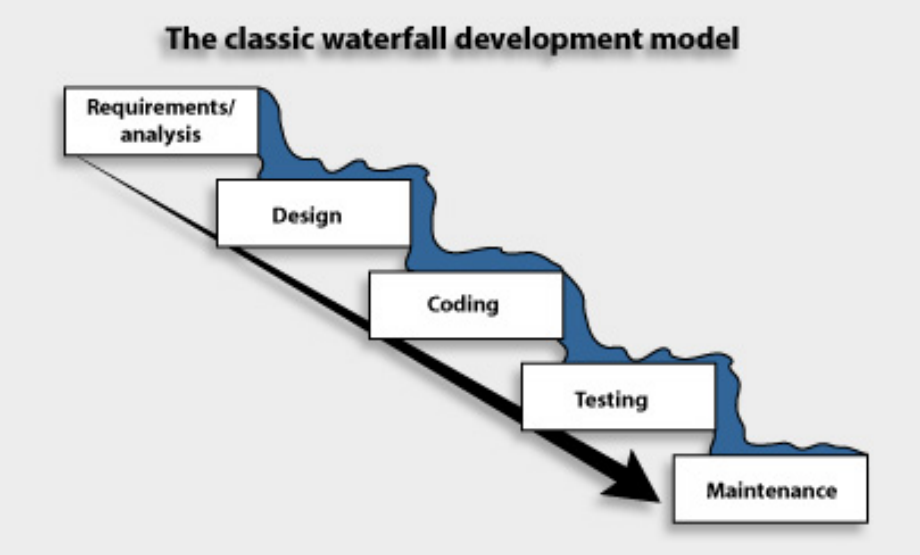
Joonis 5 Vea parandamise kulukordaja sõltuvalt vea avastamise faasist [IBMI].



Eeltoodud mõttekäik selgitas, miks tarkvara arendamises niivõrd kaua kasutati klassikalist kosemeetodit (*Waterfall*). Esimeses etapis analüüsitakse Tellija vajadused kõikidest tarkvara arendamise jaoks vajalikest aspektidest läbi (sh nii funktsionaalsed kui mittefunktsionaalsed nõuded), seejärel luuakse analüüsile vastav kood ning testitakse lõpptulemust.

[4] antud töös on välja jäetud juhud, kus tellija või täitja osapool teadlikult venitab või töötab vastu parimale lõpptulemusele

Joonis 6 Klassikalise kosemeetodi skeem [Snai].



Kosemeetod töötab ideaalse lähteülesande puhul täiuslikult, kuid siin on probleem. Päriselus pole definitsioonile vastav ideaalne lähteülesanne teostatav. Tellijal ja Täitjal on ainult võimalik oma töös püüelda ideaalse lähteülesande poole ning nõuete kogumise ja analüüsi faasis tehtavad vead on paratamatus. Samuti võib olla realisatsiooni keerukuse vähendamise eesmärgist või süsteemi töökindluse tõstmise eesmärgist tulenevalt mõistlik esialgselt planeeritud lahendus ümber disainida. David Parnas kirjutas oma raamatus *A Rational Design Process: How and Why to Fake It [Parn]* sellest juba aastal 1986*:*

“Paljudest süsteemi detailidest saame me teada alles süsteemi realiseerimise käigus. Mõned neist detailidest lükkavad ümber meie esialgse lahenduse, mistõttu peame varasemasse punkti tagasi pöörduma.“

Sellest tulenevalt on kosemeetodist oluliselt efektiivsem kasutada tarkvara arenduses agiilseid (Agile Software Development [Cock]) meetodeid, kuid nende kasutamisest üksi aga ei piisa, sest efektiivsuse tõstmiseks on vaja ikkagi lahendada enne programmeerimise etappi põhilised probleemid:

1. Kuidas testida programmeerimisse mineva funktsionaalsuse vastavust Tellija poolt püstitatud vajadustele enne kodeerimise alustamist?
2. Kuidas lahendada ära Kollase kassi probleem ehk siis kõik projekti osapooled saavad üheselt aru, milline lahendus realiseeritakse?

Arendusmetoodika võib olla kuitahes hea. Reaalsus on see, et Tellija ise saab lahenduse sobivusest aru alles pärast seda, kui on pakutud lahendust proovinud (nö käega katsunud).

Kokkuvõtteks. Ainult jooniste või pildi järgi ei ole võimalik hakata lahendust programmeerima. Programmeerija töö aluseks on tehniline ülesanne, mis sätestab töö detailse sisu, mida ta programmeerima peab hakkama. Nii Tellija kui Täitja huvides on, et see dokument oleks maksimaalselt hea ehk siis võimalikult lähedane täiuslikule lähteülesandele. Tehnilise ülesande struktuuri näide on toodud Lisas 3.

3. Lahendus

Probleem, millele töö autor noore analüütikuna 2000. aastal vastust hakkas otsima, on väljendatav järgneva mõttekäiguna:

1. Tellija tahab saavutada tarkvara abil oma ärieesmärke.
2. Enamikul tellijatel puudub ettekujutus, millist tarkvara nad täpselt vajavad oma ärivajaduste rahuldamiseks.
3. Enamik Täitja meeskonna liikmetest ei oma valdkonnaspetsiifilisi teadmisi Tellija maailmast, mistõttu nad ei ole võimelised kirjutama rätseplahenduste puhul täiuslikku (ideaalset) lähteülesannet programmeerijatele ilma tellija heakskiidu ja abita.
4. Lisaks on projektimeeskonnas Kollase kassi probleem ehk siis iga projektimeeskonna liikmel on soovitud lõpptulemusest erinev ettekujutus.

Vahemärkus: annotatsioonis on välja toodud olulise kitsendusena see, et Webmedia keskendus 90% ulatuses oma toodangus töömahukate (vähemalt 40 kasutuslooga) veebirakenduste realiseerimisele, nagu suurfirmade iseteenindused ja tagatoa (back-office) töövahendid, kus lõppkasutaja kasutajaliides oli projektide õnnestumise seisukohalt ülikriitilise tähtsusega.

Seega maksimaalse arendusefektiivsuse saavutamiseks püstitas töö autor analüüsiprotsessile järgmised eesmärgid:

1. **Testida programmeerimise lähteülesande vastavust Tellija vajadustele enne programmeerimise etapi algust –** st saada kliendilt kinnitus enne programmeerimist, et just sellist tarkvara ta endale tahab.
2. **Kirjutada täiuslik lähteülesanne –** st realiseeritav ilma täiendavaid küsimusi esitamata ning hilisema arenduse käigus muudetakse/täiendatakse minimaalselt.

Enne oma lahenduse leiutamist uuris autor ka mujal maailmas tehtut ja loomulikult ei olnud ta ainuke, kes sarnaste probleemidega maadles.

See, et põhjalike analüüsidokumentide kirjutamisest üksi jääb väheks (Tellija ei ole suuteline neid mõttega läbi töötama) ning on vaja rohkem tähelepanu pöörata visualiseerimisele, oli arusaadav ka näiteks ettevõttele Rational Software, kes suurema tuntuse saavutas aastal 2003 pärast müüki IBM-le ja kes oma sajandivahetusel ülipopulaarse Rational Unified Process-i (RUP) metoodika peamise rõhuasetuse kasutuslugude- ja protsessivisualiseerimisele üles ehitas [RUPT]. Kahjuks ei olnud see käesoleva töö autori silmis piisav.

Isiklik kogemus mitmes erinevas projektis näitas, et RUP-i aluseks olev Unified Modeling Language (UML) on mitte infotehnoloogia erialase haridusega inimestele liialt keerukas, mistõttu joonistest ei saada aru või siis teeseldakse mõistmist. Täitja võib enne programmeerimise faasi algust isegi Tellija käest kokkuvõttes saada kinnituse analüüsi dokumendile, kuid kuna Kollase kassi probleem on lahendamata ja visuaalset kokkulepet Tellija/Täitja vahel ei ole, siis tuleb hiljem ikkagi suures koguses täpsustusi ja ümbertegemist.

Klassikaline on siinkohal olukord, kus valminud tarkvaras on oluline puudujääk. Täitja viitab kinnitatud analüüsidokumendile, kus puuduolevast funktsionaalsusest pole juttugi, ja kuna Tellija on dokumendi kinnitanud, siis on õigus Täitjal. Tellija samas tunneb, et tegemist oli triviaalse funktsionaalsusega ning ta ei tulnud selle pealegi, et seda analüüsidokumendist kontrollida („Täitja ei küsinud ja Tellija ei arvanud, et peaks sellest rääkima“). Tellija õpib vigadest ning järgmises hankes on sees täiendav tingimus, et analüüsidokumendi sisu võib realisatsiooni töö käigus 15% muutuda (ehk

siis, kui Tellija on midagi ära unustanud või analüüsi dokumenti kinnitades kahe silma vahele jätnud, saab seda ilma juurde maksmata muuta/täiendada).

UML-i keerukus, võimalike vigade jaoks suurte aja- ja rahapuhvrite jätmine ning muude Tellija ja Täitja riskide pidev ennetamine, et mitte süüdi jääda, ei too Tellijat ja Täitjat üksteisele lähemale, vaid pigem tõukavad eemale.

Teistelt õppimiseks pööras töö autor pilgu ehitusvaldkonnale, kus Tellija ja Täitja vahelist koostöökogemust on arendatud aastatuhandeid.

Keskmist Tellijat huvitab maja planeerimisel eelkõige maja välimus, ning kuidas on rahuldatud maja funktsionaalsed ning mittefunktsionaalsed nõuded. Vähe on neid, kes Tellijana ise tahaksid Autocad-i (levinud joonestustarkvara) faili mõtestatult kasutada, uurida sõlmede ülesehitust, teha tugevusarvutusi konstruktsioonile jne.

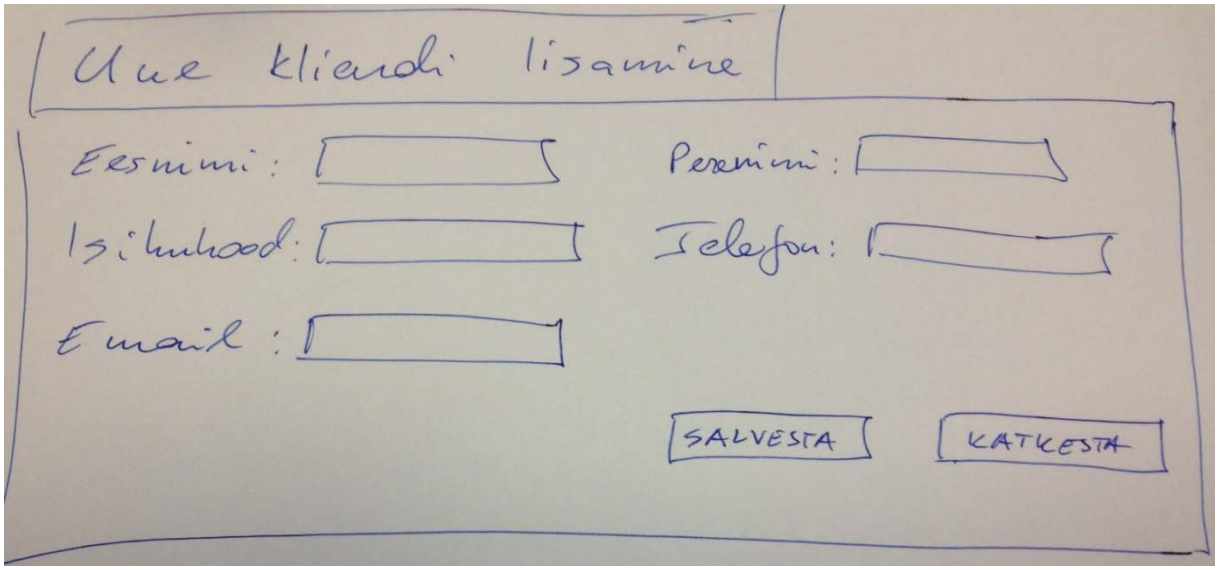
Enamiku Tellijate jaoks on oluline ainult visuaalne väljund, st arusaamise tase ning ilumeel on võimeline kinnitama ruumi planeeringu, visuaalselt hoomatava välise arhitektuuri ning sisekujunduse lahenduse. Kõik muu jääb tahes- tahtmata konstruktoritele, ehitajatele, järelevalvajatele jne.

Arvestades, et üldine teadmine ja arusaamine tarkvaralahendustest on Tellijal veel nõrgem kui maja ehitamisel, siis kuidas tekitada kindlus, et Tellija lõplikult aru saab, mida ta tahab? Vastus – loodav tarkvaralahendus tuleb visuaalselt prototüüpida.

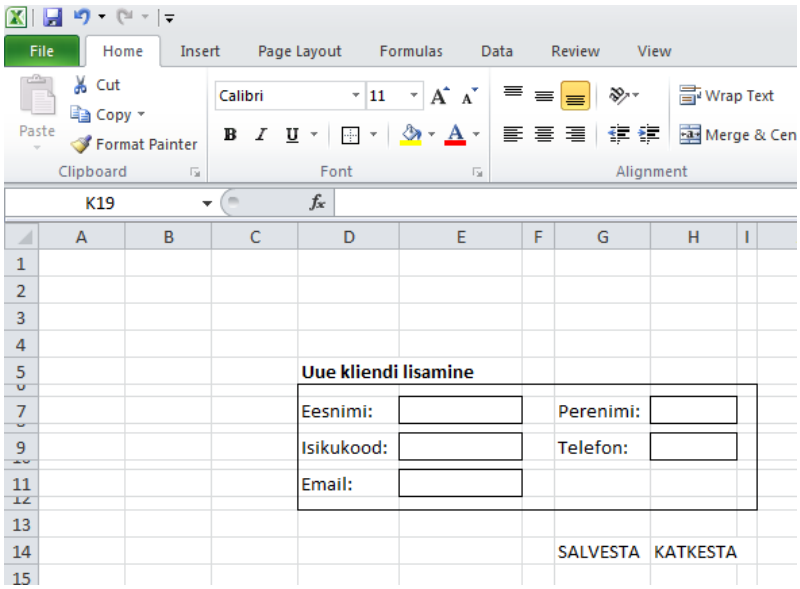
Tellija ei vaja valmis lahendust komponendist, nagu ta ei vaja ka valmis maja selleks, et kinnitada lahenduse sobivust enda jaoks. Ta peab seda komponenti või siis maja puhul arhitektuurivaateid lihtsalt visuaalselt nägema.

See tähendab, et me ei pea komponendi prototüüpimiseks teda valmis programmeerima, vaid piisab, kui visualiseerime tarkvaralahenduse endale mugaval viisil. On selleks siis käsitsi paberile joonistatud lahendus või Excelis disainitud vaade või midagi muud. Peaasi, et see on olemas ning selle abil saaks nii Tellija kui Täitja analüütik verifitseerida lahenduse vastavust nõudele (Joonised 7-8).

Joonis 7 Seinatehnika näide (Whiteboard + marker + fotoaparaat)



Joonis 8 Visualiseerimine Exceli abiga



Komponendi visualiseerimine lahendab ära Kollase kassi probleemi ehk nii Tellija kui Täitja meeskond saab aru, kuidas uus komponent oma sisult välja hakkab nägema. Kuid veel olulisem võit tuleb pildi kaudu mõtlemisel pisiasjade täpsustamise seisukohalt.

Eelnevalt visandatud joonised uue kliendi lisamise kohta võivad tunduda näitena triviaalsed, kuid analüüsides neid detailsemalt, tekivad kohe küsimused lähtuvalt võimalikest veasituatsioonidest. Kas telefoninumbri sisestamise formaat ei peaks olema kliendile ette öeldud (nt. +372XXXXXXX)? Kas rakendust peaks saama kasutada ka välisriigi kodanik, kellel standardile vastav isikukood on puudu? Kas nime väljal on pikkuse piirangud või mitte? Kas väljade järjestus on optimaalne või peaks andmete sisestamist alustama isikukoodist? Jne

Tegemist ei ole triviaalsete küsimustega ja kogemus on näidanud, et ka Tellija ise ei suuda ilma visuaalse väljundita endale sellelaadseid küsimusi esitada. Isegi esmatasandil ehk siis näiteks, kas soovitud atribuutide hulk on piisav või mitte. Kas isikukood üldse on vajalik kliendi registreerimisel või mitte?

Visualiseerimise olulisusest tulenevalt olid esimesed suuremad täiendused Webmedia tarkvara arendusprotsessis aastal 2001:

1. *Kõik visuaalset väljundit omavad komponendid tuleb visualiseerida ning klient peab komponendi visuaalse väljundi kinnitama (vastu võtma).*
2. *Tellija poolt kinnitatud komponendi visuaalne väljund on programmeerija tööülesande aluseks olevat dokumendi ehk tehnilise ülesande kohustuslik osa (vt Lisa 3: tehnilise ülesande näide).*

Viimane nõue oli karm. Programmeerijal oli keelatud komponendi realiseerimist enne alustada, kui oli olemas kliendipoolne aktsepteerimine realiseeritava komponendi visuaalsele väljundile.

Muudatuse sisseviimise tulemusena langes üle kahe korra komponentide realiseerimiseks kulutatava tööaja maht ning seda nii esmaarenduse kui kliendi ülevaatuse etapis. Programmeerija sai kohe aru, mida temalt tahetakse ning ei pidanud ise lahendust leiutama hakkama. Klient aga võttis vastu juba varem enda poolt kinnitatud lahenduse ning puudujääke esines oluliselt harvemini.

Oluline on toonitada, et visualiseerimine kõiki võimalikke veasituatsioone loomulikult ennetada ei suuda. Näiteks sätestati rahvastikuregistri projekti Reginat (koodnimi) tehes nii perekonnanime kui

ka eesnime maksimaalseks pikkuseks 50 tähemärki. Pärast mõneaastast rakenduse kasutusel olemist selgus, et araabia nimede registrisse kandmisel jääb sellest pikkusest väheks. Puhas kogemuslik teadmine ning visuaalne analüüs poleks seda küsimust aidanud lahendada.

Klassikalisele spetsifitseerimisele on alternatiive otsitud ammu. Ühe näitena on nii California Ülikoolis Ameerika Ühendriikides [Boch] kui ka Aalborgi Ülikoolis Taanis [Math] uuritud sügavuti teemat „Prototüüpimine vs spetsifitseerimine“. Mõlema ülikooli teadlased on jõudnud sarnastele järeldustele:

- Prototüüpide tegemine võimaldas valmistada tooteid enam-vähem sama jõudlusega, aga ligi 40% vähema koodiga ja 45% vähema vaevaga.
- Prototüüpidega tooted olid mõnevõrra madalama funktsionaalsuse ja töökindlusega, aga kõrgema kasutusmugavuse ja õppimislihtsusega.
- Spetsifitseerimine tekitas sidusamaid tehnilisi projekte ja lihtsamini integreeritavat tarkvara.

Antud uuringute puhul ei mõisteta prototüüpimise all mitte visuaalsete vaadete joonistamist, vaid reaalse lahenduse realiseerimist (valmisprogrammeerimist väiksemas funktsionaalsuses). Samuti oli lahendavate ülesannete juures vähemtähtis lahenduse kasutajaliides. Antud töös aga on keskendutud just lahenduse pakkumisele tarkvaratootmises, kus just kasutajaliides omab eriti olulist rolli, ning pakutud lähenemine visuaalse prototüüpimise näol on uudne. Eeltoodud teadustööd aga kinnitavad pigem agiilsete arendusmeetodite tähtsust tarkvaraarenduses. Visuaalne prototüüpimine on siinkohal üks arendusprotsessi etapp ning ei ole sõltuvuses arendusmetoodikast, à la kas agiilne või kosk.

3.1.Ajalootund: „Arene koos projektiga“

„Arene koos projektiga“ oli Webmedia hüüdlause perioodil 2001-2005. Noor ja kogenematu peamiselt tudengitest koosnev meeskond pidi oma teadmiste puudujääki pidevalt kompenseerima ületundide ja nädalavahetustel töötamisega. Iga projektiga saadi targemaks, parandati projektijuhtimist, arendusraamistikke jne. Koos „projektiga“ arenes edasi ka prototüüpimine.

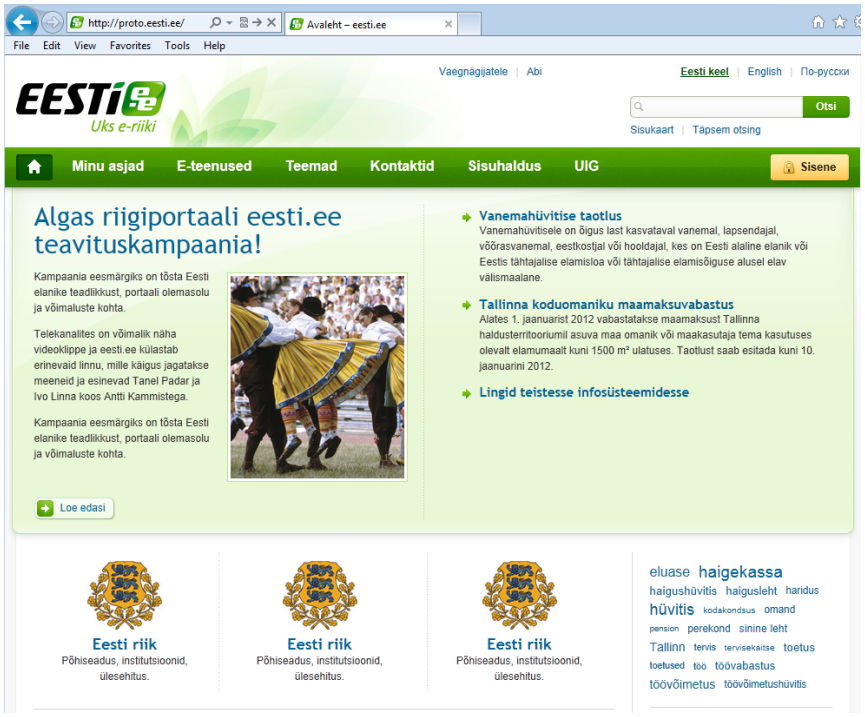
Analüüsiprotsessi ülesehitamine pildi kaudu mõtlemisele võeti klientide (Tellijate) poolt üle ootuste hästi vastu ning prototüüpimisest sai üks tugev konkurentsieelis võrreldes teiste turul tegijatega.

Kui senini oli protsess näinud välja viisil, kus Täitja tuleb Tellija juurde, istutakse koosolekulaua ääres, Tellija kirjeldab oma vajadusi, Täitja teeb märkmeid ning kirjutab kokku analüüsidokumenti, siis Webmediaga koostööd tehes asendus tuim dokumendipõrgatamine ühiseks tahvli ees mõtlemiseks. Tellija esindajatel meeldis Webmediaga koos analüüsi teha, sest nad said ka ise visualiseerimise kaudu oma vajadustest ning soovidest täpsemalt aru.

Viimastel aastatel on visuaalse prototüübi tähtsust mõistetud laiemalt ning ka Webmedia kliendid on oma teistelt partneritelt hakanud nõudma visuaalse prototüübi olemasolu või siis lausa endale püsti pannud prototüüpimise keskkonna.

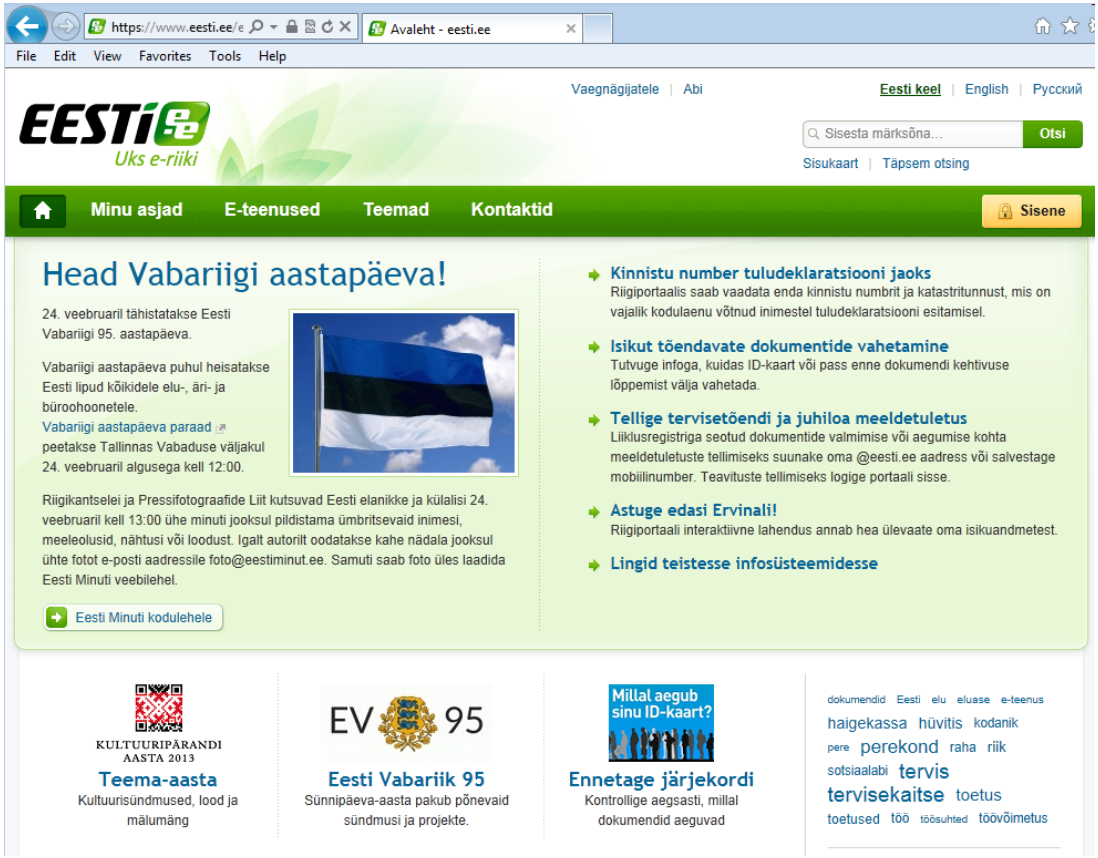
Üheks näiteks Eesti.ee, mille prototüüpkeskkond asub aadressil http://proto.eesti.ee ja näeb välja alljärgnev (Joonis 9):

Joonis 9 Eesti.ee prototüüp



Eesti.ee toodangkeskkond samal ajal on selline (Joonis 10):

Joonis 10 Eesti.ee originaal (Veebruar 2013)



Visualiseerima hakati esimesest hetkest kui Täitja projekti juhtivanalüütikul tekkis arusaam loodava süsteemi olemusest. Vahel võis see juhtuda isegi esmasel nõuete täpsustamisel. Kohustus oli vältida olukorda, kus laua ümber olijad võiksid üksteisest erinevalt aru saada. Ehk siis igal sammul tuleb vältida kollase kassi probleemi:

„Kui jutu sisu jääb segaseks, siis visualiseeri. Hoolimata sellest, kas tegemist on esmase nõuete täpsustamise, protsessimudeli või konkreetsete komponentide analüüsi või muu arendusprotsessi osaga.“

3.2.Täisfunktsionaalne prototüüp

Pärast esimestes suuremates arendusprojektides prototüüpimise kasutamist oli selge, et tehniliste ülesannete täiendamine pelgalt komponentide visuaalse väljundiga ei ole piisav. Keerukamate, eelkõige mitmest sammust koosnevate protsesside puhul peavad nii Tellija kui Täitja saama aru protsessi toimimisest tervikuna. Ilma selleta ei ole võimalik lõppkasutaja kogemust testida ning protsessi efektiivistada.

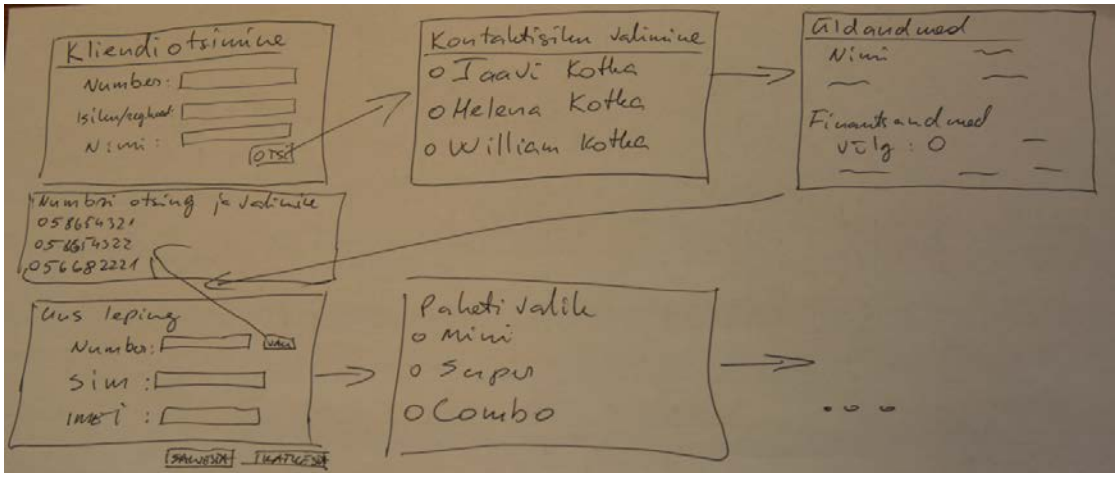
Näiteks on mobiilioperaatorite jaoks vajalik funktsionaalsus: „Uue lepingu sõlmimine olemasoleva kliendiga“. Klient tuleb esindusse ja soovib endale täiendavat SIM-kaarti koos telefoninumbriga. Protsessina näeb see välja nii:

1. Teenindaja identifitseerib kliendi (olemasoleva telefoninumbri, kliendi nime või isiku/regkoodi järgi). *Kliendi otsingu komponent. Kliendi nimekirja komponent.*
2. Kontrollib teenindatava kui kliendi esindaja esindusõiguse alust. *Kontaktisiku valimise komponent.*
3. Kontrollib kliendi staatust ning finantsinformatsiooni (võlgnevused, krediidireiting). *Kliendi üldandmete komponent ja kliendi finantsinfo komponent*.
4. Positiivse voo korral liigub edasi uue lepingu sõlmimise protsessi, kus valitakse telefoninumber, seotakse SIM-kaart ja muud lepingu alusatribuudid. *Uue lepingu lisamise komponent, numbrivaliku komponent, SIM-kaardi andmete lugemine välisseadmega.*
5. Seejärel valitakse pakett ja soodustused. *Paketi valiku komponent ja soodustuste valiku komponent*.
6. Teenuste valimine, kuna viimane sõltub paketist ja soodustustest. *Teenuste valimise komponent*.
7. Valikute ülevaatamine ja kinnitamine ning lepingu printimine. *Uue lepingu lõppvaate komponent ja kinnitamine, uue lepingu printimine*.

Üks tehing. Seitse sammu. 13+ komponenti, mis on kõik kirjeldatud eraldi tehnilise ülesandena ja omavad visuaalset väljundit. Miks neid komponente peab nii palju olema? Definitsiooni järgi kirjeldatakse iga korduvkasutatav loodava tarkvara komponent iseseisvana (ühes tehnilises ülesandes). Aga see ei ole antud töö juures oluline.

Oluline on, et Tellija tahab näha „Uue lepingu sõlmimist olemasoleva kliendiga“ kui tervikprotsessi. Ei piisa sellest, et joonistatakse üles, kuidas uue lepingu lisamine välja näeb, vaid oluline on anda kliendile terviklik kasutuskogemus. Piltlikult öeldes, mis on esimene vaade, mida lõppkasutaja näeb, mis tuleb esimese vaate järel, teise vaate järel jne (Joonis 11).

Joonis 11. Protsessijoonis vaadetega.



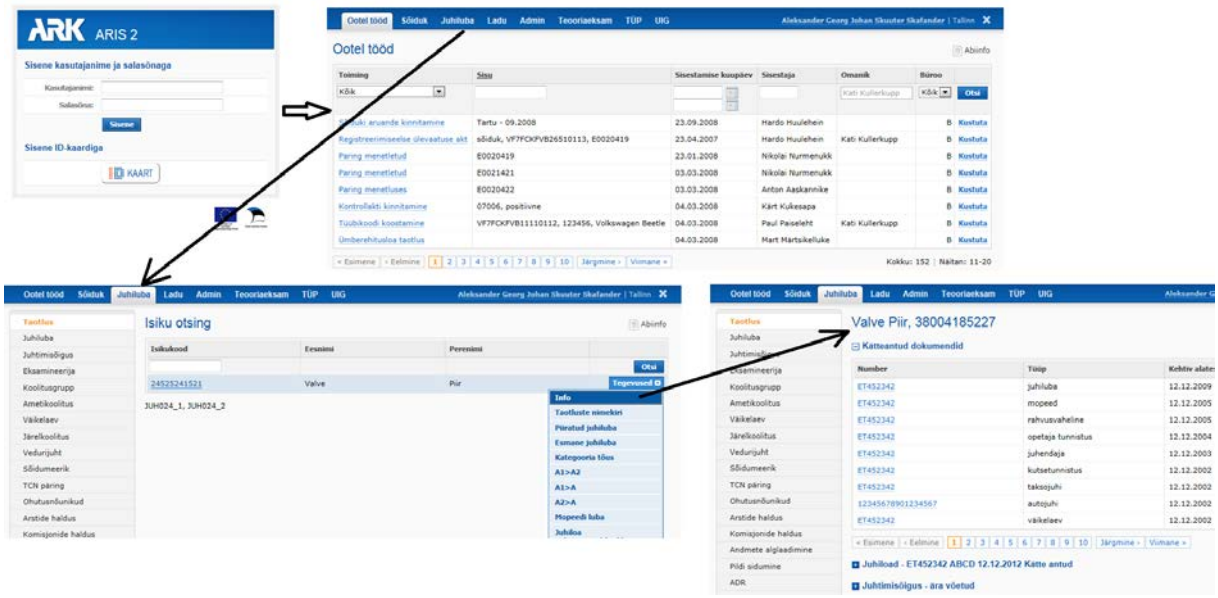
Jooniselt 11 on ilmekalt näha, et selle ülesande lahendamiseks seinatehnikast ja tahvlist (*whiteboardist*) enam ei piisa. Ka väljaprinditud komponentide detailsemate vaadetega lõppkasutaja silme ees mängimisest jääb väheks, sest väikseimgi parandus tahab ümber tegemist ning omavaheliseks sidumiseks tuleb sõna otseses mõttes sein pabereid täis kleepida ja omavahel nooltega ühendada.

Lõppkasutajad löövad uue tarkvara arendamise protsessis kaasa enamjaolt oma põhitöö kõrvalt, mistõttu nende keskendumisvõime ja vastuvõtlikkus uute protsesside ja lahenduste osas on äärmiselt piiratud. Seetõttu on nii Tellija kui ka Täitja vaatenurgast oluline anda visuaalse prototüübiga lõppkasutajale võimalikult lõpplahenduse sarnane kogemus ehk - kui pärast visuaalse prototüübi ülevaatamist lõppkasutaja küsib, et „kas nii hakkabki uus protsess ja tarkvara välja nägema?“, siis ainuke vastus saab olla „jah“.

Mis on lahendus? Siinkohal tuleb veel kord tähelepanu juhtida sissejuhatuses toodud kitsendusele. Webmedia oli 10 aastat tagasi 90% ulatuses brauseripõhiseid rakendusi tootev ettevõte ehk siis täisfunktsionaalse prototüübile lahenduse otsimist hõlbustab see oluliselt.

Kui eesmärk on anda lõppkasutajale võimalikult reaalne kasutuskogemus, siis brauseripõhise rakenduse korral on iga kasutajavaade põhimõtteliselt üks HTML fail, mis omavahel lingituna tekitavad efekti nagu olekski tegemist päris lahendusega.

Joonis 12 Isikuga seotud juhilubade andmete vaatamine



Kasutajakogemuse kvaliteedivahe on Joonisel 11 ja Joonisel 12 kujutatu osas ilmne. Eriti veel siis, kui Joonisel 12 toodud vaated on omavahel lingitud ka reaalselt. Kusjuures kõik lingid ei pea töötama. Näiteks „Muuda“ nupp loendi iga kirje järel võib viia alati ühe ja sellesama kirje detailandmete vaate lehele. Eelmise ja järgmise vaate info peab olema loogilises seoses, kuid mitte üks-ühele vastav. Eesmärk on edasi anda idee, kuidas komponendid on omavahel seotud.

Isiklik kogemus: 2004. aastal, kui Tartu Ülikooli Kliinikumis testiti lõppkasutajatega loodavat haiglainfosüsteemi lahendust kasutades täisfunktsionaalset prototüüpi, tuli juba esimesel päeval tagasiside arstidelt, et küll uus lahendus on kiire ja mugav, aga andmeid ei salvestanud korrektselt.

Hoolimata sellest, et lõppkasutajaid oli instrueeritud, et tegemist on vaid omavahel seotud ning informatsiooniga eeltäidetud vaadetega, nö piltidega, tekkis neil reaalse rakenduse kasutuse

kogemus ning nende jaoks jäi suhteliselt arusaamatuks, miks prototüübist reaalse rakenduseni jõudmiseks kulub veel üks aasta aega.

Webmedia arendusmetoodika parendamise meeskonnale oli aga selge, et täisfunktsionaalne prototüüp on ainuvõimalik tulevik.

Täisfunktsionaalne prototüüp on keskkond, mis võimaldab anda lõppkasutajale reaalset kasutuskogemust loodava tarkvaraga, ilma seda tarkvara valmis programmeerimata.

Pelgalt HTML-i lehtede omavahel sidumine on ebaefektiivne, mistõttu tekkis vältimatu vajadus prototüübi mootori järele ja arendati oma spetsiaalselt prototüüpimiseks mõeldud sisuhaldussüsteemi (*CMS - Content Management System*). Aastaks 2013 on sellest olemas juba 4. versioon ning **täisfunktsionaalne prototüüp on kohustuslik iga uue Webmedia arendusprojekti juures**.

3.3.UIG (*user interface guidelines*) olulisus

Webmedia analüüsiprotsessi peamiseks paradigma muutuseks oli liikumine tekstidokumentide genereerimiselt visualiseerimisele alates nõuete analüüsist ning lõpetades täisfunktsionaalse prototüübiga, mis annab lõppkasutajale reaalse rakenduse kasutuskogemuse enne programmeerimise faasi algust. Need sammud vähendasid oluliselt analüüsifaasis tehtavaid vigu ning tõstsid arendusprotsessi efektiivsust.

Arendusefektiivsus ning lahenduste realiseerimise kiirus kasvas aga veelgi, kui analüüsiprotsessi sai sisse viidud kolmas oluline täiendus:

Rakenduse visuaalsel prototüüpimisel on kohustuslik kasutada eeldefineeritud kasutajaliidese elemente ning kasutussuuniseid.

Inim(kasutaja)liidese juhendid (HIG või UIG) on tarkvaraarenduse dokumendid, mis kehtestavad rakenduste arendajatele reeglid, kuidas visuaalselt komponente üles ehitada ning millises situatsioonis vastavat ekraanielementi kasutada. Nende dokumentide eesmärgiks on rakenduste kasutajakogemuse paremaks tegemine, muutes seda intuitiivsemaks, lihtsasti õpitavaks ja ühesuguseks [Bro1].

Lisaks lõppkasutaja võitudele on UIG kohustuslikust kasutamisest abi Täitjale. UIG sätestab lõpliku hulga tüüpkomponentidest ja elementidest, millest tuleb rakendus kokku panna ja see toob arendusprotsessi efektiivsusele kaasa kaks positiivset ilmingut:

- UIG annab kasutusraamid, mille sees tegutsemine tõstab analüüsiprotsessi efektiivsust. Näiteks otsingud. Kui on kokku lepitud, et terve rakenduse raames kasutatakse filtriga nimekirja tüüpi otsingut, siis on see sama kõikide objektide juures. Samuti sunnib UIG raamistik otsima standardsemaid lahendusi keerukamatele kasutussituatsioonidele. Tellija on tihti sunnitud mugavustsoonist välja tulema ning keerukat protsessi lihtsustama.

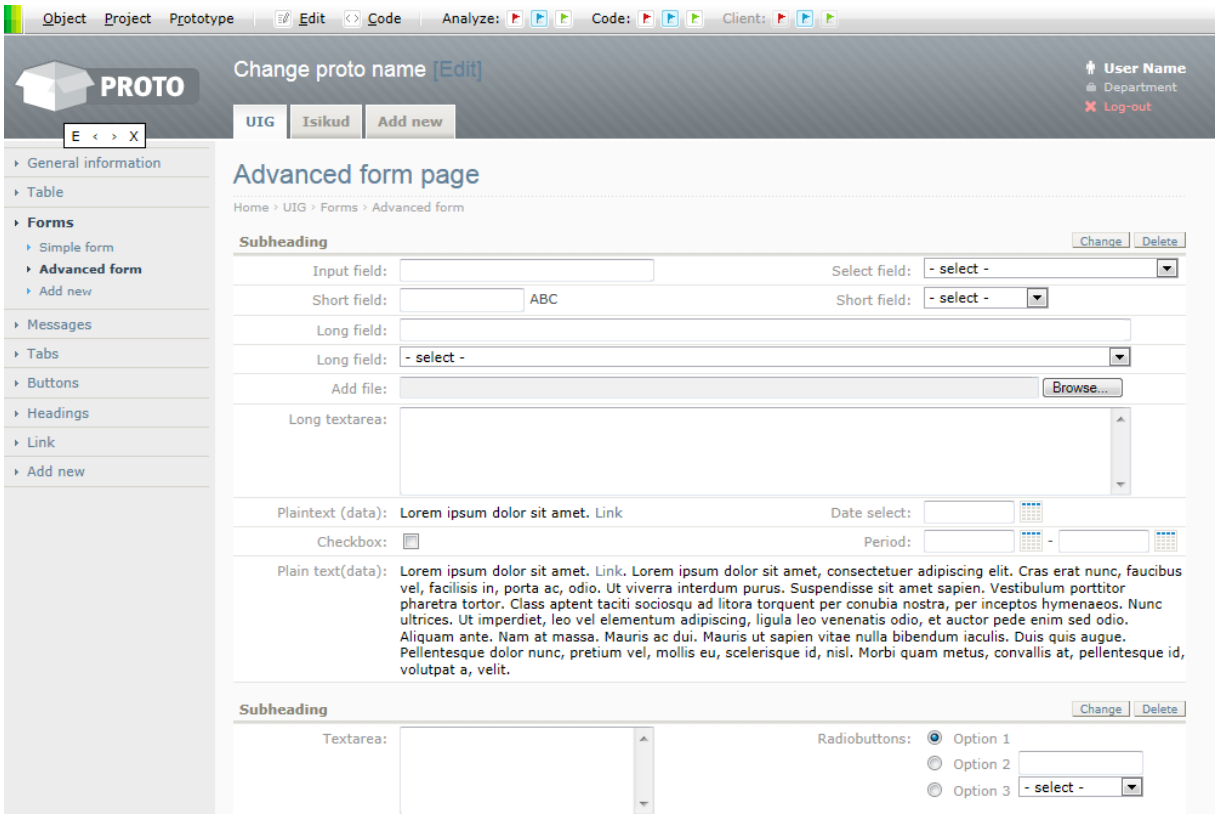
- Standardsete UIG elementide kasutamine standardiseerib ja kiirendab hilisemat programmeerimist. Kui komponendid ja elemendid oma loogikalt on üle rakenduse ühetaolised, tõstab see arenduskiirust ja võimaldab ka mitte nii kogenud tööjõu kasutamist.

Protsessiliselt on analüütikul iga projekti alguses võimalik kasutada prototüübi mootoris vaikeväärtustega (*default*) projekti, kus on ka enimkasutatavad ekraanielemendid realiseeritud. Hetkel kui kasutajaliidese analüütikud ja disainerid said valmis enda osaga projektist, muutsid nad UIG elemendid vastavaks kokkulepitud disainile ning rakendus hakkas välja nägema sellisel kujul nagu ta lõppversioonis peab olema.

Nimekiri kasutatavatest elemendi tüüpidest on näha Joonisel 13. Võib tunduda, et neid elemente on vaja disainida väga palju, siis reaalsuses on keskmiselt 2-3 erinevat tüüpi loendit, 2-3 nimekirja, TAB-i lahendus, 3-4 tüüpi nuppe, teated (veateated, hoiatused jne), tekstid (sh pealkirjad, sisutekstid) ja ongi kõik.

Autori kogemus: analüütikul või programmeerijal ei olnud lubatud iseseisvalt ilma disaineriteta uusi elemente juurde luua ning tegelikus elus erilahendusi väga vaja ei lähegi.

Joonis 13 Alusvormi (komponendi) näide prototüübi mootoris



3.4. Visuaalse prototüüpimise täiendavad positiivsed mõjud tarkvaraarenduse protsessile

Eelnevalt on kirjeldatud, kuidas visuaalne prototüüpimine Webmedia analüüsiprotsessi täiendas. Antud töös on mitmes kohas mainitud ning toodud näiteid, kuidas prototüüpimine mõjub positiivselt kogu tarkvara arendamise protsessile.

Nendest positiivsetest omadustest tahaks esile tõsta järgnevaid mõjusid:

1. Paraneb analüüsikvaliteet ning seetõttu rakenduse realiseerimise kiirus. Prototüüpimine sunnib nii Tellijat kui Täitjat detailsemalt läbi mõtlema loodavat lahendust.
2. Lõppkasutaja saab testida funktsionaalsust enne realisatsiooni ning teha endapoolseid muudatusettepanekuid.
3. Paraneb projektijuhtimise kvaliteet:
    a. Tellija ja Täitja saavad täpsemalt kokku leppida projekti arendusskoobis ning vahetulemites.
    b. Selgem ülevaade, kui palju projektist on valmis.
    c. Selgemalt eraldatud, mis on muudatus, mis on puudujääk ja mis on täitja viga.
4. Paraneb rakenduse realiseerimiseks kuluva aja ning töömahu hindamise kvaliteet, sh arusaam keerulistest ja lihtsatest komponentidest, mis võimaldavad Tellijal paremini hinnata tarkvarasse tehtavate investeeringute ja sealt saadava kasu mõttekust.

Alljärgnevalt ilmestatakse neid mõjusid detailsemate näidetega, et lisada veelgi kaalu visuaalse prototüüpimise kui kohustusliku sammu olulisusele tarkvara arendamise protsessis.

3.5.Analüüsi kvaliteedi paranemine

Komponendi spetsifitseerimisel on oluline juba analüüsi faasis pöörata tähelepanu komponendi toimimise funktsionaalsusele tervikuna. Kasutades ainult kirjeldavat meetodit ilma prototüüpimiseta, on suur oht mitte tajuda komponendi töötamist ja erijuhte tervikuna.

Toome järgnevalt lihtsustatud näite probleemi selgitamiseks.

Lähteülesanne:

Vaja on rakendust võlglaste haldamiseks.

Vajalik funktsionaalsus:

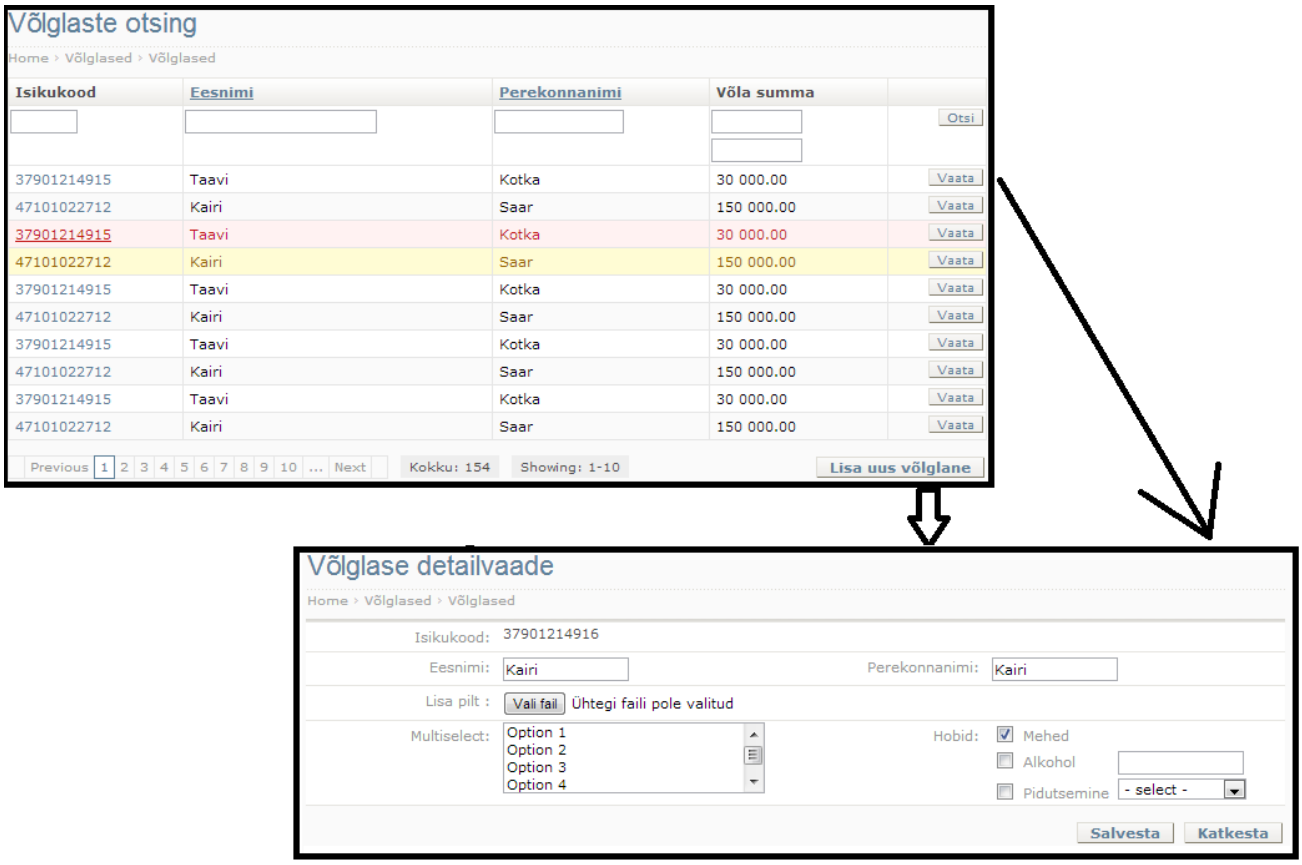
1. Peab saama lisada uut võlglast. Atribuudid:
    a. Isikukood
    b. Isiku eesnimi
    c. Isiku perekonnanimi
    d. Võla summa
    e. Staatus – kinnitamata/kinnitatud/edastatud/tühistatud
2. Peab saama otsida olemasolevaid võlglasi

Antud lähteülesanne on lahendatav kahe komponendiga:

- Loend võlglaste otsimiseks, sorteerimiseks ja filtreerimiseks
- Lihtvorm uute võlglaste lisamiseks või olemasolevate muutmiseks/tühistamiseks

Iseenesest sellest mõnerealisest lähteülesandest juba piisab, et alustada lahenduse realiseerimist ning puhtalt tervest mõistusest lähtuvalt võiks jõuda tulemuseni, mis visuaalsel kujul oleks järgmine (Joonis 14):

Joonis 14 Võlglaste nimekiri ja detailvaade



Ekstreemprogrammeerimise (*Extreme Programming* [Beck]) metoodikat kasutades olekski see juba esimese iteratsiooni tulemus, mida kliendiga läbi rääkima minna. Tulemuseni jõudmise aeg: 2 inimest x 4 tundi. Visuaalse prototüüpimise puhul on ülaltoodud Joonis 14 prototüüpimise vahetulem, mis annab võimaluse kliendiga täpsustada detaile komponentide omaduste kohta. Tulemuseni jõudmise aeg: 1 inimene x 0,5 tundi.

Ilmestamaks pisiasjade täpsustamise tähtsust, vaadeldakse siinkohal ainult loendit ehk siis Võlglaste nimekirja. Lähteülesandes polnud infot selle kohta, kui palju selliseid võlglasi üldse eksisteerib. Kas need kirjed peavad kõik ühele lehele ära mahtuma või peab jaotama lehekülgedeks. Kui viimane, siis kas kirjete arv, mis ühel lehel näha on, peab olema dünaamiliselt muudetav või mitte? Mille järgi on võlglased sorteeritud? Kas eespool vanemad või uuemad või sorteerida hoopis staatuse järgi?

Tuleb välja, et ilma reaalse lõppkasutajaga konsulteerimata ei olegi võimalik neid nüansse analüütikul ja seega ka programmeerijal oma tarkusest paika panna. Kui need nüansid täpsustamata jätta, näitab reaalne elu, et lõppkasutaja defineerib need testimise või siis juba toodangu keskkonnas tarkvara vigadena. See omakorda on Täitja vaatenurgast ebaõiglane, kuna tema jaoks on tegemist lisandunud informatsiooniga ja ei saa seetõttu olla eos viga, vaid on tegemist täiendusega (tasustamisele kuuluva tööga).

Oluline on siinkohal meelde tuletada varasemates peatükkides kirjeldatud eesmärki, et arendusprotsessis maksimaalse efektiivsuse saavutamiseks peab analüütik koos tellijaga pidevalt püüdlema täiusliku lähteülesande poole. Kui meil on tegemist ükskõik millise loendiga, siis tuleb alati täpsustada selle loendi omadusi kontrollküsimuste kaudu.

Alljärgnevalt on välja toodud valik kontrollküsimusi loendipõhistele komponentidele (NB! Küsimuste hulk ei ole lõplik) [AT08].

1. Kui suur on kirjete hulk, millele antud loend tugineb?
    a. **1 kuni 10** – tegemist on **väikeloendiga**, millele võib teha lahenduse, kus kõik kirjed päritakse alati ära ja näidatakse ka ekraanile.
    b. **11 kuni 100** – tegemist on **väike-kesk loendiga**, mille andmeid võib kõik ära pärida aga ekraanile näitamiseks tuleks jaotada portsudeks või varustada loend kerimisribaga.
    c. **101 kuni 1000** – tegemist on **kesk-suure loendiga**, mille andmeid ei tohiks korraga ära pärida ja kasutajale tuleks andmeid näidata portsude kaupa. Kui on tagatud kiire kerimine siis lubatav ka kerimisriba lisamine.
    d. **1001 kuni 10000** – tegemist on **suure loendiga**, mille andmeid ei tohi kindlasti ära pärida, tuleb mõelda ka päringu efektiivsusele ja kasutajale tuleks andmeid näidata portsude kaupa. Väga ebasoovitav on kasutada kerimisriba, mis võimaldab kerida andmeid kogu ulatuses. Kogu ulatuses kasutatavat kerimisriba võib kasutada vaid siis kui näidatavad andmed asuvad tarbimiskoha arvuti mälus või kõvakettal.
    e. **10001 ja rohkem** – tegemist on **väga suure loendiga**, mille andmeid ei tohi mingil tingimusel korraga ära pärida. Hoolikalt tuleb mõelda päringu efektiivsusele. Kasutajale tuleks andmeid näidata portsude kaupa või vaid ühte osa andmetest. Üle kõigi andmete toimiva kerimisriba kasutamine on keelatud kui andmed tulevad serverist. Kui tarbimiskoha arvuti mälust või kõvakettalt siis võib kasutada kerimisriba vaid siis kui ollakse veendunud, et see ka rahuldab kasutajaliidese kiiruslike nõuded.
2. Milline on filtervorm?
    a. **Üherealine filterriba** – loendi veergude kohal paiknevad sisestuslahtrid (tekstilahtrid, valikloendid), mis ei võta rohkem vertikaalset ruumi kui üks rida.
    b. **Mitmerealine filterriba –** reeglina kahe või kolme realine ja loendi veergude kohal paiknevad sisestuslahtrid, kus võimalik loendit kitsendada vabalt valitava ajavahemiku, mitmese valikuga valikloendi väärtustega jt. keerukamat andmesisestust võimaldavate ekraanielementide abil.
    c. **Filtervorm –** täismõõdus filtervorm, kus võimalik rakendada kõiki otsingutingimuste sisestamise ekraanielemente ja konstruktsioone.
3. Kas filtervorm säilitab oma olekut?
    a. Filtervorm mingit **olekut ei säilita** – peale päringu teostust filtervorm tühjeneb.
    b. Oleku säilitamine **päringu piires –** peale päringu teostust filtervorm ei tühjene. Andmed säilivad kuni lahkutakse loendivormilt.
    c. Oleku säilitamine **seansi piires –** sisestatud filtervormi andmed kestavad kuni kasutaja seansi lõpuni.
    d. Oleku säilitamine **andmebaasis** – sisestatud filtervormi andmed elavad üle ka kasutaja väljumise süsteemist.
4. Kas veerud on sorteeritavad?
    a. **Jäigad veerud** – ükski loendi veerg ei ole sorteeritav. Vaikimisi sorteerimine on paika pandud.
    b. **Lihtsalt sorteeritavad** – sorteeritavad on vaid veerud, mille sorteerimine on tehtav SQL „order by" operandi poolt.
    c. **Raskelt sorteeritavad** – sorteeritavad on lihtsalt sorteeritavad ja ka veerud mille järgi sorteerimine nõuab keerukamat loogikat kui see on SQL „order by" operandi poolt tehtav.
5. Kas veerud säilitavad oma olekut? – vastused on samad nagu filtervormi oleku säilimisel. Mida pikemaajalisem säilimine, seda töömahukam on seda teha.
6. Kas veerud on ümbertõstetavad?

    a. **Ei ole** – veergude järjestus pole ümber tõstetav. Tavaline veebirakendustele.
    b. **Jah on** – veerud on hiirega ümber tõstetavad.
7. Kas veergude ümber tõstmine säilitab olekut? – vastused on analoogsed filtervormi olekutele.
8. Kas veerud on peidetavad?
    a. **Ei ole** – veergude peitmine ei ole võimalik.
    b. **Jah on** – veerud on hiire paremklõpsust avanevast kohtmenüüst peidetavad või peidetavad mingi spetsiaalse tegevuse abil. Näiteks märkeruut veeru päisel.
9. Kas veergude peitmine säilitab oma olekut? – vastused on analoogsed filtervormi olekutele.
10. Kas veergude laius on muudetav?
    a. **Ei ole** – veerud on fikseeritud laiusega. Tüüpiline veebirakendustes.
    b. **Jah on** – veergude laius on hiirega muudetav.
11. Kas laiuse muutus säilitab oma olekut? – vastused on analoogsed filtervormi olekutele.
12. Kas filtervormi kujundus (lahtri suurused, nende nähtavus jt) on vastasmõjus loendi veergude kujundusega (ümber tõstmine, peitmine, laiuse muutmine)?
    a. **Vastasmõju ei ole** – kasutaja kohtkujundamine loendi veergudega ei mõjuta filtervormi.
    b. **Vastasmõju on olemas** – kasutaja kohtkujundamine asetab ümber, muudab suurusi või peidab ära filtervormi vastavaid elemente, mida kasutaja muutis loendivormil.
13. Kas loendil saab andmeid ka muuta?
    a. **Ei saa** – loend on mõeldud ainult lugemiseks.
    b. **Jah saab** – loendi andmed on loendil muudetavad. Muudatuste salvestamine toimub nupu „Salvesta” abil.
14. Kui loendi andmed on muudetavad, kas korraga saab muuta ühte rida või saab muudatusi teha paljudesse (kõikidesse ridadesse)?
    a. **Vaid üks rida** – korraga saab muuta vaid ühte rida. Muutmise algatamiseks tuleb rida välja valida või klõpsata rea kõrval oleval nupul „Muuda”.
    b. **Palju ridu** – korraga saab muuta kõiki ridu. Muudatuste jõustumiseks on all nupp „Salvesta”.
15. Kuidas toimub uue rea sisestamine?
    a. **Uut rida sisestada ei saagi.**
    b. **Siirdumisega spetsiaalvormile** – loendivormil on nupp algatamaks uue rea sisestamist.
    c. **Uue rea sisestamine loendivormil** – loendivormil on nupp uue tühja rea tekitamiseks või tühi rida on juba vaikimisi olemas. Uue rea andmete jõudmine baasi toimub läbi nupu „Salvesta”.
16. Kas loend uuendab oma andmeid ise?
    a. **Ei uuenda –** loend oma andmeid ei uuenda. Andmete uuendamiseks tuleb kasutajal klõpsata mõnda nuppu.
    b. **Jah uuendab –** loend uuendab oma andmeid teatud sagedusega või teatud sündmuste peale. Automaatne uuendamine tähendab seda, et filtervormis määratud (üks määrang on ka tühjus) kitsendusest läbi pääsenud, muutunud või uued andmed tulevad automaatselt ekraanile.
17. Mis tegevusi saab loendil teha?
    a. **Mitte midagi** – loendil ei saa peale andmete otsimise ja vaatamise midagi teha.
    b. **Siirdumine detailvormile** – loendi real topeltklõpsu tehes avaneb vastava rea kohane detailvorm.
    c. **Ridade vöötimine** – võimalik hiirega ajutiselt ära tähistad ridu, kas mingi kindla tegevuse algatamiseks või lihtsalt rea ära märkimiseks, näiteks arutluses oleva rea rõhutamiseks.
    d. **Rea kustutamine** – rea kustutamiseks tuleb rida märgistada ja klõpsata nuppu „Kustuta” või klõpsata rea lõpus olevat nuppu „Kustuta”.

    e. **Spetsiaaltegevused** – loendi reale või ridadele saab rakendada mingit rakenduse loogikast lähtuvat spetsiaalset tegevust.
18. Kas loendi andmeid/ridu saab väljastada mingisse teise vormingusse?
    a. **Ei saa** – loendi andmeid/ridu saab vaadata vaid ühes vormingus.
    b. **Trükivorming** – loendi andmeid/ridu saab väljastada trükivormingus, st välja filtreeritud ridu saab ühe-kahe hiireklõpsu abil välja trükkida. Trükivormingul on omakorda mitu võimalust. Tavaline CSS-il põhinev, spetsiaalne HTML või PDF. Töölaua rakenduse korral siis kas tugiraamistiku poolt kaasa tulev nii-öelda tasuta standardne vorming või spetsiaalne PDF või midagi sarnast.
    c. **Exceli vorming** – loendi andmeid/ridu saab eksportida Excelisse.
19. Kuidas on lahendatud mahukate andmete kuvamine loendi lahtritesse?
    a. **Mitu rida** – ühele reale mitte mahtuv tekst kuvatakse mitmele reale.
    b. **Lõigatakse maha** – lahtrisse mittemahtuv tekst lõigatakse maha - rohkem ei ole võimalik näha.
    c. **Kerimislahtrite abil** – pikad tekstid paigutatakse lahtritesse, mille sisu on võimalik kerida edasi-tagasi ja/või üles-alla. Kerimislahtrite all mõeldakse siinkohal ka tavalisi sisestuslahtreid.
    d. **Kohtsiltide abil** – lahtrite suurused on standardsed, kuid kui hiirega minna lahtri peale, siis avaneb selle kohale kohtsilt, mis sisaldab kogu teksti.
20. Kuidas jõustuvad muudatused?
    a. Nupp „**Salvesta**" – muudatused jõustuvad peale nupule „Salvesta" vajutamist. Tüüpiline veebirakenduse korral.
    b. **Fookuse liikumine** – muudatused jõuavad andmebaasi peale fookuse ära liikumist sisestusväljalt. Tüüpiline töölaua klientrakenduse korral.
21. **…**

See nimekiri ei ole lõplik. Näiteks ei sätesta see andmete kuvamise tingimusi, kui võlg on suurem kui 10 000 EUR-i; siis kuvatakse kogu kirje punasena jne. Nimekiri ilmestab, kui palju detaile ka kõige pisema komponendiga reaalselt tuleb läbi töötada, enne kui see piisavalt heaks saab.

Suurt osa neist küsimustest saab ka üldiste kasutajaliidese realiseerimise juhistega lihtsustada, kuid peatüki tuum oli demonstreerida, et ilma pildi abita ei ole Tellija projektimeeskond ega ka lõppkasutaja üldjuhul võimeline neile küsimustele vastama.

Peatüki alguses toodi võrdlus ekstreemprogrammeerimise ja visuaalse prototüüpimise vahel, kus Tellijaga töö ülevaatamiseks kulutati esimese metoodikaga 8 tundi ja teisega 1 tund. Vahe on muidugi selles, et ekstreemprogrammeerimist kasutades on olemas juba reaalselt töötav andmebaasiga lahendus, ning kui Tellija on selle lahendusega kohe rahul, ei ole mõtet prototüüpimise sammule aega kulutada. Visuaalse prototüübi vajalikkus on ikkagi otseses seoses Tellija ja Täitja meeskonna kogemustega. Kui üks pooltest ei ole pikaajalise tarkvara arendamise kogemusega ning ei oma selget nägemust loodavast lahendusest, on ka ekstreemprogrammeerimise ette mõttekas lisada visuaalse prototüüpimise samm, kuna hilisemad ümbertegemise ja täiendamise tööd kujunevad mahukaks.

3.6.Lõppkasutaja võimalus testida lahendust enne realisatsiooni

Tüüpiliseks nõudeks lähteülesandes on: *rakendus peab olema kiire ja mugav, st peab tegema võimalikult vähe klikke ja samas olema lihtsalt õpitav ja kasutajale arusaadav*.

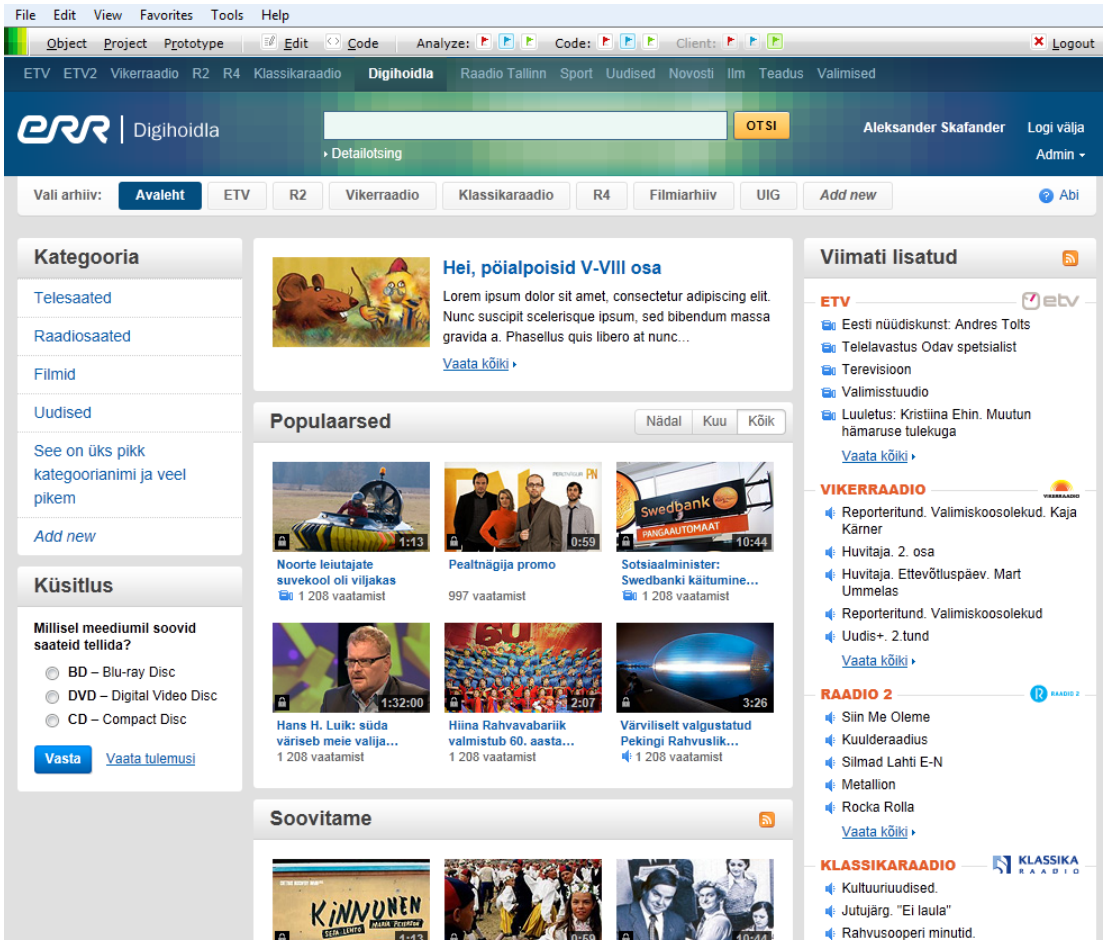
Kui 10 aastat tagasi peeti tarkvara funktsioonide rohkust oluliselt tähtsamaks tarkvara väljanägemisest, siis Apple-i uue tulemise ning „Less is more“ hüüdlause taustal on loodava rakenduse kasutajamugavuse (*usability*) ning kasutuskogemuse (*user experience*) analüüsimine ning erinevate lahenduste testimine muutunud üha tähtsamaks.

Ettevõte, kes oma ärist olulise osa ehitab üles Internetikanalile, pöörab üha rohkem tähelepanu protsessi maksimaalsele lihtsustamisele, kasutades näiteks A/B testimist jne. Samuti on just startup-maailmas eriti oluline, et Tellija saaks oma loodavat lahendust lõppkasutaja peal katsetada enne, kui ta üldse otsustab, kas investeerida tarkvara tootmisesse või mitte.

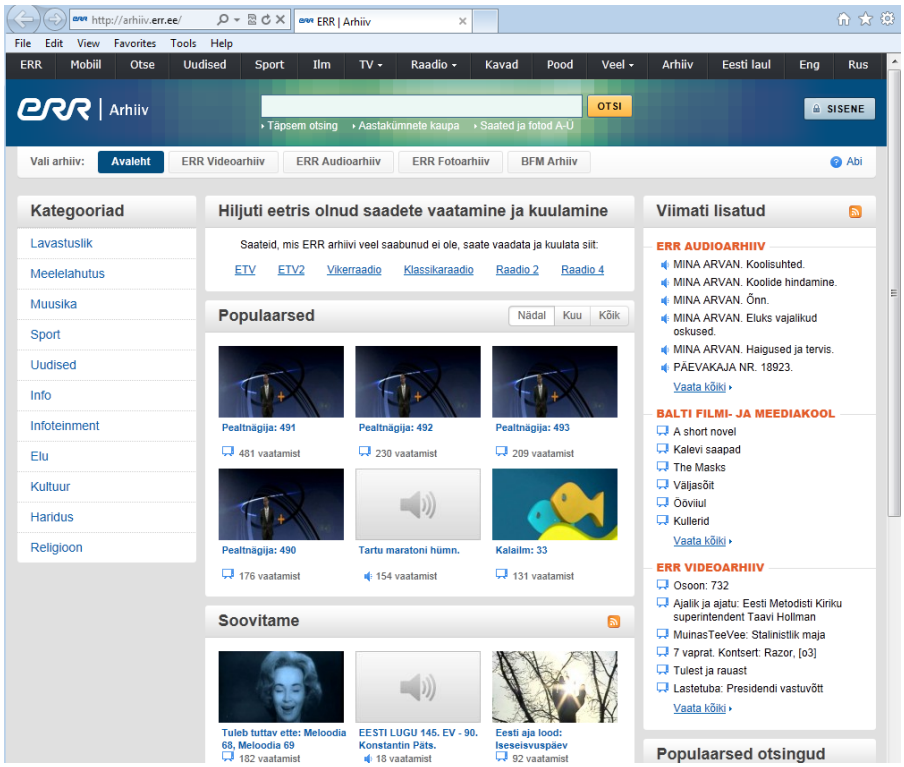
Visuaalne prototüüpimine ja prototüübimootori kaudu kasutuskogemuse andmine on siinjuures ainuvõimalik. Täitjal peab olema alus, millega istuda näiteks tulejoonel koos letiteenindajaga ning reaalselt analüüsida oma loomingut, st kas see ikka lihtsustab tööd või mitte. Alternatiiviks on teha valimid erinevatest vanuserühmadest ja testida avalikult kättesaadava teenuse toimimist. Seda mitte ainult funktsionaalsuse osas, vaid testitakse ekraanielementideni välja – ettevõtte logost, keskkonna värvidest otsingunupu suuruse ja asukohani välja.

Joonistel 15-16 on esitatud Eesti Rahvusringhäälingu arhiivikeskkonnad.

Joonis 15 Rahvusringhäälingu arhiivikeskkond Webmedia prototüübimootoris – september 2009



Joonis 16 http://arhiiv.err.ee - ERR-i arhiiv LIVE kasutuses (veebruar 2013).



Nende kahe pildi vahel on lihtne leida 10 erinevust. Prototüübis kasutatakse ERR-i logo juures sõna digihoidla, millest praeguseks on saanud sõna Arhiiv. Menüüpunkte on juurde tulnud ja sõnastus muutunud. Tänases keskkonnas on erinevalt prototüübist loobutud gallupist jne.

Need pildid peavadki olema erinevad, sest nende piltide vahe on 4 aastat. Iga keskkond, liiati veel avalik portaal, areneb edasi. Vastavalt rakenduse kasutusele muudetakse tekste, rõhuasetusi, komponentide paiknemist jne.

See, et need kaks pilti on ikka veel niivõrd sarnased, on tunnustus kasutajamugavuse disaineritele, kes on juba prototüüpimise ajal õige kontseptsiooni kokku saanud. Selle saavutamiseks korraldasid nad suuremal hulgal lõppkasutaja teste, ning kuna prototüübimootoris lingid töötavad, siis said nad lõppkasutajatele peegeldada reaalset kasutajakogemust.

Komponenti ei hakata programmeerima enne, kui prototüüp on valmis. Arvestades sellega, kui vähe tänane toodangkeskkond erineb toonasest prototüübist, võib järeldada, et täiendavaid tunde rakenduse ümbertegemiseks ei ole olnud vaja kulutada. Vähene ümbertegemise vajadus omakorda peegeldab seda, et ka finantsiliselt on see olnud Tellijale hea projekt.

3.7.Projektijuhtimise kvaliteedi paranemine

Visuaalne prototüüpimine lahendas Kollase kassi probleemi ehk projekti osapooltel on olemas ühine arusaam, mis realiseeritakse lõpptulemuseks. See võimaldab omakorda oluliselt tõsta projektijuhtimise kvaliteeti ning antud peatükk kirjeldab, kuidas prototüüpimise abil:

1. On selgemalt eraldatud, mis on muudatus, mis on puudujääk ja mis on täitja viga
2. Tellija ja täitja saavad täpsemalt kokku leppida projekti arendusskoobis ning vahetulemites
3. On selgem ülevaade, kus projektis sisuliste töödega ollakse

Igipõlised vaidlused kliendiga, kas ärivajaduste rahuldamiseks puuduv funktsionaalsus on täitja-poolne viga, mõlemapoolne töö tegemata jätmine või tellija lisandunud soov? Kuidas prototüübi abil neid situatsioone lahendada?

Toome siinkohal definitsiooni [Wiki]:

*A **software bug** is an error, flaw, mistake, failure, or fault in a computer program or system that produces an incorrect or unexpected result, or causes it to behave in unintended ways*. (Wikipedia, 2012) – Definitsioon on jäetud tõlkimata, kuna see rikuks tähenduse sügavust.

Tarkvara arendamise protsessi kulude juhtimisel tehakse Tellija ja Täitja maailmas vahet kolme tüüpi vigadel. Nad kõik on oma olemuselt vead ehk siis rakendus ei tööta nii, nagu vaja, kuid vahet tehakse vastutuse osas ehk siis piltlikult: kes vea parandamise eest maksab:

- **Tarkvaraviga** – rakenduse konkreetne funktsionaalsus ei vasta lähteülesandes püstitatule või muudele kliendi poolt kommunikeeritud ja kinnitatud soovidele. Täitja on süüdi ja peab oma kuludega vea kõrvaldama. Näiteks oli kokku lepitud, et Kohtuotsuste nimekiri on sorteeritud, uuemad kirjed eespool. Täitja ei olnud komponendile lisanud mitte mingisugust sorteerimist ja peab selle paranduse nüüd oma kuludega sisse viima.
- **Täiendus** – Tellija soovib lisada uut funktsionaalsust olemasolevale juba lõplikult kinnitatud lahendusele. Tegemist on Tellijapoolse uue sooviga, mistõttu ka muudatuse realiseerimise kulud peab kandma Tellija.
- **Puudujääk** – tarkvara on küll realiseeritud vastavalt lähteülesandele, kuid siiski ei toimi nii, nagu reaalses elus vaja ehk siis tegemist oli ebapiisava tehnilise ülesandega. Tüüpsituatsioon: Täitja väidab, et spetsifikatsioonis polnud ning seetõttu ei osanud ka aimata, et soovitud funktsionaalsus peaks vajalik olema. Täitja omakorda ei tulnud selle peale , et tema ärile niivõrd omane funktsionaalsus peaks olema detailselt lahti kirjutatud. Täpsustamine ja sellega koos sisse viidava muudatuse kulu läheb üldjuhul jagamisele. Kõikide arendustundide kompenseerimismudeli (*Time&material*) korral kannab kulud Tellija, kuna Täitjal ei ole olnud võimalik küsida raha funktsionaalsuse eest, mida ta ei teadnud, et peab seal olemas olema.

Spetsiaaltarkvara tootmises toimub enamikus projektides arvestus tunnipõhiselt, mistõttu on nii Tellija kui Täitja jaoks oluline, kes täienduste või vigade paranduse kulud peab katma. Tellija silmade jaoks lihtsad ülesanded (pane siia loendisse üks atribuut juurde, muuda siin sorteerimise reeglit, tõsta siin need väljad ümber jne) on kokkuvõttes Täitja jaoks kõik ajakadu ning võimalik rahaline kaotus.

Autori kogemuse pealt öelduna tuli isegi ülikvaliteetse analüüsi juures Tellijalt komponendi kohta keskmiselt 3-5 muudatusettepanekut. Keskmise suurusega infosüsteem Eestis, mida tervikuna tellitakse, koosneb 80-100 komponendist. Arvestades optimistlikult keskmiseks muudatuse parandamise ajaks koos täiendava testimisega 2 tundi, on kulu Täitjale: 5vigax100komponentix2tundi = 1000 tundi. Arvestades tootmise omahinna miinimumiks 25 EUR/h,

on otsene kulu täitjale 25 000 EUR-i. Kuna vigade parandamise asemel oleks nende tundide ajal saanud tegeleda tootmisega (raha teenimisega), on kulu Täitjale isegi suurem, kuna väljamüügihind on omahinnast ca 50% kõrgem.

Seega, kui rakendus ei tööta soovitud funktsionaalsusele vastavalt, on loomulik, et Tellija soovib maksimaalselt näidata vigu Täitjapoolsete eksimustena ning Täitjal on vastupidine motivatsioon. Reaalses elus näeb see välja pingelise projektikoosolekuna, kus vaieldakse iga vähegi piiripealse vea puhul, kas see on viga, puudujääk või täiendus. Tellija kasutab retoorikat: „Me ju rääkisime sellest, ma ei tulnud selle pealegi, et te seda analüüsidokumenti kirjutada ei suutnud.“ Või teine klassika: “Mis mõttes see funktsionaalsus on puudu, tegemist on ju elementaarsusega“.

Olgu siinkohal võetud näiteks isikuandmete atribuut Perekonnaseis. Täitja realiseeris lahenduse, kus Perekonnaseis on atribuut, kus säilitatakse ainult hetkel kehtivat seisundit. Sõltuvalt Tellija äriloogikast ilmnes hilisemal testimisel, et ikkagi on vaja teada ka isiku varasemat perekonnaseisu ajalugu. Üks pool ei osanud rääkida, teine pool ei osanud küsida. Puudujääk, mis tuleb parandada, kuid kelle kanda on parandamisega seotud kulud?

Visuaalse prototüüpimise kasutamine on näidanud, et selliste klassikaliste vaidluste lahendamiseks on prototüüpimine suhteliselt asendamatu abimees. Kui Tellija ja Täitja on kokku leppinud, et vealiigituse üheks alusdokumendiks saab olla ka prototüüp, väheneb vaidluste hulk oluliselt.

Kui soovitud funktsionaalsus oli kinnitatud prototüübis olemas, aga lõppvariandis Täitja poolt erinevalt või ebapiisavalt lahendatud, on tegemist veaga ja Täitja peab oma kuludega selle katma.

Kui klient soovib kinnitatud prototüübist erinevat lahendust, on tegemist täiendusega, mille parandamise kulud kannab Tellija.

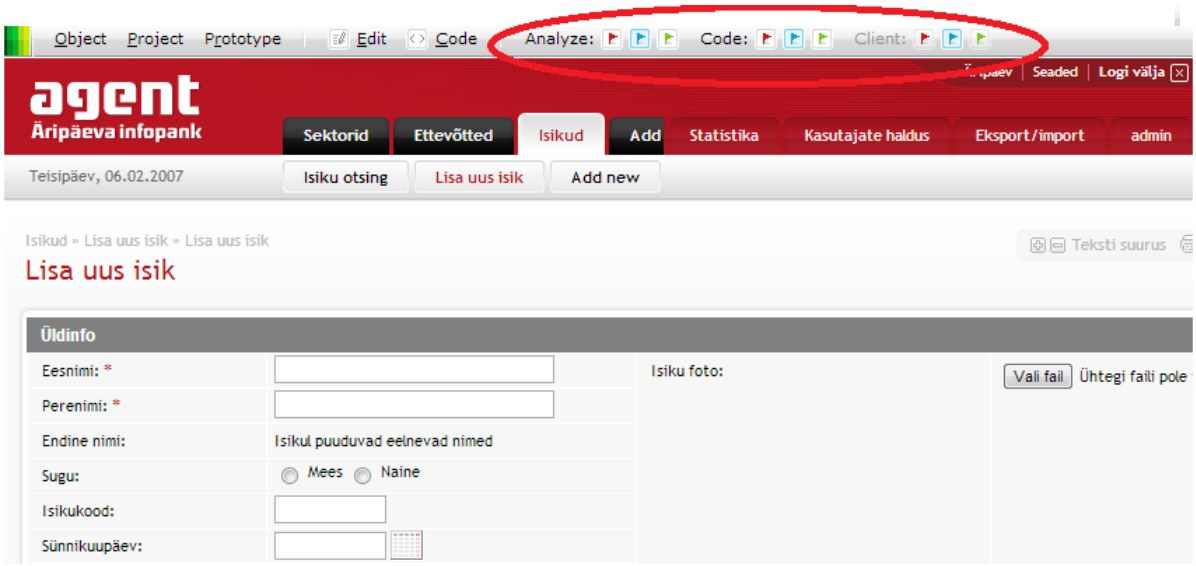
Puudujääke esineb kindlasti ka prototüübi arendusprotsessi kaasamise korral, kuid nende arv on kordades väiksem, võrreldes olukorraga, kui jätta prototüüp tegemata ja toetuda ainult tekstilisele spetsifikatsioonile. Olgu siinkohal veel kord toonitatud, et prototüübist üksi ei piisa, vaid alati on vajalik iga komponendi kohta nii prototüüp kui ka tema toimimise kirjeldus tehnilises ülesandes.

Veel on oluline märkida, et prototüüp aitab komponente ja nende funktsionaalsust paremini defineerida, kuid hilisemate vaidluste ärahoidmiseks on vaja see arendusprotsessis alati Tellijaga üle vaadata ning vahetulemused kinnitada. Ilma kinnitamiseta ei saa sellele hilisemates vaidlustes toetuda kui ametlikule kokkuleppele.

Webmedia arendusprotsessis proovis töö autor juurutada ka komponendipõhist prototüübi kinnitamist, kus Tellija ja Täitja igapäevaselt vaatavad prototüüpi üle, kuid süsteem ei hakanud tööle tänu Tellijapoolse ajaressursi vähesusele. Eesti tarkvaraarenduse projektides ei ole üldjuhul Tellija esindajatel võimalik igapäevaselt Täitjaga kaasa mõelda, mistõttu hoolimata pingutustest enamikus projektides komponendipõhist võimalust kasutama ei hakatud.

Idee oli lipukeste süsteemis, kus iga komponendi juures sai nii Täitja analüütik kui Tellija märkida, kas nende meelest on komponent valmis või mitte (vt Joonis 17). Lisaks sai programmeerija hiljem märkida, kas tal on komponent lõplikult realiseeritud või mitte. Vaikimisi olid lipud sinised ehk siis neutraalsed. Roheline tähendas valmisolekut kinnitamiseks ning punane tagasilükkamist (tagasilükkamisel tuli sisestada ka kohustuslik kommentaar).

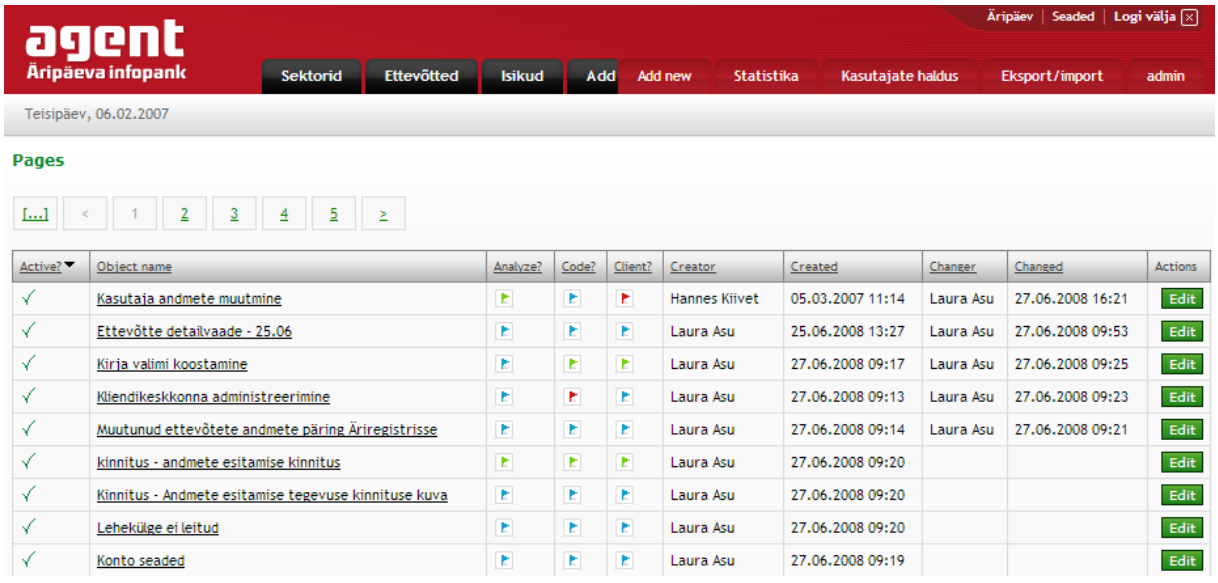
Joonis 17 Komponentide realisatsiooniprotsessi juhtimine lippudega



See süsteem pidi idee poolest andma hea ülevaate nii tellijale kui ka projektimeeskonnale, kui palju loodavast funktsionaalsusest on erinevates faasides kliendi poolt üle vaadatud ning aktsepteeritud. Reaalses elus aga toimus funktsionaalsuste ülevaatamine ja dokumentatsiooni, sh prototüübi kinnitamine projektikoosolekutel ning seisukohad fikseeriti koosoleku protokolliga.

Kui vaadata allolevat joonist (Joonis 18), siis protomootori keskkonnas oli iga projekti kohta võimalik saada edenemise kohta ülevaadet, kuid arendusprojekti juhtimisel osutus see liigkeeruliseks lähenemiseks:

Joonis 18 Prototüübi lehekülgede (vaadete) nimekiri

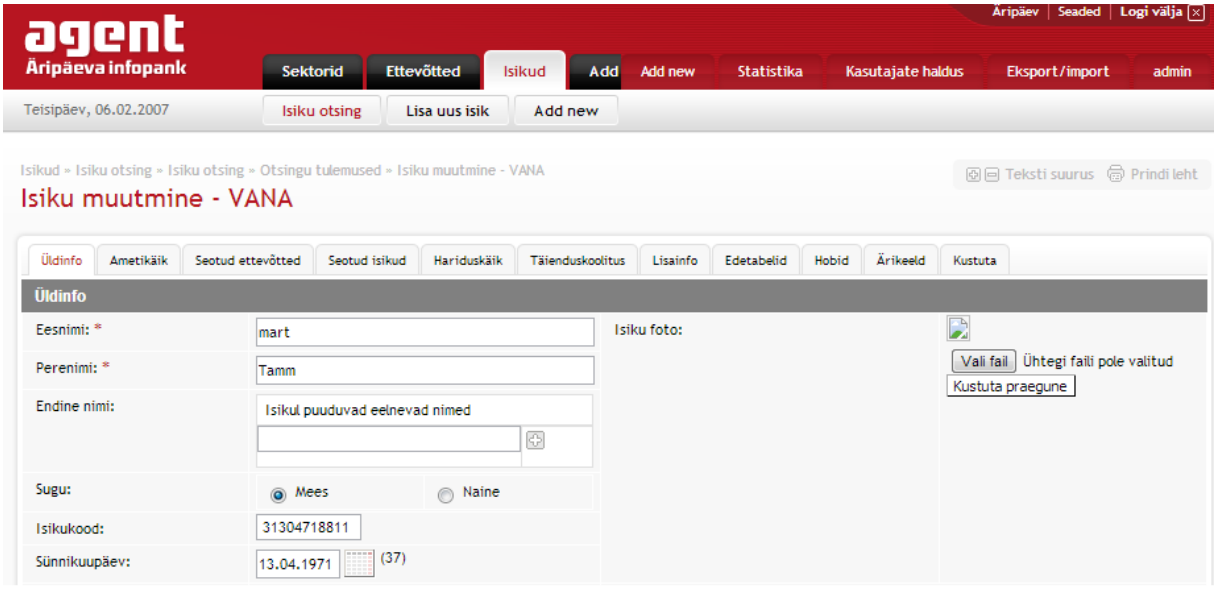


Küll aga aitas see vaade Tellijal ning Täitja projektimeeskonnal adekvaatsemalt hinnata projekti edenemist: kui keerulisemate rakenduste kallal töötab korraga mitu analüütikut, programmeerijat ja testijat, on väga raske koos hoida suurt pilti, millest loodav lahendus koosneb, kas kliendi käest on saadud kinnitused, kui palju on programmeeritud jne.

Kuna analüütikute poolt loodud iga uus komponent kajastus ka siin nimekirjas, sai projektijuht endale alusnimekirja, mida kliendiga läbi vaatamas ja kinnitamas käia. Samuti andis see projektijuhile võimaluse programmeerimise tööülesandeid mõistlikult tükeldada ning hiljem testijatele kontrollvõimaluse, kui palju reaalselt ikkagi komponendist on valminud.

Töö autori isiklik kogemus näitas, et programmeerija ühe tööülesande maht ei tohi olla suurem kui 32 tundi. Suuremate kui 4 tööpäevaste ülesannete puhul tekkis sagedamini olukord, kus komponendi keerukusest tulenevalt jäi programmeerija lahenduse otsimisel hätta ning selle asemel, et soovitud lahenduse keerukust vähendada või teistelt abi paluda, jätkati iseseisvat pusimist, suutmata ennustada tuleviku töömahte.

Joonis 19 Komplekskomponendi näide



Joonis 19 on ilmekas näide, kui palju erineb kasutusloo (*use-case*) põhine analüüs komponent-põhisest analüüsist. *Use-case* põhises analüüsis on tegemist „Isikuandmete muutmisega“. Komponendipõhise analüüsi juures on iga sakk (*Tab*) vähemalt üks komponent, mille minimaalne realiseerimise ja testimise töömaht on 6-8 tundi. Tööde nimekirjas näeb see välja märkimisväärselt erinev:

Use-case põhine analüüs	**Komponendi põhine analüüs**
Isikuandmete muutmine	Isiku üldinfo muutmine
	Isiku ametikäigu muutmine
	Seotud ettevõtete muutmine
	Seotud isikute muutmine
	Hariduskäik
	Täiendkoolitused
	Isiku Lisainfo muutmine
	Edetabelid
	Hobid
	Ärikeelud
	Isiku andmete arhiveerimine/kustutamine

Kui tarkvaraarendusega igapäevaselt mitte leiba teeniv inimene loeb vasakut veergu, siis talle vaevalt tundub, et tegemist võib olla vähemalt pooleteisenädalase tööga. Samas kui parempoolse nimekirja

puhul on 60 tunnine realiseerimise ja testimise aeg paljalt nimekirja vaadates ilmselgelt põhjendatud.

Täisfunktsionaalne prototüüp tõstab oluliselt ka projekti progressi kontrollimise võimekust. Kuna kollase kassi probleem on lahendatud ja kõigil projekti osapooltel on selge arusaam, milline peab lõpplahendus välja nägema, saavad kõik projekti osalised füüsiliselt kontrollida, kas programmeerijate poolt antud vahetulem suures pildis vastab Tellija ootustele või on realiseeritud mis iganes põhjustel ainult osaliselt.

Samuti annab see võimaluse paremini planeerida Tellijal ja Täitjal komponentide realiseerimise järjekorda sõltuvalt siis kas testimise korraldamise vajadusest või keerukusest lähtuvalt vms.

3.8. Rakenduse realiseerimiseks kuluva aja ning töömahu hindamise kvaliteedi paranemine

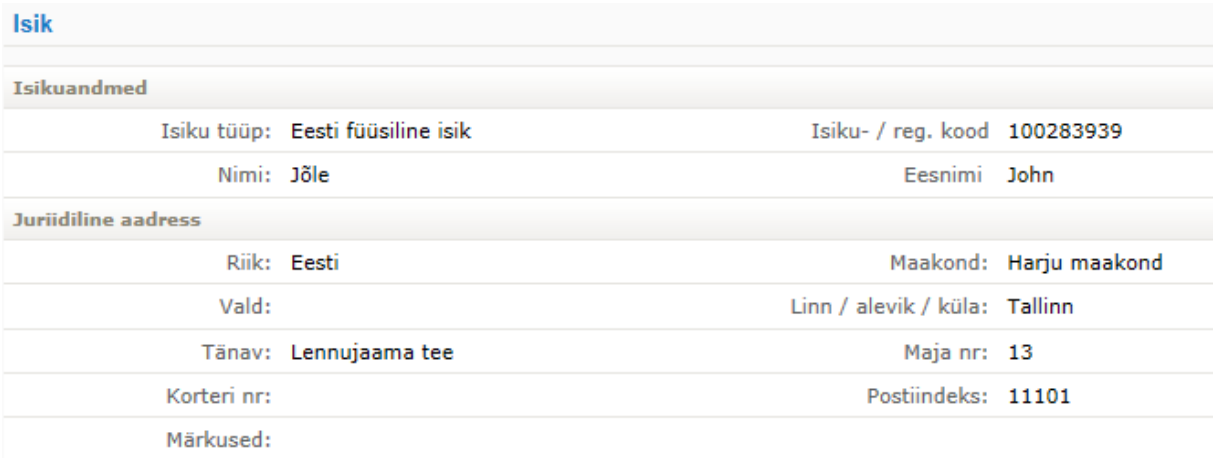
Kui palju tarkvara tootmine maksma läheb? Mis on lihtne ja mis on keeruline komponent? Kuidas on see seotud erinevate hindamismetoodikatega?

Prototüüpimise positiivse mõjuna väärib eraldi esile tõstmist fakt, kuidas tänu visualiseerimisele muutusid oluliselt täpsemaks programmeerijate mahuhinnangud. Prototüübi pealt on palju täpsemalt võimalik hinnata, kui palju ühe või teise komponendi realiseerimiseks aega kulub.

Kui hankedokumendis on näiteks kirjas nõue: *Kasutaja peab saama vaadata isikuandmeid*, siis reaalsuses võib see tähendada programmeerijale 4 tundi tööd (Näide 1), aga võib tähendada ka 60 tundi tööd (Näide 2).

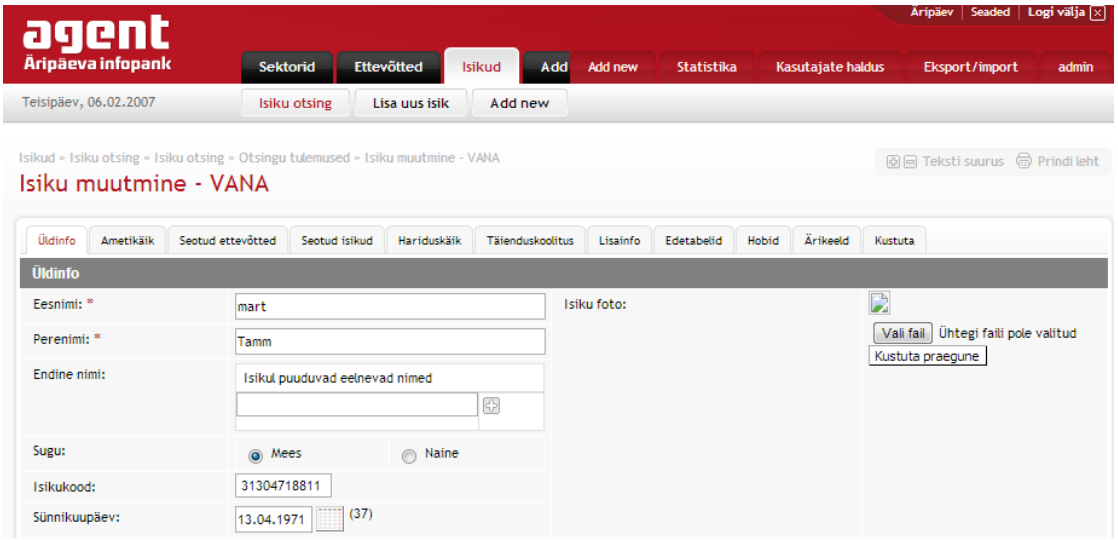
Näide 1. Alljärgnev komponent on nn lihtne vorm, mille realiseerimise töömaht on ca 4-6 tundi

Joonis 20 Lihtvormi kui komponendi näide



Näide 2. Isikuandmete vaatamise/muutmise alla kuuluva kõik sakid (*TAB*-id): Üldinfo, Ametikäik, Seotud ettevõtted jne. Tegemist on komponentide kogumiga ning selle realisatsioon võib aega võtta rohkem kui 60 tundi.

Joonis 21 Komplekskomponendi näide



Ilma visuaalse prototüübita on raske kasutada ka erinevaid hindamismetoodikaid. Näiteks väljavõte Andrus Tamboomi poolt koostatud raskusastmete metoodikast, kus iga komponenti hinnatakse vastavalt tema baasraskusastet keerukusega korrutades.

Näide 3. Lepingute ja arvete käsitlus, kus ülemloendis on lepingud, ülema detailides lepingu detailid ja arvete loend ning alama detailides on arve detailid (arve detailid ei sisalda arveridu).

tüüp: **üks-mitmele liitvorm**
raskusaste: **Keskmine (realiseerimine võtab aega kuni 16 tundi)**

Nr.	Mõõde	Kuulub sisse	Muudatus	Mõju
1.1	Olemite üks-ühele seoseid	kuni 4	kuni 6	+9%
1.2	Olemite üks-mitmele seoseid	1	0	-12%
1.3	Olemite mitu-mitmele seoseid	0	pole	0
2	Veebilehekülgede arv	kuni 3	4 kuni 5	+9%
3	Andmeelementide hulk	kuni 45	pole	0
4	Lihtklassifikaatoreid	kuni 4	5 kuni 6	+9%
5	Hierarhilisi klassifikaatoreid	kuni 1	kuni 2	+9%
6	Tegevused	kõik tegevused	ainult vaatamine	-13%
7	Väljundvorminguid	HTML	CSV (välistav)või XML	+9%
8	Kasutajaõiguste granulaarsus	kuni 6 privileegi	pole	0
9	Ärireeglid	JavaScript lihtne Java/PL	pole	0
10	Välised liidesed	puudub	pole	0
11	Korduvkasutus	ei	jah	+5%
12	Kasutaja tüüpe	1	pole	0

Ilma prototüübita on seda metoodikat võimatu kasutada. Prototüübi olemasolu korral komponente ükshaaval läbi käies ja hinnates saab aga projekti kogumahu hinnangud oluliselt täpsemaks kui nad muidu „mineviku kogemuse“ ja „kõhutunde“ pealt tuleksid.

4. Kokkuvõte

Iga spetsiaaltarkvara tootva ettevõtte eesmärgiks on teenida kasumit. Pikaaegse kasumlikkuse eelduseks nendes ettevõtetes on efektiivne arendusprotsess, mille parendamisega pidevalt tegeletakse.

Põhjuseid, mis arendusprotsessis ebaefektiivsust tekitavad, on mitmeid, kuid üheks olulisemaks juurpõhjuseks on Tellija ja Täitja meeskondade erinev arusaam ning suutmatus spetsifikatsiooni tasandil kokku leppida, milline peaks välja nägema projekti lõpptulemus. Hellitavalt sai hakatud seda probleemi kutsuma Kollase kassi probleemiks.

Lahenduse sellele probleemile leidis antud töö autor juba 2001. aastal läbi visualiseerimise, muutes kasutajaliidese prototüüpimise kohustuslikuks sammuks tarkvara tootmise protsessis. Lisaks ühise arusaama tekkimisele lõpplahendusest kaasnesid tänu visualiseerimisele arendusprotsessis ka muud positiivsed nähtused:

- Paranes Tellija kaasamõtlemise võime
- Paranes süsteemianalüüsi kvaliteet detailide osas
- Lõppkasutaja sai testida lahenduse sobivust enne programmeerimistööde algust (täisfunktsionaalne prototüüp)
- Paranes projektijuhtimise kvaliteet
- Muutusid täpsemaks programmeerimise töö mahuhinnangud
- …

Kõik see mõjutas arendusefektiivsust. Programmeerijad pidid üha vähem oma tööd ümber tegema või hilisema testimise käigus vastavalt lõppkasutajate soovidele täiendama. See omakorda tõstis nende isiklikku motivatsiooni, kuna said keskenduda uue loomisele, mitte vanade aukude lappimisele.

Kohustusliku kasutajaliidese prototüüpimise sisseviimisega olid väga rahul ka Tellijad, kelle jaoks muutusid oluliselt lihtsamaks uue lahenduse nõuete kogumise ja süsteemianalüüsi etapid. Visualiseeritud ekraanivaadetega töötamine lihtsustab tavapärase tekstilise spetsifitseerimisega võrreldes oluliselt arusaamist loodavast lõpptulemusest ning parandab Tellija kaasamõtlemise võimet pisidetailide täpsustamise tasandini.

Täisfunktsionaalne prototüüp ning visualiseerimise abil analüüsidetailide täpsustamine oli pea 10 aastat Webmedia konkurentsieelis. Hoolimata *software mockupingu, wireframingu, stroyboardingu* ja muude kollase kassi probleemi lahendavate metoodikate aktiivsemast kasutuselevõtmisest kogu maailmas viimastel aastatel, on Webmedia prototüübimootor jätkuvalt eriline, pakkudes näiteks Balsamiqust (laialt levinud skitseerimise vahend) kiiremat ning (NB!) odavamat visualiseerimist, rääkimata lõppkasutajale reaalse, see tähendab üks-ühele samase kasutuskogemuse andmisest.

Specification vs Visual Prototyping: Solution to Yellow Cat Problem

Bachelor Thesis (6 ECTS)

Taavi Kotka

Summary

The goal of any vendor of specialized software is to earn a profit. A pre-requisite for long-term profitability in such enterprises is an efficient development process that is constantly being improved.

There are many reasons why development processes become inefficient, but one of the most important root causes is the difference in understanding between the Client and Developer teams, and the inability to agree, on the specification level, as to how the project's final result should look. This issue got the pet name of the Yellow Cat Problem.

The author of this work has found a solution to this problem back in 2001, in the form of visualizing, and making the prototyping of the user interface a mandatory step in the software production process. In addition to the creation of a common understanding of the final solution, visualization also brought other positive effects to the development process:

- Improvement of the Client's ability to think along
- Improvement in the quality of system analysis, especially in small details
- The end user could test the suitability of the solution before programming even started (fully functional prototype)
- Improvement in the quality of project management
- Volume estimates for the programming work became more accurate

All of this had an impact on development efficiency. Programmers had to spend less time re-doing their work or expanding it based on end-user wishes that only came out in late phase testing. This in turn increased their personal motivation, as they managed to focus on creating the new, not patching holes in the old.

Clients were also very happy with the introduction of mandatory user interface prototyping, as for them it made the collection of requirements and system analysis stages of the new solution's development much easier. Compared to regular text specifications, working with visualized screen views significantly simplifies the understanding of the final product being created, and improves the Client's ability to think along down to the level of clarifying tiny details.

A fully functional prototype and clarification of analysis details through visualization have been competitive advantages for Webmedia (Nortal) for nearly 10 years. Despite the increasingly active global usage of software mockuping, wireframing, storyboarding and other methodologies for resolving the Yellow Cat Problem, Webmedia's prototype engine is still special, offering visualization that is faster and (note!) cheaper than, say, Balsamiq (a widespread sketching tool). Not to mention that it gives the end user an identical, one-to-one experience of the real thing.

Tsiteeritud teosed

[AT08] Andrus Tamboom, T. K. (2008). Webmediai analüüsijuhendid. rmt: T. K. Andrus Tamboom, *Webmediai analüüsijuhendid* (lk 214). Tallinn.

[Boeh] Barry W. Boehm, T. E. (1995). *PROTOTYPING VS. SPECIFYING: A MULTI-PROJECT EXPERIMENT*. Allikas: http://pdf.aminer.org/000/361/435/prototyping_vs_specifying_a_multi_project_experiment.pdf

[Beck] Beck, K. (2004). *Extreme Programming Explained: Embrace Change .*

[Broo] Brooks, F. (1975). *The Mythical Man-Month.*

[Brow] Brown, C. M. (1998). *Human-Computer Interaction Design Guidelines.*

[Cock] Cockburn, A. (2000). *Agile Software Development.*

[IBMI] IBM Institute, S. S. (kuupäev puudub). Allikas: http://www.isixsigma.com/industries/software-it/defect-prevention-reducing-costs-and-enhancing-quality/

[Inve] Invest, P. (2012). *Baltic ICT market news autumn 2012.* Vilnius: Prime Invest.

[ITPR] IT PRO Portal, S. (14. June 2010. a.). *IT PRO Portal*. Kasutamise kuupäev: 2. February 2013. a., allikas IT PRO Portal: http://www.itproportal.com/2010/06/14/top-ten-software-development-risks/

[Math] Lars Mathiassen, T. S. (1995). *Prototyping and Specifying: Principles and Practices of a Mixed Approach*. Kasutamise kuupäev: 20. February 2013. a., allikas http://iris.cs.aau.dk/tl_files/volumes/volume07/no1/03_mathiassen_p55-72.pdf

[Parn] Parnas, D. a. (1986). *A Rational Design Process: How and Why to Fake IT.*

[RUPT] RUP team, I. (2013). *IBM*. Allikas: Rational Unified Process: http://www-01.ibm.com/software/awdtools/rup/

[Snai] Snailonbike. (23. May 2010. a.). *WDMP Championship – Agile vs Waterfall*. Allikas: snailonabike.wordpress.com: http://snailonabike.wordpress.com/tag/development/

[Wiki] Wikipedia. (May 2012. a.). *http://en.wikipedia.org/wiki/Software_bug*. Allikas: Wikipedia: http://en.wikipedia.org/wiki/Software_bug

[ÄP11] Äripäev. (2011). *Äripäeva IT TOP.* Tallinn: Äripäeva kirjastus.

[ÄP13] *Äriregister*. (February 2005-2013. a.). Allikas: https://ariregister.rik.ee

Lisa 1. Mõisted

C-tase (C-level) – üldistatud termin ettevõtete tippjuhtkonna ja otsustajate kohta (C-täht tuleb inglise keelsete ametinimetuste esitähest à la Chief Executive Officer (CEO), Chief Operations Officer (COO), Chief Financial Officer (CFO) jne).

Kasutajaliidese juhend (UIG – user interface guidelines or HIG – human interface guidelines) – reeglite ja elementide kogum, mis sätestavad loodava tarkvaralahenduse kasutusmugavuse reeglid, visuaalsed disainielemendid ja nende kasutamise reeglid.

Kasutuslugu (Use-case) – kirjeldab ühte kasutussituatsiooni, kuidas kasutaja (actor) rakendust kasutades saavutab soovitud eesmärgi. Eesmärgi saavutamiseks võib olla vajalik teha mitmeid samme ning kasutada erinevaid komponente.

Komponent – antud töös tarkvarapakett, veebiteenus, ressurss või moodul. Funktsioonide või andmete kogum, mis moodustavad tarkvara realiseerimise vaatest omaette loogilise terviku.

Komponendipõhine tarkvaraarendus (Component-based software engineering (CBSE)) – tarkvara arenduse haru, mis keskendub komponentide kui korduvkasutavate elementide realiseerimisele

Prototüüp – on varajane näidis või mudel, et testida loodava rakenduse (aplikatsiooni) või protsessi kontseptsiooni ning vastavust tellimusele.

Spetsiaaltarkvara (Custom software) – on tarkvara, mis on realiseeritud rätseptööna kindla kliendi (Tellija) vajadustest lähtuvalt.

Tehniline ülesanne – alusdokument komponendi programmeerimiseks. Analüütiku töö tulem, mille järgi programmeerija komponendi realiseerib. Detailsem mall ühest tehnilise ülesande näitest on esitatud Lisas 2.

Tellija – klient, tarkvara tellija. Antud töös kasutatakse üldistatud mõistena ehk siis Tellija on koondnimetus kogu meeskonnale, kes tarkvara hankimisega on seotud ning kelle organisatsiooni ärivajadustele vastavalt lahendust realiseeritakse.

Täitja – tootja, tarkvara realiseerija ja tarnija. Koondnimetus meeskonnale, kes on vastutav Tellija vajadustest lähtuva tarkvara tarnimise eest.

Visuaalne prototüüp – staatilise infoga vaadetest koosnev näidis, mille eesmärgiks on anda edasi loodava valmislahenduse idee ning ülesehitus. Visuaalne prototüüp on käesoleva töö põhiline element, mille vajalikkusest ning kasust tarkvara arendamise protsessis antud töös peamiselt käsitletakse.

Lisa 2. Kasutatud joonised

Lisa 3. Visuaalset väljundit omava komponendi tehnilise ülesande mall

Komponendi kood

Komponendi pealkiri

Kuupäev	Ver.	Selgitus	Autor

1. Lühikirjeldus

Komponendi otstarbe, funktsiooni, mõtte jne. mõne lauseline kirjeldus.

2. Ligipääsu regulatsioon

Privileeg või privileegide loetelu ligipääsuks komponendile. Samuti muud ligipääsu reguleerivad reeglid või nõuded.

3. Komponendi sisendparameetrid

Sisendparameetrite loetelu ja nende otstarbe selgitus. Kui sisendid puuduvad, siis jääb see peatükk ära.

4. Komponendi käivituse eeldus

Täiendavad eeltingimused, mis peavad olema täidetud, et komponent saaks käivituda. Kui eeltingimused puuduvad siis jääb see peatükk ära.

5. Komponendi kirjeldus

6. Visuaal

Viit prototüübile või väljavõte prototüübi ekraanipildist.

7. Seos andmebaasiga

Tuleb kirjeldada andmete lugemise ja salvestamise reeglid. Ideaalis on kõik seosed visuaali ja andmebaasi vahel täpselt ära näidatud.

8. Tegevused

Komponendil paiknevate nuppude ja tegevuslinkide toime kirjeldus.

9. Täiendavad reeglid

Kui on täiendavaid reegleid, mis vajavad rõhutamist, siis siin on selleks koht olemas.