Tallinna Tehnikaülikool
Infotehnoloogia teaduskond
Informaatikainstituut
Infosüsteemide õppetool

Meetmeid tarkvara hallatavuse parendamiseks

magistritöö

Üliõpilane	Raul Viigipuu
Üliõpilaskood	050249IABM
Juhendaja	lektor Karin Rava

Tallinn
2015

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

_____ _____

(kuupäev) (allkiri)

Annotatsioon

Magistritöö eesmärk on käsitleda erinevaid tarkvara arendatavust ja hallatavust vähendavaid tegureid ja nende olemust ning vaadelda mõningaid viimase aja populaarseid arenduspraktikaid ja tehnoloogiaid antud kontekstis. Sellele järgnevalt pakkuda välja võimalikke meetmeid tarkvara pikaajalisest ekspluatatsioonist tulenevate negatiivsete mõjude vähendamiseks ning ühtlasi ka töökindluse ja täiendavate arenduste planeeritavuse tõstmiseks.

Olles töötanud tarkvara arendajana 10 aastat nii riiklikes organisatsioonides kui ka erafirmades, väikestes ja suurtes projektides ja meeskondades, siis oma kogemuse põhjal väidan, et valdavas enamuses spetsiaaltarkvara projektides, st tarkvara mingi konkreetse ärivajaduse lahendamiseks mingis konkreetses organisatsioonis, langeb arenduse kiirus ja prognoositavus pärast teatud hulka arendusiteratsiooni oluliselt madalamale esialgsest, mil polnud veel vaja arvestada eelnevalt looduga. Kuni selleni välja, et süsteemi arendus peatatakse ning see asendatakse uuega. Sellised arengud on organisatsioonidele tülikad, kuna esiteks muutub olemasolevatesse süsteemidesse täienduste tegemine üha raskemini planeeritavaks, teiseks ka triviaalsete muudatuste sisseviimine võib venida määramatult pikale ning ilmnevad ootamatud vead seni toiminud funktsionaalsuses. Juhul kui süsteem täiesti uuesti tehakse, nõuab see suuri rahalisi investeeringuid ning samal ajal ei ole võimalik kasutajatele olemasolevasse süsteemi täiendusi luua. Lisaks uue tarkvara juurutamine on samuti üldjuhul ressursimahukas ettevõtmine, mis aeglustab tuntavalt organisatsiooni põhiprotsesse.

Töö tulemuseks on tarkvara arenduse keerukuse põhjuste käsitlus autoriteetsete allikate põhjal, kaasaegsete arendusmeetodite ja tehnoloogiate vaatlemine selles kontekstis ning võimalike arendust toetavate praktikate välja pakkumine tuginedes töö autori kogemustele tarkvara arenduses.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 52 leheküljel, 8 peatükki, 4 joonist, 3 tabelit ja 1 koodinäite.

Abstract

The goal of this work is to describe various factors that reduce the manageability and sustainability of the development process of a special software systems. By special software I mean the kind of software that is not widely distributed and is custom built for one organization's needs. These factors are then considered in the context of some modern development practicies and technologies. Finally its goal is to provide some strategies and practices to mitigate the situation.

Based on my 10 years of experience in software development and my praticipation in software projects in various sizes, I assert that after a certain amount of iterations, software development speed and predictability falls considerably. It deteriorates until it's no longer feasible to continue to make changes and improvements and the system is replaced with a new one. This dendency is troublesome for organizations because it makes very hard to plan new developments and trivial changes to software will take unpredictable amount of time. In addition hard to debug and unexpected bugs will emerge from seemingly unrelated changes. If the system is replaced, it will halt the regular change schedule and it will require additional funds and manpower. And that in turn will affect the performance of the organization's main business processes.

The result of this work is the revalidation of well known problems in modern context and description of some methods that have worked well for the author to make software development more sustainable.

The thesis is in estonian language and contains 52 pages of text, 8 chapters, 4 figures, 3 tables and 1 code example.

Lühendite ja mõistete sõnastik

Tabel 1: Lühendite ja mõistete sõnastik

API	*Application Programming Interface* - rakendusliides Arvuti operatsioonisüsteemiga või rakendusprogrammiga määratud reeglistik, mille alusel rakendusprogramm kasutab operatsioonisüsteemi või teise rakendusprogrammi teenuseid.
UTF-8	*8-bit UCS/Unicode Transformation Format* - 8-bitine UCS/Unicode teisendusvorming Unicode'i kooditabelis leiduvate märkide kodeerimismeetod, mis kasutab märkide esitamiseks ühte kuni nelja oktetti (8-bitist baiti).
HTML	*HyperText Markup Language* - hüpertekst-märgistuskeel Enimlevinud kodeerimissüsteem (tekstivorming) veebidokumentide loomiseks. HTML koodid ehk märgendid määravad ära selle, kuidas veebileht arvutiekraanil välja näeb.
XHTML	*Extensible Hypertext Markup Language* - laiendatav hüpertekst-märgistuskeel
HTTP	*HyperText Transfer Protocol* - hüperteksti edastusprotokoll TCP/IP klient-server protokoll HTML-dokumentide vahetamiseks veebis ehk lihtsamalt öeldes andmevahetusprotokoll, mida kasutatakse Internetis dokumentide vahetamiseks
SOA	*Service-Oriented Architecture* – teenustele orienteeritud arhitektuur
SQL	*Structured Query Language* - struktuurpäringukeel Enimkasutatav päringukeel, mida toetavad kõik klient-server keskkonnale projekteeritud relatsioonandmebaasid.
REST	*Representational State Transfer* Tarkvaraarhitektuuri stiil, mida kasutatakse hüpermeedia hajussüsteemide (hajusarvutuse) valdkonnas, näiteks veebis.
SOAP	*Simple Object Access Protocol* - lihtne objektipöödusprotokoll Minimaalne komplekt kokkuleppeid programmide käivitamiseks XML'i abil üle HTTP.
CORBA	*Common Object Request Broker Architecture* Arhitektuur, mis võimaldab objektidel (programmidel) omavahel suhelda sõltumata sellest, millises programmikeeles nad on kirjutatud või millises operatsioonisüsteemis nad töötavad.
Java RMI	*Java Remote Method Invocation* Protokollikomplekt, mis võimaldab Java objektide omavahelist kaugsuhtlust üle võrgu.
RPC	*Remote Procedure Call* Protokoll mis võimaldab ühel arvutil asuval programmil täita teisel ehk serverarvutil asuvat programmi.
Race condition	Konkurentsioht Süsteemi või protsessi nõrkus, mis väljendub selles, et väljund sõltub kriitilisel ja ettearvamatul viisil sündmuste ajalisest järjestusest.
JDBC	*Java Database Connectivity* Java andmebaasipöördus. Java platvormi andmebaasipöörduse spetsifikatsioon.

Tarkvara arendatavus	Tarkvarasse paranduste ja täienduste sisse viimise lihtsuse määr. Halvasti arendatav tarkvara on segase ja mitte ilmse ülesehituse ning toimimisloogikaga. Hästi arendatav tarkvara on selge struktuuriga ning võimaldab arendajal kiiresti omandada vajaliku detailsusega ülevaade teda huvitava toimemehhanismist ning vajalikud täiendused teha.
Tarkvara hallatavus	Tarkvara hallatavus on tarkvaraga seotud tegevuste, näiteks paigaldamine, seadistamine, vigade analüüsimine, teostamise lihtsuse määr.
Tarkvara töökindlus	Tarkvara omadus tõrgeteta töötada.
Tarkvara planeeritavus	Tarkvara muutmisele kuluva ressursi adekvaatse hindamise tase.
Tarkvara arenduse jätkusuutlikkus	Tarkvara arenduse korraldamine moel, et hilisem tarkvara muutmine ja täiendamine oleksid võimalikult lihtsad ning ei tooks kaasa tarkvara töökindluse langust.
Spagetikood	Tarkvara lähtekood, millel puudub selge struktuur ning mis on tarbetult keerukas oma funktsionaalsuse kohta.
Klasterdamine	Klient-server arhitektuuril põhineva tarkvara serveri poole jooksutamine kahel või enamal arvutil, mis funktsioneerivad kliendi jaoks kui üks. Kui ühe server arvutiga midagi juhtub, võtavad teised tema töö üle.
Toodangukeskkond	*Production environment* Võrguteenusena pakutava tarkvara töökeskkond, milles lõppkasutajad pöörduvad tarkvara poole. Teisteks levinud keskkonna tüüpideks on arendus- ja testkeskkond, mida kasutatakse vastavalt tarkvara arenduseks ja testimiseks.
Hooldusaken	Tarkvara töös katkestusi põhjustavate hooldustööde tegemiseks ette nähtud aeg.
Arhitektuurne terviklikkus	Tarkvara on arhitektuurselt terviklik, kui selle ülesehitus vastab läbivalt ühtsele arendaja poolt kavandatud loogikale ja põhimõtetele.

Jooniste register

Tabelite register

Koodinäidete register

Sisukord

1. Sissejuhatus

Käesoleva magistritöös uuritakse tarkvara arenduste hallatavuse ja jätkusuutlikkuse vähenemise põhjuste olemust ning püütakse leida võimalusi olukorra parandamiseks.

Käesolevas töös käsitletakse tarkvara selle arendamise vaatepunktist.

Antud peatükk sisaldab järgmisi alampeatükke:

- Taust ja probleem
- Ülesande püstitus
- Ülevaade tööst

1.1 Taust ja probleem

Harva juhtub, et peale aktiivse arendus- ja juurutusfaasi lõppu jääb tarkvara sellisel kujul töösse oma elutsükli lõpuni. Enamasti liigub tarkvara kas haldusfaasi või järgmisesse arendusfaasi. Haldusfaasis parandatakse tarkvara kasutamise käigus ilmnenud vigu või tehakse väikseid täiendusi süsteemi kasutamise tõhustamiseks. Sellises haldusfaasis võib süsteem olla üpris kaua.

Samuti on levinud süsteemide arendamine etappide kaupa ehk siis peale ühe etapi lõpetamist algab teine arendusfaas, millele omakorda järgneb selle etapi tulemi juurutus. Etappe võib olla kuitahes palju ja nende vaheline aeg varieerub päevadest aastateni.

Üldistades võib väita, et tarkvara kogu elutsükkel on väga pikk protsess ning suure tõenäosusega vahetuvad selle aja jooksul arendusmeeskonna liikmed või terved meeskonnad mitu korda. Samuti vahetuvad süsteemi kasutajad ning tarkvara haldusega tegelevad äripoole inimesed, mis teeb teabe ülekandumise ja hankimise süsteemi kohta raskeks.

Inimeste vahetumise ja aja möödumisega tekib seoses tarkvara arendusega probleem, et tarkvara lähtekoodist ülevaate saamiseks ning mingi spetsiifilise vea parandamiseks või lisafunktsionaalsuse lisamisele kulub võrdlemisi määramatu aeg. Ka arendaja ise ei pruugi enam täpselt kõiki üksikasju mäletada kui ta piisavalt pika aja möödudes enda poolt kirjutatud lähtekoodis peab muudatusi tegema. Sama kehtib ka uue arendaja süsteemi arendusse sisse elamise aja kohta. Suuremad probleemid seisnevad tarkvara arhitektuuri kontseptuaalse mõistmises ehk miks kood on selliselt struktureeritud nagu on ja sisuliste nõuete kokku viimisega realisatsiooniga ehk et mis sisulist probleemi (kasutuslugu, äriloogika iseärasust) mingi koodilõik lahendab. Olenevalt koodi kvaliteedist võib see aeg erineda kümneid kordi! Koodi kvaliteet ja selgus hõlmab endas muu hulgas: koodielementide (klassid, meetodid, muutujad) semantilist selgust, lähtekoodi struktuuri ja kommentaaride hulka ning asjakohasust.

Vaatamata sellele, et hästi dokumenteeritud tarkvara eelised on kõigile hästi teada, siis praktikas seda siiski enamasti ei tehta. Eriti vähe leiab dokumenteerimine rakendust ühele kliendile loodavas spetsiaalse tarkvara puhul. Laiatarbetarkvara ning erinevate teekide puhul on olukord küll parem, kuid samuti kaugel ideaalist. Põhjused selleks on ilmselt täiendav aja- ja ressursikulu, mis siis ühe või teise osapoole ignorantsuse tulemusel üleliigseks peetakse, äriline surve või sobiva arenduspraktika puudumine.

Üheks uue tarkvarasüsteemi arendamise põhjuseks osutubki sageli see, et olemasolev on nii

ummikusse arendatud, et triviaalsete muudatuste tegemine ja vigade parandamine võtab ebamõistlikult palju aega. Lisaks iga muudatus toob kaasa uusi vigu seni toiminud funktsionaalsuses.

Kõnealust probleemi küll leevendab pisut viimastel aastatel laialt levinud arendusraamistike kasutusele võtmine, mis määravad suuresti rakenduse struktuuri ning erinevad tüüpprobleemide lahendused, kuid siiski annaks väike hulk hästi läbimõeldud dokumentatsiooni suure efekti arenduse tõhustamisel ja sisuliste seoste loomisel lähtekoodiga.

Lisaks tekib arendusraamistike kasutamisega seoses probleem, et arendusraamistikud arenevad üldjuhul kiiremini kui mingi konkreetne tarkvarasüsteem, mille arendamisel seda kasutatakse. Sellel on mitu negatiivset aspekti: esiteks lisab arendusraamistike versiooniuuendustega kaasas käimine täiendavat ressursikulu, kuna rakendust on vaja kohandada vastavalt raamistikule ning uuesti testida. Teiseks tuleb vanemate rakenduste arendamisel alati ennast kurssi viia antud raamistiku versiooni nüanssidega ning üldjuhul on arendusraamistike arendajatel vähe motivatsiooni vanemate versioonide toetamiseks, mis muudab aja möödudes dokumentatsiooni ja muu info leidmise üha raskemaks. See omakorda raskendab ka arenduskeskkonna seadistamist.

Kogu ülalkirjeldatud problemaatikast tulenevalt muutub olemasoleva tarkvara koodibaasi haldamine aja jooksul väga raha- ja ajakulukaks. Isegi suurem probleem on asjaolu, et paranduste rahalist ja ajalist kulu ei ole võimalik adekvaatselt hinnata, seega ka tarkvaraga seotud töid planeerida ja eelarvestada. Sageli viib see olukorrani, kus väljaarendatud süsteemi tuleb maha kanda ja uut luua tunduvalt varem kui seda nõuaksid sisulise funktsionaalsuse muutused.

Olles tarkvara arenduse valdkonnas töötanud 10 aastat ning osalenud mittetriviaalsel moel ligikaudu 30 erineva tarkvara arenduses, olen täheldanud mõningaid võrdlemisi lihtsaid kuid tõhusaid arenduspraktikaid, mis oluliselt aitavad kaasa tarkvara arendatavuse jätkusuutlikkusele. Kuna olen osalenud nii mitmeaastastes ja 10-15 arendajaga projektides kui ka keskmistes, paari-kolme arendajaga projektides, kui ka väikestes, ühe arendajaga kuni kuu ajalistes projektides, nii kogenud kui ka vähekogenud arendajatega meeskondades ning erinevates tehnilistes keskkondades ja valdkondades teostatavates projektides, siis võin väita, et need toimivad nendest kriteeriumitest sõltumata. Vaatamata nende tõhususele ei ole need praktikad siiski väga laialdaselt kasutusel ning näen nende rakendamises suurt võimalust tarkvara arenduse jätkusuutlikkuse tõstmiseks.

1.2 Ülesande püstitus

Töö eesmärkideks on analüüsida erinevaid põhjuseid, miks tarkvara hallatavus ja prognoositavus teatud arvu arendusiteratsioonide möödudes oluliselt väheneb, vaadelda selles kontekstis mõningaid kaasaegseid arenduspraktikaid ja tehnoloogiaid. Teiseks peamiseks eesmärgiks on vaadelda mõningaid maailmapraktikaid ja võimalusi antud probleemidega tegeleda ning kõige lõpuks kirjeldada võimalikult minimalistlik tarkvara dokumenteerimisnõuete ja -praktikate komplekt arendajale, mis võimaldaks kiiresti omandada ülevaate tarkvara struktuurist, sõltuvustest ja tööloogikast. Eesmärk on suhteliselt väikese lisapingutusega saavutada tarkvara hallatavuse oluline kasv.

Töö eesmärk ei ole konkreetsete projektide ja näidete põhjal efektiivsuse tõusu mõõtmine ega välja pakutud lahenduste toimivuse tõestamine. Pigem püüab töö erinevatele allikatele tuginedes võtta kokku erinevad tarkvara arenduse keerukuse põhjused, tutvustada mõningaid uuemaid tehnoloogiaid ja metoodikaid selles valguses ning autori töökogemusele tuginedes pakkuda välja

praktilisi viise probleemide leevendamiseks.

1.3 Ülevaade tööst

Töö jaguneb kolmeks sisuliseks osaks:

- Esimeses osas antakse ülevaade tarkvara arenduse olemuslikust keerukusest ning erinevatest teguritest, mis tarkvara arenduse jätkusuutlikkust langetavad.
- Teises osas tuuakse välja erinevad metoodikad ja viisid, kuidas maailmapraktikas on käsitletavaid probleeme püütud ja püütakse lahendada.
- Kolmandas osas kirjeldatakse esimestele kahele osale tuginedes reegleid ja praktikaid, mille abil tarkvara arendatavust tõsta.

2. Lahendatavad probleemid

Selles peatükis on fookuses probleemi skoobi täpsem piiritlemine. Probleemi kontekstiks võetakse eelduseks, et arhitektuurse terviklikkuse saavutamist ei võeta primaarseks eesmärgiks ning tehniline võlg on möödapääsmatu.

Käsitletakse nelja probleemi:

- Ülevaate saamine tarkvarast
- Tehnilise võla mehhaaniline langetamine
- Tarkvara töökorras oleku kontrollimine
- Tüüpvigade vältimine

2.1 Ülevaate saamine tarkvarast

Kui võtta eelduseks, et ilma korrektse arhitektuurita, keskmise või suure tehnilise võlaga tarkvara on suuresti paratamatus, siis üheks esimeseks ja võib-olla suuremaks takistuseks, millest tarkvara parandamiseks ning edasiste arenduste tegemiseks üle tuleb saada, on aeg, mis kulub arendajal tarkvaraga tutvumiseks ehk sisseelamise aeg. See on esimene praktilise tähtsusega probleem, mis esile kerkib kui arendaja tarkvaraga kokku puutub ning kavatseb täiendusi või parandusi sisse viima hakata.

Samuti on selge ja konkreetne ülevaade tarkvara funktsionaalsusest ja liidestest kasulik antud tarkvara haldusega seotud probleemide lahendamisel ning väliste muudatuste mõjude hindamiseks antud tarkvaraga seoses. Näiteks rakendus vajab tööks kirjutamisõigust teatud kataloogile serveri failisüsteemis või kui rakendus saab andmed väliselt veebiteenuselt, siis mõistliku ülevaate olemasolul oleks see kiiresti tuvastatav. Vastasel juhul tuleb seda järeldada pika katse-eksitus meetodil proovimise ja erinevates tehnilistes veateadetes järje ajamisega. Antud probleemi lahendab ka eesmärk 3. Teine näide parema kiirülevaate võimalikust kasust on see, et kui toimub muutus mõnes välises teenuste komplektis, siis oleks võimalik kiiresti veenduda, kas see mõjutab rakendust või mitte.

Ehk siis esimeseks käesolevas töös käsitletavaks probleemiks on: „**kuidas kiirendada arendaja tarkvara ülesehitusse sisseelamise aega“**.

2.2 Tehnilise võla mehhaaniline langetamine

Järgmiseks lahendatavaks probleemiks on see, millised on lihtsalt rakendatavad viisid koodi kvaliteedi tõstmiseks. Põhiliselt on siin silmas peetud staatilise koodi analüüsi pakutavaid võimalusi ning koodistiili forsseerimist. Kuigi staatiline koodi analüüs on eksisteerinud pikka aega ja töövahendid selle rakendamiseks on lihtsalt kättesaadavad ja kasutatavad, siis praktilises arendustöös leiab see siiski vähe kasutust. Ilmselt on põhjused suuresti sarnased sellele, miks tehniline võlg üldse tekib ehk aeg, ignorantsus ja teadmatus.

Teiseks käesolevas töös käsitletavaks probleemiks on: „**kuidas mehhaaniliselt, ilma sisulise teadmiseta, aeglustada tehnilise võla kasvu“**.

2.3 Tarkvara töökorras oleku kontrollimine

Kui arendaja asub tööle mõne tarkvara projektiga, siis peale üldist sisulist funktsionaalsuse tutvustust esmaseks praktiliseks sammuks on arenduskeskkonna üles seadmine. Kuigi sageli võib teha teatud oletusi töövahendite ja praktikate kohta kasutusel oleva tarkvara arendusvahendite järgi, siis sageli leidub arenduskeskkonna üles seadmisel teatud nüansse, mis võivad antud tegevuse muuta tarbetult pikaks ja ebamääraseks.

Näiteks on Java programmeerimismaailmas levinud arendusvahendiks Eclipse nimeline arenduskeskkond. Keerukamate projektide puhul võib olla vajalik teostada spetsiaalseid seadistussamme selleks, et oleks võimalik kasutada integreeritud arenduskeskkonna võimalusi ning lähtekood ehituks korrektselt. Või näiteks võib olla vajalik seadistada Java maailmas tuntud tarkvara ehitusvahendi Apache Maven kasutama mitteavalikke tarkvara repositooriume või kontrollida, et oleks lubatud kirjutamine teatud kataloogidele.

Veel üks suur valdkond, mis on kasulik nii tarkvara keskkonna üles seadmisel kui ka tarkvara administraatoritel tarkvara paigaldamis- ning haldustööde puhul, on võimaluse lisamine tarkvara konfiguratsiooni korrasoleku kontrolliks. Näiteks on levinud praktika, et keskselt hallatavate veebirakenduste puhul toimub konfiguratsiooni haldus rakenduse enda haldusest eraldi. Kui nüüd uue rakenduse versiooniga on lisatud mõni konfiguratsioonivõti, siis jääb see sageli uue paketi paigaldamisel kahe silma vahele. Juhul kui oleks mingi lihtne mehhanism, mis võimaldaks veenduda, et kõik vajalikud võtmed on seadistatud, oleks võimalik vähendada tarkvara haldustöödel ilmnevate intsidentide arvu ning suurendada kindlust, et kõik vajalikud sammud on tehtud.

Kolmandaks käesolevas töös käsitletavaks probleemiks on: „**kuidas tagada, et võimekus arendustööd alustada ei võtaks tarbetult aega, tüüpprobleemid saaksid lahendatud ning arendajatel oleks veendumus, et ühtegi olulist detaili pole kahe silma vahele jäetud“.**

2.4 Tüüpvigade vältimine

Neljandaks suureks aspektiks, millele käesolevas töös käsitletava probleemilahenduses keskendutakse, on analüüsida arendaja tööprotsessi tõhustamise võimalusi. Ehk et kirjeldada erinevaid töövõtteid, kuidas kiirendada arendaja tööd tarkvaraga. Üheks suureks tehnilise võla suurenemise põhjuseks võib lugeda seda, et muudatusi ja parandusi tehakse kiiruga ja süvenemata kompleksesse keskkonda. Selliselt on lihtne unustada paljud hea koodi juurde kuuluvad detailid. Samuti põhjustab see raskesti leitavate vigade teket, mille välja uurimiseks tuleb kulutada täiendavalt aega. Sedasorti vead võivad tekkida ka puhtalt teadmatusest.

Et probleemile konkreetsemat kuju anda, tooksin näiteks ühe kujuteldava tarkvaraprojekti, mille funktsionaalsust on võimalik kasutada nii läbi kasutajaliidese kui ka läbi veebiteenuste. Kui nüüd tehakse muudatus ühes olemas olevas funktsionaalsuses, täiendatakse tuumikklasse ja kasutajaliidest, kuid jäetakse muutmata veebiteenuse konfiguratsioonifailid, siis esmapilgul kõik töötab – kasutajaliideses saab andmeid muutunud kujul salvestada, andmebaasi jõuavad andmed õigesti ning kasutajaliideses kuvatakse ka andmeid õigesti. Kuna uut teenust ei lisandunud, vaid muutus olemas olev, siis esmapilgul näis kõik olevat korras. Alles hiljem ilmnevad probleemid ühes teises süsteemis, mis antud rakenduse veebiteenuseid kasutab. Ja siis on juba võrdlemisi keeruline tuvastada vea algset põhjust. Loomulikult tuleks antud viga välja ka põhjaliku testimise käigus, kuid kahjuks laiapõhjaline automaattestimine ei ole minu kogemuse põhjal eriti levinud, seega võib mõelda alternatiivsete meetodite peale nende tuvastamisel. Näiteks kontrollnimekirjade kasutamine,

vastava liidestuse dokumenteerimine koodi juures või spetsiifiliste integratsioonitestidega.

3. Probleemi taust

Selles peatükis kirjeldatakse lahendatava probleemi tausta ja olemust tuginedes üldtuntud järeldustele tarkvara olemusliku keerukuse kohta. Keskendutakse põhiliselt kolmele suurele valdkonnale. Esiteks tarkvara arendusmustrile Suur Mudapall, mis on endiselt kõige laialt levinum tarkvara arhitektuurne mudel. Teiseks avatakse tarkvara tehnilise võla mõju tarkvara hallatavusele ning kuidas Suure Mudapalli levik loob soodsa pinnase tehnilise võla kasvuks. Kolmandas osas võetakse kokku, miks vaatamata pikaajalisele probleemi teadvustamisele pole senini universaalset lahendust leitud.

Antud peatükk sisaldab järgmisi alampeatükke:

- Antimuster Suur Mudapall
- Tehniline võlg
- Universaalse lahenduse puudumine
- Kokkuvõte

3.1 Antimuster Suur Mudapall

Üheks laiemaks algprobleemiks, mida käesolev kirjutis püüab käsitleda ning millele lahendusi leida, on tuntud kui Suure Mudapalli antimuster [1]. Suur Mudapall kirjeldab tarkvarasüsteeme, mille struktuur on kujunenud suuresti juhuslikult, vastavalt arendaja hetkevajadustele ning lähimate arenduseesmärkide realiseerimisele, mitte süsteemi kui terviku optimaalsest lahendusest.

Kuigi on palju uurimistöid tehtud erinevate efektiivsete tarkvara arhitektuuri mustrite kohta ning välja pakutud erinevaid lahendusi vastavalt süsteemide olemuslikele ja funktsionaalsetele omadustele, siis ikkagi on tegemist kõige levinuma reaalselt eksisteeriva süsteemi disaini mudeliga Seda isegi arvestades asjaoluga, et antud arhitektuuri mustri negatiivne mõju on üldteada ja seda peetakse universaalselt ebaoptimaalseks tarkvarasüsteemide ehitamise praktikaks

Arvatavasti on iga arendaja mõne sedasorti tarkvaraga kokku puutunud. Selliste süsteemide puhul on ilmne, et nende arhitektuur pole nende ootamatule funktsionaalsuse kasvule järgi tulnud. Lisaks on paljusid algseid süsteemi osi kohandatud vastavaks mõnele üksikule uuele või muutunud nõudele, jättes arvestamata süsteemi kui terviku kompositsiooni muutunud nõuete valguses. Selle tulemusel muutuvad süsteemid järkjärgult üha raskemini hoomatavaks arendajatele. Kui erinevad süsteemi osad on ilma selge funktsionaalse kihistuseta üksteisest läbi põimunud, hakkab levima globaalse oleku jagamine süsteemiüleselt ning duplikaatkoodi osakaal suureneb. Isegi kui algselt oli süsteemi tervikstruktuur läbi mõeldud ja vastas konkreetse äriprotsessi vajadustele, siis ajapikku see selgus kaob. Üheks suureks ja levinud probleemiks organisatsioonides on veel asjaolu, et head ja kompetentsed arendajad ei soovi selliste süsteemidega töötada ning sageli lahkuvad organisatsioonist, langetades sellega organisatsiooni tarkvara arendusvõimekust ning suutlikust pakkuda oma põhiteenuseid.

Järgnevalt vaadatakse lähemalt erinevaid põhjuseid ja mõjujõude, mis viivad Suure mudapalli tekkimiseni. Miks algselt parimate kavatsustega ning asjatundlike arendajate poolt loodud süsteemid jõuavad lõpuks sellisesse ummikseisu.

Lisaks uuritakse seitset Suure mudapalli alammustrit, mis kirjeldavad probleemse tarkvara

tekkimise ja olemuse põhistsenaariume. Nendeks on:

1. Suur mudapall (*Big Ball of Mud*)
2. Äraviskamiseks kirjutatud kood (*Throwaway code*)
3. Järkjärguline kasv (*Piecemeal growth*)
4. Hoia süsteemi töös (*Keep it working*)
5. Kihtide pügamine (*Shearing layers*)
6. Vaiba alla pühkimine (*Sweeping it under the rug*)
7. Rekonstruktsioon (*Reconstruction*)

3.1.1 Suure Mudapalli tekkepõhjused

Võib üldistada, et eksisteerivad teatud universaalsed mõjujõud, mis viivad kõigi Suure mudapalli erivormide tekkeni. Järgnevalt tuuakse ära mõned levinumad nendest:

- Aeg
- Kulu
- Kogemus
- Oskused
- Nähtavus
- Keerukus
- Muutused
- Skaala

3.1.1.1 Aeg

Võib juhtuda, et ei ole piisavalt aega, et võtta arvesse pikaajalisi disaini- ja realisatsiooniotsuseid. Isegi kui süsteem on hästi projekteeritud, siis ei saa sellest ajaliste piirangute tõttu lõpuni kinni pidada ning pragmaatilistel kaalutlustel tehakse arhitektuuri kvaliteedis järeleandmisi.

Lisaks tehnilistele küsimustele tuleb arvestada ka võimalike äriliste nüanssidega, näiteks sooviga olla oma segmendis esimesena mingi tootega turul või muude väliste ajaliste piirangutega. Kuna hea arhitektuuri loomine on alati ajakulukas ja pikemaajalist kasu toov tegevus, siis sellele liialt rõhku pannes võib kogu arendustöö vajadus kaduma minna või muutuda.

Kindlasti tuleks arvestada ka asjaolu, et ette mõelda saab ainult teatud piirini ning kõike ei ole võimalik ette näha, seega väga täpse arhitektuuri välja töötamine ja selle järgimine võib teatud juhtudel hakata ka arendajatele vastu töötama.

3.1.1.2 Kulu

Läbimõeldud ja asjakohase arhitektuuri tegemine on kallis ettevõtmine, iseäranis veel juhul kui probleemvaldkond (süsteemi kontekst) on arendajatele uus ja selgeid parimaid praktikaid ei ole veel välja kujunenud. Täiusliku arhitektuurilise lahenduse realiseerimine ei ole investeeringute vaatevinklist nii kõrge prioriteediga kui võimalikult kiiresti tarnida esmane tulemus. Kuigi

tehniliselt on hea arhitektuuri eelised suuresti selged, siis äriliselt peab arvestama põhiliselt turusituatsiooniga ja investeeringute tasuvusajaga. Kvaliteetne arhitektuur ei saa väärtust tagasi teenida firmas mis on pankrotis.

3.1.1.3 Kogemus

Isegi juhul kui on olemas aeg ja kavatsus tõsiselt arhitektuursete küsimustega arvestada, võib piiravaks teguriks muutuda kogemuste vähesus probleemvaldkonnas. Samuti mõjutab arenduse käiku arendajate üldine kogemus tarkvara arenduses või ka kogemus konkreetse tehnoloogilise platvormiga, millega antud juhul töötatakse. Sageli tuleb algselt realiseerida mõned võtmekomponendid, enne kui saab arhitektuurseid piire planeerima hakata.

Kogemuste puudus võib võtta mitmeid eri vorme. Alates sellest et arendaja on just koolist tulnud ja pole professionaalse tarkvara arendusega palju kokku puutunud või kui kogenud arendaja pole kokku puutunud konkreetse probleemi valdkonnaga või arendaja, kes tunneb sisulist äriloogikat, ei oma piisavalt kogemusi arhitektuuri loomisel või kasutatava tehnoloogilise platvormiga töötamisel.

Samuti töötajate vahetumine enamasti mõjutab negatiivselt organisatsioonis tekkinud teadmust erineva taseme probleemide olemusest.

3.1.1.4 Oskused

Tarkvara arendajad erinevad oma oskuste tasemelt nii kogemuste, kalduvuste, uute teadmiste omandamise, tehniliste valikute kui ka arendusmetoodika eelistuste osas. Seega kui ühte süsteemi arendavad väga erineva nägemusega arendajad, siis ilma kokkulepitud piirideta on Suur mudapall väga kiire tekkima. Isegi kui piirangud on seatud, siis võib juhtuda, et mõne teise mõjujõu (aeg, kulu) tulemusel neid ei järgita.

3.1.1.5 Nähtavus

Kui füüsiliste hoonete arhitektuur on väga hästi silmale nähtav ja kogu planeerimise ja ehituse faasis hästi jälgitav, siis tarkvara puhul on silmale näha üksnes kasutajaliides, mis ei ütle tegeliku süsteemi arhitektuuri kohta midagi. Peale arendajate ei ole kellelgi võimalik näha kuidas süsteem tegelikult on realiseeritud.

Teised arenduse osapooled näevad süsteemi kas arhitektuuri diagrammi, esitlusslaidide või tekstilise kirjeldusena.

Ilmselt üks prevaleerivaks põhjuseks, miks arhitektuurile vähe tähelepanu pööratakse seisneb selles, et see asub „kapoti all” ning on enamusele osapooltest nähtamatu. Kui süsteem näib töötavat ja seda on võimalik tarnida, siis ei näi tarkvara sisemine mehhaanika just suure tähtsusega tegur. Samuti on raske kommunikeerida arhitektuurseid riske mittetehnilistele osapooltele, mis omakorda muudab keeruliseks nende asetamise ärilisse konteksti.

3.1.1.6 Keerukus

Tarkvara arhitektuuri keerukuse taga võib olla ka lihtne tõsisasi, et probleemvaldkond, milles tarkvara toimib, ise on keeruline. Aja möödudes hakkavad arhitektuuri mõjutama varasemad kompromissid ja erinevate organisatsioonide ajalugu ning iseärasused.

Üheks keerukuse ja erinevate komponentide vaheliste piiride hägustumise põhjuseks on see, et algselt defineeritud põhilised andmestruktuurid ja funktsionaalsed jaotused ei võimalda muutunud nõudeid realiseerida, mille tulemusena muutuvad algsed arhitektuursed põhimõtted piiranguteks,

millest uued arendused mööda lähevad.

3.1.1.7 Muutused

Tarkvara arhitektuur on hüpotees tuleviku suhtes, milles arendatav tarkvara eksisteerima hakkab. Sageli muidugi juhtub, et see hüpotees osutub valeks ning ilmnevad asjaolud, mis nõuavad olulisi muutusi valitud arhitektuuris. Või muutuvad sisulised äriprotsessid ja sellega koos ka arendatavale süsteemile esitatavad nõuded. Või muutub suvaline muu aspekt, mis antud tarkvara mõjutab.

3.1.1.8 Skaala

Suurte tarkvaraprojektide juhtimine on kvalitatiivselt erinev ülesanne väikese tarkvaraprojekti juhtimisest. Head arhitektuursed põhimõtted ja tehnikad ei pruugi projekti mahu kasvades skaleeruda ning nende asemel tuleks sõltuvalt süsteemi mahust kasutada teisi arhitektuurseid võtteid, mille täpsem käsitlus jääb väljaspoole töö skoopi.

3.1.2 Suur mudapall (*Big Ball of Mud*)

Seda antimustrit tuntakse ka kui slumm (*Shantytown*).

Suure mudapalli arhitektuuril on palju ühist slummidega ehk juhuslikult arenenud, lihtsatest ja äärmiselt madala kvaliteediga majadega piirkondadest mis on tekkinud suure, vähese kvalifikatsiooni ja madalate finantsiliste võimalustega populatsiooni kontsentreerumisel.

Slummide ülalpidamine on väga töömahukas, kuna erinevaid töid teevad madala oskusteabega inimesed vastavalt hetkevajadusele ja käepäraste vahenditega. Keskne infrastruktuur puudub (kanalisatsioon ja vesi, elektrivõrk, side), kuna see nõuaks juba keskset planeerimist, kapitali ning spetsiaalset oskusteavet. Igasugused arhitektuurilised kaalutlused jäävad kõrvale.

Üheks peamiseks põhjuseks, miks tarkvara omandab slummidele iseloomulikke omadusi, on tarkvara näiline paindlikkus. Kuna tarkvaral ei ole füüsilist mõõdet, siis tundub muudatuste tegemine lihtsam ja vähem ajamahukas kui see tegelikult on. Süsteemi lõppkasutajad puutuvad enamasti süsteemiga kokku alles selle lõppfaasis. Selleks ajaks on suuremad ja põhimõttelised tarkvara kompositsioonilised otsused juba tehtud. Peale esmast tarkvara kasutamise kogemust tekib sageli palju tagasisidet ja ettepanekuid muudatuste tegemiseks. Kuna arendusprojekti lõpptähtaeg on selleks ajaks juba lähedal ning suuri arhitektuurseid muutusi ei jõua enam sisse viia, siis ongi loodud soodne pinnas slummist tuntud kiirete ja käepäraste arenduste tekkeks.

Sarnaseid arenguid esineb ka füüsilistes ehitusprojektides, kuid tulenevalt füüsilise konstruktsiooni muutmise keerukuse ja ehituse maksumuse paremale tunnetamisele mitte sellises ulatuses nagu tarkvara puhul.

Sellise levinud arengumustri parimaks vastustrateegiaks on see, et **algselt suunata tähelepanu tarkvara funktsionaalsuse ja võimaluste arendusele ning hiljem arhitektuurile ja jõudlusele.**

Tarkvara arhitektuuri üheks omapäraks on asjaolu, et mitmed süsteemi iseloomulikud arhitektuuri elemendid ilmnevad alles peale seda, kui süsteemi funktsionaalsus on realiseeritud ja töötab.

Võib ette tulla, et suureks mudapalliks muutunud süsteemi haldavas ettevõttes muutuvad üksikud inimesed, kes on suutnud süsteemi detailidest läbi närida ja suudavad sellesse muudatusi sisse viia, väärtuslikumaks kui arhitektid. Samuti on levinud nähtus, et kiirete ja arhitektuurselt läbimõtlemata ning dokumenteerimata muudatuste tegijad on organisatsioonis kõrgelt väärtustatud, kuna asjatundmatule kõrvaltvaatajale näib sellise inimese võime kiiresti probleeme lahendada

muljetavaldav ja kasulik. Kuigi tegelikkuses muudab sellise olukorra kestmine organisatsiooni järjest rohkem ebakindlamaks, kuna tema võime funktsioneerida sõltub üksikutest asendamatutest inimestest.

Suure mudapalli levikut või taandumist mõjutab palju organisatsiooni reageerimine selle ilmingutele. Kui organisatsioon väärtustab tarkvara lihtsust ja arusaadavust, mitte mõne üksiku programmeerija läbinägelikkust ja võimet ootamatutele ja raskesti analüüsitavatele probleemidele kiirustatud lahendusi leida.

Eksisteerib kolm viisi suure mudapalli vastu võitlemiseks:

- hoida kood heas seisundis jooksvalt;

- visata esimene realisatsioon tarkvarast (mustand) ära ja alustada algusest võttes arvesse saadud kogemusi (puhtand);

- aktsepteerida suurt mudapalli ja lasta süsteemil areneda isevoolu teed arvestades võimalikke tulevasi negatiivseid tagajärgi

3.1.3 Äraviskamiseks kirjutatud kood (*Throwaway code*)

Vahel luuakse planeeritavast tarkvarast esialgne prototüüp, et konkreetsemalt uurida mingi kontseptsiooni teostatavust või demonstreerida võimalikule süsteemi kasutajale olulisemaid aspekte tulevasest lahendusest. Sellise kiire prototüübi puhul ei pöörata tähelepanu ei efektiivsusele, arhitektuurile ega ka tulevasele laiendatavusele. Fookus on arenduskiirusel ning eeldatakse, et loodu visatakse peatselt ära ja hakatakse ehitama korralikku süsteemi. Sageli aga erinevatel põhjustel saab loodud prototüüp või mustand tulevasele süsteemile baasiks. Sageli ei ole enam aega korraliku arhitektuuri jaoks ning hakatakse lisama funktsionaalsust, mis omakorda kiirendab suure mudapalli tekkeprotsessi.

Vajatakse kohest parandust väiksele probleemile, kiiresti kokku klopsitud prototüüpi või kontseptsiooni mudelit (*proof of concept*). Seetõttu luuakse vahendeid valimata kiire, ühekordne kood mis lahendab ühe konkreetse parajasti päevakorras oleva probleemi.

Antud juhul ei seisne probleem mitte kiire äraviskamiseks mõeldud koodis, mis lahendab kiiresti konkreetse probleemi, vaid selles, et see kood läbi hilisemate arenduste muutub suurema ja universaalsema tarkvara aluseks või osaks. Ehk teiste sõnadega kui äraviskamiseks mõeldud koodi ei visata minema.

Üheks võimaluseks kiire koodi mõju piiritlemiseks on kasutada kõiki võimalusi, et see isoleerida muust süsteemist, eraldi objekti, paketti, või moodulisse.

3.1.4 Järkjärguline kasv (*Piecemeal growth*)

Kuigi kosemudel saab viimasel ajal palju kriitikat, tuleb alati teadvustada, et ka kosemudel oli edukas lahendus sellele eelnevatele arenduspraktikatele. Kuna arvutid ja tarkvarasüsteemid olid tol ajal oma olemuselt lihtsamad, siis oli levinud lahendus lihtsalt arhitektuurile mõtlemata suvaliselt probleemi lahendus ära programmeerida, võttes arvesse üksnes mälu mahtu ja protsessori kiirust.

Põhjus miks kosemudel edukaks osutus, oli asjaolu, et riistvara oli kallis ja ärinõuded muutusid aeglasemalt. Samuti olid kasutajaliidesed primitiivsed, mistõttu kulutati vähem aega disainile ja rohkem sisulistele küsimustele. See võimaldas pühendada rohkem aega tarkvara läbimõtlemisele ja sisulises korrektsuses veendumiseks.

Peale seda kui muutuste vajaduse kiirus ja tarkvara keerukus kasvas, ning riistvara hind langes alla teatud kriitilise punkti, kaotas ka kosemudel oma eelised ja praktilisuse tarkvaraprojektide läbiviimisel.

Suured ja kõikehõlmavad plaanid on sageli liialt jäigad, ekslikud ja ajast maas. Tarkvara kasutajad vajavad aja möödudes muudatuste tegemist.

Kosemudeli fundamentaalseks probleemiks on see, et tehniliselt kompleksses ja kiiresti muutuvate nõuetega keskkonnas ei ole võimalik adekvaatselt tulevikku planeerida. Igas ajahetkes on võimalik lahendada tol hetkel defineeritud probleem. Samuti on enamasti reaalsus see, et kasutajad arvavad end teadvat, mida nad vajavad, kuid tegelikult nad seda ei tea. Ainuke viis sellist situatsiooni käsitleda on planeerida adapteeritavus arhitektuuri sisse.

Püüdes liialt kaugele ette näha võib anda tagasilöögi, et kulutatakse aega probleemidele, mis kunagi ei realiseeru. See on levinud nähtus.

Kui tarkvara osutub edukaks, siis see meelitab suuremat hulka osapooli tegema ettepanekuid selle edasiseks täiustamiseks. See võib suurendada arenduste skoopi suuremaks kui kasutada olevad ressursid võimaldaksid.

Lihtsad süsteemid kohanduvad palju paremini muutustele kui keerukad ja suure ning jäiga/keeruka arhitektuuriga süsteemid. Modulaarne tarkvara arhitektuur võimaldab vältida süsteemi funktsionaalsuse laialivalgumist ja soodustab konkreetsete, hästi testitud komponentide tekkimist, mille arendamine on jätkusuutlik.

3.1.5 Hoia süsteem töökorras (*Keep it working*)

Kui suured ja ärikriitilised tarkvarasüsteemid lakkavad töötamast, siis esimene prioriteet on vea eemaldamine või alternatiivse lahenduse leidmine taastamaks sisuline tööprotsess, mis tarkvarast sõltub. Peale veaolukorrast taastumist on sageli raske öelda, milline tarkvara muudatus konkreetselt viga põhjustas.

Süsteemi arendusvajadused kuhjuvad, kuid põhjalik ringi tegemine pole mõistlik, kuna võib kogu süsteemi katki teha. Selle tulemusena tuleb teha kõik mis vajalik, et hoida süsteemi töös. Kuid selle tulemusena tarkvara terviklikkus ja hallatavus väheneb.

Kirjeldatud arengu mõjusid on võimalik leevendada dokumenteerides kõiki muudatusi ja vigu dokumenteerida võimalikult detailselt, et oleks tagantjärele võimalik analüüsida tarkvara tööd. Samuti on võimalik võtta kasutusele tarkvara arenduspraktikaid, mis toetavad vigade varajast avastamist. Näiteks tuleks kogu süsteem kompileerida ja testida võimalikult tihedalt. Ideaalsel juhul peale igat muudatust.

3.1.6 Kihtide pügamine (*Shearing layers*)

Erinevad tarkvara osad (kihid) muutuvad erineva kiirusega, seega oleks mõistlik süsteemi osad komplekteerida selliselt, et sarnase muutumiskiirusega osad on koos.

Tarkvara aluseks on andmed. Andmed on otseses kontaktis süsteemi kasutajatega, kes tarbivad ja loovad andmeid.

Kood muutub aeglasemalt kui andmed, koodi muudavad arendajad, analüütikud ja disainerid.

Kui selline erineva kiirusega muutuvate süsteemi osade grupeerimine õnnestub, siis on võimalik

väiksema pingutusega sisse viia kiiresti vajaminevaid muutusi.

3.1.7 Vaiba alla pühkimine (*Sweeping it under the rug*)

Halvas seisus koodibaas toob kaasa rea probleeme, mis väljenduvad otseselt arenduskuludes. Lihtsate muudatuste tegemine võtab palju aega ning võib põhjustada uusi vigu, arendajad ei taha ja ei julge vajalikke muudatusi teha.

Segane, liigselt keerukas ning juhuslik kood on raskesti mõistetav, parandatav ja laiendatav, ning on kalduvusega ajas muutuda veelgi hullemaks, kui ei kulutata energiat selle korrastamiseks. Kui seda pole võimalik teha, siis tuleks see vähemalt eraldada muust koodist, et hiljem selle juurde tagasi tulla.

Kuigi probleemid jäävad endiselt alles, siis nad on vähemalt teada ning nende mõju piiritletud. Spagetikoodi lahtiharutamisel tuleks esmalt leida kõige tihedamalt seotud osad ning need arhitektuurselt komponentideks/teenusteks jagada. Peale seda on juba võimalik defineerida korrektsed liidesed nendega suhtluseks.

3.1.8 Rekonstruktsioon (*Reconstruction*)

Vahel ilmneb, et tarkvara on jõudnud sellisesse seisu, kus on otstarbekam vana süsteem kasutusest maha võtta ja alustada uue tarkvara loomist algusest peale.

Tarkvara ei ole võimalik enam parandada ega loogiliselt analüüsida, seega luuakse selle asemele täiesti uus süsteem.

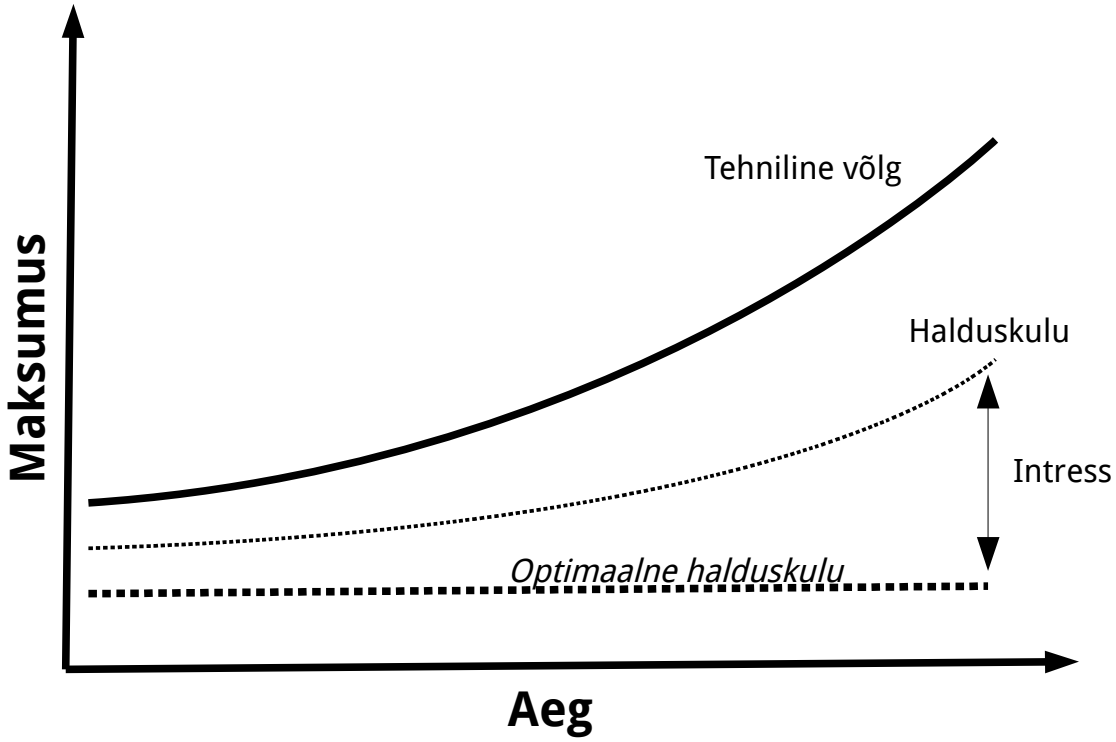
Enamasti saab süsteemi kasutuselt eemaldamise otsuse juures määravaks organisatsiooni soovimatus süsteemi arendamise ja haldamisega jätkata. Adekvaatselt planeeritud süsteem, mille haldamiseks on piisavalt ressursse, võib kesta lõpmatult kaua. Siinkohal on asjakohane analoogia füüsiliste ehitistega, vanades Euroopa linnades on piirkondi, mille vanus ulatub sadade aastateni. Kuna elu nendes on kulgenud järjepidevalt, siis on välditud ameerikalikku buumide ning surutiste tsüklist tulenevat piirkondade kiiret arendust ja hülgamist.

Süsteemi nõuded võivad muutuda niivõrd kardinaalselt, et vana süsteemi arhitektuur lihtsalt ei võimalda mõistlikul moel uusi nõudeid realiseerida. Samuti on oluliseks argumendiks kulud. Kui olemasoleva süsteemi muutmine on kulukam kui uue ehitamine, siis tuleks eelistada uue tarkvara arendamist [1].

3.2 Tehniline võlg

Järgnevalt vaatleme lähemalt mõistet tehniline võlg, selle olemust, tekkepõhjuseid ning mõju organisatsioonis erinevates ajalistes perspektiivides.

Termini tehniline võlg võttis kasutusele Ameerika programmeerija Ward Cunningham 1997 aastal [12], võrreldes tarkvara arenduses ajalise võidu saavutamiseks tehtavaid järeleandmisi koodibaasi kvaliteedis finantsmaailmas kasutatava võla ehk laenu mõistega. Intressina käsitletakse lisanduvat ressursikulu, mida vajatakse hiljem arhitektuurilisteks korrastusteks ning arenduste dokumenteerimiseks ja testimiseks. Sarnaselt finantsilisele võlale, mis teatud piirides võimaldab kiirendada äriprojektide edenemist ning tulufaasi jõudmist, kuid teatud tasemest alates hakkab järk-järgult üha koormavamalt mõjuma, kuni muutub jätkusuutmatuks ning sunnib antud kontekstis drastilisi muutusi ette võtma.



Joonis 1: Tehniline võlg

Tehniline võlg väljendub lohaka ja läbimõtlemata koodi ning arhitektuuri kasutusele jätmisega kaasnevate mõjude kuhjumises. Finantsmaailma analoogiat kasutades võib väikest tehnilist võlga tolereerida, kui sellega saavutatakse mingi oluline äriline ehk tarkvara kasutusele võtmisest tulenev eelis. Kuid selle eelduseks on, et peale tarkvara esmast väljalaset kulutatakse lisaressurssi ja korrastatakse arhitektuur, dokumenteeritakse ja testitakse uued arendused. Juhul kui seda ei tehta, siis hakkavad puudujäägid koodi kvaliteedis ja arhitektuuris kuhjuma ning evima eksponentsiaalse kasvu omadusi, mille tulemusena muutub tehniliste puuduste olemasolu üha suuremaks takistuseks vigade parandamisele ning uue funktsionaalsuse lisamisele. Sellise tendentsi jätkumine viib lõpuks olukorrani, kus tarkvara praktiliselt ei ole võimalik enam arendada ning ilmnevad raskesti silutavad vead ja koodimuudatused toovad kaasa uusi komplitseeritud vigu. Sellist olukorda võiks võrrelda finantsilise pankrotiga.

Näiteks võib vaadata niivõrd triviaalseid lohakusvigu nagu halvasti valitud muutujanimed koodis või liiga pikad funktsioonid, millele pole lisatud arendaja kavatsusi kirjeldavaid kommentaare. Alguses tundub kood veel piisavalt arusaadav ning hallatav ehk võlg on väike. Kuid kui hiljem muutunud nõuete tõttu seda sama koodi täiendatakse või muudetakse, ja ajapuudusel või ignorantsusest ei korrastata, siis võlg suureneb. Mingist hetkest alates ei julge arendajad enam põhjalikult seda koodilõiku muuta, kartes ettearvamatuid vigu, ning sellest hetkest ongi tehnilise võla eemaldamine kaelamurdvalt raskeks muutunud ehk on vaja väga põhjalikku tarkvara restruktureerimist. Viimase tegemine aga on juba oluliselt ressursimahukam kui oleks kulunud järk-järgulistele korrastustele, ning üldjuhul hakkavad mõju avaldama Suure Mudapalli peatükis kirjeldatud arengud.

Oluline on siinkohal mainida, et praktiliselt kõik tarkvarasüsteemid, mida arendatakse kommerts- või mõne asutuse kasutuseks, on peale esimese versiooni välja andmist tehnilises võlas, kuna inimesed kes ei puutu tarkvara arendusega tehniliselt kokku, ei taju tehnilise võla tegelikku olemust.

Selle põhjus on ilmselt selles, et eksponentsiaalne funktsioon ei ole inimestele intuitiivne kontseptsioon ning sama probleemi kohtab ka finantsmaailmas, kus on konkreetseks pidepunktiks rahaline vääring ja võimalus etteantud parameetritega (algsumma, intress ja aeg) erinevaid tulevikuprojektsioone välja arvutada. Samas tarkvara puhul raha ekvivalent puudub, mis muudab võla kogunemise hoomatavaks üksnes programmeerijatele ja tarkvara arhitektidele.

Tehnilist võlga jagatakse omakorda veel tahtlikuks ja mittetahtlikuks, lühiajaliseks ja pikaajaliseks. Tahtlik tehniline võlg on teadliku otsuse tulemusel muude aspektide prioritiseerimine nagu näiteks äriliste eesmärkide täitmine. Mittetahtlik on enamasti halbadest programmeerimispraktikatest tulenev ning mitteteadlik arendustöö kõrvalmõju [12].

3.3 Universaalse lahenduse puudumine

Nagu eelmises peatükis selgub, siis on tarkvara kontseptuaalse ebaoptimaalsuse ning adekvaatse arhitektuuri puudumine sageli paratamatu reaalsus. Sellega kaasnev tehnilise võla suurenemine on enamuses tarkvaraprojektides vältimatu.

Pole olemas ühtegi arendusmeetodit, ei tehnoloogias ega ka juhtimistehnikas, mis lubaks kasvõi ainult ühe suurusjärgu võrra parandada lähikümnendil tarkvara arenduse produktiivsust, veakindlust või lihtsust [2] [3].

3.3.1 Tarkvara olemuslik keerukus

Kuna aja jooksul on tarkvara arendusprotsessis kõik välised piirangud muutunud vähem tähtsaks, näiteks riistvara kättesaadavus, hind ja võimsus ning kõrgtaseme programmeerimisvahendid, siis on kõige suuremaks mõjuteguriks muutunud tarkvara olemuslik keerukus. Arvutitarkvara põhiolemus seisneb omavahel sõltuvuses olevate kontseptsioonide konstruktsioonis: andmehulgad, andmehulkade vahelised sõltuvused, algoritmid ja liidesed. Tarkvara olemus on abstraktne, mistõttu kontseptuaalsel tasandil on tegemist sama konstruktsiooniga olenemata esituskujust, olles siiski väga suurt detailsust ja täpsust nõudev. Tarkvara koostamise keerukus seisneb lahendatavale probleemile sobiva spetsifikatsiooni ja disainiga kontseptuaalse lahenduse leidmises ning veendumises, et see töötab (testimine).

Seetõttu on tarkvara loomine alati olemuslikult keeruline, olenemata valitud vahenditest või metoodikatest. Järgnevalt põgus ülevaade tarkvara olemuslikest omadustest.

3.3.1.1 Keerukus (complexity)

Digitaalsed süsteemid on iseenesest palju keerukamad kui enamus inimese poolt loodud asjad, kuna neil on väga suur hulk erinevaid võimalikke loogilisi olekuid. Tarkvarasüsteemidel on suurusjärgu võrra rohkem olekuid kui arvutitel. Tarkvara mahu kasvamine ei tähenda seda, et samad elemendid korduvad suuremal arvul, vaid erinevate elementide arv kasvab. Enamikul juhtudel elemendid suhtlevad omavahel mingil mittelineaarsel moel, mistõttu terviku keerukus kasvab samuti lineaarsest kasvust kiiremini.

Keerukust on raske kommunikeerida, mis muudab tarkvara planeerimise ebamääraseks ja takistab arendusmeeskonna suurendamist. Keerukus muudab raskeks kõiki süsteemi olekuid loetleda, rääkimata nende mõistmisest, millest tuleneb tarkvara halb töökindlus. Tarkvara liideste keerukusest tulenevad raskused liideste kasutamisel. Struktuursest keerukusest tuleneb tarkvara laiendatavuse raskused, eriti mis puudutab ettenägematute kõrvalefektide ilmnemist, ning struktuursest keerukusest tuleneb ka võimetus teadvustada endale süsteemi olekuid, mis põhjustavad

turvaprobleeme.

Lisaks tehnilistele probleemidele tulenevad keerukusest ka tarkvara halduse ja selle muudatuste juhtimisega seotud probleemid. Keerukus muudab adekvaatse pildi saamise tarkvara hetkeseisust raskeks, mis omakorda muudab juhtimisotsused juhuslikuks ning ebamääraseks. Samuti tulenevad keerukusest uute arendusmeeskonna liikmete pikad sisseelamisajad, mis muudab töötajate voolavuse äärmiselt kulukaks ja aeganõudvaks.

3.3.1.2 Vastavus välistele teguritele (comformity)

Tarkvara puhul ei ole mingit ühtset, kõike seletavat seadust ega loogikat, kuna erinevad tarkvara moodulid ja liidesed on loodud erinevate inimeste poolt. Seega on keerukus täiesti juhuslik, vastupidiselt füüsikaseadustele, mille puhul võib eeldada, et isegi kui mingile nähtusele seletust ei ole, siis on küsimus selle avastamise ajas. Tarkvara puhul sellist eeldust teha ei saa.

Igal juhul peab tarkvara vastama mingitele välistele, keskkonnast tulenevatele liidestele, mille suhtes tarkvara looja ei saa midagi ette võtta, muutmata ka tarkvara ümbritsevaid süsteeme. See on sageli ebapraktiline kui mitte võimatu.

3.3.1.3 Muutlikkus (changeability)

Tarkvarale avaldub pidevalt surve muutusteks, samuti nagu ka muud asjad mis meid ümbritsevad – majad, autod ja arvutid. Kuid kuna tarkvara on täielikult virtuaalne asi, siis on selle ulatuslik muutmine suhteliselt lihtsam kui füüsiliste asjade puhul.

Iga edukas tarkvara muutub ajas. See tuleneb kahest asjaolust. Esiteks kasutajad proovivad tarkvara rakendada ühe uute probleemide lahendamiseks, mis ei vasta täpselt algsele probleemipüstitusele. Teiseks kui tarkvara elab kauem kui tema ettenähtud kasutusaeg, siis tuleb tarkvara kohandada muutuva keskkonnaga(riistvara, tarkvara standardid, liidesed, jne).

Ehk kokkuvõtvalt tarkvara muutub igal juhul, isegi kui ta on teatud ajahetkes perfektselt(õiged eeldused, vigadeta realisatsioon) välja töötatud.

3.3.1.4 Nähtamatus (invisibility)

Tarkvara on töö ajal nähtamatu ja mittevisualiseeritav. Tarkvara ei ole võimalik geomeetriliselt kujutada, nagu näiteks hoone plaani saab. Tarkvara puhul on võimalik visualiseerida üksnes teatud aspekte sellest. Kuna tarkvara ja riistvara muutuvad ajas järjest keerukumaks, siis ei ole lootust, et tulevikus olukord paraneb. Kuna inimeste intuitsioon põhineb samuti suuresti visualiseerimisel, siis takistab see nii tarkvara kontseptuaalset kujutlemist, kui ka takistab tarkvara toimimise kommunikeerimist.

3.3.2 Erinevad „murrangulised” tehnoloogiad ja meetodid, mis pidid tarkvara produktiivsust kasvatama

Järgnevalt vaatame erinevaid tehnoloogilisi edusamme, millest loodeti olulist efektiivsuse kasvu, kuid mis ei täitnud nendele pandud lootusi.

3.3.2.1 Kõrgtaseme programmeerimiskeeled

Kõrgtaseme keeled küll vähendavad tarkvara arenduse tehnilist keerukust, võimaldades kasutada kõrgema abstraktsioonitaseme konstruktsioone ja andmestruktuure, kuid ei vähenda tarkvarasüsteemi olemuslikku keerukust, ehk olekute ja funktsionaalsuse hulka.

Programmeerimiskeelte areng on juba väga lähedal seda kasutava inimese võimele abstraktselt mõelda. Pigem ollakse juba punktis, kus keele abstraktsuse kasv hakkab arendaja produktiivsusele vastu töötama, kuna arendaja võime väga keeruliste programmeerimiskonstruktsioonide kasutamiseks on piiratud.

Praktilise näitena võiks siinkohal tuua funktsionaalse programmerimiskeele Haskell, mis küll võimaldab kasutada väga kõrge abstraktsuse tasemega keelekonstruktsioone (*monads*) ja välistada mitmeid imperatiivse programmeerimisstiili veaklasse, kuid mis pole vaatamata sellele saavutanud suurt populaarsust. Võrdluseks võib tuua firma Google-i poolt hiljuti (aastal 2009) välja töötatud programmeerimiskeele Go, mis vaatamata oma suhtelisele noorusele, on juba saavutanud märkimisväärse populaarsuse. Vastupidiselt Haskellile on Go väga minimalistlik keel, vähendades võimalikke keele konstruktsioone miinimumini, forsseerides koodi stiili ja korrektsust, kuni selleni välja, et moodulis deklareeritud, kuid mittekasutatavad teegid annavad kompileerimisel vea [16] [17].

3.3.2.2 Objekt-orienteeritud programmeerimine

Objekt-orienteeritud programmeerimise põhiliseks uuenduseks ja loodetud eeliseks oli võimalus tarkvara modelleerida rohkem reaalse elu objektide hierarhiate järgi. Praktikas ilmnes aga, et eeldus toimib loodetust märksa väiksemal hulgal tarkvara arendusvaldkondades, näiteks kasutajaliideste programmeerimisel, kuid enamasti ei sarnane tarkvara konstruktsiooni elemendid füüsilises maailmas tuntud objektidele, mistõttu defineeritavad „objektid” ei ole intuitiivselt tajutavad ning ei anna loodetud efekti produktiivsuse ja korrektsuse kasvuks.

3.3.2.3 Teegid ja raamistikud

Üheks viimaste aastate suuremaks arenguks on erinevate raamistike ja teekide kasutamine. Loogika ja eeldus seisneb selles, et mitte leiutada jalgratast ja kasutada juba loodud funktsionaalsust. Sellel lähenemisel on omad puudused.

Esiteks on iga uue raamistiku kasutusele võtmisel teatud õppimiskurv. Teiseks eeldab raamistike kasutamine pidevat selle uuendustega kaasas käimist. Lisaks kasutatakse tavaliselt raamistikust ainult väga väikest osa funktsionaalsustest, samas kui ollakse avatud vigadele ka teistes raamistiku osades.

Raamistikud kasutavad omakorda teisi teeke ja raamistikke, mis muudab lõpptoote tunduvalt suuremaks ja aeglasemaks, kui see tingimata olema peaks. Samuti ilmnevad sageli erinevate komponentide vahelised konfliktid, mistõttu kulutatakse palju aega kõrvalistele probleemidele, mis ei ole otseselt seotud sisulise eesmärgi saavutamisega. Seega arenduse kiirendamise ning lihtsustamise asemel sageli lihtsalt tegeldakse teist sorti probleemidega, ning ollakse suuremas sõltuvuses välistest teguritest. Seega ka see ei ole loodetud efekti andnud, ning tarkvara loomise olemuslik keerukus on endiselt sama.

3.3.2.4 Kasutajasõbralikud arenduskeskkonnad ja graafiline programmeerimine

Üheks suureks valdkonnaks, millele on loodetud alates 90-ndate algusest ja tegelikult pole seda ka praegu lõplikult maha maetud, on lihtsustatud, mitte-programmeerijale kergesti kasutatava keskkonna loomine. See võimaldaks domeenieksperdil vähese õppimisajaga hakata süsteemis kirjeldama äriprotsesside reegleid ning seega suhteliselt kiiresti luua ekspertsüsteem enda

valdkonnas.

Siin tekib enamasti kaks probleemi – esiteks selgub peagi, et vajatakse laiemate võimalustega töövahendit, kuna esialgne visioon ei ole piisav realistlike protsesside kirjeldamiseks ja teiseks, mis on seotud esimese probleemi lahendamisega, et ärireeglite kirjeldamine on domeenieksperdi jaoks liiga keerukas, ning seda peab tegema programmeerija. Siis aga ollakse juba halvemas olukorras, kuna programmeerija peab lisaks sisulisele funktsionaalsusele arendama ka reeglite kirjeldamise vahendit ennast. Märksa optimaalsem oleks kasutada selleks üldist laiatarbe programmeerimiskeelt, mida arendaja juba tunneb ja mis produtseeriks ka optimaalsema programmi, kuna sealt puuduks lihtsustatud arendusvahendi jaoks loodud täiendav abstraktsioonikiht [2] [3].

3.3.3 Hetkel lootusandvad meetodid

Järgnevalt kirjeldame mõningaid trende, millest praegusel hetkel loodetakse arendusprobleemidele olulist leevendust.

3.3.3.1 NoSQL

Relatsioonilised andmebaasid on olnud põhiliseks infosüsteemide andmete salvestustehnoloogiaks väga pikka aega. Põhjused, miks relatsioonilised andmebaasid on olnud niivõrd domineerivad, seisnevad selle järgmistes omadused:

- Võimaldab suurt hulka andmeid paindlikumalt hallata kui failisüsteem.
- Transaktsioonid lahendavad suure osa samaaegsete päringute(*concurrency*) ja veahaldusega seotud probleeme.
- Võimaldab kasutada andmebaasi integratsioonipunktina mitme rakenduse vahel.
- Vana ja ennast tõestanud tehnoloogiana on see muutunud standardiks, mis võimaldab erineva taustaga spetsialistidel koos tarkvaralahendusi välja töötada.
- Võimaldab paindlikku andmete juurdepääsuõiguste kontrolli.

Vaatamata relatsiooniliste andmebaaside tugevustele, on neil ka mitmeid nõrkusi, eriti viimase aja objekt-orienteeritud programmeerimistehnikate valguses:

- Relatsiooniline andmemudel erineb sellest, kuidas andmestruktuurid paiknevad rakenduse töö ajal arvuti mälus. Kuigi objekt-orienteeritud programmeerimine saavutas suure populaarsuse 90ndate alguses, siis objekt-orienteeritud andmebaasid ei saavutanud, mistõttu rakenduse programmeerijal on võimalik kasutada erinevaid andmestruktuure, mis tuleb hiljem „tõlkida” relatsioonilisele andmebaasile sobivale kujule (*impedence mismatch*). Selline andmestruktuuride ebakõla põhjustab andmete käsitlemise keerukuse kasvu, täiendavaid veavõimalusi ja jõudlusprobleeme.
- Kuigi relatsioonilisi andmebaase on kasutatud erinevate rakenduste vahelise integratsioonipunktina pikka aega, on sellel mitmeid puudusi. Suurim nendest on andmebaasi keerukuse kasv, mis arvestades tarkvara keerukuse kasvu eksponentsiaalset iseloomu muutub teatud hetkest haldamatuks. Selle tulemusel ilmnevad andmebaasi tasandil Suure Mudapalli halvad omadused.
- Relatsiooniliste andmebaaside klasterdamisel on mitmeid probleeme. Andmete mahu ja andmeoperatsioonide kasvuga on võimalik toime tulla kasutades võimsamaid

andmebaasiservereid, mille hind teatud punktist muutub väga kõrgeks, või kasutada palju odavaid tavakomponentidel põhinevaid väiksema võimsusega servereid klastris. Enamikul juhtudel eelistavad firmad klastrite kasutamist.

2005 aasta paiku toimus selge suunamuutus HTTP protokollil põhinevate veebiteenuste kasutamisele rakenduste vaheliseks suhtluseks, muutes populaarseks teenustele orienteeritud arhitektuuri(SOA). See omakorda võimaldab iga rakenduse juures hoida eraldi andmebaasi, mille struktuur sõltub ainult rakenduse enda vajadustest, mis annab palju vabamad käed andmebaasi tehnoloogia valikul.

Termin „NoSQL" tänases tähenduses tekkis 2009 aastal, kui Londonist päris tarkvara arendaja Johan Oskarsson otsis ühisnimetajat uutele andmebaasitehnoloogiatele. Sel ajal olid levinumateks alternatiivseteks andmete salvestustehnoloogiateks BigTable firmalt Google ja Dynamo firmalt Amazon. Termin muutus kiiresti populaarseks, kuigi see on pisut eksitav, kuna mõned termini alla kuuluvad uued andmebaasitehnoloogiad kasutavad päringute tegemiseks samuti päringukeelt, millel on sarnasusi traditsioonilise SQL keelega.

NoSQL andmebaaside põhilised omadused on:

- Võimaldavad teenindada suuri andmehulki klastrisse ühendatud serveritel.
- Ei kasuta andmete salvestamisel relatsioonilist mudelit.
- Andmetel puudub eeldefineeritud struktuur (*schemaless*).
- Võimaldavad salvestada andmestruktuure sellisel kujul, nagu need rakenduses kasutusel on.

Põhilisteks NoSQL andmebaaside liikideks on:

- dokumendiandmebaasid
- võti-väärtus andmebaasid (*key-value store*)
- veergude põhised andmebaasid (*column-family store*)
- graafiandmebaasid (*graph database*)

[5]

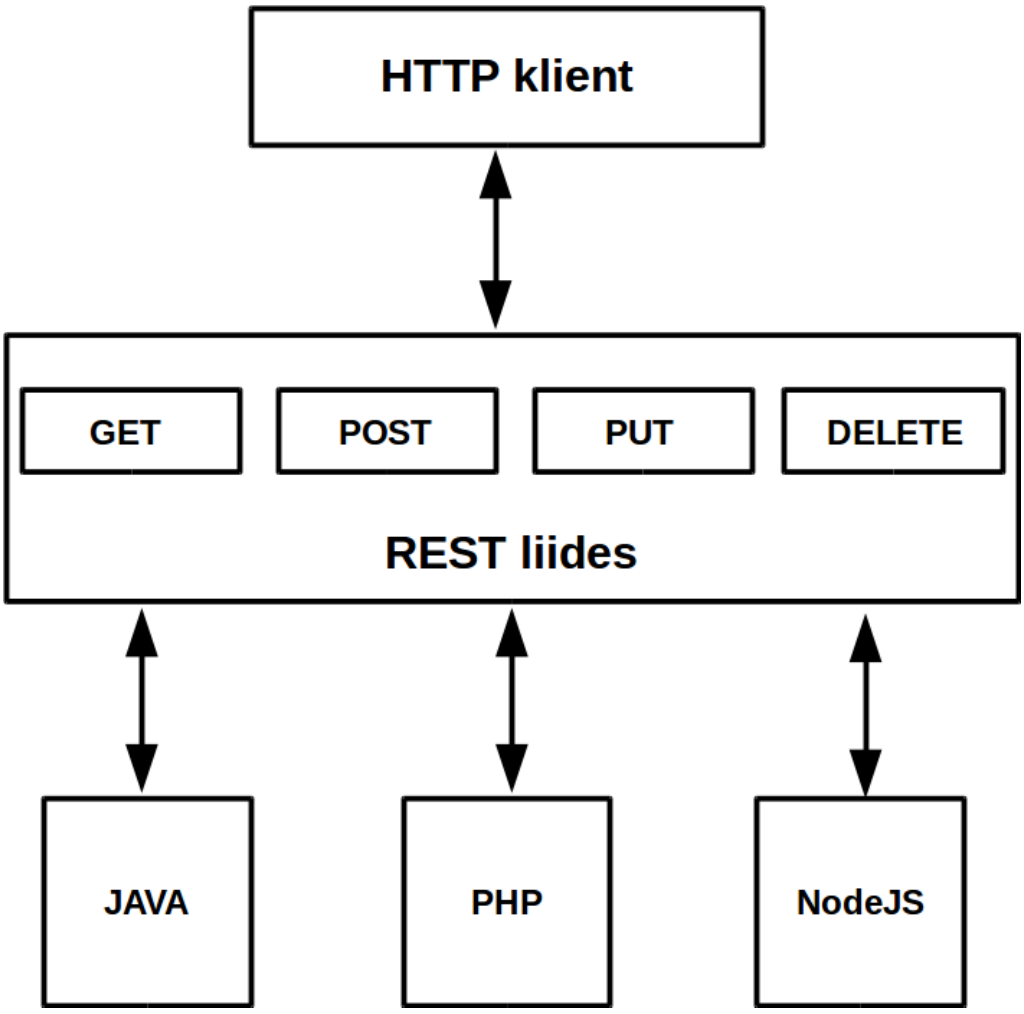
Kuigi NoSQL andmebaasid võimaldavad kiirendada tarkvara arendust, võimaldades lihtsalt muuta andmemudelit ning töötada otse objektidega, on siiski selle efekt limiteeritud, kuna rakenduse kood peab arvestama andmete ebamäärasema iseloomuga ning NoSQL andmebaasid ei ole arendajatele nii tuttavad kui relatsioonilised.

3.3.3.2 REST

REST on arhitektuuriline stiil, mis kehtestab rea piiranguid minimeerimaks võrgu latentsust ja andmevahetust ning samal ajal maksimeerides komponentide sõltumatust ning skaleeruvust. See saavutatakse pannes rõhu ühendusliidese semantikale, vastupidiselt teistele stiilidele mis keskenduvad komponentide semantikale [9].

REST liideste kasutamine on saavutanud suure populaarsuse tänu oma suhtelisele lihtsusele võrreldes teiste veebi- ja võrguliidestele, nagu näiteks CORBA, Java RMI ja SOAP. REST on ka ideaalne avalike liideste loomise vahend, mida populaarsed veebirakendused ja platvormid

laialdaselt pakuvad kolmandatele osapooltele oma rakenduste integreerimiseks.



Joonis 2: Andmevahetus REST liidese abil kliendi ja serverrakenduse vahel

Vaatamata REST arhitektuuri poolt pakutavatele plussidele, mis võimaldavad väiksemaid rakendusi üheks suuremaks platvormiks kokku siduda, on sellel ka puudusi. Suurim neist seisneb selles, et tarkvara halduskeerukus suureneb. See tuleneb suuremast komponentide arvust ning vajadusest neid seadistada ja monitoorida. Alusarhitektuuri keerukamaks muutumine toob omakorda kaasa raskesti avastatavate ja silutavate vigade tekke võimalusi.

3.3.3.3 Mikroteenused (microservices)

Mikroteenuste arhitektuur on HTTP protokollil põhinevate veebiteenuste kasutusele võtmise ning rakenduste suurema autonoomsuse tulemusel tekkinud uus arhitektuurne stiil, mis ehitab tarkvarasüsteemi üles omavahel suhtlevatest väikese funktsionaalsusega teenustest. Teenused suhtlevad üksteisega asünkroonselt kas REST teenuste kaudu või keskse juhtsiini (*event bus*) vahendusel. Selline lähenemine võimaldab väga paindlikku süsteemi arendamist ning skaleerimist, kuna võimaldab uusi funktsioone arendada eraldiseisvate komponentidena ning liidestada need vajalike teiste komponentidega. Samast komponendist võib olla igal hetkel töös mitu versiooni, mis võimaldab sujuvalt komponente produktsioonikeskkonnas välja vahetada vastavalt vajadustele. Kui mingi komponent muutub süsteemi jõudluse seisukohast pudelikaelaks, siis saab selle komponendi käivitada paralleelselt kuitahes mitmel serveril ning jagada päringud nende vahel kuni kõik päringud teenindatakse piisava kiirusega. Selline lähenemine võimaldab oluliselt vähendada Suurele

Mudapallile omaseid mõjusid.

Mikroteenuste probleemid on suuresti samad mis REST põhistel teenustel ning tulenevad samadest põhjustest. Ehk siis halduskoormus tõuseb märgatavalt ning erinevate funktsionaalsuste koordineerimine muutub keerukamaks. Samuti on sellise ülesehituse korral funktsionaalsete testide loomine raskendatud.

3.3.3.4 Programmeerijate anarhia

Programmeerijate anarhia tähistab post-agiilset arendusmudelit, kus lahendused ärivajadustele ei pärine analüütikutelt ega arhitektidelt, vaid tarkvara arendajatelt. Ehk kui kose mudeli puhul spetsifitseeritakse kõik süsteemi nõuded enne arendamist ning neid hiljem enam ei muudeta ja agiilse mudeli korral spetsifitseeritakse iteratiivselt nõuded vastavalt hetkeolukorrale, siis post-agiilse mudeli järgi kirjeldatakse arendajatele kuidas äriprotsessid toimivad hetkel ning millised tunduvad olevat põhilised probleemid. Lahendused pakuvad välja arendajad.

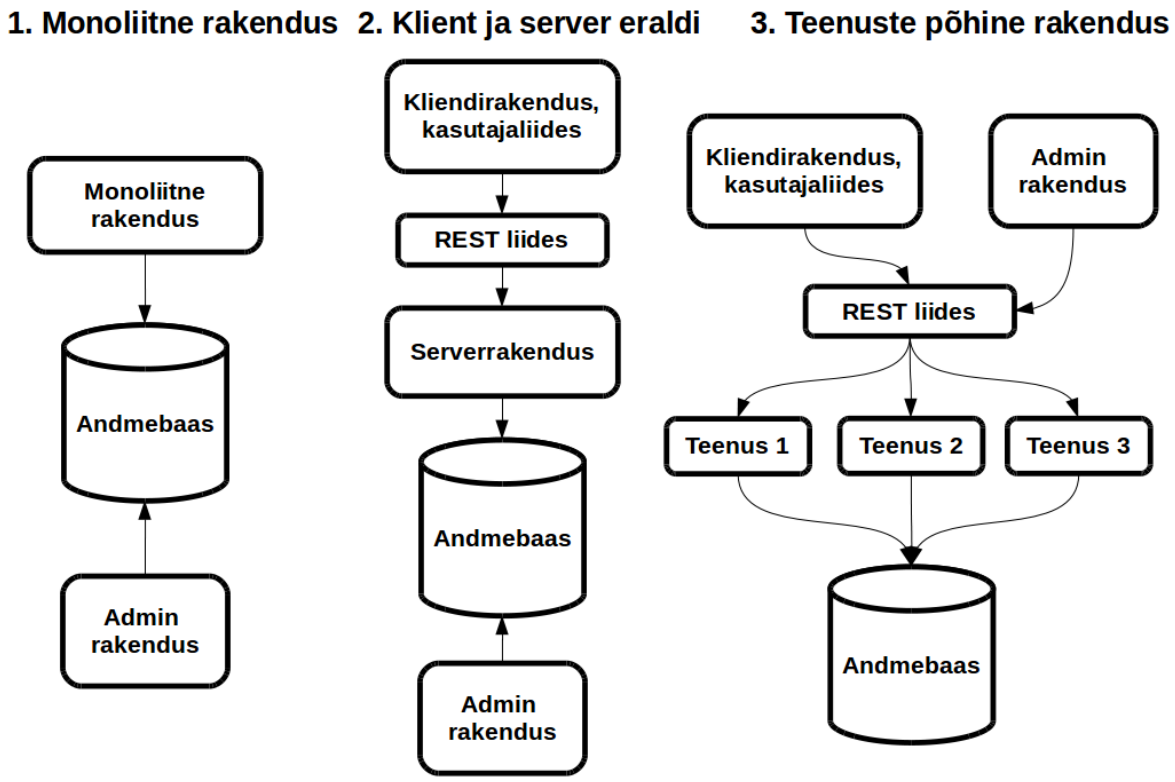
Programmeerijate anarhia korral kaotatakse kõik muud rollid arendusmeeskonnas peale arendaja ja huvigruppide ehk siis tarkvara tellija. See võimaldab viia teadmiste ülekandmisest tulenevad ebaefektiivsuse miinimumini, elimineerides vajaduse probleemiruumi erinevate rollide vahel üheselt mõistetavale kujule viia, mis viib telefonimängust tuttava info- ja ajakaoni.

Üheks põhiliseks teguriks, miks selline lähenemine suurendab nii innovatsiooni taset kui ka kasumlikkust, on asjaolu, et arendajatel on võimalik suurel määral eksperimenteerida erinevate tehniliste lahendustega, lahendamaks konkreetset probleemi. Just eksperimenteerimine on arengu mootoriks tarkvara arenduses. Programmeerijate anarhia üheks alustalaks on ebaõnnestumise hirmu kõrvaldamine meeskonna liikmetelt. Ebaõnnestumised on loomulik ning vältimatu osa, millega tuleb arvestada.

Arendajate vabadusega kaasnevad mitmed tavapärasest arenduspraktikates mittekasutatavad aspektid. Näiteks arendaja ei pea kelleltki nõusolekut küsima mingi konkreetse tehnilise lahenduse teostamiseks, vaid võib tegutseda vastavalt oma äranägemise järgi. Mitte juhindudes analüütikute poolt kirjeldatud kasutuslugudest ja arhitektide poolt määratud tehnilistest nõuetest. Isegi kui tekivad erimeelsused teiste arendajatega mingis tehnilises küsimuses, siis ei ole see probleemiks, kuna erinevad osapooled võivad lihtsalt realiseerida kumbki oma nägemuse ja hiljem võrrelda, kumb reaalsuses paremini toimib.

Sellist tehnilist mitmekesisust on võimalik praktikas hallata kasutades mikroteenustel põhinevat arhitektuuri. Samuti võimaldab mikroteenuste arhitektuur erinevate platvormide ja programmeerimiskeelte kasutamist, ainsaks tingimuseks on, et komponent suudab kokku lepitud integratsioonikanalit kasutada (HTTP, juhtsiin, sõnumid). Sellisest arhitektuurist tulenevalt ei kirjutata üldjuhul ka ühikteste, kuna kõik süsteemi komponendid on äärmiselt väiksed, vaid mõnisada rida koodi, siis on võimalik nende töö korrektsust hinnata ja muudatusi sisse viia ilma kartuseta süsteemi teisi osasid mõjutada, kuna neid lihtsalt ei ole antud komponendis. See aitab kõrvaldada ka Suure Mudapalli ilminguid koodibaasis ehk siis hallatakse palju väikseid mudapalle, mille kontrollimatut kasvu piiritletakse teenuse skoobiga [10].

Programmeerijate anarhia on kõigist senistest võimalikest nn „hõbekuulidest”, st tarkvara arendusmeetoditest mis annaksid olulise efekti produktiivsuses, innovatiivsuses ja hallatavuses, üks radikaalsemaid ja sobivates tingimustes ka paljulubavam, kuid ei tohi unustada et ka sellel on omad nõrgad küljed nagu kõigil teistel meetoditel.



Joonis 3: Monoliitne vs teenuste põhine rakendus

Esiteks traditsioonilistes organisatsioonides, mis ei ole keskendunud üksnes tarkvara arendusele, vaid tarkvara arendus on toetava iseloomuga alamüksus, ei ole ilmselt tavapärane haldustasand valmis oma näilist kontrolli tarkvara arenduse üle ära andma. See kontroll on näiline, kuna sageli tarkvara arendust juhtivad inimesed ei mõista tarkvara olemuslikku keerukust ning panustavad seal, kus see efekti ei anna, ning tõmbavad ressursse tagasi võtmetähtsusega töölõikudes. Samuti ei osata hinnata erinevate tehniliste lahenduste kõiki aspekte ja mõjusid. Usutavasti oluline osa Suure Mudapalli negatiivsetest arengutest on tingitud juhtivate organite ebakompetentsusest tarkvara arenduse küsimustes.

Teiseks saab programmeerijate anarhiat rakendada üksnes kõrgelt kvalifitseeritud töötajate korral, kes suudavad ja tahavad tehnilisi probleeme loovalt lahendada ning ei vaja välist motiveerimist ja juhendamist. Selliseid inimesi on raske leida ja palgata.

Kolmandaks eelduseks on suur usaldus arendajate ja teiste huvigruppide vahel [18].

3.4 Kokkuvõte

Võib üsna suure kindlusega väita, et nii Suure Mudapallis kirjeldatud arhitektuurse ebatäiuslikkuse ilmnemine iga suurema tarkvaraprojekti puhul ning tehniline võlg on paratamatus. Ühtlasi on selge, et tarkvara olemuslikust keerukusest ei ole võimalik lähiajal üle saada olenemata parasjagu levivatest arendusmetoodikas või arhitektuursetest trendidest, kuna probleemid, millega praeguse aja tarkvara arenduse puhul kokku puututakse, on oma olemuselt samad mis 30 aastat tagasi.

Üheks lahenduseks on muuta organisatsiooni, andes rohkem vabadust ja vastutust tehnilistele

positsioonidele nagu arendajad ja disainerid, kuna nad oskavad paremini hinnata tehniliste aspektide mõju tarkvarale. Selliste muutuste tegemist takistab sisseharjunud organisatsiooni toimimise inerts ning tehniliste positsioonide sobimatus ja soovimatus administratiivsete töödega tegeleda.

Pigem oleks otstarbekas keskenduda meetoditele ja viisidele, kuidas säilitada ülevaade tarkvara struktuurist olenemata tehnilise võla suurusest või arhitektuursest ebaoptimaalsusest. Ühtlasi peaks uurima võimalusi tehniliste meetoditega tarkvara läbipaistvuse kasvatamiseks.

4. Erinevad maailmapraktikad

Selles peatükis võetakse vaatluse alla mõned laialt levinud praktikad ja näited, kuidas töö esimeses pooles kirjeldatud problemaatika taustal määratletud probleeme on püütud lahendada.

Antud peatükk sisaldab järgmisi alampeatükke:

- Koodi üldine ülevaade ehk README fail
- Tehnilise võla piiritlemine
- Tarkvara arenduse stiil
- Testide põhine arendus

4.1 Koodi üldine ülevaade ehk README fail

Üheks põhiliseks ja ka efektiivseimaks tarkvara üldise ülevaate kommunikeerimise meetodiks on luua projekti juurde README nimeline fail, mis sisaldab infot antud tarkvaraüksuse kohta.

Olulised nüansid README faili juures on järgmised:

1. on ajakohane
2. asub lähtekoodi juurkaustas
3. on salvestatud mingis tekstipõhises formaadis

Formaadiks sobib näiteks tavaline tekst (*.txt laiendiga fail või ka ilma laiendita), Markdown formaadis tekstifail või ka lihtsalt HTML formaadis fail. Ei sobi suured dokumendiformaadid, nagu näiteks Microsoft Wordi failid (*.doc või *.docx), OpenDocument failid (.odt), ega ka RichTextFormat (*.rtf) või Portable Document Format(*.pdf). Põhiliseks põhjuseks on see, et suured dokumendiformaadid eeldavad mingit kindlat lisatarkvara olemasolu, need ei ole lihtsalt parsitavad ning on sageli kohmakad muuta(*.pdf). Lihtsalt käsitletavus on äärmiselt oluline selle tõttu, et README faile sageli kuvatakse veebis (GitHub, BitBucket või muu veebipõhine tarkvara halduskeskkond), neid hallatakse versioonihaldustarkvaraga (parsitavus) ning lihtsad tekstipõhised formaadid võimaldavad jälgida failides toimunud muudatusi ning sageli vaadatakse neid mittegraafilistes keskkondades (näiteks üle tekstiterminali füüsiliselt eemal asuvas serveris).

Levinumates koodihalduse keskkondades, näiteks GitHub ja BitBucket on väga suurt rõhku pandud README failide olemasolule. Kõik juhendid ning ka erinevad visuaalsed märguanded juhivad kasutaja tähelepanu sellele, et see fail luua ja seda kuvatakse alati esimese asjana kui keegi antud repositooriumi vaatab.

4.1.1 README faili elemendid

README fail sisaldab üldiselt järgmisi elemente:

Tabel 2: README faili elemendid [19]

Element	Kirjeldus
Konfigureerimine	Tarkvara seadistusvõimalused
Paigaldusjuhend	Lühike juhend tarkvara paigaldamiseks

Kasutusjuhend	Juhised tarkvara kasutamiseks (kui mahuliselt on võimalik)
Failid	Nimekiri tarkvara lähtekoodi failidest.
Litsents	Autoriõiguste ja kasutuslitsentsi informatsioon
Kontaktinfo	Levitaja või arendaja kontaktinfo
Teadaolevad vead	Vigade nimekiri, millest arendaja on teadlik ning võimalikud lahendused nendele.
Tüüpprobleemide lahendused	Sageli esinevate küsimuste ja probleemide vastused (mitte tarkvara vead, vaid kasutamisel segatust tekitavad nüansid)
Partnerid	Nimekiri tunnustatud koostööpartneritest
Muudatused	Muudatuste nimekiri vastavalt versioonidele
Uudised	Erinevad tarkvaraga seotud uudised.

4.1.2 Markdown

Markdown on lihtteksti vormistussüntaks, mis on loodud selliselt, et seda oleks lihtne lugeda ja kirjutada ning soovi korral HTML formaati konverteerida. Markdown on väga levinud README failide formaat. Lisaks kasutatakse seda erinevates veebi keskkondades, nagu näiteks foorumid, sisuhaldussüsteemid ja ajaveebiplatvormid. Markdown sobib selleks, et kiiresti tekstieditoriga luua kindla kujundusega dokumente.

Kuigi tegemist on väga levinud ja hästi toimiva formaadiga, on selle üheks puuduseks ühtse standardi puudumine, mistõttu paljud tööriistad ja rakendused implementeerivad sellest teatud variandi. Lisaks on loodud Markdowni põhisüntaksile mitmeid laiendusi, et tagada laiem ühilduvus teiste formaatidega ning lisada algversioonist puuduvaid teksti struktuuri- ja kujunduselemente [20] [21].

4.2 Tehnilise võla piiritlemine

Järgnevalt kirjeldan viise kuidas takistada tehnilise võla kasvu väliste vahendite abil ilma sisusse süvenemata. Mõistagi on sellistel vahenditel piiratud mõju, kuna need ei suuda parandada sisulisi puudusi, kuid arvestades asjaolu, et järgnevalt kirjeldatavate lahenduste kasutamine ei ole minu kogemuse põhjal üldlevinud, siis annab see siiski märkimisväärse efekti.

4.2.1 Koodi stiil

Esimene puhtalt visuaalne ning üksnes tarkvara arendaja vaatepunktist oluline aspekt on koodi stiil ja formaat. Küsimused nagu näiteks tabulatsiooni või tühikute kasutamine koodi treppimiseks, kuidas paigutatakse sulud, loogelised sulud ja kas konstandid kirjutatakse suurtähtedega, on koodi korrektsuse seisukohalt absoluutselt ebaolulised ning kompilaator ei pööra sellele üldse tähelepanu. Kuid töö konsistentse formaadiga koodibaasiga muudab arendaja töö olulisemalt lihtsamaks, kuna ühtlustab erinevate arendajate kirjutatud koodi ja vähendab visuaalset müra, mille kompenseerimine hajutab tähelepanu koodi sisulisest tähenduselt.

Lisaks sellele on ühtse stiili järgimisel ka omadus välistada raskesti leitavaid vigu, mis tulenevad lähtekoodi kodeeringu erinevustest. Kuigi üldstandard on maailmas UTF-8 kodeering, siis kuna näiteks Windowsi operatsioonisüsteemis ei ole see vaikekodeering, siis võib see kergesti põhjustada väga raskesti leitavaid vigu.

4.2.1.1 Stiili juhend

Paljudel levinumatel programmeerimiskeeltel on olemas ametlik koodi stiili standard, näiteks Oracle Java keelel [22] ja Google Go keelel [23]. Sageli defineerivad firmad enda spetsiifilise koodistiili juhendi, näiteks Google on Java lähtekoodile välja töötanud enda stiilijuhendi [24].

Usutavasti ei ole igal firmal praktiline nii põhjalikku stiilijuhendit luua, vaid on mõistlikum kasutada mõnda olemas olevat, näiteks eelviidatud suurfirmade poolt välja töötatud variante. Vähem aeganõudev ja praktilisem on luua ettevõttes kasutatavale arendusvahendile stiilireeglite komplekt ja seda kasutada firma siseselt ja jagada ka oma partneritele.

4.2.1.2 Koodi stiili forsseerimine

Enamusel arenduskeskkondadesse integreeritud tekstieditoridel, näiteks Java maailmas tuntud arendusvahend Eclipse, on koodi formaatimise funktsionaalsus sisse ehitatud. On võimalik ka eksportida või importida konkreetsed seadistused ja nende vahel vastavalt vajadusele ümberlülitusi teha. See võimaldab kergesti mingi koodi vastavust kehtestatud stiilile kontrollida ja tagada.

Lisaks arendusvahendite poolt pakutavatele võimalustele on paljude keelte jaoks olemas ka eraldi tööriistad sama probleemi lahendamiseks. Näiteks Google Go keeles on tööriist *gofmt*, mis tagab koodi vastavuse kehtestatud formaadile. Ühtlasi on kogu standardteekides leiduv kood formaaditud selle vahendiga.

Koodi formaadi kontrolli on võimalik teostada ehitusprotsessi osana. Koodi staatilised analüsaatorid võimaldavad kasutada ka koodi formaati kontrollivaid reeglistikke, mis võimaldab jooksvalt garanteerida formaadi korrektsust.

4.2.2 Staatiline koodi analüüs

Staatiline analüüs on tarkvara lähtekoodi analüüsi meetod, mille käigus koodi reaalselt ei käivitata, vaid püütakse leida ebaefektiivseid koodilõike, süntaksivigu, juhtida tähelepanu tüüpilistele hooletusvigadele ning halbadele programmeerimispraktikatele. Kuigi staatilist analüüsi on võimalik läbi viia ka käsitsi inimeste poolt, siis enamasti viidatakse sellele kui automatiseeritud protsessile, mida viiakse läbi spetsiaalse tarkvaralise tööriistaga, mis vastavalt eeldefineeritud reeglitele analüüsib lähtekoodi.

Kuna see toimub väga kiiresti, on võimalik kogu koodibaas suvalise tihedusega üle kontrollida ning reegleid lisada või täiendada. Samuti annab automatiseeritud staatiline analüüs alati sama tulemuse, käsitsi läbivaatusel võib esineda inimlikke vigu [4].

Põhilisteks staatilise koodianalüüsi töövahendi omadusteks on:

- Kasutuslihtsus – analüüsi tulemused peavad arvestama arendajate erinevat taset ning selgitama selgelt vea olemust.
- Reeglistik – analüüsi aluseks olev reeglistik peab olema kõrge kvaliteediga, st sisaldama oluliste vigade avastusmustreid, ning paindlik ehk reegleid peab olema võimalik seadistada vastavalt olukorrale, näiteks kindlaid reegleid teatud koodiosades välja lülitada, ning ka vajadusel uusi reegleid juurde lisada.

Lisaks võimalikule kasule tuleb alati silmas pidada ka antud töövahendi piiranguid. Automaatse staatilise koodi analüüsiga ei ole võimalik leida sisulisi vigu, vaid üksnes mehhaanilisi. Samuti tuleb kriitiliselt hinnata staatilise analüüsi aluseks olevat reeglistikku ning seda vajadusel täiendada.

Samuti peab arvestama, et teatud kontekstis võib mõni üldiselt halvaks praktikaks peetud tarkvara konstruktsioon olla õigustatud.

Konkreetseteks näideteks levinumatest staatilise analüüsi töövahenditest Java maailmas on FindBugs ja CodeNarc. Nimetatud töövahendid on liidestatud või integreeritud paljude ka teiste arendusvahenditega, nagu ehituskeskkonnad, arenduskeskkonnad ja raamistikud. Seetõttu on võimalik staatilise analüüsi etapp lihtsalt integreerida arendaja üldisesse töövoogu [13].

Näiteks Java ja Groovy keskkonnas populaarne staatiline koodianalüsaator CodeNarc sisaldab järgnevaid kontrollreeglite pakette:

Tabel 3: CodeNarc vaikereeglite paketid

Reeglite pakett	Kirjeldus
Põhiline(*basic*)	Põhilised hooletusvead tingimuslausetes, korduvused ja tühjad koodiharud.
Sulud (*braces*)	Sulgudega seotud ebakohad.
Paralleeltöötlus (*concurrency*)	Erinevad jõudluse ning potentsiaalsete veaolukordade välja toomine (*race condition*)
Konventsioonid (*convencion*)	Erinevad üldised hea stiili reeglid.
Disain (*design*)	Objekt-orienteeritud ja mustrite kohta käivad tingimused.
Mittekorduvus (*DRY – don't repeat yourself*)	Korduvate definitsioonide reeglid.
Parendused (*enhanced*)	Reeglid ebaotstarbekate koodi osade kohta, mis pole otseselt valed, kuid mis on võimalik selgemalt ja lihtsamalt lahendada.
Erindid (*exceptions*)	Erindite käsitlemise tüüpvigade tuvastamine.
Formaatimine (*formatting*)	Erinevad vormindusega seotud reeglid.
Üldised (*generic*)	Samuti üldised illegaalsete konstruktsioonide tuvastamise reeglid.
Grails	Grails on üks levinumaid Groovy keele raamistikke veebirakenduste loomiseks. CodeNarc-iga tuleb vaikimisi kaasa reeglistik Grailsi spetsiifiliste vigade leidmiseks.
Groovy keel (*groovyism*)	Reeglid idiomaatilise Groovy süntaksi kasutamiseks. Groovy võimaldab paljusid programmikonstruktsioone kirjutada lühemalt ja selgemalt kui standardne Java keel.
Importimine (*imports*)	Mittevajalike import lausetele tähelepanu juhtimine.
JDBC	Andmebaasi ühendusega seotud tüüpvead.
JUnit	JUnit testimisraamistiku spetsiifilised veaolukorrad.
Logimine (*logging*)	Logimise parimate praktikate kontroll ja tüüpvead.
Nimetamine (*naming*)	Standardsete nimekonventsioonide järgimine.
Turvalisus (*security*)	Süsteemi turvalisuse aspektid.
Serialiseerimine (*serialization*)	Serialiseerimisega seotud koodi kontroll.
Mahureeglid (*size*)	Liiga mahukate ja komplekssete klasside ning meetodite vältimine.
Ebavajalik kood	Üleliigse koodi tuvastamine, näiteks tüübi konversioonid ja koodiharud,

(*unnecessary*)	mida kunagi ei täideta või mille täitmine ei muuda programmi tööd.
Mitte kasutusel olev kood (*unused*)	Deklareeritud kui mitte kasutatavad koodi osad.

Kõiki nimetatud pakette on võimalik sisse ja väjla lülitada vastavalt konkreetse projekti vajadustele ning reguleerida ükskute reeglite tasemel. Samuti on võimalik lisada enda poolt defineeritud reeglistikke ning olemasolevaid üle laadida.

4.2.3 Kirjanduslik programmeerimine (*Literate programming*)

Literate programming on Ameerika arvutiteadlase Donald Knuthi poolt välja töötatud programmeerimise viis, mille järgi programmi kood on eelkõige probleemi selgitus teisele inimesele kasutades mõnda inimkeelt, nagu näiteks inglise või eesti keel, mille vahele on segatud makrod ja lähtekoodi lõigud mõnes programmeerimiskeeles. Teksti sisse paigutatud makrod moodustavad iselaadse pseudokoodi, mis moodustab madalama taseme üldotsetarbelise programmeerimiskeele pealse metakeele vastavalt käsitletavale probleemile.

Knuthi välja pakutud *literate programming* programmeerimisparadigma järgi ei seisne tarkvara kirjutamine arvuti ehitusest ning eripäradest lähtuvaks tegevuseks, vaid põhiliseks on järgida programmeerijate loogika ja mõtete kulgu ning olla kergesti jälgitavad inimesele, mitte masinale

Literate programming töövahenditega eraldatakse kaks eraldiseisvat programmi esitusviisi: üks mis sobib kompileerimiseks või käivitamiseks arvutil, ja teine mis sobib lugemiseks kui tavaline tekst ja mida on võimalik konverteerida sobivasse dokumendi formaati, näiteks TeX [11].

4.2.3.1 Eelised

Knuthi järgi on sellise programmeerimisviisi tulemuseks kõrgema kvaliteediga tarkvara, kuna sunnib programmerija täpsemalt ja konkreetsemalt läbi mõtlema probleemi sisulise asetuse ja lahenduskäigu, ning toob võimalikud loogikavead selgemini esile. Lisaks kujuneb arenduse käigus ühtlasi välja ka heal tasemel täpne dokumentatsioon, mis aitab nii tarkvara arendajal hiljem järgida oma algset mõtteprotsessi ning ka teistel arendajatel kergemini mõista programmi ülesehitust ja tööloogikat.

See erineb tunduvalt tavapärasest arenduspraktikast, kus lähtekoodi tööloogikat tuleb tükk-tüki haaval taastada, püüdes järgida kompilaatorist tulenevat programmi käivitusprotsessi ning lähtekoodis asuvaid kommentaare. Lisaks sellele võimaldab makrodest moodustuv metakeel paremini mõista suuremaid kontseptuaalseid programmi osi ning seeläbi saada parem tervikülevaade tarkvara ülesehitusest.

4.2.3.2 Literate programming ja dokumentatsiooni genereerimine

Literate programming-ut aetakse sageli segamini dokumentatsiooni genereerimise vahenditega, nagu näiteks Java JavaDoc süsteem, mis võimaldab spetsiaalselt märgistatud kommentaaridest genereerida programmi liidese dokumentatsiooni. Tegelikkuses on need kaks asja risti vastupidised: *literate programming* lähenemise korral on programmi lähtekood lisatud dokumentatsioonile ehk arendaja mõtteprotsessile, aga koodi genereerimise puhul eraldatakse koodile lisatud kommentaarid ühtsesse dokumentatsioonisüsteemi, mis järgib kompilaatori või muu programmi käivitusmehhanismi loogikat [26].

4.2.3.3 Ruby

Sarnase inimese mõtteviisi lähedase filosoofia alusel on loodud ka programmeerimiskeel Ruby, mille puhul on produtseeritava masinakoodi optimaalsus teisejärguline, ning inimliku mõtteprotsessi järgmine esmane prioriteet. Samuti on Ruby keele üheks aluspõhimõtteks pakkuda kogenud Ruby kasutajale ootuspärast ja mitmetimõistetavate konstruktsioonide puudumist. Need printsiibid suurendavad programmeerija produktiivsust.

Selle tagajärjel on Ruby ka üks aeglasema käivitusjõudlusega keeli maailmas, mis ei ole siiski takistanud selle populaarsuse kasvu. Koodi käivituskiiruse aeglust on võimalik kompenseerida erineval moel, näiteks rakendusi klasterdades ja erineva taseme vahepuhvrite kasutamisega [27].

4.3 Tarkvara arenduse stiil

Valdavalt on levinud tarkvara kirjutamisel imperatiivne stiil ehk rakenduse juhtimine tarkvara oleku muutmise kaudu. Teine programmeerimise paradigma on funktsionaalne programmeerimine, mis rõhutab funktsioonide rakendamist väärtustele. Funktsionaalsed programmeerimiskeeled on oma ajaloo vältel pälvinud rohkem teoreetikute kui praktikute tähelepanu [25].

4.3.1 Funktsionaalne stiil

Funktsionaalsel stiili korral juhitakse rakenduste tööd funktsioonide rakendamise teel muutumatutele algväärtustele ning välditakse oleku muutmist. Seetõttu on võimalik funktsioone käsitleda iseseisvate arvutusüksustena, mis samade sisendväärtuste korral tagastab alati sama tulemuse. See omadus tagab ka selle, et funktsionaalses stiilis kirjutatud tarkvara skaleerub paremini kui imperatiivses stiilis programmeeritud rakendused, kuna funktsioone on võimalik käivitada paralleelsete sõltumatute protsessidena, ilma et peaks arvestama globaalsest olekust tulenevate nüanssidega. Ühtlasi tagab see rakenduste suurema modulaarsuse ja töökindluse [25].

Vaatamata sellele et funktsionaalne stiil aitab üpris efektiivselt vähendada Suure Mudapalli levikut süsteemis, ei ole see siiski laialt levinud, vaid leiab põhiliselt kasutust akadeemilises keskkonnas. Selle põhjusteks võib lugeda ilmselt asjaolu, et imperatiivne arendusparadigma on inimestele kergemini omandatav ja enamus olemas olevat tarkvara on juba imperatiivses stiilis kirjutatud. Kuigi viimasel ajal on funktsionaalne programmeerimine üha rohkem kasutust leidnud, siis ei ole oodata et see saavutaks märkimisväärse kasutusulatuse lähimatel aastatel.

4.3.2 Rakenduste olek ja integreerimine

Enamus tarkvara kirjutatakse tänapäeval, nagu ka varem niikaua kui tarkvara on kirjutatud, imperatiivses programmeerimisstiilis. See tähendab et rakenduse tööd juhitakse tema oleku muutmise kaudu. Kuigi meetodi või klassi tasemel on olekut lihtne juhtida ja jälgida oleku muutusi erinevate programmis tehtavate tegevuste tagajärjel, siis rakenduse ülest või mitmest rakendusest koosneva süsteemi ülest olekut on juba oluliselt raskem jälgida, kuna süsteemi erinevad osad võivad olekut muutes põhjustada ettenägematuid kõrvalefekte mõnes teises rakenduse osas. Samuti võimaldab lihtsalt kõrvale kalduda arhitektuuri kihtidest ning ühendada rakendusi osi, mille suhtluseks näeb arhitektuur ette vahemehhanisme [8].

Rakenduse kasvades, arendajate lisandumisel ning ka muude arengute tagajärjel on tarkvara ülesed oleku muutused üheks peamiseks Suure Mudapalli ilmingute allikas. Põhjus miks globaalse oleku kasutamine on väga levinud, on asjaolu, et nii on võimalik väga kiiresti parandada vigu või lisada funktsionaalsust ilma arhitektuuri arvestamata. Juhul kui tarkvarasüsteem koosneb paljudest

rakendustest, siis võib olek olla ka rakenduste ülene. Tüüpiliselt kasutatakse rakenduste ülese oleku salvestamiseks andmebaasi. Ehk siis on üks andmebaas, mida kasutavad paljud rakendused ühiselt. Ühe rakenduse poolt sisse viidud muudatused on koheselt teisele kättesaadavad. Kuigi andmebaasi kasutamine andmevahetuskanalina muudab rakenduste integreerimise lihtsaks ning ka võrdlemisi efektiivseks, põhjustab see sageli rakenduste arendatavuse ja hallatavuse ummikseisu jõudmist, kuna koodi hulk mida mingi vea põhjuste tuvastamiseks arvesse võtta on liiga suur. Sellise situatsiooni muudab eriti tülikaks veel asjaolu, et iga rakenduse eraldi välja vahetamisest ei piisa olukorra parandamiseks, vaid välja tuleb vahetada enamasti kõik rakendused.

Seega rakenduste arendamisel on oluline hoida olek võimalikult väikeses skoobis ning selle muutmiseks kasutada arhitektuuris ettenähtud liideseid. Juhul kui tarkvarasüsteem koosneb mitmest rakendusest, siis kasutada integratsioonipunktina kas sõnumeid või veebiteenuseid. Või kasutada andmebaasi sõnumiserveri imiteerimiseks. Selliselt on võimalik ennetada Suure Mudapalli mustrit laienemist määrani, mil olukorra parandamiseks vajalikud muudatused on niivõrd suured, et takistavad vajalike arenduste sisseviimist. Ehk siis eesmärk on mitte jäigalt siduda erinevaid infosüsteemis olevaid tarkvara komponente, seetõttu ei ole lisaks andmebaasile ka näiteks RMI sobiv rakenduste integreerimise tehnoloogia [15].

Antud probleemi lahendab efektiivselt mikroteenustel põhineva arhitektuuri kasutamine, kuna rakenduse funktsionaalne skoop on väga väike, siis globaalne skoop on äärmiselt väike ning teiste teenustega suhtlemiseks ei ole muud võimalust kui valitud integratsioonitehnoloogia. Muidugi ei takista miski ka sel juhul kasutada globaalset andmebaasi oleku hoidmiseks, kuid see juba läheks vastuolla mikroteenuste arhitektuuri kasutamise eesmärkidega. Mikroteenuste arhitektuuriga kaasnev halduskeerukuse kasv on märksa kergemini käsitletav kui üle pea kasvanud tehniline võlg.

4.4 Testide põhine arendus (*Test-driven development*)

Üheks agiilsete arendusmeetodite ja ekstreemprogrammeerimise alustalaks on automaattestide kirjutamine. Testide põhine arendusmetoodika seisneb järgmistes põhimõtetes:

1. Funktsionaalsuse lisamine või täiendamine algab väikse arvu ühiktestide kirjutamisega, mis algselt kukuvad läbi, kuna funktsionaalsus ise on veel realiseerimata.
2. Realiseeritakse minimaalne vajalik funktsionaalsus testide õnnestumiseks.
3. Korrastatakse kirjutatud kood vastavalt kehtivale koodistandardile, kontrollides, et testid peale seda jäävad tööle.
4. Koodi muutmisel või teatud perioodilisusega käivitatakse kõik ühiktestid, veendumaks, et hiljutised arendused ei ole kõrvalmõjuna põhjustanud vigu muus funktsionaalsuses.

[14]

4.4.1 Võimalikud eelised

Põhilisteks testide põhise arenduse eelisteks on pikemas perspektiivis suurem kindlus tarkvara korrektses tööd ja modulaarsem kood. Kuna uue funktsionaalsuse lisamise või olemasoleva parandamisega võib kaasneda kõrvalmõjusid, mida algselt ei osata näha, siis kogu koodi testidega katvuse korral tuleb see kohe välja, kuna tarkvara ei rahulda kõiki automaatteste.

Samuti võimaldab see arendajatel koodi struktuuri ja erijuhte paremini läbi mõelda, kuna testtingimuste kirjeldamine juhib arendaja tähelepanu võimalikele veaolukordadele. Samuti kuna

ühiktest katab väga kitsast koodilõiku, siis suunab see lihtsama, paremini struktureeritud koodi kirjutamisele, mis oleks paradoksaalsel kombel ka ilma testideta paremini läbi mõeldud ja liigendatud.

Seega sobib automaattestide kasutamine suuremate ja stabiilsema domeeniloogikaga projektide arendamiseks.

4.4.2 Võimalikud puudused

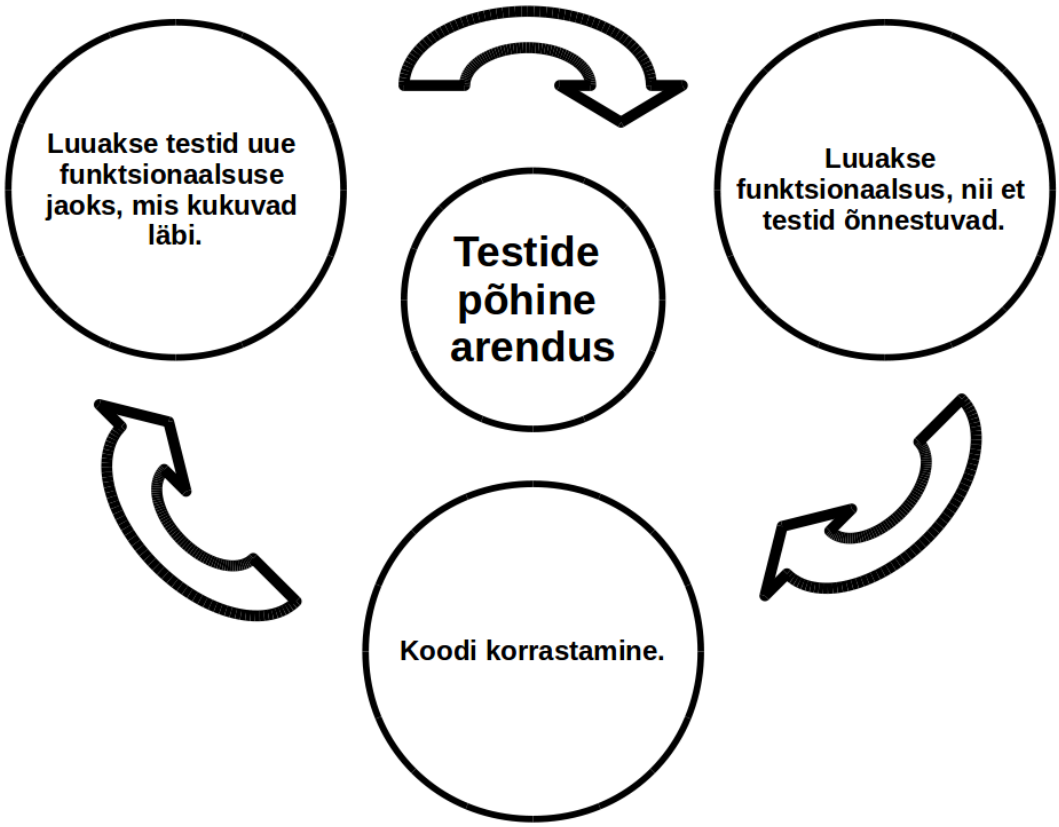
Põhiliseks testide kirjutamise hinnaks, ja ühtlasi ka põhjuseks miks neid enamasti ei kirjutata, on lisanduv ajakulu. Võib küll argumenteerida, et see aeg võidetakse tagasi hilisema kiirema arenduskiirusega, kuid see on tõsi ainult teatud juhtudel. Näiteks kui hiljem selgub, et teatud funktsionaalsus tuleb oluliselt muuta, siis esiteks läheb kaotsi esmase realisatsiooni testide kirjutamisele kulutatud aeg, ning lisaks valmib uus funktsionaalsus kauem, kuna lisaks koodile tuleb kirjutada ka uued testid.

Samuti tuleb silmas pidada, et ka testid on oma põhiolemuselt lähtekood ning nendes võib leiduda vigu või ei pruugi kõiki vajalikke stsenaariume läbi testida. Seega testide olemasolu küll suurendab kindlust, et kogu vajalik funktsionaalsus töötab ettenähtud moel, kuid ei taga seda. Ehk siis testimise põhiomadus, milleks on asjaolu, et testimine võib küll vigade olemasolu tuvastada, kuid ei taga nende puudumist, kehtib ka siin.

Lisaks testides leiduvatele vigadele, ei anna testid ka kindlust rakenduse sisulisele õigsusele. Ehk et kood võib vastata kõikidele nõuetele, sellele on kirjutatud põhjalik testide komplekt mis töötab ilma ühegi veata, kuid tarkvara ei tee seda mida kasutajad eeldavad või äriprotsess ette näeb.

4.4.3 Kokkuvõte

Nagu eelpoolkirjutatu mõista annab, on testide põhine arendus kompromiss ja sobib ehk teatud arenduskonteksti, samuti on post-agiilse arendusmeetodite poolt praktiseeritav ühiktestidest loobumine kompromiss, millel samuti omad eelised ja puudused. Seega ei saa kindlasti väita, et testidepõhine arendus tagaks alati parema tarkvaralahenduse väiksema ajaga. Kuna testide realiseerimine nõuab lisaaega funktsionaalsuse arendamiseks ja hilisemaks muutmiseks, mis pikendab funktsionaalsuse turule viimise aega, samal ajal kui programmeerijate anarhia korral on uue funktsionaalsuse turule toomise aeg lühike ning hilisem ümberkirjutamine samuti kiire ja odav, siis kindlasti on seda raske pidada ainuõigeks ja universaalselt sobivaks praktikaks.



Joonis 4: Testide põhine arendus

5. Tarkvara haldamise ja arendamise põhimõtted

Vastavalt eelkirjeldatud probleemi üldisemale olemusele ja kirjeldatud võimalikele lahendustele, tooksin järgnevalt välja erinevaid viise praktikas tarkvara arenduse efektiivsust tõstmiseks. Välja toodud lahendused põhinevad töö autori töökogemusele ning on aja jooksul ennast tõestanud.

Järgnevalt on silmas peetud, et kirjeldatavate põhimõtete rakendamine ei oleks aega-nõudev ja ei eeldaks mingi kindla profiiliga arendajaid, st kogemus ja kompetents ei sea suurel määral piire.

Antud peatükk sisaldab järgmisi alampeatükke:

- Loemind ehk README fail
- Lähtekoodi kvaliteedi tagamine
- Koodi kommenteerimine
- Tarkvara töökorras oleku kontroll
- Kontrollnimekirjad
- Riskid

5.1 Loemind ehk README fail

Nagu eelnevalt juba mainitud, on README fail sõnaline kokkuvõte olulisematest aspektidest tarkvara juures. Järgnevalt ülevaade erinevatest faili osadest.

5.1.1 Üldine

Mõne lausega kirjeldus, mis on tarkvara eesmärk ja funktsionaalsus, kes seda kasutavad. Lisaks lühike nimekiri kasutatavatest töövahenditest ja keskkondadest, et panna rakendus arendaja jaoks kiiresti tehnilisse konteksti. Näiteks mis programmeerimiskeelt ja versiooni kasutatakse, kui tegemist on veebirakendusega, siis mis konteineris või veebiserveris see on mõeldud töötama. Kui arenduseks kasutatakse raamistikke või kolmanda osapoole komponente, siis nende nimekiri ja versioonid.

5.1.2 Konfiguratsioon

Nimekiri kõikidest rakenduse konfiguratsioonifailidest, koos lühikese kirjeldusega ning asukohaga failisüsteemis. Kui konfiguratsioonifaili võib rakendusele ette anda mitmel moel, siis peaks kirjeldama mehhanismi kuidas see toimib. Oluline on siinjuures ka konfiguratsiooni tuvastamise järjekord. Kui konfiguratsiooni laadimine ei toimu vastavalt kasutatava raamistiku või platvormi konventsiooni kohaselt, siis tuleks eraldi märkida ka lõik lähtekoodis, kus konfiguratsiooni laadimine toimub.

Näiteks võib olla üks stsenaarium selline, et esmalt arvestatakse käsurealt ette antud konfiguratsiooni, kui seda ei leita, siis otsitakse vastavat parameetrit keskkonnamuutujast, kui ka sealt ei leita, siis kindlast tarkvara juures asuvast kaustast ning kõige viimaks kasutatakse sisse ehitatud vaikeväärtusi.

5.1.3 Välised liidesed

Milliseid väliseid liideseid rakendus pakub ja milliseid ise tarbib. Juurde tuleks lisada ka süsteemid

või kasutajate grupid, kes neid tarbivad või pakuvad ning millist tehnoloogiat nende juures kasutatakse. Näiteks SOAP või REST tüüpi veebiteenused, JMS sõnumivahetus, andmebaasipõhine integratsioon teiste süsteemidega, meilidel põhinev andmevahetus jne. Juurde tuleks lisada veel näitesõnumid või päringud, või siis viide kust neid võib leida.

Vajalik oleks lisada ka kirjeldus, kuidas antud teenuseid testida, veendumaks, et need on töökorras. Lisaks võib iga teenuse juures eraldi välja tuua viite seda realiseerivale kohale lähtekoodis.

5.1.4 Süsteemivälised protsessid

Rakenduse tööks on sageli vajalikud ka protsessid, mis töötavad rakendusest eraldi. Näiteks andmebaasi protseduurid ja päästikud ning operatsioonisüsteemi tasandil defineeritud tsüklilised tööd. On väga oluline, et need oleks kirjas vähemalt loemind failis, soovitatavalt tuleks neile viidata ka lähtekoodi osades, kus nendega arvestatakse.

5.1.5 Andmed

Eraldi lõik rakenduse poolt kasutatavate andmete käsitlemise kohta. See kirjeldab kuidas andmeid salvestatakse, näiteks kas andmebaasi, failidesse, indeksisse, üle võrgu teise süsteemi. Samuti on siin kirjas andmed vahemälu kohta, juhul kui see on kasutusel.

5.1.6 Arendusega alustamise juhend

Lühike juhend põhiliste sammudega, kuidas arendusega alustada ja arenduskeskkonda üles seada. Näiteks millised keskkonnamuutujad üle kontrollida, mida tähele panna konfiguratsiooni juures ja kuidas rakendust käivitada. Samuti üldisemad aspektid, nagu näiteks töövahendite kodeering, failisüsteemis vajalikele kaustadele kirjutamisõiguste määramine jne.

Siin asub ka kasutusjuhend järgnevates lõikudes kirjeldatud tarkvara töökorras oleku kontrolli kasutamiseks.

5.1.7 Konventsioonid

Juhul kui on kasutusel mingi lähtekoodi formaadistandard, siis ka see ära mainida README failis. Samuti kirjeldada ära failide nimetamise ja asukoha põhimõtted.

5.1.8 Koodi struktuur

Juhul kui rakendus ei kasuta mõne raamistiku või platvormi konventsiooni, siis tuleks eraldi välja tuua kogu koodi struktuur kas kaustade või pakettide kaupa. Kirjeldada lühidalt kuidas on lähtekood organiseeritud ning kuidas toimub andmevahetus erinevate moodulite või arhitektuursete kihtide vahel.

5.1.9 Tuntud vead, arendusvajadused

Selles osas on kirjeldatud tarkvaraga seotud teada olevad puudused, nii üldised kui ka konkreetsed. Need võivad olla tingitud teadlikust otsusest tehnilist võlga suurendada või ka tulenevad vigadest kasutatavates komponentides. Kui antud rakenduse arendus mõneks ajaks seisma pannakse ning arendajad mujale suunatakse, siis on oluline ära märkida võimalikke kitsaskohti arenduse juures. Samuti ei pruugi tehnilist laadi parenduste jaoks olla ametlikku inforuumi, nagu wiki või piletihaldus tarkvara, mis võib mõnikord sisaldada üksnes ärifunktsionaalsuse kohta käivaid andmeid või ainult konkreetsete vigade infot, siis on see sobiv koht selle kirjeldamiseks.

5.2 Lähtekoodi kvaliteedi tagamine

Koodi kvaliteeti on võimalik mehhaaniliselt tagada põhiliselt kahel viisil: lähtekoodi formaatimine vastavalt etteantud stiilireeglitele ning rakendades staatilist koodianalüüsi tuvastamaks tüüpvigu. Tuleb silmas pidada, et antud meetodid ei paranda tarkvara põhimõttelisi kompositsioonilisi ning algoritmilisi vigu, vaid üksnes väiksema ulatusega visuaalseid ning hooletusvigu. Samas konsistentselt rakendatuna annavad need summaarselt olulise efekti tarkvara hallatavuses ning töökindluses.

Kirjeldatud tegevusi on enamasti kerge integreerida ühe etapina automaatse ehitusprotsessi sisse. Selliselt on tagatud koodi pidev valideerimine kehtestatud nõuete vastu. Staatilise koodi analüüsi puhul on kasulik määrata piir, millest alates ehitusprotsess enne eeldefineeritud tasemega vigade kõrvaldamist ei õnnestu.

5.2.1 Koodi stiil

Koodi ühtne formaat võib esmapilgul tunduda vähetähtsana, kuid suuremas koodibaasis võimaldab see oluliselt vähendada visuaalset müra ning arendaja, eriti kellegi jaoks kes pole antud koodibaasiga varem töötanud, ning ta saab keskenduda koodi sisulisele funktsionaalsusele.

Nagu eelnevalt mainitud, on kõige lihtsam ja kiirem viis kasutada kasutusel oleva keele või töövahendi vaikestiili formaati. Teine viis on kasutada mõne suurfirma ametlikku stiilireeglistikku. Näiteks Java keelele on Oracle poolt välja antud ametlik stiilireeglistik ning Google pakub sellele alternatiivset formaati. Lisaks on levinud koodieditoris Eclipse vaikeformaadiks eelmainitutest erinev stiil.

On ka võimalus, et antud küsimus on arendaja eest lahendatud platvormi poolt, näiteks Google Go keele puhul on ametlik formaat keele välja töötajate poolt kindlaks määratud. Samuti paljude generatiivsete vahendite korral, näiteks Jade templeitide kompilaator NodeJS platvormil või CoffeeScript või muu sarnase transkompilaatori puhul on formaat eelnevalt määratud.

Ka lähtekoodi formaadi kontrolli on võimalik lisada ehitusprotsessi üheks etapiks. Juhul kui tarnitud kood ei vasta teatud kokkulepitud koodi formaadi reeglitele, siis ehitus ebaõnnestub. See on efektiivne vahend erinevate arendajate harjumustest või äranägemisest johtuvate lähtekoodi erisuste kõrvaldamiseks.

5.2.2 Staatiline koodianalüüs

Üldjuhul tuleks kohe projekti alguses või siis projekti sisenedes ja arenduskeskkonna üles sättimisel seadistada ka staatiline koodi analüsaator. Seda nii arendaja lokaalsesse töökeskkonda kui ka kesksesse ehitusserverisse ehitusprotsessi osana. Ajaliste piirangute tõttu võib selle esialgu käivitada vaikereeglitega ning tulemusi kasvõi ignoreerida, kuid siiski annab see lisainfot koodi hetkeolukorrast ning võimaldab kergemad hooletusvead kiiresti leida ja parandada. Pikemas perspektiivis annab see suure efekti, kuna praktikas on paljud raskesti leitavad vead oma olemuselt väga lihtsad, lihtsalt arendajad ei oska nende peale tulla ning projekti lõpufaasis võib situatsioon olla stressirikas ning keskendumist mitte soosiv. Staatiline koodianalüüs võimaldab lihtsalt ja vähese ajakuluga teatud veaklassid välistada. See omakorda võimaldab keskenduda teistele võimalikele (komplekssematele) veapõhjustele.

Juhul kui projektiga töötavad mitu meeskonda erinevatest firmadest, siis võimaldab staatiline koodianalüsaator teostada lihtsat koodi kvaliteedikontrolli. Mõlemad osapooled, nii arenduste

üleandja, kui ka vastuvõtja saavad enne kinnituse andmist veenduda, et teatud tüüpi vigu koodibaasis ei leidu. Selliselt hoitakse ära hilisemaid vaidlusi tarkvara lähtekoodi kvaliteedi üle, kuna vead avastatakse varakult. Sama asja on võimalik jälgida ka versioonihalduses, kuid sellisel juhul on see märksa aeganõudvam ja tülikam, kuna aktiivse arendustöö ajal võib lähtekood muutuda sageli ning seda muudavad mitmed arendajad. Isegi juhul kui lähtekoodi üleandmisel teostatakse koodi ülevaatus, siis teatud tüüpvigade eelnev välistamine hoiab kõikide osapoolte aega ja tähelepanu kokku ning võimaldab keskenduda olulisematele ja sisulisematele küsimustele.

Staatilise koodi analüsaatorid võimaldavad lisaks kaasa pandud reeglistikele defineerida ka kasutaja enda spetsiifilisi reegleid. Juhul kui projektis on mingeid spetsiifilisi nõudeid, mida on võimalik algoritmiliselt tuvastada, siis kindlasti tasub selle kohta eraldi reegel luua. See on hea viis automatiseerida midagi, millele muidu peaks keegi eraldi tähelepanu pöörama ning tarbetult aega kulutama. Lisaks projekti alguses defineeritud nõuetele võib pikemat aega töös olevates projektides aja jooksul silma jääda mingid tüüpilised veasituatsioonid, mis samuti on sobiv katta vastavasisulise staatilise analüüsi reegliga. Selle eelis ühiktestide ees seisneb selles, et ühiktestid testivad alati mingit konkreetset koodilõiku, kuid staatiline analüüs leiab reeglis kirjeldatud veaolukorra üle terve koodibaasi üles.

Tüüpiliselt klassifitseerib staatilise koodi analüüsi vahend vead nende olulisuse järgi kategooriatesse. Üheks otsustuskohaks tarkvara projektis ongi see, mis tasemel vigu veel aktsepteeritakse ja milliseid mitte. Juhul kui kasutatakse tarkvara ehitamisel ehitusserverit, siis on võimalik selles seadistada staatilise koodianalüüsi vigade tase, millest alates raporteeritakse ehitus ebaõnnestunuks.

5.3 Koodi kommenteerimine

Lähtekoodi kommenteerimine on üks koodi hallatavuse ja jätkusuutlikkuse tagamise võimalusi. Asjakohased ja õigesti paigutatud kommentaarid muudavad tarkvara lähtekoodi oluliselt loetavamaks, kuna väljendab koodi autori algseid kavatsusi ja paneb selle üldisesse konteksti, mis ei pruugi lihtsalt koodi enda põhjal alati ilmneda. Seetõttu on oluline kommentaaride lisamisel silmas pidada, et kommentaar ei kirjeldaks seda mida antud koodilõik teeb, vaid mis eesmärgil see on kirjutatud. Teiste sõnadega kommentaar ei peaks mitte vastama küsimusele kuidas, vaid miks.

Soovituslikult seovad ja täiendavad lähtekoodi kommentaarid loemind failis loodud tarkvara ülevaate ühtseks tervikuks. Selliselt on võimalik suvalises programmeerimiskeskkonnas saada osa kirjandusliku programmeerimise põhilisest eelisest - inimloetav programmi kirjeldus. Antud lähenemise korral küll ei teki genereeritavat tekstidokumentatsiooni tarkvara jaoks, kuid see ei omagi nii suurt praktilist väärtust kui hästi dokumenteeritud kood ise.

Samas tuleb silmas pidada, et liigsed ja sisutud kommentaarid tekitavad visuaalset ja semantilist müra nagu halvasti struktureeritud koodki ning ebatõesed ja mitteajakohased kommentaarid töötavad oma eesmärgile vastu, kuna tegelik funktsionaalsus mitte ainult ei ilmne lähtekoodi enda mõistmisel, vaid lisaks tuleb ka tuvastada vead kommentaarides [7].

Seetõttu tuleks kommentaaride lisamisel põhiliselt eelistada kontseptuaalseid ja kavatsuslikke kommentaare, ehk et kuidas antud klass või moodul sobitub üldisesse süsteemi ja millist probleemi on sellega kavatsetud lahendada. Meetodi ja ka keerulisemate konstruktsioonide kommentaare võib muidugi ka lisada, kuid reaalselt on neid raske hallata. Seega kui koodi muutmisel mingi kommentaar ei ole enam täiesti korrektne, tuleks see pigem kustutada ning veenduda, et kõrgema

taseme kommentaar oleks endiselt tõene.

Juhul kui tarkvara loomisel on olemas analüüs või muu nõuete loetelu, siis võib kommentaarides viidata konkreetsetele nõuetele, mida antud koodilõik lahendab. Selliselt on võimalik hõlbustada analüüsist tulenevate nõuete realiseerituse kontrollimist. Ühtlasi on koodi muutmisel võimalik tuvastada, kas muudetud kood samuti kõiki nõudeid täidab.

Samuti on hea taktika integreerida kommentaarid koodi sisse, näiteks anda muutujatele ja meetodi nimetustele kirjeldavad nimetused, mida on tulevasel muutjal raskem ignoreerida. Mõistlik on nõuda vähemalt klassi tasemel kommenteerimist ning nende olemasolu on võimalik staatilise koodianalüüsi käigus tuvastada.

5.4 Tarkvara töökorras oleku kontroll

Üks lihtne kuid tõhus viis nii arendajatele arenduskeskkonna üles seadmisel kui ka rakenduste administraatoritele tüüpvigade vältimiseks või varaseks avastamiseks on luua eraldi kontrollmehhanism rakenduse tööks vajalike eelduste testimiseks. See võib olla eraldi väike rakendus või skript, või võib ka olla integreeritud rakendusse sisse.

Sellise mehhanismi loomise ajakulu on äärmiselt väike võrreldes kogu tarkvara elutsükli ajalise mahuga. Millele lisandub ka potentsiaalselt tulevikus erinevate konfiguratsiooni ja muude rakenduse tööks vajalike eelduste puudustest tulenevate vigade uurimisele kulutamata jäetud aeg. Toodangukeskkonnas esinevate vigade analüüsimine toimub sageli närvilises õhkkonnas ja lihtsad võrguühenduse või operatsioonisüsteemi seadistusvead jäävad kergesti kahe silma vahele. Samas siinkirjeldatud kontrollmehhanism aitab kiiresti veenduda põhiliste tööks vajalike eelduste olemasolus. Seega on võimalik kiiresti luua baasteadmine, mis toimib ja suunata tähelepanu teistele aspektidele infrastruktuuris.

Enamasti on vajalikud kontrollid rakendustes sarnased, seega on võimalik koodi lihtsalt erinevate rakenduste vahel jagada ja sellega veelgi ajakulu vähendada.

Tuleb tähele panna, et kontrollmehhanismi loomine ei ole ühekordne tegevus, mida tehakse süsteemi loomise käigus, vaid seda tuleb kaasajastada vastavalt süsteemi muutustele. Selliselt tagatakse, et eemaldatakse mittevajalikud kontrollid ning lisatakse uued, lisatud funktsionaalsusele vastavad. On suhteliselt tõenäoline, et arenduse käigus lisandunud sõltuvused jäävad erinevatesse keskkondadesse paigaldamisel kommunikatsioonitõrgete tõttu kahe silma vahele, ning põhjustavad hooldusakende venimist ning segadust vigade tuvastamisel. Vastava kontrollmehhanismi olemasolul on võimalik sedalaadi probleeme vähendada.

Organisatsioonides tegelevad tüüpiliselt serverite ja rakenduste haldamise ja käigushoidmisega teised inimesed kui arendusega. Sageli on rakenduse haldajad ja arendajad ka erinevates organisatsioonides. Seega ei ole haldusinimesed sageli tarkvara juures kasutatavate tehnoloogiate ja raamistikega nii hästi kursis kui arendajad. Kontrollmehhanism võimaldab käsitleda mingi konkreetse raamistiku spetsiifilisi vigu ja väljastada üldisemaid, administraatorile selgemaid veateateid. Lisada võib ka lühikese juhendi, kuidas ilmnenud viga kõrvaldada.

Järgnevalt tuuakse ära tüüpilised asjad, mille kontrollimist võib kaaluda:

- Võrguressursside kättesaadavus
    - Andmebaas

    - Sõnumiserver
    - Veebiteenused
- Kõik rakenduse tööks vajalikud välise konfiguratsiooni elemendid
- Failisüsteemi kontrollid
    - Kas vajalikud kaustad ja failid eksisteerivad
    - Kas rakendusel on vajalikele kaustadele ja failidele kirjutamisõigus, näiteks logifailid, indeksifailid

Põhimõtteliselt on võimalik kirjeldatud kontrolle realiseerida ka integratsioonitestidena, ja kindlasti need ei välista üksteist, kuid tüüpiliselt on integratsioonitestide käivitamiseks vajalik arenduskeskkond koos vajalike vahenditega, mida tarkvara lõpptoodangu paigaldamise juures ei ole. Samuti ei ole tarkvara paigaldamisel vajalik kõikide integratsioonitestide käivitamine. Seega on eraldiseisev tarkvara kontroll käepärasem ja lihtsam.

5.4.1 Näide Grails raamistikus

Illustreeriva näitena on järgnevalt toodud koodilõik Grails raamistikus realiseeritud konfiguratsiooni kontrollimine rakenduse käivitamisel. Antud näiteks kontrollitakse rakenduse tööks vajalike JMS järjekordade ja andmekanali parameetrite olemasolu. Juhul kui need ei ole seadistatud, siis rakenduse käivitamine peatatakse ja logisse väljastatakse vastav teade.

```
final class ConfigurationCheck {

    private static def config = Holders.config
    private static Log log = LogFactory.getLog(this.class)

    public static void check() {

        log.info("Väliste konfiparameetriet kontroll!")

        // JMS kanalid
        List<String> confErrorList = new ArrayList<String>()
        if( ! config.queues.staatus) confErrorList.add("Staatuste JMS queue on
konfis seadistamata, lisa see 'queues.staatus' parameetrile!")
        if( ! config.queues.syndmus) confErrorList.add("Sümduste JMS queue on
konfis seadistamata, lisa see 'queues.syndmus' parameetrile!")
        if( ! config.queues.teade) confErrorList.add("Teadete JMS queue on konfis
seadistamata, lisa see 'queues.teade' parameetrile!")

        // Andmekanali parameetrid
        if(config.datalink.enabled == null) confErrorList.add("Andmekanali
käivitamine seadistamata! Lisa datalink.enabled parameeter! (true/false)")
        if( ! config.datalink.host) confErrorList.add("Andmekanali host
seadistamata! Lisa datalink.host parameeter! (IP või url)")
        if( ! config.datalink.port) confErrorList.add("Andmekanali port
seadistamata! Lisa datalink.port parameeter! (number, näiteks 50000)")

        if(confErrorList.size() > 0) {
            log.fatal "------------------------------KONFI
VEAD---------------------------------"
            confErrorList.each { errorStr ->
                log.fatal errorStr
            }
            log.fatal
"--------------------------------------------------------------------------------"
            System.exit(1); // Puuduliku konfiga rakendus ei lähe tööle!
        }

        log.info("Konf ok :)")
    }
}
```

Koodinäide 1: Rakenduse konfiguratsiooni kontroll Grails raamistikus

5.5 Kontrollnimekirjad

Paljudes valdkondades, näiteks meditsiinis, ehituses, lennunduses jne, on levinud praktikaks kasutada erinevate tegevuste läbiviimisel kontrollnimekirju, et vältida tüüpvigu ning inimlikke eksitusi. Rutiinsete tegevuste korral tagab see et midagi ei jäeta kahe silma vahele, mis võib põhjustada asjaolude kokkulangemisel raskesti selgitatavaid veaolukordi, näiteks meditsiinis operatsiooniks valmistumisel. Harva esinevate keeruliste situatsioonide korral võivad kontrollnimekirjad välja pakkuda ka optimaalse lahenduskäigu, näiteks lennuki mootoririkke korral. Mitmetes elualades on nende kasutamine kohustuslik ning nende täitmisele kehtivad omakorda täiendavad reeglid [6].

Kuigi tarkvara arenduses on kontrollnimekirjade kasutamine vähe levinud, siis nende kasutamine pakub samu turvamehhanismi mis igal pool mujalgi. Nende laialdasem rakendamine aitab samuti nii erinevate projekti tegevuste standardiseerimisel, tüüpvigade vältimisel, tarkvaratarnete

edastamisel ja vastuvõtmisel, erinevates paigaldus ja juurutusfaasi tegevustes ning uute arendajate projekti kaasamisel.

Kontrollnimekirjade kasutamine annab seda suuremat efekti, mida suurem on organisatsioon, kuigi mõnest inimesest koosnevas meeskonnas on võimalik teatud situatsioonides saavutada olulist kasu. Samuti on sageli teatud tegevusliinides aja jooksul välja kujunenud teatud suuline kontrollnimekirja laadne tegevuste järjekord. Kuid selliselt on sellest oluliselt vähem kasu, kuna seda on raske kommunikeerida, inimesed võtavad selle omaks pika aja ja eksituste tagajärjel. Sageli asuvad erinevad osapooled erinevates organisatsioonides ja füüsilistes asukohtades, mistõttu teadmine ei kandu efektiivselt edasi.

Kontrollnimekirjade eelis pikkade kordade ning eeskirjade ees seisneb nende konkreetsuses ja koheses rakendatavuses. Isegi juhul kui organisatsioonis on protsessid formaalselt dokumenteeritud ning isegi kui kõik osapooled on sellega tutvunud, siis esineb nende rakendamisel sageli veaolukordi, kuna inimesed ei suuda sobival hetkel keerukat tervikprotsessi haarata. Kontrollnimekirja puhul on eeldused, tegevused ja kontrolltingimused õiges järjekorras ja kohe teostatavate sammudena ära toodud.

5.6 Riskid

Järgnevalt kirjeldatakse mõningaid riske, mida silmas pidada eelnevalt kirjeldatud tarkvara arenduspraktikate rakendamisel.

Tarkvara dokumenteerimisel, nii loemind faili, tehnilise dokumentatsiooni, kasutusjuhendite kui ka koodi kommentaaride puhul on kõige suuremaks riskiks nende aja- ja asjakohasus. Juhul kui dokumentatsioon ei ole täielik või on vale, võib tekkida vigu ja valearvestusi mitmel tasemel. Ühtlasi võib see täiendavalt tekitada soovi seda ignoreerida ja mitte ajakohastada. Seega on oluline dokumentatsiooniga tegeleda järjepidevalt ning selle jaoks kasvõi eraldi aega planeerida.

Sarnaselt dokumentatsioonile on ka tarkvara korrasoleku kontrollmehhanismi juures oluline järjepidavus, et see kasvaks ja muutuks koos tarkvara enda arendusega.

Staatiline koodianalüüs ei kätke endas mingeid märkimisväärseid riske, mida suuremal määral seda kasutatakse, seda rohkem sellest ka kasu on võimalik saada.

Üks võimalik tegevus, mis sageli annab häis tulemusi, on projekti alguses määratleda oodatavad riskid ja perioodiliselt hinnata koodis nende realiseerumise astet ning täiendada ka riskide nimekirja ennast. Sellisel juhul ei kao tehnilise võla määr fookusest ära ning arendusega mitteseotud osapooltel on indikatsioon, mis seisus koodibaas mingil ajahetkel on. Samuti loob see pikemas perspektiivis teatud tunnetuse mittetehnilistele inimestele tehnilise võla mõju kohta.

6. Kokkuvõte

Antud töö püüdis näidata, et vaatamata uutele arendusmetoodikatele, mis teatud kontekstis täidavad küll oma eesmärki, ja tehnoloogiatele nagu NoSQL, mikroteenused, REST liidesed, raamistikud, testide põhine arendus (TDD), ei ole tulenevalt tarkvara olemuslikust keerukusest vigade arvu võimalik olulisel määral vähendada ega arenduse kiirust tõsta. Pigem võib täheldada vastupidist trendi, kuna erinevaid tarkvara rakendus- ja kasutusvaldkondi tuleb pidevalt juurde, ning sellega seoses nõuded tarkvarale järjest suurenevad, siis süsteemide komplekssus ning vajadus liidestuda teiste süsteemidega samuti üha kasvab. Erinevate tarkvara olekute ja potentsiaalsete probleemide hulk ei kasva mitte lineaarselt, vaid eksponentsiaalselt. Kui siia lisada veel järjest kõrgematasemelistest programmeerimiskeeltest ja erinevate arendusraamistike kasutamisest tulenev tarkvara arenduse liikumine järjest kõrgematele abstraktsioonitasemetele, siis tuleb ilmselt eduks lugeda seda, kui tarkvara arenduse produktiivsus ja töökindlus ei lange suurusjärgu võrra.

Teiste sõnadega ei ole ka praegu, nagu ka 30 aastat tagasi, näha mingit nö „hõbekuuli”, ehk siis tarkvara arenduse metoodikat või vahendeid, mis võimaldaks arenduse produktiivsust või töökindlust suurendada kasvõi ühe suurusjärgu võrra. Tarkvara olemuslik keerukus ei kao kuhugi, erinevad arendusmetoodikad ja tehnoloogiad lihtsalt teevad erinevaid kompromisse ja viivad selle erinevatesse kohtadesse arendusprotsessis ja käsitlevad seda erinevalt. Kusjuures keerukus ei seisne üksnes tehnoloogilistes küsimustes, vaid hõlmab ka organisatsioonilisi ja ärilisi aspekte.

Käesoleva töö autori hinnangul, mis põhineb tema pikaajalisel kogemusel tarkvara arenduses, on realistlik ja praktiline tegevusstrateegia sellises olukorras keskenduda pigem inimlikele aspektidele tarkvara arendamisel mitte tehnilistele. Ehk tarkvara kirjutatakse teisi inimesi, mitte masinaid silmas pidades. Selleks on võimalik rõhku panna semantilise kommunikatsiooni parandamisele, kasutada järjepidevalt mehaanilisi viise koodi kvaliteedi ja visuaalse müra piiritlemiseks, luua erinevaid kontrolle mis väljastavad sihtgruppi silmas pidavaid selgeid veateateid, automatiseerida võimalikult suur osa rutiinseid tegevusi, et avastada eksimusi võimalikult vara ning võimalusel välistada potentsiaalseid tüüpvigu.

Pikas perspektiivis on selliselt võimalik tõsta arenduskiirust, tagada parem muudatuste planeeritavus, tõsta tarkvara ekspluatatsiooniaega ning muuta tarkvara arendus märksa jätkusuutlikumaks kui see enamasti praegu on.

6.1 Kas eesmärgid saavutati?

Töö eesmärgiks oli analüüsida erinevaid põhjuseid, miks tarkvara hallatavus ja prognoositavus teatud arvu arendusiteratsioonide möödudes oluliselt väheneb ja vaadelda selles kontekstis mõningaid kaasaegseid arenduspraktikaid ja tehnoloogiaid. Teiseks peamiseks eesmärgiks oli vaadelda mõningaid maailmapraktikaid ja võimalusi antud probleemidega tegeleda ning lõpuks kirjeldada komplekt arenduspraktikaid mis võimaldaks kiiresti omandada ülevaate tarkvara struktuurist, sõltuvustest ja tööloogikast. Tõstes sellega tarkvara arendatavust ja hallatavust.

Kõik eesmärgid võib lugeda saavutatuks. Töös kirjeldati erinevaid koodi kvaliteedi languse ja tehnilise võla kasvu soodustavaid mõjusid. Samuti vaadeldi levinud arendusmetoodikaid ja tehnoloogiaid selles kontekstis ning kirjeldati viise kuidas selliste probleemidega tegeleda. Kirjeldati ka töö autori praktikal põhinevaid viise tarkvara hallatavuse ja arendatavuse tõstmiseks.

7. Summary

This work attempted to show and conclude that despite of the new software development methodologies, that have their advantages in certain situations, and new technologies, like NoSQL, REST interfaces, frameworks and test driven development(TDD), there is still nothing that could increase the development speed, reliability or simplicity of software systems even one order of magnitude. Rather it could be argued that the opposite trend is true, since the usage and deployment of software expands in ever increasing rate and it causes the requirements for software to increase also accordingly. And thus the complexity and the need to interface with other systems also increases, causing the count of different states in a system and potential errors to rise in exponential scale. When the tendency to increase the abstraction level of programming tools is also considered, it should be counted as success when the productivity and reliability is not declining.

In other words today, like 30 years ago, there is still no "silver bullet", that is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement within a decade in productivity, in reliability or in simplicity. The essential complexity in software does not disappear nor is it more manageable than 30 years ago, different methodologies and technologies only handle it differently and make different compromises. The problems are further amplified by the complexities of the business environment and processes in organizations that are conducting the development.

In the opinion of the author of this work, due to circumstances described in this document, it is realistic and practical to focus rather on more efficient human communication and other human aspects of software development than technical subtleties. In other words the software development should primarily focus on humans, not on machines. To do so it is possible to enhance semantic communication and descriptiveness in software, rigorously apply mechanical code quality tools and strive to reduce visual noise in source code, create different kind of integrity checks that output clear and instructive error messages and automate as much routine tasks as possible. The latter will help to discover potential error situations more early and to eliminate common mistakes.

It is likely that doing so will help to maintain development speed, predictability, software life-span and general sustainability of development.

8. Kasutatud kirjandus

1. „Big Ball of Mud”, Brian Foote, Joseph Yooder, 1999
2. "No Silver Bullet - Essence and Accident in Software Engineering", Frederick P. Brooks, Jr., 1987
3. „The Mythical Man-month”, Frederick P. Brooks, Jr., 1995
4. „Human Javascript”, Henrik Joreteg, 2014, http://read.humanjavascript.com
5. „NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence”, Pramod J. Sadalage, Martin Fowler, Addison-Wesley Professional, 2012
6. „The Checklist Manifesto: How to Get Things Right”, Atul Gawande, Picador, 2011
7. „Clean Code: A Handbook of Agile Software Craftsmanship”, Robert C. Martin, 2008
8. „AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis”, William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, Thomas J. Mowbray, 1998
9. „Principled Design of the Modern Web Architecture”, Roy T. Fielding, Richard N. Taylor, 2002
10. „Leaner Programmer Anarachy”, Fred George, Antonio Terreno, 2010
11. „Literate Programming”, Donald E. Knuth, 1984, The Computer Journal (British Computer Society) 27 (2): 97–111.
12. „An Empirical Model of Technical Debt and Interest”, Ariadi Nugroho, Joost Visser, Tobias Kuipers, 2011
13. „An overview on the Static Code Analysis approach in Software Development”, Ivo Gomes, Pedro Morgado, Tiago Gomes, Rodrigo Moreira, 2009
14. „Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies”, Thirumalesh Bhat, Nachiappan Nagappan, 2006
15. „Working Effectively with legacy code”, Michael C. Feathers, 2005, Tenth print 2009, Pearson Education, Inc.
16. Tiobe Index, [WWW] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 03.12.2014
17. Golang, Wikipedia, [WWW], https://en.wikipedia.org/wiki/Golang, 03.12.2014
18. Two years of programmer anarchy, [WWW], http://the-arm.com/two-years-of-programmer-anarchy/ , 03.12.2014
19. README, Wikipedia, [WWW], https://en.wikipedia.org/wiki/README , 03.12.2014
20. Markdown, Daring Fireball, [WWW], http://daringfireball.net/projects/markdown/ , 03.12.2014
21. Markdown, Wikipedia, [WWW], https://en.wikipedia.org/wiki/Markdown , 03.12.2014
22. Code Conventions for the Java Programming Language, Oracle, [WWW],

http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html , 03.12.2014

23. Golang stiilijuhend, [WWW], http://golang.org/doc/effective_go.html , 03.12.2014
24. Google Java Style, Google, [WWW], https://google-styleguide.googlecode.com/svn/trunk/javaguide.html , 03.12.2014
25. Functional Programming, Wikipedia, [WWW], http://en.wikipedia.org/wiki/Functional_programming , 03.12.2014
26. Literate programming, Wikipedia, [WWW], https://en.wikipedia.org/wiki/Literate_programming , 03.12.2014
27. Ruby, Wikipedia, [WWW], https://en.wikipedia.org/wiki/Ruby_(programming_language) , 03.12.2014

8.1 Standardid, tehnoloogiad, töövahendid

1. Eclipse, [WWW] http://www.eclipse.org/ , 03.12.2014
2. Apache Maven, [WWW] https://maven.apache.org/ , 03.12.2014
3. Grails, [WWW] https://grails.org/ , 03.12.2014
4. GitHub, [WWW] https://github.com/ , 03.12.2014
5. BitBucket, [WWW] https://bitbucket.org/ , 03.12.2014
6. Git, [WWW] http://git-scm.com/ , 03.12.2014
7. FindBugs, [WWW] http://findbugs.sourceforge.net/ , 03.12.2014
8. CodeNarc, [WWW] http://codenarc.sourceforge.net/ , 03.12.2014
9. NodeJS, [WWW] http://nodejs.org/ , 03.12.2014
10. Jade templeidid, [WWW] http://jade-lang.com/ , 03.12.2014
11. CoffeeScript, [WWW] http://coffeescript.org/ , 03.12.2014