LATVIJAS UNIVERSITĀTE
FIZIKAS UN MATEMĀTIKAS FAKULTĀTE
DATORIKAS NODAĻA

TESTĒŠANAS PROCESU ANALĪZE

MAĢISTRA DARBS

Autors: Līga Kļavinska
Stud. apl. nr: Mate 020011
Darba vadītājs: Dr.sc.comp. Jānis Bičevskis

Rīga 2008

ANOTĀCIJA

Darbā tiek izpētīts un analizēts konkrēta uzņēmuma sistēmu izstrādes modelis un programmatūras testēšanas dzīves cikls. Tiek apskatīta vienībtestu ietekme uz produkta kvalitāti programmatūras izstrādes dzīves ciklā, analizējot uzņēmuma divu līdzīgas sarežģītības projektu atrasto kļūdu daudzumu, prioritāti un kritiskumu, kā arī kļūdu un darāmo darbu atgriešanu biežumu programmētājam. Vienam no projektiem tika izstrādāti vienībtesti, bet otram – nē. Darbā konstatēts, ka vienībtestu ieviešana nav būtiski uzlabojusi projektu gaitu. Tiek piedāvāti izpētītās statistikas skaidrojumi un iespējamie uzņēmuma testēšanas procesu uzlabojumi.

ANNOTATION

System development model and software testing life cycle of specific company is analyzed and described in this paper. Unit testing influence on quality of product during software developing life cycle is analyzed. Two equally complex projects of the same company are chosen and compared to each other, where only in one of the projects unit tests are developed. Projects are compared by count of failures found during the project, by priority and severity of failures as well as count of returned failures and features to programmers. There was discovered, that unit testing has not given any significant improvement. Results of gathered statistics are explained and possible improvement of testing processes are given.

AUTOREFERĀTS

Darba autore uzņēmumā, kurā veikts maģistra darba pētījums, strādā par testētāju un piedalās testēšanas procesu uzlabošanas izstrādē. Viens no uzlabojumiem ir vienībtestu ieviešana izstrādes procesā. Darba autore ir piedalījusies gan projektu, kuros nebija veikti vienībtesti, gan projektu, kuros tie bija veikti, testēšanā.

Lēmums par to, ka vienībtesti jāievieš uzņēmumā tika pieņemts, bet uzņēmumā nenotika šī lēmuma ietekmes uz projektu gaitu analīze. Šī maģistra darba mērķis bija izpētīt vienībtestu ietekmi uz uzņēmuma projektu izstrādes gaitu, kā arī izpētīt uzņēmuma testēšanas procesus kopumā un piedāvāt uzlabojumus.

Darba autore izpētīja uzņēmuma programmatūras izstrādes modeli un salīdzināja to ar teorētisko V-modeli, kam uzņēmumā lietotais modelis visvairāk līdzinās. Lai novērtētu vienībtestu ietekmi, tika izvēlēti divi sarežģītības ziņā līdzvērtīgi projekti, kur vienam no projektiem tika izstrādāti vienībtesti, bet otram – nē. Autore apkopoja šo projektu kļūdu un darāmo darbu statistiku no kļūdu un darāmo darbu reģistrēšanas sistēmas un salīdzināja abu projektu rezultātus.

Statistikas pētījums deva pārsteidzošus rezultātus – vienībtestu ieviešana šajos projektos nedeva praktiski nekādus uzlabojumus. Autore meklēja iemeslus, kāpēc radusies šāda situācija, kas ir pretrunā ar teoriju. Radās trīs iespējami izskaidrojumi, kur viens no tiem ir šāds – maģistra darbā pētāmais projekts bija pirmais, kurā tika izmantoti vienībtesti, kā arī projektā bija iesaistīti programmētāji, kas tikko bija sākuši darbu uzņēmumā, tāpēc šajā projektā bija lielāks risks, ka programmētāji nezināšanas pēc var nepareizi uzprogrammēt gan pašus darāmos darbus, gan arī to vienībtestus.

Otrais izskaidrojums ir tāds, ka uzņēmumā parasti tiek veidoti līdzīgi projekti un pamatdarbības tiek pārņemtas no citiem iepriekšējiem projektiem. Tas nozīmē, ka pamatdarbību vienībtestēšana nedod gandrīz nekādu labumu, jo tās ir pārņemtas no iepriekšējiem projektiem jau notestētas.

Trešais izskaidrojums ir tieši šo divu salīdzināšanai izvēlēto projektu specifika – tā kā projekti bija ļoti līdzīgu produktu izstrāde diviem dažādiem klientiem, tad projekts, kurā tika veikti vienībtesti, ļoti daudz ko aizguva no iepriekš izstrādātā projekta, kurā vienībtesti izstrādāti netika. Kā iepriekš pieminēts, tad tā ir izplatīta prakse šajā uzņēmumā, bet šajā gadījumā aizgūta tika ne tikai pamatdarbību funkcionalitāte, bet arī ļoti daudz no

papildfunkcionalitātes.

Ņemot vērā uzņēmuma specifiku, ka daļa no funkcionalitātes tiek aizgūta no agrāk veiktajiem projektiem, autore piedāvā uzņēmumā samazināt vienībtestēšanas lomu, vienībtestēšanu veikt tikai tām komponentēm, kas ir programmētas no jauna, kā arī paātrināt integrācijas testēšanas sākumu un veikt plašākus integrācijas testus.

SATURS

TABULU SARAKSTS

ATTĒLU SARAKSTS

IEVADS

Programmatūras izstrādes procesā testēšanai ir īpaša loma, tā sastāda no 25% līdz pat 50% no izstrādes laika un izmaksām. Daudzi programmatūras izstrādes uzņēmumi izmēģina arvien vairāk un vairāk testēšanas metožu, lai samazinātu testēšanas laiku un izmaksas, kā arī palielinātu testēšanas efektivitāti un kvalitāti.[1]

Arī uzņēmums, kas tiek apskatīts šajā darbā, mēģina optimizēt programmatūras izstrādes, tai skaitā arī testēšanas, procesus. Viens no iespējamiem risinājumiem ir vienībtestu ieviešana ,tādējādi testēšana sākas jau agrīnā izstrādes stadijā un, kā zināms, jo ātrāk tiek atrasta un novērsta kļūda, jo mazākas ir tās izmaksas. Uzņēmumā tika pieņemts lēmums par vienībtestu ieviešanu un pēc virspusīgas projektu gaitas izpētes, šķita, ka izstrādes process, patērētais laiks un izmaksas ir nedaudz uzlabojušies, un tika nolemts, ka turpmāk visos projektos tiks izmantoti vienībtesti, bet sīkāka projektu gaitas analīze netika veikta.

Maģistra darba mērķis ir izanalizēt uzņēmuma testēšanas dzīves ciklu, izpētīt vienībtestu ietekmi uz izstrādes, sevišķi testēšanas, gaitu, kā arī piedāvāt uzlabojumus uzņēmuma testēšanas procesā.

Darbs sastāv no trim nodaļām, kur pirmajā nodaļā apskatīts uzņēmuma programmatūras izstrādes modelis. Nodaļā aprakstīts teorētiskais V-modelis, kas kopumā atbilst uzņēmuma programmatūras izstrādes modelim, kā arī uzņēmuma atšķirības no vispārpieņemtā V-modeļa.

Otrajā nodaļā pētīta vienībtestu ietekme uz projekta gaitu. Šim nolūkam uzņēmumā izvēlēti divi līdzīgas sarežģītības un apjoma projekti, kur vienam no projektiem tika veikti vienībtesti, bet otram – nē. No uzņēmuma kļūdu un darāmo darbu reģistrēšanas un uzturēšanas sistēmas ir apkopota testēšanas statistika par abiem projektiem un tie salīdzināti savā starpā. Par salīdzināšanas kritērijiem izvēlēti kļūdu atrašanas biežums pa projektu laika posmiem, kļūdu prioritāte, kļūdu kritiskums, kā arī kļūdu atgriešanas programmētājam biežums. Darbā salīdzināts arī darāmo darbu un izmaiņu pieprasījumu atgriešanas programmētājam biežums.

Otrajā nodaļā apkopotās statistikas rezultāti ir pārsteidzoši – tie parāda, ka šajos divos projektos vienībtestu ietekme praktiski nav jūtama – abos projektos kļūdu skaits un kritiskums ir līdzvērtīgi. Trešajā nodaļā analizēts, kāpēc vienībtestu ietekme izvēlētajos

projektos nav tik liela, kā tai vajadzētu būt. Tiek meklēti vienībtestu mazās ietekmes iemesli un iespējamie risinājumi, kā uzlabot testēšanas un izstrādes gaitu, lai testēšana uzņēmumā kļūtu efektīvāka.

1 PROGRAMMATŪRAS IZSTRĀDES MODEĻI[1] [2] [5]

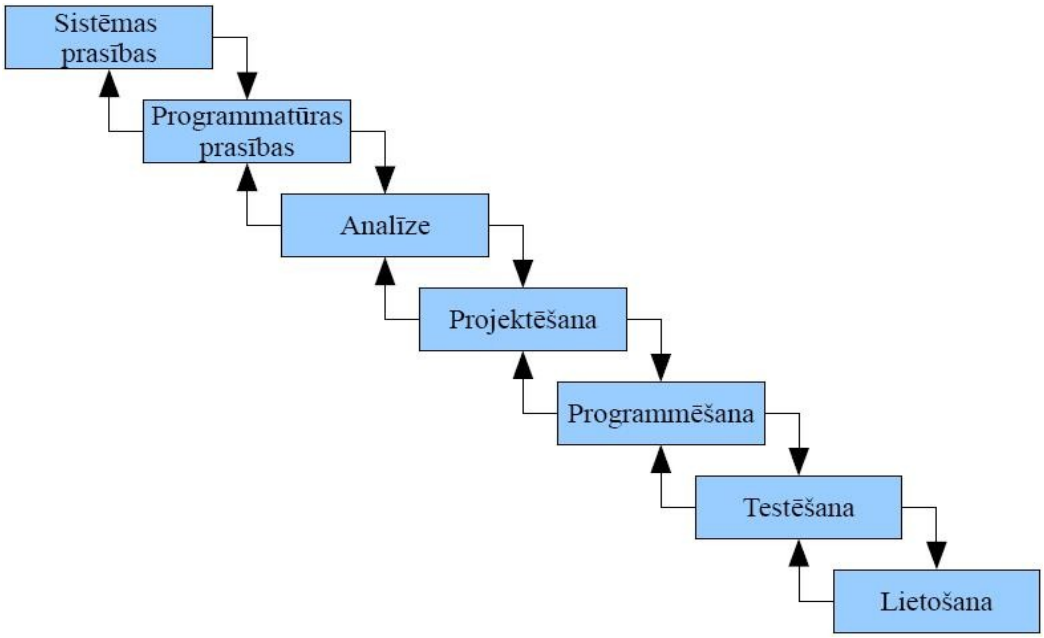
Lai sasniegtu strukturētu un kontrolējamu programmatūras izstrādi, tiek izmantoti programmatūras izstrādes modeļi un programmēšanas procesi. Eksistē daudz dažādu modeļu, piemēram, Ūdenskrituma modelis, V-modelis, Spirāles modelis, kā arī spējās programmēšanas modeļi, piemēram, ekstrēmās programmēšanas modelis.

Katrā no šiem modeļiem ir aprakstīts savs veids, kādā panākt sistemātisku un sakārtotu darba secību, kas jāievēro, izstrādājot programmatūru. Vairumā šo modeļu ir definētas izstrādes fāzes un sistēmas projektēšanas soļi. Fāzes pabeigšana, kas parasti tiek noteikta kā šķirtne, ir pienākusi tad, kad visi fāzē nepieciešamie izstrādes darbi vai dokumenti ir pabeigti un atbilst iepriekš definētiem kvalitātes kritērijiem.

Ar izstrādes modeļu palīdzību var veikt detalizētu resursu (laika, cilvēkresursu, infrastruktūras) plānošanu.

Testēšana parādās visos šajos modeļos, bet tai ir ļoti dažāda nozīme un apjoms. Kā piemērs tiks salīdzināti Ūdenskrituma (*Waterfall*) modelis un V-modelis (*V-model*).

Pirmais programmatūras izstrādes modelis ir vienkāršais un labi zināmais ūdenskrituma modelis, kura fāzes redzamas zīmējumā Zīmējums 1: Ūdenskrituma modelis



Zīmējums 1: Ūdenskrituma modelis

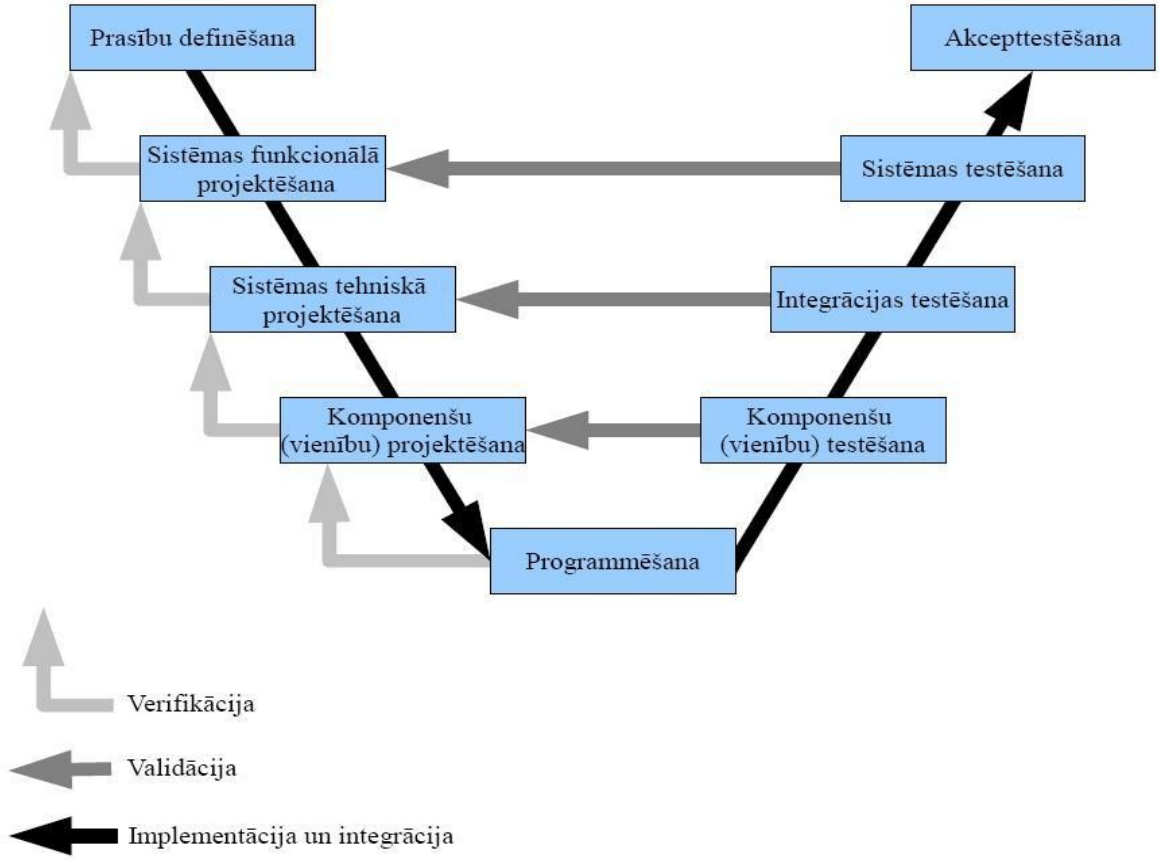
Tikai tad, kad viens no izstrādes līmeņiem ir noslēdzies, sākas nākošais līmenis. No katra līmeņa var nokļūt tikai vienu līmeni uz priekšu (ja līmenī visas prasības izpildītas) vai atpakaļ (Ja līmenī atklātas kādas nepilnības, kā dēļ nepieciešams pārskatīt kādu no augšējiem līmeņiem). Viena no vislielākajām šī modeļa nepilnībām ir tā, ka testēšana tajā tiek uzskatīta par darbību, kas jāveic tikai vienu reizi projekta beigās tieši pirms nodošanas ekspluatācijā. Testēšana šajā modelī tiek uzskatīta par pēdējās pārbaudes veidu analogi kā ražošanas pārbaude pirms produkta nodošanas klientam.

V-veida modeli var uzskatīt par Ūdenskrituma modeļa paplašinājumu, kur testēšanai tiek piešķirta daudz lielāka loma, šajā modelī testēšanas un pārbaudes aktivitātes pēc svarīguma un apjoma līdzinās projektēšanas un izstrādes aktivitātēm. Izstrādes darbības ir atdalītas no pārbaudes darbībām, katram izstrādes posmam ir pretstatīts testēšanas posms. Modelim ir V – veida forma, kur izstrādes aktivitātes sākot ar prasību definēšanu, līdz pat implementācijai atrodas lejup vērstajā "V" zarā, bet testēšanas aktivitātes atrodas uz augšu vērstajā zarā, tās sakārtotas testēšanas līmeņos, kas atbilst pretējā zara abstrakcijas līmenim. Vispārīgais V-modelis ir ļoti izplatīts un bieži tiek pielietots praksē, tāpēc tas tiks apskatīts sīkāk.

1.1 V-veida modelis[1] [2] [6]

V-veida modeļa pamatā ir apsvērums, ka izstrādes un testēšanas uzdevumi ir vienlīdz svarīgi un viens otram atbilstoši. Tas attēlots burta "V" veidā, kur uz leju vērstajā zarā ir sakārtoti izstrādes uzdevumi, bet uz augšu vērstajā – testēšanas uzdevumi, kas atbilst pretējā zara uzdevumu līmenim, kā tas ir parādīts zīmējumā Zīmējums 2: V-veida modelis

Katram testētājam un izstrādātājam būtu jāzina šis V-veida modelis, jo pat tad, ja projektā tiek izmantots cits izstrādes modelis, V-veida modeļa principus var pārveidot un izmantot arī citiem modeļiem.



Zīmējums 2: V-veida modelis

Izstrādes darbības kreisajā zarā ir tādas pašas kā ūdenskrituma modelim:

- Prasību definēšana – tiek apkopotas, specificētas un apstiprinātas pasūtītāja vai sistēmas nākotnes lietotāja vēlmes un prasības, tādējādi definējot sistēmas mērķi un vēlamās tās īpašības un funkcionalitāti.
- Sistēmas funkcionālā projektēšana – prasības tiek sakārtotas jaunās sistēmas funkcijās un dialogos.
- Sistēmas tehniskā projektēšana – tiek projektēta sistēmas implementācija, kas sevī ietver sistēmas saskarnes izveidi un sistēmas sadali mazākās apakšsistēmās (sistēmas arhitektūra), turpmāk katra apakšsistēma var tikt izstrādāta neatkarīgi no citām.
- Komponenšu (vienību) projektēšana – katrai apakšsistēmai tiek definēti tās uzdevumi, uzvedība, iekšējā struktūra un tās attiecības ar citām apakšsistēmām.
- Programmēšana – katra komponente (vienība, modulis, klase) tiek

implementēta programmēšanas valodā.

Virzoties uz priekšu pa šiem izstrādes līmeņiem, izstrādājamā sistēma tiek aprakstīta arvien detalizētāk. Ja šāda veida izstrādes laikā tiek pielaistas kļūdas, tad parasti vieglākais veids, kā tās atrast, ir meklējot tās tajā līmenī, kurā tās tika atklātas. Labais V-modeļa zars apraksta katram izstrādes līmenim atbilstošu testēšanas līmeni:

- Komponenšu (vienību) testēšana – tiek pārbaudīts, vai katra programmas komponente darbojas korekti un atbilstoši tās specifikācijai.
- Integrācijas testēšana – tiek pārbaudīts, vai komponenšu grupas savstarpēji sastrādājas tā, kā ir aprakstīts tehniskajā sistēmas projektējumā.
- Sistēmas testēšana – tiek pārbaudīts, vai visa sistēma kopumā apmierina aprakstītās prasības.
- Akcepttestēšana – tiek pārbaudīts, vai sistēma no klienta viedokļa atbilst prasībām, kas minētas līgumā.

Katrā testēšanas līmenī jāpārbauda, vai izstrādes rezultāts izpilda specificētās prasības konkrētajā abstrakcijas līmenī. Šis process, kas tiek pārbaudīti izstrādes rezultāti atbilstoši sākotnējām prasībām, tiek saukts par validāciju. Validācijas laikā testētājs novērtē, vai produkts (vai tā daļa) izpilda savu uzdevumu un ir piemērots tā sākotnējam mērķim.

V-modelī papildus validācijas testiem, ir jāveic arī tā sauktā verifikācija. Atšķirībā no validācijas, verifikācija attiecas tikai uz vienu, konkrēto izstrādes procesa posmu. Verifikācijai jānodrošina, ka dotā posma rezultāts ir izveidots pareizs un pilnīgs, atsaucoties uz tā aprakstu (konkrētā izstrādes līmeņa dokumentu). Tas nozīmē to, ka tiek pārbaudīts, vai specifikācijas tiek korekti implementētas, vai produkts atbilst specifikācijai, bet ne to, vai galarezultāts ir piemērots tā sākotnējam lietošanas mērķim.

Reālajā dzīvē katra testēšana sevī ietver abus šos aspektus – gan validāciju, gan verifikāciju, bet jāpiezīmē, ka validācijas nozīme pieaug ar katru nākošo testēšanas fāzi.

V-modelis rada iespaidu, ka testēšana sākas relatīvi vēlu, pēc sistēmas implementācijas. Šis iespaids ir kļūdains. Testi labajā V zarā ir jāsaprot kā testu izpildes līmeņi, testu sagatavošana (testu plānošana un kontrole, testu analīze un izveide) sākas daudz agrāk un tiek veikta paralēli izstrādes fāzēm kreisajā V zarā.

Testu līmeņu nošķiršana V-modelī ir kas vairāk kā tikai testēšanas aktivitāšu pagaidu sadalījums, tas drīzāk ir process, kas definē testēšanas abstrakcijas līmeņus. Šie līmeņi tehniski ir ļoti atšķirīgi, tiem ir dažādi mērķi un tajos tiek pielietotas dažādas testēšanas

metodes, dažādi testēšanas rīki un tiem nepieciešami darbinieki ar dažādām zināšanām. Turpmākajos apakšpunktos tiek aprakstīti testēšanas veidi, kādi ir V-modelī.

1.1.1 Komponenšu (vienību) testēšana[1] [2] [6]

Vienībtestēšana ir pirmais testēšanas veids, kas tiek veikts programmatūrai. Atkarībā no programmēšanas valodas, vienības tiek sauktas dažādi – par moduļiem, klasēm, programmām, funkcijām, tāpat arī attiecīgie testi var tikt saukti par klašu, moduļu, funkciju testiem. Šajā darbā tie turpmāk tiks saukti par vienībtestiem.

Vienībtestēšanas laikā tiek testētas programmatūras vienības, katra no tām individuāli un nesaistīti ar citām programmatūras vienībām. Tas nepieciešams tādēļ, lai testēšanas laikā novērstu ārējo ietekmi uz vienību, ja tāda varētu pastāvēt. Ja testēšanas rezultātā tiek atklāta kāda kļūda, tad ir skaidrs, ka tā radusies tieši tajā vienībā, kas tiek testēta.

Šajā testēšanas posmā testēšana tiek veikta ciešā saskarē ar izstrādi, testēšanai tiek izmantoti testdziņi (*testa draiveri*). Testdzinis ir programma, kas izsauc testējamo vienību un saņem tās atbildi, piemēram, ja testējamā komponente ir klase, kas aprēķina tai padoto parametru summu, tad testdzinis izsauc klasi ar vairākām testējamo parametru kopām, saņem klases atbildes, salīdzina tās ar paredzamo rezultātu un atgriež testētājam salīdzināšanas rezultātus. Bieži testdziņi spēj ierakstīt rezultātus, kā arī nolasīt ieejas datus no datnes vai datubāzes.

Lai uzrakstītu pareizu testdzini, ir nepieciešamas programmēšanas iemaņas, kā arī zināšanas par konkrēto vienību – kādi ir tās ieejas dati, kādiem jābūt izejas datiem, ko pati vienība dara. Lai pareizi uzrakstītu testdzini, jābūt pieejamam pašas vienības kodam, kā arī tas jāsaprot. Tas ir iemesls, kāpēc parasti vienībtestēšana tiek atstāta pašu izstrādātāju, nevis testētāju pārziņā. Tāpēc vienībtestēšana bieži tiek saukta arī par programmētāju (izstrādātāju) testēšanu.

Vienībtestēšanas mērķis ir panākt, lai testējamais objekts atbilst prasībām, lai tas nodrošina visu savu prasīto funkcionalitāti pilnīgi un pareizi. Šajā posmā testēti tiek ieejas – izejas dati. Šajā posmā parasti tiek pamanītas tādas kļūdas kā nepareiza aprēķināšana, izlaisti vai nepareizi izvēlēti programmas ceļi.

Svarīgi vienībtestēšanā ir neaizmirst arī tā sauktos “negatīvos testus”, kad vienībai

speciāli tiek padoti nekorekti ievaddati, lai redzētu, vai vienībā ir apstrādāti kļūdu gadījumi. Piemēram, par ievaddatiem summas aprēķināšanas klasei izvēlēties burtus. To sauc arī par robustuma testēšanu.

Vienībtestēšanas laikā vajag notestēt arī nefunkcionālās prasības, piemēram, efektivitāti un uzturamību, jo vēlākos testēšanas posmos to izdarīt vai nu nevar, vai nu var, bet ar daudz lielākām izmaksām.

Efektivitātes tests pārbauda, cik efektīvi komponente izmanto datora resursus, kā, piemēram, atmiņas izmantošana, izpildes laiks. Atšķirībā no citiem nefunkcionālajiem testiem, šos mērījumus var veikt tieši testa izpildes laikā. Efektivitātes testus ļoti reti izpilda visām programmatūras vienībām, tos parasti veic tikai kritiski svarīgākajām sistēmas daļām vai arī tām, kuru efektivitātes kritēriji ir noteikti specifikācijā. Šie testi ir svarīgi iebūvētajā programmatūrā vai reālā laika programmatūrā, kad ir vai nu ierobežoti aparatūras vai laika resursi.

Uzturamības testi ir atbildīgi par to, cik viegli saprotams, pārveidojams vai papildināms ir vienības kods. Šajā testēšanas veidā svarīgs ir laiks, kāds nepieciešams, lai programmētājs saprastu programmu un tās saturu. Testējot uzturamību ir nepieciešams pievērst uzmanību šādiem aspektiem – koda struktūrai, modularitātei, koda komentāru kvalitātei, standartu ievērošanai, dokumentācijas esamībai un saprotamībai. Uzturamību nevar testēt, uzrakstot testdzini, to vajag darīt, analizējot dokumentāciju un uzrakstīto kodu.

Vienībtestēšanas laikā testētājam ir pieejams izejas kods, kas vienībtestēšanu padara par "baltās kastes testēšanu" (*"white-box testing"*). Testētājs var izstrādāt testpiemērus, zinot programmas struktūru, tās funkcijas un mainīgos. Arī testu izpildes laikā kods ir pieejams, tādējādi ir iespējams izsekot mainīgo vērtību maiņām testa gaitā. Tas palīdz atrast kļūdas vietu, kurā vienība sāk uzvesties nepareizi. Izpildot robustuma testus, ir ļoti svarīgi, lai varētu atrast vietu, kurā jāapstrādā izņēmumgadījums.

Tomēr realitātē vienībtesti bieži tiek veikti kā „melnās kastes" („*black-box*") testi. Tas nozīmē, ka testpiemēru izstrādē netiek izmantota testējamās vienības koda analīze. No otras puses reālā sistēmas programmatūrā ir vairāki simti un pat tūkstoši elementāru vienību un nav reāli izanalizēt katras vienības kodu un izstrādāt katrai no tām testpiemērus. Šādos gadījumos bieži tiek izmantota šāda pieeja – vairākas vienības tiek integrētas lielākās vienībās un testētas tiek nu jau šīs lielākās komponentes, bieži tieši šīs lielākās komponentes arī tiek uzskatītas par mazākajām vienībām. Šādu testēšanu arī sauc par vienībtestēšanu. Šajā

gadījumā vienības parasti ir pārāk apjomīgas, lai varētu saprātīgā laikā izanalizēt vienību kodu un to izmantot vienību testēšanai.

Katram specifiskam gadījumam vajag izlemt, kādu vienībtestēšanu lai izvēlas – vai testēt pašas mazākās vienības katru atsevišķi vai apvienot tās un testēt jau lielākus apgabalus.

1.1.2 Integrācijas testēšana[1] [2] [6]

Pēc vienībtestēšanas nākošais līmenis V-modelī ir integrācijas testēšana. Uzsākot integrācijas testēšanu, tiek pieņemts, ka visas komponentes atsevišķi jau ir notestētas un ja iespējams, tad visas iepriekšējā līmenī atrastās kļūdas jau novērstas. Notestētās vienības tiek apvienotas lielākās strukturālās vienībās un apakšsistēmās, šo komponenšu apvienošanu sauc par integrāciju un to veic izstrādātāji, testētāji vai speciālas integrācijas komandas.

Pēc komponenšu apvienošanas, ir jānotestē, vai šīs komponentes korekti sadarbojas savā starpā. Integrācijas testēšanas mērķis ir atrast kļūdas integrēto komponenšu savstarpējā mijiedarbībā. Pat tad, ja komponentes katra atsevišķi ir rūpīgi notestētas un atrastās kļūdas novērstas, šajā posmā vienalga var rasties un arī rodas kļūdas. Kā piemēru šeit var minēt to, ka divas dažādas komponentes katra atsevišķi darbojas bez kļūdām, bet tad, kad viena komponente nodod otrai parametrus tālākai apstrādei, parametri tiek sajaukti vietām vai tiek nodots par vienu parametru vairāk vai mazāk, nekā definēts otrā komponentē.

Integrācijas testēšana plašākā nozīmē ietver sevī arī mijiedarbības ar citām, ārējām, sistēmām testēšanu. Reizēm to mēdz dēvēt arī par "sistēmu integrācijas testu" vai "augstāka līmeņa integrācijas testu". Šajā gadījumā izstrādes rīcībā ir tikai daļa no sistēmas, pārējā daļa ir ārējās sistēmas, kam tiešā veidā izstrādātāji nevar piekļūt. Šo riska faktoru jāņem vērā, jo var būt tā, ka testēšanas laikā sistēmu mijiedarbība ir bijusi pareiza, bet tad ārējā sistēmā tiek veiktas izmaiņas un sistēmas vairs viena otru nesaprot.

Integrācijas laikā komponentes soli pa solim tiek apvienotas lielākās struktūrās, ideālā gadījumā pēc katra šāda soļa tiktu notestēta visa sistēma – tikko struktūrai tiek pievienota jauna komponente, struktūras darbība tiek pilnībā notestēta. Bet realitātē ļoti reti ir tādi gadījumi, kad sistēma tiek veidota pilnīgi no jauna. Parasti tiek mainīta vai paplašināta jau esoša sistēma vai tā tiek sasaistīta ar citām, jau eksistējošām sistēmām. Iespējami arī gadījumi, kad kāda sistēmas daļa tiek nopirkta jau gatava. Komponenšu testēšanas laikā šīs

jau gatavās daļas visdrīzāk netiek testētas vispār. Tomēr integrācijas testos šīs daļas ir jāņem vērā un jāizpēta to sadarbošanos ar citām sistēmas daļām.

Tāpat kā vienībtestēšanā, arī integrācijas testiem nepieciešami testdziņi, tie padod testa datus testējamam objektam un saņem no tā rezultātus. Tur, kur tas ir iespējams, jāizmanto vienībtestēšanas laikā sagatavotie testdziņi, ja nepieciešams, tos pārveidojot. Ja vienībtestēšanas laikā testi ir veidoti viens ar otru savietojami un saprotami, tad nevajadzētu būt grūtībām tos izmantot arī integrācijas testēšanā.

Integrācijas testēšanas mērķis ir atklāt saskarņu un sadarbības kļūdas komponenšu un integrēto daļu starpā. Piemēri, kādas kļūdas varētu tikt atrastas integrācijas testu laikā:

- Komponente citai komponentei nodod sintaktiski nepareizus datus vai vispār nenodod nekādus datus – saņēmēja komponente nevar turpināt savu darbu vai sagrūst.
- Komunikācija starp komponentēm strādā, bet katra komponente interpretē datus citādi.
- Dati tiek pārraidīti pareizi, bet par ātru vai par vēlu, var būt, ka intervāli starp pārraidēm ir pārāk mazi vai lieli.

Nevienu no šīm kļūdām nevar atrast vienībtestu laikā, jo tās rodas tikai tad, ja komponentes savstarpēji sadarbojas.

Integrācijas testu laikā var testēt arī nefunkcionālās prasības, ja tās ir svarīgas.

1.1.3 Sistēmas testēšana[1] [2] [6]

Pēc tam, kad pabeigti integrācijas testi, nākošais līmenis ir sistēmas testēšana. Sistēmas testēšanas laikā tiek pārbaudīts, vai integrētais produkts apmierina iepriekš definētās prasības. Šis testēšanas līmenis ir nepieciešams, jo iepriekšējos testēšanas līmeņos testēšana notika pēc tehniskās specifikācijas – tas nozīmē, ka testēšana notika no programmatūras izstrādātāja skatu punkta. Sistēmas testēšanā tiek izmantots sistēmas pircēja un nākotnes lietotāja skatījums. Testētāji pārbauda, vai klienta prasības ir izpildītas un vai tās ir izpildītas precīzi un pilnībā. Daudzas sistēmas īpatnības un funkcionalitāte izpaužas tikai tad, kad darbojas visas sistēmas komponentes kopumā, tās var tikt notestētas tikai tad, kad sistēma ir pilnībā gatava.

Tad, kad integrācijas testēšana ir pabeigta, sistēma ir pilnībā uzbūvēta un sistēmas testēšana apskata sistēmu kā vienu veselumu. Lai pilnībā notestētu sistēmu, jārada testu vide,

kas ir pēc iespējas līdzīgāka tai, kādā sistēma tiks lietota reālajā dzīvē. Šajā testēšanas posmā vairs netiek izmantoti testdziņi, bet tie aparatūras un programmatūras rīki, kas tiks izmantoti tad, kad klients jau lietos sistēmu.

Lai ietaupītu laiku un izmaksas, bieži tiek pieļauta liela kļūda – tā vietā, lai testētu sistēmu atsevišķā vidē, sistēmas testi tiek veikti klienta ekspluatācijas vidē. Šādi rīkojoties, ir vairāki mīnusi. Sistēmas testu laikā noteikti radīsies daudzas kļūdas, kā rezultātā klienta ekspluatācijas videi var tikt nodarīti kaitējumi. Produkcijas vidē var rasties lieli datu zudumi un sistēmas nobrukšana. Testētājiem vai nu nav, vai ir ļoti ierobežotas tiesības mainīt klienta sistēmas parametrus un konfigurāciju, testu rezultātus var būtiski iespaidot tas, ka paralēli testiem klienta darba vidē notiek paralēli procesi no citām sistēmām. Sistēmas testus, kas ir veikti, nevar atkārtot vai var tikai ar lielām grūtībām.

Viena no lielākajām sistēmas testu izpildes grūtībām rodas tad, ja nekur nav aprakstītas klienta prasības vai tās ir aprakstītas tikai daļēji. Tādā gadījumā testētājam nav nekāda dokumenta, pēc kura vadīties, testējot sistēmu, un viņam jānodarbojas ar klienta vēlmju noskaidrošanu, kas var būt ļoti ilgs un dārgs process. Ja prasības nekur nav specificētas, tad pastāv ļoti liels risks, ka ir izstrādāta klienta vēlmēm neatbilstoša sistēma.

1.1.4 Akcepttestēšana[1] [2] [6]

Visi testēšanas līmeņi, kas aprakstīti iepriekš, ir izstrādātāja atbildībā, tie tiek veikti pirms sistēma tiek pirmo reizi rādīta klientam vai nākotnes lietotājam. Bet pirms tam, kad sistēmu sāk reāli lietot, jāiziet vēl viens testēšanas līmenis – akcepttestēšana. Šajā testēšanas līmenī uzsvars tiek likts uz sistēmas nākotnes lietotāja skatījumu un tā spriedumu par sistēmu. Akcepttestēšana ir īpaši svarīga, ja tiek izstrādāta sistēma, kas specifiska tieši šim konkrētajam pasūtītājam. Akcepttestēšana ir vienīgā testēšanas fāze, kurā tiek iesaistīts klients vai sistēmas gala lietotājs, tāpēc ir svarīgi, lai šīs fāzes testi būtu tam saprotami, reizēm akcepttestu testpiemērus veido pats klients.

Akcepttestēšanu var veikt arī paralēli pārējiem testēšanas posmiem:

- Komercproduktus, kas nav jāpārveido, var testēt pēc tam, kad tie ir instalēti vai integrēti pārējā sistēmā

- Komponenšu lietderības akcepttestus var veikt vienībtestēšanas laikā
- Jaunas funkcionalitātes akcepttestēšana var notikt pirms sistēmas testēšanas (izmantojot prototipus)

Tipiski akcepttestēšanas veidi ir šādi:

- Testēšana, lai noskaidrotu, vai izpildītas līguma saistības. Ja sistēma tiek veidota tieši klienta vajadzībām, tad klients, sadarbībā ar piegādātāju, veic akcepttestus, lai noskaidrotu, vai ir izpildītas visas līguma saistības. Ja sistēma tiek veidota paša izstrādātāja uzņēmuma iekšējai lietošanai, šis līgums var būt neformāls starp sistēmas tiešo lietotāju struktūrvienību un IT nodaļu. Testēšanas un akceptēšanas kritērijiem jābūt specificētiem līgumā, šiem kritērijiem jābūt skaidri un viennozīmīgi definētiem. Arī citiem kritērijiem, kā, piemēram, likumdošanas normatīviem, drošības prasībām, ir jābūt iepriekš noteiktiem līgumā. Praksē parasti izstrādātājs šos akcepttestus jau ir veicis sistēmas testēšanas laikā, tādējādi akcepttestu laikā var izmantot jau iepriekš sagatavotus testpiemērus. Tomēr lai izvairītos no tā, ka izstrādātājs ir pārpratis akceptēšanas kritērijus, ir vēlams, lai pasūtītājs pirms testu veikšanas testpiemērus pārskatītu. Akcepttestus jāveic klienta testu vidē, tas rada risku, ka tāpēc, ka ir cita testēšanas vide, testpiemēri, kas izstrādātāja testu vidē bija pareizi, tagad var izsaukt kļūdas. Akcepttestos jāpārbauda arī sistēmas piegādes un instalācijas procedūras.
- Lietotāju akcepttesti. Šis akcepttestu veids ir ieteicams, kad pasūtītājs un sistēmas lietotājs ir dažādi. Bieži dažādām lietotāju grupām ir dažādi priekšstati par to, kādai jābūt jaunajai sistēmai. Pat tad, ja sistēmā no tehniskā vai funkcionālā viedokļa ir izpildītas visas prasības, pastāv risks, ka lietotāji vai kāda lietotāju grupa neakceptē sistēmu, jo tās darbība neapmierina viņu nosacījumus. Tāpēc ir svarīgi veikt akcepttestēšanu visām lietotāju grupām, sadalot testpiemērus pa lietotāju ikdienas biznesa procesiem. Ja šādas akcepttestēšanas laikā tiek atklātas būtiskas nepilnības, bieži ir jau par vēlu veikt būtiskus uzlabojumus. Lai novērstu šāda gadījuma iestāšanos, ieteicams nākotnes lietotājiem parādīt sistēmas prototipus jau agrā sistēmas izstrādes stadijā.
- Operacionālie testi. Operacionālos akcepttestus veic sistēmas administratori. Šajos testos jāietver sistēmas dublējumkopiju veidošanu un atjaunošanu no

dublējumkopijām, nobrukšanas gadījumu atkopšanu, lietotāju pārvaldību, uzturēšanas uzdevumu izpildi, kā arī sistēmas drošuma pārbaudi.

- Lauku testi (Alfa un Beta testi). Ja ir paredzēts programmatūru izmantot uz daudzām dažādām izpildes vidēm, ir ļoti sarežģīti un dārgi veikt sistēmas testus uz visām šīm vidēm. Šādos gadījumos sistēmas izstrādātājs var veikt lauku testus (*field tests*). Šo testu mērķis ir atklāt lietotāju vides ietekmi uz sistēmu un, ja nepieciešams, to novērst. Šim mērķim izstrādātājs piegādā stabilas pirms relīzes programmatūras versijas iepriekš izvēlētiem klientiem, kuru darba vides adekvāti nosedz iespējamās vides, kurās sistēmai būtu jāstrādā. Tad šie klienti vai nu izpilda testpiemērus, ko noteicis izstrādātājs, vai nu darbojas ar sistēmu izmēģinājuma režīmā reālos darba nosacījumos. Problēmas, kas atrastas šajā testēšanas fāzē, kā arī klientu vispārīgs vērtējums un ieteikumi, tiek nodoti izstrādātājam, kurš tos atbilstoši apstrādā. Šādu testēšanas veidu sauc arī par Alfa vai Beta testiem, Alfa testi tiek veikti izstrādātāja, bet Beta testi – klienta pusē. Jāatceras, ka ar lauku testiem nedrīkst aizstāt izstrādātāja iekšējos testus, klientam jāsaņem jau stabila testēšanas vide.

Akcepttestu apjoms var būt ļoti dažāds, tas atkarīgs no aplikācijas riska pakāpes – ja programmatūra ir izstrādāta tieši šim klientam, tad riska pakāpe ir augsta un nepieciešama pilna akcepttestēšana. Ja sistēma ir standarta produkts un jau ilgi tiek lietota citos, līdzīgos uzņēmumos, tad akcepttests var sastāvēt tikai no sistēmas uzinstalēšanas un dažu piemēru apskatīšanas.

1.2 Uzņēmuma programmatūras izstrādes modelis

Uzņēmumā, kurā veikta testēšanas procesu izpēte un analīze, programmatūras izstrādes modelis kopumā atbilst V-modelim, kas aprakstīts iepriekšējā apakšnodaļā. Tomēr kā vairākumā reālās dzīves gadījumu, tas precīzi neatbilst teorētiskajam aprakstam.

Izstrādes fāzes ir mazliet izplūdušākas, vietām tās pārklājas, piemēram, programmēšana sākas jau tad, kad vēl nav pabeigtas iepriekšējās fāzes, kā arī no klienta tiek pieņemti izmaiņu pieprasījumi izstrādes laikā tad, kad prasību specificēšanas fāze jau noslēgusies. Arī testētāji iepazīstas ar specifikācijām jau tad, kad tās vēl nav līdz galam

pabeigtas.

Vairākas V-modelī aprakstītās fāzes uzņēmumā ir saplūdušas kopā, piemēram, netiek atsevišķi izdalītas sistēmas funkcionālā un tehniskā projektēšana, tās tiek veiktas paralēli viena otrai un šī apvienotā izstrādes fāze tiek saukta par sistēmas projektēšanu. Šajā apvienotajā fāzē lielākoties tiek aprakstīta arī lielākā daļa komponenšu. Tās komponentes, kas paliek šeit neizdalītas, tiek atstātas programmētāja brīvai interpretācijai.

Uzņēmuma pirmsākumos netika veikta arī vienībtestēšanas fāze, tā tika ieviesta vēlāk. Agrāk testēšana tika sākta tikai ar integrācijas testiem. Tomēr tika pieņemts lēmums veikt vienībtestēšanu un nu jau vairāk kā divus gadus katram izstrādājamam projektam tiek veikti vienībtesti.

Uzņēmumā ir pieņemts integrācijas testēšanu uzskatīt par jauna produkta testēšanu, savukārt sistēmas testus sauc par stabilizāciju (stabilizācija iestājas tad, kad visi darāmie darbi ir pabeigti, notestēti un atzīti par pareiziem un atlikušas tikai kļūdas). Jāpiezīmē, ka parasti projekti sevī ietver jau esošas sistēmas uzlabošanu vai papildināšanu, tas nozīmē, ka stabilizācijas laikā tiek testēta ne tikai jaunā produkcija, bet arī visa sistēmas jau esošā (vecā) funkcionalitāte. Vadoties pēc V-modeļa, šai darbībai būtu bijis jānotiek jau integrācijas testu laikā, kad tiek testēts, kā jaunās sadaļas sadarbojas ar jau esošajām. Tomēr uzņēmumā integrācijas testu laikā uzmanība tiek pievērsta tikai jaunajam projektam, veco funkcionalitāti īpaši netestējot.

Akcepttestēšana nenotiek tikai pirms pašas nodošanas, tāda notiek arī izstrādes laikā, šādu testēšanu uzņēmumā sauc par starpnodevumiem. Pārsvarā gadījumu starpnodevumu rezultātā rodas daudzi izmaiņu pieprasījumi, uzņēmuma politika ir pēc iespējas daudz šādus pieprasījumus arī apstrādāt un ieviest sistēmā pirms sistēmas gala nodošanas klientam. Šis aspekts ir pretrunā ar V-modeli, tāpēc var uzskatīt, ka uzņēmuma programmatūras izstrādes modelī ir arī dažas spējās (*agile*)[2] programmēšanas iezīmes.

Tālākajos punktos tiek aprakstītas uzņēmuma izstrādes un testēšanas fāzes, to līdzības un atšķirības ar V-modeli.

1.2.1 Prasību definēšana

Prasību definēšanas fāzes laikā tiek veiktas intervijas ar klientu un rakstīta prasību specifikācija. Šajā fāzē tiek iesaistīti arī testētāji, kas lasa specifikācijas versijas, tās analizē un

vadoties pēc savas iepriekšējās pieredzes, meklē tās vietas specifikācijā, kas varētu neatbilst sistēmas pārējai darbībai.

Šī testēšana jau prasību specificēšanas posmā ir nepieciešama tāpēc, ka uzņēmumā ir vairāki analītiķi, kas ļoti labi pārzina tikai dažas sistēmas daļas, tāpēc varētu rasties situācija, ka aprakstot jaunas funkcionalitātes prasības, nav ņemti vērā kādi sistēmas ierobežojumi.

Kā vienkāršs piemērs šeit var būt tāds, ka aprakstot jaunas sistēmas daļas darbību, ir prasība, ka klienta tālrunis nav obligāti norādāms, bet visā pārējā sistēmas klientu reģistrā tālrunis ir obligāts. Šādas nepilnības var atrast un izlabot jau prasību definēšanas fāzē, kas ir ļoti svarīgi, jo vēlāk šīs izmaiņas saskaņošana ar klientu, izlabošana sistēmā un notestēšana prasītu daudz vairāk laika un resursu.

Tiek uzskatīts, ka prasību definēšanas fāze ir noslēgusies tad, kad ir izstrādāta tā specifikācijas versija, kas ir saskaņota ar klientu un abas puses – gan izstrādātājs, gan klients ir parakstījuši specifikāciju. Tomēr reāli šī fāze nebeidzas gandrīz visa projekta garumā, jo klients ik pa brīdim iesniedz izmaiņu pieprasījumus, visbiežāk tas notiek pēc prototipa demonstrēšanas. Šie izmaiņu pieprasījumi tiek apstrādāti un tad, ja nolemts, ka izstrādes laikā ir iespējams tos veikt, tie tiek iestrādāti specifikācijā un īsu brīdi pirms projekta beigām tiek parakstīta vēl viena specifikācija, kas sevī iekļauj arī visas tās izmaiņas, ko klients pieprasījis un ir nolemts, ka tās tiks izstrādātas šajā sistēmas versijā.

1.2.2 Sistēmas funkcionālā un tehniskā projektēšana.

Kā jau iepriekš tika pieminēts, sistēmas funkcionālā un tehniskā projektēšana kā atsevišķas fāzes uzņēmumā netiek izdalītas. Tās tiek apvienotas ar prasību definēšanas fāzi. Šo posmu izstrādā sistēmas analītiķis kopā ar sistēmas arhitektu. Šai fāzei netiek arī rakstīts atsevišķs dokuments.

Sistēmas funkcionālā un tehniskā projektēšana daļēji tiek aprakstīta sistēmas specifikācijā vai tiek veidots specifikācijas pielikums (vai vairāki). Daļu no šīs fāzes apraksta projektu vai izstrādes vadītājs, kad veido darāmo darbu sarakstu programmētājiem. Darāmo darbu aprakstā parasti tiek iekļauti arī darbu tehniskie apraksti. Daļu no sistēmas tehniskā apraksta atstāj programmētāja pārziņā – programmētājam tiek uzdots izveidot darāmajā darbā aprakstīto funkcionalitāti, kā to realizēt, izlemj pats programmētājs. Darba veikšanas laikā programmētājs darāmo darbu reģistrēšanas un uzturēšanas sistēmā apraksta, kas tieši un kā

tika darīts. Tomēr galvenās vadlīnijas ir aprakstītas sistēmas prasību specifikācijā. To pieprasa arī klients, lai viņa atbalsta dienests varētu iespēju robežās izlabot radušās kļūdas un pārpratumus.

1.2.3 Komponenšu (vienību) projektēšana.

Arī komponenšu (vienību) projektēšanas fāze uzņēmumā nav atsevišķi izdalīta. Daļu komponenšu apraksta projektu vai izstrādes vadītājs, kad veido darāmos darbus programmētājam, bet lielākā daļa vienību tiek atstātas programmētāja ziņā – kad tas saņem darāmo darbu, tad pats izdomā, kā to realizēs un kādas komponentes tiks veidotas vai mainītas, ja nepieciešams, konsultējoties ar izstrādes vadītāju. Komponenšu projektēšanas fāze ir saplūdusi ar programmēšanas fāzi.

1.2.4 Programmēšana.

Programmēšanas fāze sākas jau diezgan agri – laikā, kad vēl nav beigusies prasību definēšanas fāze. Kad prasības kopumā jau definētas, atlikušas vēl tikai dažas ne tik būtiskas lietas, tad tiek uzsākta programmēšana. Tas iespējams tāpēc, ka sadarbība ar klientiem ir jau ilglaicīga un parasti tiek uzlabotas vai papildinātas jau esošas sistēmas. Tā kā ir droši zināms, ka sistēmas pamatdarbības netiks mainītas, tad tās var sākt izstrādāt jau tad, kad pārējās prasības vēl nav zināmas.

1.2.5 Komponenšu (vienību) testēšana.

Komponenšu testēšana uzņēmuma darbības sākumā netika veikta, šī fāze tika ieviesta tikai vēlāk. Komponenšu testēšanu veic programmētāji. Tikai tad, kad sistēmas versija ir izgājusi visus vienībtestus, tā tiek likta testu vidē, kur ar to strādā testētāji. Vienībtestēšanas laikā netiek testēta efektivitāte, ātrdarbība un uzturamība. Uzturamības testus veic tikai tad, ja programmētājs ir iesācējs uzņēmumā, tad viņa rakstīto kodu pirms ievietošanas sistēmā pārskata kāds pieredzējis programmētājs. Vienībtestēšanu veic visos projektos, netiek notestētas pilnīgi visas komponentes, bet tikai svarīgākās. Dažas komponentes tiek apvienotas lielākos blokos un šiem blokiem tad tiek veikta vienībtestēšana.

1.2.6 Integrācijas testēšana.

Integrācijas testēšana sākas brīdī, kad ir izstrādāta daļa no funkcionalitātes un ir iespējams veikt kādu pamatdarbību, piemēram, izdot polisi ar visvienkāršākajiem nosacījumiem. Šī testēšanas fāze notiek paralēli programmēšanai un vienībtestiem. Uzņēmumā nav speciālas integrācijas komandas, integrāciju veic paši programmētāji.

Integrācijas testu laikā netiek veidoti testdziņi vai komponenšu aizstājēji, programmēšanas darbs tiek veikts tā, lai pēc iespējas ātrāk varētu sākt veikt testēšanu bez palīgrīku izmantošanas. Tiklīdz ir gatava vēl kāda sadaļa, tā tiek pievienota sistēmai. Katru nakti tiek uzbūvēta jaunāka sistēmas versija, kur ir visas iepriekšējās dienas izmaiņas, tādējādi tās komponentes, kas jau ir uzrakstītas un notestētas vienībtestu fāzē, tiek pievienotas sistēmai un testu vidē nonāk nākošajā dienā. Tā kā programmēšana notiek pēc izstrādes plāna un tas ir izveidots tā, ka pabeigtās sadaļas arvien papildina jau notestēto sistēmu, tad parasti nav nepieciešami testdziņi, lai testētu sistēmas darbu.

Ja ir plānots, ka sistēmai jāsadarbojas ar citām iekšējām vai ārējām sistēmām, tad integrācijas testu laikā, tiklīdz tas iespējams, tiek testēta arī šī sadarbība. Integrācijas testēšanas laikā projektos parasti tiek atrasts visvairāk kļūdu.

1.2.7 Sistēmas testēšana.

Sistēmas testēšana sākas brīdī, kad pabeigta integrācijas testēšana. To nosaka tas, ka visi darāmie darbi ir paveikti un notestēti. Sistēmas testēšana uzņēmuma ietvaros tiek saukta par stabilizāciju, jo visi darāmie darbi ir pabeigti un palikušas tikai kļūdas. Ja tiek uzlabota vai papildināta esoša sistēma, nevis veidota pavisam jauna, sistēmas testēšanas laikā tiek notestētas arī pārējās sistēmas daļas, kurās nebija veiktas izmaiņas.

Sistēmas testēšana reizēm tiek sākta tad, kad vēl nav līdz galam pabeigta integrācijas testēšana. Reizēm daži testētāji turpina integrācijas testus, bet citi jau veic sistēmas testēšanu. Tas notiek brīžos, ja ir atlicis nedaudz laika līdz sistēmas nodošanas termiņam un ir iekavējušies daži nepabeigti darbi. Bet šādi gadījumi notiek pietiekoši reti, lai uzskatītu, ka kopumā sistēmas testēšana atbilst teorētiskajam V-modelim.

1.2.8 Akcepttestēšana.

Uzņēmumā akcepttestēšanu veic klients ar izstrādes uzņēmuma pārstāvju piedalīšanos. Akcepttestēšana nenotiek tikai pirms produkta nodošanas klientam, bet arī programmatūras izstrādes laikā, kad klientam tiek rādīti sistēmas prototipi. Šis posms uzņēmumā tiek saukts par starpnodevumiem. Projekta izstrādes sākumā testpiemērus izstrādā un arī izpilda izstrādātāja pārstāvis, bet tad, kad klients jau iepazinies ar sistēmas darbību, testpiemērus sāk izpildīt un vēlāk arī izstrādāt pats klients.

Starpnodevumu laikā klientam parasti rodas vairāki sistēmas uzlabošanas piedāvājumi – kādas funkcionalitātes maiņa vai jauna funkcionalitāte, šos visus izmaiņu pieprasījumus uzņēmums cenšas izpildīt līdz projekta nodošanai.

2 VIENĪBTESTĒŠANAS IETEKME UZ TESTĒŠANAS KVALITĀTI

Viens veids, kā optimizēt testēšanas darbību, ir izstrādes laikā veidot vienībtestus. Tādējādi testēšana sākas jau agrīnā izstrādes stadijā un, kā zināms, jo ātrāk tiek atrasta un novērsta kļūda, jo mazākas ir tās izmaksas. Vienībtestu priekšrocība ir arī tas, ka izstrādātājs ātrāk var atrast kļūdas cēloni, jo ir skaidri zināms, kur tieši kļūda radusies. Ja kļūdu atrod testētājs, programmētājam paiet ilgāks laiks, lai atkārtotu kļūdu, kā arī sarežģītāk saprast, kurā vietā tā jālabo.

Tomēr jāatceras, ka pastāv arī pilnīgi pretējs viedoklis – tā, kā parasti projektiem ir ierobežots laiks, tad tiek uzskatīts, ka vienībtestu rakstīšana ir neracionāla un nelietderīga programmētāju laika izmantošana, programmētājam ir jādara tikai savi tiešie pienākumi, tas ir, jāprogrammē izstrādājamais produkts, testēšanu atstājot testētājiem. Jāatceras arī, ka parasti viena programmētāja darba diena darba devējam izmaksā vairāk nekā viena testētāja darba diena, tāpēc labāk, lai pie produkta izstrādes vairāk laika ir pavadījis testētājs, nevis programmētājs.

Arī uzņēmumā, kurā veikts šī darba pētījums, ir aktuālas šīs pašas problēmas, tāpēc tika pieņemts lēmums izmēģināt turpmākajā izstrādes gaitā sākt lietot vienībtestus. Virspusēji izanalizējot projektu gaitu ar un bez vienībtestiem, šķita, ka izstrādes process, patērētais laiks un izmaksas ir nedaudz uzlabojušies un tika nolemts, ka turpmāk visos projektos tiks izmantoti vienībtesti, bet sīkāka projektu gaitas analīze netika veikta.

Šīs nodaļas mērķis ir izanalizēt vienībtestu ietekmi uz projekta, īpaši tā testēšanas gaitu un kvalitāti. Lai to varētu novērtēt, tiks izvēlēti un savstarpēji salīdzināti divi uzņēmumā izstrādāti projekti ar līdzvērtīgu sarežģītības pakāpi, no kuriem vienam ir izmantoti vienībtesti, bet otram – nē.

2.1 Kļūdu klasificēšana

Lai varētu novērtēt vienībtestu ietekmi uz testēšanas kvalitāti, jāizanalizē, cik un kādas kļūdas tika atrastas projektos ar un bez vienībtestiem un jāsalīdzina iegūtie rezultāti savā starpā. Lai to korekti varētu veikt, jāzina, kas tad īsti ir kļūda un kā tās var iedalīt. Tas

tiek aprakstīts šajā apakšnodaļā.

2.1.1 Kļūda[1] [3] [4]

Kāda notikuma iznākumu var nosaukt par kļūdainu tikai tad, ja ir skaidri zināms, kādam tieši bija jābūt dotajā notikuma iznākumam. Viens no vārda "kļūda" skaidrojumiem ir kļūda sistēmā, prasības neizpildīšanās, vēlamā iznākuma vai uzvedības (kas aprakstīts specifikācijā vai prasībās) nesakrišana ar patieso iznākumu vai uzvedību (kas iegūts testa rezultātā). Šajā nozīmē ietilpst arī tie gadījumi, kad sistēma strādā pareizi, bet nepiepilda kādas citas nefunkcionālās prasības, piemēram, strādā par lēnu, par sarežģītu. Šādas kļūdas vēl mēdz tikt sauktas par problēmām, incidentiem (angl. *failure, problem, incident*).

Vārdam "kļūda" datorprogrammu izstrādē ir arī vēl viena nozīme. Tā ir tieši kļūda kodā, iemesls, kāpēc rodas augstāk aprakstītās kļūdas jeb problēmas pašā sistēmā. Jāpiezīmē, ka ne vienmēr kļūdai sistēmā atbilst tieši viena kļūda kodā. Var būt, ka viena kļūda kodā atbilst vairākām sistēmas kļūdām, vienai sistēmas kļūdai var atbilst vairākas koda kļūdas, kā arī var gadīties slēptās kļūdas – izlabojot vienu sistēmas kļūdu, atklājas, ka ir vēl kāda kļūda, ko iepriekšējās dēļ nevarēja pamanīt.

Šajā darbā vārds "Kļūda" tiks lietots pirmajā nozīmē, jo testētāji parasti neredz kodu, bet testē tikai sistēmas darbību.

2.1.2 Vispārīgā klasificēšana[1]

Lai nodrošinātu ātru un efektīvu kļūdu un nepilnību novēršanu dažādos testēšanas posmos, jānodrošina kļūdu ziņojumu reģistrēšana. Projektā jābūt centrālajai datubāzei, kurā tiek reģistrēti un administrēti visi problēmu un kļūdu ziņojumi, kas atklāti testēšanas laikā.

Kļūdu klasificēšanā svarīgs kritērijs ir kļūdas kritiskums (*severity*), kas apzīmē kļūdas ietekmi uz sistēmu. Ir liela atšķirība, vai neizlabotās kļūdas ir sistēmas sagrūšanas gadījumi vai tikai kosmētiskas nepilnības dažos ekrānu izkārtojumos. Kritiskuma klasifikācija, kas dota [1] , ir šāda:

1. Fatāla (*fatal*) kļūda – sistēmas sagrūšana, iespējams ar datu zudumiem, šajā gadījumā testējamo objektu nav iespējams izmantot.

2. Ļoti nopietna (*very serious*) kļūda – būtiskas nepilnības, prasības nav izpildītas vai ir izpildītas nepareizi, kļūdai ir liela ietekme uz daudziem funkcionalitātes apgabaliem, testējamo objektu var izmantot tikai ar striktiem ierobežojumiem (sarežģīts vai dārgs kļūdas apiešanas ceļš).
3. Nopietna(*serious*) kļūda – novirze no funkcionālajām prasībām vai prasību pilnīga neizpildīšana, kļūdai ir būtiska ietekme uz dažiem funkcionālajiem apgabaliem, testējamo objektu var izmantot ar dažiem ierobežojumiem, šis ir visbiežāk sastopamais kļūdu kritiskuma līmenis.
4. Vidēja kritiskuma (*moderate*) kļūda – mazas novirzes, kļūdai ir vidējas nozīmes ietekme uz nedaudziem funkcionālajiem apgabaliem, sistēmu var lietot bez ierobežojumiem.
5. Maza kritiskuma (*mild*) kļūda – neliela ietekme uz nedaudziem funkcionālajiem apgabaliem, sistēmu var izmantot bez ierobežojumiem, piemēram, pareizrakstības kļūdas vai neprecīzs ekrāna lauku izvietojums.

Problēmas kritiskumu vajag piešķirt, skatoties uz tās ietekmi uz visiem sistēmas apgabaliem. Augstākminētā klasifikācija nenorāda, cik ātri konkrēto kļūdu nepieciešams izlabot. Problēmas risināšanas prioritātei ir pavisam cita klasifikācija un tās nav jājauc vai jāaizstāj vienai otru.

Nosakot kļūdas prioritāti jeb laiku, cik ātri kļūda jālabo, jāņem vērā arī citas, papildus prasības, ko nosaka produkta vai projekta pārvaldītājs (piemēram, labojuma sarežģītība, lietošanas riskantums, apgabala, ko ietekmē kļūda, lietošanas biežums u.c.). Tāpēc to, cik ātri jālabo kļūda, nosaka cits parametrs „kļūdas prioritāte". Iespējamās parametra vērtības ir sekojošas [1]:

1. Nekavējoties (*immediate*) – lietotāja bizness vai darba process ir apturēts vai nav iespējams turpināt ieplānoto testēšanu, problēmai nepieciešams tūlītējs risinājums, ja nepieciešams, tad tūlītēja labošana klienta darba vidē (ielāps, *patch*).
2. Nākošajā laidienā (*next release*) – labojums tiks ieviests nākošajā plānotajā produkta laidienā vai nākošajā (iekšējā) testa versijā.
3. Pie izdevības (*on occasion*) – labojums tiks ieviests, kad kļūdas skartās sistēmas daļas ir plānots pārskatīt vai uzlabot.
4. Atvērts (*open*) – labojumu vēl nav plānots veikt.

2.1.3 Darbā izvēlētā klasificēšana

Uzņēmumā kļūdām ir divu pakāpju klasifikācija:

1. Pēc kritiskuma (*severity*) – cik lielu iespaidu kļūda atstāj uz sistēmu, iespējamās vērtības praktiski neatšķiras no iepriekšējā nodaļā aprakstītajām, atšķiras tikai nosaukumi:
    1. No impact – viszemākā pakāpe, šo parasti liek vizuālām, gramatikas kļūdām.
    2. Low impact – mazs iespaids, šis statuss parasti tiek likts kļūdām, kas mazliet traucē darbu, bet ir apejamas, piemēram, nesaprotami kļūdu paziņojumi.
    3. Medium impact – vidējs iespaids uz sistēmu, šis statuss tiek likts kļūdām, kas būtiski iespaido sistēmas darbību, bet lietotājs var turpināt darbu, izlabojot kļūdainos datus vai dzēšot tos, kā arī kļūdām, kuras lietotājs var apiet, iegūstot sev vēlamo rezultātu kādā citā ceļā.
    4. High impact - augsts iespaids uz sistēmu, šis statuss tiek likts kļūdām, kuru rašanās rezultātā sistēma beidz darbu vai sabojā datus, kļūdām, kas būtiski traucē lietotājam darboties ar sistēmu, kas nav apejamas, kā arī kļūdām, kas ietekmē naudas aprēķinus.
    5. Critical – kritiskas kļūdas, to parādīšanās rezultātā sistēma beidz darbu, kļūdas nevar apiet, kā arī tās skar svarīgus funkcionālus apgabalus, šo kļūdu dēļ nav iespējams strādāt ar sistēmu vai kādiem tās apgabaliem.
2. Pēc prioritātes (*Priority*) – cik ātri konkrētā kļūda jālabo. Šis parametrs būtiski atšķiras no iepriekšējā punktā aprakstītā. Iepriekšējā punkta sadalījumu uzņēmumā nodrošina ar kļūdu sakārtošanu pa projektu mapēm. Kļūdu reģistrēšanas sistēmā visas kļūdas ir sadalītas pa mapēm – katram laidienam un katram produktam, kas ar laidienu ir paredzēts, ir sava mape, kurā kļūda tiek reģistrēta. Ja kļūda jālabo šajā laidienā (pēc 2.1. punktā minētā sadalījuma vērtība „nekavējoties"), tad tā tiek reģistrēta pašreiz izstrādājamās versijas mapē, ja to nav plānots veikt šajā laidienā, tad tā tiek reģistrēta (vai pārcelta) uz nākošās versijas mapi, ja to jālabo nekavējoties ar ielāpa (*patch*) palīdzību, tad tā tiek ielikta ielāpu mapē. Tātad uzņēmumā kļūdu prioritātes sadalījums ir paredzēts tikai viena projekta (vai viena laidiena) ietvariem. Prioritāte nozīmē tikai

to, kādā secībā un cik steidzami programmētājam jālabo viņam piešķirtās kļūdas. Prioritāšu iespējamās vērtības:

1. Lowest – viszemākās prioritātes kļūda;
2. Low – zemas prioritātes kļūda;
3. Medium – vidējas prioritātes kļūda;
4. High – augstas prioritātes kļūda;
5. Highest – visaugstākās prioritātes kļūda;

Uzņēmumā katru nakti notiek sistēmas versijas atjaunošana, lai testētājiem katru rītu būtu jaunākā testu vides versija. Programmētājiem jācenšas vienas dienas laikā izlabot visas visaugstākās un augstas prioritātes kļūdas, vidējas un zemas prioritātes kļūdas var nebūt izlabotas nākošajā dienā.

Kļūdu kritiskums (*severity*) projekta gaitā nemainās, bet prioritāte var mainīties atkarībā no laika, kas pavadīts izstrādē. Projekta gaitā var gadīties, ka visaugstākā prioritāte tiek piešķirta kļūdai, kuras kritiskums ir "*No impact*". Tā var gadīties tad, ja, piemēram, notiek kāda produkta versijas prezentēšana klientam, bet sistēmā ir vizuāla kļūda, ko klientam nevajadzētu redzēt. Kā arī tuvojoties projekta beigu posmam, ir tendence, ka kļūdām pieaug prioritāte. Kļūdas, kas projekta sākuma posmā varēja būt mazsvarīgas, izstrādes beigās var kļūt ļoti svarīgas. Tā ir ar vizuālām kļūdām, kas neietekmē funkcionalitāti, bet nodot projektu ar šādiem defektiem arī nevar.

Darbā tiks analizēta kļūdu rašanās projekta gaitā pēc abiem šiem faktoriem – gan kļūdu kritiskuma, gan prioritātes.

Lai adekvāti novērtētu vienībtestu ietekmi uz projekta gaitu, jāņem vērā ne tikai kļūdu rašanās biežums, bet arī paveikto darbu precizitāte un atbilstība prasībām. Uzņēmumā ir ieviests darāmo darbu saraksts (*features*), kas tiek uzglabāts tāpat, kā kļūdu saraksts. Darāmo darbu sarakstā tiek uzglabātas gan sākotnējās prasības, kas sadalītas darbu uzdevumos, gan arī projekta vēlākajos posmos izmaiņu pieprasījumi, ja tādi radušies klientam vai sistēmanalītiķim. Arī darāmajiem darbiem ir spēkā mapju sistēma. Darbiem arī tiek piešķirtas tādas pašas prioritātes kā kļūdām un tie pēc to pabeigšanas tiek testēti. Darāmajiem darbiem uzņēmumā nav kritiskuma klasificēšanas, tāpēc tāda nebūs ieviesta un apskatīta arī šajā darbā.

2.2 Salīdzināmie projekti

2.2.1 Projektu izvēle

Uzņēmums izstrādā apdrošināšanas vadības sistēmas vairākām apdrošināšanas kompānijām gan Latvijā, gan ārpus tās. Visilgāk uzņēmums sadarbojas ar divām apdrošināšanas kompānijām Latvijā, no kurām vienai tiek veikti vienībtesti, bet otrai netiek. Uzņēmumam ir izstrādāta universāla platforma IMS (Insurance Management System), kas tiek pielāgota katra klienta prasībām un vajadzībām. Lai varētu izanalizēt un salīdzināt vienībtestu ietekmi uz projekta gaitu, jāizvēlas divi projekti ar līdzīgu sarežģītību, resursiem un projekta laiku.

Izanalizējot visus uzņēmumā esošos projektus, tika izvēlēti apdrošināšanas produkta KASKO ieviešanas projekti abās organizācijās. Uzņēmumā, kurā netiek veikti vienībtesti, šis produkts tika ieviests ātrāk, tas nozīmē, ka izstrādei un prasību analīzei bija nepieciešams lielāks laiks nekā otram projektam, jo otrā var izmantot pirmā projekta pieredzi, kā arī uzņēmums jau ir guvis ieskatu konkrētā produkta apdrošināšanas specifikā. Toties otrā projektā papildus KASKO funkcionalitātei tika izstrādāta arī universāla, vienkāršāka polises izdošanas forma (Unipolise) pārējiem apdrošināšanas produktiem, kam līdzīga jau bija ieviesta pirmajā uzņēmumā. Rezultātā abi projekti sarežģītības ziņā ir līdzvērtīgi. Liela nozīme projektu izvēlē bija arī tam, ka abos šajos projektos darba autore ir piedalījusies kā testētāja, tāpēc nav tā, ka autorei kāds no projektiem nav pazīstams.

Skaidrības labad projekti turpmāk darbā tiks saukti – KASKO projekts (tas, kurā tika izstrādāts tikai KASKO produkts, šajā projektā netika izmantoti vienībtesti) un IMS projekts (tas, kurā tika izstrādāta KASKO un Unipolise, šajā projektā tika izmantoti vienībtesti).

Tiesa, novērtējot un salīdzinot šo divu projektu izstrādi, jāņem vērā, ka KASKO projektā pārsvarā tika iesaistīti programmētāji, kas jau vismaz 1 vai 2 gadus strādā uzņēmumā, tātad labi pārzina līdzšinējo izstrādāto sistēmu, kā arī ir pieredzējušāki uzņēmumā lietoto izstrādes rīku lietošanā, IMS projektā bija iesaistīti vairāki ne tik pieredzējuši programmētāji vai tādi, kas nesen pieņemti darbā, tātad vēl nav iepazinušies ar uzņēmuma specifiku.

2.2.2 Projektu dati

KASKO projektā bija iesaistīti 5 programmētāji un 3 testētāji, IMS projektā - 7 programmētāji un 3 testētāji. KASKO izstrādes laiks bija paredzēts 5 mēneši, IMS - 8 mēneši. Lai gan izskatās, ka IMS projekts bija apjomīgāks gan piesaistīto cilvēku, gan laika ziņā, tas tā nav.

KASKO projektā iesaistītie programmētāji visi bija jau ar lielu pieredzi uzņēmumā, bet IMS projektā no septiņiem programmētājiem četri bija tikko uzsākuši darbu uzņēmumā, tātad viņi bija ne tik pieredzējuši un viņiem izstrādei bija nepieciešams vairāk laika kā citiem programmētājiem, jo paralēli vēl bija jāapgūst gan pati sistēma, gan uzņēmuma programmēšanas stils. Lai gan izskatās, ka IMS projekts bija daudz ilgāks, tomēr arī tas tā gluži nav. IMS projektā pirmajos mēnešos nenotika aktīva izstrāde, sākumā projektā piedalījās tikai viens programmētājs un viens testētājs uz pusslodzi, kā arī izstrāde notika paralēli prasību specificēšanai. Citi programmētāji un testētāji projektā tika iesaistīti tikai pēc 4 mēnešiem. Projektu izstrādes apjoms un biznesa prasības ir līdzvērtīgas.

2.3 Projektu kļūdu un darāmo darbu salīdzināšana

Lai salīdzinātu abus projektus, tika izmantota uzņēmuma kļūdu un darāmo darbu reģistrēšanas sistēma, kurā glabājas gan kļūdu un darāmo darbu sākotnējie apraksti, gan komentāri labošanas gaitā, prioritāte un kritiskums, sākuma un beigu datumi, atbildīgās personas. Lai gan šī informācija ir ļoti bagātīga, tomēr tās padziļināta analīze uzņēmumā līdz šim nekad nav tikusi veikta. Vienīgais, kam projektu vadība parasti pievērsa uzmanību, ir atvērto kļūdu un darāmo darbu skaitam projekta laikā, kā arī projekta kopējam kļūdu skaitam.

Viens no iemesliem, kāpēc šāda analīze uzņēmumā netika veikta, ir tāds, ka šī sistēma ir laba datu uzglabāšanā, bet tajā nav iebūvētas praktiski nekādas atskaites. Tāpēc, lai savāktu pētīšanai nepieciešamo statistiku, ir jāiegulda liels darbs, kur daļu no pētāmajiem parametriem nekā nevar automatizēt un vienīgais veids ir manuāli iet cauri visam kļūdu un darāmo darbu sarakstam.

Kā piemērs ir kļūdu un darāmo darbu atgriešanas skaits – tas sistēmā nekur nesaglabājas, vienīgais veids, kā to noskaidrot, ir iet cauri komentāriem, kas pie kļūdas vai

darāmā darba pierakstīti, un skaitīt reizes, cik ir "testētājs-programmētājs" komentāru. Tā kā reizēm vienā atgriešanas reizē komentāru var būt gan vairāk par nepieciešamo skaitu (piemēram, programmētājs ir veicis divus komentārus, kur otrais ir pirmā papildinājums), gan arī mazāk (piemēram, programmētājs ir atgriezis kļūdu testētājam bez komentāriem), tad automatizēt šo procesu ir ja ne neiespējami, tad ļoti sarežģīti. Tāpēc tika izvēlēts vieglākais ceļš – šo analīzi veikt, manuāli izskatot visus datus.

Darbā tika izvēlēti vairāki salīdzināšanas veidi, pēc kuriem var spriest par vienībtestu ietekmi uz testēšanu. Par galvenajiem kritērijiem tika izvēlēti piereģistrēto kļūdu skaits kopā visā izstrādes ciklā, kļūdu sadalījums pēc to prioritātes (*priority*) un kritiskuma (*severity*), kā arī kļūdu sadalījums pa izstrādes mēnešiem pēc prioritātes un kritiskuma. Darbā vēl tiek salīdzināts arī programmētājam atgriezto kļūdu un darāmo darbu (*features*) skaits. Atgriezta kļūda vai darāmais darbs ir tāds, ko programmētājs uzskata par pabeigtu, nodod testēšanai, bet testētājs atrod nepilnības un atgriež programmētājam uz labošanu. Uzņēmuma politika ir atgriezt programmētājam darāmo darbu tikai tad, ja darbā ir ļoti būtiskas nepilnības, pārējos gadījumos darāmais darbs jāuzskata par paveiktu, nebūtiskās lietas piereģistrējot kā kļūdas.

2.3.1 Kopējais kļūdu skaits projektos

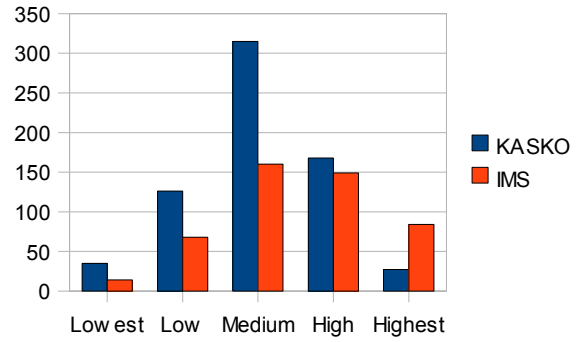
Kopumā testēšanas laikā KASKO projektā tika piereģistrēta 671 kļūda, savukārt IMS projektā bija 475 kļūdas. Tātad IMS projektā, neskatoties uz to, ka izstrāde notika ilgāku laiku, kļūdu bija par gandrīz 200 mazāk. Kļūdu sadalījums pa prioritātēm ir attēlots tabulās Tabula 1: Kļūdu skaita sadalījums pa prioritātēm un Tabula 2: Kļūdu procentuālais sadalījums pa prioritātēm, kā arī attēlots diagrammās Zīmējums 3: Kļūdu skaita sadalījums pa prioritātēm un Zīmējums 4: Kļūdu procentuālais sadalījums pa prioritātēm.

	KASKO	IMS
Lowest	35	14
Low	126	68
Medium	315	160
High	168	149
Highest	27	84
Kopā	671	475

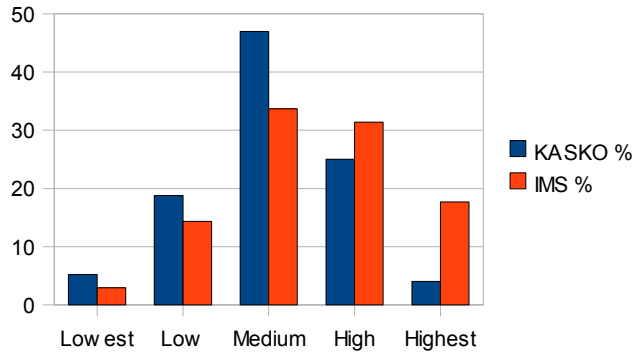
Tabula 1: Kļūdu skaita sadalījums pa prioritātēm

	KASKO %	IMS %
Lowest	5,22	2,95
Low	18,78	14,32
Medium	46,94	33,68
High	25,04	31,37
Highest	4,02	17,68

Tabula 2: Kļūdu procentuālais sadalījums pa prioritātēm



Zīmējums 3: Kļūdu skaita sadalījums pa prioritātēm



Zīmējums 4: Kļūdu procentuālais sadalījums pa prioritātēm

Kā redzams no diagrammām un tabulām, KASKO projektā bija daudz vairāk kļūdu nekā IMS projektā, tas liecina par labu vienībtestu veikšanai, jo tā kā projekti sarežģītības ziņa ir līdzvērtīgi, tad atrasto kļūdu skaitam arī būtu jābūt aptuveni vienādam. No šī mērījuma var izsecināt, ka apmēram 100 līdz 200 kļūdas tika atrastas un atrisinātas jau vienībtestu fāzē, kur programmētājs tās varēja operatīvāk atrisināt.

Tomēr ja paskatās uz kļūdu sadalījumu pēc prioritātes, redzams, ka IMS projektā bija vairāk augstas un visaugstākās prioritātes kļūdu. Tas varētu liecināt par to, ka vienībtesti nav

tikuši galā ar savu uzdevumu, jo tiem būtu jāizķer lielākās kļūdas sistēmas pamatdarbībā, kam parasti ir arī vislielākā prioritāte. Tomēr šajā gadījumā jāņem vērā IMS projekta specifika. KASKO projektā bija daudz mazāk starpnodevumu klientam, nekā tas bija IMS projektā. Un pirms starpnodevumiem parasti daudzām kļūdām tiek paaugstinātas prioritātes, lai tās tiek ātrāk izlabotas un versijā klientam neparādītos.

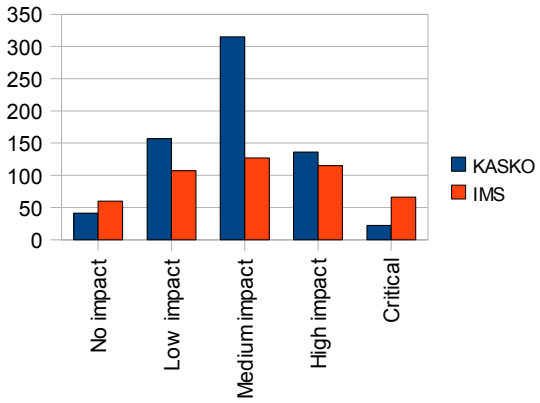
Ir svarīgi izpētīt ne tikai kļūdas pēc to prioritātes, kas projekta gaitā var mainīties, bet arī pēc to kritiskuma(*severity*) – ietekmes uz sistēmu. Šis kļūdu sadalījums visprecīzāk attēlo kļūdu ietekmi uz sistēmu kopumā, nevis konkrētajā projekta laika posmā, kad kļūda tika piereģistrēta. Sadalījums pēc kritiskuma ir redzams tabulās Tabula 3: Kļūdu skaita sadalījums pēc kritiskuma un Tabula 4: Kļūdu procentuālais sadalījums pēc kritiskuma, kā arī attēlots diagrammās Zīmējums 5: Kļūdu skaita sadalījums pēc kritiskuma un Zīmējums 6: Kļūdu procentuālais sadalījums pēc kritiskuma.

	KASKO	IMS
No impact	41	60
Low impact	157	107
Medium impact	315	127
High impact	136	115
Critical	22	66

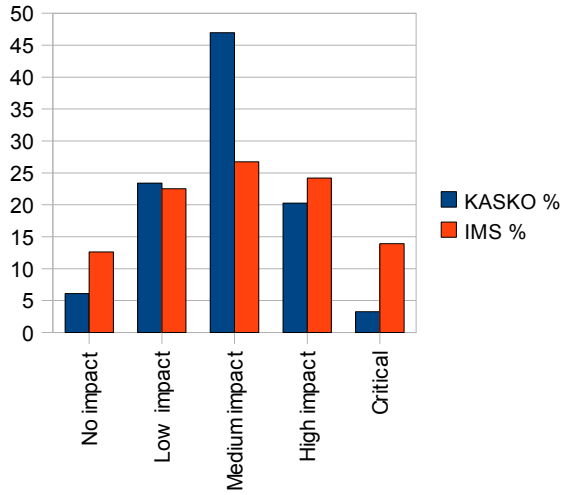
Tabula 3: Kļūdu skaita sadalījums pēc kritiskuma

	KASKO %	IMS %
No impact	6,11	12,63
Low impact	23,4	22,53
Medium impact	46,94	26,74
High impact	20,27	24,21
Critical	3,28	13,89

Tabula 4: Kļūdu procentuālais sadalījums pēc kritiskuma



Zīmējums 5: Kļūdu skaita sadalījums pēc kritiskuma



Zīmējums 6: Kļūdu procentuālais sadalījums pēc kritiskuma

Redzams, ka kļūdu skaita ziņā KASKO projektā ir daudz vairāk vidēja un zema kritiskuma kļūdu, toties IMS projektā ir vairāk augsta un visaugstākā kritiskuma kļūdu. Tātad vienībtestu ietekme nav bijusi tik liela, kā bija gaidīts, jo kritiskas kļūdas bija jāatrod jau to izpildes laikā, uz testētāju vides tām nebija jānonāk. Visticamāk, ka tāds rezultāts ir iznācis tāpēc, ka lielākā daļa programmētāju vienībtestus rakstīja pirmo reizi, kā arī programmētājiem nebija pieredzes darbā ar sistēmu. Šo rezultātu negatīvi varēja iespaidot arī tas, ka specifikācijas tika rakstītas ar pārliecību, ka visi izstrādātāji labi zina sistēmas darbību, tāpēc

daudzas lietas netika pietiekoši labi aprakstītas un programmētājs tās saprata ne tā, kā sākotnēji bija iecerēts. Daudzas no šīm augstas prioritātes kļūdām bija tieši šādi radušās – programmētājs neprecīzi saprot uzrakstīto, kā rezultātā gan sistēmā pašā, gan arī vienībtestā ir viena un tā pati kļūda. Vislabāk to vajadzētu redzēt kļūdu sadalījumā pa laika posmiem, kad izstrādes beigu daļā visām šīm neprecizitātēm vajadzētu būt apzinātām un vienībtestiem izlabotiem.

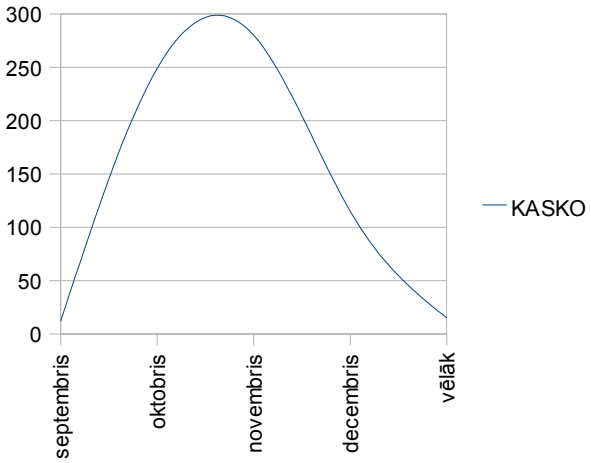
2.3.2 Kļūdu sadalījums pa projekta laika posmiem

Ir vispārzināms fakts, ka jo vēlāk tiek atrasta kļūda, jo vairāk izmaksā tās novēršana, tāpēc ļoti svarīgs faktors vienībtestu ietekmes pētīšanā, ir kļūdu sadalījums laikā.

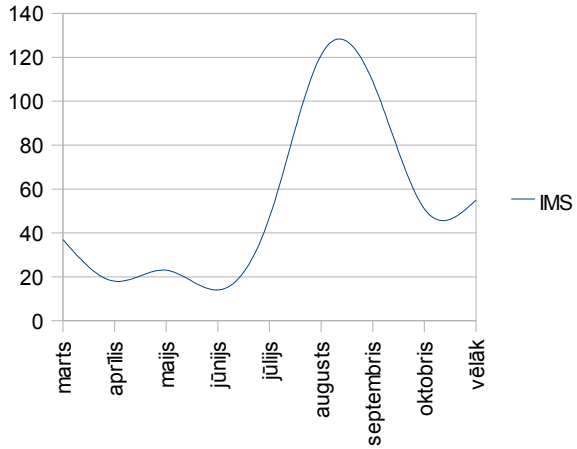
Par laika posmiem abos projektos tika izvēlēti kalendārie mēneši. Tomēr jāņem vērā, ka IMS projektā pirmajos mēnešos bija iesaistīts tikai 1 programmētājs un 1 testētājs, tāpēc sākumā atrasto kļūdu skaits ir neliels.

Jāpiezīmē, ka kļūdu sadalījumā pa mēnešiem tika ņemts vērā kļūdas reģistrēšanas (atrašanas) datums, nevis kļūdas izlabošanas datums vai laika posms starp atrašanas un izlabošanas datumiem. Tā kā jebkuru no šiem kritērijiem ņemot par salīdzināšanas parametru, rezultāti kļūdu sadalījumā pa mēnešiem ir gandrīz vienādi, tad darbā atspoguļots tikai viens no salīdzināšanas rezultātiem.

Abu projektu kļūdu kopējais sadalījums pa mēnešiem ir attēlots Zīmējums 7: KASKO kopējais kļūdu skaits pa mēnešiem un Zīmējums 8: IMS kopējais kļūdu skaits pa mēnešiem.



Zīmējums 7: KASKO kopējais kļūdu skaits pa mēnešiem

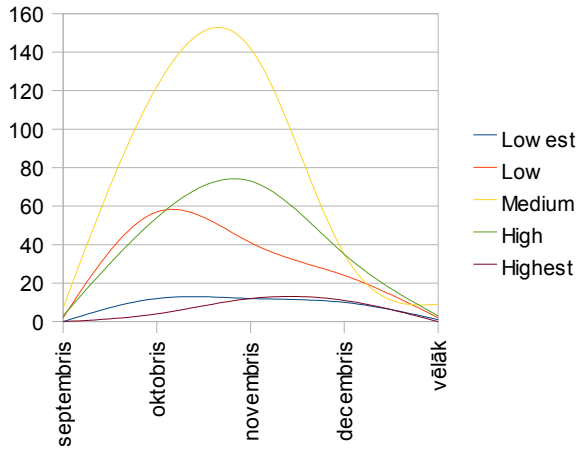


Zīmējums 8: IMS kopējais kļūdu skaits pa mēnešiem

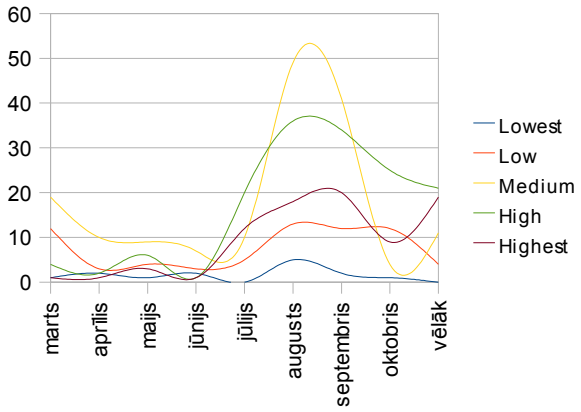
Redzams, ka kļūdu sadalījums ir apmēram vienāds, ja neskaita IMS projekta sākumu, kur gan izstrāde, gan arī testēšana notika ne tik aktīvi. IMS projektā ir vairāk kļūdu projekta beigu posmā, bet tas varētu būt tāpēc, ka uzreiz pēc šī projekta noslēguma bija neliela pauze līdz projekta ieviešanai klientam, tāpēc projekta beigu posmā, kad produkts bija gandrīz pabeigts, klientam bija daudz laika, lai pamanītu nelielas nepilnības vai nedaudz uzlabotu sistēmas izskatu vai funkcionalitāti. Tāpēc beigu posmā ir vairāk izmaiņu pieprasījumu no klienta puses nekā tas bija KASKO projektā, tāpēc arī ir vairāk iespēju rasties kļūdām.

Kļūdu prioritāšu sadalījumi abos projektos attēloti attiecīgi Zīmējums 9: KASKO

kļūdu prioritāšu sadalījums pa izstrādes mēnešiem un Zīmējums 10: IMS kļūdu prioritāšu sadalījums pa izstrādes mēnešiem.



Zīmējums 9: KASKO kļūdu prioritāšu sadalījums pa izstrādes mēnešiem

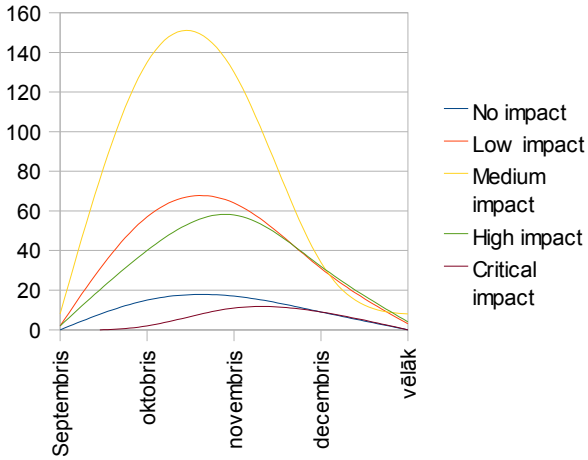


Zīmējums 10: IMS kļūdu prioritāšu sadalījums pa izstrādes mēnešiem

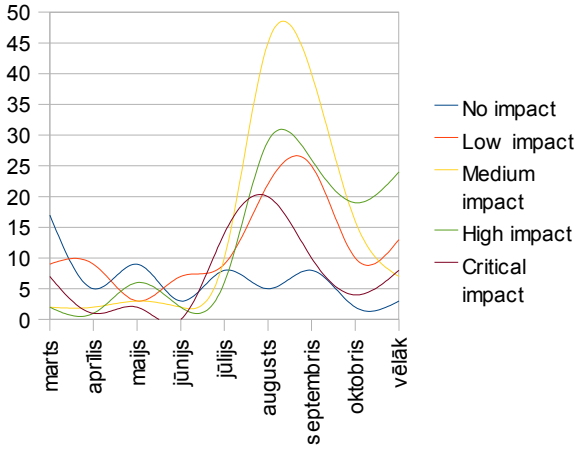
Redzams, ka abos projektos visvairāk ir vidējas prioritātes kļūdu (grafikos līknes dzeltenā krāsā). Nākošā kļūdu prioritāte ir augsta (zaļā krāsa), var pamanīt, ka IMS projektā šīs prioritātes kļūdu ir mazliet vairāk, bet tas atkal varētu būt izskaidrojams ar to, ka IMS projektā bija vairāk starpnodevumu klientam, tāpēc prioritāte pirms šiem nodevumiem daudzām kļūdām tika palielināta. Tāpēc arī IMS projektā ir salīdzinoši vairāk augstas prioritātes kļūdu. Var redzēt, ka visdrīzāk šī paša iemesla dēļ IMS projektā kļūdu prioritāšu

līkne laikā ir arī ļoti nelīdzena, "lēkājoša", salīdzinājumā ar KASKO projektu.

Kļūdu kritiskuma sadalījums pa mēnešiem ir attēlots zīmējumos Zīmējums 11: KASKO kļūdu kritiskuma sadalījums pa izstrādes mēnešiem un Zīmējums 12: IMS kļūdu kritiskuma sadalījums pa izstrādes mēnešiem.



Zīmējums 11: KASKO kļūdu kritiskuma sadalījums pa izstrādes mēnešiem



Zīmējums 12: IMS kļūdu kritiskuma sadalījums pa izstrādes mēnešiem

Redzams, ka kritiskuma sadalījums gandrīz neatšķiras no prioritāšu sadalījuma. Kā jau iepriekšējās nodaļās pieminēts, ka prioritāte IMS projektā ir augstāka tāpēc, ka šajā projektā bija vairāk starpnodevumu. Redzams, ka augsta un visaugstākā kritiskuma kļūdu ir

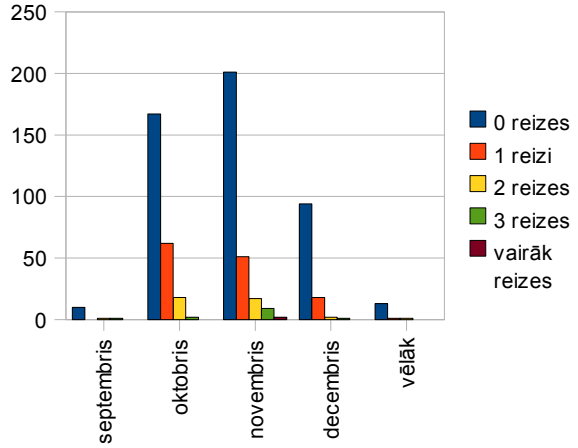
mazāk nekā attiecīgi augstas un visaugstākās prioritātes. Toties KASKO projektā kritiskuma un prioritātes līknes ir gandrīz vienādas.

2.3.3 Kļūdu atgriešana

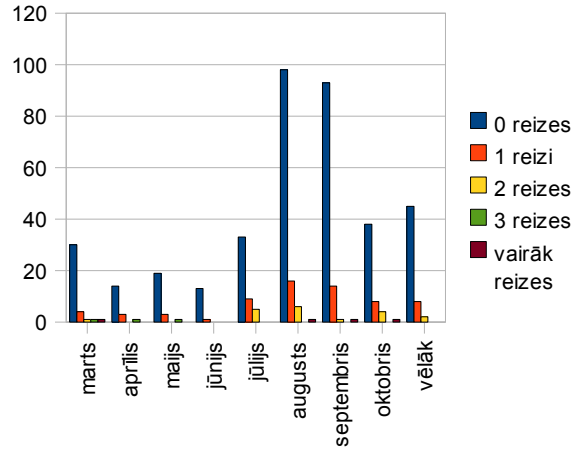
Izstrādājot darbu, autore pieņēma, ka vislabāk vienībtestu ietekmi varēs redzēt tieši no šī salīdzināšanas kritērija. Pārējos salīdzināšanas parametros ļoti lielu ietekmi izraisīja dažāds cilvēkresursu un to zināšanu līmeņu sadalījums pa projektiem, kā arī pašu projektu īpatnības. Toties šajā nodaļā tam nevajadzētu būt tik lielai ietekmei. Programmētājs savas nezināšanas dēļ var nepareizi saprast analītiķa uzrakstītās prasības, viņš var arī nezināt kādas sistēmas īpatnības, kas analītiķim jau šķiet pašsaprotamas un tāpēc tās netiek iekļautas specifikācijā. Tādējādi var rasties daudzas kļūdas, ko ar vienībtestiem atrisināt nevar un ko var atrast tikai testētājs vai programmētājs, kam jau ir pieredze uzņēmumā , kurš zina sistēmas īpatnības. Šis aspekts nemaz vai gandrīz nemaz neietekmē kļūdu atgriešanas skaitu, jo parasti kļūdas reģistrētājs ir aprakstījis gan to, kas tieši nav pareizi, gan arī to, kā īsti ir jābūt. Tāpēc paredzams, ka kļūdu atgriešanas salīdzināšanas rezultāti vistiešāk atspoguļos vienībtestu ietekmi.

KASKO projektā kopumā tika atgrieztas 186 kļūdas no 671, bet IMS 92 no 475, tātad attiecīgi 27,7% KASKO projektā un 19,4% IMS projektā. Sanāk, ka KASKO projektā tika atgriezts par 8.3% vairāk kļūdu nekā IMS projektā. Tātad kļūdu atgriešanas ziņā vienībtestu ietekme ir nedaudz uzlabojusi kopējo rezultātu. Bet projektos ir svarīgs ne tikai pats atgriešanas fakts, bet gan arī tas, cik reizes kļūda ir atgriezta programmētājam.

Kļūdu atgriešanas skaits pēc kļūdu piereģistrēšanas datuma sadalot pa mēnešiem, attēlota attiecīgi zīmējumos Zīmējums 13: KASKO kļūdu atgriešanas skaits pa mēnešiem un Zīmējums 14: IMS kļūdu atgriešanas skaits pa mēnešiem:



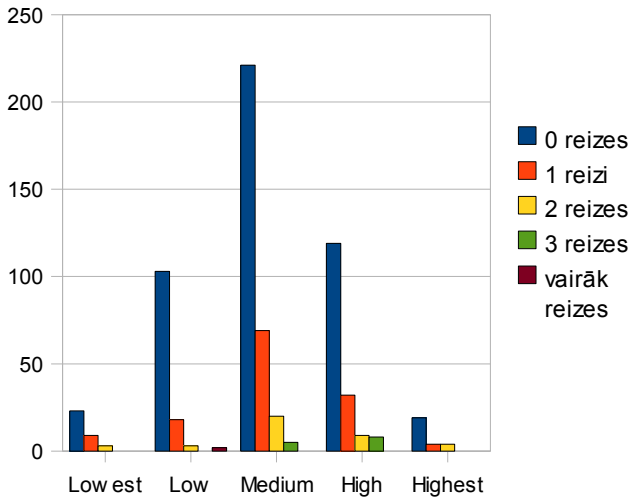
Zīmējums 13: KASKO kļūdu atgriešanas skaits pa mēnešiem



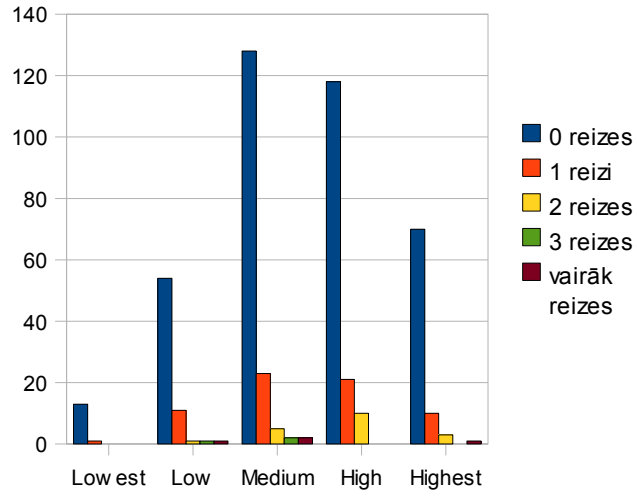
Zīmējums 14: IMS kļūdu atgriešanas skaits pa mēnešiem

Kā var redzēt, tad kopaina pa mēnešiem gandrīz neatšķiras – lielākā daļa kļūdu netiek atgrieztas, tālāk tiek atgrieztas 1 reizi, pavisam maza daļa tiek atgriezta vairāk kā vienu reizi.

Kļūdu atgriešanas sadalījums pa prioritātēm ir attēlots sekojošos zīmējumos Zīmējums 15: KASKO kļūdu atgriešanas skaits pa prioritātēm un Zīmējums 16: IMS kļūdu atgriešanas skaits pa prioritātēm:



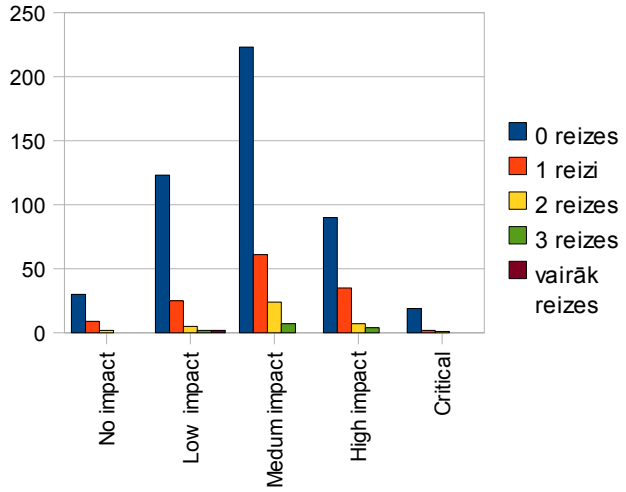
Zīmējums 15: KASKO kļūdu atgriešanas skaits pa prioritātēm



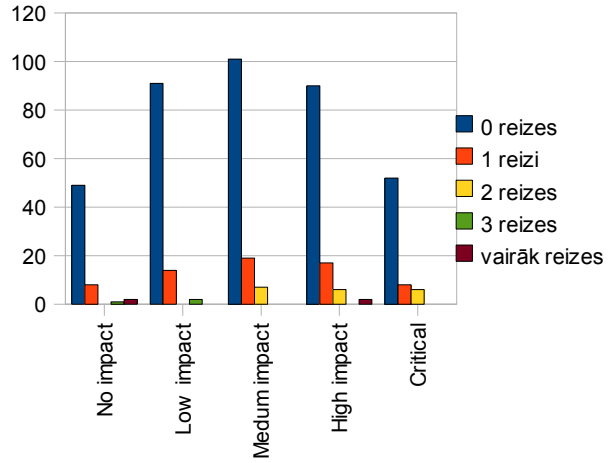
Zīmējums 16: IMS kļūdu atgriešanas skaits pa prioritātēm

Var redzēt, ka arī sadalījums pa prioritātēm ir apmēram līdzvērtīgs, redzams, ka KASKO ir mazliet vairāk atgrieztu kļūdu, bet kopumā kļūdu atgriešanas skaits ir apmēram vienāds.

Un atgriešanas sadalījums pēc ietekmes uz sistēmu ir parādīts zīmējumos Zīmējums 17: KASKO kļūdu atgriešanas skaits pēc kritiskuma un Zīmējums 18: IMS kļūdu atgriešanas skaits pēc kritiskuma:



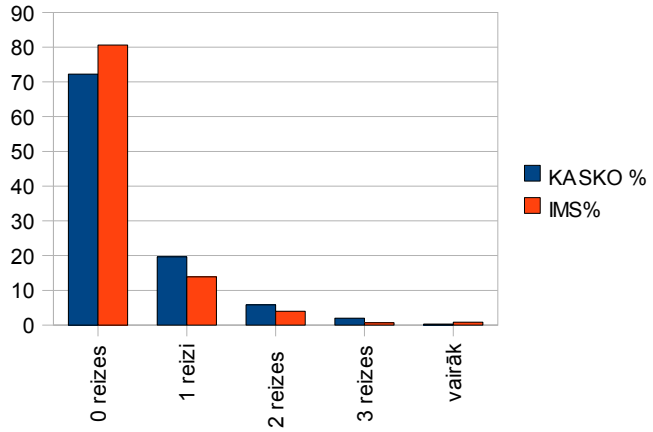
Zīmējums 17: KASKO kļūdu atgriešanas skaits pēc kritiskuma



Zīmējums 18: IMS kļūdu atgriešanas skaits pēc kritiskuma

Arī pēc ietekmes uz sistēmu kļūdas sadalījušās līdzīgi kā pēc prioritātēm, var redzēt nelielas atšķirības, bet kopumā atgriezto kļūdu skaits ir apmēram līdzvērtīgs, lai gan var redzēt, ka IMS ir mazāk atgrieztu kļūdu.

Kopumā salīdzinot KASKO ar IMS kļūdu atgriešanas procentuālo sadalījumu, kas redzams zīmējumā Zīmējums 19: Kopējais kļūdu atgriešanas skaits procentos, var redzēt, ka procentuāli IMS ir mazāk atgrieztu kļūdu. Tātad vienībtesti tomēr ir mazliet palīdzējuši uzlabot projekta gaitu un kvalitāti.



Zīmējums 19: Kopējais kļūdu atgriešanas skaits procentos

Rezumējot kļūdu atgriešanas rezultātus, var secināt, ka abu projektu kļūdu atgriešanas sadalījums gan pa mēnešiem., gan prioritātēm, gan kritiskuma ir apmēram vienāds, visvairāk atgriezts gan vidējas prioritātes, gan vidēja kritiskuma kļūdu, kā arī visvairāk atgriezto kļūdu ir projektu vidusdaļā. Visos salīdzināšanas parametros var redzēt, ka IMS atgriezto kļūdu skaits ir mazliet mazāks. KASKO ir arī lielāks skaits vairāk par vienu reizi atgrieztu kļūdu. Šo parametru visvairāk varēja ietekmēt vienībtesti, jo programmētājs, pareizi saprotot kļūdas būtību, ir iestrādājis pareizo risinājumu vienībtestos, tādējādi samazinot varbūtību vēlreiz atkārtot kļūdu, veicot kādus uzlabojumus vai pārveidojumus šajā apgabalā.

2.3.4 Darāmo darbu atgriešana

KASKO projektā tika piereģistrēti 112 darāmie darbi, IMS projektā 232. Darbu kopējais apjoms bija apmēram vienāds, IMS projektā bija sīkāks darbu sadalījums, kā arī vairāk izmaiņu pieprasījumu nekā KASKO projektā. Tas daļēji izskaidrojams ar to, ka IMS projektā bija vairāk starpnodevumu un produkta versiju rādīšanas klientam. Kā liecina prakse, klientam parasti pēc šādiem nodevumiem rodas daudz vēlmju, ko varētu izmainīt.

Tā kā lielākā daļa darāmo darbu ir izveidoti projekta sākumposmā, kā arī izstrāde

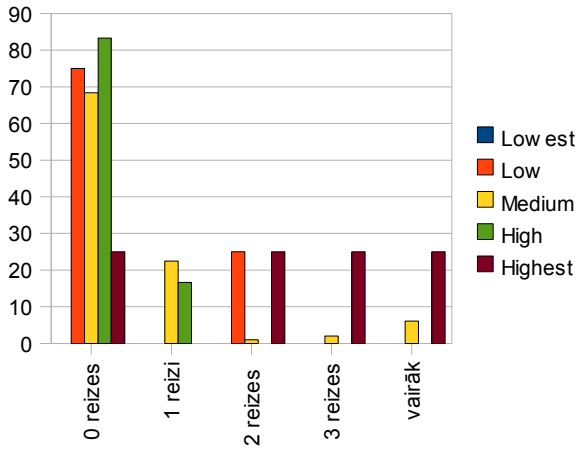
vienā darbā parasti notiek diezgan ilgi, tad darbu atgriešanas analizēšana pa mēnešiem kā tas bija ar kļūdām, netiks veikta, jo īsti nav skaidrs, kurā mēnesī iekļaut darbu, ja tas, piemēram, piereģistrēts 1. augustā, sākts darīt 20. novembrī, sākts testēt un attiecīgi atgriezts 3. decembrī pirmo reizi un 1. janvārī otro. Netiks analizēti ari darāmie darbi pēc to ietekmes uz sistēmu, jo uzņēmumā šāds sadalījums pastāv tikai kļūdām. Darbu sadalījums pa prioritātēm ir attēlots tabulā Tabula 5: Darāmo darbu sadalījums abos projektos:

	KASKO	IMS	KASKO %	IMS %
Lowest	0	4	0	1,72
Low	4	23	3,57	9,91
Medium	98	126	87,5	54,31
High	6	46	5,36	19,83
Highest	4	33	3,57	14,22

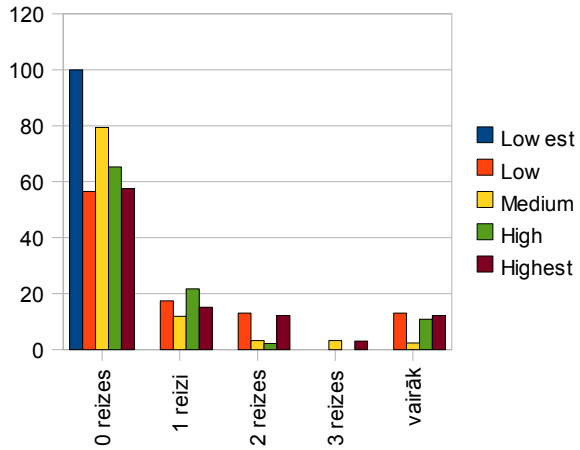
Tabula 5: Darāmo darbu sadalījums abos projektos

Tā kā ir tik lielas skaitliskas atšķirības darāmo darbu sarakstā, tad analīze un salīdzināšana veikta procentos pret kopējo kļūdu skaitu.

No Zīmējums 20: KASKO atgrieztie darāmie darbi pa prioritātēm un Zīmējums 21: IMS atgrieztie darāmie darbi pa prioritātēm var redzēt, ka KASKO projektā salīdzinoši augsts ir visaugstākās, zemas un vidējas prioritātes darāmo darbu atgriešanas procents, savukārt IMS projektā atgriezto darbu sadalījums pa prioritātēm ir gandrīz vienāds.

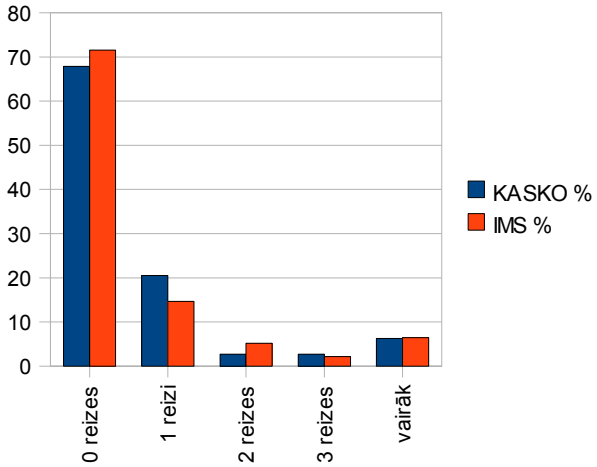


Zīmējums 20: KASKO atgrieztie darāmie darbi pa prioritātēm



Zīmējums 21: IMS atgrieztie darāmie darbi pa prioritātēm

Kā redzams grafikā Zīmējums 22: Kopā darāmo darbu atgriešana abos projektos, tad kopsummā atgriezto darbu līknes ir gandrīz vienādas, tomēr jāpiezīmē, ka ar visu to, ka IMS projektā bija iesaistīti mazāk pieredzējuši izstrādātāji, atgriezto darbu kopējais procents IMS tomēr ir zemāks nekā KASKO. IMS kopā tika atgriezts 28.45% darāmo darbu, turpretī KASKO 32.14%. Varētu uzskatīt, ka nepilni 5 procenti nav rādītājs, kas pierāda vienībtestu pozitīvu ietekmi uz projekta kvalitātes uzlabošanos, bet tomēr ņemot vērā apstākļus, tas tomēr ir vērā ņemams sasniegums un, autores-prāt, bez vienībtestu rakstīšanas atgriešanas procents būtu daudz lielāks.



Zīmējums 22: Kopā darāmo darbu atgriešana abos projektos

2.4 Pētījuma rezultāts un secinājumi

Darba izstrādes laikā iegūtais rezultāts ir pārsteidzošs un vispārpieņemtajai teorijai neatbilstošs. Tiek uzskatīts, ka vienībtestu rakstīšana ievērojami samazina testēšanas laiku un kļūdu skaitu, jo pirms produkta nodošanas testēšanā vienībtestu laikā jābūt izķertai ļoti lielai daļai kļūdu. Lai gan salīdzinot abus projektus, varēja redzēt, ka IMS projektā (projekts, kurā tika izstrādāti vienībtesti) bija mazliet labāki rezultāti, tomēr uzlabojums ir mazāks, nekā bija sagaidāms.

KASKO projektā tika piereģistrēta 671 kļūda, savukārt IMS projektā bija 475 kļūdas, tas nozīmē, ka IMS projektā bija par 196 jeb par 29% kļūdām mazāk nekā KASKO projektā. Šis rādītājs ir vienīgais, kurā vienībtestu ietekme ir jūtami manāma.

Bija sagaidāms, ka IMS projektā būs daudz mazāk augstas un visaugstākās prioritātes un kritiskuma kļūdu, bet pētījuma rezultātā iegūtie dati rāda, ka kļūdu sadalījums pēc prioritātes un kritiskuma ir ļoti līdzīgs abos projektos, pie kam KASKO projektā šādu kļūdu ir pat procentuāli mazliet mazāks.

Arī pētot kļūdu atrašanas daudzumu laikā rezultāti bija pārsteidzoši. Ja KASKO projektā kļūdu sadalījums laikā vairāk līdzinās vispārpieņemtajam – projekta sākuma un beigu posmos ir maz atrasto kļūdu, bet vidusdaļā to ir visvairāk, tad IMS projektā atrasto kļūdu skaits ir ļoti nevienmērīgi sadalījies, pie tam projekta beigu posmā ir salīdzinoši ļoti daudz atrastu kļūdu, kur daudzas no tām ir pat augstas un visaugstākās prioritātes un kritiskuma. Šo tendenci varētu daļēji izskaidrot ar to, ka IMS projekta sākotnējā posmā nenotika intensīva izstrāde, bet beigu posmā tika saņemts ļoti daudz izmaiņu pieprasījumu, tomēr tik ļoti liela ietekme šim faktoram nevarētu būt.

Pētot kļūdu un darāmo darbu atgriešanu programmētājam, arī ir jūtama neliela vienībtestu pozitīvā ietekme IMS projektā, bet tā nav tik izteikta, kā bija gaidīts. Darāmo darbu atgriešana IMS projektā ir par 3.69% mazāk nekā KASKO projektā, bet kļūdu atgriešana IMS projektā ir par 8.3% mazāk nekā KASKO projektā. Šo rezultātu varētu daļēji izskaidrot ar to, ka IMS projektā strādāja ne tik pieredzējuši programmētāji kā KASKO projektā, tātad viņi nebija tik ļoti iepazinušies ar uzņēmuma specifiku un varēja kaut kur pieļaut kļūdas, kas uzņēmumā ilgāk strādājošiem programmētājiem negadītos. Tomēr šis fakts

pilnībā neizskaidro tik mazu atšķirību starp projektiem.

Kopumā izanalizējot visus datus, var secināt, ka vienībtesti ir pozitīvi ietekmējuši projekta testēšanas gaitu, bet šis uzlabojums nav pietiekoši liels. Varbūt šāds rezultāts ir tāpēc, ka šis bija pirmais projekts uzņēmumā, kurā tika veidoti vienībtesti, un tāpēc tie netika uzrakstīti pietiekoši lielā skaitā vai pietiekoši kvalitatīvi.

3 VIENĪBTESTU MAZĀS IETEKMES IZSKAIDROJUMS

Tā kā pētījums par vienībtestu ietekmi uz testēšanas kvalitāti ir devis tik pārsteidzošus rezultātus, kas ir pretrunā ar vispārpieņemto faktu, ka vienībtestēšana būtiski uzlabo projekta gaitu un kritiski samazina atrasto kļūdu daudzumu, tad jāizpēta, kāpēc ir radušies šādi dati un kā iegūtos rezultātus lai izskaidro.

3.1 Projektu integrācijas stratēģijas[1] [7] [8] [9]

Plānojot projekta izpildi, jāizdomā, kādā veidā integrēt komponentes, lai plānoto testēšanu veiktu pēc iespējas agrāk, kā arī pēc iespējas vienkāršāk, lai varētu sasniegt pēc iespējas lielāku testēšanas efektivitāti. Efektivitāte ir atkarīga no testēšanas izmaksām (testēšanas personāla un pielietojamo testēšanas rīku izmaksas) un no testēšanas gūtā labuma (atklāto kļūdu kritiskums un skaits). Testēšanas vadības uzdevums ir izplānot un ieviest projektā optimālo integrēšanas stratēģiju.

Reālajā dzīvē parasti dažādas programmatūras komponentes tiek izstrādātas dažādā laikā, ar dažu nedēļu vai pat mēnešu atšķirību. Nav pieļaujama situācija, kad testētāji gaida, līdz visas projekta daļas ir pabeigtas, un tikai tad veic integrācijas testus.

Pirmā izeja, kas redzama šīs problēmas risināšanā, ir integrēt komponentes tajā secībā, kādā tās tiek izstrādātas. Tas nozīmē, ka tiklīdz komponentei ir veikti visi vienībtesti un atrastās kļūdas novērstas, tiek pārbaudīts, vai to nevar integrēt ar kādu citu jau notestētu komponenti vai apakšsistēmu. Ja tāda komponente vai apakšsistēma ir, tad jaunā komponente tiek pievienota un tiek veikti integrācijas testi jau ar šo klātpievienoto komponenti.

Jārēķinās, ka šādi veicot integrācijas testus, sistēmā trūks vairāku komponenšu, bez kurām testus veikt nevar. Tas nozīmē, ka būs jāuzprogrammē šo trūkstošo daļu aizstājēji, lai komponentes varētu komunicēt. Jo ātrāk tiek sākta integrācijas testēšana, jo vairāk šādu daļu būs jāprogrammē. Tātad jāatrod tas projekta izstrādes laika posms, kurā sākt integrācijas testus tā, lai tie sāktos pēc iespējas agrāk un lai būtu pēc iespējas mazāk jāprogrammē šie daļu aizstājēji.

Katram projektam būs sava stratēģija, kas atkarīga no projekta individuālajiem

apstākļiem. Lai atrastu šo stratēģiju, jāizanalizē sekojoši projekta aspekti:

- Sistēmas arhitektūra – tā nosaka, cik sistēmā ir komponentes un kā tās ir atkarīgas viena no otras.
- Projekta plāns – tas nosaka, kad tiek izveidotas sistēmas konkrētās sadaļas un kad tām būtu jābūt gatavām testēšanai, nosakot implementācijas kārtību, būtu jākonsultējas ar testētāju vai testēšanas daļas vadītāju.
- Testēšanas plāns – nosaka, kuri sistēmas aspekti jātestē, cik intensīvi un kurā testēšanas līmenī tas jādara.

Testēšanas pārvaldniekam jāņem vērā šīs vispārīgās vadlīnijas un jāizdomā piemērotākā integrācijas stratēģija. Vispārīgas integrācijas stratēģijas ir aprakstītas nākošajās apakšnodaļās. Lejupejošā un augšupejošā integrācija var tikt pielietota sistēmās, kurās ir stingri definēta hierarhija, kas reālajā dzīvē nav bieži sastopama parādība. Parasti reālajā dzīvē tiek izmantots šo stratēģiju sajaukums.

*3.1.1 Lejupejošā (***top-down***) integrācija*[1] [7] [8] [9]

Testēšana sākas ar augšējā līmeņa komponenti, kas izsauc citas komponentes, bet neizsauc pati sevi. Visām pārējām komponentēm tiek programmēti aizvietotāji. Ja šis solis ir veiksmīgs, integrācija turpinās ar zemāka līmeņa komponentēm. Augstākais līmenis, kas jau ir notestēts, var tikt izmantots kā testdzinis.

Stratēģijas plusi – nav nepieciešami testdziņi vai arī ir nepieciešami tikai ļoti nedaudzi un vienkārši, jo augstākā līmeņa komponentes, kas jau ir notestētas, tiek izmantotas kā galvenā testēšanas vides daļa. Galvenās sistēmas komponentes ir notestētas iespējami agri, kā arī ir iespējams pēc iespējas agrāk nodemonstrēt klientam sistēmas visu galveno funkcionalitāti.

Stratēģijas mīnuss – zemāka līmeņa komponentes, kas vēl nav integrētas, jāaizstāj ar aizvietotājiem, kas var būt ļoti dārgs process.

*3.1.2 Augšupejošā (***bottom-up***) integrācija*[1] [7] [8] [9]

Testēšana sākas ar elementārām sistēmas komponentēm, kas neizsauc citas komponentes, izņemot operētājsistēmas funkcijas. No notestētajām komponentēm tiek

veidotas lielākas apakšsistēmas un tad šīs, integrētās daļas tiek testētas.

Stratēģijas plusi – nav nepieciešami komponenšu aizvietotāji, tādējādi ir vieglāk veidot testpiemērus.

Stratēģijas mīnusi – augstāka līmeņa komponentes jāsimulē ar testdziņu palīdzību, sistēma kopumā nav notestējama, līdz nav pievienotas visas komponentes.

*3.1.3 Sendviča (*Sandwich*) integrācija*[1] [9]

Stratēģija ir lejupejošās un augšupejošās stratēģiju kombinācija, kur sistēmas augšējiem līmeņiem tiek izmantota lejupejošā stratēģija, bet pakļautajiem līmeņiem – augšupejošā. Ja sistēmai ir tikai trīs līmeņi, tad metode ir vienkārši izprotama – zemākajam līmenim izmanto augšupejošo stratēģiju, augstākajam lejupejošo, tātad testēšana notiek vidējā līmenī. Ja sistēmai ir vairāki līmeņi, tad nav metodes, kādā stratēģijas sadalīt, vislabākā pieeja ir censties minimizēt nepieciešamo testdziņu un aizvietotāju skaitu. Modificētajā sendviča stratēģijā testēšana visos līmeņos notiek paralēli – vidējā līmenī tiek izmantoti gan testdziņi, gan aizvietotāji, augšējā līmenī – aizvietotāji, bet zemākajā līmenī – tikai testdziņi.

Stratēģijas plusi – visus sistēmas līmeņus var testēt paralēli vienu otram, tādējādi testēšana sākas ātrāk un uzreiz var notestēt komponentes, kas tikko pabeigtas.

Stratēģijas mīnusi – ir nepieciešami gan testdziņi, gan komponenšu aizvietotāji, kas var prasīt lielu resursu ieguldījumu.

3.1.4 Ad-hoc integrācija[1]

Komponentes tiek integrētas tādā secībā, kādā tiek pabeigta to izstrāde.

Stratēģijas pluss – tiek ietaupīts laiks, jo katra komponente tiek integrēta tik agri, cik vien iespējams.

Stratēģijas mīnuss – ir nepieciešami gan testdziņi, gan komponenšu aizvietotāji.

3.1.5 Mugurkaula integrācija[1]

Tiek uzbūvēts sistēmas skelets jeb mugurkauls (tā sistēmas daļa, kas ir vissvarīgākā), kurā tiek integrētas pārējās sistēmas komponentes.

Stratēģijas pluss – komponentes var tikt integrētas jebkādā secībā.

Stratēģijas mīnuss – sistēmas skeleta jeb mugurkaula uzbūvēšanai ir nepieciešams papildus darbs.

3.1.6 Lielā sprādziena integrācija[1] [9]

Tiek sagaidīts, kad izstrādātas visas sistēmas komponentes, un tās visas tiek ievietotas sistēmā vienlaicīgi. Šis integrācijas veids parasti tiek izvēlēts, kad izstrādātājiem vispār nav integrācijas stratēģijas.

Stratēģijas pluss – nav nepieciešami ne testdziņi, ne komponenšu aizvietotāji.

Stratēģijas mīnusi – ir zaudēts testēšanas laiks, jo tiek gaidīts līdz izstrādes beigām. Tā kā testēšanas fāzei parasti tiek atvēlēts maz laika, tad šis ir ļoti būtisks mīnuss. Otrs lielākais mīnuss ir tas, ka visas kļūdas notiek vienlaicīgi, ir ļoti grūti vai pat neiespējami sasniegt to, ka sistēma vispār darbojas. Kā arī ir ļoti sarežģīti un laikietilpīgi atrast vietu, kur kļūda radusies, kā arī izlabot kļūdu.

3.1.7 Uzņēmumā izmantotais integrācijas veids

Uzņēmumā nav stingri definētas integrācijas stratēģijas, no pirmā acu uzmetiena šķiet, ka tiek izmantota ad-hoc integrācija – tikko kāda no komponentēm ir izstrādāta, tā tiek iekļauta sistēmas būvējumā un nākošajā dienā jau ir pieejama uz testu vides un testētāji to jau var sākt testēt.

Tomēr, izpētot vairākus uzņēmuma projektu izstrādes plānus, kļūst skaidrs, ka integrācijas stratēģija vairāk līdzinās mugurkaula (*backbone*) stratēģijai. Izstrāde ir plānota tā, lai vispirms tiktu izstrādātas sistēmas pamatdarbības, un tad, kad tas paveikts, pamatdarbības tiek "apaudzētas" ar pārējo nepieciešamo funkcionalitāti. Tad, kad ir izstrādāts pamatdarbību skelets, testētāji sāk testēt sistēmu, paralēli atrasto kļūdu labošanai tiek izstrādāta pārējā sistēmas funkcionalitāte.

Sistēmas integrācijas stratēģiju vislabāk raksturo šāds piemērs. Tiek izstrādāts jauns apdrošināšanas produkts, piemēram – KASKO polises izdošana. Vispirms tiek izstrādāta polises izdošana ar minimāliem nepieciešamiem datiem – vēl nav ne polises prēmijas aprēķina, ne sasaistes ar CSDD, ne polises izdrukas. Vairāki lauki nav aizpildāmi, izvēlnēs ir

dažas pagaidu vērtības, bet polisi izdot var. Tad, kad šis posms ir pabeigts, tiek iesaistīti testētāji. Pēc tam sistēmā pakāpeniski tiek pievienota pārējā funkcionalitāte – prēmijas aprēķins, izdrukas, sasaiste ar CSDD, rēķinu veidošana, polises anulēšana, pārtraukšana, pielikumu veidošana, tiek pievienoti speciālie nosacījumi, ekspertu iesaistīšana un cita nepieciešamā funkcionalitāte.

Redzams, ka uzņēmuma integrācijas stratēģija pēc paša darbības veida līdzinās ad-hoc stratēģijai (tikko kas jauns ir izstrādāts, tas tiek integrēts sistēmā un uzreiz arī testēts), bet pēc integrācijas galarezultāta – mugurkaula stratēģijai – vispirms tiek izstrādātas un integrētas sistēmas pamatdarbības, pēc tam tām laika gaitā tiek likta klāt pārējā funkcionalitāte.

Tā kā lielākā daļa uzņēmuma projektu ir sistēmas papildināšana vai uzlabošana, tad parasti pamatdarbības jau ir izstrādātas citos sistēmas apgabalos un tās ir vai nu tieši jāpārņem no kopējās sistēmas funkcionalitātes, vai nu jāpārņem un jāpielāgo jaunajam projektam. Tas nozīmē, ka daļa no pamatkomponentēm jau ir notestēta iepriekšējos posmos un vienībtesti šīm komponentēm nekādas būtiskas kļūdas atklāt nevar. Potenciāli lielākais kļūdu skaits šādā gadījumā varētu rasties integrācijas procesā – kā šīs citu projektu komponentes sadzīvos ar jaunā projekta komponentēm.

Arī abos 2. nodaļā apskatītajos projektos (2.2 Salīdzināmie projekti) tika izmantota šī pieeja – pamatdarbības, kas visā pārējā sistēmā neatšķiras (polises izdošanas darbība, pielikuma izdošanas darbība, personas pievienošana klientu reģistram u.c.), tika paņemtas no iepriekšējiem projektiem. Tas nozīmē, ka pamatdarbību vienībtestēšanai nav lielas nozīmes, ko arī uzskatāmi parāda kļūdu statistika – projektos ar un bez vienībtestēšanas kļūdu skaits ir līdzvērtīgs.

3.2 Projektā iesaistīto cilvēku sagatavotība

Analizējot tieši šos, 2. nodaļā pētītos projektus (2.2 Salīdzināmie projekti), nevar nepieminēt to, ka IMS projekts uzņēmumā bija pirmais projekts, kurā tika veikti vienībtesti. Tas nozīmē, ka izstrādātājiem nebija pieredzes testu rakstīšanā un iespējams, ka tāpēc vienībtesti nav devuši lielu ietekmi uz testēšanas gaitu.

IMS projektā vienībtesti tika pielietoti apmēram 70% komponenšu, no kurām daļai netika rakstīti atsevišķi vienībtesti, bet komponentes tika apvienotas lielākās struktūrās, kas tika uzskatītas par vienu vienību un tai tad tika rakstīti testi. Daži vienībtesti aptvēra lielākas

vienību grupas, piemēram, tests, kas veic polises izdošanu vai prēmijas aprēķinu – šajās darbībās bija iesaistītas daudzas atsevišķas komponentes, no kurām daļai bija atsevišķi rakstīti vienībtesti.

Atlikušās 30% komponentes netika testētas, jo tās vai nu bija pietiekoši elementāras un tika uzskatīts, ka tām vienībtesti nav nepieciešami, vai nu bija pārņemtas no citiem projektiem.

Izpētot vienībtestu pārklājumu, redzams, ka dažās vietās tomēr bija nepieciešami vienībtesti vai tie bija nepieciešami mazliet komplicētāki – gandrīz nav tā saucamo "negatīvo testu", daudzi testi pārbauda tikai vienu, vienkāršāko gadījumu, kur būtu nepieciešamas rūpīgākas pārbaudes.

Jāatceras arī tas, ka 4 no 7 IMS projekta programmētājiem šis bija pirmais projekts uzņēmumā, tas arī ir viens no iemesliem, kāpēc IMS projektā integrācijas un sistēmas testēšanas laikā tika atrasts vairāk kļūdu nekā KASKO projektā.

Tā kā analītiķis, projekta specifikācijas rakstot, bija izlaidis daudzas, viņaprāt, pašsaprotamas lietas, tad radās daudzi pārpratumi, kad programmētājs bija nepareizi izpratis lietas būtību un gan pašā programmā, gan arī attiecīgajos vienībtestos ielaidis vienas un tās pašas kļūdas, tad saprotams, ka vienībtesti šādas kļūdas atklāt nevar, to atklāj tikai testētājs integrācijas testēšanas laikā. Lai gan tādu kļūdu nebija īpaši daudz, tomēr apmēram 5% no visām projektā atrastajām kļūdām bija tieši šī iemesla dēļ – programmētājs nebija pareizi izpratis analītiķa rakstīto, jo viņam nebija pieredzes darbā ar sistēmu.

Arī pašā integrācijas testēšanas procesā IMS projektā tika atrasts vairāk kļūdu tāpēc, ka produktu izstrādāja nepieredzējuši programmētāji, kas pieļāva kļūdas, ko programmētājs ar pieredzi uzreiz pamanītu un novērstu. Šis pats iemesls ir tam, ka IMS projektā ir salīdzinoši vairāk atgrieztu kļūdu un darāmo darbu. Jaunie programmētāji parasti izlaboja tieši to, kas rakstīts kļūdas ziņojumā, nenovērtējot, kādu ietekmi viņa labojums rada pārējās sistēmas daļās. Pieredzējis programmētājs, kas zina sistēmas strukturālo uzbūvi, jau apmēram nojauš, ko viņa labojums varētu ietekmēt un pirms tā ievietošanas testu vidē, pats veic nepieciešamās pārbaudes.

3.3 Komponenšu pārņemšana no citiem projektiem

Analizējot 2. nodaļā salīdzinātos projektus, jāņem vērā arī tas, ka IMS projektā bija

ļoti daudzas programmatūras daļas, kas tika vai nu pilnībā pārņemtas no KASKO projekta, vai nu pārņemtas un pielāgotas IMS projekta prasībām. Kā jau minēts nodaļā 3.1.7 Uzņēmumā izmantotais integrācijas veids, uzņēmumā ir pieņemts pārņemt no citiem produktiem pamatdarbības, kas visā sistēmā kopumā neatšķiras. Tomēr šajā gadījumā iet runa par daudz vairāk kā tikai pamatdarbību pārņemšanu.

Tā kā KASKO projekta un IMS projekta prasības KASKO produktam bija ļoti līdzīgas, tad liela daļa no KASKO produkta izstrādātā koda tika pārņemta IMS projektā, veicot nepieciešamos labojumus. IMS projekta Unipolises produkta izstrādē arī liela daļa koda tika pārņemta no citiem, agrāk realizētiem uzņēmuma projektiem. Rezultātā lielākā daļa no komponentēm jau bija notestētas agrākos projektos, tāpēc vienībtestēšanas laikā šīm komponentēm atrastās kļūdas nebija daudz un nevarēja pozitīvi ietekmēt testēšanas gaitu.

Projektos, kuros tiek pārņemta daļa no funkcionalitātes no kāda cita projekta vai produkta, lielākā problēma ir nevis tajā, ka komponentes var būt kļūdainas, jo tas jau ir notestēts tad, kad komponentes tiek izveidotas, bet gan tajā, kā šīs jaunpieņemtās un pārveidotās komponentes integrēsies ar pārējām projekta komponentēm, kas arī ļoti labi atspoguļojas IMS projekta kļūdu statistikā – tā pēc apjoma gandrīz neatšķiras no KASKO projekta, kurā vienībtesti vispār netika veikti.

Tātad, ņemot vērā to, ka liela daļa no komponentēm IMS projektā bija aizgūtas no citiem projektiem, kuros tās jau bija notestētas, vienībtestu ietekmei šajā projektā būtu jābūt minimāli jūtamai, ko pierāda arī iepriekšējā nodaļā aprakstītā statistika. Vienībtesti bija nepieciešami tikai tai programmatūras komponenšu daļai, kas projektā tika programmētas no jauna.

SECINĀJUMI

Maģistra darbā ir izpētīts uzņēmuma izstrādes modelis un tas salīdzināts ar teorētisko V-modeli. Ir izpētīta vienībtestu ietekme uz testēšanas gaitu, salīdzinot divus projektus, no kuriem vienā tika izstrādāti vienībtesti, bet otrā – nē, kā rezultātā nonākts pie secinājuma, ka vienībtestu ieviešana uzņēmumā nav devusi būtiskus uzlabojumus šo projektu izstrādes dzīves ciklā.

Ir piedāvāti vairāki izskaidrojumi, kāpēc vienībtestu ietekme uz testēšanas gaitu ir tik neliela. Viens no izskaidrojumiem attiecas uz visiem uzņēmuma projektiem. Uzņēmumā ir ieviesta mugurkaula (*backbone*) integrācijas stratēģija un pamatfunkcionalitāte uzņēmumā izstrādātajiem projektiem pārsvarā ir viena un tā pati un tiek pārņemta un pārveidota, lai atbilstu jaunā projekta prasībām. Vienībtestu pielietošana šai "mugurkaula" daļai nedod lielu kļūdu skaita samazinājumu, jo šī funkcionalitāte lielākoties jau ir notestēta agrākajos projektos, tāpēc testu izpildes rezultātā nekādas būtiskas kļūdas atklāt nav iespējams un lielāko daļu kļūdu, ja tādas rodas, ir iespējams atrast tikai integrācijas testu laikā, kad šīs pārņemtās daļas tiek integrētas ar no jauna veidojamām projekta specifiskām daļām.

Šis izskaidrojums nav pretrunā arī ar vispārējo teoriju. Šo konkrēto gadījumu var pielīdzināt komercproduktam, kas tiek nopirkts un tad pielāgots konkrētām pircēja vajadzībām.[1] Šādā gadījumā vienībtestus vai nu nav jāraksta vispār, vai nu jāraksta tikai tām produkta daļām, kas ir pārveidotas. Vienībtestu rakstīšanai tām produkta daļām, kas nav mainītas, nedos nekādu labumu. Šajā gadījumā par "komercprodukta" neaiztikto daļu var uzskatīt pamatfunkcionalitāti, kas ir pārņemta no citiem projektiem un jau notestēta, bet par pielāgoto daļu – jaunizveidotās vai izmainītās komponentes.

Pārējie divi izskaidrojumi ir projektu specifiski, bet tie jāņem vērā arī nākošajos projektos, jo var gadīties, ka šīs situācijas atkārtojas. Viens no izskaidrojumiem ir līdzīgs iepriekš minētajam – tā kā salīdzināmie projekti bija ļoti līdzīgi un tas, kurā tika veikti vienībtesti, tika veikts vēlāk, tad ļoti liela daļa funkcionalitātes bija pārņemta no iepriekš izstrādātā projekta, kurā netika veikti vienībtesti. Papildus augstāk minētajai pamatfunkcionalitātei, kas raksturīgs lielākajai daļai projektu, šajā projektā tika pārņemta arī liela daļa no pārējās sistēmas funkcionalitātes. Tas vēl vairāk samazina vienībtestu lomu, jo šajā projektā bija daudz vairāk pārņemtas funkcionalitātes, kas iepriekšējos projektos jau bija notestēta. Rezultātā vienībtesti šajā projektā nebija nepieciešami lielai daļai funkcionalitātes,

jo tie nevar atklāt nekādas būtiskas kļūdas jau notestētās vienībās.

Trešais izskaidrojums ir tāds, ka maģistra darbā pētāmais projekts uzņēmumā bija pirmais, kurā tika veikti vienībtesti, tāpēc vienībtestu izstrādē un izpildē varēja rasties nepilnības programmētāju zināšanu trūkuma dēļ. Kā arī šī projekta izstrādē tika iesaistīti programmētāji, kam šis bija pirmais projekts uzņēmumā, tāpēc tie nebija pilnībā iepazinušies ar uzņēmuma darbības jomas specifiku, tā rezultātā radās daudzas kļūdas, kad programmētājs nebija pareizi sapratis savu uzdevumu un gan pašā sistēmā, gan arī attiecīgajos vienībtestos ielaidis vienas un tās pašas kļūdas. Šis faktors ietekmēja gan to, kāpēc vienībtesti ir tik maz ietekmējuši projekta gaitu, gan arī palielināja integrācijas un sistēmas testēšanas laikā atrasto kļūdu skaitu.

Lai gan šī problēma, ka programmētājiem nebija iepriekšējas pieredzes vienībtestu izstrādē, uzņēmumā uz šo brīdi jau ir novērsts, jo ir izstrādāti vairāki projekti ar vienībtestiem, turpmākajos projektos jāņem vērā risks, kāds pastāv, ja vienā projektā iesaista vairākus programmētājus ar mazu pieredzi vai nu programmēšanā vispār, vai nu uzņēmuma izstrādes specifikā. Ja šādu situāciju pieļauj, tad jārēķinās, ka testēšanas posmos tiks atrasts daudz vairāk kļūdu un gan kļūdas, gan darāmie darbi tiks atgriezti vairāk, tādējādi palielināsies izstrādes laiks un izmaksas.

Rezultātā piedāvājamais risinājums ir, ņemot vērā uzņēmuma specifiku, samazināt vienībtestu izstrādes laiku un apjomu, neveidojot testus tām projekta komponentēm, kas vai nu tiek tieši pārņemtas no citiem projektiem, vai nu tiek mazliet izmainītas. Vienībtestus izstrādāt tikai tām komponentēm vai komponenšu grupām, kas tiek izstrādātas no jauna vai nu pārņemtas no citiem projektiem un būtiski izmainītas. Tiek piedāvāts arī būtiski palielināt integrācijas testēšanas apjomu un integrācijas testēšanu sākt pēc iespējas ātrāk, jo pastāv risks, ka komponentes, kas pārņemtas no citiem projektiem, nepareizi sadarbosies vai nu savā starpā (ja tās tiek pārņemtas no dažādiem projektiem), vai nu ar projektā jaunizveidotajām komponentēm. Un šādas kļūdas var atklāt tikai integrācijas testēšanas laikā.

Tomēr jāatceras, ka šajā darbā sīkāk ir izpētīti tikai divi projekti, kas ir par maz, lai veiktu striktus spriedumus par visu uzņēmumu. Tāpēc autores ieteikums ir pirms katra projekta izpētīt, vai šajā darbā aprakstītā situācija tam atbilst un tikai tad, ja tā ir, samazināt vienībtestu izstrādes apjomu.

Vēl viens būtisks ieteikums uzņēmuma projektu izstrādes cikla uzlabošanai, kas radies šī darba izstrādes laikā, ir pēc katra projekta noslēguma analizēt kļūdu un darāmo darbu

reģistrēšanas un uzturēšanas sistēmas datus. Šajā sistēmā ir ļoti daudz noderīgas informācijas, kas vai nu vispār netiek apstrādāta un ņemta vērā , vai nu tiek analizēta tikai virspusēji. Kļūdu un darāmo darbu statistikas apkopošana un izpēte var palīdzēt analizēt projektu gaitu, kā arī pieņemt lēmumus turpmākai uzņēmuma izstrādes un testēšanas procesu uzlabošanai.

IZMANTOTĀ LITERATŪRA

[1] A. Spillner, T. Linz, H. Schaefer „Software testing Foundations" 1st Edition 2006, dpunkt.verlag GmbH, Heidelberg 266 lpp.

[2] R. Pressman"Software engineering: a practitioner's approach " 6th Edition 2005, McGraw-Hill Higher Education, Boston 912 lpp.

[3] Brīvā enciklopēdija Wikipedia [tiešsaiste]. – [atsauce 15.05.2008.]. Pieejams: http://en.wikipedia.org/

[4] Interneta vārdnīca [tiešsaiste]. – [atsauce 15.05.2008.]. Pieejams: http://dictionary.reference.com/

[5] R. Sorensen "A Comparison of Software Development Methodologies", Software Technology support center mājas lapa [tiešsaiste]. – [atsauce 15.05.2008.]. Pieejams: http://www.stsc.hill.af.mil/crosstalk/1995/01/Comparis.asp

[6] E. Notenboom "Multiple V-model in relation to testing", Universität Siegen mājas lapa [tiešsaiste]. – [atsauce 15.05.2008.]. Pieejams: http://pi.informatik.uni-siegen.de/stt/23_1/01_Fachgruppenberichte/FG217/06_MultipleV.pdf

[7] Dr. Jerry Gao "Software Testing & Strategies", San José State University College of Engineering mājas lapa [tiešsaiste]. – [atsauce 15.05.2008.]. Pieejams: www.engr.sjsu.edu/gaojerry/course/287/test-**strategy**.ppt

[8] Bruce R. Maxim "Software Testing Strategies" University of Michigan-Dearborn mājas lapa [tiešsaiste]. – [atsauce 15.05.2008.]. Pieejams: www.engin.umd.umich.edu/CIS/course.des/cis375/ppt/lec25.ppt

[9] Izdales materiāli, "Software Testing ", University of Alaska Anchorage mājas lapa [tiešsaiste]. – [atsauce 15.05.2008.]. Pieejams: www.math.uaa.alaska.edu/~afkjm/cs401/handouts/testing.ppt

Ar savu parakstu apliecinu, ka maģistra darbs veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai. **Piekrītu sava darba publicēšanai internetā.**

Autors: _____

(Autora paraksts)

Ar savu parakstu apliecinu, ka esmu lasījis augstāk minēto maģistra darbu un atzīstu to par p i e m ē r o t u / n e p i e m ē r o t u (nevajadzīgo svītrot) aizstāvēšanai Latvijas Universitātes datorzinātņu maģistrantūrā.

Darba vadītājs: _____

(Vadītāja paraksts)

Darbs iesniegts Datorikas nodaļā _____.

(Iesniegšanas datums)

Ar šo es apliecinu, ka darba elektroniskā versija ir augšupielādēta LU informatīvajā sistēmā.

Metodiķe: _____.

(Metodiķes paraksts)

Recenzents: _____

(Recenzenta paraksts)

Darbs aizstāvēts maģistra gala pārbaudījuma komisijas sēdē

_____ prot. Nr. _____, vērtējums _____

(Darba aizstāvēšanas datums)

Komisijas sekretārs: _____

(Sekretāra paraksts)