LATVIJAS UNIVERSITĀTE

DATORIKAS FAKULTĀTE

Prioritāšu mehānisma izmantošana analītisku algoritmu testēšanā

MAĢISTRA DARBS

Autore: **Ilga Baiža**

Stud. apl. Nr. ib11113

Darba vadītājs: Dr. sc. comp. Jānis Bičevskis

RĪGA, 2017

ANOTĀCIJA

Darbs veltīts sarežģītu algoritmu izpildes testēšanai sistēmas uzturēšanas laikā. Lietotājam, kurš nav IT speciālists, ir nepieciešams pārliecināties, ka algoritms, veicot datu atlasi un analīzi, korekti tos interpretē, un algoritma aprēķina rezultāts ir pareizs. Darbā tiek apskatīta resursu vadības sistēmas Horizon patēriņa starpības sadales koeficienta analīze. Izveidots algoritma apraksts un blokshēmas.

Tiek piedāvāts problēmu atrisināt ar starprezultātu izvadi izmantojot prioritāšu mehānisma palīdzību. Risinājums pielietots patēriņa starpības sadales koeficienta analīzei, atzīmējot starprezultātu izvades punktus atbilstoši noteiktajām prioritātēm. Analizējot klientu pieteikumus, autors secināja, ka starprezultātu izvade izmantojot prioritāšu mehānismu uzlabotu sistēmas lietotāju izpratni par algoritma darbību.

Atslēgvārdi: testēšana, prioritāšu mehānisms, nekustamo īpašumu pārvaldība

ABSTRACT

The theme of the Master's thesis: **Priorities in debugging for analytic algorithms**

Work is dedicated to complex algorithm execution testing in system's maintenance time. User, who is not an IT professional, needs to make sure that the algorithm through data selection and analysis interprets them correctly and the result of the algorithm is correct. Work examines resource management system's Horizon consumption difference distribution coefficient analysis. Algorithm's description and flowcharts have been made.

It is proposed to solve the problem with intermediate output using priority mechanism. The solution has been applied to consumption difference distribution coefficient analysis, noting the intermediate output points in accordance with the priorities set out. Analyzing customer applications author concluded that the intermediate output using priority mechanisms would improve user awareness of the algorithm operation.

Key words: testing, priorities in debugging, real estate management

AUTOREFERĀTS

Autors darbā ir apskatījis analītisku algoritmu izprašanas problēmu. Darba autors piedāvā algoritma starprezultātu iegūšanai izmantot prioritāšu mehānismu, kas ļauj starprezultātus saglabāt tikai tad, kad tas nepieciešams un tikai tādā apjomā kā tas nepieciešams. Šādā veidā iespējams analītiskiem algoritmiem nepasliktināt ātrdarbību un iegūt izpratni par rezultāta rašanos.

Maģistra darba ietvaros apskatīta testēšanas teorija un testēšanas automatizācija. Salīdzināta vienībtestēšana ar lietotāja saskarnes testiem un izdarīti secinājumi par to plusiem un mīnusiem. Pēc iegūtās informācijas izvirzīta darbā apskatāmā problēma un veikta literatūras izpēte iespējamiem risinājumiem. Autors izmantotajā literatūrā un avotos iekļāvis 16 dažādas vienības. Izpētei izmantotas grāmatas, raksti žurnālos un rakstu krājumos. Izmantoti arī Latvijas Republikas tiesību akti, lai izskaidrotu biznesa prasības, kas no tiem cēlušās.

Darbā aplūkotā problēma ir apskatīta pietiekoši detalizēti, izklāsts nav virspusīgs, pētījuma rezultāti ir sniegti pārskatāmi ar ilustratīviem attēliem. Autors darba ietvaros izveidojis sistēmas Horizon nekustamo īpašumu pārvaldības moduļa patēriņa starpības sadales koeficienta analīzes aprakstu un algoritma blokshēmas, sistēmas dokumentācijā pieejamais darbības apraksts bija neprecīzs un neaktuāls, shēmas šim algoritmam līdz šim nebija zīmētas. Autors blokshēmās atzīmējis un aprakstījis potenciālus starprezultātu punktus un sadalījis tos prioritātēs, lai varētu iegūt dažādas detalizācijas starprezultātu izvadi. Izmantojot izveidoto aprakstu, blokshēmas un starprezultātu izvadi ar prioritāšu mehānismu autors izanalizējis klientu pieteikumus par problēmām ar šo analīzi.

Veicot literatūras avotu izpēti autors apskatīja kādi risinājumi tiek piedāvāti prasību labākai izpratnei. Autors darbā secina, ka vizuālizācija veicina izpratni par algoritma darbību un palīdz izskaidrot tā rezultātu, kas atbilst literatūras avotos rakstītajam. Lai pārliecinātos par risinājumu derīgumu autors analizēja klientu pieteikumus un secināja, ka starprezultātu izvade dod papildus informāciju un prioritāšu mehānisms atļauj izvērtēt kādas detalizācijas dati ir nepieciešami, lai apskatītu problēmu. Risinājums šobrīd nav ieviests praksē, jo tas nav atkarīgs no darba autora. Pētījuma rezultāti tiks iesniegti sistēmas izstrādātājam un tas pieņems lēmumu par šādas funkcionalitātes iekļaušanu sistēmā

Lai pārliecinātos, ka darbā nav valodas kļūdu, autors lieto pareizrakstības pārbaudes rīkus, kā arī ir veicis darba caurskati. Darbā ir izmantota latviešu valodā oficiāli pieņemtās nozares terminoloģija. Darba autors ir iepazinies ar Datorikas fakultātes maģistra darba izstrādes un

aizstāvēšanas metodiskajiem norādījumiem. Darbā visi attēli un idejas, kas ir aizgūti no citiem avotiem ir ar literatūras atsaucēm.

SATURS

APZĪMĒJUMU SARAKSTS

Labais līgums – dzīvokļa apsaimniekošanas līgums, kura dati neatbilst ministru kabineta noteikumos 19.[1] punktā minētajiem nosacījumiem un patēriņa starpības sadales koeficienta analīzē tiek piešķirts mainīgais nulle.

Sliktais līgums – dzīvokļa apsaimniekošanas līgums, kura dati atbilst ministru kabineta noteikumos 19.[1] punktā minētajiem nosacījumiem, patēriņa starpības sadales koeficienta analīzē tiek piešķirts mainīgais viens un tam ir piestādāma ūdens patēriņa starpība.

Skaitītājs (sistēmā Horizon) – pakalpojuma pieslēgums konkrētam objektam, piemēram, dzīvoklim ir pievilkta aukstā ūdens caurule, pa kuru var tikt piegādāts ūdens.

Skaitītāja eksemplārs (sistēmā Horizon) – fiziska mērierīce, kas nodrošina patērētā daudzuma uzskaiti pakalpojumam.

IEVADS

Klientam lietojot sistēmas, kas atvieglo viņa ikdienas darbu un efektivizē uzņēmuma procesus, ir svarīgi uzticēties sistēmas rezultātiem. Sistēmu ieviešana uzņēmumā var aizvietot cilvēka manuālu darbu veicot aprēķinus vai analīzi, ja šos procesus ir iespējams aprakstīt sistēmas algoritmā. Kļūdas sistēmā tās lietotājam var radīt šaubas, kā rezultātā var tikt veikts dubults darbs, jo lietotājs paralēli sistēmai veiks darbības manuāli, lai pārliecinātos ka sistēma strādā korekti.

Testēšana ir viens no svarīgākajiem programmatūras izstrādes dzīvescikla posmiem, mūsdienās testēšanas pamatmērķis ir nodrošināt stabili strādājošu produktu, kura aprēķini ir uzticami, un uz kura sniegto rezultātu pamata var pieņemt finansiāli svarīgus lēmumus. Regresa testēšana ir nozīmīgs testēšanas veids, ar kura palīdzību nodrošināt, ka klientam tiek piegādāts augstas kvalitātes produkts. Pamatojoties uz to, ka regresa testēšana ir laikietilpīgs un atkārtojošs process no versijas uz versiju, to nepieciešams automatizēt, vai vismaz veikt analīzi kāpēc tas nav izdarāms, kā arī pie šīs analīzes periodiski atgriezties. Tā kā testēšanai paredzētie resursi ir ierobežoti un testēšanas pilnības sasniegšana vēljoprojām ir aktuāla problēma, tad testēšanas resursi ir jāizmanto efektīvi. Viena no iespējām ir samazināt ar kļūdu beigušos testu apstrādes laiku.

Bieži daudz laika tiek iztērēts mēģinot saprast sistēmas darbību un noteikt vai sistēma darbojas korekti. Ja sistēmas lietotājs nezina kā sistēma veic konkrētu darbību, tam nav iespējams izprast darbības rezultātu un uzticamība samazinās vēl vairāk.

Maģistra darbā autors pētīs iespēju uzlabot analītisku algoritmu testēšanas iespējas. Maģistra darbā tiks veikta literatūras par testēšanu apskate, aprakstīta problēma un apskatīti iespējamie risinājumi un iepriekš veiktie pētījumi par šāda veida risinājumiem. Autors maģistra darbā veiks sistēmas Horizon nekustamo īpašumu pārvaldības moduļa aprēķinu un analīzes algoritmu izpēti un apskatīs iespējas uzlabot testēšanas procesu izmantojot prioritāšu mehānismu.

Maģistra darba mērķis ir izpētīt iespējas analītisku algoritmu testēšanas uzlabošanā un izmantojot prioritāšu mehānismu aprakstīt algoritma starprezultātu punktus.

Lai sasniegtu kursa darbā izvirzīto darba mērķi, tika uzstādīti šādi darba uzdevumi:

a) Veikt apskatu par testēšanu un tās pamatjēdzieniem, testēšanas automatizāciju.
b) Apkopot pieejamo informāciju par vienībtestēšanu un lietotāja saskarnes testēšanu.
c) Aprakstīt risināmo problēmu un apkopot iespējamos risinājumus.
d) Izveidot patēriņa starpības sadales koeficienta analīzes aprakstu un algoritma blokshēmu

e) Atzīmēt patēriņa starpības sadales koeficienta analīzes algoritma iespējamos starprezultātu punktus un to prioritātes
f) Apkopot rezultātus un secinājumus

Darba pirmajā nodaļā veikts pārskats par testēšanu un tās jēdzieniem, otrajā nodaļā apskatīta testēšanas automatizācija veicot sīkāku ieskatu regresa testēšanā un testa virzītā izstrādē. Trešajā nodaļā aprakstīta vienībtestēšana un ceturtajā nodaļā apskatīti lietotāja saskarnes testi. Izdarīti secinājumi, ka salīdzinot vienībtestēšanu ar lietotāja saskarnes testiem viena no vienībtestu priekšrocībām ir programmētāja iespēja testus, kas beigušies ar kļūdu, izpētīt izmantojot atkļūdošanas rīkus. Darba piektajā nodaļā analizēta apskatāmā problēma un tās iespējamie risinājumi. Sistēmu lietotāji analītisku algoritmu gadījumā sastopas ar problēmu, ka sistēmas izdotais rezultāts nav izskaidrojams un viņi to nevar izdarīt ar saviem spēkiem. Kā iespējamie risinājumi tiek izvirzīti algoritma apraksta vizualizācija un prioritāšu mehānisma ieviešana starprezultātu punktiem ar papildus informācijas izvades iespējām. Darba sestajā nodaļā aprakstīta pētāmā resursu vadības sistēma Horizon. Septītājā nodaļa aprakstīti patēriņa starpības sadales koeficienta analīzes pamatprincipi. Pirmajā apakšnodaļā izveidots algoritma apraksts un blokshēmas, kas atspoguļo tā darbību. Otrajā apakšnodaļā aprakstīti starprezultātu punkti un prioritāšu mehānisma princips. Trešajā apakšnodaļā veikta problēmu analīze, apskatot sistēmas lietotāju pieteikumus un analizējot tos izmantojot blokshēmas un prioritāšu mehānisma rezultātu starppunktus.

1. PĀRSKATS PAR TESTĒŠANU UN JĒDZIENIEM

Ražošanas produktiem to daļas vai gala produkts parasti tiek pārbaudīti vai tie atbilst prasībām. Vispārīgi tas, kas atbilst ražošanas produktiem, ir piemērojams arī programmatūras izstrādei. Tā kā programmatūras izstrādes produkts nav fizisks, tad tā novērtējums ir sarežģītāks. Vienīgais veids kā tieši izmeklēt produktu ir lasīt izstrādes dokumentāciju, bet tādā veidā netiks pārbaudītas programmatūras dinamiskā uzvedība. Šāda veida pārbaudi veic testēšana programmatūru izpildot uz datora. Tās darbības nepieciešams salīdzināt ar prasībām. Testēšana samazina programmatūras lietošanas risku. [14] Piemēram, kļūdas grāmatvedības sistēmā var radīt nepareizus datus par saistībām pret kreditoriem vai debitoru saistītām, kā ari nepareizus uzņēmuma finanšu rādītājus un pat likumdošanas pārkāpšanu.

Situāciju var klasificēt par nepareizu tikai tad, kad ir zināms kāds ir sagaidāmais rezultāts. Programmatūras testēšanā izdala terminus problēma, kļūme, defekts un kļūda [2]. Problēma ir programmatūras darbības neatbilstība, tam ko ir gaidījis lietotājs. Atšķirību starp rezultātu, kas tiek iegūts izpildot testu un sagaidāmo rezultātu, kas definēts prasībās vai specifikācijā, sauc par kļūmi. Šis jēdziens apraksta situāciju, kad programmatūras lietotājs saskaras ar problēmu lietošanā. Kļūmes izraisa defekti programmatūrā. Katrs defekts programmatūrā ir kopš tās izstrādes vai izmaiņām, bet to iespējams ieraudzīt tikai lietojot programmu, kad ieraugām kļūmes. Defektus izraisa kļūdas, ko radījis cilvēks. Kļūda ir programmētāja darbība, kā rezultātā viņš uzraksta programmu ar defektu. [2, 14] Problēmas rodas programmētāja kļūdas dēļ. Defekti var būt arī dokumentācijā, ne tikai programmā. Liela daļa programmatūras problēmu sakņojas tās prasību, specifikāciju defektos [2].

Lietotāja atrastā problēma var norādīt uz kļūdām programmētāja darbā, defektiem programmatūras specifikācijā, piemēram, neprecīzām prasībām, pretrunām starp prasībām vai trūkstošām prasībām. Problēma var norādīt arī uz jaunu prasību nepieciešamību, no kā izriet nepieciešamība attīstīt programmatūru un pievienot jaunu funkcionalitāti. [2]

Šobrīd nav zināma programmatūras sistēma bez kļūdām, apskatot netriviālas sistēmas[14]. Bieži kļūdas netiek atklātas izņēmumu gadījumos, kuri nav apskatīti izstrādes un testēšanas procesā. Izsakot pieņēmumu, ka neviena sistēma nav bez kļūdām, netiek apgalvots, ka visas sistēmas ir neuzticamas. Ir zināmas pietiekami daudz sistēmas, kas darbojas dienu no dienas bez problēmām.

Testēšana nepierāda kļūdu neesamību. Lai to izdarītu testam būtu jāizpilda programma visos iespējamos veidos ar visām iespējamām datu vērtībām. Praksē šāds testu skaits tiek uzskatīts par neierobežotu, šāda veida testēšana visām kombinācijām nav iespējama. Tā kā pilna testēšana nav

iespējama, tad testēšanas ieguldījumam ir jābūt samērojamam ar rezultātu. [14] Testēšanai ir jāturpinās tik ilgi kamēr defektu atrašanas un labošanas izmaksas ir zemākas nekā izmaksas par kļūmi ekspluatācijā [11]. Teorētiski šāds uzskats ir pareizs, bet praktiski nekad nav iespējams zināt kādas kļūmes ir ekspluatācijā un kādas izmaksas tās var radīt, līdz tās tiek atklātas un jau rada zaudējumus. Sistēmas ar augstu darbības risku ir jātestē pilnīgāk, nekā sistēmas, kuras kļūmju dēļ nerada lielus zaudējumus. Resursu vadības sistēmai nepareiza datu saglabāšana vai aprēķinu veikšanas nozīmē ļoti augstu risku. Var tikt ietekmēti finanšu dati, sabojātas attiecības ar sistēmas lietotāja klientiem un sadarbības partneriem, kas nozīmē papildus zaudējumus. Sistēma, kurā regulāri ir kļūmes, kas rosina tās lietotājus veikt paralēlu uzskaiti un dubultas pārbaudes, var kļūt neuzticama un veicināt sistēmas nomaiņu.

Tā kā pilna pārtestēšana nav iespējama ir ļoti svarīgi izprast kā lietot ierobežotos testēšanas rezultātus. Testus nepieciešams veidot un izpildīt strukturēti un sistemātiski. Testēšana jāplāno, lai būtu iespējams atrast pēc iespējas vairāk defektu, bet lai izvairītos no liekiem testiem, kuri nevarēs atrast defektus vai sniegt informāciju par programmatūras kvalitāti. [14]

Pieņemot, ka projektēšanas laikā problēmas atrašana un izlabošana maksā 1 vienību, tad implementācijas fāzē tās atrašanas un izlabošanas izmaksas jau jāreizina ar 10, bet sistēmtestēšanai vai akcepttestēšanas fāzē – jāreizina ar 100. Ja klients vai lietotājs problēmu atrod jau ekspluatācijas laikā, tad cena jau jāreizina ar 1000. [2]

Lai programmatūras izstrāde notiktu strukturēti un kontrolēti tiek izmantoti dažādi programmatūras izstrādes modeļi un izstrādes procesi. Šobrīd pasaulē ir zināmi dažādi modeļi. Populārākie no tiem ir ūdenskrituma modelis, V modelis, spirāles modelis un dažādi inkrementāli modeļi un spējās metodes. Testēšanas parādās katrā no šiem dzīves cikla modeļiem, bet ar dažādu nozīmi un dažādā apjomā. [14] Jāatceras, ka reālajā dzīvē organizācijās atšķirtas viena un tā paša dzīves cikla izmantošana. Var tikt veiktas nobīdes no ideālā dzīves cikla, lai pielāgotos konkrētam projektam. [4]

Pirmais pamata modelis, ko apskatīsim ir ūdenskrituma modelis. Šis modelis ir ļoti plaši pazīstams. Tikai, kad viens izstrādes līmenis ir noslēdzies tiek uzsākts nākamais. Šajā modelī izšķir sistēmas prasību, programmatūras prasību, analīzes, dizaina, programmēšanas, testēšanas un lietošanas līmeņus. Tikai starp viens otram sekojošiem līmeņiem ir atgriezeniskā saite, kas ja nepieciešams atļauj veikt izmaiņas iepriekšējā līmenī. Galvenais trūkums šim modelim ir tas, ka testēšana tiek saprasta kā vienreizēja darbība projekta beigās tieši pirms nodošanas lietošanai. [14]

Ūdenskrituma moduļa uzlabojums ir V-modelis. Tajā izstrādes aktivitātes ir nodalītas no pārbaudes aktivitātēm. Testēšanas aktivitātes notiek paralēli ar izstrādes aktivitātēm visa izstrādes procesa laikā. Tās ir sadalītas attiecīgos testu līmeņos. Šajā modelī testēšanas aktivitātēm ir tik pat svarīga loma kā izstrādei un tās ir savstarpēji saistītas. Šajā modelī ir divi zari – izstrādes, kurā notiek projektēšana un programmēšana, bet otrā zarā norisinās testēšana. Izstrāde tiek sadalīta prasību definēšanā, funkcionālajā sistēmas projektējumā, tehniskajā sistēmas projektējumā, komponenšu specifikācijā un programmēšanā. Visvieglāk kļūmes ir atrast līdzīgas abstrakcijas procesā izstrādēm, tāpēc katram izstrādes procesam atbilst līdzīgas abstrakcijas testēšanas līmenis. Komponentu testēšana pārbauda, ka katra sistēmas komponente strādā atbilstoši tās specifikācijai. Integrāciju testēšana pārbauda vai komponenšu grupas sadarbojas tā kā tas norādīts tehniskajā sistēmas projektējumā. Sistēmas testēšana pārbauda kopumā atbilst noteiktajām prasībām. Akcepttestēšana pārbauda vai sistēma atbilst līgumā noteiktajām prasībām no klienta skatu punkta. V-modelis var radīt iespaidu, ka testēšana tiek uzsākta salīdzinoši vēlu, bet šāds uzskats ir nepareizs. Sagatavošanās testēšanai – testu plānošana un kontrole, testu analīze un veidošana – notiek vienlaicīgi ar attiecīgo izstrādes fāzi. [14]

Spējās izstrādes modeļi ar atšķiras no tradicionāliem izstrādes dzīves cikliem ar to kā testēšanas un izstrādes aktivitātes ir integrētas, ar projekta darba produktiem, ieejas un izejas kritērijiem dažādos testēšanas līmeņos, rīku izmantošana un cik neatkarīgi var tikt veikta testēšana. Viena no galvenajām spējās izstrādes atšķirībām no tradicionāliem izstrādes dzīves cikliem ir ļoti īsu iterāciju ideja. Katras iterācijas rezultāts ir strādājoša programmatūra, kas piegādā vērtību biznesa ieinteresētajai pusei. Katras iterācijas sākumā notiek tās plānošana. Kad iterācijas tvērums ir nodefinēts, izvelētie lietotājstāsti tiek izstrādāti, integrēti un testēti. Iterācijas ir dinamiskas, izstrādes, integrācijas un testēšanas aktivitātes notiek viscaur katrai iterācijai ar ievērojamu paralēlismu un pārklāšanos. Testēšanas aktivitātes notiek viscaur iterācijai nevis kā nobeiguma aktivitāte. Tipiski testētāja darba produkti spējās izstrādes projektā iekļauj automatizētus testus, dokumentāciju, piemēram, testa plānus, manuālos testus, defektu ziņojumus un testu rezultātu žurnālus. Spējās izstrādes laikā tradicionālie testēšanas līmeņi pārklājas biežo izmaiņu dēļ. Iterācijas laikā lietotājstāsts tipiski iziet cauri divām galvenajām testēšanas aktivitātēm - vienībtestēšanai un raksturiezīmju testēšanai. Raksturiezīmju testēšana dažreiz tiek sadalīta divās aktivitātēs. Verifikācijas testēšana, kas bieži ir automatizēta un iekļauj testēšanu pret lietotājstāsta akceptēšanas kritērijiem. Validācijas testēšana, kas parasti notiek manuāli un tiek pārbaudīts vai raksturiezīme ir piemērota lietošanai, lai uzlabotu progresa redzamību un iegūtu īstu atgriezenisko

saiti no biznesa iesaistītajām pusēm. Vēl paralēli iterācijas laikā notiek regresa testēšana , kas iekļauj atkārtotu automatizētu vienībtestu un raksturiezīmju verifikācijas testu no kārtējās un iepriekšējām iterācijām izpildi. [4]

2. TESTĒŠANAS AUTOMATIZĀCIJA, PAMATOJUMS, SALĪDZINĀJUMS

Ja programmatūra tiek testēta manuāli, cilvēki ievada datus programmatūrā un veic darbības līdzīgi kā to darīs programmatūras lietotāji. Automatizētā testēšana nozīmē, ka testi tiek izpildīti automatizēti. Testēšana nodarbojas ar kļūdu meklēšanu, bet tas nav automatizētās testēšanas pamatuzdevums. Automatizētā testēšana mēģina novērstu kļūdu rašanos. Uzrakstīt stabilu testa skriptu ir sarežģīti un to uzturēt ir vēl grūtāk. Skatoties no projekta skatupunkta testu skripti nav obligāti, jo klients par to nemaksā. Kad testi kļūst sarežģīti vai dārgi uzturēšanā, rodas vēlme atteikties no testēšanas. Testu izveidošanas un uzturēšanas izmaksas ir viens no lielākajiem izaicinājumiem automatizētajā testēšanā, īpaši, ja projektā ir ļoti daudz testu. [13]

Kā augsta līmeņa automatizētās testēšanas mērķus var nosaukt, kvalitātes uzlabošanos, testējamās sistēmas izpratnes uzlabošanos, risku samazināšanos, vieglu testu izpildi, rakstīšanu un uzturēšanu, pēc iespējas mazāku testu uzturēšanu sistēmai turpinot attīstīties. [13]

Jau 1966. gadā SAE International, kas mūsdienās ir starptautiska apvienība vairāk nekā 128 000 kosmosa, automobiļu un komerctransporta inženieriem un tehniskajiem ekspertiem, savā nacionālajā kosmosa inženierijas un ražošanas sanāksmē runā par tēmu manuālā testēšana pret automatizēto testēšanu. Rakstā tiek atzīts, ka testēšanas automatizācija ir ieguldījumu vērta. Tiek aprakstīts, ka tieši kosmosa programmatūra ir attīstījusi tehnikas, kas ļauj automatizēt sistēmas, kas iepriekš nebija savietojamas ar automatizāciju sarežģītības vai izmaksu dēļ. [9]

Automatizētā testēšana izstrādes projektos tiek izmantota jau vairākas dekādes. Testēšana primāri tika veikta sistēmas testu kontekstā, lietojot speciālu datortehniku un programmatūru, kas bija programmēta ar testu skriptiem. Šāda veida testēšanas infrastruktūra nebija piemērota vienībtestēšanai, kā arī nebija plaši pieejama plašās datortehnikas iesaistīšanas dēļ. Pēdējo desmitgažu laikā ir kļuvuši pieejami vairāk vispārēju testu automatizācijas rīki, kas lietotnes testē caur to lietotāja saskarni. Daži no šiem rīkiem izmanto skriptu valodas, lai definētu testus, pievilcīgāki rīki izmanto robota lietotājus. Ne vienmēr esošie rīki apmierina testētājus un testēšanas pārvaldniekus augsto uzturēšanas izmaksu dēļ, ko izraisa trauslo testu problēma.[13]

Automatizēti testi bieži ir neveiksmīgi vienkāršu iemeslu dēļ. Ir svarīgi saprast šādas automatizācijas ierobežojumus, lai izvairītos no problēmām, kas bieži tiek asociētas ar uzvedības, saskarnes, datu un konteksta izmaiņām. Ja tiek mainīta sistēmas uzvedība, piemēram, ja prasības tiek mainītas un sistēma tiek pielāgota jaunajām prasībām, jebkādi testi, kas pārbauda mainīto funkcionalitāti visticamāk būs neveiksmīgi. Šis attiecas uz testēšanu neatkarīgi kāda testu

automatizēšanas pieeja tiek izmantota. Uzvedības izmaiņas var radīt vēl lielāku ietekmi uz testēšanas procesu, ja izmanītā funkcionalitāte tiek izmantota, lai sāktu citus testus. Testējot biznesa loģiku, kas iekļauta sistēmā, izmantojot lietotāja saskarni, var rasties problēmas. Jebkādas izmaiņas saskarnē var likt testam beigties ar kļūdu, piemēram, izvēļņu nosaukumu izmaiņas. Visiem testiem tiek pieņemts sākuma stāvoklis. Pārsvarā tas tiek definēts, kā nosacījumi datiem, kas jau ir sistēmā. Ja dati tiek mainīti, tad tests var beigties ar kļūdu, ja tests ir bijis atkarīgs no mainītajiem datiem. Sistēmas uzvedību var ietekmēt lietu, kas atrodas ārpus sistēmas, stāvoklis. Ārējie faktori iekļauj iekārtu stāvokli (printeri, serveri), citas lietotnes vai sistēmas pulkstenis. Testi, kas ir saistīti ar šo kontekstu nebūs stabili, ja vien netiks pārņemta kontrole pār kontekstu. [13]

Spējās izstrādes modeļos vairumā komandu notiek testu automatizēšana visos testēšanas līmeņos. Tas nozīmē, ka testētāji tērē laiku veidojot, izpildot, pārraugot, un uzturot automatizētos testus un to rezultātus. Kamēr programmētāji koncentrējas uz vienībtestu veidošanu, testētājiem vajadzētu koncentrēties uz automatizētu integrācijas, sistēmas un sistēmu integrācijas testu veidošanu. Šāda veida nepieciešamība noved pie tā ka spējās izstrādes komandās ir nepieciešamības pēc testētājiem ar spēcīgām tehniskām un testu automatizācijas zināšanām. Programmētāji visbiežāk veido automatizētus vienībtestus. Šie testi var tikt veidoti pēc koda izstrādes, bet dažkārt programmētāji veido testus inkrementāli pirms katras rakstāmā koda porcijas, lai radītu veidu kā pārbaudīt, ka uzrakstītais kods strādā kā paredzēts. Šāda veida pieeja tiek saukta par testu virzītu izstrādi, bet realitātē testu forma vairāk atbilst izpildāma zema līmeņa projektējumam. [4]

Spējās izstrādes modelī tiek automatizēti vienībtesti un funkcionālie testi. Tā kā dažkārt funkcionālo testu veikšana aizņem vairāk laika, tad tie tiek izpildīti retāk nekā vienībtesti. Piemēram, vienībtesti var tikti izpildīti ikreiz pēc koda izmaiņām, bet funkcionālie testi ik pēc dažām dienām. Viens no automatizētās testēšanas mērķiem ir apstiprināt, ka būvējums ir funkcionējošs un instalējams. Ja kāds no testiem ir neveiksmīgs nepieciešams veikt labojumus līdz nākamajiem testiem. Tas pieprasa ieguldījums reālā laika testa rezultātu ziņošanā. Šāda veida ieguldījumi samazina izmaksas , ko rada neefektīvi cikli "būvējums – instalācija – kļūda – būvējums - instalācija", tā kā izmaiņas, kas ir salauzušas būvējumu tiek atrastas ātri. [4]

2.1 Regresa testēšana

Programmatūras produkts netiek pabeigts ar pirmo izlaisto versiju. Tā vietā turpinās nepārtraukta izstrāde. Tiek veikti kļūdu labojumi, ārējās sistēmas var mainīties, kas rada

nepieciešamību pēc izmaiņām sistēma, kā arī klienta vajadzības var mainīties, kas rada izmaiņas esošajās prasībās vai jaunas prasības. [14]

Gadījumā, ja esošā programmatūra tiek izmainīta, laboti defekti vai pievienotas jauns daļas, to nepieciešams pārtestēt. Programmatūras izmaiņām mēdz būt blakus efekti, tāpēc nepieciešams atkārtot esošos testus. [14] Regresa testēšana pārtestē agrāk testētus programmatūras segmentus vai visu sistēmu, lai pārliecinātos, ka tie vēl funkcionē atbilstoši specifikācijai pēc izmaiņu ieviešanas citā lietotnes daļā, piemēram, pieliekot klāt jaunu funkcionalitāti vai labojot problēmas jau esošajā. Regresa testēšana noder esošu lietotņu testēšanai, jo palīdz noķert kļūdas, kas ir ieviesušās netīši. [2; 13] Regresa testēšanu nepieciešams veikt visos līmeņos. Testu scenāriji, ko izmanot regresa testēšanā tiek izpildīti vairākas reizes, tāpēc tiek jābūt labi dokumentētiem un atkārtoti izmantojamiem. [14]

Regresa testēšanas mērķi ir noteikt vai sistēmas dokumentācija arvien ir aktuāla, ka sistēmas testu dati un nosacījumi arvien ir aktuāli, ka agrāk testētās sistēmas daļas funkcionē pareizi pēc izmaiņu ieviešanas. Veicot regresa testēšanu manuāli, tas var aizņemt daudz laika, būt nogurdinošs darbiniekam, kas var palielināt testētāju pieļauto kļūdu iespējamību. [2]

Regresa testēšanu pielieto, kad ir liels risks, ka jaunās izmaiņas var ietekmēt sistēmas nemainītās daļas. Izstrādes laikā regresa testēšanu var veikt pēc noteikta izmaiņu daudzuma veikšanas. Ja regresa testēšanas process ir automatizēts, tad to iespējams veikt pirms katras jaunas sistēmas versijas vai tās papildinājumu/labojumu likšanas ražošanā. [2]

Plānojot regresa testēšanu nepieciešams saprast cik plašai tai jābūt. Viena no iespējām ir defektu pārtestēšana, kad tiek izpildīti tie testi, kas ir atklājuši kļūmes. Izmainītās funkcionalitātes testēšana izpilda testus par programmas daļām, kas ir izmainītas vai labotas. Jaunās funkcionalitātes testēšana izpilda testus visām programmas daļām vai elementiem, kas ir jauni integrēti. Pilns regresa tests nozīmē visas sistēmas testēšanu. Tikai defektu pārtestēšana vai tikai izmainīto daļu pārtestēšana ir nepietiekama. Vienkāršas sistēmas koda izmaiņas var veidot blakusefektus jebkurā patvaļīgā sistēmas daļā. Testējot tikai izmaiņas netiek pārbaudīts kādas sekas izmaiņas ir radījušas neizmainītajā sistēmas daļā. Šajā gadījumā problēmas rada programmatūras sarežģītība. Ar saprātīgām izmaksām ir iespējams tikai aptuveni noteikt iespējamās izmaiņu ietekmes. It sevišķi grūti tas ir sistēmās ar nepietiekamu dokumentāciju un trūkstošām prasībām. Pārsvarā šādas problēmas rodas ar vecām sistēmām. [14]

Spējās izstrādes modelī katras iterācijas laikā paralēli jaunās funkcionalitātes testiem notiek regresa testēšana, kas iekļauj atkārtotu automatizētu vienībtestu un raksturiezīmju verifikācijas testu

izpildi no tekošās un iepriekšējām iterācijām. Izmaiņu notikšana visa projekta laikā ir viens no pamata principiem spējajā izstrādē. Esošo funkciju izmaiņas ietekmē testēšanu, it sevišķi jūtamas sekas ir regresa testēšanā. Testu automatizācijas izmantošana ir viens no veidiem kā pārvaldīt testēšanu, kas nepieciešama saistībā ar izmaiņām. Tomēr ir svarīgi, ka izmaiņu daudzums nepārsniedz projekta komandas spēju tikt galā ar riskiem, kas saistīti ar šīm izmaiņām. [4]

2.2 Testu virzīta izstrāde

Testu virzīta izstrāde vairāk koncentrējas uz programmatūras, kas vēl ir jāuzraksta, darbībām. Testu veidošana pirms izstrādes nodrošina testējamās programmatūras izpratni pirms to sāk veidot. Ja izveidotie testi atspoguļo kā programmatūra tiks izmantota, tiek nodrošināts, ka tiek būvēta pareizā programmatūra. Šāda testu veidošana apskatot dažādus scenārijus detalizētā veidā palīdz identificēt sadaļas, kur prasības ir neskaidras vai pretrunīgas. Šāda analīze palīdz uzlabot prasību kvalitāti, līdz ar to uz labojot programmatūras, ko tās apraksta, kvalitāti. [13]

Kad testu rakstīšana ir pabeigta un testi pēc pārbaudes ir kļūdaini, jo jaunā funkcionalitāte vēl nav uzprogrammēta, var pārslēgties uz izstrādi, lai panāktu, ka testu rezultāti ir pozitīvi. Testi var tikt izmantoti arī kā progresa mērījumi. Arvien papildinot kodu, tiek veiksmīti izpildīti vairāk testi. Strādājot tiek tupināts izpildīt iepriekš izveidotos testus, kā regresa testus, lai pārliecinātos, ka izmaiņas nav radījušas nevēlamus blakus efektus. Šādas automatizētās testēšanas pieeja ļauj atzīmēt jauno funkcionalitāti un neļauj tai tikt netīšām izmainītai.[13;4] Williams, Macimilien un Vouk savā publikācijā secina, ka kods, kas izstrādāts izmantojot testu virzītu izstrādi, funkcionālajos testos un regresa testēšanā uzrāda ievērojami mazāk defektus nekā līdzīgs produkts izmantojot tradicionālāku izstrādi. [15]

3. VIENĪBTESTI

Programmētājiem veicot testēšanu atgriezeniskā saite par programmatūras kvalitāti tiek iegūta daudz ātrāk. Programmētāji, kas apgalvo, ka var uzrakstīt kodu, kas strādā ar pirmo reizi un turpinās strādāt vienmēr, ir sastopami reti. Lai nodrošinātu, ka programmatūra strādā jau ar pirmo reizi arī programmētāji veic testēšanu, lai pārbaudīt, ka programmatūra strādā, kā tas bija paredzēts rakstot kodu. Daži programmētāji rīkojas līdzīgi kā testētāji un pārbauda sistēmu kopumā, bet pārsvarā tie dod priekšroku pārbaudīt sistēmu vienību pēc vienības. Vienības var būt lielākas sastāvdaļas vai individuālas klases, metodes vai funkcijas. Galvenā atšķirība, kas atšķir šāda veida testus no testētāju veidotajiem testiem, ir tā, ka vienības tiek testētas sekojot programmatūras projektējumam nevis prasību interpretācijai. [13]

Pēc V-modeļa testēšanas līmeņiem šādu testēšanu iedala komponentu testēšanas līmenī. Šajā līmenī galvenā atšķirība ir komponenšu nošķirtība no pārējās sistēmas. Izolācija ir nepieciešama, lai izvairītos no ārējās ietekmes. Ja testa rezultāts parāda problēmu, tās cēlonis noteikti ir komponentē, kura tiek testēta. [14]

Šāda veida testēšanu, kad tiek pārbaudītas nelielas komponentes jau izstrādes laikā sauc par vienībtestēšanu. To parasti veic programmētāji jeb izstrādātāji. Šādā veidā pārbaudot iespējams noteikt, ka funkcija strādā kā tas ir gaidāms. Tiek pārbaudīts, ka pie noteiktiem ievades datiem funkcija atgriež vēlamo rezultātu un spēs apstrādāt gadījumus, kad ievades dati ir nekorekti. Gala rezultātā vienībtestēšana dod iespēju atrast kļūdas programmatūras algoritmos vai loģikā, lai uzlabotu koda kvalitāti. Vienībtestēšanā atrastās programmatūras kļūdas parasti nereģistrē problēmu pārvaldības sistēmās. Parasti izstrādātāji tās operatīvi novērš, turpinot komponentes izstrādi. [2]

Rakstot arvien vairāk un vairāk testu tiek iegūti testu komplekti, ko iespējams izpildīt jebkurā laikā izstrādes procesā, lai pārbaudītu koda kvalitāti. Šādā veidā būs iespējams izvairīties no salauztas funkcionalitātes citu izmaiņu dēļ. Esošajiem testiem dodot neveiksmīgu rezultātu būs skaidrs, kam nepieciešams pievēstu uzmanību.

Protams šādu testu veidošana jau agri izstrādes posmā un to uzturēšana ir saistīta ar izmaksām, ko veido ieguldītais laiks, bet projektam augot iespējams testus, kas jau ir izveidoti, izpildīt un pārliecināties, ka esošā funkcionalitāte nav salauzta pievienojot jaunu funkcionalitāti.

Spējās izstrādes modelī tiek izmantoti automatizēti rīki gan programmēšanai, gan testēšanai, kā arī programmatūras izstrādes pārvaldīšanai. Programmētāji nepārtraukti atzīmē kodu un vienībtestus konfigurācijas pārvaldības rīkā, lietojot automatizētus būvējumu un testēšanas

satvarus. Šie satvari palīdz nepārtrauktai integrācijai ar jaunu programmatūru, ļaujot vienībtestus izpildīt atkārtoti pēc jauna koda pievienošanas. [4]

Ja programmētājs pats testē savu darbu, to nepieciešams darīt ļoti kritiski. Pārsvarā cilvēki nevar paskatīties uz sevis radīto produktu pietiekami no malas. Cilvēkiem nav raksturīga vēlme pierādīt savas kļūdas, drīzāk tie vēlas pierādīt, ka viņu veidotais kods darbojas labi. Testējot sevis radītu programmatūru cilvēki mēdz būt pārāk optimistiski. Pastāv iespēja aizmirst loģiskus testa scenārijus, jo darbinieks vairāk ir ieinteresēts programmēšanā nevis testēšanā, tāpēc testē paviršī. Ja programmētājs ir pieļāvis būtisku kļūdu dizainā, piemēram, pārpratis formulējumu, tad visticamāk ar viņa paša veidotiem testiem to atrast neizdosies. Šāda veida scenārijs neienāks prātā. Lai apietu šāda veida problēmas iespējams strādāt pāros, kad programmētāja darbu testē cits programmētājs, bet jārēķinās, ka iepazīšanās ar testēšanas objektu aizņem papildus laiku, kas nebūtu nepieciešams pašam testējot. [14]

Komponenšu testēšanas galvenais uzdevums ir pārbaudīt, ka konkrētais testa objekts izpilda visu funkcionalitāti, kas paredzēta specifikācijā. Šajā gadījumā funkcionalitāte nozīmē testa objekta ievadizvades uzvedību. Tiek veidoti vairāki testēšanas scenāriji, kas pārbauda noteiktas ievadizvades kombinācijas. Tipiskās kļūdas, kas tiek atrastas funkcionālās komponenšu testēšanas laikā ir nepareizi aprēķini, trūkstoši vai nepareizi izvēlēti ceļi. [14]

Vēlāk testējot jau visu sistēmu kopumā nevis atsevišķu vienību iespējams testētā vienība tiks izsaukta neatbilstoši specifikācijai. Šādā gadījumā vienībai nevajadzētu apturēt savu darbību vai izraisīt visas sistēmas avāriju. Vienībai šādā situācijā jāvar rīkoties saprātīgi un stabili. Tāpēc stabilitātes testēšana ir vēl viena svarīga komponenšu testēšanas sastāvdaļa. Šī veida testēšana ir līdzīga funkcionālajai testēšanai, bet vienības izsaukšana vai testu dati ir vai nu nepareizi vai īpaši gadījumi, kas nav minēti specifikācijā. Šādus testus sauc arī par negatīvajiem testiem. Vienības reakcijai būtu jābūt piemērotai izņēmumu apstrādei. Ja šādi gadījumi netiek apstrādāti var rasties kļūdas, piemēram, dalīšana ar nulli, kas var izraisīt sistēmas darbības apstāšanos. [14]

Komponentes īpašības, kam ir liela ietekme un tās nav iespējams notestēt augstākos testēšanas līmeņos vai tas ir dārgāk, nepieciešams pārbaudīt komponenšu testēšanas laikā, piemēram, efektivitāte un uzturamība. Efektivitāte nosaka cik efektīvi komponente izmanto datora resursus. Tiek izvērtēti dažādi aspekti – atmiņas izmantošana, skaitļošanas laiks, diska vai tīkla piekļuves laiks un vienības funkciju un algoritma izpildes laiks. Šos nefunkcionālos testēšanas laikā iespējams precīzi izmērīt, piemēram, atmiņas izmantošanu kilobaitos, atbildes laiku milisekundēs. Efektivitātes testi reti tiek veikti visai sistēmai, tā ir jāpārbauda tikai sistēmas daļas, kur tā ir kritiska

vai ja efektivitātes prasības ir iekļautas prasībās. Šāda testēšana parasti tiek veikta sistēmām, kurām ir ierobežoti datortehnikas resursi vai reāllaika sistēmās, kad pastāv laika ierobežojumi. [14]

Uzturamība apzīmē visas programmatūras īpašības, kam ir ietekme uz to cik viegli vai grūti ir izmainīt programmatūru vai turpināt to izstrādāt. To apraksta cik lielas pūles programmētājam ir jāpieliek, lai saprastu programmu un tās kontekstu. Šeit jārunā gan par programmētāju, kas ir veidojis programmu un to nepieciešams darīt pēc vairākiem mēnešiem vai gadiem, gan programmētāju, kas pārņem pienākumus par kodu, ko uzrakstījis cits. Koda uzturamībai svarīgi ir koda struktūra, modularitāte, koda komentāru kvalitāte, standartu ievērošana, saprotamība un dokumentācijas vērtība. Šāda veida īpašības nevar notestēt dinamiskā testēšana. Šeit nepieciešams izmantot statisko testēšanu un apskates. Tā kā vienībtestēšanas laikā tiek apskatīta atsevišķa komponente ir vērtīgi iekļaut šāda veida analīzi. [14] Pieejot izstrādei no vienībtestēšanas perspektīvas visticamāk tiks rakstīts kods, kas ir viegli testējams. Tas nozīmē, ka tiks veidotas vairāk mazas un koncentrētākas funkcijas, kas nodrošina vienu darbību, nevis lielas funkcijas, kas nodrošina vairākas dažādas darbības, tādā veidā uzlabojot uzturamību.

Testētājam, kas veic vienībtestēšanu, parasti ir pieeja pirmkodam, kas padara to par baltās kastes testēšanas sfēru. Testus iespējams veidot izmantojot zināšanas par vienības struktūru, funkcijām un mainīgajiem. Pieeja kodam var būt izdevīga arī testu izpildei. Ar speciālu rīku palīdzību ir iespējams sekot līdz programmatūras mainīgajiem testa izpildes laikā. Tas palīdz pārbaudot pareizu vai nepareizu vienības darbību. Vienības stāvoklis var ne tikai tik novērots, bet arī ietekmēts, kas ir svarīgi veicot stabilitātes testēšanu, jo testētājs var izraisīt speciālus izņēmumu gadījumus. [14]

Plānojot vienībtestēšanu jāņem vērā, ka reālas programmatūras sistēmas sastāv no simtiem un tūkstošiem vienkāršu komponenšu. Analizēt kodu, lai izveidotu testa scenārijus ir iespējams tikai dažām izvēlētām komponentēm. Vēlāk vienkāršās komponentes tiks apvienotas lielākās vienībās, kuras iespējams izmantot vienībtestēšanai. Iespējams šīs vienības jau ir pārāk lielas, lai veiktu novērojumus un iejaukšanos koda līmenī ar pienācīgu iedarbību. Plānojot vienībtestēšanu nepieciešams saprast vai testēt vienkāršās daļas vai tikai lielākās vienības. Iespējams testu veidošanu atvieglotu testu virzīta izstrāde, kas testu scenāriji tiek sagatavoti un automatizēti pirms programmēšanas. Koda daļas tiek testētas un uzlabotas līdz visi testi ir veiksmīgi. [14]

4. LIETOTĀJA SASKARNES TESTI

Tradicionālā testēšanas definīcija ir cēlusies no kvalitātes nodrošināšanas. Programmatūra tiek testēta, jo tajā ir jābūt defektiem. Tātad testēšana tiek veikta, līdz nav iespējams pierādīt, ka programmatūrā vēl arvien ir defekti. Tradicionāli testēšana notiek pēc programmatūras pabeigšanas. Tā rezultātā testēšana ir kvalitātes mērīšanas rīks nevis kvalitātes nodrošināšanas veids. Daudzos uzņēmumos testēšanu nodrošina kāds cits, nevis programmētājs. Šādas testēšanas nodrošinātā atgriezeniskā saite ir vērtīga, bet tiek iegūta pārāk vēlu programmatūras izstrādes ciklā, tāpēc tās vērtība samazinās. Šāda veida testēšanai ir iespaids uz termiņu pagarināšanu, jo atrastās problēmas tiek nodotas izstrādei atkārtotam darbam, un pēc labojumiem nepieciešama atkārtota testēšana. [13]

Testētājs, kas ir neatkarīgs no izstrādes, var palielināt testēšanas kvalitāti un vispusību, jo viņš testa objektu apskata bez aizspriedumiem. Tas nav viņa produkts, kā varētu uzskatīt programmētājs. Programmētāja radītie pieņēmumi un iespējamie pārpratumi ne vienmēr sakritīs ar testētāja pieņēmumiem un pārpratumiem. Tā kā testētāja pienākums ir ziņot par kļūmēm, tad tam var būt negatīva ietekme uz attiecībām starp testētājiem un programmētājiem. Meklēt un pierādīt citu cilvēku kļūdas var būt problemātiski. Bieži problēmas sagādā tas, ka testētāju atrastās kļūmes nav iespējams atkārtot programmētāju vidēs. Nepieciešams fiksēt detalizētu kļūmes un testa vides aprakstu, lai atklātu atšķirības vidēs, kas var izraisīt dažādo uzvedību. [14]

Lietotāja saskarnes testēšana atbilst V-modeļa integrāciju testēšanai. Pat ja ir notikusi pilnīga komponenšu jeb vienībtestēšana, var rasties saskarnes izraisītas problēmas, piemēram, lietotāja saskarne nav nodevusi visu informāciju par atzīmētajiem laukiem. Integrāciju testēšanas uzdevums ir atrast sadarbības un sadarbspējas problēmas. Integrāciju testēšana pārbaudīs arī ārējo sistēmu darbību, ko ir izstrādājuši citi uzņēmumi, piemēram, datubāze. [14] Šis ir viens no iemesliem kāpēc vienībtestēšana nevar tikt aizstāta ar lietotāja saskarnes testēšanu. Abi testēšanas veidi pārbauda testējamo sistēmu dažādos līmeņos.

Integrāciju testēšanas mērķis ir atrast saskarnes un sadarbības problēmas, kā arī konfliktus starp integrētajām vienībām. Problēmas var rasties jau savienojot tikai divas komponentes. To savienošana var nestrādāt, jo to saskarnes nav savietojamas, trūkstošu failu dēļ, vai komponentes ir sadalītas neatbilstoši projektējumam. Problēmas, kas rodas programmatūras daļu savienojumu izpildes dēļ, ir grūtāk atrodamas. Tās ir komunikācijas vai datu apmaiņas kļūdas, piemēram, komponente nodod sintaktiski nepareizus vai nekādus datus, saņēmēja komponente nestrādā, komponente interpretē saņemtos datus savādāk, dati tiek nosūtīti pareizi, bet nepareizā laikā vai

pārāk bieži. [14] Šāda veida kļūmes nevarētu atrast vienībtestēšana, jo tās izraisa divu vienību savstarpējā sadarbība.

Veicot tikai integrāciju testēšanu iespējami šādi trūkumi. Lielākā daļa kļūmju, kas notiks testa izpildē, būs saistīti ar funkcionāliem defektiem atsevišķās komponentēs. Šādā gadījumā tiek veikts netieši izteikts komponenšu tests, bet vidē, kas nav piemērota piekļuvei individuālai komponentei. Tā kā katra komponente netiek skatīta atsevišķi ir kļūmes, kuras netiks izraisītas un līdz ar to netiks atrastas. Ja testa laikā notiek kļūme var būt sarežģīti vai pat neiespējami atrast tās izcelsmi un nošķirt cēloni. [14] Vienībtestu paralēla neveikšana rada papildus slodzi uz integrāciju testiem un tiek noķerti mazāk defekti, vairāk laika aizņem rezultātu analīze un tiek patērēts vairāk papildus laika, nekā veicot arī vienībtestēšanu.

5. PROBLĒMA UN IESPĒJAMIE RISINĀJUMI

Klienti sistēmas ieguvumus visvairāk izjūt, kad kāda pietiekami sarežģīta darbība nav jāveic viņiem manuāli, bet to var izdarīt sistēma. Viņi šādos gadījumus arī gaida, ka sistēma šīs darbības veiks bez kļūdām. Neuzticamas sistēmas gadījumā klientam var rasties vēlme paralēli veikt manuālas darbības un pārliecināties, ka sistēma darbojas korekti. Ar katru nākamo kļūmi klienta uzticībai samazinoties palielinās klienta manuālais darbs un samazinās sistēmas priekšrocības.

Dažkārt sistēmās tiek izstrādāti algoritmi, kuru precīza darbība zināma tikai programmētājam. Šie algoritmi parasti veic analīzi izmantojot sistēmas datus. Sistēmas lietotājam tiek attēlots tikai rezultāts un datu atlases un pārbaužu daļa paliek neredzama. Algoritmi tiek aprakstīti arī klientu prasībās no biznesa viedokļa, kā arī testēšanas scenārijos ar konkrētiem datu piemēriem.

Šādos gadījumos lietotāji sistēmas saskarnē norāda tikai konfigurējamos parametrus un izsauc analīzi vai aprēķinu, kas fonā veic komplicētas darbības, kas ir iestrādātas sistēmā vai konfigurētas sistēmas sākuma konfigurācijā. Ja klienta konfigurācijā ir iespējams aprakstīt aprēķinu algoritmus, tas sarežģa testēšanas darbu, jo visas klienta iespējamās konfigurācijas nav iespējams notestēt. Šādu algoritmu gadījumā ļoti svarīgi ir arī kvalitatīva regresa testēšana pēc kļūdas labojuma vai izmaiņu veikšanas. Sistēmas, kuru lieto daudz klienti gadījumā var būt situācija, ka jaunam klientam uzsākot lietošanu rodas jaunas prasības un sistēmu nepieciešams uzlabot. Šādā gadījumā pārējie klienti sagaida, ka viņu prasības turpinās strādāt bez sarežģījumiem.

Iespējams šādu algoritmu testēšanai primāri jāizmanto vienībtestēšana, lai lietotāja saskarnes testiem būtu jāvar atrast tikai kļūdas, kas saistītas lietotāja saskarnes darbību vai datu atlasīšanu. Apskatot šādu iespējamo risinājumu jāanalizē kādi ieguldījumi nepieciešami, lai varētu izveidot vienībtestus šādiem algoritmiem. Ja šāds algoritms ir iestrādāts sistēmā, kura neatbalsta vienībtestēšanu vai to izveidošana un uzturēšana ir sarežģīta un prasa neatbilstošus resursus, nepieciešams testēšanu balstīt uz lietotāja saskarnes testiem.

Gadījumos, kad tests ir neveiksmīgs vai sistēmas lietotājs ir saskāries ar nepareizu rezultātu klients un piesaka kļūmi, nepieciešams izprast atsevišķo konfigurāciju un saprast vai tiešām sistēmā ir defekts vai klients ir kļūdījies datu ievadīšanā vai savā analīzē. Pie pietiekami sarežģītiem aprēķiniem problēmas cēloņa atrašana aizņem daudz laika un resursu. Izvirzītās problēmas risinājuma mērķis ir uzlabot testēšanu sarežģītiem, lietotājam neredzamiem aprēķiniem.

Lai izprastu algoritma darbību un uzzinātu iespējamos problēmas cēloņus, lietotājam nepieciešams zināt kādas darbības algoritms veic, kādus datus analizē, kā arī kādu salīdzināšanu

veic. Ja lietotājam pieejama algoritma darbības informācija, iespējams veikt manuālu analīzi, ja sistēma piedāvā atlasīt nepieciešamos datus lietotāja saskarnē. Algoritma darbības informāciju lietotājam var sniegt prasību dokuments. Klasiski sistēmas prasību dokumenti tiek veidoti teksta veidā, bet analītisku algoritmu aprakstīšanai iespējams izmantot shematiskas prasības, kas padara informāciju klientam uztveramāku un izprotamāku. Problēmas sistēmā var izraisīt kaskādes veida algoritmi kā arī ierobežojumi, kas tos ietekmē. Izprotot dažādu datu savstarpējās attiecības sarežģītā sistēmā, būs skaidrāk saprotami tās darbībās principi. Prasību vizualizācija dod iespēju izprast sarežģītus analīzes algoritmus, jo lietotājam ir iespēja izsekot sistēmas darbībai. [7] Pētījumus par vizualizācijas (diagrammu) ieteikmi uz kognīciju uzsāka Larkin un Simon [6] . Kā galvenos diagrammu ieguvumus viņi apraksta informācijas tuvumu, indeksāciju pēc atrašanās vietas un uzmanības nobīdes samazināšanos. [12] Castner un Larkin ir ierosinājuši, ka laba reprezentācija samazina kognitīvās apstrādes slodzi divos veidos. Atļaujot lietotājam aizvietot mazāk pūles pieprasošos vizuālos operatorus ar vairāk pūles pieprasošiem loģiskiem operatoriem. Tiek samazināts uzdevuma izpildei nepieciešamās informācijas meklēšanas laiks. [5]

Prasību dokumenti tipiski iekļauj nelielu dabīgās valodas aprakstu katrai prasībai. Papildus izpratnei dokumentam var tikt pievienoti prasību grupu attēlojums strukturētā diagrammā, lai veicinātu kopīgu izpratni par pilnu prasību komplektu. Šeit gan noteikti jāatzīmē, ka šādai vizualizācijai ir jābūt aktuālai un izsekojamām tās izmaiņām. Dažkārt prasības var tikt aprakstītas dabiskās valodas aprakstos, kas ir izkaisīti pa vairākiem dokumentiem, kas apraksta dažādu ieinteresēto pušu skatupunktus. Šādu prasības ir subjektīvi interpretējamas un var ietekmēt sistēmas lietotāju spēju pieņemt objektīvus lēmumus par sistēmas darbību. Šīs problēmas iespējams risināt ar vizuālu analīzi. Dabiskā valodā rakstītas prasības tiek pakļautas subjektīvas interpretācijas iespējai. Vizuāls skaidrojums var atbalstīt sistēmas lietotājus, lai noteiktu tās darbības scenārijus. [8; 7]

Datu vizualizācija pārveido tos vieglākai uztverei cilvēka redzes maņai, algoritmu vizualizācija pārstrukturē skaitļu un simbolu datus vizuāli uztveramās formās. Prasību vizuālizācijas mērķis ir nodrošināt dziļāku informācijas izpratni. Prasību veidošana ir process programmatūras izstrādē, kas pieprasa intensīvu komunikāciju starp dažādām ieinteresētajām pusēm, lai apspriestu un vienotos par nepieciešamajām izmaiņām vai uzlabojumiem sistēmā. Šādā situācijā vizuāla reprezentācija ir vērtība komunikācijas procesā. Efektīva vizuālā reprezentācija nodrošina kognitīvo atbalstu, izceļoti svarīgākos mijiedarbības punktus un konkrētas lietošanas aspektus. Prasības darbojas kā saziņas līdzeklis starp klientiem, lietotājiem un izstrādātājiem.

Veidojot kopīgu izpratni par nepieciešamo funkcionalitāti ir viena no svarīgākajām lietām, lai tiktu atrastas neprecizitātes specifikācijā. Vizualizācija ir viens no veidiem kā palīdzēt lietotājiem sniegt ieskatu lielos un sarežģītos datu kopumos. Sarežģītu prasību vizualizācija var potenciāli samazināt kognitīvo slodzi izceļot svarīgos saskares punktus un aprakstītās sistēmas uzvedību. [6; 8]

Vizualizācijas process apskatot to vienkārši ir soļu secība, kas iekļauj apkopošanu, ilustratīvu attēlošanu, analīzi un datu interpretāciju. To sauc par vizualizācijas datu plūsmas modeli. Vizualizācijas galvenais uzdevums ir datu pārveidošana grafiskā reprezentācijā. [8]

Jones, Harrold un Stasko savā publikācijā apraksta testēšanas informācijas vizualizēšanu, lai palīdzētu lokalizēt kļūdas. Tā ir viena no visdārgākajām un laikietilpīgākajām atkļūdošanas sastāvdaļām. Lai atrastu kļūdas iemeslu programmētājiem nepieciešams identificēt koda daļas, kas tiek izmantotas testa izpildei un izvēlēties aizdomīgo, kas varētu ietvert kļūdas. Aprakstītā metode izmanto krāsas, lai vizuāli atzīmētu koda daļas, kas piedalās izpildītajā testu kopā, atdalot neveiksmīgos un izietos testus. Izmantojot vizuālu atzīmēšanu programmētājam ir iespēja apskatīt koda daļas, kas ir iesaistītas neveiksmīgo testu izpildē. Vizualizācijai tiek izmantoti testēšanas rezultātā iegūtie dati. Šādā veidā tiek noteikts testēšanas pārklājums, atrodot koda zonas, kuras netiek testētas. Koda daļas krāsu šajā metodē nosaka testu, kas izmanto šo koda daļu rezultāti. Ja visi testi būs beigušies veiksmīgi, tad daļu iekrāso zaļu, un ja visi testi beigušies neveiksmīgi, tad koda daļu iekrāso sarkanu. Gadījumiem, ar abu veidu testa rezultātiem tiek izmantotas gradienta krāsas. Šis risinājums palīdz izprast sistēmas darbību un kvalitāti programmētājiem un samazina sistēmas atkļūdošanas laiku, bet nepalīdz sistēmas lietotājiem, kam nav iespējas redzēt kodu un maģistra darbā apskatāmās problēmas risināšanai neatbilst. [10]

Salīdzinot vienībtestēšanu un lietotāja saskarnes testēšanu autors ieraudzīja atšķirību, ka programmētājam ir pieejams speciāls rīks ar kuru iespējams testēšanu laist pa soļiem un aplūkot starprezultātus. Gadījumā, kad lietotāja saskarnes tests beidzas negatīvi, ir grūti analizēt kļūmes cēloni, jo tas var būt nepareizi ievaddati, nepareizi atlasītie dati, problēmas integrācijā ar datubāzi vai problēma pašā algoritmā vai tā daļā. Ja ir redzams tikai gala rezultāts, nav iespējams noteikt precīzu cēloni. Šādā gadījumā testētājam, pirms nodot programmētājam automatizētā testa atrasto defektu, ir jāsaprot kāpēc tas ir radies, un vai tiešām programmētājam būs kļūda ko labot.

Iespējamais risinājums ir sistēmas lietotājiem radīt iespēju apskatīt sistēmas algoritma starprezultātus, kas dodu papildus iespēju tā darbības analīzē. Jāapzinās, ka starprezultātu piefiksēšana palēnina kopējo algoritma darbības laiku. Tā kā šāda iespēja nav nepieciešama visos sistēmas aprēķinos, bet tikai pārbaudot nekorektu rezultātu, tad nepieciešams to varēt ieslēgt un

izslēgt. Ja algoritma starprezultātiem iespējami dažādi detalizācijas līmeņi, tad iespējams veidot dažādas detalizācijas starprezultātus.

Bičevskis un Borzovs savā publikācijā 1982. gadā apraksta prioritāšu mehānismu starprezultātu iegūšanā. Publikācijas autori apraksta ideju sistēmai pievienot diagnostikas īpašības – iespēju iepriekš paredzētos punktos iegūt informāciju. Lai to nodrošinātu, jau veidojot algoritmu tiek paredzēta iespēja konkrētos punktos atzīmēt papildus izvada iespēju, ko iespējams izslēgt un ieslēgt pēc vajadzības. Aprakstītājā sistēmā izslēgšanas un izslēgšanas iespēju nodrošināja prioritātes. Speciāli programmētajām makrokomandām tika piešķirtas prioritātes un atkarībā no sistēmas izpildes prioritātes tika izpildītas zemākas kategorijas prioritātes. [16]

Šāda veida risinājumi ļautu samazināt laiku, kas tiek patērēts apstrādājot automatizētās testēšanas rezultātus vai nepareizus rezultātus, ko ikdienas situācijā ir atradis sistēmas lietotājs, un atļautu ātrāk nonākt pie problēmas cēloņa, tātad risinājuma. Lietotājam nepieciešams noteikt vai nepareizu rezultātu rada datu problēmas vai cēlonis ir kļūme programmatūrā. Metodēm, kas var samazināt laiku, kas nepieciešams, lai atrastu defektus, ir liela ietekme uz programmatūras izstrādes un uzturēšanas izmaksām un kvalitāti.

6. SISTĒMAS APRAKSTS

Horizon ir resursu vadības sistēma, kas nodrošina ievērojami plašākas iespējas nekā tikai grāmatvedības uzskaiti. Horizon pilda arī personāla vadības sistēmas, noliktavas uzskaites programmas, ražošanas vadības sistēmas un citu biznesa vadības sistēmu funkcijas.

Horizon ir resursu vadības sistēma, kura no pirmsākumiem ir ražota Latvijā. Pirmie resursu vadības sistēmas programmēšanas darbi tika uzsākti 1991. gadā. Tolaik sistēmu sauca Apvārsnis. 1993. gadā grāmatvedības sistēma Apvārsnis tika piedāvāta tirgū un ieguva pirmos klientus. Mainoties tehnoloģijām 1998. gadā Apvārnis no DOS vides pilnībā pārgāja uz Windows vidi. 2008. gadā sistēma piedzīvo zīmola maiņu un turpmāk tiek dēvēta par resursu vadības sistēmu Horizon. Kopš 2005. gada sistēmu ražo pēc kvalitātes vadības sistēmas atbilstoši starptautiskā standarta ISO 9001 prasībām. [3]

Horizon arhitektūra ir veidota trijos līmeņos: datubāzes, biznesa loģikas (lietojuma) un datu attēlošanas (lietotāja saskarne) līmenī, atsevišķi nodalot lietotāja saskarnes valodu. Risinājums ir neatkarīgs no izmantotās datubāzes platformas, atbalsta gan MS SQL, gan arī Oracle SQL datubāzi. Tā kā šīs datubāzes ir pieejamas vairākām operētājsistēmām, bet risinājuma biznesa loģika ir ietverta klienta darbstacijas programmatūrā, tad sistēmas servera programmatūru ir iespējams darbināt uz jebkuras platformas, kas atbalsta minētos datubāžu risinājumus. Klienta darbstacijas programmatūra darbojas uz MS Windows grupas operētājsistēmām. Biznesa loģikas līmenis ir realizēts klienta darbstacijas programmatūrā. Klienta darbstacijas programmatūra pieslēdzas serverim, izmantojot risinājumu SQL direct. Savukārt sadarbība ar citām datu pārraides tīklos esošām sistēmām tiek nodrošināta, izmantojot tehnoloģiju .net vai j2ee. Visi biznesa loģikas elementi — pavadzīmes, konti, kodi — ir izveidoti kā sistēmas objekti, kas var apvienoties, pārmantot citu objektu īpašības. Sistēmā ir izveidots pamats, uz kura īsā laikā iespējams izveidot jaunu funkcionalitāti. [3]

Šobrīd uzņēmumā vienībtestēšana notiek minimāli un tiek automatizēti lietotāja saskarnes testi. Tas kas nav automatizēts tiek veikts manuālos regresa testos, bet retāk un ne vienmēr pilnā apjomā. Ja uzņēmuma iekšējā testēšana nav noķērusi defektu, jo nav veikti testi vai testu segums nav bijis atbilstošs konkrētā klienta konfigurācijai, defekts nonāk klientam publicētā versijā un to var atklāt klients. Ja klients izstrādātājam piesaka kļūdu, nepieciešams veikt līdzīgu analīzi kā neveiksmīga regresa testa rezultātā. Šeit jāņem vērā konkrētā klienta dati, konfigurācija un vide.

Sistēmas testēšanu nodrošina izstrādes nodaļas testētāji. Izstrādes nodaļa ir sadalīta komandās pēc funkcionalitātēm un to sastāvs nodrošina izstrādes ciklu no analīzes līdz testēšanai. Komandas strādā izmantojot spējās izstrādes metodoloģiju. Sistēmas testēšanu var iedalīt 3 grupās – jaunu izstrāžu testēšana, kļūdu labojumu testēšana un regresa testēšana.

Jaunas izstrādes tiek veidotas pēc klienta pieprasījuma vai atbilstoši sistēmas attīstības rīcības plānam. Jaunās izstrādes pēc to programmēšanas testē atbilstoši produkta īpašnieka izveidotajām prasībām vai lietotājstāstiem. Pēc jaunas izstrādes testētājam nepieciešams arī papildināt regresa testu apjomu nākamajai versijai ar šīs izstrādes scenārijiem. Ja tiek pieņemts lēmums šai funkcionalitātei nodrošināt automatizētu regresa testēšanu, tad tiek veidoti testpiemēri un automatizēšanas rīkā tiek aprakstīti un pievienoti automatizēto testu kopai.

Sistēmas kļūdu labojumus pārbauda testētājs. Sistēmas izstrādē tiek izmantota laidiena procedūra. Ja kļūdu nepieciešams izlabot, bet tā nav klientam jāpiegādā kādā no publicētajām versijām, tad testētājs to notestē versijā, kurā šobrīd notiek izstrāde, bet, ja labojumam nepieciešama piegāde ar laidienu klientam publicētā versijā, tad testētājs to testē gan izstrādes aktuālajā versijā, gan piegādes versijā pēc kļūdas labojuma ienešanas.

Četras reizes gadā tiek publicēta jauna sistēmas versija, kurā iekļauj jaunās izstrādes un zemas kategorijas kļūdu labojumus. Pirms versijas publicēšanas nepieciešams veikt regresa testēšanu, lai pārliecinātos, ka visa esošā funkcionalitāte vēl arvien darbojas. Pamatā regresa testēšanu veic testētāji. Regresa testēšanu sistēmai nodrošina ar manuālu testēšanu, automatizētu testēšanu izmantojot lietotāja saskarni un vienībtestēšanu. Izstādes nodaļa tiecas uz to, lai tiktu automatizēta pēc iespējas lielāka kopa svarīgu testpiemēru. Automatizētie testi tiek izpildīti ne tikai pirms versijas publicēšanas, bet arī pirms jaunu laidienu publicēšanas.

Horizon ietvars sākotnēji netika veidots vienībtestu vajadzībām. Tagad ir izveidots risinājums, bet to plašāk ir iespējams izmantot sistēmas kodola testēšanai, kas nav atkarīgs no datubāzes vaicājumiem vai lietotāja saskarnes. Vienībās, kur ir iesaistīta datubāze nav nodalīti izsaukumi. Vienībtestu veidošana tiek sarežģīta, jo nepieciešams veikt nodalīšanu viscaur algoritmam. Vienībtestu veidošana šāda veida sistēmai ir sarežģīta un bieži neveiksmīgu testu iemesls ir kļūda pašā vienībtestā nevis sistēmā. Vienībtestus nav iespējams veidot arī tām sistēmas funkcionalitātēm, kas tiek nodrošinātas ar datubāžu procedūru iespējām.

Resursu vadības sistēmas modulis Nekustamo īpašumu pārvaldība nodrošina nekustamo īpašumu pamatdatu uzskaitīšanu, kā arī ieņēmumu un izmaksu uzskaiti nekustamajam īpašumam, nodrošinot klientu un līgumu uzskaiti. Sistēmas moduli izmanto uzņēmumi, kas apsaimnieko

dažādus nekustamos īpašumus. Tie var būt uzņēmuma īpašumā esošie nekustamie īpašumi vai nekustamā īpašuma objekti, kas nodoti uzņēmumam apsaimniekošanā.

Viena no sistēmas moduļa iespējām ir izrakstīt ikmēneša apsaimniekošanas rēķinus klientiem, kas ietver arī komunālo maksājumu aprēķinus. Vienkāršākajiem pakalpojumiem tiek noteikta konstanta perioda (mēneša, ceturkšņa) maksa, bet aprēķini var ietvert dažādus mainīgos no kuriem atkarīgs rēķina apjoms, piemēram, dzīvokļu skaitītāji, mājas kopējie skaitītāji, platības koeficienti. Dažkārt aprēķinu algoritmus nevar izvēlēties pats apsaimniekošanas uzņēmums vai komunālo pakalpojumu sniedzējs, tos nosaka ministru kabineta noteikumi vai mājas kopsapulces balsojums. Horizon nekustamo īpašumu pārvaldības modulis nodrošina katram klientam iespēju konfigurēt savas aprēķinu formulas izmantojot dažādus mainīgos un matemātiskās un loģikas iespējas. Viena no iespējām, kas sistēma veidota lai izpildītu Latvijas Republikas likumdošanu ir patēriņa starpības sadales koeficienta analīze, kas arī tiks izmantots par pamatu maģistra darbā.

7. PATĒRIŅA STARPĪBAS SADALES KOEFICIENTA ANALĪZE

Veicot ūdens patēriņa aprēķinus daudzdzīvokļu mājās var tikt uzskaitīti mājas kopējie ūdens skaitītāji un dzīvokļu individuālie ūdens skaitītāji. Veicot aprēķinus periodā var rasties starpība starp mājas kopējā skaitītāja uzrādīto daudzumu un dzīvokļu individuālo skaitītāju daudzumu summu. Ūdens patēriņa starpības sadali nosaka ministru kabineta noteikumi Nr. 1013 "Kārtība, kādā dzīvokļa īpašnieks daudzdzīvokļu dzīvojamā mājā norēķinās par pakalpojumiem, kas saistīti ar dzīvokļa īpašuma lietošanu". Šo noteikumu 19. punktā aprakstīts kam nepieciešams apmaksāt izveidojušos starpību. Starpība tiek sadalīta uz atsevišķo īpašumu skaitu, ņemot vērā noteikumu 19. prim, kur noteikts, ka ūdens patēriņā starpību maksā atsevišķi īpašnieki, ko nosaka papildus analīze par konkrētā dzīvokļa datiem. Ūdens patēriņa starpības sadali ietekmē skaitītāja esamība, skaitītāja verifikācijas dati, skaitītāju rādījumu nodošanas vēsture. Noteikumu 28. punkts papildus nosaka dzīvokļa īpašnieka prombūtnes iespējamību, kad iespējams interpretēt, ka ūdens starpību tam nevar piestādīt. [1]

Pēc autora pieredzes dažādi apsaimniekošanas nozares uzņēmumi ministru kabineta noteikumus Nr. 1013 interpretē dažādi, tāpēc sistēmā tiek nodrošinātas datu analīzes konfigurācijas iespējas. Tas palielina iespējamo scenāriju skaitu attiecīgi sarežģot testēšanu. Sistēmu uzsākot lietot jauniem uzņēmumiem, iespējams nepieciešamas izmaiņas analīzes algoritmā, jo jaunā sistēmas lietotāja likumdošanas interpretāciju sistēma neatbalsta. Analīzē iesaistīti dažādu sistēmas dokumentu dati – līguma dati, skaitītāju dati, skatītāju rādījumi, prombūtnes dokumenti. Resursu vadības sistēma Horizon patēriņa starpības sadales analīzi nodrošina patēriņa starpības sadales koeficienta analīze.

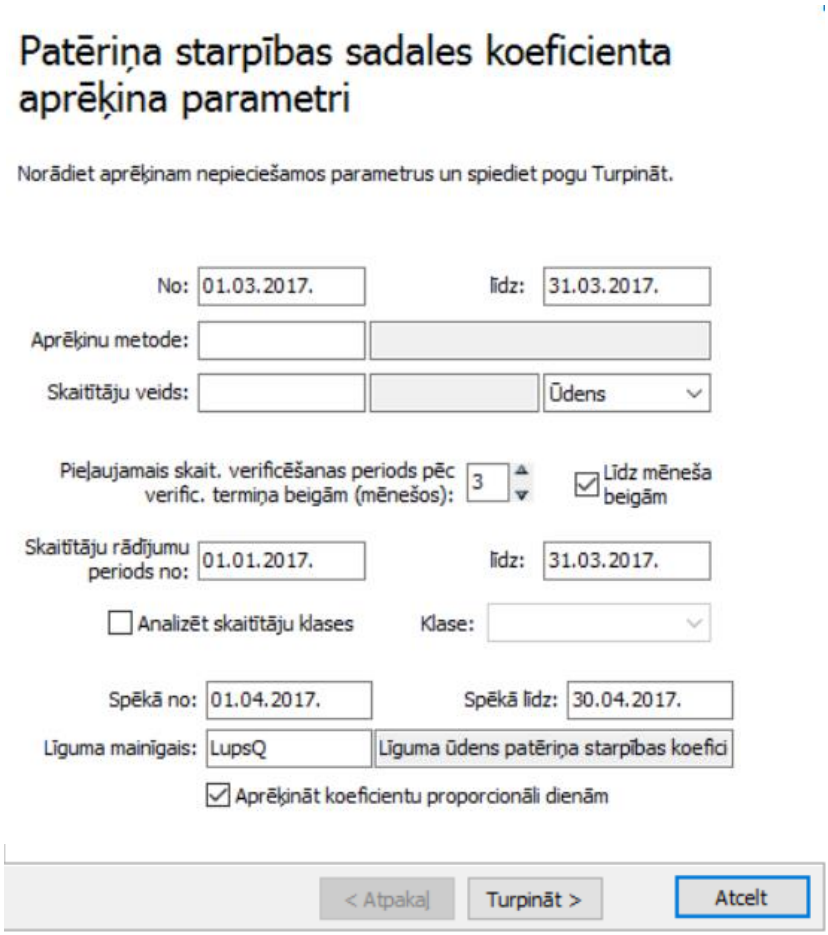
7.1 Algoritms

Sistēmā analīzes algoritms tiek veikts līgumam, kas ar klientu noslēgts par dzīvokļa apsaimniekošanu. Analīzes rezultātā līgumam tiek pievienots mainīgais, kurš nosaka rezultātu. Mainīgais 0 nozīmē, ka līgums neatbilst ministru kabineta noteikumos 19. prim punktā minētajiem noteikumiem, turpmāk, "labais līgums". Mainīgais 1 nozīmē, ka līgums atbilst kādai noteikumos aprakstītajām situācijām un tam ir piestādāma ūdens patēriņa starpība, turpmāk, "sliktais līgums". Analīzi iespējams arī konfigurēt, lai mainīgais tiktu aprēķināts proporcionāli un "sliktais līgums" gadījumā tiktu piešķirta vērtība no 0.01 līdz 1.00 atkarībā no analīzes datiem.

Pētot uzņēmumā pieejamo izstrādes dokumentāciju redzams, ka ir pieejams sākotnējās izstrādes apraksts. Pēc kāda laika, kurā tika veikti labojumi un papildinājumi, ir izveidots jauns apraksts, lai saprastu kā kopumā darbojas algoritms. Veidojot aprakstu atklātas dažādas neprecizitātes algoritma darbībā un veikts refaktorings. Pēc tam dokumentācija nav atjaunota, līdz ar to nav zināms aktuālais algoritms. Autors veica algoritma izpēti un aprakstīšanu pēc sistēmas koda un biznesa loģikas.

Sistēmā analīzi iespējams veikt vienam līgumam vai vairākiem līgumiem vienlaicīgi tos iezīmējot. Tiek piedāvāta iespēja analīzi veikt uz servera norādot konkrētu līgumu filtru, lai palielinātu ātrdarbību un netraucētu lietotāju analīzes laikā veikt citas darbības sistēmā.

Pirms algoritma izpildes lietotājam jāveic konfigurācija, kas ietekmē analizējamo datu atlasi. Konfigurācijas logs attēlots 7.1. attēlā. Tā kā klientu prasības ir atšķirīgas, tad analīzes konfigurācija sniedz iespēju to veikt atbilstoši klienta likumdošanas interpretācijai, bet palielina algoritma sarežģītību.



7.1. att. **Patēriņa starpības sadales koeficienta aprēķina konfigurācijas logs**

Lai saprotamāk aprakstītu algoritmu blokshēmā, autors ir ieviesis konfigurācijas parametru apzīmējumus, kas attēloti tabulā 7.1. Pēc aprēķina uzsākšanas turpmākās darbības netiek attēlotas lietotāja saskarnē. Gadījumos, kad tiek analizēts vairāk kā viens līgums sistēmas lietotājs var redzēt progresa joslu un analizējamo līgumu skaitu un pabeigto analīžu skaitu.

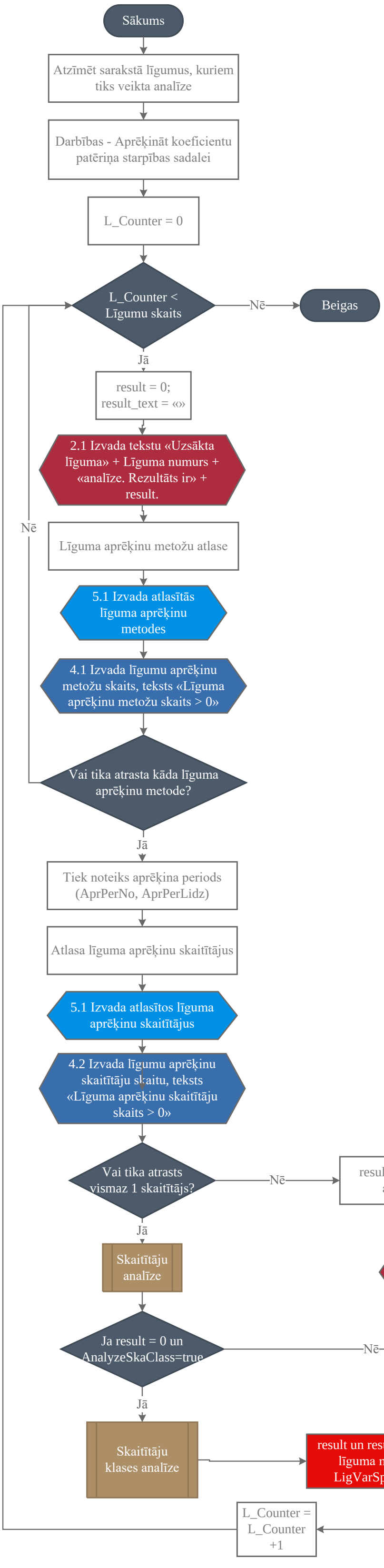
7.1.tabula

Konfigurācijas parametru apzīmējumi

	Nosaukums	**Apzīmējums**	**Datu tips**
1.	No	PER_NO	Datums
2.	Līdz	PER_LIDZ	Datums
3.	Aprēķinu metode	AprMet	Klasifikators
4.	Skaitītāja veids		Klasifikators
5.	Skaitītāja pamatveids		Klasifikators
6.	Pieļaujamais skait. verificēšanas periods pēc verifik. termiņa beigām (mēnešos)	VerifPeriods	Vesels skaitlis
7.	Līdz mēneša beigām	LidzMenB	Būla
8.	Skaitītāju rādījumu periods no	SKARAD_NO	Datums
9.	Skaitītāju rādījumu periods līdz	SKARAD_LIDZ	Datums
10.	Analizēt skaitītāju klases	AnalyzeSkaClass	Būla
11.	Klase	SkaClass	Klasifikators
12.	Spēkā no	LigVarSpekaNo	Datums
13.	Spēkā līdz	LigVarSpekaLidz	Datums
14.	Mainīgais	LigVar	Klasifikators
15.	Aprēķināt koeficientu proporcionāli dienām	PropDien	Būla

Analīzes process sākas ar līguma datu analīzi. Aprēķina uzsākšana un līgumu analīze ir redzama attēlā 7.2 blokshēmas veidā. Ja analīze tiek veikta vairākiem līgumiem, tad tie tiek apskatīti cikla veidā. Lai kontrolētu progresu un nodrošinātu cikla darbību ir ieviests mainīgais L_Counter, kas norāda cik līgumi ir apstrādāti. Analīzei nepieciešami arī mainīgie result un result_text, kur algoritma laikā tiek saglabāta potenciālā līguma mainīgā un tā piezīmju vērtība. Pirms katra līguma apstrādes šo mainīgo vērtības tiek atgrieztas sākuma vērtībās. Pirmajā atlasē tiek pieprasītas līguma aprēķinu metodes. Tiek atlasītas tās metodes, kas atbilst konfigurācijas parametrā norādītajai

vērtībai AprMet, atbilstošā periodā, kas norādīts konfigurācijā (PER_NO un PER_LIDZ). Ja netiek atrasta neviena metode, tad šī līguma analīze tiek izbeigta un tiek analizēts nākamais līgums. Ja tiek atrasta atbilstoša līguma aprēķinu metode, tad izmantojot tās periodu tiek noteikts turpmākais aprēķina periods.

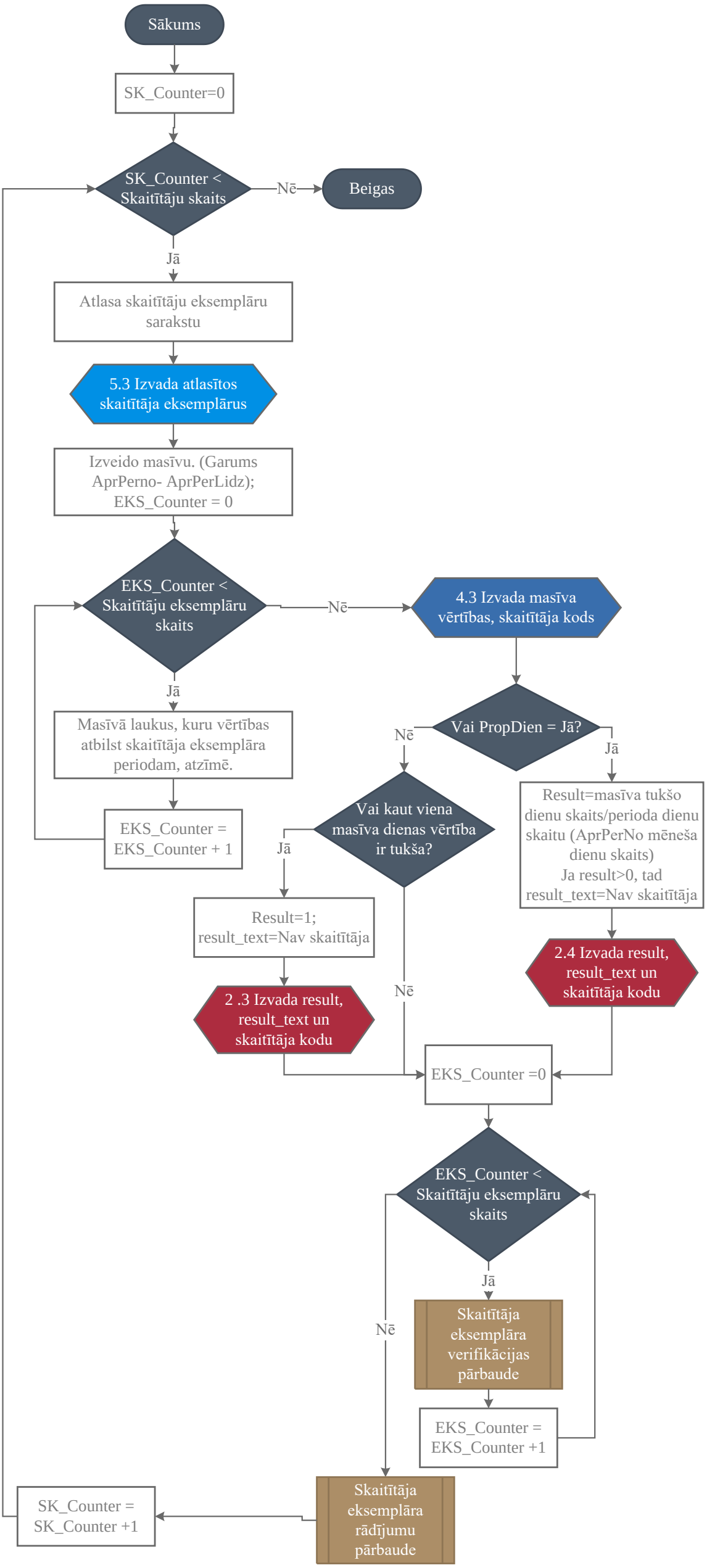


7.2. att. **Līgumu analīzes blokshēma patēriņa starpības sadales koeficienta aprēķinā**

Līgumiem, kuros ir analīzes metode, tiek atlasīti tai piesaistītie aprēķinu skaitītāji. Šajā atlasē tiek izmantoti sākotnējās konfigurācijas parametri skaitītāja veids un skaitītāja pamatveids. Ja kāds no šiem parametriem ir aizpildīts, tad veido atlases filtru ar šīm vērtībām. Tālāk tiek veikta skaitītāju analīze, kas aprakstīta atsevišķā apakšprocesā (attēls 7.3).

Sistēmā ir definēti divi dažādi jēdzieni – skaitītājs un skaitītāja eksemplārs. Par skaitītāju sistēmā uzskata pakalpojuma pieslēgumu konkrētam objektam, piemēram, dzīvoklim ir pievilkta aukstā ūdens caurule, pa kuru var tikt piegādāts ūdens. Patērētā ūdens daudzuma uzskaiti nodrošina skaitītāja eksemplārs, kas ir reālā mērierīce, ko var uzstādīt un var arī neuzstādīt objektā. Ja ministru kabineta noteikumos numur 1013 19.[1] 2. ir minēts, ka mājas starpību piestāda tādiem dzīvokļiem, kuros nav uzstādīti ūdens patēriņa skaitītāji, tad Horizon izpratnē jābūt uzstādītam skaitītāja eksemplāram. [1]

Skaitītāju analīzē tiek veidots cikls katra līguma aprēķinu skaitītāja analīzei. Sākotnēji tiek pārbaudīts vai visā analīzes periodā ir spēkā kāds skaitītāja eksemplārs. Lai to analizētu tiek izveidots masīvs, kura garums atbilst analīzes perioda dienu skaitam. Ciklā tiek pārbaudīti visi skatītāja eksemplāri un atzīmētas dienas, kuras atbilst tā darbības periodam. Pēc tam sistēma veic pārbaudi vai konfigurācijas parametros ir atzīmēts parametrs Aprēķināt koeficientu proporcionāli dienām. Ja rezultāts jāuzrāda proporcionāls, tad tiek veikts aprēķins masīva tukšo dienu skaits dalīts ar perioda dienu skaitu. Ja rezultāts ir lielāks par nulli, tad rezultātam tiek pievienotas piezīmes “Nav skatītāja” un rezultātā saglabāta proporcija. Ja rezultātu nebija nepieciešams aprēķināt proporcionāli, tad tiek veikta pārbaude vai kaut viena masīva dienas vērtība ir tukša un ja tā ir, tad piešķirts rezultāts viens ar piezīmēm “Nav skaitītāja”.

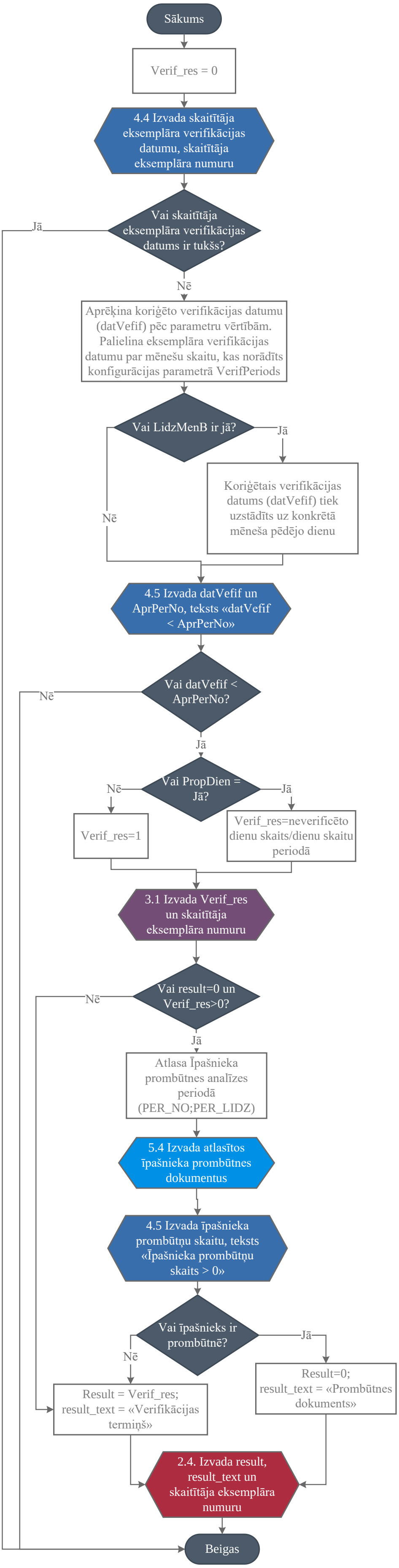


7.3. att. **Skaitītāju analīzes blokshēma patēriņa starpības sadales koeficienta aprēķinā**

Pēc skaitītāja eksemplāra esamības pārbaudes tiek uzsākts jauns cikls cauri visiem skaitītāja eksemplāriem, lai veiktu verifikācijas pārbaudi (attēls 7.4).

Ciklā katram skaitītāja eksemplāram tiek veikta verifikācijas pārbaude. Skaitītāja verifikāciju nosaka nākamās verifikācijas datums. Ja šis datums ir tukšs, tad skaitītāja eksemplārs uzreiz tiek uzskatīts par verificētu un turpina nākamās pārbaudes. Ja nākamās verifikācijas datums ir aizpildīts, tad tiek aprēķināts jauns datums, kas atbilst parametros norādītajām atkāpēm. Lietotājam iespējams norādīt pieļaujamo skaitītāja verificēšanas periods pēc verifikācijas termiņa beigām, kā arī atļaut skatītāja verifikāciju veikt līdz mēneša beigām. Tātad tiek pielielināta nākamā verifikācijas datuma vērtība par mēnešu skaitu, kas pieļauti verifikācijai, kā arī datumu mainīt uz mēneša beigām, ja konfigurācijas parametrs līdz mēneša beigām ir atzīmēts. Ja iegūtais datums ir lielāks par analīzes perioda no, tad skaitītāja eksemplārs vēl arvien ir derīgs un tiek veikta tālāka analīze. Ja skatītājs nav verificēts (iegūtais datums ir mazāks par analīzes perioda no datumu), tad tiek veikta pārbaude vai nepieciešams rezultātu aprēķināt proporcionāli. Ja rezultāts jāaprēķina proporcionāli, tad neverificēto dienu skaits tiek dalīts ar dienu skatu periodā, citādāk rezultāts ir viens. Ja līdz verifikācijas pārbaudei līgums ir uzskatīts par labo un verifikācijas pārbaudes rezultāts ir lielāks par nulli, tad tiek veikta īpašnieka prombūtnes pārbaude.

Vadoties pēc likumdošanas īpašniekam ir tiesības pakalpojumu sniedzējam pieteikt prombūtni, kuras laikā tam nav iespējams verificēt skaitītājus un nodot rādījumus. Prombūtnes laikā verifikācijas vai nenodotu rādījumu dēļ klients nemaksā mājas ūdens starpību. Sistēma atlasa visas prombūtnes analīzes periodā. Ja tiek konstatēta prombūtne, tad rezultāts ir nulle un piezīmēs tiek ierakstīts “Prombūtnes dokuments”, bet ja prombūtne nav atrasta, tad rezultāts ir iepriekšējās verifikācijas pārbaudes rezultāts ar piezīmēm “Verifikācijas termiņš”.



7.4. att. **Skaitītāju eksemplāru verifikācijas analīzes blokshēma patēriņa starpības sadales koeficienta aprēķinā**

Pēc skaitītāja verifikācijas pārbaudes tiek veidots cikls cauri visiem skaitītāja eksemplāriem un apskatīti skaitītāja rādījumi (attēls 7.5). Pirmais tiek pārbaudīts vai līguma aprēķinu skaitītāja spēkā no ir lielāks par parametru PER_NO. Ja ir iestājies šāds gadījums, tad tiek uzskatīts, ka skatītājs ir tikko uzstādīts objektā un īpašniekam tiek piešķirts sadzēšanas laiks, lai nodotu pirmos rādījumus.

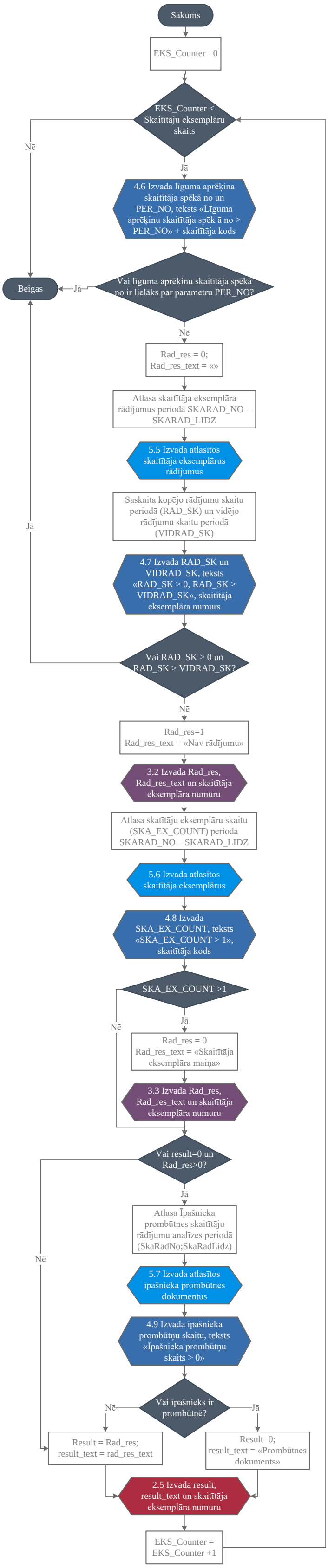
Ja skaitītājs uzstādīts objektā pirms analīzes perioda sākuma, tad tiek veikta skaitītāju rādījumu atlase un saskaitīts kopējais skaitītāju rādījumu skaits un vidējo rādījumu skaits periodā. Sistēmā to nosaka skaitītāja rādījuma daudzuma tips. Iespējamās vērtības ir reģistrēts apjoms, importēts daudzums, vidējais daudzums un importēts vidējais. Par klienta iesniegtiem rādījumiem tiek uzskatīti tikai reģistrēts apjoms, ko sistēmas lietotājs ir ievadījis ar roku, piemēram, pēc klienta zvana, un importēts daudzums, kas ir importēts sistēmā ar Microsoft Excel failu palīdzību, piemēram, no attālinātās skaitītāju rādījumu nolasīšanas sistēmām. Pārējie divi daudzuma tipi ir sistēmas vai tās lietotāja aprēķināti vidējie rādījumi no klienta iepriekš ziņotajiem rādījumiem, tāpēc nevar tikt uzskatīti par klienta ziņotiem rādījumiem. Analīzes rezultātu nosaka ar pārbaudēm vai rādījumu skaits ir lielāks par nulli, tātad vai klients vispār ir nodevis kādu rādījumu periodā un vai rādījumu skaits ir lielāks par vidējo rādījumu skaitu, kas nozīmē, ka vismaz viens no rādījumiem ir klienta ziņots. Ja šie notikumi izpildās, tad līgums nevar būt slikts rādījumu dēļ un tiek uzskatīts, ja viens no skaitītāja eksemplāriem ir labs, tad viss skatītājs ir labs, līdz ar to uzreiz tiek analizēts nākamais skaitītājs. Ja šie nosacījumi neizpildās, tad pagaidu rezultātā tiek saglabāts, ka rezultāta vērtība ir viens un piezīmes ir "Nav rādījumu". Pagaidu rezultāts tiek veidots, jo ir iespējamas situācijas, kas anulē slikto stāvokli rādījumu nenodošanas dēļ. Tās tiek apskatītas tālāk algoritmā.

Ja skaitītāju rādījumu analīzes periodā ir mainījis skaitītāja eksemplāru, tad nevar iegūt slikta līguma statusu skaitītāju rādījumu nenodošanas dēļ. Tiek saskaitīts skaitītāju eksemplāru skaits skaitītāju rādījumu analīzes periodā. Ja tas ir lielāks par 1, tad tiek uzskatīts, ka ir veikta skaitītāju eksemplāru maiņa un līgums nevar būt slikts. Pagaidu rezultātam tiek piešķirta vērtība nulle un piezīmes "Skaitītāja eksemplāra maiņa".

Īpašnieka prombūtnes gadījumā, līdzīgi kā ar skaitītāja verificēšanu, tiek piemērota speciāla apstrāde. Īpašnieka prombūtnes tiek analizētas tikai tad, ja rezultāts, līdz rādījumu pārbaudei ir bijis nulle un pagaidu rezultāts rādījumu analīzei ir lielāks par nulli. Šādā gadījumā tiek atlasītas īpašnieka prombūtnes skaitītāju rādījumu analīzes periodā. Ja tiek konstatēta īpašnieka prombūtne, tad rezultātam tiek piešķirta vērtība nulle un piezīmes ierakstīts "Prombūtnes dokuments". Ja

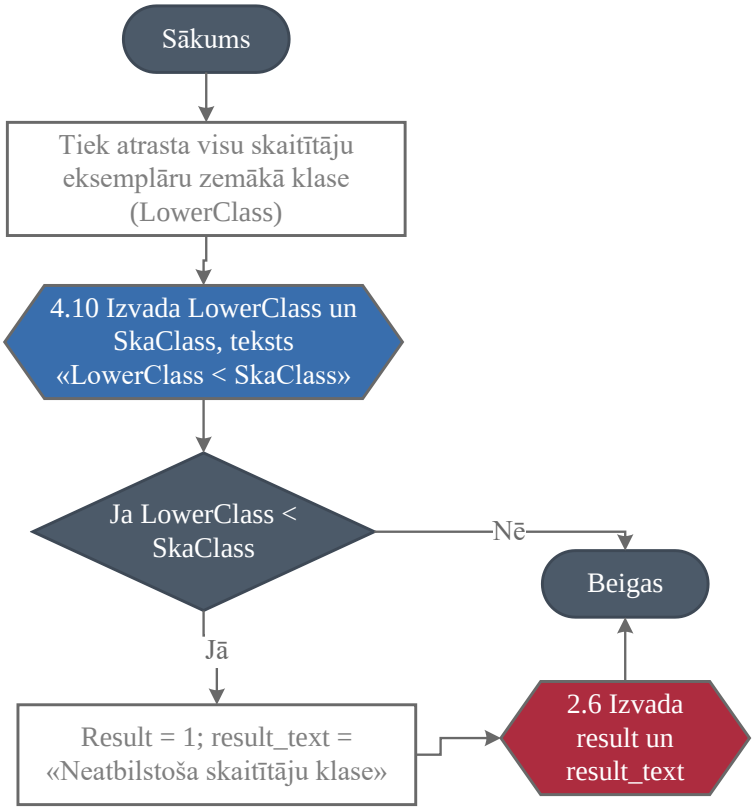
īpašnieka prombūtne netiek atrasta, tad rezultātam tiek piešķirts rādījumu pagaidu rezultāts un tā piezīmes.

Pēc tam tiek veikta nākamā eksemplāra pārbaude. Kad visu skaitītāja eksemplāru pārbaudes ir veiktas, cikls pāriet pie nākamā skaitītāja. Tādā veidā tiek pārbaudīti visi līguma aprēķinu skaitītāji.



7.5. att. **Skaitītāju eksemplāru radījumu analīzes blokshēma patēriņa starpības sadales koeficienta aprēķinā**

Pēc skaitītāju analīzes, līguma analīzes noslēgumā, tiek veikta skaitītāju klases analīze (attēls 7.6), ja tā ir pieprasīta konfigurācijas parametros. Kā redzams attēla 7.2 blokshēmā šī analīze tiek veikta tikai tad, ja līgums līdz šim ir bijis "labais līgums" un konfigurācijas parametros ir atzīmēta vērtība analizēt skaitītāju klases. Sistēma analizē visu skaitītāju klašu vērtības un nosaka zemāko no tām. Ja atrastā klase ir zemāka par konfigurācijas parametros norādīto skaitītāja klasi, tad mainīgajam result tiek piešķirta vērtība 1 un teksts "Neatbilstoša skaitītāju klase". Ja skaitītāja klase nav mazāka par parametros norādīto, tad šis process beidzas nemainot mainīgo vērtības.



7.6. att. **Skaitītāju klases analīzes blokshēma patēriņa starpības sadales koeficienta aprēķinā**

Algoritma noslēgumā mainīgo result un result_text vērtības tiek ierakstītas līguma mainīgajā. Šī darbība tiek veikta uzreiz tad, ja tiek noskaidrots, ka līgumā nav skaitītāju, ja nav nepieciešama skaitītāju klases analīze vai arī pēc skaitītāja klases analīzes, ja tāda notikusi (attēls 7.2). Līguma mainīgais un tā periods tiek noteikts analīzes konfigurācijas parametros.

Pēc tam šis mainīgais tiek izmantots mājas apjoma un apjoma aprēķinos, lai veidotu ikmēneša komunālo maksājumu rēķina rindiņas attiecīgajam līgumam. Sistēma analizē mainīgo vērtības un nosaka vai līgumam ir jāmaksā ūdens patēriņa starpība vai nav.

Tā kā Horizon sistēmas dokumentācijā nebija aktuāla patēriņa starpības sadales koeficienta analīzes apraksta, tad autors pirms prioritāšu mehānisma izpētes izveidoja algoritma darbības aprakstu un blokshēmas, kas jau šobrīd ir izmantojamas sistēmas darbības izskaidrošanai tās lietotājiem. Nākamajā apakšnodaļā autors skaidros prioritāšu mehānisma ieviešanu balsoties uz šajā apakšnodaļā atrodamajām algoritma blokshēmām.

7.2 Starprezultātu punkti un prioritāšu mehānisms

Autors kā iespējamu risinājumu algoritma darbības izpratnei 5. nodaļā minēja prioritāšu mehānismu starprezultātu iegūšanā. Pēc algoritma aprakstīšanas izmantojot blokshēmas iespējams atzīmēt punktus, kuros potenciāli var būt nepieciešams iegūt starprezultātus.

Aplūkojot algoritma shēmu un biznesa procesu autors izdala piecu prioritāšu punktus – gala rezultāta piezīmju vērtības, pagaidu rezultāta izmaiņas algoritma gaitā, pagaidu starprezultātu izmaiņas algoritma gaitā, rezultātu nosakošu salīdzināšanu vērtības un atlases vaicājumu rezultāti. Punktu prioritātes nosauktas sākot no nedetalizētākās. Prioritātēs, kurās nepieciešams piereģistrēt lielāku informācijas daudzumu, būtu nepieciešams veidot žurnāla failus, kuros saglabāt informāciju. Prioritāšu mehānisma izmantošana starprezultātu izvadē nodrošinātu ātrdarbības saglabāšanu analīzei. Izpildot analīzi noklusētā režīmā jeb ar pirmo prioritāti nebūtu nepieciešams saglabāt papildus informāciju, kas notiktu, ja starprezultātu izvade tiktu nodrošināta vienmēr. Prioritāšu mehānisms ļauj lietotājam izvēlēties detalizāciju kādā iegūt izvades žurnāla failu. Ikmēneša analīzi sistēmas lietotājs var veikt masveidā noklusētā režīmā, bet ja rodas šaubas par sistēmas darbību vai rezultāta korektumu, tad iespējams atsevišķus gadījumus izpildīt ar augstākām prioritātēm.

Starprezultātu punkti atzīmēti autora veidotajās blokshēmās, izmantojot sešstūra elementu. Punkta prioritāti apzīmē pildījuma krāsa. Prioritātēm piešķirtās krāsas redzamas attēlā 7.7.

7.7. att. **Starprezultātu punktu prioritāšu krāsu apzīmējumi**

Ar pirmo prioritāti aprakstīta gala rezultāta piezīmju vērtības piereģistrēšana. Ar pirmo prioritāti sistēma darbojas pēc noklusējuma un saglabātajam līguma mainīgajam pievieno piezīmes par to, kāda iemesla dēļ ir radies rezultāts, kas ir saglabāts. Piemēram, ja līgums ir slikts, jo skaitītājs nav verificēts, tad piezīmēs tiek ierakstīts "Verifikācijas termiņš". Šo iespēju nodrošina piezīmju piešķiršana jau pagaidu rezultātiem un tad saglabāšana arī gala rezultāta gadījumā. Šīs prioritātes darbības vietu blokshēmā iespējams redzēt attēlā 7.2, kur līgumā tiek ierakstīta result_text vērtība, shēmā šī darbība atzīmēta ar koši sarkanu krāsu. Tā kā sistēmā šī prioritāte jau ir iestrādāta, tad tā atzīmēta ar kvadrāta formas procesa apzīmējumu. Šī prioritāte nodrošina sistēmas lietotājam uzzināt analīzes piešķirtās vērtības iemeslu. Piezīmes tiek pievienotas visos gadījumos, kad līgums ir slikts un izņēmumu gadījumos, kad līgums paradīts par labu, kaut gan varēja būt slikts. Iespējamās piezīmju vērtības ir "Nav aprēķina skaitītāja", "Nav skaitītāja", "Nav rādījumu", "Skaitītāja eksemplāra maiņa", "Prombūtnes dokuments", "Verifikācijas termiņš", "Neatbilstoša skaitītāja klase". Apskatot piezīmju izmantojumu algoritmā, autors iesaka piezīmes "Nav skaitītāja" labot uz "Nav skaitītāja eksemplāra", lai tās izteiktu īsto datu analīzes nozīmi.

Otro prioritāti izmanto, lai saglabātu pagaidu rezultāta izmaiņas algoritma gaitā. Lai iegūtu papildus informāciju par analīzes gaitu otrā prioritāte izvada mainīgā result un result_text vērtības analīzes gaitā. Veicot analīzi ar otro prioritāti katra līguma cikla sākumā, kad result un result_text mainīgais tiek nonullēts, tiek izvadītas to vērtības un līguma numurs, kura analīze tiek uzsākta, kā redzams attēlā 7.2, starprezultātu punktā 2.1. Turpmāk analīzes laikā, ja tiek mainīta result un result_text vērtība, pēc tam tiek ievietots starprezultātu punkts, kas izvada jaunās vērtības. Nepieciešamības gadījumā tiek izvadīta papildus informāciju, piemēram, attēlā 7.4 starprezultātu punktā 2.4 tiek izvadīts skaitītāja eksemplāra numurs, kas dod papildus informāciju kuram skaitītājam ir beidzies verifikācijas termiņš. Šīs prioritātes izvadi iespējams nodrošināt saglabājot žurnāla failu vai izvadot informāciju sistēmas logā, ko pēc tam iespējams saglabāt kā teksta failu.

Trešā prioritāte izvada papildus informāciju par situācijām, kad sistēmā tiek saglabāti pagaidu starprezultāti, jo tos var mainīt izņēmumu gadījumi. Viens no šādiem gadījumiem ir skaitītāja eksemplāra verifikācijas pārbaude. Algoritma blokshēmā ir ieviests mainītais Verif_res, kurā tiek saglabāts rezultāts par verifikācijas pārbaudi. Attēlā 7.4. redzams starprezultātu punkts 3.1, kur tiek izvadīta mainīgā Verif_res vērtība un skaitītāja eksemplārs, kuram tā piešķirta. Otrs šāda veida mainīgais tiek izmantots skaitītāja eksemplāru rādījumu analīzē. Algoritmu blokshēmā

ir ieviests mainīgais Rad_res un Rad_res_text. Attēlā 7.5 redzami starprezultātu punkti 3.2 un 3.3, kuri izvada mainīgo vērtības un skaitītāja eksemplāra numuru, kas tiek analizēts.

Ceturtā prioritāte izvada rezultātu nosakošu salīdzināšanu vērtības. Šāda veida informācija var palīdzēt noteikt vai problēma ir salīdzināmajos lielumos vai salīdzināšanas procesā. Šajās situācijās sistēmai nepieciešams izvadīt salīdzināmās vērtības un lietotāja ērtībai sniegt komentāru par salīdzināšanas nosacījumu. Attēlā 7.2 redzams starprezultātu punkts 4.1, kurā tiek izvadīta informācija par līgumu aprēķinu metožu skaitu un tekstu "Līguma aprēķinu metožu skaits > 0", kas paskaidro veicamās salīdzināšanas būtību. Autors kopumā blokshēmās atzīmējis desmit šādas prioritātes punktus pirms nosacījumu elementiem.

Piektās prioritāte izvada atlases vaicājumu rezultātus. Šāda veida izvadi visērtāk būtu veikt saglabājot datus žurnāla failos. Šī ir prioritāte ar vislielāko detalizāciju. Starprezultātu punkti izvietoti vietās, kur sistēma veic atlases vaicājumus datubāzei. Šīs prioritātes izpilde dod iespēju lietotājam pārbaudīt datu atlases pareizību un arī iepriekšējo prioritāšu rezultātu pareizību, dodot iespēju pilnībā izprast konkrētās analīzes gaitu. Attēlā 7.3 redzams starprezultātu punkts 5.3, kas izvada atlasītos skaitītāja eksemplārus. Šāda informācija ļauj apskatīt eksemplāru sarakstu, kas tiks analizēti. Attēlā 7.5 starprezultātu punkts 5.5 izvada atlasītos skaitītāja eksemplāra rādījumus. Pēc tam algoritms aprēķina mainīgos – rādījumu skaitu atkarībā no daudzums tipa. Šāda starppunkta izvadītā informācija palīdz saprast, ka visi rādījumi tiek atrasti un vai tie pēc tam tiek pareizi saskaitīti rādījumu skaitā. Autors kopumā algoritma blokshēmās ir atzīmējis septiņus šādus punktus.

7.3 Kļūdu analīze izmantojot prioritāšu mehānismu

Lai saprastu prioritāšu mehānisma izmantošanas praktisko nozīmi starprezultātu izvadē autors apskatīs sistēmas lietotāju problēmu pieteikumus, kas izveidoti pieteikumu sistēmā. Sistēmas lietotājam rodas nepieciešamība vērsties pie programmatūras risinājuma, ja rodas aizdomas, ka sistēma analīzi veic nepareizi, vai nav iespējams izprast rezultāta iemeslu. Sistēmas lietotājam ir iespējams savu problēmu nodot klientu atbalsta speciālistam, kura uzdevums ir apskatīt lietotāja pieteikto kļūdu vai konsultāciju. Par kļūdu klasificējas problēma, kur nepieciešams koda labojums, jo sistēma nestrādā tā kā paredzēts. Konsultācijas ietvaros lietotājam tiek sniegta informācija par sistēmas darbību vai veiktas konfigurācijas izmaiņas, lai sistēma strādātu kā vēlams, bez izmaiņām sistēmas kodā.

Ja lietotāja izveidotais pieteikums ir kļūda, tad klientu atbalsta speciālista darbs ir kļūdu atkārtot un nodot izstrādei labošanai. Ja lietotājs ir pieteicis konsultāciju, tad paskaidrojumus sniedz klientu atbalsta speciālists. Dažkārt lietotājs uzskata, ka pieteikums ir kļūda, kaut gan īstenībā problēmas cēlonis ir nepareiza konfigurācija vai datu neprecizitāte. Šādus gadījumus nepieciešams atpazīt tieši klientu atbalsta speciālistam un sniegt skaidrojumu klientam par situāciju, bet dažkārt ar esošajiem līdzekļiem tas nav iespējams un nepieciešams prasīt papildus informāciju izstrādei. Šajā apakšnodaļā autors apskatīs šādas situācijas, kad lietotājs vai klientu atbalsta speciālists nav varējis izprast problēmas cēloni – tā ir kļūda vai konsultācija – pašu spēkiem.

Klienta problēmas pieteikums numur 1 ir saistīts ar īpašnieka prombūtnes reģistrēšanu. Līgumam reģistrēta īpašnieka prombūtne no pirmā oktobra līdz trīsdesmit pirmajam decembrim. Tiek veikta patēriņa starpības koeficienta analīze par oktobra mēnesi un sistēma izdod rezultātu, ka līgums ir slikts ar piezīmēm "Nav rādījumu". Tā kā līgumam ir reģistrēta īpašnieka prombūtne, tad šāda veida piezīmes nav pareizas, jo līgums nevar būt slikts rādījumu nenodošanas dēļ.

Klientu atbalsta speciālists ir mēģinājis šādu situāciju atkārtot savā testa vidē tādā pašā sistēmas versijā. Ievadot īpašnieka prombūtni līgumam, kurš nav nodevis rādījumus, tika ņemta vērā prombūtne un rezultāts bija nulle ar piezīmēm "Prombūtnes dokuments". Klientu atbalsta speciālists nesaskatīja būtisku atšķirību klienta piemērā un viņa veidotajā, tāpēc pats nevarēja izprast sistēmas darbību. Tika pieņemts lēmums veidot iekšējo pieteikumu izstrādei, lai noskaidrotu atšķirības sistēmas darbībā iemeslus. Situācijas apskatīja programmētājs un paskaidroja, ka atšķirību rada skaitītāju skaits, kuriem nav nodoti rādījumi. Klienta vidē līgumā bija divi skaitītāji un nevienā no tiem nebija nodotu rādījumu, bet klientu atbalsta speciālista vidē bija piemērs ar vienu skaitītāju, kas strādā pareizi. Sistēmas kļūda radās ciklā, jo pirmajam skaitītājam prombūtne tika identificēta, bet otrajam tā vairs netika meklēta un tika izdots rezultāts viens un piezīmes "Nav rādījumu".

Šajā gadījumā varam secināt, ka pirmās prioritātes starprezultāti nedeva pietiekamu skaidrojumu par situāciju, bet norādīja virzienu kur meklēt problēmu – rādījumu pārbaudē. Autors apskatīs kā šāda situācija izskatītos nākamo prioritāšu darbības gadījumā.

Izpildot analīzi ar otro prioritāti tiktu iegūts papildus žurnāla fails. Izpildot piemērus sistēmā potenciāli varētu iegūt žurnāla failu, kas redzams attēlā 7.8. Šajā gadījumā skaidri redzama piemēru atšķirība, ka klienta piemērā tiek analizēti divi skaitītāju eksemplāri un klientu atbalsta speciālista piemērā tikai viens. Rodas iespaids, ka nepareizi strādā tieši otrā skaitītāja eksemplāra analīze, bet nav zināms iemesls, kāpēc netiek ņemta vērā īpašnieka prombūtne.

Klienta piemērs:

```
1 Uzsākta līguma TESTKL001 analīze. Rezultāts ir 0.
2 0, "Prombūtnes dokuments", SKEKS1
3 1, "Nav rādījumu", SKEKS2
```

Klienta atbalsta speciālista piemērs:

```
1 Uzsākta līguma TESTKAS001 analīze. Rezultāts ir 0.
2 0, "Prombūtnes dokuments", SKEKS1
```

7.7. att. **Iespējamais žurnāla fails pieteikuma Nr. 1 analīzes izpildei ar otro prioritāti**

Lai vēl precīzāk izprastu situāciju autors, izveidoja izvades failus, ko būtu iespējams iegūt ar analīzes izpildi ar trešo prioritāti. Izpildot piemērus ar trešo prioritāti iespējams redzēt rādījumu analīzes starprezultātus (attēls 7.9). Šim piemēram situācijas atrisināšanai šāda informācija nepalīdz. Tikai norāda, ka šobrīd tiek domāts pareizā virzienā, ka problēma ir īpašnieka prombūtnes dokumentu pārbaudē otrajam skaitītāja eksemplāram.

Klienta piemērs:

```
1 Uzsākta līguma TESTKL001 analīze. Rezultāts ir 0.
2 Rad_res 1, "Nav rādījumu", SKEKS1
3 0, "Prombūtnes dokuments", SKEKS1
4 Rad_res 1, "Nav rādījumu", SKEKS2
5 1, "Nav rādījumu", SKEKS2
```

Klienta atbalsta speciālista piemērs:

```
1 Uzsākta līguma TESTKAS001 analīze. Rezultāts ir 0.
2 Rad_res 1, "Nav rādījumu", SKEKS1
3 0, "Prombūtnes dokuments", SKEKS1
```

7.9. att. **Iespējamais žurnāla fails pieteikuma Nr. 1 analīzes izpildei ar trešo prioritāti**

Turpinot padziļinātu analīzi iespējams veikt ar ceturto prioritāti, kas dod ieskatu rezultātus ietekmējošu salīdzināšanu vērtībās. Pieteikumu numur viens piemēru rezultāti izpildot analīzi ar ceturto prioritāti redzami attēlā 7.10. Aplūkojot klienta piemēra žurnāla failu, redzams, ka pirmā skaitītāja analīzē 12. rindā tiek veikta salīdzināšana īpašnieka prombūtnes esamībai, bet pēc tam analizējot otro skaitītāju šādas salīdzināšanas nav. Pēc analīzes blokshēmas (attēls 7.5) tādai būtu jābūt starp 20. un 21. rindu. Jāievēro, ja otrajam skaitītājam un tā eksemplāram netiek veikta pārbaude par skaitītāja eksemplāra maiņu, kas pirmajam skaitītājam redzama 11. rindā. No šādiem

izvades datiem var secināt, ka sistēma apskatot otro skaitītāju un skaitītāja eksemplāru neveic pārbaudes, kas paredzētas izņēmuma gadījumiem, kad slikts līgums var kļūt par labu.

Klientu atbalsta speciālistam šāds izvada fails var dot informāciju, ka šajā gadījumā problēma nav sistēmas datos un atšķirību klienta piemērā un testa vides piemērā rada analizējamo līguma aprēķinu skaitītāju skaits. Apskatot detalizētāku žurnāla failu, šajā gadījumā ar ceturto prioritāti saprotams, ka sistēma neveic visas darbības, kas paredzētas analīzes blokshēmā, un visticamāk tā ir sistēmas kļūda, kas nododama izstrādei labošanai. Ar šādu iespēju klientu atbalsta speciālists var pats izprast sistēmas darbību un ar klientu un izstrādi komunicēt ar jau detalizētāku informāciju bez izstrādes konsultācijām.

Klienta piemērs:

```
Uzsākta līguma TESTKL001 analīze. Rezultāts ir 0.
1, Līguma aprēķinu metožu skaits > 0
2, Līguma aprēķinu skaitītāju skaits >0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1, SK1
21.03.2019, SKEKS1
30.06.2019, 01.10.2016, datVefif < AprPerNo
1, Īpašnieka prombūtņu skaits > 0
21.03.2015, 01.10.2016, Līguma aprēķinu skaitītāja spēkā no > PER_NO, SK1
0, 0, RAD_SK > 0, RAD_SK > VIDRAD_SK, SKEKS1
Rad_res 1, "Nav rādījumu", SKEKS1
1, SKA_EX_COUNT > 1
1, Īpašnieka prombūtņu skaits > 0
0, "Prombūtnes dokuments", SKEKS1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1, SK2
26.11.2018., SKEKS2
28.02.2019., 01.10.2016, datVefif < AprPerNo
1, Īpašnieka prombūtņu skaits > 0
26.11.2014, 01.10.2016, Līguma aprēķinu skaitītāja spēkā no > PER_NO, SK2
3, 3, RAD_SK > 0, RAD_SK > VIDRAD_SK, SKEKS2
Rad_res 1, "Nav rādījumu", SKEKS2
1, "Nav rādījumu", SKEKS2
```

Klienta atbalsta speciālista piemērs:

```
Uzsākta līguma TESTKAS001 analīze. Rezultāts ir 0.
1, Līguma aprēķinu metožu skaits > 0
1, Līguma aprēķinu skaitītāju skaits > 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1, SK1
01.01.2020, SKEKS1
30.04.2020, 01.10.2016, datVefif < AprPerNo
1, Īpašnieka prombūtņu skaits > 0
01.01.2016, 01.10.2016, Līguma aprēķinu skaitītāja spēkā no > PER_NO, SK1
0, 0, Rad_SK > 0, RAD_SKA > VIDRAD_SK, SKEKS1
Rad_res 1, "Nav rādījumu", SKEKS1
0, "Prombūtnes dokuments", SKEKS1
```

7.10. att. **Iespējamais žurnāla fails pieteikuma Nr. 1 analīzes izpildei ar ceturto prioritāti**

Klienta problēmas pieteikums numur 2 ir saistīts ar skaitītāju esamības analīzi. Līgumam ir četri aprēķinu skaitītāji, bet sistēmas analīzes rezultāts ir viens ar piezīmēm "Nav skaitītāja". Klients sniedzis papildus informāciju, ka dzīvoklī veikta skaitītāju eksemplāru maiņa trīsdesmitajā un trīsdesmit pirmajā maijā. Patēriņa starpības sadales koeficienta analīze veikta par maija mēnesi.

Klientu atbalsta speciālists apskatot piemēru neatrada datu problēmas, jo visu analīzes periodu līgumā bija spēkā skaitītājs un skaitītāja eksemplārs. Pārbaudot savā testa vidē konsultants secina, ka gadījumā, ja skaitītāja maiņa analīzes mēnesī nav veikta, tad sistēma skaitītājus atrod veiksmīgi. Lai saprastu kāpēc sistēma neatrod skaitītājus gadījumā, ja veikta skaitītāja maiņa klientu atbalsta speciālistam nākas izveidot iekšējo konsultācijas pieteikumu un prasīt skaidrojumu izstrādei.

Izstrāde pēc piemēra izpētīšanas secināja, ka problēma ir skaitītāju eksemplāru atlases periodā. Ja skaitītāja eksemplāra maiņas jaunais skaitītājs tika uzstādīts patēriņa starpības koeficienta analīzes perioda beigu datumā, tad sistēma to neatrada. Skaitītāju eksemplāru atlasē pastāvēja nosacījums, ka skaitītāja eksemplāra spēkā no datums ir mazāks par aprēķina periodu līdz. Tātad situācija, kad skaitītāja eksemplāra spēkā no datums sakrīt ar aprēķina periodu līdz netika atlasīta.

Izpildot analīzi ar pirmo prioritāti tiek izgūts rezultāts viens ar piezīmēm "Nav skaitītāja", kas nozīmētu, ka kādu no aprēķina perioda dienām skaitītājam nav atrasts skaitītāja eksemplārs. Aplūkojot datus redzams, ka sistēmā katrā analīzes dienā ir reģistrēts skaitītāja eksemplārs.

Lai izprastu situāciju sīkāk autors izveidos potenciālus analīzes žurnāla failus, ja būtu pieejami starprezultāti ar prioritāšu mehānismu. Žurnāla faila piemērs veicot analīzi ar otro prioritāti redzams attēlā 7.11. Varam secināt, ka analīze visos četros skaitītājos nav atradusi aktīvu skaitītāja eksemplāru kādā no analīzes dienām.

```
1 Uzsākta līguma TEST002 analīze. Rezultāts ir 0.
2 1, "Nav skaitītāja", SK1
3 1, "Nav skaitītāja", SK2
4 1, "Nav skaitītāja", SK3
5 1, "Nav skaitītāja", SK4
```

7.11. att. **Iespējamais žurnāla fails pieteikuma Nr. 2 analīzes izpildei ar otro prioritāti**

Šajā piemērā trešās prioritātes starprezultāti nedotu papildus informāciju, jo tie ir saistīti ar gadījumiem, kad analīzē tiek izmantoti papildus starprezultātu mainīgie, lai apstrādātu skaitītāju eksemplāru verifikācijas un rādījumu neesamības pielaides īpašnieka prombūtnes vai skaitītāja

eksemplāra maiņas gadījumā. Tā kā šajā piemērā ar skaitītāju rādījumiem un skaitītāju eksemplāru verifikāciju viss ir kārtībā, tad starprezultātu mainīgie netiks veidoti.

Apskatot izvades failu, kas potenciāli rakstos piemēra analīzi veicot ar ceturto prioritāti (attēls 7.12), redzams, ka skaitītājam periodā pēdējā dienā nav aktīva eksemplāra. Kā redzams ceturtajā, vienpadsmitajā, astoņpadsmitajā un divdesmit piektajā rindiņā masīva pēdējā vērtība ir 0, kas nozīmē, ka 31.05.2016 nav bijis spēkā esošs skaitītāja eksemplārs. Klients piemērā ir aprakstījis, ka no 31.05.2016 ir stājies spēkā jauns skaitītāja eksemplārs. Tātad šie eksemplāri nav atlasīti vai netiek ņemti vērā masīva aizpildē. Apskatot izvades failu redzams, ka skaitītāja eksemplāra verifikācijas pārbaude arī tiek veikta tikai vienam eksemplāram katram skaitītājam. Tas rada aizdomas, ka tiešām nav atlasīti jaunie skaitītāja eksemplāri. Vai nestrādā vēl kāds cikls vairāku skaitītāja eksemplāru pārbaudei.

```
Uzsākta līguma TEST002 analīze. Rezultāts ir 0.
1, Līguma aprēķinu metožu skaits > 0
4, Līguma aprēķinu skaitītāju skaits > 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0, SK1
1, "Nav skaitītāja", SK1
25.05.2016, SKEKS1
31.08.2016, 31.05.2016, datVefif < AprPerNo
0, Īpašnieka prombūtņu skaits > 0
25.05.2012, 01.05.2016, Līguma aprēķinu skaitītāja spēkā no > PER_NO, SK1
3, 0, RAD_SK > 0, RAD_SK > VIDRAD_SK, SKEKS1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0, SK2
1, "Nav skaitītāja", SK2
25.05.2016, SKEKS3
31.08.2016, 31.05.2016, datVefif < AprPerNo
0, Īpašnieka prombūtņu skaits > 0
25.05.2012, 01.05.2016, Līguma aprēķinu skaitītāja spēkā no > PER_NO, SK2
3, 0, RAD_SK > 0, RAD_SK > VIDRAD_SK, SKEKS3
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0, SK3
1, "Nav skaitītāja", SK3
25.05.2016, SKEKS5
31.08.2016, 31.05.2016, datVefif < AprPerNo
0, Īpašnieka prombūtņu skaits > 0
25.05.2012, 01.05.2016, Līguma aprēķinu skaitītāja spēkā no > PER_NO, SK3
3, 0, RAD_SK > 0, RAD_SK > VIDRAD_SK, SKEKS5
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0, SK4
1, "Nav skaitītāja", SK4
25.05.2016, SKEKS7
31.08.2016, 31.05.2016, datVefif < AprPerNo
0, Īpašnieka prombūtņu skaits > 0
25.05.2012, 01.05.2016, Līguma aprēķinu skaitītāja spēkā no > PER_NO, SK4
3, 0, RAD_SK > 0, RAD_SK > VIDRAD_SK, SKEKS7
```

7.12. att. **Iespējamais žurnāla fails pieteikuma Nr. 2 analīzes izpildei ar ceturto prioritāti**

Lai pārliecinātos par eksemplāru atlasīšanu nepieciešams veikt analīzi ar piekto prioritāti, kas izvada atlases vaicājumu rezultātus. Piektās prioritātes analīzes iespējamos rezultātus autors apskatījis līdz starprezultātu punktam 5.3 (attēls 7.3), kur tiek izvadīti atlasītie skaitītāja eksemplāri. Iespējamais izvades fails parāda, ka tiek atlasīti tikai četri no astoņiem skaitītāja eksemplāriem, kas ir spēkā analīzes periodā (attēls 7.13). No izvades faila varam secināt, ka problēma ir tieši atlases vaicājumā un tā ir kļūda, ko nepieciešams nodot izstrādei labojuma veikšanai.

	Līguma numurs	Aprēķinu metode	Spēkā no	Spēkā līdz
1	TEST002	UdStarp	01.01.2003	

	Skaitītāja kods	Spēkā no	Spēkā līdz	Skaitītāja veids	Skaitītāja pamatveids
1	SK1	25.05.2012		Aukstais ūdens	Ūdens
2	SK2	25.05.2012		Aukstais ūdens	Ūdens
3	SK3	25.05.2012		Karstais ūdens	Ūdens
4	SK4	25.05.2012		Karstais ūdens	Ūdens

	Skaitītāja eks. numurs	Spēkā no	Spēkā līdz	Nāk. verifikācijas datums
1	SKEKS1	25.05.2012	30.05.2016	25.05.2016
2	SKEKS3	25.05.2012	30.05.2016	25.05.2016
3	SKEKS5	25.05.2012	30.05.2016	25.05.2016
4	SKEKS7	25.05.2012	30.05.2016	25.05.2016

7.13. att. **Iespējamais žurnāla fails pieteikuma Nr. 2 analīzes izpildei ar piekto prioritāti**

Klienta pieteikums numur trīs ir saistīts ar skaitītāju esamību. Klients veicis patēriņa starpības sadales koeficienta analīzi par augustu un līgumā iegūtais rezultāts ir viens ar piezīmēm "Nav aprēķina skaitītāja". Klients izveidojis pieteikumu, jo neizprot iegūtu rezultātu un uzskata, ka sistēmā ir kļūda, jo skaitītāji sistēmā ir pievienoti. Klientam līdz šim dzīvoklī nav bijuši skaitītāji, tie uzstādīti 01.08.2016. Šajā gadījumā uz klienta pieteikumu spēja atbildēt pats klientu atblasta speciālists, bet šajā gadījumā prioritāšu mehānisma starprezultātu izvade, būtu ļāvusi pašam klientam izprast situāciju. Klientu atbalsta speciālists apskatot datus secināja, ka sistēmas lietotājs ir ievadījis objekta skaitītājus, bet nav pievienojis tos līgumā konkrētajai aprēķinu metodei. Sistēma atlasa skaitītājus no līguma aprēķinu skaitītāju saraksta, kur šie skaitītāji nebija pievienoti.

Veicot patēriņa starpības sadales koeficienta analīzi ar otro prioritāti tiktu izveidots žurnāla fails, kas redzams attēlā 7.14. Papildus informāciju šis fails nesniedz, jo skaitītāji tiek pārbaudīti analīzes sākumā un kā redzams attēlā 7.2 blokshēmā, ja līgumam netiek atrasti skaitītāji, tad tiek pieņemts rezultāts un tālāka analīze nenotiek.

```
1  Uzsākta līguma TEST003 analīze. Rezultāts ir 0.
2  1, "Nav aprēķinu skaitītāja"
```

7.14. att. **Iespējamais žurnāla fails pieteikuma Nr. 3 analīzes izpildei ar otro prioritāti**

Tā kā netiek atrasti skaitītāji, tad izpildīt analīzi ar trešo prioritāti nebūs jēgas, jo pagaidu starprezultāti, kurus izvada trešās prioritātes starppunktos, atrodami tikai skaitītāju analīzē. To vai algoritms neatrod skaitītājus vai nepareizi veic salīdzināšanu iespējams saprast izpildot analīzi ar ceturto prioritāti. Attēlā 7.15 redzamas iespējamais izvades fails analīzes izpildei ar ceturto prioritāti. Trešajā rindā redzams, ka sistēma ir atradusi nulle līguma aprēķinu skaitītājus. Tā kā šeit tiek izmantots jēdziens līguma aprēķinu skaitītājs, paskaidrojot salīdzināšanu, ko veic sistēma, pārbaudot, ka sistēmā ir vismaz viens līguma aprēķinu skaitītājs, sistēmas lietotājam tiek dota norāde, kur dati trūkst. Šāda veida paskaidrojums lietotājam ar zināšanām par nekustamo īpašumu pārvaldības moduli norāda, ka jāpārbauda līguma dati. Ja līgumā būtu aprēķinu skaitītāji, tad klients varētu pieteikt pieteikumu, ka nenotiek pareiza datu atlase, bet šajā pieteikumā klients pats būtu varējis atrast vietu sistēmā, kur dati neatbilst vēlamajam.

```
1  Uzsākta līguma TEST003 analīze. Rezultāts ir 0.
2  1, Līguma aprēķinu metožu skaits > 0
3  0, Līguma aprēķinu skaitītāju skaits > 0
4  1, "Nav aprēķinu skaitītāja"
```

7.15. att. **Iespējamais žurnāla fails pieteikuma Nr. 3 analīzes izpildei ar ceturto prioritāti**

Maģistra darbā veidotie iespējamie žurnāla faili ir autora veidoti un šobrīd šāds risinājums nav iestrādāts sistēmā. Faili ir autora interpretācija par iespējamo risinājumu.

Apskatot klientu pieteikumus autors secina, ka šāda veida papildinājumi sistēma atvieglotu sistēmas lietotāju iespējas izprast patēriņa starpības sadales koeficienta analīzes rezultātu. Apskatot piemērus izmantojot starprezultātu izvadi ar prioritāšu mehānismu redzams, ka samazinās cilvēkresursu izmantošana. Sistēmas lietotājs pats var izdarīt secinājumus par analīzes darbību neiesaistot izstrādes darbinieku, kas sniegtu informāciju no sistēmas koda vai atkļūdošanas rīka starprezultātiem. Veicinot sistēmas lietotāju zināšanas par tās darbību samazinātos nepieciešamais uzturēšanas laiks sistēmas izstrādātājam, uzlabotos sistēmas uzticamība un samazinātos manuālais darbs, ko sistēmas lietotājs iespējams veic paralēli, lai pārliecinātos par sistēmas rezultātu pareizību.

Maģistra darbā veiktā izpēte un praktiskā darba rezultāti tiks iesniegti sistēmas izstrādātājam, lai tiktu izvērtēta iespēja iestādāt šāda veida starprezultātu izvadi sistēmā. Maģistra darbā veiktais praktiskais darbs ir sagatavots darba uzdevums sistēmas izstrādātājam, lai nākotnē vieglāk risinātu lietotāju problēmas.

REZULTĀTI

1. Kad sistēma jau ir izstrādāta un ilgtermiņā ir jāveic tās uzturēšana, nepārtraukti tiek veikti uzlabojumi, kas izraisa izmaiņas sistēmā. Šāda situācijā ļoti svarīga ir regresa testēšana, kurai ar ierobežotiem resursiem ir jāpārbauda pēc iespējas vairāk funkcionalitātes.
2. Automatizētajam testam beidzoties neveiksmīgi nepieciešama manuāla analīze defekta cēloņa meklēšanai. Sarežģītu algoritmu gadījumā programmētājiem ir iespējams ar atkļūdotāja rīku pa soļiem apskatīt darbības ar starprezultātiem. Testētājiem testējot ar lietotāja saskarni parasti šādas iespējas noklusēti nav paredzētas.
3. Lai automatizētu algoritma izprašanas procesu, autors, aplūkojot pieejamo literatūru, secināja, ka iespējams izmantot starprezultātu izvadi ar prioritāšu mehānismu. Šāda metode dos papildus informāciju sistēmas lietotājam, ir iespējams izvēlēties informācijas detalizāciju un nerada lielu ietekmi uz sistēmas ātrdarbību.
4. Maģistra darba ietvaros tika izveidots apraksts resursu vadības sistēmas Horizon nekustamo īpašumu pārvaldības moduļa patēriņa starpības sadales koeficienta analīzes algoritmam un izveidota tā vizualizācija izmantojot blokshēmas. Izmantojot shēmas sistēmas lietotājs var manuāli atsekot analīzes darbībai un izdarīt secinājumus par iegūto rezultātu.
5. Patēriņa starpības sadales koeficienta analīzei izveidotas prioritātes, kurām algoritma blokshēmās atzīmēti starprezultātu izvades punkti. Pēc šādu punktu iestrādāšanas sistēmā, tās lietotājam būs iespēja dažādā detalizācijā aplūkot analīzes starppunktus, kas dos precīzāku ieskatu algoritma atlasītajos datos, veiktajās salīdzināšanās un iegūtajos rezultātos.
6. Maģistra darba ietvaros analizēti klientu pieteikumi, izmantojot potenciālus starprezultātu izvades žurnālu failus un prioritāšu mehānismu.

SECINĀJUMI

1. Iepazīstoties ar pieejamo literatūru par testēšanu un tās automatizāciju, tika izvirzīta problēma par analītisku algoritmu izpratni. Sistēmas lietotājam algoritma, kas izdod tikai rezultātu, fonā veicot datu atlasi un analīzi, izpratnei nepieciešama papildus informācija vai iespēja ieskatīties algoritma darbībā.
2. Strādājot ar analītiskiem algoritmiem sistēmās var rasties problēmas izprast to rezultātus un zust uzticība sistēmas darbībai. Sistēmas lietotājam ir svarīgi izprast analīzes gaitu, ja nepieciešams izskaidrot rezultātu. Sistēmas lietotājs var būt gan klients jeb pasūtītājs, gan klientu atbalsta speciālists, gan testētājs.
3. Lai aprakstītu analītiska algoritma prasības iespējams izmantot dabisko valodu vai vizualizācijas. Izpētot pieejamo literatūru autors secina, ka dabiskās valodas prasības vēlams papildināt ar vizuālizāciju shēmu veidā, lai vienādotu ieinteresēto pušu izpratni un samazinātu kognitīvās apstrādes slodzi.
4. Izveidojot algoritma aprakstu un blokshēmas, kā arī prioritātes un starprezultātu izvades punktus darba gaitā bija iespējams analizēt klienta izveidotus pieteikumus. Izpētot piemērus autors secina, ka šāda metode var palielināt izpratni par analīzes darbību un iegūtajiem rezultātiem.

IZMANTOTĀ LITERATŪRA UN AVOTI

[1] "Kārtība, kādā dzīvokļa īpašnieks daudzdzīvokļu dzīvojamā mājā norēķinās par pakalpojumiem, kas saistīti ar dzīvokļa īpašuma lietošanu", Latvijas vēstnesis, 197 (3981), 18.12.2008. [atsauce: 13.04.2017.] Pieejams: https://likumi.lv/doc.php?id=185342.

[2] "Programmatūras testēšanas rokasgrāmata vadītājiem", SQUALIO, 2014; - [atsauce 12.01.2017.]. Pieejams: http://odo.lv/ftp/docs/squalio_testesanas_rokasgramata.pdf.

[3] Sistēmas Horizon oficiālā mājaslapa, [atsauce 19.04.2017.] Pieejams: http://horizon.lv.

[4] R. Black, A. Claesson, G. Coleman, B. Cornanguer, I. Forgacs, A. Linetzki, T. Linz, L. Van der Aalst, M. Walsh, S. Weber, *Foundation Level Extension Syllabus Agile Tester*, ISTQB 2014, pp. 18-26

[5] S. Casner , J.H. Larkin, "Cognitive Efficiency Considerations for Good Graphic Design", *11th Annual Conf. Of the Cognitive Science Society*, August 1989.

[6] N. Dulac, T. Viguier, N. Leveson, M.A. Storey, "On the Use of Visualization in Formal Requitements Specification", *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*, 2002

[7] R.A. Gandhi, S.W Lee, "Visual Analytics for Requirements-driven Risk Assessment", *Requirements Engineering Visualization, 2007. REV 2007. Second International Workshop on*, 2007

[8] O.C.Z Gotel, F.T. Marchese, S.J. Morris, "On Requirements Visualization", *REV '07 Proceedings of the Second International Workshop on Requirements Engineering Visualization*, 2007

[9] R. Gray, "Automation versus Manual Testing", *SAE Technical Paper 660694*, 1966.

[10] J.A. Jones, M.J. Harrold, J. Stasko "Visualization of Test Information to Assist Fault Localization", *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, 2002

[11] T. Koomen, M. Pol, *Test Process Improvement: A Practical Step by Step Guide to Structured Testing*, Addison-Wesley, 1999

[12] J.H. Larkin, H.A. Simon, "Why a Diagram is (Sometimes) Worth Ten Thousand Words", *Cognitive Science, 11(1):65-99*, Jan-Mar 1987.

[13] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*, Addison-Wesley, 2007.

[14] A. Spillner, T. Linz, H. Schaefer, *Software testing foundations*, dpunkt.verlag, 2006.

[15] L. Williams, E.M. Maximilien, M. Vouk, “Test-driven development as a defect-reduction practice”, *Software Reliability Engineering 14th International Symposium (ISSRE 2003), IEEE*, 2003.

[16] J.Bičevskis, J.Borzovs. “*Prioriteti v otladke boļših programmnih sistem*.” *Programmiro vaņije, No. 3*, 1982, lpp. 31-34.

Maģistra darbs **„ Prioritāšu mehānisma izmantošana analītisku algoritmu testēšanā”** izstrādāts LU Datorikas fakultātē.

Darba teksta galīgā versija izgatavota 22.05.2017.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: _____.

(Autora paraksts un datums)

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 un datums)

Darbs iesniegts **maģistrantūras sekretariātā** _____.

(Iesniegšanas datums)

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

Studiju metodiķe: _____.

(Metodiķes paraksts)

Recenzents: __Profesors, Dr.dat. Guntis Arnicāns____

(Akad.amats, zin.grāds, vārds, uzvārds)

Darbs aizstāvēts maģistra gala pārbaudījuma komisijas sēdē

_____ prot. Nr. _____

(Darba aizstāvēšanas datums)

Komisijas sekretārs: _____

(Sekretāra paraksts)