LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

# ALGU APRĒĶINA MODUĻA AUTOMATIZĒTĀ TESTĒŠANA

MAĢISTRA DARBS

Autors: **Kristīne Tanne**
Stud. apl. Nr. kt12022
Darba vadītājs: Dr. dat. Dainis Dosbergs

RĪGA 2018

# ANOTĀCIJA, ATSLĒGVĀRDI

Ikvienai sistēmai ir nepieciešama testēšana, lai pārliecinātos, ka sistēma strādā, kā nepieciešams. Bieži testētājiem daudz laiku aizņem vienu un to pašu testu regulāra izpilde pēc izmaiņu veikšanas. Šādu testu automatizēšana atbrīvo testētāju laiku citu svarīgāku testu veikšanai.

Maģistra darba mērķis ir izpētīt automatizēto testēšanu un metodes, izanalizēt apskatīto informāciju, izvēlēties atbilstošākās automatizēto testu izstrādes metodes un uz to pamata izstrādāt automatizētos testus Algu aprēķina modulī.

Darbā ir aprakstīts testēšanas jēdziens, automatizētās testēšanas principi, līmeņi un ietvari, un apskatīta un salīdzināta programmatūra automatizēto lietotāja saskarnes un datu bāzu testu izstrādei. Ir apskatīts Algu aprēķina un tam un saistītie moduļi, un to integrācija ar pārējām SIA ZZ Dats sistēmām. Balstoties uz sistēmā izmantotajām tehnoloģijām un rīkiem, ir veikta rīku un metožu izvēle. Vadoties pēc izvēlētajām metodēm un rīkiem, ir izstrādāti lietotāja saskarnes un vienību testi sistēmā. Izmantojot izstrādātos testus ir iespējams regulāri pārbaudīt vai sistēma funkcionē kā nepieciešams un nav radušās kļūdas tās darbībā.

**Atslēgvārdi:** informācijas sistēma, testēšana, testu automatizēšana, algu aprēķins, lietotāja saskarnes testi, vienību testi

# ABSTRACT, KEYWORDS

*Automated testing of Salary calculation module*

Every system requires testing to make sure that the system works as intended. Often, a lot of testers' time is being used on performing regular, repeated tests after each change in a system. The automation of such tests increase testers' available time for other more important tests.

The goal of the Master's Thesis is to study automated testing and methods, to analyze the studied information, to choose the most appropriate automated test development methods, and to develop automated tests based on these methods in the Salary calculation module.

In the Master's Thesis the author has described the concept of testing, automated testing principles, levels and frameworks, as well as described and compared software for developing automated user interface and database tests. Author has described the Salary calculation module and its related modules, and their integration with other SIA ZZ Dats systems. Based on the technology and tools used in the system, a choice of tools and methods has been made. Based on the chosen tools and methods, the author has developed user interface and unit tests in the Salary calculation module. By using the developed tests, it is possible to regularly check whether the system is functioning properly and if there are any errors in its operation.

**Keywords:** information system, testing, test automation, salary calculation, user interface tests, unit tests

# AUTOREFERĀTS

Darbs ir balstīts uz praktiska procesa veikšanu, izstrādājot automatizētos testus darba laika uzskaites funkcionalitātē SIA ZZ Dats (turpmāk tekstā ZZ Dats) Algu aprēķina moduļa ietvaros. Algu moduļa darba laika uzskaite ir uz tīmekļa vietnes balstīta autora izstrādāta funkcionalitāte ar sarežģītiem algoritmiem, un tā ir integrēta ar vairākām citām sistēmām, tādēļ ir svarīgi, lai tajā nebūtu kļūdu un sistēma izpildītu darba laika aprēķinus pareizi.

Darbs dod pārskatu par testu automatizēšanas procesu, darbā ir aprakstīta testu izstrādes procesa gaita, un tiek izskatītas galvenās problēmas un to risinājumi, izstrādājot automatizētus testus. Maģistra darbā ir veikts arī teorētiskais pētījums par testēšanu, automatizēto testēšanu un automatizētās testēšanas rīkiem tīmekļa vietņu un datu bāzes vienību testēšanai, uz kā pamata tika veikti lēmumi rīku izvēlē un testu izstrādē.

Lai sasniegtu izvirzītos mērķus, tika veikti sekojoši uzdevumi:

- apskatīts testēšanas jēdziens;
- apskatīts automatizētās testēšanas jēdziens, aprakstot testu automatizācijas līmeņus un ietvarus, uz kuriem ir iespējams balstīt automatizētos testus;
- aprakstīti un izvērtēti rīki lietotāja saskarnes un datu bāzes automatizēto testu izstrādei, salīdzinot apskatītos variantus;
- aprakstīta ZZ Dats izstrādātā Vienotā pašvaldību sistēma, Resursu vadības sistēma G-VEDIS, Algu aprēķina modulis un darba laika uzskaites funkcionalitāte, to sasaiste un izmantotās tehnoloģijas un rīki;
- aprakstīts Algu aprēķina moduļa versijas dzīves cikls, aprakstot arī testēšanas procesu un aktuālās problēmas;
- izvērtēti apskatītie rīki un izvēlēti atbilstošākie automatizēto testu izstrādei Algu aprēķina moduļa darba laika uzskaites funkcionalitātē;
- veikta lietotāja saskarnes automatizēto testu izstrāde, izmantojot rīkus “Microsoft Visual Studio” un “Selenium”;
- veikta datu bāzes automatizēto vienību testu izstrāde, izmantojot rīkus “Oracle PL/SQL Developer” un “utPLSQL”;
- izstrādātie datu bāzes testi integrēti ar automatizēto darbu izpildes sistēmu *Jenkins*, īstenojot regulāru testu izpildīšanu ik dienu, rezultātu piereģistrēšanu un e-pastu izsūtīšanu darba autoram neveiksmīgu testu gadījumos.

Darba rezultātā ir nodrošināti automatizētie testi Algu aprēķina moduļa darba laika uzskaites funkcionalitātē, kas tiek izpildīti regulāri un programmētājiem palīdz laicīgi konstatēt un novērst problēmas pirms tās sasniedz testētājus vai lietotājus.

Risinājums ir sasniedzis darbā izvirzītos mērķus, jo ir analizēta informācija par testēšanu un automatizēto testēšanu, apskatīta Algu aprēķina sistēmas darba laika uzskaites funkcionalitāte un tās sasaiste ar citiem moduļiem, uz apskatītās informācijas pamata pieņemti lēmumi par konkrētu rīku izmantošanu testu izstrādē, un veikta testu izstrāde. Izstrādātie automatizētie testi tiek regulāri izpildīti, pārbaudot katru darba laika uzskaites funkcionalitātes aspektu, un brīdina programmētājus par kļūdām gan sistēmas lietotāja saskarnes darbībā, gan algoritmu darbībā. Gadījumos, kad izpildītie automatizētie testi ir konstatējuši kļūdas sistēmas darbībā, ko programmētāji nav pamanījuši, pēc testu rezultātu noskaidrošanas programmētājiem tiek paziņota kļūdainā darbība, un to ir iespējams izlabot īsā laika periodā.

# SATURA RĀDĪTĀJS

# ATTĒLU SARAKSTS

# APZĪMĒJUMU SARAKSTS

**ASP.NET** – uzņēmuma *Microsoft* izstrādāts vienots tīmekļa izstrādes modulis [TTASP]

**C# -** uzņēmuma *Microsoft* izstrādāta programmēšanas valoda

**CSS** – kaskadētas stila lapas

**DBPS Oracle** – datu bāzu pārvaldības sistēma Oracle. *Oracle Database* ir relāciju datubāzes pārvaldības sistēma, kuru izstrādā *Oracle Corporation* [WIKOD]

**Firefox** – uzņēmuma *Mozilla* izstrādāta tīmekļa pārlūkprogramma

**HTML** – hiperteksta iezīmēšanas valoda

**Java –** vairāku platformu programmēšanas valoda

**JavaScript -** firmas *Netscape* izveidota valoda, kas ļauj globālā tīmekļa izstrādātājiem veidot interaktīvas vietnes. *JavaScript* var sadarboties ar valodas *HTML* pirmkoda programmām, tādējādi ļaujot globālā tīmekļa izstrādātājiem piepildīt šis vietnes ar dinamisku saturu [LZATE]

**Jenkins -** nepārtrauktās integrācijas sistēma, kas nodrošina programmatūras koda kompilēšanu, būvēšanu un testēšanu [ODORJ]

**Kursors datu bāzē** – kontroles struktūra, kurā var ielasīt vairākus tabulas ierakstus un iet tiem cauri [WIKCD]

**Microsoft Visual Studio** – integrētā izstrādes vide, kas ir paredzēta datora programmatūras, mājas lapu un lietotņu izstrādei [WIMVS]

**MS SQL Server** – uzņēmuma *Microsoft* izstrādāta relāciju datu bāzu vadības sistēma

**MVC (Model-View-Controller)** – satvars tīmekļa lietotņu būvei, kas iedalās 3 komponentēs – modelis, skats, kontrolieris, kuri savā starpā sazinās viens ar otru [MVCW3]

**Oracle SQL Developer** – uzņēmuma *Oracle* izstrādāta integrēta izstrādes vide darbam ar *SQL Oracle* datu bāzēm [WIOSD]

**PHP -** atklātā pirmkoda skriptu valoda, kura sākotnēji bija paredzēta servera puses lietojumos dinamiska tīmekļa lapu ģenerēšanai [WIPHP]

**PL/SQL Developer** – uzņēmuma *Allround Automations* izstrādāta integrētā izstrādes vide [AAAPS]

**RVS G-VEDIS** – SIA ZZ Dats (turpmāk tekstā ZZ Dats) izstrādāta resursu vadības sistēma

**Selenium** – pārnesams programmatūras testēšanas ietvars tīmekļa lietotnēm [WISEL]

**Sinonīmi (datu bāzē)** – tabulas, skata vai cita objekta aizstājvārds [WIKSD]

**utPLSQL spraudnis** – universāla atvērtā koda vienību testēšanas sistēma, kas paredzēta *Oracle PL/SQL* datu bāzēm [UTPLS]

**Visual FoxPro** – uzņēmuma *Microsoft* izstrādāta objektorientētās programmēšanas vide, kas paredzēta *Windows* datoru lietojumprogrammatūras izveidei [VFPST]

**VPS (Vienotā Pašvaldību Sistēma)** – ZZ Dats izstrādāts savstarpēji integrēts informācijas tehnoloģiju risinājums pašvaldību biznesa procesu atbalstam [ZZRIS].

# IEVADS

Pēdējos gados programmatūras izstrādē arvien aktuālāka kļūst testēšanas automatizācija. Programmatūrai kļūstot apjomīgākai un palielinoties tās funkcionalitātei, paildzinās testēšanā pavadītais laiks, jo pēc katrām izmaiņām ir ne tikai jātestē labotā vieta, bet arī jāatkārto vieni un tie paši testi, pārbaudot vai iepriekš strādājusī funkcionalitāte vēl joprojām strādā, kā paredzēts. Šādi testi testētājiem aizkavē laiku citu, svarīgāku testu veikšanai. Rodas nepieciešamība pēc regulāru un atkārtojami izpildāmu testu automatizācijas.

Ikvienai sistēmai ir nepieciešama testēšana, lai pārliecinātos, ka sistēma atbilst prasībām un ka tā nesatur kļūdas. Automatizējot testus un tos regulāri izpildot, ir iespējams laicīgi atrast kļūdas sistēmā un tās novērst, pirms tās ir sasniegušas testētājus vai pat klientus. Testu automatizācija nozīmē speciālu vai pašu izstrādātu rīku izmantošana iepriekš izstrādātu testu izpildīšanai un to atgriezto rezultātu salīdzināšanai ar sagaidītajiem.

Darba autors strādā pie Algu aprēķina moduļa (tīmekļa lietotnes) izstrādes un meklē risinājumus automatizēto testu izstrādei un ieviešanai moduļa darba laika uzskaites tabulu funkcionalitātē. Ieviešot automatizētos testus sistēmā, tiks nodrošināta pārliecība, ka sistēmā nav problēmu, un ka sistēmas pamata darbību disfunkcija netraucēs testētājiem veikt savu darbu, jo atrastās kļūdas būs iespējams novērst laicīgi. Algu aprēķina modulis kā datu bāzes vadības sistēmu izmanto *Oracle*, bet lietotāji informācijai piekļūst caur lietotāja saskarni tīmekļa lietotnē. Tas nozīmē, ka galvenokārt vislielākā uzmanība tiek pievērsta tīmekļa lietotņu automatizētajai testēšanai, un datoros instalējamu programmatūru automatizēta testēšana darbā aprakstīta tiks minimāli.

Darba mērķis ir apskatīt testēšanas un automatizētās testēšanas teoriju un galvenās vadlīnijas, apskatīt un analizēt dažas populārākās testu automatizēšanas iespējas un to atbilstību testējamajai sistēmai, un, balstoties uz analīzi, izstrādāt automatizētos testus Algu aprēķina moduļa darba laika uzskaites funkcionalitātē. Apskatot Algu aprēķina moduli, darba laika uzskaites funkcionalitāti, to saistītās sistēmas un moduļus, un to integrāciju un izmantotās tehnoloģijas un rīkus, autors izvēlas atbilstošākos testu automatizācijas rīkus un tehnikas, un tās pielieto darba laika uzskaites funkcionalitātes automatizēto testu izstrādē.

Darbs ir sadalīts četrās daļās:

1. daļā tiek apskatīts testēšanas jēdziens, testēšanas stratēģijas un līmeņi.

2. daļā tiek izpētīta automatizētā testēšana – apskatīti testu automatizācijas līmeņi, ietvari, uz kuru pamata var veikt automatizēto testēšanu, aprakstīti automatizētās testēšanas pamatprincipi, priekšrocības un trūkumi, kā arī aprakstīts automatizētās testēšanas dzīves cikls.

Daļas beigās tiek apskatītas automatizēto testu izveides iespējas lietotāja saskarnē un datu bāzē, salīdzinot tās.

3. daļā ir aprakstīta Vienotā pašvaldību sistēma, resursu vadības sistēma G-VEDIS, Algu aprēķina modulis, darba laika uzskaites tabulu funkcionalitāte un visu šo sistēmu sasaiste. Ir aprakstīts Algu aprēķina moduļa versijas izstrādes dzīves cikls un galvenās problēmas ar kurām saskarās testētājs, veicot manuālo testēšanu. Daļā ir apskatītas sistēmu izmantotās tehnoloģijas un rīki, uz kuriem balstoties tiek izvēlētas atbilstošākās no darba 2. daļā apskatītajām automatizēto testu izveides iespējām, kuras tiks realizētas.

4. daļā ir aprakstīti darba autora izstrādātie automatizētie testi lietotāja saskarnes testēšanai un datu bāzu vienību testēšanai, izmantojot darba 3. daļā izvēlētās testēšanas metodes un rīkus.

Aiz darba pirmās, otrās un trešās daļas ir kopsavilkums, kas apkopo tās galvenos punktus.

Darba beigās rezultātos ir apkopots viss autora paveiktais, savukārt secinājumos apkopots visās daļās aprakstītais.

# 1. TESTĒŠANAS JĒDZIENS

Testēšana ir programmatūras un aparatūras darbības pārbaude, izmantojot testdatus. Testēšanas mērķis var būt defekta atklāšana, tā atrašanās vietas lokalizēšana vai arī testējamā objekta dinamisko parametru (piemēram, ātrdarbības) noskaidrošana [ETVMR], kā arī testēšanu veic, lai validētu vai apliecinātu, ka programma atbilst biznesa un tehniskajām prasībām, un strādā, kā paredzēts [ISWIS].

## 1.1. Testēšanas stratēģijas

Trīs galvenās testēšanas stratēģijas ir melnās kastes testēšana, baltās kastes testēšana un pelēkās kastes testēšana, kas tiek izmantotas programmatūras izstrādē [TCBWT].

### 1.1.1. Melnā kaste

Funkcionālajā testēšanā, jeb melnās kastes testēšanā tiek testētas funkcijas ar ieejas datiem un tiek pārbaudīts to izejas rezultāts. Tiek salīdzināta programmas uzvedība pret specifikāciju bez zināšanām par koda uzbūvi [HLGAT]. Melnās kastes testos tiek izmantota programmatūras lietotāja saskarne. Pat tad, kad ir veikti labojumi programmatūras iekšienē, testiem būtu jābūt veiksmīgiem, ja vien nav mainīta programmatūras funkcionalitāte prasību līmenī. Testētājs zina, ko programmai ir jādara, tomēr nezina, kā programma to dara [TCBWT].

Melnās kastes testēšanas priekšrocības ir [TCBWT]:

a) efektivitāte lieliem koda apgabaliem,
b) nav nepieciešama piekļuve kodam,
c) testētāji un programmētāji strādā viens no otra neatkarīgi un testēšana nav objektīva.

Melnās kastes testēšanas trūkumi ir [TCBWT]:

a) ierobežots pārklājums, jo tiek izpildīta tikai ierobežota testu scenāriju kopa,
b) neefektīva testēšana dēļ testētāja ierobežotajām zināšanām par programmatūras iekšējo darbību.

### 1.1.2. Baltā kaste

Strukturālā testēšana, jeb baltās kastes testēšana pievēršas programmas ceļu izvēlei un atbilstošo testa piemēru kopas veidošanai. Tiek salīdzināta programmas uzvedība attiecībā pret kodā paredzēto [HLGAT]. Baltās kastes testētāji zina koda uzbūvi un izmanto šīs zināšanas

testu veikšanai. Lai veiktu baltās kastes testēšanu, ir nepieciešamas zināšanas par programmēšanas valodu, kurā programma rakstīta [TCBWT].

Baltās kastes testēšanas priekšrocības ir [TCBWT]:

a) efektīvs veids kļūdu un problēmu atrašanai,
b) ļauj atrast apslēptās kļūdas,
c) palīdz optimizēt kodu,
d) dēļ nepieciešamajām zināšanām par programmatūras iekšējo darbību tiek iegūts maksimāls programmatūras funkcionalitātes pārklājums.

Baltās kastes testēšanas trūkumi ir [TCBWT]:

a) var neatrast nerealizētu vai trūkstošu funkcionalitāti,
b) ir nepieciešamas augstas zināšanas par programmatūras koda uzbūvi un valodu
c) ir nepieciešama piekļuve kodam.

### 1.1.3. Pelēkā kaste

Pelēkās kastes testēšana ir baltās un melnās kastes testēšanas apvienojums. Veicot pelēkās kastes testēšanu, programmatūras iekšējā struktūra ir daļēji zināma. Tas nozīmē, ka ir piekļuve iekšējai datu struktūrai un algoritmiem, lai veidotu testpiemērus, bet testēšana notiek lietotāja pusē izmantojot melnās kastes testēšanu. Pelēkās kastes testēšanu var izmantot dažādos testēšanas līmeņos, bet visbiežāk to izmanto integrācijas testēšanā [STFGB].

## 1.2. Testēšanas līmeņi

Testēšana iedalās četros lielos līmeņos: vienību testēšana, integrācijas testēšana, sistēmas testēšana un akcepttestēšana [RDBTD]. Programmatūras izstrādes ciklā visi šie testēšanas veidi ir jāizpilda secīgi viens pēc otra [TSTIT].

### 1.2.1. Vienību testēšana (Moduļu testēšana)

Vienība ir vismazākā lietotnes daļa (piemēram, funkcijas, klases, procedūras). Vienību testēšana ir testēšanas veids, ar kuru tiek testētas individuālas vienības, koda rindiņas, lai noskaidrotu vai tie ir atbilstoši lietošanai, un lai parādītu, ka katra individuāla komponente ir atbilstoša prasībām un paredzētajai funkcionalitātei. Vienību testus parasti sastāda un izpilda lietojumprogrammatūras izstrādātāji, lai pārbaudītu kodu pirms programmas vienības tiek apvienotas kopīgā funkcionalitātē. Pēc šīs apvienošanas seko integrācijas testēšana, kad tiek pārbaudīta šo atsevišķo vienību vai moduļu sadarbība [TSTUT, RDBTD].

Priekšrocības vienību testēšanas veikšanā ir problēmu atrašana agrīnā programmatūras stāvoklī, kas minimizē programmatūras izstrādes riskus, kā arī minimizē izmaksas programmatūras kļūdu labošanai, kas atklātas tad, kad programma ir gandrīz pabeigta [RDBTD]. Ja tiek sastādīti kvalitatīvi vienību testi un tie tiek izpildīti ikreiz, kad ir veiktas kādas izmaiņas kodā, tas nodrošina mazāku kļūdu skaitu programmatūrā [TSTUT].

Piemērs vienību testēšanai reālajā dzīvē: katra automašīnas komponente tiek izstrādāta individuāli (sēdekļi, spoguļi, motors, bremzes). Pēc izstrādes tiek pārbaudīts vai katra komponente individuāli strādā, kā nepieciešams [TSTST].

### 1.2.2. Integrācijas testēšana

Integrācijas testēšana nozīmē vairāku individuālu programmatūras moduļu testēšana kopā, lai pārbaudītu vai tie savā starpā strādā un nerada kļūdas. Daudzas sistēmas izstrādā vairāki izstrādātāji, un to izpratne un programmēšanas loģika var atšķirties vienam no otra. Ja programmatūras moduļus izstrādā vairāki izstrādātāji, tad integrācijas testēšana kļūst par vajadzību [TSTIT, RDBTD]. Var būt gadījumi, kad katram modulim vienību testēšana izpildās veiksmīgi, tomēr pēc moduļu apvienošanas rodas kļūdas, kad tie savā starpā mijiedarbojas.

Ir četri integrācijas testēšanas veidi [TSTIT]:

- lielā sprādziena testēšanas pieeja: visas vienības tiek sakombinētas, veidojot pilnīgu programmatūras sistēmu vai lielu tās daļu, un tas tiek izmantots integrācijas testēšanā;
- lejupēja testēšana: tiek testēti augšējie integrētie moduļi, kuros integrē zemākus moduļus vienu pa vienam. Procesu atkārto līdz visi moduļi ir integrēti un notestēti;
- augšupēja testēšana: pirmās testē zemāka līmeņa komponentes, pēc tam tās tiek izmantotas, lai atvieglotu augstākā līmeņa komponentu testēšanu. Procesu atkārto līdz hierarhijā esošās augšējās komponentes ir notestētas;
- sviestmaizes testēšana: pieeja kombinē lejupējo un augšupējo testēšanu.

Piemērs integrācijas testēšanai reālajā dzīvē: kad katra automašīnas daļa ir samontēta ar citu daļu, šī saliktā kombinācija tiek pārbaudīta – vai montāžā nav radušās nekādas blakusparādības un vai visas sastāvdaļas darbojas kopā, kā paredzēts [TSTST].

### 1.2.3. Sistēmas testēšana

Sistēmas testēšana nozīmē tieši to, kādu ideju nosaukums rada – tiek testēta visa sistēma kā kopums, lai pārbaudītu, ka kopējais produkts atbilst noteiktajām prasībām [RDBTD].

Sistēmu testēšanu parasti veic komanda, kas ir neatkarīga no izstrādes komandas, lai novērtētu sistēmas kvalitāti objektīvi [TSTST].

Piemērs sistēmas testēšanai reālajā dzīvē: pēc tam, kad visas detaļas ir saliktas un automašīna ir gatava, ir jāpārbauda dažādi aspekti, piemēram, vienmērīga braukšana, pārtraukumi, pārnesumi, nobraukums un automašīnas mūžs [TSTST].

### 1.2.4. Akcepttestēšana

Akcepttestēšana ir testēšanas process, kur tiek pārbaudīts vai produktam tiek dota *zaļā gaisma* vai nē. Šīs testēšanas mērķis ir novērtēt vai sistēma atbilst galalietotāju prasībām un vai tā ir gatava izmantošanai [TSTUA]. Veicot akcepttestēšanu, testēšanas komanda var noskaidrot, kā produkts darbosies tad, kad tas būs instalēts lietotāja sistēmā [RDBTD].

Akcepttestēšanai ir divi veidi – alfa testēšana un beta testēšana. Alfa testēšanu veic uz vietas, kas nozīmē, ka testēšanas komandā ir iekļauti arī izstrādātāji un biznesa analītiķi. Savukārt beta testēšanu veic klienta pusē, un to veic reāli klienti, kas nozīmē, ka izstrādātāji un biznesa analītiķi procesā netiek iekļauti [TSTUA].

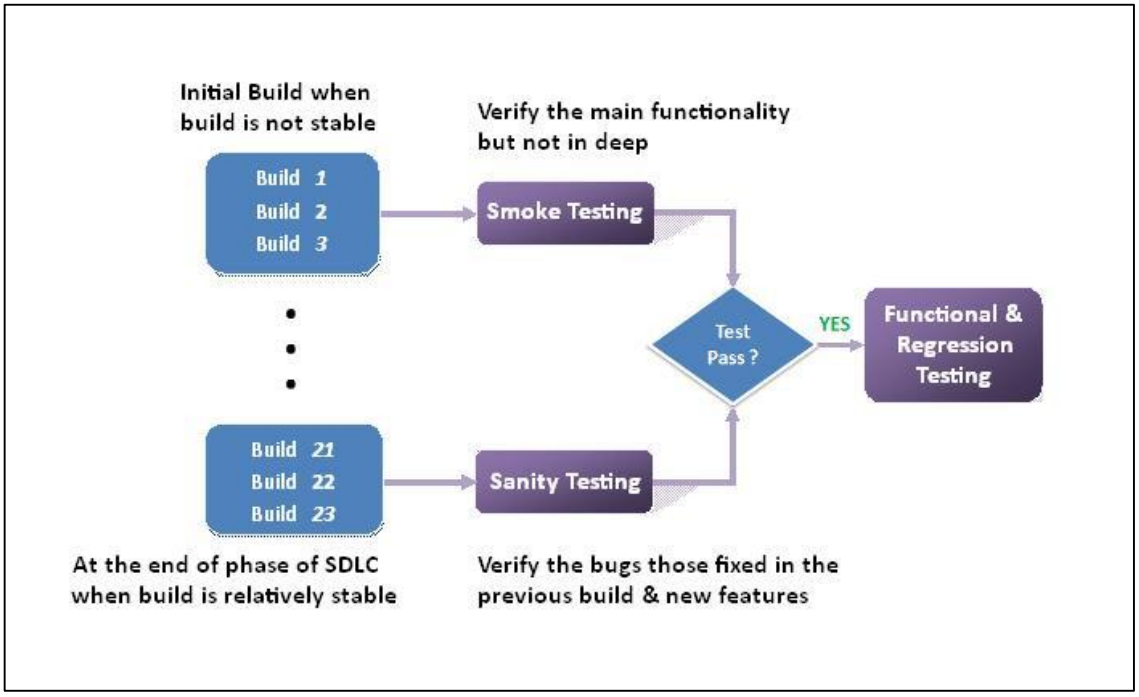
### 1.2.5. Citi testēšanas veidi

Papildus četriem aprakstītajiem testēšanas veidiem (līmeņiem), ir vēl dažādi citi testēšanas veidi, kas ir mazāk obligāti. Tos var novietot zem kāda no četriem līmeņiem, vai arī starp tiem. Lielākie no atlikušajiem testēšanas veidiem ir dūmu (Smoke) testēšana, veselības (Sanity) testēšana un regresijas testēšana. Tomēr ir arī tādi testēšanas veidi kā stresa testēšana, lietojamības testēšana un veiktspējas testēšana, kas nepārbauda konkrēti vai programmatūra spēj izpildīt nepieciešamās darbības, bet gan vai tā ir ērti lietojama, kā arī vai tā atbilst nefunkcionālajām prasībām (piemēram, spēj strādāt ar 1000 paralēliem lietotājiem, spēj izpildīt 100 darbības vienlaicīgi, u.t.t.).

**Dūmu (Smoke) testēšana**

Dūmu testēšana ir paredzēta, lai noskaidrotu vai programmatūra ir stabila un ir gatava tālākai testēšanai. Dūmu testēšanu veic pirms jebkādas citas testēšanas, lai pārliecinātos, ka programmatūras kritiskā funkcionalitāte strādā korekti. Šī testēšana pārbauda programmatūras darbību virspusēji, tas nozīmē, ka netiek testētas sīka un dziļa funkcionalitāte, bet gan tikai pamata darbības. Dūmu testi var pārbaudīt sekojošus aspektus – vai lietotājs var piekļūt programmatūrai, vai lietotājs var navigēt no viena loga uz otru, u.c. vienkārša līmeņa funkcionalitātes. Dūmu testos arī tiek pārbaudīts vai grafiskā lietotāja saskarne reaģē, kā nepieciešams. Dūmu testus veic sākotnējās programmatūras būvējuma stadijās, kad tā līdz

galam nav stabila un kad funkcionalitāti ir paredzēts krietni papildināt vai labot (skat. 1.1. att.) [STCSM].




*1.1. att.* **Dūmu un veselības testēšanas izpildes procesa diagramma [STCSM]**

Dūmu testi palīdz atrast moduļu integrācijā ieviestās problēmas, palīdz atrast problēmas agrīnā testēšanas fāzē, kā arī palīdz testētājiem iegūt pārliecību, ka nesenie labojumi sistēmā nav sabojājuši svarīgu sistēmas funkcionalitāti [STCSM].

**Veselības (Sanity) testēšana**

Tad, kad testētājam tiek nodota programmatūra ar nelieliem kļūdu labojumiem kodā vai funkcionalitātē, tiek veikta veselības testēšana, lai pārbaudītu vai ir salabotas kļūdas, kas bija pieteiktas iepriekšējās programmatūras versijās, un vai nav salauzta iepriekš strādājusī funkcionalitāte. Veselības testēšanas mērķis ir pārbaudīt vai plānotā funkcionalitāte strādā tā, kā paredzēts. Veselības testēšana palīdz izvairīties no liekas laika un izmaksu izšķēršanas, ja būvējums ir neveiksmīgs. Testētājam būtu jānoraida būvējumu, ja tajā tiek atrastas kļūdas, veicot veselības testēšanu [STCSA].

Veselības testēšanu veic pēc regresijas testēšanas, lai pārliecinātos, ka labojumi un izmaiņas programmatūrā nav salauzušas pamata funkcionalitāti. Parasti veselības testēšana tiek veikta sistēmas izstrādes dzīves cikla beigās (skat. 1.1. att.), t.i., pirms programmatūras izlaišanas [STCSA].

**Regresijas testēšana**

Regresijas testēšana ir jau iepriekš testētas sistēmas atkārtota testēšana pēc modifikāciju veikšanas sistēmā. To ir nepieciešams veikt, lai pārbaudītu vai atrastās kļūdas ir novērstas, kā arī lai pārliecinātos, ka sistēmā veiktās izmaiņas nav radījušas defektus sistēmas nemainītajā daļā. Katrai klāt pievienotai vai modificētai programmatūras daļai tiek veidoti jauni vai tiek laboti esošie regresijas testi. Katrai no jauna atrastajai kļūdai tiek izveidots tests, kuru izpildot var pārliecināties, ka kļūda ir novērsta un nav atkārtoti radusies. Veicot regresijas testēšanu, tiek darbināti bāzes testi, ar kuriem pārliecinās, ka pēc kļūdu labošanas programmatūra funkcionē tieši tāpat, kā iepriekš. Regresijas testēšanu izmanto, kad sistēmai pievieno jaunu funkcionalitāti, veic labojumus esošajā funkcionalitātē, labo kļūdas, kā arī ja ir salabota kāda ar veiktspēju saistīta kļūda. Ja regresijas testēšana tiek veikta ar roku, nevis izmantojot automatizācijas rīkus, tad tas ir ļoti laikietilpīgs darbs, jo ir nepieciešams atkārtot vienus un tos pašus testus atkal un atkal pēc katra labojuma. Regresijas testēšanu ir nepieciešams veikt pat tad, kad ir veiktas ļoti nelielas izmaiņas kodā, jo šīs izmaiņas var radīt neparedzētas kļūdas eksistējošā funkcionalitātē [IECRT].

**Lietojamības testēšana**

Lietojamības testēšana tiek veikta no klienta perspektīvas, lai novērtētu, cik lietotāja saskarne ir lietotājdraudzīga. Testi nosaka, cik viegli lietotājs mācās rīkoties ar sistēmu, cik labi lietotājs individuāli spēj veikt nepieciešamās darbības pēc apmācībām, cik pievilcīgs ir dizains izmantošanai [CPWIS].

**Veiktspējas testēšana**

Veiktspējas testi pārbauda sistēmas ātrumu un efektivitāti, un pārliecinās, ka sistēma izdod pieprasītos rezultātus noteiktā laikā, kā noteikts prasībās. Veiktspējas testēšana iedalās slodzes testēšanā un stresa testēšanā [CPWIS, ODOTR].

Stresa testēšana pārbauda, kā programmatūra uzvedās nelabvēlīgos, ekstremālos apstākļos. Testēšana notiek ārpus specifikāciju robežām [CPWIS]. Ja stresa testēšanas laikā slodze ir ilgstoša, tad sistēma var palikt tik lēna, ka to var uzskatīt kā apstājušos, kā arī sistēma var apstāties pavisam, ja tiek pārsniegts kāds no sistēmas fiziskajiem ierobežojumiem, piemēram, operatīvās atmiņas izmērs [ODOTR].

Slodzes testēšana atkarībā no stresa testēšanas ir pozitīvs tests, kurā nosaka, ar cik lielu slodzi sistēma spēj tikt galā. Izpildot slodzes testu, vienmērīgi tiek palielināta sistēmas slodze un tiek vērots, kā izmainās veiktspējas rādītāji – sistēmas slodze, izmantotais procesora laiks un atmiņa, un servera atgrieztā satura laiks [ODOTR].

## 1.3. Kopsavilkums

Testēšana ir nepieciešama ikvienā projektā, pat vismazākajos, jo bez testēšanas nav iespējams pārliecināties par programmas pareizumu. Testēšana nozīmē ne tikai pārbaudi vai programmatūrā nav kļūdu, bet arī pārbaudi vai programmatūra visas prasītās darbības veic korekti. Nefunkcionālā testēšana testē tādas lietas, kā ātrdarbība, veiktspēja, lietojamība, un citus svarīgus, bet ne kritiskus programmatūras aspektus. Vienību testēšana un dūmu testēšana palīdz atrast kļūdas programmatūras agrīnā stāvoklī, un pateicoties tām, var samazināt kļūdu labošanas izmaksas, jo kļūda, kas atrasta testēšanas pēdējā stadijā vai no klienta, prasīs daudz vairāk laiku novēršanai.

# 2. AUTOMATIZĒTĀ TESTĒŠANA

Programmatūras testēšanā testu automatizēšana ir specializētu programmatūru, jeb rīku izmantošana, kas ir nošķirti no programmatūras, ko testē. Automatizētā testēšana kontrolē testu izpildi un salīdzina testu rezultātus ar paredzētajiem rezultātiem [KAHDA]. Automatizēto testu izpildīšana nevar aizvietot manuālo testēšanu, tomēr tā var manāmi uzlabot testēšanas kvalitāti, kā arī samazināt resursu izmantojumu testēšanas nolūkos. Ir testi, kurus būtu ļoti apgrūtinoši veikt, izmantojot automātiskos testus (pārbaudes, kur izmantojams cilvēka prāts un viedoklis), kā arī ir testi, kurus daudz vienkāršāk ir veikt, izmantojot automatizēto testēšanu (piemēram, liela lietotāju daudzuma nosimulēšana programmā).

Automatizējot testus, ir svarīgi izvēlēties programmatūru (rīkus), kurai var piekļūt, un ar ko spēj darboties visi iesaistītie dalībnieki, kas ietver manuālos testētājus un izstrādātājus [SBLAW].

## 2.1. Automatizētās testēšanas pamatprincipi

Veicot automatizēto testēšanu, ir iespējams saskarties ar dažādām problēmām, no kurām ir iespējams izvairīties, ja testēšana tiek veikta, balstoties uz automatizētās testēšanas pamatprincipiem, jeb labo praksi, kas uzskaitīti zemāk.

**Uzturēt testus vienkāršus un īsus**

Lielākoties tad, kad tiek veidoti vienību testi, tie testē vienu funkciju. Tomēr bieži vien cilvēki to aizmirst un pievieno vairākus vienas funkcijas izsaukumus, katru reizi padodot citādākas ieejas datu kombinācijas, tādā veidā vienā testā testējot dažādus testpiemērus. Tādā gadījumā, ja kāds no izsaukumiem izpildās neveiksmīgi, pārējie izsaukumi netiek izpildīti, kā rezultātā testa rezultāti nesniedz precīzu informāciju par visām iespējamajām nepilnībām (neveiksmīgiem izsaukumiem), kas iespējami. Tādēļ testus ir nepieciešams veidot pēc iespējas mazākus. Jo garāks ir tests, jo trauslāks tas kļūst. Galvenais iemesls šādai smagai testu veidošanai ir tāds, ka testētāji, veidojot testus, cenšas aptvert pilnu lietošanas piemēru, kā to darītu manuālajā testēšanā [QATAP].

**Uzturēt testus neatkarīgus**

Nevienam testam nevajadzētu būt atkarīgam no cita testa. Šim noteikumam nav pārāk grūti sekot, tomēr šī ir viena no vispieļautākajām problēmām testētāju vidū. Tad, kad tiek veidots lietošanas piemēra tests, bieži vien katrs solis tajā tiek rakstīts kā atsevišķs tests, kā rezultātā viens tests ir atkarīgs no cita testa (atvērta forma, mainīti dati, u.c.). Problēma šajā ir

tāda, ka katru izveidoto testu ir jāvar izpildīt individuāli. Tad, kad testi tiek palaisti, parasti viena testa neveiksmīgas izpildes gadījumā tiek sākts automātiski pildīt nākamo testu, izlaižot neveiksmīgi izpildīto, vai arī tests tiek pārtraukts un dati tiek atstāti tādi, kādi tie bija neveiksmīgās testa izpildes brīdī [QATAP].

**Veidot idempotentus testus**

Ja testu var izpildīt atkārtoti vairākas reizes pēc kārtas ar vieniem un tiem pašiem rezultātiem, tad tas tiek uzskatīts par idempotentu. Ja tests nav idempotents, tad ir nepieciešams veikt daudz mērījumus, lai nodrošinātu, ka tas ir izpildīts tieši vienu reizi, un pēc testa izpildes ir nepieciešams atjaunot izpildes vidi atkārtotai testa izpildei.

**Minimizēt testu pārklāšanos**

Ja ir vairāki testi, kuri pārbauda vienu un to pašu sistēmas daļu, tad, augot testu izmēriem un skaitam, lieki palielināsies testu laika izmaksas – tiks testēts viens un tas pats funkcionalitātes apgabals atkārtoti bez vajadzības [XPPOT].

## 2.2. Automatizētās testēšanas izmantošana, priekšrocības un trūkumi

Daļa testētāju un izstrādātāju iespējams domā, ka automatizētā testēšana atvieglo darbu, jo pēc testu izveides šķietami pašam vairs nav jādara tik daudz darba, tādēļ to būtu jāievieš vienmēr un visur, kad iespējams. Tomēr ne visos gadījumos automatizēto testu ieviešana ir labākais variants.

Automatizēto testēšanu nav vēlams ieviest, kad tiek veidota jauna vai mainīga funkcionalitāte, jo pēc testu izveides funkcionalitāte var mainīties, un tas nozīmē vairāk lieka darba testa pārveidošanai, lai tas strādātu ar veiktajiem programmatūras labojumiem. Lielākoties automatizēto testu ieviešana, kamēr vēl līdz galam nav izstrādāta funkcionalitāte, vai tā var mainīties, ir lieka laika izšķiešana [OWTAT].

Ja programmatūrai ir ārkārtīgi sarežģīta funkcionalitāte, automatizētā testa ieviešana var aizņemt daudz laika un tas var būt neizdevīgi. Tādos gadījumos labāks variants ir pieturēties pie manuālās testēšanas [OWTAT].

Automatizēto testēšanu vislabāk pielietot regresijas testiem (kad ir nepieciešams atkārtoti testēt vienu un to pašu funkcionalitāti, kas tiek pārnesta uz nākamo programmatūras versiju), dūmu testiem (ātra vispārīga sistēmas pārbaude), datu virzītai testēšanai (ja nepieciešams vienu un to pašu funkciju vai formu notestēt ar daudz, dažādiem ievades datiem, piemēram, meklēšana vai pieslēgšanās forma) [OWTAT].

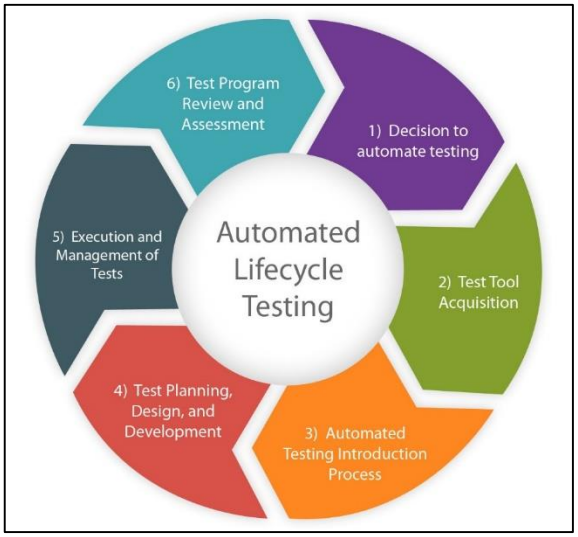
Automatizētajai testēšanai ir sekojošas priekšrocības [BAVMT]:

- testi izpildās ātri un efektīvi – lai arī sākotnējā testu iestatīšana aizņem laiku, līdzko testi ir izveidoti, tie ir atkārtojami izmantojami;
- testi ir efektīvi izmaksu ziņā – lai arī īstermiņā automatizācijas rīki ir dārgi, tomēr ilgtermiņā tie ietaupītu naudu. Tie ne tikai dara vairāk, kā cilvēks varētu paveikt konkrētā laikā, bet arī ātrāk atrod nepilnības;
- visi var apskatīt testu rezultātus – tad, kad viena persona veic testēšanu, pārējā komanda nevar redzēt testēšanas rezultātus, savukārt automātiskajai testēšanai rezultāti tiek piereģistrēti, un ikviens ar pieejas datiem tos var apskatīt. Tas nodrošina lielāku komandas sadarbību un labāku gala produktu.

Automatizētajai testēšanai tomēr ir arī savi trūkumi – rīki var būt dārgi, kā arī to uzstādīšana un izmantošana joprojām aizņem laiku. Kaut arī automātiskie testi atradīs lielāko daļu kļūdu sistēmā, tiem tomēr ir savi ierobežojumi – retums automatizētās testēšanas programmatūru var noteikt vizuālās izmaiņas (piemēram, attēla krāsās vai fonta izmērā). Šādas izmaiņas pilnībā var noteikt tikai manuālā testēšana. Tas nozīmē, ka ne visa testēšana var tikt paveikta ar automātiskajiem rīkiem [BAVMT].

## 2.3. Automatizētās testēšanas dzīves cikls

Automatizētās testēšanas dzīves cikls sastāv no 6 galvenajām fāzēm (skat. 2.1. att.) – lēmums veikt automatizēto testēšanu, testēšanas rīka iepazīšana, automatizētās testēšanas iepazīšanas procesa, testu plānošanas, projektēšanas un izstrādes, testu izpildes un pārvaldības, kā arī testēšanas pārskatīšanas un novērtēšanas [KTQAS].




*2.1. att.* **Automatizētās testēšanas dzīves cikls [KTQAS]**

**Lēmums veikt automatizēto testēšanu**

Lēmums veikt automatizēto testēšanu ir pirmā automatizētās testēšanas dzīves cikla fāze. Šī fāze pārklāj visu automatizētās testēšanas lēmuma procesu. Šīs fāzes laikā ir svarīgi testēšanas komandai izskatīt automatizētās testēšanas gaidas un atzīmēt potenciālos ieguvumus no automatizētās testēšanas, ja tā tiks veikta atbilstoši. Ir nepieciešams arī izvēlēties testa rīkus, kas palīdzēs veikt nepieciešamās testēšanas darbības [EDTAT].

**Testēšanas rīka iepazīšana**

Tā kā testēšanas rīkam vajadzētu atbalstīt lielāko daļu no organizācijas testēšanas prasībām, gadījumos, kad tas ir iespējams, testēšanas inženierim ir jāpārskata sistēmas inženierijas vide un citas organizatoriskās vajadzības, kā arī jāizstrādā testēšanas rīku vērtēšanas kritēriju saraksts. Balstoties uz šo sarakstu, personāls novērtē testēšanas rīkus [EDTAT].

**Automatizētās testēšanas iepazīstināšanas process**

Automatizētās testēšanas iepazīstināšanas process sastāv no testēšanas procesa analīzes un testēšanas rīka apsvēršanas un izskatīšanas. Testēšanas procesa analīze nodrošina, ka kopumā testēšanas process un stratēģija ir savā vietā, un ka tie tiek mainīti, kad nepieciešams, lai ļautu veiksmīgi ieviest automatizēto testēšanu. Testēšanas rīka apsvēršanas un izskatīšanas fāzē ir ietverti soļi, kas pēta, vai, ņemot vērā projekta testēšanas prasības, pieejamās testēšanas vides un personāla resursus, konkrētajā projektā būtu izdevīgi izmantot automatizētās testēšanas rīku iekļaušanu bez īpaša projekta [EDTAT].

**Testu plānošana, projektēšana un izstrāde**

Testu plānošanas fāze atspoguļo nepieciešamību pārskatīt ilgstošas testu plānošanas aktivitātes. Šajā fāzē testa grupa nosaka testa procedūras izveides standartus un vadlīnijas, aparatūru, programmatūru, kas nepieciešama testa vides atbalstam, testēšanas grafiku, izpildījuma mērījumu prasības, kā arī testa konfigurācijas un vides kontroles procedūru un defektu izsekošanas procedūru [EDTAT].

Testu projektēšana pieprasa noteikt nepieciešamo testu skaitu, veidus, kā tiks veikts tests (ceļi, funkcijas), kā arī testa nosacījumi, kas jāveic. Ir nepieciešams noteikt testu projektēšanas standartus un tiem sekot [EDTAT].

Lai automatizētie testi būtu atkārtoti izmantojami, atkārtojami un uzturami, ir nepieciešams sekot testu izstrādes standartiem [EDTAT].

**Testu izpilde un pārvaldība**

Šajā fāzē testētāju komanda pievēršas testu projektēšanai un izstrādei. Komandai ir uzdevums izpildīt testēšanas programmatūrai noformulētos testus. Izpildot testa procedūras, testēšanas komandai jāievēro testēšanas procedūras izpildes grafiku. Testēšanas procedūras izpildes grafiks rada testa plānā noteikto stratēģiju – tiek izpildīti integrācijas, sistēmas un lietotāju akcepttesti. Visas šīs testēšanas fāzes kopā veido soļus, kas nepieciešami, lai sistēmu notestētu kopumā [EDTAT].

**Testēšanas pārskatīšana un novērtēšana**

Lai veiktu pastāvīgas uzlabošanas darbības, ir nepieciešams visa testa dzīves cikla laikā veikt testēšanas programmatūras pārbaudes un novērtēšanas darbības. Papildus ir nepieciešams arī novērtēt metrikas, kā arī ir nepieciešams veikt gala pārbaudes un novērtēšanas darbības, lai testēšanas procesu varētu uzlabot [EDTAT].

## 2.4. Testu automatizācijas līmeņi

Lietojumprogrammatūras testi iedalās trijos līmeņos – vienību testi (*unit*), funkcionālie (integrācijas) testi un lietotāja saskarnes testi (*user interface*) [BQLTA].

### 2.4.1. Vienību testi

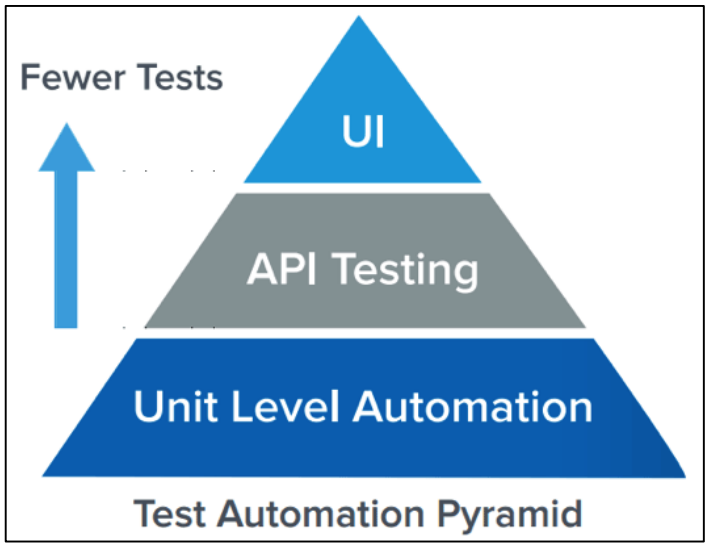
Katrs vienību tests apstiprina, ka funkcija atgriež paredzēto rezultātu, ja tiek padoti zināmi ievades dati. Automatizētu vienību testu izpilde ir ļoti ātra, un tos ir vēlams izpildīt ikreiz, kad ir veiktas izmaiņas kodā. Tā kā šie testi ir nelieli un specifiski, atrast kļūdas rašanās vietu var salīdzinoši ātri [TTABE].

### 2.4.2. Funkcionālie (integrācijas) testi

Funkcionālie testi ir koda līmeņa skripti (tādi, kas nedarbojas lietotāja saskarnē), kas testē pilnīgu procesu, kurā iesaistīti vairāki objekti. Katrs objekts individuāli tiek testēts vienību testēšanā, savukārt funkcionālajā testēšanā viens tests izpilda vairākos vienību testos ietvertu funkcionalitāti kā secīgu darbību lietošanas piemēru. Piemērs funkcionālajam testam ir "pasūtījuma iesniegšana", kas ne tikai veic pasūtījumu, bet arī pārbauda vai pasūtījumu datu bāze ir atjaunota, vai pasūtījuma numurs ir pareizs, un vai pasūtījuma veicējam ir aizsūtīts pareizais apstiprinājuma e-pasts [CAUTT].

### 2.4.3. Lietotāja saskarnes testi

Lietotāja saskarnes testu līmenī ir iespējams testēt gan lietotāja saskarni, gan funkcionalitāti, tomēr tas nenozīmē, ka saskarnes testos ir jāveic visu funkcionālo testēšanu. Jo zemāks līmenis, jo detalizētāki ir testi, tādēļ lietotāja saskarnes testos ir vēlams testēt tikai to, kas notiek caur lietotāja saskarni, un kas nav notestējams zemākā līmenī [TBHTA]. Lietotāja saskarnes automatizētajiem testiem ir nepieciešama uzturēšana ik reizi, kad mainās lietotāja saskarne, un tieši tādēļ, ka ir tik daudz faktoru, kas rodas, veicot testu, kas emulē klikšķus uz ekrāna (piemēram interneta ātrums), šādi testi var rezultēties viltus kļūdās (*false negative*) [TTABE].




*2.2. att.* **Automatizēto testu piramīda [SBLAW]**

Ja visu trīs līmeņu testus izpilda secīgi, tad katrā nākamajā līmenī ir jāpārbauda tikai funkcionalitāte, ko zemākais līmenis nav pārbaudījis (piemēram, no lietotāja saskarnes nav nepieciešams testēt funkcionālo līmeni, jo tas ir paveikts, izmantojot funkcionālos testus), tas nozīmē mazāk testu izveidi. Aprakstītais ir attēlots 2.2. att. – galveno automatizēto testu daļu sastāda vienību testi, aiz kuriem seko funkcionālā testēšana, kas testē ar funkcionālajiem testiem nenotestējamos aspektus, un tikai tad funkcionalitāti un aspektus, kas vēl nav notestēti līdz šim, pārbauda ar saskarnes testiem [TTABE].

## 2.5. Testu automatizācijas ietvari

Testu automatizācijas ietvars ir vadlīniju kopums testpiemēru izstrādei. Tā ir automatizētas testēšanas konceptuāla daļa, kas palīdz testētājiem efektīvāk izmantot resursus [TPTAF]. Testu automatizācijas ietvari palīdz uzlabot testu efektivitāti, samazināt

uzturēšanas izmaksas, minimizēt manuālu iejaukšanos testos, maksimizēt testu pārklājumu, kā arī kods ir atkārtoti izmantojams [SBLAT].

Ir sekojoši galvenie testu automatizācijas ietvari [STATT]:

- lineārais ietvars,
- modulārais ietvars,
- datu virzītais ietvars,
- atslēgvārdu virzītais ietvars,
- hibrīdais ietvars.

**Lineārais ietvars**

Lineārais ietvars ir visvienkāršākais no visiem testu automatizācijas ietvariem. Veidojot testus pēc lineārā ietvara principa, visa ar testu saistītā informācija ir nodefinēta pašā testā. Tas nozīmē, ka tests pats satur informāciju, uz kura objekta ir jāspiež, kādu informāciju ir jāievada katrā konkrētā ievades laukā, u.c., kas nozīmē, ka tests ir nodefinēts vienam konkrētam testēšanas piemēram, un, ja ir nepieciešams testu izpildīt uz citiem ievades datiem, ir jālabo pašu testa kodu. Tieši šī iemesla dēļ šos testus uzturēt ir sarežģīti un ir nepieciešami regulāri labojumi, lai testus izmantotu atkārtoti. Testi ir vienkārši, ātri izveidojami, un ir nepieciešama minimāla plānošana, tomēr testiem nav atkārtota lietojamība [STATT, SBLAT].

Lineāro ietvaru var izmantot maza līmeņa projektos, kuros nav daudz lietotāja saskarnes ekrānu. Tad, kad kāds automatizācijas rīks tiek izmantots pirmo reizi, tas parasti uzģenerē kodu lineārā formā. Izņemot šos iemeslus, būtu jāizvairās no lineārā ietvara izmantošanas projektos [STATT].

Piemērs lineāram testam ir izpildīt sekojošus punktus: atvērt "*http://test.com/login*" mājas lapu, ievadīt "*user1*" lietotāja vārda laukā, ievadīt "*password*" paroles laukā, spiest uz pogu "*Log in*", pārbaudīt vai lapas teksts satur informāciju "*Hello, user1!*".

**Modulārais ietvars**

Īstenojot testu veidošanu pēc modulārā ietvara, lietotni ir nepieciešams sadalīt atsevišķās vienībās, funkcijās vai iedaļās, kur katra tiks testēta izolēti no pārējām. Pēc lietotnes sadalīšanas moduļos, tiek izveidoti testu skripti katram modulim, un tālāk tie tiek apvienoti lielākos testos hierarhiskā veidā. Šie lielākie testi pārstāvēs dažādus testpiemērus [SBLAT].

Testiem, kas balstīti uz modulārā ietvara, objekti ir nodefinēti vienu reizi un ir atkārtoti izmantojami visās testa funkcijās. Tiek veidotas nelielas funkcionalitātes testēšanas funkcijas, kas veic konkrētu darbu. Ja tiek veiktas izmaiņas lietojumprogrammatūrā, tad ir tikai jānosaka modulis un tā saistītie individuālie testa skripti, kuros veikt izmaiņas, un pārējie testu skripti

var tikt atstāti neskarti. Testpiemēru veidošana uz modulārā ietvara pamata aizņem daudz mazāk laiku, nekā uz lineārā ietvara balstītie testi. Tomēr modulārajam ietvaram ir trūkumi – dati vēl joprojām ir stingri iekodēti testa skriptā, un viens skripts nav izmantojams vairākām datu kopām, kā arī šādu testu izveidošana prasa labas objektorientētās programmēšanas zināšanas un prasmes [STATT, SBLAT].

Atšķirība no lineāriem testiem ir sekojoša (salīdzinājums ar iepriekš aprakstīto piemēru): katra veicamā darbība tiek definēta kā atsevišķa funkcija (t.i., funkcija "*Atvērt mājas lapu*" ar ieejas parametru "*MājaslapasSaite*", funkcija "*Ievadīt tekstu ievades laukā*" ar ieejas parametriem "*IevadāmaisTeksts*" un "*IevadesLaukaID*", funkcija "*Nospiest pogu*" ar ievades parametru "*PogasID*" un funkcija "*Verificēt lapas tekstu*" ar ievades parametru "*SagaidāmaisTeksts*". Bez aprakstītajām funkcijām vēl ir funkcija "*Pieslēgt lietotāju*", kurai tiek padoti visi tālāk padodamie ievades parametri, un kas izsauc visas 4 aprakstītās funkcijas. Šī funkcija ir tā, kas kalpo kā individuāls uz modulārā ietvara balstīts tests.

**Datu virzīts ietvars**

Datu virzītajā ietvarā testu dati ir atdalīti no skriptu loģikas, kas nozīmē, ka testiem padodamos datus var uzturēt ārēji. Ļoti bieži testētājiem ir jātestē vienu funkcionalitāti vairākas reizes ar dažādiem ievades datiem, dēļ kā ir ļoti svarīgi, ka testu dati netiek stingri iekodēti pašā testā, kā tas notiek uz lineārā vai modulārā ietvara balstītos testos [SBLAT].

Uz datu virzīta ietvara balstītiem testiem ir sekojošas priekšrocības – testus var izpildīt ar dažādām ievades datu kopām, un ir vienkāršāk un ātrāk testēt dažādus scenārijus, jo ir tikai jāizmaina vai jāpapildina datus ārējā avotā. Tomēr šādu testu veidošana aizņem daudz laika, un ir nepieciešams ļoti pieredzējis testētājs, lai šādus testus izveidotu [SBLAT].

Salīdzinājumā ar modulāriem testiem, datu virzītie testi uzglabā ārēju testpiemēru datu avotu ar kolonnām (katram parametram sava kolonna) un rindām (katrā rindā ir savs testpiemērs), kas tiek padots testam. Tas nozīmē, ka tests tiks izsaukts tik reizes, cik rindas ir datu tabulā.

**Atslēgvārdu virzīts ietvars**

Veidojot testus pēc atslēgvārdu ietvara, ārējos avotos atrodas ne tikai parametri, bet arī izpildāmās darbības. Tas nozīmē, ka galējā testu izpildes ideja netiek nodefinēta kodā, bet gan ārējos vienkārši pārskatāmos un modificējamos avotos, failos vai tabulās. Ir nepieciešams nodefinēt lineāras funkcijas (kas izpilda vienu konkrētu darbību), un tālākās darbības notiek, vadoties pēc ārējā avotā nodefinētiem soļiem (skat. 2.1. tabulu), kur tiek norādīta kontrole ar kuru nepieciešams mijiedarboties, darbība (funkcija), ko uz šo kontroli jāizpilda, kā arī

nepieciešamie ieejas parametri. Katrai rindai tabulā tiek izpildīta atbilstošā darbība [STATT, SBLAT].

*2.1. tabula*

**Uz atslēgvārdu virzītā ietvara balstīta testa atslēgvārdu tabulas piemērs**

| Darbība (funkcija) | Objekts (kontrole) | Ieejas parametri |
|---|---|---|
| AtvērtLapu | | http://test.com/login |
| IevaddītTekstu | LietotājaVārdaLauks | user1 |
| IevaddītTekstu | ParolesLauks | password |
| SpiestPogu | LogIn | |
| PārbaudītTekstu | HelloBloks | Hello, user1! |

Lai izveidotu testus uz atslēgvārdu ietvaru pamata, ir nepieciešamas minimālas kodēšanas zināšanas, tomēr ietvara izstrāde un uzstādīšana prasa augstas izmaksas, jo tas ir laikietilpīgs un sarežģīts darbs, kur ir nepieciešams nodefinēt atslēgvārdus, kas tiek apstrādāti no testa datiem, un objektu bibliotēkas. Individuāls atslēgvārds var tikt izmantots vairākos testa skriptos, kas nozīmē, ka kods ir atkārtoti izmantojams. Tomēr atslēgvārdu uzturēšana var būt sarežģīta, augot testu izmēriem – būs nepieciešams regulāri papildināt atslēgvārdu tabulas [SBLAT].

**Hibrīdais ietvars**

Hibrīdais ietvars var būt divu vai vairāku iepriekš aprakstīto ietvaru apvienojums, no tiem paņemot to stiprās puses un mazinot to vājās vietas. Šis ietvars var, piemēram, izmantot modulāro pieeju kombinācijā ar datu virzīto vai atslēgvārdu virzīto ietvaru, kā arī var tikt izmantoti skripti, lai izpildītu darbības, kuras būtu pārāk sarežģīti realizēt, izmantojot tikai atslēgvārdu virzīto ietvaru [STATT].

## 2.6. Automatizēto testu izveides iespējas

Liela daļa sistēmu sadalās divās pamata daļās – datu bāze un lietotāja saskarne, jeb pati lietotne. Šādām sistēmām ir nepieciešams testēt sistēmu gan saskarnē (testējot funkcijas, kas atrodas lietotnes kodā), gan arī datu bāzes funkcionalitāti, ja tajā tiek uzturētas funkcijas, procedūras un izpildītas darbības.

### 2.6.1. Lietotāja saskarnes testēšana

Lietotāja saskarnes testēšana ir sistēmas vai programmatūras grafiskās lietotāja saskarnes testēšana, lai pārliecinātos, ka sistēma atbilst prasībām. Grafiskās lietotāja saskarnes testi spēj

atdarināt tādas darbības kā peles klikšķis, rullēšana, u.c. lietotāja veiktās darbības, mijiedarbojoties ar sistēmu. Testi spēj arī mainīt objektu vērtības, piemēram teksta laukam nolasīt un piešķirt vērtību. Tomēr testu rakstura iezīmes ir atkarīgas no izmantotajām tehnoloģijām programmatūras izstrādē [TPUIT].

Lietotāja saskarnes testos ir svarīgi pārbaudīt visas navigācijas darbības (apmeklēt un notestēt visas lietotājam pieejamās sadaļas), pārbaudīt datu ticamību (katrā sadaļā pārliecināties, ka uzrādītie dati ir patiesi un netiek uzrādīta nekorekta vai neatbilstoša informācija), pārbaudīt objektu stāvokļus (pārbaudīt vai pēc darbības izpildes ir pieejamas kontroles, pogas, un vai tās ir aktīvas vai neaktīvas), kā arī pārbaudīt datuma un skaitļu lauka formātus (vai datums tiek uzrādīts pareizā formātā, un vai skaitļiem ir pareizi noformatēti cipari aiz komata, u.c.) [TPUIT].

Lietotāja saskarnes automatizētos testus ir iespējams izstrādāt pašam. Ja sistēma ir maza izmēra un sastāv no maza daudzuma darbību, tad iespējams, ka nav vērts implementēt programmatūru, ar kuru sistēma tiks testēta, jo tas var aizņemt vairāk laika, nekā manuālu testu izstrāde. Tomēr ja sistēma ir maza, bet augoša, un ir skaidrs, ka funkcionalitātes klāsts un programmatūras izmērs augs, efektīvāk tomēr ir izmantot jau esošu programmatūru, kas ir parūpējusies par to, lai nav jāprogrammē pašu testu darbības funkcionalitāti dažādiem scenārijiem, bet ir tikai jāizstrādā pašus testus.

#### 2.6.1.1. Selenium automatizēto testu ietvars

Vispopularizētākā metode automatizēto testu izstrādē ir *Selenium*. Ar tā palīdzību ir iespējams testēt tīmekļa vietnes, izmantojot dažādas tīmekļa pārlūkprogrammas, un tas atbalsta testu izstrādi dažādās valodās, to skaitā *Java*, *PHP* un *C#*. *Selenium* testi ļoti bieži piedāvā ierakstīšanas un atskaņošanas iespējas, ar kā palīdzību testus ir iespējams izstrādāt ar minimālām programmēšanas zināšanām [DATAS].

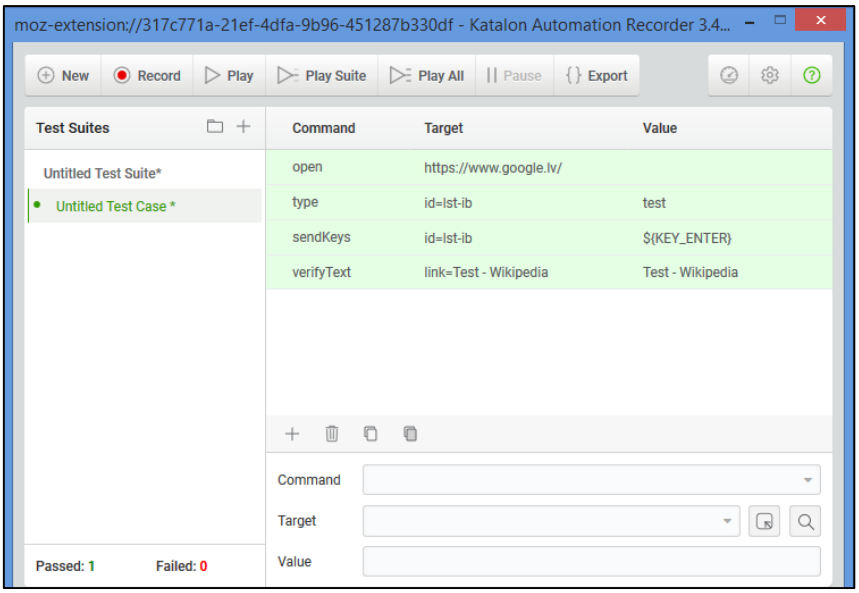
Izvēloties *Selenium* kā programmatūras testēšanas automatizācijas variantu, vēl ir jāizvēlas rīks, kurā *Selenium* ir iebūvēts, vai arī jāizveido kodu pašam, kas izmanto *Selenium* piedāvātā ietvara kodu un funkcijas.

**Katalon Recorder un Katalon Studio**

Visātrāk iegūstamais variants *Selenium* rīka izvēlei ir interneta pārlūkprogrammai *Mozilla Firefox* vai *Google Chrome* pievienot bezmaksas spraudni "Katalon Recorder", kas tieši izmanto *Selenium* piedāvāto kodu. Pēc spraudņa pievienošanas, ir iespējams atvērt testu logu (skat. 2.3. att.), spiest pogu "Ierakstīt" un izpildīt izvēlētas darbības – spiest uz pogām vai saitēm, pieprasīt validēt ievades lauku vērtības vai tekstu konkrētā blokā, mainīt šīs vērtības un veikt citas darbības. Pēc testa apturēšanas ir iespējams apskatīt visus ierakstītos soļus, tos dzēst

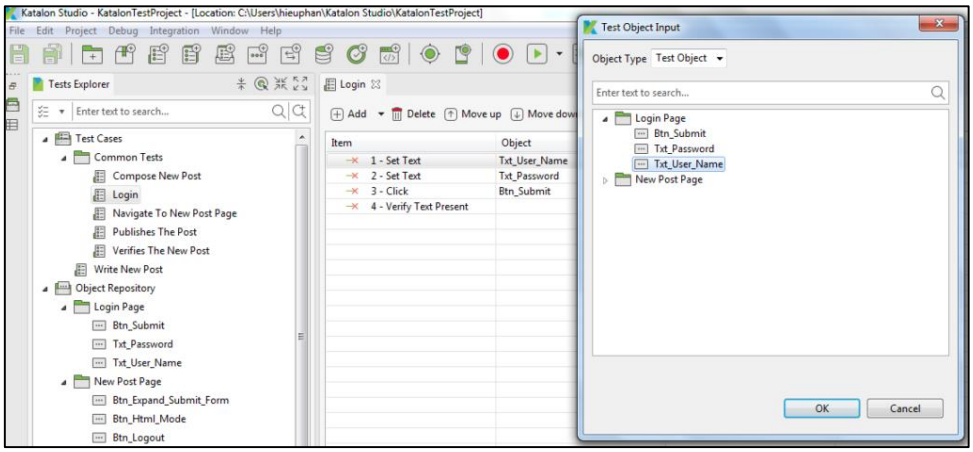
un testu atkārtoti pieprasīt izpildīt. Šis risinājums nepieprasa nekādas programmēšanas zināšanas, tomēr ja ir nepieciešams saprast, ko konkrētā darbības rinda dara un kāpēc, kā arī labot kādu darbību, tas var būt apgrūtinoši, jo spraudnis izmanto specifisku koda uzbūvi, kas nav līdzīga citām programmēšanas valodām.

*Katalon Recorder* pievienot tīmekļa pārlūkprogrammai un sākt ar to darboties ir ļoti ātri un vienkārši, un nav nepieciešamas iemaņas spraudņa lietošanā vai programmēšanā. Tomēr ir ļoti ierobežots izpildāmo darbību klāsts un nav iespējams veikt labojumus darbībās, jo objektu identifikācija ir stingri iekodēta programmatūrā bez iespējas tai piekļūt vai to labot.



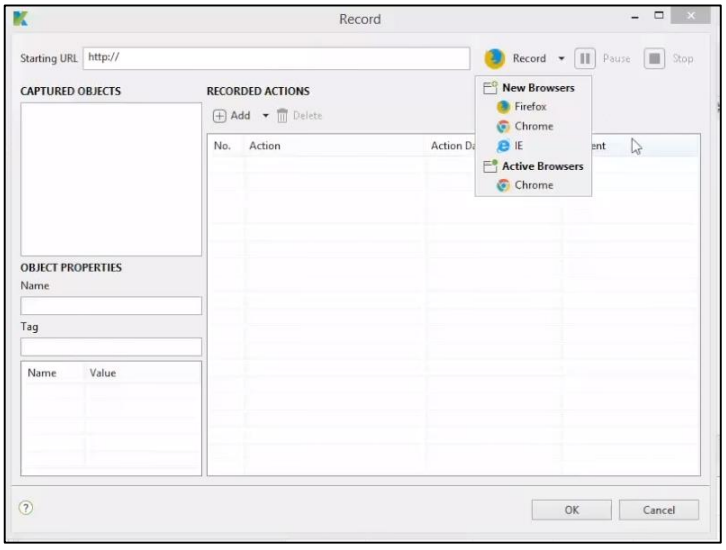

*2.3. att.* ***Katalon Recorder*** **spraudņa logs**

Papildu sarežģītāku darbību veikšanai un augstāka līmeņa testēšanai, no *Katalon* atbilstošāka ir datorā instalējama programmatūru *Katalon Studio* (skat. 2.4. att.). Tā ir balstīta uz *Selenium* kodu, tomēr tai ir pievienots arī individuāls kods un funkcionalitāte, ko nenodrošina daudz citu *Selenium* izmantojošo programmatūru vai spraudņu. *Katalon Studio* izmantošanai ir nepieciešamas zināšanas par tās lietošanu un ar to nav iespējams darboties tik vienkārši, kā ar pārlūkprogrammas spraudni [KATKS].




*2.4. att.* ***Katalon Studio*** **programmatūras logs ar testu ierakstīšanas iespēju [KATKS]**

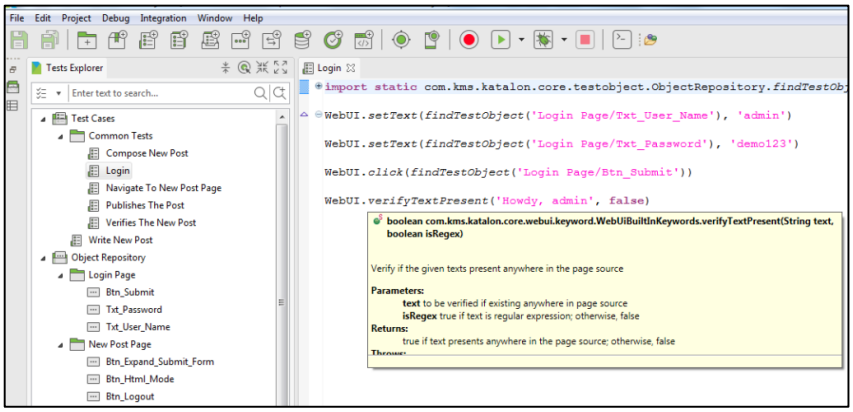
*Katalon Studio* piedāvā ierakstīšanas un atskaņošanas principu, kas nozīmē, ka nav nepieciešamas programmēšanas zināšanas, lai izstrādātu testus. Spiežot ierakstīšanas pogu, tiek atvērts logs (skat. 2.5. att.), kur ir jāizvēlas tīmekļa vietne, kuru testēt (sākotnējā saite), kā arī tīmekļa pārlūkprogramma, kura tiks izmantota testu ierakstīšanai un atskaņošanai (tiek atbalstītas *Firefox*, *Chrome*, *Internet Explorer*). Pēc datu akceptēšanas, izvēlētajā pārlūkprogrammā tiek atvērta sākotnējā saite un katra lietotāja veiktā darbība tiek ierakstīta. Kad ir veiktas nepieciešamās darbības, atgriežoties programmatūras logā, ir redzamas piefiksētās darbības un izmantotie objekti (ievades lauki, pogas, tekstu bloki, u.c.), kuras var labot, dzēst, un beigās visu saglabāt kā testpiemēru.




*2.5. att.* ***Katalon Studio*** **darbību ierakstīšanas konfigurācijas logs**

Katra testpiemēra darbības arī vēlāk ir iespējams labot, mainot to uz kādu no daudzām piedāvātajām darbībām, kā arī var pievienot jaunas darbības un norādīt objektu, ar kuru tās veikt. Katram objektam var izvēlēties vienu vai vairākus parametrus, pēc kā to atpazīt (teksts, identifikators, nosaukums, klase, u.c.).

Programmatūra piedāvā arī pārslēgties uz logu, kur testa informācija un ierakstītās darbības ir pārveidotas uz kodu, kuru var labot un kuram var pievienot klāt darbības (skat. 2.6. att.). Tad, kad testi ir izstrādāti, tos ir iespējams atskaņot, izmantojot "Run" pogu. Pēc testu izpildes tiek parādīta informācija par katru veikto darbību un tās izpildes statusu (veiksmīgs, neveiksmīgs), kā arī var apskatīt grafikus ar katru testu izpildes reizi, testu skaitu, kas ir izpildījušies veiksmīgi un neveiksmīgi.



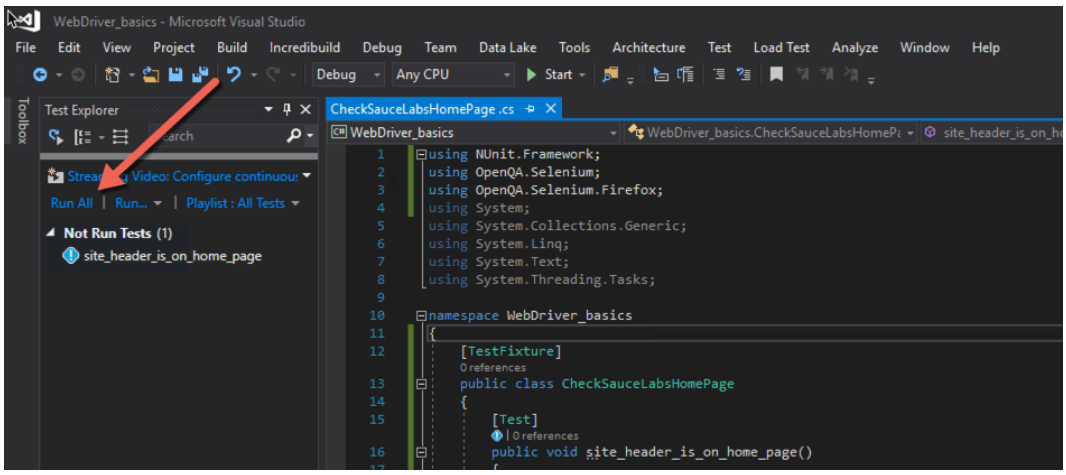

*2.6. att.* ***Katalon Studio*** **testa skriptēšanas režīms**

### ***Microsoft Visual Studio*** **pakotne ar** ***Selenium*** **atbalstu**

Maksas variants rīkam, kas izmanto *Selenium*, izvēlei ir *Microsoft Visual Studio* programmatūra. Lai to izmantotu, projektam ir nepieciešams pievienot pakotnes, kas papildinās projekta kodu ar iepriekš sagatavotām funkcijām un klasēm automatizēto testu izveidei.

Testi ir izstrādājami tikai rakstot tos ar roku kā kodu. Tas nozīmē, ka šo risinājumu nav iespējams lietot bez programmēšanas zināšanām, ir nepieciešamas padziļinātas programmēšanas zināšanas. Ir nepieciešamas arī pamata zināšanas darbam ar *Visual Studio*, lai izstrādātu testus ar šo metodi. Kods tiek rakstīts *C#* valodā, un elementu manipulācijai tiek izmantota *JavaScript* valoda.

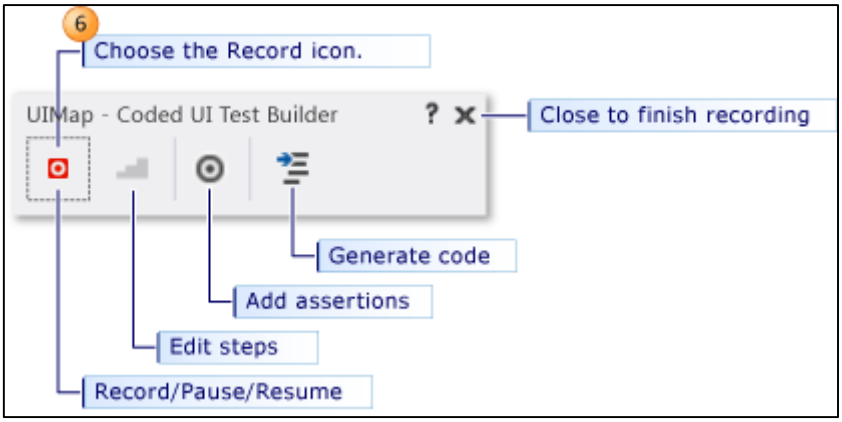
Pēc testu izstrādes var palaist individuālus testus, neveiksmīgi izpildītos testus, ne reizi nepildītos testus vai arī visus testus (skat. 2.7. att.). Testus veidot un uzturēt *Visual Studio* ir parocīgi tad, ja pati programmatūra, kas tiek testēta, arī tiek izstrādāta uz šīs pašas programmatūras, jo tad programmatūra un tās testi atrodas vienuviet [DMCTS].




*2.7. att.* ***Microsoft Visual Studio Selenium*** **testa izpildes logs ar darbību "Palaist visus testus" [SGSWW]**

### 2.6.1.2. Datorā instalējamas programmatūras testēšana

Ja testējamā programmatūra nav tīmekļa vietne, bet gan datora programmatūra, *Selenium* nav iespējams pielietot. *Microsoft Visual Studio* programmā ir iespējams arī testēt datora programmatūru, izmantojot "Coded UI Tests" komponenti (skat. 2.8. att.), kas ļauj izstrādāt kodētus lietotāja saskarnes testus. Ar tās palīdzību var ierakstīt lietotāja veiktās darbības, no tām uzģenerēt kodu, kuru var labot un atkārtoti izpildīt [DMUUA]. Padziļinātāk datora programmatūras automatizētās testēšanas varianti netiek apskatīti, jo maģistra darba mērķis ir izstrādāt automatizētos testus tīmekļa vietnei.




*2.8. att.* ***Microsoft Visual Studio*** **kodēta lietotāja saskarnes testa ierakstīšanas loga kontroļu paskaidrojumi [DMUUA]**

### 2.6.1.3. Rīku priekšrocības un trūkumi

Tā kā darba mērķis ir izanalizēt un realizēt testus tīmekļa lietotnei, tiks salīdzināti tikai tīmekļa lietotnes automatizēto testu izstrādes varianti.

*Katalon Recorder* priekšrocība ir iespēja to izmantot bez programmēšanas zināšanām. Rīka trūkumi ir tā pieejamo darbību klāsts, jo rīks atbalsta tikai "ierakstīt un atskaņot" principu, un tam nav piekļuves uzģenerētajam kodam, lai veiktu manuālus labojumus testos.

Arī *Katalon Studio* priekšrocības ietver iespēju rīku izmantot bez programmēšanas zināšanām. Testus ir iespējams labot, kā arī ar roku var pievienot izvēlētas darbības ar izvēlētu objektu. Pieejamo darbību klāsts ir ļoti liels, kā arī ir iespējams piekļūt uzģenerētajam kodam un to labot. Rīka trūkumi ir "ierakstīt un atskaņot" principa izmantošana, kas nozīmē tikai piedāvāto darbību veikšanu testa izstrādē, jo nav iespēja pašam izstrādāt funkcijas, kas veic nepieciešamās darbības. Ir nepieciešamas nelielas rīka zināšanas, lai darbotos ar rīku.

*Microsoft Visual Studio Selenium* testu priekšrocības ietver iespēju izstrādāt testu funkcijas ar roku, kas nozīmē neierobežotu izpildāmo darbību klāstu. Ja testējamā programmatūra tiek izstrādāta uz *Microsoft Visual Studio* programmatūras, tad testus var izstrādāt tajā pašā projektā, kurā tiek izstrādāta testējamā programmatūra. Tas nozīmē, ka nepieciešama nav atsevišķa programmatūra testu izstrādei, un testu uzturēšana ir vienkāršāka, jo kods un testi atrodas vienuviet. Risinājuma trūkumi ietver nepieciešamību zināt programmatūras zināšanas, lai izstrādātu testus, kā arī ir nepieciešamas zināšanas darbam ar *Microsoft Visual Studio* programmatūru. Visus testus ir nepieciešams rakstīt ar roku, jo risinājums neatbalsta "ierakstīt un atskaņot" principu.

### 2.6.2. Datu bāzes testēšana

Lietotāja saskarnes testi pārbauda, kā sistēma darbojas no grafiskās lietotāja saskarnes puses, tomēr šādi netiek pārbaudīts viss programmatūras kods. Gadījumos, kad datu bāze nesastāv tikai no tabulām, bet arī no pakotnēm, funkcijām un procedūrām (datu bāzu vadības sistēmu *MS SQL Server*, *PL/SQL Developer*, u.c. gadījumos), ir ieteicams testēt arī datu bāzes funkcijas un procedūras, jeb veikt vienību testēšanu.

Kamēr ikvienas tīmekļa vietnes lietotāja saskarni ir iespējams testēt ar vienu programmatūru gandrīz neatkarīgi no valodas, kurā programmatūra rakstīta, datu bāzei tā nav. Datu bāzes vienību testus ir nepieciešams veidot, zinot kāds ir kods, kas nozīmē, ka programmatūra katrai datu bāzes pārvaldības sistēmai atšķirsies.

Datu bāzes testiem priekšrocības pret lietotāja saskarnes testiem ir tādas, ka ir iespējams pārbaudīt datu stāvokli starp darbībām, kā arī var validēt informāciju, kas tiek uzglabāta tikai datu bāzē un sistēmā lietotājiem rādīta netiek. Testos, kas izpildāmi datu bāzē ir iespējams arī izpildīt transakcijas atrites darbību, kas atgriež datus sākotnējā stāvoklī.

#### 2.6.2.1. Testu izstrāde bez palīglīdzekļiem

Datu bāzes testus ir iespējams izstrādāt bez palīglīdzekļiem, vienā procedūrā vai testa logā izsaucot sistēmas darbības vai procedūras, un starp to izsaukšanas reizēm pārbaudot vai dati ir izveidoti un atgriezti pareizi. Šādam risinājumam visticamāk ir nepieciešams izstrādāt funkcijas, kurām var padot divas vērtības, un kas atgrieztu pazīmi vai tās sakrīt vai nesakrīt. Pēc saņemtā rezultāta testā var izlemt tālāk veicamās darbības, piemēram, pārtraukt procedūras izpildi, izpildīt transakcijas atriti un atgriezt ārējā parametrā rezultātu vai izvadīt konsolē neveiksmīgā rezultāta detalizētu pārskatu.

Manuālu testu izstrādes risinājums var būt parocīgs, ja sistēmā nav daudz darbību, funkciju un procedūru, ko testēt. Tomēr ja ir nepieciešams pārbaudīt un salīdzināt sarežģītas datu struktūras (piemēram, tabulas saturs vai kursora informācija), tas var būt apgrūtinoši, un specializētu funkciju un pārbaužu izstrāde var aizņemt laiku.

#### 2.6.2.2. Testu izstrāde ar palīglīdzekļiem

Maģistra darbā apskatāmajai sistēmai tiek izmantota *Oracle PL/SQL* datu bāzes programmēšanas valoda, tādēļ tiek meklēti un apskatīti risinājumi, kas atbalstītu automatizēto testu izstrādi tieši šajā programmēšanas valodā.

**utPLSQL**

Ikvienā *Oracle* datu bāzē ir iespējams izmantot "utPLSQL" spraudni. Spraudnis ir pieejams bezmaksas, un, to veiksmīgi instalējot, tiek pievienots lietotājs ar datu bāzi, kas satur pakas, kuru procedūras ir izmantojamas testos. Tad, kad ir uzinstalēts spraudnis, lai izveidotu testus, ir nepieciešams izveidot paku ar procedūrām, kas veic datu labošanu, atlasi un citas darbības, kuras ir nepieciešams testēt. Pēc darbību izstrādes, starp darbībām ir iespējams pievienot atgriezto vai atlasīto vērtību pārbaudi pret sagaidāmajām vērtībām. 2.9. attēlā ir redzams procedūras piemērs, kur tiek pārbaudīts vai funkcijas "betwnstr" atgrieztais rezultāts sakrīt ar "2345".

```
procedure basic_usage is
begin
  ut.expect( betwnstr( '1234567', 2, 5 ) ).to_equal('2345');
end;
```

*2.9. att.* ***utPLSQL*** **testa procedūras piemērs ar vērtības validāciju [UTPLS]**

Lai procedūra tiktu uzskatīta kā tests un paka tiktu uzskatīta kā testa paka, tām ir nepieciešams pievienot atbilstošās anotācijas, skat. 2.10. att.

```
create or replace package test_betwnstr as

  -- %suite(Between string function)

  -- %test(Returns substring from start position to end position)
  procedure basic_usage;
```

*2.10.* ***utPLSQL*** **testa paka ar anotācijām pakai un procedūrai, lai tās būtu izpildāmas [UTPLS]**

Pēc anotāciju pievienošanas un procedūru izstrādes, pakas testus var izpildīt testa logā, izpildot darbību "begin ut.run('test_betwnstr'); end;", kur "test_betwnstr" ir pakas nosaukums. Pēc testa izsaukšanas un izpildes datu izvades logā lietotājam tiks izvadīta informācija par katru izpildīto testu un tā rezultātu, uzrādot konkrētā testa informāciju, ja ir tāds, kas ir neveiksmīgs (skat. 2.11. att.).

```
Between string function
  Returns substring from start position to end position
  Returns substring when start position is zero (FAILED - 1)

Failures:

  1) zero_start_position
      Actual: '123456' (varchar2) was expected to equal: '12345' (varchar2)
      at ""UT3_USER.TEST_BETWNSTR"", line 10

Finished in .232584 seconds
2 tests, 1 failed, 0 errored, 0 disabled, 0 warning(s)
```

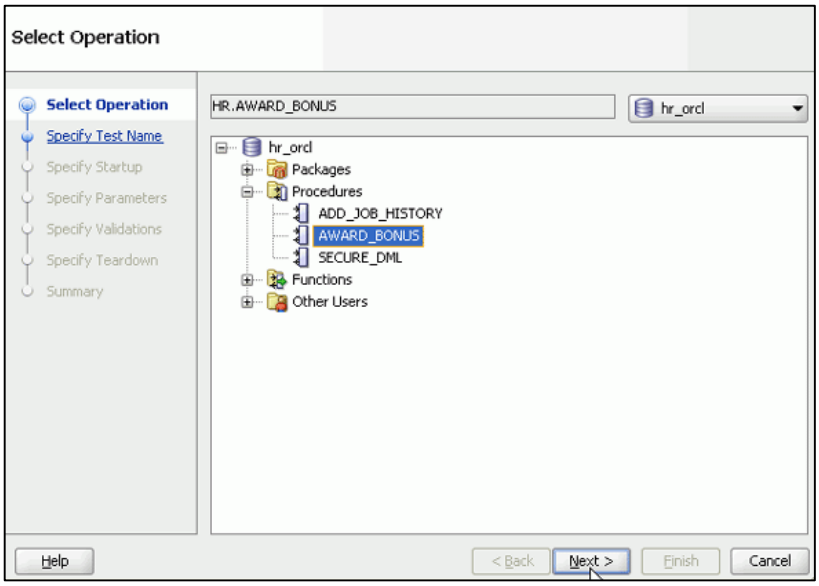
*2.11. att.* ***utPLSQL*** **testa atgrieztais rezultāts datu izvades logā [UTPLS]**

### SQL Developer Unit Testing

Ja datu bāzes koda izstrādei tiek izmantota datu bāzu pārvaldības sistēma *SQL Developer*, tad ir iespējams izmantot tajā iebūvēto vienību testu izstrādes funkcionalitāti. Gadījumos, kad tiek izmantota cita datu bāzu vadības sistēma, *SQL Developer* vienību testus var izmantot jebkurā gadījumā, tomēr tad būtu nepieciešams strādāt ar divām programmām – viena koda izstrādei, bet otra – koda testēšanai.

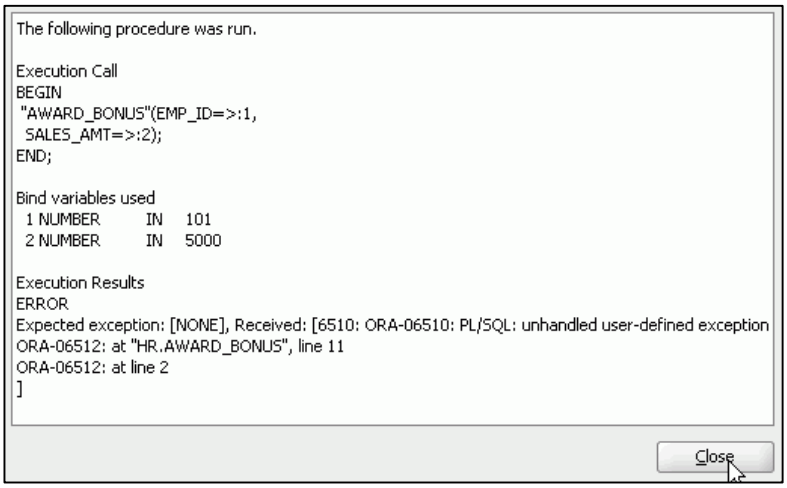
Vienību testus *SQL Developer* ir paredzēts izstrādāt uz lietotāja "unit_test_repos", kas ir atšķirīgs no datu bāzes koda rakstīšanai izmantotā lietotāja. Pieslēdzoties šim lietotājam, var izvēlēties veidot jaunu testu, un programmatūras uzrādītajā logā (skat. 2.12. att.) izvēlēties procedūru, kuru testēt, izvēloties testa nosaukumu. Ir iespējams norādīt darbības, ko veikt pirms un pēc testa izpildes, piemēram, nokopēt ietekmēto rindu uz citu tabulu pirms testa un atjaunot rindas saturu pēc testa. Testējamai procedūrai ir jānorāda visi ieejošie un izejošie parametri, to tips un vērtības ar kurām testu izpildīt. Vienam testam ir iespējams izveidot vairākas

implementācijas ar dažādiem ieejas parametriem. Testam var norādīt sagaidāmo rezultātu, piemēram, nesaņemt neko vai arī saņemt izņēmumu (exception) ar konkrētu numuru [ORAUT].




*2.12. att.* ***Oracle SQL Developer*** **vienību testa izveides logs ar testējamā objekta izvēli [ORAUT]**

Tad, kad testi ir izstrādāti, tos ir iespējams izpildīt, saņemot pretī informāciju par testa izpildes procesu un rezultātiem. Ja tests ir izpildījies neveiksmīgi, tas tiek paziņots lietotājam (skat. 2.13. att.).




*2.13. att.* ***Oracle SQL Developer*** **vienību testa atgrieztais rezultāts [ORAUT]**

Ja ir nepieciešams izmainīt testu, izmaiņas ir veicamas caur lietotāja saskarni, nevis caur testa kodu (kā tas ir *utPLSQL*) [SBRFU]. Viens vienību tests var notestēt tikai vienu procedūru vai funkciju, kas nozīmē, ka nav iespējams testēt scenārijus, kuros tiek izsauktas vairākas secīgas procedūras. Testu nav arī iespējams izstrādāt caur kodu, kas nozīmē ierobežotas darbības testu izstrādē un pārbaužu veikšanā.

#### 2.6.2.3. Rīku priekšrocības un trūkumi

Vienību testu izstrādei, testus rakstot ar roku, priekšrocības ir iespēja tos izstrādāt, kādus vēlas. Testos ir iespējams ievietot saglabāšanas punktu, uz kura izpildes brīdi ir iespējams atjaunot datu bāzes datus jebkurā testa brīdī. Testu formātu var izvēlēties, kādu nepieciešams, un vienā testā var iekļaut vairākus apakštestus. Vienību testu trūkumi ietver nepieciešamību visus testus rakstīt ar roku. Tas nozīmē, ka ar roku jāizstrādā ne tikai pašus testus, bet arī salīdzināšanas darbības, kuras visticamāk ir jāizpilda pēc katras izsauktās darbības. Lai izpildītu visus izstrādātos testus, ir jāizstrādā atsevišķa procedūra, kurā secīgi tiks izsaukti visi izstrādātie testi. Ja nepieciešams testa izpildītājam paziņot testa izgāšanās iemeslu, ir jāizstrādā atkļūdošanas informācijas izvadīšana uz ekrāna.

*utPLSQL* priekšrocības ietver jau sagatavotas testu funkcijas, kuras var izmantot, salīdzinot divas vērtības. Ir iespējams salīdzināt arī sarežģītākas konstrukcijas, piemēram, kursorus. Ar vienas rindiņas palīdzību ir iespējams palaist visus atzīmētos testus visās atzīmētajās pakās vai vienā izvēlētā pakā. Pēc katra testa izpildes automātiski tiek veikta transakcijas atrite, kas nodrošina datu bāzes satura neskaršanu, izpildot testu, jo dati tiks atjaunoti sākotnējā stāvoklī. Pēc testa izpildes ir piekļuve katra testa izpildes statusa informācijai, kā arī darbību, kuras izpildījušās neveiksmīgi, aprakstam. Risinājuma trūkumi ir nepieciešamība ar roku ņemt nost anotācijas testiem, kurus nav nepieciešams izpildīt gadījumos, kad tiek izpildīti visi izstrādātie testi.

*Oracle SQL Developer* automatizēto testu izstrādei nav nepieciešamas programmēšanas zināšanas, jo viss notiek caur lietotāja saskarni kas ir priekšrocība. Tomēr risinājumam ir vairāki trūkumi, to skaitā testu izstrādes ierobežojumi – testus nevar rakstīt ar roku, kas nozīmē samazinātu izstrādājamo testu diapazonu, kā arī viens tests var pārbaudīt tieši vienu vienību datu bāzē. Ja kods netiek izstrādāts uz *SQL Developer* programmas, tad ir jāizmanto atsevišķa programma testu izstrādei un atsevišķa - koda izstrādei.

## 2.7. Kopsavilkums

Automatizējot testus, ir svarīgi, lai tie būtu maksimāli īsi, nav atkarīgi viens no otra, kā arī ir atkārtojami. Automatizētos testus ir vēlams ieviest tad, kad sistēmai ir paredzamas mazas izmaiņas esošajā funkcionalitātē, citādi ir nepieciešams regulāri atjaunināt testus, ja tiek mainīta sistēmas funkcionalitāte.

Lai izstrādātu automatizētos testus, nepietiek ar kāda rīka paņemšanu un mēģinājumu uzreiz veidot uz tā testus. Pirms testa izvēles izstrādes komandai nepieciešams kopīgi izlemt, kādas ir automatizētās testēšanas gaidas, un, balstoties uz tām, kopīgi jāizvēlas atbilstošākais

rīks. Ja testus paredzēts izstrādāt vairākām personām, tad ir jāvienojas par testu izstrādes vadlīnijām, lai testu kodā nevaldītu haoss.

Ir noderīgi kombinēt lietotāja saskarnes un datu bāzes vienību testus, jo nav iespējams ar vienu no tiem notestēt gan katru funkciju, gan lietotāja saskarni.

Testu automatizācijas ietvara izvēle ir atkarīga no sistēmas izmēra un veida, tomēr, ja tiek izvēlēts gatavs rīks, tad tas visticamāk jau ir balstīts uz kādu no aprakstītajiem ietvariem. Ja rīks ir balstīts uz kādu zemāka līmeņa ietvaru, piemēram lineāro, tad dažos gadījumos ir iespējams veikt labojumus un mainīt tā ietvaru uz augstāka līmeņa. Piemērs šim gadījumam var būt atsevišķa faila izveide, kurā tiek uzglabāti mainīgie, kurus izmantot testā, un šo mainīgo nolasīšana pirms testu izpildes.

Rīkus, kas izmanto "Ierakstīt un atskaņot" principu, ir parocīgi izmantot, ja nav programmēšanas zināšanu, un ja testus izstrādā testētāji, nevis programmētāji. Tomēr šiem testiem ir savi trūkumi – var būt brīži, kad šī metode ieraksta pārāk daudz darbības (piemēram, peles kustināšanu vai gaidīšanu pirms kādas darbības veikšanas), kā arī dažreiz rīki var neatbalstīt darbības, kuras nepieciešams ierakstīt, un tiek ierakstīts pārāk maz informācijas. Mēdz būt gadījumi, kad testu nav iespējams atkārtot identiski, jo ir mainījušies elementu identifikatori, pēc kuriem tests nosaka uz kura objekta ir jāspiež. Testus, kas veidoti pēc šī principa, ir grūti uzturēt – ja kaut kas mainās sistēmā, tad visticamāk testus būs nepieciešams atkārtoti ierakstīt, ja netiek nodrošināta piekļuve uzģenerētajam kodam [LBTRT], savukārt ar piekļuvi uzģenerētajam kodam var būt problēmas kaut ko mainīt, ja kods nav veidots vienkāršā un saprotamā valodā.

Lietotāja saskarnes testu izstrādei bez programmatūras zināšanām visatbilstošākais variants ir izmantot *Katalon Studio*, jo ar to ir iespējams izstrādāt testus bez koda izstrādes, un tie piedāvā pietiekami lielu testējamo darbību klāstu. Tomēr priekš izstrādātājiem visatbilstošāk ir izmantot elastīgāku testu izstrādes risinājumu, kur katru darbību var izstrādāt tieši tādu, kāda nepieciešama līdz vissīkākajai detaļai. Šādai situācijai no apskatītajiem variantiem visatbilstošāk ir izmantot *Visual Studio* programmatūru ar *Selenium* pakotņu nodrošināto kodu kā pamatu savu testu rakstīšanai.

Gadījumos, kad sistēma ir vienkārša, testējamo objektu nav daudz un nav sarežģītu scenāriju sistēmas izmantošanā, datu bāzu vienību testus var izstrādāt uz *Oracle SQL Developer* programmas piedāvātā rīka. Šajā rīkā testu izstrāde notiek caur veidni, kur, nerakstot kodu, var izvēlēties testējamos objektus un ievadīt visas nepieciešamās vērtības. Sistēmām ar sarežģītiem algoritmiem un testējamajām konstrukcijām parocīgāk ir izmantot *utPLSQL* spraudni, kas ļauj izstrādāt testus procedūras veidolā ar neierobežotām darbībām starp pārbaudes reizēm. Izmantojot šo spraudni, ir iespējams izstrādāt testus, kas vadās pēc lietošanas scenārijiem un

vienā testā veic vairākas secīgas darbības, un darbību starpposmos pārbauda rezultātu. Manuālā transakcijas atrite ļauj ar vairākiem secīgiem testiem notestēt ne tikai individuālas darbības kā vienību testus, bet arī notestēt lietojuma scenārijus, kas sastāv no vairākām secīgām darbībām, padarot ar *utPLSQL* izstrādātos vienību testus par funkcionālajiem integrācijas testiem.

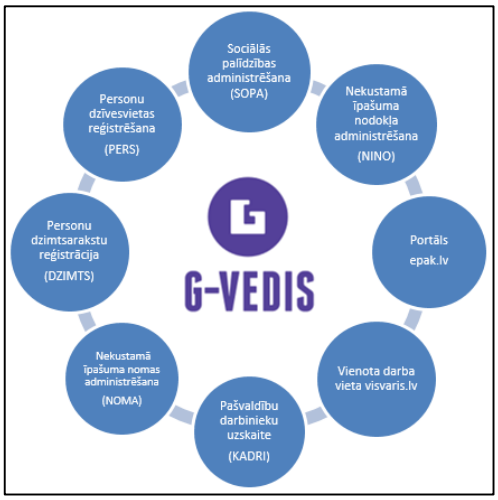
# 3. RVS G-VEDIS ALGU APRĒĶINA MODUĻA DARBA LAIKA UZSKAITES TABULU FUNKCIONALITĀTE

Darba ietvaros ir plānots ieviest automatizēto testēšanu resursu vadības sistēmas G-VEDIS Algu aprēķina moduļa darba laika uzskaites funkcionalitātei. Lai rastu priekšstatu par sistēmu kopumu, kurā ietilpst testējamā sistēma, šajā darba daļā autors apskata *ZZ Dats* izstrādāto Vienoto pašvaldību sistēmu un resursu vadības sistēmu G-VEDIS, kas ietilpst Vienotajā pašvaldību sistēmā. Daļā ir apskatīts arī Algu aprēķina modulis, kas ir G-VEDIS sistēmas komponente, un tajā ietilpstošā darba laika uzskaites funkcionalitāte. Darba daļā tiek apskatīta sistēmu uzbūve, sistēmu nodrošinātā funkcionalitāte, izmantotās tehnoloģijas, rīki un risinājumi. Tiek apskatīts Algu aprēķina moduļa versiju dzīves cikls, kā arī apskatīti sistēmai atbilstošākie automatizācijas varianti, kuri tiks realizēti sistēmā darba ietvaros.

## 3.1. Vienotā pašvaldību sistēma un VISVARIS

Lai atvieglotu pašvaldību iekšējos un starpiestāžu sadarbības procesus, ZZ Dats ir izstrādājis savstarpēji integrētu IT risinājumu visām pašvaldībām – Vienoto pašvaldību sistēmu (turpmāk tekstā VPS) [ZZRIS].

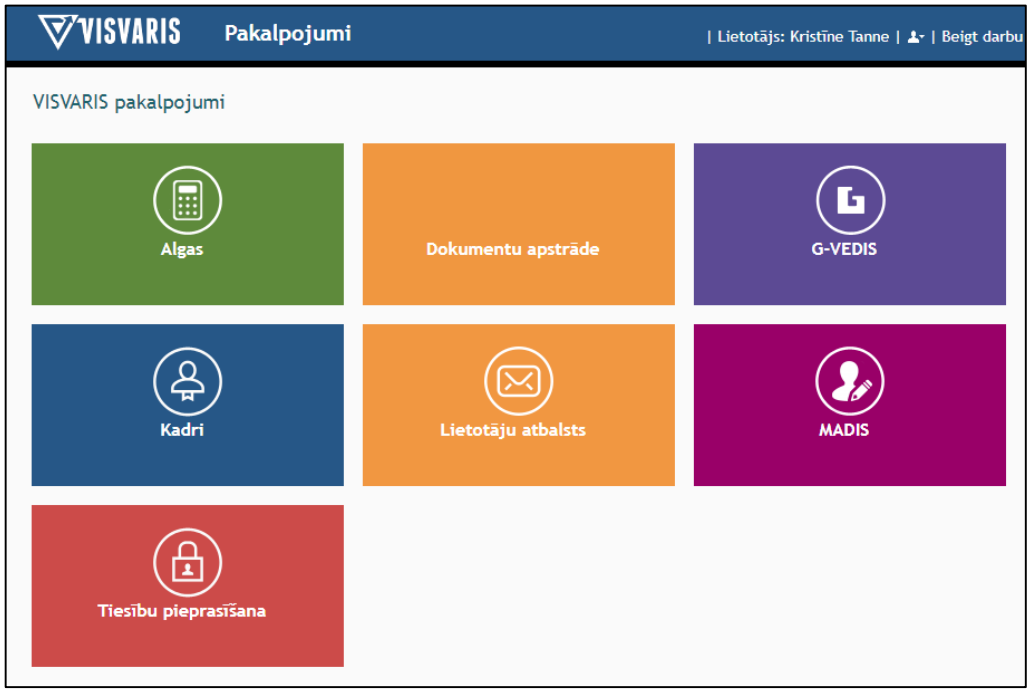
VPS sastāv no vairākām lietojumprogrammām, piemēram, nekustamā īpašuma nodokļa administrēšanas (NINO), sociālās palīdzības administrēšanas (SOPA), personu dzimtsarakstu reģistrācijas (DZIMTS), personu dzīvesvietas reģistrēšanas (PERS), nekustamā īpašuma nomas administrēšanas (NOMA), administratīvo pārkāpumu uzskaites un kontroles (APUS) un resursu vadības sistēmas G-VEDIS (skat. 3.1. att.) [ZZRIS].




*3.1. att.* **VPS risinājuma uzbūves shēma**

VPS ir integrēta ar daudziem valsts nozīmes reģistriem - iedzīvotāju, uzņēmumu, kadastra, adrešu u. c., kā arī ar citām informācijas sistēmām — Valsts kasi, komercbankām, Valsts ieņēmumu dienestu, Valsts izglītības informācijas sistēmu u. c. VPS ir izveidota ar mērķi nodrošināt integrētu informācijas tehnoloģiju atbalstu pašvaldību funkciju biznesa procesiem. Sistēma ir piemērota visām pašvaldībām centralizētas e-pārvaldes izveidei [ZZRIS].

Daļa no VPS lietojumprogrammām ir pieejama caur tīmekļa pārlūkprogrammu. Visas pieejamās lietojumprogrammas ir apvienotas vienā resursā www.visvaris.lv (turpmāk tekstā VISVARIS). VISVARIS strādā kā vienota darba vide VPS apakšsistēmām, kuras ir izstrādātas tīmekļa vidē. VISVARIS tiek attīstīts kā centralizēts risinājums un tam ir vienota piekļuve tīmekļa vietnē. Pieslēdzoties, lietotājam tiek piedāvāti pieejamie pakalpojumi, kurus var izmantot (skat. 3.2. att.).




*3.2. att.* **VISVARIS pakalpojumi**

Lietotājam pieejamo pakalpojumu klāsts ir atkarīgs no piešķirtajām tiesībām. VISVARIS lietotājiem nodrošina vienotu autentifikāciju (vienu reizi pieslēdzoties un pārslēdzoties uz citu pakalpojumu, lietotājs paliks pieslēgts sistēmā).

## 3.2. RVS G-VEDIS

Viena no VPS lietojumprogrammām ir finanšu un resursu vadības sistēma G-VEDIS (turpmāk tekstā G-VEDIS). G-VEDIS ir informācijas sistēma resursu vadībai un grāmatvedības uzskaitei. Sistēma nodrošina pilnu grāmatvedības ciklu, sākot ar pirmdokumentu ievadi un beidzot ar pārskatu sagatavošanu. Tā ir specializēta pašvaldībām un to pakļautības iestādēm.

G-VEDIS nodrošina vienotu pašvaldības un tās padotības iestāžu finanšu un grāmatvedības datu uzskaiti. Visās iestādēs uzskaite tiek veikta pēc vienotas, pašvaldībā apstiprinātas metodikas [ZZRVG]. G-VEDIS strādā kā datora programmatūra, kas izmanto klienta – servera modeli. Tā ir lietotņu struktūra, kas sadala procesus starp serveriem un pakalpojuma pieprasītājiem, jeb lietotājiem [WICSM]. Izmantojot G-VEDIS tīmeklī (VISVARĪ), ir iespējams paplašināt tā funkcionalitāti. G-VEDIS programmatūra tiek uzturēta klientu datorā, bet dati tiek ņemti no serveriem, kas tiek glabāti mākonī.

G-VEDIS sastāv no vairākām lielām daļām, to skaitā ēdināšanas plānošanas modulis, PII apmeklējumu uzskaites modulis, personāla uzskaites modulis KADRI, Algu aprēķina modulis un Algu aprēķina moduļa darba laika uzskaites tabulu aizpildes apakšmodulis.

## 3.3. Algu aprēķina modulis un darba laika uzskaites tabulu funkcionalitāte

Algu aprēķina modulis ir ļoti nozīmīga G-VEDIS sastāvdaļa. Modulis ir VISVARIS sastāvdaļa, un ir pieejams ikvienam klientam, izmantojot tīmekļa pārlūkprogrammu.

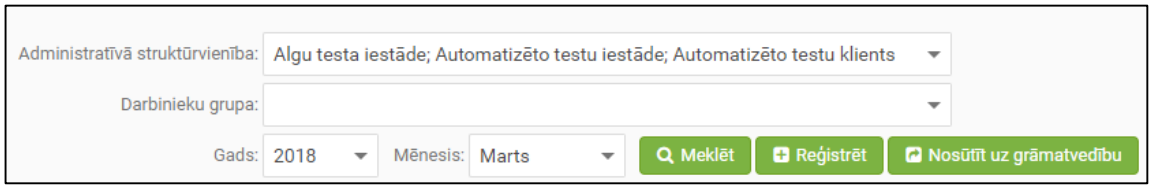
Darba laika uzskaites tabulu aizpildes funkcionalitāte (turpmāk tekstā DLUT) ir daļa no Algu aprēķina moduļa, kas ir izdalīta no pamata funkcionalitātes un tiek uzturēta atsevišķā modulī. Darba laika uzskaites funkcionalitāte ir izdalīta no Algu aprēķina moduļa, jo ar to pamatā strādā personāla speciālisti un struktūrvienību vadītāji, kuru darba pienākumi ir darba laika uzskaites tabulu aizpilde.

DLUT datus darba laika uzskaitei saņem no personāla uzskaites un algu grāmatvedības bloka KADRI, tos apstrādā, un nosūta uz grāmatvedību Algu aprēķina modulī, kur tie tālāk tiek apstrādāti, apstiprināti un izmantoti darbinieku algu aprēķinā. Modulī KADRI darbiniekiem tiek ievadītas slodzes, amati un prombūtnes, kas tiek ņemtas vērā, reģistrējot datus DLUT. Prombūtņu, to skaitā darba nespējas lapu informācija tiek iegūta no e-veselības, bet prombūtnes ir iespējams ievadīt arī manuāli.

Darba laika uzskaites tabulu sadaļā ir iespējams sagatavot darba laika uzskaites tabulas darbiniekiem. Dati tiek sagatavoti uz darbinieka amatu, un tiem pēc sagatavošanas ir iespējams precizēt stundas, ievadīt nakts darba stundas, aizvietošanas stundas, virsstundas, kā arī stundas darbam svētku dienās un brīvdienās. Pēc labojumu veikšanas datus var nosūtīt uz Algu aprēķina moduli, kur algu grāmatveži tos tālāk izmanto algu aprēķinā. Nosūtītajai informācijai ir iespējams veikt korekcijas un atkārtoti iesniegt precizējumus grāmatvedībai.

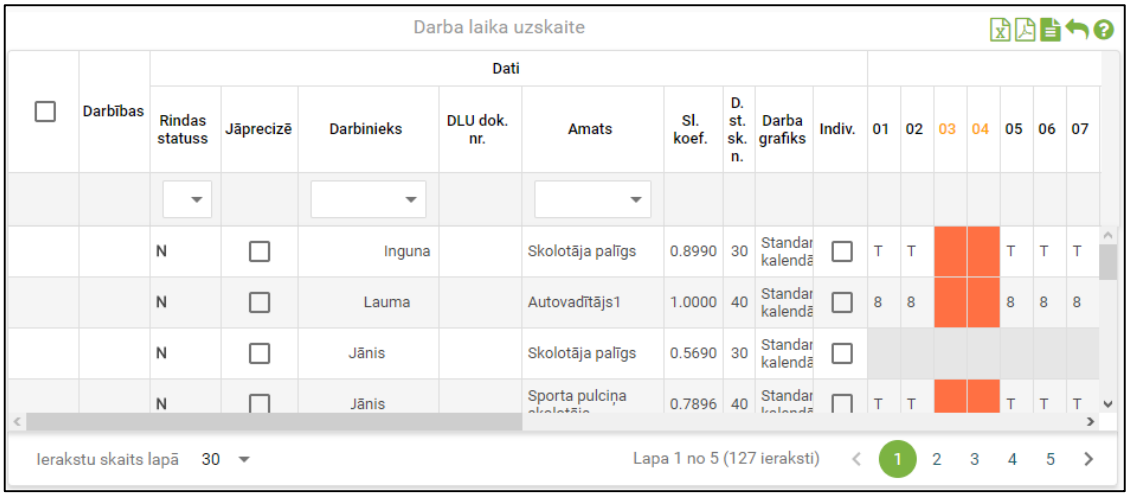
Lai rastu priekšstatu par funkcionalitāti, kurai nepieciešams izstrādāt automatizētos testus, turpmāk šajā nodaļā ir apskatīta darba laika uzskaites tabulu funkcionalitāte no lietotāja saskarnes perspektīvas.

Datus darba laika uzskaites tabulām ir iespējams atlasīt pēc obligātiem meklēšanas kritērijiem "Administratīvā struktūrvienība", "Gads", "Mēnesis", un pēc neobligāta meklēšanas kritērija "Darbinieku grupa" (skat. 3.3. att.).




*3.3. att.* **Darba laika uzskaites tabulu saraksta atlases meklēšanas kritēriji**

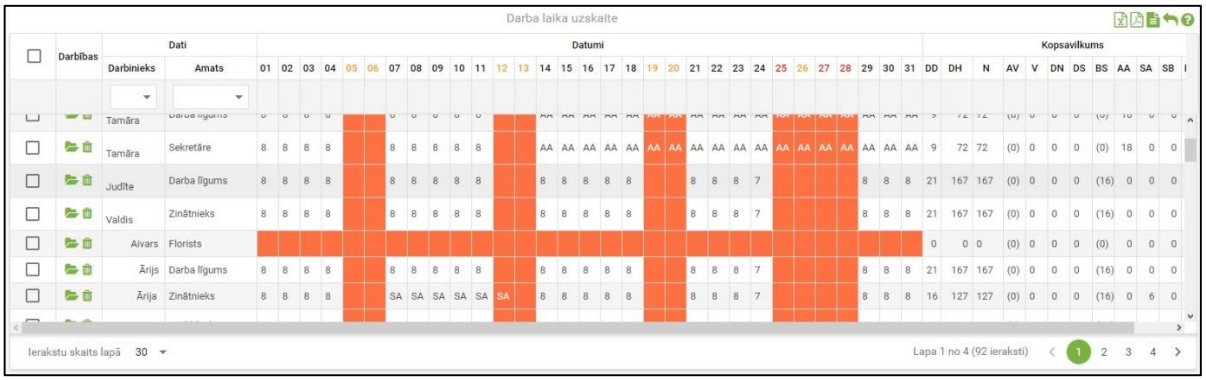
Spiežot pogu "Meklēt", sarakstā (skat. 3.4. att.) tiek atainoti ieraksti, kas ir atrasti pēc ievadītajiem meklēšanas kritērijiem. Šie ieraksti tiek parādīti dinamiski, un datu rindas nav tieši reģistrētas datu bāzē. Lietotājam ir iespējams eksportēt attēloto sarakstu .xls vai .pdf formātā, pieprasīt lietotāja instrukcijas lejupielādi, sarakstam var slēpt, uzrādīt un mainīt secību kolonnām, kārtot pēc jebkuras kolonnas, filtrēt datus pēc atlasīto datu rindu statusa, darbinieka vārda un amata, kā arī atjaunot kolonnu noklusētos iestatījumus.




*3.4. att.* **Darba laika uzskaites saraksts ar nereģistrētām rindām**

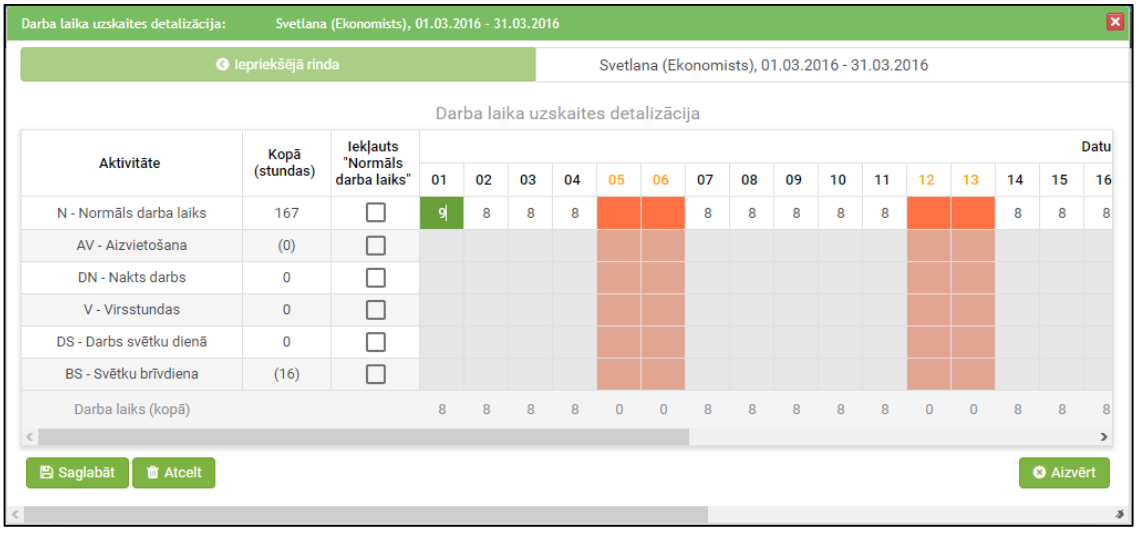
Sarakstā redzamos, bet nereģistrētos ierakstus (skat. 3.4. att.) ir iespējams piereģistrēt datu bāzē, izmantojot reģistrācijas pogu. Pēc ierakstu reģistrācijas (skat. 3.5. att.), lietotājam ir pieejama rindu detalizācijas atvēršana, kur ir iespējams labot nostrādātās stundas (skat. 3.6. att.). Ierakstu reģistrācijas brīdī stundas tiek aizpildītas no amatam norādītā standarta darba grafika (katrai dienai ir norādītas paredzamās darba stundas). Sarakstā ir apskatāmas ne tikai reģistrētās stundas katrā datumā, bet arī prombūtnes kods, ja kādā no datumiem ir prombūtne,

kā arī kopsavilkums katram aktivitātes veidam un prombūtnēm. Kopsavilkumā tiek uzrādīta ievadīto stundu kopsumma katram aktivitātes veidam un no darba grafika ņemtu plānoto stundu kopsumma prombūtnēm.




*3.5. att.* **Darba laika uzskaites saraksts ar reģistrētām rindām**

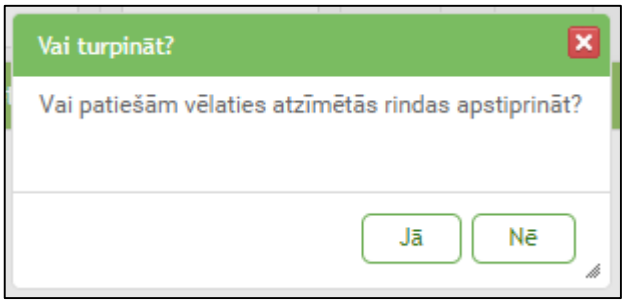
Atverot vienas rindas detalizāciju, ir iespējams labot reģistrētās stundas katram aktivitātes veidam, un labojumus saglabāt, kā arī var pārslēgties uz citām sarakstā esošajām rindām. Dienas, kurās veikti labojumi pirms saglabāšanas, tiek iekrāsotas zaļā krāsā (skat. 3.6. att.). Detalizācijā ir iespējams ievadīt arī virsstundas, nakts darbu, darbu svētku dienā un darbu citos aktivitātes veidos. Ja darbiniekam periodā ir prombūtne, tad tā tiek ņemta vērā, reģistrējot datus, un detalizācijā prombūtnes datumos darba laiks reģistrēts netiek. Katra prombūtne, kas ietilpst periodā, tiek pievienota kā jauna rinda detalizācijā, un ir iespējams redzēt plānotās darba stundas tās laikā.




*3.6. att.* **Darba laika uzskaites rindas detalizācijas skats ar nesaglabātām izmaiņām 1. datumā**

Pēc labojumu veikšanas rindās, izvēlētām rindām var izpildīt darbību "Apstiprināt". Apstiprinātās rindas tālāk ir iespējams nosūtīt uz grāmatvedību, Algu aprēķina modulī automātiski izveidojot jaunu darba laika uzskaites dokumentu, kurā visas izvēlētās un nosūtītās

rindas tiek pievienotas. Izvēlētas rindas ir iespējams arī aizpildīt pēc plāna, atcelt apstiprinājuma statusu vai uzlikt to, kā arī rindas ir iespējams dzēst. Pēc dzēšanas rindas tiek uzrādītas kā nereģistrētas (skat. 3.4. att.) un tās var atkārtoti reģistrēt.




*3.7. att.* **Darba laika uzskaites rindu apstiprināšanas brīdinājuma logs**

Izpildot darbības "Dzēst", "Apstiprināt", "Atcelt apstiprinājumu" un "Aizpildīt pēc plāna", pirms darbības izpildīšanas lietotājam tiek uzrādīts apstiprinājuma logs, kur darbību var arī atcelt (skat. 3.8. att.).

Tad, kad rinda ir nosūtīta uz grāmatvedību, tai ir iespējams noņemt apstiprinājuma statusu, bet tādā gadījumā lietotājam tiks uzrādīts brīdinājums "*1 rinda, kurai tika noņemts apstiprinājums, ir nosūtīta uz grāmatvedību. Pēc labojumu veikšanas, atkārtoti sūtot uz grāmatvedību, tiks veidots precizējums!*" – tas nozīmē, ka, ja rindai labo stundas kādā no aktivitātes veidiem, tad rindai tiks uzlikta pazīme "Jāprecizē" un darbība "Nosūtīt uz grāmatvedību" labos iepriekš nosūtīto rindu, ja tā nav aiztikta, vai arī veidos precizēto rindu, ja tā ir izmantota algu aprēķināšanai.

## 3.4. Izmantotās tehnoloģijas un rīki

Ir divu veidu VPS moduļi – datora programmatūra un tīmekļa vietnes. Visiem moduļiem ir vairākas kopīgi izmantojamas datu bāzes, tomēr katrs modulis izmanto konkrētu tam paredzētu datu bāzi. Ir arī kopīga funkcionalitāte, tā skaitā lietotāju reģistrācija un autorizācija, ieraksta izmaiņu vēstures uzglabāšanas modulis, u.c.

VPS arhitektūrai raksturīga iezīme ir datu bāzu pārvaldības sistēmas *Oracle* izmantošana datu uzglabāšanas un apstrādes vajadzībām. Oracle datu bāzes kods tiek izstrādāts un uzturēts integrētās izstrādes vidē *PL/SQL Developer*. VPS moduļu saziņu starp sistēmām un *Oracle* tabulām, veicot datu atlasi, labošanu vai atskaišu izguvi, nodrošina glabātās procedūras. Procedūras tiek uzglabātas pakās, kas palīdz tās sagrupēt un pārskatīt, vienā pakā uzturot procedūras un funkcijas darbībām ar vienu tabulu vai tabulu grupu. Procedūrām un funkcijām tiek padoti ievaddati, kurus izmantojot, tiek veidoti un izpildīti vaicājumi, veiktas datu ievietošanas, labošanas, dzēšanas un atlases darbības.

RVS G-VEDIS saziņai starp lietotāju un datu bāzi izmanto *Visual FoxPro* programmatūru, kurā vaicājumi tiek izveidoti pirms pieprasījuma sūtīšanas uz datu bāzi, un datu bāzē tiek tikai izpildīts saņemtais vaicājums. *Visual* FoxPro Programmatūras piedāvājumā ir iespēja datus no datu bāzes ielasīt *Visual FoxPro* objektos – lokālajos kursoros, kurus tālāk var apstrādāt klienta datorā [MVFPH]. G-VEDIS izmanto šos piedāvātos kursorus, tajos ielasot *Oracle* vaicājuma atgrieztos datus. G-VEDIS ir Windows datoru lietojumprogramma, kas nodrošina iespēju lietotājiem darboties vienlaicīgi ar vairākiem atvērtiem logiem, kur katrā ir cits saraksts vai forma. Tomēr, kamēr vienā logā tiek izpildīta kāds process (datu izguve, labošana, u.c.), pārējie logi tiek padarīti neaktīvi. Tas nozīmē, ka lietotājs vienlaikus var veikt tikai vienu paralēlu darbību.

Tīmekļa lietotnēm salīdzinājumā ar datora programmatūru ir savas priekšrocības – ir vienkāršāk uzturēt programmatūru, jo jaunu versiju piegāde tiek nodota visiem uzreiz un nav gadījumu, kad kāds izmanto sistēmu ar vecāku versiju, kurā ir vēl neizlabota ievainojamība, kamēr pārējiem lietotājiem ir jaunākā programmatūras versija. Tīmekļa vietne arī piedāvā darboties vienlaicīgi ar vairākiem atvērtiem logiem, tomēr atšķirība starp datora programmatūru ir spēja pieprasīt vairākas darbības izpildīt vienlaikus, ja tās tiek izsauktas no dažādām cilnēm.

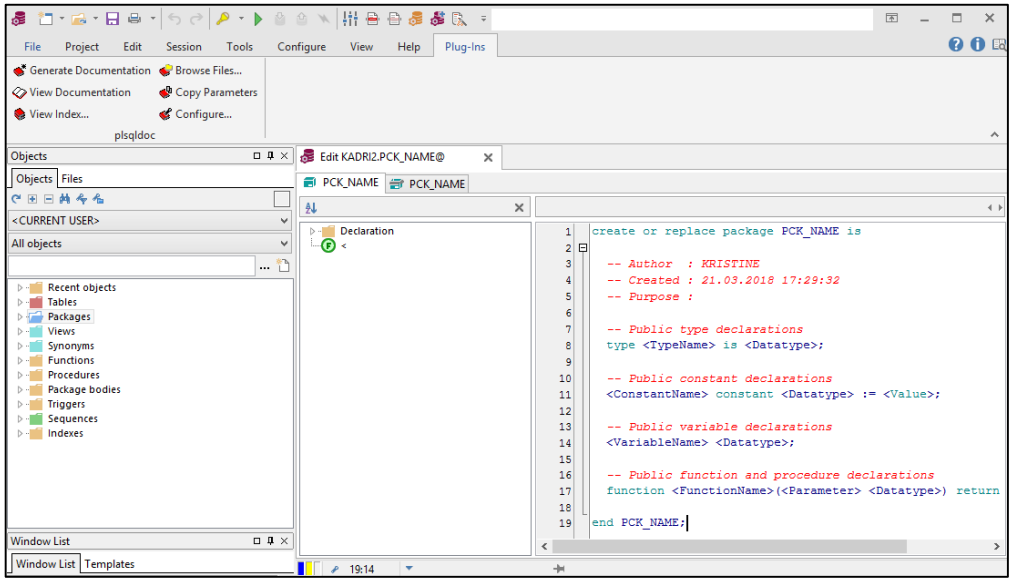
Datu atlases un apstrādes (pievienošanas, labošanas) procedūras moduļiem, kas balstīti uz tīmekļa vietni, notiek tikai Oracle datu bāzē, savukārt uz datora programmatūrām balstītiem moduļiem šīs darbības notiek gan datu bāzē, gan arī caur programmatūru (lietojumprogrammatūras kodā pa taisno izpilda darbības, kas veic datu manipulācijas datu bāzē).

Ir svarīgi apzināties testējamās sistēmas izstrādē izmantotās tehnoloģijas un rīkus, jo tie nosaka ierobežojumus sistēmā. Tā kā Algu aprēķina modulis un darba laika uzskaites tabulas ir tīmekļa lietotnes, tās sastāv no divām daļām – lietotāja saskarne un datu bāze. Sistēma tiek izstrādāta uz divām pamata programmatūrām – *PL/SQL Developer* (datu bāzei), un *Microsoft Visual Studio* (aplikācijai). Pati sistēma strādā kā tīmekļa lietotne, kas nozīmē, ka klientiem nav nepieciešams datorā instalēt programmatūru, lai lietotu sistēmu.

### 3.4.1. Datu bāzu tehnoloģijas un rīki Algu aprēķina modulī

*PL/SQL Developer* (skat. 3.9. att.) ir integrētā izstrādes vide (IDE), kas paredzēta programmatūras izstrādei *Oracle Database* vidē, izmantojot *PL/SQL* programmēšanas valodu [AAAPS]. Datu bāzē ir pakas, kuras satur procedūras un funkcijas. Objektu sadalījums pakās palīdz nodrošināt, ka objekti ir sagrupēti pēc to funkcionalitātes, tas nozīmē, ka

procedūras, kas veic darbības ar darba laika uzskaites funkcionalitātē izmantojamajiem datiem un objektiem, atrodas vienā pakā.




*3.8. att.* ***PL/SQL Developer*** **programmatūras lietotāja saskarne ar jaunas pakas veidošanas logu**

Procedūras Algu aprēķina modulī sastāv no sekojošiem galvenajiem veidiem:

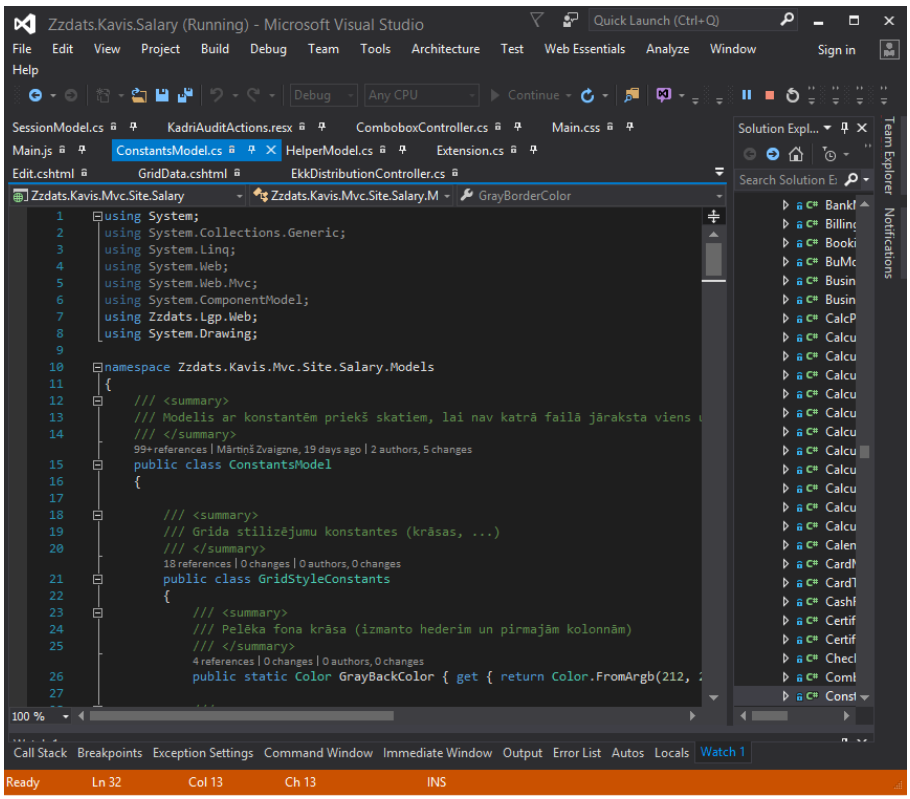
a) datu atlase – izmantojot saņemtos ievades parametrus, tiek izpildīts vaicājums un kursorā atgriezts vaicājuma rezultāts. Rezultāts var būt saraksts ar vairākām rindām vai arī dati vienā rindā, piemēram, viena ieraksta dati vai viena aprēķināta vērtība,

b) datu labošana, dzēšana, aprēķināšana vai citas datu apstrādes darbības. Rezultātā var atgriezt kļūdas paziņojumu, ja tāds ir, kā arī atsevišķos gadījumos atgriež izveidotā ieraksta identifikatoru vai citus pieprasītos datus kā individuālas vērtības.

Datu bāzē tiek uzglabātas arī funkcijas, bet lielākoties tās ir paredzētas izmantošanai datu bāzē, piemēram, vaicājumā katrai rindai kādā kolonnā uzrādīt funkcijas rezultātu.

### 3.4.2. Lietotnes tehnoloģijas un rīki Algu aprēķina modulī

*Microsoft Visual Studio* (skat. 3.10. att.) ir integrētā izstrādes vide, kas ir paredzēta datora programmatūras, mājas lapu un lietotņu izstrādei [WIMVS]. Algu aprēķina moduļa lietotnes izstrādei tiek izmantota programmēšanas valoda *C#* (kopā ar *ASP.NET MVC*), kā arī *HTML*, *CSS* un *JavaScript* lietotāja saskarnes izstrādei.




*3.9. att.* ***Microsoft Visual Studio*** **programmatūras lietotāja saskarne ar klases definīciju**

### MVC šablons

Sistēmas lietotne sastāv no 3 daļām, kā nosaka *MVC* (Model-View-Controller) šablons – modelis, skats, kontrolieris [ASPMO]:

a) Modelis atbild par datu izguvi no datu bāzes un par datu nodošanu datu bāzei. Modelī tiek nodefinēta katra datu objekta uzbūve ar laukiem, kuros glabā informāciju. Datus modelī ielasa objektos, veic ar tiem darbības, un pēc tam atjauninātu informāciju nosūta atpakaļ uz datu bāzi.

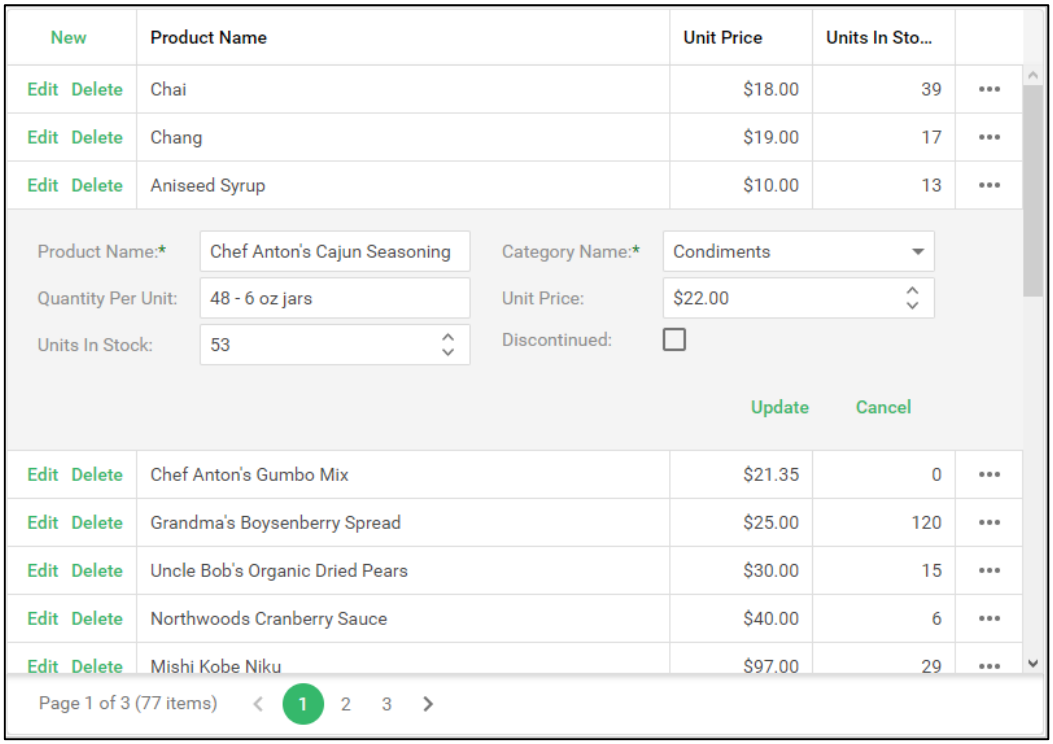
b) Skati ir sistēmas komponente, kas atbild par lietotnes lietotāja saskarnes uzrādi. Parasti lietotāja saskarne tiek izveidota no modeļa datiem, piemēram, ja modelis atlasa darbinieku datus, tad skatā tie tiek uzrādīti sarakstā un ar iespēju tos labot.

c) Kontrolieri apstrādā skata un modeļa mijiedarbību. Modelī atlasītie dati tiek nodoti kontrolierim, kas savukārt tos nodod konkrētam skatam. Pēc labojumu veikšanas skatā, dati tiek padoti uz kontrolieri, kas datus nodod modelim, kas savukārt var izmantot šos datus datu bāzes satura labošanai vai datu atlasīšanai.

### *DevExpress* ietvars vadīklām

Liela daļa Algu aprēķina moduļa sastāv no sarakstiem, kur tiek uzrādīti apstrādāti dati no datu bāzes. Lai lietotājam piedāvātu ērtu darbošanos ar datiem, kas tiek uzrādīti sarakstā, tiek

izmantots *DevExpress* ietvars. *DevExpress* ir trešās puses *ASP.NET* vadīklu nodrošinātājs, kas pielāgo *ASP.NET* vadīklu elementus, padarot tos elastīgākus un vienkāršāk lietojamus, nekā *ASP.NET* vadīklas [QWADX]. *DevExpress* piedāvā lielu vadīklu izvēli, to skaitā sarakstus (Grid, skat. 3.11. att.), ievades laukus, atskaites, diagrammas un citas.




*3.10. att.* ***DevExpress GridView*** **saraksta kontroles piemērs [DXGVD]**

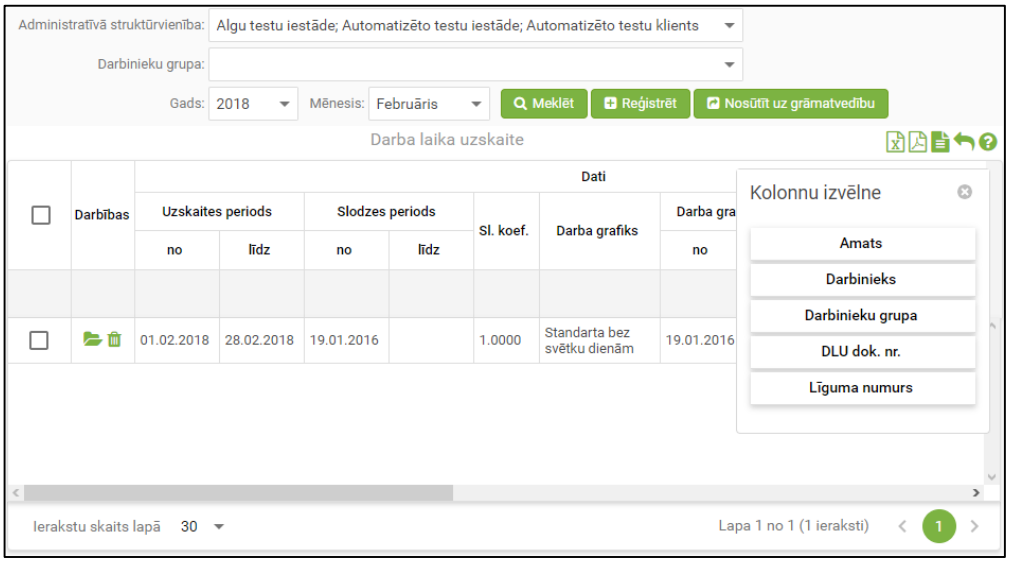
**Sistēmas saskarnes uzbūve**

Saskarne Algu aprēķina sistēmā parasti sastāv no divām galvenajām daļām – meklēšanas filtra un saraksta (skat. 3.12. att.). Meklēšanas filtrs sastāv no dažādu veidu ievaddatiem atkarībā no meklējamā informācijas veida, to skaitā datumiem, brīva formāta teksta laukiem, izkrītošajām izvēlnēm, kā arī datu izvēles caur jaunu logu, kurā meklējamo ierakstu var atlasīt pēc brīva teksta meklēšanas filtra. Meklēšanas filtrā pirms meklēšanas var izvēlēties, kā filtrēt datus, un pēc pogas "Meklēt" nospiešanas sarakstā tiek atlasīti dati, baltoties uz izvēlētajiem kritērijiem.

Sarakstā uzrādītās rindas ir iespējams iezīmēt un izpildīt atzīmētajām rindām apstrādes darbības. Gandrīz visiem sarakstiem ir iespējams personalizēt kolonnas, tās paslēpjot, pievienojot, kā arī kārtojot pēc izvēlētas kolonnas.

Sarakstā rindām, kuras ir iespējams labot, var atvērt vaļā labošanas formu, kurā tiek attēloti visi labojamie dati. Labošanas formā var mainīt vērtības ievades laukos un tikai pēc pogas "Saglabāt" nospiešanas dati tiek nosūtīti uz datu bāzi un saglabāti, kas nodrošina labojumu atcelšanu, ja labojumi veikti nekorekti vai neparedzēti. Dažiem sarakstiem nav datu

labošanas iespēja tieši rindā redzamajiem datiem, bet ir detalizācijas saraksta apskates iespējamība, kur var veikt detalizētāku datu koriģēšanu.

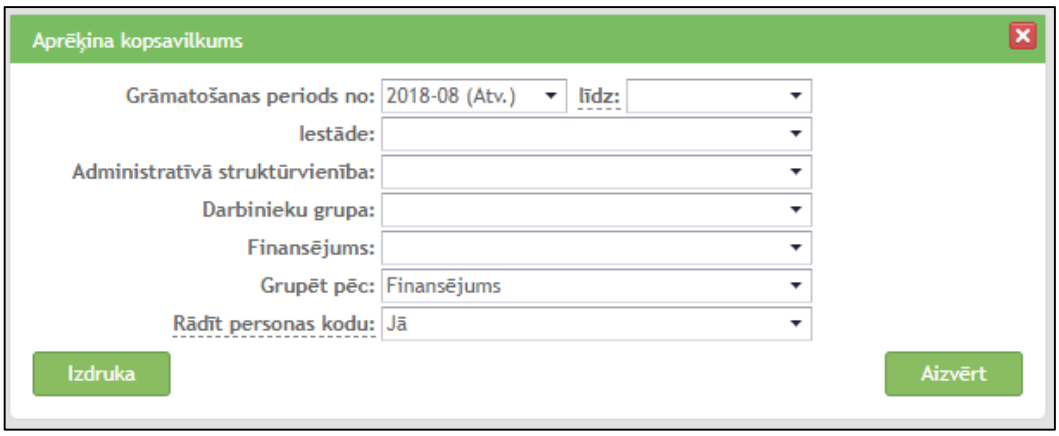



*3.11. att.* **Algu aprēķina moduļa saskarnes uzbūves piemērs ar meklēšanas filtru un sarakstu**

Datu labošanas forma VISVARIS pakalpojumos tiek realizēta divos veidos:

- kartīte, kur katrā rindā var labot viena ieraksta datus ne tikai uzrādītajām kolonnām, bet arī papildus citus datus, kas netiek uzrādīti sarakstā. Dažās sadaļas kartītē ir iespējams piekļūt arī detalizācijas vai saistīto datu skatam, kuram arī ir iespējams labot uzrādītos datus,
- iekļautā labošanas forma sarakstā, kas aizvieto sarakstā uzrādāmos datus ar ievades laukiem un ļauj labot datus vairākās rindās (ierakstos) uzreiz, bet tikai sarakstā uzrādītajām kolonnām.

Lietotājam nav iespējams izvēlēties datu labošanas veidu – tas ir realizēts katrā lietotnes sadaļā vienā no aprakstītajiem veidiem – atkarīgi no tā, kurš no veidiem katrā gadījumā ir visatbilstošākais.



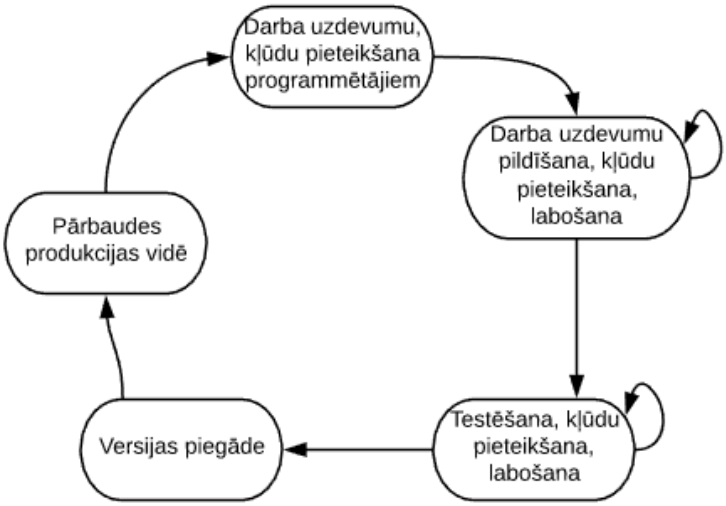

*3.12. att.* **Algu aprēķina moduļa atskaites "Aprēķina kopsavilkums" atlases kritēriji**

Lielu daļu sarakstu ir iespējams eksportēt XLS vai PDF formātā, individuāliem sarakstiem vai rindām ir iespēja tos eksportēt XML formātā. Papildus šīm atskaitēm ir vēl iespējams piekļūt speciāli sagatavotām atskaitēm, balstoties uz vairāku sadaļu datiem – šādas atskaites tiek izgūtas nevis balstoties uz vienas rindas datiem vai izdodot uzrādīto sarakstu, bet gan balstoties uz datu bāzē aprēķinātu informāciju par vairākiem periodiem vai pēc pieprasītajiem atskaites filtra kritērijiem (skat. 3.13. att.).

## 3.5. Versijas dzīves cikls Algu aprēķina modulī

Vidēji reizi divos mēnešos Algu aprēķina modulim notiek jaunas versijas piegāde, un ir nepieciešama pilna šīs versijas pārbaude. Kopā ar versijas piegādi Algu modulī, tiek piegādāta arī versija darba laika uzskaites tabulām. Ar jaunām versijām tiek piegādāti sistēmas kļūdu labojumi, jauna funkcionalitāte kā arī izmaiņu pieprasījumi.

Algu aprēķina moduļa mērķis ir nodrošināt uzņēmumā nodarbinātam personālam darba laika uzskaiti, darba algas aprēķinu un izmaksu, nodrošinot t.sk. atbilstošu atskaišu veidošanu.




*3.13. att.* **Algu aprēķina moduļa versijas piegādes dzīves cikls**

Algu aprēķina sistēma sastāv no trim vidēm – izstrādes vide, testa vide un produkcijas vide. Izstrādes vidē strādā programmētāji – visi labojumi, kas tiek veikti, ir tajā pieejami uzreiz. Testa vidē strādā testētāji, un ik dienu tiek likti labojumi no izstrādes vides uz testa vidi ar programmētāju veiktajiem labojumiem. Produkcijas vidē strādā klienti, un tur vienmēr stāv pēdējā aktuālā sistēmas versija.

Versijas dzīves cikls Algu aprēķina modulī sastāv no trim galvenajām darbībām – izstrāde, testēšana, atkļūdošana, kas ietver sekojošus soļus (skat. 3.14. att.):

a) darba uzdevumu un labojumu nodefinēšana un piešķiršana programmētājiem (balstoties uz klientu prasībām, pieteiktajām kļūdām, izmaiņām likumos vai citiem

faktoriem, tiek nolemts labot esošo funkcionalitāti vai papildināt sistēmu ar jaunu funkcionalitāti);

b) izstrāde (programmētājs izpilda piešķirtos darba uzdevumu un nodod tos testēšanai), testēšana (testētājs pārbauda programmētāju atrisinātos vienumus, testēšanas brīdī konstatē kļūdas un piesaka tās atbildīgajam programmētājam), testētāju pieteikto kļūdu labošana. Šajā fāzē ik dienu tiek likti programmētāju sistēmas un datu bāzes labojumi no izstrādes vides uz testa vidi;
c) testēšanas periods, kad vairs netiek piešķirti jauni darba uzdevumi, bet tikai testēta visa sistēma, piereģistrētas un labotas kļūdas. Arī šajā fāzē ik dienu tiek likti labojumi no izstrādes vides uz testa vidi;
d) programmatūras versijas piegāde produkcijas vidē (uz visiem serveriem tiek uzliktas lietotnes un datu bāzes izmaiņas);
e) programmatūras pārbaude produkcijas vidē (pēc versijas piegādes tiek pārbaudīts vai produkcijā netiek atrastas kļūdas un vai versijas piegāde ir bijusi veiksmīga).

### 3.5.1. Testēšanas process Algu aprēķina modulī

Pēc iepriekš aprakstītā ir noskaidrots, ka testētāji sistēmu testē visa versiju piegādes cikla laikā – tad, kad programmētāji izpilda darba uzdevumus un to risinājums tiek uzlikts uz testēšanas servera, testētāji pārbauda jauno funkcionalitāti. Arī kļūdu labojumi tiek likti uz testēšanas servera un to labojumus pārbauda testētāji. Pēc visu versijas darba uzdevumu izpildes jauni darbi vairs netiek veidoti, un tiek tikai reģistrētas atrastās kļūdas, kuras tiek labotas, un kuru labojumi tiek testēti. Kad ir pienākusi versijas diena, tiek likta sistēma un datu bāze no testēšanas vides uz produkcijas vidi. Produkcijas vidē ir nodalīta vieta mākonī, kas kalpo kā akcepttesta vide. Pēc versijas uzlikšanas testētāji akcepttesta vidē izpilda testus un pārliecinās, ka sistēma strādā. Ja akcepttesta vidē sistēmas funkcijas strādā, tad tās strādās arī pārējā produkcijas vidē.

Tad, kad versija ir uzlikta un notestēta, visi līdzšinējie programmētāju labojumi tiek likti no izstrādes vides uz testa vidi, un testēšanas process sākās atkal no jauna, tikai nākamajā versijā.

### 3.5.2. Problēmas manuālajā testēšanā

Gadās tādas situācijas, kad pēc programmētāju labojumu uzlikšanas testa vidē, tiek konstatēts, ka kāda no sistēmas daļām nestrādā kā nepieciešams, piemēram, netiek atvērta sadaļa, vai arī netiek uzrādīti dati sarakstā. Šādā situācijā testētājiem nav iespējams strādāt ar

konkrēto sistēmas daļu, kamēr kļūda nav novērsta. Līdzīga situācija ir iespējama tad, kad tiek uzlikta versija produkcijā – lai pārliecinātos, ka visa sistēma strādā, testētājam ir nepieciešams "izstaigāt" cauri visām sistēmas daļām un pārbaudīt katru komponenti un darbību. Šādas pārbaudes aizņem laiku un kavē testētāju darbu sistēmas pārbaudē, kā arī kavē sistēmas normālas darbības atjaunošanu.

Lietotāja saskarnes automatizētā testēšana atvieglotu testētāju darbu – ja pēc versijas uzlikšanas testa vidē vai produkcijā tiktu izpildīta automatizētā testēšana, kas bez lietotāja iejaukšanās pārbaudītu visas sistēmas komponentes, būtu iespējams ātrāk identificēt kļūdas, tās izlabot, un tiktu mazāk aizkavēts testētāju darbs.

## 3.6. Kopsavilkums

Darba 2. daļā tika veikta iespējamo rīku un metožu analīze, lai izstrādātu automatizētos testus. Ir nepieciešams izstrādāt automatizētos testus Algu aprēķina moduļa darba laika uzskaites tabulu aizpildes funkcionalitātei gan lietotāja saskarnē, gan arī datu bāzē. Balstoties uz darba 2. un 3. daļās apskatīto, modulis ir tīmekļa lietotne un tai ir jāizstrādā uz *Selenium* balstīti testi. Tīmekļa lietotnei kods tiek izstrādāts uz *Visual Studio* programmas, kas dod priekšrocību testu izstrādei uz tās pašas programmas. Tā kā testējamā sistēma ir liela izmēra un tā sastāv no dažādu veidu sarežģītām kontrolēm, to skaitā *DevExpress* sarakstiem, tos būtu sarežģīti notestēt bez *JavaScript* koda rakstīšanas, caur kuru var manipulēt *DevExpress* kontroles – pieprasīt atzvanīšanu (satura atjaunošanu no servera), piešķirt vai izgūt vērtības ievades lauku kontrolēm un veikt citas darbības. Vienīgais risinājums, kurš atļauj *JavaScript* koda izpildi testos, ir *Selenium* testi uz *Visual Studio* programmas.

Algu aprēķina un darba laika uzskaites algoritmi ir sarežģīti. Kamēr lietotāja saskarnē, spiežot uz pogas, varētu domāt, ka tiek izpildīta viena darbība, datu bāzē šī darbība sastāv no vairākām sarežģītām darbībām, tādēļ vienību testēšana Algu aprēķina sistēmai noteikti ir nepieciešama. Tas nozīmē, ka ir nepieciešams izstrādāt datu bāzes testus. Datu bāzes komponentes tiek izstrādātas un uzturētas, izmantojot *PL/SQL Developer* datu bāzu pārvaldības sistēmu. Testu izstrāde uz tās pašas programmas, kurā tiek rakstīts kods, būtu ļoti parocīga, jo visas darbības ir veicamas vienā programmā. Svarīga datu bāzes procedūru daļa ir kursori, kas tiek atgriezti datu uzrādīšanai lietotāja saskarnē. *utPLSQL* spraudnis piedāvā izstrādāt vienību testus un pārbaudīt datu pareizumu, to skaitā salīdzināt kursoru datus, tādēļ tas ir atbilstošs risinājums datu bāzu vienību testu izstrādei.

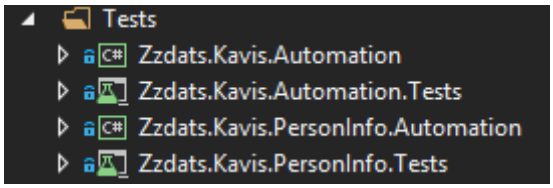
# 4. AUTOMATIZĒTO TESTU IEVIEŠANA ALGU APRĒĶINA MODULĪ

Balstoties uz apskatītajiem automatizēto testu izstrādes variantiem darba 2. daļā un Algu aprēķina sistēmas analīzes darba 3. daļā, tika izstrādāts kopsavilkums darba 3. daļas beigās, kurā ir noskaidrots, ka visatbilstošāk automatizēto testu izstrādei Algu aprēķina sistēmas darba laika uzskaites funkcionalitātei ir izmantot *Visual Studio* programmā piedāvātos *Selenium* testus, un datu bāzē izstrādāt testus uz *utPLSQL* spraudņa pamata. Abi varianti nodrošina testu izstrādi tajā pašā programmā, kurā tiek izstrādāts kods. Izmantojot šos variantus, ir iespējams izstrādāt testus bez ierobežojumiem, jo tos var rakstīt ar roku lietotāja saskarnei, un var pievienot neierobežotus procedūru izsaukumus un pārbaudes viena testa ietvaros *utPLSQL* testos.

Darba autors šajā darba daļā turpmāk apraksta automatizēto testu izstrādi Algu aprēķina sistēmas darba laika uzskaites funkcionalitātē. Darba gaitā ir izstrādāti lietotāja saskarnes testi uz *Selenium* pamata *Visual Studio* programmā, kā arī datu bāzes vienību testi, izmantojot *utPLSQL* spraudni *PL/SQL Developer* programmā.

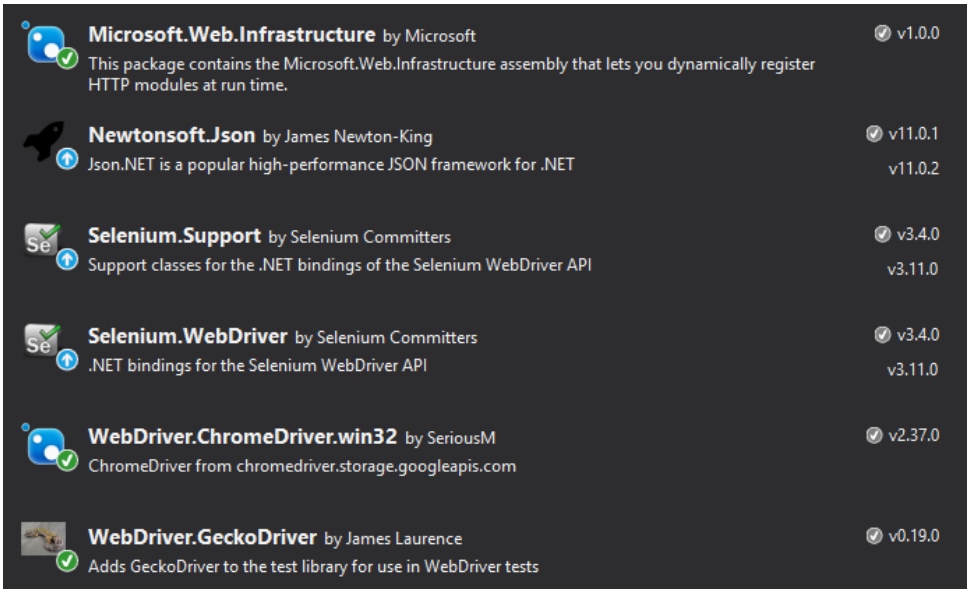
## 4.1. *Selenium* automatizēto testu izstrāde

*ZZ Dats* jau tiek pielietota automatizētā saskarnes testēšana, izmantojot *Visual Studio* programmatūru, tādēļ ir jau sagatavoti divi projekti ar kodu, uz kuru pamata var izstrādāt kodu automatizētajiem testiem projektā.




*4.1. att.* **Automatizētās testēšanas projekti moduļa testēšanas koda izstrādei**

Projekts "*Zzdats.Kavis.Automation*" (skat. 4.1. att.) satur kopējo informāciju un infrastruktūru testu automatizācijai. "*Zzdats.Kavis.Automation.Tests*" satur kopējo testa informāciju. Papildus šiem diviem projektiem, ir nepieciešams izveidot vēl divus – "*Zzdats.Kavis._Projekta_nosaukums_.Automation*", kas satur konkrētā moduļa lapu sadalījumu un "*Zzdats.Kavis. ._Projekta_nosaukums_.Automation.Tests*", kas satur konkrētā moduļa testu realizāciju. Pirmie divi projekti netiek mainīti un ir vienādi visām sistēmām (moduļiem), tomēr divi pārējie projekti tiek izstrādāti un pielāgoti individuāli katram modulim.




*4.2. att.* **Nepieciešamās *Microsoft Visual Studio* pakotnes automatizēto *Selenium* testu izstrādei projektā**

Lai būtu iespējams izstrādāt automatizēto testu kodu un izpildīt testus, projektam ir nepieciešams pievienot pakotnes, kas pievieno klāt projektam papildus funkcionalitāti (skat. 4.2. att.). To ir iespējams paveikt, izmantojot *Visual Studio* iebūvētu rīku "*NuGet*".

```
<appSettings>
    <add key="TestDriver" value="Chrome" />
    <add key="ImplicitTimeOut" value="20" />
    <add key="BaseUrl" value="https://visvaris-tv.zzdats.lv/" />
    <add key="User" value="auto_lo" />
    <add key="Password" value="          " />
    <add key="PersonInfoPrefix" value="PersonInfo" />
    <!--Selenium server parameters-->
    <add key="SeleniumServerUrl" value="http://selenium.zzdats.lv:4444/wd/hub" />
    <add key="SeleniumServerTimeout" value="60" />
</appSettings>
```

*4.3. att.* **Konfigurācijas faila fragments automatizēto testu izpildei DLUT**

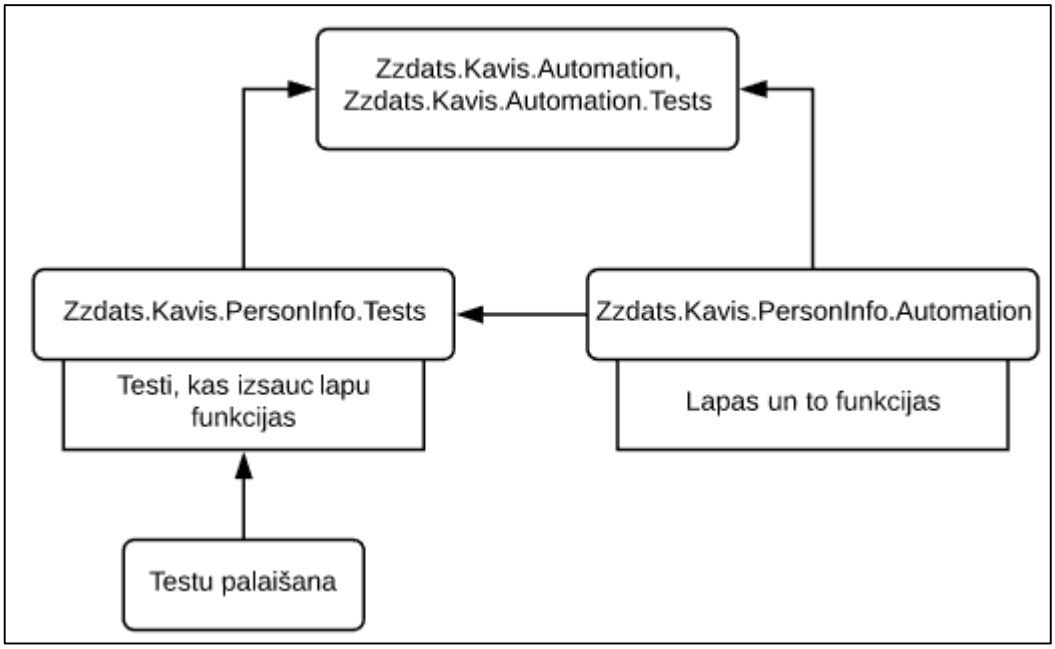
Pēc projektu izveides ir jānodefinē konfigurācija, kas tiks izmantota testu palaišanai un izpildei. Konfigurācijā tiek norādīts, kādu tīmekļa pārlūkprogrammu automatizētie testi izmantos, kāda ir pamata adrese, kurā notiks testēšana, kā arī lietotāja vārds un parole ar kuriem notiks pieslēgšanās (skat. 4.3. att.). Šo parametru nodefinēšana konfigurācijas failā nav obligāta, tomēr ir parocīga un atvieglo darbu testu izstrādē, jo visa konfigurācija atrodas vienuviet, kā rezultātā uzturēšana ir vienkārša.

Ir iespējams izstrādāt vairākus konfigurācijas failus – vienu testa videi, bet otru – produkcijas videi. Tas nozīmē, ka ir iespējams izstrādāt testus, kas pārbaudīs sistēmas darbību ne tikai uz testa vides serveriem, bet arī uz produkcijas vides serveriem. Testi produkcijas vidē

palīdz pārliecināties par sistēmas stabilitāti pēc jaunas versijas uzlikšanas. Programmētāji ir nodrošināti ne tikai ar atsevišķu lietotāju testa vides automatizēto testu veikšanai, bet arī ar produkcijas vides automatizēto testu lietotāju, kuru izmanto, lai pieslēgtos pirms testu izpildes.

**Testu izstrāde**

Tad, kad ir veikta testu projektu konfigurācija, var sākt veidot testus. "Automation" projektā tiek veidots individuāls fails katrai lapai (sistēmas daļai), kurā tiek uzglabātas testu funkcijas. "Tests" projektā nodefinē pašus testus, kas izsauc funkcijas no "Automation" projekta. Izpildot darbību "Palaist visus testus", tiek izpildīts tas, kas ir nodefinēts zem "Tests" projekta (skat. 4.4. att.).




*4.4. att.* **Automatizēto testu projektu sasaiste**

DLUT testu projektam ir nodefinētas 15 kopīgās palīgfunkcijas (tās ir pieejamas un izsaucamas no jebkuras testu grupas), lai pamata funkcionalitāti varētu izsaukt no jebkura testa, un to nebūtu atkārtoti jāraksta. Funkcijas ir aprakstītas 4.1. tabulā.

*4.1. tabula*

**Autora izstrādātās kopīgās testu palīgfunkcijas**

| Funkcija | Nolūks |
|---|---|
| `FindElement( driver, by, timeoutInSeconds)` | Atgriež elementu (pēc parametra by – tips satur veidu, pēc kura atlasīt (id, klase) un tā nosaukumu) ar gaidīšanu līdz elementa ielādei. Ja `timeoutInSeconds` ir lielāks par 0, tad gaida norādīto laiku, līdz atrod nepieciešamo elementu, pretēji uzreiz atgriež norādīto elementu |
| `FindElements( driver, by, timeoutInSeconds)` | Atgriež sarakstu ar vairākiem elementiem, strādā pēc augstāk aprakstītās funkcijas principa |
| `SetDevExpComboBoxValue( driver, element, value)` | Uzstāda *DevExpress* izkrītošās izvēlnes vērtību |

*4.1. tabulas turpinājums*

| Funkcija | Nolūks |
|---|---|
| `SetDevExpTextBoxValue( driver, element, value)` | Uzstāda *DevExpress* teksta lauka vērtību |
| `SetSelectBoxValue( driver, element, value)` | Uzstāda izkrītošās izvēlnes vērtību |
| `SetTextBoxValue( driver, element, value)` | Uzstāda teksta lauka vērtību |
| `SetTextBoxValueWTrigger( driver, element, value)` | Uzstāda teksta lauka vērtību, papildus izpildot "`.keyup()`" darbību teksta laukam (simulē vērtības labošanu ar roku) |
| `GetElementValue( driver, element)` | Atgriež elementa vērtību |
| `GetDevExpElementValue( driver, element)` | Atgriež *DevExpress* elementa vērtību |
| `WaitCallbackEnd( driver, gridName, timeoutInSeconds)` | Sagaida *DevExpress* objekta atzvanīšanas beigas, gaidot noteiktu sekunžu laiku |
| `ClickWScroll( element)` | Rullē līdz objektam un spiež uz tā |
| `GetGridPageRowCount( driver, gridName)` | Atgriež redzamo rindu skaitu *DevExpress* sarakstā |
| `CheckFileDownloaded( filename)` | Datora lejupielādes mapē iet cauri lejupielādētajiem failiem un pārbauda vai kāds atbilst pieprasītajam faila nosaukumam, un ir izveidots pēdējās minūtes laikā. Atrod failu un izdzēš to |
| `CheckFile( driver, button, filename, time)` | Izsauc `ClickWScroll` uz padoto pogu, pagaida pieprasīto laika periodu un izsauc `CheckFileDownloaded`, padodot sagaidāmo faila nosaukumu |
| `CheckInputValById( driver, inputId, inputVal)` | Pārbauda vai teksta lauks satur pieprasīto vērtību |

Darba laika uzskaites tabulu testiem ir nodefinētas funkcijas, kuras tiek izmantotas testu definēšanā. Šīs funkcijas var tikt izsauktas atkārtoti viena testa ietvaros. Funkcijas ir aprakstītas 4.2. tabulā.

*4.2. tabula*

**Autora izstrādātās Darba laika uzskaites testu funkcijas**

| Funkcija | Nolūks | Papildus nosacījumi |
|---|---|---|
| `GoTo()` | Navigē atvērto pārlūkprogrammas logu uz sadaļas saiti | - |
| `wait` | Izpilda padoto darbību līdz paiet konkrēts laika periods (t.i. nepilda darbību ilgāk kā, piemēram, 10 sekundes) | - |
| `Open()` | Izsauc `GoTo()` un pārbauda vai ir redzams DLU saraksts, izmantojot `FindElement()` | - |
| `CloseDetalization()` | Aizver rindas detalizāciju, ja tāda ir atvērta | - |
| `CloseErrorMsg()` | Aizver kļūdas paziņojumu vai modālo logu | - |

*4.2. tabulas turpinājums*

| Funkcija | Nolūks | Papildus nosacījumi |
|---|---|---|
| `Search( count, adminDepartmentId, workerGroupId, year, month)` | Aizpilda meklēšanas kritērijus ar saņemtajiem datiem, spiež pogu "Meklēt" un pārbauda vai saraksts satur pieprasīto rindu skaitu | Jābūt atvērtai lapai ar sarakstu |
| `Register(count)` | Izpilda darbību "Reģistrēt" un pārbauda vai saraksts satur noteiktu reģistrētu rindu skaitu | Jābūt atvērtai lapai ar sarakstu |
| `OpenDetalization()` | Spiež "Atvērt detalizāciju" rindai un pārbauda vai eksistē detalizācijas saraksts | Jābūt atvērtai lapai ar sarakstu, un jābūt atlasītam vismaz vienam reģistrētam ierakstam |
| `CheckDetalization( inputId, inputVal, plus)` | Pārbauda vai detalizācijā eksistē ievades lauks, vai tas satur pieprasīto vērtību. Ja padod parametru "plus", tad veic papildus pārbaudi: pārbauda vai poga "Saglabāt" ir paslēpta, piešķir ievades laukam citu vērtību, spiež pogu "Atcelt", pārbauda vai saraksts atjaunojās uz veco vērtību | Jābūt atvērtai rindas detalizācijai |
| `SaveDetalizationValue( inputId, inputVal)` | Saglabā detalizācijas vienā laukā konkrētu vērtību.<br>Piešķir ievades laukam padoto vērtību, spiež "Saglabāt" un pēc saraksta atjaunošanās pārbauda vai sarakstā ir redzama jaunā ievadītā vērtība | Jābūt atvērtai rindas detalizācijai |
| `FillByPlan(inputId, inputVal, stepNr)` | Aizpilda visas rindas sarakstā pēc plāna.<br>Izsauc `CloseDetalization()`, iezīmē visas saraksta rindas, pārbauda vai ir redzama poga "Aizpildīt pēc plāna", spiež pogu, akceptē ar "Jā", atver detalizāciju un izsauc `CheckDetalization()`, padodot noklusēto vērtību | Jābūt atvērtai lapai ar sarakstu, un jābūt atlasītam vismaz vienam reģistrētam ierakstam |
| `Approve(inputId, plus, stepNr)` | Apstiprina visas rindas sarakstā.<br>Izsauc `CloseDetalization()`, iezīmē visas saraksta rindas, pārbauda vai ir redzama poga "Apstiprināt", spiež pogu, akceptē ar "Jā". Ja padod parametru "plus", tad veic papildus pārbaudi: izsauc `OpenDetalization()` un pārbauda vai ievades lauka labošana ir bloķēta | Jābūt atvērtai lapai ar sarakstu, un jābūt atlasītam vismaz vienam reģistrētam ierakstam |

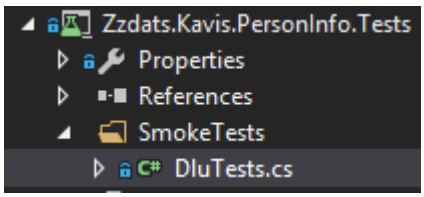
*4.2. tabulas turpinājums*

| Funkcija | Nolūks | Papildus nosacījumi |
|---|---|---|
| `ValidateColumnValue( columnName, columnVal, rowNumber)` | Validē saraksta vienas rindas vienas kolonnas vērtību: Saraksta vienas kolonnas vērtību izvada uz ekrāna kā kļūdas paziņojumu, ar `wait` darbību pārbauda vai izvadītā vērtība sakrīt ar pieprasīto, un izsauc `CloseErrorMsg()`. | Jābūt atvērtai lapai ar sarakstu |
| `SendToAccounting( docNr, docNrId, stepNr)` | Nosūta visas apstiprinātās rindas uz grāmatvedību: Izsauc `CloseDetalization()`, spiež uz pogas "Nosūtīt uz grāmatvedību", ar `wait` darbību sagaida, līdz atveras forma, pārbauda vai ir pieejams pieprasītais ievades lauks, aizpilda ar pieprasīto informāciju, spiež uz pogas "Izveidot", un ar `ValidateColumnValue`() pārbauda vai rindas dokumenta numurs sakrīt ar ievadīto | Jābūt atvērtai lapai ar sarakstu. Jābūt vismaz vienai apstiprinātai un nenosūtītai rindai |
| `SendToAccounting_Repeat( docNr, stepNr)` | Nosūta visas apstiprinātās rindas uz grāmatvedību; izpildāms atkārtoti pēc datu labošanas: Izsauc `CloseDetalization()`, spiež uz pogas "Nosūtīt uz grāmatvedību". Pārbauda vai piedāvā labot jau esošās rindas (nevis veidot jaunu dokumentu), spiež uz pogas "Izveidot", un ar `ValidateColumnValue`() pārbauda vai rindas dokumenta numurs sakrīt ar iepriekš izveidoto | Jābūt atvērtai lapai ar sarakstu. Jābūt vismaz vienai labotai, apstiprinātai un nenosūtītai rindai |
| `Unapprove(stepNr)` | Atceļ visu rindu apstiprinājumu. Izsauc `CloseDetalization()` un `CloseErrorMsg()`, sagaida līdz saraksts ir atjaunojies, pārbauda vai ir redzama poga "Atcelt apstiprinājumu", spiež uz šīs pogas, akceptē darbību ar "Jā", sagaida brīdinājumu par precizējuma veidošanu pēc datu labošanas, izsauc `CloseErrorMsg()` | Jābūt atvērtai lapai ar sarakstu. Jābūt vismaz vienai apstiprinātai un nosūtītai rindai |

*4.2. tabulas turpinājums*

| Funkcija | Nolūks | Papildus nosacījumi |
|---|---|---|
| `DeleteRow(count, stepNr)` | Dzēš vienu rindu sarakstā. Izsauc `WaitCallbackEnd()`, spiež uz pirmās pogas "Dzēst", akceptē darbību ar "Jā", izsauc `WaitCallbackEnd()`, pārbauda vai saraksts satur pieprasīto pogas "Dzēst" skaitu (dzēstām rindām pogas nav) | Jābūt atvērtai lapai ar sarakstu. Jābūt vismaz vienai reģistrētai rindai, ko dzēst |
| `GetXls(filename)` | Izgūst un dzēš saraksta atskaiti XLS formātā. Atrod pogu faila izgūšanai, izsauc papildus funkciju `CheckFile()`, atgriež rezultātu | Jābūt atvērtai lapai ar sarakstu |
| `GetPdf(filename)` | Izgūst un dzēš saraksta atskaiti PDF formātā. Atrod pogu faila izgūšanai, izsauc papildus funkciju `CheckFile()`, atgriež rezultātu | Jābūt atvērtai lapai ar sarakstu |
| `DeleteAllRows(stepNr)` | Dzēš visas rindas sarakstā. Izsauc `CloseDetalization()`, izsauc `WaitCallbackEnd()`, iezīmē visas saraksta rindas, izsauc `WaitCallbackEnd()`, pārbauda vai ir redzama poga "Dzēst", akceptē dzēšanu ar "Jā", izsauc `WaitCallbackEnd()`, pārbauda vai visu rindu statuss ir "Nereģistrēts" | Jābūt atvērtai lapai ar sarakstu. Visām rindām sarakstā jābūt reģistrētām |
| `GetUserManual(filename)` | Izgūst un dzēš lietotāja instrukciju. Spiež uz pogas "Lietotāja instrukcija", ar `wait` darbību sagaida, līdz atveras logs, kas satur "Lejupielādēt" pogu, spiež uz pogas, izsauc papildus funkciju `CheckFile()`, atgriež rezultātu | Jābūt atvērtai lapai ar sarakstu |
| `ClearData(stepnr)` | Konkrētajiem filtra kritērijiem atjauno datus uz sākotnējiem. Atceļ visu rindu iezīmēšanu, izsauc `WaitCallbackEnd()`. Pārbauda vai ķeksis "Iezīmēt visus" ir ieķeksēts – ja ir, tad dati ir nereģistrēti un ir atjaunoti, pretēji izsauc `Unapprove()`, izsauc `DeleteAllRows()`. | - |

Tad, kad ir izstrādātas funkcijas, kuras izmantos testi, tiek nodefinēti paši testi ar kritērijiem. Zem "Tests" projekta atsevišķā failā (skat. 4.5. att.) tiek izstrādāta funkcija, kas satur nepieciešamo funkciju izsaukumus, un šie izsaukumi kopā ir uzskatāmi kā viens tests.




*4.5. att.* **Darba laika uzskaites testa faila atrašanās vieta projektā**

Darba laika uzskaites testu kopums pārbauda visas moduļa pogas, darbības un sarakstus. Katra izveidotā funkcija tiek izsaukta vismaz vienu reizi, nodrošinot, ka viss sistēmas lietotāja saskarnē tiek pārbaudīts. "DluTests.cs" failā ir izveidota "Dlu_Test()" funkcija, kas satur vienkopus nodefinētus un piešķiramus parametrus priekš testiem (skat. 4.6. att.). Ja ir nepieciešams izpildīt testu vairākos periodos (t.i., ar vairākiem filtra kritērijiem), tad "Dlu_Test()" funkciju var nokopēt un ievietot zem tās kā atsevišķu funkciju, izmainot filtra parametrus, kas tiek izmantoti testos. Kā arī šādā veidā var notestēt vairākus scenārijus bez funkciju mainīšanas – ir nepieciešams tikai izveidot jaunu testa funkciju, izmainīt parametrus un izsaukt testus tādā secībā, kādā nepieciešams.

```
//Meklēšanas parametri
string institutionId = "1010"; //iestādes ID
string adminDepartmentId = "437"; //Administratīvās struktūrvienības ID
int? workerGroupId = null; //Darbinieku grupas ID
int year = 2016; //Gads
int month = 3; //Mēnesis
int rowCount = 1; //Rindu skaits sarakstā

//Detalizācijas parametri:
string valueBeginning = "8"; //Vērtība, kādai jābūt ievadlaukā
string valueEdit = "9"; //Vērtība, uz kādu labos
string activityTypeId = "4"; //Aktivitātes tipa ID, kuram testē vērtību
string day = "1"; //Datums, kuram testē vērtību

//Nosūtīšanas grāmatvedībai parametri
string docNr = "nr"; //veidojamā dlu dok. numurs
string docNrId = "Dlu_Create_DocNr"; //dokumenta numura ievades lauka ID
```

*4.6. att.* **"Dlu_Test()" funkcijas parametri un to vērtību piešķiršana**

Pēc parametru piešķiršanas tiek izsauktas funkcijas secīgi, izpildot konkrētu testēšanas scenāriju (skat. 4.7. att.). Scenārijs izpilda sekojošas darbības:

a) atver sadaļu,
b) meklē ierakstus,
c) atjauno datus uz sākotnējo stāvokli,
d) reģistrē rindas,
e) atver vienas rindas detalizāciju,

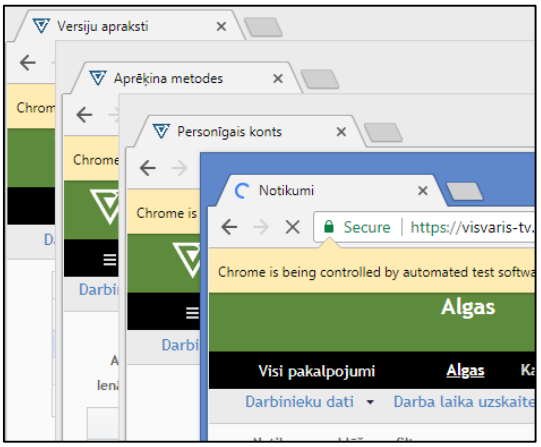
f) detalizācijā pārbauda vai stundas izvēlētam datumam izvēlētā aktivitātes veidā sakrīt ar pieprasītajām,
g) detalizācijā saglabā citu vērtību,
h) aizpilda rindas datus pēc plāna,
i) apstiprina visas rindas un pārbauda vai detalizācijā ir nobloķēta vērtību ievade,
j) nosūta apstiprinātās rindas uz grāmatvedību un pārbauda nosūtīto rindu izveidotā dokumenta numuru,
k) atceļ visu rindu apstiprinājumu,
l) atver vienas rindas detalizāciju,
m) detalizācijā saglabā citu vērtību, nekā tika saglabāta f) solī,
n) validē, ka labotajai rindai tiek uzrādīta pazīme "Jāprecizē",
o) apstiprina visas rindas,
p) nosūta apstiprinātās rindas uz grāmatvedību, pārbaudot, ka sistēma piedāvā labot esošās rindas, nevis veidot jaunu dokumentu, un pēc nosūtīšanas pārbauda, ka nav mainījies nosūtīto rindu dokumenta numurs (tas norāda, ka ir labots esošais dokuments),
q) atceļ visu rindu apstiprinājumu,
r) dzēš vienu rindu,
s) reģistrē rindas no jauna,
t) dzēš visas rindas,
u) pieprasa saraksta PDF atskaiti,
v) pieprasa saraksta XLS atskaiti,
w) pieprasa lietotāja rokasgrāmatas izguvi.

```
//Atver sadaļu
Assert.True(Page.Open(), "DLU tabulas sadaļas atvēršana");

//Meklē
Assert.True(Page.Search(rowCount, admDepartmentFullId, workerGroupId, year, month), "Ierakstu meklēšana");

//Atjauno datus
Assert.True(Page.ClearData(ref stepNr), "Datu atjaunošana " + stepNr);

//Reģistrē rindas
Assert.True(Page.Register(rowCount), "Ierakstu reģistrēšana");

//Atver detalizāciju
Assert.True(Page.OpenDetalization(), "Detalizācijas atvēršana");

//Detalizācijā pārbauda vai stundas datumā aktivitātes veidam sakrīt ar pieprasīto
Assert.True(Page.CheckDetalization(inputId, valueBeginning, true), "Detalizācijas vērtību pārbaude");
```

*4.7. att.* **Fragments no "Dlu_test()" funkcijas darbību izsaukšanas**

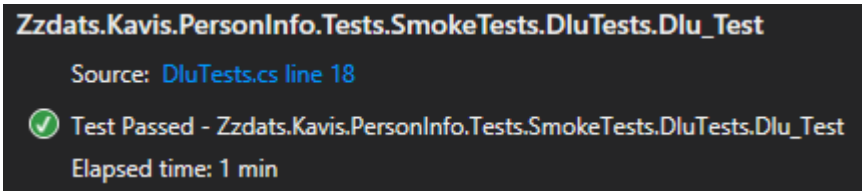
Testa palaišanas brīdī tiek atvērta izvēlētā tīmekļa pārlūkprogramma un secīgi pildītas visas pieprasītās darbības. Lietotājam ir redzams logs, kurā tiek veiktas darbības, un tajā ir

iespējams iejaukties. Logs ir kā parasts tīmekļa pārlūkprogrammas logs, tikai tajā bez lietotāja iejaukšanās automātiski tiek pildītas darbības. Lietotājam tīmekļa pārlūkprogrammas logā ir redzams paziņojums, ka to kontrolē automatizētās testēšanas programmatūra. Gadījumos, kad izpildāmo testu skaits ir liels, *Visual Studio* vienlaicīgi atver vairākus tīmekļa pārlūkprogrammas logus un izpilda vienlaicīgi vairākus testus, lai ietaupītu laiku (skat. 4.8. att.). Testi ir neatkarīgi viens no otra un viens otru neietekmē, tādēļ šādi drīkst rīkoties.



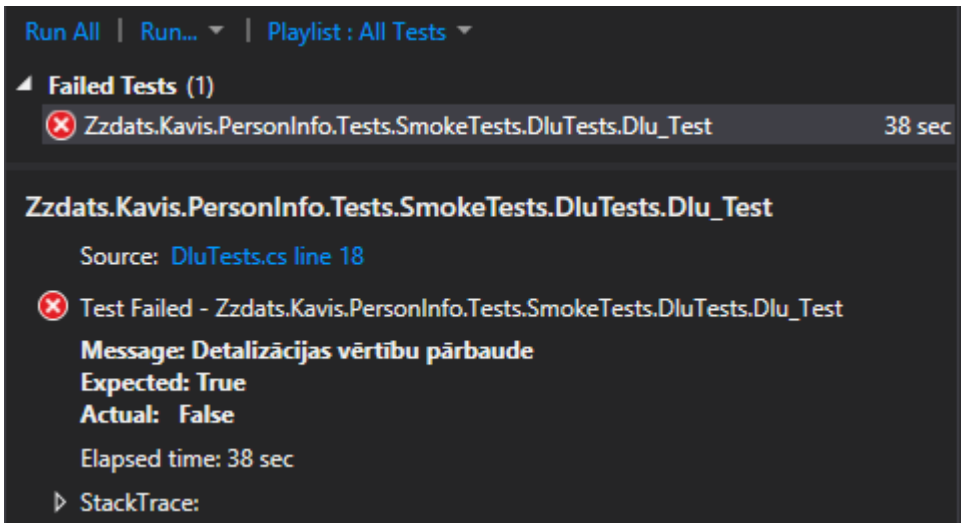

*4.8. att.* **Vairāki *Google Chrome* tīmekla pārlūkprogrammas logi, kuros vienlaicīgi notiek vairāku testu pildīšana**

Veiksmīgas testa izpildes gadījumā lietotājam tiek parādīta informācija par testu – nosaukums un izpildes laiks (skat. 4.9. att.).




*4.9. att.* **Veiksmīgas automatizētā testa izpildes paziņojums programmā *Visual Studio***

Ja tests ir bijis neveiksmīgs, tad lietotājam tiek uzrādīta informācija par testiem, kuri ir veiksmīgi izpildījušies līdz neveiksmīgajam testam, neveiksmīgi izpildītā testa nosaukums un nodefinētais kļūdas paziņojums neveiksmīgas testa izpildes gadījumā (skat. 4.10. att.).




*4.10. att.* **Neveiksmīgas automatizētā testa izpildes paziņojums programmā *Visual Studio***


*4.11. att.* **Automatizētās testu kopas izpildīšanas rezultāti ar veiksmīgiem un neveiksmīgiem testiem**

Gadījumos, kad, pildot testus, kāds tests ir bijis neveiksmīgs, tad tas tests tiek apturēts, bet tiek turpināts pildīt pārējos testus. Testu izpildes apturēšanas vai pabeigšanas brīdī lietotājam tiek uzrādīta informācija par katru veiksmīgi un neveiksmīgi izpildīto testu, kā arī testa izpildes laiku (skat. 4.11. att.).

**Problēmas ar kurām var saskarties automatizēto testu izstrādē**

Izstrādājot automatizētos testus, darba autors saskārās ar vairākām problēmām, kurām nācās atrast risinājumus, un kuru dēļ bija nepieciešams veikt uzlabojumus testos.

Gadījumā, kad tests neizpildās veiksmīgi, visticamāk, ka testa vidū ir radusies kļūda. Tādā gadījumā ir grūti noteikt, kāds tieši ir iemesls testa neveiksmīgai izpildei, it īpaši tad, ja tests sastāv no daudziem soļiem. Risinājums šādai situācijai ir references mainīgā "stepNr" nodefinēšana (skat. 4.12. att.), kuram pēc katras darbības piešķir soļa numuru. Neveiksmīgas

testa izpildes gadījumā  šajā mainīgajā tiek padots pēdējā piešķirtā soļa numurs pirms testa pārtraukšanas, un pēc tā var noteikt, kura bija pēdējā izpildītā darbība testā un pie kura soļa radās problēma. Pēdējā izpildītā soļa numurs tiek izvadīts klāt testa izpildes paziņojumam (paziņojums bez soļa numura apskatāms 4.10. att.).

```
public bool DeleteAllRows(ref int stepNr)
{
    try
    {
        stepNr = -1;

        //Aizver detalizāciju (ja ir atvērta)
        CloseDetalization();
        stepNr = 0;

        //Sagaida, kad saraksts ir beidzis atjaunoties
        Driver.Instance.WaitCallbackEnd(GridName, timeout);
        stepNr = 1;

        //Iezīmē visas saraksta rindas
        js.ExecuteScript("DLUGrid.SelectRows();");
        stepNr = 2;

        //Sagaida, kad saraksts ir beidzis atjaunoties
        Driver.Instance.WaitCallbackEnd(GridName, timeout);
        stepNr = 3;
```

*4.12. att.* **Piemērs automatizētā testa funkcijai, kas izmanto parametru "stepNr"**

Pēc neveiksmīgas testu kopas izpildes, ja testus vēlās palaist atkārtoti, to nebija iespējams izdarīt, jo jau pie pirmā testa tika atgriezta kļūda par testa neveiksmīgu izpildi. Tas notika tādēļ, ka netika atjaunots sākotnējais datu stāvoklis, bet pirmā soļa veikšanai bija nepieciešams konkrēts datu stāvoklis veiksmīgai testa izpildei. Tādēļ tika pievienota funkcija "ClearData" (skat. 4.13. att.), kas atjauno datus uz sākotnējo stāvokli no jebkura testa stāvokļa. Tādi testa stāvokļi bija vairāki, šajā gadījumā – reģistrēta rinda (nepieciešams visas rindas dzēst) un apstiprināta vai nosūtīta rinda (nepieciešams rindām atcelt apstiprinājumu, un tad tās dzēst).

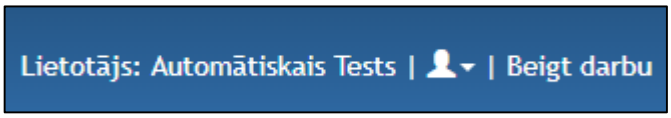
```
public bool ClearData(ref int stepNr)
{
    try
    {
        stepNr = 11;

        //At-iezīmē visas rindas
        js.ExecuteScript("DLUGrid.UnselectRows()");
        stepNr = 12;
        //Sagaida, kad saraksts ir beidzis atjaunoties
        Driver.Instance.WaitCallbackEnd(GridName, timeout);
        stepNr = 13;
        //Ja "Iezīmēt visus" ir true
        if ((bool)js.ExecuteScript("return cbDluSelectAll.GetValue() == true;"))
        {//Ja ir true, tad dati jau ir atjaunoti
            stepNr = 14;
            return true;
        }
        else
        {
            stepNr = 15;
            //Noņem rindām statusu "Apstiprināts"
            Unapprove(ref stepNr);
            stepNr = 16;
            Driver.Wait(TimeSpan.FromSeconds(1));
            //Dzēš rindas
            return DeleteAllRows(ref stepNr);
        }
```

*4.13. att.* **Funkcijas "ClearData" kods**

Bieži nācās saskarties ar viltus negatīvu rezultātu dēļ tā, ka, pieprasot pildīt kādu darbību, nebija atjaunojušies dati sarakstā pēc darbības izpildes vai nebija aizvēries iepriekšējais logs. Tādā gadījumā visvienkāršākais variants ir uzlikt konkrētu sekunžu gaidīšanu līdz nākamās darbības veikšanai. Tomēr tas nav efektīvi, jo nāksies gaidīt pat tad, ja dati būs jau pieejami darbības izpildei. Tādēļ tika pievienota darbība, kas pārbauda vai nepieciešamā poga ir pieejama, vai saraksts ir atjaunojies un dati ir pieejami, bet tas nozīmē vēl vienas pārbaudes pievienošanu pirms īstās darbības izpildes pieprasīšanas. Visefektīvākais variants, kurš arī tika realizēts, ir pievienot funkciju "wait" (skat. 4.2. tabulu), kas mēģina piekļūt objektam vai izpildīt kādu darbību, bet neveiksmīga rezultāta gadījumā mēģina atkārtoti, līdz ir pagājis izvēlētais laika periods. Piemēram, ja tiek padota darbība "Spiest uz pogas Nosūtīt" un laika intervāls 5 sekundes, tad funkcija mēģinās spiest uz pogas – ja izdosies, tad darbība beigs darbu, bet ja neizdosies, tad mēģinās atkal. Ja ir pagājušas 5 sekundes un nav izdevies, tikai tad funkcija atgriezīs neveiksmīgu rezultātu. Piemērs funkcijas "wait" pielietojumam ir redzams 4.14. attēlā.

```
//Sagaida, līdz atverās forma un ir redzama poga "Nosūtīt"
if (wait.Until(drv =>
        (bool)((IJavaScriptExecutor)drv)
        .ExecuteScript("return $('#Dlu_CreateDluDocument').is(':visible')")))
{
```

*4.14. att.* **funkcijas "wait" pielietojums**




*4.15. att.* **Automatizēto testu lietotāja informācija sistēmā**

Automatizētajiem testiem ir nepieciešami konstanti dati – tiem vienmēr ir jābūt pieejamiem, un tos neviens cits nedrīkst izmantot vai mainīt, citādi testi būs jāmaina vai arī dati būs jāveido no jauna. Tādēļ testēšanai tiek izmantots speciāls lietotājs (skat. 4.15. att.), kuram ir piešķirtas tiesības uz objektiem, kas paredzēti tieši automatizētai testēšanai, un kuriem nevar piekļūt neviens cits lietotājs. Tas nodrošina datu pastāvīgumu.

## 4.2. Datu bāzes automatizēto testu izstrāde

Algu aprēķina moduļa datu bāzes kods tiek izstrādāts uz *PL/SQL Developer* programmatūras, kurai ir pieejams *utPLSQL* spraudnis. Pievienojot spraudi programmatūrai, tiek pievienots "UT3" lietotājs, kurā ir pieejamas pakas ar kodu, kas ir izmantojams automatizēto testu izstrādē. Lai izveidotais lietotājs spētu apstrādāt automatizēto testu darbības, tam ir nepieciešamas tam atbilstošas tiesības, to skaitā sesijas, procedūru, tipu, tabulu, skatu un sinonīmu veidošana. Tāpat arī ir nepieciešams nodefinēt sinonīmus šī lietotāja pakām, lai jebkurš cits datu bāzes lietotājs varētu piekļūt informācijai šajās pakās un izstrādāt automatizētos testus [UTPLS].

Programmatūrai pievienotais spraudnis piedāvā izstrādāt pakas ar procedūrām – katra procedūra tiek uzskatīta kā tests un tiek izpildīta individuāli un neatkarīgi no citām. Lai paka tiktu uzskatīta kā testa paka, un testi tajā tiktu izpildīti, ir nepieciešams pievienot atbilstošo anotāciju pakas specifikācijā ("%suite" apzīmē, ka paka ietilpst vienību testos, skat. 4.16. att.). Anotācijā ir nepieciešams arī norādīt testa pakas nosaukumu, šajā gadījumā tas ir "kadri2.dlu".

```
create or replace package ut_dlu_pck is

   -- %suite
   -- %suitepath(kadri2.dlu)
```

*4.16. att.* ***PL/SQL*** **testa pakas specifikācijas anotācija, kas atzīmē, ka pakā atrodas testi**

Lai procedūra tiktu uzskatīta kā tests un izpildīta, tai arī ir nepieciešama atbilstoša anotācija - "%test". Anotācijā papildus ir norādāms testa nosaukums vai apraksts, kurš var atšķirties no procedūras nosaukuma (skat. 4.17. att.). Ar anotāciju ir iespējams pateikt, lai procedūru izpilda pirms visiem citiem testiem (piemēram, ja ir nepieciešams sagatavot datus pirms testa uzsākšanas) (anotācija "%beforeall", skat. 4.17. att.), kā arī var izstādāt procedūras, kuras ir izpildāmas pēc visu testu izpildīšanas, pievienojot anotāciju "%afterall".

```
-- %test
-- %displayname(SDP_strukt_list)
PROCEDURE SDP_strukt_list;
```

(a)

```
-- %beforeall
PROCEDURE BEFORE_TEST;
```

(b)

*4.17. att.* **PL/SQL testa pakas procedūras deklarāciju anotācijas paraugi**

(a) procedūra, kas izpildāma kā tests, (b) procedūra, kas izpildāma pirms visām citām

Procedūras kodā ir iespējams veikt datu atlasi, labošanu un pārbaudes tā, kā tas tiktu darīts jebkurā citā procedūrā, un ar vienību testu spraudņa pievienoto kodu ir iespējams pārbaudīt šos atlasītos datus vai vērtības. Datu pārbaude ietver vienas vai vairāku vērtību vai objektu salīdzināšanu, to skaitā (skat. 4.18. att.) pārbaudi vai vērtība ietilpst skaitļu intervālā, pārbaudi vai vērtība ir vai nav tukša, ir lielāka vai mazāka par citu vērtību, vai satur citu vērtību, vai atbilst konkrētai regulārai izteiksmei, kā arī vai sakrīt ar citu vērtību [UTPLS]. Pārbaudē vai elements sakrīt ar citu elementu ir iespējams salīdzināt arī datu bāzu kursorus, kas nozīmē nevis viena vārda vai skaitļa salīdzināšanu, bet gan liela izmēra tabulu uzbūves (kolonnu, tipu) un datu salīdzināšanu. Salīdzinot kursorus, ir iespējams norādīt kolonnas, kuras ir nepieciešams ignorēt – tas nozīmē, ka ja vienā kursorā būs kolonna, bet otrā nebūs, vai arī kolonnas vērtības abos kursoros atšķirsies, tas netiks ņemts vērā.

```
be_between
be_empty
be_false
be_greater_than
be_greater_or_equal
be_less_or_equal
be_less_than
be_like
be_not_null
be_null
be_true
equal
match
```

*4.18. att.* **Pilns *utPLSQL* spraudņa pieejamo pārbaužu saraksts [UTPLS]**

Pārbaudes tiek veiktas, izmantojot *utPLSQL* darbību "UT.EXPECT" – pirmajā parametrā tiek padota atlasītā vērtība, kuru ir nepieciešams pārbaudīt, otrajā parametrā tiek padots

paziņojums, kurš tiks izvadīts neveiksmīgas pārbaudes gadījumā. Salīdzināšanas pārbaudes izveidotajam objektam ir iespējams izsaukt 4.18. attēlā aprakstītās pārbaudes, piemēram, zemāk aprakstītajā gadījumā (skat. 4.19. att.) ir pārbaudes, ka vērtība nav tukša un ka vērtība sakrīt ar "0". Ja kāda no šīm pārbaudēm atgriež negatīvu rezultātu, tad tests tiek uzskatīts par izgāzušos un šajā testā tālākas pārbaudes netiek veiktas. Tā kā testi ir individuāli un neatkarīgi viens no otra, viena testa izgāšanās neietekmē pārējo testu izpildi.

```
--TESTS pārbauda vai strādā procedūra SLODZES_DARBA_PERIODI_D
PROCEDURE SDP_delete IS
  v_sdp_id    NUMBER := h_get_sdp_id();
  v_sk        NUMBER;
  V_ERR       VARCHAR2(2000);
  V_ERR2      VARCHAR2(2000);
  v_strv      NUMBER := v_strv2_id;
  V_ROW_DATA  KADRI2.SLODZES_DARBA_PERIODI%ROWTYPE;
  V_ROW_DATA2 KADRI2.SLODZES_DARBA_PERIODI%ROWTYPE;
  v_sdp_ids   slodzes_darba_periodi_pck.NUMBER_TYPE;
BEGIN

  sessionstart;

  --Pārbauda vai tika izgūts iepriekšējā testā reģistrētais ieraksts
  ut.EXPECT(A_ACTUAL => v_sdp_id,
            A_MESSAGE => 'Parbaude vai izguts registretas rindas id'
            ).to_be_not_null();

  --Izsauc dzēšanu
  SLODZES_DARBA_PERIODI_PCK.SLODZES_DARBA_PERIODI_D(IN_SDP_ID => v_sdp_id,
                                                    OUT_ERR   => V_ERR);

  --Atlasa vai ieraksts ir dzēsts
  SELECT COUNT(1) INTO v_sk
  FROM kadri2.SLODZES_DARBA_PERIODI sdp
  WHERE sdp.id = v_sdp_id;

  --Pārbauda
  ut.EXPECT(A_ACTUAL => v_sk,
            A_MESSAGE => 'Parbaude vai ieraksts izdzests'
           ).to_equal(0);

  sessionend;

END SDP_delete;
```

*4.19. att.* **Vienību testa procedūras "SDP_delete" kods**

Katrā izstrādātajā procedūrā darba autors izsauc ieraksta, ar kuru veicamas darbības, identifikatora izguvi un piešķir atgriezto vērtību mainīgajam (mainīgais "v_sdp_id", skat. 4.19. att.). Tas nodrošina piekļuvi ieraksta identifikatoram, kurš tiek izmantots testos. Testa sākumā

tiek izsaukta lietotāja sesijas izveidošana un procedūras beigās – sesijas pārtraukšana. Sesijas izveide ir nepieciešama tādēļ, ka modulī visas izsauktās procedūras pārbauda vai lietotājs ir pieslēgts un vai lietotājam ir tiesības veikt izvēlētās darbības ar izvēlētajiem objektiem.

Ieraksta dzēšanas darbības tests (skat. 4.19. att.) vispirms pārbauda vai ir veiksmīgi izgūts iepriekšējos testos reģistrētās rindas identifikators. Tests tālāk izdzēš reģistrēto rindu, izsaucot procedūru, kas tiek izmantota lietotāja saskarnē brīdī, kad lietotājs spiež uz pogas "Dzēst". Pēc ieraksta dzēšanas tiek pārbaudīts vai ieraksts patiešām ir dzēsts, atlasot skaitu, cik ieraksti atlasās ar iepriekš reģistrētā ieraksta identifikatoru. Ja tiek atlasīts skaits 0, tad rindas dzēšana ir bijusi veiksmīga un tests ir veiksmīgi nokārtots.

```
-- %test
-- %displayname(SDP_register)
-- %rollback(manual)
procedure SDP_register;
```

*4.20. att.* ***PL/SQL*** **testa pakas procedūras deklarāciju anotācija automātiskas transakcijas atrites atspējošanai**

Noklusēti testiem ir iestatīta automātiska transakcijas atrite pēc testa izpildes, kas nozīmē, ka citos testos nav redzamas agrākā testā veiktās datu izmaiņas. Lai mainītu šo funkcionalitāti, pakas specifikācijā procedūras deklarācijai ir nepieciešams pievienot anotāciju "rollback(manual)" (skat. 4.20. att.), kas nosaka, ka netiks veikta transakcijas atrite pēc testa. Tas nodrošina, ka testā veiktās izmaiņas ir redzamas turpmākajos testos, piemēram, testā bez automātiskas transakcijas atrites tiek piereģistrēti ieraksti un nākamajos testos var izmantot šos reģistrētos datus.

Ļoti nozīmīga datu salīdzināšanas darbība darba laika uzskaitēm ir "to_equal", kas salīdzina divus datu objektus un pieprasa, lai tie sakrīt. Šīs darbības priekšrocība ir spēja salīdzināt kursorus. Darba laika uzskaitē pielietojums šai darbībai ir sekojošā scenārijā – vispirms tiek atlasīti uzģenerētie (nereģistrētie) ieraksti, ielasot atgrieztos datus kursorā, tad dati tiek piereģistrēti, un tiek atlasīti piereģistrētie dati citā kursorā. Abu kursoru dati (visas rindas un kolonnas) tiek salīdzināti, un datu nesakritības rezultātā tiek uzskatīts, ka tests nav nokārtots. Tā kā ir kolonnas, kuru saturs atšķiras nereģistrētiem un reģistrētiem ierakstiem, piemēram, ieraksta identifikators un ieraksta statuss, ir nepieciešams salīdzināšanas darbībai norādīt kolonnas, kuru saturu ir nepieciešams ignorēt. To ir iespējams izdarīt, parametrā "a_exclude" padodot simbolu virkni ar kolonnu nosaukumiem, kas atdalīti ar komatu (skat. 4.21. att.).

```
--TESTS pārbauda vai darbība "Reģistrēt" piereģistrē datus pareizi un vai pirms un pēc reģistrā
--Testējamās darbības: "Atlasīt nereģistrētos datus", "Reģistrēt", "Atlasīt reģistrētos datus"
PROCEDURE SDP_register IS
  V_BEFORE_REF SYS_REFCURSOR;
  V_AFTER_REF SYS_REFCURSOR;
  v_str VARCHAR2(4000);
  v_str2 VARCHAR2(4000);
BEGIN
    SESSIONSTART;
    BEGIN

      --Atlasa nereģistrēto ierakstu datus kursorā V_BEFORE_REF - GET_LIST_GENERATED
      slodzes_darba_periodi_pck.GET_LIST_GENERATED(IN_STRUKT_ID  => v_strukt_id,IN_DGRUPA_ID

      --Izpilda darbību "Reģistrēt" ADD_SDP_MULTI_CRITERIA
      slodzes_darba_periodi_pck.ADD_SDP_MULTI_CRITERIA(IN_IESTADE_ID => v_iestade_id,IN_STRUKT

      --Atlasa reģistrēto ierakstu datus kursorā V_AFTER_REF - GET_LIST_EXISTING
      slodzes_darba_periodi_pck.GET_LIST_EXISTING(IN_STRUKT_ID  => v_strukt_id,IN_DGRUPA_ID  =

   EXCEPTION WHEN OTHERS THEN
     ut.EXPECT(A_ACTUAL => 1, A_MESSAGE => 'Kluda darbibu izpilde: ' || SQLERRM).to_equal(0);
   END;

   --Pārbauda vai reģistrācija neatgrieza kļūdu
   ut.EXPECT(A_ACTUAL => v_str,
             A_MESSAGE => 'Parbaude par registraciju bez kludas pazinojuma')
             .to_be_null();

   --Salīdzina VAI V_BEFORE_REF un V_AFTER_REF sakrīt (ignorē kolonnas: P_PIESAISTITS, SDP_ID,
   ut.expect( V_AFTER_REF, 'Parbaude vai registretas rindas sakrit ar generetajam' )
              .to_equal( V_BEFORE_REF, a_exclude => 'P_PIESAISTITS,SDP_ID,KOPSAV_DARBS_SVETKU_

   SESSIONEND;

EXCEPTION WHEN OTHERS THEN SESSIONEND;

END SDP_register;
```

*4.21. att.* **Vienību testa procedūras "SDP_register" kods ar kursoru salīdzināšanas darbību**

Darba laika uzskaites testēšanai kopā ir izstrādāti 18 testi, kas ir aprakstīti 4.3. tabulā. Testi pārbauda katru no izsaucamajām procedūrām darba laika uzskaites funkcionalitātē. Katrs tests testē vienu vai vairākas procedūras darba laika uzskaites pakā. Datu pārbaude un validēšana pēc darbību izpildes ietver arī datu pārbaudīšanu tabulās, pārliecinoties, ka visas stāvokļu pārejas un datu labojumi notiek korekti.

*4.3. tabula*

**Darba laika uzskaites funkcionalitātes izstrādātie automatizētie vienību testi un to apraksts**

| Tests | Nolūks | Papildus nosacījumi |
|---|---|---|
| BEFORE_TEST | Atjauno datus pirms testu uzsākšanas – pēc nodefinētajiem filtra kritērijiem atceļ ierakstu apstiprinājuma statusu un izdzēš esošos ierakstus, ja tādi eksistē | Izpildās pirms visiem testiem, sagatavojot datus |

*4.3. tabulas turpinājums*

| Tests | Nolūks | Papildus nosacījumi |
|---|---|---|
| SDP_strukt_list | Atlasa administratīvo struktūrvienību sarakstu, kas tiek attēlots izkrītošajā izvēlnē filtrā, un salīdzina vērtības ar sagaidāmajiem datiem, pieprasot, lai tie sakristu | |
| SDP_register | Atlasa nereģistrēto ierakstu datus kursorā, piereģistrē ierakstus, atlasa reģistrēto ierakstu datus kursorā un salīdzina abu kursoru informāciju, pieprasot, lai tā būtu vienāda | Neizmanto automātisku transakcijas atriti (reģistrētie dati ir pieejami turpmākiem testiem) |
| SDP_idp_u | Vienam ierakstam izsauc Algu moduļa grāmatvedības dokumenta piešķiršanu un pārbauda vai tas ir nomainījies. Izsauc tā paša ieraksta vērtības atjaunošanu uz tukšu un pārbauda vai vērtība ir tukša | |
| SDP_strv_u | Vienam ierakstam izsauc citas administratīvās struktūrvienības piešķiršanu un pārbauda vai tā ir nomainīta | |
| SDP_ik_renew | Atlasa viena ieraksta informāciju, maina ieraksta slodzes koeficientu un darba grafiku uz citu, izsauc šo datu atjaunošanu no saistītā amata kalendāra uzstādījumiem un pārbauda vai dati ir atjaunojušies uz sākotnējiem | |
| SDP_approve | Vienam ierakstam izsauc statusa maiņu uz "Apstiprināts", pārbauda vai statuss ir nomainījies, izsauc statusa maiņu atpakaļ uz "Reģistrēts", pārbauda vai statuss ir nomainījies | |
| SDP_clear_hours | Vienam ierakstam izsauc visu stundu dzēšanu (katrā dienā maiņu uz 0), atlasa maksimālo stundu vērtību ierakstam un pārbauda vai tā ir 0 | |
| SDP_delete | Izsauc viena ieraksta dzēšanu un pārbauda vai ieraksts tika dzēsts | |
| SDP_delete_multi | Izsauc visu saraksta ierakstu dzēšanu un pārbauda vai tie ir dzēsti | |
| SDP_rows_delete | Izsauc viena ieraksta detalizācijas vērtību dzēšanu un pārbauda vai ieraksti ir dzēsti | |
| SDP_rewrite | Vienam ierakstam visas detalizācijas stundas maina uz "0.5", izsauc stundu atjaunošanu un pārbauda vai ir palicis kāds ieraksts ar stundām, kas sakrīt ar "0.5" | |

*4.3. tabulas turpinājums*

| Tests | Nolūks | Papildus nosacījumi |
|---|---|---|
| SDP_detalized_list | Izsauc viena ieraksta detalizācijas atlasi un pārbauda kritisko (tādu, kurām būtu vienmēr jābūt atbilstošām) kolonnu informāciju, salīdzinot to ar sagaidāmo | |
| SDP_hours_u | Izsauc viena ieraksta detalizācijas stundu labošanu, atlasa ieraksta detalizācijas kopsavilkumu aktivitātes veidam "Normāls darba laiks" un pārbauda vai stundas ir sasummējušās korekti | |
| SDP_get_dpstv | Izsauc administratīvās struktūrvienības informācijas atlasi, ko uzrāda logā "Nosūtīt uz grāmatvedību" un pārbauda vai dati tiek atlasīti un sakrīt ar filtra kritērijos norādītajiem | |
| SDP_has_rights | Izsauc funkciju, kas pārbauda vai padotā administratīvā struktūrvienība lietotājam ir pieejama – pārbauda vai uz pieejamu administratīvo struktūrvienību funkcija atgriež "1" un vai uz nepieejamu tā atgriež "0" | |
| SDP_dlu_jauns | Apstiprina vienu ierakstu, izsauc ieraksta nosūtīšanu uz grāmatvedību, padodot veidojamā dokumenta numuru, pārbauda vai dokuments ir izveidots, pārbauda vai tas satur norādīto dokumenta numuru un vai dati nosūtītajā rindā sakrīt ar grāmatvedībā nosūtītās rindas informāciju.<br>Izsauc ieraksta statusa maiņu atpakaļ uz reģistrētu un pārbauda vai procedūra atgriež brīdinājumu par precizējuma veidošanu iepriekš nosūtītajai rindai | Neizmanto automātisku transakcijas atriti (mainītie dati ir pieejami turpmākiem testiem) |
| SDP_japrecize_u | Atlasa iepriekšējā testā nosūtītās rindas grāmatvedības ieraksta identifikatoru, nodzēš to, izsauc pazīmes "jāprecizē" uzlikšanu un pārbauda vai procedūra atrada ierakstam saistīto grāmatvedības rindu un uzlika pazīmi (pazīme tiek likta tikai, ja eksistē grāmatvedības dokuments, kurš ir izveidots no rindas) | |

Lai testos nebūtu atkārtoti jāraksta viena un tā pati informācija, kā arī lai būtu iespējams viegli pārvaldīt filtra kritērijus, pēc kuriem jāmeklē un jāapstrādā ieraksti katrā testā, ir izveidotas funkcijas, kas atgriež visus nepieciešamos parametrus un kritērijus datu apstrādei (skat. 4.22. att.). Bez funkcijām, kas atgriež parametrus, ir izstrādātas vēl papildus procedūras, kas veic sekojošas darbības: atgriež viena ieraksta identifikatoru pēc filtra kritērijiem, ar kuru veikt apstrādes darbības katrā testā, uzsāk testa lietotāja sesiju, un pārtrauc lietotāja sesiju.

```
--Filtra kritēriji un citi parametri
FUNCTION v_strukt_id RETURN NUMBER IS BEGIN RETURN 437; END;
FUNCTION v_dgrupa_id RETURN NUMBER IS BEGIN RETURN NULL; END;
FUNCTION v_iestade_id RETURN NUMBER IS BEGIN RETURN 1010; END;
FUNCTION v_gads RETURN NUMBER IS BEGIN RETURN 2016; END;
FUNCTION v_menesis RETURN NUMBER IS BEGIN RETURN 3; END;
FUNCTION v_dper_id RETURN NUMBER IS BEGIN RETURN 176; END; --Darba periods

FUNCTION v_idp2_id RETURN NUMBER IS BEGIN RETURN 89; END;  --Jebkāds eksistējošs IDP ID
FUNCTION v_strv2_id RETURN NUMBER IS BEGIN RETURN 659; END;  --Jebkāds eksistējošs STRV ID
```

*4.22. att.* **Vienību testu pakas funkcijas, kas atgriež filtra kritērijus un parametrus, ko jāizmanto testos**

Pēc funkciju izstrādes ir iespējams palaist visus izstrādātos automatizētos testus *PL/SQL Developer* programmatūras testa logā, izsaucot darbību "ut.run();". Visu izpildīto testu rezultāti tiek izvadīti datu izvades logā, uzskaitot katru izpildīto testu un tā izpildes statusu, ja tas ir bijis neveiksmīgs (skat. 4.23. att., pirmais tests sarakstā ir izpildījies neveiksmīgi).

```
kadri2
  dlu
    ut_dlu_pck
      SDP_strukt_list [,453 sec] (FAILED - 1)
      SDP_register [3,313 sec]
      SDP_idp_u [,109 sec]
      SDP_strv_u [,032 sec]
      SDP_ik_renew [,031 sec]
      SDP_approve [,078 sec]
      SDP_clear_hours [,094 sec]
      SDP_delete [,062 sec]
      SDP_delete_multi [,094 sec]
      SDP_rows_delete [,047 sec]
      SDP_rewrite [,719 sec]
      SDP_detalized_list [,031 sec]
      SDP_hours_u [,094 sec]
      SDP_get_dpstv [,922 sec]
      SDP_has_rights [,015 sec]
      SDP_dlu_jauns [1,203 sec]
      SDP_japrecize_u [,032 sec]
```

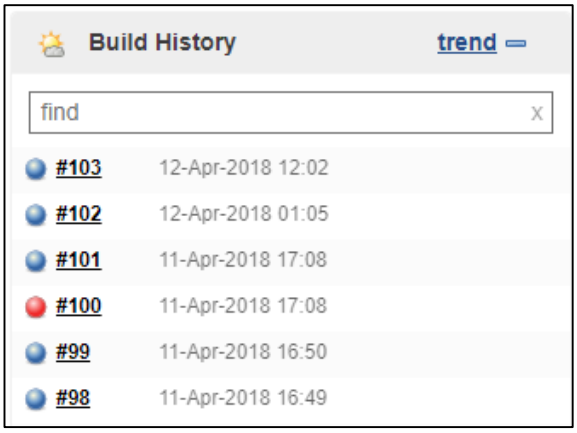
*4.23. att.* ***utPLSQL*** **testa datu izvades informācija pēc testu izpildīšanas**

Pēc katra izpildītā testa uzskaites tiek izvadīta informācija par neveiksmīgajiem testiem, uzskaitot katru neveiksmīgo darbību un sniedzot papildus informāciju. Šajā gadījumā neveiksmīgā darbība ir kursoru salīdzināšana un izvades logā tiek uzrādītas abu kursoru vērtības, parādot lietotājam to, ka tās nesakrīt (skat. 4.24. att.).

```
Failures:

  1) sdp_strukt_list
     "Parbaude vai atlasas pareizas strukturvienibas un iestades"
     Actual:
         <ROW><ID>659</ID><NOSAUKUMS>Algu testa iestāde</NOSAUKUMS><K
         <ROW><ID>659</ID><NOSAUKUMS>Algu testa iestāde</NOSAUKUMS><K
         <ROW><ID>659</ID><NOSAUKUMS>Algu testa iestāde</NOSAUKUMS><K
         <ROW><ID>659</ID><NOSAUKUMS>Algu testa iestāde</NOSAUKUMS><K
         <ROW><ID>437</ID><NOSAUKUMS>&quot;Sesku&quot; privātskola</N
         <ROW><ID>437</ID><NOSAUKUMS>&quot;Sesku&quot; privātskola</N
         <ROW><ID>433</ID><NOSAUKUMS>VPIP</NOSAUKUMS><KLIENTS>DEMO Vi
      (refcursor)
     was expected to equal:
         <ROW><ID>659</ID><IESTADE_ID>1010</IESTADE_ID></ROW>
         <ROW><ID>659</ID><IESTADE_ID>1015</IESTADE_ID></ROW>
         <ROW><ID>659</ID><IESTADE_ID>1006</IESTADE_ID></ROW>
         <ROW><ID>659</ID><IESTADE_ID>1011</IESTADE_ID></ROW>
         <ROW><ID>437</ID><IESTADE_ID>1010</IESTADE_ID></ROW>
         <ROW><ID>437</ID><IESTADE_ID>1006</IESTADE_ID></ROW>
         <ROW><ID>433</ID><IESTADE_ID>1010</IESTADE_ID></ROW>
      (refcursor)
     at "KADRI2_UT.UT_DLU_PCK", line 119 ut.expect( V_AFTER_REF, 'Par
```

*4.24. att.* ***utPLSQL*** **testa datu izvades informācija pēc testu izpildīšanas (turpinājums)**

ZZ Dats piedāvā risinājumu, kas automātiski katru dienu izpilda izveidotos automatizētos testus datu bāzē. Tas nodrošina pārliecību, ka visas procedūras strādā un tajās nav radušās problēmas, jo ik dienu tiek izpildīti visi testi. Piedāvātā sistēma ir balstīta uz *Jenkins* un ir pieejama ZZ Dats darbiniekiem tīmeklī. *Jenkins* ir nepārtrauktās integrācijas sistēma, kas nodrošina programmatūras koda kompilēšanu, būvēšanu un testēšanu [ODORJ]. Tajā ir iespējams norādīt datu bāzi, kurā izpildīt testus, testu izpildes laika intervālu (ik dienu, ik nedēļu, u.t.t.) un testu izpildes laiku, kā arī e-pastu, uz kuru ziņot par neveiksmīgiem testiem. Mājas lapā automatizēto testu izpildes izveidotajā lapā var apskatīt visas būvējuma reizes un to statusu (veiksmīgs vai neveiksmīgs) (skat. 4.25. att.). Noklusēti testi tiek izpildīti katru dienu 1 naktī (attēlā redzams, ka tests "#102" ir izpildīts automātiski naktī). Testus ir iespējams arī izpildīt jebkurā citā brīdī, to pieprasot tīmekļa vietnē, spiežot uz pogas "Build now".




*4.25. att.* **ZZ Dats piedāvātā automatizēto testu izpildes risinājuma būvējuma vēstures pārskats (sarkans – neveiksmīgs tests, zils – veiksmīgs tests)**

Katru būvējumu ir iespējams atvērt un apskatīt tā detalizāciju – ir iespēja piekļūt XML failam, kas satur datu izvades loga informāciju (skat. 4.26. att.).

```
<testsuites tests="18" skipped="0" error="0" failure="1" name="" time="2.5160000000"
>
<testsuite tests="18" id="1" package="kadri2"  skipped="0" error="0" failure="1"
name="kadri2" time="2.5160000000" >
<testsuite tests="18" id="2" package="kadri2.dlu"  skipped="0" error="0" failure="1"
name="dlu" time="2.5160000000" >
<testsuite tests="18" id="3" package="kadri2.dlu.ut_dlu_pck"  skipped="0" error="0"
failure="1" name="ut_dlu_pck" time="2.5160000000" >
<system-out>
<![CDATA[
]]>
</system-out>
<testcase classname="kadri2.dlu.ut_dlu_pck"  assertions="3" skipped="0" error="0"
failure="1" name="SDP_strukt_list" time="0.1250000000"  status="Failure">
<failure>
<![CDATA[
"Parbaude vai izguts registretas rindas id"
Actual: NULL (number) was expected to be not null
at "KADRI2_UT.UT_DLU_PCK", line 69 ut.EXPECT(A_ACTUAL => v_sdp_id, A_MESSAGE =>
'Parbaude vai izguts registretas rindas id').to_be_not_null();
```

*4.26. att.* **ZZ Dats piedāvātā automatizēto testu izpildes risinājuma būvējuma XML faila fragments ar testa informāciju no datu izvades loga**

ci@test.zzdats.lv Jenkins build is back to normal : Zzdats.LOPS.KADRI2.Database #101
See <http://ci.zzdats.lv/job/Zzdats.LOPS.KADRI2.Database/101/display/redirect> <end>

ci@test.zzdats.lv Build failed in Jenkins: Zzdats.LOPS.KADRI2.Database #100
See <http://ci.zzdats.lv/job/Zzdats.LOPS.KADRI2.Database/100/display/redirect>

*4.27. att.* **ZZ Dats piedāvātā automatizēto testu izpildes risinājuma būvējuma izsūtītie e-pasti**

Lai programmētājam, kas izstrādāja testus, nebūtu ik dienu jāiet mājas lapā un jāpārbauda vai automātiski izpildītie testi ir bijuši veiksmīgi, ļoti noderīga ir e-pastu izsūtīšana par neveiksmīgiem testiem, kā arī par gadījumiem, kad būvējums ir atjaunojies uz veiksmīgu un testos vairs nav kļūdu (skat. 4.27. att.). Uz norādīto e-pasta adresi pēc katra būvējuma tiek izsūtīta ziņa, kurā ir pievienota informācija par datu bāzi, kurā veikts būvējums, kas noder gadījumos, kad viens programmētājs izstrādā testus vairākās datu bāzēs. Bez datu bāzes tiek nosūtīts arī būvējuma statuss (neveiksmīgs vai arī kļuvis par veiksmīgu), kā arī saite, uz kuras spiežot var piekļūt sistēmai un apskatīt detalizētāku testu informāciju.

# REZULTĀTI

Maģistra darba rezultātā tika sasniegts izvirzītais mērķis – izpētīt testu automatizācijas iespējas, izvēlēties atbilstošākās un realizēt tās, izstrādājot automatizētos testus Algu aprēķina moduļa darba laika uzskaites funkcionalitātē gan lietotāja saskarnē, gan datu bāzē. Darbā tika apskatīta ZZ Dats Vienotā pašvaldību sistēma, resursu vadības sistēma G-VEDIS, Algu aprēķina modulis un darba laika uzskaites funkcionalitāte, to arhitektūra un ierobežojumi.

Tika apskatītas automatizēto testu izstrādes metodes tīmekļa vietnēm un datu bāzu vienību testēšanai, un tās analizētas. Veicot metožu analīzi, tika noskaidrots, ka tādām sistēmām, kā Algu aprēķina sistēma, vislabāk ir izvēlēties ar roku rakstītus testus, kas balstīti uz *Selenium* ietvara, jo "ierakstīt un atskaņot" princips nepiedāvā tik lielu elastīgumu testu izstrādē un uzturēšanā. Datu bāzu vienību testu izstrādei Algu aprēķina sistēmā izvēlēts *utPLSQL* spraudnis, uz kura pamata tika izstrādāti datu bāzes vienību testi.

Darba laika uzskaites funkcionalitātei autors izstrādāja lietotāja saskarnes testus, pārbaudot katru lietotājam pieejamo kontroli, to skaitā sarakstus, pogas un datu modificēšanas iespējas. Lietotāja saskarnes testi tika izstrādāti uz *Visual Studio* programmatūras, un tos ir ērti izpildīt pēc koda maiņas, jo nav jāmaina programmatūra, lai izpildītu testu. Testi ir izstrādāti, balstoties uz modulārā ietvara vadlīnijām, kas nozīmē, ka testu funkcijas ir vairākkārt izmantojamas dažādos testos. Pēc nepieciešamības var pievienot papildus testa funkcijas, pārbaudot citu mēnešu datus vai izpildot citādus scenārijus. Vienlaicīgi ir iespējams palaist visus testus, un ja kādi no izpildītajiem testiem pēc izpildes bija neveiksmīgi, tie programmētājam tiek paziņoti, sarežģītiem testiem papildus uzrādot konkrēto rindu, kuras pārbaude bija neveiksmīga.

Autors datu bāzes vienībām (procedūrām) izstrādāja vienību testus, ar kuriem tiek notestēta katra lietotājam pieejama procedūra. Testi ir integrēti ar automatizēto darbu izpildes sistēmu *Jenkins*, kas nodrošina regulāru testu izpildi naktī un e-pastu izsūtīšanu par neveiksmīgiem testiem ar iespēju apskatīt testa detalizētu pārskatu ar neveiksmīgās darbības aprakstu.

Testu izstrāde darba autoram ir palīdzējusi atrast radušās kļūdas un tās novērst, pirms tās nonāk pie testētājiem. Tā kā ik dienu tiek liktas programmētāju veiktās izmaiņas no izstrādes vides uz testa vidi, regulāra vienību testu izpilde ir ļoti svarīga.

Katrā no Algu aprēķina moduļa vidēm regulāri tiek izpildīti automatizētie testi pēc labojumu veikšanas un versijas likšanas. Testi katrā vidē automatizēti pārbauda vai vide strādā, kā nepieciešams. Pēc katras versijas uzlikšanas tiek izpildīti lietotāja saskarnes automatizētie testi, un ar automatizēto darbu izpildes sistēmu *Jenkins* tiek izpildīti datu bāzes automatizētie testi gan testa vidē, gan produkcijas vidē.

# SECINĀJUMI

Izpētot automatizētās testēšanas teoriju, tika noskaidrots, ka vissvarīgākie ir vienību testi, tomēr nedrīkst aprobežoties tikai ar tiem – ir jāveic arī lietotāja saskarnes testēšana, jo to vienību testi nepārbauda. Lietotāja saskarnes testi ir daudz jūtīgāki pret labojumiem, nekā vienību testi, jo pat vismazākā izmaiņa var panākt, ka tests ir neveiksmīgs. Ir svarīgi arī veikt integrācijas testus, jo katra vienība var strādāt pareizi, tomēr tās kopā var mijiedarboties nekorekti un neizpildīt lietojuma scenārijus kā nepieciešams.

Automatizētie testi neaizstāj testētāju darbu, bet gan to atvieglo. Konstanti regulāri atkārtojami testi ikdienā bieži vien netiek izpildīti to lielā apjoma un laika trūkuma dēļ, tādēļ automatizētie testi šādiem gadījumiem ir ļoti parocīgi.

Ir svarīgi izvēlēties atbilstošu ietvaru, pēc kura vadlīnijām tiek izstrādāti automatizētie testi, jo pareizā ietvara izvēle krietni atvieglos testu uzturēšanu.

Mainoties sistēmas funkcionalitātei, ir nepieciešams atjaunot un papildināt automatizētos testus.

Regulāra automatizēto testu izpilde nodrošina laicīgu kļūdu atrašanu un mazākas to novēršanas izmaksas.

Algu aprēķina modulis sastāv no lietotāja saskarnes un datu bāzes objektiem, tādēļ šajās sistēmās ir jātestē ne tikai lietotāja saskarne, bet arī datu bāze, izmantojot vienību testus.

Automatizētajiem testiem, kas izstrādāti uz *utPLSQL* un *Selenium* ietvara *Visual Studio* programmatūrā, ir iespējams vienkārši noskaidrot kļūdas rašanās cēloni. Var noskaidrot, kurā testā ir radusies kļūda, kā arī kura ir bijusi konkrētā pārbaude, kas ir izpildījusies neveiksmīgi. Šī informācija var palīdzēt neilgā laikā izlabot kļūdu.

Automatizēto testu izstrāde Algu aprēķina modulī palīdz pārbaudīt sarežģītu algoritmu izpildes korektumu un palīdz pārliecināties, ka ir pieejama katra forma, kontrole un darbība lietotāja saskarnē.

Ar *Selenium* automatizētajiem ar roku rakstītajiem testiem var pārbaudīt jebkādu konstrukciju tīmekļa lietotnes sistēmu. Tas ir ļoti noderīgi Algu aprēķina un Darba laika uzskaites formu testēšanā, jo tās sastāv no labojamām tabulām ar ļoti daudz ievadlaukiem, saraksta ar interaktīviem elementiem, dažādu veidu ievades elementiem un pogām, kuru testēšanu ir sarežģīti veikt ar testiem, kas izstrādāti pēc "ierakstīt un atskaņot" principa.

Datu bāzu automatizētie vienību testi palīdz pārliecināties, ka katra datu bāzē izstrādātā vienība strādā pareizi, jo datu bāzē tiek uzglabātas funkcijas un procedūras, kuras nav iespējams tieši izsaukt, izmantojot lietotāja saskarni.

Ieplānota automatizēto datu bāzu testu izpilde naktī nodrošina testu veikšanu laikā, kad ar sistēmu nestrādā lietotāji, tādējādi neietekmējot sistēmas veiktspēju darba laikā.

# IZMANTOTĀ LITERATŪRA UN AVOTI


[AAAPS] Allround Automations PL/SQL Developer. [tiešsaiste]. – [atsauce 19.03.2018.]. Pieejams: https://www.allroundautomations.com/plsqldev.html

[ASPMO] ASP.NET MVC Overview. [tiešsaiste]. – [atsauce 19.03.2018.]. Pieejams: https://msdn.microsoft.com/en-us/library/dd381412(v=vs.108).aspx

[BAVMT] Automated vs. Manual Testing: The Pros and Cons of Each. [tiešsaiste]. – [atsauce 08.01.2018.]. Pieejams: http://www.base36.com/2013/03/automated-vs-manual-testing-the-pros-and-cons-of-each/

[BQLTA] 3 Levels of Testing Automation. [tiešsaiste]. – [atsauce 16.01.2018.]. Pieejams: http://blog.qatestlab.com/2015/08/19/levels-testing-automation/

[CAUTT] Automated Testing | Types of Tests. [tiešsaiste]. – [atsauce 16.01.2018.]. Pieejams: http://www.continuousagile.com/unblock/test_types.html

[CPWIS] What is software testing? What are the different types of testing? [tiešsaiste]. – [atsauce 03.04.2018.]. Pieejams: https://www.codeproject.com/Tips/351122/What-is-software-testing-What-are-the-different-ty

[DATAS] Top 10 Automated Software Testing Tools. [tiešsaiste]. - [atsauce 03.04.2018.]. Pieejams: https://dzone.com/articles/top-10-automated-software-testing-tools

[DMCTS] Get started with Selenium testing in a CD pipeline. [tiešsaiste]. – [atsauce 03.04.2018.]. Pieejams: https://docs.microsoft.com/en-us/vsts/build-release/test/continuous-test-selenium?view=vsts

[DMUUA] Use UI Automation To Test Your Code. [tiešsaiste]. – [atsauce 03.04.2018.]. Pieejams: https://docs.microsoft.com/en-gb/visualstudio/test/use-ui-automation-to-test-your-code

[DXGVD] Responsive Layout. [tiešsaiste]. – [atsauce 23.03.2018.]. Pieejams: https://demos.devexpress.com/MVCxGridViewDemos/Adaptivity/ResponsiveLayout?device=tablet&rotate=0

[EDTAT] Elfriede Dustin. The Automated Testing Life-cycle Methodology (ATLM). [tiešsaiste]. – [atsauce 03.01.2018.]. Pieejams: https://www.cmcrossroads.com/sites/default/files/article/file/2015/Automating%20Software%20Testing-%20A%20Life-Cycle%20Methodology.pdf

[ETVMR] Terminu vārdnīca (Moodle rīks). [tiešsaiste]. – [atsauce 03.01.2018.]. Pieejams: https://estudijas.lu.lv/mod/glossary/view.php?id=47882&mode=search&hook=testēšana&sortkey=&sortorder=asc&fullsearch=1&page=3

[HLGAT] Testēšanas metodes. [tiešsaiste]. – [atsauce 03.01.2018]. Pieejams: http://home.lu.lv/~garnican/Testesan/test_n40.htm

[IECRT] What is Regression testing in software? [tiešsaiste]. – [atsauce 21.01.2018.]. Pieejams: http://istqbexamcertification.com/what-is-regression-testing-in-software/

[ISWIS] What is Software Testing? [tiešsaiste]. – [atsauce 03.01.2018.]. Pieejams: http://istqbexamcertification.com/what-is-software-testing/

[KAHDA] Kolawa, Adam; Huizinga, Dorota**.** Automated Defect Prevention: Best Practices in Software Management. Wiley-IEEE Computer Society Press, 2007. 74 lpp.

[KATKS] Katalon Studio: Free Powerful Automated Testing Tool. [tiešsaiste]. – [atsauce 13.04.2018.]. Pieejams: https://www.katalon.com/katalon-studio/

[KTQAS] Quality Assurane Services. [tiešsaiste]. – [atsauce 03.01.2018.]. Pieejams: https://www.kelltontech.com/quality-assurance-services

[LBTRT] Top 8 Reasons to Avoid “Record and Playback” in Test Automation. [tiešsaiste]. – [atsauce 03.04.2018.]. Pieejams: https://leaptest.com/blog/top-8-reasons-to-avoid-record-and-playback-in-test-automation

[LZATE] LZA Terminoloģijas komisija. [tiešsaiste]. – [atsauce 03.04.2018.]. Pieejams: http://termini.lza.lv/

[MVFPH] Microsoft Visual FoxPro 9.0. [tiešsaiste]. – [atsauce 17.04.2018.]. Pieejams: https://docs.microsoft.com/en-us/previous-versions/visualstudio/foxpro/mt490117(v%3dmsdn.10)

[ODORJ] Jenkins nepārtrauktās integrācijas sistēma. [tiešsaiste]. – [atsauce 16.04.2018.]. Pieejams: https://odo.lv/Recipes/Jenkins

[ODOTR] Testēšanas rokasgrāmata. [tiešsaiste]. – [atsauce 03.04.2018.]. Pieejams: https://odo.lv/ftp/docs/Testesanas_rokasgramata.pdf

[ORAUT] Performing a Unit Test of Your PL/SQL in Oracle SQL Developer 2.1 . [tiešsaiste]. – [atsauce 17.04.2018.]. Pieejams: http://www.oracle.com/webfolder/technetwork/tutorials/obe/db/11g/r2/prod/appdev/sqldev/sqldev_unit_test/sqldev_unit_test_otn.htm

[OWTAT] Oracle: When to Automate Your Testing (and When Not To). [tiešsaiste]. – [atsaue 08.01.2018.]. Pieejams: http://www.oracle.com/technetwork/cn/articles/when-to-automate-testing-1-130330.pdf

[QATAP] Test Automation Principles. [tiešsaiste]. – [atsauce 08.01.2018.]. Pieejams: http://www.qaautomation.net/?p=287

[QWADX] What actually is DevExpress and what is its relationship with Visual Studio and ASP.NET.? [tiešsaiste]. – [atsauce 04.04.2018]. Pieejams: https://www.quora.com/What-actually-is-DevExpress-and-what-is-its-relationship-with-Visual-Studio-and-ASP-NET

[RDBTD] Differences between the different levels of tests. [tiešsaiste]. – [atsauce 03.01.2018.]. Pieejams: https://reqtest.com/uncategorised/differences-between-the-different-levels-of-tests/

[SBLAT] Test Automation Frameworks. [tiešsaiste]. – [atsauce 17.01.2018.]. Pieejams: https://smartbear.com/learn/automated-testing/test-automation-frameworks/

[SBRFU] Recommendations for unit testing PL/SQL programs. [tiešsaiste]. – [atsauce 17.04.2018.]. Pieejams: http://stevenfeuersteinonplsql.blogspot.com/2015/03/recommendations-for-unit-testing-plsql.html

[SBLAW] What is Automated Testing. [tiešsaiste]. – [atsauce 03.01.2018.]. Pieejams: https://smartbear.com/learn/automated-testing/what-is-automated-testing/

[SGSWW] Getting Started with WebDriver in C# Using Visual Studio. [tiešsaiste]. – [atsauce 03.04.2018.]. Pieejams: https://saucelabs.com/resources/articles/getting-started-with-webdriver-in-c-using-visual-studio

[STATT] How to Develop Test Scripts Using Top 5 Most Popular Test Automation Frameworks (Examples). [tiešsaiste]. – [atsauce 17.01.2018.]. Pieejams: http://www.softwaretestinghelp.com/automation-testing-tutorial-5/

[STCSA] Sanity Testing. [tiešsaiste]. – [atsauce 21.01.2018.]. Pieejams: http://www.softwaretestingclass.com/sanity-testing/

[STCSM] Smoke Testing. [tiešsaiste]. – [atsauce 21.01.2018.]. Pieejams: http://www.softwaretestingclass.com/smoke-testing/

[STFGB] Gray Box Testing. [tiešsaiste]. – [atsauce 16.01.2018.]. Pieejams: http://softwaretestingfundamentals.com/gray-box-testing/

[TBHTA] Automated GUI Testing: How to Get It Right. [tiešsaiste]. – [atsauce 16.01.2018.]. Pieejams: https://testlio.com/blog/how-to-automate-gui-testing/

[TCBWT] Black-box vs White-box testing. [tiešsaiste]. – [atsauce 03.01.2018.]. Pieejams: https://technologyconversations.com/2013/12/11/black-box-vs-white-box-testing/

[TPTAF] Test Automation Framework. [tiešsaiste]. – [atsauce 17.01.2018.]. Pieejams: https://www.techopedia.com/definition/30670/test-automation-framework

[TPUIT] User Interface Testing. [tiešsaiste]. – [atsauce 03.04.2018.]. Pieejams: https://www.tutorialspoint.com/software_testing_dictionary/use_interface_testing.htm

[TSTIT] What is Integration Testing? [tiešsaiste]. – [atsauce 11.01.2018.]. Pieejams: http://toolsqa.com/software-testing/integration-testing/

[TSTST] What is System Testing? [tiešsaiste]. – [atsauce 11.01.2018.]. Pieejams: http://toolsqa.com/software-testing/system-testing/

[TSTUA] What is User Acceptance Testing? [tiešsaiste]. – [atsauce 11.01.2018.]. Pieejams:

http://toolsqa.com/software-testing/user-acceptance-testing-uat/

[TSTUT] What is Unit Testing? [tiešsaiste]. – [atsauce 11.01.2018.]. Pieejams: http://toolsqa.com/software-testing/unit-testing/

[TTABE] The test automation basics every software developer should know. [tiešsaiste]. – [atsauce 16.01.2018.]. Peejams: https://techbeacon.com/test-automation-basics-every-software-developer-should-know

[TTASP] ASP.NET. [tiešsaiste]. – [atsauce 17.04.2018.]. Pieejams: https://techterms.com/definition/aspnet

[UTPLS] utPLSQL. [tiešsaiste]. – [atsauce 11.04.2018.]. Pieejams: http://utplsql.org/

[VFPHO] Visual FoxPro Home. [tiešsaiste]. – [atsauce 04.04.2018.]. Pieejams: https://msdn.microsoft.com/en-us/vfoxpro/bb190225.aspx

[WICSM] Client – server model – Wikipedia. [tiešsaiste]. – [atsauce 03.04.2018.]. Pieejams: https://en.wikipedia.org/wiki/Client%E2%80%93server_model

[WIKCD] Cursor (databases) – Wikipedia. [tiešsaiste]. – [atsauce 17.04.2018.]. Pieejams: https://en.wikipedia.org/wiki/Cursor_(databases)

[WIKSD] Synonym (satabase) – Wikipedia. [tiešsaiste]. – [atsauce 17.04.2018.]. Pieejams: https://en.wikipedia.org/wiki/Synonym_(database)

[WIMVS] Microsoft Visual Studio. [tiešsaiste]. – [atsauce 23.03.2018.]. Pieejams: https://en.wikipedia.org/wiki/Microsoft_Visual_Studio

[WIODC] Open Database Connectivity – Wikipedia. [tiešsaiste]. – [atsauce 04.04.2018.]. Pieejams: https://en.wikipedia.org/wiki/Open_Database_Connectivity

[WIOSD] Oracle SQL Developer. [tiešsaiste]. – [atsauce 17.04.2018.]. Pieejams: https://en.wikipedia.org/wiki/Oracle_SQL_Developer

[WIPHP] PHP – Vikipēdija. [tiešsaiste]. – [atsauce 17.04.2018.]. Pieejams: https://lv.wikipedia.org/wiki/PHP

[WISEL] Selenium (software). [tiešsaiste]. – [atsauce 03.04.2018.]. Pieejams: https://en.wikipedia.org/wiki/Selenium_(software)

[XPPOT] Principles of Test Automation. [tiešsaiste]. – [atsauce 08.01.2018.]. Pieejams: http://xunitpatterns.com/Principles%20of%20Test%20Automation.html

[ZZRIS] Risinājumi | ZZ Dats. [tiešsaiste]. – [atsauce 03.04.2018.]. Pieejams: http://www.zzdats.lv/risinajumi/

[ZZRVG] Resursu vadība un grāmatvedības uzskaite | ZZ Dats. [tiešsaiste]. – [atsauce 03.04.2018.]. Pieejams: http://www.zzdats.lv/risinajumi/resursu-vadibas-sistema-g-vedis-3/

Maģistra darbs “Algu aprēķina moduļa automatizētā testēšana” izstrādāts LU Datorikas fakultātē.

Darba teksta galīgā versija izgatavota 21.05.2018.

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 maģistratūras sekretariātā _____.
(Iesniegšanas datums)

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

Studiju metodiķe : _____.
(Metodiķes paraksts)

Recenzents : __Docents, Dr.dat. Vineta Arnicāne_____
(Akad.amats, zin.grāds, vārds, uzvārds)

Darbs aizstāvēts maģistra gala pārbaudījuma komisijas sēdē

_____ prot. Nr. _____

(Darba aizstāvēšanas datums)

Komisijas sekretārs: _____
(Sekretāra paraksts)