LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

EDGARS DIEBELIS

PROGRAMMATŪRAS PAŠTESTĒŠANA

PROMOCIJAS DARBS
DATORZINĀTNES DOKTORA GRĀDA IEGŪŠANAI

Nozare: datorzinātnes
Apakšnozare: datu apstrādes sistēmas un datortīkli

Zinātniskais vadītājs
profesors Dr. sc. comp. JĀNIS BIČEVSKIS

RĪGA – 2012

ANOTĀCIJA

Darbs satur pārskatu par 5 gadu pētījumu rezultātiem paštestēšanas jomā. Paštestēšana 2007.gadā tika definēta kā viena no viedttehnoloģiju virzieniem, kurus apvieno vēlme aprīkot programmatūru ar dzīvu būtņu īpašībām – piemēroties mainīgai ārējai videi, optimizēt sevi, aizsargāties pret apdraudējumiem un adekvāti reaģēt neparedzamās situācijās. Paštestēšanas mērķis – dot iespēju jebkurā dzīves cikla momentā pārliecināties par programmatūras darbības pareizību. Pētījums veikts vairākos etapos: vispirms formulēta paštestēšanas pieejas koncepcija, funkcionalitāte un tās lietojumi dažādās programmatūras darbības vidēs, kam seko paštestēšanas funkcionalitātes implementācija, iebūvējot izstrādājamajā programmatūrā testēšanas atbalsta iespējas. Pēc tam paštestēšanas pieeja salīdzināta ar tradicionālo testēšanas atbalsta rīku iespējām un aprobēta konkrētā banku informācijas sistēmā, kā arī novērtēta paštestēšanas pieejas efektivitāte. Pētījuma gala secinājums – paštestēšana sniedz virkni priekšrocību programmatūras kvalitātes nodrošināšanai par relatīvi zemām izmaksām, tajā pašā laikā nodrošinot arī to funkcionalitāti, ko nodrošina tradicionālie testēšanas atbalsta rīki.

Atslēgvārdi

Programmatūras testēšana, Viedtehnoloģijas, Paštestēšana, Testu automatizēšana.

SATURA RĀDĪTĀJS

APZĪMĒJUMU SARAKSTS

Apzīmējums	Skaidrojums
IT	Informācijas tehnoloģijas
HTTP	Hiperteksta transporta protokols
HTTPS	Hiperteksta drošas pārsūtīšanas protokols
SOAP	Vienkāršais objektu piekļuves protokols
Testa punkts	Komanda, pie kuras tiek izpildītas sistēmas testēšanas darbības. Testa punkts iekļauj arī kontrolpunktus.
Kontrolpunkts	Vieta datora programmā, kurā pārbauda procesa izpildes pareizību (*checkpoint*) [letonika.lv, 2012a].
SLIM	Vienkāršu sarakstu izsaukšanas metode SLIM (*Simple List Invocation Method*)
RFB	Attālināto kadru buferis (*Remote framebuffer*) – vienkāršs protokols attālinātai piekļuvei grafiskajai lietotāja saskarnei [RealVNC, 2012].
VNC	Virtuālā tīkla skaitļošana VNC (*Virtual Network Computing*) – grafiskās darbvirsmas koplietošanas sistēma, kas izmanto RFB protokolu, lai attālināti kontrolētu citu datoru (tas pa tīklu pārsūta klaviatūras un peles notikumus no viena datora uz citu un saņem pretī izmaiņas darbvirsmas attēlā) [Wikipedia, 2012a].
TCP/IP	Pārraides vadības protokols / intertīkla protokols
RFT	Rational Functional Tester
VVUS	Valūtas un vērtspapīru uzskaites sistēma
API	Lietojumprogrammas saskarne (*Application Program Interface*) ir specifikācijā paredzēta saskarne, kas tiek izmantota programmatūras komponentu savstarpējā komunikācijā [Wikipedia, 2012f].
TDD	Testu virzīta izstrāde (Test-driven development) ir programmatūras izstrādes process, kas balstās uz īsiem atkārtotiem izstrādes cikliem: sākotnēji programmētājs izveido kļūdainu automatizēti izpildāmu testa piemēru, kas nosaka vēlamo uzlabojumu vai jaunu funkciju, tad izstrādā kodu veiksmīgai testa izpildei un visbeidzot sakārto kodu atbilstoši pieņemtajiem standartiem [Wikipedia, 2012g].

IEVADS

Informācijas tehnoloģiju straujā attīstība ir izveidojusi nebijušas sarežģītības kompleksus, kurus [Kephart, Chess, 2003] autori novērtē ar vārdiem „skaitļošanas sistēmu sarežģītība tiecas uz cilvēka iespēju robežām". Viņi norāda, ka visaptverošās (*pervasive*) skaitļošanas sapnis – triljoniem datorsistēmu, kas vienlaikus pieslēgtas internetam – ātri vien var kļūt nevadāmas un pārvērsties par ļaunu „murgu". Informācijas sistēmas, kuras parasti tiek veidotas kā universālas sistēmas, konkrētam lietojumam tiek piemērotas ar konfigurēšanas palīdzību. "Ierindas" lietotājam, kuru skaits strauji pieaug, to paveikt kļūst neiespējami. [Kephart, Chess, 2003] Autori paredz vēl tālāku informācijas sistēmu sarežģītības palielināšanos, cilvēkam padarot par gandrīz neiespējamu veikt sistēmu instalēšanas, konfigurēšanas, optimizēšanas un uzturēšanas darbus.

Vienu no iespējamiem problēmas risinājumiem 2001. gadā ir piedāvājis IBM ar autonomisko skaitļošanas (*autonomic computing*) manifestu [IBM, 2001]. Tā galvenais mērķis ir veidot informācijas sistēmas, kas ir spējīgas pašas sevi vadīt (*self-management*), un tādejādi pārvarēt barjeru starp lietotājiem un arvien sarežģītāko informāciju tehnoloģiju pasauli.

Manifestā tika piedāvātas četras galvenās īpašības, kas raksturo autonomisko skaitļošanu:

- Paškonfigurācija (*Self-configuration*). Automatizēta komponentu un sistēmu konfigurācija nodrošina augšējā līmeņa politiku. Pārējā sistēma pielāgojas automātiski un nemanāmi.
- Pašoptimizācija (*Self-optimization*). Komponentes un sistēmas pastāvīgi meklē iespējas uzlabot savu veiktspēju un efektivitāti.
- Pašatjaunošanās (*Self-healing*). Sistēma automātiski atklāj, diagnosticē un labo atklātās programmatūras un aparatūras problēmas.
- Pašaizsardzība (*Self-protection*). Sistēma automātiski aizsargājas pret ļaunprātīgiem uzbrukumiem vai kaskādes neveiksmēm. Tā izmanto agrīnu brīdināšanu, lai paredzētu un novērstu visas sistēmas nepilnības.

2003.gadā IBM šo sarakstu precizēja līdz astoņiem raksturlielumiem [Kephart, Chess, 2003], pievienojot spēju „pazīt pašai sevi" un vadīt savus resursus, pazīt savas darbības vidi un atbilstoši reaģēt – adaptēt un adaptēties, darboties heterogēnā pasaulē atbilstoši tās atvērtiem standartiem, kā arī apslēpt risinājuma tehnisko sarežģītību. Šobrīd jāatzīst, ka manifestā izvirzītie mērķi ir sasniegti tikai daļēji un turpinās autonomisko sistēmu teorētiska izpēte un praktiska realizācija [Kephart, 2011]. 2011. gadā bija pagājuši 10 gadi kopš IBM manifesta, kuru laikā ir mēģināts implementēt manifestā piedāvātās idejas. Kā vienu no piemēriem, kur šīs idejas novestas līdz realizācijai reālās

sistēmās, var minēt programmu versiju automātisku atjaunošanos mobilās iekārtās. Kaut arī paštestēšana kā atsevišķa autonomiskas skaitļošanas sadaļa nav minēta, tā neapšaubāmi kalpo to pašu mērķu sasniegšanai kā autonomiskā skaitļošana, par kuras aktualitāti nav jāšaubās.

2007. gadā ar līdzīgu kā autonomiskas sistēmas mērķi tika piedāvāta viedtehnoloģiju pieeja [Bičevska, Bičevskis, 2007]. Tā piedāvāja virkni praktiski realizējamus uzlabojumus informācijas sistēmu funkcionalitātē, kas ļautu vienkāršot to uzturēšanu un ikdienas lietošanu, tādejādi tuvojoties autonomisko sistēmu galvenajam mērķim. Kā viedtehnoloģiju implementēšanas galvenais paņēmiens tiek piedāvāts nevis autonomu universālu komponenšu izveide, bet iekļaut viedtehnoloģiju iespējas konkrētās sistēmas programmās tieši.

Zemāk minētas tipiski, empīrisku eksperimentu un praktisku pētījumu ietvaros identificēti septiņi viedtehnoloģiju veidi, taču tie, protams, nav vienīgie iespējamie:

- Dinamisks biznesa modelis – mainoties darījumprocesiem, informācijas sistēmas funkcionalitāte, pateicoties formalizētam biznesa modelim, mainās automātiski. To ir iespējams sasniegt lietojot Domēna specifiskās valodas (*Domain Specific Language*) pieeju [Bicevskis et. al., 2011], kura piedāvā specificēt biznesa modeli tādā precizitātes līmenī, ka biznesa modelis ir izpildāms (interpretējams) tieši. Šī pieeja ar labiem rezultātiem ir realizēta [Bicevskis et. al., 2011] [Cerina-Berzina et. al., 2011] vairākās notikumu orientētās informācijas sistēmās Latvijā.
- Iebūvēta versiju pārvaldība un datu sinhronizācija – automātiska programmatūras versiju izplatīšana no centrālās glabātuves uz lokālām darbstacijām un serveriem, ieskaitot datu struktūru konversijas un datu transportu. Pētījuma rezultātus [Bičevska, Bičevskis, 2008a] ir izdevies ieviest finanšu un budžeta pārskatu sistēmā visā Latvijā.
- Ārējās vides testēšana – automatizēta ārējās (produkcijas) vides parametru pārbaude un atkarībā no konstatētajiem rezultātiem iespēja pielāgot programmatūru specifiskām vides īpašībām un/vai informēt izstrādātājus par nepieciešamajām/veiktajām darbībām. Piedāvātais risinājums [Rauhvargers, 2008] [Rauhvargers, Bičevskis, 2008] [Rauhvargers, Bičevskis, 2009] paredz speciālā valodā izveidot programmas pasi, kas satur prasības ārējai videi (operētājsistēma, bibliotēku versijas, datoru tehniskie parametri u.c.), lai programmatūra spētu optimāli darboties. Pētījuma rezultātus lieto vairākās sistēmās visā Latvijā.
- Paštestēšana – programmatūras spēja pārbaudīt pašas integritāti un darbspējas, izpildot iepriekš uzkrātus testpiemērus ar programmatūrā iebūvētām testēšanas atbalsta iespējām. Promocijas darbs ietver līdz šim veikto pētījumu [Bičevska, Bičevskis,

2008b] [Bičevska, Bičevskis, 2008c] [Diebelis et. al., 2009] [Diebelis, Bičevskis, 2010] [Diebelis, Bičevskis, 2011] [Diebelis, 2012] [Vizulis, Diebelis, 2012] [Diebelis, Bičevskis, 2012] rezultātus, kas detalizēti aplūkoti nākamajās nodaļās.

- Datu kvalitātes uzraudzība – pastāvīga datu glabātuvē uzkrāto datu iekšējās nepretrunības un pilnīguma kontrole (pārraudzība).
- Pieejamības testēšana – informācijas sistēmas nepārtrauktas pieejamības uzraudzība, ieskaitot ziņošanas mehānismu par programmatūras darbības statusu un identificētajām papildus nepieciešamajām komponentēm/ versijām.
- Drošības un slodzes testēšana – sistēmas drošības, veiktspējas un slodzes situācijas uzraudzība.

No 2007.gada viedtehnoloģiju jomā ir gūti reāli praktiski rezultāti vismaz četrās jomās: dinamisks biznesa modelis, iebūvēta versiju pārvaldība un datu sinhronizācija, ārējās vides testēšana un paštestēšana.

Darba zinātniskā novitāte – nodrošinot programmatūrā iebūvētu savas darbības korektuma pārbaudi, tiek piedāvāta jauna oriģināla pieeja programmatūras testēšanas jomā, kas kalpo autonomisku sistēmu un viedtehnoloģiju mērķu sasniegšanai. Paštestēšana pieeja piedāvā iespēju pārbaudīt programmu integritāti un darbspējas, izpildot iepriekš uzkrātus testpiemērus ar programmatūrā iebūvētām testēšanas atbalsta iespējām. Tā rezultātā:

- paštestēšanas pieeja nodrošina iespēju veikt testēšanu produkcijas vidē;
- paštestēšanas pieeja nodrošina iespēju lietotājiem bez padziļinātām IT zināšanām definēt un izpildīt testu piemērus;
- paštestēšanas pieeja nodrošina iespēju veikt ārējo saskarņu (sadarbības ar ārējām sistēmām) testēšanu;
- paštestēšanas pieeja nodrošina iespēju veikt pilnu sistēmas testēšanu, piemēram, reģistra tipa sistēmas gadījumā, sākot ar datu ievadi līdz ievadīto datu saglabāšanai datu bāzē, nosūtīšanai ārējai saskarnei, eksportam failā utt.;
- paštestēšanas pieeja nodrošina iespēju veikt sistēmas testēšanu, kas iepriekš nav īpaši jāsagatavo testu izpildei (atbilstošu datu sagatavošana testu izpildei). Paštestēšanas pieeja nodrošina kontroli, vai uzkrātos testus konkrētajā situācijā ir iespējams izpildīt;
- paštestēšanas piedāvātās iespējas nenodrošina neviens no līdz šim apzinātiem un pieejamiem testēšanas atbalsta rīkiem, un tādejādi ir neapšaubāmi inovatīva izstrāde.

Darbs sastāv no piecām nodaļām, kur pirmā veltīta darbā lietoto jēdzienu skaidrojumam, bet pārējās četras atbilstoši galveno jautājumu analīzei:

- paštestēšanas koncepcija;
- paštestēšanas implementācija;
- paštestēšanas un tradicionālo testēšanas rīku iespēju salīdzinājums;
- paštestēšanas efektivitātes mērījumi.

Paštestēšanas koncepcija

Paštestēšanas ideja paredz iespēju programmatūrai pirms darbības uzsākšanas sevi automātiski pārbaudīt līdzīgi kā pēc datora ieslēgšanas tas nekavējoties pārbauda savu gatavību darbam. Ieslēdzot datoru, tiek veikta datora paštestēšana, tiek automātiski veiktas pārbaudes vai darba kārtībā ir visas nepieciešamās datora sastāvdaļas, tādas kā cietais disks, operatīvā atmiņa, procesors, video karte, skaņas karte utt. Ja kāda no sastāvdaļām ir bojāta vai nav pieejama, kā rezultātā datora darbība nav iespējama, tad par to tiek paziņots lietotājam. Paštestēšanas mērķis ir analoģisks kontrolei, kas tiek veiktas tūlīt pēc datora ieslēgšanas – pirms sistēmas lietošanas automātiski pārbaudīt, vai sistēma nesatur kļūdas, kas traucē sistēmas lietošanu. Pirmie paštestēšanas pieejas pētījumi, kas iekļauti šajā darbā, ir atspoguļoti 2009. gada maģistra darbā [Takeris, 2009] un 2008., 2009. gadā publicētām zinātniskām publikācijām [Bičevska, Bičevskis, 2008b] [Bičevska, Bičevskis, 2008c] [Diebelis et. al., 2009].

Būtiska loma paštestēšanas pieejas realizācijā ir testa punktiem, kas tiek iebūvēti pašā sistēmā un nodrošina lietotāju veikto darbību un ierakstīto testa piemēru izpildes reģistrēšanu testa piemēra failā. Testa punktu iebūvēšanu sistēmā nodrošina to, ka sistēma tiek testēta no sistēmas „iekšpuses", nevis no „ārpuses" kā to veic trešo kompāniju izstrādātie testēšanas rīki. Iebūvējot testēšanas darbības pašā sistēmā ir iespējams notestēt sistēmas funkcionalitāti, ko nespēj notestēt rīki, kas veic sistēmas testēšanu no ārpuses.

Paštestēšanas implementācija

Lai paštestēšanas pieeja nepaliktu tikai teorijas līmenī, 2010. gadā tika izstrādāts paštestēšanas rīks, kas reālos projektos spētu pierādīt paštestēšanas pieejas lietderību. Paštestēšanas rīks sastāv no diviem pamata moduļiem:

- paštestēšanas modulis – bibliotēka (.dll fails), kas satur paštestēšanas funkcijas, kuru izsaukumi tiek iebūvēti testējamā sistēmā;
- paštestēšanas testu pārvaldes modulis – lietojumprogramma, kas nodrošina jaunu testu veidošanu, esošu testu rediģēšanu, izpildi, rezultātu salīdzināšanu, rīka konfigurēšanu un citas iespējas.

Darbā aprakstītā rīka realizācija balstīta uz aizstāvētiem maģistra, bakalaura darbiem [Vizulis, 2011] [Mihailovs, 2010], kuru vadītājs ir darba autors un 2010., 2011. gadā publicētām zinātniskām publikācijām [Diebelis, Bičevskis, 2010] [Diebelis, Bičevskis, 2011]. Rīks pašreiz netiek lietots reālā produkcijā, bet ir eksperimentāli pārbaudīts. Paštestēšanas tehnoloģija testa režīmā ir aprobēta banku sistēmas Vērtspapīru un valūtas uzskaites sistēmā. Šajā gadījumā banka noteica aprobāciju, kas ir sensitīva pret ar biznesu funkcionalitāti nesaistītu rīku darbībām produkcijas vidē.

Paštestēšanas un tradicionālo testēšanas rīku iespēju salīdzinājums

Paštestēšanas pieejas veiksmīgai tālākai attīstībai nepieciešams novērtēt, kādi ir līdzīgu rīku būvēšanas teorētiskie pamati un kādu vietu šāda pieeja ieņem starp pasaulē populārākajiem testēšanas rīkiem. Salīdzināmo rīku izvēle tika veikta ņemot vērā uzņēmuma „Automated Testing Institute" [ATI, 2012a] viedokli. Uzņēmums katru gadu nosaka vadošos automatizētās testēšanas rīkus dažādās nominācijās. Darbā paštestēšanas pieejas novērtēšanai izvēlēti rīki, kas pēdējo gadu laikā kādā no nominācijām ir ieguvuši balvu. Darbā aprakstītais rīku salīdzinājums balstīts uz maģistra darbu [Vizulis, 2011], kura vadītājs ir darba autors.

Paštestēšanas pieeja nodrošina virkni standarta funkcionalitātes (tādas kā testu ierakstīšana un atspēlēšana, automatizētu testu izpildi, testu pārvaldību u.c.), ko nodrošina lielākā daļa testēšanas rīku. Bez standarta funkcionalitātes paštestēšanas pieeja nodrošina vairākas iespējas, ko nenodrošina darbā aprakstītie testēšanas rīki: testēšana produkcijas vidē, lietotājam bez IT zināšanām veidot testus un veikt baltās kastes testēšanu,

Paštestēšanas efektivitātes mērījumi

Loģisks rezultāts idejai, idejas salīdzinājumam ar līdzīgām, idejas realizācijai ir idejas novērtējums (efektivitātes mērījumi). Lai veiktu paštestēšanas pieejas efektivitātes mērījumus, paštestēšanas pieeja netika iestrādātā esošā sistēmā. Iestrādājot paštestēšanas pieeju sistēmā, nebūtu iespējams noteikt paštestēšanas pieejas ieguvumu, jo nebūtu iespējams salīdzināt to ar situāciju, kad sistēmā paštestēšanas pieeja netiktu iebūvēt. Tādēļ autors veica konkrētas sistēmas esošu kļūdu pieteikumu analīzi, norādot, vai paštestēšanas pieeja konkrēto kļūdu būtu identificējusi.

Paštestēšanas koncepts tika attīstīts ERAF 2.1.2.2.1. apakšaktivitātes „Jaunu produktu un tehnoloģiju izstrāde" projekta Nr. L-JPI-09-0012 "Programmatūras paštestēšanas tehnoloģija" ietvaros laika periodā no 13.07.2009 līdz 13.01.2011. Šajā projektā bija iesaistīts pētnieku un programmētāju kolektīvs, kurā promocijas darba autors darbojās kā pētnieks un faktiskais projekta vadītājs. Galveno zinātnisko rezultātu izstrādē darba autora ieguldījums ir šāds: otrā un trešā nodaļa – 70-80% , ceturtā nodaļa – 50% un piektā nodaļa – 100% no kopējā apjoma.

1. PROGRAMMATŪRAS TESTĒŠANA

Amerikāņu zinātnieks Rodžers Pressmans programmatūras testēšanai devis šādu definīciju: „Programmatūras testēšana ir ļoti būtisks programmatūras kvalitātes nodrošināšanas elements un attēlo galīgo specificēšanas, projektēšanas un koda veidošanas pārskatu" [Pressman, 2000]. Lai parādītu testēšanas termina plašo nozīmi, darbā ir iekļautas vēl šādas divas globālajā tīmeklī atrodamas definīcijas: „Programmatūras testēšana ir programmatūras darbības pārbaude, izmantojot testdatus. Testēšanas mērķis var būt defekta atklāšana, tā atrašanās vietas lokalizēšana vai arī testējamā objekta dinamisko parametru (piemēram, ātrdarbības) noskaidrošana" [letonika.lv, 2012b] un „Programmatūras testēšana ir izmeklēšana, kuru veic, lai ieinteresētajām personām sniegtu informāciju par produkta vai pakalpojuma kvalitāti" [Kaner, 2006].

Izlasot definīcijas, ir noprotams, ka programmatūras testēšanas jēdziens ir ļoti plašs. Tādēļ, lai precizētu programmatūras testēšanas būtību, šajā nodaļā tiks apskatītas testēšanas metodes un līmeņi. Tas arī palīdzēs novērtēt to atbilstību turpmākajās nodaļās, apskatot paštestēšanas pieeju un testēšanas rīkus.

1.1. Testēšanas metodes

1.1.1. Baltās kastes jeb strukturālā testēšana

Baltās kastes testēšana koncentrējas uz programmatūras iekšējo struktūru. Šāda veida testēšanai nepieciešama pieeja testējamā objekta struktūrai, t.i., pirmkodam.

Izmantojot šo testēšanas metodi, testētājs atkarībā no izvēlētā testēšanas pilnības kritērija iegūst testus, kas:

- garantē, ka visi neatkarīgie ceļi modelī ir izpildīti vismaz vienu reizi;
- pārbauda visas loģisko operatoru patiesās un nepatiesās vērtības;
- izpilda visus ciklus to darbības robežās;
- izpilda iekšējās datu struktūras, lai noteiktu to pareizību. [Pressman, 2000]

1.1.2. Melnās kastes jeb funkcionālā testēšana

Melnās kastes testēšana koncentrējas uz programmatūras funkcionālajām prasībām. Piemēram, testētājs testē specifikācijā minēto funkcionalitāti un prasības, nodrošinot noteiktus ievaddatus. Testētājam nekas nav zināms par programmatūras iekšējo uzvedību un struktūru.

Šajā testēšanā parasti tiek meklētas šādas kļūdas:

- neatbilstoša vai iztrūkstoša funkcionalitāte;

- saskarnes kļūdas;
- datu struktūru vai ārējo datu piekļuves kļūdas;
- uzvedības vai veiktspējas kļūdas;
- inicializācijas un beigu kļūdas. [Pressman, 2000]

1.1.3. Pelēkās kastes testēšana

Pelēkās kastes testēšana apvieno abas iepriekšminētās testēšanas metodes. Gan baltās kastes, gan arī melnās kastes testēšana var tikt izmantota vienlaikus. Piemēram, vienību un zema līmeņa komponenšu testēšanai parasti izmanto strukturālo metodi, savukārt lielu komponenšu un sistēmu testēšanai visbiežāk pielieto funkcionālo metodi. [Beizer, 1995]

1.1.4. Secinājumi

Neiedziļinoties paštestēšanas būtībā, varētu šķist, ka paštestēšana viennozīmīgi izmanto melnās kastes metodi, jo, gan veidojot jaunu, gan arī izpildot testu, testētājam nav jāiedziļinās testējamā objekta iekšējā struktūrā, bet gan jāpārzina funkcionālās prasības. Pēc šāda principa parasti notiek funkcionālā testēšana. Tomēr paštestēšanā ļoti nozīmīga loma ir testa punktiem, kurus nepieciešams iestrādāt testējamās sistēmas kodā un līdz ar to jāiedziļinās arī tās iekšējā struktūrā. Šī iemesla dēļ paštestēšana pieeja nodrošina testēšanu gan pēc baltās, gan pelēkās, gan pelēkās kastes metodes.

1.2. Testēšanas līmeņi

Veidojot jaunus testus, izmantojot kādu no iepriekš minētajām metodēm, nepieciešama arī vienots testēšanas līmenis, pēc kura vadīties, jo testētājam jāzina, ko testēt, kā testēt, kad testēt un cik daudz testēt. Šo jautājumu atbildēšanai tiek izmantotas testēšanas līmeņi, tādēļ, lai būtu skaidra to būtība, turpmākajās apakšnodaļās apskatītas izplatītākās no tām.

1.2.1. Vienībtestēšana

Vienībtestēšana veic testēšanu pa atsevišķām sistēmas komponentēm vai moduļiem. Ar šādas testēšanas palīdzību var atklāt tikai komponentes vai moduļa kontekstā esošās kļūdas. Šo testu veidošanai, kā likums, nepieciešama pieeja testējamā objekta iekšējai struktūrai (baltās kastes metode).

1.2.2. Integrācijas testēšana

Integrācijas testēšana iedalās divos veidos:

- Neinkrementāla – vispirms savieno visas komponentes un tad testē visu sistēmu kopumā.
- Inkrementāla – atsevišķas komponentes tiek pakāpeniski savienotas līdz tiek iegūta sistēma. Testēšana notiek pēc katras komponentes pievienošanas. Komponenšu pievienošanu var sākt ar galveno komponenti (lejupejošā integrācija) vai arī ar atomārām komponentēm (augšupejošā integrācija) un tad pievienot tām saistītās komponentes. [Pressman, 2000]

1.2.3. Sistēmas testēšana

Sistēmas testēšana pārbauda vai pilnībā integrēta sistēma nodrošina tai izvirzītās prasības vidē, kas maksimāli tuvināta produkcijas videi [IEEE, 1990].

1.2.4. Regresa testēšana

Regresa testēšana ir izmainītās programmatūras selektīva atkārtota testēšana, lai pārliecinātos, ka kļūdas ir izlabotas un ka iepriekš strādājošajā funkcionalitātē izmaiņu rezultātā nav radušās jaunas kļūdas [Webopedia, 2012].

Parasti regresa testēšana tiek veikta automatizēti ar kādu testēšanas rīku, kas nodrošina testu kopu veidošanu, uzkrāšanu, atkārtotu izpildi un rezultātu analīzi.

1.2.5. Akcepttestēšana

Akcepttestēšana ir programmatūras vai aparatūras galīgā pārbaude, kas parāda izstrādātā produkta funkcionālās iespējas. Šī pārbaude nosaka, vai pārbaudāmais objekts darbojas atbilstoši līgumā fiksētajām prasībām (specifikācijai), t. i., vai tas atbilst akceptēšanas kritērijiem [letonika.lv, 2012c].

1.2.6. Secinājumi

Kā redzams, testēšanas līmeņu ir ļoti daudz. Katra savā veidā piedāvā paaugstināt testējamās sistēmas kvalitāti. Mūsdienās jaunās testēšanas pieejas nereti apvieno vairākus testēšanas līmeņus, no katra aizņemoties labākās īpašības un tādējādi uzlabojot testēšanas kvalitāti.

1.3. Nefunkcionālā testēšana

1.3.1. Atjaunošanās testēšana

Atjaunošanās testēšana paredzēta sistēmām, kurām ir ļoti svarīgi noteiktā laika posmā atjaunoties pēc bojājumiem. Testēšanas procesā sistēma tiek sabojāta dažādos veidos un pēc tam tiek pārbaudīts, vai atjaunošanās notiek veiksmīgi. [Pressman, 2000]

1.3.2. Drošības testēšana

Drošības testēšana ir pārbaude, vai sistēmā iebūvētie aizsardzības mehānismi spēj aizsargāt sistēmu no nesankcionētas piekļuves.

Jebkurai sistēmai iespējams nesankcionēti piekļūt. Svarīgi ir tas, cik daudz laika un resursu tam nepieciešams. Drošības testēšanas mērķis ir šo ieguldījumu padarīt lielāku par ieguvumu, nesankcionēti piekļūstot sistēmai. [Pressman, 2000]

1.3.3. Stresa testēšana

Stresa testēšanas mērķis ir novērtēt sistēmas uzvedību aiz specifikācijā noteiktām sistēmas lietošanas robežām [Spillner *et. al.*, 2007]. Stresa testēšana tiek veikta, lai novērtētu sistēmas darbību pārslodzes gadījumā.

1.3.4. Slodzes testēšana

Slodzes testēšanas mērķis ir izmērīt sistēmas uzvedību pieaugošas, ilgstošas slodzes apstākļos, piemēram, pieaugot lietotāju apstrādājamo transakciju skaitam [Spillner *et. al.*, 2007].

1.3.5. Veiktspējas testēšana

Veiktspējas testēšana novērtē sistēmas atbilstību specifikācijā noteiktajām veiktspējas prasībām [letonika.lv, 2012d]. Testēšanas mērķis ir pārbaudīt veiktspējas raksturlielumus, piemēram, reakcijas laiku, resursu izmantošanu u.c.

1.4. Funkcionālā testēšana

Funkcionālās testēšanas mērķis ir novērtēt sistēmas funkcionalitātes kvalitāti. Ar testu palīdzību tiek noteikts, vai sistēma darbojas atbilstoši definētajām pasūtītāja prasībām. Parasti funkcionālā testēšana tiek veikta, izmantojot lietotāja saskarni, padodot tai ievaddatus un pārbaudot izvaddatus.

1.5. Automatizēta testēšana

Testēšanas automatizēšana ir tādu testu izstrāde un izpilde, kurus var izpildīt bez cilvēka tiešas līdzdalības, kuri veic testējamās sistēmas aktuālā un sagaidāmā stāvokļa salīdzināšanu [IndicThreads, 2012].

Ja agrāk programmatūras testēšana notika tikai manuāli, testētājam cenošoties imitēt lietotāja veiktās darbības testējamajā sistēmā, tad mūsdienās bieži vien testēšanas process tiek automatizēts, jo praksē ir pierādījies, ka bieži vien lētāk ir ieguldīt resursus automatizētas testēšanas rīka izstrādē vai pielāgošanā, atbilstoši testēšanas projektam, nevis veikt daudzkārtēju manuālu sistēmas testēšanu. Protams, testu automatizēšanas ieviešanas izdevīgums ir jānovērtē katram programmatūras projektam atsevišķi, bet lielos projektos bez kvalitatīvas automatizētas testēšanas mūsdienās vairs nevar iztikt.

Šī iemesla dēļ pasaulē pastāv ļoti daudz dažādu automatizētas testēšanas rīku, kuri viens no otra atšķiras ar piedāvātajām iespējām un uzbūves. Savam testēšanas projektam izvēloties kādu no esošajiem rīkiem vai veidojot jaunu rīku, svarīgi zināt ne tikai tā piedāvātās iespējas, bet arī tā uzbūvi, jo automatizēta testēšana sniedz augstāku kvalitāti un efektivitāti, ja tā tiek balstīta uz kāda no testu automatizēšanas ietvariem un/vai pieejām. Protams, svarīgi ir arī tas, kurš no šiem ietvariem un/vai pieejām tiek izvēlēts attiecīgajam projektam, tādēļ turpmākajās apakšnodaļās tiks apskatīti populārākie testu automatizēšanas ietvari un pieejas.

1.5.1. Testu automatizēšanas ietvari

Testu automatizēšanas ietvars ir pieņēmumu, koncepciju un rīku kopums, kas sniedz atbalstu automatizētai programmatūras testēšanai [Automatic Testing Frameworks, 2011a].

Šādu ietvaru mērķis ir padarīt testēšanas procesu lētāku un efektīvāku. Vienlaikus testu automatizēšanas ietvara uzdevums ir [Wikipedia, 2012b]:

- nodefinēt formātu, ar kuru iespējams veikt automatizētu testēšanu;
- izveidot mehānismu, kā iekļūt vai vadīt testējamo sistēmu;
- izpildīt testus;
- veidot testu rezultātu pārskatus.

Mūsdienās testu automatizēšanas ietvari ir plaši izplatīti, tādēļ jebkurā daudzmaz nopietnā programmatūras izstrādes projektā tiek aplūkota iespēja izmantot vismaz vienu testu automatizēšanas ietvaru. Tomēr, lai arī cik tālu šie ietvari būtu attīstījušies, neviens nepiedāvā risinājumu, kas derētu jebkuram projektam, jebkuram testēšanas līmenim. Tādēļ svarīgi izvēlēties

pareizu testēšanas ietvaru katra projekta vajadzībām. Tas ne tikai paaugstinās testēšanas efektivitāti un kvalitāti, bet arī samazinās izmaksas.

Turpmākajās apakšnodaļās apskatīti populārākie testu automatizēšanas ietvari.

1.5.1.1. Lineārais ietvars

Lineārais ietvars ir vienkāršākais no testu automatizēšanas ietvariem. Tas veido viendimensionālus automatizētus testus. Katrs tests sastāv no atsevišķiem skriptiem un katra komanda precīzi atbilst testa ierakstīšanas brīdī veiktajām darbībām. Lineārie skripti nav objektorientēti, t.i., nav modularitātes un atkalizmantojamības, kā arī netiek izsaukti ārēji skripti. Kā redzams 1.1. attēlā lineārais skripts, satur komandas, testējamos objektus un statiskus datus, kuri kopā imitē lietotāja darbības testējamajā sistēmā.

```
Input „John” into Username textbox
Input „White” into Password textbox
Click Login button
If „Welcome Screen” exists then
       Pass the test
Else
       Fail the test
End If
```

1.1. att. **Lineāra testa skripta fragments** [ATI, 2012b]

Lineāro ietvaru parasti izmanto vienkāršākie testēšanas rīki, kuri testu automatizēšanu nodrošina ar ierakstīšanas un atspēlēšanas funkcionalitāti. 1.1. tabulā uzskaitīti šī ietvara izmantošanas plusi un mīnusi.

1.1. tabula

Lineārā ietvara plusi un mīnusi [ATI, 2012b]

Plusi
Ātra testu izstrāde, jo nav nepieciešama detalizēta plānošana.
Īsāks rīka apguves laiks, jo ietvara vienkāršības dēļ nav jāvelta laiks rīka piedāvāto iespēju apguvei un testa skripts atbilst precīzi ierakstītajām darbībām.
Vienkopus apvieno informāciju par testu, jo testa skripts visu ar testu saistīto informāciju iekļauj vienā failā.
Neatkarīgi testa skripti, jo izmainot vienu testa skriptu, tas nekādā veidā neietekmē citus.
Vieglāka kļūdu atrašana, jo tests atrodas vienā testa skriptā.
Mīnusi
Ierakstītos testa skriptus bieži vien nevar atspēlēt, ja notikušas nelielas izmaiņas testējamajā sistēmā.
Nav atkalizmantojamības, jo vienādie skripti tiek dublēti.
Viendimensionāli skripti, jo tos var izpildīt tikai vienai testējamajai sistēmai un vienā veidā. Ja nepieciešams izpildīt tikai daļu no testa vai citā secībā, tad nepieciešama laikietilpīga testa skripta analīze.
Grūti lasāmi skripti, jo nav atkalizmantojamu objektu un notiek skriptu dublēšana.

Skriptu pārvaldīšanai nepieciešamas padziļinātākas zināšanas, jo nav modularitātes, kas palīdz noteikt, ko dara noteikts skripta fragments.

1.5.1.2. Datu vadītas testēšanas ietvars

Datu vadītas testēšanas ietvars nodrošina automatizētu datu vadītu testēšanu. Datu vadīta testēšana visus testā iekļautos ievaddatus un verificējamos izvaddatus parametrizē tā, lai testa skriptā neparādītos statiski dati. Tā vietā katram testa parametram tiek nodefinēta datu kopa, kas nodrošina testu ar dažādiem ievaddatiem un izvaddatiem. Šāds ietvars nodrošina iespēju vienu testu izpildīt ar dažādiem ievaddatiem un verificējamajiem izvaddatiem. Tomēr šis ietvars joprojām satur viendimensionālus skriptus.

1.2. tabula

Datu vadītas testēšanas datu tabula

ParamUsername	**ParamPassword**
John	White
Jim	Black
George	Red

Vienkāršākais piemērs datu vadītas testēšanas ietvaram ir aplūkojams testa datu tabulā (skat. 1.2. tabulā), kurā katra rinda apzīmē vienu testa datu kopu, bet kolonnas – testa parametrus. Parasti testa datu tabula tiek glabāta datubāzē, izklājlapā, XML failā vai citos formātos. Papildinot iepriekš aplūkoto lineārā ietvara testa skripta fragmentu (skat. 1.1. attēlā), iespējams iegūt datu vadītu testu (skat. 1.2. attēlā), kas parametru vērtības iegūst no 1.2. tabulas rindām.

```
Input <ParamUsername> into Username textbox
Input <ParamPassword> into Password textbox
Click Login button
If „Welcome Screen” exists then
       Pass the test
Else
       Fail the test
End If
```

1.2. att. **Datu vadīta testa skripta fragments** [ATI, 2012b]

Piemērā labi redzams, ka šī pieeja ir lineārā ietvara papildinājums ar dinamiskiem datiem. Iepazīstoties ar darba nākamajām apakšnodaļām, būs redzams, ka šī ietvara pieeja tiek pielietota arī citos testu automatizēšanas ietvaros kā neatņemama sastāvdaļa. Mūsdienās, veidojot jaunu ietvaru, bieži vien tiek apvienoti kopā zemāka līmeņa testu automatizēšanas ietvari. 1.3. tabulā uzskaitītie plusi un mīnusi attiecas uz lineāru ietvaru, kas papildināts ar datu vadītas testēšanas ietvaru.

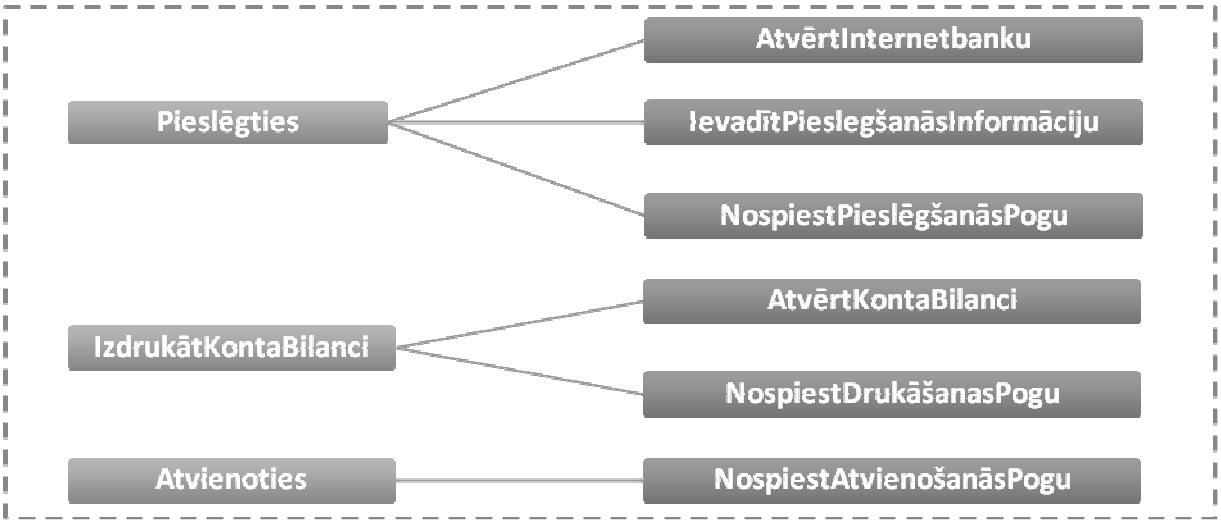
1.3. tabula

Datu vadītas testēšanas ietvara plusi un mīnusi [ATI, 2012b]

Plusi
Testu atkalizmantojamība ar dažādiem datiem.
Viegli realizējams, jo nepieciešamas nelielas izmaiņas esošajos skriptos, piemēram, statiskie dati tiek aizstāti ar parametriem.
Mīnusi
Mīnusi sakrīt ar lineārā ietvara mīnusiem (skat. 1.1. tabulā)

1.5.1.3. Funkcionālās dekompozīcijas ietvars

Funkcionālās dekompozīcijas ietvars testus sadala hierarhiskas funkcijās, kuras veic noteiktas darbības testējamajā sistēmā. Katrs tests tiek veidots tikai no iepriekš izstrādātām funkcijām. Savukārt funkcijas tiek veidotas no citām funkcijām vai arī testa skriptiem. Ja nav pieejama testa izveidei nepieciešamā funkcija, tad tā ir jāizstrādā no jauna. Tādējādi veidojas hierarhija, kas abstrahē darbības testējamajā sistēmā līdz tādam līmenim, ka testētājam nav nepieciešamas zināšanas par testējamās sistēmas uzbūves īpatnībām. [QTP, 2012]



1.3. att. **Internetbankas testa piemērs ar divu līmeņu funkciju dekompozīciju**

Testa funkcijas var iedalīt šādās kategorijās [ATI, 2012b]:

- **utilītfunkcijas** – veic ietvara vai testējamās sistēmas darbības, piemēram, žurnalēšanas funkcija;
- **navigācijas funkcijas** – veic navigāciju uz noteiktu testējamās sistēmas pozīciju;
- **kļūdu apstrādes funkcijas** – veic negaidītu notikumu apstrādi testēšanas laikā;
- **citas funkcijas**, kuras neietilpst augstāk minētajās kategorijās.

Šī ietvara izmantošanas plusi un mīnusi uzskaitīti 1.4. tabulā. Tas ļauj precīzāk novērtēt šī rīka pielietošanas iespējas.

1.4. tabula

Funkcionālās dekompozīcijas ietvara plusi un mīnusi [ATI, 2012b]

Plusi
Augsta atkalizmantojamība, jo jebkuru skripta fragmentu, kas tiek izmantots vairāk kā vienu reizi var pārveidot par funkciju, kuru var izsaukt gan viena testa, gan visa testēšanas projekta ietvaros.
Neatkarīgi testa skripti, jo, lai arī tiek izmantotas ārējas komponentes, pats tests netiek citur izmantots.
Testu vadītas izstrādes atbalsts, jo, ja testējamā sistēma vēl nav izstrādāta, bet ir sistēmas prasību specifikācijas, tad reālu funkciju vietā var izveidot pagaidu tukšas funkcijas un veidot testa scenārijus.
Lasāmi skripti, jo sadalīti mazās daļās.
Vieglāka kļūdu pārvaldība, jo iespējams izveidot funkciju, kas veic kļūdu pārvaldību.
Mīnusi
Sarežģītākiem testiem nepieciešamas padziļinātākas zināšanas, lai nodrošinātu funkciju pārvaldību.
Nepieciešama precīza funkciju dokumentācija, jo, pieaugot funkciju skaitam, arvien grūtāk nodrošināt funkciju nedublēšanos un hierarhiju.
Sarežģītāka uzturēšana, jo, mainot vienu funkciju, mainās vairāki testi.

1.5.1.4. Atslēgvārdu vadītas testēšanas ietvars

Atslēgvārdu vadītas testēšanas ietvars testu veidošanu sadala divos posmos – plānošana un realizācija [Wikipedia, 2012c].

Plānošana notiek, izmantojot atslēgvārdu testa tabulas (skat. 1.5. tabulā), kurās, līdzīgi kā vārdnīcās, ir atslēgvārdi. Atslēgvārds šajā gadījumā apzīmē vispārinātu darbību sistēmā, piemēram, nospiest pogu, aizpildīt ievadlauku utt. Tabulā šie atslēgvārdi sakārtoti secīgi, atbilstoši veicamajām darbībām testējamajā sistēmā. Papildus atslēgvārdiem katra tabulas rinda satur objektu, ar kuru veikt attiecīgo darbību, un datus, kuri nepieciešami darbības veikšanai. Vēl šajā atslēgvārdu tabulā var iekļaut dažāda veida papildinformāciju, kas nepieciešama testēšanas rīkam, lai veiksmīgi izpildītu testu.

1.5. tabula

Atslēgvārdu testa tabula

Darbība	**Objekts**	**Dati**
Ievadīt	Teksta ievadlauks „Lietotājvārds”	Jānis
Ievadīt	Teksta ievadlauks „Parole”	Bērziņš
Nospiest	Poga „Pieslēgties”	Kreisais peles klikšķis

Realizācija ir atkarīga no testēšanas rīka vai ietvara. Parasti testu izpilde notiek, izmantojot draiverus, kuri nolasa atslēgvārdu tabulas ierakstus un izpilda atslēgvārdam atbilstošu kodu. Dažreiz testu izpilde notiek pa tiešo, neizmantojot draiverus. Šādā gadījumā atslēgvārds tiek realizēts testējamajā sistēmā un nav nepieciešams papildus izstrādes process.

Plānošanas posmu parasti veic testētājs, jo atslēgvārdu tabulu veidošanai nav nepieciešamas nedz programmēšanas, nedz arī testējamās sistēmas realizācijas zināšanas, pietiek pārzināt testējamās sistēmas prasību specifikāciju. Savukārt realizācijas posmu veic sistēmas izstrādātājs, kurš katram atslēgvārdam izstrādā kodu, kas uz testējamās sistēmas izpilda atslēgvārdam atbilstošo darbību. Tādējādi tiek atdalīta testu izstrādāšana no realizācijas.

Atslēgvārdu vadītas testēšanas ietvars balstās uz datu vadītas testēšanas un funkcionālās dekompozīcijas ietvaru pamatprincipiem un dod iespēju parametrizēt ievaddatus un verificējamos izvaddatus, kā arī ar atslēgvārdu palīdzību izsaukt funkcijas jeb draiverus, kas izpilda noteiktas darbības testējamajā sistēmā.

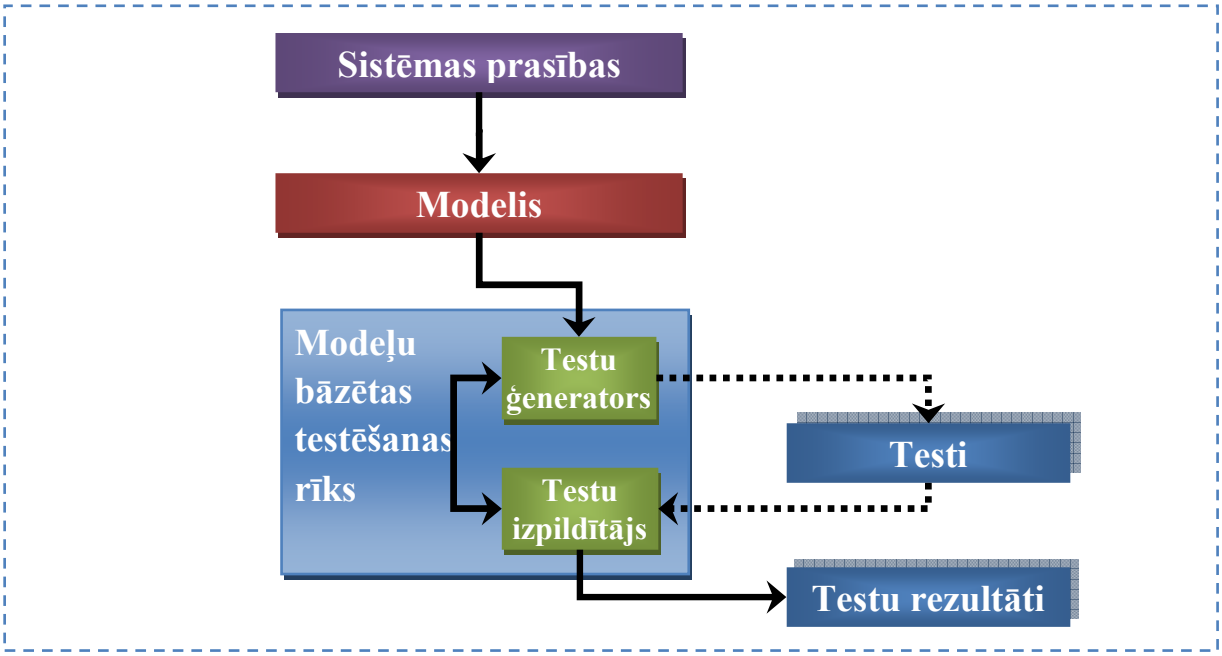
1.6. tabula

Atslēgvārdu vadītas testēšanas ietvara plusi un mīnusi [ATI, 2012b]

Plusi
Augsta atkalizmantojamība, izveidotās funkcijas iespējams izmantot ne tikai viena testa vai testēšanas projekta ietvaros, bet arī dažādās testējamajās sistēmās.
Testu vadītas izstrādes atbalsts, jo, izmantojot sistēmas prasību specifikāciju un atslēgvārdu tabulu, pirms sistēmas izstrādes var veidot testus.
Viegli lasāmi skripti, pateicoties atslēgvārdu tabulām.
Augsta skriptu standartizācija, jo atslēgvārdu tabulas palīdz standartizēt izsaucamās funkcijas.
Vieglāka kļūdu pārvaldības ieviešana, jo var izmantojot atkalizmantojamās šī ietvara komponentes, piemēram, kļūdu pārvaldības funkcionalitāti.
Atslēgvārdu testa tabulas ir vieglāk saprotamas cilvēkiem bez zināšanām IT jomā.
Efektivitāte ilgtermiņā, jo ieviešana ir sarežģīta, bet testu izveide ir vienkāršāka nekā iepriekšējiem trim ietvariem.
Mīnusi
Sarežģītāka uzturēšana, jo, mainot vienu atslēgvārda realizāciju, mainās vairāki testi.
Nepieciešama precīza atslēgvārdu dokumentācija, jo, pieaugot to skaitam, var rasties grūtības ar to pārvaldību.
Sarežģīta testu veidošanas funkcionalitātes realizācija

1.5.1.5. Modeļu bāzēts ietvars

Modeļu bāzēts ietvars saņem informāciju par testējamo sistēmu no tās modeļa un, balstoties uz šo informāciju, dinamiski izveido testus. Šāds ietvars nodrošina pilnīgu platformas neatkarību, jo testētājs ar modeļu (parasti ar stāvokļu pārejas diagrammu) palīdzību apraksta darbības, kuras iespējams veikt ar testējamo sistēmu, un skaidri paredzamās reakcijas. Pēc tam testu automatizēšanas ietvars apkopo modeļos ietverto informāciju un uzģenerē testējamās sistēmas testus.



1.4. att. **Modeļu bāzēts ietvars**

1.4. attēlā ilustrētajā modeļu bāzēta ietvara uzbūvē redzams, ka testu ģenerēšanas modulis no modeļa ģenerē testus, bet testu izpildīšanas modulis šos testus izpilda un izvada testu rezultātus. Šāda veida pieeju sauc par **bezsaistes modeļu bāzētu ietvaru**. Savukārt, ja testu ģenerators neveido ārējus testus, bet gan pārveidoto modeļu informāciju pa tiešo padod testu izpildītājam, tad šādu pieeju sauc par **tiešsaistes modeļu bāzētu ietvaru**. Tiešsaistes pieejā ir viegli vadīt testējamās sistēmas nenoteiktību (piemēram, neparedzētas kļūdas, negaidīti izvaddati), jo testu ģenerators redz testējamās sistēmas izvaddatus un var attiecīgi pielāgot testu ģenerēšanu. Bezsaistes pieeja dod iespēju uzģenerētos testus izpildīt vairākkārtīgi dažādās vidēs. [Utting, 2008]

1.7. tabula

Modeļu bāzēta ietvara plusi un mīnusi [ATI, 2012b]

Plusi
Minimāls skriptu daudzums.
Iespēja testēt dažādas sistēmas, jo modeļi nav piesaistīti vienai noteiktai sistēmai vai videi.
Testēšanā tiek iesaistīts mākslīgais intelekts, kas izpēta sistēmu līdzīgi kā to dara lietotājs.
Lielāka iespēja atklāt kļūdas, jo katrs uzģenerētais tests ir savādāks.
Mīnusi
Nepieciešamas augstas zināšanas par testējamo sistēmu.
Nepieciešamas tehniskas zināšanas par testēšanas rīka funkcionalitāti.
Grūtāk novērtējams testu automatizēšanas ieguvums, jo katrs tests ir savādāks.

1.5.1.6. Secinājumi

Bez šiem augstāk minētājiem testu automatizēšanas ietvariem, mūsdienas pastāv dažādas šo ietvaru modifikācijas, kuras iegūtas papildinot vai apvienojot kopā citus ietvarus. Bieži vien šādas

ietvaru modifikācijas izstrādā automatizēto testēšanas rīku izstrādātāji, kuri, pielāgojot sava rīka koncepcijai kādu no esošajiem ietvariem, cenšas paaugstināt testēšanas efektivitāti un kvalitāti.

Lai arī katrs jaunizveidotais ietvars papildina ar jaunām iespējām iepriekšējos, ne vienmēr tas sniedz efektīvu un kvalitatīvu testēšanu, jo, izvēloties noteiktas testēšanas projektam testu automatizēšanas ietvaru, nepieciešams izvērtēt ne tikai ieguvumus, bet arī ieguldītos resursus.

1.5.2. Testu automatizēšanas pieejas

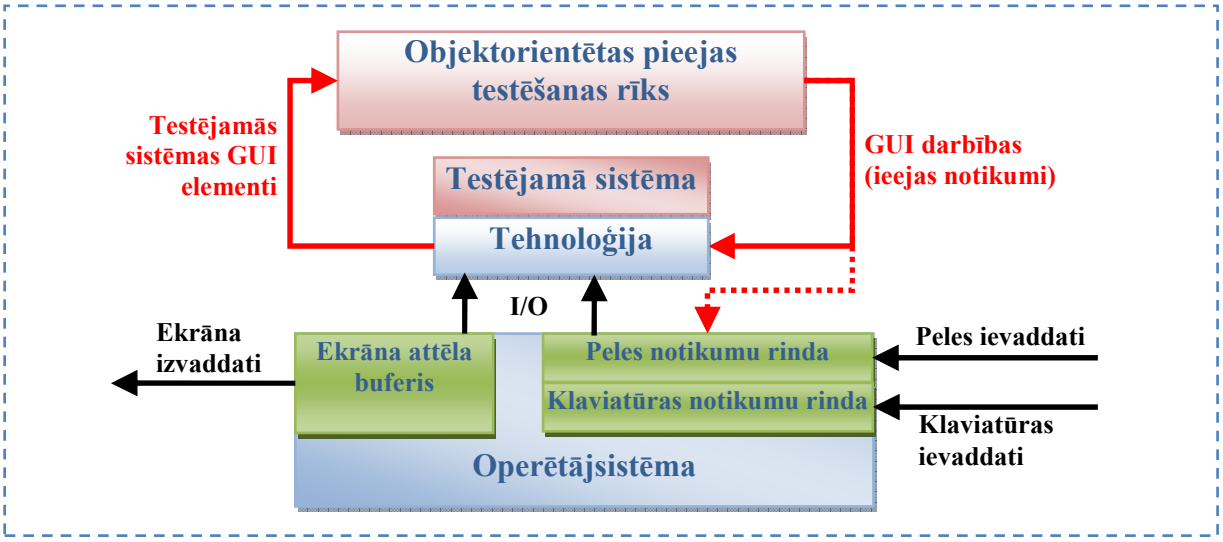
Testu automatizēšanas pieeja nosaka, kādā veidā testēšanas rīks piekļūst testējamajai sistēmai, lai spētu veikt automatizētu testu izpildi. Mūsdienās parasti tiek izmantotas divas dažādas testu automatizēšanas pieejas – objektorientēta un attēlu balstīta [T-Plan, 2012a].

1.5.2.1. Objektorientēta pieeja

Objektorientētas pieejas rīki parasti ir piesaistīti vienai vai vairākām tehnoloģijām, kuras tie spēj atpazīt, proti, identificēt noteiktus grafiskās lietotāja saskarnes objektus, nolasīt to atribūtus un sadarboties ar tiem.

Lai izsauktu dažādas darbības testējamajā sistēmā, visbiežāk objektorientētas pieejas rīki izmanto testējamās sistēmas lietojumprogrammas saskarni (API), caur kuru iespējams piekļūt testējamās sistēmas iekšējai struktūrai. Piemēram, programmēšanas valodā C#.NET izstrādātai sistēmai, lai imitētu pogas nospiešanu, jāizsauc attiecīgās pogas metode „PerformClick()”. Savukārt daži rīki darbojas operētājsistēmas līmenī, izsaucot klaviatūras vai peles notikumus.

Testa rezultātu pārbaude notiek, pārbaudot noteikta grafiskās lietotāja saskarnes objekta pašreizējo stāvokli, piemēram, pārbaudot, vai eksistē teksta lauks ar nosaukumu „Parole” un tekstu „qwerty123”. Tāpat tiek salīdzināti arī dažādi testējamās sistēmas iekšējās struktūras elementi, piemēram, funkciju atgrieztie rezultāti, mainīgo vērtības utt.



1.5. att. **Objektorientēta pieeja** [T-Plan, 2012a]

Šādas pieejas izmantošanai ir šādi plusi:

- iespējams izmantot tikai konkrētai testējamajai tehnoloģijai raksturīgas iespējas;
- stabili testa skripti, jo iespējams piekļūt noteiktiem grafiskās lietotāja saskarnes elementiem, to atribūtiem un izsaukt darbības;
- vienkārša un efektīva testa rezultātu pārbaude;
- saprotams attiecīgās testējamās sistēmas tehnoloģijas speciālistiem.

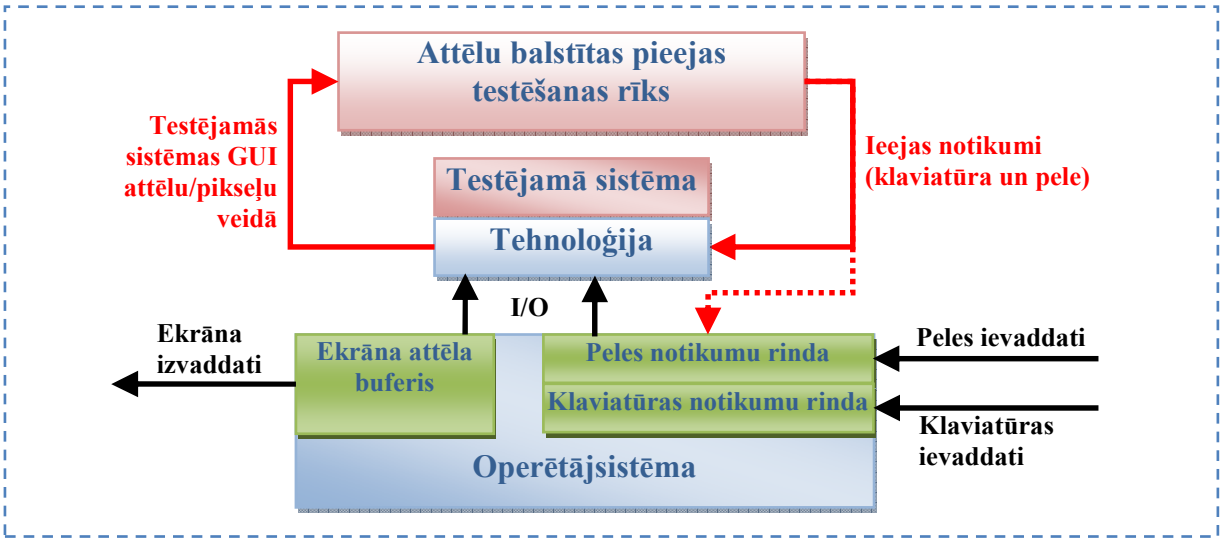
Tāpat šādai pieejai piemīt arī mīnusi:

- iespējams testēt tikai tās lietojumprogrammas, kas realizētas attiecīgajā tehnoloģijā;
- rodas problēmas, ja testējamā sistēma realizēta vairākās atšķirīgās tehnoloģijās;
- neatklāj kļūdas, kas saistītas ar grafiskās lietotāja saskarnes attēlošu;
- tehnoloģijas atjauninājumu gadījumā, parasti nepieciešams veikt izmaiņas arī testēšanas rīkā;
- testu izstrādes procesā nepieciešamas programmēšanas zināšanas.

1.5.2.2. Attēlu balstīta pieeja

Attēlu balstītas pieejas rīki parasti testu automatizēšanu realizē operētājsistēmas līmenī, ļaujot piekļūt visām nepieciešamajām iekārtām, ieskaitot virtualizācijas un attālinātās pieejas lietojumprogrammas. Līdzīgi kā objektorientētās pieejas metode, kas darbojas operētājsistēmas līmenī, arī attēlu balstītā pieeja ievaddatu sūtīšanai uz testējamo sistēmu izmanto klaviatūras un peles notikumus, bet tā vietā, lai izsaukumus veiktu ar komandrindu izsaukumiem, attēlu balstītas pieejas rīks šos notikumus iekļauj pa tiešo operētājsistēmas notikumu rindā.

Grafiskās lietotāja saskarnes elementiem attēlu balstītas pieejas rīki piekļūst, izmantojot operētājsistēmas ekrāna buferi pikseļu līmenī. Pēc katriem ievaddatiem, ko rīks aizsūta uz testējamo sistēmu, tiek iegūts ekrāna attēls. Testa pārbaude notiek, izmantojot iegūto attēlu salīdzināšanas metodes. Biežāk izmantotās ir attēla pārmeklēšanas un objektu atpazīšanas metodes.



1.6. att. **Attēlu balstīta pieeja** [T-Plan, 2012a]

Izmantojot testēšanas rīku, kurš izmanto attēlu balstītu pieeju tiek iegūts:

- testēšana, kas pilnībā notiek tā, kā to dara sistēmas lietotājs;
- platformas un tehnoloģiju neatkarīgs – pilnīga testējamās sistēmas tehnoloģiju neatkarība, jo testēšana notiek operētājsistēmas līmenī;
- nav nepieciešams piekļūt testējamās sistēmas iekšējai struktūrai;
- viegli apgūstama pieeja.

Protams, pastāv arī dažas nepilnības:

- nav pieejas testējamās sistēmas grafiskās lietotāja saskarnes elementiem un to atribūtiem;
- ierobežotas verificēšanas iespējas;
- jūtīgums pret testa vides izmaiņām, piemēram, ekrāna fona attēla nomaiņa.
- testa vides maiņas gadījumā, nākas veikt atkārtotu testējamās sistēmas ekrāna attēlu iegūšanu.
- nevar notestēt funkcionalitāti, kas netiek attēlota uz ekrāna.

1.5.2.3. Secinājumi

Tāpat kā testu automatizēšanas ietvaru gadījumā, arī testu automatizēšanas pieejas gadījumā nepieciešama rūpīga gan testējamās sistēmas, gan testa vides analīze, lai izvēlētos pareizo pieeju

testu automatizēšanai. Pirms izvēles izdarīšanas, speciālisti iesaka izvērtēt šādus kritērijus [T-Plan, 2012a]:

- testēšanas „dziļums" – vai ir nepieciešams piekļūt grafiskās lietotāja saskarnes elementiem, vai – nav;
- tehnoloģiju loks – vai testējamā sistēma sastāv no vienas, vai vairākām tehnoloģijām un vai nav nākotnē paredzēts ieviest jaunu tehnoloģiju;
- migrācijas un atjaunošanas plāni – vai testa vide būs stabila, vai nebūs jāveic migrācija uz citu vidi vai platformu;
- testu izstrādātāju zināšanas – vai testu izstrādātājiem ir programmēšanas zināšanas, vai būs pietiekoši laika, lai viņi apgūtu tehnoloģijai specifiskas iespējas.

2. PAŠTESTĒŠANAS PIEEJA

Tajās mūsdienu tehnoloģijās, kuru kļūdu dēļ iespējamas nopietnas sekas, ļoti plaši tiek izmantota iebūvētā paštestēšanas pieeja (*Built-in Self Test*). Iebūvētā paštestēšana ir mikroshēmas funkcija, kas var pārbaudīt visu vai daļu no iekšējās funkcionalitātes [Webopedia, 2011]. Šādā veidā pati tehnoloģija spēj pati sevi notestēt un savlaicīgi noteikt kļūdas, un paziņot par tām pirms radušās nopietnas finansiālas vai katastrofālas sekas. Šīs paštestējošās mikroshēmas izmanto ieročos, aviācijā, medicīnā un citās sarežģītās tehnoloģijās.

Šāda veida testēšanas pieeju ļoti efektīvi iespējams pielietot ne tikai mikroshēmām, bet arī programmatūrai. 2007. gadā paštestēšanas pieeja tika definēta kā viens no viedtehnoloģiju pamata principiem un tika šādi definēta: „Par informāciju sistēmas paštestēšanu tiek saukta tās spēja automātiski izpildīt iepriekš sagatavotus testus savas darba spējas pārbaudīšanai" [Bičevska, Bičevskis, 2007].

Kā redzams, gan mikroshēmā iebūvētās paštestēšanas, gan programmatūras paštestēšanas pieejas definīcijas daudz neatšķiras, jo būtībā tās abas no iekšienes veic funkcionalitātes testēšanu ar iepriekš sagatavotiem testiem. Virspusēji skatoties, atšķirība ir tikai tā, ka viena iebūvēta mikroshēmā, bet otra – programmatūrā. Tomēr programmatūras paštestēšanas pieejā ir vairākas programmatūras testēšanai specifiskas īpašības, tādēļ tālāk šajā nodaļā tiks aprakstīta paštestēšanas koncepcija un uzbūve.

Darbā ir aplūkota jauna testēšanas tehnoloģija, kas, līdzīgi kā tradicionālie testēšanas rīki, nodrošina testu veidošanu un testu automātisku izpildi, dažādu testēšanas līmeņu testēšanu, testēšanu pēc dažādām testēšanas metodēm, funkcionālo un nefunkcionālo testēšanu. Galvenā atšķirība – testēšanas mehānisms daļēji tiek iebūvēts pašā testējamā sistēmā, tādējādi nodrošinot plašākas testēšanas un citas jaunas iespējas, ko tradicionālie testēšanas atbalsta rīki nenodrošina.

2.1. Koncepcija

Kā aprakstīts [Bičevska, Bičevskis, 2008b] [Bičevska, Bičevskis, 2008c] paštestēšanas pieeja ietver divus komponentus:

- Sistēmas kritiskās funkcionalitātes testa piemēri, kas pārbauda to darbību kopumu, bez kura programmatūra nav lietojama.
- Programmā iebūvēts automātiskās testēšanas mehānisms (regresā testēšana), kas nodrošina automātisku testu izpildes un rezultātu salīdzināšanu ar etalona vērtībām iespējas.

Kritiskā funkcionalitātes definēšana un tās pārbaudes testu sagatavošana, kā likums, ir prasību analīzes un testēšanas procesa sastāvdaļa. Paštestēšanas pieejas realizācija programmatūrā prasa, vismaz daļēji, tradicionālo testēšanas atbalsta rīku iespējas iekļaut veidojamā sistēmā. Paštestēšanas funkcionalitātes realizācijas rezultāts ir sistēmas papildināšana ar paštestēšanas funkciju izsaukumiem un paštestēšanas funkciju bibliotēka (.dll fails). Līdz ar sistēmas izstrādi programmētājs sagatavo arī kritiskās funkcionalitātes testu piemērus. Paštestēšanas pieeja ir realizēta ar paštestēšanas rīku, kas, līdzīgi kā tradicionālie testēšanas atbalsta rīki, nenosaka konkrētu stratēģiju testu veidošanā un testu izpildē, tādā veidā to atstājot programmētāju rokās. Jāatzīmē, ka ideja par iebūvētu atbalstu programmu testēšanai piedāvāta jau visai sen [Bichevskii, Borzov, 1982] [Chengying *et. al.*, 2007] un tā realizēta atsevišķos praktiskos projektos. Neapšaubāmi, paštestēšanas pieejas iespējas prasa papildus darbu sistēmas izstrādē. Tomēr šīs pieejas priekšrocības kvalitatīvas sistēmas izstrādē un, it īpaši, ilgstošā sistēmas uzturēšanā attaisno šos papildus izdevumus.

Galvenā paštestēšanas īpatnība ir spēja programmatūru testēt jebkurā laikā gan izstrādes, gan testa, gan produkcijas vidē. Līdz ar to, veidojot paštestēšanas mehānismu, izstrādātājiem jāgarantē, ka informācija produkcijas datu bāzēs rakstīta netiek, bet tās var izmantot lasīšanas režīmā. Tādejādi sistēmas testēšanu var veikt gan testa, gan produkcijas vidē, netraucējot tās lietošanu, un katrā testēšanas reizē ir iespējams veikt uzkrātajiem testiem atbilstoša regresa testēšana. Protams, pēc katras sistēmas izmaiņas ir lietderīgi papildināt testu kopu, tādejādi nodrošinot paštestēšanas ietvaros pārbaudāmās kritiskās funkcionalitātes pārklājumu.

Ja pasūtītājs neuzticas testu veikšanai produkcijas vidē (ar šo problēmu noteikti varētu saskarties banku sistēmu gadījumā, kas uztur augstas drošības prasības), tad paštestēšanu iespējams veikt testa vidē. Tādejādi ar laiku iespējams radīt pārliecību pasūtītājā, ka korekta paštestēšanas funkcionalitātes darbība ir iespējama arī produkcijas vidē.

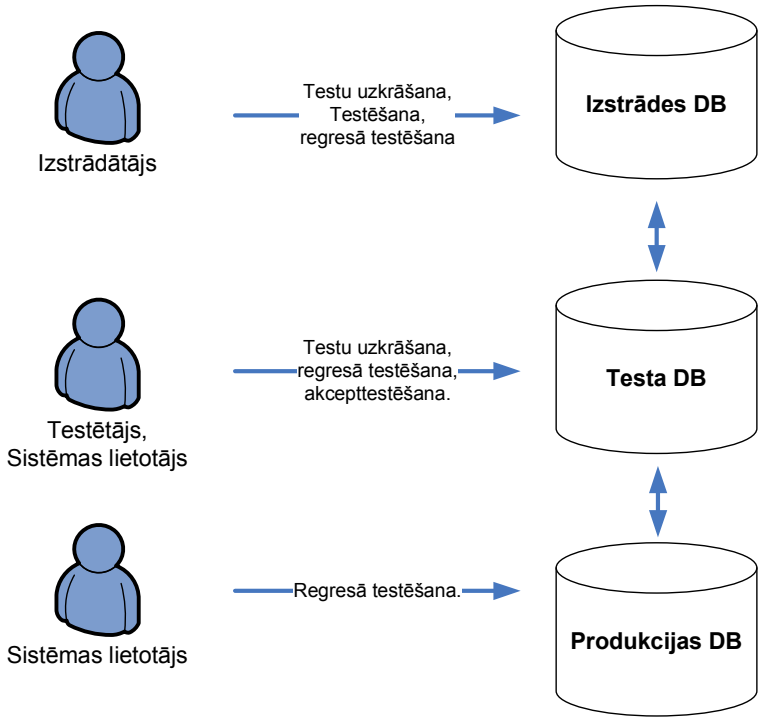
2.2. Testēšanas etapi

Lai nodrošinātu kvalitatīvu izstrādes rezultātu, tad sistēmu parasti testē trīs etapos (skat. 2.1. attēlā) dažādās vidēs:

- Izstrādes vide. Izstrādes vidē tiek veikta sistēmas izstrāde, labotas sistēmas kļūdas un izstrādāti sistēmas uzlabojumi.
- Testa vide. Testa vide tiek izmantota izlaboto kļūdu un jauno uzlabojumu akcepttestēšanai. Lai nodrošinātu pēc iespējas pietuvinātu vidi sistēmas lietošanas

(produkcijas) videi, piemēram, reizi mēnesī testā vidē tiek uzstādīts produkcijas vides dublējums.

- Produkcijas vide. Vide, kurā strādā sistēmas lietotāji. Produkcijas vidē kļūdas labojumi vai uzlabojumi tiek uzstādīti tikai pēc tam, kad ir veikti akcepttesti Izstrādes un Testa vidēs.



2.1. att. **Sistēmas testēšanas etapi**

Galvenās paštestēšanas pieejas raksturojošās iezīmes:

- kopā ar programmatūru tiek nodoti testpiemēri, ar kuriem ir veikta automatizēta sistēmas paštestēšana;
- testēšanu var atkārtot Produkcijas vidē, nebaidoties sabojāt Produkcijas vides datu bāzi;
- paštestēšanā netiek pārbaudīta lietotāju saskarnes, veiktspējas un citas nefunkcionālas īpašības.

Lai panāktu augstu sistēmas kvalitāti, testēšanu, ieskaitot automatizētu paštestēšanu, ir nepieciešams veikt secīgi visos testēšanas etapos, kas aplūkoti nākošajās nodaļā.

2.2.1. Paštestēšana Izstrādes vidē

Izstrādes vidē tiek veidotas jaunākās datu struktūras un programmatūras izejas tekstu faili. Tomēr izstrādātājiem ne vienmēr Izstrādes vidē ir pieejamas sistēmas visas ārējās saskarnes, piemēram, ja sistēmai jāsadarbojas ar grāmatvedības sistēmu, kuru izstrādājuši citi piegādātāji, tad

Izstrādes vidē šo grāmatvedības sistēmu uzstādīt ir problemātiski licenču un drošības apsvērumu dēļ. Tādejādi nepieciešams veidot ārējo saskarņu simulēšanu (aizbāžņus), kas ne vienmēr spēj pilnvērtīgi nodrošināt ārējās saskarnes aizstāšanu. Tādējādi sadarbības ar ārējām sistēmām testēšana ir viena no grūtāk risināmām problēmām programmu testēšanā vispār. Darbā parādīts, ka Paštestēšanas mehānismi spēj sniegt būtisku atbalstu norādītās problēmas risināšanā.

Izstrādes vidē sistēmas izstrādātājiem papildus sistēmas funkcionalitātes izstrādei ir jānodrošina Paštestēšanas mehānisma iestrādāšanu sistēmā. Testēšanu Izstrādes vidē nodrošina paši sistēmas izstrādātāji:

- sagatavo testa piemērus, kas pārklāj sistēmas kritisko funkcionalitāti;
- nodrošina testu piemēru importu no Testa vai Produkcijas vides;
- izpilda testēšanu, to veicot automatizēti ar paštestēšanas funkcionalitātes atbalstu.

Neapšaubāmi, programmētājs vai izstrādes programmētāju grupa nenodos neatkarīgai testēšanai vai pasūtītājam savu sistēmu, ja, lietojot paštestēšanu, tiek atklātas kļūdas. Tikai pēc veiksmīgas sistēmas testēšanas programmētājs nodod savu sistēmu neatkarīgai testēšanai Testa vidē. Neatkarīgā testēšanas grupa spēj atklāt atšķirīgas specifikāciju izpratnes, ja tādas eksistē, lietotāja saskarnes neērtības un citas nefunkcionālas prasību neatbilstības, kas nav saistītas ar sistēmas kritisko funkcionalitāti. Tomēr, kā rāda prakse, izmantojot paštestēšanas pieeju neatkarīgai testēšanai reti izdodas atklāt defektus. Paštestēšana neatbrīvo sistēmas izstrādātājus no sistēmas testēšanas, bet sniedz jaunas iespējas veikt automatizētu regreso testu izpildi bez ārēju rīku atbalsta.

2.2.2. Paštestēšana Testa vidē

Testa vide ir paredzēta akcepttestēšanai, ko pirms izstrādātāju piegādātās programmatūras uzstādīšanas Produkcijas vidē veic testētāji – problēmas apgabalu pārzinoši sistēmas lietotāji. Testa videi no Produkcijas vides vajadzētu atšķirties tikai ar piegādātām sistēmas izmaiņām. Šajā vidē sistēmas testus iespējams veikt reālākos sistēmas darbības apstākļos kā Izstrādes vidē. Turklāt šajā vidē iespējams veikt precīzāku ārējo saskarņu testēšanu, jo sistēmas lietotājiem Testa vidē ir pieejamas ārējās saskarnes, kas ne vienmēr ir pieejamas Izstrādes vidē.

Sistēmas konfigurācijas pārvaldnieks nodrošina testu piemēru uzstādīšanu no Produkcijas vides, izstrādātāju piegādātās programmatūras, sagatavoto testu un to rezultātu uzstādīšanu Testa vidē. Sistēmas lietotāji Testa vidē:

- akceptē vai noraida no Izstrādes vides saņemtos testus. Izstrādes vidē sagatavotie testi Testa vides paštestēšanā tiek iekļauti tikai tad, kad sistēmas lietotājs Testa vidē tos ir apstiprinājis;

- sagatavo testa piemērus, kas papildina izstrādātāju sagatavotos testa piemērus;
- veic piegādātās sistēmas testēšanu, tādā veidā pārliecinoties, vai programmatūras funkcionalitāte nav sabojāta ar jaunās funkcionalitātes vai izmaiņu ieviešanu, kā arī tiek pārbaudīts vai Testa vidē ir piegādāti visi ar jaunām izstrādēm saistītie nodevumi (datu struktūras, izejas tekstu faili).

2.2.3. Paštestēšana Produkcijas vidē

Produkcijas vidē ir pieejami reālie dati, kā arī ir pieejamas visas ārējās saskarnes produkcijas režīmā. Tādējādi Produkcijas vidē ir iespējams sastapt situācijas, kuras dažādu iemeslu dēļ neizdodas izveidot Testa un Izstrādes vidēs.

Sistēmas konfigurācijas pārvaldnieks nodrošina testu uzstādīšanu Produkcijas vidē, bet sistēmas lietotāji Produkcijas vidē:

- sagatavo jaunus testa piemērus, kas papildina izstrādātāju, Testa vidē sagatavotos testa piemērus ar reāliem piemēriem no Produkcijas vides;
- veic automatizētu testu izpildi (sistēmas paštestēšanu), pie kam reālie dati Produkcijas vidē netiek mainīti. Produkcijas dati sistēmas paštestēšanas režīmā tiek izmantoti tikai lasīšanas (*read-only*) režīmā. Datu saglabāšana notiek Testu izpildes datu bāzē. Testu rezultātā iespējams pārliecināties, vai piegādātā programmatūra nav nekorekti mainījusi esošo sistēmas funkcionalitāti un vai Produkcijas vidē uzstādīti visi nepieciešamie programmatūras nodevumi.

2.3. Darbības režīmi

Neatkarīgi no sistēmas darbības vides (Izstrādes, Testa vai Produkcijas vide) sistēmu, kura satur paštestēšanas funkcionalitāti, ir iespējams lietot vairākos režīmos:

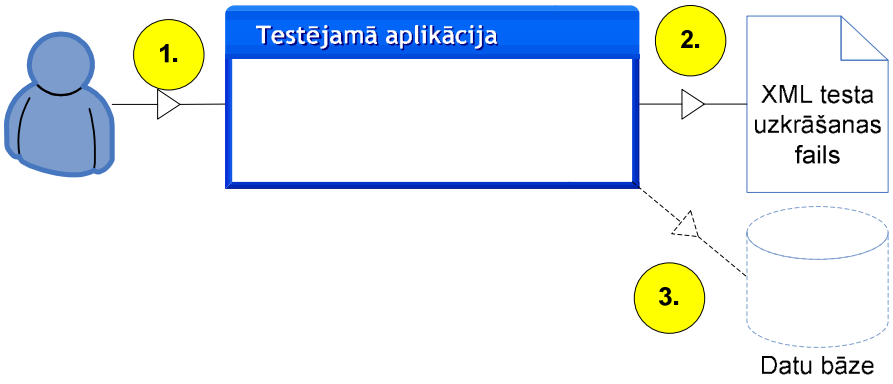
- Testu uzkrāšanas režīms. Šajā režīmā tiek veikta jaunu testu definēšana, esošu testu rediģēšana un dzēšana;
- Paštestēšanas režīms. Šajā režīmā tiek veikta automatizēta programmatūras paštestēšana, automatizēti izpildot uzkrātos testa piemērus;
- Lietošanas režīms. Režīmā netiek veiktas nekādas testējošas darbības. Šajā režīmā sistēmas lietotājs izmanto sistēmas pamata funkcionalitāti;
- Demonstrācijas režīms. Ja reiz datu bāzē ir uzkrāta informācija par korekti veiktajiem testiem, tad šo testu kopu būtu lietderīgi izmantot demonstrācijās, jaunu darbinieku apmācībai, sistēmas demonstrēšanai citiem interesentiem u.c. gadījumos.

Par katra režīma tehnisko realizāciju ir aprakstīts nākamajās nodaļās.

2.3.1. Testu uzkrāšanas režīms

Sistēma testu uzkrāšanas režīmā protokolē visas nepieciešamās informācijas lasīšanas, rakstīšanas un vadības darbības, ierakstot tās Testu piemēru failā. Tāpat arī sistēmas darbības rezultāti tiek saglabāti Testu piemēru failā. Lai nodrošinātu paštestēšanas režīmu produkcijas vidē, tiek izmantota papildus datu bāze, kurā tiek veikta testu reģistrēšana un izpildīšana. Produkcijas vides gadījumā testu uzkrāšanas laikā produkcijas vides datu bāze ir pieejama lasīšanas režīmā.. Izstrādes un testa vidēs papildus datu bāze nav nepieciešama un tiek izmantota viena datu bāze gan testu uzkrāšanai, gan testu atspēlēšanai. Papildus šis režīms var tikt izmantots, lai lietotājs, atklājot sistēmas darbības kļūdu, varētu šo darbību virkni, kas ved pie programmatūras nekorektas darbības, ierakstīt testu failā un nodot to uzturēšanas personālam kopā ar incidenta ziņojumu.

Testus, kā likums, sagatavo programmētājs atbilstoši savai specifikācijas izpratnei. To skaits un saturs līdz ar sistēmas attīstību tiek palielināts.



Testu uzkrāšanas režīms

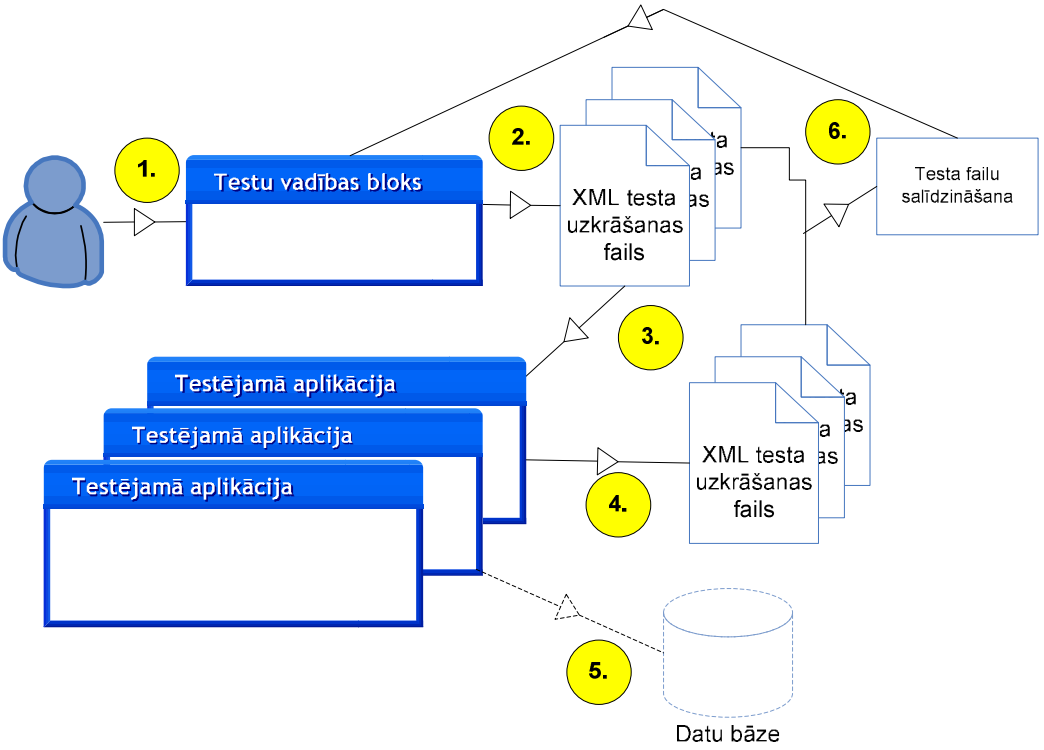
2.2. att. **Testu uzkrāšanas režīms**

1. Lietotājs testu uzkrāšanas režīmā reģistrē testa piemēru. Lietotājs lieto to pašu lietojumprogrammu, kas tie lietota ikdienā biznesa mērķu sasniegšanai.
2. Testa uzkrāšanas režīmā lietojumprogramma testa failā reģistrē sistēmā veiktās darbības.
3. Ja lietojumprogrammā tiek veikta datu saglabāšana, tad dati tiek saglabāti datu bāzē. Produkcijas vides gadījumā saistītie dati tiek ielasīti no produkcijas datu bāzes, savukārt saglabāšana tiek veikta testu izpildes datu bāzē, nevis produkcijas datu bāzē.

2.3.2. Paštestēšanas režīms

Šajā režīmā tiek veikta automātiski sistēmas paštestēšana, automātiski izpildot testa piemērus no uzkārtiem Testa piemēru failiem. Izstrādes un testa vidēs testi tiek izpildīti vienā datu bāzē.

Savukārt produkcijas vidē tiek izmantota testu uzkrāšanas un izpildes datu bāze. Produkcijas vides reālā datu bāzes paštestēšanas režīmā ir pieejama tikai lasīšanas režīmā.



Paštestēšanās režīms

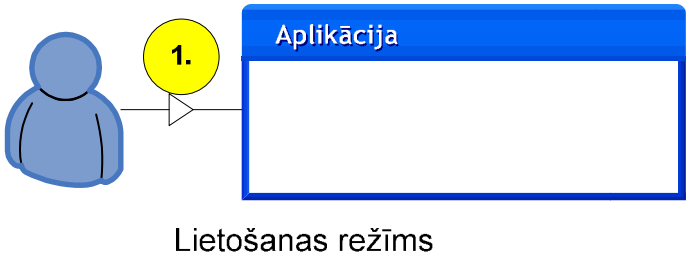
2.3. att. **Paštestēšanas režīms**

1. Lietotājs, lai izsauktu sistēmas paštestēšanu, atver testu vadības bloka logu.
2. Logā lietotājs ielasa testu failus un norāda, kurus no ielasītajiem failiem izpildīt.
3. Testu vadības bloks nolasa informāciju no testa faila, kas tiek izpildīts, un veic atbilstošās darbības, kas norādītas testa failā.
4. Paštestēšanas režīmā, tāpat kā testu uzkrāšanas režīmā tiek veidots testa fails. Fails tiek veidots izmantojot to pašu pieeju, kas tika izmantota testa uzkrāšanas režīmā.
5. Ja testa piemērā tiek veikta datu saglabāšana, tad dati tiek saglabāti datu bāzē. Produkcijas vides gadījumā saistītie dati tiek ielasīti no reālās datu bāzes, savukārt saglabāšana tiek veikta testu reģistrēšanas un izpildes datu bāzē, nevis reālajā datu bāzē.
6. Pēc testu izpildes tiek salīdzināts testa uzkrāšanas režīmā izveidotais fails ar paštestēšanas režīmā izveidoto failu. Ja failu saturs sakrīt, tad testa izpilde ir notikusi veiksmīgi, savukārt, ja nesakrīt, tad tas ir bijis neveiksmīgs. Informācija par testa rezultātu tiek attēlota lietotāja testu vadību blokā, kur neveiksmīga testa izpildes gadījumā, lietotājam ir iespējams detalizētāki iepazīties ar iemesliem, kādēļ tests nav izpildījies.

2.3.3. Lietošanas režīms

Šajā režīmā ir lietderīgi iestrādāt kontroli, ja noticis sistēmas darbības incidents, atbildīgajam personālam tiek nosūtīts atbilstošs ziņojums ar incidentu saistīto informāciju.

Sistēmas lietošanas režīmā netiek veiktas ar sistēmas testēšanu saistītas darbības. Lietošanas režīmu sistēmas lietotāji izmanto sistēmas pamata funkcionalitātes lietošanai.



2.4. att. **Lietošanas režīms**

1. Sistēmas lietotājs lietošanas režīmā strādā ar sistēmu, nenojaušot par Testu uzkrāšanas un Testu izpildes datu bāzēm.

2.3.4. Demonstrācijas režīms.

Režīms, kuru var izmantot sistēmas demonstrācijām. Lietotājs var veikt sistēmas demonstrācijas, izmantojot Testu piemēru failā uzkrātos testus. Demonstrēšanas režīmā automātiski tiek aizpildītas formas lauki ar testa datiem no Testu piemēra failiem. Tādā veidā demonstrējot sistēmas funkcionalitāti. Testu piemēru failā uzkrātie testi vienmēr izpildīsies un demonstrētājs var būt pārliecināts par veiksmīgu demonstrāciju, pretēji, kā nereti gadās, kad demonstrētājam jāsajūtas neveikli kādas negaidītas kļūdas parādīšanas brīdī. Demonstrācijas režīma process sakrīt ar paštestēšanas procesu (skat. 2.3. attēlā). Atšķirība starp procesiem ir tajā, ka paštestēšana var tikt veikta lietotājam neredzamā režīmā, savukārt demonstrācija tiek veikta pa soļiem lietotājam redzamā režīmā. Reizē ar demonstrāciju ir iespējams veikt arī sistēmas paštestēšanu. Šāda pieeja ir daudz lēnāka, bet ir iespējams identificēt kļūdu redzamā režīmā, izpildot konkrēto darbību, kas nolasīta no testa faila.

2.4. Paštestēšana sistēmas dzīves cikla modeļos

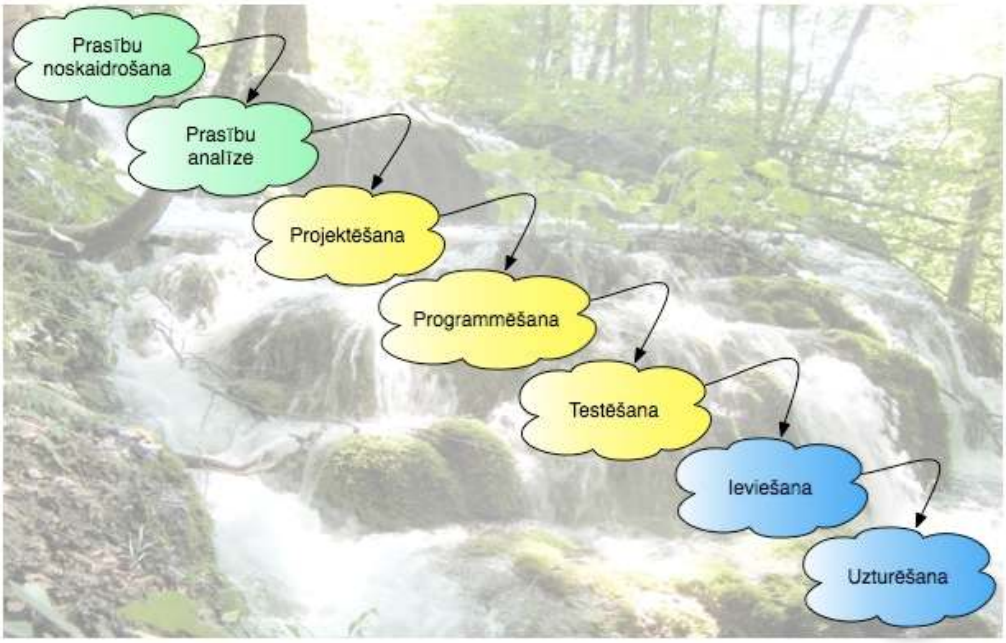
Neatkarīgi no tā, kāds sistēmas dzīves cikla modelis tiek lietots – ūdenskrituma, spirāles, prototipēšanas, v-formas vai arī kāds cits – tajos visos ir atvēlēta loma sistēmas prasību izzināšanai, projektēšanai, izstrādei un, protams, testēšanai. Atkarībā no izvēlētā modeļa atšķirīga var būt minēto procesu secība, realizācijas veids, atkārtošanās skaits vai laika daudzums, kas modelī tiek veltīts katram no procesiem, tomēr tiem visiem mērķis ir veiksmīga informācijas sistēmas izstrāde. Katram

sistēmu izstrādātājam ir zināms, ka nevar izvēlēties vienu tādu sistēmas dzīves cikla modeli, kurš būtu ideāli piemērots jebkura veida problēmu risināšanai. Līdzīga situācija ir ar testēšanas veidiem un to ieviešanu dažādos modeļos – nav iespējams izdalīt atsevišķus dzīves cikla modeļus, kuri obligāti būtu jālieto tajos gadījumos, kad vēlamies programmatūrā ieviest paštestēšanu. Pirms piemērotākā modeļa izvēles jāņem vērā arī fakts, ka paštestēšanas ieviešana neizmaina tos programmatūras izstrādes dzīves cikla izvēles apsvērumus, kuri bija par iemeslu esošajai dzīves cikla izvēlei. Protams, var gadīties situācija, kad konkrētās sistēmas izstrādei vienlīdz piemēroti ir vairāki modeļi, un tādā gadījumā fakts, ka tiek plānots ieviest paštestēšanu, var palīdzēt izdarīt izvēli par labu modelim, kurā paštestēšana būs ieviešama pēc iespējas vieglāk un ātrāk, tomēr tie būs atsevišķi gadījumi, no kā nākas secināt, ka, lai efektīvi izmantotu programmatūras paštestēšanas iespējas, to jāspēj pielāgot jebkuras informāciju sistēmas izstrādes dzīves ciklam.

2.4.1. Ūdenskrituma modelis

Mūsdienās, apskatot konkrētus sistēmas dzīves cikla modeļus, ir ierasts sākt ar vienu no senākajiem modeļiem – ūdenskrituma modeli. Ūdenskrituma modeļa aizsākumi meklējami 1970. gadā, kad Dr. Vinstons Roiss (Winston W. Royse) publicēja savu skatījumu par lielo informācijas sistēmu izveidošanas pārvaldību. Ūdenskrituma idejas pamatā ir divi „vitāli svarīgie kopīgie soļi visiem programmu piegādātājiem, neatkarīgi no tā, cik liela un sarežģīta programma tiek izstrādāta. Pirmais no tiem ir analīzes solis, kuram seko programmēšanas solis“ [Royce, 2012]. Attīstot šo ideju, Vinstons Roiss pakāpeniski nokļuva līdz modelim, kuru mēs mūsdienās pazīstam kā ūdenskrituma jeb lineārās secības, jeb klasiskā dzīves cikla modeli. Klasiskais ūdenskrituma modelis ir parādīts 2.5. attēlā, kur redzams, ka tas sastāv no septiņiem cits citam sekojošiem etapiem:

- sistēmas prasību noskaidrošana un iegūto prasību analīze;
- programmatūras projektēšana, programmēšana un testēšana;
- izstrādātās programmatūras ieviešanas un uzturēšana.



2.5. att. **Ūdenskrituma modelis**

Kā redzams, dotajā modelī ir paredzēta vieta sistēmas testēšanai, tomēr rodas jautājums, vai gadījumā, ja tiek izmantota ne tikai testēšana, bet arī paštestēšana – vai arī paštestēšanai būtu jānotiek tikai ūdenskrituma modeļa testēšanas fāzē? Jāņem vērā apstāklis, kā tiek veikta testēšana ūdenskrituma modeļa izmantošanas gadījumā – dotajai programmatūrai ir izstrādātas prasības, uz kurām tiek bāzēta programmatūras prasību specifikācija un apraksts. Tālāk, lielākoties izmantojot tikai šos dokumentus, tiek izstrādāts pašas programmatūras kods un, kad koda izstrāde ir pabeigta, tiek veikta izstrādātā produkta testēšana.

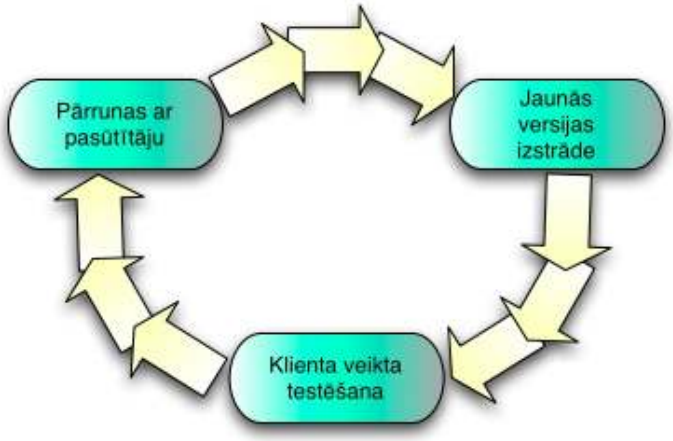
Ja paštestēšana netiek izmantota, tad testēšanas laikā tiek veikta gan atsevišķu sistēmas vienumu jeb moduļu testēšana, gan kopīga visas sistēmas testēšana. Kamēr vien testēšana nav izrādījusies veiksmīga, programmatūra tiek nodota atpakaļ programmētājiem kļūdu novēršanai. Kad beidzot visi testi ir veiksmīgi izieti, notiek programmatūras ieviešana un, atkarībā no līgumsaistībām, arī tās uzturēšana. Ikdienas pieredze liecina, ka lielākajā daļā gadījumu ir nepieciešams veikt izmaiņas izstrādātajā programmatūrā, ko var izraisīt gan izmaiņas algoritmu aprēķinos, gan nepieciešamība saglabāt kādus citus, iepriekš neapzinātus datus, gan arī testētāju nepamanītās kļūdas.

Līdz šim aprakstītais ūdenskrituma modelis ataino situāciju paštestēšanas neizmantošanas gadījumā, tomēr, ja projektā vēlas ieviest paštestēšanu, tad arī ūdenskrituma modeli ir nepieciešams pielāgot paštestēšanai. Jau sistēmas projektēšanas laikā ir jāparedz sistēmas darbību gan testu uzkrāšanas un izpildīšanas, gan, protams, arī sistēmas lietošanas režīmā. Nākošais etaps pēc

sistēmas projektēšanas ir programmēšana – pašsaprotami, ka programmēšanas laikā nepieciešams iestrādāt paštestēšanas funkcionalitāti izstrādājamajā informāciju sistēmā, kas palielina programmēšanai nepieciešamo laiku. Pēdējais solis pirms sistēmas nodošanas tās pasūtītājam ir testēšana, kuru nepieciešams veikt tādā pašā veidā, kā to darītu gadījumā, ja paštestēšanas funkcionalitāte nebūtu iestrādāta sistēmā. Ja paštestēšanas funkcionalitāte ir iestrādāta, tad papildus moduļu un integrācijas testēšanai, ir ļoti rūpīgi jāpārliecinās arī par paštestēšanas funkcionalitātes pareizu darbību. Ja kļūdaina izrādīsies paštestēšanas funkcionalitāte, tad nepareizi strādās arī automātiskā testēšana. Tikai tad, kad sistēmas pamatfunkcionalitāte ir notestēta, ir pienācis brīdis, kad nepieciešams sākt ierakstīt un uzkrāt jaunos testus. Kāds ir ieguvums no paštestēšanas ieviešanas pēc ūdenskrituma modeļa izstrādātā sistēmā, ja testus sāk uzkrāt tikai pēc testēšanas beigām – ir jāatceras, ka „Diemžēl programmatūras izstrādes dzīves cikla modeļi galveno uzmanību pievērš programmatūras izstrādes procesam, tikai atturīgi pieskaroties programmatūras uzturēšanas procesam. Praksē situācija ir citāda. Programmatūras sākotnējā izstrāde ir tikai pirmais nelielais solis programmatūras kopējā dzīves laikā“ [Lubiņa, 2012]. Paštestēšanu ir vērts iestrādāt veidojamajā sistēmā tajos gadījumos, ja pēc sistēmas sākotnējās izveidošanas tā tiks papildināta ar jaunu funkcionalitāti, kas ietekmē līdzšinējo sistēmas funkcionalitātes darbību.

2.4.2. Prototipēšanas modelis

Otrs modelis, kas visbiežāk tiek apskatīts, runājot par informācijas sistēmu izstrādi, ir prototipēšanas jeb iteratīvais modelis. Prototipēšanas modeļa gadījumā programma tiek izstrādāta pakāpeniski, ar katru iterāciju aizvien vairāk tuvojoties programmatūras īstajai versijai. „Mirklī, kad ir pabeigta programmatūras prasību analīze un projektēšana, sākas programmatūras izstrādes process. Kad prototips ir izveidots, tas tiek nodots pasūtītājam tā izvērtēšanai. Pasūtītājs veic izstrādātā nodevuma testus un sniedz savus komentārus izstrādātājam par to, vai dotais produkts atbilst tam, ko pasūtītājs ir vēlējies un sagaidījis saņemt. Pēc galīga iterāciju skaita pasūtītājs saņem galīgo programmatūras versiju“. [Stylusinc, 2009]



2.6. att. **Prototipēšanas modelis**

Vienkāršots prototipēšanas modeļa grafiskais attēlojums ir apskatāms 2.6 attēlā — tajā redzams, kā cikliski cits citam seko konsultēšanās ar pasūtītāju par programmatūras prasībām posms, šo prasību realizācija un realizācijas testēšana. Tieši realizācijas testēšana ir posms, pēc kura pasūtītājs izsaka savu viedokli par sistēmas pašreizējo stāvokli – ja sistēma pasūtītāju apmierina, tad tā tiek ieviesta reālajā ekspluatācijā, pretējā gadījumā seko kārtējā iterācija, veicot atlikušos uzlabojumus un realizējot nepieciešamās izmaiņas.

Apskatot prototipēšanas modeli, līdzīgi kā tas bija ūdenskrituma modeļa gadījumā, rodas jautājums, vai arī prototipēšanas modeļa izmantošanas gadījumā nenākas saskarties ar jau iepriekš aprakstīto situāciju, ka paštestēšana ir izmantojama tikai pēc izstrādes beigām, nevis tās laikā? Vai ir iespējams iekļaut prototipēšanas modelī paštestēšanu tādā veidā, lai tā, atšķirībā no ūdenskrituma modeļa, būtu noderīga ne tikai sistēmas uzturēšanas laikā, bet jau kārtējā prototipa izstrādes laikā? Prototipēšanas modeļa gadījumā atbilde ir jā, tas ir iespējams, it īpaši, lielu sistēmu izstrādes gadījumos. Lai to veiksmīgi izdarītu, testu kopu būtu vēlams veidot pēc kārtējā prototipa nodošanas ar testiem pārklājot to programmatūras daļu, kura jau ir izstrādāta un kuru vairs nav paredzēts mainīt. Tādā gadījumā pēc katra nākamā prototipa izstrādāšanas nebūs nepieciešams veikt manuālu regresa testēšanu, bet to varēs izdarīt automātiski, tādējādi ietaupot laiku. Ņemot vērā reālo projektu izstrādes laika iterāciju skaitu un paštestēšanas ieviešanas izmaksas, ir jāatzīst, ka arī šajā gadījumā, līdzīgi kā pārējos modeļos, paštestēšana īpaši noderīga ir ilgstošas uzturēšanas gadījumā, bet mazāk noderīga mazu sistēmu izstrādes gadījumos.

2.4.3. Agile pieeja

Līdz šim darbā tika apskatīti abi no teorijas viedokļa un, iespējams, arī reālās izplatības populārākie sistēmas dzīves cikla modeļi – ūdenskrituma un prototipēšanas modelis. Šie modeļi ir

piemēroti paštestēšanas ieviešanai un izmantošanai tomēr, ja ir vēlēšanās apskatīt ne tikai populārākos modeļus, kas iesaistīti paštestēšanas attīstībā, tad noteikti nedrīkst aizmirst par Agile jeb spējās izstrādes modeli. Spējā izstrāde kā pieeja sistēmas izstrādei ir samērā jauna – tās pirmsākumi ir meklējami 2001. gadā, kad 17 ar informāciju sistēmu izstrādi saistītu cilvēku grupa satikās kādā ASV kalnu kūrortā un, iepriekš neplānoti, savstarpēju diskusiju rezultātā izveidoja spējās izstrādes modeli. Tā formulējums balstījās uz četriem vaļiem:

- koncentrēšanās uz personām un sadarbību, nevis uz procesiem un sistēmām;
- reāli lietojamas programmatūras, nevis sarežģītu dokumentu, izstrādi;
- sadarbību ar klientu, nevis pārrunām par līguma noteikumiem;
- iespēju reaģēt uz pārmaiņām, nevis automātisku plāna izpildi. [Lubiņa, 2012]

Vienlaikus nedrīkst aizmirst, ka spējās programmatūras izstrādes modelis nav tas, pie kura programmatūras izstrādes industrija būtu apstājusies – Agile ir kalpojusi par pamatu tādu metodoloģiju attīstībai kā Scrum un ekstrēmā programmēšana. Kā zināms, šīs ir metodes, kurām pamatā ir ļoti veiksmīga komunikācija ar pasūtītāja un ļoti ātra programmatūras izstrāde ar strauji mainīgām prasībām. Strādājot šādā režīmā, pēc jaunas funkcionalitātes ieviešanas var neatlikt laika pilnai līdzšinējās funkcionalitātes pārtestēšanai, kas, savukārt, negatīvi ietekmē izstrādājamās sistēmas kvalitāti. Risinājumi dotajai problēmai ir vairāki, tomēr viens no efektīvākajiem kvalitātes uzlabošanas veidiem spējās izstrādes gadījumā ir paštestēšanas ieviešana un izmantošana – tā ļautu mainīt saistītus modeļus, īsā laikā pārliecinoties, ka izmaiņas nav ietekmējušas pārējās sistēmas funkcionalitāti. Rezultātā vēl aizvien būtu iespējams gūt spējās izstrādes ieguvumus, nesamazinot sistēmas kvalitātes prasības, kas, savukārt, ir liela priekšrocība konkrētam sistēmas dzīves cikla modelim.

Abos iepriekš aplūkotajos sistēmas dzīves ciklu modeļos testu piemēru izstrāde un testēšana tiek veikta pēc izstrādes, savukārt Agile modelī testa piemēri tiek izstrādāti pirms sistēmas izstrādes (*Test Driven Development – TDD*) [Wikipedia, 2012h]. Sistēmas izstrādes tiek veikta, lai nodrošinātu veiksmīgu sagatavoto testa piemēru izpildi. Ņemot vērā, ka paštestēšana daļēji tiek iestrādāta izstrādājamā sistēmā un testi tiek veidoti darbinot pašu sistēmu, tad sākotnējo testu sagatavošanai paštestēšanas pieeja nav izmantojama, bet, ņemot vērā sistēmas straujo mainību, ko nosaka Agile, paštestēšanas mehānisma iebūvēšana sistēmā ir pat ļoti vēlama, tādā veidā nodrošinot automātisku nemainītās funkcionalitātes testēšanu, mainītās funkcionalitātes testa piemēru izmaiņas. TDD paredz, ka programmētājiem jāveido automatizēti izpildāmi testa piemēri [Wikipedia, 2012g], kas nav vienkāršs uzdevums, turklāt tradicionālie testēšanas atbalsta rīki nenodrošina pilnu (sākot ar datu ievadi un beidzot ar datu saglabāšanu) sistēmas testēšanu, kā arī

nenodrošina testēšanu pēc baltās kastes metodes. Tas nozīmē, ka programmētājam, lai nodrošinātu sistēmas pilnu (lietotāju saskarnes, biznesa loģikas, datu bāzes u.c.) testēšanu, testu izstrādē ir jāizmanto gan tradicionālie testēšanas atbalsta rīki, gan jāiegulda laiks dažādu savu testa piemēru, kurus ne vienmēr ir iespējams izveidot ar tradicionāliem testēšanas atbalsta rīkiem, izstrādē un to automatizēšanā. Paštestēšanas pieeja nenodrošina testu piemēru veidošanu, kamēr nav izstrādāts kods, bet sistēmas esošā koda testēšanas problēmu tā atrisina.

Gan paštestēšanas pieeja, gan Agile paredz programmatūras piegādi kopā ar testu piemēriem. Tradicionālie testēšanas rīki nodrošina testa piemēru eksportēšanu dažādu skriptu formātos, savukārt paštestēšanas pieeja eksportē pilnu testa scenāriju xml faila formātā. Akcepttestēšanā izpildot tradicionālo testēšanas rīku skriptus, vizuāli bieži nav saprotams, kādas darbības skripts ir veicis (ir tikai pazīme: tests veiksmīgs, tests neveiksmīgs) un ir nepieciešams iepazīties ar testu piemēru dokumentāciju, kurā aprakstīti visi testu piemēri, turpretī paštestēšanas pieeja akcepttestēšanā nodrošina testu piemēru vizuālu izpildi, tādā veidā radot pārliecību par testa veiksmīgu/ neveiksmīgu izpildi. Turklāt ir iespējams pievienot, testētājprāt, iztrūkstošos testa piemērus.

Apskatītie informācijas sistēmu izstrādes modeļi nebūt nav vienīgie, kurus iespējams izmantot paštestēšanas nodrošināšanai. Vienlīdz labi iespējams izmantot arī spirāles vai V-veida modeli, tomēr arī šajos modeļos galvenās idejas paliek nemainīgas – paštestēšanas ieviešana prasa papildu darbu izstrādes laikā.

2.5. Datu bāzes sagatavošana testu atkārtotai izpildei

Testu izpildei ir svarīga datu kopa, uz kuras tests tiks izpildīts. Datu bāze testu uzkrāšanas brīdī ir noteiktā stāvoklī. Nav paredzams, vai testu izpildes brīdī datu bāzes stāvoklis sakritīs ar datu bāzes stāvokli, kāds tas bija testa uzkrāšanas brīdī. Visticamāk, ka stāvokļi nesakritīs, jo uzreiz pēc testa reģistrēšanas datu bāzes stāvoklis ir mainījies un nesakrīt ar stāvokli, kāds tas bija pirms testa reģistrēšanas. Tas nozīmē, ka ir gadījumi, kad testi var neizpildīties datu bāzes stāvokļa izmaiņu rezultātā. Piemēram, tiek reģistrēts tests, kurā klientam no konta tiek izskaitīta noteikta naudas summa. Pēc testa atkārtotas izpildes, kādā brīdī klientam nauda kontā vairs nebūs un testa izpilde būs neveiksmīga, jo no klienta konta nebūs iespējams izskaitīt naudu. Tā kā datu bāze nepārtraukti mainās, tad nodrošināt uzkrātā testa izpildīšanu uz to pašu datu kopu, uz kuru tests tika reģistrēts, ir sarežģīti un laikietilpīgi. Tālāk nodaļā ir aprakstīti datu atkārtotu testu izpildes risinājumi, ņemot vērā datu bāzes mainīgumu. Paštestēšanas realizācijas laikā tika aplūkoti šādi datu bāzes sagatavošanas risinājumi atkārtotu testu izpildei:

- Datu bāzes dublējuma sagatavošana. Pirms katra testa reģistrēšanas tiek sagatavots datu bāzes dublējums;
- Pretējo testu ģenerēšana. Tiek izstrādāta tehnoloģija, kas lietotāja definētam testam sagatavo pretējo testu, kas atgriež datu bāzi sākotnējā stāvoklī;
- Pretējo testu reģistrēšana. Tests tiek veidots tā, lai tas ietvertu pilnu notikumu kopu;
- Secīga visu testu izpildīšana. Paštestēšanas režīmā secīgi pēc kārtas tiek izpildīti visi reģistrētie testi;
- Testu prioritātes. Lietotājs paštestēšanas režīmā izpilda tikai tos testus, kas nav atkarīgi no datu kopas. Pārējos testus izpilda izstrādātājs;
- Testu izpildes kritēriji. Sistēmā tiek iebūvēti testa veiksmīgas izpildes kritēriji.

No augstāk aprakstītiem risinājumiem sistēmas paštestēšanā tiks izmantots Testu izpildes kritēriji. Tālāk nodaļas apakšnodaļās detalizētāki tiek aplūkoti augstāk minēto risinājumu priekšrocības un trūkumi.

2.5.1. Datu bāzes dublējuma sagatavošana

Pirms katras testa reģistrēšanas tiek sagatavots aktuālais datu bāzes dublējums. Dublējums tiek uzglabāts tik ilgi, kamēr tiek pielietoti uzkrātie testi, kas reģistrēti, izmantojot konkrēto dublējumu. Pirms testa atkārtotas izpildes tiek uzstādīts atbilstošais datu bāzes dublējums. Šādam risinājumam ir būtiska priekšrocība:

- Testa izpildes brīdī ir pieejams tāds pats datu bāzes stāvoklis, kāds tas bija reģistrējot testu. Risinājums testa izpildes brīdī nodrošina pilnīgi analoģiskus apstākļus kādi tie bija testa reģistrēšanas brīdī.

Šādam risinājumam ir vairāki trūkumi:

- Risinājums ir ļoti laikietilpīgs. Šāda pieeja nozīmē, ka paštestēšanas programmatūrai jānodrošina automātiska dublējuma noņemšana un uzstādīšana atsevišķi katram testam. Protams, testus ir iespējams grupēt pa dublējumiem, bet tas neatrisina problēmu gadījumā, ja testu apjoms nepārtraukti tiek papildināts;
- Papildus ir nepieciešama precīza dublējumu uzkrāšana, uzglabāšana;
- Testu, tātad arī dublējumu skaitam pieaugot, pieaugs arī cietā diska fiziskās atmiņas izmantošanas apjoms, kas apjomīgākām datu bāzēm nozīmētu arī rūpīgu cietā diska un tā atmiņas monitorēšanu.

Tā kā risinājumam ir vairāk trūkumu nekā priekšrocību, tad šāds risinājums netiks iekļauts paštestēšanas tehnoloģijas realizācijā.

2.5.2. Pretējo testu ģenerēšana

Katram lietotāja reģistrētam testa piemēram paštestēšanas programmatūra automātiski sagatavo pretējo testu. Piemēram, ja lietotājs reģistrē testu, kurā no klienta konta tiek izskaitīta noteikta naudas summa, tad paštestēšanas programmatūra automātiski izveido testu, kas klienta kontā ieskaita norādīto naudas summu. Paštestēšanas režīmā, izpildot šo testu, pirmais tiks izpildīts automātiski ģenerētais skripts, tad lietotāja reģistrētais tests. Tādā veidā tiek nodrošināts, ka testa izpilde nebūs atkarīga no datu bāzes stāvokļa. Tehniski tas varētu tikt realizēts automātiski veidojot pretējās komandas skriptu, piemēram, ja testa piemērs satur delete komandu, tad pretējais tests tiktu izveidots ar insert komandu.

Tehnoloģija, kas nodrošina pretējo testu ģenerēšanu ir pietiekami sarežģīta, lai to neiekļautu paštestēšanas programmatūras realizācijā.

2.5.3. Pretējo testu reģistrēšana

Risinājuma mērķis ir virzīt lietotāja testa reģistrēšanu tā, lai tie saturētu pilnu notikumu kopu. Paštestēšanas programmatūra kontrolē lietotāja, kurš reģistrē testa piemēru, darbības, liekot lietotājam, ar sākotnējiem testa piemēriem nodrošināt datu kopu, ar kuru lietotājs pēc tam reģistrē citus testu piemērus. Piemēram, noteiktas naudas summas izskaitīšanas no klienta naudas konta gadījumā lietotājam pirms naudas summas izskaitīšanas no naudas konta nepieciešams veikt naudas summas ieskaitīšanu naudas kontā. Tādā veidā nodrošinot datu kopu, ar kuru veikt naudas summas izskaitīšanu no klienta naudas konta.

Lai nodrošinātu šo risinājumu, paštestēšanas programmatūrai ir jānodrošina dažādas kontroles, kuru realizācija ir sarežģīta.

Risinājuma pielāgošana sarežģītās lietojumprogrammās, kas satur daudz klasifikatoru vai saistīto datu, šāda pieeja būs ļoti laikietilpīga, jo pirms lietojumprogrammas testa reģistrēšanas, lietotājam jāizveido klasifikatoru un saistīto datu testa piemēri. Šādiem gadījumiem risinājums varētu paredzēt datu kopu prioritāšu definēšanu, kurām datu kopām jābūt iekļautām testa piemēros, kurām nē. Tas savukārt sarežģī risinājuma tehnisko realizāciju.

Pastāv iespēja mutiski informēt lietotājus par testu definēšanas kārtību, neiekļaujot papildus kontroles paštestēšanas programmatūrā. Paredzams, ka šādā gadījumā drīz vien pēc jaunu testu reģistrēšanas, liela daļa atkārtoti izpildīto testu rezultāti būs neveiksmīgi.

2.5.4. Secīga visu testu izpildīšana

Risinājums nodrošina šādu secīgu soļu izpildi:

- Brīdī, kad sistēmā tiek ieviesta sistēmas paštestēšanas funkcionalitāte, no reālās datu bāzes tiek noņemts datu bāzes dublējums un uzstādīts testu uzkrāšanas un testu izpildes datu bāzēm. Turpmāk jauns dublējums no reālās bāzes netiek ņemts. Testu uzkrāšanas un izpildes datu bāzēm vienlaicīgi ar reālo datu bāzi tiek uzstādītas tikai datu bāzu objektu (procedūru, funkciju, trigeru, skatījumu u.c.) jaunākās versijas un datu struktūru labojumi;
- Lietotāji testu uzkrāšanas datu bāzē reģistrē jaunus testus;
- Sistēmas paštestēšanas režīmā testu izpildes datu bāzei tiek uzstādīts sākotnēji noņemtais reālās datu bāzes dublējums, jaunākās datu bāzu objektu versijas un datu struktūras, tad automātiski secīgi pēc kārtas izpildīti visi testu uzkrāšanas datu bāzē reģistrētie lietotāju testi. Tādā veidā tiek nodrošināts, ka testi tiek uzkrāti un izpildīti datu bāzēs ar noteiktu datu kopu. Tas nozīmē, ka tests vienmēr tiks uzkrāts un izpildīts uz noteiktas datu kopas, kas nav iespējams regulāra reālās vides dublējuma uzstādīšanas gadījumā.

Risinājuma priekšrocība:

- Testa izpildes brīdī ir datu bāze nodrošina tādu pašu datu kopu, kāda tā bija reģistrējot testu.

Šādam risinājumam ir vairāki trūkumi:

- Risinājums ir diezgan laikietilpīgs. Šāda pieeja nozīmē, ka paštestēšanas programmatūrai jānodrošina automātiska dublējuma uzstādīšana pirms testu izpildes;
- Nav iespējams izpildīt atsevišķu(-s) testu(-s). lai pārbaudīt konkrētu testu, jāveic visu testu izpilde;
- Datu bāzes objektu versiju un struktūru izmaiņu uzkrāšana. Paštestēšanas režīmā testi, kuros tiek izmantotas datu bāzes struktūras un objekti, jāizpilda ar testa uzkrāšanas brīdī aktuālām datu bāzes struktūrām un objektu versijām;
- Sarežģīta tehniskā realizācija. Jāuzkrāj datu bāzes objektu un datu struktūru izmaiņas un jāprot tās pielietot paštestēšanas režīmā.

Tā kā risinājumam ir vairāk trūkumu nekā priekšrocību, tad šāds risinājums netiks iekļauts paštestēšanas tehnoloģijas realizācijā.

2.5.5. Testu prioritātes

Ne visi testi ir atkarīgi no datu bāzes stāvokļa. Piemēram, testa piemērs, kur klienta naudas kontā tiek ieskaitītas noteikta naudas summa. Šādu testu reģistrēšanai nav nepieciešamas papildus

kontroles, datu kopas sagatavošana (pieņemot, ka klientam ir aktīvs naudas konts). Savukārt, testa piemēram, kas veic naudas izskaitīšanu no klienta naudas konta, ir nepieciešama papildus kontrole vai datu kopas sagatavošana. Risinājums paredz prioritāšu definēšanu testu piemēriem atkarībā no iepriekš definētas datu kopas:

- 1. prioritāte – ar šo prioritāti automātisku testu izpildi veic sistēmas izstrādātāji, kuriem ir IT zināšanas un spēj sagatavot datu bāzes stāvokli atbilstoši testu prasībām. Izstrādātāji testu piemēriem sagatavo atbilstošos skriptu failus, lai testu izpildē būtu pieejama nepieciešamā datu kopa.

  Arī lietotājiem ir iespējams veikt testus ar 1. prioritāti, bet šajā gadījumā testu rezultāti, kas atkarīgi no konkrētas datu kopas, iespējams, nesakritīs ar rezultātu, kas reģistrēts testa uzkrāšanas režīmā. Pēc neveiksmīgas testu izpildes lietotājam ir iespējams iepazīties ar detalizētāku testa izpildes aprakstu, pēc kura lietotājam ir iespējams izdarīt secinājumus (piemēram, naudas summas izskaitīšana no klienta naudas konta nav iespējama, jo klientam kontā nav naudas).
- 2. prioritāte – ar šo prioritāti automātisku testu izpildi veic sistēmas lietotāji, kuriem nav IT zināšanu. Veicot sistēmas paštestēšanu ar 2. prioritāti, netiek veikti testi, kas atkarīgi no iepriekš reģistrētu datu kopas.

Daļa testu izpildes paliek izstrādātāja ziņā. Lietotāji izpilda tikai tos testus, kas nav atkarīgi no datu bāzes izmaiņām. Piedāvātajā risinājumā lietotājs ir ļoti atkarīgs no izstrādātāja, tādēļ aprakstītais risinājums netiks iekļauts paštestēšanas programmatūrā.

2.5.6. Testu izpildes kritēriji

Lai testi nebūtu atkarīgi no datu kopas, nepieciešams kontrolēt testa veiksmīgas izpildes pamata kritērijus, bez kuriem testa izpilde nav iespējama. Kontroli nodrošina testējamā sistēmā iestrādātie testa izpildes kritēriji, kas testa izpildes laikā pārliecinās par norādītā kritērija izpildes iespējamību. Ja testa punktā norādītais kritērijs izpildās, tad testa izpilde tiek turpināta, pretējā gadījumā testa izpilde tiek pārtraukta, atzīmējot testa izpildi kā neveiksmīgu. Papildus tiek norādīts testa pārtraukšanas iemesls, kuru lietotājs var novērst un veikt testa izpildi atkārtoti. Piemēram, tiek izpildīts noteiktas naudas summas izskaitīšanas no klienta naudas konta testa piemērs. Šajā testā varētu tikt iestrādāti un kontrolēti šādi testa izpildes kritēriji:

- Klienta naudas kontā ir pieejams testa piemērā norādītais naudas daudzums;
- Klientam ir reģistrēts testa piemērā norādītais naudas konts;
- Klienta naudas konts nav slēgts.

Risinājuma priekšrocības:

- Testa izpildes brīdī datu bāzei nav nepieciešams nodrošināt tādu pašu datu kopu, kāda tā bija reģistrējot testu;
- Risinājums nav laikietilpīgs. Testu izpildei nav nepieciešams uzstādīt datu bāzes dublējumu;
- Ir iespējams izpildīt konkrētu(-s) testu(-s), neveicot visu reģistrēto testu izpildi;
- Salīdzinoši vienkārša tehniskā realizācija, jo paštestēšanas programmatūras realizācijas pieeju.

Šādam risinājumam ir trūkums:

- Izstrādātājam papildus testējamā sistēmā jāiestrādā testa izpildes kritēriji.

Tālāk darbā ir aprakstīts jēdziens Testa punkts. Testu izpildes kritēriji paštestēšanas programmatūrā tiek realizēti kā testa punkti, kas sakrīt ar kopējo paštestēšanas realizācijas pieeju. Paštestēšanas programmatūras realizācijā ir iekļauts Testu izpildes kritēriji risinājums.

2.6. Testa punkts un tā realizācija

Testa punkts – komanda, pie kuras tiek izpildītas sistēmas testēšanas darbības. Precīzāk – testa punkts ir programmēšanas valodas komanda programmas tekstā, pirms kuras izpildes tiek ievietotas testēšanas darbību komandas. Testa punkts nodrošina konkrētu darbību un lauku vērtību saglabāšanu pie testu uzkrāšanas, kā arī programmas izpildes rezultāta piereģistrēšanu pie testu atkārtotas izpildes. Izmantojot testa punktus, iespējams atkārot sistēmas notikumu izpildi.

Kā aprakstīts iepriekšējās nodaļās, paštestēšanas iespējas tiek iestrādātas testējamā sistēmā, konkrēti – tiek pierakstītas pie testa punktiem. kurus iespējams iekļaut sistēmā vismaz divos veidos:

- Fiziski iejaucoties sistēmas programmatūras izejas kodā. Programmētājs izstrādājot sistēmu, programmatūras kodā iestrādā arī testa punktus, kas reģistrē sistēmas darbības;
- Speciālistam, kas definē biznesa procesu shēmas, norādot testa punktus biznesa procesā. Šajā gadījumā ir nepieciešama biznesa procesu un sistēmas savstarpējā atbilstība un papildus līdzekļi testēšanas darbību pārnešanai uz sistēmu.

Paštestēšanas pieeja nosaka, ka testu punkti testējamā sistēmā jāizvieto tā, lai tie nosegtu sistēmas kritisko funkcionalitāti. Tajā pašā laikā paštestēšanas pieeja pieļauj testa punktu izvietošanu tā, lai nodrošinātu:

- testēšanu pēc baltās kastes metodes, iestrādājot testa punktus, kas pārklāj visu sistēmas funkcionalitāti;

- svarīgākās funkcionalitātes testēšanu, piemēram, datu saglabāšanu, iestrādājot dažus svarīgus testa punktus. Jo mazāk iestrādātu testa punktu, jo sistēma mazāk jūtīga pret pirmkoda izmaiņām.

Programmētāja ziņā ir izvēlēties nepieciešamo testa punktu izvietošanas stratēģiju – kritiskās funkcionalitātes testēšana vai testēšana pēc baltās kastes metodes, vai testa punktu izvietošana, ņemot vērā jūtība pret pirmkoda izmaiņām.

Sākotnēji izstrādājot paštestēšanas programmatūras koncepciju, bija paredzēts izstrādāt tikai testa punktus, kas nodrošina datu saglabāšanas datu bāzē, datu atlasīšanas no datu bāzes notikumu, reģistrēšanu. Svarīgi bija pārbaudīt vai atkārtoti izpildot datu bāzes komandu (INSERT, UPDATA, SELECT, procedūras vai funkcijas izsaukums u.c.), datu bāzē saglabātais vai no datu bāzes atlasītais rezultāts sakristu ar pirmajā reizē veikto datu saglabāšanu vai datu atlasīšanu.

Attīstot paštestēšanas koncepciju, radās ideja izmantot testa punkta pieeju visu sistēmas notikumu reģistrēšanai. Tātad testa punkti reģistrē ne tikai datu saglabāšanas datu bāzē vai datu atlasīšanas no datu bāzes notikumus, bet arī citus lietojumprogrammas notikumus (lauku aizpildīšana lietojumprogrammas formā, lietojumprogrammas notikumu izsaukšana utt). Šādas izmaiņas nodrošina arī lietotāju saskarnes, biznesa loģikas testēšanu, kā arī šāda pieeja nodrošināja iespēju lietotājam sistēmu lietot demonstrācijas režīmā. Tādā veidā ar salīdzinoši nelieliem ieguldījumiem būtiski tika palielināta paštestēšanas funkcionalitāte.

2.6.1. Testa punkta realizācija

Lai realizētu testa punktus, ir nepieciešams noteikt kā strādā sistēma, kāds ir izstrādātās sistēmas modelis un kādu informāciju būs nepieciešams piefiksēt, reģistrējot testa piemēru. Tas ir nepieciešams, lai:

- varētu izstrādāt vai pielietot jau gatavus testa punktus, kas spēj piefiksēt nepieciešamo informāciju testējamā sistēmā;
- apzinātu, kur tieši programmatūras kodā nepieciešams pievienot testa punktus;
- spētu izstrādāt vai pielietot jau eksistējošo testu atspēlēšanas moduli.

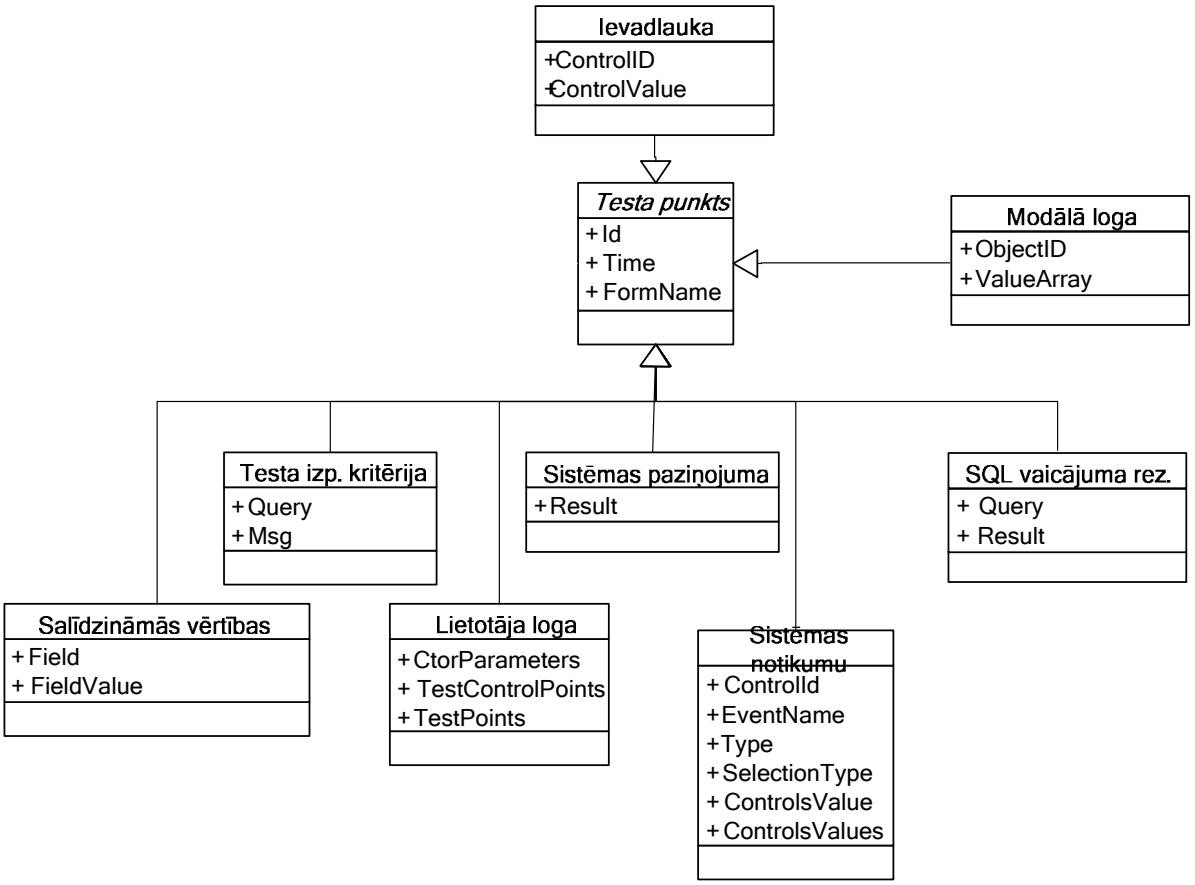
Katram testa punktam ir papildus klase, kas satur visu nepieciešamo informāciju, kuru saņem testa punkts. Testu punktu objekti ir izstrādāti, lai atvieglotu darbu ar testa scenāriju – scenārija izveidošanu un tā atspēlēšanu. Pašreiz paštestēšanas programmatūrā ir ieviestas 2.7. attēlā norādītās testu punktu objektu klases, ar kuru palīdzību ir iespējams piefiksēt lietotāja/sistēmas darbības:

- Lietotāja loga testa punkts – testa punkts, kas apzīmē lietotāja logu. Visi testa punkti, kuri atrodas zem šī testa punkta, pieder šī testa punkta lietotāja logam un nosaka visas

darbības šajā logā. Šī tipa objektiem ir jāsatur izsauktā loga nosaukumu, kā arī vadīklas testa punktus, kurus dinamiski var pievienot. Ja neizpildās testa kritēriji, tad tests netiek turpināts, jo tā izpildes rezultāts būs negatīvs;

- Salīdzināmās vērtības testa punkts – testa punkts, kas dod iespēju piefiksēt vērtības, kuras lietotājam nav pieejamas (piemēram, programmatūras iekšējie mainīgie, ārējās saskarnes dati) un pēc testa atspēlēšanas salīdzināt šīs vērtības. Tas ir noderīgi pārbaudei vai ir mainījusies vērtības, kuras nedrīkst mainīties, vai arī sakrīt rezultāts pēc kādas funkcijas izpildes;
- Testa izpildes kritērija testa punkts – testa punkts nodrošina testa kritēriju saglabāšanu. Testa kritērijs ir pieprasījums uz datubāzi, kas jāatgriež 0 vai 1. Ja pieprasījums atgrieza 0, tad nav testa izpilde netiek turpināta, jo testa rezultāts būs negatīvs. Testa punkts kontrolē vai testa izpilde ir iespējama (piemēram, pirms naudas izskaitīšanas no naudas konta, tiek pārbaudīts, vai konts nav slēgts un vai kontā ir nauda). Izmantojot testu kritēriju testa punktus, iespējams noteikt kritērijus, ar kādiem uzkrātais tests izpildīsies. Sistēmas paštestēšanas režīmā testa izpildes kritēriju punkti pārbauda, vai izpildās testa punktos norādītie nosacījumi. Ja nosacījums neizpildās, tad tests ir neveiksmīgs un lietotājam ir pieejams detalizēts testa izpildes apraksts, kurā precīzi tiek norādīts iemesls, kādēļ tests nav izpildījies.
- Ievadlauka testa punkts – testa punkts tiek izmantots, lai testa piemēra failā piereģistrētu lietotāja ievadītos datus lietotāja saskarnes ievadlaukos;
- Sistēmas paziņojuma testa punkti – testa punkts piereģistrē sistēmas paziņojumā nospiesto pogu;
- Modālā loga testa punkti – tā kā paštestēšanas pieeja nenodrošina modālo logu testēšanu, tad modālo logu testa punkti tiek izmantoti, lai būtu iespējams simulēt modālo logu darbību, tos neizsaucot
- Lietojumprogrammas notikuma testa punkti – testa punkts reģistrē dažādas testējamajā sistēmā veiktās darbības, piemēram, pogas nospiešanu, dubultklikšķus utt.
- SQL vaicājuma rezultāta testa punkti – ka datu bāzes pieprasījuma rezultāta testa punkts piereģistrē testa failā vērtības, kas tika atgrieztas no datubāzes. Testa punkts tiek izmantots pēc datu saglabāšanas, lai salīdzinātu datu bāzē saglabātos datus, kādi saglabāti reģistrējot testu, kādi saglabāti atkārtoti izpildot testu;

- Paštestēšanas testa punkti – Dažādi testa punkti, kas nodrošina paštestēšanas funkcionalitātes sekmīgu darbību iebūvētajā sistēmā – piemēram, konfigurācijas ielasīšana, rezultāta XML faila izveidi, visu izpildīto SQL vaicājumu pārtveršanu utt.

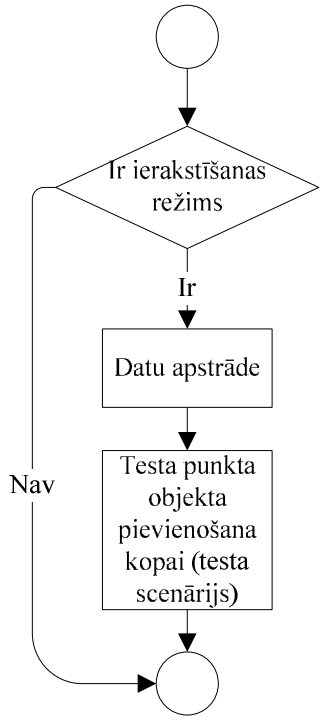


2.7. att. **Testa punktu objektu UML diagramma**

Katrs testa punkts programmatūras kodā tiek pievienots ar tā attiecīgās funkcijas izsaukumu. Visas funkcijas – testa punkti, kas nodrošina testa punktu objektu pievienošanu scenārijam, atrodas vienā statiskā klasē un pašas ir statiskas, kas ļauj izsaukt šīs funkcijas bez galvenās klases inicializēšanas no jebkuras vietas sistēma.

Katrs testa punkta izsaukums, pievieno attiecīga testa punkta tipa objektu scenārijam (skat. 2.8. attēlā). Parasti tas ir testa punkta objekta izveidošana un pievienošana scenārijam, bet, piemēram, Ievadlauka testa punkts var pārrakstīt iepriekšējo Ievadlauka testa punktu, gadījumā ja darbība notiek vienā Ievadlaukā (piemēram, tekstā lauka tiek ievadīta vērtība ‘a’ un pēc tam ‘b’, kas kopā veido ‘ab’, tad scenārijā tiek piefiksēts, ka tekstā laukā ir ievadīta vērtība ‘ab’).

Lai testa punkta pievienošana nepatērētu laiku uz apstrādi, gadījumā, ja lietotājs darbojas programmatūras lietošanas režīmā, katrā funkcijā ir iestrādāta pārbaude, kas pārbauda vai sistēma tiek lietota ierakstīšanas režīmā. Ja sistēma neatrodas ierakstīšanas režīmā, tad funkcija beigs savu darbību uzreiz pēc tās izsaukšanas. Tādā veidā netraucējot lietotāju darbību.

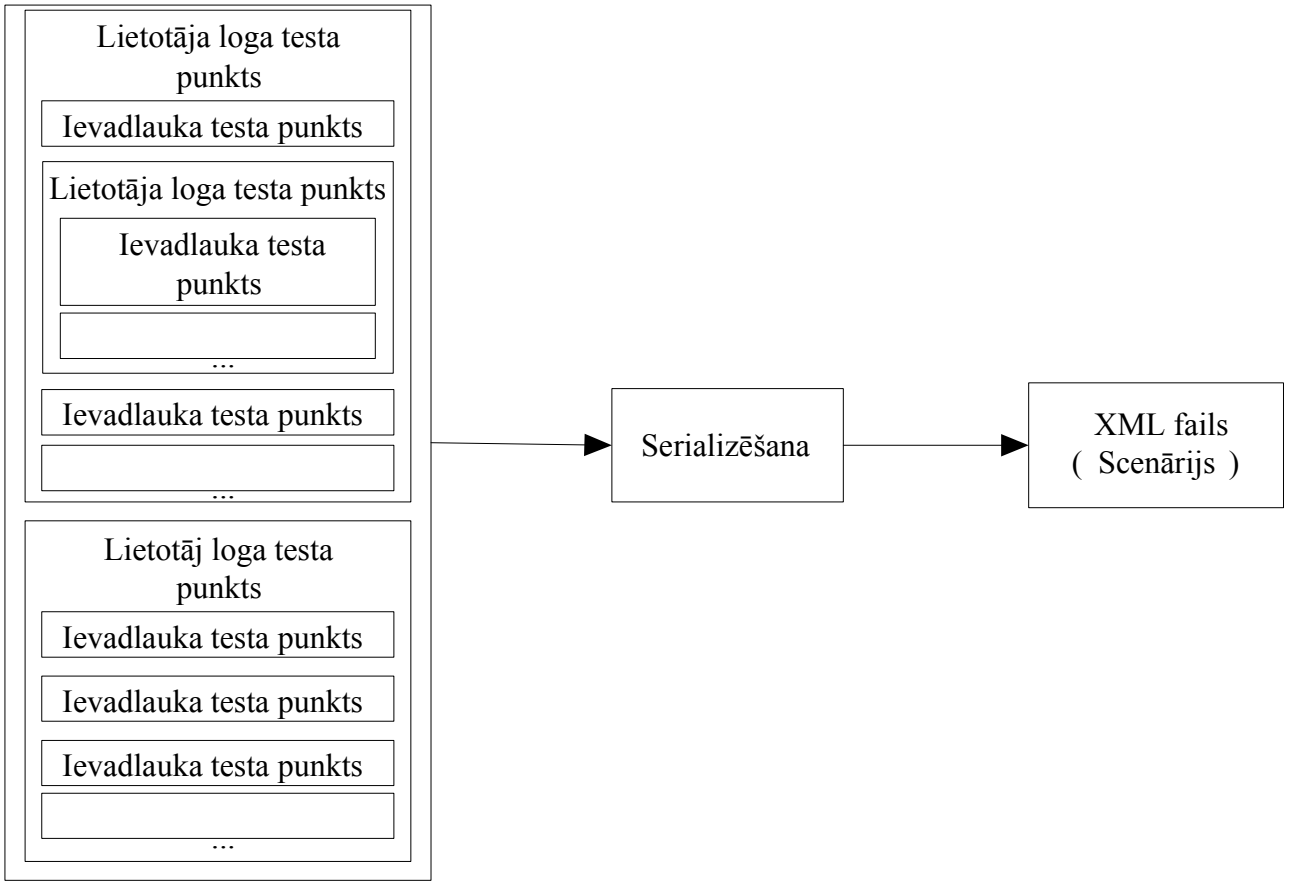


2.8. att. **Testa punktu darbības process**

Testa punktus sistēmas izstrādātāji sistēmā izvieto tā, lai tiktu pārklāta sistēmas kritiskā funkcionalitāte. Pilna testa scenārija gadījumā Ievadlauka testa punktu ir nepieciešams pievienot visu ievadlauku vērtību izmaiņu notikumos, pretēja gadījumā nebūs iespējams atkārtot lietotāja darbības.

Lai nodrošinātu vieglu, pārskatāmu un ērti lietojamu darbu ar testa failiem, testu scenārijiem, bija jānodrošina strukturizēta testa scenārija glabāšana. Scenāriju reģistrēšanai un uzkrāšanai tiek izmantota XML faila struktūra.

Testa scenārijs satur secīgi reģistrētus testa punktus. Testa scenārijs tiek veidots ar .NET platformas serializēšanas palīdzību (skat. 2.9. attēlā), lai testa atspēlēšanas brīdi tiktu izveidota tāda pati testa faila struktūra, kas tika izveidota testa ierakstīšanas brīdī.



2.9. att. **Scenārija transformācija XML failā**

Tā kā katram testa punktam ir savs identifikators, kas nosaka darbību izpildes secību, ir viegli atspēlēt ierakstītas darbības un kļūdas vai nesakritības gadījumā izstrādātājiem ir viegli atrast vietu, kur tieši notika kļūda.

2.6.2. Testu punktu izmantošana sistēmas darbības režīmos

Testa punkts var tikt pielietos šādos sistēmas darbības režīmos:

- Testu uzkrāšanas režīms. Lietotājiem veidojot jaunu testu, sistēmā iestrādātos testa punktos reģistrē testēšanas darbībās fiksēto norādīto informāciju testa failā. Testa punktos var tikt reģistrēta dažāda veida informācija, piemēram, aizpildītā lauka vērtība, komandpogas nospiešana, vērtības no saraksta izvēle u.c.;
- Paštestēšanas režīms. Programmatūra automātiski izpilda testa failos reģistrētos notikumus, uzkrāšanas laikā ievadītos notikumus nomainot ar to izvēli no testa faila. Testu izpildes gaitā sistēmā izvietotie testa punkti veido tādu pašu testa failu kā testu uzkrāšanas režīmā. Pēc testu izpildes tiek salīdzināts testu uzkrāšanas režīmā izveidotais testa fails ar paštestēšanas režīmā izveidoto testa failu. Ja failu saturs sakrīt, tad tests ir veiksmīgi izpildījies, pretējā gadījumā testa izpilde ir neveiksmīga;

- Demonstrāciju režīms. Demonstrācijas režīmam tiek izmantoti testu uzkrāšanas režīmā izveidotie un paštestēšanas režīmā veiksmīgi izpildījušies testu faili. Demonstrācijas režīmā definētā laika intervālā vai lietotājam pa soļiem izpildot komandas no testa faila var tikt demonstrēta sistēmas funkcionalitāte gan jaunu sistēmas lietotāju apmācīšanai, gan sistēmas funkcionalitātes demonstrēšanai potenciālo sistēmas pircēju piesaistīšanai.

Nākamajā 2.1. tabulā ir aprakstīti būtiskāko testa punktu nozīme katrā no sistēmas darbības režīmiem.

2.1. tabula

Testu punkti sistēmas darbību režīmos

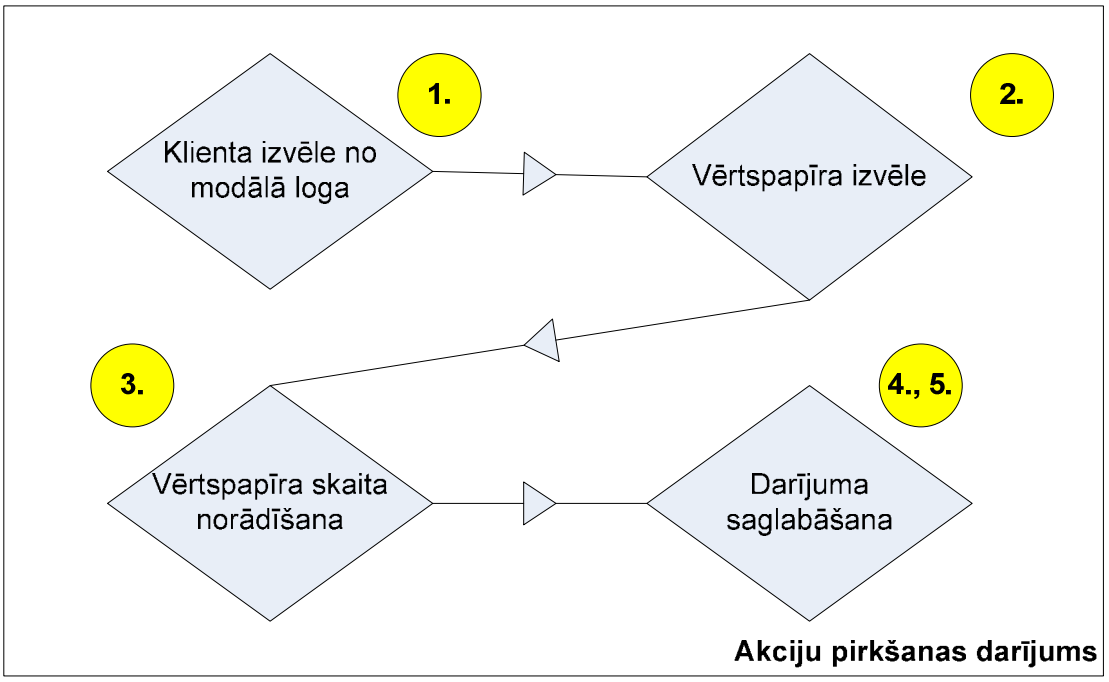
Testu uzkrāšanas	**Paštestēšanas**	**Demonstrācijas**
1. Ievadlauka testa punkts		
Lauka nosaukuma un laukā norādītās vērtības reģistrēšana testa failā.	Lauka nosaukuma un lauka vērtības nolasīšana no testa faila un vērtības ierakstīšana atbilstošajā laukā.	Skat. paštestēšanas režīmu.
2. Salīdzināmās vērtības testa punkts		
Dažādu vērtību, tai skaitā aprēķina rezultātā iegūstamo vērtību, reģistrēšana testa failā.	Izšķir divus gadījumus (papildus atribūts, kas norāda uz konkrēto gadījumu): • Testu izpildīšanas laikā aprēķinātās vērtības salīdzināšana ar testu uzkrāšanas laikā testa failā reģistrēto vērtību; • Testu uzkrāšanas laikā testa failā reģistrētās vērtības izmantošana testu izpildē, nepārrēķinot pašu vērtību.	Testu uzkrāšanas laikā testa failā reģistrētās vērtības izmantošana testu izpildē, nepārrēķinot pašu vērtību.
3. Sistēmas paziņojuma testa punkts		
Paziņojuma izsaukuma un lietotāja veiktās operācijas reģistrēšana testa failā.	Paziņojums lietotājam netiek attēlots. Testu izpildīšana notiek, ņemot vērā testa failā reģistrēto lietotāja veikto operāciju.	Šajā režīmā sistēmas paziņojums tiek attēlots uz ekrāna.
4. Modālā loga testa punkts		
Modālā loga izsaukuma un atgriežamo vērtību reģistrēšana testa failā.	Modālais logs testu izpildes laikā netiek atvērts, bet tiek izmantotas testu uzkrāšanas režīmā reģistrētās modālā loga vērtības no testa faila.	Šajā režīmā modālais logs tiek attēlots uz ekrāna.
5. SQL vaicājuma rezultāts testa punkts		
Testa rezultātu reģistrēšana testa failā. Testa rezultāts var būt gan konkrēta lauka vērtība, gan vaicājuma rezultāts.	Salīdzina testu izpildīšanas rezultātā iegūtās vērtības ar vērtībām, kas testu uzkrāšanas režīmā reģistrētas testa failā.	Testa punkts šajā režīmā netiek izmantots.
6. Lietojumprogrammas notikuma testa punkts		

Lietojumprogrammas notikumu reģistrēšana testa failā.	Lietojumprogrammas notikumu nolasīšana no testa faila un izpildīšana.	Skat. paštestēšanas režīmu.
7. Testu izpildes kritērija testa punkts		
Darbība šajā režīmā netiek izmantota.	Kontrolē vai testu izpilde esošajā datu bāzes stāvoklī ir iespējama.	Skat. paštestēšanas režīmu.

2.6.3. Testu punktu lietošanas piemērs

Lai parādītu testa punktu lietošanu, nākamajā attēlā ir parādīts akciju pirkšanas darījuma process. Akciju pirkšanas darījuma reģistrēšana sastāv no šādiem pamata soļiem:

- Klienta norādīšana;
- Vērtspapīra izvēle;
- Vērtspapīra skaita norādīšana;
- Darījuma saglabāšana.



2.10. att. **Akciju pirkšanas darījuma process**

Lai ieviestu paštestēšanu akciju pirkšanas darījuma procesā, sistēmā tiktu izvietoti šādi pieci testa punkti un pie tiem tiek pierakstītas dažāda veida testēšanas darbības:

- *Modālais logs testa punkts* uzkrāšanas failā reģistrē modālajā logā izvēlēto klientu.
- *Ievadlauka testa punkts* testa uzkrāšanas failā reģistrē darījumā norādīto vērtspapīru.
- *Ievadlauka testa punkts* testa failā reģistrē darījumā norādīto vērtspapīru skaitu.
- *Lietojumprogrammas notikuma testa punkts* testa failā reģistrē Saglabāt pogas nospiešanas notikumu.

- *SQL vaicājuma rezultāta testa punkts* testa failā reģistrē datus, kas pēc pogas Saglabāt nospiešanas, saglabāti datu bāzē.

Reģistrējot akcijas pirkšanas darījuma testa piemēru, katrs no punktiem testa uzkrāšanas failā piereģistrē informāciju, kas tiek izmantota testa atspēlēšanā. Atspēlējot akciju pirkšanas darījuma testu, paštestēšanas programmatūra no testa faila pa soļiem nolasa un izpilda failā reģistrētās darbības. Izpildot testa failā norādītās darbības, tiek veidots jauns testa fails. Pēc visu darbību izpildes savstarpēji tiek salīdzināti testu faili, kuriem veiksmīga testa izpildes gadījumā ir jāsakrīt. Ja faili nesakrīt, tad lietotājam testēšanas programmatūras lietojumprogrammā ir iespējams identificēt vietu (komandu) testa failā, kas izpildījusies kļūdaini.

3. PAŠTESTĒŠANAS IMPLEMENTĀCIJA

Paštestēšanas programmatūra ir daļa no izstrādājamās programmatūras. Tas nozīmē, ka pie programmatūras izstrādātājiem, pasūtītājiem un lietotājiem nav nepieciešamības uzstādīt papildus testēšanas rīkus, ar kuru palīdzību veikt sistēmas testēšanu. Sistēmas testēšanu nodrošina paštestēšanas programmatūra, kas daļēji ir iebūvēta pašā testējamā sistēmā.

Paštestēšanas pieejas sniegto iespēju novērtēšanai reālos programmatūras projektos 2010. gadā tika uzsākta paštestēšanas rīka izstrāde. Lai rīks atbilstu paštestēšanas pieejas koncepcijai, tas tika sadalīts šādos divos moduļos:

- paštestēšanas modulis – C# .NET bibliotēka (.dll fails), kas nodrošina paštestēšanas lietojumprogrammas saskarni (API) testa punktu iebūvēšanai testējamajā sistēmā;
- paštestēšanas testu pārvaldes modulis – C# .NET lietojumprogramma, kas nodrošina testu veidošanu, rediģēšanu, izpildi, rezultātu salīdzināšanu, rīka konfigurēšanu un citas ar testu pārvaldību saistītas darbības.

Paštestēšanas rīka izstrāde ir eksperimentāla un satur virkni ierobežojumus, kas ir attēloti rīku salīdzināšanas tabulā (4.6. tabula). Norādītie trūkumi, ierobežojumi nebūt nenozīmē, ka paštestēšana tos neatbalsta pēc būtības. Darba galvenais paštestēšanas implementācijas mērķis bija parādīt, ka paštestēšanas pieejas koncepts ir dzīvotspējīgs un ir iespējama tā implementācijas un aprobācija. Tādēļ paštestēšana tika implementēta konkrētai informāciju sistēmai, kas darbojas uz Windows platformas un ir realizēta .Net (C#, VB, C++). Minēto ierobežojumu dēļ pašreiz nav iespējama paštestēšanas rīka rūpnieciska ražošana. Lai to nodrošinātu, nepieciešami ieguldījumi, cilvēku resursi. Iespējams, nākotnē būs projekti, kas atbalstīs paštestēšanas rīka pilnveidošanu līdz tā rūpnieciskai ražošanai, izplatīšanai un uzturēšanai.

Turpmākajās apakšnodaļās tiks aprakstīta katra moduļa tehniskā realizācija.

3.1. Paštestēšanas modulis

Kā jau iepriekš tika minēts, paštestēšanas modulis ir C# .NET bibliotēka (SelfTesting.Core.dll), kas satur metodes, kuras, ievietojot testējamās sistēmas programmatūras kodā, nodrošina testa punktu veidošanu. Būtībā paštestēšanas modulis nodrošina testa ierakstīšanas funkcionalitāti, kuru pēc tam testa vairākkārtējai izpildei un testa rezultātu pārbaudei izmanto paštestēšanas testu pārvaldes modulis.

Lai nodrošinātu veiksmīgu testu ierakstīšanu un pēc tam arī atspēlēšanu, testējamajā sistēmā obligāti jāiekļauj paštestēšanas testa punkti, kas pirms sistēmas palaišanas ielasa paštestēšanas

konfigurāciju (skat. 3.1. attēlā 5. rindā) un pēc sistēmas aizvēršanas izveido testa punktu XML failu (skat. 3.1. attēlā 7. rindā).

```
static class Program
{
    [STAThread]
    static void Main(String[] Parameters)
    {
        Tester.LoadConfiguration();
        Application.Run(new frmLogin(Parameters));
        Tester.CreateXml();
    }
}
```

3.1. att. **Paštestēšanas testa punktu iekļaušanas piemērs**

Papildus, lai būtu iespējams izmantot visas konfigurācijā piedāvātās iespējas, testējamās sistēmas programmatūras kodā jāveic nelielas izmaiņas, proti, datu bāzes savienojuma klases iniciēšanu (skat. 3.2. attēlā) jāveic, izmantojot paštestēšanas moduļa piedāvāto starpniekklasi (skat. 3.3. attēlā).

```
Globals.OraConn = new OraConn();
```

3.2. att. **Datu bāzes savienojuma klases iniciēšana**

```
Globals.OraConn = SelfTestingProxyGen.CreateObject<OraConn>();
```

3.3. att. **Datu bāzes savienojuma klases iniciēšana caur starpniekklasi**

Šādā veidā tiek panākts, ka visi uz datu bāzi sūtītie vaicājumi un to atgrieztie dati iet caur starpniekklasi, tādējādi ļaujot paštestēšanas rīkam tos piereģistrēt un līdz ar to arī salīdzināt. Detalizētāk par šo iespēju ir aprakstīts 3.2. apakšnodaļā.

Pēc tam, kad ir izpildīti paštestēšanas iebūvēšanas priekšnosacījumi, var sākt ievietot testa punktus programmatūras kodā. Visbiežāk tiek izmantoti ievadlauku vērtību un lietojumprogrammas notikumu testa punkti, kurus parasti ievieto notikumu apstrādes metožu sākumā. 3.4. attēlā redzamā metode *Tester.LogFieldAndValue* vienlaikus piereģistrē gan notikumu, gan arī ievadlauka vērtību, ja tāda ir. Savukārt, lai atspēlēšanas laikā paštestēšanas rīks mācētu atvērt formu un pēc tam tajā veikt dažādas darbības, formas konstruktora metodē jāievieto paštestēšanas testa punkta metodi *Tester.AddFormTest*, kas piereģistrē formas atvēršanu. Tāpat 3.4. attēlā redzams pielietojums kritērija testa punkta metodei *Tester.LogCriterion*, kas šajā gadījumā pārbauda, vai datu bāzē eksistē izkrītošajā izvēlnē izvēlētā valsts. Ja neeksistē, tad tiek parādīts paziņojums „Neizdevās atlasīt valsti!" un pārtraukta tālāka testa izpilde, jo ir skaidrs, ka tests neizpildīsies.

```
public frmUser()
{
    Tester.AddFormTest(this);
    ...
}

private void txtUsername_TextChanged(object sender, EventArgs e)
{
    Tester.LogFieldAndValue(txtUsername);
    ...
}

private void cmbCountry_SelectedIndexChanged(object sender, EventArgs e)
{
    if (cmbCountry.SelectedIndex > -1)
    {
        Tester.LogCriterion("SELECT COUNT(*) FROM COUNTRIES WHERE NAME = '"
              + cmbCountry.Text + "'", "Neizdevās atlasīt valsti!");
    }
    ...
    Tester.LogFieldAndValue(cmbCountry);
}

private void btnClose_Click(object sender, EventArgs e)
{
    Tester.LogFieldAndValue(btnClose);
    ...
    this.Close();
}
```

3.4. att. **Testa punktu iekļaušana programmatūras kodā**

Tad, kad ir iegūts neliels priekšstats par to, kā notiek testa punktu iebūvēšana testējamajā sistēmā, svarīgi uzzināt, kur un kā testa punkti veic piereģistrēšanu.

Testa ierakstīšanas vai atspēlēšanas brīdī, paštestēšanas modulis veido testa punktu XML failu, kas pēc testējamās sistēmas aizvēršanas tiek saglabāts konfigurācijā norādītajā vietā.

Katrai testa punkta metodei ir atbilstošs testa punkts XML failā. 3.5. attēlā attēlots fragments no testa punktu XML faila, kas tiek uzģenerēts, ja izpilda 3.4. attēlā redzamos testa punktus. Testa punktu XML faila sākumā vienmēr tiek norādīts testējamās sistēmas izpildfails un režīms, kādā tika izpildīts tests. Tālāk seko visi testa kritēriji, kuri tiek izpildīti testa izpildes sākumā, lai pārliecinātos, vai ir vērts turpināt testa izpildi. Pēc tam seko programmatūras kodā iebūvēto testa punktu piereģistrētās vērtības XML formātā.

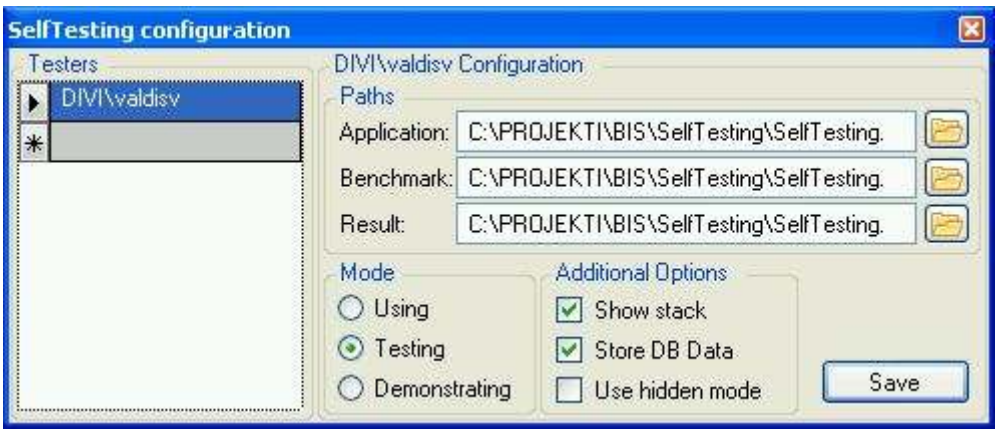
```
<?xml version="1.0" encoding="windows-1257"?>
<Test xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <App>IST.exe</App>
  <Mode>Testing, ShowStack</Mode>
  <Criterions>
    <Criterion Id="1" IsImportant="false">
      <Condition>SELECT COUNT(*) FROM COUNTRIES WHERE NAME = 'Latvija'</Condition>
      <Type>Query</Type>
      <Msg>Neizdevās atlasīt valsti!</Msg>
    </Criterion>
  </Criterions>
  <Points>
    [...]
    <FormAction Id="6" Form="IST.frmUser" IsImportant="false" Action="Open" />
    <Control Id="7" Form="IST.frmUser" Name="txtUsername"
       Event="txtUsername_TextChanged" ControlType="System.Windows.Forms.TextBox">
      <StringValue>Janis.Berzins</StringValue>
    </Control>
    <Control Id="8" Form="IST. frmUser " Name="cmbCountry"
       Event="cmbCountry_SelectedIndexChanged"
       ControlType="System.Windows.Forms.ComboBox">
      <StringValue>Latvija</StringValue>
    </Control>
    [...]
    <Control Id="33" Form="IST.frmUser" Name="btnClose" Event="btnClose_Click"
ControlType="System.Windows.Forms.Button" />
    [...]
  </Points>
</Test>
```

3.5. att. **Testa punktu XML faila fragments**

Kā tika minēts šīs apakšnodaļas sākumā, tad paštestēšanas modulis nodrošina paštestēšanas lietojumprogrammas saskarni (API), ar kuras palīdzību var veikt testa ierakstīšanu un testa punktu XML faila izveidi. Tomēr ar to vien nepietiek, jo nepieciešams no testējamās sistēmas neatkarīgs ārējs rīks, kas, balstoties uz paštestēšanas moduļa izveidoto testa punktu XML failu, nodrošina testa vairākkārtēju izpildi, rezultātu salīdzināšanu, konfigurēšanu un citu automatizētu testēšanas darbību veikšanu. Šim nolūkam paštestēšanas rīks piedāvā paštestēšanas testu pārvaldes moduli, ar kuru tiek iepazīstināts nākošajā apakšnodaļā.

3.2. Paštestēšanas testu pārvaldes modulis

Paštestēšanas testu pārvaldes modulis ir atsevišķa lietojumprogramma, kas kā bibliotēku izmanto iepriekš aprakstīto paštestēšanas moduli. Šī moduļa pamatuzdevums ir nodrošināt testu automatizētu atspēlēšanu, izmantojot ierakstītos testa punktu XML failus un testējamajā sistēmā iebūvētos testa punktus.



3.6. att. **Paštestēšanas konfigurācijas logs**

Lai sāktu testa izpildi, vispirms jāveic paštestēšanas rīka konfigurēšana (skat. 3.6. attēlā). Ņemot vērā, ka bieži vien testējamā sistēma atrodas uz servera un to izmanto vairāki lietotāji, svarīgi katram lietotājam nodrošināt savu konfigurāciju, tādēļ konfigurācijas loga sadaļā „Testers" var norādīt testētāju, t.i., lietotāja vārdu.

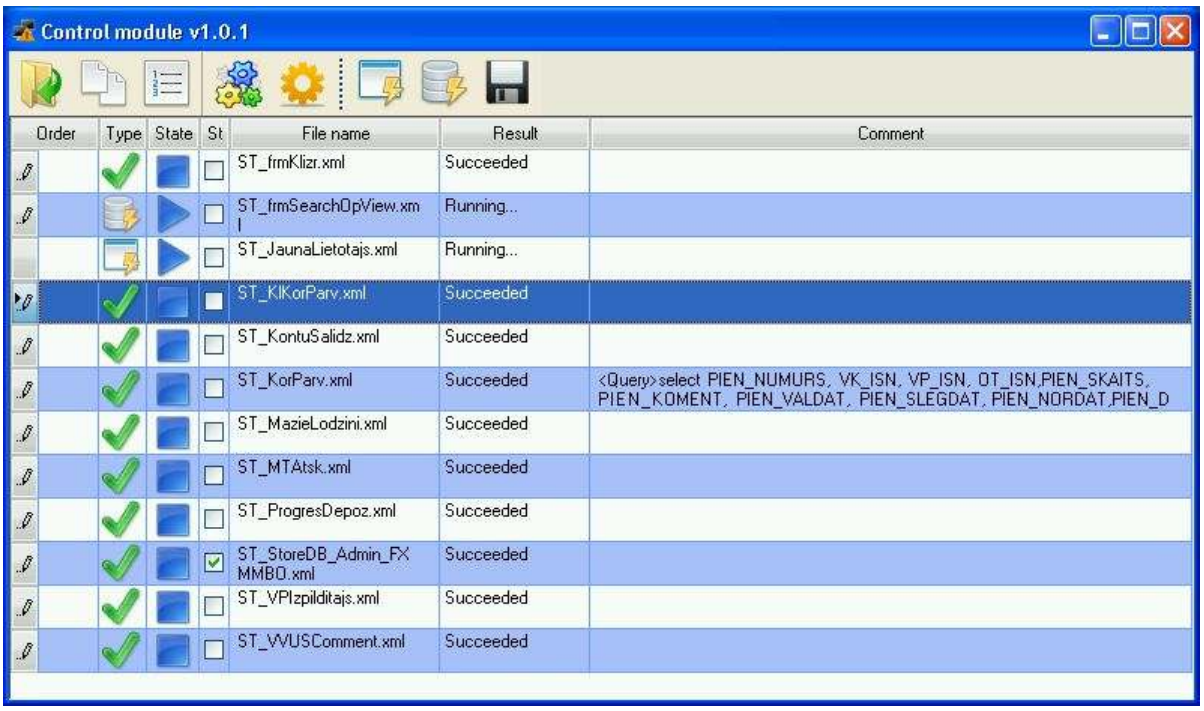
Katram lietotājam jānorāda ceļi uz testējamo sistēmu, ierakstīto un atspēlēto testa punktu XML failiem. Tāpat arī jānorāda viens no testēšanas režīmiem (skat. 2.3. Darbības režīmi):

- lietošanas režīms (*Using mode*);
- testēšanas režīms (*Testing mode*);
- demonstrēšanas režīms (*Demonstrating mode*).

Izvēloties testēšanas režīmu, papildus tiek piedāvātas šādas iespējas:

- **rādīt testa punktu steku** (*Show stack*) – testa ierakstīšanas un izpildīšanas laikā tiek parādīts logs, kurā secīgi sarindoti visi attiecīgi izpildāmie vai ierakstītie testa punkti.
- **saglabāt visus datu bāzes datus** (*Store DB Data*) – testa ierakstīšanas un izpildīšanas laikā testa punktu XML failā tiek piereģistrēti visi uz datu bāzi nosūtītie vaicājumi un saņemtie dati, tādējādi ļaujot veikt detalizētu datu bāzes testēšanu.
- **lietot neredzamo režīmu** (*Use hidden mode*) – testa izpildīšanas laikā testējamā sistēma netiek parādīta uz ekrāna, bet tiek izpildīta fonā.

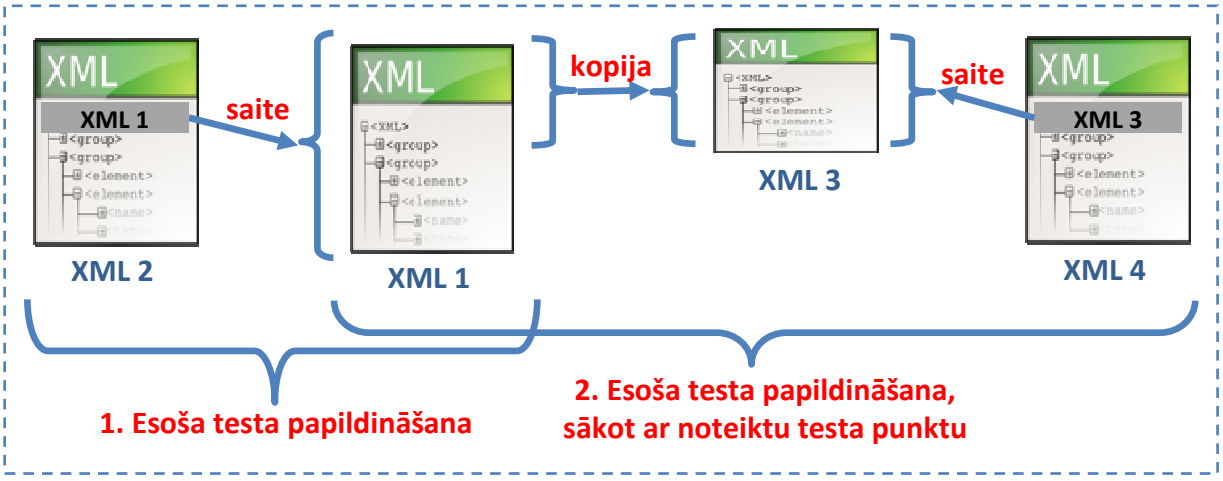
Jāatzīmē, ka šī konfigurācija attiecas ne tikai uz testu izpildīšanas funkcionalitāti, bet arī uz testu ierakstīšanu. Piemēram, ja konfigurācijā tiek izvēlēts lietošanas režīms, tad, lietojot testējamo sistēmu, nenotiks testēšana, tāpat nevarēs arī izpildīt ierakstītos testus (šādā veidā var arī norādīt, kuri sistēmas lietotāji drīkst veikt testēšanu un kuri nedrīkst). Tāpat arī, ja konfigurācijā tiek izvēlēts testēšanas režīms, rādot testa punktu steku, tad gan testus izpildot, gan lietotājam darbinot testējamo sistēmu, tiks parādīts testa punktu steka logs.



3.7. att. **Paštestēšanas testu pārvaldes modulis**

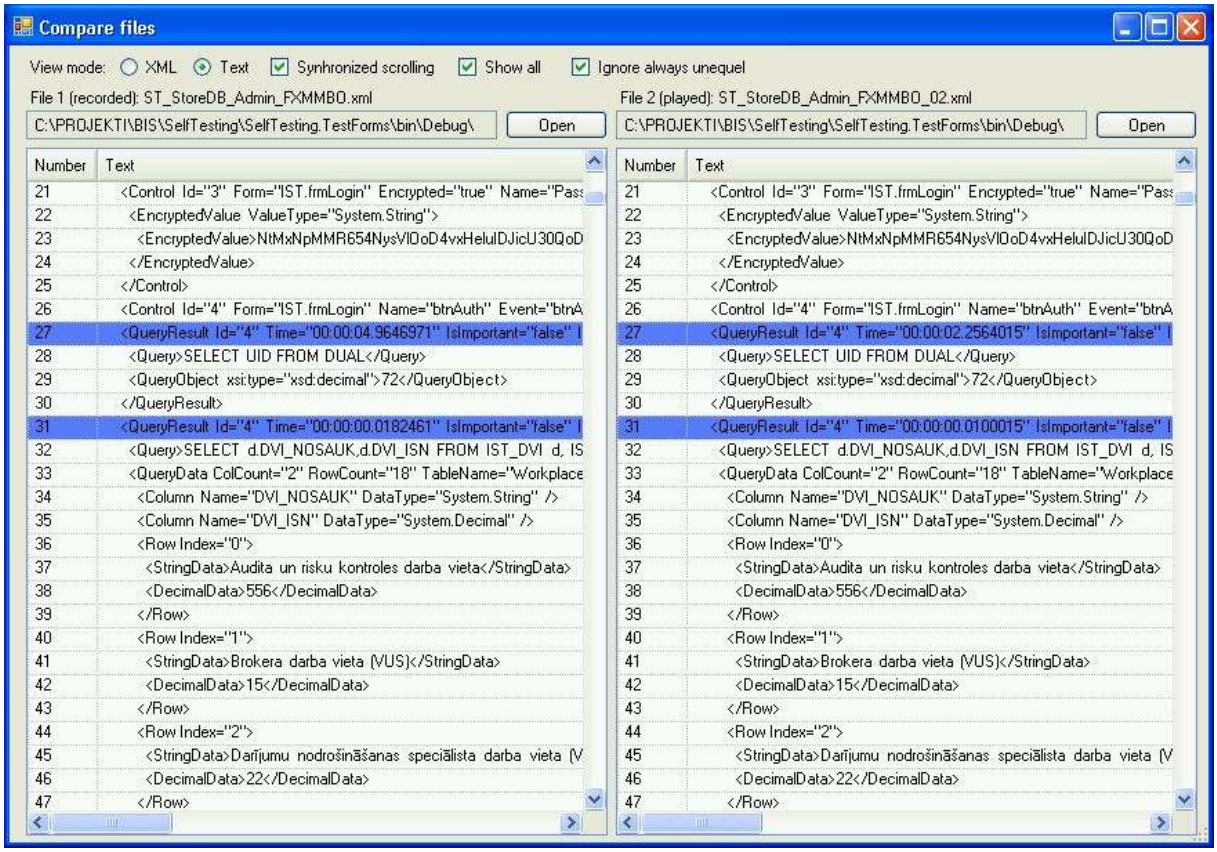
Pēc konfigurēšanas paštestēšanas testu pārvaldes modulī (skat. 3.7. attēlā) jāielasa ierakstītie testi. Iezīmējot sarakstā esošos testus, tos var izpildīt pa vienam vai arī vairākus uzreiz (iespējama secīga vai vienlaicīga izpilde). Neatkarīgi no tā, cik testi iezīmēti, tos var izpildīt trīs veidos:

1. **Lietojumprogrammas testa izpilde** – secīgi tiek izpildītas etalona testa punktu XML failā ierakstītās darbības. Pēc darbību izpildes tiek uzģenerēts jauns testa punktu XML fails;
2. **Datu bāzes testa izpilde** – secīgi tiek izpildīti visi etalona testa punktu XML failā ierakstītie vaicājumi. Pēc darbību izpildes tiek uzģenerēts jauns testa punktu XML fails;
3. **Esoša testa papildināšana** (skat. 3.8. attēlā 1. punktu) – secīgi tiek izpildītas etalona testa punktu XML failā ierakstītās darbības līdz pēdējai komandai. Pēc tam testējamā sistēma pāriet testu ierakstīšanas režīmā, ļaujot testētājam veikt dažādas darbības. Pēc darbību veikšanas tiek uzģenerēts jauns testa punktu XML fails, kura sākumā ir saite uz etalona testa punktu XML failu, bet pēc tam seko no jauna ierakstītie testa punkti.



3.8. att. **Esoša testa papildināšana**

Izpildot testu, jebkurā brīdī iespējams to atcelt, apturēt un atsākt. Papildus tam, kad tests ir apturēts, sākot no pašreizējās komandas var uzsākt jauna testa izveidi (skat. 3.8. attēlā 2. punktu), kuram kā saite pievienots testa punktu XML fails, kas iekļauj visus līdz šim izpildītos testa punktus. No jauna ierakstītie testa punkti tiek pievienoti aiz šīs saites.



3.9. att. **Testa rezultātu salīdzināšanas logs**

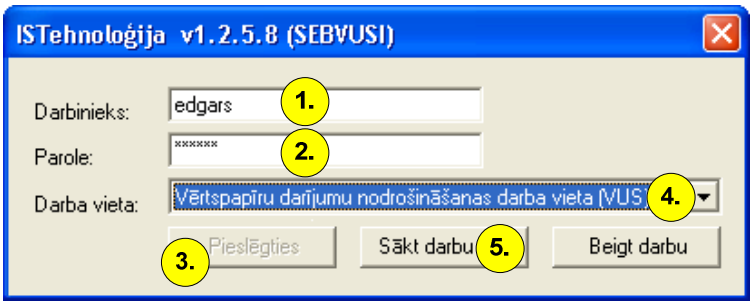
Pēc katra testa izpildes tiek uzģenerēts jauns testa punktu XML fails, kas tiek salīdzināts ar etalona testa punktu XML failu. Balstoties uz šo salīdzināšanu tiek noteikts, vai tests ir izpildījies veiksmīgi, vai nav. Izmantojot testa rezultātu salīdzināšanas logu (skat. 3.9. attēlā), katram testam

var apskatīt arī detalizētu testa izpildes pārskatu, tādējādi uzskatāmā veidā identificējot testa neizpildīšanās iemeslus un kļūdas testējamajā sistēmā.

Jāpiebilst, ka, ņemot vērā, ka testa rezultātu salīdzināšana notiek, salīdzinot katru testa punktu XML failā, brīdī, kad notiek testa punktu iebūvēšana testējamajā sistēmā, ļoti svarīgi ir novērtēt, vai testa punkta piereģistrētajai vērtībai, katru reizi izpildot testu, jābūt vienādai. Ja nav jābūt vienādai, tad testa punktam jāuzstāda pazīme „Nesvarīgs", tādējādi, ja salīdzināšanā šīs vērtības nesakritīs, tas nenozīmēs, ka tests ir neveiksmīgi izpildījies. Šāda pazīme iestrādāta Ievadlauka testa punktiem. Tipisks piemērs šāda pazīmes izmantošanai ir datuma lauks, kas noklusēti ielasa tekošo datumu.

3.3. Paštestēšanas piemērs

Tālāk ir aplūkots sistēmas pieslēgšanās loga testa piemērs, kurā ir izvietoti testa punkti.



3.10. att. **Sistēmas pieslēgšanās loga testa piemērs**

Pieslēgšanas logā lietotājs veic šādas darbības:

1. Ievada lietotāja pieteikumvārdu.
2. Ievada lietotāja paroli.
3. Nospiež poga Pieslēgties.
4. Izvēlas darba vietu.
5. Nospiež pogu Sākt darbu.

Pēc augstāk norādīto darbību veikšanas testa punkti testa failā izveido ierakstus.

```
- <Actions>
  - <Control Id="1" Name="Username" Event="Username_TextChanged" ControlType="System.Windows.Forms.TextBox">
      <Value xsi:type="xsd:string">edgars</Value>
    </Control>
  - <Control Id="2" Name="Passwd" Event="Passwd_TextChanged" ControlType="System.Windows.Forms.TextBox">
      <Value xsi:type="xsd:string">123456</Value>
    </Control>
    <Control Id="3" Name="btnAuth" Event="btnAuth_Click" ControlType="System.Windows.Forms.Button" />
  - <Query Id="4">
      <Query>SELECT d.DVI_NOSAUK,d.DVI_ISN FROM IST_DVI d, IST_SLD s WHERE s.DVI_ISN=d.DVI_ISN AND s.DAR_ISN=72 order by 1</Query>
    </Query>
  - <Control Id="5" Name="Workplace" Event="Workplace_SelectedIndexChanged" ControlType="System.Windows.Forms.ComboBox">
      <Value xsi:type="xsd:int">20</Value>
    </Control>
    <Control Id="6" Name="StartWork" Event="StartWork_Click" ControlType="System.Windows.Forms.Button" />
</Actions>
```

3.11. att. **Testa faila piemērs**

Dokumentā norādītā testēšanas funkciju bibliotēka var tikt izmantota MS Visual Studio vidē izstrādātos projektos. Nepieciešamības gadījumā tā var tikt viegli papildināta ar jaunām funkcijām.

3.4. Testu automatizēšanas ietvari

Paštestēšanas pieeja balstās uz **lineārā testu automatizēšanas ietvara**, proti, testa ierakstīšanas laikā visi testa punkti precīzi atbilst testa punktu XML failā esošajām komandām.

Parasti lineārā ietvara testa skripti ir viendimensionāli un neiekļauj atkalizmantojamību. Tomēr paštestēšanas rīks daļēji atbalsta atkalizmantojamību, ļaujot veidot testus, kuriem testa punktu XML fails satur saiti uz citu testa punktu XML failu. Šādā veidā, piemēram, tiek panākts, ka, veidojot jaunu testu, nav nepieciešams katru reizi ierakstīt darbības, kas saistītas ar lietotāja pieslēgšanos testējamajai sistēmai. Tā vietā var kā saiti iekļaut jau gatavu testa punktu XML failu, kas satur tikai tās komandas, kas nepieciešamas, lai pieslēgtos sistēmai.

Pagaidām paštestēšanas pieeja neiekļauj tādus pasaulē plaši izplatītus testu automatizēšanas ietvarus kā datu un atslēgvārdu vadītas testēšanas ietvarus, tādēļ viens no šī darba mērķiem ir apskatīt, kā šie ietvari realizēti citos rīkos, un novērtēt iespējas un nepieciešamību tos ieviest paštestēšanas rīkā.

3.5. Testēšanas līmeņi, funkcionālā testēšana

Tā kā testējamajā sistēmā ir iespējams iebūvēt testa punktus, ar kuru palīdzību var testēt gan sistēmas ārējo struktūru, gan arī iekšējo, paštestēšana atbalsta četrus dažādus testēšanas līmeņus un funkcionālo testēšanu.

Vienībtestēšana. Paštestēšanas pieeja nosaka testa punktu iebūvi testējamajās sistēmas programmatūras kodā. Līdz ar to testēšanas rīkam ir iespēja piekļūt testējamās sistēmas iekšējai struktūrai un veikt dažādu vienumu testēšanu.

Integrācijas testēšana. Pēc katras testējamās sistēmas komponentes pievienošanas jebkurā brīdī iespējams izpildīt uzkrātos paštestēšanas testus.

Regresa testēšana. Paštestēšanas pieejas viens no pamatnosacījumiem ir spēja automātiski, vairākkārtīgi izpildīt uzkrātos testus, izmantojot testējamajā sistēmā iebūvētos testa punktus.

Akcepttestēšana. Akcepttestēšanu paštestēšanas pieejā iespējams nodrošināt, apvienojot regresa un funkcionālās testēšanas īpašības. Attiecībā uz akcepttestēšanu paštestēšanas pieeja piedāvā iespēju sistēmas lietotājiem pašiem veidot akcepttestus, vienkārši darbinot testējamo sistēmu un neizmantojot papildus testēšanas rīkus. Šādā veidā tiek ietaupīts laiks, kas jāvelta izstrādātājiem, automatizējot pasūtītāja noteiktos akceptēšanas kritērijus.

Funkcionālā testēšana. Paštestēšanas pieeja atbalsta funkcionālo testēšanu, testa punktu XML failā piereģistrējot testējamajai sistēmai padotos ievaddatus un saņemtos izvaddatus.

3.6. Piedāvāto iespēju kopsavilkums

Paštestēšanas pieejas izmantošana testēšanas projektā piedāvā šādas iespējas:

- .NET atbalsts;
- Windows lietojumprogrammu testēšana;
- Testēšana produkcijas vidē;
- Datu bāzes testēšana;
- Sistēmas lietotājs bez padziļinātām IT zināšanām pats var veidot testus;
- Vienlaicīga vairāku testu izpilde;
- Iespējama paralēlu darbību veikšana;
- Zema un augsta līmeņa testa punkti, kas iekļauj kontrolpunktus (*checkpoints*);
- Iespējami ātra testu izpilde, jo rīks spēj noteikt komandas izpildes beigas;
- Testēšana pēc baltās kastes metodes;
- Vienkārša rīka uzbūve un rīka funkcionalitātes apgūšana.

3.7. Secinājumi

Paštestēšanas pieeja paredz iespēju programmatūrai pirms darbības uzsākšanas sevi automātiski pārbaudīt: pirms sistēmas lietošanas automātiski tiek pārbaudīts, vai sistēma nesatur kļūdas, kas traucē sistēmas lietošanu.

Būtiska loma paštestēšanas pieejas realizācijā ir testa punktiem, kas tiek iebūvēti pašā sistēmā un nodrošina lietotāju veikto darbību un ierakstīto testa piemēru izpildes reģistrēšanu testa piemēra failā. Testa punktu iebūvēšana sistēmā nodrošina to, ka sistēma tiek testēta no sistēmas "iekšpuses", nevis no "ārpuses" kā to veic trešo kompāniju izstrādātie testēšanas rīki. Iebūvējot testēšanas darbības pašā sistēmā, ir iespējams notestēt sistēmas funkcionalitāti, ko nespēj notestēt rīki, kas veic sistēmas testēšanu no ārpuses.

Paštestēšanas pieeja turpina savu attīstību un ar katru dienu tā tiek pilnveidota. Pašlaik, izstrādājot paštestēšanas rīku, šī pieeja tiek izmēģināta reālos programmatūras testēšanas projektos, lai novērtētu tās priekšrocības, trūkumus un tālākos attīstības virzienus.

Jau šobrīd ir pierādījies, ka paštestēšanas rīka priekšrocība ir tā piedāvātā ērtā testu ierakstīšana, kurai nav nepieciešams izmantot ārējus testēšanas rīkus. Tā vietā, darbinot testējamo sistēmu tā, kā tas tiek darīts ikdienā. Bez šīs priekšrocības ir arī citas papildiespējas, kuras nepiedāvā citi funkcionālās testēšanas rīki. Viena no tādām ir testēšana produkcijas vidē, vēl viena – pasūtītāja veidoti akcepttesti, kas ļauj ietaupīt laiku, jo pasūtītājs pats var veidot automatizētus akcepttestus. Tāpat paštestēšana piedāvā arī detalizētu datu bāzes testēšanu, testa ierakstīšanas laikā

piereģistrējot visus uz datu bāzi sūtītos vaicājumus un saņemtos datus, kā arī piedāvājot testa punktus, kuri ar vaicājumu palīdzību ļauj noteikt datu bāzes pareizību.

Viens no paštestēšanas pieejas mīnusiem ir tas, ka nepieciešams ieguldīt papildus resursus, lai veiktu testa punktu iebūvēšanu testējamās sistēmas programmatūras kodā, tomēr ilgtermiņā apjomīgās sistēmās tas noteikti atmaksāsies, jo būs mazāks kļūdu skaits, kā arī samazināsies laiks, kas nepieciešams sistēmas testēšanai un kļūdu novēršanai.

Ir virkne iespēju, ko paštestēšanas rīks uz doto brīdi neatbalsta:

- Testējamo tehnoloģiju loks (atbalsta tikai .NET ietvara programmēšanas valodas C#, VB un C++);
- datu un atslēgvārdu vadītas testēšanas ietvarus (atbalsta tikai lineāro testēšanas ietvaru) u.c.

Tā kā paštestēšanas rīks vēl turpina savu attīstību tāpat kā pati paštestēšanas pieeja, tad iepriekš minēto iespēju atbalsts ir rīka attīstības iespējas. Rīka izstrādē primārais uzdevums nebija izveidot universālu testēšanas rīku, kas atbalsta ļoti daudz tehnoloģijas, ietvarus un piedāvā daudz dažādas iespējas, bet gan novērtēt testēšanas pieejas lietderīgumu, priekšrocības, trūkumus un attīstības virzienus. Tādēļ sākumā paštestēšanas rīks veidots, galveno uzmanību pievēršot tieši paštestēšanas koncepcijas implementācijai, nevis dažādām papildiespējām.

Kopumā, lai arī paštestēšanas rīkam vēl ir daudz attīstības iespēju, tas jau pašlaik paceļ programmatūras testēšanu jaunā līmenī, ļaujot sistēmas lietotājiem ērtā veidā pašiem veidot testus, vienlaikus saglabājot iespēju veikt detalizētu sistēmas iekšējās struktūras testēšanu.

4. TESTĒŠANAS RĪKI UN TO SALĪDZINĀJUMS AR PAŠTESTĒŠANU

Mūsdienās pieejams plašs klāsts ar testēšanas rīkiem, kuri paredzēti dažādu testēšanas līmeņu realizēšanai dažādām sistēmām. Izstrādājot savu testēšanas rīku, svarīgi ir novērtēt, kādus testēšanas rīkus piedāvā citi izstrādātāji un kādas ir to priekšrocības un trūkumi salīdzinājumā ar paštestēšanas rīku, tādējādi nosakot iespējamos tālākos rīka attīstības virzienus

Precīzu salīdzināšanas rezultātu iegūšanā būtisks nosacījums ir pareiza apskatāmo rīku izvēle. Ņemot vērā to, ka paštestēšanas rīks tiešā veidā izmanto automatizētas testēšanas principus, salīdzināšanai darbā tika nolemts izvēlēties dažādus automatizētos testēšanas rīkus.

Pēc tam, kad ir izvēlēts testēšanas rīka veids, būtiski ir noteikt, kuri rīki ir populārākie. Šī mērķa izpildīšanai tika izmantots IT un testēšanas jomā autoritāti guvušā uzņēmuma „Automated Testing Institute" viedokli. Uzņēmums kopš 2009. gada maija publicē žurnālu „Automated software testing magazine" [Automated Software Testing Magazine, 2012], kas ir viens no populārākajiem automatizētās testēšanas jomā, un ir izveidojis savu tīmekļa vietni [ATI, 2012a]. Šajā vietnē atrodami pieredzējušu IT profesionāļu raksti par automatizētas testēšanas pieeju, ietvariem un rīkiem, uzskaitīti un īsi aprakstīti 716 dažādi pasaulē pieejamie automatizētas testēšanas rīki. Tāpat aktīvi darbojas arī slēgtais forums, kurā var piereģistrēties tikai pēc uzņēmuma apstiprinājuma (šobrīd tajā reģistrējušies ~8000 dalībnieku).

Katru gadu šī organizācija rīko automatizētai testēšanai veltītu konferenci Verify/ATI [Verify/ATI, 2012], kurā notiek jaunu pieeju un rīku demonstrēšana un apmācība. Kopš 2009. gada ffuzņēmums nosaka industrijas vadošos automatizētas testēšanas rīkus dažādās nominācijās, piešķirot tiem „ATI Automation Honors" balvu [ATI, 2012c]. Šai balvai jebkurš vietnes apmeklētājs var pieteikt tos automatizētas testēšanas rīkus, kuriem nominēšanas periodā ir bijis nozīmīgs laidiens (*release*). Pēc nominēšanas perioda beigām, pārsvarā balstoties uz tīmekļa vietnes reģistrētajiem lietotājiem, tiek atlasīti biežāk nominētie rīki. Pēc tam speciāla komiteja, kas veidota no IT profesionāļiem, veic šo rīku izpēti (informācija par rīkiem tiek iegūta no to oficiālajām tīmekļa vietnēm, dokumentācijas, dažādiem rakstiem, blogiem, forumiem utt.) un sašaurina pieteikto rīku klāstu līdz pieciem finālistiem katrā no šim kategorijām un apakškategorijām (2010. gada kategorijas) [ATI, 2011]:

- Labākais atklātā pirmkoda automatizētas vienībtestēšanas rīks;
    - Apakškategorijas: C++, Java, .NET.
- Labākais atklātā pirmkoda automatizētas funkcionālās testēšanas rīks;
    - Apakškategorijas: tīmeklis, Java, .NET, Flash/Flex.
- Labākais atklātā pirmkoda automatizētas veiktspējas testēšanas rīks;

- Apakškategorijas: tīmeklis, tīmekļa servisi/SOA.

- Labākais komerciālais automatizētas funkcionālās testēšanas rīks;
  - Apakškategorijas: tīmeklis, Java, .NET, Flash/Flex, tīmekļa servisi/SOA.
- Labākais komerciālais automatizētas veiktspējas testēšanas rīks;
  - Apakškategorijas: tīmeklis/HTTPS, tīmekļa servisi/SOA.

Labākie rīki katrā no kategorijām un apakškategorijām tiek noteikti interneta (75%) un speciālās komisijas (25%) balsojumā.

Ņemot vērā, ka neizdevās atrast citu līdzvērtīgu (ar tik daudz atsaucēm, īpaši no testēšanas rīku oficiālajām tīmekļa vietnēm) šāda līmeņa automatizētas testēšanas rīku apbalvošanu un vērā ņemamus testēšanas rīku popularitātes reitingus vai salīdzinājumus, tika izvēlēti rīki, kuri ieguvuši balvu kādā no augstāk minētajām nominācijām.

Turpmākajās apakšnodaļās apskatīti daži paštestēšanas pieejai līdzīgākie laureāti. Katram rīkam novērtēta koncepcija, izmantotie testēšanas ietvari un līmeņi un apkopotas piedāvātās iespējas.

4.1. TestComplete

TestComplete ir SmartBear izstrādātais automatizētas testēšanas rīks, kas nodrošina Windows un tīmekļa lietojumprogrammu testēšanu, ir viens no vadošajiem pasaulē starp funkcionālās testēšanas rīkiem. To pierāda arī fakts, ka šis rīks ir ieguvis „ATI Automation Honors" balvu, kā 2010. gada labākais komerciālais automatizētas funkcionālās testēšanas rīks un to savos testēšanas projektos izmanto tādas pasaulē pazīstamas kompānijās kā Adobe, Corel, Falafel Software, ARUP Laboratories, QlikTech u.c. [SmartBear Software, 2012]

4.1.1. Koncepcija

TestComplete rīks funkcionālo testu veikšanai izmanto atslēgvārdu vadītas testēšanas ietvaru, bet papildus ļauj veidot testus arī ar skriptiem. Kopumā TestComplete darbības koncepcija ir salīdzinoši vienkārša. Kā redzams 4.1. attēlā, šis rīks ar starpprocesu komunikācijas un dažādu iebūvēto palīgrīku palīdzību veic testējamajā sistēmā veikto darbību ierakstīšanu un pēc tam arī izpildīšanu.

4.1. att. **TestComplete koncepcija**

Pēc katra testa izpildes, TestComplete izveido detalizētu pārskatu par testa izpildi, attēlojot katras komandas izpildes rezultātu un atspēlēšanas laikā iegūtos ekrāna attēlus. Šādā veidā TestComplete ļauj uzskatāmā veidā noteikt testēšanas laikā atrastās kļūdas.

4.1.2. Testu automatizēšanas ietvari

TestComplete piedāvā kvalitatīvu automatizētu testēšanu, izmantojot atslēgvārdu un datu vadītas testēšanas ietvarus.

Atslēgvārdu vadītas testēšanas ietvars pilnībā spēj nodrošināt to pamatfunkcionalitāti, kuru piedāvā testa skripti. Protams, sarežģītāku testu gadījumā bez testa skriptiem neiztikt, tādēļ atslēgvārdu režīmā iespējams izsaukt atsevišķus testa skripta fragmentus vai pat visu testu nokonvertēt uz testa skriptu.

Datu vadītas testēšanas ietvars nodrošina iespēju testējamajai sistēmai dinamiski padot ievaddatus, tādējādi padarot testus vispusīgākus. Ievaddati var tikt ielasīti gan no populārākajām datubāzēm, gan arī no izklājlapu, CSV, XML un INI failiem. Lielu ievaddatu kopas izveidei, TestComplete piedāvā savu testa datu ģenerēšanas rīku, kas satur iepriekš definētus likumus, pēc kuriem vadoties, tas uzģenerē populārākos datu veidus.

4.1.3. Testēšanas līmeņi, funkcionālā testēšana un nefunkcionālās testēšanas aspekti

Mūsdienu testēšanas rīki neaprobežojas tikai ar vienu testēšanas līmeņa nodrošināšanu. Parasti rīki nodrošina vienu līdz trīs testēšanas līmeņiem, bet TestComplete piedāvā četrus.

Integrācijas testēšana. Līdzīgi kā daudzi automatizētās testēšanas rīki arī TestComplete ļauj pārliecināties par testējamās sistēmas pareizību pēc katras pievienotās komponentes.

Vienībtestēšana. TestComplete vienībtestēšanu nodrošina trīs dažādos veidos:

1. izpildīt vienībtestus, izmantojot jau gatavu vienībtestēšanas rīku (NUnit, JUnit, DUnit un MSTest) funkcionalitāti;

2. izmantot TestComplete piedāvātās vienībtestēšanas funkcijas, kuras var iekļaut automatizētajos testos;
3. izmantot TestComplete piedāvāto iespēju piekļūt testējamās sistēmas objektu atribūtiem un metodēm, tādējādi savā veidā papildinot jau gatavos skriptus ar vienībtestiem.

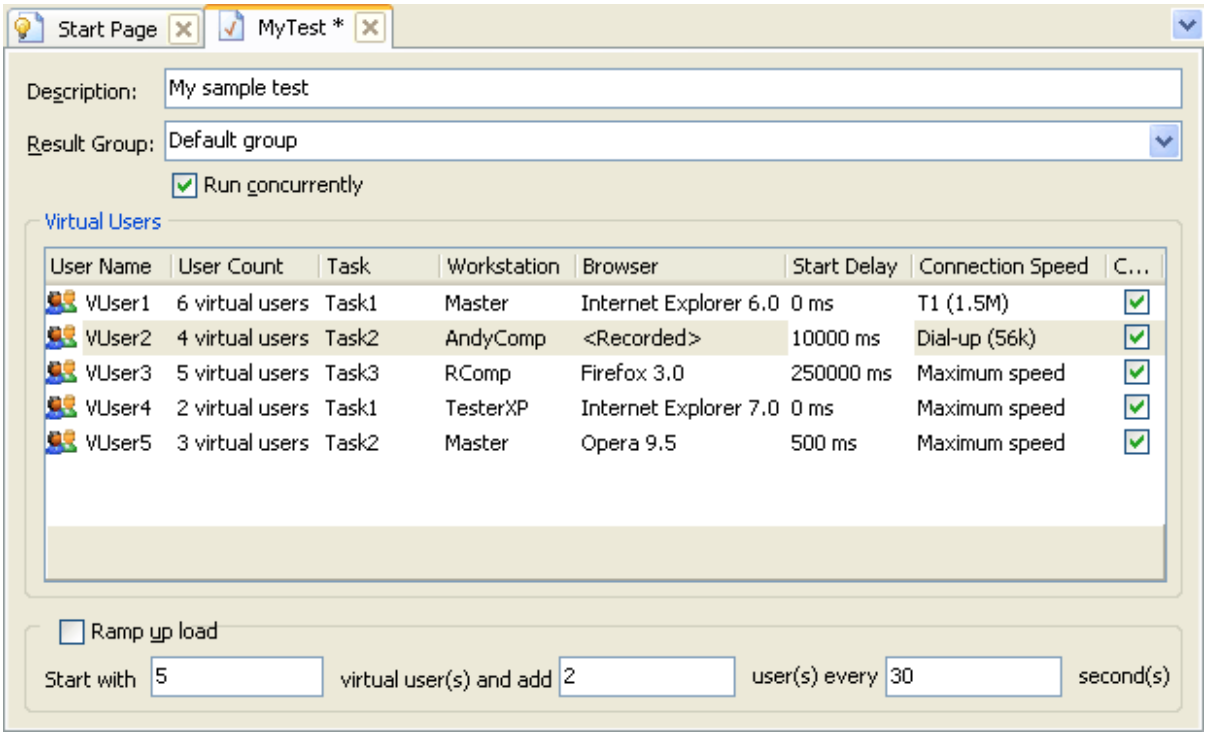
Regresa testēšana. Iekļaujot jebkuru ar rīku izveidoto testu regresa testu kopā, rīks nodrošina to vairākkārtēju izpildīšanu, pārvaldību un rezultātu analīzi.

Akcepttestēšana. Izmantojot funkcionālās testēšanas iespējas klients ar šī rīka palīdzību var automatizēti pārliecināties, vai testējamā sistēma atbilst akceptēšanas kritērijiem.

Funkcionālā testēšana. Bez augstāk norādītiem testēšanas līmeņiem TestComplete nodrošina arī funkcionālo testēšanu. Izmantojot iepriekšminēto atslēgvārdu balstītu vai skriptu testēšanu, rīks nodrošina sistēmas lietotāja saskarnes testēšanu.

TestComplete nodrošina arī divus nefunkcionālās testēšanas aspektus.

Slodzes un stresa testēšana. Tiek nodrošināta iespēja ierakstīt visus HTTP, HTTPS un SOAP pieprasījumus, kas tiek sūtīti uz tīmekļa serveri. Pēc tam šos pieprasījumus iespējams koriģēt un pavairot tā, lai tiktu imitēti vairāki lietotāji. Rīks piedāvā arī slodzes testu pārvaldīšanas un analīzes iespējas (skat. 4.2. attēlu).



4.2. att. **TestComplete slodzes testēšanas vadības logs**

4.1.4. Piedāvāto iespēju kopsavilkums

„SmartBear software" izstrādātais rīks TestComplete piedāvā šādas svarīgas testēšanas iespējas:

- Atslēgvārdu un datu vadītas testēšanas ietvari;
- Skriptēšana testu pilnveidošanai;
- Windows un tīmekļa lietojumprogrammu testēšana;
- .NET, WPF, ASP.NET, Silverlight, XBAP, Adobe AIR, Flex, Flash, Java, JavaFX atbalsts;
- Internet Explorer 8 un Firefox 3.6 atbalsts;
- Windows 7, Vista, XP, 2000, Windows Server 2003 un 2008, Windows Mobile, Pocket PC un emulatoru atbalsts;
- Spraudņi;
- Ērta objektu pievienošana un atjaunošana ar Object Spy;
- Testa datu ģenerēšana.

4.2. FitNesse

FitNesse [FitNesse, 2012] ir atklātā pirmkoda automatizēts akcepttestēšanas rīks, kas ļauj, savstarpēji sadarbojoties testētājiem, izstrādātājiem un pasūtītājiem, veidot testus Wiki vidē [Testing Geek, 2012]. Wiki ir tīmekļa vietņu satura pārvaldības sistēma, kas ļauj veidot jaunas vai rediģēt esošas lapas ar teksta redaktora vai vienkāršotas iezīmēšanas valodas palīdzību [Wikipedia, 2012d].

2010. gadā šis rīks ir ieguvis „ATI Automation Honors" balvu kā labākais atklātā pirmkoda automatizētas funkcionālās testēšanas rīks apakškategorijā „.NET".

FitNesse balstās uz melnās kastes testēšanas principiem un Agile manifestiem:

- Cilvēki un mijiedarbība pāri procesiem un rīkiem;
- Strādājoša programmatūra pāri visaptverošai dokumentācijai;
- Sadarbība ar klientu pāri līgumu saskaņošanai;
- Reaģēšana uz izmaiņām pāri sekošanai plānam.

Šī rīka mērķis ir padarīt akcepttestēšanu automatizētu, viegli veidojamu un lasāmu arī cilvēkiem bez padziļinātām IT zināšanām. Tādēļ savus testus var veidot arī pasūtītājs, jo testu veidošanas princips maksimāli pietuvināts biznesa loģikai. Šāda pasūtītāja iesaistīšanās testēšanas procesā palīdz precīzāk nodefinēt sistēmas prasības un izstrādātājiem ir vieglāk saprast, kas jādara sistēmai.

4.2.1. Koncepcija

FitNesse testēšanas koncepcija balstās uz četrām komponentēm:

1. Wiki lapa, kurā tests izveidots testu tabulas veidā (*Decision table*);
2. Testēšanas sistēma, kas interpretē Wiki lapu;
3. Testa armatūra (*test fixture*), kuru izsauc testēšanas sistēma;
4. Testējamā sistēma, kuru darbina testa armatūra. [Wikipedia, 2012e]

Testu izstrādes procesā no jauna jāveido tikai Wiki lapa un testa armatūra. Pārējo nodrošina FitNesse rīks un testējamā sistēma.

Testu veidošana notiek, izmantojot Wiki lapas. Wiki lapā tiek veidotas testu tabulas, kuras attēlo ieejas un sagaidāmos izejas datus. Atkarībā no izmantotās testēšanas sistēmas FitNesse rīks nodrošina dažādu veidu testu tabulas. Lai attēlotu FitNesse rīka darbības principu, vispirms tiks apskatīta vienkāršākā testu tabula – lēmumu tabula.

Ja nepieciešams izveidot testu, kas notestē klasi, kas veic skaitļu kāpināšanu pakāpē, tad Wiki lapā jāizveido šāda lēmumu tabula:

```
|Kapinasana|
|baze|kapinatajs|rezultats?|
|2|5|32|
|4|2|16|
|1.5|2|2.25|
```

Wiki vide šo tekstu pārvērš vizuāli uzskatāmākā formātā (skat. 4.1. tabulu).

4.1. tabula

FitNesse lēmumu tabula

KapinasanasTests		
baze	**kapinatajs**	**rezultats?**
2	5	32
4	2	16
1.5	2	2.2

Izpildot šo testu, FitNesse šo tabulu nodod norādītajai testēšanas sistēmai, kura to interpretē un izsauc šādu sistēmas izstrādātāja veidotu testa armatūru (šajā piemērā testa armatūra uzrakstīta C# programmēšanas valodā):

```
public class KapinasanasTests
{
    private double _baze;
    private double _kapinatajs;

    public void setBaze(double baze)
    {
        _baze = baze;
    }
```

```
    public void setKapinatajs(double kapinatajs)
    {
        _kapinatajs = kapinatajs;
    }

    public double rezultats()
    {
        return TestejamaSistema.KapinasanasKlase.Kapinat(_baze, _kapinatajs);
    }
}
```

Interpretēšana notiek starp tabulas virsrakstiem un testa armatūras kodu. Lēmumu tabulas pirmā rinda satur testa armatūras klases nosaukumu „KapinasanasTests". Otrajā rindā tiek norādītas testa armatūras klases metodes, kuras tiek izsauktas tādā secībā, kādā nodefinētas tabulas kolonnas. Tām metodēm, aiz kurām seko jautājuma zīme, tiek nolasīta arī atgrieztā vērtība, kura pēc tam tiek salīdzināta ar sagaidāmo vērtību. Pārējās metodes veic ievaddatu uzstādīšanu testa armatūras klasei. Testa izpildes beigās tiek salīdzinātas atgrieztās vērtības ar sagaidāmajām vērtībām un rezultāts attēlots šādā formātā:

4.2. tabula

FitNesse lēmumu tabula pēc testa izpildes

KapinasanasTests		
baze	**kapinatajs**	**rezultats?**
2	5	32
4	2	16
1.5	2	2.2 expected 2.25 actual

Izmantojot šādu koncepciju, sistēmas testēšana tiek vispārināta līdz biznesa loģikai, tādējādi ļaujot šajā procesā iesaistīties pasūtītājiem, kuriem papildus jāpagūst tikai Wiki lapu un testa tabulu veidošanas principi, nevis programmēšanas valoda. Papildus tam Wiki lapa var tikt veidota arī kā akcepttestēšanas dokumentācija, jo testa tabulas uzskatāmā veidā attēlo kritērijus, kuriem jāizpildās, lai izstrādātā sistēma atbilstu pasūtītāja vēlmēm.

4.2.2. Testu automatizēšanas ietvari

FitNesse jau koncepcijas līmenī balstās uz datu un atslēgvārdu vadītas testēšanas ietvariem. Jāatzīmē, ka atšķirībā no citiem automatizētās testēšanas rīkiem, šis nepiedāvā iespēju abus ietvarus izmantot vienlaicīgi. Jāizvēlas vai nu izmantot atslēgvārdu vadītu pieeju ar statiskiem datiem, vai arī datu vadītu pieeju bez atslēgvārdiem.

Atslēgvārdu vadītas testēšanas ietvars realizēts ar SLIM skriptu tabulu un FIT darbību armatūru palīdzību, Kā redzams

4.3. tabulā, katra tabulas rinda apzīmē vienu lietotā darbību. Slīprakstā iezīmētās komandas ir testa armatūrā esošo metožu izsaukumi. Pirms metodes izsaukšanas komandā tiek izņemtas atstarpes un nākamais burts aiz šīs atstarpes tiek pārveidots par lielo (piemēram, komanda „number of login attempts" tiks pārveidota par „numberOfLoginAttempts"). Savukārt pasvītrotās komandas ir FitNesse (precīzāk FIT) iebūvētās komandas, kuras nosaka kāda darbība jāveic ar sekojošajās kolonnās esošajiem datiem.

4.3. tabula

FitNesse skriptu tabula [FitNesse, 2011]

script	*login class*	Valdis	Parole
login			
check	*login message*	Valdis veiksmīgi pieslēdzies	
note	Šis ir komentārs		
show	*number of login attempts*		

Šādā pārskatāmā veidā, neiedziļinoties testējamās sistēmas iekšējā struktūra un izmantojot tikai iebūvētās komandas un testa armatūru, FitNesse nodrošina iespēju ar atslēgvārdiem imitēt lietotāja darbības testējamajā sistēmā.

Datu vadītas testēšanas ietvars realizēts ar SLIM lēmumu tabulu un FIT kolonnu armatūru palīdzību. Kā 4.2. tabulā redzams, tad katra lēmumu tabulas rinda ļauj izpildīt vienu un to pašu testu ar dažādiem ievaddatiem un sagaidāmajiem rezultātiem. Tāpat šīs tabulas iespējams izeksportēt un ieimportēt no izklājlapas, tādējādi atvieglojot testu definēšanu.

4.2.3. Testēšanas līmeņi, funkcionālā testēšana

Tāpat kā daudzi citi testēšanas rīki, arī FitNesse piedāvā vairākus testēšanas līmeņus un funkcionālo testēšanu.

Akcepttestēšana. FitNesse sākotnēji tika izstrādāts, kā akcepttestēšanas rīks, ar kuru iespējams nodefinēt testus pat pirms testējamās sistēmas koda uzrakstīšanas. Šī rīka piemērotību akcepttestēšanai nosaka arī uz tekstu balstītai testu veidošanas princips, kas ļauj iesaistīt arī cilvēkus bez IT zināšanām, bet ar biznesa loģikas zināšanām.

Vienībtestēšana. FitNesse nodrošina arī vienībtestu automatizētu izpildi. Ja testa armatūrā tiek rakstīti vienībtesti, kas testē noteiktu sistēmas moduli un atgriež noteiktu rezultātu, tad FitNesse spēj šos testus ne tikai izsaukt automatizēti, bet arī nodrošina dažādus ievaddatus, rezultātu salīdzināšanu un pārvaldību, ko ne vienmēr piedāvā vienībtestēšanas rīki.

Regresa testēšana. Jebkuru testu vai testu kopu var automātiski vairākkārtīgi izpildīt un pārvaldīt ar Wiki vides palīdzību.

Integrācijas testēšana. Pielāgojot iepriekš aprakstīto testēšanas līmeņu funkcionalitāti, iespējams veikt efektīvu integrācijas testēšanu pēc katras komponentes pievienošanas.

Funkcionālā testēšana. Izmantojot vai nu SLIM skriptu tabulas vai nu FIT darbību armatūras, iespējams veikt sistēmas funkcionālo testēšanu. Tā vietā, lai piefiksētu noteiktus lietotāja saskarnes notikumus, FitNesse armatūrā piedāvā rakstīt metodes, kas izsauc šos notikumus. Pēc tam testa tabulā iespējams rakstīt secīgus metožu izsaukumus, tādējādi imitējot lietotāja darbības ar sistēmu.

4.2.4. Piedāvāto iespēju kopsavilkums

FitNesse rīks piedāvā šādas iespējas:

- Atslēgvārdu un datu vadītas testēšanas ietvari;
- Windows un tīmekļa lietojumprogrammu testēšana;
- .NET, ASP.NET, Java, Delphi, HTML, Ruby, Python, Perl atbalsts;
- Wiki vide;
- Neatkarība no platformas;
- Atklātais pirmkods;
- Spraudņi;
- Lietotāju darbību scenāriju attēlošana neatkarīgi no lietotāja saskarnes;
- Izveidotie testi kalpo kā testēšanas dokumentācija.

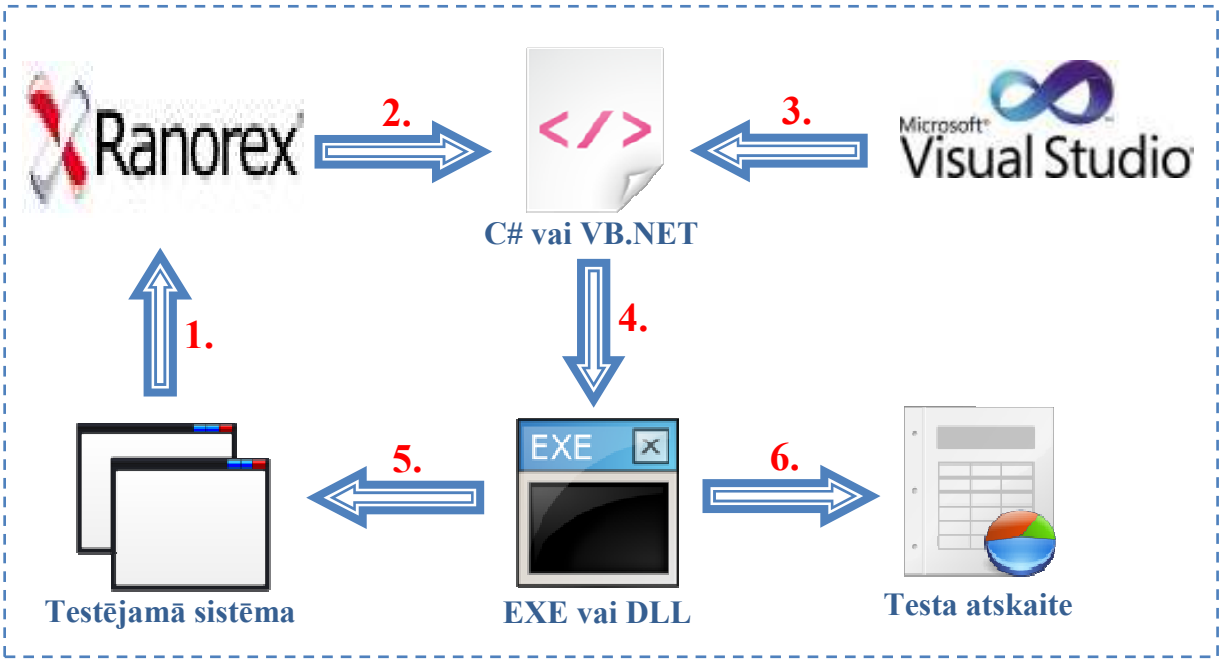
4.3. Ranorex

Ranorex [Ranorex, 2012] ir tipisks grafiskās lietotāja saskarnes testēšanas rīks, kas dod iespēju gan testētājiem, gan izstrādātājiem ātri un viegli veidot jaunus un pārvaldīt esošos funkcionālos testus. Šo rīku atzinīgi ir novērtējusi organizācija „Automated Testing Institute", piešķirot tam balvu kā 2010. gada labākajam komerciālajam automatizētas funkcionālās testēšanas rīkam apakškategorijās „.NET" un „Flash/Flex". Kategorijā kopumā šis rīks ieguva 2. vietu, piekāpjoties iepriekš apskatītajam rīkam TestComplete.

Šī testēšanas rīka klienti ir dažādās pasaulē pazīstamas kompānijas, piemēram, Bosch, General Electrics, FujitsuSiemens, Yahoo, RealVNC u.c.

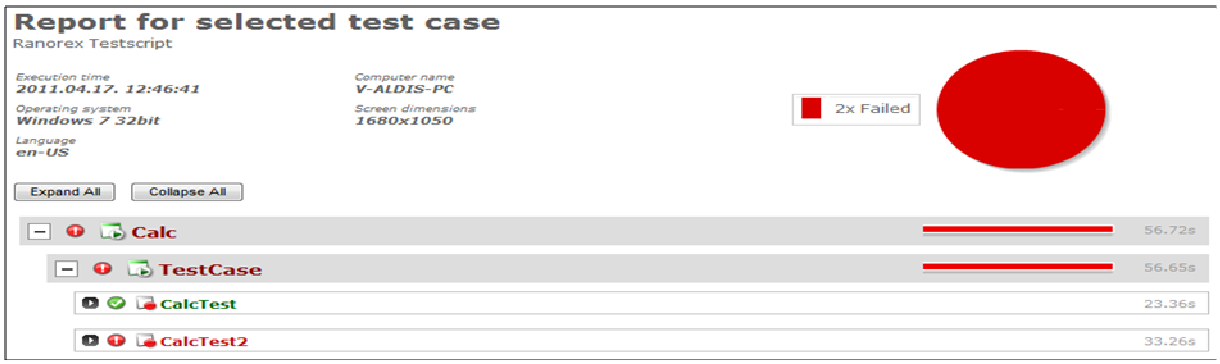
4.3.1. Koncepcija

Kā jau visi populārākie mūsdienu grafiskās lietotāja saskarnes testēšanas rīki, arī Ranorex koncepcija balstās uz atslēgvārdu vadītu testēšanu, piefiksējot objektu, darbību un identifikatoru.



4.3. att. **Ranorex koncepcija**

1. Ranorex ieraksta testējamajā sistēmā veiktās darbības.
2. Ierakstītās darbības Ranorex pārveido par C# vai VB.NET kodu.
3. Lai testēšanu padarītu ērtāku arī izstrādātājiem, izveidoto C# vai VB.NET kodu iespējams rediģēt arī ar Visual Studio izstrādes vidi.
4. No koda tiek uzkompilēts izpildāms fails vai bibliotēka.
5. Uzkompilētā testēšanas „programma" izpilda ierakstītās darbības uz testējamās sistēmas.



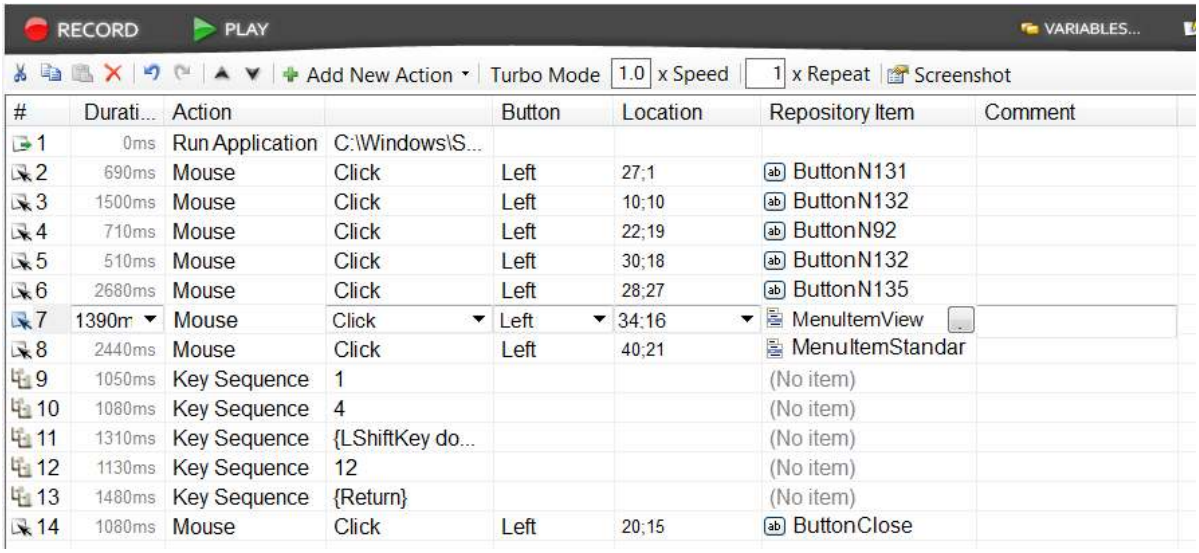
4.4. att. **Ranorex testu kopas atskaite**

6. Pēc testa izpildes tiek uzģenerēta testa atskaite, kas attēlo testa izpildes statusu – veiksmīgs vai neveiksmīgs. Katram testam var aplūkot arī detalizētu informāciju (skat. 4.4. attēlā).

4.3.2. Testu automatizēšanas ietvari

Kā jau vairums mūsdienu automatizētas funkcionālās testēšanas rīki, arī Ranorex iekļauj gan atslēgvārdu, gan datu vadītas testēšanas ietvarus.

Atslēgvārdu vadītas testēšanas ietvars. Testu rediģēšana Ranorex Studio notiek pēc atslēgvārdu vadītas pieejas (skat. 4.5. attēlā), proti, katra komanda sastāv no izpildes laika, darbības, testējamā objekta, ievaddatiem un komentāra. Lai arī šāda pieeja ir pielāgots atslēgvārdu vadītas testēšanas ietvars, tomēr mērķis, veidot testus bez skriptiem, ir panākts.



4.5. att. **Ranorex Studio testu redaktors**

Datu vadītas testēšanas ietvars. Pēc testa ierakstīšanas Ranorex Studio piedāvā ievaddatus pārvērst par mainīgajiem un katram mainīgajam norādīt ievaddatu kopu. Šādā veidā rīks piedāvā izpildīt vienu un to pašu testu ar dažādiem ievaddatiem. Līdzīgi kā citi rīki, kas izmanto datu vadītas testēšanas ietvaru, arī Ranorex piedāvā datu eksportu un importu no izklājlapas.

4.3.3. Testēšanas līmeņi, funkcionālā testēšana

Ranorex nepiedāvā daudz dažādus testēšanas līmeņus, bet tā vietā koncentrējas uz kvalitatīvu funkcionālo un regresa testēšanu.

Regresa testēšana. Rīks piedāvā testus apvienot testu kopās un izpildīt tos vairākkārtīgi, kā arī veikt iegūto rezultātu analīzi.

Integrācijas testēšana. Izmantojot grafiskās lietotāja saskarnes un regresa testēšanas iespējas, Ranorex spēj nodrošināt daļēju integrācijas testēšanu, kuru vajadzētu papildināt arī ar testējamās sistēmas iekšējās struktūras (baltās kastes) testiem.

Akcepttestēšana. Akceptēšanas kritēriju atbilstību klients var pārbaudīt, izmantojot Ranorex piedāvāto automatizēto funkcionālo un regresa testēšanu.

Funkcionālā testēšana. Līdz ar grafiskās lietotāja saskarnes testēšanu šis rīks nodrošina arī efektīvu funkcionālo testēšanu.

4.3.4. Piedāvāto iespēju kopsavilkums

Ranorex rīks piedāvā šādas iespējas [Automatic Testing Frameworks, 2011b]:

- Atslēgvārdu un datu vadītas testēšanas ietvari;
- Windows un tīmekļa lietojumprogrammu testēšana;
- .NET, WPF, Silverlight, Qt, MFC, Delphi, Java SWT, Flash/Flex;
- C#, VB.NET un Python koda ģenerēšana;
- Iespēja izstrādātājiem darboties sev ierastā vidē – Visual Studio;
- Iespējām bagāta izstrādes vide Ranorex Studio;
- Lietotāja saskarnes elementu identificēšana un filtrēšana ar RanoreXPath un repozitorijs.

4.4. T-Plan Robot

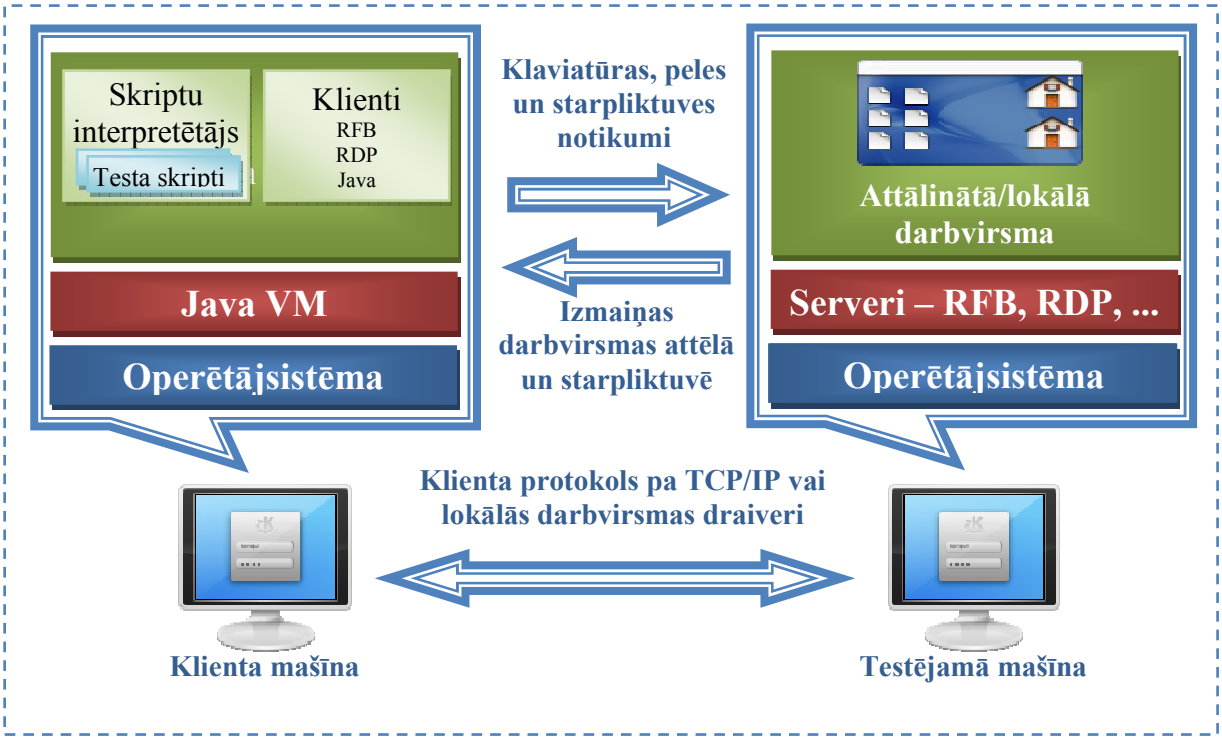
T-Plan Robot [T-Plan, 2012b] ir elastīgs un universāls grafiskās lietotāja saskarnes testēšanas rīks, kas būvēts uz attēlu balstītas testēšanas principiem. Atklātā pirmkoda rīks sistēmas testēšanu veic no lietotāja viedokļa, t.i., vizuālā. Tā kā rīks spēj testēt sistēmas, kuras nevar notestēt uz objektorientētas pieejas balstīti rīki, 2010. gadā tas ieguvis „ATI Automation Honors" balvu kā labākais atklātā pirmkoda automatizētas funkcionālās testēšanas rīks apakškategorijā „Java". Šī uzņēmuma klientu lokā ir šādas kompānijas – Xerox, Philips, Fujitsu-Siemens, Virgin Mobile u.c.

4.4.1. Koncepcija

Atšķirībā no daudziem citiem tipiskiem funkcionālās testēšanas rīkiem, T-Plan Robot neizmanto nedz datu, nedz arī atslēgvārdu vadītu testēšanu. Tā vietā tās izmanto attēlu balstītu testēšanas pieeju.

Šādas pieejas izmantošana T-Plan Robot ļauj būt neatkarīgam no tehnoloģijas, uz kuras būvēta vai uzstādīt testējamā sistēma. Šis rīks var notestēt jebkuru sistēmu, kas tiek attēlota uz operētājsistēmas darbvirsmas.

T-Plan Robot darbojas ar darbvirsmu attēliem, ko saņem no attālināto darbvirsmu tehnoloģijām vai citām tehnoloģijām, kuras rada attēlus. Pagaidām šis rīks atbalsta tikai statisku attēlu testēšanu un RFB protokolu, kas plašāk zināms kā virtuālā tīkla skaitļošana VNC (*Virtual Network Computing*). Nākotnē paredzēts pievienot atbalstu attālinātajam darbvirsmas protokolam RDP (*Remote Desktop Protocol*) un lokālās grafiskās kartes draiverim. [T-Plan, 2012c]



4.6. att. **T-Plan Robot koncepcija** [T-Plan, 2012c]

4.6. attēlā redzams, ka testēšanas process notiek pēc klienta – servera principa. Klients un serveris var sadarboties pa tīklu, izmantojot TCP/IP protokolu, vai arī lokāli, izmantojot darbvirsmas draiveri. T-Plan Robot testēšanas rīks darbojas kā klients, kas uz serveri sūta klaviatūras, peles un starpliktuves notikumus. Savukārt testējamā sistēma atrodas uz servera, kas klientam sūta izmaiņas darbvirsmas attēlā un starpliktuvē. Uz klienta sistēmas uzstādītais T-Plan Robot testēšanas rīks darbojas uz Java virtuālās mašīnas bāzes, tādējādi padarot šo rīku par platformas neatkarīgu.

Testu ierakstīšana notiek, pieslēdzoties attālinātajai darbvirsmai un darbinot testējamo sistēmu. Tajā brīdi T-Plan Robot piefiksē nosūtītos ievaddatus un saņemtās izmaiņas darbvirsmas attēlā vai starpliktuvē un izveido testa skriptu.

Testa atspēlēšanas notiek izpildot testa skriptu, kurā piefiksēti uz testējamo sistēmu nosūtāmie ievaddati. Saņemtās izmaiņas darbvirsmas attēlā un starpliktuvē tiek salīdzinātas ar testa ierakstīšanas brīdi piefiksētajām. Šajā brīdi tiek izmantotas rīka attēlu salīdzināšanas metodes.

Klienta – servera arhitektūra var tikt nodrošināta trīs dažādos veidos [T-Plan, 2012c]:

1. Viena operētājsistēma ar vairākām darbvirsmām – atbalsta tikai Linux/Unix operētājsistēma, jo tā ļauj darbināt vairākus VNC serverus vienlaikus;
2. Viens dators ar vairākām operētājsistēmas instancēm – atbalsta visas operētājsistēmas, jo, izmantojot virtualizācijas tehnoloģijas (piemēram, VirtualBox, VMware u.c.), uz virtuālā datora var uzstādīt VNC serveri;
3. Divi atsevišķi datori – atbalsta visas operētājsistēmas, jo, savienojot datorus tīklā, viens darbojas kā klients, bet otrs kā serveris, uz kura uzstādīta testējamā sistēma.

4.4.2. Testu automatizēšanas ietvari

T-Plan Robot lielākais mīnus ir tas, ka tas neiekļauj nedz datu, nedz arī atslēgvārdu vadītas testēšanas ietvaru. Tā vietā tiek izmantots pats vienkāršākais no ietvariem – lineārais.

Lineārais ietvars. Šajā rīkā visas ierakstītās darbības precīzi atbilst testa skripta komandām. Testa skripts satur statiskus datus, nedodot iespēju testu izpildīt vairākkārtīgi ar dažādiem ievaddatiem un izvaddatiem.

4.4.3. Testēšanas līmeņi, funkcionālā testēšana

No testēšanas līmeņiem T-Plan Robot piedāvā regresa, integrācijas un akcepttestēšanu. Bez zemāk pārskaitītiem testēšanas līmeņiem T-Plan Robot nodrošina arī funkcionālo testēšanu.

Regresa testēšana. Rīks piedāvā testu ierakstīšanu, vairākkārtēju atspēlēšanu, grupēšanu testu kopās, rediģēšanu un iegūto rezultātu analīzi.

Integrācijas testēšana. Ņemot vērā, ka T-Plan Robot piedāvā tikai melnās kastes testēšanu, tad testējamās sistēmas integrācijas testēšanu var veikt tikai daļēji.

Akcepttestēšana. T-Plan Robot savdabīgā veidā ar attēlu balstītu funkcionālo testēšanu, ļauj klientam pārliecināties par testējamās sistēmas atbilstību akceptēšanas kritērijiem.

Funkcionālā testēšana. Izmantojot testējamās sistēmas attēlu balstītas grafiskās lietotāja saskarnes testēšanu, T-Plan Robot nodrošina automatizētu funkcionālo testēšanu.

4.4.4. Piedāvāto iespēju kopsavilkums

T-Plan Robot testēšanas rīks piedāvā šādas iespējas [T-Plan, 2012d]:

- Attēlu balstīta testēšana;

- Jebkuras sistēmas, kura tiek attēlota uz darbvirsmas, testēšana;
- Statisku attēlu testēšana;
- Atklātais pirmkods;
- Testēšana tuvināta lietotāja lietošanas principiem;
- Rīks neietekmē testējamās sistēmas vidi.

4.5. Rational Functional Tester

Uzņēmuma IBM piedāvātais produkts Rational Functional Tester [IBM RFT, 2012a] (turpmāk RFT) ir objektorientētas pieejas automatizēts funkcionālās testēšanas rīks, kas ir viena no sastāvdaļām IBM Rational programmatūras dzīves cikla nodrošināšanas rīku klāstā.

Šis rīks ir viens no populārākajiem testēšanas rīkiem, taču „ATI Automation Honors" balvu nav ieguvis. 2009. un 2010. gadā bija finālists starp labākajiem komerciālajiem automatizētas funkcionālās un veiktspējas testēšanas rīkiem. Tas parāda, ka mūsdienās arvien vairāk parādās jauni efektīvi automatizētas testēšanas rīki, kuriem savas pozīcijas testēšanas rīku tirgū pamazām sāk zaudēt arī Rational Functional Tester.

4.5.1. Koncepcija

IBM RFT ir radīts, lai nodrošinātu automatizētu funkcionālo un regresa testēšanu tiem testētājiem un izstrādātājiem, kuriem nepieciešama augstākās klases Java, .NET un tīmekļa lietojumprogrammu testēšana.

4.7. att. **Rational Functional Tester koncepcija** [IBM, 2011]

RFT rīkam nav pašam savas grafiskās lietotāja saskarnes. Tā vietā RFT izmanto Java lietotājiem pazīstamo izstrādes vidi Eclipse, bet .NET izstrādātājiem – Visual Studio (skat. 4.7. attēlā). Šajās izstrādes RFT uzstādīšanas laikā pievieno papildus funkcionalitāti, kas ļauj ierakstīt, atspēlēt, rediģēt un pārvaldīt testus. [IBM, 2011]

Ierakstot testu, visas darbības, kas notiek testējamajā sistēmā, RFT uzreiz pārveido par Java vai VB.NET testa skriptiem. Ierakstīšanas procesa laikā testētājam pašam jāveido kontrolpunkti, lai RFT piefiksētu sagaidāmo sistēmas stāvokli (piemēram, lauka vērtību, objekta atribūtu, sistēmas ekrāna attēlu utt.) un pēc tam, testu atspēlējot, šo stāvokli salīdzinātu ar pašreizējo.

Atspēlējot testu, RFT izpilda ierakstīšanas laikā uzģenerētos testa skriptus un salīdzina katras izpildītās darbības rezultātu ar ierakstītās darbības rezultātu. Ja testa skriptā ir nodefinēti kontrolpunkti, tad RFT veic verificēšanu attiecībā pret testa skriptā nodefinēto sagaidāmo sistēmas stāvokli. Pēc katra testa skripta izpildes RFT uzģenerē testa rezultātu atainojošu HTML failu (žurnālu), kurā tiek attēlotas visas neatbilstības sagaidāmajam sistēmas stāvoklim.

4.5.2. Testu automatizēšanas ietvari

Tā kā rīkam nav pašam savas grafiskās lietotāja saskarnes, bet gan ar testēšanas funkcionalitāti tiek papildināta gan Eclipse, gan Visual Studio izstrādes vide, tad tiek izmantots tikai datu vadītas testēšanas ietvars. Atslēgvārdu vadītas testēšanas nodrošināšanai IBM piedāvā dažādus sarežģītus pagaidu risinājumus, bet pagaidām vēl nav izstrādāts risinājums, kurš ļautu testētājam bez skriptu palīdzības veidot automatizētus funkcionālos testus.

Datu vadītas testēšanas ietvars. Testa ierakstītājam iespējams norādīt, ka ierakstīšanas laikā visi ievaddati tiek pārveidoti par parametriem un dati tiek saglabāti izklājlapā. Tāpat arī rediģēšanas režīmā statiskus datus var pārveidot par parametriem un norādīt, ka ievaddati jāņem no izklājlapas. Šādā veidā lietotājs var pievienot papildus ievaddatus un paaugstināt testējamās sistēmas testu pārklājumu, neveicot izmaiņas testā.

4.5.3. Testēšanas līmeņi, funkcionālā testēšana

Testēšanas līmeņu ziņā RFT testēšanas rīks piedāvā regresa testēšanu, akcepttestēšanu un daļēju integrācijas testēšanu, kā arī kvalitatīvu un detalizētu funkcionālo testēšanu.

Regresa testēšana. Rīks piedāvā iespējām bagātu objektu ierakstīšanu, vairākkārtēju atspēlēšanu, objektu rediģēšanu utt.

Integrācijas testēšana. Ņemot vērā, ka funkcionālā un regresa testēšana šajā rīkā nodrošina tikai melnās kastes testēšanas metodi, tad integrācijas testēšana ar šo rīku ir iespējama tikai daļēji, atstājot sistēmas iekšējās struktūras testēšanu citu rīku pārziņā.

Akcepttestēšana. Funkcionālās un regresa testēšanas sniegto iespēju apvienojumā, RFT dod iespēju veikt akcepttestēšanu.

Funkcionālā testēšana. RFT rīks funkcionālo testēšanu nodrošina, izmantojot objektorientētas pieejas funkcionālo testēšanu, kas neaprobežojas ar grafiskās saskarnes elementu virspusēju testēšanu, bet gan paplašina to līdz objektu atribūtu līmenim.

4.5.4. Piedāvāto iespēju kopsavilkums

IBM RFT rīks piedāvā šādas iespējas [IBM RFT, 2012b]:

- Atslēgvārdu un datu vadītas testēšanas ietvari;
- Windows un tīmekļa lietojumprogrammu testēšana;
- .NET, ASP.NET, Java, HTML, Siebel, SAP, Flex atbalsts;
- Java un VB.NET koda ģenerēšana;
- Nav jāapgūst jauna vide, jo tiek integrēts Eclipse un Visual Studio;
- „Storyboard" testēšanas atbalsts;
- Ērta testējamās sistēmas objektu pievienošana un atjaunošana ar ScriptAssure;
- Integrējams ar citiem IBM Rational programmatūras dzīves cikla pārvaldības rīkiem;
- Iespēja papildināt rīku ar nestandarta vadīklu atbalstu;
- Testa skriptu versiju pārvaldība;
- Uz termināļiem balstītu sistēmu testēšana;

4.6. HP Unified Functional Testing Software

Hewlett Packard piedāvātais rīks HP Unified Functional Testing Software [HP UFTS, 2012] (turpmāk HP UFTS) ir augstas kvalitātes automatizētas funkcionālās un regresa testēšanas rīku kopums, kas sastāv no diviem atsevišķiem rīkiem – HP Functional Testing Software (turpmāk HP FTS) un HP Service Test Software (turpmāk HP STS).

HP FTS plašāk pazīstams kā HP QuickTest Professional (agrāk arī kā Mercury QuickTest Professional) jeb HP QTP. HP FTS pamatā ir tas pats HP QTP, bet papildināts ar dažādiem paplašinājumiem. Tas ir aizstājējs agrāk populāram rīkam Mercury WinRunner, kuru 2006. gadā pārpirka HP. Tā kā šī rīka liela daļa funkcionalitātes pārklājās ar (vai tika pārnesta uz) HP FTS, tad 2008. gadā tika nolemts pārtraukt šī rīka atbalstīšanu un ieteikts līdz tam ierakstītos testus pārnest uz HP FTS. Savukārt HP STS ir HP pašu izstrādāts rīks, kas nodrošina automatizētu funkcionālu servisu testēšanu ar aktivitāšu diagrammu palīdzību. [HP UFTS, 2011a]

Apvienojot šos rīkus kopā, HP ieguva vienu no pasaulē pieprasītākajiem funkcionālās un regresa testēšanas rīkiem, kas atzinību guvis arī 2009. gada „ATI Automation Honors" balvu

pasniegšanas ceremonijā kā labākais komerciālais automatizētas funkcionālās testēšanas rīks, bet 2010. gadā – iekļuvis šīs kategorijas četru finālistu vidū.

4.6.1. Koncepcija

HP UFTS ir tipisks atslēgvārdu un datu vadītas testēšanas rīks, kas vienlaikus gan atvieglo testu veidošanu, rediģēšanu, gan nodrošina plašu testējamās sistēmas testu pārklājumu.

Lai veiktu pilnvērtīgu testējamās sistēmas funkcionālo testēšanu, nepietiek ar grafiskās lietotāja saskarnes testēšanu, jo sistēmas funkcionalitāte neaprobežojas tikai ar vizuālo funkcionalitāti. Liela daļa funkcionalitātes „paslēpta" zem grafiskās lietotāja saskarnes komponenšu līmenī. Īpaši tas ir izteikts sistēmām, kas darbojas pēc principa „klients – serveris". Šādas sistēmas tiek sauktas par daudzlīmeņu sistēmām.

Lai nodrošinātu arī šādu sistēmu testēšanu, HP UFTS balstās uz daudzlīmeņu testēšanas koncepciju (skat. 4.8. attēlā). Daudzlīmeņu sistēmu testēšanu c iedala trīs līmeņos [HP UFTS, 2011b]:

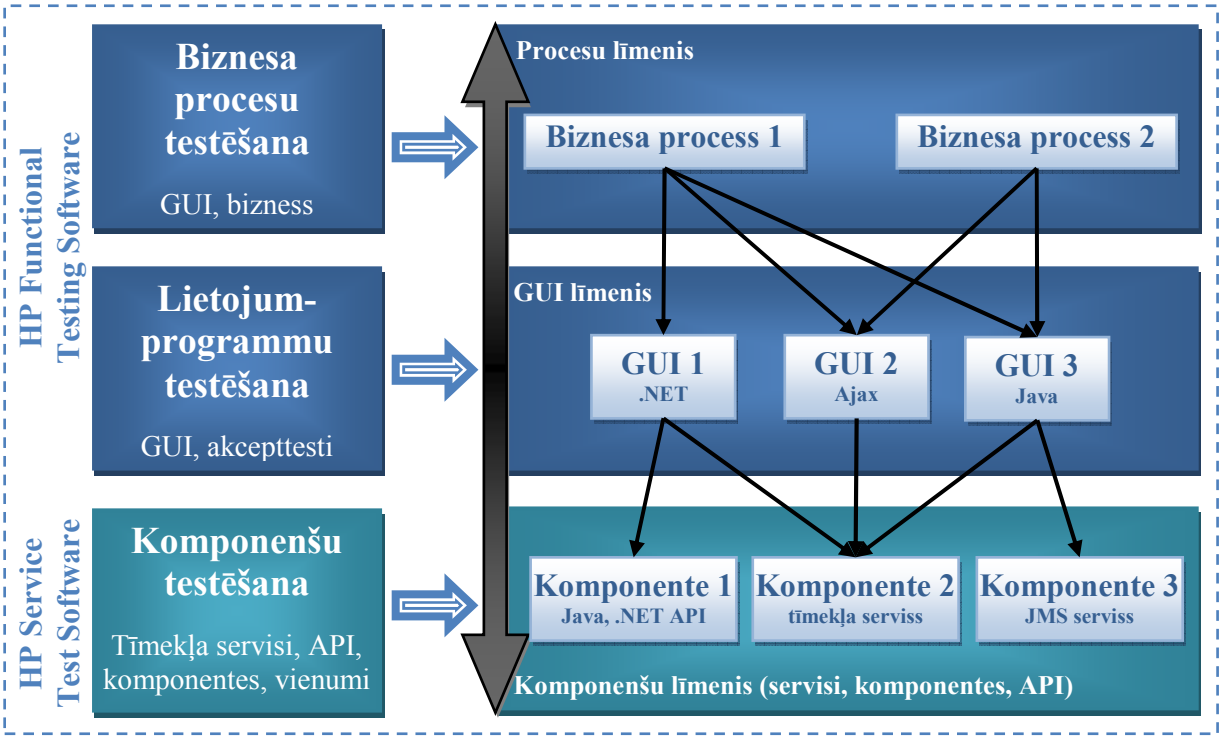
- grafiskās lietotāja saskarnes testēšana;
- servisu un komponenšu testēšana;
- daudzlīmeņu testēšana.

Grafiskās lietotāja saskarnes testēšanu nodrošina HP FTS, kas to sadala divos līmeņos – biznesa procesu testēšana un lietojumprogrammu testēšana. Biznesa procesu testēšanu iespējams veikt, pateicoties testu ierakstīšanas un atspēlēšanas funkcionalitātei un atslēgvārdu vadītai testēšanai, kas būtiski atvieglo testu veidošanu un rediģēšanu un padara to tuvāku biznesa loģikai. Savukārt, lai nodrošinātu kvalitatīvu lietojumprogrammu testēšanu, HP FTS ar integrēto skriptu veidošanas un atkļūdošanas vidi piedāvā arī pilnu piekļuvi grafiskās lietotāja saskarnes objektiem un to īpašībām. Tādējādi HP FTS vienlaikus var testēt gan procesu, gan GUI līmeni.

Servisu un komponenšu testēšanu nodrošina HP STS, kurš ērtā veidā ļauj veidot automatizētus funkcionālos testus testējamās sistēmas neredzamajai daļai. Testu veidošana notiek ar aktivitāšu diagrammu palīdzību un tikai sarežģītāku testu gadījumā nepieciešama programmēšana, tādējādi ļaujot veidot testus lietotājiem bez programmēšanas zināšanām.

Daudzlīmeņu testēšanu nodrošina HP UFTS, kas vienotā risinājumā apvieno HP FTS un HP STS. HP UFTS piedāvā testēt transakcijas, kas apvieno kopā daudzlīmeņu testējamās sistēmas līmeņus. Šādā veidā vienā testa scenārijā iespējams veikt gan grafiskās lietotāja saskarnes, gan servisu un komponenšu testēšanu.

Testu rezultāti tiek uzģenerēti pēc katra testa izpildīšanas visos trīs testēšanas līmeņos. Gan HP FTS, gan HP STS testu rezultātus attēlo divos līmeņos. Pirmais līmenis vispārīgi attēlo testa rezultātus un statistiku, bet otrais līmenis piedāvā detalizētu katras komandas izpildes rezultātu un kļūdas gadījumā izsmeļošu aprakstu. Savukārt, HP UFTS testu rezultātus apvieno vienotā atskaitē, kur katra komanda sadalīta vai nu HP FTS, vai nu HP STS aktivitātēs.



4.8. att. **HP UFTS daudzlīmeņu testēšanas koncepcija** [HP UFTS, 2011b]

Kā redzams 4.8. attēlā tad HP FTS nodrošina gan biznesa procesu, gan lietojumprogrammu testēšanu, testējamajā sistēmā ļaujot notestēt attiecīgi procesu un GUI līmeņus. Katrs biznesa process tiek nodrošināts ar vienu vai vairākām grafiskajām lietotāja saskarnēm dažādās realizācijās.

Grafiskā lietotāja saskarne ir redzamā testējamās sistēmas daļa, kas veic dažādu komponenšu un servisu izsaukumus un saņem no tiem rezultātus, ko attēlot lietotājam. Kā noprotams, tad liela daļa sistēmas funkcionalitātes atrodas sistēmas komponenšu līmenī, tādēļ HP STS piedāvā dažādu servisu, lietojumprogrammu saskarņu, komponenšu un vienumu funkcionālo testēšanu.

HP FTS no testēšanas koncepcijas viedokļa nepiedāvā jaunu pieeju, tomēr apvienojumā ar HP STS, šis rīks spēj piedāvāt atšķirīgu, daudzpusīgu funkcionālo testēšanu trīs līmeņos. Šāda daudzlīmeņu funkcionālā testēšana ļauj veikt testēšanu pirms grafiskās lietotāja saskarnes izstrādes, tādējādi paātrinot sistēmas izstrādi un paaugstinot komponenšu un servisu kvalitāti. Līdz ar to tiek paaugstināta arī vispārējā grafiskās lietotāja saskarnes kvalitāte.

4.6.2. Testu automatizēšanas ietvari

HP UFTS iekļauj gan atslēgvārdu vadītu, gan datu vadītu testēšanas ietvaru koncepciju.

Atslēgvārdu vadītas testēšanas ietvars HP FTS rīkā realizēts tipiskā veidā ar grafiskās lietotāja saskarnes palīdzību, tādējādi ļaujot testus veidot bez programmatūras koda rakstīšanas. Katra testa komanda sastāv no darbības, objekta, ar kuru izpildīt darbību, un datiem, kuri nepieciešami darbības izpildei.

Datu vadītas testēšanas ietvars realizēts gan HP FTS, gan HP STS līdzīgā veidā ar izklājlapu palīdzību. Izklājlapā katra kolonna atbilst noteiktam parametram, bet rindas satur ievaddatus. Tests tiek izpildīts tik reizes, cik rindas nodefinētas izklājlapā. Testu izstrādes vidē ieintegrēta izklājlapa ļauj daudz ērtākā veidā pārvaldīt šo datu vadīto testēšanu.

4.6.3. Testēšanas līmeņi, funkcionālā testēšana

Apvienojot kopā vienotā risinājumā gan HP FTS, gan HP STS, HP UFTS nodrošina funkcionālo, regresa, integrācijas un akcepttestēšanu.

Regresa testēšana. Gan HP FTS, gan HP STS nodrošina testu ierakstīšanu, vairākkārtēju atspēlēšanu, objektu piesaisti, rediģēšanu un citas regresa testēšanai raksturīgas darbības.

Integrācijas testēšana. Gan HP FTS, gan HP STS nodrošina testu pārvaldīšanu, ļaujot izveidotos testus izpildīt pēc katras integrācijas, pārliecinoties, vai pievienotā komponente nav radījusi kļūdas citās komponentēs.

Akcepttestēšana. HP UFTS nodrošina iespēju cilvēkiem bez padziļinātām zināšanām IT jomā veidot un izpildīt sarežģītus testus gan grafiskajai lietotāja saskarnei, gan servisiem un komponentēm, tādējādi ļaujot pārliecināties, ka sistēma atbilst akceptēšanas kritērijiem.

Funkcionālā testēšana. HP UFTS nodrošina kvalitatīvu daudzlīmeņu funkcionālo testēšanu, to sadalot divās daļās. Grafiskās lietotāja saskarnes testēšanu veic HP FTS, bet servisu un komponenšu testēšanu veic HP STS.

4.6.4. Piedāvāto iespēju kopsavilkums

HP UFTS rīks piedāvā šādas iespējas [HP STS, 2011]:

- Atslēgvārdu un datu vadītas testēšanas ietvari;
- Windows un tīmekļa lietojumprogrammu testēšana;
- Servisu testēšana;
- Daudzlīmeņu lietojumprogrammu testēšana;
- .NET, ASP.NET, Java, Delphi, Flex, Silverlight, HTML, SAP atbalsts;

- WSDL, JMS, HTTP, RESt atbalsts;
- VBScript un C# skriptu valodas;
- Ērta objektu pievienošana un atjaunošana ar Object Spy;
- Ērta, intuitīva testu veidošana ar grafiskās lietotāja saskarnes palīdzību bez programmēšanas;
- Iespēja papildināt rīka funkcionalitāti, izmantojot lietojumprogrammas saskarni;
- Integrējams ar citiem HP programmatūras dzīves cikla pārvaldības rīkiem.

4.7. Selenium

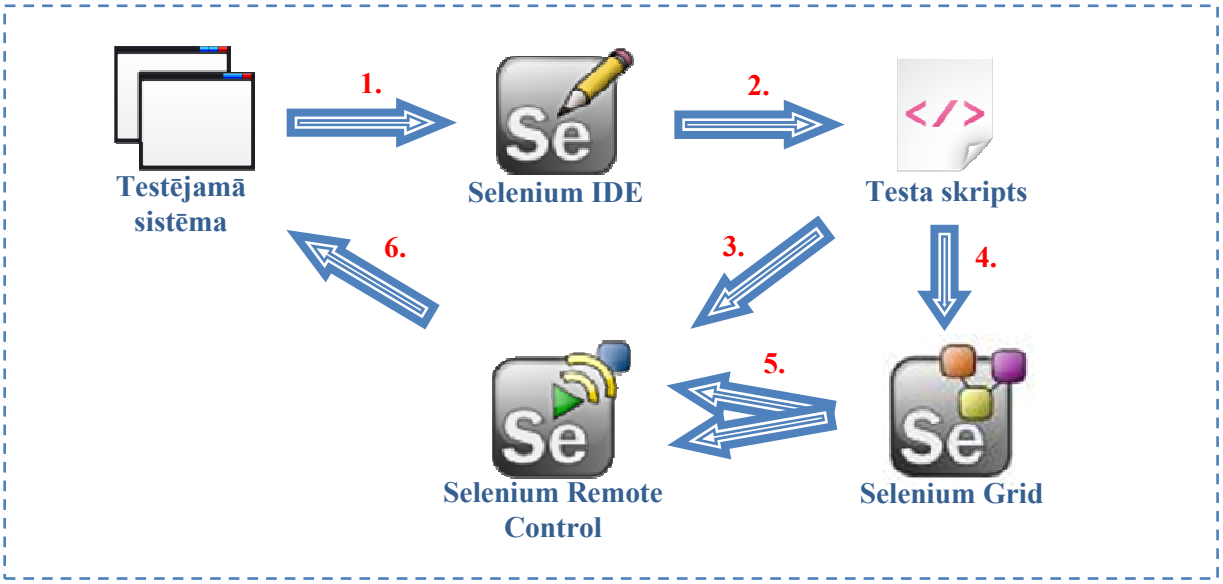
Selenium [SeleniumHQ, 2012] ir OpenQA izstrādāts atklātā pirmkoda tīmekļa lietojumprogrammu testēšanas ietvars, kas sastāv no vairākiem testēšanas rīkiem. Tīmekļa lietojumprogrammu testēšanas jomā šis ietvars jau vairāk kā piecus gadus ir stabils līderis un pēdējos divos gados ieguvis „ATI Automation Honors" balvu kā labākais atklātā pirmkoda automatizētas funkcionālās testēšanas rīks. Būtisks šī rīka attīstības faktors ir tas, ka to savos testēšanas projektos izmanto tādas IT kompānijas kā Google, Mozilla, LinkedIn u.c.

4.7.1. Koncepcija

Selenium ietvars sastāv no trim dažādiem rīkiem [SeleniumHQ, 2012]:

- **Selenium IDE** – Selenium skriptu izstrādes vide, kas ļauj ierakstīt, atspēlēt, rediģēt un atkļūdot testus.
- **Selenium Remote Control** – pamatmodulis, kas ļauj rakstīt testus dažādās programmēšanas valodās un tos izpildīt uz jebkuras pārlūkprogrammas.
- **Selenium Grid** – kontrolē vairākus Selenium Remote Control instances tā, lai testus varētu izpildīt vienlaicīgi uz dažādām platformām.

Testu veidošanai Selenium izstrādājis savu programmēšanas valodu Selenese, lai nodrošinātu iespēju rakstīt testus dažādās programmēšanas valodās. Savukārt testu izpildīšanai izmanto tīmekļa serveri, kas darbojas kā starpnieks starp pārlūkprogrammu un tīmekļa pieprasījumiem, šādā veidā nodrošinot pārlūkprogrammu neatkarību.



4.9. att. **Selenium koncepcija** [SeleniumHQ, 2012]

1. Selenium IDE, izmantojot Mozilla Firefox pārlūkprogrammu, ieraksta testējamajā sistēmā veiktās darbības.
2. Visas ierakstītās darbības tiek ierakstītas Selenese programmēšanas valodā, bet nepieciešamības gadījumā, tos iespējams izeksportēt C#, Java, Perl, PHP, Python vai Ruby programmēšanas valodās. Pēc tam kādā no šo programmēšanas valodu izstrādes rīkiem var papildināt esošos vai veidot jaunus testa skriptus.
3. Selenium Remote Control testa skriptā ierakstītās komandas pārveido par tīmekļa pieprasījumiem.
4. Selenium Grid ielasa testa skriptu un izveido vairākas Selenium Remote Control instances.
5. Selenium Grid vienlaicīgi izsauc izveidotās Selenium Remote Control instances, padodot kā parametru ielasīto testa skriptu.
6. Selenium Remote Control pārsūta uz testējamo sistēmu tīmekļa pieprasījumus, kuri atbilst testa skriptā nodefinētajām komandām.

Jāpiebilst, ka Selenium ietvara koncepcija nenosaka 4.9. attēlā redzamo secību. Atkarībā no testētāja zināšanām, pieredzes un projekta vajadzībām šī secība var mainīties. Pirmo testu veidošanai nepieredzējis testētājs noteikti izmantos Selenium IDE izstrādes vidi. Tādā gadījumā testēšana notiks, izmantojot pilnu ciklu (scenāriji 4.9. attēlā – 1., 2., 3., 6. vai 1., 2., 4., 5., 6.). Sarežģītāku testu veidošanai pieredzējis testētājs parasti izlaiž pirmos divus soļus, jo testu veidošana notiek kādā no Selenium atbalstīto programmēšanas valodu izstrādes vidēm un izpildīšanu nodrošina Selenium Remote Control un Selenium Grid (scenāriji 4.9. attēlā – 3., 6. vai 4., 5., 6.).

4.7.2. Testu automatizēšanas ietvari

Selenium tikai daļēji iekļauj gan datu, gan arī atslēgvārdu vadītas testēšanas ietvarus.

Datu vadītas testēšanas ietvaru Selenium iekļauj daļēji, jo tam nepieciešami papildus rīki un konfigurācija, bet rīka koncepcijā tas nav ietverts.

Atslēgvārdu vadītas testēšanas ietvars tiek nodrošināts ar tīmekļa pārlūkprogrammas Mozilla Firefox paplašinājuma palīdzību.

4.7.3. Testēšanas līmeņi, funkcionālā testēšana

Lai gan nereti Selenium tiek uzskatīts par vienībtestēšanas rīku, tas nav domāts vienībtestēšanai. Selenium ir vadošais tīmekļa lietojumprogrammu funkcionālās un regresa testēšanas rīks. Papildus ar to iespējams veikt arī testējamās sistēmas integrācijas testēšanu un akcepttestēšanu.

Regresa testēšana. Testu ierakstīšanu Selenium nodrošina tikai ar Mozilla Firefox paplašinājuma Selenium IDE palīdzību, bet to vairākkārtēju atspēlēšanu dažādās vidēs nodrošina gan Selenium RC, gan Selenium Grid, gan Selenium Core, gan arī savā testēšanas projektā iekļauti testu skripti.

Integrācijas testēšana. Līdzīgi kā lielākā daļa automatizētās funkcionālās testēšanas rīki, arī Selenium piedāvā daļēju integrācijas testēšanu pēc jaunu komponenšu pievienošanas.

Akcepttestēšana. Izmantojot funkcionālās un regresa testēšanas funkcionalitāti ar Selenium iespējams pārbaudīt testējamās sistēmas atbilstību prasībās noteiktajiem akceptēšanas kritērijiem.

Funkcionālā testēšana. Selenium funkcionālos tests veic ar HTTP pieprasījumu un tīmekļa pārlūkprogrammu palīdzību.

4.7.4. Piedāvāto iespēju kopsavilkums

Selenium ietvars piedāvā šādas iespējas:

- Iespējām plašu tīmekļa lietojumprogrammu testēšana;
- Java, HTML, Flash, Flex, Silverlight, HTML, Ruby atbalsts
- Testu veidošana C#, Java, Perl, PHP, Python un Ruby programmēšanas valodās;
- Programmēšanas valoda Selenese;
- Platformu un tīmekļa pārlūkprogrammu neatkarība;
- Testu atkļūdošana;
- Atklātais pirmkods;
- Daudz dažādu spraudņu un papildinājumu.

4.8. Paštestēšanas rīka salīdzinājums ar testēšanas rīkiem

Darba mērķis ir novērtēt, kādas ir paštestēšanas rīka piedāvātās iespējas salīdzinājumā ar citiem rīkiem, un noskaidrot tālākos attīstības virzienus, tādēļ iepriekšējās nodaļās tika detalizēti apskatīti paštestēšanas rīks un dažādi automatizētas testēšanas rīki. Tomēr no apjomīgiem rīku aprakstiem grūti noteikt, kādas ir tā priekšrocības un trūkumi salīdzinājumā ar citiem. Tāpat krietni grūtāk ir novērtēt, kurš rīks būtu piemērotāks noteiktam testēšanas projektam. Šī iemesla dēļ svarīgi ir šos apskatītos rīkus salīdzināt pēc noteiktiem kritērijiem. Tādēļ nākamajās apakšnodaļās tiks aprakstīti izvirzītie rīku salīdzināšanas kritēriji un, vadoties pēc tiem, veikta rīku salīdzināšana.

4.8.1. Salīdzināšanas kritēriji

Salīdzināšanas kritēriji lielākoties tika izvirzīti, vadoties pēc apskatīto septiņu rīku piedāvātajām iespējām. Ja kāds apskatītais rīks piedāvā jaunu, būtisku iespēju, tad tā, kā atsevišķs kritērijs, tiek pievienota pie kopīgiem rīku salīdzināšanas kritērijiem. Kopumā, izvēloties salīdzināšanas kritērijus, tika ņemts vērā:

1. svarīgākās testēšanas īpašības;
2. automatizētas testēšanas rīku svarīgākās īpašības;
3. salīdzināmo rīku piedāvātās iespējas.

Nākamajā 4.4. tabulā ir uzskaitīti testēšanas rīku salīdzināšanas kritēriji un atbilstošā kritērija jautājums, uz kuru tiek sniegta atbilde kopējās rīku salīdzināšanas tabulās (skat. 4.5. un 4.6. tabulās). Paštestēšanas un darbā aplūkoto rīku salīdzināšanai izmantoti 4.4. tabulā aprakstītie kritēriji.

4.4. tabula

Testēšanas rīku salīdzināšanas kritēriji

Salīdzināšanas kritērijs	Jautājums
Testēšanas metode (TM)	Kuras testēšanas metodes atbalsta?
Testu automatizēšanas pieeja (TAP)	Kuras testu automatizēšanas pieejas izmanto?
Testu automatizēšanas ietvars (TAI)	Kurus testu automatizēšanas ietvarus izmanto?
Testēšanas līmenis	Kurus testēšanas līmeņus atbalsta?
Funkcionālā testēšana	Vai atbalsta funkcionālo testēšanu?
Nefunkcionālā testēšana	Kurus nefunkcionālos testēšanas aspektus atbalsta?
Platforma	Kuras operētājsistēmas atbalsta?
Testējamā tehnoloģija	Kuras tehnoloģijas (parasti programmēšanas valodas) atbalsta?
Testu ierakstīšana un atspēlēšana	Vai nodrošina testu ierakstīšanu un automatizētu vairākkārtēju atspēlēšanu?
Darbvirsmas	Vai nodrošina darbvirsmas lietojumprogrammu testēšanu?

lietojumprogrammu testēšana	
Tīmekļa lietojumprogrammu testēšana	Vai nodrošina tīmekļa lietojumprogrammu testēšanu?
Servisu testēšana	Vai nodrošina servisu testēšanu?
Datu bāzes testēšana	Vai iespējams atsevišķi testēt tikai sistēmas datu bāzi?
Testēšana produkcijas vidē	Vai nodrošina testēšanu produkcijas vidē?
Sistēmas lietotājs var veidot testus	Vai lietotājs bez padziļinātām IT zināšanām var veidot testus?
Vienlaicīga vairāku testu izpilde	Vai nodrošina paralēlu testu izpildi?
Paralēlu darbību veikšana	Vai testu izpildes laikā, veicot paralēlas darbības, netiek izjaukta testa izpilde?
Testējamā objekta noteikšana	Vai spēj atšķirt testējamo objektu no citiem operētājsistēmas objektiem?
Testa rezultātu analīze	Vai pēc testa izpildes piedāvā testu rezultātu analīzi?
Testu rediģēšana	Vai tiek piedāvāts izveidoto testu redaktors?
Ekrāna attēli	Vai testa ierakstīšanas/atspēlēšanas laikā tiek iegūti testējamās sistēmas ekrāna attēli?
Kontrolpunkti	Vai tiek piedāvāti kontrolpunkti?
Objektu validēšana	Vai tiek nodrošināta objektu validēšana, ja notikušas izmaiņas testējamajā sistēmā?
Objektu pārlūks	Vai tiek piedāvāts testējamās sistēmas objektu pārlūks/redaktors?
Testu žurnāls (log)	Vai tiek veidots testa izpildes žurnāls?
Testu izpildes laiku plānotājs	Vai iespējams norādīt laiku, kad uzsākt testu izpildi, piemēram, naktī?
Komandas izpildes beigu noteikšana	Vai rīks spēj noteikt, kad beigusies iepriekšējās komandas izpilde (šim nolūkam netiek izmantots noteikts gaidīšanas laiks)?
Spraudņi un paplašinājumi	Vai nodrošina iespēju veidot savus spraudņus un paplašinājumus, rīka funkcionalitātes papildināšanai?
Izstrādātājs	Kura kompānija vai persona izstrādājusi (pašlaik pieder) testēšanas rīks?
Cena	Cik maksā rīks?
Lietošanas ērtums (1 – 5)	Vērtību skalā no 1 (ļoti neērts) līdz 5 (ļoti ērts), cik ērta ir testu veidošana (subjektīvs viedoklis)?
Rīka programmēšanas valoda	Kurās programmēšanas valodās iespējams veidot testus?
Klients	Kuras kompānijas izmanto rīku savos testēšanas projektos?

4.8.2. Salīdzināšanas rezultāti

Katra izvirzītā salīdzināšanas kritērija novērtēšanai tika izmantota rīka oficiālajā mājas lapā un dažādās uzticamās tīmekļa vietnēs pieejamo informāciju, specifikāciju aprakstus un palīdzības logus, kā arī, praktiski izmēģinot testēšanas rīku, pārliecinājās par šo kritēriju atbilstību īstenībai. Liela daļa testēšanas rīku piedāvātās iespējas aprakstītas iepriekšējās nodaļās.

Testēšanas rīku salīdzināšanas rezultātu objektivitātes palielināšanai maksimāli ir mēģināts izvairīties no subjektīvā viedokļa, cenšoties pēc iespējas vairāk salīdzināšanu veikt pēc rīka piedāvāto iespēju esamības vai neesamības principa. Tādēļ 4.5. un 4.6. tabulās redzamajā salīdzinājumā kritēriji novērtēti ar šādām trīs atbildēm:

- Ir – nozīmē, ka attiecīgais rīks atbalsta kritērijā minēto funkcionalitāti;
- Daļēji – nozīmē, ka attiecīgais rīks tikai daļēji atbalsta kritērijā minēto funkcionalitāti;
- Nav – nozīmē, ka attiecīgais rīks neatbalsta kritērijā minēto funkcionalitāti

Darbā izdalītas divas salīdzinājumu tabulas, kur 4.5. tabulā ir apkopota informācija par salīdzināšanas kritērijiem, kurus paštestēšanas rīks atbalsta un 4.6. tabulā, kur apkopota informācija par salīdzināšanas kritērijiem, ko paštestēšana rīks pašreiz neatbalsta.

4.5. tabula

Testēšanas rīku salīdzinājums (paštestēšanas rīks nodrošina)

Kritērijs		Paštestēšanas rīks	TestComplete	FitNesse	Ranorex	T-Plan Robot	Rational Functional Tester	Selenium	HP Unified Functional Testing Software
TM	Melnā kaste	Ir	Ir	Ir	Ir	Ir	Ir	Ir	Ir
	Baltā kaste	Ir	Nav	Nav	Nav	Nav	Nav	Nav	Nav
	Pelēkā kaste	Ir	Ir	Ir	Nav	Nav	Nav	Nav	Nav
TAP	Objektorientēta pieeja	Ir	Ir	Ir	Ir	Nav	Ir	Ir	Ir
TAI	Lineārais ietvars	Ir	Nav	Nav	Nav	Ir	Nav	Nav	Nav
Testēšanas līmenis	Vienībtestēšana	Ir	Ir	Ir	Nav	Nav	Nav	Nav	Nav
	Integrācijas testēšana	Ir	Ir	Ir	Daļēji	Daļēji	Daļēji	Daļēji	Ir
	Regresa testēšana	Ir	Ir	Ir	Ir	Ir	Ir	Ir	Ir
	Akcepttestēšana	Ir	Ir	Ir	Ir	Ir	Ir	Ir	Ir
Funkcionālā testēšana		Ir	Ir	Ir	Ir	Ir	Ir	Ir	Ir
Platforma	Windows 2000	Ir	Ir	Ir	Nav	Ir	Ir	Ir	Ir
	Windows Server 2003 un 2008	Ir	Ir	Ir	Ir	Ir	Ir	Ir	Ir
	Windows XP	Ir	Ir	Ir	Ir	Ir	Ir	Ir	Ir
	Windows Vista	Ir	Ir	Ir	Ir	Ir	Ir	Ir	Ir
	Windows 7	Ir	Ir	Ir	Ir	Ir	Ir	Ir	Ir
Testējamā tehnoloģija	.NET (C#, VB, C++)	Ir	Ir	Ir	Ir	Ir	Ir	Nav	Ir
Testu ierakstīšana un atspēlēšana		Ir	Ir	Nav	Ir	Ir	Ir	Ir	Ir
Darbvirsmas		Ir	Ir	Ir	Ir	Ir	Ir	Nav	Ir
Datu bāzes testēšana		Ir	Ir	Ir	Nav	Nav	Nav	Nav	Ir
Testēšana produkcijas vidē		Ir	Nav	Nav	Nav	Nav	Nav	Nav	Nav
Sistēmas lietotājs var veidot testus		Ir	Nav	Nav	Nav	Nav	Nav	Nav	Nav
Vienlaicīga vairāku testu		Ir	Nav	Ir	Nav	Nav	Nav	Ir	Nav
Paralēlu darbību veikšana		Ir	Nav	Ir	Nav	Nav	Nav	Nav	Ir
Testējamā objekta noteikšana		Ir	Ir	Ir	Nav	Nav	Ir	Ir	Ir
Testa rezultātu analīze		Ir	Ir	Ir	Ir	Ir	Ir	Nav	Ir
Kontrolpunkti		Ir	Ir	Nav	Ir	Ir	Ir	Ir	Ir
Komandas izpildes beigu noteikšana		Ir	Ir	Ir	Nav	Nav	Ir	Ir	Ir

Testēšanas rīku salīdzinājums (paštestēšanas rīka attīstība)

Kritērijs		Paštestēšanas rīks	TestComplete	FitNesse	Ranorex	T-Plan Robot	Rational Functional Tester	Selenium	HP Unified Functional Testing Software
TAP	Attēlu balstīta pieeja	Nav	Nav	Nav	Nav	Ir	Nav	Nav	Nav
Testu automatizēšanas ietvars	Datu vadītas testēšanas ietvars	Nav	Ir	Ir	Ir	Nav	Ir	Daļēji	Ir
	Funkcionālās dekompozīcijas ietvars	Nav	Nav	Nav	Nav	Nav	Nav	Nav	Nav
	Atslēgvārdu vadītas testēšanas ietvars	Nav	Ir	Ir	Ir	Nav	Ir	Ir	Ir
	Modeļu bāzēts ietvars	Nav	Nav	Nav	Nav	Nav	Nav	Nav	Nav
Nefunkcionālā testēšana	Atjaunošanās testēšana	Nav	Nav	Nav	Nav	Nav	Nav	Nav	Nav
	Drošības testēšana	Nav	Nav	Nav	Nav	Nav	Nav	Nav	Nav
	Stresa testēšana	Nav	Ir	Nav	Nav	Nav	Nav	Nav	Nav
	Slodzes testēšana	Nav	Ir	Nav	Nav	Nav	Nav	Nav	Nav
	Veiktspējas testēšana	Nav	Nav	Nav	Nav	Nav	Nav	Nav	Nav
Platforma	Pocket PC	Nav	Ir	Ir	Nav	Ir	Nav	Nav	Nav
	Windows Mobile	Nav	Ir	Nav	Nav	Ir	Nav	Nav	Ir
	Unix	Nav	Nav	Ir	Nav	Ir	Daļēji	Ir	Nav
	OS X	Nav	Nav	Ir	Nav	Ir	Nav	Ir	Nav
	Citas	Nav	Nav	Nav	Nav	Ir	Nav	Daļēji	Nav
Testējamā tehnoloģija	Java	Nav	Ir	Ir	Nav	Ir	Ir	Ir	Ir
	Delphi	Nav	Ir	Ir	Ir	Ir	Nav	Nav	Ir
	ASP.NET	Nav	Ir	Ir	Ir	Ir	Ir	Ir	Ir
	Flash/Flex	Nav	Ir	Nav	Ir	Ir	Ir	Ir	Ir
	Silverlight	Nav	Ir	Nav	Ir	Ir	Nav	Ir	Ir
	HTML	Nav	Ir	Ir	Ir	Ir	Ir	Ir	Ir
	Ruby	Nav	Nav	Ir	Nav	Ir	Nav	Ir	Nav
	Python	Nav	Nav	Ir	Nav	Ir	Nav	Nav	Nav
	Perl	Nav	Nav	Ir	Nav	Ir	Nav	Nav	Nav
	Siebel	Nav	Nav	Nav	Nav	Nav	Ir	Nav	Nav
	SAP	Nav	Nav	Nav	Nav	Nav	Ir	Nav	Ir
	Citas	Nav	Nav	Nav	Nav	Ir	Nav	Daļēji	Nav
Tīmekļa lietojumprogrammu testēšana		Nav	Ir	Ir	Ir	Ir	Ir	Ir	Ir
Servisu testēšana		Nav	Nav	Nav	Nav	Nav	Nav	Nav	Ir
Testu rediģēšana		Nav	Ir	Ir	Ir	Ir	Ir	Ir	Ir
Ekrāna attēli		Nav	Ir	Nav	Ir	Ir	Ir	Nav	Ir
Objektu validēšana		Nav	Ir	Nav	Ir	Nav	Ir	Ir	Ir
Objektu pārlūks		Nav	Ir	Nav	Ir	Nav	Ir	Nav	Ir
Testu žurnāls (log)		Nav	Ir	Ir	Ir	Ir	Ir	Ir	Ir
Testu izpildes laiku plānotājs		Nav	Ir	Nav	Nav	Nav	Nav	Nav	Nav
Spraudņi un paplašinājumi		Nav	Ir	Ir	Nav	Ir	Ir	Ir	Ir

Nākamajā tabulā (4.7. tabula) ir aplūkoti papildus rīku salīdzināšanas kritēriji, kuri netika iekļauti iepriekšējās rīku salīdzināšanas tabulās.

4.7. tabula

Testēšanas rīku salīdzinājums 2

Testēšanas rīks	Izstrādātājs	Cena (EUR)	Ērtums	Klients	Rīka prog. valoda
Paštestēšanas rīks	Datorikas institūts DIVI	*	5	*	C#
TestComplete	SmartBear Software	~ 1 400	3	Adobe, Corel, Falafel Software, ARUP Laboratories, QlikTech u.c.	VBScript, Jscript, C++Script, C#Script, DelphiScript
FitNesse	Robert C. Martin	Bezmaksas	4	*	Java
Ranorex	Ranorex	~ 1 190	3	Bosch, General Electrics, FujitsuSiemens, Yahoo, RealVNC u.c.	C++, Python, C#, VB.NET
T-Plan Robot	T-Plan	Bezmaksas	3	Xerox, Philips, FujitsuSiemens, Virgin Mobile u.c.	Java
Rational Functional Tester	IBM	2 700 – 11 000	4	*	Java, VB.NET
Selenium	OpenQA	Bezmaksas	3	Google, Mozilla, LinkedIn u.c.	C#, Java, Perl, PHP, Python, Ruby
HP Unified Functional Testing Software	Hewlett Packard	3 000 – 10 000	4	*	VBScript, C#

* – nav vai nav zināms

4.8.3. Secinājumi

Salīdzinot astoņus testēšanas rīkus, tika izvirzīti šādi secinājumi:

- Paštestēšanas rīks ir vienīgais, kas nodrošina testējamās sistēmas testēšanu no „iekšpuses", tādējādi nodrošinot testēšanu pēc baltās kastes metodes, kas ļauj detalizētāk notestēt sistēmu.
- Starp darbā apskatītajiem astoņiem rīkiem tikai viens bija tāds, kas kā testu automatizēšanas pieeju izmanto attēlu balstītu pieeju. Pārējie izmanto objektorientētu pieeju. Tādējādi var secināt, ka visbiežāk testu automatizēšanai tiek izmantota objektorientēta pieeja;

- Salīdzinoši ar darbā aprakstītajiem rīkiem paštestēšanas rīks nodrošina plašu klāstu ar testēšanas līmeņiem, jo tikai TestComplete papildus vienību, integrācijas, regresa, funkcionālajai un akcepttestēšanai piedāvā arī stresa un slodzes testēšanu, kamēr pārējie aprobežojas tikai ar trīs testēšanas līmeņiem. Jāatzīmē, ka, ņemot vērā paštestēšanas iespējas izpildīt testus paralēli, stresa un slodzes testēšanas atbalstu būtu salīdzinoši vienkārši ieviest arī paštestēšanas rīkā;
- Vienīgais no apskatītājiem testu automatizēšanas rīkiem, kas nenodrošina testu ierakstīšanas un atspēlēšanas funkcionalitāti ir FitNesse. Tā vietā tiek piedāvāta ērta testu veidošana tabulu formātā;
- Reti kurš testēšanas rīks nodrošina tādas paštestēšanas rīkā pieejamas iespējas kā datu bāzes testēšanu, vienlaicīgu testu izpildi un paralēlu darbību veikšanu testēšanas laikā, savukārt testējamā objekta noteikšanu, testu rezultātu analīzi un kontrolpunktu pievienošanu testam nodrošina gandrīz visi testēšanas rīki;
- Tikai paštestēšanas rīks dod iespēju testēšanu veikt produkcijas vidē un testējamās sistēmas lietotājam bez IT zināšanām veidot testus, tomēr tas arī ir vienīgais, kas nenodrošina testu rediģēšanu un žurnālu;
- Objektu validēšanu un pārlūku piedāvā tikai komerciālie testu automatizēšanas rīki;
- Ekrāna attēlu iegūšana un testu izpildes laiku plānotājs ir papildiespējas, kuras piedāvā tikai daži testēšanas rīki;
- Ja, izvēloties testēšanas rīku, svarīga ir iespējami ātra testu izpilde, tad visticamāk nederēs rīki (Ranorex un T-Plan Robot), kas nespēj noteikt komandu izpildes beigas, bet tā vietā gaida noteiktu laiku;
- Paštestēšanas rīks un Ranorex ir vienīgie, kas nedod iespēju rīku papildināt ar esošiem vai izstrādāt jaunus spraudņus un paplašinājumus;
- Atklātā pirmkoda testēšanas rīki spēj piedāvāt tās iespējas, ko piedāvā komerciālie testēšanas rīki;
- Ne tikai paštestēšanas rīkam pirms testu veidošanas jāiegulda papildus resursi, arī FitNesse rīkam vispirms jāizveido testu armatūra, lai šie testi varētu veiksmīgi izpildīties.

Virkni pašreiz paštestēšanas rīka neatbalstāmo no 4.6. tabulā aprakstītiem kritērijiem ir iespējams ieviest ar salīdzinošie nelieliem rīka uzlabojumiem. Piemēram, lai rīks nodrošinātu veiktspējas testēšanu, nepieciešams ieviest jaunu testa punktu, kas kontrolētu darbību izpildes veiktspēju.

Paštestēšana ir jauna oriģināla pieeja, kas neatpaliek no citiem rīkiem un dažos jautājumos ir neapšaubāmi labāka.

5. PAŠTESTĒŠANAS EFEKTIVITĀTES MĒRĪJUMI

Acīmredzot praktiski nav iespējams pielietot un salīdzināt divus dažādus konceptus vienādos apstākļos – ar vienādu personālu un vienā projektā. Tāpēc paštestēšanas efektivitāte tika novērtēta, balstoties uz retrospektīvu incidentu ziņojumu analīzi konkrētam pietiekami liela apjoma projektam. Tika izvēlēta Valūtas un Vērtspapīru uzskaites sistēma (VVUS), kuras attīstība turpinās jau 8 gadus. Ar automatizētu incidentu ziņojumu uzskaites sistēmu Bugzilla [Bugzilla, 2012] bija fiksēti visi, kopskaitā 1171 incidenta ziņojumi. Katrs incidenta ziņojums tika izanalizēts noskaidrojot, vai incidenta ziņojumu varēja novērst ar atbilstoša testa punkta ielikšanu programmatūrā un testa uzkrāšanu. Tā kā incidentu uzskaites sistēma ir integrēta ar darba laika uzskaites sistēmu, tad efektivitātes mērījumos tiek izmantots ne tikai reģistrēto incidentu pieteikumu skaits, bet arī incidentu pieteikumu risināšanai faktiski patērētais laiks stundās.

Incidentu ziņojumu apstrādes rezultāts ir subjektīvs darba autora viedoklis, tomēr apjomīgais incidentu ziņojumu skaits, statistika atspoguļo tendences.

5.1. Valūtas un vērtspapīru uzskaites sistēma

Ņemot vērā, ka darba autora vadība vairāk kā septiņu gadu garumā notiek Valūtas un vērtspapīru uzskaites sistēmas (VVUS) uzturēšana un attīstība, par paštestēšanas pieejas izmēģinājumu sistēmu tika izvēlēta tieši VVUS. VVUS dažādās konfigurāciju versijās tiek lietota četrās Latvijas bankās:

- kopš 1996. gada AS "SEB banka";
- kopš 2007. gada AS "Reģionālā investīciju banka";
- kopš 2008. gada VA/S "Latvijas Hipotēku un zemes banka";
- Kopš 2012. gada AS „Rigensis Bank”.

VVUS ir divu līmeņu arhitektūrā (klients-serveris) būvēta sistēma, kas sastāv no:

- klienta lietojumprogrammas (vairāk kā 200 formas), kas izstrādāta: Centura SQL Windows, MS Visual Studio C#, MS Visual Studio VB.Net;
- datu bāzes vadības sistēmas Oracle 10g (317 tabulas, 50 procedūras, 52 trigeri, 112 funkcijas).

VVUS sastāv no diviem saistītiem moduļiem:

- vērtspapīru uzskaites modulis (VUS);
- valūtas operāciju uzskaites modulis (VOIS).

5.2. Vērtspapīru uzskaites modulis

Programmatūras mērķis ir nodrošināt vērtspapīru operāciju izpildi un analīzi. Vērtspapīru uzskaites sistēma dod iespēju bankai uzskaitīt klientus, kas izmanto vērtspapīru kontus, vērtspapīrus un ar tiem veiktos darījumus gan klientu, gan pašas bankas vārdā. Bez datu uzskaitīšanas, sistēma lietotājus nodrošina ar analītisku informāciju.

VUS nodrošina bankas un bankas klientu īpašumā esošo vērtspapīru un ar tiem saistīto darījumu uzskaiti. VUS pamata funkcijas:

- Darījumu pieteikumu ievade. VUS nodrošina šādu darījumu uzskaiti: vērtspapīru pirkšana/pārdošana, vērtspapīru pārvedumi/ieskaitījumi, fondu daļu darījumi, vērtspapīru pārvedumi starp korkontiem, vērtspapīru dereģistrācija, akciju bloķēšana uz akcionāru pilnsapulci, REPO un REVERSE REPO darījumi, vērtspapīru apgrūtināšana, opciju tirdzniecība;
- Darījumu pieteikumu statusu kontrole un darījumu apstrāde pēc noteikta scenārija;
- Informācijas apmaiņa ar ārējām sistēmām. VUS nodrošina datu apmaiņu ar šādām ārējām sistēmām: bankas interneta banku, fondu daļu emitenta sistēmu, Latvijas Centrālo depozitāriju, Rīgas Fondu biržu, bankas centrālo sistēmu, Bloomberg, SWIFT, Līdzekļu pārvaldības sistēmu (LPS);
- Izpildīto darījumu reģistrācija un apstrāde;
- Dažādu komisiju (brokeru pakalpojumu, vērtspapīru turēšanas komisiju u.c.) aprēķināšana un iekasēšana;
- Bankas portfeļa vērtspapīru pārvērtēšana, amortizācija un uzkrājuma aprēķināšana;
- Pret partneriem definēto limitu kontrole;
- Salīdzina bankas korkontu atlikumus ar korespondējošo banku sūtītajiem vērtspapīru korkontu izrakstiem;
- Ar vērtspapīriem saistīto naudas maksājumu izveide un apstrāde. VUS nodrošina šādu maksājumu uzskaiti: dividenžu izmaksas (arī par dereģistrētām akcijām), nodokļu ieturēšana/atgriešana, kuponu izmaksas, pamatsummas izmaksas, vērtspapīram dzēšoties;
- Atskaišu sagatavošana (ieskaitot FKTK atskaites, LB atskaites, klientu vērtspapīru kontu izrakstus).

VUS nodrošina uz darījumiem bāzētu vērtspapīru kontu atlikumu operatīvu pārrēķinu. VUS tiek nodrošināts ar maināmu darbavietu konfigurāciju.

5.2.1. Valūtas operāciju uzskaites modulis

Programmatūras mērķis ir uzskaitīt un analizēt valūtas darījumus (valūtas maiņas, starpbanku depozītus u.c). Sistēma uzskaita valūtas darījumus, klientus, kas veic valūtas darījumus. Bez datu uzskaitīšanas, sistēma lietotājus nodrošina ar analītisku informāciju par klientiem, valūtas darījumiem, valūtas kursiem.

VOIS galvenās funkcijas:

- Darījumu ievade. VOIS nodrošina šādu darījumu uzskaiti: starpbanku FX, FX Forwrd, SWAP (riska, bezriska) darījumi, starpbanku depozītu ieguldīšanas, piesaistīšanas, pagarināšanas darījumi, klientu FX, FX Forward, SWAP (riska, bezriska) darījumi, klientu depozītu ieguldīšanas, piesaistīšanas, pagarināšanas darījumi, mainīgās likmes darījumi, starpbanku Order darījumi, klientu Order darījumi, procentu likmju mijmaiņas (IRS) darījumi, opciju darījumi, Valsts kases darījumi, nodrošinājumu darījumi, valūtu mijmaiņas darījumi;
- Darījumu statusu kontrole un darījumu apstrāde pēc noteikta scenārija (stāvokļu pāreja);
- Informācijas apmaiņa ar ārējām sistēmām. VOIS nodrošina apmaiņu ar šādām ārējām sistēmām: bankas centrālā sistēma, Reuter, UBS TS tirdzniecības platforma, SWIFT;
- Valūtu pozīciju uzturēšana (bankas kopējā pozīcija, pozīcijas par portfeli);
- Limitu definēšana un kontrole;
- Darījumu imports no Reuter un Interneta tirdzniecības platformām (TS; UBS TS u.c.);
- Valūtas kursu un % likmju imports no Reuter;
- Marginālā tirdzniecība (*Margin Trading*);
- Nostro kontu atlikumi;
- Atskaišu sagatavošana.

VOIS tiek nodrošināts ar maināmu darbavietu konfigurāciju.

5.3. Paštestēšanas efektivitātes mērījumu pieeja

Tika atlasītas visi (kopskaitā 1171 kļūdas pieteikumi) Valūtas un vērtspapīru uzskaites sistēmas kļūdu pieteikumi, kas reģistrēti 8 gadu garumā. Katrs kļūdas pieteikums tika izvērtēts pēc nākamajā tabulā (skat. 5.1. tabulā) aprakstītiem kritērijiem. Tabulā ir redzams, ka ne visi kļūdu pieteikumi ir klasificēti kā kļūdas. Lietotāji kļūdu uzskaites sistēmā reģistrēja kļūdas, bet detalizētākas kļūdas izmeklēšanas, iepazīšanas rezultātā kļūda atsevišķos gadījumos tika pārklasificēta par lietotāju kļūdu, uzlabojumu vai konsultāciju.

5.1. tabula

Kļūdu pieteikumu veidi

Kļūdas veids	Apraksts
Dublikāts	Kļūdas pieteikuma veids tika piemērots tiem kļūdas pieteikumiem, kas atkārto esoša, spēkā esoša kļūdas pieteikumu.
Lietotāju kļūda	Kļūdas pieteikuma veids tika piemērots tiem kļūdas pieteikumiem, kurus risinot atklājās, ka paša lietotāju darbības vainas dēļ radusies kļūdas pieteikumā aprakstītā kļūda.
Kļūda – nenovēršama	Kļūdas pieteikuma veids tika piemērots kļūdas pieteikumiem, kas aprakstīja reālu sistēma esošu kļūdu, kuru neidentificētu paštestēšanas pieeja, pieņemot, ka sistēmā ir iestrādāts paštestēšanas pieejas mehānisms.
Kļūda – novēršama	Kļūdas pieteikuma veids tika piemērots kļūdas pieteikumiem, kas aprakstīja reālu sistēma esošu kļūdu, kuru identificētu paštestēšanas pieeja, pieņemot, ka sistēmā ir iestrādāts paštestēšanas pieejas mehānisms.
Uzlabojums	Kļūdas pieteikuma veids tika piemērots tiem kļūdas pieteikumiem, kas, izvērtējot pieteikumu, izrādījās sistēmas funkcionalitātes uzlabojums.
Konsultācija	Kļūdas pieteikuma veids tika piemērots tiem kļūdas pieteikumiem, kas tika atrisināti ar konsultāciju palīdzību.

Paštestēšanas pieejas efektivitātes mērījumiem svarīgākie kļūdu pieteikumu veidi ir Kļūda – nenovēršama un Kļūda – novēršama. Tādēļ šo veidu kļūdu pieteikumi ir aplūkoti detalizētāk, balstoties uz nākamajās tabulās (skat. 5.2. un 5.3. tabulās) redzamajiem kļūdu veidu sadalījumiem.

5.2. tabula

Kļūda - nenovēršama pieteikumu veida sadalījums pēc kļūdu veidiem

Kļūdas veids	Apraksts
Ārējā saskarnes kļūda	Kļūda datu apmaiņā ar ārējo saskarni.
Datoru konfigurācijas kļūda	Lietotāja datora konfigurācijas neatbilstība VVUS prasībām.
Datu tipu kļūda	Datu tipu nekonsekventa, nekorekta lietošana. Neatbilstība starp formas laukiem un datu bāzes tabulu laukiem.
Lietotāja saskarnes kļūdas	Vizuālas izmaiņas. Piemēram, formā lauks ir neaktīvs, vai arī atskaitē netiek attēlots logo.
Lietotāju paralēlo darbību kļūda	Vienlaicīgas vairāku lietotāju darbības ar vienu ierakstu sistēmā.
Prasību interpretācijas kļūda	Nepilnīgas pasūtītāja prasības. Izpildītāja kļūdaina prasību interpretācija.
Specifisks gadījums	Dažādi specifiski testu scenāriji, kuru izpildes rezultātā rodas sistēmas kļūda.

5.3. tabula

Kļūda - novēršama pieteikumu veida sadalījums pēc testa punktu veidiem

Testa punkts, kas identificētu konkrēto kļūdu
Faila rezultāta testa punkts
Ievadlauka testa punkts
Lietojumprogrammas notikuma testa punkts
Salīdzināmās vērtības testa punkts
Sistēmas paziņojuma testa punkts
SQL vaicājuma rezultāta testa punkts

Tabulā norādītais Faila rezultāta testa punkts pēc tehniskās implementācijas ir tas pats Salīdzināmās vērtības testa punkts. Faila rezultāta testa punkts efektivitātes mērījumos izdalīts atsevišķi, lai uzsvērtu tieši ar failu apstrādi identificētās kļūdas.

5.4. Mērījumu rezultāti

5.4.1. Pieteikumu sadalījums pēc pieteikuma veida un tiem veltītā laika

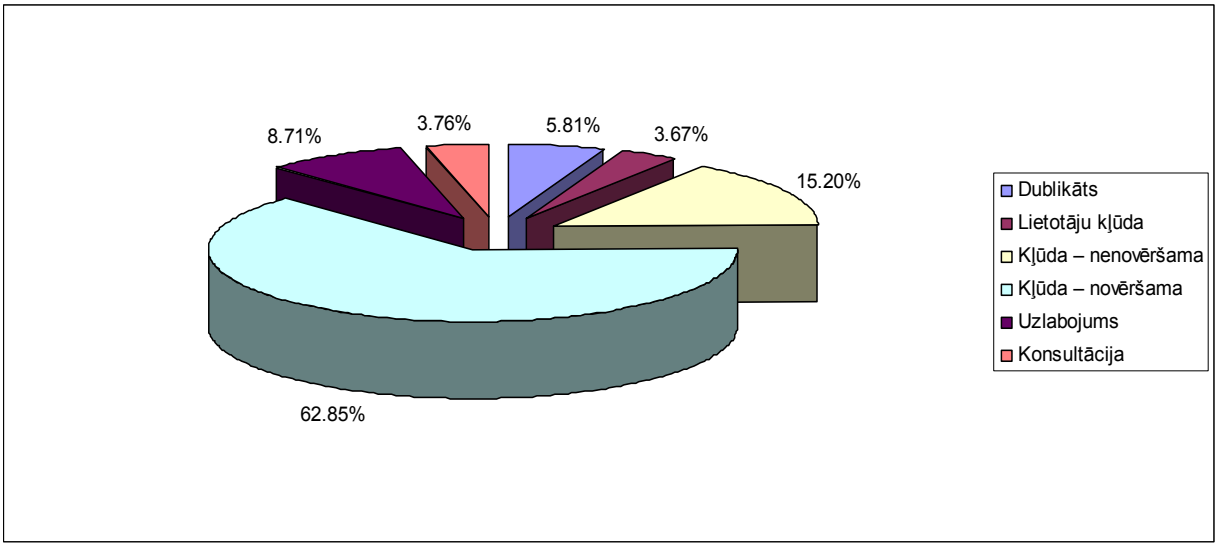
Nākamajā 5.4 tabulā ir apskatāms kopējais kļūdu pieteikumu sadalījums pa kļūdu veidiem un tiem veltīto laiku stundās. Tabula satur šādas kolonnas:

- Pieteikumu veids – kļūdas pieteikuma veids (skat. 5.1. tabulā);
- Skaits – kļūdas pieteikumu skaits;
- % no kopējā – pieteikuma veida skaits procentos no kopējā kļūdu pieteikumu skaita;
- Stundas – veltītais laiks kļūdas pieteikuma veida risināšanā;
- % no kopējā – pieteikuma veidam veltītais laiks procentos no kopējā visiem kļūdu pieteikumiem veltītā laika.

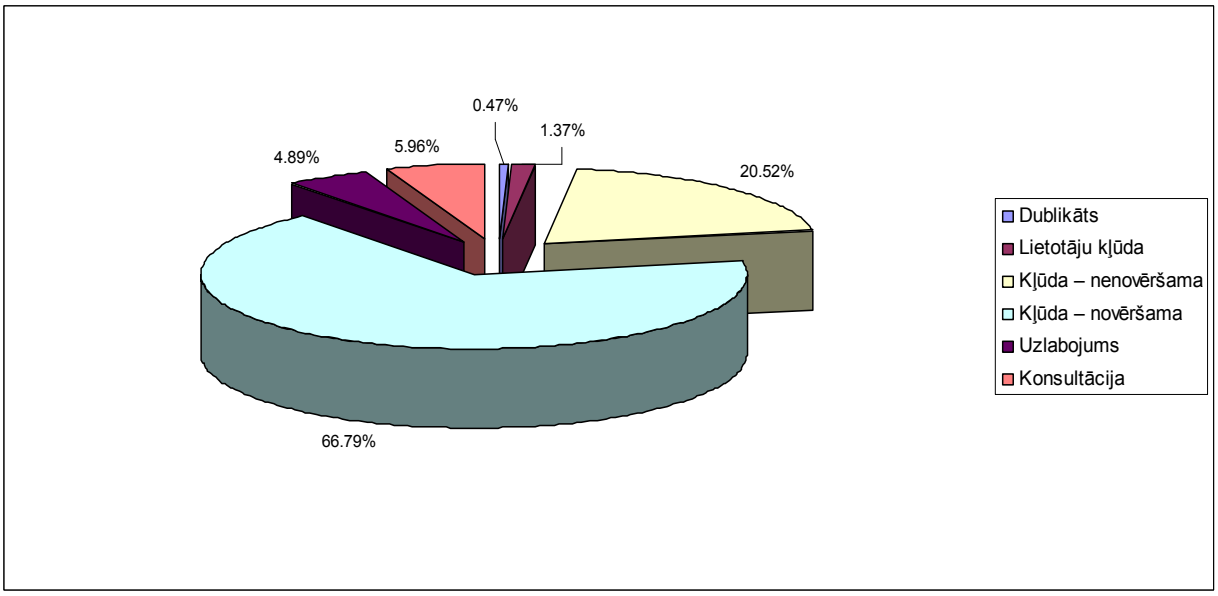
5.4. tabula

Kopējais kļūdu pieteikumu sadalījums pa kļūdu veidiem un tiem veltītā laika

Pieteikuma veids	Skaits	% no kopējā	Stundas	% no kopējā
Dublikāts	68	5.81	23.16	0.47
Lietotāju kļūda	43	3.67	67.46	1.37
Kļūda – nenovēršama	178	15.2	1011.96	20.52
Kļūda – novēršama	736	62.85	3293.74	66.79
Uzlabojums	102	8.71	241.36	4.89
Konsultācija	44	3.76	293.92	5.96
Kopā:	**1171**	**100**	**4931.6**	**100**



5.1. att. **Kopējais kļūdu pieteikumu sadalījums pa kļūdu veidiem**



5.2. att. **Kopējais kļūdu pieteikumu sadalījums pēc kļūdu veidiem veltītā laika**

Ņemot vērā iegūto informāciju, autors secina, ka:

- No kopējā kļūdu pieteikuma skaita (1171) paštestēšanas pieeja VVUS jau izstrādes laikā identificētu 62.85% (pieteikumu skaits: 736, novēršanai patērētās stundas: 3293.74) no visiem nepilnu deviņu gadu laikā reģistrētiem kļūdu pieteikumiem. Tas nozīmē, ka izstrādātāji jau izstrādes vidē spētu identificēt šīs kļūdas un tās novērst, tādējādi būtiski uzlabojot sistēmas kvalitāti, kā arī iemantojot pasūtītāja uzticību pret sistēmas kvalitāti.
- Kā norādīts [Pressman, 2004], jo ātrāk kļūda tiek identificēta, jo kļūdas novēršanas izmaksas ir mazākas. Tiek nosaukta starpība pat vairāk kā desmit un pat simts reizes.

Bez tam sistemātiska testēšanas nodrošina to, ka līdz pasūtītājam nonāk tikai 3% no visām kļūdām [Bičevskis, 1998]. 5.4. tabulā norādītais laiks, kas veltīts kļūdu novēršanai būtu neapšaubāmi mazāks, ja kļūda tiktu identificēta jau izstrādes vidē. Pieņemot, ka kļūdas identificēšanas izstrādes vidē ļautu ietaupīt 10% no kļūdas novēršanas laika, kas tika identificētas pie pasūtītāja, deviņu gadu garumā tiktu ietaupītas aptuveni 329h, kas ir līdzvērtīgas aptuveni 41 darba dienai.

- No kopējā kļūdu pieteikumu skaita paštestēšanas pieeja VVUS nespētu identificēt 15.2% kļūdu (pieteikumu skaits: 178, novēršanai patērētās stundas: 1011.96) no kopējā kļūdu pieteikumu skaita.
- No kopējā kļūdu pieteikuma skaita 78.05% (pieteikumu skaits: 914, novēršanai patērētās stundas: 4305.7) ir reālas kļūdas, kas tika labotas, pārējie 11.95% (pieteikumu skaits: 257, patērētās stundas: 625.9) kļūdu pieteikumi bija lietotāju kļūdas, uzlabojumi, konsultācijas un kļūdu dublikāti.
- Kļūdu atrisināšanai veltītais laiks procentuāli (87.31%) no kopējā kļūdu pieteikumiem patērētā laika ir lielāks kā procentuālais (78.05%) kļūdu skaits no kopējā kļūdu pieteikumu skaita. Tas nozīmē, ka kļūdu risināšanai, novēršanai proporcionāli pret pārējiem kļūdu pieteikumiem (uzlabojumiem, lietotāju konsultācijām, lietotāju kļūdu identificēšanai, kļūdu dublikātu identificēšanai) ir patērēts vairāk laika.

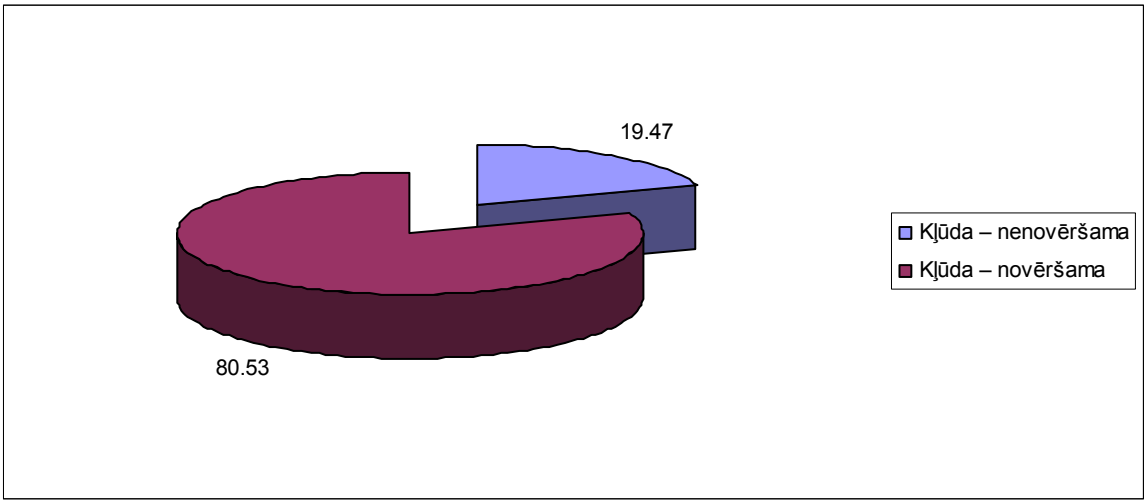
5.4.2. Pieteikumu sadalījums pēc reālo kļūdu veidiem

Kā autors minēja iepriekšējā nodaļā, par reālām kļūdām ir uzskatāmi šādi darbā izdalīti kļūdu pieteikumu veidi: Kļūda – nenovēršama, Kļūda – novēršama. Nākamajā 5.5. tabulā atsevišķi tiek aplūkots reālo kļūdu pieteikumu sadalījums. Tabulas kolonnu apraksts sakrīt ar iepriekšējā nodaļā aprakstītām kolonnām.

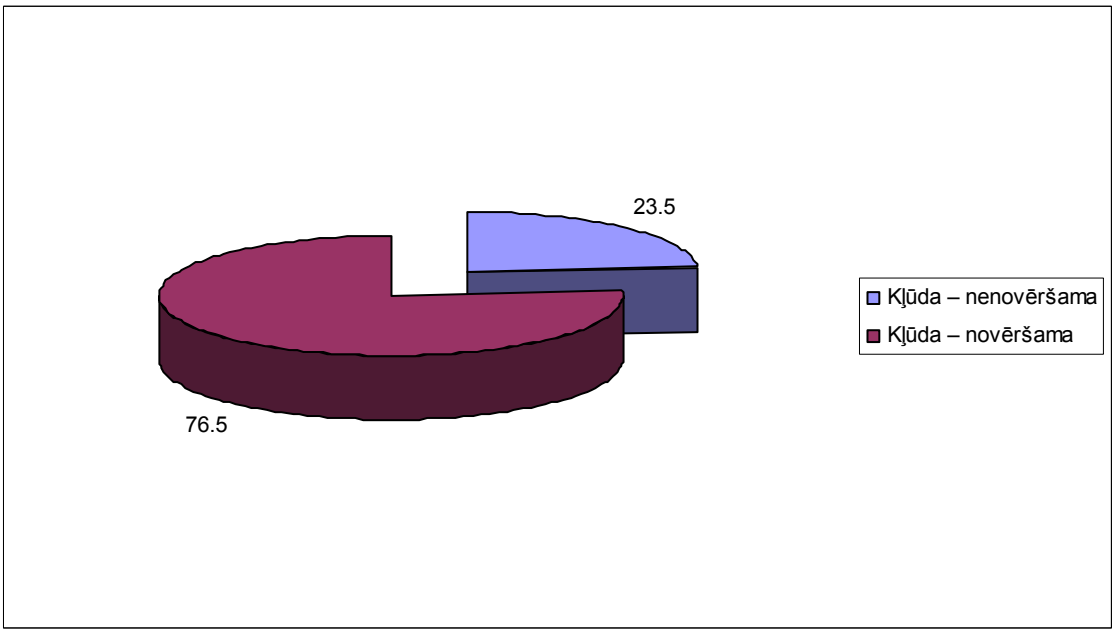
5.5. tabula

Kļūdu pieteikumu sadalījums pa kļūdu veidiem

Kļūdas veids	Skaits	% no kopējā	Stundas	% no kopējā
Kļūda – nenovēršama	178	19.47	1011.96	23.5
Kļūda – novēršama	736	80.53	3293.74	76.5
Kopā:	**914**	**100**	**4305.7**	**100**



5.3. att. **Kļūdu pieteikumu sadalījums pa kļūdu veidiem**



5.4. att. **Kļūdu pieteikumu sadalījums pa kļūdu veidiem pēc veltītā laika**

Ņemot vērā iegūto informāciju, autors secina, ka:

- Paštestēšanas pieeja VVUS identificētu 4/5 no visām reģistrētām kļūdām.
- Paštestēšanas pieejas identificējošo kļūdu atrisināšanai veltītais laiks procentuāli (76.5%) no kopējā kļūdu atrisināšanai patērētā laika ir mazāks nekā procentuālais (80.53%) kļūdu skaits no kopējā kļūdu skaita. Tas nozīmē, ka paštestēšanas pieejas identificēto kļūdu risināšanai, novēršanai proporcionāli pret kļūdām, kuras netiktu identificētas ar paštestēšanas mehānismu, ir patērēts mazāk laika.

5.4.3. Pieteikumu sadalījums pa gadiem

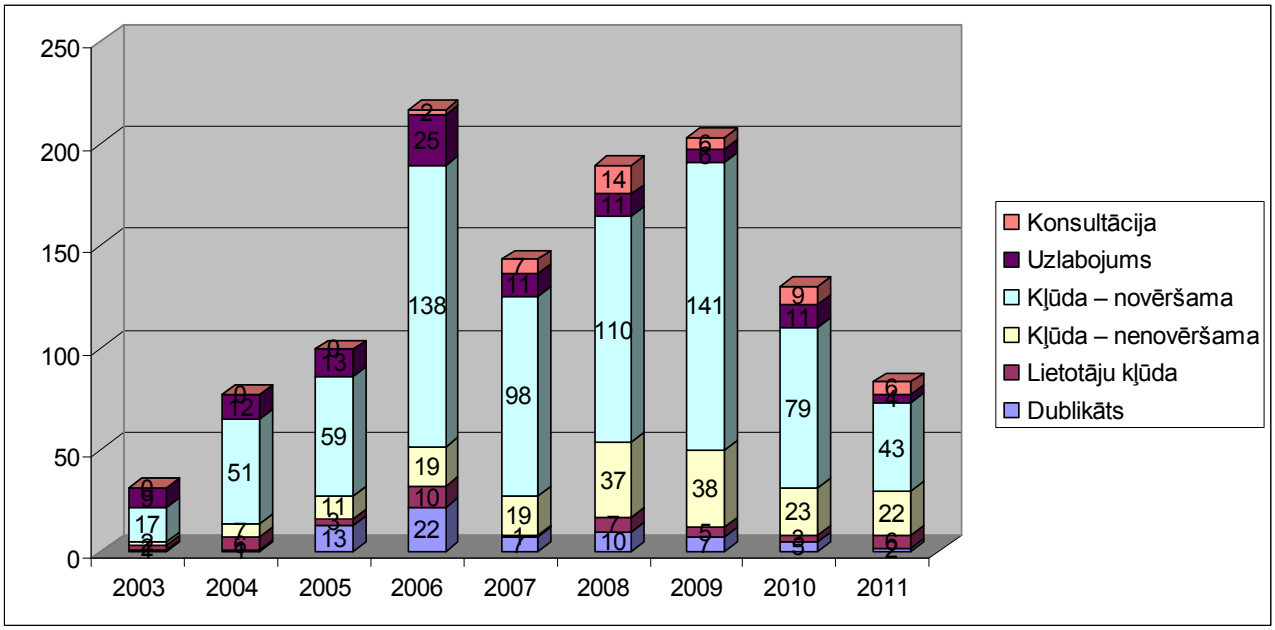
Nākamajā 5.6. tabulā ir apskatāms kļūdu pieteikumu sadalījums pa gadiem. Tabula satur šādas kolonnas:

- Pieteikumu veids
  - kļūdas pieteikuma veidu (skat. 5.1. tabulā) skaits pa gadiem;
  - % no kopējā – kļūdu pieteikuma veids procentuāli no kopējā kļūdu pieteikumu skaita gadā;
- 2003-2011 – gadi, kas aplūkoti, analizējot VVUS kļūdu pieteikumus;
- Kopā – kļūdu pieteikumu skaita summas.

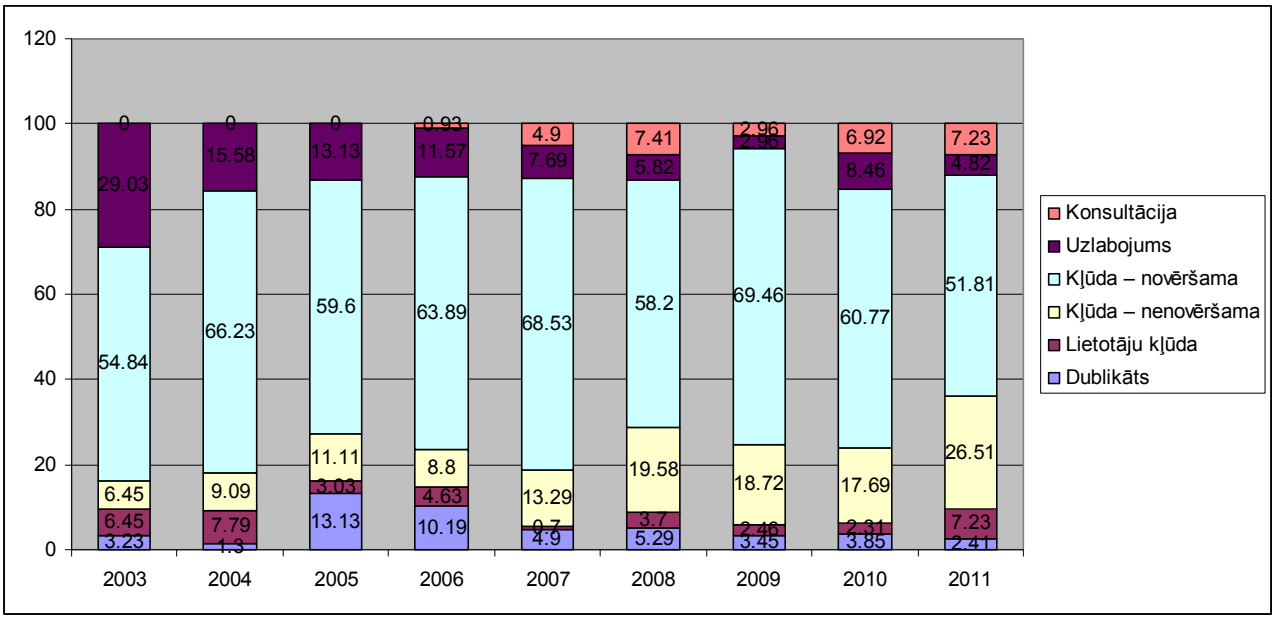
5.6. tabula

Pieteikumu sadalījums pa gadiem

Pieteikuma veids	2003	2004	2005	2006	2007	2008	2009	2010	2011	Kopā
Dublikāts	1	1	13	22	7	10	7	5	2	**68**
% no kopējā	3.23	1.3	13.13	10.19	4.9	5.29	3.45	3.85	2.41	
Lietotāju kļūda	2	6	3	10	1	7	5	3	6	**43**
% no kopējā	6.45	7.79	3.03	4.63	0.7	3.7	2.46	2.31	7.23	
Kļūda – nenovēršama	2	7	11	19	19	37	38	23	22	**178**
% no kopējā	6.45	9.09	11.11	8.8	13.29	19.58	18.72	17.69	26.51	
Kļūda – novēršama	17	51	59	138	98	110	141	79	43	**736**
% no kopējā	54.84	66.23	59.6	63.89	68.53	58.2	69.46	60.77	51.81	
Uzlabojums	9	12	13	25	11	11	6	11	4	**102**
% no kopējā	29.03	15.58	13.13	11.57	7.69	5.82	2.96	8.46	4.82	
Konsultācija	0	0	0	2	7	14	6	9	6	**44**
% no kopējā	0	0	0	0.93	4.9	7.41	2.96	6.92	7.23	
Kopā (skaits):	**31**	**77**	**99**	**216**	**143**	**189**	**203**	**130**	**83**	**1171**



5.5. att. **Pieteikumu sadalījums pa gadiem**



5.6. att. **Pieteikumu skaita procentuālais sadalījums pa gadiem**

Apskatot tabulu un grafikus ir redzams, ka:

- Atsevišķos gados atsevišķiem kļūdu pieteikumu veidiem ir lielāki kāpumi vai kritumi, kā arī konsultācija tipa pieteikumi ir reģistrēti sākot ar 2005. gadu, bet procentuālais sadalījums pa gadiem aptuveni sakrīt ar kopējo pieteikumu skaita procentuālo sadalījumu.
- Ik gadu lielāko daļu no pieteiktajām kļūdām paštestēšanas pieeja spētu identificēt.

5.4.4. Kļūdu pieteikumu sadalījums pa gadiem

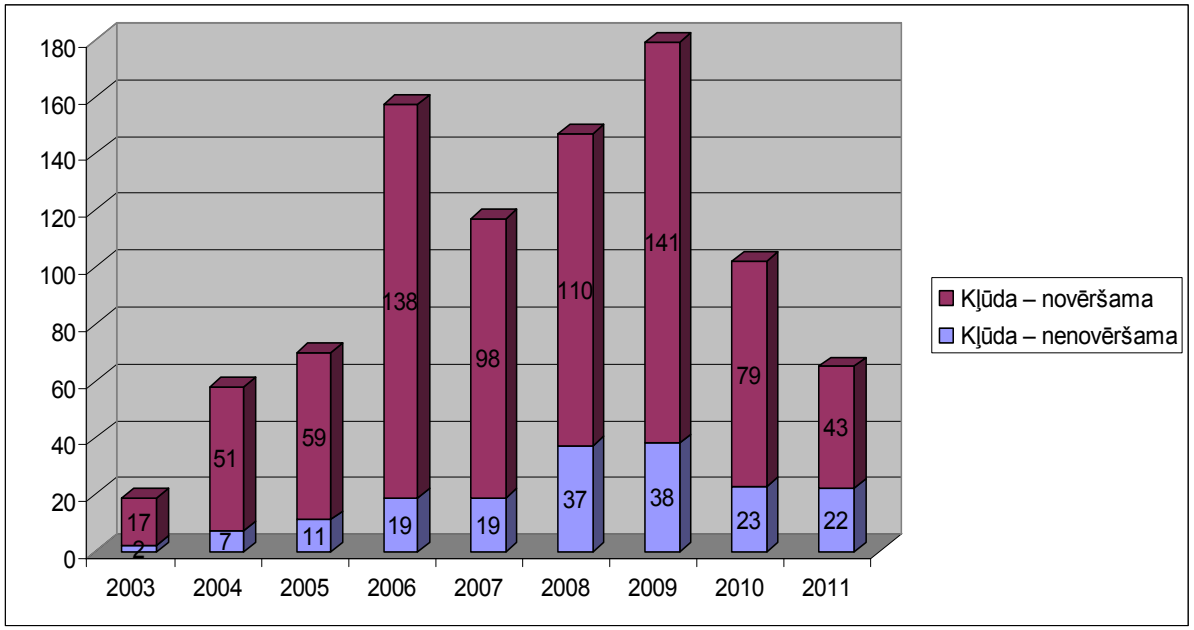
Nākamajā 5.7. tabulā atsevišķi tiek aplūkots reālo kļūdu pieteikumu sadalījums. Tabula satur šādas kolonnas:

- Pieteikumu veids
  - kļūdas pieteikuma veidu (skat. 5.1. tabulā) skaits pa gadiem;
  - % no kopējā – kļūdu pieteikuma veida skaits pa gadiem procentos no kopējās kļūdu pieteikuma veida skaita;
- 2003-2011 – gadi, kas aplūkoti, analizējot VVUS kļūdu pieteikumus;
- Kopā – kļūdu pieteikumu skaita summas.

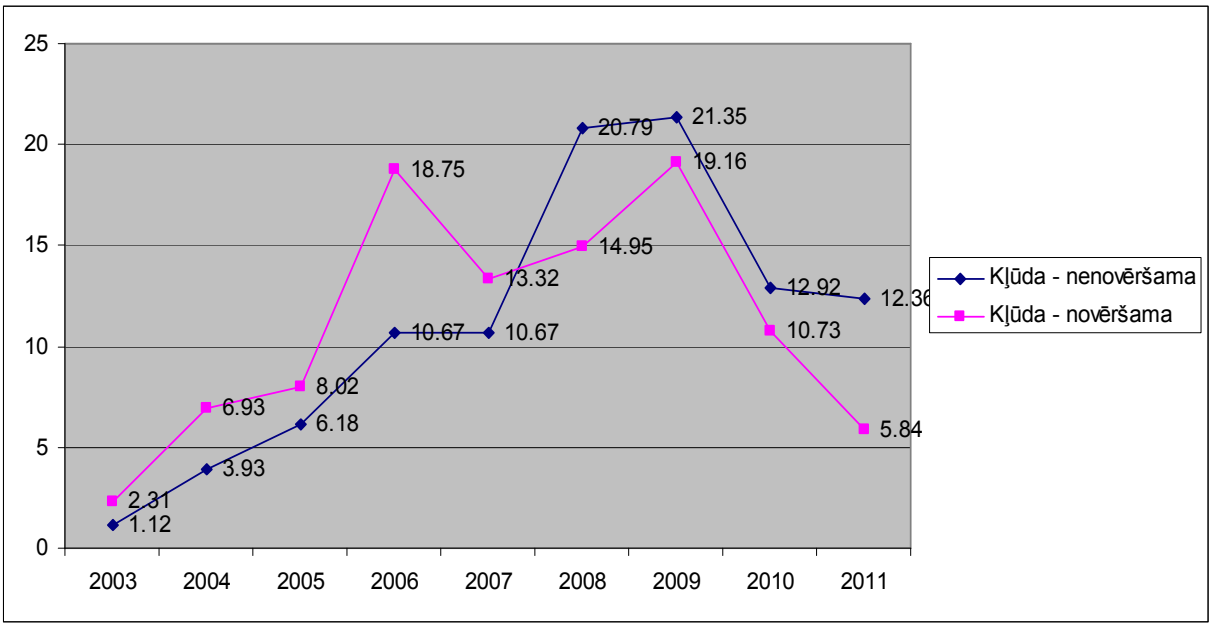
5.7. tabula

Kļūdu pieteikumu sadalījums pa gadiem

Pieteikuma veids	2003	2004	2005	2006	2007	2008	2009	2010	2011	Kopā
Kļūda – nenovēršama	2	7	11	19	19	37	38	23	22	**178**
% no kopējā	1.12	3.93	6.18	10.67	10.67	20.79	21.35	12.92	12.36	**100**
Kļūda – novēršama	17	51	59	138	98	110	141	79	43	**736**
% no kopējā	2.31	6.93	8.02	18.75	13.32	14.95	19.16	10.73	5.84	**100**
Kopā:	**19**	**58**	**70**	**157**	**117**	**147**	**179**	**102**	**65**	**914**



5.7. att. **Kļūdu pieteikumu skaits sadalījumā pa gadiem**



5.8. att. **Kļūdu pieteikumu procentuālais sadalījums pa gadiem**

Ņemot vērā iegūto informāciju, autors secina, ka:

- Paštestēšanas pieeja VVUS identificētu lielāko daļu pieteikto kļūdu.
- Paštestēšanas pieejas identificēto un neidentificēto kļūdu skaita izmaiņas procentos pa gadiem ir līdzīgas.
- Pirmos piecus gadus paštestēšanas pieejas identificējamo kļūdu skaita izmaiņas procentuāli ir lielākas, savukārt, sākot ar 2008. gadu lielākas kļūdu skaita izmaiņas ir kļūdām, ko paštestēšanas pieeja neidentificētu.

5.4.5. Pieteikumu apjoms attiecībā pret attīstības apjomu sadalījumā pa gadiem

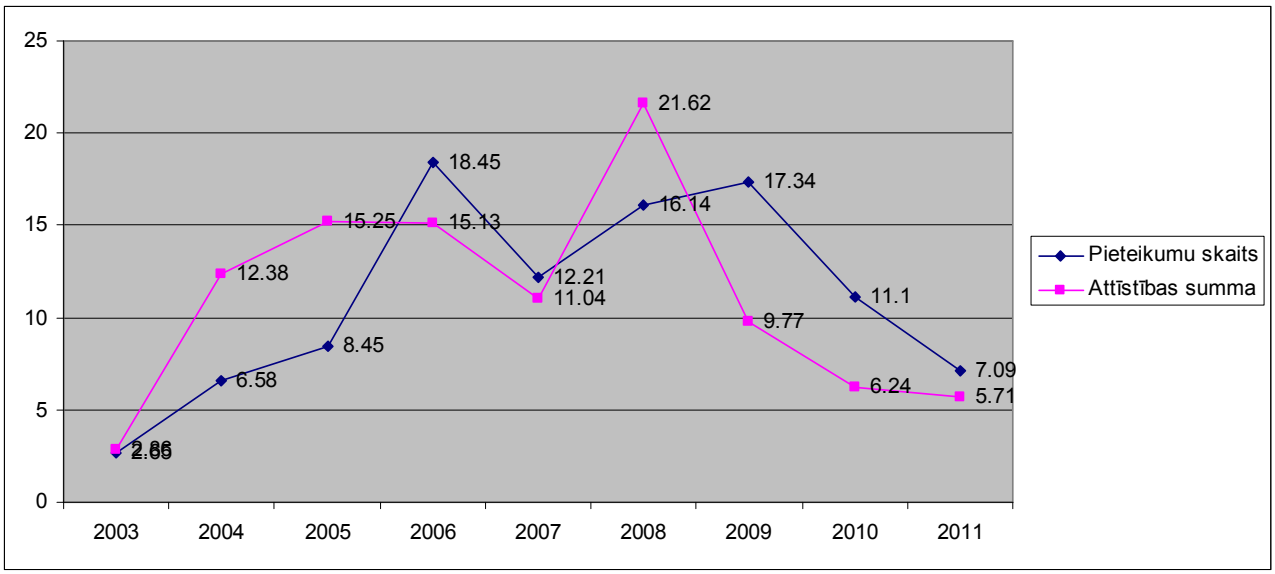
Nākamajā 5.8. tabulā ir apskatāms kļūdu pieteikumu skaita attiecība pret attīstības apjomu pa gadiem. Tabula satur šādas kolonnas:

- Gads – gadi, kas aplūkoti, analizējot VVUS kļūdu pieteikumus;
- Pieteikumu skaits – reģistrēto VVUS kļūdu pieteikumu skaits pa gadiem;
- % pret kopējo – pieteikumu skaita procentuālais sadalījums pa gadiem;
- Attīstības summa % pret kopējo – attīstībai ieguldītās summas procentuālais sadalījums pa gadiem. Attīstības summas ir konfidenciālas, tādēļ darbā tās nav iekļautas.

5.8. tabula

Pieteikumu skaits attiecībā pret attīstības apjomu pa gadiem

Gads	Pieteikumu skaits	% pret kopējo	Attīstības summa % pret kopējo
2003	31	2.65	2.86
2004	77	6.58	12.38
2005	99	8.45	15.25
2006	216	18.45	15.13
2007	143	12.21	11.04
2008	189	16.14	21.62
2009	203	17.34	9.77
2010	130	11.1	6.24
2011	83	7.09	5.71
Kopā:	**1171**	**100**	**100**



5.9. att. **Pieteikumu skaits attiecībā pret attīstības apjomu**

Šajā apakšnodaļā aplūkotā informācija tieši neatspoguļo informāciju par paštestēšanas pieejas efektivitāti, bet ir interesanti salīdzināt pieteikumu skaita izmaiņas (%) un attīstības apjoma izmaiņas (%) deviņu gadu garumā. Ņemot vērā iegūto informāciju, var secināt, ka pieaugot attīstības apjomam, pieaug arī kļūdu pieteikumu apjoms. Secinājums ir diezgan loģisks, bet šajā gadījumā tam ir arī reāls pamatojums, piemērs.

5.4.6. Paštestēšanas pieejas un identificējamo kļūdu sadalījums pa kļūdu veidiem

Nākamajā 5.9. tabulā ir apskatāms paštestēšanas pieejas neidentificējamo kļūdu sadalījums pa kļūdu veidiem. Tabula satur šādas kolonnas:

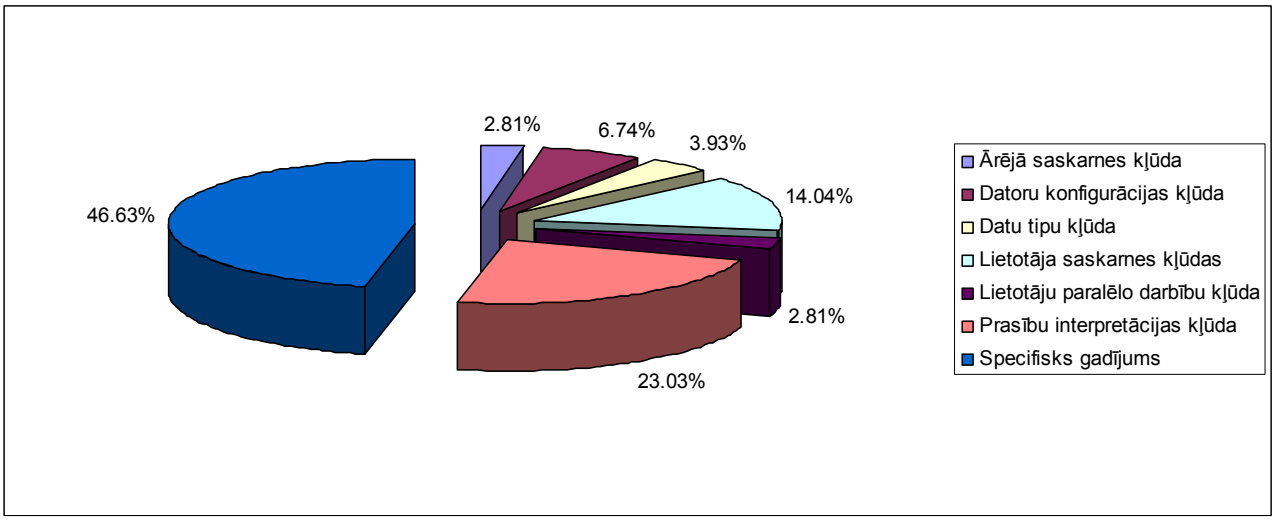
- Kļūdas veids – viens no septiņiem kļūdas veidiem;

- Skaits – kļūdas veida skaits;
- % no kopējā – kļūdas veida skaits procentos no kopējā kļūdu skaita.

5.9. tabula

Paštestēšanas pieejas neidentificējamo kļūdu sadalījums pa veidiem

Kļūdas veids	Skaits	% no kopējā
Ārējā saskarnes kļūda	5	2.81
Datoru konfigurācijas kļūda	12	6.74
Datu tipu kļūda	7	3.93
Lietotāja saskarnes kļūdas	25	14.04
Lietotāju paralēlo darbību kļūda	5	2.81
Prasību interpretācijas kļūda	41	23.03
Specifisks gadījums	83	46.63
Kopā:	**178**	**100**



5.10. att. **Paštestēšanas pieejas neidentificējamo kļūdu sadalījums pa veidiem**

Ņemot vērā iegūto informāciju, autors secina, ka:

- Lielāko daļu (gandrīz 50%) no kļūdām, kuras nespētu identificēt paštestēšanas ir specifiskie gadījumi (testa scenāriji). Tie ir nestandarta testa scenāriji:
    - kas izstrādājot sistēmu nav aplūkoti. Piemēram, šāds scenārijs no kļūdas apraksta: „ievadu pieteikuma vārdu, paroli, spiežu enter, spiežu vēlreiz enter, spiežu Sākt darbu – kokā izvēlos jebkuru skatu, kuram apakšā ir objekti. Rezultātā kļūda!”. Kā redzams, tas ir specifisks gadījums, kuru izstrādātājs nav aplūkojis. Kritiskās funkcionalitātes testēšanai tiktu izveidots testa piemērs, kurā tiktu iekļauta enter taustiņa nospiešana vienu reizi, bet šāds testa piemērs kļūdu neradītu. Savukārt, testa scenārijs ar divkāršu enter nospiešanu, kas

radītu kļūdu, netiktu izveidots, jo tas ir specifisks gadījums, ko lietotājs ikdienas darbā neveic.
    - ko paštestēšanas pieeja nespēj pārbaudīt. Piemēram, šāds kļūdas apraksts: „Drukājot darījumu, tiek izdrukāta viena lieka balta lapa". Paštestēšanas pieeja nespēj pārbaudīt, vai pēdējā izdrukātā lapa ir balta.
- Piektdaļu no visiem kļūdas veidiem, ko paštestēšanas pieeja nespētu identificēt, ir prasību interpretācijas kļūdas. Šāda veida kļūdas rodas, kad tiek izstrādāti sistēmas papildinājumi/uzlabojumi, bet rezultāts neatbilst pasūtītāja prasībām, jo izstrādātājs ir savādāk interpretējis pasūtītāja prasības. Stundu skaits (41h) deviņu gadu garumā, manuprāt, nav liels un ir pieļaujams.
- Paštestēšanas pieeja nespēj identificēt lietotāju saskarnes kļūdas (vizuālās izmaiņas, datuma formāta, lauka pieejamības izmaiņas u.c.).
- Daļa reģistrēto kļūdu ir saistītas ar lietotāja datora konfigurācijas neatbilstību pret sistēmas prasībām. Paštestēšanas pieeja šāda veida kļūdas spētu identificēt tikai uz paša lietotāja datora, bet neidentificētu uz testēšanas datora, kam konfigurācija atbilst sistēmas prasībām.
- Daļa kļūdu, ko nespēj identificēt paštestēšanas pieeja, ir datu tipu kļūdas, kas ietver:
    - loga lauku garuma atbilstības ar datu bāzes tabulas lauku garuma pārbaudi;
    - mainīgā datu tipa maksimālās vērtības pārsniegšana.
- Paštestēšanas pieeja nespēj identificēt kļūdas, kas atkarīgas no ārējo saskarņu datiem. Paštestēšanas pieeja spēj uzkrāt un izpildīt testa piemērus, kas satur ārējo saskarņu datus, bet ne vienmēr spēj novērtēt saņemto datu pareizību. Ar paštestēšanas pieeju varam pārbaudīt nemainīgu datu atlasīšanu no ārējās saskarnes (piemēram, vēsturisku valūtas kursa atlasīšana), bet nevaram pārbaudīt mainīgu datu atlasīšanas pareizību (piemēram, pašreizējā klienta naudas konta atlikuma atlasīšana). Konkrētā gadījumā reģistrētās kļūdas nebija pašas sistēmas kļūdas, bet ārējo saskarņu kļūdas.
- Lietotāju paralēlo darbību kļūdas saistītas ar to, ka sistēmā nekorekti iestrādāts transakciju mehānisms. Vairākiem lietotājiem vienlaicīgi koriģējot vienu ierakstu, uzkārās sistēma. Šāda veida kļūdas paštestēšanas pieeja neidentificē.

5.4.7. Paštestēšanas pieejas identificējamo kļūdu sadalījums pa testa punktu veidiem

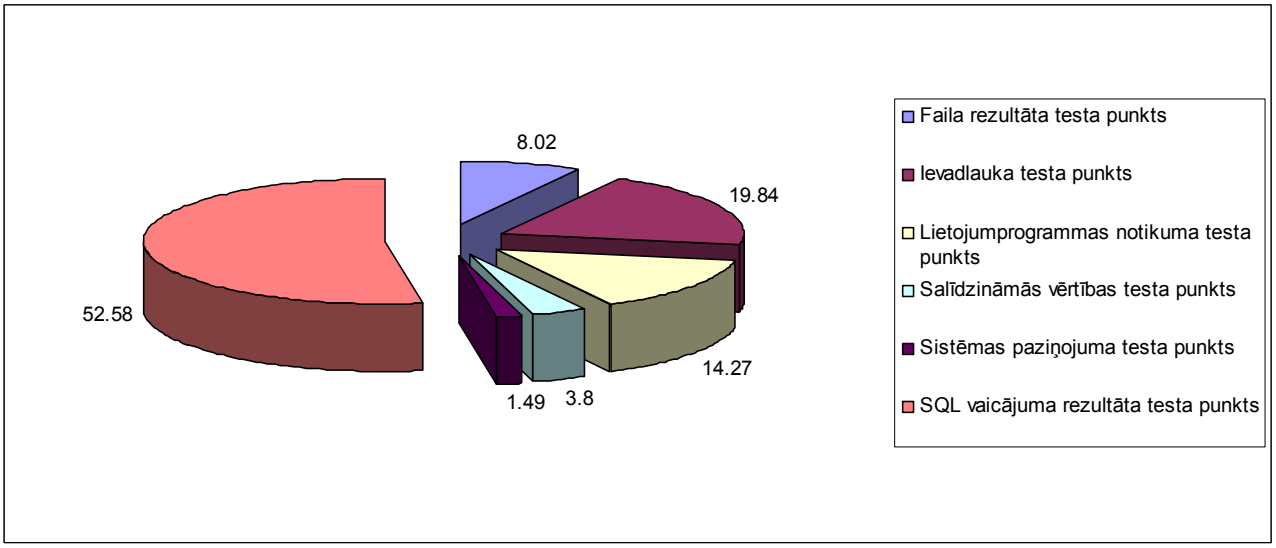
Nākamajā 5.10. tabulā ir apskatāms paštestēšanas pieejas identificējamo kļūdu sadalījums pa kļūdu veidiem un tiem veltītā laika. Tabula satur šādas kolonnas:

- Testa punkts – testa punkta nosaukums, kas identificētu kļūdu;
- Skaits – skaits, cik kļūdas testa punkts identificētu;
- % no kopējā – testa punkta novēršamo kļūdu skaits procentos no visu testa punktu novēršamo kļūdu skaita;
- Stundas – stundas, cik veltītas šo testa punktu identificējamo kļūdu risināšanā;
- % no kopējā – stundu skaits procentos no kopējo stundu skaita, kas veltītas testa punktu identificēto kļūdu novēršanai.

5.10. tabula

Paštestēšanas pieejas identificējamo kļūdu sadalījums pa testa punktu veidiem

Testa punkts	Skaits	% no kopējā	Stundas	% no kopējā
Faila rezultāta testa punkts	59	8.02	150.03	4.56
Ievadlauka testa punkts	146	19.84	827.14	25.11
Lietojumprogrammas notikuma testa punkts	105	14.27	364.24	11.06
Salīdzināmās vērtības testa punkts	28	3.8	93.53	2.84
Sistēmas paziņojuma testa punkts	11	1.49	58.84	1.79
SQL vaicājuma rezultāta testa punkts	387	52.58	1799.96	54.65
Kopā:	**736**	**100**	**3293.74**	**100**



5.11. att. **Paštestēšanas pieejas identificējamo kļūdu sadalījums par testa punktu veidiem**

Ņemot vērā iegūto informāciju, autors secina, ka:

- Vairāk kā pusi no visām reģistrētām kļūdām spētu identificēt SQL vaicājuma rezultāta testa punkts. Testa punkts veic datu atlasīšanu no datu bāzes un to salīdzināšanu ar etalona vērtībām. Testa punkta pārsvars ir skaidrojams ar to, ka VVUS pamata mērķis ir datu saglabāšana un atskaišu sagatavošana no saglabātajiem datiem.

- Piekto daļu no paštestēšanas pieejas identificējamām kļūdām identificētu ievadlauka testa punkts. Testa punkts veic lauka vērtības salīdzināšanu ar etalona vērtību.
- Lietojumprogrammas testa punkts identificē visas sistēmas kļūdas, kas notiek pēc lietojumprogrammas darbības izpildes (piemēram, pogas nospiešana).

5.5. Secinājumi

Statistikas analīzes rezultātā ir gūta pārliecība, ka paštestēšanas ieviešana būtiski ļautu ietaupīt sistēmas izstrādes laiku un uzlabotu sistēmas kvalitāti. Kā to parāda analīze, tad ne vienmēr paštestēšanas pieeja spēj identificēt visas sistēmas kļūdas. Balstoties uz veikto analīzi, turpmākās darbības paštestēšanas pieejas attīstība tiks vērsta uz pašreizējās paštestēšanas pieejas neidentificējamo kļūdu veidu samazināšanu.

NOBEIGUMS

Darbā veikto pētījumu rezultātā autors ir guvis pārliecību, ka paštestēšanas pieeja sniedz virkni priekšrocību salīdzinājumā ar līdz šim apzinātiem un pieejamiem testēšanas rīkiem un ļauj būtiski uzlabot sistēmas kvalitāti, un ietaupīt laiku sistēmas izstrādes, testēšanas un uzturēšanas dzīves cikla posmos. Ne vienmēr programmatūras paštestēšana nosegs visu sistēmas funkcionalitāti, tomēr autors ir pārliecināts, ka tā aizstās būtisku daļu no kopējā testēšanas apjoma un būtiski uzlabos sistēmas kvalitāti.

Paveiktais parāda, ka paštestēšanas pieejas piedāvātās iespējas ir ne tikai līdzvērtīgas tām, ko piedāvā citi pasaulē atzīti testēšanas rīki, bet spēj piedāvāt arī tādas iespējas, ko citi pasaulē atzīti testēšanas rīki nepiedāvā, piemēram, testēšanu gan pēc baltās kastes, gan pēc pelēkās, gan pēc melnās kastes metodes, ārējo saskarņu testēšanu vai iespēju sistēmas lietotājam bez padziļinātām IT zināšanām veidot testus.

Paštestēšanas pieejas pašreizējā implementācija sevī iekļauj tikai pašu būtiskāko funkcionalitāti, lai reālos projektos varētu pārliecināties par paštestēšanas pieejas konceptuālu lietderību. Kā to parāda testēšanas rīku salīdzināšanas tabula (4.6. tabula), tad paštestēšanas rīkam pastāv vairāki attīstības virzieni:

- papildus jaunu testu automatizēšanas ietvara(-u) ieviešana;
- testu redaktora un žurnāla ieviešana;
- objektu pārlūka un validēšanas ieviešana;
- slodzes, stresa un, iespējams, arī citu testēšanas veidu ieviešana;
- atbalstāmo platformu un testējamo tehnoloģiju loka paplašināšana;
- tīmekļa lietojumprogrammu un servisu testēšanas ieviešana;
- iespēja papildināt rīka piedāvāto funkcionalitāti ar spraudņu un paplašinājumu palīdzību;
- [Bichevskii, Borzov, 1982] aprakstītās prioritāšu idejas ieviešana;
- integrācija ar programmatūras izpildes vides testēšanu [Rauhvargers, Bičevskis, 2008].

Papildinot paštestēšanas rīka funkcionalitāti ar iepriekš minētajām iespējām un papildinot to ar iespēju veikt paštestēšanu produkcijas vidē, kas pagaidām attīstīta tikai koncepcijas līmenī, autors tic, ka tas ļautu paštestēšanas rīkam iegūt stabilu vietu starp pasaulē pieejamajiem testēšanas rīkiem.

Paštestēšanas pieeja nākotnē varētu tikt veiksmīgi pielietota arī citos projektos, piemēram, iegultas programmatūras izstrādē, kur tradicionālie testēšanas atbalsta rīki ir visai grūti pielietojami.

Darba galvenie rezultāti un būtiskākie secinājumi:

- Darbā tiek piedāvāta, izstrādāta, aprobēta un novērtēta jauna testēšanas tehnoloģija/pieeja, kādu pašreiz nepiedāvā neviens no līdz šim apzinātiem un pieejamiem testēšanas atbalsta rīkiem;
- Paštestēšana neaizvieto tradicionālo programmatūras testēšanu, bet maina pašas testēšanas procesus, būtiski paplašinot programmētāja lomu programmatūras testēšanā;
- Paštestēšana prasa papildus darbu paštestēšanas funkcionalitātes iekļaušanai programmatūrā, kritiskās funkcionalitātes testu, un testēšanas procedūru izstrādei;
- Kā to parāda efektivitātes mērījumu statistika, ar paštestēšanas palīdzību būtiski tiek ietaupīts laiks uz esošās funkcionalitātes atkārtotu (regresa) testēšanu.
- Paštestēšana nodrošina to, ka nemainītās programmatūras funkcionalitāte saglabāsies, bet jaunā programmatūra strādās atbilstoši jauniem testiem, kurus ir apstiprinājis sistēmas lietotājs.
- Paštestēšanas pieeja piedāvā vairākas jaunas iespējas, ko principā nepiedāvā tradicionālie testēšanas atbalsta rīki:
    - sistēmas testēšanu pēc baltās kastes metodes;
    - iespēju testus veikt produkcijas vidē;
    - lietotājiem bez IT zināšanām uzkrāt testa piemērus;
    - ārējo saskarņu testēšanu;
    - pilnu sistēmas testēšanu (sākot ar datu ievadi un beidzot ar datu saglabāšanu datu bāzē).
- Paštestēšanas funkcionalitātes ieviešana ir noderīga inkrementālos izstrādes modeļos, īpaši sistēmām, kuras tiek attīstītas pakāpeniski un uzturētas daudzu gadu garumā, mazāk noderīga lineārajos izstrādes modeļos.
- Paštestēšanas funkcionalitāti vēlams iestrādāt jau programmatūras projektēšanas un izstrādes laikā, lai samazinātu izmaksas un jau sākotnēji nodrošinātu augstu sistēmas kvalitāti.
- Testa punkti būtiski atvieglo testu pierakstīšanu un automātisku izpildīšanu. Testa punkti nodrošina testa pierakstīšanu ērtā un viegli lasāmā veidā.

Rīkam vēl nepieciešama attīstība tieši tehniskās realizācijas ziņā. Darbā paveiktais parāda, ka paštestēšanas pieeja spēs veiksmīgi funkcionēt reālos projektos un dos būtisku ieguldījumu programmatūras izstrādē un IT jomā kopumā. Par to liecina tas, ka paštestēšanas mehānisma

izmantošanā ir ieinteresētas tās Latvijas bankas, kuras izmanto SIA „Datorikas institūts DIVI" izstrādāto Valūtu un Vērtspapīru uzskaites sistēmu, tādēļ šajā sistēmā jau šobrīd ir iestrādāts paštestēšanas mehānisms.

Paštestēšanas pieejas attīstības nākotnes plānos ietilpst klientu loka paplašināšana, izveidojot to par automatizētas testēšanas rīku tirgū pieejamu produktu, kas izmantojams dažādos projektos. Darba autors cer, ka tiks iegūti nepieciešamie līdzekļi šī paštestēšanas pieejas attīstībai.

Darbā veiktā pētījuma rezultāti publicēti vairākos (6) recenzētos izdevumos, prezentēti divās starptautiskās zinātniskās konferencēs. Par darba tēmu izstrādāti divi kursa darbi, viens bakalaura darbs un divi maģistra darbi.

LITERATŪRAS AVOTI

[ATI, 2011] **Automated Testing Institute (ATI).** Schedule & Selection Process. [Tiešsaiste]: Automated Testing Institute web site [Citēts: 16.01.2011]. Pieejams: http://www.automatedtestinginstitute.com/home/index.php?option=com_content&view=article&id=1283&Itemid=156

[ATI, 2012a] **Automated Testing Institute (ATI).** [Tiešsaiste]: Automated Testing Institute web site [Citēts: 04.05.2012]. Pieejams: http://www.automatedtestinginstitute.com/home/

[ATI, 2012b] **Automated Testing Institute (ATI).** Frameworks. [Tiešsaiste]: Automated Testing Institute web site [Citēts: 04.05.2012]. Pieejams: http://www.automatedtestinginstitute.com/home/index.php?option=com_content&view=article&id=69&Itemid=75

[ATI, 2012c] **Automated Testing Institute (ATI).** ATI Automation Honors. [Tiešsaiste]: Automated Testing Institute web site [Citēts: 04.05.2012]. Pieejams: http://www.automatedtestinginstitute.com/home/index.php?option=com_content&view=article&id=1262&Itemid=131

[Automated Software Testing Magazine, 2012] **Automated Software Testing Magazine.** [Tiešsaiste]: Automated Testing Institute web site [Citēts: 04.05.2012]. Pieejams: http://www.automatedtestinginstitute.com/home/index.php?option=com_conten%20t&view=article&id=1276&Itemid=122

[Automatic Testing Frameworks, 2011a] **Automatic Testing Frameworks.** [Tiešsaiste]: Automatic Testing Frameworks web site [Citēts: 19.04.2011]. Pieejams: http://www.automatictestingframeworks.com/?q=test_automation_framework

[Automatic Testing Frameworks, 2011b] **Automatic Testing Frameworks.** Ranorex. [Tiešsaiste]: Automatic Testing Frameworks web site [Citēts: 17.04.2011]. Pieejams: http://www.automatictestingframeworks.com/?q=ranorex

[Beizer, 1995] **Beizer, B.** *Black-box testing: techniques for functional testing of software and systems.* John Wiley & Sons Inc., 1995

[Bičevskis, 1998] **Bičevskis, J.** The Effictiveness of Testing Models. ***In:*** *Proc. of 3d Intern. Baltic Workshop "Databases and Information Systems"*, Riga, 1998

[Bicevskis *et. al.*, 2011] **Bicevskis, J., Cerina-Berzina, J., Karnitis, G., Lace, L., Medvedis, I., Nesterovs, S.** Practitioners View on Domain Specific Business Process Modeling. ***In:*** *Databases and Information Systems VI - Selected papers from 9th International Baltic*

Conference, DB&IS 2010 (Barzdins, J., Kirikova, M., eds.), IOS Press, vol. 224, 2011. pp.169-182

[Bichevskii, Borzov, 1982] **Bichevskii YY, Borzov YV.** Prioriteti v otladke bolsih programmnih sistem [Prioritātes lielu programmu sistēmu atkļūdošanā]. *Programmirovanie*, 1982, vol. 3, pp.31-34

[Bičevska, Bičevskis, 2007] **Bičevska, Z., Bičevskis, J.** Smart Technologies in Software Life Cycle. ***In:*** *Proceedings of Product-Focused Software Process Improvement. 8th International Conference, PROFES 2007, July 2-4, 2007* (Münch, J., Abrahamsson, P., eds.), Riga, Latvia, vol.4589/2007, 2007. pp.262-272

[Bičevska, Bičevskis, 2008a] **Bičevska, Z., Bičevskis, J.** Application of Smart Technologies in Software Development: Automated Version Updating. ***In:*** *Scientific papers, vol. 733* (Bārzdiņš, J., Freivalds, R.M., Bičevskis, J., eds.), University of Latvia, 2008. pp.24 - 37

[Bičevska, Bičevskis, 2008b] **Bičevska, Z., Bičevskis, J.** Self-testing: A New Testing Approach. ***In:*** *Proceedings of the Eighth International Baltic Conference Baltic DB&IS 2008, June 2-5, 2008, Tallinn, Estonia* (Haav, H.-M., Kalja, A. eds.), 2008. pp.179-189

[Bičevska, Bičevskis, 2008c] **Bicevska, Z., Bicevskis, J.** Applying Self-Testing: Advantages and Limitations. ***In:*** *Databases and Information Systems V - Selected Papers from the Eighth International Baltic Conference, DB&IS 2008, June 2-5, 2008, Tallinn, Estonia* (Haav, H.-M., Kalja, A., eds.), IOS Press, vol. 187, 2008. pp.192-202

[Bugzilla, 2012] **Bugzilla.** [Tiešsaiste]: Bugzilla web site [Citēts: 9.03.2012]. Pieejams: http://www.bugzilla.org/

[Cerina-Berzina *et. al.*, 2011] **Cerina-Berzina, J., Bicevskis, J., Karnitis, G.** Information systems development based on visual Domain Specific Language BiLingva. ***In:*** *Selected Papers from the 4th IFIP TC 2 Central and East Europe Conference on Software Engineering Techniques*, *CEE-SET 2009, Krakow, Poland,* LNCS, vol. 7054, Springer, 2011. pp.124-135

[Chengying *et. al.*, 2007] **Chengying M., Yansheng L., Jinlong Z.** Regression testing for component-based software via built-in test design. ***In:*** *Proceedings of the ACM symposium on Applied computing, March 11 - 15, 2007, Seoul, Korea*, 2007. pp.1416-1421

[Royce, 2012] **Royce, W., W.** Managing the development of large software systems. [Tiešsaiste]: Department of Computer Science web site [Citēts: 05.05.2012]. Pieejams: http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf

[Diebelis *et. al.*, 2009] **Diebelis, E., Takeris, V., Bičevskis, J.** Self-Testing - New Approach to Software Quality Assurance. ***In:*** *Proceedings of the 13th East-European Conference on*

Advances in Databases and Information Systems, ADBIS 2009, September 7-10, 2009, Riga, Latvia (Grundspenkis, J., Kirikova, M., Manolopoulos, Y., Morzy, T., Novickis, L., Vossen, G., eds.), 2009. pp.62-77

[Diebelis, Bičevskis, 2010] **Diebelis, E., Bičevskis, J.** An Implementation of Self-Testing. ***In:*** *Proceedings of the 9th International Baltic Conference on Databases and Information Systems, Baltic DB&IS 2010, July 5-7, 2010, Riga, Latvia* (Barzdins, J., Kirikova, M., eds.), 2010. pp.487-502

[Diebelis, Bičevskis, 2011] **Diebelis, E., Bičevskis, J.** Test Points in Self-Testing. ***In:*** *Databases and Information Systems VI - Selected papers from 9th International Baltic Conference, DB&IS 2010* (Barzdins, J., Kirikova, M., eds.), IOS Press, vol. 224, 2011. pp.309 - 321

[Diebelis, 2012] **Diebelis, E.** Efficiency Measurements of Self-Testing. ***In:*** *Scientific papers*. (pieņemta publicēšanai).

[Diebelis, Bičevskis, 2012] **Diebelis, E., Bičevskis, J.** Software Self-Testing. ***In:*** *Proceedings of the 10th International Baltic Conference on Databases and Information Systems, Baltic DB&IS 2012, July 8-11, 2012, Vilnius, Lithuania.* (pieņemta publicēšanai)

[FitNesse, 2011] **FitNesse.** Test Table Styles. [Tiešsaiste]: FitNesse web site [Citēts: 11.04.2011]. Pieejams: http://fitnesse.org/FitNesse.UserGuide.TestTableStyles

[FitNesse, 2012] **FitNesse.** The fully integrated standalone wiki, and acceptance testing framework. [Tiešsaiste]: FitNesse web site [Citēts: 14.04.1012]. Pieejams: http://fitnesse.org/

[HP STS, 2011] **HP Service Test Software (HP STS)**. [Tiešsaiste]: HP Unified Functional Testing software web site [Citēts: 09.05.2011]. Pieejams: https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24^1382_4000_100__

[HP UFTS, 2011a] **HP Unified Functional Testing Software (HP UFTS)**. Looking for Mercury Interactive? [Tiešsaiste]: HP Unified Functional Testing software web site [Citēts: 09.05.2011]. Pieejams: https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-10^36653_4000_100__&jumpid=reg_R1002_USEN#

[HP UFTS, 2011b] **HP Unified Functional Testing Software (HP UFTS)**. [Tiešsaiste]: HP Unified Functional Testing software web site [Citēts: 06.05.2011]. Pieejams: https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24^1352_4000_100__

[HP UFTS, 2012] **HP Unified Functional Testing Software (HP UFTS)**. Automate functional testing of transactions spanning multiple layers. [Tiešsaiste]: HP Unified Functional Testing software web site [Citēts: 14.04.2012]. Pieejams: http://www8.hp.com/us/en/software-solutions/software.html?compURI=1172957

[IBM, 2001] **International Business Machines Corporation.** *AUTONOMIC VISION & MANIFESTO.* [Tiešsaiste]: IBM web site, 2001 [Citēts: 02.03.2012]. Pieejams: http://www.research.ibm.com/autonomic/manifesto/

[IBM, 2011] **IBM.** IBM Help System. [Tiešsaiste]: IBM web site [Citēts: 19.04.2011]. Pieejams: http://publib.boulder.ibm.com/infocenter/rfthelp/v7r0m0/index.jsp?topic=/com.ibm.rational.test.ft.proxysdk.doc/topics/c_pr_architecture.html

[IBM RFT, 2012a] **IBM Rational Functional Tester (RFT).** Functional testing automation for quality driven software delivery. [Tiešsaiste]: IBM Rationa Functional Tester web site [Citēts: 14.04.2012]. Pieejams: http://www-01.ibm.com/software/awdtools/tester/functional/

[IBM RFT, 2012b] **IBM Rational Functional Tester (RFT).** Rational Functional Tester Features and Benefits.. [Tiešsaiste]: IBM Rationa Functional Tester web site [Citēts: 05.05.2012]. Pieejams: http://www-01.ibm.com/software/awdtools/tester/functional/features/index.html

[IEEE, 1990] **IEEE.** *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries.* IEEE, 1990.

[IndicThreads, 2012] **IndicThreads.** An Introduction to Software Test Automation. [Tiešsaiste]: IndicThreads web site [Citēts: 04.05.2012]. Pieejams: http://www.indicthreads.com/1329/an-introduction-to-software-test-automation/

[Kaner, 2006] **Kaner, C.** Exploratory Testing. ***In:*** *Assurance Institute Worldwide Annual Software Testing Conference*, Florida Institute of Technology, Quality, Orlando, FL, November 2006.

[Kephart, Chess, 2003] **Kephart, J., O., Chess, D., M.** The Vision of Autonomic Computing. *Computer Magazine*, 2003, vol. 36, pp.41-50

[Kephart, 2011] **Kephart, J., O.** Autonomic Computing: The First Decade. ***In:*** *ICAS 2011: Proceedings of the 8th International Conference on Autonomic Computing*, 2011. pp.1-2

[letonika.lv, 2012a] **letonika.lv.** Kontrolpunkts. [Tiešsaiste]: letonika.lv web site [Citēts: 04.05.2012]. Pieejams: http://www.letonika.lv/groups/default.aspx?r=1107&q=Kontrolpunkts&id=2071795&g=1

[letonika.lv, 2012b] **letonika.lv.** Testēšana. [Tiešsaiste]: letonika.lv web site [Citēts: 04.05.2012]. Pieejams: http://www.letonika.lv/groups/default.aspx?r=10331062&q=testing&cid=398581&&g=2

[letonika.lv, 2012c] **letonika.lv.** Akcepttests. [Tiešsaiste]: letonika.lv web site [Citēts: 04.05.2012]. Pieejams: http://www.letonika.lv/groups/default.aspx?r=10331062&q=acceptance%20test&cid=246761&&g=2

[letonika.lv, 2012d] **letonika.lv.** Veiktspējas testēšana. [Tiešsaiste]: letonika.lv web site [Citēts: 04.05.2012]. Pieejams: http://letonika.lv/groups/default.aspx?r=1107&q=veiktsp%C4%93jas%20test%C4%93%C5%A1ana&id=2069576&&g=1

[Lubiņa, 2012] **Lubiņa, D.** Programmu izstrāde ar Agile žiglāka un ražīgāka. [Tiešsaiste]: Exigen Services web site [Citēts: 05.05.2012]. Pieejams: http://www.exigenservices.lv/newsroom/exigen-services-in-the-news/09-01-08-2021

[Mihailovs, 2010] **Mihailovs, P.** *Paštestēšanas principu realizācija*. Bakalaura darbs, Latvijas Universitāte, Datorikas fakultāte, Rīga (2010)

[Pressman, 2000] **Pressman, R., S.** *Software Engineering - A Practioner's approach 5th edition.* The McGraw-Hill Companies Inc., 2000.

[Pressman, 2004] **Pressman, R., S.** *Software Engineering - A Practitioner's Approach. 6th edition.* The McGraw-Hill Companies Inc., 2004

[QTP, 2012] **QTP Tutorials & Interview Questions.** Functional Decomposition Framework. [Tiešsaiste]: QTP Tutorials & Interview Questions web site [Citēts: 04.05.2012]. Pieejams: http://qtp.blogspot.com/2008/10/functional-decomposition-framework.html

[Ranorex, 2012] **Ranorex.** Proffesional Test Automation Tools. [Tiešsaiste]: Ranorex web site [Citēts: 14.04.1012]. Pieejams: http://www.ranorex.com/

[Rauhvargers, 2008] **Rauhvargers, K.** On the Implementation of a Meta-data Driven Self Testing Model. ***In:*** *Software Engineering Techniques in Progress* (Hruška, T., Madeyski, L., Ochodek, M., eds.), Brno, Czech Republic, 2008. pp.153-166

[Rauhvargers, Bičevskis, 2008] **Rauhvargers, K., Bičevskis, J.** Automating the Software Environment Testing Process. ***In:*** *Proceedings of the Eighth International Baltic Conference Baltic DB&IS 2008, June 2-5, 2008, Tallinn, Estonia* (Haav, H.-M., Kalja, A., eds.), 2008. pp.155-166

[Rauhvargers, Bičevskis, 2009] **Rauhvargers, K., Bičevskis, J.** Environment Testing Enabled Software - a Step Towards Execution Context Awareness. ***In:*** *Databases and Information Systems V - Selected Papers from the Eighth International Baltic Conference, DB&IS 2008* (Haav, H.-M., Kalja, A., eds.), 2009. pp.169-179

[RealVNC, 2012] **RealVNC.** The RFB Protocol. [Tiešsaiste]: RealVNC web site [Citēts: 04.05.2012]. Pieejams: http://www.realvnc.com/docs/rfbproto.pdf

[SeleniumHQ, 2012] **SeleniumHQ.** Selenium Web Application Testing System. [Tiešsaiste]: SeleniumHQ web site[Citēts: 14.04.1012]. Pieejams: http://seleniumhq.org/

[SmartBear Software, 2012] **SmartBear Software.** TestComplete Automated Testing. [Tiešsaiste]: SmartBear Software web site [Citēts: 04.05.2012]. Pieejams: http://www.automatedqa.com/products/testcomplete/

[Spillner *et. al.*, 2007] **Spillner A., Linz T., Schaefer H.** *Software Testing Foundations: A Study Guide for the Certified Tester Exam.* Rocky Nook Inc, 2007

[Stylusinc, 2009] **Stylusinc.** [Tiešsaiste]: Stylusinc web site [Citēts: 24.05.2009]. Pieejams: http://www.stylusinc.com/Common/Concerns/SoftwareDevtPhilosophy.php

[T-Plan, 2012a] **T-Plan.** Image Based vs Object Oriented Testing. [Tiešsaiste]: T-Plan web site [Citēts: 04.05.2012]. Pieejams: http://www.t-plan.com/robot/docs/articles/img_based_testing.html

[T-Plan, 2012b] **T-Plan.** T-Plan Robot. [Tiešsaiste]: T-Plan web site [Citēts: 14.04.1012]. Pieejams: http://www.t-plan.com/robot/

[T-Plan, 2012c] **T-Plan.** T-Plan Robot Enterprise v2 Tutorial. [Tiešsaiste]: T-Plan web site [Citēts: 05.05.1012]. Pieejams: http://www.t-plan.com/robot/docs/tutorials/v2.0/index.html

[T-Plan, 2012d] **T-Plan.** T-Plan Robot Enterprise 2.3 Language Reference. [Tiešsaiste]: T-Plan web site [Citēts: 05.05.1012]. Pieejams: http://www.t-plan.com/robot/docs/v2.3ee/scripting/commref.html

[Takeris, 2009] **Takeris, V.** *Paštestēšana banku sistēmās*. Maģistra darbs, Latvijas Universitāte, Datorikas fakultāte, Rīga (2009)

[Testing Geek, 2012] **TestingGeek.** FitNesse Introduction. [Tiešsaiste]: Testing Geek web site [Citēts: 04.05.1012]. Pieejams: http://www.testinggeek.com/index.php/testing-tools/test-execution/95-fitnesse-introduction

[Utting, 2008] **Utting, M.** The Role of Model-Based Testing. ***In:*** *Verified Software: Theories, Tools, Experiments* (Meyer, B., Woodcock, J., eds.), LNCS, vol. 4171, Springer, 2008. pp.510-517

[Webopedia, 2011] **Webopedia.** Built-In Self-Test. [Tiešsaiste]: Webopedia web site [Citēts: 22.01.2011]. Pieejams: http://www.webopedia.com/TERM/B/Built_In_Self_Test.html

[Webopedia, 2012] **Webopedia.** Regression testing. [Tiešsaiste]: Webopedia web site [Citēts: 04.05.2012]. Pieejams: http://www.webopedia.com/term/r/regression_testing.html

[Wikipedia, 2012a] **Wikipedia.** Virtual Network Computing. [Tiešsaiste]: Wikipedia web site [Citēts: 04.05.2012]. Pieejams: http://en.wikipedia.org/wiki/Virtual_Network_Computing

[Wikipedia, 2012b] **Wikipedia.** Test automation framework. [Tiešsaiste]: Wikipedia web site [Citēts: 04.05.2012]. Pieejams: http://en.wikipedia.org/wiki/Test_automation_framework

[Wikipedia, 2012c] **Wikipedia.** Keyword-driven testing. [Tiešsaiste]: Wikipedia web site [Citēts: 04.05.2012]. Pieejams: http://en.wikipedia.org/wiki/Keyword-driven_testing

[Wikipedia, 2012d] **Wikipedia.** Wiki. [Tiešsaiste]: Wikipedia web site [Citēts: 04.05.2012]. Pieejams: http://en.wikipedia.org/wiki/Wiki

[Wikipedia, 2012e] **Wikipedia.** FitNesse. [Tiešsaiste]: Wikipedia web site [Citēts: 04.05.2012]. Pieejams: http://en.wikipedia.org/wiki/FitNesse

[Wikipedia, 2012f] **Wikipedia.** API. [Tiešsaiste]: Wikipedia web site [Citēts: 04.05.2012]. Pieejams: http://en.wikipedia.org/wiki/API

[Wikipedia, 2012g] **Wikipedia.** Test-driven development. [Tiešsaiste]: Wikipedia web site [Citēts: 12.05.2012]. Pieejams: http://en.wikipedia.org/wiki/Test-driven_development

[Wikipedia, 2012h] **Wikipedia.** Agile software development. [Tiešsaiste]: Wikipedia web site [Citēts: 12.05.2012]. Pieejams: http://en.wikipedia.org/wiki/Agile_software_development

[Verify/ATI, 2012] **Verify/ATI.** VerifyATI Testing & Test Automation Conference. [Tiešsaiste]: Verify/ATI web site [Citēts: 02.03.2012]. Pieejams: http://www.verifyati.com

[Vizulis, 2011] **Vizulis, V.** *Paštestēšanas pieejas un testēšanas rīku salīdzinājums*. Maģistra darbs, Latvijas Universitāte, Datorikas fakultāte, Rīga (2011)

[Vizulis, Diebelis, 2012] **Vizulis, V., Diebelis, E.** Self-Testing Approach and Testing Tools. ***In:*** *Scientific papers*. (pieņemta publicēšanai).