PROGRAMMATŪRAS TESTĒŠANAS ROKASGRĀMATA VADĪTĀJIEM

Versija 1.0

RĪGA - 2014

SATURS

1. IEVADS

Mūsdienās labas un kvalitatīvas programmatūras izstrādes pamatnosacījums ir programmatūras testēšana, kuras galvenais mērķis ir atrast problēmas programmatūrā un palīdzēt nodrošināt kvalitatīvas programmatūras izstrādi. Ar testēšanas palīdzību, piemēram, var noteikt, vai komplicēts algoritms strādā atbilstoši tā sākotnējai iecerei, vai sistēma spēj izpildīt tai izvirzītās slodzes prasības vai arī iespējams veikt pārbaudi pirms sistēmas pieņemšanas ekspluatācijā. Programmatūras testēšana dod iespēju vadītājiem izprast programmatūras gatavību, apzināties riskus, kas saistīti ar programmatūras palaišanu ekspluatācijā, lai pieņemtu pamatotus lēmumus un izvairītos no dažādām negaidītām un nepatīkamām situācijām, kas saistītas ar programmatūras problēmu konstatēšanu to lietojot jau klientiem.

Programmatūras testēšanas rokasgrāmata vadītājiem ir veidota kā materiāls ar pamatinformāciju par programmatūras testēšanu. Lasītājam nav nepieciešamas priekšzināšanas par programmatūras izstrādi un testēšanu. Rokasgrāmatā vairāk tiek aplūkoti jautājumi, kas saistīti ar programmatūras izstrādes un testēšanas procesu organizāciju, mazāk pievēršoties testēšanas tehniskajām metodēm. Par katru jautājumu informācija ir īsa, lai rokasgrāmata kopumā būtu ar mazu apjomu un tādēļ ērtāk pārskatāma. Rokasgrāmatā ir dotas norādes uz avotiem, ko izmantot, ja ir nepieciešamība un vēlme uzzināt vairāk par konkrēto jautājumu. Grāmatas beigās ir dots saraksts ar avotiem, ko var izmantot, lai papildus iegūtu informāciju par programmatūras testēšanu.

Vadītājiem priekšstata iegūšanai par programmatūras testēšanas nepieciešamību un testēšanas nozīmi kvalitatīvas programmatūras izstrādē īpaši rekomendējam šādas rokasgrāmatas sadaļas: „8.1. Testēšanas stratēģija” un „9. Testēšanas komanda”.

Profesionāļiem rekomendējam sadaļas: „3. Testēšanas veidi”, „4. Testēšanas pieejas”, „5. Testēšanas vieta programmatūras izstrādes dzīves ciklā”, „6. Testēšanas līmeņi”, kurās ietverta informācija par testēšanas vietu programmatūras izstrādes procesā un tehniska rakstura informācija par testēšanas pamatjautājumiem.

2. PAMATJĒDZIENI

2.1. Kas ir programmatūras testēšana

Testēšana identificē problēmas programmatūrā un sniedz informāciju par programmatūru un riskiem, kas saistīti ar programmatūras laišanu tirgū.

Testēšana var būt dažāda. Ir divas svarīgākās pieejas:

- „tradicionālā" pieeja – testēšana balstās uz prasībām un specifikācijām, testēšana tiek veikta „sistemātiski", kas dažkārt noved pie liela apjoma dokumentācijas izveidošanas. Tiek veikta virkne aktivitāšu (testēšanas plānošana, testpiemēru specificēšana, testpiemēriem nepieciešamo testēšanas datu izveide, testpiemēru izpilde ar testējamo programmatūru). Šī pieeja prasa ieguldīt darbu (un resursus) pirms tiek atklāta pirmā problēma programmatūrā;
- „spējā" pieeja [CBC01] – testēšana tiek balstīta uz risku analīzi un programmatūras kontekstu, pragmatiski izvērtējot, kādas testēšanas aktivitātes veikt pirms programmatūras nodošanas ekspluatācijā.

„Tradicionālā" jeb „sistemātiskā" pieeja vairāk ir piemērota lieliem vai ilgiem projektiem, testēšanas komandu iesaistot projektā relatīvi agri, dažkārt jau projektēšanas fāzē. Tas ļauj veikt programmatūras detalizētu analīzi un izstrādāt testpiemērus ilgākam laika periodam. Šī pieeja prasa pakāpeniskus izstrādes ciklus.

„Žiglā" pieeja ir piemērotāka mazāka izmēra programmatūrai un īsākiem laika periodiem, tā ir pielietojama arī žiglās programmatūras dzīves cikla gadījumā.

Testēšana ir aktivitāšu kopums, kuru uzdevums ir identificēt problēmas programmatūrā, novērtēt tās kvalitātes pakāpi, kā arī palīdzēt nodrošināt lietotāju apmierinātību ar programmatūru.

2.2. Kas ir kvalitāte

Kvalitāte ir pakāpe, kādā komponente, sistēma vai process atbilst noteiktajām prasībām un/vai lietotāju/klientu vajadzībām un gaidām.

Programmatūras kvalitāti raksturo funkcionalitāte, uzticamība, lietojamība, produktivitāte, uzturamība un pārvietojamība [SLS07].

Programmatūras funkcionalitātes kvalitāti izsaka tās piemērotība veiktajiem uzdevumiem un lietotāju/klientu mērķiem, pareizība un precizitāte, spēja sadarboties ar citām sistēmām un drošība.

Programmatūras uzticamība ir tās spēja strādāt bez kļūdām zināmu laika periodu zināmos apstākļos.

Programmatūras lietojamību izsaka tās piemērotība lietotājam no efektivitātes, produktivitātes un lietderīguma viedokļa. Lietojamību vērtējot, ņem vērā programmatūras saprotamību, apgūstamību, lietošanas ērtumu un atraktivitāti.

Programmatūras produktivitāti izsaka tās spēja nodrošināt atbilstošu veiktspēju (veiktspēja - laiks, kādā tā reaģē uz lietotāju pieprasījumiem) ar konkrētiem resursiem un noteiktos apstākļos.

Programmatūras uzturamība raksturo, cik viegli to būs papildināt ar jaunu funkcionalitāti un labot problēmas esošajā funkcionalitātē.

Vērtējot programmatūras pārvietojamību, pārbauda tās prasības un vajadzības, kas ir nepieciešamas, lai to uzliktu vidē, kur tai būs jādarbojas – klienta organizācijā ar tās datortehniku, programmatūru, kas tiek lietota paralēli testējamajai programmatūrai, un datortīklu.

2.3. Kas ir problēma (kļūda)

Testēšanā izšķir terminus problēma, kļūme, defekts un kļūda.

Kļūda (error) ir cilvēka darbība, kas izraisa defektu.

Defekts (defect) ir trūkums, bojājums (piem., kādā precē, izstrādājumā), novirze no kvalitātes normām vai standartiem [AkadTerm]. Defekts ir cilvēka darbības rezultāts, kas piemīt testējamajai programmatūrai.

Kļūme (failure) ir situācija, kad datu apstrādes sistēma vai kāda tās sastāvdaļa bojājumu dēļ daļēji vai pilnīgi zaudē spēju izpildīt tai paredzētās funkcijas. Kļūme ir procesa izpildīta defekta rezultāts (neatkarīgi no tā, vai process ir automatizēts, vai nē) [AkadTerm].

Problēma ir programmatūras darbības neatbilstība tam, ko no tās sagaida lietotājs.

Kļūda ir programmētāja darbība, kā rezultātā viņš uzraksta programmu ar defektu. Izpildot defektīvu programmatūru, ir redzama kļūme vai problēma, piemēram, negaidīta programmatūras uzvedība.

Defekti var būt ne tikai programmā, bet arī dokumentācijā, kas ir programmatūras sastāvdaļa. Liela daļa programmatūras problēmu sakņojas tās prasību, specifikāciju defektos.

Problēma ne vienmēr ir kļūdas izpausme. Lietotāja saskatīta problēma var norādīt:

1) uz kļūdām programmētāju darbā, nekorekti realizējot programmatūras specifikācijā prasīto;
2) uz defektiem programmatūras specifikācijā – pretrunām starp prasībām, iztrūkstošām prasībām, neviennozīmīgi interpretējamām prasībām;
3) uz nepieciešamību attīstīt programmatūru tālāk, veidojot jaunu funkcionalitāti.

2.4. Testēšana un kvalitātes nodrošināšana

Kvalitātes nodrošināšana nozīmē, ka procesi organizācijā tiek veikti un pielietoti korekti, atbilstoši nozares labākajai vai rekomendētajai praksei. Nepārtraukti procesu uzlabojumi uzlabo to efektivitāti un lietderīgumu [Per06]. Kvalitātes nodrošināšanas viens no mērķiem ir augstāku izstrādes un testēšanas brieduma līmeņa sasniegšana.

Pasaulē ir izveidoti testēšanas procesu izveides ietvari, piemēram, *Test Maturity Model* (TMM) , *Test Process Improvement* (TPI) modelis un *Test Improvement Model* (TIM). TMM [BHS+01] ietvarā ir vairāki līmeņi, kas parāda testēšanas brieduma pakāpi. Katrā līmenī ir vairāki testēšanas brieduma mērķi, kurus organizācija var izmantot gan kā attīstības mērķus, gan arī kā novērtēšanas modeli, lai iegūtu izpratni par savu situāciju testēšanas jomā.

TPI [KP99] balstās uz brieduma līmeņa novērtējumu dažādiem svarīgākajiem aspektiem, piemēram, dzīves cikla modelim vai metrikām, kā arī esošā brieduma līmeņa salīdzināšanu pret līmeņu prasībām. Tam piemīt arī ierobežojumi, īpaši mērogojamības ziņā [FD07] un pielietošanā praksē [Jun09].

TIM [ESU97] atbilstošā izstrāde koncentrējas uz ietvaru ar līmeņiem un svarīgākajiem virzieniem, kā arī pašnovērtējuma procedūru. TIM modelis ļauj neatkarīgi novērtēt esošo situāciju svarīgākajos testēšanas virzienos un dod novērtētajai organizācijai tās tā brīža testēšanas procesu “karti”.

2.5. Kas veic testēšanu

Testēšanu dažādos programmatūras izstrādes dzīves cikla periodos var veikt dažādi cilvēki – izstrādātāji parasti veic vienībtestēšanu, lietotāji – akcepttestēšanu, izstrādes komandā iekļauti testētāji vai ārējie jeb neatkarīgie testētāji – integrācijas un sistēmtestēšanu.

Ja problēmas identificē lietotājs ikdienas lietošanas procesā, tas nenozīmē, ka lietotājs ir kļuvis par testētāju.

Ja hakeris meklē iespēju uzlauzt programmatūru, viņš nav tās drošības testētājs, viņš ir uzbrucējs.

2.6. Kāpēc nav iespējama pilnīga notestēšana

Nav iespējams pilnībā notestēt programmatūru vai notestēt pilnīgi visu [KFN99].

Piemēram, ja testē mazu kalkulatoru, „notestēt visu" nozīmē pārbaudīt to ar visām iespējamām ievadvērtību, darbību un konfigurāciju (programmatūras un datortehnikas) kombinācijām. Tas nozīmē tik lielu testpiemēru skaitu, ka praktiski nav reāli tos visus izpildīt. Tāpēc pilnībā notestēt var tikai ļoti triviālu programmatūru, kas ir ļoti rets gadījums.

Tādēļ testēšanā ir nepieciešams saprātīgi samazināt izpildāmo testpiemēru skaitu tā, lai no vienas puses sasniegtu testēšanas mērķus un no otras puses darbs būtu reāli paveicams dotajā laikā un ar dotajiem resursiem. Izmantojot risku analīzi, tiek lemts, kurus testpiemērus izstrādāt un kurus izpildīt. Testēšanas komandas uzdevums ir samazināt ar programmatūru saistītos riskus.

2.7. Kad uzsākt testēšanu

Testēšanu ir vēlams izsākt iespējami agri programmatūras izstrādes dzīves ciklā [KBP02]. Jo vēlāk tiek atrasta problēma, jo dārgāk ir to izlabot. Ja problēmu neidentificē jau programmatūras projektēšanas laikā, tad tiek patērēts laiks un resursi:

1) lai noprogrammētu problemātisko projektējumu;
2) lai problēmu atrastu, programmatūru testējot;
3) lai kļūdu izlabotu, kad tā atrasta;
4) un visbeidzot, lai vēlreiz notestētu, vai patiešām problēma ir izlabota un vai labojot nav nejauši radušās jaunas problēmas programmatūrā Tāpat tiek novērsti iespējamie zaudējumi, kas problēmas dēļ varētu būt radušies klientam, ja problēma tiktu atrasta jau programmatūras ekspluatācijas laikā.

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

Ja testēšanu veic secīgi pa līmeņiem, piemēram, vienībtestēšana -> integrācijas testēšana -> sistēmtestēšana -> akcepttestēšana, tad katrā nākošajā līmenī testēšanu uzsāk tad, kad ir pabeigta testēšana iepriekšējā līmenī.

2.8. Kad beigt testēt

Testēšanu beidz, ja:

- zināmā laika periodā atrastās problēmas pēc savas nozīmības un smaguma ir relatīvi nesvarīgas, salīdzinot ar to atrašanai veiktās testēšanas izmaksām;
- veicot testēšanu pa līmeņiem – ja ir sasniegti nosacījumi, kas ir testēšanas plānošanas laikā noteikti kā testēšanas beigu nosacījumi šajā līmenī [Bla04].

Testēšanas beigšanas nosacījumus izstrādā atbilstoši testēšanas mērķiem. Ja kāds no nosacījumiem nav izpildīts, tiek veidoti papildus testpiemēri, lai šo nosacījumu izpildītu.

Detalizēti analizējot programmatūru, ir iespējams identificēt programmatūras apgabalus, ko nevar notestēt, jo, piemēram, tās ir izņēmuma nesasniedzamas situācijas.

2.9. Testēšana un atkļūdošana

Testēšana nozīmē identificēt vienu vai vairākus raksturlielumus, kas saistīti ar programmatūru. Katram testpiemēram ir:

- mērķis (raksturlielums, ko novērtē);
- veids, kā novērtē raksturlielumu (noteikta procedūra);
- aktivitāte (testpiemēra procedūras izpilde);
- rezultāts (vērtējamā raksturlieluma atbilstība vai neatbilstība sagaidāmajam).

Ja raksturlielums neatbilst sagaidāmajam, runā par „problēmu”. Problēma ir kļūdaina programmatūras koda izpildes rezultāts. Lai novērstu problēmu, ir nepieciešams atrast kļūdu programmatūrā un izlabot to.

Kļūdu programmatūrā atrod un izlabo izstrādātāji un šo procesu sauc par atkļūdošanu. Atkļūdošana pēc būtības nav testēšanas procesa sastāvdaļa, bet reāli dzīvē cilvēki bieži veic vienlaikus dažādas funkcijas, īpaši tas ir raksturīgi projektos, kuros strādā spējā programmatūras izstrādes stilā.

3. TESTĒŠANAS VEIDI

3.1. Manuālā testēšana un automatizētā testēšana

Manuālā testēšana nozīmē, ka testēšanu veic manuāli – cilvēki vada ievaddatus programmatūrā un veic nepieciešamās darbības ar programmatūru līdzīgi kā to darīs lietotāji.

Automatizētā testēšana nozīmē, ka testpiemēri tiek izpildīti un novērtēti automatizēti.

3.2. Funkcionālā testēšana un nefunkcionālā testēšana

Funkcionālā testēšana nozīmē, ka tiek pārbaudīta programmatūras funkcionalitāte – darbības, ko tā veic, piemēram, ļauj veikt pārskaitījumus internetbankā, nopirkt preci internetveikalā vai parāda dokumentu un ļauj to izdrukāt. Vairumā gadījumu programmatūras funkcijas ir aprakstītas tās prasībās un/vai specifikācijās.

Nefunkcionālā testēšanā pārbauda tādus programmatūras raksturlielumus, kas ir svarīgi programmatūras lietošanā, bet kas tiešā veidā neietekmē tās funkcionālos rezultātus. Nefunkcionālās testēšanas piemēri ir:

- lietojamības testēšana – programmatūras lietošanas ergonomiskuma novērtēšana;
- veiktspējas testēšana – pārbauda, līdz kādai maksimālajai noslodzei programmatūras spēj nodrošināt savu darbību. Noslodzi var veidot – vienlaikus pieslēgušos lietotāju skaits, vienlaikus veicamo operāciju skaits, datu bāzes aizpildījuma apjoms, utt..

3.3. Statiskā testēšana un dinamiskā testēšana

Dinamiskā testēšana nozīmē programmatūras testēšanu, to darbinot.

Statiskā testēšana [Vee02] ir programmatūras pārbaude, to nedarbinot. Statiskās testēšanas piemēri ir – pārvaldības apskates (management reviews, tehniskās apskates (technical reviews), koda inspekcijas, caurskates (walk-throughs) un auditi.

Pārvaldības apskašu rezultātā tiek sekots līdzi darbu progresam, to izpildes atbilstībai plāniem un laika grafikiem ar mērķi novērtēt pārvaldības pieeju efektivitāti un to atbilstību noteiktajiem mērķiem.

Tehnisko apskašu gaitā tiek vērtēta programmatūra kā produkts, vai tajā nav neatbilstības ar pielietojamajiem standartiem un noteiktajām specifikācijām. Tajās vērtē, piemēram, programmatūras specifikācijas un prasības, dokumentu, testēšanas un lietotāja dokumentāciju, programmatūras versijas sagatavošanas procedūras, instalācijas procedūras, utt.

Inspekciju mērķis ir atrast kļūdas programmatūrā, pārbaudot, vai, piemēram,:

- programmatūra atbilst tās specifikācijām un prasībām;
- programmatūra atbilst noteiktajiem kvalitātes raksturlielumiem;
- pārbauda pirmkodu un algoritmus.

4. TESTĒŠANAS PIEEJAS

4.1. Melnās kastes testēšana

Melnās kastes testēšana [Bei95] notiek, vadoties no programmatūras specifikācijām un citu informāciju par to, bet neizmantojot informāciju par programmatūras pirmkodu.

Viena no populārākajām melnās kastes testēšanas metodēm ir programmatūras ievaddatu sadalīšana ekvivalences klasēs jeb grupās, pieņemot, ka vienā grupā esošie dati izraisīs līdzīgus programmatūras rezultātus un uzvedību. Testēšanā izmanto vienu vai dažus pārstāvjus no katras ekvivalences klases.

Cita metode – speciālo un robežvērtību testēšana – izmantojot specifikāciju, konstatē, kuras ievadvērtības ir izņēmuma gadījumi vai kuras ir „uz robežas", piemēram, ja ievadlaukā var ievadīt vērtības no 1 līdz 10, tad robežvērtības ir 1 un 10.

Tāpat var plānot testēšanu pēc principa „iziet visas izvēlnes", „izkustināt visas vadīklas dialoglogā."

4.2. Baltās kastes testēšana

Baltās kastes testēšanas metodes izmanto informāciju par programmatūras pirmkodu – tās struktūru jeb arhitektūru.

Piemēram, ja testēšana tiek veikta vienībtestēšanas līmenī, tad testkomplekts tiek plānots tā, lai tiktu izdarbinātas visas programmatūras pirmkodā rindiņas, izvēļu zari un nosacījumi.

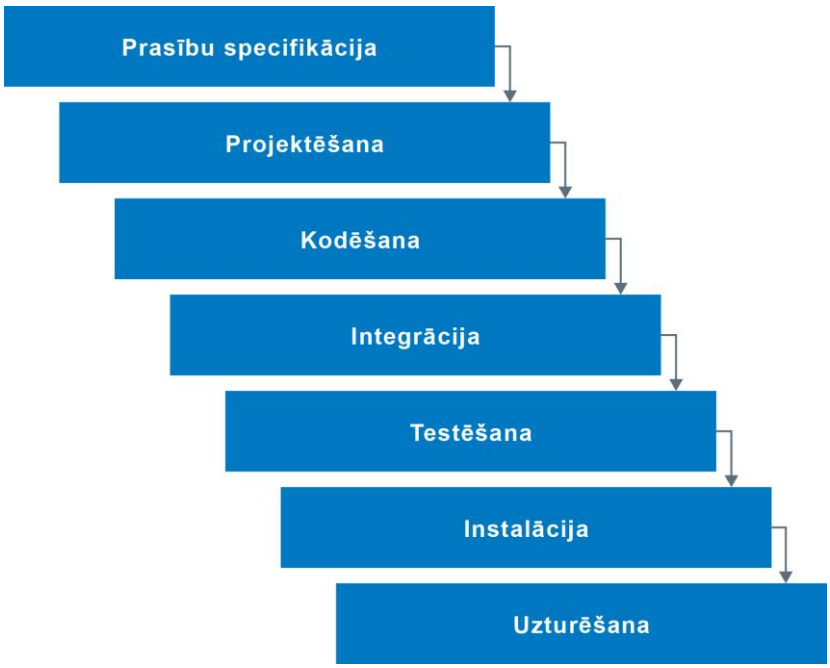
Struktūras testēšanas paveikšanas līmeni mēra ar „pārklājumu" – cik liels procents koda ir izdarbināts testēšanas laikā.

4.3. Pelēkās kastes testēšana

Pelēkās kastes testēšanā izmanto gan baltās, gan melnās kastes testēšanas principus [Cав01]. Piemēram, plāno testpiemērus, izmantojot gan specifikāciju, cenšoties noklāt visas prasības, gan arī cenšas izdarbināt visas pirmkoda rindiņas.

5. TESTĒŠANAS VIETA PROGRAMMATŪRAS IZSTRĀDES DZĪVES CIKLĀ

5.1. Pamata ūdenskrituma modelis



Ūdenskrituma modelim raksturīgi, ka programmatūras izstrāde notiek secīgi pa posmiem. Vispirms tiek izstrādātas programmatūras prasības, izveidojot programmatūras specifikāciju. Kad lietotāji vai pasūtītāji apstiprinājuši specifikāciju, uz tās pamata tiek veikta programmatūras projektēšana.

Programmatūras projektējums savukārt kļūst par bāzi tās kodēšanas procesam, kad programmētāji sāk reāli rakstīt programmatūras pirmkodu. Parasti visa programmatūra tiek sadalīta mazākās daļās jeb moduļos, ko atkal var sadalīt sīkāk, tā turpinot, kamēr katrs programmētājs vai programmētāju grupa var programmēt savu daļu. Programmētāji veic pirmā līmeņa testēšanu jeb vienībtestēšanu, pārbaudot, vai viņu uzrakstītie moduļi vai programmatūras daļas strādā atbilstoši specifikācijai.

Integrācijas fāzē programmētāji „liek kopā" programmatūras daļas jeb integrē tās, nepieciešamības gadījumā pieprogrammējot klāt vēl nepieciešamo, lai tās sadarbotos atbilstoši specifikācijai. Arī šajā fāzē programmētāji veic testēšanu, lai noskaidrotu, vai savienotās funkcijas darbojas. To sauc par integrācijas testēšanu.

Pēc tam seko programmatūras testēšanas fāze. Testēšanas fāzē raksturīgi, ka, pirmkārt, testēšanu parasti veic nevis programmētāji, bet testētāji, otrkārt, testētāji parasti neveic vienībtestēšanu, bet pamatā koncentrējas uz lielāka apjoma integrācijas testēšanu,

pārbaudot jau vairāk dažādu moduļu sadarbību, kā arī uz sistēmtestēšanu, kad pārbauda jau visas programmatūras darbību kopumā.

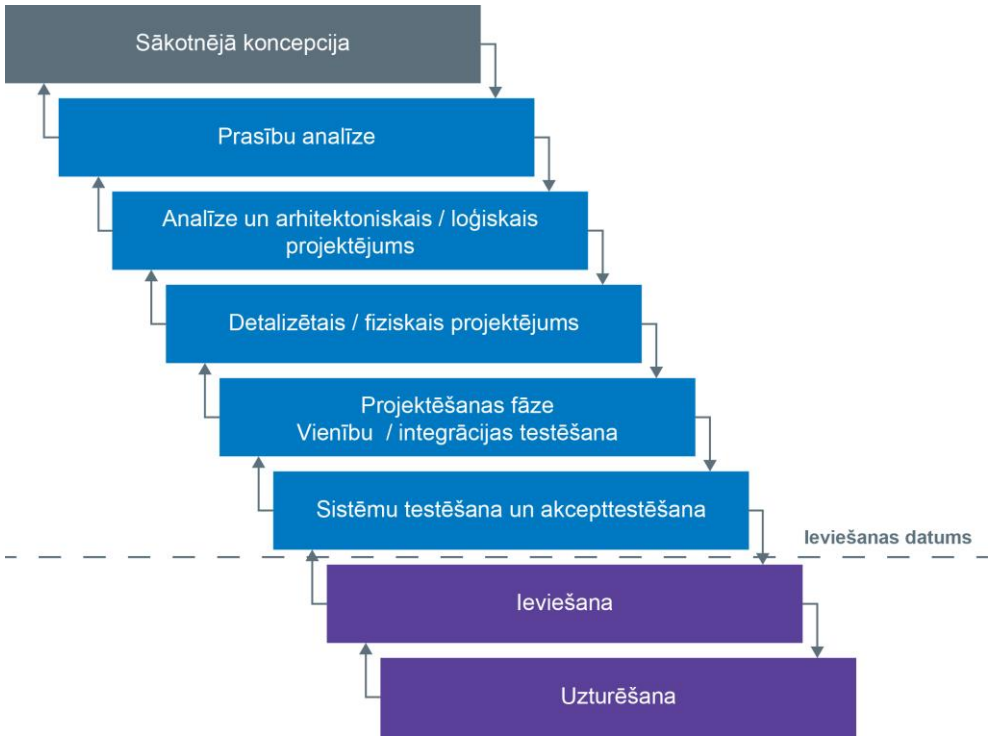
Ūdenskrituma modeļa testēšanas fāzē notiek arī jaunizveidotās programmatūras akcepttestēšana, kuras laikā tās pasūtītāji un, ja iespējams, lietotāji pārbauda tās atbilstību prasībām.

Instalācijas fāzē programmatūrai tiek izveidota instalācija un tā tiek instalēta uz atbilstošas datortehnikas. Uzturēšanas fāzē programmatūrai tiek labotas atrastās problēmas un nepieciešamības gadījumā tā tiek papildināta ar jaunu funkcionalitāti. Uzturēšanas fāzē notiek gan jaunās funkcionalitātes vai veikto labojumu vienībtestēšana, integrācijas un nepieciešamības gadījumā sistēmtestēšana un akcepttestēšana. Uzturēšanas fāzei ir raksturīga regresa testēšana, kad tiek pārbaudīts, vai pēc jauno labojumu vai piepildinājumu pievienošanas programmatūrai pārējā tās funkcionalitāte vēl darbojas atbilstoši specifikācijai.

Ūdenskrituma modelis tipiski tiek lietots lielu sistēmu izstrādei lielās organizācijās ar diezgan augstu birokrātisma līmeni, kur ir svarīgi pēc katra izstrādes posma parakstīt atbilstošus dokumentus par to pabeigšanu. Pēc ūdenskrituma modeļa darbs norit arī situācijās, kad dažādas fāzes izpilda dažādi ārpakalpojumu sniedzēji.

Lielākais ūdenskrituma modeļa trūkums ir fakts, ka sākotnējās fāzēs radušās kļūdas turpmākajās fāzēs tiek turpināts iestrādāt programmatūrā, rezultātā bieži iegūstot programmatūru, kas neatbilst gaidītajam. Kad kļūdas ir atrastas, tās var būt neiespējami vai ļoti dārgi labojamas, jo parasti tiek atrastas ļoti vēlu – testēšanas vai pat tikai uzturēšanas fāzēs.

5.2. Atvasinātais ūdenskrituma modelis



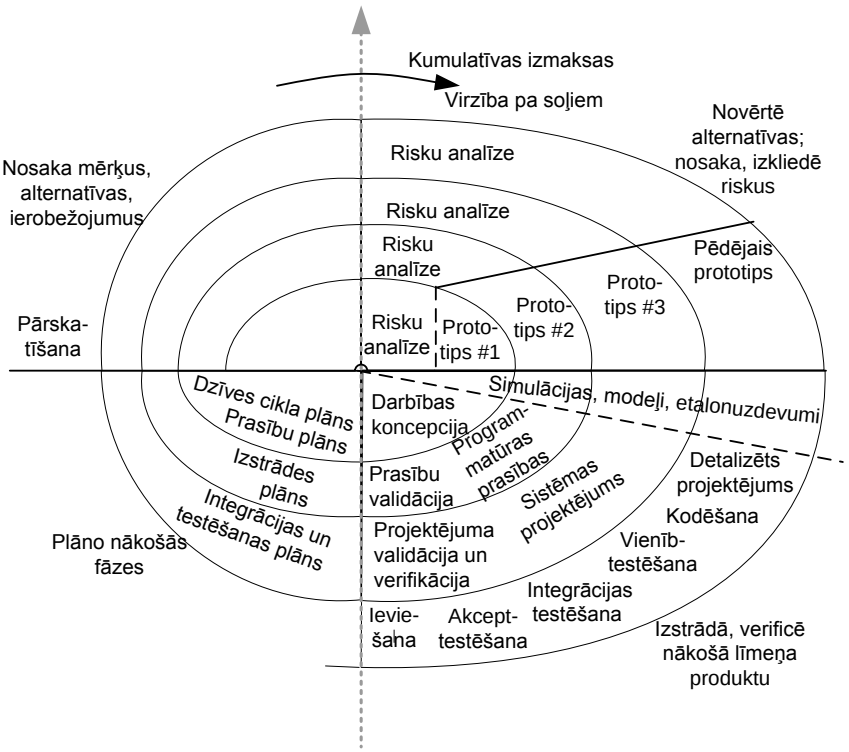
Attēlā dotajā atvasinātajā ūdenskrituma modelī [Jor02] ir mazliet sīkāk sadalītas programmatūras izstrādes fāzes nekā iepriekšējā nodaļā minētajā ūdenskrituma modelī, bet tie ir divi no iespējamiem fāžu sadalījuma modeļiem. Abos gadījumos ir izcelti svarīgākie programmatūras izstrādes dzīves cikla posmi.

Atvasinātais ūdenskrituma modelis pēc būtības atšķiras no vienkāršā ūdenskrituma modeļa ar to, ka nepieciešamības gadījumā ir iespējams atgriezties iepriekšējās fāzēs.

Piemēram, ja analīzes un arhitektoniskā/loģiskā projektējuma veidošanas laikā tiek konstatētas būtiskas nepilnības vai pretrunas programmatūras specifikācijā, tad izstrāde var atgriezties atpakaļ prasību analīzes fāzē, lai precizētu prasības.

Modeļa priekšrocība ir tā, ka, ja problēma vai kļūda ir atrasta, ir iespējams atgriezties iepriekšējā fāzē un mēģināt to izlabot.

5.3. Spirālveida modelis



Spirālveida modelis ir ūdenskrituma modeļa jauns attīstības pakāpiens. Programmatūra tiek attīstīta cikliski, Katra cikla sākumā tiek identificēti izstrādes mērķi, tiek analizētas iespējamās alternatīvas un ierobežojumi. Pēc tam seko risku analīze, projektēšana un detalizēta prototipa izstrāde, komponenšu kodēšana un testēšana, to integrācija un akcepttestēšana.

Spirālveida modelis paredz, ka katras iterācijas vai cikla laikā netiek mainītas izstrādājamās programmatūras prasības.

5.4. Inkrementālais modelis

Inkrementālajam modelim ir raksturīgs, ka sākotnēji tiek izveidotas dažas komponentes, kurām izstrādes grupa pēc tam pievieno jaunas un jaunas komponentes, kamēr izstrāde ir pabeigta.

Inkrementālo modeli var realizēt divos veidos:

- kad izstrādājamā programmatūra tiek sadalīta pa daļām, kuras pakāpeniski tiks izstrādātas pirms projektējuma fāzes, piemēram, prasību izstrādes laikā;
- pakāpeniski izstrādājamās daļas tiek iecerētas projektēšanas laikā, lai sekmētu izstrādi. Ir iespējams, ka pēc tam, kad programmatūras tiek sadalīta daļās, katra no

tām tiek izstrādāta un testēta atsevišķi, bet pirms programmatūras nodošanas ekspluatācijā tās tiek savstarpēji saintegrētas.

Inkrementālajā izstrādē parasti pamatkomponente tiek izstrādāta atbilstoši kādam no secīgajiem modeļiem – kādai ūdenskrituma, V vai W modeļu variācijām. Pārējās papildus komponentes tiek izstrādātas atbilstoši kādam no secīgajiem vai iteratīvajiem modeļiem.

Dažkārt programmatūras, kas tiek izstrādāta pēc inkrementālā modeļa, tiek nodota ekspluatācijā pakāpeniski, katrreiz papildinot to ar jaunu funkcionalitāti.

5.5. Ātrā lietotņu izstrāde (Rapid Application Development –RAD)

RAD modelis nozīmē programmatūras dažādu funkcionalitāšu un sastāvdaļu paralēlu, vienlaicīgu izstrādi, pēc tam tās savstarpēji saintegrējot. Komponenšu izstrāde parasti tiek organizēta kā paralēli mini-projekti, kuru laikā izveido prototipus, kas ļauj lietotājiem novērtēt topošo programmatūru un izteikt savas vēlmes.

RAD modelis ļauj nepieciešamības gadījumā ātri modificēt komponentes, bet jebkurā gadījumā ir nepieciešams fiksēt prasības un specifikācijas uz kādu laiku, iespējams, nelielu), kamēr tās tiek noprogrammētas, notestētas un atdotas ekspluatācijā.

Modeļa viena no vissvarīgākajām priekšrocībām ir tā, ka lietotāji jau agri un pakāpeniski var novērtēt topošo programmatūru kopumā un katru tās komponenti atsevišķi.

RAD modelis ir bāze, uz kura ir attīstījušās spējās izstrādes metodes – piemēram, ekstrēmā programmēšana (eXtrem Programmming - XP) un Scrum.

5.6. Spējais modelis

Spējā modeļa pamats ir:

1) augsti kvalificētas izstrādes komandas, kurās ietilpst izstrādātāji, testētāji un lietotāji, kurām ir dotas pilnvaras pieņemt lēmumus par projektēšanas un implementācijas jautājumiem;
2) daudz un ātras iterācijas, ar lietotāju atgriezenisko saiti.

Spējā modeļa viens no piemēriem ir XP izstrādes metodoloģija, kas piedāvā:

1) biznesa vai lietotāju scenāriju izveidi, tādējādi definējot izstrādājamās programmatūras funkcionalitāti;
2) nemitīgu un bagātīgu informāciju no lietotājiem vai klientiem par izstrādājamo programmatūru – viņi palīdz izstrādes komandai definēt un izskaidrot lietošanas scenārijus, kā arī veic akcepttestēšanu;

3) pāru programmēšana – programmē un testē divi cilvēki, t.i., katrai funkcijai viens veido programmu, bet otrs – testpiemērus;
4) testpiemēru izstrāde notiek pirms programmatūras pirmkods tiek uzprogrammēts, panākot, ka tiek izstrādāts tikai nepieciešamais pirmkods (netiek noprogrammēts nekas lieks);
5) ļoti bieža izveidoto komponenšu integrācija, pat vairākas reizes dienā;
6) katrai problēmai pēc iespējas realizē vienkāršāko un parocīgāko risinājumu.

XP izstrādes modeļa realizācija nozīmē ļoti biežas un daudz iterācijas, tādēļ nepieciešama spēcīga konfigurāciju un versiju pārvaldība, testēšanas automatizācijas rīki, lai varētu pārtestēt pēc katras iterācijas un integrācijas.

XP modeļa trūkumi:

1) testēšanas rīks tiek lietots, lai fiksētu lietotāju ieteikto lietošanas scenāriju, testēšana tiek lietota kā neatkarīga pieeja, lai pārbaudītu (verificētu un validētu) programmatūru;
2) nav iespējams novērtēt kopējās izmaksas, laiku un ieguldāmā darba apjomu pirms projekta sākuma, kā arī pirms katras iterācijas vai lietošanas scenārija realizācijas; nevar novērtēt, cik daudz koda nāksies pārrakstīt, jo uzzinot jaunas prasības, nāksies konstatēt, ka jau uzprogrammētais ir jāmodificē vai jāmaina pilnībā.

XP modeļa priekšrocības:

1) bieža komunikācija starp lietotājiem, testētājiem un izstrādātājiem;
2) izveidotās programmatūras ātra nodošana ekspluatācijā, sliktākajā gadījumā ar ierobežotu funkcionalitāti.

Dažkārt spējās izstrādes principi tiek integrēti secīgās izstrādes modeļos, kur, piemēram, pēc sākotnējās projektēšanas fāzes tiek izveidots lietošanas scenāriju komplekts un iedots izstrādātājiem, bet pēc tam izmantots akcepttestēšanas laikā. Spējais modelis šādā gadījumā tiek izmantots tikai atsevišķu izstrādājamās programmatūras komponenšu vai daļu izstrādē, bet ne kā visas sistēmas kopējais izstrādes modelis.

5.7. Kā pazīt „savu" modeli?

Izstrādes modelis ir atkarīgs no izstrādes konteksta. Dažādiem izstrādes modeļiem var būt dažāds dokumentācijas formalizācijas līmenis, tātad arī dažāds pieejamo datu apjoms testpiemēru izstrādei. Tāpat arī modeļa izvēle ir atkarīga no organizācijas, kurā notiek izstrāde:

1) vidē ar augstu formalizācijas/birokrātisma pakāpi piemērotāks ir secīgais izstrādes modelis (var būt ar inkrementālā modeļa izmantojumu). Šādā situācijā tiek saskaņotas

prasības (SLA – Service Level Agreements) starp ieinteresētajām pusēm (stakeholders) neatkarīgi no tā, vai tās ir no vienas vai dažādām organizācijām un kā tās ģeogrāfiski novietotas – strādā blakus vai atrodas vairāku tūkstošu kilometru attālumā;

2) neformālā vidē parasti lieto inkrementālo un iteratīvo modeļus, kad izstrādātāji, testētāji un lietotāji atrodas ģeogrāfiski vienā vietā;
3) vidē ar stingriem noteikumiem vai valsts iestādēs, piemēram, medicīnas, kosmosa, militārās, ar kodolenerģiju saistītās iestādēs, raksturīga secīgie izstrādes modeļi, jo tie atļauj sekot līdz termiņiem, veikt dažādas pārbaudes, piemēram, projektējuma pārbaudes vai kritiskā projektējuma pārbaudes.

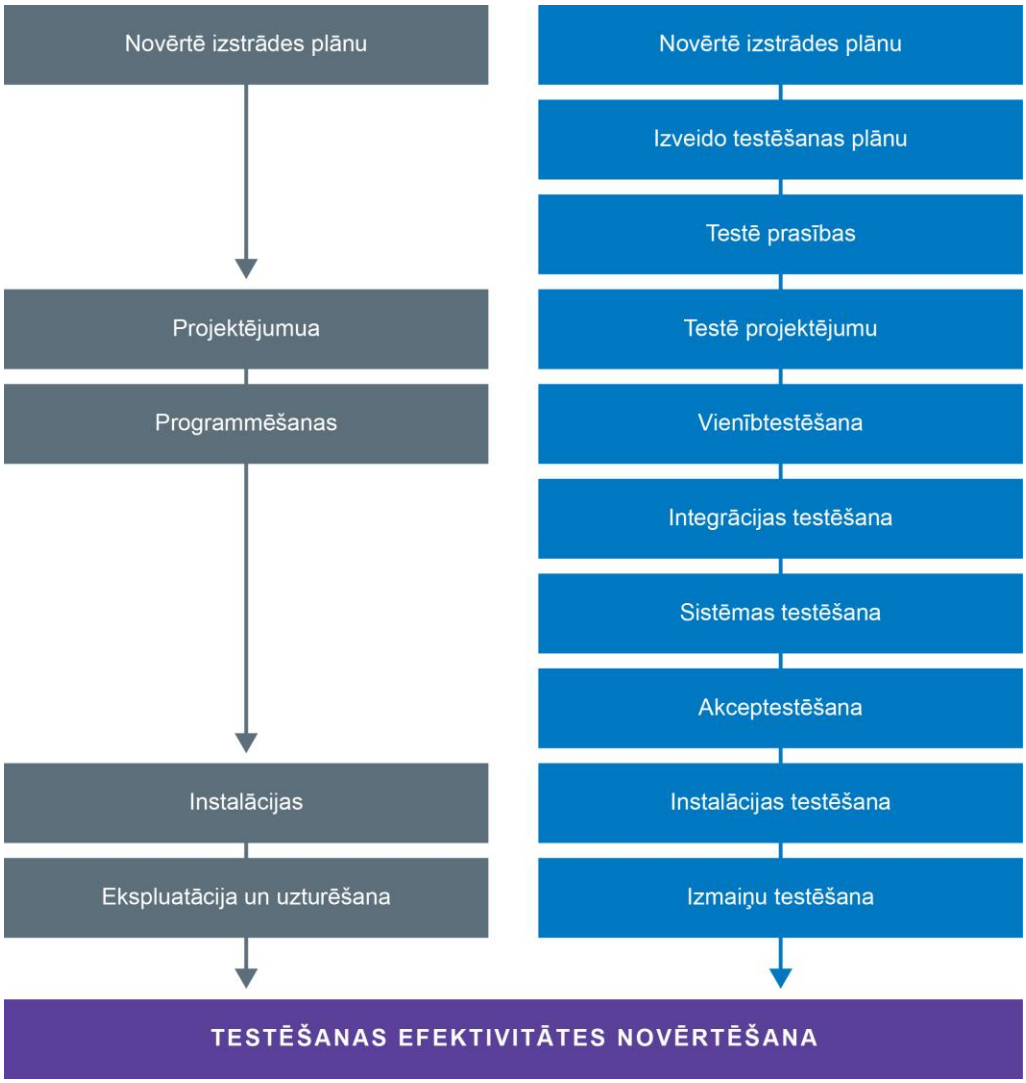
Spējie modeļi prasa lietotāju, klientu un citu ieinteresēto pušu iesaistīšanos, kas ne vienmēr ir iespējams, tādēļ šo metožu pielietošana ir ierobežota.

Izstrādes modeļu salīdzinājums

Svarīgs aspekts	Ūdenskrituma	V-modelis	Inkrementālais (pakāpeniskais)	Spirāliskais (cikliskais)	Spējais
Vadītāju cieša iesaistīšanās	✔	✔	✔	✔	
Viegla atskaitīšanās	✔	✔	✔	✔	
Visas prasības nepieciešamas pirms darbu uzsākšanas	✔	✔	✘		
Viegla resursu piešķiršana	✔	✔	✔	✔	
Ātri pieejama programmatūra, ko jau var darbināt			✔	✔	✔
Risku un izmaksu kontrole ar prototipēšanu			✔	✔	✔
Vadībai problēmas, jo trūkst skaidru mērķu			✔	✔	✔
Prasa skaidru saskarni starp komponentēm			✔	✔	✔
Ne pārāk savietojams ar formālām kontrolēm			✔	✔	✔
Tendence pārvietot problēmas uz nākošo piegādi			✔	✔	✔

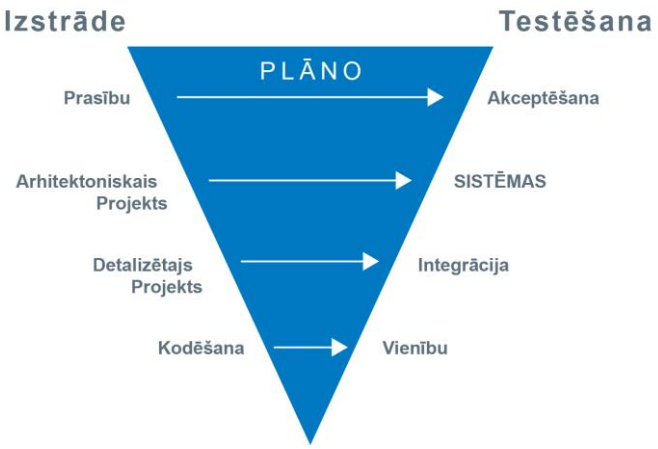
* inkrementālais modelis var sākt gan ar pilnībā definētām prasībām, gan arī ar mazāk definētiem mērķiem

6. TESTĒŠANAS LĪMEŅI



Parasti izšķir sekojošus testēšanas līmeņus – vienību testēšana, integrācijas testēšana, sistēmtestēšana un akcepttestēšana. Taču testēšanas plānošanu un izpildi veic saskaņā ar V-modeli (attēlā labajā pusē), darbus veicot paralēli programmatūras izstrādes dzīves cikla aktivitātēm (attēlā kreisajā pusē).

6.1. Testēšanas līmeņi un plānošana



Attēlā ir parādīts, kādus testēšanas līmeņus plāno kādos programmatūras izstrādes dzīves cikla posmos. Svarīgākais – jo „lielāka" līmeņa testēšana, jo agrāk to jau var sākt plānot [Per02], piemēram, akcepttestēšanā veicamos testus var plānot jau paralēli programmatūras prasību izstrādei, kas nozīmē, ka testētāji šajā brīdī var dot savu ieguldījumu preventīvā problēmu novēršanā programmatūrā, analizējot prasības un vērtējot, piemēram, to atbilstību lietotāju vēlmēm, gan arī to savstarpējo nepretrunību, precizitāti un viennozīmīgumu.

6.2. Pārskats par testēšanas līmeņiem

Parametrs	Līmenis			
	Vienību	**Integrācijas**	**Sistēmas**	**Akcepttestēšana**
Kas veic?	Izstrādātāji	Izstrādātāji un testētāji	Testētāji	Testētāji & lietotāji
Datortehnika, operāciju sistēma	Program-mētāja vide	Programmētāja vide	Sistēmtestu dators vai vairāki	Ražošanas spogulis
Kopdzīves program-matūra	Nekāda	Nekāda	Nekāda / reālā	Reālā
Saskarnes	Nekādi	Iekšējie	Simulēti un reāli	Simulēti un reāli
Testdatu avoti	Ar rokām izveidoti	Ar rokām izveidoti	Reālie un ar rokām izveidoti	Reālie
Testdatu apjoms	Mazs	Mazs	Liels	Liels
Stratēģija	Vienību	Vienību / konstrukciju grupas	Sistēma kopumā	Simulēta ekspluatācija

6.3. Vienībtestēšana

Vienībtestēšana nozīmē programmatūras nelielas komponentes testēšanu. Vienībtestēšanu parasti veic programmētāji jeb izstrādātāji. Vienībtestēšanā atrastās programmatūras kļūdas parasti nereģistrē problēmu pārvaldības sistēmās. Parasti izstrādātāji tās operatīvi novērš, turpinot komponentes izstrādi.

6.4. Integrācijas testēšana

Integrācijas testēšana nozīmē pārbaudīt vienas vai vairāku izveidotu programmatūras komponenšu sasvstarpējās sadarbības pārbaudi. Integrācijas testēšanu veic vai nu izstrādātāji (nelielām komponentēm vai programmatūras daļām) vai testētāji – lielām programmatūras daļām, piemēram, dažādu apakšsistēmu integrācijas testēšanu.

Integrācijas testēšanu var veikt dažādos veidos – no lejas uz augšu, no augšas uz leju un pēc sviestmaizes principa – jaukti, t.i., daļu programmatūras testēt no augša uz leju, daļu no lejas uz augšu, vadoties no iespējām un nepieciešamības katrā konkrētajā situācijā.

6.5. Sistēmtestēšana

Sistēmtestēšana nozīmē, ka tiek veikti testi, kās pārbauda izstrādājamo programmatūru kopumā, pēc iespējas testos iesaistot visas izstrādātās programmatūras komponentes, kā arī pārbaudot sadarbību ar citu ārējo programmatūru, piemēram, pieslēgumus ar valsts nozīmes sistēmām, no kurām saņem vai kam nosūta informāciju.

Sistēmtestēšanu parasti veic testētāji, nevis programmētāji.

6.6. Akcepttestēšana

Akcepttestēšana ir turpmāko sistēmas lietotāju veikta testēšana apstākļos, kas maksimāli tuvināti atbilst produkcijas vides apstākļiem. Tā demonstrē, kā izstrādātā sistēma atbilst izvirzītajām funkcionalitātes un kvalitātes prasībām. Sekmīga akcepttesta iziešana nozīmē, ka pasūtītājs atzīst, ka pamatā piekrīt izstrādātā produkta atbilstībai viņa izvirzītajām prasībām.

Ideālā situācijā akcepttestēšanas plāns un testpiemēri tiek izveidoti reizē ar prasībām.

Parasti akcepttestēšana nav visaptveroša, taču tā ir ļoti vērtīga, lai parādītu pasūtītājam programmatūras atbilstību prasībām un ieinteresētu lietotājus. Tādēļ bieži kā akcepttestēšanas metrika tiek lietots prasību pārklājums.

Svarīgi ir akcepttestēšanā programmatūras konfigurāciju pārvaldību veikt tikpat stingri kā ražošanā.

Akcepttestēšanas paveidi:

- funkcionālie akcepttesti - sistēmas turpmākie lietotāji pārbauda sistēmas funkcionalitāti;
- produkcijas akcepttesti - sistēma tiek darbināta produkcijas vidē testa režīmā ilgāku laiku;
- lietojamības testi.

Lietotāju pienākumi akcepttestēšanā:

- prasību definēšana;
- biznesa risku noteikšana;
- akcepttestēšanas plāna izveide, pārbaude, uzturēšana;
- reālistisku scenāriju testu veidošana;
- apgādāšana ar reālistiskiem testdatiem;
- testu izpildīšana;
- dokumentācijas novērtēšana;
- testu izejas datu novērtēšana;
- noteikt akcepta kritērijus.

6.7. Regresā testēšana

Regresā testēšana pārtestē agrāk testētus programmatūras segmentus vai visu sistēmu, lai pārliecinātos, ka tie vēl funkcionē atbilstoši specifikācijai pēc izmaiņu ieviešanas citā lietotnes daļā, piemēram, pieliekot klāt jaunu funkcionalitāti vai labojot problēmas jau esošajā.

Mērķi:

- nosaka, vai sistēmas dokumentācija arvien ir aktuāla;
- nosaka, ka sistēmas testu dati un nosacījumi arvien ir aktuāli;
- nosaka, ka agrāk testētās sistēmas daļas funkcionē pareizi pēc izmaiņu ieviešanas.

Ja regresās testēšanas process nav automatizēts, tad:

- tas var paņemt daudz laika;
- tas var būt ļoti nogurdinošs (kas palielina testētāju pašu kļūdu iespējamību);
- ir rūpīgi jāpārdomā ieguldījumu/ieguvumu efekts.

Regresās testēšanas piemēri:

- atkārto iepriekš veiktos testus, lai redzētu, vai nemainītās sistēmas daļas darbojas pareizi;
- pārskata agrāk sagatavotās manuālās procedūras, ko izpilda sistēmas darbības laikā, vai tā ir korektas pēc izmaiņām sistēmā.

Regreso testēšanu pielieto:

- kad ir liels risks, ka jaunās izmaiņas var ietekmēt sistēmas nemainītās daļas;
- izstrādes laikā regreso testēšanu var veikt pēc zināma sistēmā iekļauto izmaiņu skaita;
- uzturēšanas laikā lēmumu pieņem, vadoties no sistēmas svarīguma un zaudējumiem, kas varētu rasties sistēmas rūpīgas netestēšanas rezultātā;
- ja regresās testēšanas process ir automatizēts, tad to veic pirms katras jaunas sistēmas versijas vai tās papildinājumu/labojumu likšanas ražošanā.

7. NEFUNKCIONĀLĀ TESTĒŠANA

Ja funkcionālā testēšana koncentrējas uz programmatūras darbību jeb funkcijām, ko tā veic, tad nefunkcionālā testēšana fokusējas uz to, kā šīs darbības tiek veiktas.

Līdzīgi kā funkcionālā testēšana, arī nefunkcionālā testēšana var tikt veikta visos testēšanas līmeņos – gan vienību, gan integrācijas, kā arī sistēmtestēšanas un akcepttestēšanas līmenī.

Nefunkcionālās testēšanas veidus pēc mērķa var iedalīt vairākās grupās, piemēram:

- lietojamības testēšana – pārbauda, vai programmatūra ir lietošanā pietiekami ērta, saprotama, viegli apgūstama un atraktīva;
- veiktspējas testēšana (stresa, slodzes) – pārbauda, vai programmatūra spēj korekti strādāt ar normāli sagaidāmo datu apjomu ar paredzēto datortehniku un citu infrastruktūru, gan arī kāda ir maksimālā noslodze (datu apjomiem, transakciju biežumam un apjomam, utt.), pie kā programmatūra strādā korekti;
- uzturamības (maintenance) testēšana – pārbauda, vai programmatūra ir pietiekami ērti un lēti pavadāma, t.i., cik viegli un lēti tajā būs iekļaut jaunu funkcionalitāti, labot esošo, kā arī veikt nepieciešamo testēšanu;
- pārvietojamība – pārbauda programmatūras spēju tikt pārvietotai no vienas vides citā.

7.1. Veiktspējas testēšana

Veiktspējas testēšanas (performance testing) laikā nosaka, vai testējamā programmatūra spēj pietiekami ātri reaģēt situācijā, kad tā tiek lietota pie normālā, paredzētā noslogojuma. Bieži programmatūras prasībās ir norādītas veiktspējas prasības, piemēram, ka programmai uz kādu pieprasījumu atbilde ir jādod 1-2 sekunžu laikā. Veiktspējas testēšana pārbauda šo prasību izpildi.

7.2. Slodzes testēšana

Slodzes testēšanā (load testing) pārbauda sistēmas spēju strādāt pie lielām noslodzēm, kā paredzēts specifikācijā, nosakot sistēmas fizisko un loģisko iespēju robežas. Slodzes testēšanas jēga ir saprast, cik lielu slodzi programmatūra spēj izturēt, un redzēt, kā tā uzvedas pie lielām slodzēm – piemēram, vai tās veiktspēja būtiski netiek traucēta, vai nerodas kādi drošības apdraudējumi negaidītu veiktspējas efektu dēļ, u.c..

Piemēram, slodzes testēšanas mērķi var būt - noskaidrot, vai:

- lietotne spēj apstrādāt konkrētā laika periodā paredzēto transakciju apjomu;
- lietotne spēj apstrādāt vairāk kā konkrētā laika periodā paredzēto transakciju apjomu;
- lietotne ir strukturāli spējīga apstrādāt lielus datu apjomus;

- sistēmai ir, ieskaitot komunikāciju līnijas, pietiekami resursi, lai nodrošinātu pietiekamus apgrozījumlaikus;
- ar sistēmu cilvēki var veikt savus darbus un uzturēt vēlamos apgrozījumlaikus.

Slodzes testēšanu veic, pēc iespējas precīzi simulējot ražošanas vidi, nodrošinot, piemēram, ka:

- ar sistēmu strādājošie cilvēki (gan lietotāji, gan apkalpojošais personāls) lieto standarta dokumentāciju;
- transakcijas ievada cilvēki, kas vēlāk ar sistēmu strādās ražošanā;
- tiešsaistes sistēmas testējot cilvēki ievada transakcijas normālā un lielākā apjomā, ilgstošā laika periodā;
- pakešu sistēmas testē ar lielām ievada paketēm, turklāt jāievada vairāk nekā viena transakciju pakete.

7.3. Datu apjomu testēšana

Datu apjomu (vlume testing) testēšanā pārbauda sistēmas uzvedību situācijās, kad notiek darbs ar lieliem datu apjomiem, piemēram, ar lieliem datu failiem. Datu apjomu testēšanas laikā sistēma, piemēram, apzināti tiek pakļauta testu virknei, kuru mērķis ir radīt lielu testējamās sistēmas datu apjomu, kuri tiek glabāti, ievadīti, laboti, izvadīti sistēmā.

Arī datu apjomu testēšanu veic, pēc iespējas precīzi simulējot ražošanas vidi.

7.4. Stresa testēšana

Nosaka, vai sistēma var funkcionēt ar lieliem apjomiem – maksimālajiem apjomiem, kas paredzēti specifikācijā un arī lielākiem apjomiem, lai noskaidrotu, vai sistēma spēj strādāt arī smagākās situācijās.

„Stresojamo" jomu piemēri:

- ievada transakcijas – tiek ievadīts liels daudzums datu gan paralēli, gan secīgi;
- iekšējās tabulas – datu bāzes tabulas tiek aizpildītas ar lielu daudzumu datu;
- disku vieta – diski tiek piepildīti ar datiem gandrīz pilni;
- izeja – tiek prasīts liels daudzums rezultātu paralēli vai secīgi;
- komunikācijas – tiek noslogotas sistēmas saskarnes ar citām sistēmām, kā arī iekšējās saskarnes starp pašas programmatūras dažādām daļām, piemēram, ja tā ir tīmekļa lietotne, noslogo tieši liela apjoma datu pārsūtīšanu no lietotāja uz datu bāzi un otrādi katram lietotājam un vēl daudziem lietotājiem paralēli;
- datoru jauda – tiek pārbaudīts, ar kādu operatīvās atmiņas, disku atmiņas, procesoru jaudu, videokartes atmiņu, u.c. mazāko un, ja nepieciešams, arī lielāko, apjomu programmatūra spēj strādāt korekti;

- saskarsme ar cilvēkiem – vai programmatūras spēj nodrošināt lielu skaitu lietotāju vienlaicīgi, vai tā spēj tikt galā, kad lietotāji tīši vai netīši to lieto nekorekti vai neparedzētā apjomā, secībā, nolūkā.

7.5. Lietojamības testēšana

Lietojamība ir programmatūras spēja būt saprotamai, apgūstamai, lietojamai un atraktīvai lietotājam, kad tā tiek lietota. Tādēļ lietojamības testēšanas laikā novērtē, cik viegli un ērti ir gala lietotājiem sistēmu apgūt un lietot.

Vispārīgi runājot, novērtē gan lietotāja dokumentāciju, gan programmatūras funkcionalitāti, ko lietotājs lietos, gan arī programmatūras spēju atkopties, kad notikušas kādi traucējumi vai atteices.

Īpaša uzmanība lietojamības testēšanā tiek pievērsta programmatūras saskarnei – dažādiem logiem, dialogiem, izvēlnēm, iespējām strādāt ar taustiņkombinācijām, „karstajiem" taustiņiem, peli.

Problēmas:

- lietotājiem ir dažādas vajadzības, prasmes, gaumes, prasības, sagatavotība un izglītība;
- lietotāji parasti nezina vai nespēj precīzi formulēt, ko vēlas;
- izstrādātājiem parasti nav skaidrs, ko precīzi izsaka prasības, jo prasībās lietojamības jautājumi bieži netiek specificēti.

Lietojamības testu piemēri:

- objektu izvietojums un izlīdzinājums ekrānā;
- lietotāja saskarnes tests (atvērt visas izvēlnes, izmēģināt visus objektus);
- pamata funkcionalitātes pārbaude (File+Open+Save...);
- peles labās pogas klikšķu jūtība;
- loga izmēru maiņa, minimizēt, maksimizēt;
- ritjoslu pārbaude;
- navigācija ar klaviatūru un peli, iezīmēšana, vilkšana;
- drukāšana, novietojot lapu guļus vai stāvus;
- pārbauda F1, palīdzības izvēlni;
- lauku apstaigāšanas secība ar Tab taustiņu un navigācija visos dialogos un izvēlnēs.

7.6. Drošības testēšana

Drošības testēšanas (security testing) mērķis ir noskaidrot, vai programmatūra ir aizsargāta pret ārējiem uzbrukumiem. Drošības testēšanā pārbauda sistēmas konfidencialitāti, integritāti,

kā arī sistēmas un tajā saglabāto datu pieejamību. Drošības testēšanā parasti veic ielaušanās testu un konfigurācijas izvērtēšana – cik lietotnes fiziskā, loģiskā arhitektūra un infrastruktūrair atbilstoša drošības prasībām.

Drošība ir aizsardzības sistēma, lai:

- sargātu konfidenciālu informāciju;
- pārliecinātu trešās puses, ka viņu dati ir aizsargāti.

Drošības testēšanu lieto, kad lietotnes aizsargātie dati un/vai aktīvi ir organizācijai nozīmīgi.

Drošības testēšana pārbauda arī aizsargprocedūru un pretpasākumu adekvātumu. Šis ir ļoti specializēts testēšanas process, ko īpaši nespecializējušies testētāji saviem spēkiem parasti var veikt tikai viduvējā līmenī. Drošības testēšanā plaši tiek pielietota „negatīvā" testēšana, kad pārbauda, kas notiek, ja sistēmu lieto „nepareizi" - ne tā, kā paredzēts specifikācijā, neparedzēti.

Drošības testēšanu jāveic gan pirms sistēmas atdošanas ražošanā, gan pēc tās.

Piemēri:

- nosaka, vai sargājamie resursi ir identificēti un pieeja tiem ir definēta;
- novērtē, vai izstrādātās drošības procedūras ir pareizi realizētas un funkcionē atbilstoši specifikācijām;
- tiešsaistes sistēmas var mēģināt "uzlauzt", lai identificētu un novērstu neautorizētu avotu pieeju.

8. TESTĒŠANAS DOKUMENTĀCIJA

Visu testēšanas aktivitāšu, piemēram, testēšanas plānošanas, testu datu sagatavošanas, testpiemēru projektēšanas, testpiemēru izpildes, verifikācijas un validācijas laikā tiek veidota un uzturēta testēšanas dokumentācija.

Detalizētai testēšanas dokumentācijai ir gan priekšrocības, gan trūkumi:

- precīzi un detalizēti testpiemēri ir mazāk elastīgi, ja programmatūra, kuru tie testē, tiek mainīta, bet – tie ļauj testēšanā izmantot mazāk pieredzējušus testētājus;
- mazāk precīzi un vispārīgāk aprakstīti (vai pat maz dokumentēti) testpiemēri ir grūtāk atkārtojami, ja to izpildes laikā tiek konstatēta kāda problēma, it īpaši, ja nav detalizēts, kādu informāciju testētājam jāievada programmatūrā, tāpat šie testpiemēri ir grūtāk izpildāmi lietotājiem, kam nav labu zināšanu par testējamās programmatūras biznesa jomu;
- precīzos un detalizētos testpiemēros ir precīzāk uzdoti sagaidāmie rezultāti un tāpēc ir skaidrāki kritēriji, lai novērtētu, vai testpiemērs ir izpildījies veiksmīgi vai nē, kā arī ir mazāk subjektīvas interpretācijas iespējas testētājam. Taču šādu testpiemēru uzturēšana ir grūtāka un dārgāka;
- mazāk detalizētus testpiemērus ir vieglāk izprojektēt un aprakstīt, bet grūtāk ir novērtēt kopējo testēšanas pārklājumu.

Testēšanas dokumentāciju var veidot dažādi, gan saskaņā ar kādu no standartiem, piemēram, saskaņā ar IEEE 829 standarta 2008.gada versiju, gan arī piemērojoties programmatūras izstrādes metodoloģijai.

8.1. Testēšanas stratēģija

Testēšanas stratēģija ir dokuments, kurā tiek aprakstītas testēšanā ieinteresētās puses, viņu vajadzības, kā arī testēšanai pieejamie resursi, kā rezultātā konkrētajai testēšanai tiek doti konkrēti mērķi, uzdevumi un uz kā pamata tiek izstrādāta testēšanas pieeja.

8.1.1. Izmantojamie resursi

Resursi ir – laiks, budžets un cilvēki (skaits un kvalifikācija gan testēšanā, gan programmatūras izstrādē, gan testējamās programmatūras biznesa jomā), kas tiek atvēlēti testēšanai. Ierasta problēma – testēšanai ir atvēlēts pārāk maz laika, jo programmatūra ir atdota testēšanai pārāk vēlu, budžets nav atvēlēts pietiekams, bet testētāju komandas kvalifikācija varētu būt augstāka.

8.1.2. Ieinteresētās puses

Testēšanas rezultātos ieinteresēto pušu var būt diezgan daudz, piemēram,:

- lietotāji;
- pasūtītāji;
- izstrādes komandas vadītāji;
- atbildīgie par kvalitātes nodrošināšanu;
- augsta līmeņa vadītāji.

8.1.3. Testēšanas mērķi

Katra no ieinteresētajām pusēm dod testēšanai savus mērķus, piemēram:

- lietotājiem ir svarīgi, lai programmatūra strādā uzticami, ir ar labu lietojamību;
- pasūtītājiem ir svarīgi, lai programmatūrā nebūtu atlikušas smagas kļūdas, kas, ja tiks atklātas vēlāk darba gaitā, var izraisīt smagas sekas (finansiāli, morāli, utt.) un/vai kuru izlabošana vēlāk var dārgi maksāt;
- izstrādātājiem ir svarīgi gūt apliecinājumu, ka viņu izstrādātā programmatūra strādā korekti, bet principā viņi nav ieinteresēti, ka testētāji atklāj smagas, grūti labojamas kļūdas, jo to labošana prasīs laiku un resursus, par ko pasūtītājs papildus nemaksās;
- kvalitātes nodrošināšanas speciālistiem ir svarīgi, lai testēšanas procesi norit un dokumentācija tiek sagatavota atbilstoši kvalitātes prasībām un procedūrām;
- augsta līmeņa vadītājiem ir svarīgi, lai biznesa procesi netiek traucēti un necieš uzņēmuma prestižs, piemēram, programmatūras nekorektas darbības vai sliktas lietojamības dēļ.

Ieinteresēto pušu mērķi var būt pretrunīgi, piemēram, pēc dažādiem principiem ir jāveido testpiemēru komplekti, ja mērķis ir atrast smagas kļūdas vai pārliecināties, ka programmatūra atbilst specifikācijai.

8.1.4. Testēšanas uzdevumi

Testēšanas uzdevumi izriet no testēšanas mērķiem, par kādiem ieinteresētās puses ir vienojušās, piemēram, konkretizējot, kādus programmatūras aspektus testēs, kādus apgabalus kādā intensitātē testēs, ko netestēs vai daļēji testēs. Testēšanas uzdevumi ir jāsaskaņo ar pieejamajiem resursiem.

8.1.5. Testēšanas pieeja

Testēšanas pieeja ir konceptuālā līmenī izstrādāts testēšanas plāns, ietverot jau arī tehniskos testēšanas līdzekļus, piemēram, metodes, rīkus, kurus paredzēts izmantot, lai realizētu testēšanas uzdevumus un sasniegtu testēšanas mērķus.

Aprakstot testēšanas pieeju, tiek konkretizēti testēšanas ierobežojumi un vides risinājumi – uz kādas tehnikas, kādā infrastruktūras un līdzprogrammatūras vidē notiks testēšana.

8.2. Testēšanas plāns

Ja testēšanas stratēģija ir vispārējs konkrētās programmatūras testēšanau raksturojošs dokuments, tad testēšanas plānā tiek detalizēti aprakstīta konkrētā testēšanas kampaņa. Piemēram, var būt atsevišķi testēšanas plāni visas sistēmas akceptestēšanai, atsevišķu apakšsistēmu integrācijas testēšanai, veiktspējas testēšanai. Testēšanas plāna mērķis ir mērķtiecīgi aprakstīt testēšanas komandas darbu.

Testēšanās plānā tiek iekļauta sekojoša informācija:

- unikāls plāna identifikators un versija;
- ievads, kurā īsi raksturo dokumentu un to izveides kontekstu;
- apraksta testējamos objektus (ko testēs – kādu programmatūru, datortehniku, dokumentāciju, utt.) un testēšanas mērķi;
- testēšanas uzdevumi - ko pārbaudīs, piemēram, vai testēs funkcionalitātes atbilstību specifikācijai, vai varbūt lietojamību vai drošību;
- raksturlielumi, kas netiks testēti – ir svarīgi aprakstīt testēšanas robežas;
- plānotā testēšanas pieeja – izmantotie pamatprincipi, metodes, tehnikas, testēšanas aktivitāšu sadalījums pa testēšanas līmeņiem vai uzdevumiem;
- kritēriji, kā noteiks, vai programmatūra ir testus izgājusi vai nē; tie bieži ir atkarīgi no programmatūrā atlikušo problēmu skaita un smaguma, par ko savukārt spriež pēc atrasto problēmu skaita un smaguma;
- testēšanas pārtraukšanas un atsākšanas kritēriji – apraksta iemeslus, kad testēšana tiks pārtraukta, jo resursu izmantošana kļūst neefektīva (piemēram, neturpina testēšanu, ja programmatūra vienkāršos iepriekšfiksētos gadījumos vai arī programmatūra nav korelēti uzinstalējusies);
- testēšanas nodevumi, ieskaitot starpnodevumus un gala nodevumus, piemēram, testu dati, statistika, skripti, scenāriji, pārskati;
- testēšanas darbības, kas jāveic, lai sagatavotu nodevumus;
- prasības videi, kurā notiks testēšanas darbības (datortehnika, līdzprogrammatūra, testēšanas rīki);
- prasības testēšanas komandai attiecībā uz uzdevumu izpildes un nodevumu kvalitāti;
- testēšanā iesaistīto cilvēku lomas, nepieciešamais zināšanu un prasmju līmenis gan no testēšanas tehniskā viedokļa (rīki, metodes), gan no testējamās programmatūras biznesa jomas viedokļa;
- veicamo darbību laika plāns – uzdevumu secība, atkarības un izpildes ilgums;
- riski un aktivitātes risku samazināšanai, piemēram, testpiemēru izpildes apjoma katram uzdevumam atkarība no uzdevuma relatīvā svarīguma (piemēram, kritiskuma, iespējamo atrasto problēmu smaguma);
- testa plāna akceptējums no ieinteresētajām pusēm, kas var būt gan testētāju komanda (novērtējot uzdevumus, kas jāveic), gan izstrādes grupa (lai pārliecinātos, ka viņi spēs

piegādāt vajadzīgās kvalitātes programmatūru sagaidāmajos termiņos), gan klienti un lietotāji (kuri akceptē vai neakceptē norādītos pārklājuma kritērijus), gan augstāka līmeņa vadība (kas novērtē uzrādīto mērķu nepieciešamību).

8.3. Testa scenārijs, specifikācija

Testu var specificēt dažādi. Viens no variantiem ir, norādot:

- specifikācijas identifikatoru;
- programmatūras funkcijas, ko testēs;
- sīkāk apraksta pieeju, kā testēs;
- specificē, kas jādara, uzdodot scenāriju jeb darbību virkni, kas jāveic;
- apraksta kritērijus, kā novērtēt, vai testpiemērs beidzies veiksmīgi, vai nē.

8.4. Testpiemērs

Testpiemēru veido saskaņā ar tā specifikāciju. Testpiemēra apraksts var saturēt sekojošas sadaļas:

- testpiemēra identifikators;
- ko testē;
- ievadvērtību apraksts;
- rezultātu apraksts;
- vides, kurā veicams testpiemērs, apraksts;
- īpašo prasību apraksts, piemēram, kādiem datiem jābūt iepriekš sagatavotiem vai kādā stāvoklī jābūt programmatūrai, pirms testpiemēru izpilda; tāpat arī – kādas darbības jāveic, lai sakārtotu testēšanas vidi pēc testpiemēra izpildes;
- saistība ar citiem testpiemēriem.

Ievadvērtības un rezultātus var aprakstīt ļoti konkrēti, piemēram, ka dialoga laukā *Uzvārds* jāievada vērtība „Ozoliņš”, bet apraksts var būt veidots arī vispārīgāk, piemēram, dialoga laukā *Uzvārds* jāievada burtu kombinācija, ne garāka par 30 simboliem.

Ja ievadvērtības ir aprakstītas konkrēti, ir vieglāk konkrēti aprakstīt arī rezultātus, piemēram, ka cilvēkam ar uzvārdu Ozoliņš alga ir 350EUR. Vispārīgi aprakstītiem testpiemēriem arī rezultāti ir vispārīgi aprakstīti, piemēram, algai ir jābūt robežās no 100EUR līdz 1000EUR. Vispārīgi aprakstīto testpiemēru izpildē testētāji var izvēlēties ievadvērtības saskaņā ar specifikāciju un pēc savas pieredzes, lai testēšana būtu iespējami efektīvāka, taču ir grūtāk novērtēt rezultātu – testētājam tas ir jārēķina līdzi.

8.5. Testēšanas pārskati

Testēšanas pārskati tiek sniegtas ieinteresētajām pusēm saskaņā ar testēšanas mērķiem, piemēram, par atrastajām problēmām un to smagumu, par veikto testēšanas apjomu, novērtējot gan notestētās programmatūras apjomu (prasību pārklājumu, koda pārklājumu), gan veikto testu skaitu salīdzinājumā pret iecerēto vai specificēto testu skaitu, utt..

9. TESTĒŠANAS KOMANDA

Testēšanas komandas var atšķirties, piemēram, gan pēc testētāju zināšanām testēšanā un par testējamās programmatūras biznesa jomu, gan pēc pakļautības (piemēram, vai tā ir neatkarīga, vai iekļauta izstrādes komandas sastāvā), gan pēc atrašanās vietas tīri ģeogrāfiski – vai tā strādā kopā ar izstrādātāju komandu vai atrodas citur – citā ēkā, citā valstī.

9.1. Iekšējā vai ārējā

Testētāju komandu sauc par iekšējo, ja tā ir vienā organizācijā ar programmatūras izstrādātāju komandu. Testētāju komanda ir ārējā, ja tā ir no citas organizācijas.

9.2. Izstrādātāji vai neatkarīga testētāju komanda

Testētāju komanda var būt iekļauta izstrādātāju komandas sastāvā un pakļauta izstrādes grupas vadītājam, bet tā var būt arī neatkarīga. Neatkarīgā testētāju komanda varbūt gan iekšēja, gan ārēja.

9.3. IT cilvēki vai. ne-IT cilvēki

Testētāju komandā var tikt iekļauti gan cilvēki ar zināšanām IT jomā, gan arī cilvēki bez tām, bet ar zināšanām testējamās programmatūras biznesa jomā. Lai testēšana būtu efektīva, testētāju komandā ir nepieciešami cilvēki ar zināšanām gan par testēšanu, gan arī par testējamās programmatūras biznesa jomu.

9.4. Testēšanas neatkarība

Ir iespējami dažādi testēšanas neatkarības līmeņi:

- pilnīgs neatkarības trūkums – testēšanu veic paši izstrādātāji, tāpat arī sistēmanalītiķi var paši pārvērtēt izstrādātās specifikācijas;
- daļēja neatkarība – testēšanu veic nevis pats programmatūras izstrādātājs vai sistēmanalītiķis, bet gan viņa kolēģis; šī pieeja īpaši raksturīga tā saucamajā pāru programmēšanā;
- cilvēks vai cilvēku grupa, kas specializējušies testēšanā, bet kuri ir izstrādes komandas sastāvdaļa;
- specializēta testētāju komanda tajā pašā organizācijā kā izstrādes grupa, bet kas nav pakļauta izstrādes grupas vadītājam;
- specializēta testētāju komanda citā organizācijā kā izstrādes grupa.

Pieaugot testētāju neatkarības līmenim samazinās iespēja, ka testētāju viedokli par testējamo programmatūru un tās specifikāciju ietekmē izstrādātāji, tādēļ testētāji spēj paskatīties uz to no cita skatu punkta un, iespējams, ieraudzīt vairāk problēmu iespējami agrāk programmatūras dzīves ciklā.

No otras puses, ja testētāji ir pārāk atrauti no izstrādātājiem, viņi nevar tik viegli iegūt un izmantot savā darbā dažādu noderīgu informāciju gan par programmatūras tehnisko struktūru, gan par dažādiem tehnoloģiskiem risinājumiem, gan par dažādām problēmām izstrādes laikā, kas varētu norādīt, kur potenciāli programmatūrā varētu būt problēmas.

9.5. Testēšanas psiholoģiskie aspekti

Testētājiem un izstrādātājiem ir principiāli dažāds domāšanas veids, jo viņiem ir dažādi mērķi:

- izstrādātājs meklē vienu, pēc iespējas labāku, risinājumu konkrētajai problēmai un izstrādā programmu atbilstoši šis risinājumam;
- testētājs meklē visus iespējamos defektus, kas varētu programmatūrā būt, ko izstrādātājs varētu būt aizmirsis, pārpratis, kur kļūdījies.

Izstrādātājiem parasti ir divas lomas – izstrādātāja un testētāja. Izstrādātāji testē programmatūru vismaz vienībtestēšanas līmenī, bet ir iespējams, ka arī integrācijas un sistēmtestēšanas līmeņos. Taču cilvēkam ir grūti ieraudzīt savas kļūdas, tādēļ ir svarīgi, ka programmatūru testē arī citi cilvēki, ne tikai izstrādātāji.

Ja testētāji ir pakļauti izstrādes grupas vadītājam, pastāv risks, ka vadītājs neļauj identificēt kāda noteikta tipa problēmas, piemēram, lietojamības problēmas novērtē kā nesvarīgas. Rezultātā testētāji vispār neveic atbilstoša veida problēmu meklēšanu un reģistrēšanu.

10. TESTĒŠANAS RĪKI UN TO VEIDI

Testēšanā rīkus lieto galvenokārt divu iemeslu dēļ:

- lai automatizētu nogurdinošu, bieži atkārtojamu testpiemēru izpildi;
- lai izpildītu un novērotu darbus, ko nevar viegli veikt cilvēki, piemēram, veiktspējas testēšanā analizētu programmatūras reaģēšanas laiku (ko parasti mēra sekundes simtdaļās).

Testēšanas rīkus pēc to veicamajiem uzdevumiem var iedalīt sekojoši:

- testēšanas pārvaldības rīki:
  - projektu pārvaldības rīki plānošanai un aktivitāšu veikšanas novērošanai;
  - nodrošina testēšanas izsekojamību sākot ar testu specificēšanu līdz izpildei un testēšanas rezultātu novērošanai;
  - nodrošina problēmu pārvaldību no reģistrācijas līdz izlabošanai,
  - nodrošina versiju un konfigurācijas pārvaldību testējamajām komponentēm, kā arī visām komponentēm, draiveriem, dokumentiem, kas izmantoti testēšanas aktivitātēs;
  - atbalsta risku analīzi, ieskaitot riski identifikāciju, novērtēšanu un risku novērošanu;
  - nodrošina pārskatus par testēšanu;
- prasību pārvaldības rīki ļauj
  - saglabāt prasības un specifikācijas;
  - pārbaudīt prasību saskaņotību;
  - norādīt prasību un specifikāciju prioritātes;
  - nodrošināt katra testa un tā izpildes izsekojamību, tādējādi dodot iespēju novērtēt prasību, specifikāciju un testu pārklājumu;
- pārskašu nodrošināšanas rīki:
  - pārskašu procesu atbalsta rīki ļauj plānot un izsekot līdzi pārskašu procesiem;
  - ļauj pārvaldīt pārskašu dokumentus;
  - nodrošina izsekojamību starp programmatūras kodu un pārskašu dokumentiem;
- statiskās analīzes rīki:
  - ļauj identificēt defektus pirms uzsākta programmatūras dinamiskā izpilde;
  - nodrošina koda atbilstību kodēšanas standartiem;
  - izdara mērījumus koda un arhitektūras metrikām [Hut03];
  - analizē programmatūras struktūru un tās daļu savstarpējās atkarības;
- modelēšanas rīki – tos izmanto izstrādājamās programmatūras biznesa uzvedības modelēšanai un iegūto modeļu novērtēšanai;
- testu specificēšanas un testdatu izveidošanas un pārvaldības rīki:
  - testdatus ģenerē, izmantojot prasības, grafiskās lietotāju saskarnes, arhitektūras un uzvedības modeļus, kodu;

    - rada un darbojas ar datu failiem, datu bāzēm, individuālajiem datiem vai ziņojumiem, kas izmantoti testu izpildē;
    - veido liela apjoma datus testēšanai, piemēram, veidojot datu kombinācijas dažādiem programmatūras ievadparametriem atbilstoši to vērtību ekvivalences klasēm vai robežvērtībām;
    - analizē statistikas datus attiecībā pret dotiem nosacījumiem un riskiem;
    - anonimizē datus, kas ņemti no ražošanas vides, lai nodrošinātu klientu datu konfidencialitāti;
- testu izpildes rīki – palīdz ievadīt ievaddatus, izmantojot lietotāja saskarni, programmatūras API saskarni vai komandrindu; parasti šajos rīkos ir iekļautas datu salīdzināšanas iespējas, lai var novērtēt testpiemēru izpildes rezultātu atbilstību sagaidāmajiem rezultātiem;
- testēšanas vižu pārvaldības rīki – ļauj aizvietot neesošu aprīkojumu vai programmatūru ar simulatoriem (simulē aizvietojamo daļu darbību) vai emulatoriem (izdod aizvietojamo daļu darbības rezultātus, bet nesimulē to darbību); tāpat šī rīki nodrošina dažādas vides testēšanai – operāciju sistēmas, datortehniku;
- pārklājuma mērīšanas rīki;
- datu salīdzināšanas rīki – izmanto testu izpildes rezultātā iegūto datu, datu failu vai datu bāzu salīdzināšanai ar sagaidāmajiem testu rezultātiem;
- drošības testēšanas rīki;
- veiktspējas un slodzes testēšanas rīki;
- citi rīki.

Rīku izmantošanas priekšrocības:

- samazina nepieciešamību veikt manuāli atkārtotas, vienādas darbības, piemēram, veikt regresos testus;
- palielina regresās testēšanas uzticamību un atkārtojamību, kā arī spēju salīdzināt lielus datu apjomus;
- spēj simulēt un palielināt lietotājiem veicamo transakciju daudzumu, piemēram, slodzes, stresa un veiktspējas testēšanā, kā arī simulēt sarežģītas un neparastas programmatūras izpildes vides;
- ļauj viegli veikt metriku mērījumu datus, arī cilvēkam grūti nomērāmus datus, piemēram, par datoru tīkla caurlaidību vai programmatūras reakcijas laikiem;
- vienkāršo pārskatu veidošanu.

Rīku lietošanas riski:

- pārāk zemu novērtēts apjoms ieguldāmajam darbam, lai iegūtu rezultātus no rīka lietošanas;
- nenovērtējot paša rīka ietekmi uz mērāmajiem lielumiem, piemēram, rīka ietekmi uz datortīkla noslogojumu vai datora ātrdarbību;

- sagaidot pārāk daudz no rīku izmantošanas;
- nenovērtējot darba apjomu, kas nepieciešams testu skriptu un datu uzturēšanai, kad testējamā sistēma mainās.
- rīku versiju un konfigurāciju pārvaldības trūkums;
- rīku izstrādātāju vai piegādātāju bankrots vai darbības izbeigšana.

11. PROBLĒMU PĀRVALDĪBA

- Programmatūras testēšanas galvenais mērķis ir atrast problēmas programmatūrā un palīdzēt nodrošināt augsti kvalitatīvas programmatūras piegādi pasūtītājam vai lietotājam. Tāpēc ir svarīgi, lai atrastie defekti tiktu izlaboti.
- Problēmas parasti tiek reģistrētas problēmu pārvaldības rīkā, kas nodrošina to apstrādes procesa pārredzamību un iespējami ērtu veikšanu.
- Testētājam ir nepieciešams konstatēt programmatūras neatbilstošu uzvedību vai nu iegūstot testpiemēra izpildes rezultātā datus, kas neatbilst sagaidāmajiem rezultātiem vai konstatējot programmatūras negaidītu uzvedību, piemēram, negaidītu ātrdarbības pazemināšanos, kādu datu sabojāšanu.

Kad testētājs problēmu ir konstatējis, tiek izveidots problēmziņojums par to, kas tiek reģistrēts problēmu pārvaldības rīkā. Pēc tam reģistrētā problēma tiek novērtēta, piemēram, cik tā ir nopietna attiecībā pret lietotājiem, attiecībā pret darba apjomu, kas nepieciešams tās izlabošanai. Pēc tam tiek pieņemts lēmums, vai to labos, kas un kāds labos. Kad problēmu izstrādātāji uzskata par izlabotu, testētāji veic testēšanu, lai pārliecinātos, ka tā patiešām ir izlabota un ka labojums nav atstājis negatīvu ietekmi uz pārējo programmatūras funkcionalitāti. Ja testēšanas rezultāts ir pozitīvs, tad tiek uzskatīts, ka problēma ir izlabota.

12. IETEICAMĀ LITERATŪRA

[AkadTerm] Akadēmiskā terminu datubāze AkadTerm: http://termini.lza.lv/term.php

[Bei95] **Beizer, B.** Black Box Testing, New York: John Wiley, 1995

[BHS+01] **Burnstein, I., Homyen, A., Suwanassart, T., Saxena, G., Grom, R.** *A testing Maturity Model for Software Test Process Assessment and Improvement***, In:** Fundamental Concepts for the Software Quality Engineer, (Daughtrey, T., ed.), ASQ Quality Press, 2001

[Bla04] **Black, R.** *Critical Testing Processes,* Addison-Wesley, 2004

[CBC01] **Culbertson, R., Brown, C., Cobb, G.** *Rapid Testing,* Prentice Hall PTR, 2001

[ESU97] **Ericson, T., Subotic, A., Ursing. S.,** TIM - A Test Improvement Model, Software Testing, *Verification & Reliability (STVR)*, 7, 4, 1997, John Wiley & Sons, Inc., pp. 229-246

[FD07] **Farooq, A., Dumke, R.R.** Research directions in verification & validation process improvement, *SIGSOFT Software Engineering Notes*, 32, 4, 2007

[Hut03] **Hutcheson, M. L.** *Software Testing Fundamentals: Methods and Metrics*, Wiley Publishing Inc., Indianapolis, Indiana, 2003

[Jor95] **Jorgensen, P. C.** *Software Testing: A Craftman's Approach,* Boca Raton, London, New York, Washington D.C., CRC Press, 1995

[Jun09] **Jung, E**. A Test Process Improvement Model for Embedded Software Developments, ***In:*** *Proc. Of the 9th Internatinal Conference on Quality Software*, 2009, Jeju, South Korea

[KBP02] **Kaner, C., Bach, J., Pettichord, B.** *Lessons Learned in Software Testing: A Context Driven Approach* New York, Chichester, Weinheim, Brisbane, Singapore, Toronto: John Wiley & Sons, Inc., 2002

[KFN99] **Kaner, C., Falk, J., Nquyen, H. Q.** *Testing Computer Software,* 2nd ed. New York, Chichester, Weinheim, Brisbane, Singapore, Toronto: John Wiley & Sons, Inc., 1999

[KP99] **Koomen, T., Pol, M.** Test Process Improvement: A practical step-by-step guide to structured testing, Addison-Wesley, Great Britain, 1999

[Per00] **Perry, W. E.** *Effective Methods for Software Testing,* John Wiley & Sons, 2nd ed., 2000

[Per06] **Perry, W. E.** *Effective Methods for Software Testing*, Wiley Publishing, Inc, Indianapolis, Indiana, 3rd ed., 2006

[SLS07] **Spillner, A., Linz, T., Schaefer, H.** Software Testing Foundations. A Study Guide for the Certified Tester Exam. Foundation Level, ISTQB compliant, Rocky Nook Inc., 2007

[Vee02] **Veenendaal, E.** *The Testing Practitioner*, UTN Publishers, 2002

[Сав01] **Савин, Р.** Тестирование Дот Ком, или Пособие по жёсткому обращению с багами в интернет-стартапах, Дело, 2007

SQUALIO

Kr. Barona 13/15-41, Rīga, LV-1011, Latvija
tālrunis: +371 66119005 / fakss: +371 66119005
info@squal.io / www.squal.io