LATVIJAS UNIVERSITĀTE

DATORIKAS FAKULTĀTE

VINETA ARNICĀNE

SAREŽĢĪTĪBU IETEKMĒTA PROGRAMMATŪRAS TESTĒŠANA

Promocijas darbs

datorzinātņu doktora (Dr. sc. comp.) zinātniskā grāda iegūšanai

Nozare: datorzinātnes
Apakšnozare: datu apstrādes sistēmas un datortīkli

Zinātniskais vadītājs:

profesors, Dr. sc. comp.

Jānis Bičevskis

R ī g a - 2013

IEGULDĪJUMS TAVĀ NĀKOTNĒ

Darbs izstrādāts ar Eiropas Sociālā fonda atbalstu projektā «Atbalsts doktora studijām Latvijas Universitātē» (Nr.2009/0138/1DP/1.1.2.1.2./09/IPIA/VIAA/004), kā arī projekta „Datorzinātnes pielietojumi un tās saiknes ar kvantu fiziku" (līguma Nr. 2009/0216/1DP/1.1.1.2.0/09/APIA/VIAA/044) ietvaros.

Darba vadītājs:

Profesors, Dr. sc. comp. Jānis Bičevskis
Latvijas Universitāte

Recenzenti:

Profesors, Dr. habil. sc. comp. Juris Borzovs
Latvijas Universitāte

Asoc. profesore, Dr. sc. comp. Rudīte Čevere
Latvijas Lauksaimniecības universitāte

Profesors, Dr. Phil. Vladislavs V. Fomins
Vytautas Magnus University

Darba aizstāvēšana notiks Latvijas Universitātes Datorzinātņu promocijas padomes atklātā sēdē 2013.gada 23. aprīlī plkst. 16:00 LU Matemātikas un informātikas institūtā Raiņa bulvārī 29, Rīgā, 413.auditorijā.

Ar darbu un tā kopsavilkumu var iepazīties LU bibliotēkā Kalpaka bulvārī 4

Padomes priekšsēdētājs Jānis Bārzdiņš

ANOTĀCIJA

Promocijas darbs ir veltīts programmatūras testēšanas metodoloģisko, tehnoloģisko un organizatorisko aspektu izpētei ar mērķi nodrošināt uzticamas programmatūras izveidi, akcentējot sarežģītības ietekmi uz programmatūras testēšanas procesiem. Pētījuma rezultātā tiek piedāvāts reorganizēt testēšanas procesus, pielietojot tajos daudzaģentu un komplekso sistēmu darbības principus.

Darbā testēšanas tehnoloģisko aspektu jomā aplūkots vērtību apgabalu testēšanas modelis un novērtēta testēšanas sarežģītība kā izpildāmo testu skaits dažādiem ekvivalences klašu un robežvērtību testēšanas jeb domēntestēšanas kritērijiem. Identificētas tradicionālās testēšanas, kas balstīta uz vērtību apgabalu testēšanas modeļiem, galvenā problēma – tehnoloģiskā sarežģītība, kas praktiskos lietojumos padara neiespējamu šo testēšanas modeļu lietošanu.

Darbā tiek piedāvātas 2 oriģinālas idejas testēšanas organizatoriskās struktūras uzlabošanai, kas ļauj vismaz daļēji pārvarēt testēšanas tehnoloģisko sarežģītību:

1. Uztvert *testēšanas sistēmu* - cilvēku, tehnikas un materiālu organizētu sakopojumu, kas nepieciešams, lai izpildītu ieinteresēto pušu dotos testēšanas uzdevumus dotajā laikā un ar dotajiem resursiem - kā kompleksu sistēmu un, izmantojot kompleksu sistēmu darbības principus, ar adekvātu testēšanas organizatorisko struktūru un procesu pārvaldību pārvarēt testēšanas tehnoloģisko sarežģītību. Papildus šai oriģinālajai pieejai, pamatojoties uz komplekso sistēmu darbības principiem, darbā izstrādāti metodiski ieteikumi testēšanas procesu pārvaldībai.

2. Piedāvāts testēšanas teorijā oriģināls modelis - SUP modelis - testēšanas efektivitātes paaugstināšanai, kuru raksturo nepieciešamība ņemt vērā sistēmu izstrādē iesaistīto dalībnieku atšķirīgu skatījumu uz veidojamo sistēmu. Šis modelis pierāda ne-IT speciālistu iesaistīšanas testēšanā nepieciešamību. Tiek piedāvāts programmatūras testēšanas procesos pielietot daudzaģentu sistēmas organizatoriskos principus, kas, iesaistot galalietotājus sistēmas izstrādē un testēšanā, ļauj samazināt testēšanas sarežģītību un palīdz risināt testētāju kvalifikācijas problēmas.

Darbā aprakstītie principi pārbaudīti vairākos praktiskos testēšanas projektos, kas apliecina piedāvātās pieejas lietderību.

Darba rezultāti ir publicēti 8 publikācijās (3 no tām ir indeksētas *ISI Web of Knowledge*, 2 no tām indeksētas *Scopus*).

Atslēgvārdi: programmatūras testēšana, programmatūras testēšanas sarežģītība, testēšanas metodes, kompleksas sistēmas, daudzaģentu sistēmas

SATURS

Ievads 6

Tēmas aktualitāte un novitāte 6

Promocijas darba mērķi un uzdevumi 8

Pētījumā izvirzītās tēzes 8

Metožu raksturojums 8

Promocijas darba galvenie rezultāti 9

Darba zinātniskā un praktiskā nozīmība 10

Darba rezultātu publikācijas un to prezentācija konferencēs 10

Darba struktūra 12

1. Programmatūras testēšanas sarežģītības jēdziens 14

2. Programmatūras testēšanas metožu sarežģītība un efektivitāte 17

2.1. Ekvivalences klašu un robežgadījumu testēšanas metožu sarežģītība 18

2.1.1. Pamatjēdzieni domēntestēšanā 20

2.1.2. Domēntestēšanas metožu raksturojums 21

2.1.3. Ekvivalences klašu testēšanas metožu sarežģītība 23

2.1.4. Robežvērtību testēšanas metožu sarežģītība 26

2.2. Domēntestēšanas metožu iekļautība 34

2.3. Kombinācijtestēšanas metožu sarežģītība 39

2.4. Modeļbāzēto testēšanas metožu sarežģītība 41

2.5. Testēšanas metožu efektivitātes un produktivitātes salīdzināšana 47

2.6. Nodaļas secinājumi 49

3. Programmatūras testēšanas procesu sarežģītība 52

3.1. Testēšanas procesi, to sarežģītība 52

3.2. Testēšanas sistēma kā kompleksa sistēma 53

3.2.1. Līdzšinējie pētījumi 54

3.2.2. Testēšanas sistēmas sarežģītības 55

3.2.3. Testēšanas sistēma un kompleksu sistēmu īpašības 58

3.2.4. Testēšanas sistēma un komplekso sistēmu principi 60
3.3. Nodaļas secinājumi 63
4. Testēšanas sarežģītības ietekmēšana 65
4.1. SUP modelis un testēšana 65
4.1.1. SUP modeļa jēdziens 67
4.1.2. SUP segmentu interpretācija 75
4.1.3. Testēšanas procesa uzlabošana 77
4.2. Daudzaģentu sistēmu pielietojums sarežģītības samazināšanā 80
4.2.1. Reālās pasaules aprakstīšana ar daudzaģentu modeli 80
4.2.2. Testēšanas procesa modeļa izveide 83
4.2.3. Modeļa izveide un uzturēšana 88
4.3. Ne-IT cilvēku iesaistīšana testēšanā 89
4.3.1. Ne-IT testētāji 90
4.3.2. Ne-IT testētāju intuitīvais testēšanas stils 91
4.3.3. Ne-IT cilvēku apmācība programmatūras testēšanai 93
4.3.4. Ne-IT testētāju darba vadīšanas īpatnības 95
4.3.5. Reālu projektu testēšanas piemēru analīze 97
4.3.6. Ieguvumi no projektu piemēru analīzes 101
4.4. Galalietotāju iesaistīšana izstrādē un testēšanā 102
4.4.1. Ietvaram piemērotās informācijas sistēmas 104
4.4.2. Sistēmas arhitektūra 105
4.4.3. Sistēmas datu ontoloģija 106
4.4.4. Lietotāju saskarnes šablonu DSL 107
4.4.5. Pielietojumi 109
4.5. Nodaļas secinājumi 112
Rezultāti 117
Izmantotā literatūra un avoti 119

IEVADS

Promocijas darbs ir izstrādāts laika posmā no 2006. līdz 2012.gadam Latvijas Universitātes Datorikas fakultātē, kas darba izstrādes laikā atdalījās no Fizikas un matemātikas fakultātes. Darba vadītājs ir profesors Jānis Bičevskis. Šis darbs turpina LU Datorikas fakultātē un pirms tam Fizikas un matemātikas fakultātē tradicionāli veiktos pētījumus programmatūras testēšanas jomā.

Tēmas aktualitāte un novitāte

Programmatūras testēšana kā programmatūras izstrādes procesu sastāvdaļa ir strauji attīstījusies kopš 20.gs. vidus [GH88]. Ir izstrādāta testēšanas teorija, praksē izveidojušās un plaši tiek izmantotas dažādas testēšanas tehnikas un metodes [Ber07]. Tomēr vēl arvien testēšanas kvalitāte nav pietiekama, jo bieži par spīti ieguldītajiem resursiem, kas veido pat 30%-60% no visā programmatūras izstrādē ieguldītajiem resursiem [Par01], programmatūras ekspluatācijas laikā tiek atrastas nopietnas kļūdas. Kā vienu no iemesliem avotos min arī programmatūras testēšanas sarežģītību, nekonkretizējot, kā sarežģītība izpaužas un kā samazināt tās ietekmi uz testēšanas efektivitāti.

Jēdziens *programmatūras testēšanas sarežģītība* avotos parādās jau kopš 20.gs. pēdējām dekādēm, pētot dažādu testēšanas metožu sarežģītību un efektivitāti [Tai80], [Mye92], [FW93a], [ZHM97], [Hut03], [TI11]. Kā metrika programmatūras testēšanas sarežģītībai tiek lietots testkomplekta, ko veido testēšanas metode, apjoms jeb testpiemēru skaits testkomplektā.

Runājot par sarežģītību kā jēdzienu, nav viena vispāratzīta sarežģītības jēdziena definīcijas, tāpat kā nav metrikas, kas viennozīmīgi ļautu novērtēt un salīdzināt dažādu objektu vai sistēmu sarežģītību.

Vārdnīca [DPA+95] terminu *sarežģītība* skaidro: „*Raksturojums, kas parāda ar kādas datu apstrādes problēmas risināšanu saistītās grūtības un ko parasti izsaka kāda šīs problēmas risināšanas gaitā patērējamā resursa terminos. Tā, piemēram, algoritmu var raksturot ar tā izpildes laiku atkarībā no apstrādājamo datu apjoma*”.

Runājot par konkrētas programmatūras testēšanu no tās organizatoriskā viedokļa, var domāt par to, kā par projektu, kura mērķis ir izpildīt ieinteresēto pušu sniegtos uzdevumus.

Darbā tiek aplūkotas divas testēšanas projekta sarežģītības - tehnoloģiskā un organizatoriskā sarežģītība [Bac96], [XL04]. Ar tehnoloģisko sarežģītību saprot tās grūtības, kas rodas tīri tehniski, izpildot testēšanas uzdevumu, piemēram, cik testpiemēru nepieciešams izpildīt, lai notestētu programmatūru atbilstoši kādai testēšanas metodei, bet ar organizatoriskajām – grūtības testēšanas procesu pārvaldībā.

Programmatūras testēšanas tehnoloģiskā sarežģītība reālam projektam var būt ļoti liela. Piemēram, ja pielieto izsmeļošās testēšanas metodi, jau pavisam nelielu programmatūru nav iespējams pilnībā notestēt, jo testpiemēru, kurus nepieciešams izpildīt, ir tik ļoti daudz. Projekta organizatorisko sarežģītību izsaka gan projekta darbībā iesaistīto elementu skaits, piemēram, cilvēku, datoru skaits, elementu dažādība, kā arī mijiedarbības un komunikāciju saišu skaits starp šiem elementiem.

Promocijas darbs ir veltīts praktiski nozīmīgas problēmas – testēšanas kvalitātes – risināšanai, pamatojoties uz pētījumiem par testēšanas sarežģītību – kā tā ietekmē testēšanas procesus un galarezultātus un kā to ietekmēt, lai uzlabotu testēšanas rezultātus.

Darba zinātniskā novitāte: – uzskatot, ka programmatūras testēšanas stratēģiju un politiku liela mērā nosaka un virza tās sarežģītība dažādās izpausmju formās, novērtēta testēšanas tehnoloģiskā sarežģītība domēntestēšanas un citām metodēm un, uztverot programmatūras testēšanas projektu kā kompleksu sistēmu, izstrādāts organizatorisko paņēmienu kopums, kas palīdz pārvarēt programmatūras testēšanas projektu tehnoloģisko sarežģītību:

- Darbā novērtēta testēšanas sarežģītība kā izpildāmo testu skaits 17 dažādiem ekvivalences klašu un robežvērtību testēšanas jeb domēntestēšanas kritērijiem, kā arī izveidota to iekļautības hierarhija.
- Darbā tiek piedāvāts, uztverot testēšanas sistēmu kā kompleksu sistēmu un kompleksu sistēmu darbības principus kā metodisku ieteikumus, mainīt testēšanas procesu organizatorisko pārvaldību, kas ļauj pārvarēt tehnoloģisko sarežģītību. Tiek piedāvāts programmatūras testēšanas procesos pielietot daudzaģentu sistēmas organizatoriskos principus, kas ļauj samazināt lokāli katram izpildītājam testēšanas darbu sarežģītību.
- Darbā tiek piedāvāts testēšanas teorijā oriģināls modelis (SUP modelis) testēšanas efektivitātes paaugstināšanai, kuru raksturo nepieciešamība ņemt vērā

sistēmu izstrādē iesaistīto ieinteresēto pušu atšķirīgu skatījumu uz veidojamo sistēmu. SUP modelis pierāda ne-IT speciālistu iesaistīšanas testēšanā nepieciešamību.

Promocijas darba mērķi un uzdevumi

Promocijas darba galvenais mērķis ir piedāvāt risinājumu programmatūras testēšanas kvalitātes problēmai. Darbā parādīts, ka problēmu vismaz daļēji var atrisināt, mainot testēšanas procesu organizatorisko struktūru tā, lai samazinātu katra testēšanā iesaistītā cilvēka darba lokālo sarežģītību, tādējādi ļaujot samazināt programmatūras testēšanas kopējo tehnoloģisko sarežģītību.

Promocijas darba uzdevumi ir sekojoši:

- Novērtēt programmatūras testēšanas tehnoloģisko sarežģītību, kā testēšanas metožu sarežģītību to ģenerēto testpiemēru skaita nozīmē.
- Izstrādāt priekšlikumus programmatūras testēšanas procesu organizatoriskās struktūras un procesu pārvaldības izmaiņām ar mērķi samazināt testēšanas tehnoloģiskās sarežģītības ietekmi uz testēšanas rezultātiem.

Promocijas darba un tā kopsavilkuma saturs apliecina, ka izvirzītais darba mērķis un formulētie konkrētie uzdevumi ir sasniegti.

Pētījumā izvirzītās tēzes

Promocijas darba ietvaros ir izvirzītas šādas pamata tēzes:

- Programmatūras testēšanas tehnoloģiskākā sarežģītība ir tik liela, ka atbilstoši domēntestēšanas metodei nav iespējams pilnībā notestēt vairumu praksē sastopamo programmatūru.
- Problēmas, ko izraisa programmatūras testēšanas tehnoloģiskā sarežģītība, ir iespējams pārvarēt, mainot testēšanas procesu organizatorisko struktūru un procesu pārvaldības principus.

Metožu raksturojums

Darbā pielietotas gan teorētiskas analīzes, gan praktiskas pielietojuma metodes:

- Novērtēta domēntestēšanas 17 metožu, kombinācijtestēšanas metožu un modeļbāzēto testēšanas metožu sarežģītība kā izpildāmo testpiemēru skaits. Izveidota domēntestēšanas metožu iekļautības hierarhija.
- Novērtēta testēšanas sistēmu atbilstība kompleksajām sistēmām un doti metodiski ieteikumi testēšanas procesu pārvaldībā atbilstoši komplekso sistēmu darbības principiem. Piedāvāts programmatūras testēšanas procesos pielietot daudzaģentu sistēmu organizatoriskos principus.
- Izstrādāts SUP modelis testēšanas efektivitātes paaugstināšanai, kas prasa ņemt vērā programmatūras izstrādē ieinteresēto pušu atšķirīgu skatījumu uz veidojamo programmatūru.
- Izvērtēta ne-IT speciālistu iesaistīšanas testēšanā un programmatūras izstrādē nepieciešamība un ar piemēriem demonstrētas iespējas to veikt.

Promocijas darba galvenie rezultāti

Darbs veltīts testēšanas metodoloģisko, tehnoloģisko un organizatorisko aspektu izpētei ar mērķi nodrošināt uzticamas programmatūras izveidi. Testēšana, kas balstīta uz testēšanas modeļa izveidi ar sekojošu programmu testēšanu atbilstoši izvēlētajam modelim, sastopas ar nopietnām grūtībām – gan modeļa izveide, gan tai sekojoša testēšana ir sarežģīta un prasa lielus resursus. Tādējādi nereti testētāji atļaujas sistemātiskas testēšanas vietā aprobežoties ar dažu tipisku lietošanas gadījumu pārbaudi, kas neizbēgami noved pie programmu izraisītām informācijas sistēmu kļūdām. Darbam ir 4 galvenie rezultāti:

- Testēšanas tehnoloģisko aspektu jomā aplūkots vērtību apgabalu testēšanas modelis un novērtēta testēšanas sarežģītība kā izpildāmo testu skaits dažādiem domēntestēšanas kritērijiem. Izveidota domēntestēšanas metožu iekļautības hierarhija. Identificētas tradicionālās testēšanas, kas balstīta uz vērtību apgabalu testēšanas modeļiem, galvenā problēma – sarežģītība, kas praktiskos lietojumos padara neiespējamu šo testēšanas modeļu lietošanu.
- Lai pārvarētu testēšanas tehnoloģisko sarežģītību, tiek piedāvāts testēšanas sistēmu uztvert kā kompleksu sistēmu un kompleksu sistēmu darbības principus izmantot, lai sasniegtu labākus testēšanas rezultātus. Papildus oriģinālai pieejai testēšanai, darbā izstrādāti metodiski ieteikumi testēšanas procesu pārvaldībai, pamatojoties uz komplekso sistēmu darbības principiem.

- Lai pārvarētu testēšanas tehnoloģiskās sarežģītības problēmu, tiek piedāvāts testēšanas teorijā oriģināls modelis (SUP modelis) testēšanas efektivitātes paaugstināšanai, kuru raksturo nepieciešamība ņemt vērā sistēmu izstrādē iesaistīto dalībnieku atšķirīgu skatījumu uz veidojamo sistēmu. Šis modelis pierāda ne-IT speciālistu iesaistīšanas testēšanā nepieciešamību.
- Tiek piedāvāts programmatūras testēšanas procesu pārvaldībā pielietot daudzaģentu sistēmas organizatoriskos principus, kas ļauj samazināt testēšanas sarežģītību un palīdz risināt testētāju kvalifikācijas problēmas, iesaistot galalietotājus sistēmas izstrādē un testēšanā.

Darba zinātniskā un praktiskā nozīmība

Promocijas darbā ir pētīta programmatūras testēšanas sarežģītība divos virzienos – tehnoloģiskā sarežģītība jeb testēšanas metožu sarežģītība un organizatoriskā jeb testēšanas procesu sarežģītība.

Viens no darbā pētītajiem apgabaliem ir domēntestēšanas metožu sarežģītība. Dažādu testēšanas metožu sarežģītība pasaulē jau ir tikusi pētīta jau ilgstoši. Tomēr ekvivalences klašu testēšanas un robežgadījumu testēšanas metožu sarežģītība nav bijusi pētīta. Darbā apkopotas 17 dažādas avotos minētas domēntestēšanas metodes un novērtētas to sarežģītības augšējās un apakšējās robežas, kā arī izveidots to iekļautības modelis. Tas varētu veicināt testētāju izpratni par domēntestēšanas metožu dažādību, pielietošanas piemērotību atkarībā no testēšanas mērķiem un pieejamajiem resursiem, tādējādi uzlabojot testēšanas efektivitāti.

Otrs promocijas darbā pētītais apgabals ir programmatūras testēšanas sistēma. Ir pētītas testēšanas sistēmas kā kompleksas sistēmas īpašības un rasts izskaidrojums dažādu empīriski atrastu un avotos aprakstītu testēšanas vadīšanas un veikšanas metožu un paņēmienu efektivitātei. Šie pētījumi var veicināt testētāju, viņu vadītāju un citu testēšanā ieinteresēto pušu izpratni par testēšanā notiekošo, palīdzot efektīvāk vadīt testēšanas procesus.

Darba rezultātu publikācijas un to prezentācija konferencēs

Promocijas darba galvenie rezultāti ir atspoguļoti 8 publikācijās.

Publikācijas, kuru tapšanā ir nozīmīgs promocijas darba autores ieguldījums (80%-100%):

- „End-User Development Framework with DSL for Spreadsheets” [Arn11].
- „Complexity of Equivalence Class and Boundary Value Testing Methods” [Arn09].
- „Using the Principles of an Agent-Based Modeling for the Evolution of IS Testing Involving Non-IT Testers” [AA08] (indeksēta *ISI Web of Knowledge*).
- „Use of Non-IT Testers in Software Development” [Arn07] (indeksēta *ISI Web of Knowledge* un *Scopus*).
- „Statistikas sistēmu datu ievades un atskaišu formu izstrādes metodoloģija” [Arn00].

Publikācijas, kuru tapšanā ir mazāk nozīmīgs promocijas darba autores ieguldījums (40%-60%):

- „Evolutionary Reduction of the Complexity of Software Testing by Using Multi-Agent System Modeling Principles” [AA11].
- „Opportunities to Improve Software Testing Processes on the Basis of Multi-Agent Modeling” [AA09a] (indeksēta *ISI Web of Knowledge* un *Scopus*).
- „Using the Sponsor-User-Programmer Model to Improve the Testing Process” [AA09b].

Par darba rezultātiem autore ir referējusi 3 starptautiskās zinātniskās konferencēs un 6 vietējās zinātniskās konferencēs:

- Referāts „*Sarežģītības vadīta programmatūras testēšana*” LU 70. zinātniskajā konferencē, Informācijas tehnoloģiju sekcijā, 2012. gadā, Rīgā, Latvijā.
- Referāts „End-User Development Framework with DSL for Spreadsheets” starptautiskās konferences 10th International Conference, BIR 2011, Associated Workshops and Doctoral Consortium, 2011. gadā, Rīgā, Latvijā.
- Referāts „*Programmatūras testēšanas sarežģītība*” Apvienotajā Pasaules latviešu zinātnieku III kongresā un Letonikas IV kongresā „Zinātne, sabiedrība un nacionālā identitāte”, sekcijā „Tehniskās zinātnes”, 2011. gadā, Rīgā, Latvijā.
- Referāts „*Equivalence Class and Boundary Value Testing Methods – Well-Known and Unexplored”* Testēšanas profesionāļu 10. konferencē "Testēšanas teorija un prakse", 2009. gadā, Rīgā, Latvijā.
- Referāts „*Sarežģītības noteikta testēšana*” LU 97. zinātniskajā konferencē, Informācijas tehnoloģiju sekcijā, 2009. gadā, Rīgā, Latvijā.

- Referāts „Using the Principles of an Agent-Based Modeling for the Evolution of IS Testing Involving Non-IT Testers” starptautiskajā konferencē Baltic DB & IS, 2008. gadā, Tallinā, Igaunijā.
- Referāts „*Testēšanas sarežģītība* LU 76. zinātniskajā konferencē, Informācijas tehnoloģiju sekcijā, 2008. gadā, Rīgā, Latvijā.
- Referāts „Use of Non-IT Testers in Software Development” konferencē 8th International PROFES (Product Focused Software Development and Process Improvement) conference, 2007.gadā, Rīgā, Latvijā.
- Referāts „*Testēšanas sarežģītība*” LU 65. zinātniskajā konferencē, Informācijas tehnoloģiju sekcijā, 2007. gadā, Rīgā, Latvijā.

Kā līdzautore darba autore ir tikusi pārstāvēta 2 konferencēs:

- Referāts Arnicans, G., Arnicane, V. “*Simplified design of test cases based on models*” konferencē 12th Annual Software Testing Conference: Forming Basis of Globally Mature, May 26, 2011, Rīgā, Latvijā.
- Referāts V. Arnicāne, G. Arnicāns, J. Bičevskis, J. Borzovs „*Vēlmes, realitāte un tendences programmatūras testēšanā*” konferencē Latvijas testēšanas profesionāļu 9.konference “Testēšanas teorija un prakse”, 22.maijā, 2008, Rīgā, Latvijā.

Darba struktūra

Promocijas darbs ir iepriekš uzskaitītajās publikācijās aprakstītā pētnieciskā un praktiskā darba loģisks nobeigums.

Promocijas darbs izklāstīts 124 lapaspusēs. Darbs satur 29 attēlus, 2 tabulas, ievadu, četras pamata nodaļas, nobeigumu, literatūras avotu sarakstu.

Darbam ir sekojoša struktūra:

- Darba pirmajā nodaļā ir aplūkots sarežģītības jēdziens un programmatūras testēšanas sarežģītības jēdziens, to vēsturiskā attīstība un izpētē sasniegtie rezultāti pasaulē.
- Darba otrajā nodaļā analizētas programmatūras testēšanas tehnoloģiskā sarežģītība metožu ģenerēto testkomplektu apjomu nozīmē. Ir apkopotas 17 domēntestēšanas metodes, novērtēta to sarežģītība un izveidota to hierarhija iekļautības nozīmē. Dots ieskats kombinācijtestēšanas un modeļbāzēto testēšanas metožu sarežģītībā.

- Darba trešajā un ceturtajā nodaļās analizētas programmatūras testēšanas organizacionālās sarežģītības un iespējas tās ietekmēt ar mērķi uzlabot testēšanas rezultātus. Trešajā nodaļā programmatūras testēšana aplūkota no projektu sarežģītības viedokļa, kā arī īpaši pētīta testēšanas sistēma kā kompleksa sistēma.
- Darba ceturtajā nodaļā pētītas dažādas iespējas ietekmēt programmatūras testēšanas sarežģītību, mainot testēšanas sistēmas organizatoriskās sarežģītības. Piedāvāts izmantot SUP modeli, analizētas iespējas izmantot ne-IT testētājus, daudzaģentu sistēmu organizatoriskos principus testēšanas procesu organizācijā un gala-lietotājus kā programmētājus programmatūras izstrādē.
- Nobeiguma nodaļā uzskaitīti darbā gaitā radušies secinājumi un turpmākie programmatūras testēšanas sarežģītību pētījumi virzieni.

1. PROGRAMMATŪRAS TESTĒŠANAS SAREŽĢĪTĪBAS JĒDZIENS

Runājot par sarežģītību kā jēdzienu, cik darba autorei zināms, nav viena vispāratzīta sarežģītības jēdziena skaidrojuma vai definīcijas, tāpat kā nav metrikas vai metriku kopas, kas viennozīmīgi ļautu novērtēt uz salīdzināt dažādu objektu, sistēmu sarežģītību.

Vārdnīca [DPA+95] terminu „sarežģītība" skaidro: „*Raksturojums, kas parāda ar kādas datu apstrādes problēmas risināšanu saistītās grūtības un ko parasti izsaka kāda šīs problēmas risināšanas gaitā patērējamā resursa terminos. Tā, piemēram, algoritmu var raksturot ar tā izpildes laika atkarību no apstrādājamo datu apjoma*".

Pētījumos tiek meklēta tāda sarežģītības definīcija un tās metrika, kas būtu lietderīga, t.i., būtu ne vien teorētiski, bet arī praktiski pielietojama un ļautu salīdzināt dažādus objektus.

Programmatūras testēšana ir process, ko veic cilvēku, datortehnikas un resursu kopums, kas veidots ar mērķi realizēt ieinteresēto pušu dotos uzdevumus dotā laika un budžeta ietvaros.

Programmatūras testēšanas uzdevums ir dod informāciju par testējamo programmatūru. Informācijas vērtība ir atkarīga no konteksta, kādā tā tiek pielietota. Informācija ir apstrādāti dati vai fakti par objektiem, notikumiem vai personām, kas ir jēgpilni tās saņēmējam.

Programmatūras testēšanas rezultātus saņem dažādas programmatūras izstrādē iesaistītās puses – izstrādātāji, lietotāji, izstrādātāju vadītāji, lietotāju vadītāji, izstrādes finansētāji, u.c.. Katrai no šīm pusēm ir savas intereses attiecībā uz testējamo programmatūru un tādēļ arī tās sagaida dažāda rakstura informāciju par to. Piemēram, vadītājus interesē programmatūras kvalitātes novērtējums, izstrādātājiem ir svarīgas atrastās smaga rakstura (viņu skatījumā) kļūdas un to apraksta kvalitāte, lietotājiem ir svarīgas ne tikai atrastās funkcionālas problēmas, bet arī lietojamības problēmas. Katra no pusēm var dot savām interesēm atbilstošu uzdevumu testētājiem. Parasti testētājiem nepieciešams veikt vairākus uzdevumus paralēli, bet katrs no tiem var prasīt dažādu pieeju testēšanā. Papildus tam testēšanai tiek atvēlēts ierobežots laika, budžeta un citu resursu apjoms.

Tātad programmatūras testēšanai jābūt efektīvai, produktīvai un lietderīgai. Testēšanas efektivitāti izsaka pakāpe, kādā testēšana sasniedz vēlamo rezultātu, piemēram, atrod funkcionālas, lietojamības vai cita veida problēmas, novērtē

programmatūras kvalitāti. Testēšanas produktivitāti izsaka pakāpe, kādā tā sasniedz vēlamo rezultātu zināmā laika periodā. Bet testēšanas lietderīgumu izsaka pakāpe, kādā tā palīdz iesaistīto pušu mērķu sasniegšanai.

Darba jeb projekta sarežģītību var izteikt divējādi [Bac96]:

- Ar kvantitatīvajām metrikām, piemēram, uzdevumā iesaistīto savstarpēji atkarīgo elementu, apakšuzdevumu, komponenšu, utml. skaits.
- Ar kvalitatīvajām metrikām, piemēram, kā uzdevuma veikšanas grūtību līmeņa novērtējums.

Pirmajā gadījumā testēšanas sarežģītību var raksturot ar, piemēram, testējamās programmatūras prasību skaitu, komponenšu skaitu, prasību skaitu, pielietojamo testēšanas metožu izpildei nepieciešamo testpiemēru skaitu.

Otrajā gadījumā testēšanas sarežģītību ietekmē dažādu faktoru kopums, piemēram, testējamās programmatūras kritiskums, tās uztveramības sarežģītība, specifikācijas kvalitāte, iesaistīto pušu doto uzdevumu skaidrība, pieejamo resursu pietiekamība.

Principā sistēmu sarežģītības novērtējums ir atkarīgs no novērtētāja un viņa izmantotā metriku kopuma [Gra86]. Programmatūras testēšanas sarežģītības novērtējums nav izņēmums, it sevišķi kvalitatīvo metriku novērtējums ir subjektīvs un atkarīgs no novērotāja jeb vērtētāja.

Kvantitatīvās metrikas vairāk ir pielietojamas programmatūras testēšanas metožu sarežģītības novērtēšanā, novērtējot, piemēram, testpiemēru skaitu metodes veidotā testkomplektā, testkomplekta veidošanas un/vai uzturēšanas algoritmu izpildes laika ilgumu.

Kvalitatīvās metrikas vairāk izmantojamas testēšanas procesu organizatorisko aspektu novērtējumos. Piemēram, testpiemēra kvalitāte – cik informējošs ir katrs testpiemērs, tātad, cik daudz informācijas tas dos gan tad, ja tā izpilde beigsies veiksmīgi, gan arī tad, ja neveiksmīgi. Līdzīgi arī - cik viegli vai grūti savu darbu ir veikt testētājam – vai viņam darba apjoms un paveikšanas iespējas ir šķiet aptveramas un saprotamas.

Kopumā var teikt, ka programmatūras testēšanas sarežģītību veido tehnoloģiskā sarežģītība jeb pielietoto testēšanas metožu sarežģītība un organizatoriskā jeb testēšanas procesu sarežģītība. Tehnoloģisko sarežģītību var izteikt kā funkciju no testēšanas metožu tehnoloģiskajiem aspektiem, piemēram, testkomplekta, ko veido testēšanas metode apjoms, testkomplekta izveidošanas algoritma sarežģītība, testkomplekta uzturēšanas algoritma sarežģītība. Testēšanas procesu sarežģītību izsaka testēšanā

iesaistīto cilvēku skaits, to kvalifikācija, testēšanas organizacionālais ietvars, testēšanā ieinteresēto pušu doto uzdevumu sarežģītība, resursu – budžeta, laika – apjoms, testējamās programmatūras apjoms, sarežģītība un kritiskums, utml..

2. PROGRAMMATŪRAS TESTĒŠANAS METOŽU SAREŽĢĪTĪBA UN EFEKTIVITĀTE

Nav vienas vispāratzītas programmatūras testēšanas metožu sarežģītības metrikas.

Tai [Tai80] definē testēšanas sarežģītību kā testpiemēru skaitu, kas nepieciešami, lai demonstrētu programmas korektību.

Myers [Mye92] uzskata, ka testēšanas sarežģītība nozīmē apjomu resursiem, kas nepieciešami testēšanai. Viņa skatījumā testēšanas sarežģītību ietekmē testējamās programmatūras sarežģītība (tās garums, *McCabe* sarežģītība [MB89]), ceļa izpildes sarežģītība un testkomplekta izveides sarežģītība, kā arī testkomplekta izpildes sarežģītība.

Dažādi autori ir pētījuši dažādu testēšanas metožu sarežģītību – to veidotā testkomplekta apjomu, testkomplekta izveides un uzturēšanas algoritmu sarežģītību. *Frankl* un *Weyuker* demonstrēja testēšanas metožu savstarpējās iekļautības jēdzienu, izmantojot datu plūsmas testēšanas metodes [FW93b]. *Zhu et. al.* ir parādījuši kombināciju testēšanas metožu iekļautību [ZHM97].

Hutchenson uzsver, ka testpiemēru skaits testkomplektā nav praksē labi pielietojams sarežģītības mērs, tādēļ, ka testpiemēri var būtiski atšķirties [Hut03], piemēram, to izpildei nepieciešamo testēšanas resursu ziņā. Viņa piedāvā ierobežot testpiemēru apjomu, piemēram, ar no klaviatūras ievadāmo simbolu skaitu vai ar peles klikšķu skaitu, kas nepieciešami, lai izpildītu vienu testpiemēru. Šādā veidā viņa panāk testpiemērus ar vienādu sarežģītību – tie patērē vienādu laika un citu resursu daudzumu gan testpiemēru izpildei, gan dokumentēšanai.

Tsui un *Iriele* [TI11] norāda, ka testēšanas sarežģītība ir metrika, kas ietver visu testēšanas aktivitāšu sarežģītības. Šajā gadījumā ar testēšanas aktivitātēm tiek saprastas tādas darbības kā testpiemēru specificēšana, testēšanas vides sagatavošana un notīrīšana, testpiemēru izpildes un rezultātu dokumentēšana un analīze.

Mēģinot samazināt testēšanas sarežģītību, testētāji tajā pašā laikā cenšas nezaudēt testēšanas efektivitāti attiecībā pret ieinteresēto pušu dotajiem uzdevumiem.

Turpmākajās 5 apakšnodaļās tiks iztirzāta domēntestēšanas metožu sarežģītība, to hierarhija iekļautības nozīmē, dots ieskats kombinācijtestēšanas un modeļbāzētās testēšanas metožu sarežģītības novērtējumos, kā arī pētījumos par testēšanas metožu un produktivitātes salīdzināšanu.

2.1. Ekvivalences klašu un robežgadījumu testēšanas metožu sarežģītība

Ekvivalences klašu un robežgadījumu testēšanas metožu jeb domēntestēšanas metožu pamatpieeja ir sadalīt visu testējamā objekta ievaddomēnu apakšdomēnos jeb ekvivalences klasēs tā, ka programmas ievadi no viena apakšdomēna tiek apstrādāti vienādā veidā, tas ir, izraisa programmai viena veida izvadu vai uzvedību [SLS07], [Vee02]. Katrs programmas ievads reprezentē visus ievadus no tā apakšdomēna, kuram pats pieder. Šīs metodes pamatā ir pieņēmums, ka, ja notestē programmu ar dažiem labi izvēlētiem reprezentantiem no vienas klases, tad tiek atrastas visas vai vairums no kļūdām, kuras parādītos, ja tiktu testēti visi klases elementi [SLS07], [Vee02], [Bei90], [Bei95], [Jor95], kā arī, ja reprezentanti kļūdu neuzrāda, tad kļūda neparādīsies, programmu darbinot arī ar citiem klases elementiem [Bei90], [Bei95], [Jor95].

Programmas ievaddomēna ekvivalences klases var tikt iegūtas ar programmas strukturālo analīzi – ceļu analīzi un funkcionālo analīzi – sadalīšanu ekvivalences klasēs, domēnanalīzi, robežvērtību analīzi [Bei90].

Programmas ceļu analīzes pieeja ir pazīstama kopš 20.gs. 70-tajiem gadiem. Tās pamatā ir pieņēmums, ka domēna kļūda parādās tad, ja programmas izpilde aiziet pa nepareizu ceļu tās izpildes kokā [How76]. Izmantojot simbolisko izpildi, ir iespējams iegūt ceļu izpildes nosacījumus [BBK75] vai nevienādību sistēmu [BBS+79], [Cla76], [How75], [How76], kuras risinājums noteiks to ievaddomēna apakškopu, kas nodrošina konkrētā ceļa izpildi.

Sadalīšana ekvivalences klasēs tika pētīta kopš 20.gs. 80-tajiem gadiem [Cla81], [HT90], [JW89], [RC85], [WJ91]. Šī metode sadala ievaddomēnu ekvivalences klasēs, pamatojoties uz programmas specifikāciju [Bei90], [Bei95], [Jor95], [SLS07], [Vee02]. Pirmais solis ir katram programmas parametram noteikt pieļaujamo vērtību kopu jeb domēnu. Tās ir pieļaujamo vērtību ekvivalences klases, viena katram parametram. Otrais solis ir noteikt katra parametra nepieļaujamo vērtību ekvivalences klasi. Nepieļaujamā vērtība ir tāda, ko ir iespējams „iedot" programmas parametram, bet kas nav iekļauta pieļaujamo vērtību klasē. Nākamajos soļos ekvivalences klases tiek uzlabotas, sadalot sīkāk tā, lai dažādas specifikācijas prasības, dažāda programmatūras uzvedība atbilstu dažādām ekvivalences klasēm.

Pēc tāda paša principa ekvivalences klasēs var sadalīt arī programmas izvadvērtību domēnu. Rezultātā ievaddomēns ekvivalences klasēs tiek saskaldīts pēc

principa, ka, ja divi programmas ievadi izraisa izvadu, kas pieder dažādām programmas izvadddomēna ekvivalences klasēm, tad tie pieder arī pie dažādām ievaddomēna ekvivalences klasēm [SLS07].

Robežvērtību analīze īpašu izmanību pievērš ekvivalences klašu robežvērtībām, jo prakse rāda, ka robežvērtības bieži parāda kļūdas programmā. Robežvērtību testēšanas metodes testpiemēros izvēlas robežvērtības, izņēmumu jeb speciālās, īpašās vērtības, kā arī vērtības, kuras ir tuvu iepriekšminētajām vērtībām.

Ir tādu metožu grupa, kuras dažādos literatūras avotos tiek sauktas par domēntestēšanas metodēm vai domēnanalīzes metodēm [MCC+87]. Šīs metodes ņem vērā arī savstarpējo atkarību vai mijiedarbību starp programmas ievadparametriem [Bei90], [Bei95], [MCC+87], [WC80], [CHR82], [AWZ92]. Šajās metodēs ievaddomēnu parasti aplūko kā ģeometrisku figūru, bet tās šķautnes uzskata par robežām. Vairumā gadījumu tiek analizēti domēni ar lineārām robežām [AWZ92], [Bei90], [CHR82], [JW94], [MCC+87], [WC80], taču ir metodes, kas atļauj pētīt arī nelineārās robežas [Bei95], [HF98], [JW94], [ZA92].

Tātad ekvivalences klases nosaka, izmantojot ceļu analīzi un/vai sadalīšanu ekvivalences klasēs. Viena vērtība no katras klases var tikt izvēlēta kā tās reprezentants testēšanai. Ja programmai ir vairāki parametri un tiem, savukārt, ir vairākas ekvivalences klases, nepieciešams izvēlēties kādu stratēģiju, kā savstarpēji kombinēt to reprezentantus kā testpiemēru ievadvērtības. Līdzīga situācija ir arī ar robežvērtību analīzes rezultātā iegūtajām vērtībām. Atbilstošās domēntestēšanas metodes turpmāk šajā darbā tiks sauktas attiecīgi par ekvivalences klašu testēšanas (*equivalence class testing* (ECT)) metodēm un robežgadījumu testēšanas (*boundary value testing* (BVT)) metodēm.

Šajā darbā par domēntestēšanas metodes sarežģītību tiks saukts testpiemēru skaits vismazākajā testkomplektā, ko ģenerē šī metode. Testēšanas metodes var salīdzināt vēl pēc diviem parametriem – relatīvās efektivitātes kļūdu atrašanā un cenas [WWH91]. Metodes sarežģītība ir cieši saistīta ar tās izmaksām. Ja sarežģītība ir liela, tas nozīmē, ka metodes ģenerētājā testkomplektā iespējams liels skaits testpiemēru. Pieaugot testkomplekta apjomam, nepieciešams vairāk resursu testēšanai, piemēram, testētāju, datortehnikas, programmatūras, laika, naudas. No otras puses, jāņem vērā metodes efektivitāte. Metode tiek uzskatīta par efektīvu tad, ja programmatūra, kas notestēta atbilstoši tai, ir gandrīz korekta [WWH91]. Bet metodei ir jābūt arī racionālai – ja testpiemērs uzrāda kļūdu, ir labāk, ja ir neliels skaits gadījumu, ko pārbaudīt, uzskatot, ka tie varētu būt izraisījuši kļūdu.

Tādējādi ir ļoti svarīgi plānojot testēšanu ņemt vērā gan metožu sarežģītību, gan to efektivitāti.

2.1.1. Pamatjēdzieni domēntestēšanā

Pieņemsim, ka P ir programma ar N ievadmainīgajiem X_i un kur $1 \le i \le N$. Katram parametram ievaddomēns D_i ir sadalīts M_i ekvivalences klasēs un to galapunkti ir robežvērtības.

Robežvērtību testēšanas jēga ir pārbaudīt situācijas, kad ievadparametriem ir katras ekvivalences klases ekstremālās vērtības (maksimālā, minimālā), mazliet mazākas (par mazu vērtību ε) vai mazliet lielākas nekā ekstremālās vērtības, kā arī kad vērtība ir *nomināla* – tā atrodas ekvivalences klases iekšienē ievērojami tālāk no klases galapunktiem nekā par attālumu ε. Vērtību ε nosaka testētājs atkarībā no domēna datu tipa un specifikācijas, piemēram, veseliem skaitļiem ε būs 1, bet vērtība, kas izsaka naudas vienības latos ar precizitāti līdz santīmam kā ε ir lietderīgi pieņemt 0.01.

Katram parametram X_i atbilstošā domēna D_i, katru sakārtotu elementu ekvivalences klasi d_{ij} grafiski var parādīt kā 2.1. attēlā. Klases minimālā robežvērtība ir $x_{ij\ min}$, maksimālā robežvērtība ir $x_{ij\ max}$, kur $1 \le j \le M_i$. Klases nominālā vērtība ir $x_{ij\ nom}$. Vērtības $x_{ij\ min-}$, $x_{ij\ max-}$ ir mazliet mazākas par atbilstošajām robežvērtībām, bet $x_{ij\ min+}$, $x_{ij\ max+}$ ir mazliet lielākas. Pieraksta vienkāršības labad darbā tiks lietoti apzīmējumi *min-, min, min+, nom, max-, max, max+* apzīmējumu $x_{ij\ min-}$, $x_{ij\ min}$, $x_{ij\ min+}$, $x_{ij\ nom}$, $x_{ij\ max-}$, $x_{ij\ max}$, $x_{ij\ max+}$ vietā, kur tas nevar izraisīt pārpratumus.

Katram *i, j* ir spēkā sekojošas nevienādības, kad $1 \le i \le N$ un $1 \le j \le M_i$:

$x_{ij\ min} - x_{ij\ min-} \le \varepsilon_{ij}$ $x_{ij\ min+} - x_{ij\ min} \le \varepsilon_{ij}$ $x_{ij\ nom} - x_{ij\ min} \ge \varepsilon_{ij}$

$x_{ij\ max+} - x_{ij\ max} \le \varepsilon_{ij}$ $x_{ij\ max} - x_{ij\ max-} \le \varepsilon_{ij}$ $x_{ij\ max} - x_{ij\ nom} \ge \varepsilon_{ij}$

$x_{ij\ min-}$ $x_{ij\ min}$ $x_{ij\ min+}$ $x_{ij\ nom}$ $x_{ij\ max-}$ $x_{ij\ max}$ $x_{ij\ max+}$ X_i

2.1. att. **Ekvivalences klase d_{ij}: [$x_{ij\ min}$, $x_{ij\ max}$], tās robežvērtības $x_{ij\ min}$, $x_{ij\ max}$, iekšējie OFF punkti $x_{ij\ min+}$, $x_{ij\ max-}$, ārējie OFF punkti $x_{ij\ min-}$, $x_{ij\ max+}$ un nominālā vērtība $x_{ij\ nom}$**

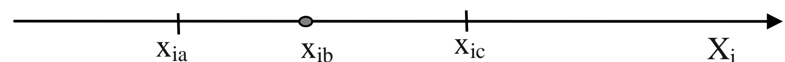
Vērtības, kas ir mazliet lielākas par minimālo vērtību ($x_{ij\ min+}$), mazliet mazākas par maksimālo vērtību ($x_{ij\ max-}$), šajā darbā tiks sauktas par *iekšējiem OFF punktiem* (tās atrodas ekvivalences klases iekšpusē), bet vērtības, kas ir mazliet mazākas par minimālo vērtību ($x_{ij\ min-}$) vai mazliet lielākas par maksimālo vērtību ($x_{ij\ max+}$)– par *ārējiem OFF punktiem*.

Tātad ir divi pieņēmumi:

Katram parametram X_i visu ekvivalences klašu apvienojums veido domēnu D_i. Tādējādi, $D_i = \bigcup_{j=1}^{M_i} d_{ij} \quad \forall i, j$, kur $1 \le i \le N$ un $1 \le j \le M_i$.

Ekvivalences klasēm nav kopīgu vērtību, tas ir, to šķēlums ir tukša kopa– $\forall i, j, k$, kur $1 \le i \le N$, $1 \le j \le M_i$, $1 \le k \le M_i$ un $j \ne k$ $d_{ij} \bigcap d_{ik} = \varnothing$.

Kaut gan katram parametram ekvivalences klasēm nav kopīgu vērtību, tām var būt kopīgas robežvērtības. Piemēram, kā parādīts 2.2. attēlā, ja X_i domēnu veido slēgts intervāls $[x_{ia}, x_{ib}]$ un no kreisās puses atvērts, bet no labās – slēgts intervāls $(x_{ib}, x_{ic}]$, tad tiem ir kopīga robežvērtība x_{ib}. Parametra X_i visu kopīgo robežvērtību kopas lielums darbā tiks apzīmēts ar L_i.



2.2. att. **Kopējā robežvērtība x_{ib} intervāliem $[x_{ia}, x_{ib}]$ un $(x_{ib}, x_{ic}]$, L_i=1**

Katram parametram X_i vērtības, kas pieder tā domēnam D_i sauc par *pieļaujamām* (*valid*) vērtībām, bet visas pārējās vērtības – par *nepieļaujamām* (*invalid*). Arī nepieļaujamās vērtības var tikt sadalītas ekvivalences klasēs. Parametra X_i nepieļaujamo vērtību ekvivalences klašu kopas apjoms darbā tiks apzīmēts ar Q_i.

Robežgadījumu testēšanas metodes ir pielietojamas arī gadījumos, kas parametra domēnu veido diskrētu vērtību kopa. Šādos gadījumos ir svarīgi atrast funkciju, saskaņā ar kuru domēna elementus var sakārtot, piemēram, kā mēnešus Jan, Feb, Mar, [..] Dec, simbolu rindu leksikogrāfiskā secība, vai datu secība izkrītošajā izvēlnē.

2.1.2. Domēntestēšanas metožu raksturojums

Domēntestēšanas metodes sarežģītība ir mazākā iespējamā metodes ģenerētā testkomplekta apjoms jeb mazākais testpiemēru, kas nepieciešami, lai notestētu programmatūru atbilstoši konkrētajai metodei, skaits.

Katras testēšanas metodes būtiska sastāvdaļa ir adekvātuma kritērijs (*adequacy criterion*). Adekvātuma kritērijs nosaka, kādi testpiemēri tiek izvēlēti, vai testu kopa ir adekvāta metodei, kā arī apsvērumus, kas jāņem vērā testēšanas procesā [ZHM97].

Domēntestēšanas metožu adekvātuma kritērijus raksturo trīs aspekti:

- Kāda veida vērtības izmantot testēšanā, piemēram, tikai robežvērtības vai tikai pieļaujamo vērtību klašu reprezentantus.
- Domēna klašu pārklājuma princips.

- Stratēģija, kā izvēlētās vērtības tiek kombinētas testpiemēros saskaņā ar datu pārklājuma principu.

Pirmais aspekts ņem vērā testējamo datu semantisko informāciju atbilstoši testēšanas metodes nosacījumiem, piemēram, vai izmanto tikai robežvērtības, vai izmanto kāda veida OFF punktus, vai ar nepieļaujamajām vērtībām testē, vai nē.

Otrais aspekts atspoguļo testēšanas metodes stratēģiju testējamo datu pārklājumu iegūšanā. Vienkāršais pārklājuma kritērijs *katrs-izmantots* neņem vērā, kā no dažādu parametru domēniem izvēlētās vērtības tiek savstarpēji kombinētas, bet sarežģītāki pārklājuma kritēriji, piemēram, pāru pārklājums, prasa veidot dažādu parametru vērtību kombinācijas.

Katrs-lietots (*1-pakāpes*, *each-used*, *1-wise*) pārklājums prasa, ka katra parametra katra izvēlētā vērtība ģenerētajā testkomplektā tiek izmantota vismaz vienu reizi [GOA05].

Pāru (*2-pakāpes, pair-wise, 2-wise*) pārklājums prasa, ka testkomplekta testpiemēros parādās katrs iespējamais pāris, ko var izveidot no dažādu parametru vērtībām.

t-pakāpes (*t-wise*) pārklājums nosaka, ka testkomplekta testpiemēros jābūt jebkurai iespējamai kombinācijai no t parametru vērtībām. N-pakāpes pārklājums ir specgadījums no t-pakāpes pārklājuma, kur N ir programmas parametru skaits.

Pieaugot pārklājumā līmenim *t*, pieaug arī testkomplekta apjoms. *Katrs-lietots* un *N-pakāpes* pārklājumi tiek plaši izmantoti domēntestēšanas metodēs.

Trešais aspekts nosaka, kādā veidā tiek realizēts izvēlēto datu kombinatoriālais pārklājums. Piemēram, pārklājums *katrs-lietots* var tikt sasniegts dažādi:

- Katra parametra katrai izvēlētajai vērtībai tiek ģenerēts atsevišķs testpiemērs, kurā visiem pārējiem parametriem ir nominālvērtības. Šajā gadījumā testkomplekta apjoms ir visu parametru izvēlēto vērtību summa.
- Katrā testpiemērā visu parametru vērtības ir kādas no izvēlētajām vērtībām. Rezultātā testkomplekta apjoms būs vienāds tā parametra izvēlēto vērtību kopas apjomu, kam tā ir vislielākā.

Otrās stratēģijas priekšrocība ir mazāks testkomplekta apjoms, taču, ja kāds no testpiemēriem atklāj kļūdu, ir ļoti grūti pateikt, kāpēc. Nākas veikt papildus izpēti, lai noskaidrotu, kura parametra vērtība bija kļūdas atradēja. Pirmās stratēģijas priekšrocība ir iespēja uzreiz vainot izmantoto robežvērtību, jo pārējās vērtības ir nominālas. To sauc par *vienas kļūdas pieņēmumu* (*single fault assumption*), kas nosaka, ka ļoti reti kļūdu programmatūras uzvedībā izraisa vairāku kļūdu kombinācija programmatūras pirmkodā

[Jor95]. Un tādēļ testpiemērā pietiek izmantot tikai vienu potenciālu kļūdas uzrādītāju vērtību.

Divu kļūdu pieņēmums nozīmē, ka testpiemēros robežvērtības liek diviem parametriem, kamēr pārējiem parametriem atstāj nominālvērtības.

Vispārīgi var runāt par *N kļūdu pieņēmumu*, kad metode prasa katrā testpiemērā pielietot robežvērtības visiem programmas ievadparametriem.

Ja testkomplekts ir izveidots atbilstoši vienas kļūdas pieņēmumam un kāds testpiemērs uzrāda kļūdu, tas ir labs iemesls, lai uzskatītu, ka parametrs, kuram bija robežvērtība, tiek nekorekti apstrādāts. Taču šāds testkomplekts būs lielāks jeb sarežģītāks nekā testkomplekts, kas veidots ar N kļūdu pieņēmumu. No otras puses – ja neveiksmīgi beidzas testpiemērs ar n kļūdu pieņēmumu, ir daudz sarežģītāk pateikt, kura parametra vai kuru parametru vērtības netika pareizi apstrādātas.

2.1.3. Ekvivalences klašu testēšanas metožu sarežģītība

Turpmāks tiks pārskatas domēntestēšanas metodes, kas aprakstītas literatūrā. Ir divu veidu domēntestēšanas metodes – ekvivalences klašu testēšanas metodes un robežvērtību testēšanas metodes. Vairumā gadījumu literatūrā tiek aprakstīts tikai, kādas vērtības izmantot testēšanā, bet netiek noteikts, kā tās kombinēt testpiemēros, tāpat tikai dažām metodēm literatūrā ir atrodams sarežģītības novērtējums.

Attēlu vienkāršības labad tajos tiks demonstrēta situācija ar kādu programmu Z, kurai ir divi parametri X_1 un X_2. Parametra X_1 domēns ir intervāls $[x_{1a}, x_{1d}]$, kurš ir sadalīts trīs ekvivalences klasēs - $[x_{1a}, x_{1b})$, $[x_{1b}, x_{1c})$, un $[x_{1c}, x_{1d}]$. Parametra X_2 domēns ir intervāls $[x_{2u}, x_{2z}]$, kurš ir sadalīts divās ekvivalences klasēs - $[x_{2u}, x_{2v})$ un $[x_{2v}, x_{2z}]$. Ārpus domēna ekvivalences klasēm ir nepieļaujamās vērtības. Piemēram, parametram X_1 nepieļaujamo vērtību ekvivalences klases ir intervāli (∞, x_{1a}) un (x_{1d}, ∞), līdzīgi parametram X_2 – (∞, x_{2u}) un (x_{2z}, ∞).

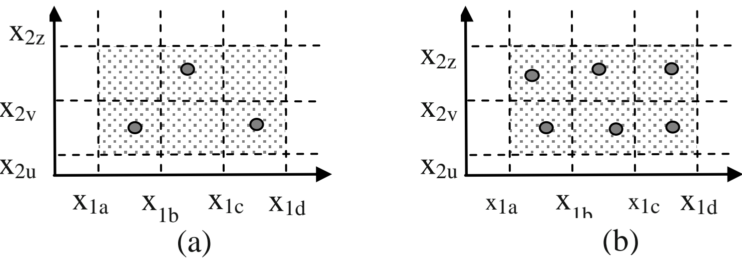
Ja bezgalība ir pieļaujamo vērtību ekvivalences klases robeža, tad testēšanā reāli to aizvieto ar galīgu robežu – maksimālo vai minimālo robežu, ko dators var lietot atbilstošajam datu tipam. Nepieļaujamo vērtību ekvivalences klasēm robežvērtības netiek pētītas [SLS07].

2.1.3.1. Vājā ekvivalences klašu testēšanas metode

Vājā ekvivalences klašu testēšanas metode pārbauda vienu reprezentantu no katra parametra pieļaujamo vērtību katras ekvivalences klases [CBC01], [Cop03], [Jor95], [KBP02], [KFN99], [May04], [NJH01], [Vee02]. Metodes pamatā ir vienas kļūdas

pieņēmums. To lietojot, tiek pieņemts, ka ir pietiekami, ja katru ekvivalences klasi testē vienreiz.

Piemēra programmai Z vājā ekvivalences klašu testēšanas metode ģenerē trīs testpiemērus, piemēram, kā tas attēlots 2.3(a). attēlā. Testpiemēru skaits sakrīt ar parametra X_1 ekvivalences klašu skaitu, jo tam ir lielākais skaits ekvivalences klašu no abiem parametriem – 3.



2.3. att. **Testpiemēri vājajai ekvivalences klašu testēšanas metodei (a) un stiprajai ekvivalences klašu testēšanas metodei (b)**

Vispārīgi, ja programmai ir N parametri, vājā ekvivalences klašu testēšanas metode rada testkomplektu, kura apjoms ir $\max_{i=1}^{N}(M_i)$.

2.1.3.2. Stiprā ekvivalences klašu testēšanas metode

Stiprā ekvivalences klašu testēšanas metode [Jor95], [MS01] izmanto N kļūdu pieņēmumu. Testkomplektā iekļauj testpiemēru no visu parametru ekvivalences klašu Dekarta reizinājuma katra elementa, piemēram, kā redzams 2.3(b). attēlā.

Stiprā ekvivalences klašu testēšanas metode ļauj sasniegt divus mērķus – pārklāt visas ekvivalences klases un pārbaudīt programmas visas dažādo ievadu veidu kombinācijas.

Vispārīgā gadījumā testkomplekta apjoms šai metodei ir $\prod_{i=1}^{N} M_i$.

2.1.3.3. Robustā vājā ekvivalences klašu testēšanas metode

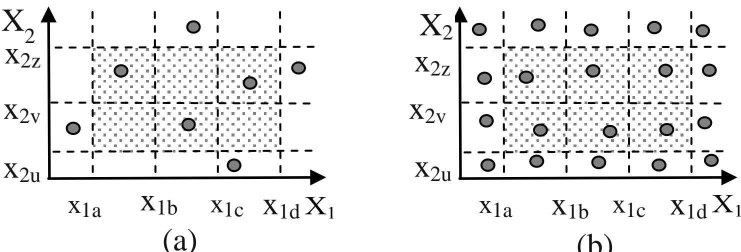
Robustā vājā ekvivalences klašu testēšanas metode ir līdzīga vājai ekvivalences klašu testēšanas metodei, taču tā papildus aplūko nepieļaujamo vērtību ekvivalences klases [BN03], [CBC01], [Cop03], [Cra02], [Jor95], [KBP02], [KFN99], [LF03], [May04], [Per06], [SLS07], [Vee02], [Wat01] saskaņā ar sekojošu algoritmu:

1) pieļaujamajām vērtībām izvēlas tikai vienu vērtību no katras ekvivalences klases, turklāt katrā testpiemērā izmanto tikai pieļaujamas vērtības;

2) nepieļaujamajām vērtībām izvēlas vienu vērtību no katras ekvivalences klases un katrā testpiemērā vienam parametram piemēro vērtību no nepieļaujamajām vērtībām,

bet pārējiem visiem – pieļaujamās vērtības. Metode neļauj izmantot nepieļaujamās vērtības vienā testpiemērā diviem vai vairākiem parametriem (sk. 2.4(a). attēlu).

N parametru gadījumā ģenerēto testpiemēru skaits ir $\max_{i=1}^{N}(M_i)+\sum_{i=1}^{N}Q_i$, kur Q_i nepieļaujamo vērtību ekvivalences klašu kopas apjoms parametram X_i.



2.4. att. **Testpiemēri robustajai vājajai ekvivalences klašu testēšanas metodei (a) un robustajai stiprajai ekvivalences klašu testēšanas metodei (b)**

2.1.3.4. Robustā stiprā ekvivalences klašu testēšanas metode

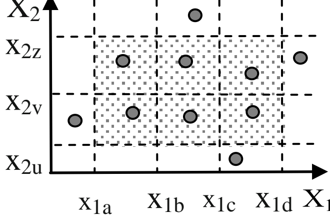
Robustā stiprā ekvivalences klašu testēšanas metode [Dus02], [Jor95], [Wat01] testkomplektā iekļauj testpiemēru no visu parametru visu - gan pieļaujamo, gan nepieļaujamo vērtību ekvivalences klašu Dekarta reizinājuma katra elementa, piemēram, kā tas ir parādīts 2.4(b). attēlā.

N parametru gadījumā ģenerēto testpiemēru skaits ir $\prod_{i=1}^{N}(M_i+Q_i)$.

2.1.3.5. Robustā jauktā ekvivalences klašu testēšanas metode

Robustā jauktā ekvivalences klašu testēšanas metode [SLS07] testkomplektā iekļauj pa testpiemēram no visu parametru pieļaujamo vērtību ekvivalences klašu Dekarta reizinājuma katra. No nepieļaujamo vērtību ekvivalences klasēm ņem pa vienam reprezentantam un veido testpiemērus, kuros vienam parametram ir nepieļaujama vērtība, bet visiem pārējiem – pieļaujamās. Nav pieļauts, ja diviem vai vairāk parametriem vienā testpiemērā ir nepieļaujamas vērtības (sk. 2.5. attēlu).

N parametru gadījumā ģenerēto testpiemēru skaits ir $\prod_{i=1}^{N}M_i+\sum_{i=1}^{N}Q_i$.



2.5.att. **Robustās jauktās testēšanas metodes testpiemēri**

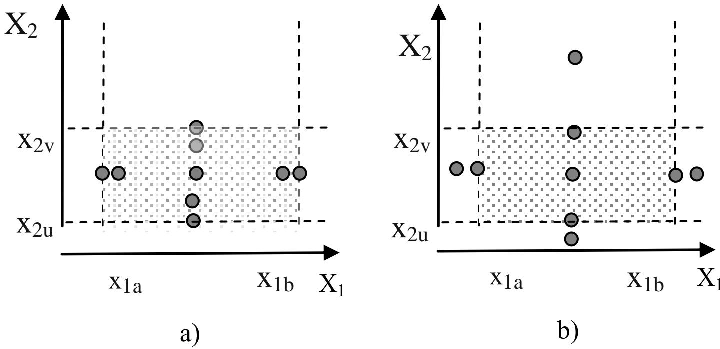
2.1.4. Robežvērtību testēšanas metožu sarežģītība

2.1.4.1. Vājā iekšējā robežvērtību testēšanas metode

Vājā iekšējo robežvērtību testēšanas metode [Jor95], [LF03], [MS01], [SLS07] pārbauda ekvivalences klašu robežvērtības, iekšējos OFF punktus un nominālās vērtības. Metodes pamatā ir vienas kļūdas pieņēmums – katrā testpiemērā vienam parametram ir pārbaudāmā vērtība, bet pārējiem – nominālās, kā tas divdimensionālam gadījumam shematiski parādīts 2.6(a). attēlā.

Metodes sarežģītības movērtējums.

Ja programmai ir tikai viens parametrs X_1, iegūstam 5 testpiemērus no katras pieļaujamo vērtību ekvivalences klases – min, min+, nom, max-, max. Tā kā pieļaujamo vērtību ekvivalences klašu skaits ir M_1, testpiemēru skaits testkomplektā ir $5M_1$.



2.6. att. **Testpiemēri vājai iekšējai robežvērtību testēšanas metodei (a) un vājajai ārējai robežvērtību testēšanas metodei (b)**

Ja programmai ir divi parametri, testpiemērus ģenerē pēc sekojoša algoritma:

1) Pirmajam parametram izvēlas pirmās ekvivalences klases nominālo vērtību, otrajam parametram izvēlas tā domēna pirmās ekvivalences klases min, min+, max-, max punktus, veidojot pirmos 4 testpiemērus.

2) Atkārto pirmo soli visām pārējām parametra X_2 .ekvivalences klasēm.

Tādējādi kopā pa pirmajiem diviem soļiem ir iegūti $4M_2$ testpiemēri.

3) Tā kā blakusesošajā ekvivalences klasēm ir kopīgas robežvērtības, ir izveidoti vienādi testpiemēri. Optimizējam testkomplektu – izslēdzam liekos testpiemērus.

Ir iegūti $4M_2 - L_2$ testpiemēri.

4) Atkārto 1–3 soļus pārējām parametra X_1 ekvivalences klasēm.

Tagad ir iegūti $(4M_2 - L_2)M_1$ testpiemēri.

5) Atkārto 1–4 soļus ar parametriem mainītās lomās – tagad otrajam parametram no ekvivalences klasēm ņemam nominālvērtības, bet no pirmā parametra ekvivalences klasēm – robežvērtības.

Kopā piecos soļos ir iegūti $(4M_1 - L_1)M_2 + (4M_2 - L_2)M_1$ testpiemērus.

6) Tagad pievieno testpiemērus, kuros abiem parametriem ir nominālvērtības, iegūstot šajā solī M_1M_2 testpiemērus.

Kopējais testkomplekta apjoms divu parametru gadījumā ir:

$$(4M_1 - L_1)M_2 + (4M_2 - L_2)M_1 + M_1M_2 = 9M_1M_2 - L_1M_2 - L_2M_1.$$

Ja pielieto vājo iekšējo robežvērtību testēšanas metodi programmai ar N parametriem, iegūst testkomplektu ar $(4N+1)\prod_{i=1}^{N} M_i - \sum_{i=1}^{N}(L_i \prod_{j=1, j\neq i}^{N} M_j)$ testpiemēriem. Šī ir vājās iekšējās robežvērtību testēšanas metodes sarežģītības apakšējā robeža.

Ja algoritmam izlaiž trešo soli – tātad nemeklē liekos testpiemērus, tad iegūst, ka vājā iekšējās robežvērtību testēšanas metode nedos vairāk par $(4N+1)\prod_{i=1}^{N} M_i$ testpiemēriem. Šī ir vājās iekšējās robežvērtību testēšanas metodes sarežģītības augšējā robeža.

2.1.4.2. *Vājā ārējā robežvērtību testēšanas metode*

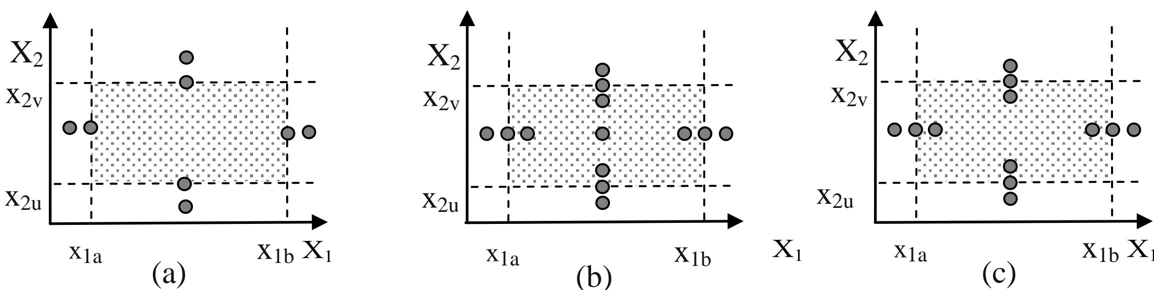
Vājā ārējā robežvērtību testēšanas metode pārbauda ekvivalences klašu robežvērtības, ārējos OFF punktus un nominālos punktus, kā shematiski parādīts 2.6(b). attēlā [BN03], [KFN99]. Šī metode ir ļoti līdzīga vājajai iekšējo robežvērtību testēšanas metodei un atšķiras no tās ir OFF punktu izvēli. Vājā iekšējā RVT metodei izmanto iekšējos OFF punktus, bet vājā ārējā RVT– ārējos OFF punktus.

Ģenerētā testkomplekta apjoms ir tāds pats kā vājajai iekšējo RVT metodei – tas nav mazāks par $(4N+1)\prod_{i=1}^{N} M_i - \sum_{i=1}^{N}(L_i \prod_{j=1, j\neq i}^{N} M_j)$ testpiemēriem un nav lielāks par $(4N+1)\prod_{i=1}^{N} M_i$ testpiemēriem.

2.1.4.3. *Vājā vienkāršā ārējā robežvērtību testēšanas metode*

Vājā vienkāršā ārējā robežvērtību testēšanas metode pārbauda ekvivalences klašu robežvērtības un ārējos OFF punktus [BN03], [EM07], [KFN99], [May04], [MS01], [Vee02]. Atšķirībā no vājās ārējās RVT metodes, šī metode neizmanto nominālpunktus. (sk. 2.7(a). attēlu).

Ģenerētā testkomplekta apjoms ir mazāks nekā vājai iekšējai RVT vai vājajai ārējai RVT metodei – apakšējā sarežģītības robeža ir $4N\prod_{i=1}^{N} M_i - \sum_{i=1}^{N}(L_i \prod_{\substack{j=1\\ j\neq i}}^{N} M_j)$ testpiemēri, bet augšējā - $4N\prod_{i=1}^{N} M_i$ testpiemēri.



2.7. att. **Testpiemēri vājajai vienkāršajai ārējai RVT metodei (a), robustajai vājajai RVT metodei (b) un robustajai vājajai vienkāršajai RVT metodei**

2.1.4.4. Robustā vājā robežvērtību testēšanas metode

Robustā vājā RVT metode [BN03], [Cop03], [Cra02], [Jor95], [LF03], [Vee02], [Wat01] pārbauda ekvivalences klašu robežvērtības, iekšējos un ārējos OFF punktus, kā arī nominālpunktu (sk. 2.7(b). attēlu).

Ģenerētā testkomplekta apjomu var aprēķināt līdzīgi kā vājajai iekšējai RVT metodei – tas būs ne mazāks par $(6N+1)\prod_{i=1}^{N} M_i - \sum_{i=1}^{N}(L_i \prod_{\substack{j=1\\ j\neq i}}^{N} M_j)$ testpiemēriem un ne lielāks par $(6N+1)\prod_{i=1}^{N} M_i$ testpiemēriem.

2.1.4.5. Robustā vājā vienkāršā robežvērtību testēšanas metode

Robustā vājā vienkāršā RVT metode [Cop03], [Vee02] pārbauda ekvivalences klašu robežvērtības, iekšējos un ārējos OFF punktus, bet, atšķirībā no robustās vājās RVT, netestē nominālpunktus (sk. 2.7(c). attēlu).

Ģenerētā testkomplekta apjomu aprēķina līdzīgi kā vājajai iekšējai RVT metodei, iegūstot ne mazāk kā $6N\prod_{i=1}^{N} M_i - \sum_{i=1}^{N}(L_i \prod_{\substack{j=1\\ j\neq i}}^{N} M_j)$ testpiemērus un ne vairāk kā $6N\prod_{i=1}^{N} M_i$ testpiemērus.

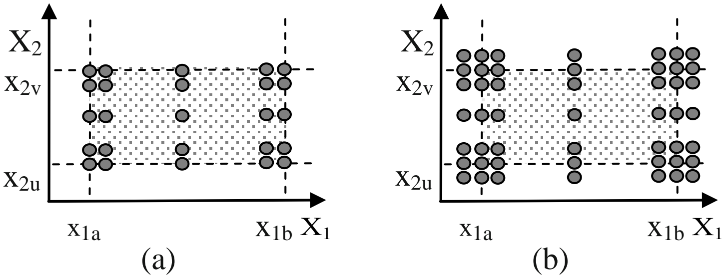
2.1.4.6. Sliktākā gadījuma robežvērtību testēšanas metode

Sliktākā gadījuma robežvērtību testēšanas metode [GTW+03], [Jor95] testē robežvērtības, iekšējos OFF punktus un nominālo punktu. Vājā iekšējā RVT metode izmantoja tikai vienas kļūdas pieņēmumu, taču sliktākā gadījuma RVT metode lieto

vairāku kļūdu pieņēmumu. Tas nozīmē, ka metode pārbauda, kas notiks, ja vienam vai visiem parametriem būs īpašās vērtības (sk. 2.8(a). attēlu).

Ja programmai ir tikai viens parametrs X_1, iegūst 5 testpiemērus no katras ekvivalences klases. Pēc tam izslēdz no testkomplekta liekos testpiemērus, kas radušies gadījumos, kad ekvivalences klasēm ir kopīgas robežvērtības.

Tādējādi viena parametra gadījumā testkomplektā iegūst $5M_1 - L_1$ testpiemērus.



2.8. att. **Testpiemēri sliktākā gadījuma RVT metodei (a) un robustajai sliktākā gadījuma RVT metodei**

Ja programmai ir divi parametri, testpiemērus ģenerē ar sekojošu algoritmu:

1) Pirmajam parametram kā vērtību izvēlas tā pirmās ekvivalences klases minimālo vērtību, bet otrajam parametram kā vērtības maina min, min+, max, nom, max+ vērtības no tā pirmās ekvivalences klases.

2) Atkārto pirmo soli parametra X_1 pirmās ekvivalences klases min+, max, nom, max+ vērtībām.

Pa pirmajiem diviem soļiem kopa iegūti $5x5$ testpiemēri.

3) Atkārto 1. un 2. soli katrai no atlikušajām parametra X_1 ekvivalences klasēm un optimizē testkomplektu, izņemot ārā liekos jeb dubultos testpiemērus.

Tagad ir iegūti $5x(5M_1 - L_1)$ testpiemēri testkomplektā.

4) Atkārto 1-3. soļus katrai no atlikušajām parametra $X_{2\ ekvivalences}$ klasēm.

Pavisam testkomplektā tagad ir $(5M_1 - L_1)x(5M_2 - L_2)$ testpiemēri.

N parametriem iegūst $\prod_{i=1}^{N}(5M_i - L_i)$ testpiemērus, kā metodes sarežģītības apakšējo robežu, bet kā sarežģītības augšējo robežu sliktākā gadījuma RVT metode dod ne vairāk kā $\prod_{i=1}^{N}5M_i = 5^N\prod_{i=1}^{N}M_i$ testpiemērus.

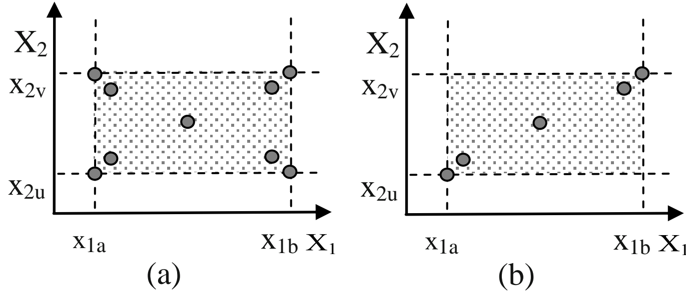
2.1.4.7. Robustā sliktākā gadījuma robežvērtību testēšanas metode

Robustā sliktākā gadījuma robežvērtību testēšanas metode [Dus02], [Jor95] testē robežvērtības, iekšējos un ārējos OFF punktus un nominālvērtību (sk. 2.8(b). attēlu).

Testkomplekta veidošanas algoritms ir analoģisks kā sliktākā gadījuma robežvērtību testēšanas metodei, programmai ar N parametriem iegūstot vismaz $\prod_{i=1}^{N}(7M_i - 2L_i)$ testpiemērus un ne vairāk kā $7^N \prod_{i=1}^{N} M_i$ testpiemērus.

2.1.4.8. *Stūra vājā iekšējā robežvērtību testēšanas metode*

Stūra vājā iekšējā robežvērtību testēšanas metode darbojas saskaņā ar daudzu kļūdu pieņēmumu. Tā testē gadījumus, kad visiem parametriem ir viena veida īpašās vērtības – robežvērtības, iekšējie OFF punkti vai nominālās vērtības, kā tas shematiski parādīts 2.9(a). attēlā [GTW+03], [SLS07].



2.9. att. **Testpiemēri stūra vājajai RVT metodei (a) un vājajai diagonālajai iekšējai RVT metodei (b)**

Ja programmai ir divi parametri, testpiemēri tiek ģenerēti ar sekojošu algoritmu:

1) Ņem abu parametru pirmo ekvivalences klašu robežvērtības un izveido 4 testpiemērus - (min, min), (min, max), (max, min) un (max, max).

2) Ņem abu parametru pirmo ekvivalences klašu iekšējos OFF punktus un izveido atkal izveido 4 testpiemērus - (min+, min+), (min+, max-), (max-, min+) un (max-, max-).

3) Abu parametru pirmo ekvivalences klašu nominālpunkti dod vienu testpiemēru (nom, nom).

Kopā līdz šim ir iegūti $2x2^2 + 1$ testpiemēri pirmajam abu parametru ekvivalences klašu kopu Dekarta reizinājuma elementam. Pa visiem elementiem kopā iegūst $(2x2^2 + 1)M_1M_2$ testpiemērus.

4) Tagad izslēdz liekos jeb dubultos testpiemērus, ko rada ekvivalences klašu kopējās robežvērtības un iegūst $(2^{2+1} + 1)\prod_{i=1}^{2} M_i - 2\sum_{i=1}^{2}(L_i \prod_{\substack{j=1 \\ j \neq i}}^{2} M_i)$ testpiemērus.

N parametru gadījumā stūra vājā iekšējā RVT metode testkomplektā dod vismaz $(2^{N+1}+1)\prod_{i=1}^{N} M_i - 2\sum_{i=1}^{N}(L_i \prod_{\substack{j=1 \\ j \neq i}}^{N} M_j)$ testpiemērus.

Ja neveic 4. algoritma soli, tad iegūst, ka stūra vājā iekšējā RVT metode testkomplektā dod ne vairāk kā $(2^{N+1}+1)\prod_{i=1}^{N} M_i$ testpiemērus.

2.1.4.9. Vājā diagonālā iekšējā robežvērtību testēšanas metode

Vājā diagonālā iekšējā RVT metode arī atbilst daudzu kļūdu pieņēmumam, taču tā testē gadījumus, kad visiem programmas parametriem ir viena tipa speciālās vērtības, piemēram, ja tās ir robežvērtības, tad visiem parametriem ir minimālās vērtības vai visiem parametriem ir maksimālās vērtības, līdzīgi arī ar iekšējiem OFF punktiem un nominālvērtībām, kā shematiski parādīts 2.9(b). attēlā [GTW+03].

Ja programmai ir divi parametri, testkomplekts tiek veidots ar sekojošu algoritmu:

1) Ņem robežvērtības pirmajai ekvivalences klasei katram no parametriem un iegūst 2 testpiemērus (min, min), (max, max).

2) Ņem iekšējos OFF punktus pirmajai ekvivalences klasei katram no parametriem un iegūst 2 testpiemērus (min+, min+) un (max-, max-).

3) Nominālpunkti pirmajai ekvivalences klasei katram no parametriem dod vēl vienu testpiemēru (nom, nom).

Kopā iegūti 5 testpiemēri pirmajam abu parametru ekvivalences klašu kopu Dekarta reizinājuma elementam. Pa visiem elementiem kopā iegūst $5M_1M_2$ testpiemērus.

4) Tagad izslēdz liekos jeb dubultos testpiemērus, ko rada ekvivalences klašu kopējās robežvērtības un iegūst $5M_1M_2 - L_1L_2$ testpiemērus.

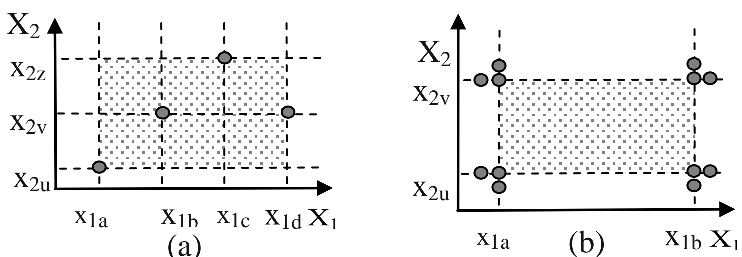
N parametru gadījumā vājā diagonālā iekšējā RVT metode testkomplektā dod vismaz $5\prod_{i=1}^{N} M_i - \prod_{i=1}^{N} L_i$ testpiemērus.

Ja neveic 4. algoritma soli, tad iegūst, ka vājā diagonālā iekšējā RVT metode testkomplektā dod ne vairāk kā $5\prod_{i=1}^{N} M_i$ testpiemērus.

2.1.4.10. Daudzdimensiju robežvērtību testēšanas metode

Daudzdimensiju RVT metode prasa vismaz vienu testpiemēru katras ekvivalences klases katrai robežvērtībai (2.10(a). attēls) [CBC01], [Jor95], [KBP02], [KFN99], [MS01], [NJH01], [SLS07], [Vee02].

N parametru gadījuma testkomplektu veidos $\max_{i=1}^{N}(2M_i - L_i)$ testpiemēri jeb ne vairāk kā $\max_{i=1}^{N}(2M_i)$ testpiemēri, ja nav pārklājušos robežvērtību.



2.10. att. **Testpiemēri daudzdimensiju RVT metodei (a) robustajai stūra ārējai RVT metodei (b)**

2.1.4.11. Robustā stūra ārējā robežvērtību testēšanas metode

Robustā stūra ārējā RVT metode testē gadījumus, kad programmas parametriem ir robežvērtības vai ārējie OFF punkti kā shematiski parādīts 2.10(b). attēlā [Cop03].

Vispirms aplūko gadījumu, kad vienam no parametriem vērtība ir OFF punkts.

Ja programmai ir tikai viens parametrs X_1,tad iegūst 4 testpiemērus no katras pieļaujamo vērtību ekvivalences klases – min-, min, max un max+. Tā kā ekvivalences klašu skaits ir M_1, pārklājošos robežvērtību skaits ir L_1, testpiemēru skaits testkomplektā būs $4M_1$-L_1

Ja programmai ir divi parametri, testkomplekts tiek veidots ar sekojošu algoritmu:

Vispirms testpiemērus veido ar ārējiem OFF punktiem.

1) Kombinējot katru otrā parametra robežvērtību ar katru pirmā parametra ārējo OFF punktu, iegūst $2M_1(2M_2 - L_2)$ testpiemērus.

2) Atkārto pirmo soli, mainot parametrus vietām, un iegūst $2M_2(2M_1 - L_1)$ testpiemērus.

3) Veido testpiemērus, kad abiem parametriem ir robežvērtības un iegūst - $(2M_1 - L_1)(2M_2 - L_2)$ testpiemērus.

Kopā divdimensiju gadījumā iegūst $2M_1(2M_2 - L_2) + 2M_2(2M_1 - L_1) + (2M_1 - L_1)(2M_2 - L_2)$ testpiemērus.

Vispārinot uz N parametriem, iegūst, ka robustā stūra ārējā RVT metode ģenerē vismaz $\sum_{i=1}^{N}(2M_i \prod_{\substack{j=1 \\ j \neq i}}^{N}(2M_j - L_j)) + \prod_{i=1}^{N}(2M_i - L_i)$ testpiemērus.

Metodes sarežģītības augšējā robeža ir $\prod_{i=1}^{N} 2M_i + \sum_{i=1}^{N}(2M_i \prod_{\substack{j=1 \\ j \neq i}}^{N} 2M_j) = 2^N \prod_{i=1}^{N} M_i + \sum_{i=1}^{N}(2^N \prod_{j=1}^{N} M_j) = 2^N (N+1) \prod_{i=1}^{N} M_i$ testpiemēri testkomplektā.

2.1.4.12. Robustā stiprā robežvērtību testēšana

Robustā stiprā RVT metode [SLS07] izmanto pieļaujamo vērtību ekvivalences klašu visas robežvērtības, iekšējos un ārējos OFF punktus un sadala tās divās kopās – visas pieļaujamās vērtības vienā kopā un visas nepieļaujamās vērtības - otrā (sk. 2.11. (a) un (b) attēlus).

Metode ļauj brīvi kombinēt testpiemērā visas pieļaujamās vērtības.

Nepieļaujamo vērtību kopa tiek reducēta. Ja ir divas vai vairāk vērtības no vienas un tās pašas ekvivalences klases, tad no tām kopā tiek atstāta tikai viena. Metode prasa tieši vienu testpiemēru katrai vērtībai, kas ir atstāta šajā kopā, turklāt katrā testpiemērā pieļauj tikai vienu nepieļaujamo vērtību, pārējo parametru vērtībām jābūt no pieļaujamo vērtību kopas.

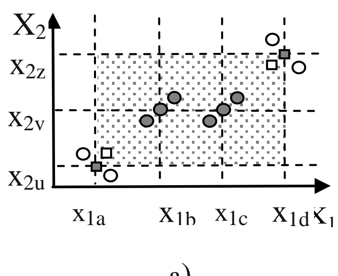
Katram parametram ir L_i kopīgās jeb pārklājošās robežvērtības. Tās dod pieļaujamo vērtību kopā $3L_i$ vērtības – robežvērtību un tās abus OFF punktus. Katram parametram ir $2M_i$-$2L_i$ nepārklājošās robežvērtības, tātad būs tieši $2M_i$-$2L_i$ nepieļaujamās vērtības – ekvivalences klašu robežvērtību ārējie OFF punkti, kas atrodas nepieļaujamo vērtību apgabalos, un būs tieši $2M_i$-$2L_i$ pieļaujamās vērtības – pieļaujamo vērību ekvivalences klašu robežvērtību iekšējie OFF punkti.

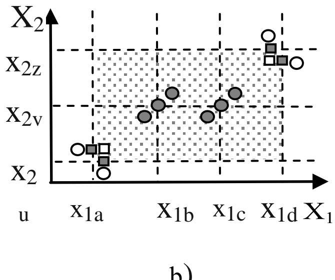
Tādējādi, tā kā metodes adekvātuma kritērijs ļauj brīvi kombinēt pieļaujamās vērtības, iegūst, ka mazākais iespējamais testpiemēru, kurus veido tikai pieļaujamās vērtības, skaits testkomplektā programmai ar N parametriem ir $\max_{i=1}^{N}(3L_i + 2M_i - 2L_i) = \max_{i=1}^{N}(2M_i + L_i)$, bet nepieļaujamo vērtību kopa dod $\sum_{i=1}^{N}(2M_i - 2L_i)$ testpiemērus.

Vienīgais jautājums ir par $2M_i$-$2L_i$ robežvērtībām. Tās var būt pieļaujamas (piederēt kādai ekvivalences klasei), bet var būt arī nepieļaujamas.

Apakšējo robustās stiprās RVT metodes sarežģītības robežu iegūst, ja pieņem, ka visas robežvērtības ir pieļaujamas:

$$\max_{i=1}^{N}(2M_i + L_i) + \sum_{i=1}^{N}(2M_i - 2L_i) + \max_{i=1}^{N}(2M_i - 2L_i) =$$
$$= \max_{i=1}^{N}(2M_i + L_i) + 2\sum_{i=1}^{N}(M_i - L_i) + 2\max_{i=1}^{N}(M_i - L_i)$$





a) b)

2.11. att. **Testpiemēri robustajai stiprajai robežvērtību testēšanai: (a) apakšējās sarežģītības robežas gadījums, (b) – augšējās. Melnie punkti reprezentē testpiemērus pārklājošos robežu gadījumā, baltie kvadrāti – nepārklājošos robežu iekšējo OFF punktu testpiemērus. Baltie punkti reprezentē pārklājošos robežu ārējos OFF punktus, bet melnie kvadrāti rāda nepārklājošos robežvērtību testpiemērus, kad (a) gadījumā tās ir pieļaujamas vērtības, (b) gadījumā – kad tās ir nepieļaujamās vērtības**

Augšējo robustās stiprās RVT metodes sarežģītības robežu iegūst, ja pieņem, ka visas robežvērtības ir nepieļaujamas:

$$\max_{i=1}^{N}(2M_i + L_i) + \sum_{i=1}^{N}(2M_i - 2L_i) + \sum_{i=1}^{N}(2M_i - 2L_i) = \max_{i=1}^{N}(2M_i + L_i) + 4\sum_{i=1}^{N}(M_i - L_i)\ .$$

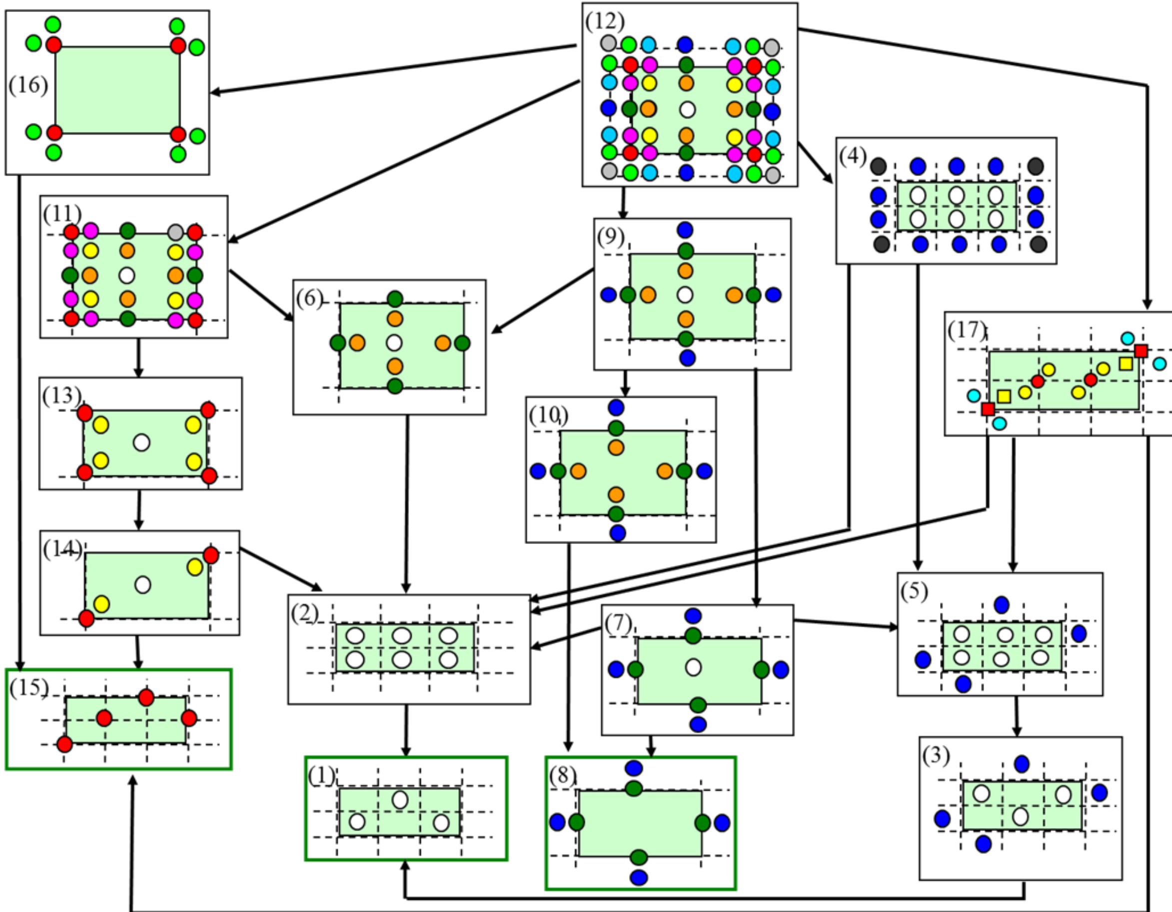
2.2. Domēntestēšanas metožu iekļautība

Avotos bieži testēšanas metožu salīdzināšanā izmanto to sakārtošanu pēc iekļautības principa. Šis princips ir ticis, piemēram, pielietots datu plūsmas testēšanas metožu adekvātuma kritēriju salīdzināšanā, kā arī dažādu citu strukturālu pārklājuma metožu kritēriju salīdzināšanā [CPR+85], [FW88], [Nta88], [RW85], [Wei89], [WGM85]. Citi testēšanas kritēriju salīdzināšanas veidi ir tikuši pētīti, salīdzināti *Weyuker*, *Weiss* un *Hamlet* darbos [Ham89], [WWH91].

Iekļautības viena no definīcijām ir sekojoša: "*pieņemsim, ka C1 un C2 ir divi programmatūras testēšanas metožu adekvātuma kritēriji. Saka, ka C1 iekļauj C2 jebkurai testējamai programmai p, visām specifikācijām s un visām testu kopām t tad, ja no tā, ka testu kopa t ir adekvāta pret C1 programmai p attiecībā pret specifikāciju s, izriet, ka t ir adekvāta arī pret C2 programmai p attiecībā pret specifikāciju s*" [Ham89]. Citiem vārdiem, C1 iekļauj C2, ja katrs testkomplekts, ko ģenerē C1, ir adekvāts arī pret C2.

2.12. attēlā ir parādīta domēntestēšanas metožu hierarhija, saskaņā ar iekļautības principu. Katras metodes pamatprincips testpiemēru izvēlē attēlā shematiski parādīts

atbilstošajā taisnstūrī. Bultas virziens rāda iekļautību, piemēram, robustā sliktākā gadījuma RVT metode iekļauj jebkuru no pārējām metodēm.



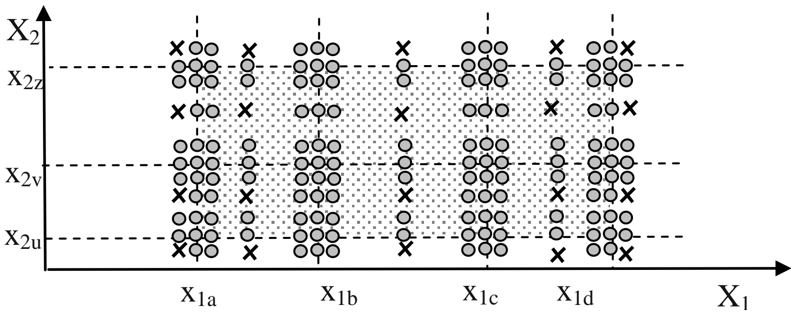
2.12. att. **Domēntestēšanas metožu hierarhija saskaņā ar iekļautības principu, , kur: (1) - Vājā EKT metode, (2) - Stiprā EKT metode, (3) - Robustā vājā EKT metode, (4) - Robustā stiprā EKT metode, (5) - Robustā jauktā EKT metode, (6) - Vājā iekšējo RVT metode, (7) - Vājā ārējo RVT metode, (8) - Vājā vienkāršā ārējo RVT metode, (9) - Robustā vājā RVT metode, (10) - Robustā vājā vienkāršā RVT metode, (11) - Sliktākā gadījuma RVT metode, (12) - Robustā sliktākā gadījuma RVT metode, (13) - Stūra vājā iekšējā RVT metode, (14) - Vājā diagonālā iekšējā RVT metode, (15) - Daudzdimensiju RVT metode, (16) - Robustā stūra ārējā RVT metode, (17) - Robustā stiprā RVT metode**

Vairumā gadījumu iekļautība ir acīmredzama, taču dažas situācijas ir vērts aplūkot tuvāk.

2.2.1.1. Robustā sliktākā gadījuma RVT metode iekļauj robusto stipro EKT metodi

Robustā stiprā EKT metode prasa testpiemēru no katra elementa Dekarta reizinājumam, ko veido visu parametru pieļaujamo un nepieļaujamo vērtību ekvivalences klases. Robustā sliktākā gadījuma RVT metode izmanto iekšējos OFF

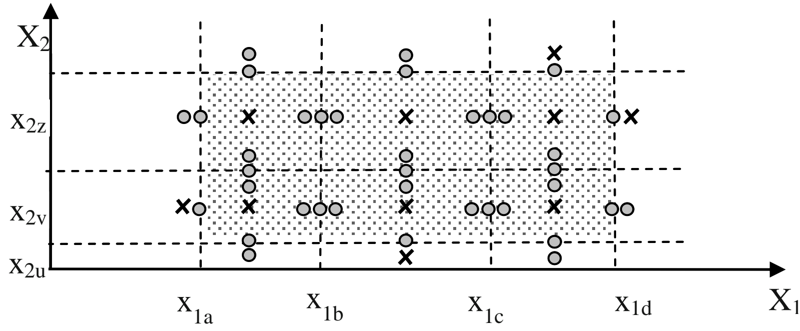
punktus un nominālpunktu no visu elementu katras pieļaujamo vērtību ekvivalences klases (robežvērtības nav derīgas, jo ir iespējams, ka tās nepieder pieļaujamo vērtību ekvivalences klasei), kā arī ārējos OFF punktus no visu parametru nepieļaujamo vērtību ekvivalences klašu Dekarta reizinājuma elementiem. Divdimensionālajā gadījumā situāciju shematiski var parādīt kā 2.13. attēlā. Krustiņi un punktiņi kopā veido testkomplektu robustajai sliktākā gadījumā RVT metodei, bet krustiņi vieni paši attēlo testkomplektu, kurš apmierina robustās stingrās EKT metodes kritēriju. Šādā veidā no robustās sliktākā gadījuma RVT metodes testkomplekta vienmēr var iegūt testkomplektu, kurā atbilst robustajai stingrajai EKT metodei.



2.13. att. **Krustiņi un punktiņi kopā veido testkomplektu robustajai sliktākā gadījumā RVT metodei, bet krustiņi vieni paši attēlo testkomplektu, kurš apmierina robustās stingrās EKT metodes kritēriju**

2.2.1.2. Vājā ārējā RVT metode iekļauj robusto jaukto EKT metodi

Robustās jauktās EKT metodes kritērijs prasa testpiemēru no katra elementa Dekarta reizinājumam, ko veido visu parametru pieļaujamo vērtību ekvivalences klašu kopas, un vienu elementu no visu parametru katras nepieļaujamo vērtību ekvivalences klases. Vājā ārējā RVT metode izmanto nominālpunktu no katra elementa Dekarta reizinājumam, ko veido visu parametru pieļaujamo vērtību ekvivalences klašu kopas ārējos OFF punktus no piegulošajām nepieļaujamo vērtību ekvivalences klasēm.

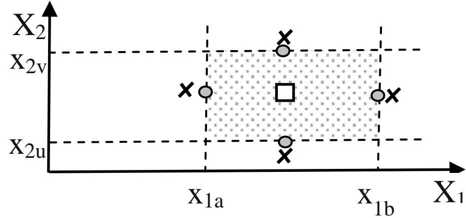


2.14. att. **Krustiņi un punkti kopā veido testkomplektu vājajai ārējai RVT metodei, bet krustiņi vieni paši veido adekvātu testkomplektu robustajai jauktajai EKT metodei**

Divdimensionālā gadījumā shematiski situāciju var parādīt kā 2.14. attēlā. Krustiņi un punkti kopā veido testkomplektu vājajai ārējai RVT metodei, bet krustiņi vieni paši veido adekvātu testkomplektu robustajai jauktajai EKT metodei. Šādā veidā no katra vājās ārējās RVT metodes ģenerētā testkomplekta var iegūt testkomplektu, kurš ir adekvāts robustās jauktās EKT metodes kritērijam.

2.2.1.3. Vājā vienkāršā ārējā RVT metode neiekļauj robusto vājo EKT metodi

Situācijā, kas parādīta 2.15. attēlā, parametram X_1 ir viena pieļaujamo vērtību ekvivalences klase, kuras robežvērtības tai nepieder – intervāls (x_{1a}, x_{1b}) un parametram X_2 ir līdzīga situācija – ekvivalences klase ir intervāls (x_{2u}, x_{2v}). Vājā vienkāršā ārējā RVT metode nenodrošina testpiemēru, kad abiem parametriem ir pieļaujamas vērtības, taču tādu testpiemēru prasa robustā vājā EKT metode.



2.15. att. **Krustiņi un punkti kopā veido testkomplektu vājajai vienkāršajai ārējai RVT metodei. Krustiņi vieni paši der kā testpiemēri robustās vājās EKT metodes testkomplektam. Bet kvadrātiņš parāda testpiemēru, kuru prasa robustās vājās EKT metodes kritērijs, bet nenodrošina vājā vienkāršā ārējā RVT metode**

2.1. tabulā apkopotas apskatīto ekvivalences klašu testēšanas un robežvērtību testēšanas metožu sarežģītības apakšējo un augšējo robežu novērtējumi. Viegli redzēt, ka robustā sliktākā gadījuma RVT metodei, kura ir iekļautības hierarhijas koka saknē 2.12. attēlā, ir arī visaugstākie sarežģītības novērtējumi. Metodēm, kuras atrodas iekļautības hierarhijas zemāko līmeņos, ir arī zemāki sarežģītības novērtējumi.

Ir acīmredzami, ka, ja metodes C1 adekvātuma kritērijs iekļauj metodes C2 adekvātuma kritēriju, tad C1 sarežģītība ir lielāka par vai vienāda ar C2 sarežģītību.

Tomēr nevar apgalvot, ka, ja metodes C1 sarežģītība ir lielāka par metodes C2 sarežģītību, tad C1 iekļauj C2. Vājās vienkāršās ārējās RVT metodes sarežģītība ir lielāka par vājās EKT metodes sarežģītību. Taču vājā vienkāršā ārējā RVT metode testē pavisam cita tipa punktus nekā vājā EKT metode, tādējādi vājā vienkāršā ārējā RVT metode neiekļauj vājo EKT metodi.

2.1. tabula

Darbā apskatīto domēntestēšanas metožu sarežģītības novērtējumu apkopojums, kur N – programmas parametru skaits, M_i – parametra X_i pieļaujamo vērtību ekvivalences klašu skaits, Q_i – parametra X_i nepieļaujamo vērtību ekvivalences klašu skaits, un L_i – parametra X_i pieļaujamo vērtību ekvivalences klašu kopīgo robežvērtību skaits

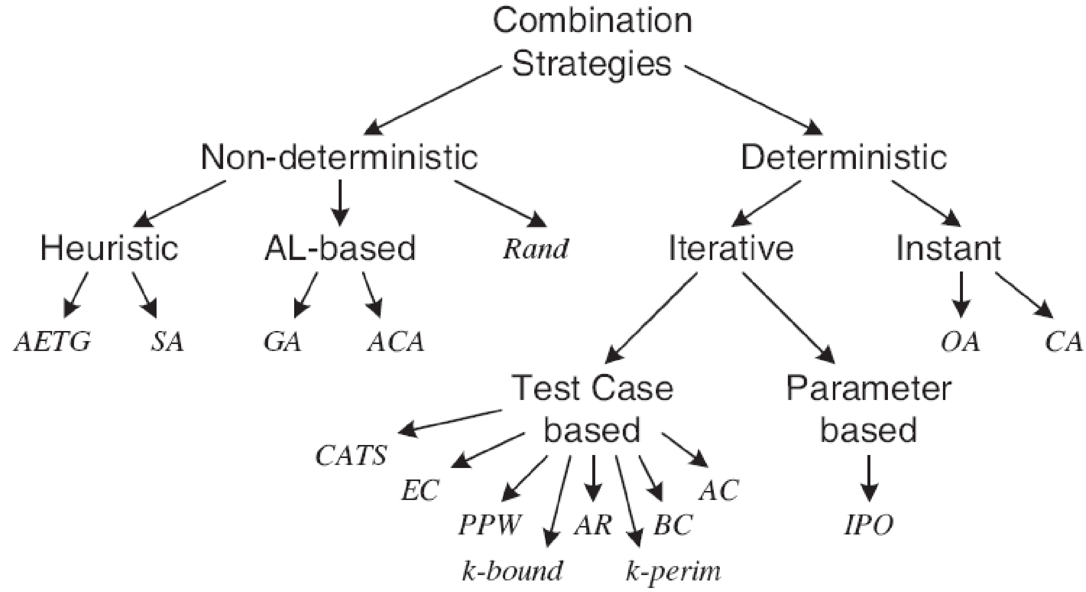
Npk.	Metode	Sarežģītība	
		Apakšējā robeža	Augšējā robeža
1.	Vājā EKT metode	$\max_{i=1}^{N}(M_i)$	Sakrīt ar apakšējo robežu
2.	Stiprā EKT metode	$\prod_{i=1}^{N} M_i$	Sakrīt ar apakšējo robežu
3.	Robustā vājā EKT metode	$\max_{i=1}^{N}(M_i) + \sum_{i=1}^{N} Q_i$	Sakrīt ar apakšējo robežu
4.	Robustā stiprā EKT metode	$\prod_{i=1}^{N}(M_i + Q_i)$	Sakrīt ar apakšējo robežu
5.	Robustā jauktā EKT metode	$\prod_{i=1}^{N} M_i + \sum_{i=1}^{N} Q_i$	Sakrīt ar apakšējo robežu
6.	Vājā iekšējā RVT metode	$(4N+1)\prod_{i=1}^{N} M_i - \sum_{i=1}^{N}(L_i \prod_{j=1, j\neq i}^{N} M_j)$	$(4N+1)\prod_{i=1}^{N} M_i$
7.	Vājā ārējā RVT metode	$(4N+1)\prod_{i=1}^{N} M_i - \sum_{i=1}^{N}(L_i \prod_{j=1, j\neq i}^{N} M_j)$	$(4N+1)\prod_{i=1}^{N} M_i$
8.	Vājā vienkāršā ārējā RVT metode	$4N\prod_{i=1}^{N} M_i - \sum_{i=1}^{N}(L_i \prod_{j=1, j\neq i}^{N} M_j)$	$4N\prod_{i=1}^{N} M_i$
9.	Robustā vājā RVT metode	$(6N+1)\prod_{i=1}^{N} M_i - 2\sum_{i=1}^{N}(L_i \prod_{j=1, j\neq i}^{N} M_j)$	$(6N+1)\prod_{i=1}^{N} M_i$
10.	Robustā vājā vienkāršā	$6N\prod_{i=1}^{N} M_i - 2\sum_{i=1}^{N}(L_i \prod_{j=1, j\neq i}^{N} M_j)$	$6N\prod_{i=1}^{N} M_i$
11.	Sliktākā gadījuma RVT metode	$\prod_{i=1}^{N}(5M_i - L_i)$	$5^N \prod_{i=1}^{N} M_i$
12.	Robustā sliktākā gadījuma RVT metode	$\prod_{i=1}^{N}(7M_i - 2L_i)$	$7^N \prod_{i=1}^{N} M_i$
13.	Vājā stūra iekšējā RVT metode	$(2^{N+1}+1)\prod_{i=1}^{N} M_i - 2\sum_{i=1}^{N}(L_i \prod_{j=1, j\neq i}^{N} M_i)$	$(2^{N+1}+1)\prod_{i=1}^{N} M_i$
14.	Vājā diagonālā iekšējā RVT metode	$5\prod_{i=1}^{N} M_i - \prod_{i=1}^{N} L_i$	$5\prod_{i=1}^{N} M_i$
15.	Daudzdimensionālā RVT metode	$\max_{i=1}^{N}(2M_i - L_i)$	$2\max_{i=1}^{N}(M_i)$
16.	Robustā stūra ārējā RVT metode	$\sum_{i=1}^{N}(2M_i \prod_{j=1, j\neq i}^{N}(2M_j - L_j)) + \prod_{i=1}^{N}(2M_i - L_i)$	$2^N(N+1)\prod_{i=1}^{N} M_i$

Npk.	Metode	Sarežģītība	
		Apakšējā robeža	Augšējā robeža
17.	Robustā stiprā RVT metode	$\max\limits_{i=1}^{N}(2M_i + L_i) + 2\sum\limits_{i=1}^{N}(M_i - L_i) + 2\max\limits_{i=1}^{N}(M_i - L_i)$	$\max\limits_{i=1}^{N}(2M_i + L_i) + 4\sum\limits_{i=1}^{N}(M_i - L_i)$

2.3. Kombinācijtestēšanas metožu sarežģītība

Kombinācijtestēšanas metodes veido testpiemērus, kombinējot programmatūras parametru vērtības saskaņā ar kādu kombinēšanas stratēģiju, veidojot vērtību pārklājumus.

2.16. attēlā parādīts kombinācijtestēšanas metožu sadalījums pēc kombinēšanas stratēģiju tipiem [GOA05].



2.16.att. **Kombinācijtestēšanas metožu sadalījums pēc kombinēšanas stratēģiju tipiem [GOA05]**

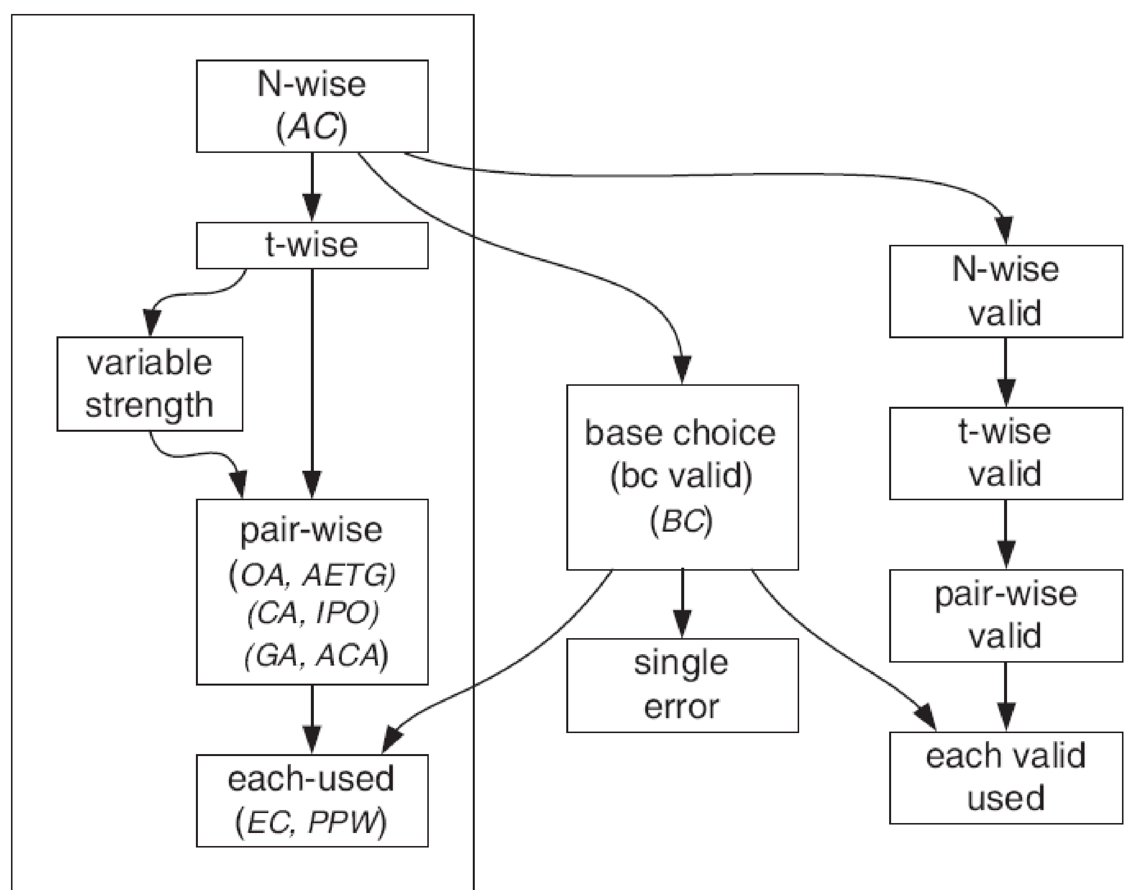
Kombinācijtestēšanas metožu pārklājumu kritēriji:

- *Katrs-lietots* (*1-pakāpes*, *each-used*, *1-wise*) pārklājums prasa, ka katra parametra katra izvēlētā vērtība ģenerētajā testkomplektā tiek izmantota vismaz vienu reizi [GOA05]. Piemēram, 100% pārklājums atbilstoši kritērijam *Katrs-pieļaujamais-lietots* (*Each-valid-used*) nozīmē, ka katra parametra katra pieļaujamā vērtība tiek iekļauta vismaz vienā testpiemērā, kurā pārējo parametru vērtības arī ir pieļaujamas.

- *Pāru* (*2-pakāpes, pair-wise, 2-wise*) pārklājums prasa, ka testkomplekta testpiemēros parādās katrs iespējamais pāris, ko var izveidot no dažādu parametru vērtībām.
- *t-pakāpes* (*t-wise*) pārklājums nosaka, ka testkomplekta testpiemēros jābūt jebkurai iespējamai kombinācijai no t parametru vērtībām. *N-pakāpes* pārklājums ir specgadījums no t-pakāpes pārklājuma, kur N ir programmas parametru skaits.

Pieaugot pārklājuma līmenim *t*, pieaug arī testkomplekta apjoms. *Katrs-lietots* un *N-pakāpes* pārklājumi tiek plaši izmantoti domēntestēšanas metodēs.

2.17. attēlā parādīta kombinācijtestēšanas metožu iekļautība, balstoties uz izmantotajiem pārklājuma kritērijiem.



2.17. att. **Kombinācijtestēšanas pārklājuma kritēriju un atbilstošo metožu iekļautība, pamatojoties uz kombinēšanas stratēģiju pārklājumu kritērijiem [GOA05]**

Iepriekšējā apakšnodaļā jau tika apskatīta iekļautības definīcija, kas nozīmē, ka C1 un C2 ir divi programmatūras testēšanas metožu adekvātuma kritēriji, tad C1 iekļauj C2, ja katrs testkomplekts, ko ģenerē C1, ir adekvāts arī pret C2 [Ham89]. Faktiski tas nozīmē arī, ka, C1 iekļauj C2 tad un tikai tad, ja no fakta, ka C1 sasniedzis 100% pārklājumu, izriet, ka ir sasniegts arī C2 100% pārklājums.

Tātad, izmantojot 2.17. attēlu, kurā ir parādīta kombinācijtestēšanas metožu iekļautības hierarhija, var teikt, ka vismazākā sarežģītība būs metodēm, kuras nodrošina pārklājuma kritērijus *Katrs-lietots (Each-used,* metodes *EC* un *EPW)*, *Viena-kļūda (Single error)* un *Katrs-pieļaujamais-lietots* (*Each-valid-used*), bet vislielākā – kritērijam *N-pakāpes* pārklājums (metode *AC*).

100% pārklājums atbilstoši kritērijam *Katrs-pieļaujamais-lietots* (*Each-valid-used*) nozīmē, ka katra parametra katra interesējošā pieļaujamā vērtība tiek iekļauta vismaz vienā testpiemērā, kurā pārējo parametru vērtības arī ir pieļaujamas [GOA05].

Pārklājums atbilstoši kritērijam *Viena-kļūda* (*Single_error*) nozīmē, ka katra parametra katra interesējošā kļūdainā vērtība jeb vērtība, kas izraisa programmas darba pārtraukšanu vai paziņojumu par kļūdu, tiek iekļauta vismaz vienā testpiemērā, kurā pārējo parametru vērtības ir pieļaujamas [GOA05]. Šis kritērijs realizē *vienas kļūdas pieņēmumu.*

Pārklājums atbilstoši kritērijam *Katrs-lietots* (*Each-used*) nozīmē, ka katra parametra katra interesējoša pieļaujamā vērtība tiek iekļauta vismaz vienā testpiemērā, kurā pārējo parametru vērtības arī ir pieļaujamas [GOA05].

Pārklājuma kritēriju *Katrs-lietots* un *Katrs-pieļaujamais_lietots* sarežģītība ir vienāda ar maksimālo atlasīto vērtību skaitu pa visiem parametriem. Kritērija *Viena-kļūda* sarežģītība ir vienāda ar visu parametru visu atlasīto vērtību skaita summu. Tātad šo kritēriju sarežģītības novērtējums vistiešākajā veidā ir atkarīgs no tā, kā interpretē jēdzienus „*interesējošās pieļaujamās vērtības*” un „*interesējošās kļūdainās vērtības*”. Triviālākajā gadījumā pietiek atlasīt vienu pieļaujamo un vienu nepieļaujamo vērtību katram parametram un tās kombinēt atbilstoši pārklājuma kritērijiem. Šāda stratēģija varētu būt noderīga dūmu testiem, lai noskaidrotu, vai programmatūra vispār strādā, bet ne testiem programmatūrā esošo kļūdu atrašanai. Ja vēlas testēšanu veikt dziļāk, vērtību grupēšanai izvēlei nākas izmantot domēntestēšanas metodēs pielietotos vērtību atlases kritērijus.

2.4. Modeļbāzēto testēšanas metožu sarežģītība

Modelis ir formāla abstrakcija, kas kalpo konkrētam mērķim [Bae10]. Modeli uzskata par formālu, ja tā struktūra un semantika atbilst tieši vai netieši noteiktam metamodelim.

Modeļbāzēta testēšana lieto testējamas sistēmas un dažkārt arī tās vides formālus modeļus, lai ģenerētu testpiemērus un novērtētu, cik veiksmīgi testpiemēri ir izpildījušies.

Var teikt, ka testēšana vienmēr ir modeļbāzēta, jo tā ir balstīta uz kļūdu modeļa. Kļūdu modelis skatās uz sistēmu ar mērķi ieraudzīt, kur visticamāk varētu būt kļūdas. *Binder* kļūdu modeli definē sekojoši [Bin99]: ”*Kļūdu modelis identificē testējamās sistēmas komponentes un saites starp tām, kurās, visticamāk, ir kļūdas*”. 2.18. attēlā ir redzama modeļbāzētās testēšanas taksonomija [UPL05].

Modeļbāzētās testēšanas pamatpieejas [UL07]:

1. Testpiemēru ģenerēšana no sistēmas domēna modeļa.
2. Testpiemēru ģenerēšana no vides modeļa
3. Testpiemēru un to orākulu ģenerēšana no uzvedības modeļa.
4. Testpiemēru skriptu ģenerēšana no abstraktajiem testiem.

Domēna modelis satur informāciju par ievadvērtību domēniem un testpiemēru ģenerēšana nozīmē gudri atlasīt apakškopu no šīm vērtībām, savstarpēji tās kombinēt, veidojot testpiemērus.

Vides modelis apraksta testējamās sistēmas sagaidāmo vidi. Piemēram, tas var būt sistēmas lietošanas statistiskais modelis [Pro03], kurš var saturēt tādu informāciju kā – cik bieži kuras daļas tiek izpildītas, kādi ir ievadvērtību lietojuma sadalījumi.

Domēna un vides modeļi nesatur nekādu informāciju par sagaidāmajiem testpiemēru rezultātiem jeb orākuliem. Šādu informāciju var iegūt no uzvedības modeļa, kas apraksta arī saistību starp sistēmas ievaddatiem un izvadu. Šis ir vienīgais modeļu veids no 4 iepriekšminētajiem, kas ļauj izveidot testpiemēru ar tā orākulu.

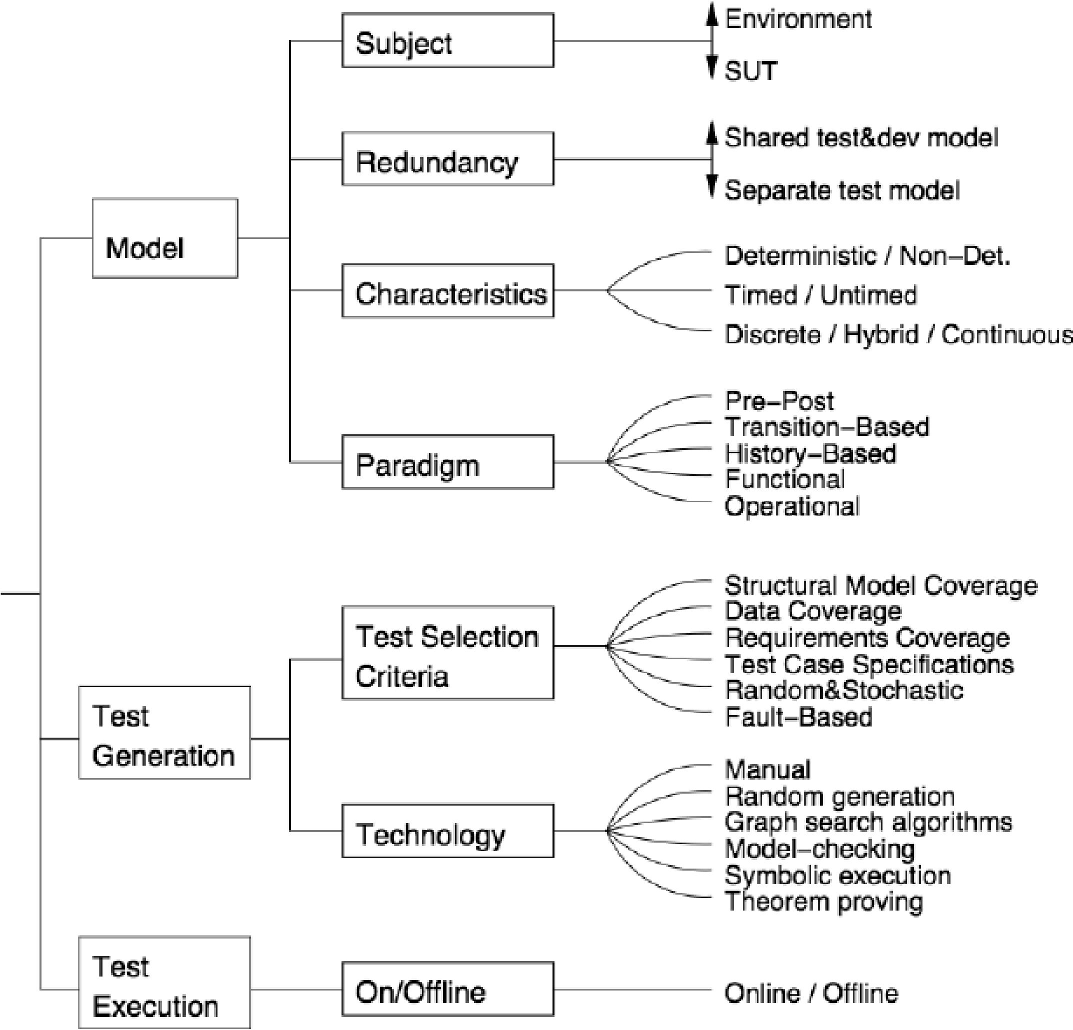
Abstraktie testi apraksta testpiemērus augstā līmenī, piemēram, ar UML secības diagrammu vai kā augsta līmeņa procedūru izsaukumus. No šā abstraktā apraksta tiek ģenerēti izpildāmi zema līmeņa testpiemēru skripti. Abstraktie testi satur informāciju par testējamās sistēmas API.

Modeļbāzētajā testēšanā var izdalīt sekojošus etapus:

1. Modeļu veidošana.
2. Testpiemēru ģenerēšana.
3. Orākulu atrašana testpiemēriem.
4. Modeļu un testkomplektu uzturēšana.

Modeļu pierakstam var izmantot galīgos automātus, stāvokļu diagrammas, klašu diagrammas, secības diagrammas, lietošanas scenāriju diagrammas, aktivitāšu diagrammas, sadarbības diagrammas, objektu diagrammas [NSV+07], Markova ķēdes,

gramatikas un citus veidus, piemēram, lēmumu tabulas, lēmumu kokus. Modeļu veidošana notiek manuāli, izmantojot testējamās sistēmas specifikāciju vai, ja specifikācija ir nepilnīga vai nav vispār, piefiksējot reālo sistēmas uzvedību un konsultējoties ar citām sistēmas izstrādē piesaistītajām pusēm par vēlamo sistēmas uzvedību.



2.18. att. **Modeļbāzētās testēšanas taksonomija [UPL05]**

Testkomplekta, ko veido, izmantojot modeļbāzēto testēšanu, apjoms ir atkarīgs no pārklājuma kritērija, ko izvēlas, un modeļa veida. Galīgo stāvokļu automātiem pārklājumi var būt pār ievadiem, stāvokļiem un pārejām, piemēram, visu stāvokļu pārklājums. Šo pārklājumu var iegūt manuāli vai automātiski traversējot caur modeli. Algoritma sarežģītība galīga automāta visu virsotņu pārklājuma izveidošanai ir līdzīga visu virsotņu apstaigāšanai orientētā grafā - V^2, kur V – virsotņu skaits

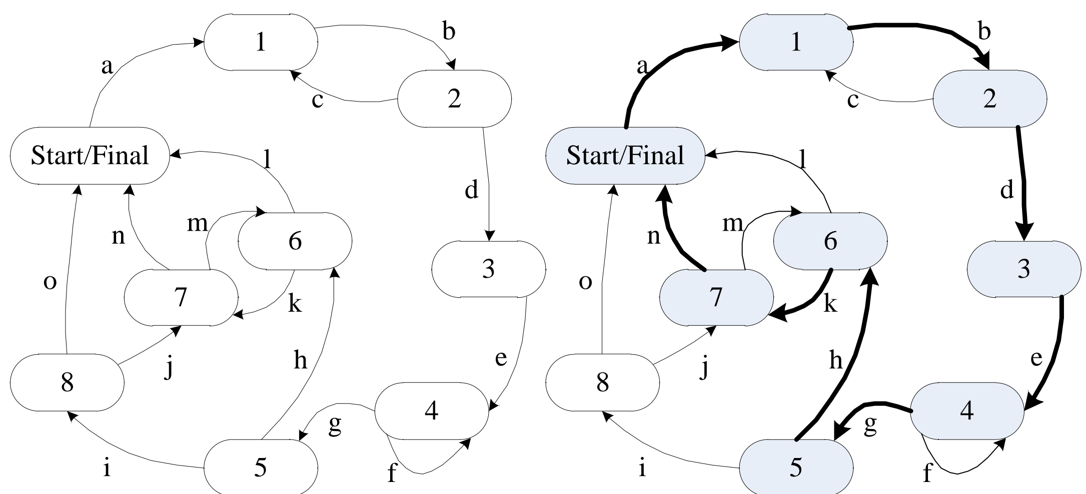
Pasaulē nav visaptverošu pētījumu par dažādu pārklājumu veidu efektivitāti [AW93], [RR00].

Ja modeli attēlo kā galīgo stāvokļu automātu, piemēram, kā 2.19. (a) attēlā, tad testpiemērs ir ceļš orientētā grafā. Modeļa grafa virsotnes ir automāta stāvokļi, bet katra šķautne norāda pāreju no viena stāvokļa uz citu. Iezīmes pie šķautnēm norāda darbību, kuras rezultātā notiek pāreja, piemēram, ar peles kreiso taustiņu uzklikšķinot uz kāda elementa saskarnē vai nospiežot kādu taustiņu kombināciju, vai ievadot kādas ievadvērtības.

Ja modeļa katra virsotne reprezentē programmas vienu ievadparametru, bet šķautnes – atlasītās pārbaudāmās ievadvērtības, tad tas grafiski ataino domēnu modeli.

Testējot cenšas panākt dažādus modeļa grafu pārklājumus, piemēram, iziet pa visām grafa šķautnēm, iziet pa visiem grafa stāvokļiem. Ja vienā virsotnē ieiet vairākas šķautnes un/vai iziet no tās vairākas šķautnes, var pārbaudīt visu ieejošo un izejošo šķautņu kombinācijas, var šo kombināciju daudzumu 'pagarināt' iekļaujot 3 un vairāk citu citai sekojošas šķautnes.

Tāpat var mēģināt notestēt visus ceļus modeļa grafā. Ja grafs ir ciklisks, tad ierobežo maksimālo ciklošanās skaitu, piemēram, ciklus iziet ne vairāk kā vienu vai divas reizes.



2.19. att. **Galīgo stāvokļu automāts un testpiemēra attēlojums tajā**

Programmatūras modeļa grafu var uztvert kā orientētu grafu.

Orientēts grafs $\mathcal{G} = (V, E)$, kur V ir virsotņu kopa, E ir orientētu šķautņu kopa, pie tam $E \subseteq V^2$ [Rab08].

Grafs $\mathcal{G}' = (V', E')$ ir $\mathcal{G}$ apakšgrafs, ja $V' \subseteq V$ *un* $E' \subseteq E$ [Rab08].

Orientēts ceļš $\mathcal{P} = (P, E_P)$ ar garumu $n \geq 0$ no virsotnes v_0 uz virsotni v_{n-1} ir orientēts grafs, kur $P = \{ v_0, \ldots , v_{n-1} \}$ un , visas virsotnes v_i ir dažādas un $E_P = \{\{ v_i, v_{i+1}\} \mid 0 \leq i \leq n - 2\}$.

Grafs $\mathcal{G}$ ir saistīts, ja visām virsotnēm v, $w \in V(\mathcal{G})$, eksistē orientēts ceļš grafā $\mathcal{G}$ vai nu no v uz w, vai no w uz v [Rab08].

Virsotņu apakškopa $X \subseteq V(\mathcal{G})$ *rada* orientētu grafa $\mathcal{G}$ apakšgrafu $\mathcal{G}[X]$ sekojoši:

(N1) $V(\mathcal{G}[X]) = X$ un

(N2) $E(\mathcal{G}[X]) = \{ \{u, v\} \in E(\mathcal{G}) : u, v \in X.$

Grafu teorijā aplūko vairākas grafu sarežģītības metrikas. Programmatūras testēšanas sarežģītības novērtēšanai var izmantot ceļu platuma un koku platuma metrikas.

Definīcija (*Koku dekompozīcija un koku platums*). Ja $\mathcal{G}$ ir orientēts grafs, *koku dekompozīcija* grafam $\mathcal{G}$ ir pāris $(\mathcal{T}, \mathcal{X})$, kur $\mathcal{T}$ ir koks un $\mathcal{X} = (X_t : t \in V(\mathcal{T}))$, un kur $\forall i$, $1 \leq i \leq n$, $X_i \subseteq V(\mathcal{G})$ un:

(T1) $\bigcup_{t \in V(\mathcal{T})} X_t = V(\mathcal{G})$,

(T2) katrai šķautnei $\{u, v\} \in E(\mathcal{G})$ eksistē virsotne $t \in V(\mathcal{T})$ tāda, ka $\{u, v\} \subseteq X_t$,

(T3) katrai virsotnei $v \in V(\mathcal{G})$ koka $\mathcal{T}$ apakšgrafs, ko rada kopa, kam tā pieder $\{ t : v \in X_t \}$, ir saistīts koks.

Dekompozīcijas $\mathcal{X}$ *platums* ir $\max\{|X_i| : 1 \leq i \leq t\} - 1$. Grafa $\mathcal{G}$ *koku platums* ir mazākais visu grafa $\mathcal{G}$ koku dekompozīciju platums.

Koku dekompozīciju grafam $\mathcal{G}$ veido tāds koks $\mathcal{T}$ un tā apakškoku kopa $\mathcal{X}$, ka apakškoki ietver visas grafa $\mathcal{G}$ virsotnes (nosacījums (T1)) un visas šķautnes (T2). Turklāt, jebkurai šķautnei $e = \{u, v\} \in E(\mathcal{T})$, ja to izņem no $\mathcal{T}$, iegūst divus jaunus apakškokus, kur viens no tiem, $\mathcal{T}_u$, satur virsotni u, bet otrs - $\mathcal{T}_v$ – satur virsotni v (T3).

Nosacījumus (T2) un (T3) definīcijā var aizvietot ar sekojošu nosacījumu:

(T4) ja $t, t', t'' \in V(\mathcal{T})$ un virsotne t' atrodas ceļā no virsotnes t uz virsotni t'', tad $X_t \cap X_{t''} \subseteq X_{t'}$.

Koku platums izsaka to, cik tuvu grafa struktūra ir kokam. Jo koku platums ir lielāks, jo vairāk ciklu ir grafā, un sarežģītāka ir grafa testēšana, jo nepieciešams veidot papildus testpiemērus tieši ciklu testēšanai.

Definīcija (*Ceļu dekompozīcija un ceļu platums*). Ja $\mathcal{G}$ ir orientēts grafs, *ceļu dekompozīcija* grafam $\mathcal{G}$ ir virkne $X_1, \ldots, X_n$, kur $\forall i$, $1 \leq i \leq n$, $X_i \subseteq V(\mathcal{G})$, tāda, ka

(P1) $\bigcup_{i=1}^{n} X_i = V(\mathcal{G})$,

(P2) ja $i \leq j \leq k$, tad $X_i \cap X_k \subseteq X_j$, un

(P3) katrai šķautnei $e = \{u, v\} \in E(\mathcal{G})$ eksistē $i \leq n$ tāds, ka $\{u, v\} \subseteq X_i$.

Ceļu dekompozīcijas $X_1, \ldots, X_n$ platums ir $\max\{|X_i| : 1 \leq i \leq k\} - 1$. Grafa $\mathcal{G}$ *ceļu platums* ir mazākais visu grafa $\mathcal{G}$ ceļu dekompozīciju platums.

Ceļu platums izsaka mazāko iespējamo ceļu skaitu, kas pārklāj grafu $\mathcal{G}$. Ja grafs $\mathcal{G}$ apraksta programmatūras modeli, tad tā ceļu platums izsaka minimālo testpiemēru skaitu, kas nepieciešams, lai pārklātu visus ceļus grafā jeb iegūtu grafa ceļu pārklājumu.

Ja grafam ir ceļu platums *k*, tad tam koku platums ir ≤ *k*. Šīs metrikas var atšķirties ievērojami, piemēram, ja grafs $\mathcal{G}$ ir koks, tad tam koku platums ir 1, bet ceļu platums var būt daudz lielāks.

Lai noteiktu koku platumu, nepieciešams atrisināt sekojošu problēmu: ja ir dots orientēts grafs $\mathcal{G}$ un naturāls skaitlis k, pateikt, vai grafa $\mathcal{G}$ koka platums ir ne lielāks par k. Šī problēma ir NP-pilna [ACP87]. Taču ir grafu klases, kam šis jautājums ir atrisināms polinomiālā laikā

Thorup ir parādījis, vadības plūsmas grafam programmām bez GOTO operatora izmantojuma, kas uzrakstītas tādās valodās kā C un Pascal, koku platums ir maza konstante [Tho98]. Savukārt daudzas problēmas var tikt atrisinātas lineārā vai polinomiālā laikā, ja grafa koku platums ir ierobežots [KT05].

Tātad, lai konstruētu modeļa grafa pārklājumu un noteiktu testpiemēru skaitu, nepieciešams (1) noteikt grafa koku platumu un konstruēt tos, un (2) katram kokam noteikt ceļu platumu un konstruēt tos.

Orientētā grafā, kas ir koks, uzkonstruēt visus ceļus var lineārā laikā. Taču vispārīgā gadījumā orientētam grafam noteikt koku daudzumu un uzkonstruēt tos praktiski nav iespējams.

Turklāt, ja grafa šķautnēm ir piesaistītas programmas parametru vērtības vai to varianti, tad ir jāpārbauda arī to dažādie varianti, atkal atgriežoties pie domēntestēšanas metodēm parametru vērtību atlasē un kombinēšanā, taču, kā tika parādīts 2.1. nodaļā arī to sarežģītība praktiskam uzdevuma ir pārāk liela.

Līdz ar to arī ar modeļu palīdzību pārklājumu veidoto testkomplektu apjoms praksē var izrādīties neizpildāmi liels.

2.5. Testēšanas metožu efektivitātes un produktivitātes salīdzināšana

Dažādi autori ir mēģinājuši ar analītiskām metodēm salīdzināt dažādu testēšanas metožu efektivitāti un produktivitāti. Testēšanas metodēm ir tikusi vērtēta to spēja atrast kļūdas, kā arī to atbilstošā izveidotā testkomplekta apjoms [FI98], [FW93a], [FW93b], [Ham89], [HFG+94], [Hut94], [Off92]. Diemžēl vēl arvien nav efektīva problēmas risinājuma [FI98]. Vairumā sākotnējo pētījumu lietotas teorētiskās un formālās metodes [FI98], [Off92]. Darbu [FW93a], [FW93b], [Hon96] rezultāti atļauj dažādu testēšanas metožu sarežģītību salīdzināt pēc divu veidu metrikām.

Testēšanas metožu efektivitāti un produktivitāti avotos analizē divos veidos. Pirmais veids – sagaidāmais atrasto kļūdu skaits, otrais – varbūtība atrast vismaz vienu kļūdu.

Pieņemsim, ka ir programma *p*, kurai ir specifikācija *s*. Tā tiek testēta atbilstoši metodes *C* kritērijiem.

Testēšanas metodes kritērijs *C* nosaka, vai testkomplekts *T* ir atbilstošs, lai adekvāti iztestētu programmu *p* attiecībā pret specifikāciju *s* saskaņā doto testēšanas metodi. Var teikt, ka *T* ir *C*-adekvāts attiecībā pret *p* un *s*.

Pieņem, ka $SD_C(p,s)=\{D_1, D_2, ..., D_k\}$ – ir saskaņā ar metodi *C* sadalīta *(p,s)* ievaddatu telpas apakšdomēnu multikopa (multikopa ir tāda kopa, kuras elementiem nav jābūt unikāliem, t.i., tā var saturēt vairākus vienādus elementus, tātad apakšdomēni var pārklāties),

Pieņem, ka $d_i=|D_i|$ (apakšdomēna D_i elementu skaits) un m_i ir skaits kļūdām, kuras var iegūt, izpildot programmu *p* ar visām D_i vērtībām.

Frankl un *Weyuker* piedāvā metrikas M_1 , M_2 un M_3. Metrika M_1 [WJ91] (formula (1)) izsaka varbūtību, ar kādu metodei *C* atbilstošs testkomplekts atradīs vismaz vienu kļūdu programmā *p*, kuras specifikācija ir *s*.

$$M_1(C,p,s) = \max_{1 \le i \le k}\left(\frac{m_i}{d_i}\right) \quad (1)$$

Metrika M_2, definēta ar formulu (2) [FW93b], izsaka varbūtību, ar kādu tiks atrasta kļūda, ja tieši viens testpiemērs no katra apakšdomēna tiks izvēlēts un katra testpiemēra t_i izvēlēšanas varbūtība no apakšdomēna D_i ir ar varbūtību $\frac{1}{n}$.

$$M_2(C,p,s) = 1 - \prod_{i=1}^{k}\left(1 - \frac{m_i}{d_i}\right) \quad (2)$$

M_3 ir M_2 vispārinājums [FW93b], ņemot vērā n testpiemērus no katra apakšdomēna (formula 3).

$$M_3(C,p,s,n) = 1 - \prod_{i=1}^{k}\left(1 - \frac{m_i}{d_i}\right)^n \text{, kur } n \geq 1 \quad (3)$$

Frankl un *Weyuker* [FW93a] piedāvā arī metriku *E* (formula (4)), kas ir sagaidāmais kļūdu, ko atradīs testēšana atbilstoši kritērijam *C*, skaits.

$$E(C,p,s) = \sum_{i=1}^{k}\left(\frac{m_i}{d_i}\right) \text{, kur } k \geq 1 \quad (4)$$

Visas šīs metodes grūti piemērojamas praktiskajā dzīvē, jo nav zināms, kā noteikt atrodamo kļūdu skaitu m_i katrā apakšdomēnā D_i. Tuvināti metrikas var izmantot situācijā, kad jau programmatūra ir testēta un tātad ir zināms, cik kļūdu atrasts katrā apakšdomēnā. Tad, saņemot no izstrādātājiem jaunu programmatūras versiju, testētāji uz vēstures bāzes var izvērtēt, kuras metodes kurus apakšdomēnu testēšanai izmantot, pieņemot, ka jaunās kļūdas būs līdzīgā veidā atrodamas kā iepriekšējās.

Morasca un *Serra-Capizzano* [MSC04] vispārina *Frankl* un *Weyuker* rezultātus.

$c_i = d_i - m_i$ – vērtību skaits apakšdomēnā D_i, kas neizraisa kļūdas.

Katram apakšdomēnam $a \in SD$, $p^{(k)}(a)$ nozīmē varbūtību, ka testkomplekta k-tais testpiemērs ir ņemts no apakšdomēna *a* ($\sum_{a \in SD} p^{(k)}(a) = 1$).

Skaitu, cik reizes tiek sagaidīts, ka vērtības apakšdomēna tiks ņemtas testkomplekta veidošanai, apzīmē ar *att(a)* ($\sum_{a \in SD} att(a) = z$, kur *z*- testpiemēru skaits metodei *C* adekvātā testkomplektā T_C, $z = |T_C|$)

Varbūtība, ka programmas kļūdu izraisa k-tais atlasītais testpiemērs, sauc par *kļūdu indeksu* un apzīmē ar $\theta^{(k)}$.

Programmas *korektuma indeksu* (varbūtība, ka k-tais testpiemērs kļūdu neizraisa) pie k-tā testpiemēra apzīmē ar $\gamma^{(k)} = 1 - \theta^{(k)}$

Apakšdomēnam atbilstošo lielumus apzīmē kā $\theta^{(k)}(a)$ un $\gamma^{(k)}(a) = 1 - \theta^{(k)}(a)$.

Sagaidāmo kļūdu skaitu *E*, ko atradīs testēšana atbilstoši kritērijam *C* , var izteikt ar formulu 5.

$$E = \sum_{a \in SD} att(a)\theta(a) \quad (5)$$

Ne vienmēr vairāki testkomplekti, kas ir adekvāti metodei ar kritēriju, būs vienādi efektīvi kļūdu atrašanā. *Vilkomir et.al.* [VKB03] ievieš testēšanas metodes tolerances jēdzienu. *Tolerance* – jebkura testkomplekta, kas ir adekvāts metodes kritērijam C, spēja uzrādīt līdzīgu efektivitātes līmeni. Kritērijiem ar augstu toleranci dažādu testkomplektu efektivitāte atšķirsies maz, bet kritērijiem ar zemu toleranci tā var atšķirties būtiski. Viena no tolerances metrikām var būt efektivitātes sadalījuma standarta novirze [VKB03], kā izteikt formulā (6).

$$R^2(C,p,s) = \frac{1}{d_S}\sum_{i=1}^{d_S}\left(E_{t_i}(C,p,s) - E(C,p,s)\right)^2 \qquad (6)$$

Diemžēl apskatītās metrikas neļauj salīdzināt tās testēšanas metodes, kas visas izmanto vienādu domēnu sadalījumu apakšdomēnos, bet dažādas stratēģijas testpiemēru izvēlē no katra apakšdomēna.

Dažādu testēšanas metožu efektivitāte vērtēta arī eksperimentos [BS87], [HFG+94], [FW93a], [FW93b], [Hut94], bet analizēto programmu daudzums un apjoms ir pārāk mazi, lai rezultāti būtu vispārināmi.

2.6. Nodaļas secinājumi

Ekvivalences klašu un robežvērtību analīzes metodes tiek bieži pieminētas grāmatās par programmatūras testēšanu. Vairumā gadījumu tajās aprakstīti pamatprincipi, kā izveidot ekvivalences klases, kā atrast robežvērtības, kā arī parādīti daži veidi, kā, tās izmantojot, veidot testpiemērus. Katrs autors parasti uzsver pāris paņēmienus, bet neanalizē to vājās un stiprās puses, tikai demonstrē dažus piemērus. Pozitīvi izceļas *Jorgensen*, kurš pieminējis gan vairāk metožu, gan mēģinājis tās saklasificēt un dot tām nosaukumus [Jor95].

Šajā nodaļā ir mēģināts apkopot un aprakstīt literatūrā biežāk minētās ekvivalences klašu un robežvērtību testēšanas metodes. Darbā arī novērtēta un apkopota domēntestēšanas metožu sarežģītība, kā arī dota metožu hierarhija pēc iekļautības kritērija. Testēšanas metožu sarežģītība skatīta kā metodes ģenerētā minimālā testkomplekta apjoms.

Autore secina, ka, ja kāda domēntestēšanas metode iekļauj citu metodi, tad pirmās metodes sarežģītība ir lielāka, nekā iekļautās metodes sarežģītība. Taču pretēji apgalvot ir aplami.

Darbā analizēti domēntestēšanas metožu adekvātuma kritēriji no trim aspektiem – testēšanā izvēlēto vērtību veida, datu pārklājuma principa un pēc stratēģijas, kas tiek

lietota, lai izvēlētās dažādu programmas parametru vērtības savstarpēji kombinētu testpiemēros, ievērojot datu pārklājuma principu.

Kaut arī literatūrā ir ekvivalences klašu testēšana un robežvērtību testēšana bieži tiek pieminēta, daži svarīgi aspekti vēl arvien ir nepietiekami izpētīti. Viens no tiem ir, kā testēšanas efektivitāti ietekmē datu pārklājuma kritērija izvēle un programmas dažādu parametru speciālo un robežvērtību kombināciju pārklājums.

Nodaļā dots kombinācijtestēšanas pārklājuma kritēriju pārskats, kā arī kombinācijtestēšanas pārklājuma kritēriju un atbilstošo metožu hierarhija pēc iekļautības principa.

Secināts, ka kombinācijtestēšanas metožu sarežģītība tiešā veidā ir saistīta ar datu atlases kritērijiem. Triviālākajā gadījumā pietiek atlasīt vienu pieļaujamo un vienu nepieļaujamo vērtību katram parametram un tās kombinēt atbilstoši pārklājuma kritērijiem. Šāda stratēģija varētu būt noderīga dūmu testiem, lai noskaidrotu, vai programmatūra vispār strādā, bet ne testiem programmatūrā esošo kļūdu atrašanai. Ja vēlas testēšanu veikt dziļāk, vērtību grupēšanai izvēlei nākas izmantot domēntestēšanas metodēs pielietotos vērtību atlases kritērijus, kuru atlasītās vērtības kombinēt atbilstoši kombinācijtestēšanas pārklājumu principiem, reālās situācijās iegūstot testkomplektu ar ļoti lielu apjomu vai arī testkomplektu ar mazu apjomu, bet, iespējams, ar mazu efektivitāti kļūdu atrašanā programmatūrā.

Nodaļā ir arī aplūkota modeļbāzēto metožu sarežģītība un tās novērtējums, izmantojot grafu teoriju.

Modeļbāzētās testēšanas nodaļā tika parādīti pasaulē zināmie rezultāti par to, ka lai iegūtu testkomplektu atbilstoši modelim, nepieciešams:

1. Modeļa ciklisko grafu pārvērst par aciklisku grafu jeb koku kopu.
2. Katram iegūtajam kokam neieciešams iegūt testpiemēru komplektu, kas spēj nodrošināt koka pārklājumu saskaņā ar kādu kritēriju.
3. Koka pārejām jeb šķautnēm var būt pielikti nosacījumi to iziešanai, kas ir datu apgabali vai pārskaitītas vērtības. Lai notestētu šķautnēm piesaistītās vērtības, nākas izmantot domēntestēšanas pieejas, turklāt nepieciešams izmantot kombinācijtestēšanas metodes, lai katrai šķautnei izvēlētās vērtības kombinētu gan savstarpēji (ja ir vairāki parametri, kam šķautne maina vērtības), gan vērtību kombinācijas uz vienā vai vairākos soļos pēc kārtas izvēlēto šķautņu parametru vērtības.

Tas nozīmē, ka testētāji var būt atkal spiesti konstatēt lielu testēšanas sarežģītību – testpiemēru skaitu, ko nepieciešams izpildīt lai notestētu modeli atbilstoši izvēlētam kritērijam vai to kopai.

Rezultātā nākas secināt, ka testēšanas metodes nesniedz pietiekami efektīvu iespēju testpiemēru skaita reducēšanai, ievērojami nezaudējot testēšanas efektivitāti – spēju atrast problēmas testējamajā programmatūrā, bet iekļaujoties testēšanas projekta organizatoriskajos ierobežojumos. Tātad programmatūras testēšanas tehnoloģiskā sarežģītība reālam projektam var būt praksē nepārvarami ļoti liela. Tādēļ nākas meklēt citus risinājumus.

Apakšnodaļās 2.1. un 2.2. aprakstītie rezultāti ir darba autores jaunieguldījums un ir publicēti [Arn09]. Apakšnodaļā 2.1. autore ir no literatūras apkopojusi dažādas ekvivalences klašu un robežgadījumu testēšanas metodes, devusi tām nosaukumus, ja tie jau nebija doti avotā [Jor95], novērtējusi metožu sarežģītību un izveidojusi to iekļautības hierarhiju. Vairumā gadījumu literatūrā ir bijis dots tikai īss metodes raksturojums, tikai dažām metodēm ir avotā [Jor95] dots arī sarežģītības novērtējums.

Apakšnodaļā 2.4. autore ar rezultātiem no grafu teorijas parāda, ka arī modeļbāzētās testēšanas sarežģītība praksē pārāk liela, lai šo pieeju varētu pielietot vispārējai programmatūras testēšanai.

3. PROGRAMMATŪRAS TESTĒŠANAS PROCESU SAREŽĢĪTĪBA

Mūsdienās programmatūra ir ne tikai ļoti komplicēta, bet arvien vairāk sistēmu uzvedas kā kompleksās sistēmas, it īpaši, ja tiek ņemta vērā apkārtējās vides ietekme uz sistēmu. Viena no komplekso sistēmu pamatīpašībām ir fakts, ka sistēmas kopējā uzvedība nevar tikt aprakstīta tikai kā atsevišķu sastāvdaļu uzvedību summa. Programmatūras testēšanas procesus var uzlabot, gan uzlabojot izpratni par testēšanas procesiem kā tādiem, gan arī par testējamo programmatūru. Testētājiem sarežģītība nav jānovērš, viņiem ir jārēķinās ar to, lai sasniegtu savus mērķus. Daudzu testēšanas problēmu cēloņus palīdz izprast komplekso sistēmu teorija. Šajā nodaļā programmatūras testēšanas sistēmu tiek piedāvāts uzlūkot kā kompleksu sistēmu. Svarīgākie nodaļas rezultāti ir publicēti rakstā [AA11].

3.1. Testēšanas procesi, to sarežģītība

Process ir sistemātiska operāciju izpilde kāda noteikta rezultāta iegūšanai [DPA+95]. Programmatūras testēšanas sarežģītību ietekmē arī tas, kā tiek organizēti procesi testēšanā. Runājot par konkrētas programmatūras testēšanu no tās organizatoriskā viedokļa, var domāt par to, kā par projektu - projekts ir parasti īslaicīgs veidojums, kurā iesaistītas, iespējams, vairākas organizācijas vai to daļas, konsultanti, apakšuzņēmēji, u.c. dalībnieki un kura mērķis ir sasniegt vai pārspēt ieinteresēto pušu cerības.

Baccarini piedāvā projekta sarežģītību novērtēt, izmantojot divas dimensijas, kur pirmā nozīmē diferenciāciju un savstarpējo sasaistītību, bet otrā – tehnoloģisko un organizatorisko sarežģītību [Bac96].

Ar diferenciāciju saprot elementu daudzumu un dažādību projektā. No organizatoriskā viedokļa var runāt par horizontālo un vertikālo diferenciāciju, kur vertikālā nozīmē organizacionālo vertikālo struktūru (līmeņu skaitu), bet horizontālā – gan organizatorisko vienību (departamentu, grupu, utt.) skaitu, gan darbu sadalīšanu pēc struktūras (līdz vienkārši veicamiem uzdevumiem) vai pēc izpildītāju specializācijas.

Ar sasaistītību saprot projekta elementu savstarpējo mijiedarbību daudzumu, saites starp dažādām darbībām, kas veicamas. Šajā nozīmē projekta sarežģītība var tikt interpretēta arī kā dažādu faktoru kopums [Bac96] – projekta kritiskums, projekta

caurredzamība, projekta uzdevumu skaidrība. Šajā nozīmē par sarežģītību var runāt arī, ņemot vērā tā izprašanas grūtības. Tāpēc sarežģītība ir subjektīva un atkarīga no novērotāja.

Projekta sarežģītību otra dimensiju veido organizatoriskā sarežģītība un tehnoloģiskā sarežģītība. Projekta organizatorisko struktūru veido komunikācijas un atskaišu saites, lēmumu pieņemšanas tiesības, atbildības uzlikšanas tiesības, uzdevumu uzlikšanas tiesības. Tehnoloģiskā sarežģītība izsaka veicamo uzdevumu izpildes grūtības.

Xia un *Lee,* apskatot IS izstrādes projektu sarežģītības, arī izdala divas sarežģītību dimensijas – (1) organizatoriskā sarežģītība un tehnoloģiskā sarežģītība, (2) strukturālā un dinamiskā sarežģītības [XL04]. Vēlāk tehnoloģiskā sarežģītība IS izstrādes projektos jau tiek saukta par IT sarežģītību [XL05], uzsverot, ka IS izstrādes projekti atšķiras no citu veidu projektiem ar savu sarežģītību

Bacarini uzskata, ka sarežģītība pēc būtības atšķiras no diviem citiem projektu raksturlielumiem – apjoma un nenoteiktības [Bac96], bet *Turner* un *Cochrane* piedāvā nenoteiktību kā vēl vienu projektu sarežģītības dimensiju [TC93]. Savukārt *Williams* runā par strukturālo sarežģītību projekta iekšējās struktūras nozīmē un par uz nenoteiktību balstītu sarežģītību projekta neskaidrās vai mainīgās dabas nozīmē [Wil99].

3.2. Testēšanas sistēma kā kompleksa sistēma

Pētījumi rāda, ka programmatūras testēšanai tiek patērēti 30-60% no visiem programmatūras izstrādē izlietotajiem resursiem [Par01]. Lielais resursu patēriņš un fakts, ka sasniegtie rezultāti nereti neatbilst gaidītajiem, liek domāt par iemesliem un veidiem, kā uzlabot programmatūras testēšanas procesus.

Šajā darbā tiek pieļauts, ka daudzu problēmu cēlonis ir testēšanas procesu arvien pieaugošā sarežģītība. Programmatūras testēšanas sistēmu var uzlūkot kā kompleksu sistēmu. Komplekso sistēmu spēks izpaužas spējā pašorganizēties, adaptēties, emerģēt un attīstīties (*evolve*) [Mit03]. Zināšanas par kompleksajām sistēmām ļauj labāk saprast programmatūras testēšanu un ļauj meklēt jaunas netradicionālas pieejas procesu pārvaldībā.

Testpiemēri var tikt veidoti ar nolūku sasniegt dažādus mērķus, tomēr programmatūras testēšanai ir divi galvenie mērķi – atrast kļūdas testējamā programmatūrā, kā arī demonstrēt programmatūras atbilstību tās prasībām [GH88].

Var domāt, ka labākā metode visu mērķu sasniegšanai ir izsmeļošā testēšana (*exhaustive testing*) – darbināt testējamo programmatūru, izmantojot visas iespējamās ievadparametru vērtību kombinācijas visos iespējamos testējamās programmatūras stāvokļos un visos iespējamos tās vides- līdzprogrammatūras un datorinfrastruktūras stāvokļos [SLS07].

Tomēr, kaut arī izsmeļošā testēšana spēj atrast visas vai gandrīz visas kļūdas testējamajā programmatūrā, praksē un vairumā gadījumu arī teorētiski to veikt nav iespējams, jo gan izpildāmais testpiemēru skaits parasti ir neizpildāmi liels, gan arī iespējamo programmatūras un tās vides stāvokļu kombināciju skaits ir liels. Tas nozīmē, ka arī programmatūras testēšanas sarežģītība visu izpildāmo testpiemēru visās iespējamās vidēs skaita nozīmē var būt ļoti liela.

Pilnas izsmeļošās testēšanas izpildes neiespējamība liek programmatūras testēšanā iesaistītajām pusēm un testētājiem pārvarēt testēšanas tehnoloģisko sarežģītību. To var panākt dažādos veidos. Piemēram, iesaistītās puses neuzdod testētājiem uzdevumus, kas prasa atrast visas kļūdas, neprasa pierādīt vai garantēt, ka testējamā programmatūra darbojas tieši tā, kā noteikts tās specifikācijā, un nedara to, kas tai nav jādara. Lai samazinātu sava darba apjomu, testētāji pielieto dažādas metodes, tehnikas, metodoloģijas. Viņu mērķis ir veikt testēšanu tā, lai tā būtu pēc iespējas efektīva, ņemot vērā ieinteresēto pušu dotos uzdevumus, kā arī laika un budžeta ierobežojumus.

3.2.1. Līdzšinējie pētījumi

Programminženierijā nav viena vispārpieņemta testēšanas sarežģītības metrika. Tomēr šajā jomā ir bijuši pētījumi un laika gaitā ir mainījusies izpratne par testēšanas sarežģītību.

3.2.1.1. Testēšanas procesu organizācija un pārvaldība

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, 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".

3.2.1.2. *Sistēmu sarežģītības*

Dažādu jomu pētnieki ir pētījuši dažādus sarežģītības aspektus. Piemēram, *Thorsten* [TBW06] ir definējis trīs pārklājošas sarežģītību grupas – laika sarežģītības, organizācijas sarežģītības un sistēmiskās sarežģītības. Laika sarežģītības var būt statiskas un dinamiskas. Statiskās laika sarežģītības parāda sistēmas metriku konkrētā laika momentā. Savukārt dinamiskās laika sarežģītības rāda mērījumu izmaiņas laikā.

Organizācijas sarežģītības izsaka sistēmas strukturālo sarežģītību (elementu, saišu un komunikāciju starp elementiem skaitu un dažādību), kā arī procesu sarežģītības (procesu skaitu un dažādību).

Sistēmiskās sarežģītības izsaka sistēmas iekšējo un ārējo sarežģītību [Jos04]. Iekšējo sarežģītību uzsaka sistēmas organizatoriskā struktūra un procesi, kas notiek tajā. Ārējo sarežģītību veido dati, enerģija un resursi, kas ienāk sistēmā no tās apkārtējās vides.

Sarežģītības parasti nav neatkarīgas. Kad vienas no tām mainās, mainās arī citas.

3.2.2. Testēšanas sistēmas sarežģītības

Programmatūras testēšanai kā procesam ir ekonomiskie, tehnoloģiskie un pārvaldības aspekti [Bur03]. No ekonomiskā viedokļa testēšanai ir jāiekļaujas laika un budžeta ierobežojumos un jāizpilda ieinteresēto pušu dotie uzdevumi. Tie būtiski ietekmē testēšanas politiku un stratēģiju. Tehnoloģiskie aspekti attiecas uz testēšanas metodēm, tehnikām. Pārvaldības aspekti saistīti ar organizatoriskiem jautājumiem, piemēram, testēšanas politikas un stratēģijas izstrādi, testēšanas dokumentēšanu, testētāju apmācību, testēšanas procesu metriku mērīšanu, nodrošināšanu, lai testēšanas procesi attīstās un nepārtraukti uzlabojas.

3.2.2.1. *Testēšanas sistēma (TS)*

Ar *sistēmu* saprot mijiedarbojošos vienību vai elementu kopu, kas veido integrētu veselumu, kura mērķis ir veikt kādu funkciju [Sky05]. No praktiska sistēmu pārvaldības

viedokļa var uzskatīt, ka *sistēma* ir cilvēku, tehnikas un materiālu organizēts sakopojums, kas nepieciešams, lai sasniegtu noteiktu mērķi un ko saista kopā komunikāciju saites [Sky05].

Ja programmatūras testēšanu uzlūko kā sistēmu, var teikt, ka *testēšanas sistēma* ir sociotehniska sistēma [Gee04], [JR00], [MW04], [Paj00], kurā ietilpst:

- Iesaistītie cilvēki – testētāji, lietotāji, programmatūras izstrādātāji veido organizāciju, kas ir laikā mainīga kompleksa sociāla sistēma.
- Testējamā sistēma – programmatūra mūsdienās var būt kompleksa sistēma pati par sevi ar daudzām neatkarīgām komponentēm, kas savstarpēji saistītas ar ciešākām vai vaļīgākām saitēm. Šo komponenšu uzdevums ir strādāt neparedzamā vidē un piemēroties tai.
- Datortehnika, programmatūra – testēšanas rīki, līdzprogrammatūra, operāciju sistēma un atbilstošā infrastruktūra.

Testēšanas sistēmas mērķis ir izpildīt ieinteresēto pušu dotos uzdevumus jeb sasniegt viņu uzdotos mērķus dotajā laikā un ar dotajiem resursiem.

Testēšanas sistēmas robežas ir diezgan nenoteiktas un mainās laika gaitā. Piemēram, programmatūras izstrādātāji vai lietotāji var būt un var nebūt sistēmas sastāvdaļas, atkarībā no veicamajiem uzdevumiem.

Katra no TS apakšsistēmām attīstās un mainās dzīves laikā. Testējamā programmatūra tiek papildināta ar jaunu funkcionalitāti, tiek identificēti un uzlaboti moduļi, kuros ir problēmas. Testētāju komanda var mainīties atkarībā no testēšanas līmeņa, veida un pielietotās metodes. Datortehnikas un programmatūras vide mainās un tiek mainīta, lai pārbaudītu testējamās programmatūras darbu visdažādākajās vidēs, kā arī, lai testētāju darbu padarītu pēc iespējas ērtāku. Testēšanas rīki tiek uzlaboti, atsevišķi testēšanas procesu elementi tiek automatizēti.

Testēšanas sistēmas rezultāti veido testēšanas artefaktus – atrastās un aprakstītās kļūdas, testu projektēšanas dokumentāciju, testkomplektus, testu izpildes dokumentāciju, testēšanas atskaites.

3.2.2.2. Laiks, budžets, uzdevumi – TS ārējās sarežģītības

Testēšanas sistēmas ārējā sarežģītība [Jos04] ir funkcija no laika un budžeta ierobežojumiem, kā arī ieinteresēto pušu dotajiem uzdevumiem. Laiks, budžets un uzdevumi ietekmē gan viens otru, gan arī testēšanas sistēmas rezultātus.

Testēšanas sistēmas ārējās sarežģītības rodas no ieinteresēto pušu prasībām un apkārtējās vides. Noteiktie laika termiņi, budžets un uzdevumi, kas jāizpilda, ir

savstarpēji saistītas sarežģītības, kas ļoti būtiski ietekmēs testēšanas sistēmas rezultātus. Lai strādātu optimāli, testēšanas sistēmai ir vai nu jāpiemērojas šīm sarežģītībām vai tās jāmaina, ietekmējot ārējo vidi – ieinteresētās puses, testējamās programmatūras izstrādātājus.

Pieņemsim, ka adekvāta testēšana nozīmē tādu testēšanu, kas izpilda testēšanas sistēmai dotos uzdevumus. Katrā konkrētā situācijā katrai sarežģītībai ir kāda noteikta robeža, kuru pārsniedzot adekvāta testēšana vairs nav iespējama. Piemēram, ja testēšanas sistēmai ir dots uzdevums veikt pilnu testēšanu atbilstoši kādam kritērijam, tad parasti to nav iespējams veikt, ja testējamā programmatūra ir pietiekami liela un kritērijs nav ļoti vispārīgs. Jebkuram uzdevumam ir tāda laika un resursu robeža, zem kuras šis uzdevums vairs nav paveicams.

Vienkāršojot uzdevumus, dodot vairāk laika un resursu, adekvātas testēšanas veikšanas iespēja kļūst reālāka. Laika un resursu apjoma palielināšana, nevienkāršojot uzdevumus, ne vienmēr ir pietiekama doto uzdevumu izpildei.

Testēšanas sistēmas ārējā sarežģītība ir relatīvi zemāka, ja dotais laiks un resursi ir atbilstoši veicamajiem uzdevumiem.

3.2.2.3. Testēšanas sistēmas iekšējās sarežģītības

Testēšanas sistēmas iekšējo sarežģītību veido visu aktivitāšu, kas veicamas testēšanas laikā, sarežģītības kopā. Iekšējās sarežģītības var iedalīt divās grupās – tehnoloģiska rakstura sarežģītības un organizatoriskās sarežģītības.

Ja nav būtisku laika un resursu ierobežojumu, tehnoloģisko sarežģītību var izteikt kā testpiemēru skaitu testkomplektā, kas adekvāti testē testējamo programmatūru attiecībā saskaņā ar testēšanas sistēmai dotajiem uzdevumiem. Ja ņem vērā laika un budžeta ierobežojumus, tad iekšējo sarežģītību ietekmē arī sarežģītība algoritmam, ko izmanto testpiemēru ģenerēšanai testkomplektā, testkomplekta uzturēšanas sarežģītība, kad testējamā programmatūra mainās, testkomplekta izpildes sarežģītība.

Laika un budžeta trūkuma apstākļos testētāji samazina testpiemēru skaitu testkomplektā, tajā pašā laikā mēģinot nezaudēt tā efektivitāti. Lai varētu to izdarīt, pētnieki un testēšanas praktiķi ir izstrādājuši dažādas testēšanas metodes un tehnikas.

Vispirms ir minamas risku vadītās metodes, kas atļauj vairāk testēt svarīgākos testējamās programmatūras apgabalus, piemēram, potenciāli biežāk lietojamos, sarežģītākos, tādus, kuros kļūda radītu vissmagākās sekas.

Nākamās ir metodes, kas teorētiski ļauj samazināt testpiemēru skaitu testkomplektā, saglabājot testēšanas efektivitāti, piemēram, ekvivalences klašu un

robežvērtību testēšanas metodes, kombinācijtestēšanas metodes, modeļbāzētās testēšanas metodes.

Laika sarežģītības var samazināt, izmantojot testēšanas automatizāciju bieži izpildāmiem testpiemēriem.

Paralēli tehnoloģiskajām sarežģītībām testēšanas sistēmā ir arī organizatoriskās sarežģītības, piemēram, testēšanas komandas lielums, testēšanas dažādu prasmju līmenis un dažādība komandā.

Organizatorisko sarežģītību ietekmē arī testēšanas procesu pārvaldības stils, piemēram, vai tiek izmantoti kādi testēšanas procesu pārvaldības ietvariem, kā, piemēram, *Test Maturity Model* (TMM), *Test Process Improvement* (TPI) modelis un *Test Improvement Model* (TIM).

3.2.3. Testēšanas sistēma un kompleksu sistēmu īpašības

Testēšanas sistēmā ir vairākas cieši saistītu problēmu grupas. Pirmā no tām saistīta ar testēšanas sistēmu kā sociālu sistēmu jeb organizāciju. Tai pieskaita tādas problēmas, kas, piemēram, saistītas ar cilvēku motivāciju, hierarhisku subordināciju, cilvēku fizisko izvietojumu izmantotajās telpās, cilvēku spējām. Vairums no šīm problēmām eksistē jebkurā organizācijā, ne tikai testēšanas sistēmās.

Nav vienas vispārpieņemtas kompleksas sistēmas definīcijas. Šajā darbā tiek izmantota sekojoša definīcija: „[...] *kompleksa sistēma* ir jebkura sistēma, kas sastāv no daudzām mijiedarbojošām komponentēm (aģentiem, procesiem, utt.), kuru kopējās aktivitātes ir nelineāras (tās nav secināmas no individuālo komponenšu aktivitāšu kopuma) un selektīvu grūtību rezultātā izrāda hierarhisku pašorganizāciju".

Organizācijas ir kompleksas sistēmas [LR03]. Tādēļ tiek piedāvāts uztvert testēšanas sistēmu kā kompleksu sistēmu. Komplekso sistēmu tipiskākās īpašības [And03], [Cil98], [Mit03] no testēšanas sistēmas viedokļa ir sekojošas.

• **Aģenti, to savstarpējā mijiedarbība un atkarība**

Aģents veic noteiktas aktivitātes vidē, par kuru viņš zina un kas var atbildēt uz izmaiņām. Testēšanas sistēmā ir daudz dažādu aģentu, piemēram, cilvēku, līdzprogrammatūras, testēšanas rīku un testējamās programmatūras autonomās komponentes. Aģenti sistēmas iekšienē ir savstarpēji saistīti ar ļoti daudz saitēm un mijiedarbībām.

• **Pašorganizācija, emerģence, evolūcija**

Šīs ir vienas no vissvarīgākajām kompleksas sistēmas īpašībām, jo tās apraksta sistēmu tādā nozīmē, ka sistēmas kopējā uzvedība nevar tikt uzskatīta par tās dažādo

komponenšu uzvedību summu. Pašorganizācija un emerģenta uzvedība rada jaunu kārtību. Sistēmas elementi organizē sevi ar mērķi sasniegt vēlamo rezultātu.

Pūļa testēšana (*crowd testing*) un pētnieciskā testēšana (*exploratory testing*) ir piemēri metodēm, kas ļauj testētājiem pašorganizēties un izrādīt emerģenci. Ja testēšanas aktivitātes notiek stingri pēc iepriekš izveidota sīka plāna, tad testētājiem ir ļoti grūti pašorganizēties.

Testēšanas sistēmas var evolucionēt katrā testēšanas ciklā vai kampaņā, ja testētāji vāc metrikas par savu darbu, analizē pieļautās kļūdas, veiksmīgos risinājumus, izdara secinājumus, kurus izmanto piemērotās situācijās turpmākajā darbā.

• **Koevolūcija, sinerģija**

Sistēma laika gaitā attīstās un evolucionē, iegūstot jaunas īpašības, kas tai palīdz tikt galā ar iekšējo un ārējo sarežģītību. Aģenti var koevolucionēt, attīstīties, adaptēties viens pie otra un strādāt saskaņoti, sasniedzot savstarpēju sinerģiju.

• **Nelinearitāte**

Sistēmas nelineārā daba var izpausties divos veidos. Pirmais ir cēloņa un efekta veids – neliela aktivitāte var izraisīt ievērojamas sekas. Otrais ir, ka sistēmas uzvedība nav iepriekš paredzama. Testēšanas sistēma nav sistēma, par kuru iepriekš var pateikt, kura tās darbība novedīs pie vēlamā rezultāta – nopietnu kļūdu identificēšanas testējamajā sistēmā. Tajā pat laikā, ja problēma ir identificēta, šis fakts var novest pie tālejošām sekām, piemēram, pat pie visas testēšanas stratēģijas izmaiņām.

• **Equilibrium un far-from-equilibrium stāvokļi**

Testēšanas sistēmu var uzskatīt par tuvu *equilibrium* stāvoklī esošu, ja vairs nav nepieciešams vai dažādu iemeslu dēļ iespējams izstrādāt jaunus testpiemērus vai arī izpildīt jau izstrādātos testus. Ir divi pamatgadījumi, kad testēšanas sistēma nonāk stāvoklī tuvu *equilibrium.*

Pirmais gadījums ir, kad testējamā sistēma tiek uzturēta, bet tā netiek mainīta vai papildināta ar jaunu funkcionalitāti, kā arī netiek mainīta testējamās sistēmas vide. Katras izmaiņas testējamā programmatūrā vai tās līdzprogrammatūrā, jaunas prasības no ieinteresētajām pusēm virza testēšanas sistēmu prom no *equilibrium* stāvokļa.

Otrs gadījums ir, kad testēšanas sistēmai trūkst resursu. Piemēram, ja budžets ir iztērēts, testēšanas sistēma apstājas, ja vien testētāji nestrādā par brīvu.

• **Pozitīvās un negatīvās atgriezeniskās saites**

Pozitīvās atgriezeniskās saites izmaina sistēmu un rada nepieciešamību evolucionēt, mācīties un emerģēt [GU07], [Hey09]. Pozitīvo atgriezenisko ciklu testēšanā veido jaunatklātās kļūdas testējamajā sistēmā, jo tas nozīmē, ka atbilstošajā

programmatūras daļā var būt vēl kļūdas, tātad ir vērts un ir nepieciešams to testēt papildus. No otras puses, jaunas kļūdas atrašana norāda arī, ka tās atklāšanā izmantotā testēšanas metode ir konkrētā gadījumā efektīva un ir vērts to izmantot arī turpmāk, testējot apgabalus, kur tā vēl nav pielietota.

3.2.4. Testēšanas sistēma un komplekso sistēmu principi

Testēšanas sistēmas ir uztveramas kā kompleksas adaptīvas sistēmas [Mit03]. Kompleksu adaptīvu sistēmu priekšrocības rada to pašorganizēšanās, adaptācijas, emerģences un evolucionēšanas spējas. Testēšanas sistēmu vadība organizē un motivē testētāju komandu, dod testētājiem testēšanas mērķus, kas saņemti no ieinteresētajām pusēm, organizē nepieciešamos resursus un sistēmas vidi. Kā kompleksas sistēmas inženieriem viņiem jānodrošina darbības vide un konteksts, kurā sistēma var veiksmīgi pildīt savus pienākumus, izmantojot visas tās iespējas tikt galā ar sarežģītību [Bec00].

Kompleksas sistēmas darbību nevar aprakstīt ar likumiem, kas precīzi nosaka tās darbošanos un rezultātus, jo šāda sistēma vienmēr iekļauj procesus, kas nav iepriekšparedzami – emerģenci un pašorganizēšanos. Šī iemesla dēļ kompleksu sistēmu projektēšanā, izstrādē, uzturēšanā un pārvaldībā izmanto principus [PV00], [SM09], nevis striktus un precīzus likumus. Turpmāk darbā aplūkoti principi, kas ir raksturīgi kompleksu sistēmu inženierijas procesos [MBB06], [NK06] un kurus darbā tiek piedāvāts izmantot programmatūras testēšanas praksē.

Lokāla darbība, globālas sekas. Sistēmas globālie mērķi, pašorganizēšanās un adaptēšanās tiek sasniegti, aģentiem darbojoties lokāli [Aar05]. Testēšanas sistēmā aģenti katrs testē noteiktu programmatūras apgabalu, izmantojot kādu testēšanas metodi. Viņi savā starpā saskaņo mazu aģentu grupu darbus. Aģenti var nemitīgi pārgrupēties, mainoties uzdevumiem vai darba kontekstam, ar mērķi veikt darbu iespējam labāk. Viņi var strādāt paralēli pie viena un tā paša programmatūras apgabala, izmantojot gan vienādas, gan dažādas testēšanas metodes, piemēram, viens var lietot ekvivalences klašu testēšanas metodi, kamēr cits izmantot pētniecisko testēšanu. Tāpat var būt arī aģenti, kas izmanto dažādas pārklājumu metodes. Piemēram, ja aģents rūpējas par komandu pārklājumu, tad viņš cenšas iegūt informāciju pār pārklājumu, ko nodrošina testi, kurus veikuši citi aģenti, lai veiktu programmatūras pārklājuma procentuālu novērtējumu. No otras puses, šis aģents var dot informāciju par programmatūras apgabaliem, kuriem ir vājš pārklājums, citiem aģentiem lokāli no viņa viedokļa cenšoties pēc iespējas labāk izpildīt savu uzdevumu – nodrošināt pēc iespējas labāku komandu pārklājumu, bet

globālā mērogā aģenta darbība uzlabo testēšanas kvalitāti – tā kļūst plašāka, samazinās iespēja, ka paliek netestēti kādi programmatūras apgabali.

Aģenti darbojas un koordinējas [DV10] lokāli, maksimizējot to savstarpējo sinerģiju, bet parādās arī globāls pašorganizēšanās un adaptācijas efekts. Izmantojot pārklājumu aģenta darba rezultātus, citi aģenti var fokusēties uz netestētiem vai maz testētiem programmatūras apgabaliem. Šādi tiek panākts sinerģētisks aģentu sadarbības efekts – pārklājumu aģentam ir izdevīgi, ka citi aģenti ar savu darbu palielina pārklājumu, jo tad viņš tuvojas sava darba izpildes mērķim – sasniegt pēc iespējas lielāku pārklājumu, ideālā gadījumā 100%. No otras puses, citiem aģentiem ir izdevīgi uzzināt, kuri apgabali ir netestēti vai maz testēti, tātad tajos, iespējams var atrast kļūdas.

Testēšanas rezultativitāti veicina arī **sacensība starp aģentiem**, kas ir kā stimuls nezaudēt darba efektivitāti. Testēšanas metriku komplektu [CPR04] var pielietot, lai nodrošinātu atgriezenisko saiti par pielietotajām testēšanas stratēģijām. Mērķis ir nodrošināt, lai testēšanas process dod gan pēc iespējas objektīvāku testējamās programmatūras novērtējumu, gan ļauj izdarīt secinājumus par pielietoto testēšanas taktiku un vai turpināt testēšanu saskaņā ar to.

Lai tiktu galā ar sarežģītībām, ko rada iekšējās un ārējās vides dažādi apstākļi, ir svarīgi ievērot **negaidītā gaidīšanas principu**. Programmatūras testēšana pēc būtības ir pilna ar gaidītiem pārsteigumiem. Katra atrastā kļūda ir gaidīts pārsteigums. Ja testējamā programmatūra nav kvalitatīva, arī katrs testkomplekts, kas nav atklājis kļūdas, ir gaidīts pārsteigums. Aģents, kas atradis kļūdu, iespējams, var turpināt darbu ar augstāku prioritāti attiecībā pret citiem aģentiem, jo tieši šī aģenta metode bija tā, kas identificēta problēmu. No otras puses, tas nozīmē, ka ir vērts atbilstošo programmatūras apgabalu testēt pamatīgāk, jo tajā, iespējams, ir vēl citas problēmas. Ja pārsteigums ir gaidīts, ir iespējams tam sagatavoties, nodrošinot nepieciešamās rezerves, lai būtu pietiekami elastīgi sistēmas aktivitāšu organizēšanā. Lai tiktu galā ar katra pārsteiguma radīto sarežģītību, nepieciešams (1) iespēju kopums, kā turpināt darbu pēc pārsteiguma, (2) zināšanas, kā pielietot šīs iespējas un kādas sekas būs šīm darbībām, (3) atgriezeniskā saite, lai kontrolētu sekas [GH05].

Iespēju kopumu var iegūt, izmantojot **dublēšanu un dažādība** visos līmeņos. Dublēšana nozīmē, ka vairāki identiski aģenti veic vai spēj veikt vienu un to pašu darbu vienā un tajā pašā veidā. Savukārt dažādība nozīmē, ka vienu un to pašu darbu veic viena tipa vairāki aģenti, bet ne identiski aģenti. Rezultātā viens un tas pats darbs tiek paveikts dažādos veidos. Dabiskas atšķirības starp cilvēkiem, kas darbojas kā viena un tā paša tipa aģenti, ir priekšrocība, jo, ja šādi aģenti testē vienu un to pašu

programmatūras apgabalu, viņi var iegūt dažādus rezultātus, kas palīdz turpmākajam darbam. Aģentu dažādība palīdz piemēroties dažādām situācijām, kā arī palīdz izvairīties no kļūdām, ko izdara paši aģenti emerģences procesā. Dublēšanu un nevienādība palīdz sistēmai darboties arī nedrošos apstākļos. Ja testētājs saslimst un tādējādi viņa aģents uz laiku pārtrauc darbu, viņa kolēģi var turpināt darbu, nodrošinot strukturāli drošu sistēmu un labu organizatorisko struktūru. Ir labāk, ja vienu un to pašu programmatūras apgabalu testē vismaz viens cilvēks-aģents un vismaz viens automatizēts aģents vai vairāki dažādi cilvēki-aģenti, nekā divi identiski automatizēti aģenti. Dažādība dod lielāku iespēju, ka tiks izlabotas kļūdas aģentu darbā, ja tādas ir, kā arī tiks atrastas kļūdas testējamā programmatūrā, jo tiks izveidoti dažādi testpiemēri.

Daļu zināšanu sistēmā ienes testētāju iepriekšējā pieredze un prasmes, daļu iegūst kā pieredzi pašas testēšanas sistēmas darbības laikā. Ir labāk kopēt automatizētos aģentus, kam jau ir iegūta pieredze, nekā katrreiz no jauna izveidot jaunu aģentu un ļaut tam izgūt savu pieredzi. Līdzīgi arī cilvēkiem-aģentiem labāk savstarpēji apmainīties ar iegūto pieredzi testēšanā, lai uzlabotu testēšanas rezultātus.

Sistēmā esošo aģentu pieredze konkrētas metodes pielietošanā padara iespējamu samazināt jaunā aģenta darba sarežģītību. Tas nozīmē, ka sistēma var uzkrāto pieredzi izmantoto turpmākajā darbībā.

Atgriezeniskās saites palīdz veidot testēšanas stratēģiju, pamatojoties uz sasniegtajiem rezultātiem [CPR04]. Nav jēgas strikti pieturēties pie iepriekšizveidotiem plāniem, ja testējamā sistēma un pati testēšanas sistēma nepārtraukti mainās. Nepieciešams ņemt vērā informāciju, kas iegūta par testējamo sistēmu – problēmas, kas ir vai nav atklātas katrā testējamās sistēmas apgabalā, to smaguma pakāpi, informāciju par testējamās sistēmas mainīto vai papildināto funkcionalitāti, u.c.. Šāda pieeja ļauj veidot testēšanas stratēģiju īstermiņā – līdz nākamajai reizei, kad situācija atkal tiks pārvērtēta. Tajā pašā laikā nepieciešams ņemt vērā testēšanas metožu savstarpējo sinerģiju, kad vienas metodes rezultātus var izmantot cita metode, uzlabojot vai optimizējot darbu.

Atgriezeniskās saites ļauj intensīvāk lietot konkrētai programmatūrai konkrētajā testēšanas posmā visefektīvākās testēšanas metodes. Ja metode ir devusi rezultātus – atradusi problēmu, tad ir vērts turpināt pielietot šo pašu metodi, jo var cerēt, ka tā atradīs vēl citas problēmas.

Atgriezeniskās saites palīdz fokusēties arī uz testēšanas vidi un procesiem nekā testēšanas rezultātiem, tādējādi radot apstākļus, kuros testēšanas var emerģēt un evolucionēt. Citiem vārdiem, ir svarīgi nodrošināt, lai testēšanas procesus pārāk stingri

neiegrožo dažādi plāni un nosacījumi, jo citādi sistēma nevarēs darboties, izmantojot visu savu potenciālu.

Iepriekš minētos komplekso sistēmu pārvaldības principus praktiķi jau ir ievērojuši un tos tiek ieteikts izmantot. Piemēram, *Kaner et. al.* [KBP02] iesaka testētājiem rēķināties, ka var būt izmaiņas testējamā programmatūrā īsi pirms termiņu beigām, ka var mainīties iesaistīto pušu dotie uzdevumi testētājiem, ka nepieciešams elastīgi mainīt testēšanas stratēģiju un plānus, adaptējoties situācijai (negaidītā gaidīšanas princips), ja testētājs netiek galā ar kādu uzdevumu, dot to citam testētājam vai nolikt malā un nedarīt vispār, nevis tērēt cilvēka laiku pie darba, kas viņās neveicas (lokāla darbība, globālas sekas), rotēt testētājus pa dažādiem testējamās programmatūras apgabaliem, testēt pa pāriem (dažādība un dublēšana), regulāri veidot atskaites par situāciju un izsūtīt iesaistītajām pusēm, nekoncentrēties tikai uz viena veida metrikām par testēšanu, veidot to komplektu (atgriezeniskās saites).

3.3. Nodaļas secinājumi

Nodaļas mērķis bija aplūkot testēšanu no sarežģītības perspektīvas, pieņemot, ka neadekvātu testēšanas rezultātu iemesls bieži vien ir augsta testēšanas sarežģītība.

Nodaļā aplūkoti programmatūras testēšanas sarežģītību veidi. Tika apskatītas testēšanas sistēmas iekšējās un ārējās sarežģītības. Lai pārvarētu testēšanas tehnoloģisko sarežģītību, tiek piedāvāts testēšanas sistēmu uztvert kā kompleksu sistēmu un kompleksu sistēmu darbības principus izmantot, lai sasniegtu labākus testēšanas rezultātus. Papildus oriģinālai pieejai testēšanai, darbā parādīti metodiski ieteikumi kā principi, kas jāievēro testēšanas procesu pārvaldībā:

- Daudzveidība – vienus un tos pašus programmatūras apgabalus var testēt gan vairāki testētāji ar vienādām metodēm, gan arī ar dažādām. Pirmajā gadījumā testēšana atšķirsies, jo katra testētāja pieredze, skatījums un prasmes ir dažādas, bet rezultātā iespējams ir samazināt kļūdu pašu testētāju darbā ietekmi uz rezultātu.
- Lokāla darbība, globālas sekas - katras testētājs dara savu darbu savā programmatūras apgabalā, bet, ja atrod kļūdu ar lielu ietekmi, tad tas var ietekmēt visu citu testētāju darbu, liekot viņiem mainīt savu darbu stratēģiju.

- Sinerģija un kooperācija – testētājiem lokāli sadarbojoties, piemēram, savstarpēji apmainoties ar informāciju par testēšanas rezultātiem, par paveikto, var notikt testētāju darba pašorganizēšanās un optimizēšanās.
- Negaidītā gaidīšanas princips – testēšanas sistēmai nemitīgi jābūt gatavai pēkšņai situācijas maiņai, piemēram, ja atrasta kļūda ar būtisku ietekmi, mainās lietotāju prasības testējamai programmatūrai, mainās laika un resursu ierobežojumi, un jāprot reaģēt uz šādiem gadījumiem.
- Atgriezenisko saišu nodrošināšana - testēšanas metriku komplektu var pielietot, lai nodrošinātu atgriezenisko saiti par pielietotajām testēšanas stratēģijām. Mērķis ir nodrošināt, lai testēšanas process dod gan pēc iespējas objektīvāku testējamās programmatūras novērtējumu, gan ļauj izdarīt secinājumus par pielietoto testēšanas taktiku un vai turpināt testēšanu saskaņā ar to.

Kompleksas sistēmas darbību nevar aprakstīt ar likumiem, tās darbības pamatā ir principi, kas tiek ievēroti, pieņemot lēmumus un izdarot izvēles. Principu mērķis ir ļaut testēšanas sistēmai samazināt tās sarežģītību, elastīgi adaptējoties jauniem apstākļiem un pašorganizējoties. Daļu no šiem principiem var realizēt, izmantojot testēšanas metodes un principus [KBP02], kas tiek lietoti praksē jau līdz šim.

Nepieciešams veikt papildus pētījumus, lai identificētu testēšanas problēmas, ko izraisa testēšanas sistēmu kompleksā daba, kā nepieciešams izstrādāt testēšanas metodes un metodoloģijas, kas palīdz šīs problēmas samazināt. Nepieciešams pētīt līdzšinējo praksē lietoto testēšanas metožu pielietošanas robežas, lai izstrādātu jaunas, pielietojot zināšanas par kompleksajām sistēmām.

Tomēr tajā pašā laikā nepieciešami pētījumi, vai un cik daudz ir pieļaujams atbalstīt testēšanas sistēmu uzvedībā neparedzamību un emerģenci, ņemot vērā gan sasniegtos testēšanas rezultātus, gan arī resursus, kas ieguldīti to iegūšanā.

Nodaļas svarīgākie rezultāti ir publicēti [AA11]. Darba autores jaunieguldījums ir idejas par testēšanas sistēmu, testēšanas sistēmu kā kompleksu sistēmu un testēšanas sistēmu komplekso sistēmu īpašību demonstrējums.

4. TESTĒŠANAS SAREŽĢĪTĪBAS IETEKMĒŠANA

Lai palīdzētu testēšanas sistēmai sasniegt optimālus rezultātus, tās pārvaldībai ir iespējams izmantot dažādas sviras, kas ietekmē testēšanas sistēmas sarežģītību.

Ļoti svarīga svira ir testēšanas procesu pārvaldības metodoloģija.

4.1. apakšnodaļā ir piedāvāts SUP modelis, kas mudina apzināt visas testējamās programmatūras lietošanā ieinteresētās puses, kā arī šo pušu intereses jau konkrēti programmatūras prasību līmenī. Apakšnodaļas rezultāti ir publicēti rakstā [AA09b].

4.2. apakšnodaļā ir piedāvāts izmantot atsevišķus pārvaldības elementus no daudzaģentu sistēmām. Apakšnodaļas rezultāti ir publicēti rakstos [AA08] un [AA09a].

4.3. apakšnodaļā ir analizētas ne-IT cilvēku iesaistīšanas iespējas testēšanā, šīs pieejas pozitīvās un negatīvās sekas. Apakšnodaļas rezultāti ir publicēti rakstā [Arn07].

4.4. apakšnodaļā ir aprakstīta pieredze ne-IT cilvēku iesaistīšanā ne tikai testēšanā, bet pat informācijas sistēmas izstrādē kā programmētājiem. Apakšnodaļas rezultāti ir publicēti rakstos [Arn00] un [Arn11].

4.1. SUP modelis un testēšana

Eksistē daudzas programmatūras izstrādes metodoloģijas, kuras detalizēti apraksta gan izstrādes procesu kopumā, gan arī testēšanas procesu, norādot tā vietu kopējā izstrādes procesā. Diemžēl daudzo metodoloģiju pieejamība vēl nenozīmē, ka tās tiek izmantotas praksē. No otras puses, industrijā "[..] programmatūras procesa uzlabošanas būtiski mērķi ir lietotāju patieso vajadzību apmierināšana, projektu novērtējumu un pārskatāmības no klientu puses uzlabošana" [OHa00]. Bet programmatūras izstrādē ir iesaistītas arī personas, piemēram, pasūtītājs un lietotājs, kas reti kad pārzina IT tehnoloģijas un izstrādes metodoloģiju

Pēdējos gados testēšanas procesā arvien biežāk iesaistās cilvēki bez specifiskām zināšanām, jo programmatūras testēšanas speciālistu trūkums arvien palielinās. Bez sagatavošanas un apmācības šie darbinieki veic nekvalitatīvu testēšanu un neapzinās savu lomu visā procesā [Arn07]. Tāpat ir arī ar pasūtītāju un galveno programmas izstrādes finansētāju, viņam arī ir nepieciešamas vienkāršotas pamata zināšanas, lai kontrolētu situāciju. Tādējādi, "modelēšanas metodikas, metodoloģijas un stratēģijas

visas palīdz vienkāršot prasību apstrādes tehnikas tā, ka tehnikas var veiksmīgi pielietot tipiski praktiķi" [CA07].

Lai atvieglotu situācijas izpratni daudzām programmatūras izstrādē iesaistītajām personām, ir izveidots pirmais konceptuālais variants SUP modelim. Šis modelis vienkāršotā veidā faktiski no putna lidojuma aplūko vairākas svarīgas lietas, kas saistās ar programmatūras kvalitāti un testēšanu. Modelis daudzās izstrādē iesaistītās personas sadala trīs lielās klasēs: Pasūtītājs (*Sponsor*) - personas, kas pasūta programmatūru, ir tās īpašnieki un finansē izstrādi, Lietotājs (*User*) - personas, kas ir programmatūras mērķauditorija un savā darbā izmantos šo programmatūru un Izstrādātājs (*Programmer*) - personas, kas izveido programmatūru. No šo personu klašu nosaukumu angļu valodā sākuma burtiem ir veidots modeļa nosaukums – *SUP modelis*.

SUP modeļa kontekstā var lietot jēdzienu *Produkts* kā „jebkurus programmatūras dzīves ciklā izveidotus nodevumus – izejas kodus, dokumentus, diagrammas, utml." līdzīgi kā [Vee02].

SUP modeļa ieviešanas pamatmērķis ir uzlabot testēšanas procesu, uzlabojot iesaistīto personu izpratni par notiekošo.

Literatūrā tiek izteikti vismaz divi dažādi skatījumi uz programmatūras kvalitāti [CF00], [Lew05], [Sol00]. Viens no skatījumiem uz programmatūras kvalitāti literatūrā un bieži vien arī praksē ir – vai programmatūra atbilst tai izvirzītajām prasībām. Tiek uzskatīts, ka prasības ir dokumentētas rakstiskā veidā. Šāda pieeja ir tipiska ārpakalpojumu projektos, jo palīdz uzturēt formālās attiecības starp pasūtītāju un izpildītāju.

Cits skatījums uz kvalitāti ir no lietotāja viedokļa – vai programmatūra atbilst tam, ko sagaida lietotājs. Programmatūra ir kvalitatīva, ja lietotājiem tā noderīga un ērta lietošanā. Šāda pieeja tipiskāka gadījumos, kad top programmatūra plašam lietotāju lokam un peļņa ir atkarīga no tā, vai lietotājs ar savu naudu balso par šo programmatūru vai arī nevēlas to pirkt.

SUP modelis iekļauj sevī abus augstāk minētos principus kā vienlīdz svarīgus un tikai modeļa izmantotājs lemj, vai ņemt vērā abus skatus vai tikai kādu vienu no tiem. Pirmais skatījums uz kvalitāti balstās uz to, ka ir dokumentētas prasības attiecībā pret programmatūru. Līdz ar to SUP modelis iekļauj *Dokumentu skatu* (SUP-D) uz visu notiekošo. Otrais princips ir grūtāk pārvaldāms, jo kvalitāte tiek mērīta ar personu izjūtām, gaidām, vīziju par topošo programmatūru. To modelī apraksta ar *Vīziju skatu* (SUP-V). Modelī tiek ņemts vērā, ka var būt atšķirības starp vienas un tās pašas personas vīziju par produktu un šīs personas uzrakstītajām prasībām dokumentos.

Testēšanas aktivitātes jāuzsāk iespējami agri programmatūras izstrādes dzīves ciklā un jāveic visās izstrādes stadijās [Per00]. Tās ietver arī plānošanas stadiju un ik pēc laika validē, vai veiktie darbi atbilst visu iesaistīto pušu interesēm. Ar SUP modeļa palīdzību var paskatīties uz procesu jebkurā tā stadijā un konstatēt aptuveno stāvokli – kuras lietas ir mazāk ņemtas vērā, vai testēšana ir pietiekoši plaša un adekvāta.

4.1.1. SUP modeļa jēdziens

SUP modelis pēc būtības ir kopa ar dažādiem skatiem uz topošo programmatūru. SUP modeļa galvenās sastāvdaļas ir to Izpildītāju – Pasūtītāju, Lietotāju un Izstrādātāju (*Sponsor*, *User*, *Programmer*) dažādie skati uz veidojamo programmatūru: vīzijas skati *SUP Vīzijas* (SUP-V), dokumentu skati *SUP Dokumenti* (SUP-D). Skati pārklājas un veido skatu sektorus. Skatu sektoriem var tikt piekārtoti svari, kas nosaka, cik nepieciešami tie ir ieinteresētajām pusēm. Katra izveidotā Produkta uzvedības nianse un īpašība atbilst kādam no sektoriem. Veicot pasākumus, kas samazina nesvarīgo sektoru apjomu un palielinot svarīgo sektoru apjumu, tiek uzlabota Produkta kvalitāte.

4.1.1.1. SUP modeļa pirmsākumi

Ideja par SUP modeli ir attīstījusies vairāku gadu laikā, meklējot cēloņus, kāpēc programmatūras testēšana bieži ir neadekvāta, nepilnīga, liekus resursus patērējoša un spontāna bez skaidra plāna. Kā galvenais cēlonis tika konstatēts fakts, ka zināšanas par programmatūras testēšanu izstrādes komandai ir viduvējas vai pat vājas. Vēl sliktāk ar šīm zināšanām ir biznesa pārstāvjiem – programmatūras Pasūtītājiem un Lietotājiem. Katram ir savs priekšstats par testēšanu un tas nereti noved pie nesaprašanās.

Tika meklēts modelis, ko saprastu visas programmatūras izstrādā iesaistītās puses un ko saprotamos jēdzienos varētu izskaidrot vienas prezentācijas vai lekcijas laikā. Sākotnējais modelis lielā mērā balstījās uz [Jor95] aprakstīto principu, ka uz programmatūras uzvedību var paskatīties no specifikācijas skatupunkta un reālās programmas skatupunkta. Vizuāli to var iztēloties kā divus apgabalus, kas daļēji pārklājas, jo parasti specifikācija atšķiras no programmas. Testētājs ar testpiemēriem meklē atšķirības starp šīm kopām. Izvēlēto testpiemēru kopa testē dažādus šo apgabalu segmentus un arī apgabalu ārpus tiem. Dažādas testēšanas metodes dažādi pārbauda programmatūras uzvedību. Ar vizuālu attēlu palīdzību (Vīnes diagrammām) lasītājam ir vieglāk saprast, ko metodes testē un ko netestē.

Izmantojot šo prezentācijas metodi, pamazām tika konstatēts, ka faktiski tas apskata testēšanu tikai no programmētāju viedokļa. Reālajos apstākļos prezentācijas

veidu vajadzēja būtiski koriģēt līdz tas tuvojās reālai dzīvei un atbilda vairākām iesaistīto personu grupām.

4.1.1.2. Izpildītāji SUP modelī un to skati

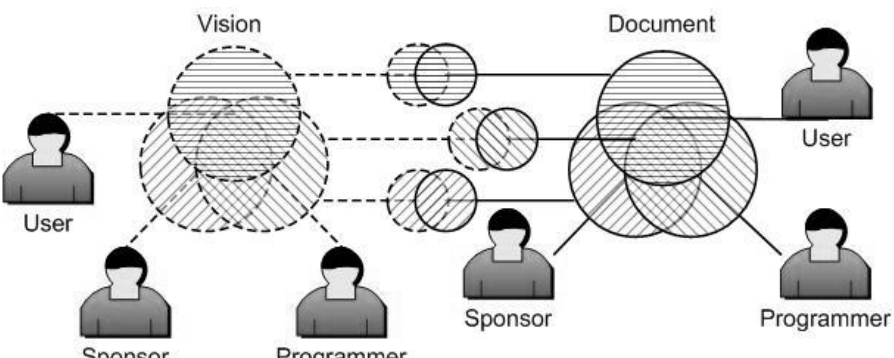
SUP modelī centrālo vietu ieņem svarīgākās programmatūras izstrādē iesaistīto personu grupas un to skats uz gala Produktu.

4.1.1.3. Izpildītāji un to klasifikācija

Programmatūras izstrādē ir izdalītas trīs svarīgas iesaistīto personu grupas (4.1. att.), kuras nosacīti varētu nosaukt sekojoši:

- **Pasūtītāji (*Sponsor*)** - personas vai to pārstāvji, kas pasūta programmatūru, finansē tās izstrādi, grib gūt no tās finansiālu labumu.
- **Lietotāji (*User*)** - personas, kas ir programmatūras mērķauditorija un kas lietos šo programmatūru darba pienākumu veikšanai vai savām personīgajām vajadzībām.
- **Programmētāji (*Producer*)** - personas, kas veic programmatūras izstrādi, kas par darbu saņem materiālus labumus vai gandarījumu.

Atsevišķos gadījumos kāda konkrēta fiziska persona vienlaicīgi atrodas vairākās grupās. Šādā gadījumā jāuzskata, ka viņa spēlē vairākas lomas vienlaicīgi. Tam ir gan priekšrocības, jo samazinās dažādo skatu skaits, bet ir arī trūkumi, jo skats var būt šaurs un kļūdains no citu personu viedokļa.



4.1. att. **Pasūtītāju, Lietotāju un Programmētāju skati (vīzijas un dokumentētās prasības) uz Produktu un attiecības starp tiem**

4.1. tabulā parādīti grafiskie apzīmējumi programmatūras līmeņiem (stāvokļiem) SUP modelī.

4.1. tabula

Grafiskie apzīmējumi programmatūras līmeņiem (stāvokļiem) SUP modelī

Apgabala veids	Nozīme
Ar pārtrauktu robežu	Rakstiski nefiksētas idejas, priekšstati par programmatūru, tās vīzija.
Ar nepārtrauktu robežu	Rakstiski noformētas prasības (prasību specifikācija, programmatūras specifikācija, u.c.).
Iekrāsots apgabals	Prasības, kas realizētas programmatūrā (programmas uzvedība un īpašības, pirmkoda īpašības, dokumentācijas īpašības).

4.1.1.4. Izpildītāju dažādie skati uz produktu

No citas puses programmatūra vai atsevišķi tās aspekti var tikt apskatīti trīs līmeņos:

Ideja, vīzija par to, kādai jābūt programmatūrai un kas tai būtu jādara. To var uzskatīt par dokumentāli nefiksētu programmatūras mentālo modeli. Turpmāk šis līmenis jeb skata tiks apzīmēts ar vārdu *Vīzija.*

Vīzijas, priekšstati, prasības rakstiskā formā. Tie ir formāli dokumenti, vēstules, diagrammas, u.c., kas ietilpst pasūtītāju prasību specifikācijā, lietotāju pieprasījumos un sūdzībās, izstrādātāju izveidotajā programmatūras projektējumā, u.c.. Turpmāk šis līmenis jeb skata tiks apzīmēts ar vārdu *Dokumenti.*

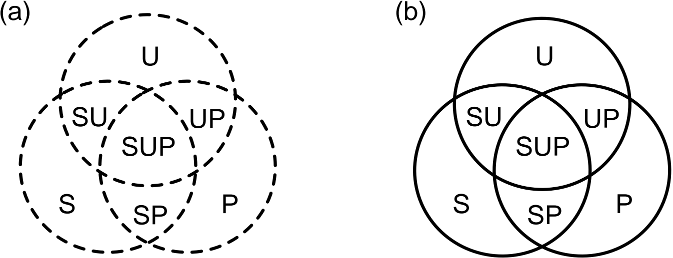
Izveidotā programmatūra, kas sastāv no darbināmas programmas, tās koda un atbalstošās dokumentācijas. Turpmāk šis līmenis jeb skata tiks apzīmēts ar vārdu *Produkts*.

4.1.1.5. Izpildītāju vīzijas

Katram no programmatūras izstrādē iesaistītajiem ir savs priekšstats jeb vīzija, kā programmatūrai būtu jāuzvedas, kādas ir tās īpašības. Jo vairāk šis priekšstats atbilst gala Produkta reālajai uzvedībai un īpašībām, jo Produkts viņu acīs ir kvalitatīvāks.

Katram Izpildītāju reprezentantam parasti ir savs viedoklis par vajadzīgo sistēmas uzvedību, kas bieži vien nesakrīt ar pārējo viedokli. 4.2.(a) attēlā parādīta dažādo vīziju savstarpējā atkarība. Iesaistīto personu kopīgo skats darba ietvaros tiks apzīmēts ar SUP-Vīzija (saīsināti SUP-V). Katram segmentam ir dots burtu identifikators, kur katrs burts simbolizē atbilstošā pārstāvja skata daļu (S – Pasūtītāji (*Sponsor*), U – Lietotāji (*User*), P – programmētāji (*Producer*). Katrs segments šeit un turpmāk darbā grafiski

reprezentē uzrakstītu (SUP-D skatā) vai neuzrakstītu (SUP-V skatā) programmatūras uzvedības un/vai īpašību prasību kopu.



4.2. att. **Skats SUP-Vīzija (a), skats SUP-Dokumenti (b) un to segmenti**

4.1.1.6. Izpildītāju dokumentētās prasības

Programmatūras izstrādes laikā tiek dokumentētas visu pušu prasības un tiek veidota programmatūra. Programmatūras izstrādātāji veido ne tikai pirmkodu, bet arī vajadzības gadījumā papildina specifikāciju, lai tā atbilstu esošajai situācijai, veido lietotāju rokasgrāmatas, palīdzības sistēmu un citu dokumentāciju.

Visus dokumentus var sadalīt nelielos elementāros apgalvojumos. Katra iesaistītā persona katram apgalvojumam var norādīt, vai tas atbilst viņas priekšstatam par programmatūru. Saklasificējot visus dokumentu apgalvojumus, var iegūt skatu uz dokumentiem *SUP-Dokumenti* (saīsināti SUP-D). Katram iegūtajam segmentam var piešķirt identifikatoru tāpat kā *SUP-Vīzija* (sk. 4.2.(b) att.). Tiek pieņemts, ka prasībai ir formāls akcepts no personas, kuras interešu lokam pieder atbilstošais segments jeb kurai šī prasība „pieder".

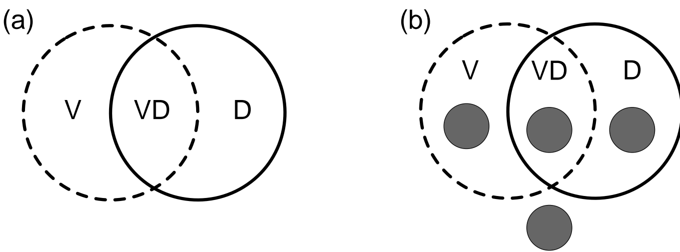
Jāuzsver, ka daļa no dokumentiem ir arī gala Produkta sastāvdaļa.

4.1.1.7. Vīziju un dokumentēto prasību saistība

Sistēmas pasūtīšanas un izstrādes gaitā daļa no katras iesaistīto pušu domām tiek fiksētas rakstiski – Lietotāji veido rakstiskas prasības, Pasūtītāji – augsta līmeņa prasības un specifikācijas, Programmētāji – programmatūras projektējumus, papildina prasību specifikāciju, programmē kodu, izstrādā lietotāju rokasgrāmatas un citu dokumentāciju. Taču ne visas vēlmes tiek izteiktas rakstiskā veidā vai var būt tikt realizētas. Situāciju var shematiski attēlot kā 4.3. attēlā. Segmenta identifikatorā ar V ir norāda uz SUP-Vīzijai atbilstošu segmentu un ar D – uz SUP-Dokumentiem atbilstošu segmentu.

Ja, piemēram, aplūko Lietotājus, tad redzams, ka viņu vīzija (apgabals ar pārtrauktu robežu - V un VD) atšķiras no rakstiski fiksētajām viņa prasībām (apgabals ar nepārtrauktu robežu- VD un D). Tām ir kopēja daļa VD, kad prasības izsaka vēlamās

programmatūras īpašības. Taču ir ļoti grūti sasniedzams stāvoklis, kad visas lietotāju vēlmes tiktu izliktas uz papīra. Neaprakstītās prasības ir segmentā V. Tāpat iespējams, ka kādas prasības tiek uzrakstītas kļūdaini – tās neatbilst reālajām lietotāju vēlmēm un vīzijai par sistēmu (segments D). 4.3. attēla (b) gadījumā redzams iekrāsots laukums ārpus abiem V un D apgabaliem. Šādi tiek parādītas prasības, kuras ir implementētas, taču neietilpst ne Lietotāja Vīzijā, ne Dokumentos. Tādas varētu kādas Pasūtītāju prasības, kas lietotājiem neinteresē, piemēram, par drošību sistēmā, vai specifiskas Programmētāju vēlmes, piemēram, speciāli veidots kods testēšanas atvieglošanai avi automatizētai testēšanai.



4.3. att. **Saistība starp Vīziju un Dokumentiem no Lietotāju, Pasūtītāju vai Programmētāju viedokļa, kur (a) gadījumā neviena prasība vēl nav implementēta, (b) gadījumā – dažas prasības no katra reģiona ir jau implementētas (apzīmētas ar iekrāsotu laukumu)**

Ja uz 4.3. attēlu raugās no Pasūtītāju vai arī no Programmētāju viedokļa, tad situācija veidojas analoģiska Lietotāju gadījumam, kas tikko tika apskatīts.

Ideālā gadījumā vīziju aplim V un dokumentu aplim D būtu jāsakrīt. Tas nozīmētu, ka visas Lietotāju, Pasūtītāju vai Programmētāju vēlmes ir rakstiski fiksētas kā prasības, turklāt tas izdarīts kvalitatīvi – uzrakstītais atbilst vēlamajam. Programmētājiem tas nozīmētu, ka viņu vēlmes un priekšstati par sistēmu ir realizēti tās projektējumā, dokumentācijā un programmatūras dzīves cikla vēlākajos posmos jau nokodētas. Turklāt tas viss izdarīts bez kļūdām un pārpratumiem. Reālajā dzīvē diemžēl šādu stāvokli sasniegt ir neiespējami gan lielo izmaksu dēļ, gan valodas neviennozīmības dēļ.

4.1.1.8. Izpildītāju skatu mijiedarbība

Atgādināsim, ka kāda daļa no priekšstatiem jeb vīzijām pārklājas visiem – ko vēlas Lietotāji, Sponsori un realizējuši vai gatavi realizēt Programmētāji (SUP). Daļa domu ir kopīgas Pasūtītājiem un Lietotājiem (SU), cita daļa kopīgas Programmētājiem un Lietotājiem (UP), vēl cita – Pasūtītājiem un Programmētājiem (SP). Katrai no

iesaistītajām pusēm parasti ir arī tāda priekšstatu daļa par programmatūru, kas nav zināma pārējiem, vai kam nepiekrīt pārējās iesaistītās puses (4.2. attēlā apzīmētas ar S pasūtītājiem, U - Lietotājiem un P - Programmētājiem).

Līdzīga aina ir arī ar dokumentētajām prasībām. Sākotnēji SUP modelis izmantoja tikai vienu skatu, kur vienāda nosaukuma segmenti tika salikti kopā. Diemžēl reālajā dzīvē, kā jau minēts, vienas un tās pašas personas vīzija dažādu iemeslu pēc var nesakrist ar dokumentētajām prasībām. Tādēļ nākas sākt izmantot vēl citus skatus.

4.1.1.9. Produkts un tā saistība ar skatiem

Līdz šim pamatā tika runāts tikai par Produkta vīziju un rakstiskām prasībām. Izstrādes mērķis ir izveidot gala Produktu, kas maksimāli atbilst visu Izpildītāju Vīzijām. Taču tikai daļa no Vīziju prasībām tiks dokumentētas, un arī no dokumentētājām ne visas tiks implementētas (4.3(b). attēlā iekrāsotie apgabali reprezentē realizēto programmas kodu, kas atbilst prasībām atbilstošajos apgabalos). Diemžēl cilvēki dažādi interpretē prasības un pieļauj arī kļūdas. Līdz ar to ir daļa no Produkta apmierina Vīziju, daļa arī Dokumentus, daļa vai abus reizē (iekrāsotais laukums apgabalā VD 4.3(b). attēlā). Daļa no vēlmēm un prasībām nav īstenotas (neiekrāsotie apgabali). Produktam ir arī uzvedība un īpašības, kas nav nedz bijušas ne vīzijās un vēlmēs, nedz arī pasūtinātas dokumnetāri, bet ir realizētas (iekrāsotais laukums segmentiem ar vārdu).

Piemēra pēc turpmāk tiks aplūkotas dažas interpretācijas Lietotāju gadījumā.

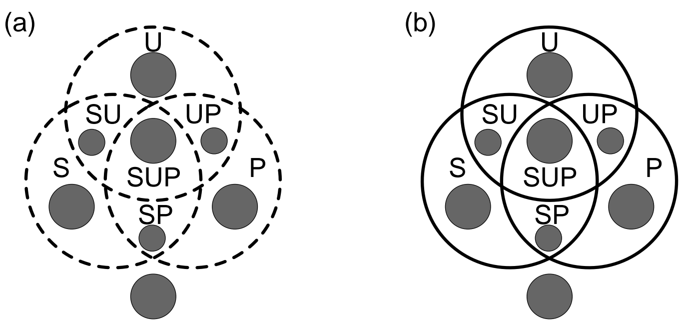
Vīzijas un Dokumentu prasību šķēluma gadījumā (apgabals VD) parādīts, ka ir prasības, kas jau ir noprogrammētas, jo ietilpst ieēnotajā apgabalā un ir prasības, kas vēl nav realizētas, jo ne viss šķēluma apgabals ir ieēnots.

Ja realizētais programmas pirmkods atbilst prasībām no apgabala V, tad tas nozīmē, ka Programmētāji ir realizējuši tādas Lietotāju vēlmes, kuras Lietotāji rakstiski nav pieprasījuši, bet kuras tomēr viņiem ir nepieciešamas un atbilst viņu vēlmēm.

Ja realizētais programmas kods atbilst prasībām no apgabala D, tad tas nozīmē, ka Programmētāji ir realizējuši kļūdainās Lietotāju prasības (rakstiskais dokuments neataino patiesās Lietotāju vēlmes).

Programmas kods, kas atbilst apgabalam ārpus V, VD un D nozīmē visu pārējo, kas ir realizēts, bet par ko Lietotājiem nav ne vīziju, ne prasību, piemēram, tās ir realizētas kādas specifiskas pasūtītāja prasības vai programmatūras uzturēšanas un testēšanu atvieglojošas prasības, vai nevienam nevajadzīga funkcionalitāte.

4.4. attēla (a) un (b) daļas kopā parāda SUP modeļa struktūru, kas ir modeļa pamats. Viens no secinājumiem, kas parasti rodas iepazīstoties ar SUP modeli, ka testēšanai ir jābūt ļoti daudzpusīgai, lai pārbaudītu dažādas lietas (4.4.(b) attēlā vien ir 16 dažādi apgabali – neiekrāsoti apgabali U, SU, UP, SUP, S, P, SP, iekrāsotie apgabali iepriekšminētajos apgabalos un kā arī neiekrāsotais un iekrāsotais apgabals ārpus nosauktajiem apgabaliem – kopā 7 + 7 + 1 +1 = 16 apgabali).



4.4. att. **SUP skati programmatūras izstrādes gaitā – (a) Vīzijas par programmatūru (SUP-V modelis) un jau implementētā programmatūra, (b) dokumentētās programmatūras prasības (SUP-D modelis) un jau implementētā programmatūra**

4.1.1.10. Skatu segmenti

Izpildītāju dažādo skatu dēļ veidojas dažādas kombinācijas – situācijas, kas tiek reprezentētas ar atbilstošu segmentu. Piemēram, 4.4.(b). attēls ataino 16 dažādas situācijas. Katram segmentam nav grūti atrast interpretāciju reālajā dzīvē. Tālāk ir dotas dažu situāciju interpretācijas. Katras situācijas atpazīšanai un apstrādei var tikt izmantotas dažādas metodes.

Ne visas situācijas ir vienādas no to nepieciešamības un lietderības viedokļa. Katra konkrētā produkta izstrādes gadījumā svari var tikt piešķirti pēc sava principa. Tomēr vienas kompānijas ietvaros, ņemot vērā tās īpatnības un kultūru, ir iespējams izveidot svarošanas principu, kas varētu derēt daudziem projektiem.

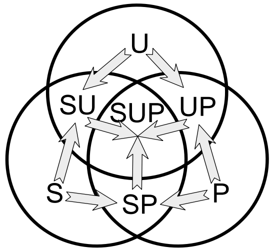
Vislielākais svars faktiski visos projektos ir SUP sektora iekrāsotais laukums. Tas nozīmē, ka Pasūtītāju, Lietotāju un Programmētāju skati par atbilstošo programmatūru uzvedību vai īpašību sakrīt un Produktā tas tieši tā arī ir realizēts.

Nākamie pēc svara ir iekrāsotie laukumi SP, UP un SU sektoros, t.i., ir realizēta programmatūras daļa, kurai trūkst tikai viena Izpildītāja atbalsta. Savstarpējās šo sektoru attiecības atkarīgas no tā, kurš Izpildītājs ir dominējošais.

Nepatīkamās situācijas, kurām jāliek mazi svari, ir neiekrāsotie laukumi sektoros – vēlamā vai pieprasītā uzvedība nav realizēta. No šīm situācijām sliktākais ir SUP sektors, jo ir vienprātība par uzvedības nepieciešamību.

Neskaidrāka situācija ir ar programmatūru, kas nevienam nav vajadzīga un, iespējams, ir radusies kļūdas dēļ (iekrāsotais laukums ārpus sektoriem).

Kad konkrēts Produkts tiek izstrādāts, dažādu reģionu svari var tikt piekārtoti pēc konkrēti izvēlētiem principiem. Organizācijā var izstrādāt arī vienotus principus kā piekārtot svarus dažādu projektu ietvaros izstrādāto Produktu SUP modeļos. To darot, ir jāņem vērā konkrētās organizācijas specifika un kultūra.



4.5. att. **Bultas norāda no apgabaliem ar mazāku svaru uz apgabaliem ar lielāku svaru.**

Kvalitātes uzlabošanas stratēģija ar sektoru palīdzību var tikt izteikta samērā vienkārši:

- Veic aktivitātes, kas atpazīst Produkta uzvedības nianses un īpašības konkrētā sektorā.
- Nosaka mērķi, t.i., sektoru, kurā vēlas redzēt uzvedību vai īpašību.
- Izvēlas ceļu no sākotnējā sektora līdz mērķa sektoram, pārejot uz sektoriem, kuriem ir kopīga saskarsme un, visticamāk, lielāks svars (var eksistēt vairāki ceļi, izvēlas attiecīgajos apstākļos optimālo, piemēram, kā 4.5. attēlā).

4. Veic aktivitātes, lai realizētu izvēlēto ceļu.

1.soli pamatā veic, izmantojot dažādas testēšanas metodes. 4.soli var realizēt, panākot atbilstošas izmaiņas Produktā vai ar skaidrošanu un pārliecināšanu mainot citu Izpildītāju skatu uz programmatūru.

4.1.1.11. SUP modeļa dinamika

Darbojoties ar SUP modeli, jāņem vērā, ka tā sastāvdaļas laika gaitā maina savu stāvokli. Pirmkārt, tas ir saistīts ar to, ka cilvēka viedoklis par lietām mainās. Tas, kas agrāk nepatika, tagad var patikt un otrādi. Ja mainījušies uzskati, kas ir dokumentēti, tad dokumentācija sāk saturēt kļūdu. Programmatūras uzturētāji, savukārt, var veikt izmaiņas, kas vairs neatbilst kādai Vīzijai vai Dokumentiem.

Atkarībā no skartā segmenta, izmaiņas var būt gan krasas, gan nelielas, gan pozitīvas, gan negatīvas. SUP modelis var palīdzēt novērtēt, cik būtiskas izmaiņas notiek, un to var izmantot, lai novērtētu, vai tās kopumā ir vēlamas, vai nevēlamas.

4.1.2. SUP segmentu interpretācija

SUP modeļa viens no galvenajiem mērķiem ir iepazīstināt Izpildītājus ar tipiskām situācijām, ko apraksta sektori, lai spētu testēšanas procesu virzīt to atpazīšanai un pēc tam veikt aktivitātes situācijas mainīšanai vēlamā virzienā.

Piemēram, 4.4. attēla (b) gadījumā parādītas SUP-D un Produkta attiecības:

- **SUP iekrāsotais apgabals** – ideāla situācija šī skata ietvaros, jo tā ir specifikācijas prasību kopa, par kuru rakstiski izdevies savstarpēji vienoties gan Pasūtītājiem, gan Lietotājiem, gan Programmētājiem, un Produkts šīm prasībām pilnībā atbilst. Šai segmentai daļai būtu jābūt pēc iespējas lielākai. No testēšanas viedokļa pietiktu, ja šai daļai atbilstu tikai neliels akcepttests formalitātēm. Lieki testpiemēri tikai dod psiholoģisko drošību, bet ir lieku resursu patērētāji, jo pēc būtības šo testpiemēru darbināšana nemaina produkta kvalitāti. Šim segmentam atbilstošie testpiemēri ir ļoti noderīgi specifikācijas precīzākai izskaidrošanai, izstrādes laikā programmētājam un noder kā piemēri lietotāja dokumentācijā.
- **SUP neiekrāsotais apgabals** – kaut kādu iemeslu pēc nerealizētais produktā. Jāizvērtē, cik daudz resursu nepieciešams, lai produkts iegūtu šo īpašību. Testēšana tradicionāli pamatā uzmanību velta tieši šim apgabalam, jo tas ir acīmredzams Programmētāju brāķis. Tomēr jāpatur prātā, ka varbūt SUP-V skatā atbilstošā uzvedība ir citā sektorā un īstenībā ir jāmaina specifikācija nevis pirmkods. Nav izslēgts, ka šādai situācijai ir kāds būtisks iemels un tas ir jānoskaidro.
- **SU iekrāsotais apgabals** – uzvedība, ko Lietotāji vēlas, Pasūtītāji pieprasa, un Programmētāji, pretēji savam oficiālajam viedoklim, ir realizējuši. Ja izdodas pārliecināt Programmētājus oficiāli akceptēt izveidoto, tad atbilstošās prasības „pārceļas” uz SUP iekrāsoto apgabalu (jāpārliecina, jo citādi Programmētāji šo uzvedību var izņemt no Produkta).
- **SU neiekrāsotais apgabals** – uzvedība, ko Lietotāji vēlas, Pasūtītāji pieprasa, bet Programmētāji to nesola un, iespējams, nav paredzējuši realizēt. Ja izdodas atrast kompromisu, projekta attīstības gaitā atbilstošās prasības no šī apgabala varētu pārcelties uz SUP apgabalu.
- **UP apgabals** - Lietotāji uzrakstījuši prasības, kuras Programmētāji gatavi realizēt, bet Pasūtītāji nav gatavi par tām maksāt vai arī tās netiek kādu iemeslu dēļ iekļautas projekta specifikācijā. Šāda situācija bieži rodas kompāniju iekšējo projektu izstrādes gaitā, kad Programmētāji cieši komunicē ar Lietotājiem, bet specifikācija netiek aktualizēta, piemēram, laika trūkuma dēļ. Rīcība atkarīga no

tā, vai attiecīgā prasība jau realizēta vai nē. Var mēģināt pārliecināt Pasūtītāju par izmaiņām specifikācijā. Iespējams, ka viss kārtībā, jo varbūt visi ir apmierināti ar stāvokli, vienīgi tas nav fiksēts specifikācijā (ir ietaupīti resursi). Taču iesējams, ka netiek precīzi realizēts kāds būtisks Pasūtītāju mērķis. Piemēram, Pasūtītājs vēlas ierobežot Lietotāju piekļuvi datiem, bet Lietotāji labāk vēlas tikt klāt visiem datiem, ko arī Programmētājam, iespējams, realizēt ir vieglāk.

- **SP apgabals** – Pasūtītāju specifikācijā norādīts, programmā realizēts, bet to neprasa Lietotāji. Šajā apgabalā var būt prasības, par kurām Lietotāji dažkārt neinteresējas un tādēļ neprasa, piemēram, sistēmas drošības jautājumi, specifiska programmatūra testēšanas atvieglošanai, kļūdu apstrāde, audita takas veidošana. Tomēr var gadīties, ka nekompetences dēļ pieprasīta/izveidota Lietotājiem traucējoša un pat kaitīga funkcionalitāte. Nepieciešams darbs ar Lietotājiem un Pasūtītājiem, lai saprastu, kā rīkoties. Pasūtītāji, protams, savu naudu var tērēt, kā viņiem patīk, bet varbūt tā ir kļūda Pasūtītāju specifikācijā.
- **S apgabals** – Pasūtītāju specifikācijā ierakstīts, bet Lietotāji to nav prasījuši un Programmētāji nav gatavi realizēt. Šeit var būt prasības, kas, Programmētājiem turpinot darbu, pārceļas uz SP apgabalu. No otras puses, iespējams, šeit esošās prasības netiks realizētas programmā, jo Lietotāji tām kategoriski nepiekrīt. Rīcība līdzīga, kā SP sektorā.
- **U apgabals** – prasības, ko Lietotāji vēlas, bet Pasūtītāji kādu iemeslu dēļ nav ar mieru tās iekļaut specifikācijā un nodot izstrādei Programmētājiem. Lietotājiem ir iespēja komunicēt ar Pasūtītājiem, lai panāktu prasību iekļaušanu specifikācijā (tās nonāk apgabalā SU), vai arī tieši ar Programmētājiem, lai panāktu prasību nonākšanu apgabalā PU un pēc tam vienotos ar Pasūtītājiem par prasību iekļaušanu specifikācijā. Diemžēl praksē bieži ir situācija, kad Lietotāji nopietni tiek piesaistīti tikai izstrādes beigās vai vispār netiek iesaistīti.
- **P apgabals** – uzvedība, kas fiksēta programmatūras projektējumā vai Programmētāju veidotos prasību dokumentos, ko nav prasījuši ne Lietotāji, ne arī tās bijušas specifikācijā, ko devuši Pasūtītāji. Iespējams, šī ir lieka funkcionalitāte. Iespējams arī, ka šī funkcionalitāte nepieciešama, bet ne Pasūtītāji, ne Lietotājs par to nav iedomājušies un tādēļ nav iekļāvuši specifikācijā. Bieži šādā statusā nonāk funkcionalitāte, kas nepieciešama sistēmu darbinošam tehniskajam personālam, lai veiktu kādas tehniskas darbības, piemēram, audita pierakstu veidošana, lai noskaidrotu, kā reāli tiek izpildītas

kādas darbības vai aprēķini. Specifikācijās parasti netiek iekļauta zema līmeņa kļūdu apstrāde, piemēram, lai ievadformas ievadlaukā Cena varētu ievadīt tikai ciparus un decimālo atdalītāju un tikai divus ciparus aiz decimālā atdalītāja.

Ja SUP modeļa apmācībā dod pietiekoši daudz reālu piemēru, tad Izpildītāji ātri vien iemācās atpazīt situācijas un domāt par veicamajām aktivitātēm.

4.1.3. Testēšanas procesa uzlabošana

Prasības ir programmatūras izstrādes projekta pamats. Testētājiem jāspēj izdarīt abas lietas – gan pārbaudīt prasību atbilstību Izpildītāju vēlmēm, gan noteikt, vai specifikācijā fiksētās prasības ir korekti realizētas programmatūras kodā. Jau paša SUP modeļa pārzināšana var palīdzēt uzlabot testēšanas procesu, jo tās ir strukturētas zināšanas par iespējamajām situācijām un ļauj skatīties uz testēšanu savādāk, nepieciešamības gadījumā pārkārtot savas aktivitātes. SUP modelis ir vienkāršots skats uz programmatūras izstrādi. Tas norāda testētājiem un Izpildītājiem virzienus kur un kā darboties, bet nerunā par testēšanu detalizētā līmenī.

4.1.3.1. Atbalsts testēšanas procesa vadīšanā

SUP modeļa izmantošana var palīdzēt uzlabot testēšanas procesu, jo strukturētas zināšanas par iespējamajām situācijām ļauj pārvērtēt testēšanas aktivitātes un atbilstoši mainīt testēšanas stratēģiju.

SUP modelis var būt rīks, kas palīdz sākt testēšanu un testēšanas plānošanu agri programmatūras izstrādes dzīves ciklā, veikt to plaši un dažādu, vadoties no Izpildītāju interesēm. Šie aspekti ir ļoti svarīgi kvalitātes nodrošināšanai [Per00]. SUP modelis atbalsta iteratīvu projekta kvalitātes pārvērtēšanu ar mērķi uzlabot testēšanas procesus. [HZJ06], [PM06], [Vee02].

Testēšana ļauj projekta vadītājiem uzzināt faktus par programmatūru – kāda veida kļūdas atrastas, cik daudz kļūdas atrastas un kurā programmatūras funkcionālajā apgabalā tās atrastas. Kvalitātes testēšana arī parāda, kāda veida kļūdas varētu vēl būt, kaut arī nav atrastas. Tas ir iteratīvs process. Katrreiz, kad programmatūra tiek testēta, jāņem vērā, vai nav piemirstas kādas izmaiņas projektā. SUP modelis dod iespēju iteratīvi analizēt situāciju programmatūras izstrādes projektā un pieņemt informētus lēmumus, lai panāktu programmatūras izstrādes procesu uzlabošanos.

SUP ļauj uzlabot programmatūras procesus, parādot projektā vājās vietas, piemēram, prasības, kuras Programmētāji pēc Lietotāju lūguma realizējuši, bet kuras nav atspoguļotas Pasūtītāja specifikācijā.

SUP ļauj racionālāk izlietot resursus, piemēram, nevis programmatūras izstrādes beigās, t.i., tipiskā testēšanai atvēlētajā laikā (izstrādes fāzē) konstatēt, ka Pasūtītāju specifikācijā izvirzītās prasības (S apgabals) neatbilst Lietotāju vēlmēm un tāpēc Lietotājiem nav pieņemamas (nonākušas apgabalā SP). Jācenšas savlaicīgi sekot, lai visas Pasūtītāja specifikācijas prasības, kas skar Lietotāju, tiktu saskaņotas ar Lietotājiem pirms Programmētāji sāk tās realizēt (vēlams ceļš pa sektoriem S-SU-SUP)

4.1.3.2. Projekta problēmu atrašana, izmantojot SUP skatus

Programmatūras izstrādes sākumā, piemēram, prasību izstrādes posmā un pirms tā, kad tiek definēti sistēmas uzdevumi (SUP-V skats), visas vēlmes atrodas apgabalos S, U un P. Testētāji var strādāt ar prasībām, piemēram, no U reģiona. Viņi var veikt lietojamības testēšanu, pārbaudīt šīs prasības ar mērķi palīdzēt Lietotājiem:

- Specificēt viņu vēlmes.
- Saskatīt, kuras prasības ir pretrunīgas.
- Sagatavoties pamatot savas prasības Pasūtītāju acīs.

Vēlāk modeļa SUP-D skats testēšanā prasa domāt par projektu kopumā:

- Kādas ir prasības, par kurām Izpildītāji ir vienojušies un par kurām nav?
- Kuras prasības jau ir implementētas, kuras vēl nav?
- Pārbaudīt, vai nav kāda papildus funkcionalitāte, kas nav precīzi pieprasīta, bet ir vajadzīga.

SUP modeļa skati dod testētājiem skaidru skatījumu uz prasībām:

- Par kādām lietam domā katrs no Izpildītājiem (reģioni S, U, P)?
- Kuras prasības tiks izdiskutētas nākotnē (reģioni SU, SP, UP)?
- Kuras prasības ir jau noprogrammētas katrā reģionā? Kuras nav? Kāpēc?
- Vai prasību kopa ir pilnīga? Piemēram, vai tiek fiksētas prasības, par ko kāda no pusēm nav īsti zinoša, bet cita nav ieinteresēta šādu prasību eksistencē. Pie tādām diemžēl pieder lietojamības prasības, kuras tiešā veidā neizriet no biznesa prasībām. Ir gadījumi, kad specifikācijā tiek norādīts, kādam jābūt maksimālajam sistēmas atbildes laikam uz lietotāja pieprasījumu. Bet reti tiek norādīts, kādiem jābūt, piemēram, minimālajiem fontiem ekrānformās, kaut tas ir būtiski Lietotājiem, jo ar sistēmu mēdz strādāt cilvēki ar dažādu redzes asumu.
- Vai dokumentētās prasības patiešām ir tās, ko Lietotāji, Programmētāji un Pasūtītāji redz savās Vīzijās?
- Vai starp dažādu pušu prasībām nav pretrunu?

Vienlaicīgi ar prasību harmonizācijas darbu starp Lietotājiem, Programmētājiem un Pasūtītājiem testētāji var sākt sagatavot lietošanas scenārijus, testu specifikācijas testēšanai nākotnē, izjūtot reālās Pasūtītāju un Lietotāju vajadzības.

4.1.3.3. Testēšanas metožu izvēle un izmantošana

Kamēr tiek strādāts ar prasībām, kuras vēl nav noprogrammētas, tikmēr testētāji savā darbā var izmantot statiskās metodes – inspekcijas, pārskatus, papīra prototipa lietojamības testēšanu.

Kopā ar citiem izstrādātājiem var veidot un nemitīgi papildināt projekta kopējo vārdnīca, prasību krātuvi.

Papildus tam testētāji gatavot testēšanas plānus, lietošanas scenārijus prasībām, kuras ir jau SUP apgabalā, domājot par tuvojošos dinamiskās testēšanas laiku.

Līdzko parādās noprogrammētas Produkta daļas, testētāji ne vien turpina līdz šim veikto darbu ar prasībām, bet arī sāk pārbaudīt prasību un koda atbilstību:

- Vai prasības noprogrammētas korekti?
- Vai nav noprogrammētās prasības, par kurām vēl nav visu Izpildītāju vienošanās?
- Vai kodā nav ieprogrammēta lieka funkcionalitāte – kas nekur nav prasīta?

Šajā laikā testētāji izmanto tradicionālās dinamiskās testēšanas metodes, lai noskaidrotu, vai prasības ir noprogrammētas, kā paredzēts.

Neparedzēti noprogrammētas prasības vai pilnīgi lieku funkcionalitāti var atrast, izmantojot strukturālās testēšanas metodes (baltās kastes metodes), pētniecisko testēšanu, ad hoc testēšanu, kā arī veicot koda pārskatīšanas.

Melnās kastes testēšanas metodes, kas balstās tikai uz Pasūtītāju rakstiskajām prasībām, darbojas SUP-D skata visos sektoros ar S burtu nosaukumā, maz pārbauda Lietotāju intereses un nespēj atrast lieko programmatūru, kas atrodas sektorā P vai ārpus visiem sektoriem.

Baltās kastes testēšanas metodes, kas izmanto pirmkodu, vienas pašas nespēj pārbaudīt sektorus S, SU un U, jo pamatā darbojas ar visiem sektoriem, kuru nosaukumā ir P burts.

Interesanti strādā statiskās testēšanas metodes, ja, inspicējot pirmkodu, vienlaicīgi tiek skatīti arī prasību dokumenti. Testēšana notiek visos sektoros, bet ir divi būtiski trūkumi – nepieciešama augsta testētāja kvalifikācija un nevar atrast specifiskas kļūdas, kas skar programmas darbināšanu (piemēram, pārbaudīt sistēmas reakcijas laiku).

Akcepttestēšana, kas ir ļoti populāra un ar kuru bieži cenšas iztikt, pamatā darbojas tikai ar sektoru SUP, kas ir nepietiekama testēšana, ja mērķis ir atrast problēmas programmatūrā.

4.2. Daudzaģentu sistēmu pielietojums sarežģītības samazināšanā

Ja iepriekšējā apakšnodaļā aprakstītais SUP modelis aplūkoja programmatūras izstrādē iesaistītos Izpildītājus, sadalot to trīs klasēs – Lietotājos, Sponsoros un Programmētājos, tad šajā nodaļā tiek aplūkota pati testēšanas sistēma kā daudzaģentu sistēma.

4.2.1. Reālās pasaules aprakstīšana ar daudzaģentu modeli

Daudzaģentu modeļi tiek izmantoti komplekso sistēmu modelēšanai. Daudzaģentu modeļi savos pamatos balstās uz reālās pasaules dažādiem principiem: aģents ir cilvēciskas būtnes analogs, aģents veic tiem uzticētās funkcijas, aģenti savā starpā sadarbojas, aģenti reaģē uz apkārtējās vides izmaiņām, aģenti dzimst, mirst, izglītojas, cenšas sasniegt gan savus mērķus, gan visas sistēmas globālos mērķus.

Darbā piedāvātā pieeja balstās uz pieņēmumu, ka ilgstošu pētījumu rezultātā ir atrastas svarīgākās lietas un principi reālajā komplekso sistēmu pasaulē, kas ļauj ar primitīvāku modeli aprakstīt veidu, kā tikt galā ar sistēmai uzdoto uzdevumu risināšanu. Uz daudzaģentu modeli balstītā programmatūra spēj inteliģenti aizstāt fizisku cilvēku daudzu uzdevumu veikšanā. Ir izmantota transformācijas shēma „reālā pasaule” → „modelis” → „programmatūra, kas balstās uz modeli”. Darbā tiek uzskatīts, ka var veiksmīgi veikt transformāciju jeb pāreju arī pretējā virzienā no „modelis” uz „reālā pasaule”, jo modelī tika iekļauti vissvarīgākie reālās pasaules principi, atmetot nebūtisko, kas reālajā dzīvē varētu būt pat traucējošs faktors. Darbā ir piedāvāts paskatīties uz testēšanas procesu kā uz daudzaģentu sistēmu.

Eksistē dažādi ietvari, kas nosaka daudzaģentu sistēmu izveides arhitektūru un principus. Lai izskaidrotu darbā piedāvātās pieejas būtību, varētu paņemt jebkuru ietvaru, kas šķiet pieņemams un saprotams, jo pamatjēdzieni un principi tajos ir līdzīgi. Darbā par pamatu ir paņemts [Aar05], [Jen00] piedāvātais ietvars, jo tas tika veidots, balstoties uz cilvēku organizācijas principiem.

Svarīgākie jēdzieni, kas tiek paturēti piedāvātajam modelim, ir aplūkoti turpmākajās apakšnodaļās.

4.2.1.1. Aģents

Viena no aģenta definīcijām ir „Aģents ir autonoma datorsistēma, kas atrodas kādā vidē un ir spējīgs uz elastīgu, autonomu rīcību šajā vidē ar mērķi izpildīt tos uzdevumus, kuru dēļ tas ir izveidots” [Aar05]. Katrs aģents ir ierobežots savās izziņas iespējās (ko var izdarīt, saprast, apgūt un ar kādu efektivitāti), fiziskajās iespējās (kādi resursi ir pieejami), laika iespējās (kāds ir aģenta dzīves ilgums, kādi ir laika limiti veicamajiem uzdevumiem) un institucionālajos ierobežojumos (ko drīkst darīt juridiski, politiski un ētiski).

Aģentam ir precīzi definētas robežas (ķermenis) un saskarne (iespēja komunicēties ar citiem aģentiem). Aģenti atrodas specifiskā vidē, ko var izzināt ar sensoriem (sensors) un mainīt to pēc saviem ieskatiem un spējām. Aģenti pilda tiem nospraustos uzdevumus, cenšoties sasniegt personīgos un visas sistēmas mērķus. Aģenti ir autonomi, kas spēj noteikt gan savu stāvokli, gan uzvedību. Aģenti spēj analizēt situāciju un risināt problēmu, reaģējot uz apkārtējās vides izmaiņām vai paši izraisot nepieciešamās vides izmaiņas, lai citi aģenti veiktu tiem uzticētos uzdevumus.

Aģenti tiek iedalīti dažās lielās klasēs pēc to izmantošanas nolūka, piemēram, var izvēlēties *Operatoru*, *Menedžeri*, *Plānotāju* un *Īpašnieku*. *Operatora* tipa aģenti veic reālo zema līmeņa tehnisko darbu, kas dod vēlamo rezultātu. *Menedžera* tipa aģenti veic organizatorisko un administratīvo darbu, pārvaldot citus aģentus. *Plānotāja* tipa aģenti veic stratēģiskās plānošanas darbus, izvēlas konkrētu risinājumu uzdevumam, ja ir vairākas iespējas. *Īpašnieka* tipa aģenti pārvalda dažādus resursus. Nav ideālas organizacionālās sistēmas, katrai situācijai ir cits optimālais risinājums, kas ir atkarīgs no konkrētajiem aģentiem (pat konkrētas aģenta eksemplāra), uzdevuma un apkārtējās vides, kas var būt gan stabila, gan ļoti mainīga un neparedzama.

4.2.1.2. Uzdevums

Uzdevums ir darbs, kas jāpaveic, lai sasniegtu globālos un arī individuālos aģentu mērķus. Augstākā līmeņa uzdevumi parasti ir tik komplicēti, ka pat labam speciālistam rodas problēmas ar tā veikšanu. Izmantojot dekompozīcijas principu, uzdevumi tiek sadalīti sīkākos apakšuzdevumos tik ilgi, kamēr iegūtie apakšuzdevumi ir pietiekami vienkārši, lai katra apakšuzdevuma izpildītājam būtu skaidrs, kā to var izpildīt. Veidojas uzdevuma dalījumu hierarhija. Starp uzdevumiem un apakšuzdevumiem var izveidot dažādas atkarības, kas veido atkarību grafu jeb tīklu. Uzdevumus papildus var grupēt pēc vēlamajiem principiem, lai tos būtu vieglāk pārvaldīt.

4.2.1.3. *Darbība*

Darba jeb uzdevuma veikšanai tiek izmantotas operācijas, kurus veic aģenti un kas darbojas ar pieejamajiem objektiem. Par objektu var kalpot dati, informācija, zināšanas, rīki, glabātuves (piemēram, datu bāzes), materiālie resursi. Tipiskās darbības ar objektiem ir to radīšana, mainīšana, iznīcināšana, patērēšana, izmantošana.

Ieteicams ir izvēlēties dažas tipiskas operāciju klases un izmantot paraugus, kā šīs operāciju klases izmantot. Operācijas atkarībā no iesaistīto aģentu uzvedības var būt gan aktīvas (aģents pats iniciē operācijas sākšanu, parūpējas par visiem nepieciešamajiem ieejas datiem un resursiem, maina apkārtējo vidi, nodod rezultātus citiem aģentiem), gan pasīvas (aģents gaida, lai citi aģenti pēc iespējas izdara vairāk – piegādā ieejas objektus, piemēram, iniciē operācijas sākšanos un paņem rezultāta objektus).

Operācijas var būt gan algoritmizējamas, t.i., tās var aprakstīt ar precīzu algoritmu, gan nealgoritmizējamas, t.i., aģents katrā konkrētā situācijā pats izdomā un realizē inovatīvu risinājumu. Datorizētā sistēmā pamatā ir tikai algoritmizējamas operācijas.

4.2.1.4. *Koordinācija un tās mehānismi*

Koordinācija var būt gan vertikāla, kad ir vadītājs un padotais, gan horizontāla, kad aģenti ir vienlīdzīgāki savās tiesībās. Koordinācija var tikt izmantota, lai sasniegtu globālos visas sistēmas mērķus, vai arī atsevišķu indivīdu – aģentu vai aģentu grupas privātos mērķus, kas var būt arī nevēlami kopējam sistēmas mērķim.

Koordinācijas aktivitātes grupē klasēs. Piemēram, *Stratēģija* ir aktivitātes, lai izplānotu visas sistēmas darbošanos mērķa sasniegšanai; *Pārraudzība* ir aktivitātes, lai kontrolētu padotos aģentus; *Kooperācija* ir aktivitātes, lai viens otram nepakļauti aģenti vienotos abiem vai sistēmai kopumā izdevīgām darbībām.

Koordinācijas tipiskie mehānismi ir tieša pakļautība (nodrošina vertikālo sadarbību, vadītājs pārvalda padotā darbu), darba standartizācija (sadarbība tiek aprakstīta ar precīziem standartiem jeb instrukcijām, kā jāveic sadarbība un kopējs darbs), savstarpēja piemērošanās (neformālā komunicēšanās).

4.2.1.5. *Organizatoriskā struktūra un saites starp tās komponentēm*

Reālajā dzīvē sistēmās pastāv organizacionālā struktūra, kas nosaka sistēmas elementu grupas un to savstarpējās „juridiskās attiecības". Kompānijā tā ir klasiskā organizacionālā struktūra ar darbinieku amatiem. Amatu ir daudz un tiem atbilstošie pienākumi ir dažādi. Amatus var grupēt klasēs („lomās"). Katrs aģents pilda kādu amatu jeb lomu. Reizēm ir lietderīgi aprakstīt arī neformālās organizacionālās struktūras.

Starp organizacionālās struktūras elementiem un objektiem var būt dažādas relācijas. Relāciju piemēri ir sekojoši: ražotājs/klients (viens aģents ražo objektus, bet cits tos patērē), kopējs ierobežots objekts (aģentiem savu operāciju veikšanai ir nepieciešams viens un tas pats objekts), ziņojums (viens aģents ziņo otram), konflikts (aģentam jāziņo citam aģentam par kāda konflikta rašanos), komanda un instrukcija (komandu un instrukciju nodošana no viena aģenta otram), deleģēšana (aģents savu atbildību un tiesības deleģē citam aģentam).

4.2.1.6. Aģenta spējas

Lai aģents paveiktu viņam uzdoto uzdevumu, viņam jāpiemīt nepieciešamajām spējām. Darbā piedāvātajai pieejai izvēlēts ilgu gadu laikā noslīpētu spēju aprakstīšanas modelis *Piecu iespēju modelis* [Aar05]. Tajā spēju tipiem ir izdalītas piecas tipiskākās grupas – *komunikācija*, *kompetence*, *pats*, *plānotāj*s un *vide*. Tās ir svarīgākās lietas, par kurām jādomā, lai aģenti būtu pēc iespējas „jaudīgāki", veicot uzdevumus sistēmā.

Komunikācija nodrošina aģenta sadarbību ar citiem aģentiem un apkārtējo vidi un nepieciešamo zināšanu uzturēšanu. *Kompetence* ir spēju kopa (zināšanas un metodes), kas ļauj veikt tiešo tehnisko uzdevuma izpildi. *Pats* nodrošina aģenta „intīmo dzīvi", t.i. aģents uztur savus mērķus, veicamos darbus, iespējas, uzrauga sevi, uztur un pilnveido sevi, vada savu darbību. *Plānotājs* paredzēts, lai aģents pats varētu izlemt par stratēģiju, kā darboties, kādā secībā un ar kādiem paņēmieniem veikt uzdevumus, t.i., plānot savu darbību. *Vides* spēju kopa nodrošina iespēju iegūt informāciju par apkārtējo vidi, citiem aģentiem un notiekošajiem procesiem.

4.2.2. Testēšanas procesa modeļa izveide

Veidojot daudzaģentu sistēmu modeļus, parasti idejas tiek meklētas reālajā dzīvē. Un meklējumu moto ir *„Kā cilvēku organizatoriskie principi var tikt izmantoti multiaģentu arhitektūrās?*". Darbā ir izvēlēts pretējais virziens, ko varētu formulēt ar moto „*Kā daudzaģentu sistēmu arhitektūra un veidošanas principi var tikt izmantoti testēšanas procesa labākai organizēšanai*?". Mērķis ir atrast pamatprincipus, kas ir it sevišķi jāievēro, ja programmatūras testēšanā dažādu iemeslu dēļ iesaistās nespeciālisti – cilvēki bez īpašām zināšanām programmatūras testēšanā.

Ir pamanīts, ka cilvēki bieži vien sarežģītās situācijās rīkojas neefektīvi. Daži nedara neko, citi aktīvi dara kaut ko, kas nav piemērots dotajā gadījumā. Retajam izdodas virzīties uz nospraustā mērķa pusi. Savukārt, ja situācija ir vienkāršāka un cilvēks izprot lietu kārtību, tad viņš jūtās komfortablāk, darbojas efektīvāk un

mērķtiecīgāk. Bez tam cilvēks par labāko rīku un metodi atzīst to, ar kuru viņš prot strādāt.

Daudzaģentu modelis piedāvā daudzas lietas vienkāršot, izmantojot dažādus zināmos paņēmienus, kā samazināt sistēmas sarežģītības pakāpi. Var izmantot paņēmienus, ko iesaka programmatūras izstrādes sarežģītības samazināšanai [Boo07]:

Dekompozīcija. Lielā problēma tiek sadalīta mazākās apakšproblēmās. Dalīšana tiek veikta tik ilgi, kamēr nonāk līdz saprotamām un vieglāk risināmām apakšproblēmām, kuras var risināt pat izolētībā no citām apakšproblēmām. Sarežģītība samazinās, jo katrs risinājuma posms ir saprotams, risinājums ir triviālāks un ir drošāk un kvalitatīvāk realizējams.

Abstrakcija. Tiek lietoti dažādi vienkāršoti modeļi, lai izceltu svarīgāko un paslēptu atbilstošajam līmenim nebūtiskās detaļas. Sarežģītība samazinās, jo var koncentrēties uz būtiskākajiem problēmas aspektiem, konceptuāli saprast notiekošo, izvēlēties labāko risinājumu vai stratēģiju, samazināt smagu kļūdu pieļaušanas varbūtību.

Organizācija. Tiek identificētas un pārvaldītas attiecības starp risinājumu nodrošinošajām komponentēm. Komponentes var tikt grupētas un uzskatītas par augstāka līmeņa universālāku un viengabalaināku komponenti. Tiek noteikti paņēmieni, kā organizēt sadarbību starp komponentēm sarežģīta kopīga uzdevuma veikšanai. Var minimizēt viena un tā paša darba dublēšanu.

Testēšanas procesa vienkāršošanai tiek piedāvāts sākt ar būtiskākā atdalīšanu no mazāk būtiskām lietām. Svarīgākā atdalīšanu veicam pakāpeniski. Pirmais solis ir aģentbāzētu modeļu izveidošana. Šie modeļi apraksta tikai svarīgākās lietas, jo visu dabā eksistējošo aprakstīt nevar. Modelis var būt neformāls, piemēram, tas var būt nekur rakstiski nefiksēts mentāls modelis, bet tas var tikt arī pierakstīts uz papīra vai datorā, izmantojot formalizētu modelēšanas vidi. Taču arī šie modeļi nespeciālistiem vēl ir par sarežģītu un piedevām tie vairāk ir domāti inteliģentas programmatūras izveidei. Otrais solis ir no aģentbāzētiem modeļiem izvēlēties nedaudzas lietas un principus, kas ir pietiekoši vienkārši gan sapratnei, gan atstāj lielu brīvības pakāpi to pielāgošanai dzīvei. Darba autores praktiskā pieredze rāda, ka, izprotot šo pieeju, var veiksmīgi pielietot pat tikai atsevišķus principus kaut arī pats modelis faktiski ir tikai vīzijas veidā.

Būtiskākie principi, kas var tikt adaptēti no daudzaģentu sistēmām un to veidošanas paņēmieniem ir sekojoši. Burtu izmantošana identifikatoros: A – aģents (no vārda angļu valodā *Agent*), T – uzdevums (*Task*), O – darbība (*Operation*), C –

koordinācija (*Coordination*), S – organizācijas struktūra (*Structure of organization*), F – funkcionēšana jeb aģenta spējas (*Functioning*).

A1: Darba darītāji ir pēc iespējas primitīvi aģenti. Aģents ir cilvēks ar savu spēju kopumu vai programmatūra, kas nepieciešama konkrēta uzdevuma paveikšanai. Aģentu piemēri: testpiemēra izpildītājs, testpiemēra rezultāta novērtētājs, testpiemēru veidotājs atbilstoši zaru pārklājuma kritērijam C1, atrastās kļūdas dokumentētājs, visu programmatūras ekrāna formu saraksta ieguvējs.

A2: Aģenti tiek grupēti tipiskās klasēs vai grupās un tiek noteiktas attiecības starp tām vai to ietvaros. Tiek nodrošināta aģentu pakļautības noteikšana (vadītājs-izpildītājs), aģenta augšējā līmeņa specializācijas noteikšana (operators, vadītājs, plānotājs, resursu pārvaldnieks), zemāka līmeņa specializācija (testēšanas rezultāta novērtētājs, kļūdu ziņojuma rakstītājs, testpiemēru veidotājs), specifisku uzdevumu veikšanas dalībnieks (regresā testēšana, scenāriju testēšana, veiktspējas testēšana).

A3: Aģenti, kas eksistē vienā fiziskā personā vai datorā, tiek apvienoti aģentūrā. Pēc principa A1 persona vai dators sevī var saturēt daudzus primitīvus aģentus. Atbilstoši šo aģentu darbība ir ierobežota – vienlaicīgi darbojas viens aģents izdalītās laikspraugas ietvaros. Var tikt simulēta vairāku aģentu paralēla darbība vienas aģentūras ietvaros.

A4: Regulāri jānoskaidro pieejamās aģentūras un tajā mītošos aģentus. Tas nozīmē, ka jābūt lietas kursā par sistēmā pieejamajiem izpildes resursiem, lai var plānot konkrētas aktivitātes (izvēloties operācijas, ko var paveikt ar pieejamajiem resursiem; meklējot iespējas iegūt jaunu aģentūru ar nepieciešamo aģentu vai veidojot jaunu aģentu eksistējošā aģentūrā).

A5: Noskaidro katras aģentūras spējas izveidot jaunus aģentus. Jāzina katra darbinieka iespējas iegūt jaunas prasmes (radīt sevī jaunu aģentu), jāzina datora un tajā instalētās programmatūras izmantošanas un konfigurēšanas iespējas vai programmatūras automātiskās adaptēšanās iespējas.

A6: Aģentūras efektīvas izmantošanas plānošana. Katrai aģentūrai ir savas darbošanās izmaksas. Zemas kvalifikācijas darbiniekus izmanto primitīvu un standartizētu uzdevumu veikšanai. Tipiskas, bieži izpildāmas un datorizējamas operācijas liek veikt datoram (testēšanas automatizācija). Nestandarta un inovatīviem darbiem jāizmanto augstas kvalifikācijas darbinieki, cenšoties tiem nedot uzdevumus, ko var veikt zemākas kvalifikācijas darbinieks.

T1: Uzdevumus skalda līdz primitīviem apakšuzdevumiem. Jābūt saskaņotībai ar principu A1. Jo primitīvāki uzdevumi un tos veicošie aģenti, jo mazāka būs sistēmas

sarežģītība (tiek pieņemts, ka sistēmas „sapratnes sarežģītība" samazinās straujāk nekā pieaug „sastāvdaļu kvantitātes sarežģītība", jo var izmantot abstrakcijas un grupēšanas piedāvātos līdzekļus). Parādās lielākas iespējas darbu optimālas izpildes plānošanai, jo vienu lielāku uzdevumu var veikt vairākas aģentūras (piemēram, viena daļa darbinieku plāno testpiemērus atbilstoši izvēlētajai testēšanas metodei, otra daļa izpilda, trešā daļa novērtē rezultātus, ceturtā veido testēšanas atskaites).

T2: Kritisko uzdevumu noteikšana. Izvērtējot riskus un dažādu iesaistīto pušu (pasūtītājs, programmētājs, lietotājs) intereses tiek piešķirtas uzdevumu izpildes prioritātes. Izvērtēšana sākas ar augstākā līmeņa uzdevumiem. Apakšuzdevumu izvērtēšana tiek veikta tikai kritiskajiem virsuzdevumiem. Tas palīdz noteikt testēšanas stratēģiju, jaunu uzdevumu radīšanas nepieciešamību un nosacījumus uzdevumu izpildes plāna veidošanai.

T3: Uzdevumu grupu ar lielu izpildes laiku atrašana. Noskaidro, kuru uzdevumu izpilde ir savstarpēji atkarīga pēc dažādiem kritērijiem, kas ietekmē kopējo izpildes laiku (piemēram, uzdevumi jāveic secīgi vai uzdevumi patērē kopīgu resursu).

T4: Uzdevumu ar ierobežotu izpildītāju skaitu noskaidrošana. Parasti ir uzdevumi, kuru izpilde prasa augstu kvalifikāciju un ir atbilstošu aģentu un aģentūru deficīts. Aģentūras ar augstu kvalifikāciju jārezervē šo kritisko darbu veikšanai, atturot tos no citu vienkāršāku uzdevumu veikšanas.

O1: Dala operāciju līdz primitīvām apakšoperācijām. Līdzīgi kā principā T1 samazina risinājuma sarežģītību un palielina iespējas manevrēt ar aģentūru izvēli katras apakšoperācijas izpildei. Vieglāk ir atsekot līdzi operācijas izpildei.

O2: Svarīgāko operāciju klašu atrašana un to darbības algoritma noteikšana. Tiek noteikti tipiski risinājumi svarīgāko un biežāk veicamo uzdevumu veikšanai. Ar paraugu palīdzību tiek aprakstīts, kā operācija jāveic. Datorizējamas operācijas vēlāk var tikt noprogrammētas un radīti programmatūras aģenti.

O3: Aizsardzība pret strupceļa situācijām. Tā kā liela daļa aģentūru ir fiziskas personas, kurām ir liela brīvības pakāpe savu lēmumu pieņemšanā, tad jāparūpējas par operācijas normālas izpildes kontroli. Praksē persona (aģentūra) var padarīt atsevišķus savus aģentus par pasīviem vai pat nedot tiem laikspraugu darbu veikšanai. Tā rezultātā kādas operācijas izpilde var pilnībā apstāties un ietekmēt arī citu aģentūru (personu) uzvedību. Aģentu, kas ir programmatūra, darbība ir vieglāk koriģējama un prognozējama.

C1: Nosaka katram uzdevumam vēlamāko koordinācijas mehānismu. Atkarībā no testēšanas procesa nodrošināšanas stratēģijas un pieejamajām aģentūrām nosaka

vēlamākos sadarbības mehānismus gan starp aģentiem, gan starp aģentūrām. Testēšanas process ir ļoti mainīgs un dinamisks atkarībā no projekta fāzes un izmantotajām testēšanas metodēm. Līdz ar to vienlaicīgi sistēmā tiek izmantoti dažādi kooperācijas varianti.

C2: Kooperēšanās veicināšana starp aģentiem un aģentūrām. Izveido kooperēšanās iespējas, uzrādot izdevīgumu to izmantošanai, nodrošina vidi, kas veicina globālo mērķu sasniegšanu, ļauj aģentūrām pašām lemt par kooperēšanos. Mērķis varētu būt pašorganizējošās sistēmas izveide, jo tās ir bieži efektīvākas un dzīvotspējīgākas kritiskos apstākļos. Nepieciešams uzmanīties, lai aģentūras neaizrautos ar privāto mērķu piepildīšanu, ignorējot visas sistēmas mērķus.

C3: Esošo kooperācijas mehānismu izmantošana. Testēšanas procesā pamatā tomēr darbojas dzīvi cilvēki un organizācijā jau eksistē konkrēti kooperēšanās modeļi starp konkrētiem indivīdiem. Nav ideālā koordinācijas mehānisma starp personām, jo katrs dod priekšroku savam vēlamajam mehānismam vai to kombinācijai atkarībā no savām personīgajām īpašībām, personas brieduma pakāpes un darbības mērķiem. Cilvēks ne labprāt pieņem straujas izmaiņas savā dzīvē un jācenšas izmantot jau eksistējošo kooperācijas modeli, pamazām to pārveidojot vēlamajā virzienā.

S1: Izvēlas vēlamāko organizacionālo struktūru katrai uzdevumu klasei. Testēšanas sistēmā ir atbilstošās organizācijas oficiālā juridiskā personu (aģentūru) organizacionālā struktūra, kas jāņem vērā. Tajā pašā laikā pastāv arī neformālās attiecības starp aģentūrām. Ir iespējams izvēlēties un pārvaldīt gan formālās, gan neformālās struktūras. Jābūt saskaņai ar principiem C1.

S2: Izmanto pastāvošās organizacionālās struktūras. Analoģija ar principu C3.

F1: Noskaidro aģentu un aģentūru svarīgākās spējas. Ir jāzina, kādi izpildītāju resursi ir pieejami, lai plānotu testēšanas stratēģiju un aktivitātes.

F2: Noskaidro svarīgākās aģentu spējas, kas nepieciešamas kritisku uzdevumu veikšanai. Jāsaskaņo ar rezultātiem, ko iegūst pielietojot principus T1, T2 un T3. Varam izdalīt iztrūkstošās spējas, kas ir starpība starp noskaidrotajām spējām principos F1 un F2.

F3: Alternatīvo spēju atrašana. Apzinās variantus, kā iztrūkstošās spējas var aizstāt ar esošajām spējām, iespējams, izvēloties citus risinājumus problēmai. Testēšanā vienu un to pašu mērķi var panākt ar dažādām metodēm un līdz ar to arī dažādām spējām. Notiek dinamiska adaptēšanās tiem apstākļiem, kādi ir konkrētajā situācijā.

F4: Jaunu spēju iegūšana. Apzina iespējas, kā iegūt iztrūkstošās spējas un nodrošina to pakāpenisku iegūšanu. Tā ir jaunu aģentu radīšana, izmantojot darbinieku apmācīšanu, iegādājoties jaunu programmatūru vai konfigurējot esošu programmatūru.

Uzskaitīto principu vienkāršotas izmantošanas stratēģija ir minēta darbā [AA08]. Principu ieviešana testēšanas darba uzlabošanai notiek pakāpeniski, evolucionējot uz labāku darbības modeli.

4.2.3. Modeļa izveide un uzturēšana

Novērojumi rāda, ka testēšana praksē bieži tiek organizēta pēc vienkārša un saprotama principa – katrai programmatūras funkcionālai daļai tiek piekārtots testētājs ar uzdevumu to notestēt. Tālākais jau ir atkarīgs, kā testētājs ar savām spējām tiks galā ar šo uzdevumu.

Šai labi saprotamai pieejai ir būtiska problēma - mazkvalificētam testētājam aktivitātes parasti aprobežojas ar vienkāršu tipisku testpiemēru darbināšanu. Šādu testētāju iemaņas ir tik nelielas un sadrumstalotas, ka plānveidīga un nopietna testēšana netiek veikta.

Lai realizētu darbā piedāvātos principusm praksē visu izšķir, vai testēšanas komandā ir vismaz viens labs testētājs un vai viņš ir spējīgs organizēt pieejamos resursus mērķa sasniegšanai. Labs speciālists var izplānot optimālu testēšanas stratēģiju, balstoties uz savām spējām. Diemžēl pietrūkst resursu, kas varētu realizēt plānus. Nevar iedot nekvalificētam testētājam augsta līmeņa uzdevumu, jo viņš nezina, kā to paveikt.

Viens no risinājumiem ir lielos uzdevumus sadalīt līdz primitīviem apakšuzdevumiem un pēc tam izplānot šo apakšuzdevumu izpildi ar visas testēšanas komandas palīdzību. Tā ir būtiski savādākā pieeja, jo uzdevumu deleģēšana vairs nav pa lieliem funkcionāliem blokiem, bet gan pa nelieliem skaidri definētiem uzdevumiem, ko darbinieks ar savām spējām var paveikt. Darbā piedāvātā daudzaģentu pieeja atvieglo pāriešanu uz šādu testēšanas organizēšanas paradigmu.

Vai ir jāveido daudzaģentu sistēmas modelis formālā veidā? Diemžēl pašreizējā attīstības stadijā atbilde drīzāk ir „nē", nekā „jā". Testēšanas komandām nav kapacitātes šī uzdevuma veikšanai (kvalificēti speciālisti, brīvs laiks). Mazos projektos modeli vieglāk uzturēt dažu vadošo darbinieku galvā neformālā veidā. Lielos projektos bez specializētiem rīkiem modeli veidot un uzturēt ir pārāk resursietilpīgi.

Tajā pašā laikā jāatzīmē, ka daļēji izveidot formālu vai pusformālu modeli ir iespējams. Testēšana sevī iekļauj gan cilvēku roku darbu, gan datoru veiktus automatizētus procesus. Atsevišķām aktivitātēm varētu tikt izstrādāti standarti un

procedūras, kā veikt uzdevumus. Testēšanas rīku izmantošana arī var tikt samērā viegli aprakstīta.

Daudzaģentu sistēmas pieejai ir priekšrocība, ka pēc vienotiem principiem var aprakstīt cilvēka un datora sadarbību jeb simbiozi kopējo mērķu sasniegšanai. Tas atvieglo gan datora operāciju aizstāšanu ar cilvēku darbu, gan iespēju formalizēt operācijas, aprakstot tās ar algoritmiem, lai spētu izveidot uz programmatūru bāzētus aģentus.

Izmantojot aprakstītos principus, var izveidoties daudzus apakšuzdevumi, kas dublējas. Piemēram, lielie funkcionālie bloki izmanto vienu un to pašu apakšfunkcionalitāti, ko cenšas notestēt katrs testētājs, bet pietiktu ar vienu reizi, jo, darbinot testpiemēru vienām vajadzībām, tas varētu testēt arī citu jomu. Lai realizētu šo pieeju, nepieciešama informācijas apmaiņa starp testētājiem, piemēram, kopējas testpiemēru bāzes starp testētājiem, un kooperācijas testpiemēru sagatavošanā un darbināšanā.

Piedāvātie principi palīdz labāk arī uzrādīt tās lietas, kas reāli jāmāca darbiniekiem. Sadalot lielu uzdevumu nelielos saprotamos apakšuzdevumos un uzrādot vienkāršu paņēmienu, kā tos atrisināt, ātrāk atklājam katra darbinieka vājās vietas un redzam, kas jāapgūst, lai darbinieks spētu tikt galā ar lielu uzdevumu viens pats.

Ieteikums testēšanas procesa plānotājam un vadītājam ir pirms katra uzdevuma deleģēšanas kādam darbiniekam vismaz galvā izveidot apakšmodeli uzdevuma veikšanai. Rakstā [AA08] ir parādīts piemērs, ka pat it kā triviāls uzdevums „Pārbaudi, vai informatīvajā sistēmā var atvērt un aizvērt visus logus” var radīt daudzas problēmas, nesaprašanās un nekvalitatīvu izpildi un neadekvātu reaģēšanu uz rezultātiem. Izveidoto apakšmodeli var daļēji pierakstīt uz papīra un iedot darbiniekam kā veicamā uzdevuma specifikāciju. Pat šāds neliels uzlabojums var jūtami uzlabot testēšanas procesa kvalitāti. Ja uzdevumu specifikācijas tiek uzkrātas, tad pamazām veidojas lielā modeļa fragmentu kopa un zināšanu bāze par to.

4.3. Ne-IT cilvēku iesaistīšana testēšanā

Uzņēmumos, kuru pamatdarbība nav programmatūras izstrāde (ne-IT uzņēmumos), trūkst profesionālu testētāju. Bieži vien pat trūkst labas kvalifikācijas IT speciālistu vispār. Viens no mītiem, pie kura pieturas daudzas kompānijas, ir, ka testēšana nav sarežģīta un grūta, to var veikt jebkurš darbinieks [Bla04]. Pat

programmatūras izstrādes kompānijas par testētājiem izmanto programmētājus-iesācējus.

Ne-IT kompānijas bieži testēšanā iesaista cilvēkus bez zināšanām testēšanā, tajā skaitā arī vispār bez zināšanām IT jomā. Tomēr ļoti bieži viņi ir topošās programmatūras lietotāji un atbilstošās biznesa jomas eksperti [CJ02], [Rac01]. Ne-IT testētāji mēģina pašizglītoties par testēšanas jautājumiem, bet tas nav viegli izdarāms, jo ir ļoti maz informācijas avotu, kas viņiem būtu piemēroti. Vairums literatūras prasa IT priekšzināšanas.

Saduroties ar problēmu, ka testēšana nedod vēlamos rezultātus, uzņēmumi ir gatavi maksāt par šo cilvēku izglītošanu testēšanas jautājumos.

Šajā darba nodaļā izmantota autores vairāk kā 10 gadu laikā industrijā iegūtā pieredze darbā ar ne-IT testētājiem un viņu apmācībā. Ir aplūkoti svarīgākie novērojumi un secinājumi attiecībā uz jautājumiem, ko ne-IT testētāji intuitīvi prot bez apmācības, ko viņiem mācīt par testēšanu, kādas testēšanas metodes viņi spēj uztvert un pielietot, kā vadīt ne-IT testētājus.

4.3.1. Ne-IT testētāji

Cilvēkiem, ko uzņēmumi izmanto kā testētājus, var būt dažāds zināšanu līmenis gan IT jomā, tajā skaitā par testēšanu, gan par kompānijas biznesu. Uzņēmumos trūkst profesionālu testētāju, tiem ir grūti piesaistīt testētāju lomā arī cilvēkus ar labām IT zināšanām par testētājiem [Hut03], kaut viņiem būtu pat vājas zināšanas testēšanā. Tāpēc kompānijas ir spiestas kā testētājus izmantot cilvēkus, kam ir labas zināšanas par uzņēmuma biznesu, bet ir ļoti vājas vai nav nekādu zināšanu par testēšanu un IT vispār. Turpmāk darbā viņi tiks saukti par par ne-IT cilvēkiem vai ne-IT testētājiem, bet cilvēki ar IT zināšanām par IT cilvēkiem vai IT testētājiem.

Ne-IT cilvēki nodarbojas ar testēšanu uz dažādu nosacījumu pamata. Dažkārt viņi tiek pieņemti patstāvīgā darbā kompānijas IT departamentā kā testētāji, citkārt testēšana viņiem ir kā papildus pienākums blakus tiešajiem pienākumiem kompānijas biznesā.

Ne-IT cilvēku priekšrocība ir dziļas zināšanas biznesa jomā, kurai tiek veidota IT programmatūra [CJ02], bieži arī daudzu gadu garumā mērāma pieredze šajā jomā. Taču parasti viņi slikti pārzina sistēmas izstrādē izmantotās tehnoloģijas, sistēmas arhitektūru, vāji orientējas testēšanas metodēs, neprot izveidot labus problēmziņojumus.

Dziļās zināšanas par uzņēmuma biznesu atļauj ne-IT testētājiem labi strādāt augstākajos testēšanas līmeņos, kur jāveic funkcionālā un akcepttestēšana – sistēmtestēšanas un akcepttestēšanas līmeņos. Ja ne-IT testētājam jāveic vienībtestēšana

vai zema līmeņa integrācijas testēšana, ātri var rasties nopietnas problēmas. Tikpat komplicēti šādiem cilvēkiem ir veikt specifiska veida testēšanu, piemēram, veiktspējas testēšanu. Regresajā testēšanā viņiem ir grūti izanalizēt, kuri programmatūras apgabali ir funkcionāli saistīti ar jauno vai mainīto programmatūras daļu, lai iztestētu arī šos apgabalus.

4.3.2. Ne-IT testētāju intuitīvais testēšanas stils

Pirms veikt apmācības katrā no organizācijām, tika intervēti gan ne-IT testētāji, gan viņu vadītāji, gan kolēģi – programmētāji, sistēmanalītiķi. Mērķis bija izprast, kādas problēmas viņiem ir un ko viņi sagaida no apmācībām. Tika apkopotas vairumam no viņiem kopīgās problēmas – gan par ne-IT testētāju testēšanas stilu, gan par prasmēm, neprasmēm, gan viņu vadīšanu.

Ja ne-IT cilvēkus neapmāca testēšanā, viņi to veic intuitīvi. Viņu testēšanas stilam ir dažas kopīgas raksturīgas iezīmes. Daži svarīgi ne-IT testētāju darba aspekti:

- **Izmanto funkcionālo testēšanu,** vairāk testē *Ad hoc* stilā – nesistemātiski, nekonsekventi, bieži sistēmu vērtē tikai no vizuālā viedokļa, pārbauda tikai standartlietošanas gadījumus, „pareizo taciņu", ar pareiziem datiem, darbības veic secībā, kā parasti tās vajadzētu veikt.
- **Baidās pārtraukt testējamās sistēmas darbu** – kaut arī intuīcija vai pieredze saka, kā varētu programmatūras darbu varētu pārtraukt, izsaucot avārijas situāciju, tomēr to nedara, lai neradītu problēmas kolēģiem un arī paši varētu turpināt darbu. Tāpēc netiek izveidoti problēmziņojumi, jo netiek precīzi izpētīta kļūmīgā situācija.
- **Ar testpiemēru saprot tikai veicamo darbību virkni, datus ne**. Testu datu plānošanu, pat ja veic, dara to pārāk primitīvi. Datus dažkārt uztver kā vienādus, kas īpaši testa rezultātu vai tā gaitu nevar mainīt, tādēļ ievadāmos, izmantojamos datus testpiemēru plānošanas gaitā nefiksē. Testa rezultātu bieži novērtē „uz aci"- „apmēram tā jau laikam bija jābūt".
- **Intuitīvi izmanto robežgadījumus, datu sadalīšanu ekvivalences klasēs**. Ja testdatiem pievērš uzmanību, intuitīvi robežgadījumus pārbauda.
- **Nespēj strādāt ar datu bāzi** – sagatavot datus datu bāzē, nolasīt datus no tabulām, tajā skaitā lietotāju saskarnēs neparādītos datus, LOG-u datus, nespēj pārbaudīt, vai ekrānā vai atskaitē redzamā informācija atbilst datiem datu datubāzē.

- **Testējot izmanto realitātei tuvus testa datus, spēj veidot biznesam raksturīgas situācijas, darbību virknes** – tā ir liela testētāju un labu pārzinātāju priekšrocība. Viņi nāk no biznesa vides, ar biznesa zināšanām. No otras puses viņiem šķiet lieki pārbaudīt programmatūras nestandarta lietošanas veidus, pieņemot, ka lietotāji zina, ko viņi dara.
- **Neatbalsta testēšanas uzsākšanu agri** - neredz prasību testēšanas nepieciešamību, viņiem ir grūtības testpiemēru veidošanā no prasībām. Viņiem ir problemātiski stādīties priekšā, kāda būs sistēma un kā šos testpiemērus patiešām varēs izpildīt. Viņi uzskata, ka sistēma jātestē un testi jāveido, kad tā uzprogrammēta, kad var to reāli redzēt un darbināt.
- **Dažkārt stereotipiska domāšana** - ja organizācijā jau lietota kāda IT sistēma, tad tā uzliek zīmogu domāšanai gan par biznesa organizāciju, gan par to, kā sistēmai jāizskatās, kas tai jādara un kā. Ir grūti kardināli mainīt pieeju, kaut reizēm to prasa gan jaunās tehnoloģijas, gan nepieciešamās izmaiņas biznesa procesos.
- **Izjūt grūtības problēmu fiksēšanā un aprakstīšanā** – nereaģē uz „jocīgu" sistēmas uzvedību, pat atkārtotu, izskaidrojot to ar dažādiem blakus faktoriem, kas tos varētu būt izraisījuši. Bieži viņi nereaģē pat, ja ievēro neparastu sistēmas uzvedību Nemēģina apzināti panākt problemātiskās situācijas atkārtošanos.
- **Neprot izveidot labus problēmziņojumus** – kodolīgus, skaidrus, precīzus, īsus, kas labi atspoguļo problēmas būtību. Ļoti reti kāds pārskata problēmziņojumu ar mērķi saprast, vai programmatūras izstrādātāji spēs atrast kļūdas, kuras izraisījušas problēmziņojumu.
- **Nezina, kā īsi un atbilstoši dokumentēt savu darbu** - nezina, kā aprakstīt savu darbu dokumentācijā, dažkārt arī baidās to darīt, jo nejūtas pietiekami kompetenti, citreiz nevēlas dokumentēšanai veltīt laiku. Ir gadījumi, kad nesaskata jēgu dokumentācijai, un tādēļ uzskata to par bezjēdzīgu laika patēriņu.
- Testēšanā **iet IT cilvēku (programmētāju, analītiķu, viņu vadītāju) pavadā** [CBG+04]**,** [May04]**.** Jūtas nedroši, neaizstāv pamatot savus spriedumus gadījumos, kad prasības ir nepilnīgas (piemēram, nav lietojamības, veiktspējas prasību). Ja sistēmas izstrādātāji dažādu iemeslu dēļ, piemēram, lai uzlabotu statistiku, uzstāj, ka ne-IT testētāju atrastā problēma nav vērā ņemama problēma, ne-IT testētāji tās bieži neuzrāda. Viņi tad apraksta tikai problēmu grupas, ko atzīst izstrādātāji, kas ir acīmredzamas jeb ļoti nopietnas. Faktiskais rezultāts – būtiskiem sistēmas aspektiem testējot netiek veltīta pietiekama uzmanība vai tie

vispār netiek testēti. Ne-IT testētāji samierinās ar to, ko sistēmas izstrādātāji ir spējuši izveidot, nevis pastāv uz savu viedokli un noformē to kā problēmziņojumu vai izmaiņu pieprasījumu. Viņi gaida, kad par līdzīgu problēmu ziņo reālie lietotāji un tikai tad ziņo par citām atrastām tāda veida problēmām.

Ne-IT testētājiem sagādā grūtības situācijas, kad nepieciešamas programmētāja vai profesionāla testētāja zināšanas, piemēram:

- Kā notestēt reālo biznesa situāciju – sistēmas ilgstošu lietošanu, servera pulksteņa un datuma pārlikšanu, piemēram, uz mēneša beigām, gada beigām.
- Kā atlasīt testpiemērus, lai testēšanas būtu pietiekami efektīva, ar vienu testpiemēru notestētu dažādus sistēmas.
- Datu bāzes aizpildīšana ar testa datiem, ja sistēmai nav atbilstošu lietotāja saskarņu to ievadam vai ja tie nāk no sistēmas ārējām saskarnēm.
- Lielāka apjoma testdatu sagatavošana.
- Testēšanas programmatūras izveidošana, uzturēšana, arī pasūtīšana.
- Ārējo saskarņu testēšana.

Ne-IT testētāji labprāt vēlas mācīties par testēšanu, ja tas nav pārāk sarežģīti un pārāk ilgi. Taču ir problēmas ar mācību materiāliem. Cilvēkiem bez IT izglītības var būt grūti izmantot esošās grāmatas un citus avotus par testēšanu, jo tie parasti prasa noteiktas priekšzināšanas datorzinātnēs, piemēram, pamatiemaņas programmēšanā.

No otras puses kompānijas vēlas ātri apmācīt ne-IT cilvēkus testēšanā, lai sasniegtu iespējami labākus rezultātus. Tādēļ nākas izmantot apmācības, meklēt cilvēkus, kas spēj piemērot apmācību materiālus viņu prasībām.

4.3.3. Ne-IT cilvēku apmācība programmatūras testēšanai

Gatavojot apmācības kursu programmatūras testēšanā, jāpiemērojas ne-IT testētāju vajadzībām un darba autore praksē ņem vērā sekojošus aspektus:

- Viņu pienākumus un atbildību programmatūras izstrādes projektā, jo pārskaitījumus, kas atrasti avotos [Bla02], [Wat01] nepieciešams koriģēt.
- Viņu problēmas, kas konstatētas interviju laikā un problēmas un par ko ir zināms, kas viņiem tās varētu būt.
- Viņu pieredzi testēšanā un projekta biznesa jomā.
- Viņu izglītību, vecumu un dzimumu.

Ne-IT cilvēki vēlas, lai apmācība testēšanā:

- Būtu bez „tehniskiem smalkumiem” – baitus, bitus, simbolu kodus labāk nemaz nepieminēt.
- Nebūtu jāiedziļinās programmatūras izstrādes metodoloģijā, uzskatot, ka tā ir vadītāju problēma, bet viņi darīs, ko viņiem liek darīt.
- Pēc iespējas atbilstu viņu līdzšinējam darba stilam, jo tad var uzskatīt, ka viņi jau visu dara pareizi, tātad nav nepieciešams neko būtiski mainīt.
- Cik iespējams, vairāk piemēru no viņu testējamās sistēmas un esošajām situācijām, lai vieglāk apgūt un apgūto sākt pielietot.
- Nerādītu neko sarežģītu, kaut tikai pēc izskata sarežģītu, pat ja tas būtu efektīvs lietošanā.
- Neprasītu izmantot neko tādu, kam varētu būt sakars ar programmēšanu vai programmatūras izstrādes tehnoloģijām.

Ko var mācīt par testēšanu ne-IT cilvēkiem, kādas prasmes un zināšanas ir relatīvi vienkārši apgūstamas, kas ļaus viņiem izmantot savu stipro pusi – labu biznesa pārzināšanu?

- **No prasībām veidot lietošanas gadījumus** – ne-IT testētāji šeit var izmantot savas plašās zināšanas par biznesu, tā procesiem, likumsakarībām, atkarībām. Labi noder zināšanas modeļu zīmēšanā, cēloņu-seku diagrammu [Bei90], [Vee02] izmantošanā, kā pamatot un dokumentēt savu testa scenāriju izvēli.
- **Robežgadījumu un ekvivalences klašu lietošana testa datu izvēlē** – intuitīvi pielietotās metodes papildinot ar zināšanām, var vienkārši iedot testētājiem spēcīgu ieroci labu testpiemēru veidošanā.
- **Testpiemēru skaita samazināšanas metodes** - zināšanas par ekrānformu elementu grupēšanu pēc atkarībām, par ortogonālo masīvu, lēmumu tabulu, kombināciju analīzes izmantošanu.
- **Testpiemēru pierakstīšana** – kā pierakstīt testa gaitu, ievadāmos datus, sagaidāmos rezultātus.
- **Zināšanas par lietojamības prasībām** – kādam vēlams būt elementu izvietojumam, izskatam ekrānformās un atskaitēs, kāda ir nevēlama prakse, populāras kļūdas. Šīs zināšanas ļauj testētājiem būt pārliecinātākiem par sevi, dod iespēju pamatot savus uzskatus, pretoties izstrādātāju spiedienam. Dažkārt uzņēmumā ir savas vadlīnijas, kādām ir jāizskatās lietotāju saskarnēm, tad testētāji var izvērtēt, vai šīs vadlīnijas ir kvalitatīvas un pieņemamas.
- **Risku analīze** [Per00], [Vee02] – ļauj efektīvāk organizēt savu darbu ierobežota laika un resursu apstākļos, testēt vissvarīgākās prasības, veikt viskritiskākos

testus. Dažkārt testētāji nevēlas ar to nodarboties, uzskatot, ka tā ir vadītāju kompetence, ka tā prasa no viņiem pārāk daudz lieka birokrātiska darba.

- **Pamatiemaņas SQL vaicājumu veidošanā** – dod iespēju testētājam skatīties datu bāzē datus. Sarežģītos gadījumos labāk, ja vaicājumus izveido kāds no izstrādes grupas, bet testētāji prot tos palaist sev nepieciešamā brīdī. Piemēram, ar šādu vaicājumu palīdzību testētāji var pārbaudīt sistēmas LOG ierakstus.
- **Par testētāju lomu projektā, par testēšanas mērķiem** [Bei90], [KBP02], [Per00] – rezultātā testētāji labāk apzinās savu uzdevumu atrast kļūdas, nevis pierādīt, ka sistēma strādā nevainojami. Apzinās savas iespējas – kāda veida kļūdas viņu spēj atrast, kādas ne. Viņi zina, ka viņi nespēj nodrošināt programmatūras kvalitāti un viņiem par to nav nevienam jāatbild [Hut03].
- **Problēmziņojumu veidošana** – kam jābūt tajos, kādiem tiem jābūt. Vēlams atgādināt arī par vienkāršākajām problēmu fiksēšanas metodēm, piemēram, kā iegūt ekrāna attēlu, kā ar grafisko redaktoru izgriezt no attēla vajadzīgo daļu, kā ielikt attēlu problēmziņojuma dokumentā.
- **Reāli piemēri** – mācībās vislabāk izmantot piemērus no sistēmas, kas jātestē – prasības, ekrānformas, atskaites. Mācībās tad tiek izmantoti testētājiem pierastie jēdzieni, biznesa termini, lai vieglāk ir atcerēties mācīto. Pat tad, ja izmanto materiālus no jau notestētas un ražošanā atdotas sistēmas daļas, dažkārt visiem ir pārsteigums, cik daudz nepamanītu lietu ir paslīdējis garām.
- **Metrikas** [Bei90], [CPR04], [Per00] **-** kā mērīt savu un komandas darba apjomu, sava un komandas darba pārklājumu, kā saprast, ka ir problēmas testu plānošanā un izstrādē.
- **Testētāju grupas vieta projekta komandā, testēšanas un komandas darba psiholoģiskie aspekti** [CBG+04], [Hut03], [KBP02], [May04] – testētājiem jāzina programmatūras izstrādes komandas iekšējās organizācijas priekšrocības un trūkumi. Jāpārzina psiholoģiskā klimata efekti gan testēšanas komandā, gan programmatūras izstrādes komandā.

4.3.4. Ne-IT testētāju darba vadīšanas īpatnības

Visiem testētājiem ir daudz kopēju vadības problēmu, neatkarīgi no tā, cik viņi zina par IT. Ne-IT testētāju vadītāji tomēr var nokļūt īpašā situācijā dēļ fakta, ka ne-IT testētāji izjūt lielāku pazemību, kad saskaras ar programmatūras izstrādātājiem. Tas ir

tādēļ, ka viņu nejūtas droši IT jomā. Tomēr daļa problēmu ir tādas pašas, kā IT testētāju vadītājiem [Ash03], [Bla99], [Bla02], [Hut03], [KBP02], [KFN99].

Testētājam jājūt, ka viņa vārdam ir autoritāte un ka viņa darbs ir vajadzīgs. Apmācībās mēdz atklāties lietas, ko acīmredzot ikdienā testētāji neuzdrošinās pateikt saviem vadītājiem – ka par viņu darbu pārāk maz interesējas, ka viņi jūtas pamesti, ka netiek ņemti vērā viņu darba rezultāti, ka viņiem ir doti norādījumi kāda veida problēmas neuzrādīt, ka viņi kaut ko savā darbā neprot vai nesaprot, bet nezina, pie kā iet pēc palīdzības. Tādēļ vēlams, ka testētāju apmācībās ir klāt arī viņu vadītāji.

Vadītājam ir ļoti uzmanīgi jāseko līdzi testētāja izmantotā laika sadalījumam. Ne-IT testētājam vismaz sākumā nepieciešams papildus laiks, lai pārdomātu, izprojektētu testus. Ja laika ir par maz, testēšana reducējas uz funkcionalitātes testēšanu, izpildot standarta lietošanas gadījumus. Ilgstoši šādi strādājot testētājam sāk šķist, ka ar to testēšanai arī pilnīgi pietiek. Piemēram, pārbaudot kļūdas labojumu, tiek izpildīts tikai tests, kas atrada šo problēmu, un netiek domāts par blakus efektiem, ko šis labojums varētu radīt.

Ir bīstami, ja ne-IT testētāji ir pakļauti izstrādes grupas vadītājam. Bieži ne-IT testētāji pakļaujas IT cilvēku viedoklim, neuzstāj uz savu pieredzi un viedokli, kādai jābūt sistēmai. Ja vadītājs kaut kādu iemeslu dēļ nav ieinteresēts, ka kļūdas tiek uzrādītas, piemēram, lietojamības kļūdas, kas radušās sistēmas vājas projektējuma kvalitātes dēļ un kuras tagad būtu daudzējādā nozīmē dārgi labot, tad testētājiem tiek apgalvots, ka šīs kļūdas nav nemaz kļūdas un par tām problēmziņojumi nav jāveido, ka tās nav jāreģistrē. Sekas – testētājs mierīgi gaida, kamēr par šīm problēmām ziņos jau lietotāji, jo lietotāju viedokli vadītājs tomēr respektē.

Gan izstrādātājiem, gan testētājiem nepieciešams veidot arī savus iekšējos problēmu reģistrus, ja oficiālais kļūdu paziņojumu saraksts tiek mākslīgi ierobežots, gan izstrādātājiem, gan testētājiem nepieciešams veidot arī savus iekšējos problēmu reģistrus, kaut parastas elektroniskas tabulas veidā. Testētājiem šāds reģistrs palīdz atsaukt atmiņā arī tos gadījumus, kad bija aizdomas par kļūdu, bet neizdevās to atkārtot vai lokalizēt pietiekami, lai izveidotu problēmziņojumu. Paturot prātā šos aizdomīgos gadījumus, var efektīvāk šīs kļūdas atklāt vēlāk vai citās situācijās, kad tās atkal izpaužas.

Gan izstrādātājiem, gan testētājiem nepieciešams veidot savus iekšējos problēmu reģistrus. Dažkārt projektu vadītāji oficiālo kļūdu paziņojumu sarakstu mākslīgi ierobežo, norādot, ka kāda tipa kļūdas tajā nav nepieciešams reģistrēt. Piemēram, parasti viņi uzstāj, ka nav jāreģistrē grūti atkārtojamas kļūdas. Citreiz

norāda, ka nav jāreģistrē neliela smaguma kļūdas, pamatojot to ar apsvērumu, ka ir pārāk dārgi tās labot un tāpēc nav jēgas reģistrēt, jo netiks labotas. Ļoti bieži šādā statusā nokļūst lietojamības kļūdas lietotāju saskarnēs, jo izstrādātāji apgalvo, ka „testētāji jau nezina, ko lietotājiem vajag" vai „lietotāji ir apmierināti ar šādu saskarni". Citreiz laika trūkuma dēļ izstrādātāji dod testēt programmatūru pat ar zināmām kļūdām tajā. Šajos un līdzīgos gadījumos gan izstrādātājiem, gan testētājiem nepieciešams veidot arī savus iekšējos problēmu reģistrus, kaut parastas tabulas veidā. Testētājiem šāds reģistrs palīdz atsaukt atmiņā arī tos gadījumus, kad bija aizdomas par kļūdu, bet neizdevās to atkārtot vai lokalizēt pietiekami precīzi, lai izveidotu problēmziņojumu. Paturot prātā šos aizdomīgos gadījumus, var efektīvāk šīs kļūdas atklāt vēlāk, kad tās atkal ir parādījušās. Izstrādātājiem šāds saraksts ir kā atgādinājums par kļūdām, kas ir jāizlabo, cik iespējams, drīzāk.

Jāveido vārdnīca – lai programmatūras izstrādes cilvēki un no biznesa puses nākušie testētāji „runātu vienā valodā"- lietotu vienus un tos pašus jēdzienus vienā un tajā pašā nozīmē.

4.3.5. Reālu projektu testēšanas piemēru analīze

Autore piedāvā aplūkot trīs reālus gadījumus no prakses, kā ne-IT testētāji tika iesaistīti programmatūras testēšanā.

4.3.5.1. Projekts I

Programmatūras izstrādes projekts notika valsts finansu institūcijā, kurā strādāja ap 500 darbinieku. Organizācijai bija centrālais ofiss un dažas filiāles visā Latvijā. Finansu institūcija nopirka programmatūru un pēc tam adaptēja to divu gadu laikā. Sākotnēju adaptācijas darbu veica trīs cilvēku grupa, vēlāk tā tika paplašināta līdz 12 cilvēkiem, ieskaitot septiņus sistēmanalītiķus un programmētājus, kā arī divus biznesa cilvēkus, kas noteica precīzi organizācijas prasības programmatūrai, kā arī trīs konsultanti no kompānijas, kas izstrādāja un pārdeva programmatūru.

Pirmos testēšanas līmeņus veica izstrādes grupa. Ne-IT testētāji tika iesaistīti sistēmtestēšanā un akcepttestēšanā.

Pašā projekta sākumā no katra organizācijas departamenta tika izvirzīti divi pārstāvji jeb pilnvarotie lietotāji, kuru uzdevums bija konsultēt izstrādātājus par to, kādas programmatūras funkcijas nepieciešams attīstīt. Vairumā gadījumu lietotāju pārstāvji bija ne-IT testētāji – bez jebkādas izglītības IT jomā vai programmatūras testēšanā.

Lietotāju pārstāvjiem bija divi uzdevumi. Izstrādes laikā viņiem bija pieeja testēšanas videi un viņiem bija uzdots notestēt visas operācijas, ko viņi vēlāk varētu veikt ikdienas darbā, tiklīdz šīs operācijas tika izstrādātas un pievienotas testēšanas videi. Tā bija viņu veiktā sistēmtestēšana. Otrais uzdevums bija akcepttestēšana – dažu darba dienu ilgumā viņi testēšanas vidē veica visas savas ikdienā veiktās darbības, kā arī testēja mēneša un gada beigās veicamās specifiskās operācijas.

Lietotāju pārstāvjiem bija apmācības par sistēmas lietošanu, bet viņi netika apmācīti testēšanā.

Sākotnēji lietotāju pārstāvji sistēmas lietošanā bija ļoti pasīvi. Kad departamentiem tika informācija par statistiku – cik notikušas pieslēgšanās, cik laika pavadīts sistēmā, tad testētāji sāka iemēģināt programmatūras lietošanu. Kad mazliet to bija apskatījuši, pārstāja tajā ko darīt, taču neizgāja no sistēmas, cerībā tādējādi uzlabot statistiku par sistēmā pavadīto laiku.

Kad tika darīta zināma statistika arī par sistēmā darbinātajām ekrānformām, atskaitēm, cik ilgu laiku ar katru no tām strādāts, sākās reāls darbs, bet ne iecerētajā veidā. Pilnvarotie lietotāji testēja funkcionalitāti tikai tad, kad to palūdza izstrādātāji.

Lietotāju pārstāvji neveica testēšanu paralēli viņu ikdienas darbam. Tā vietā tika apmaksāts viņu darbs nedēļas nogalēs un brīvdienās, kuru laikā viņi veica akcepttestēšanu.

Rezultāti. Pilnvarotie lietotāji kā ne-IT testētāji bija pasīvi, kamēr netika parādīts, ka viņu darbs tiek reģistrēts, un netika organizēta plānveida vispārēja testēšana. Viņi minēja vairākus iemeslus, pirmkārt, viņiem tāpat bija jāpadara visi ikdienas pienākumi papildus testēšanai, otrkārt, trūcis zināšanu, kā īsti veikt testēšanu.

Taču ne-IT testētāji bija neaizstājami komplicētu funkcionālo testu veidošanā un to rezultātu novērtēšanā kā orākuli, lai saprastu, vai testi veiksmīgi izieti, vai nē.

Kad tika atrastas problēmas vai radās jautājumi, ne-IT testētāji tos nosūtīja pa e-pastu izstrādātājiem, kuri izpētīja situāciju un veidoja problēmziņojumus.

Ne-IT testētāji izvairījās strādāt ar situācijām, kas reizēm ikdienā gadās, ir pieļaujamas, bet netipiskas, piemēram, no specifisku datu ievadīšanas.

Tomēr testētāji atrada vairākas funkcionālas problēmas un pēc sistēmas ieviešanas ekspluatācijā palīdzēja kolēģiem tās lietošanas apgūšanā. Tas uzlaboja sistēmas apguves ātrumu un kvalitāti.

4.3.5.2. *Projekts II*

Nākošais gadījums bija kompānijā, kurā strādāja 1500 darbinieku. Tai bija centrālais ofiss, 30 filiāles, kā arī daudz mazu pārstāvniecību visā Latvijā. Arī šeit programmatūra tika nopirkta, bet pēc tam adaptēta kompānijas vajadzībām tās IT departamentā. Izstrādātāji veica vienībtestēšanu un daļu no integrācijas testēšanas. IT departamentā bija neatkarīga testēšanas grupa, kura veica sistēmtestēšanu un plānoja akcepttestēšanu, kuru reāli veica lietotāji – ne-IT testētāji.

Kad programmatūra jau bija tuvu nodošanai ražošanā, no katras kompānijas filiāles un departamenta tika aicināti pieteikties 1-2 brīvprātīgie lietotāju pārstāvji. Viņiem kā ne-IT testētājiem tika veiktas apmācības sistēmas lietošanā un testēšanā. Viņiem tika veikta apmācība vienas darbadienas apjomā par to, kā veikt testēšanu.

Ne-IT testētājiem bija pieejama testēšanas vide. Sākotnēji viņi iepazinās ar to un sistēmu, spēlējoties ar to un konsultējoties ar izstrādātājiem. Pēc tam notika plānveida testēšanas aktivitātes. Izstrādes grupas sastāvā esošie profesionālie IT testētāji bija sastādījuši uzdevumus jeb lietošanas gadījumus, kas katram no ne-IT testētājiem jeb lietotāju pārstāvjiem jāveic. Visi uzdevumi kopā nodrošināja sistēmas prasību pārklājumu. Ne-IT testētājiem bija ieteikts papildināt šos uzdevumus ar gadījumiem no savas pieredzes, izvēlēties testdatus un aktivitātes pēc saviem ieskatiem.

Daļa uzdevumu bija jāveic konkrētā datumā un laikā visiem reizē – tika pārbaudīta sistēmas spēja darboties ar realitātei tuvu noslodzi – daudziem pieslēgumiem, daudziem pieprasījumiem vienlaikus. Lietotāju apmācības process tika izmantots, lai testētu sistēmas veiktspēju.

Ar brīvprātīgo lietotāju palīdzību tika pārbaudīta programmatūras reakcija uz noslodzi, kuru simulēt ir sarežģīti un dārgi.

Rezultāti. Brīvprātīgie lietotāji pēc sistēmas nodošanas ražošanā palīdzēja kolēģiem apgūt sistēmas lietošanu. Pārdomātā testu organizācija ļāva pārbaudīt vairākus testēšanas aspektus vienlaicīgi. Taču ne visi no ne-IT testētājiem darbojās vienlīdz labi un motivēti. Brīvprātīgo lietotāju aktīvākā un rezultatīvākā (no problēmu atrašanas viedokļa) daļa vēlāk regulāri piedalījās programmatūras kārtējo versiju testēšanā.

4.3.5.3. *Projekts III*

Trešais projekts bija kompānijā ar daudzām filiālēm. Kompānija nolēma piesaistīt ārpakalpojumu sniedzējus programmatūras daļas izstrādei. Uzņēmuma iekšējā izstrādes komandā bija programmatūras izstrādātāji, sistēmanalītiķi un testētāji. Testētāji bija tieši pakļauti programmatūras izstrādes grupas vadītājam.

Programmatūra tika izstrādāta vairāku gadu garumā, kuru laikā daļa no tās izstrādātāju grupas cilvēkiem bija nomainījušies. Procesa beigās izrādījās, ka izstrādes grupā ir palikuši tikai daži cilvēki no tiem, kas uzsāka darbu tā sākumā.

Testētāji tika piesaistīti, kad programmatūras izstrāde tuvojās beigām. Tie bija ne-IT testētāji, kas bija jau strādājuši kā pilna laika testētāji jau vecajai programmatūrai. Dažiem no viņiem bija 10 un vairāk gadu ilga pieredze testēšanā, taču viņi visi testēšanā bija nonākuši no lietotāju puses un nekad nebija apmācīti testēšanā.

Programmatūras izstrādes procesā bija iesaistīti arī divi pilnvaroti lietotāji no katra organizācijas departamenta. Viņu pienākums bija palīdzēt izstrādātājiem, ja radās jautājumi par uzņēmuma biznesa procesiem, kā arī apstiprināt sistēmas artefaktus – programmatūras prasības, prototipus, ekrānu formas, atskaišu formas, u.c. Izstrādātāji veica vienībtestēšanu, bet pilnvarotie lietotāji – akcepttestēšanu. Pārējā testēšana bija ne-IT testētāju rokās.

Testētāji brīvi lasīja daļēji formalizētā valodā pierakstītas sistēmas prasības, veidoja testēšanas plānus, testu projektējumus, testu scenārijus. Testēšanu nācās veikt ļoti ātri, jo izstrāde paņēma vairāk laika nekā bija paredzēts un pilnas sistēmas prezentācijas termiņš strauji tuvojās.

Programmatūrā bija ļoti maz iestrādātu iespēju, kas palīdzētu testētājiem vieglāk tikt galā ar savu darbu, piemēram, nebija ierakstu LOG tabulās vai iespējas kādā veidā iegūt dažādu garu aprēķinu starprezultātus. Šāda situācija bija radusies tāpēc, ka programmatūras prasībās bija uzrādītas tikai funkcionālās vajadzības, bet personāla kadru mainības dēļ programmatūras izstrādātāji ļoti stingri ievēroja programmatūras prasībās noteikto.

Ne-IT testētājiem tika organizētas apmācības testēšanā, kuru ilgums pārsniedza 40 astronomiskās stundas. Tās uzlaboja testētāju kvalifikāciju, savstarpējo sapratni starp testētājiem, izstrādātājiem un vadītājiem, taču darba kultūras maiņa un procesu maiņa notika lēnām un strauji uzlabojumi testēšanas kvalitātē nebija.

Testētāji bija raduši lietot kā testa datu bāzi ražošanas datu bāzes kopiju, nevis veidot savus testa datus. Viņi uzskatīja, ka ražošanas datu bāzē jau satur visus nepieciešamos testēšanai gadījumus. Datu apjoma un konfidencialitātes dēļ, šoreiz reālie dati nebija pieejami un ne-IT testētājiem testa datu sagatavošana sagādāja nopietnas problēmas. Sistēma dažus svarīgus datus saņēma no ārējām sistēmām un tos nevarēja ievadīt, izmantojot lietotāju saskarnes, bet nekādi palīgrīki, lai tos ievadītu datu bāzē vai simulētu saskarnes ar ārējām sistēmām, arī nebija izveidoti.

Rezultāti: Laika trūkuma dēļ izstrādātāji nevarēja pietiekami atbalstīt testētājus, bet testētāji pamatā koncentrējas uz funkcionalitātes svarīgākajiem testiem.

Testētājiem bija grūti izveidot testa datus, kuru ievadam nav lietotāja saskarnes. Testētājiem ir problemātiski notestēt sistēmas ārējās saskarnes.

Nepietiekamu tehnisko zināšanu un prasmju dēļ testētājiem bija grūtības veikt zema līmeņa integrācijas testēšanu. Pamatā testētāji veica funkcionālo sistēmtestēšanu.

Apmācību rezultātā testētāji spēja noteikt prasības programmatūrai, kuru izpilde nepieciešama testēšanas darba atvieglošanai. Apmācības uzlaboja testētāju kvalifikāciju, savstarpējo sapratni starp testētājiem, izstrādātājiem un vadītājiem, taču darba kultūras maiņa un procesu maiņa notika lēnām un strauji uzlabojumi testēšanas kvalitātē nebija. Kompānijā tika uzsākts nopietns darbs, lai reorganizētu visu testēšanas procesu kopumā.

4.3.6. Ieguvumi no projektu piemēru analīzes

Praksē tika secināts, ka vislabāk ne-IT testētājus var izmantot kombinācijā ar profesionāliem IT testētājiem. Tad viņi papildina viens otra kompetences. Ne-IT testētāji veiksmīgi var testēt sistēmas atbilstību biznesa procesiem, programmatūras lietojamību, lietotāja saskarnes– ekrānformas un atskaites.

Vēlams, lai ne-IT testētāju darbu atbalstītu izstrādātāji, palīdzot darbā ar testdatu izveidošanu datu bāzē, SQL vaicājumu veidošanu un citos jautājumos, kur nepieciešamas specifiskas IT zināšanas.

Ne-IT testētāji var izveidot realitātei atbilstošus lietošanas scenārijus, testdatus, ņemot vērā arī robežgadījumu un ekvivalences klašu analīzi. Izstrādātājiem to paveikt ir grūti, jo viņi sliktāk pārzina biznesa apgabala nianses. Ne-IT testētāji var veidot vienkāršus SQL vaicājumus datu bāzē, lai pārbaudītu datus un pētītu LOG tabulu ierakstus.

Ne-IT testētāji var labi novērtēt sistēmas lietotāju saskarņu lietojamību – vai darba vietas ir ērtas, vai visas nepieciešamās funkcijas var izpildīt, vai sistēma lieki nenoslogo redzi, atmiņu, vai nav grūti lietojama.

Ne-IT testētāju apmācības testēšanā ne vienmēr būtiski uzlabo testētāju darba rezultātus. Piemēram, laika trūkuma apstākļos viņiem nav iespējams izmantot savas zināšanas pietiekami labi.

SUP modelis labi pamato ne-IT testētāju iesaistīšanas testēšanā lietderīgumu – viņi pārzina biznesu, pārzina testējamo programmatūru un var ļoti labi novērtēt, kuras no prasībām ir kuras ieinteresētās puses kompetencē, kuras prasības programmatūrā ir

realizētas atbilstoši biznesa cilvēku vajadzībām, bet principi, kas aizgūti no multiaģentu sistēmu modelēšanas pieredzes, palīdz veiksmīgāk organizēt ne-IT testētāju darbu.

4.4. Galalietotāju iesaistīšana izstrādē un testēšanā

Šajā darba apakšnodaļā aprakstīta autore praktiskā pieredze ne-IT cilvēku iesaistīšanā ne tikai programmatūras testēšanā, bet arī tās izstrādē.

Ir daudz organizāciju, kurām ir nepieciešams saglabāt un analizēt datus, kurus tās iegūst kā tabulveida dokumentus. Arī atskaites, ko veido no šiem datiem ir tabulu formā. Tā kā biznesa vide un lietotāju prasības mainās ļoti strauji, atbilstoši bieži mainās prasības biznesu atbalstošajām informācijas sistēmām. Nepieciešams ļaut ievadīt uzņēmumu informācijas sistēmās jaunu veidu dokumentus ar datiem, kā arī ģenerēt no tām citu veidu atskaites, lai palīdzētu pieņemt biznesa lēmumus. Uzturēt nemitīgi mainošos un attīstošos programmatūru ir dārgi. Tādēļ tiek piedāvāts pēc iespējas vairāk programmatūras uzturēšanā iesaistīt gala lietotājus.

Lietotājiem patīk strādāt savā ierastajā programmatūras vidē, viņiem patīk, ja jaunizveidotās informācijas ir ļoti viegli saprotamas, apgūstamas un lietojamas Tajā pat laikā lietotāji prasa, lai jaunās prasības izstrādātāji realizētu ātri, bet informācijas sistēmas saglabātu savu uzticamību. Lietotāji vēlas datus no informācijas sistēmām iegūt ātri un ar mazu piepūli, lai varētu pieņemt lēmumus biznesā.

Lietotāji ir labi eksperti par lietotāju saskarnēm – viņi pamatā zina, kā kurām atskaitēm ir jāizskatās un kāda veida ievadformas viņiem būtu vislabākās [Arn07]. Lietotāji bieži izmanto elektroniskās tabulas, lai radītu specifikācijas tabulveida dokumentiem, kurus nepieciešams pievienot informācijas sistēmai.

Rezultātā tika izveidots ietvars, kas piedāvā izmantot elektroniskās tabulas kā lietotāju saskarni vairāku iemeslu dēļ. Pirmkārt, lietotāji ir pazīstami ar elektroniskajām tabulām, jo izmanto tās savā ikdienā. Otrkārt, izklājlapās esošus datus ir viegli izmantot turpmākai apstrādei, piemēram, veidot no tiem pivottabulas, grafikus, kopēt uz teksta dokumentiem, prezentācijām. Treškārt, profesionāļi – programmatūras izstrādātāji var tikt atsvabināti no vienkāršu lietotāju saskarņu veidošanas. Ja gala lietotāji izveidojuši pietiekami labas dokumentu sagataves izklājlapās, tad šī pieeja ietaupa daudz programmētāju un sistēmanalītiķu laika.

Gala lietotāji bieži programmē ar mērķi izveidot programmas, ko izmantot savā ikdienas darbā [KAB+11]. Gala lietotāju veikto programmatūras izstrādi var definēt kā „metožu, tehniku un rīku kopu, ko var izmantot programmatūras lietotāji, kas darbojas

kā neprofesionāli programmatūras izstrādātāji, lai nepieciešamības gadījumā radītu, mainītu vai papildinātu programmatūras artefaktus" [LPW06]. Darbā aprakstītajā gadījumā lietotāji izveidoja elektronisko tabulu šablonus, izmantojot ļoti vienkāršu vizuālu domēnspecifisku valodu [YC02]. Šablonus var uzskatīt par programmām, ja programmēšanu definē kā „programmas plānošanas un rakstīšanas procesu, kur programmu definē kā specifikāciju kolekciju, kas var saņemt ievadvērtības un ko var izpildīt (vai interpretēt) iekārta ar skaitļošanas spējām" [KAB+11].

Pētījumi apliecina, ka gala lietotāji plaši izmantot programmēšanā izklājlapas [BG01], [Dra05], [Erw09], [MKB06], [OG08]. *Erwig et.al.* apraksta programmu ģeneratoru [EAC+05], kas transformē šablonu par lietošanai gatavu izklājlapu, kurā ir arī konkrētajai tabulai piemērotas operācijas šūnu labošanai, rindiņu un kolonnu ieveidošanas un izmešanas operācijas. Viņu programma arī pasargā lietotāju no datu ievada kļūdām, kā arī palīdz lietotājam rindiņu vai kolonnu izmešanas vai ievietošanas gadījumā.

Darbā piedāvātais ietvars arī atbalsta datu saglabāšanu datu bāzē un datu iegūšanu no tās. Visu datu apraksti ir reģistrēti sistēmas ontoloģijā. Ontoloģiju veido datu elementu apraksti, modelējot domēna zināšanu bāzi, kuras pamatelementi ir klases, atribūti un saites [LÖ09].

Ietvara veidotā ontoloģija satur informāciju par dokumentu veidiem, ko sistēma saņem vai izdod, izmantojot lietotāju saskarnes, kā arī katra ievaddokumentu veida katra datu elementa aprakstu. Tā satur arī datu kvalitātes un integritātes pārbaudēm nepieciešamo informāciju.

Ontoloģijā saglabātie datu apraksti tiek izmantoti, lai norādītu uz datiem elektronisko tabulu šablonos, izmantojot izveidoto domēnspecifisko valodu. Tāpat tie tiek izmantoti storētajās procedūrās, kad tiek atlasīti datu komplicētām atskaitēm.

Ietvara programmatūra atbalsta datu ievadu no izklājlapām, kas tiek uzģenerētas no šabloniem, datu integritātes kontroles, izmantojot likumus, kas saglabāti ontoloģijā, kā arī vienkāršu atskaišu veidošanu atbilstoši dotajiem šabloniem. Komplicētām atskaitēm datus atlasa storētās procedūras, kuras raksta profesionāli programmētāji.

Šajā pieejā profesionāli programmētāji un sistēmanalītiķi veic tikai komplicētus uzdevumus, vienkāršu ikdienas darbus spēj izdarīt gala lietotāji.

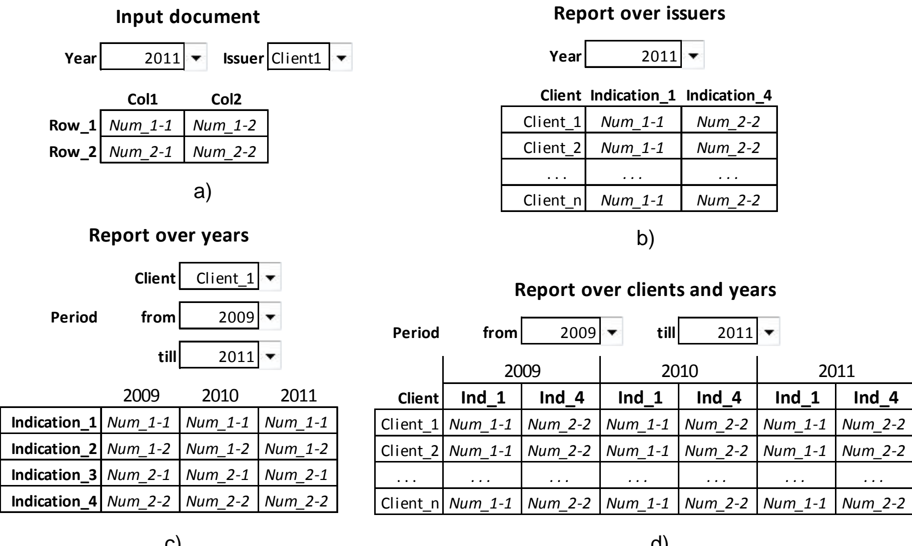
Darbā aprakstītajai pieejai ir vairākas priekšrocības. Pirmkārt, tā samazina profesionālu programmatūras izstrādātāju veicamo darbu apjomu, otrkārt, ar tās palīdzību var sasniegt lielāku lietotāju apmierinātību ar programmatūru, jo izklājlapas izskatās precīzi tādas, kādas viņi tās specificēja. Treškārt, šī pieeja samazina

programmatūras testēšanas sarežģītību, jo šablonu testēšana parasti ir vienkāršāka nekā pilnībā no jauna veidotas saskarnes testēšana.

4.4.1. Ietvaram piemērotās informācijas sistēmas

Piedāvātais ietvars ir paredzēts tādu informācijas sistēmu izstrādes atbalstam, kas uzkrāj ar tabulveida dokumentiem ienākošus datus. Atskaites šajās sistēmās arī parasti ir tabulveida dokumenti. Šajās sistēmās nepieciešams arī veikt datu savstarpējās integritātes datu kvalitātes kontroles.

Ievaddokumentus un atskaites var sagrupēt četrās grupās. Pirmajā grupā ir *fiksēta izmēra* dokumenti, kā, piemēram, 4.6.(a) attēlā attēlotajā dokumentā. Parasti šajos dokumentos dati ir izkārtoti tabulās ar iepriekš zināmu, fiksētu kolonu un rindiņu skaitu. Nākamo trīs grupu dokumentiem ir *plūstošais izmērs*. Otrās grupas dokumentos ir tabula ar fiksētu virsrakstu daļu un nezināmu rindiņu skaitu (4.6.(b) att.). Katrā rindiņā ir viena un tā paša tipa informācija, kā noteikt tabulas virsraksta daļā. Trešā dokumentu grupa ir līdzīga otrajai, bet rindiņas un kolonnas ir savstarpēji mainītās lomās - tajā ir fiksēts rindiņu skaits, bet iepriekš nezināms kolonnu skaits (4.6.(c) att.). Un visbeidzot, ceturtā grupa satur dokumentus, kam specifikācijas veidošanas brīdī nav precīzi zināms ne rindiņu, ne kolonnu skaits (4.6.(d) att.). Reālo kolonnu un rindiņu skaitu nosaka atskaites dati, piemēram, kā klienti atskaitēs, kas redzams attēlos 4.6.(b) un 4.6.(d) vai izvēlētās vērtības atskaites vadīklās, kā parādīts attēlos 4.6.(c) un 4.6.(d).



4.6. att. **(a) Ievaddokumenta shematisks piemērs, dažādi atskaišu veidi - (b) vertikāli *plūstoša* atskaite, (c) horizontāli *plūstoša* atskaite, (d) vertikāli un horizontāli *plūstoša* atskaite**

Ievaddokumenti parasti ir pirmajā un otrajā grupā, bet trešās un ceturtās grupas dokumenti parasti ir atskaites.

Katram dokumentam ir divas daļas. Pirmajā daļa ir informācija par dokumentu kopumā, piemēram, kas to izdevis, laika periods, uz kuru attiecas informācija. Dokumenta otrajā daļā ir dati, piemēram, dažādi skaitļi un apraksti, ko šie skaitļi nozīmē, arī kolonnu un rindiņu virsraksti.

Kā shematisku piemēru var aplūkot *fiksēta* tipa dokumentu kā 4.6.(a) attēlā. Tas tiek iesniegts reizi gadā. Tas satur četrus rādītājus Indication1, Indication2, Indication3 un Indication4, kas attēlā kodēti kā Num_1_1, Num_1_2, Num_2_1, Num_2_2. Rādītāji var būt, piemēram, atbilstošajā gadā gūtie ieņēmumi, peļņa vai samaksātie nodokļi. Reālo skaitļu vietā 4.6.(a) attēlā ir parādīti šo datu elementu kodi.

No dokumentu, kuru tips ir kā 4.6.(a) attēlā attēlots, datiem, var ģenerēt visu četru dokumentu tipu grupu stila atskaites. Piemēram, (1) summējošās atskaite par visiem iesniedzējiem par konkrēto gadu (izskatīsies tieši tāpat kā ievadforma, tikai nebūs klienta izvēles vadīklu), (2) par iesniedzējiem jeb klientiem par konkrēto gadu par kādiem konkrētiem rādītājiem, kā 4.6.(b) attēlā, (3) par konkrētu klientu *plūstošā* atskaite pa labi pa gadiem, kā 4.6.(c) attēlā, un (4) abos virzienos *plūstošā* atskaite pa gadiem un klientiem, kā 4.6.(d) attēlā.

Piemēram, ja aplūko tuvāk atskaiti 4.6.(b) attēlā, var redzēt, ka šeit ir vieni un tie paši datu elementu kodi katram klientam un katrā kolonnā. Tā nav kļūda, tie ir datu elementu kodi. Ja tiktu veidota reāla atskaite, tad katrā atskaites šūnā koda vietā būtu atbilstošais skaitlis klientam, par konkrēto gadu, piemēram, 2011. gadu.

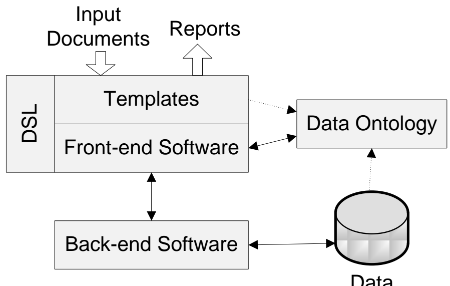
4.4.2. Sistēmas arhitektūra

Ietvars nodrošina informācijas sistēmai arhitektūru, kā shematiski parādīts 4.7. attēlā. Arhitektūrai ir četras svarīgākās daļas – datu ontoloģija, datu glabātuve, klienta un servera programmatūras.

Ontoloģijā ir aprakstīti sistēmas biznesa dati. Visi dati sistēmā nonāk pa kopnēm jeb dokumentiem. Katrā dokumentā ir kopa ar datu elementiem. Ontoloģijā ir informācija par dokumentu tipiem un datu elementu tipiem, kā arī datu savstarpējās integritātes un konsistences sakarību likumi.

Datu glabātuvē tiek glabāti sistēmas biznesa dati – ievadītie dokumenti un to datu elementu vērtības.

Klienta puses programmatūra ir kopums no standarta izklājlapu programmatūras, sistēmas šabloniem un ietvara programmatūras, kas veic šablonu interpretāciju atbilstoši ietvara domēnspecifiskajai valodai.

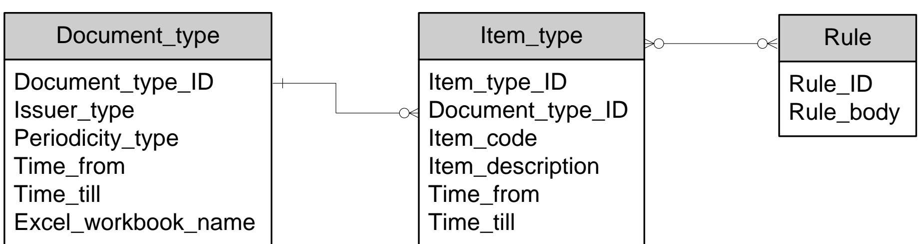


4.7. att. **Uz ietvara pamata veidotas informācijas sistēmas arhitektūra**

Servera puses programmatūra veic drošības nodrošināšanu, strādājot ar datu glabātuvi, to veido arī storētās procedūras datu bāzē, kas uzrakstītas ar mērķi nodrošināt komplicētākās atskaites ar datiem.

4.4.3. Sistēmas datu ontoloģija

Biznesa datu ontoloģija satur informāciju par ievaddokumentu, kuri tiks saglabāti glabātuvē, tipiem. Tā satur arī izvaddokumentu jeb atskaišu, kas tiks ģenerētas no sistēmas, tipu aprakstus tipiem. Katram dokumentu tipam ir kopa ar atribūtiem, piemēram, kāds ir dokumenta tipa periodiskums – gads, mēnesis, ceturksnis, kura organizācija vai institūciju grupa iesniegs atbilstošā tipa dokumentus, datumus no kura līdz kuram dokumentu tips ir spēkā iesniegšanai, kā arī šablona izklājlapas, kas nodrošina atbilstošo biznesa datu ievadu, labošanu vai atskaites ģenerēšanu, vārdu.



4.8. att. **Datu ontoloģijas konceptuāla shēma**

Interesantāka informācija ontoloģijā ir par datu elementu tipiem. Katram dokumentu tipam piesaistīts viens vai vairāki datu elementu tipi. Ontoloģijā ir saglabāta kopa ar atribūtiem katram datu elementu tipam, piemēram, atsauce uz dokumenta tipu, kam tas piesaistīts, elementa kods, elementa apraksts, arī datums, no kura līdz kuram

datu elementa tips ir spēkā izmantošanai. 4.8. attēlā attēlota datu ontoloģijas konceptuālā shēma.

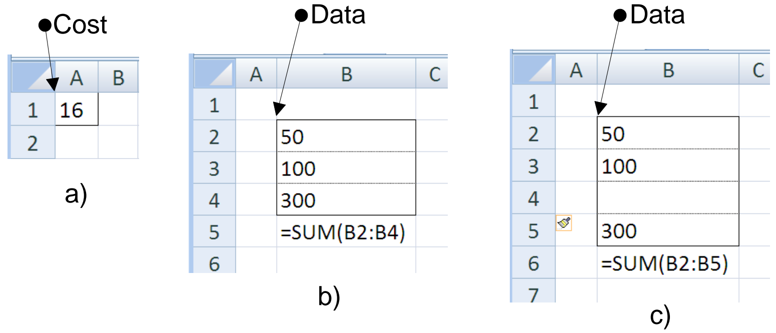
Katra datu elementu tipa kods ir unikāls informācijas sistēmas mērogā. Šī īpašība atļauj viegli un precīzi atsaukties uz nepieciešamo datu elementu tipu sistēmas lietotāja saskarnes šablonos, kā arī datu kvalitātes un integritātes pārbaudes likumos.

4.4.4. Lietotāju saskarnes šablonu DSL

Mērķis bija iedot lietotājiem rīku, kas viņiem atļautu veidot viņu biznesa sistēmu lietotāju saskarņu šablonus. Turpmākajās apakšnodaļas tiks aplūkotas svarīgākās izklājlapu īpašības, ko ietvara programmatūra izmanto, kad interpretē dokumentu šablonu.

4.4.4.1. Svarīgākās izklājlapu īpašības, ko izmanto DSL

Apgabals ir izklājlapas šūna, kā 4.9.(a) attēlā, vai nepārtraukta sekojošu šūnu grupa, kā redzams 4.9.(b) attēlā.



4.9. att. **Apgabalu piemēri – (a) vienas šūnas apgabals, (b) apgabals kā secīgu šūnu grupa, (c) apgabals kā secīgu šūnu grupa pēc vienas rindas iespraušanas pirms pēdējās apgabala šūnas**

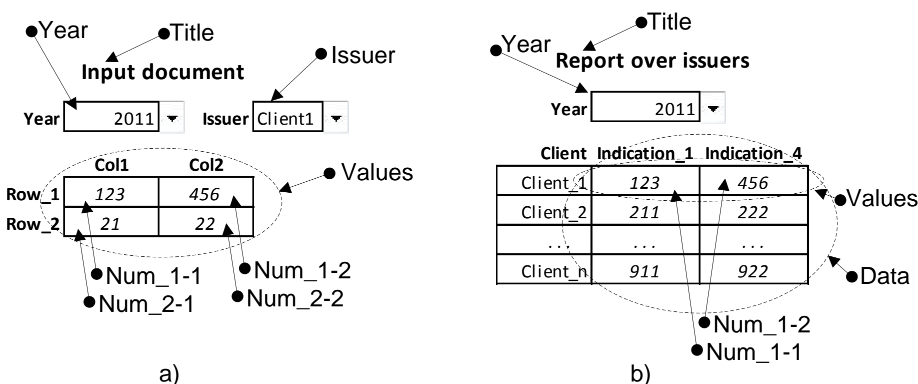
Ja apgabals ir sekojošu šūnu grupa, tas stiepjas, ja jaunas šūnas tiek iespraustas starp kādām apgabala šūnām, kā parādīts attēlos 4.9.(b) un 4.9.(c). Attēlā 4.9.(b) ir apgabals *Data*, ko veido šūnas B2:B4. Kad iesprauž jaunu šūnu pēc šūnas B4, izklājlapas funkcionalitāte automātiski nodrošina, ja iezīme *Data* tagad attiecas gan uz bijušajām šūnām, gan arī jauno šūnu apgabalu – B2:B5. Kas ir svarīgi – formulas, kuras atsaucas uz datu apgabaliem, izmantojot relatīvo adresāciju, tiek automātiski mainītas atbilstoši jaunajai situācijai, kā parādīts attēlos 4.9.(b) un 4.9.(c). Absolūtās adreses formulās paliek nemainītas.

4.4.4.2. DSL elementi

Ietvara šablonos DSL elementus interpretē izklājlapas datu makro vai programmas, kas radītas citās vidēs, bet izmantojot izklājlapas API funkcijas, manipulē ar to un tās datiem. Visi DSL elementi izklājlapas apgabali, kam piešķirti vārdi. Priekšrocība ir apstāklī, ka lietotāji var veidot šablonus izmantojot „*What You See Is What You Get*" principu. Piedāvātā DSL ir ļoti maza. Tajā ir tikai daži valodas elementi un tos var grupēt sekojoši:

- Statiskie elementi, piemēram, dokumenta virsraksts, tabulas virsrakstu rinda, kas satur datu elementus.
- Vadīklu elementi – lai nodrošinātu dokumentu atribūtus atbilstoši to tipiem, piemēram, izkrītošās izvēlnes gadam, gadam un ceturksnim vai gadam un mēnesim atbilstoši dokumenta tipa periodiskumam, kas norādīts šablonā, izkrītošā izvēlne dokumenta izdevēja izvēles vadīklai.
- Datu elementi – izklājlapas šūnas, kur dati tiks ievadīti vai modificēti.
- Citi specifiski DSL elementi – šūnas un apgabali ar iezīmēm, slēptās izklājlapas ar tehniskajiem datiem – mēnešiem, ceturkšņiem, gadiem, SQL vaicājumu veidošanai nepieciešamie fragmenti, datu integritātes pārbaužu likumi.

Šo elementu pielietošanas noteikumi ir pietiekami vienkārši, ka to spēj izdarīt arī lietotāji. Piemēram, 4.10.(a) attēlā parādīts fiksēta dokumentu tipa paraugs.



4.10. att. **Dokumentu šablonu piemēri, uzrādot apgabalu nosaukumus – (a) fiksēta dokumentu tipa šablons, (b) vertikāli plūstoša dokumentu tipa šablons**

Statiskie elementi ir parastas simbolu virknes, skaitļi vai citas vērtības, kas ievadīti izklājlapas šūnās ar iezīmi. Tās nolasa DSL interpretators, tās nekad netiek mainītas. Piemēram, 4.10.(a) attēlā apgabals *Title* ir statiskais elements.

Vadīklu elementi ir apgabali *Month* un *Issuer*. Šie elementi izskatā kā parastas šūnas, bet, kad lietotājs mēģina izmainīt to saturu, tad viņš var izvēlēties tikai vienu

vērtību no iepriekš definēta saraksta. Šablona veidotājs vadīklu elementi vērtību sarakstus iekopē slēptā izklājlapā, kas saglabā šablona izklājlapas darba burtnīcā.

Datu elementi ir apgabali, kas sastāv no vies šūnas, bet visus datu elementus izklājlapā ietver apgabals *Values*. Programmatūra ņem visus ir datus no apgabala *Values*, bet no katras individuālās šūnas iezīmes paņem datu elementa tipa kodu, ko izmanto, datus saglabājot vai nolasot.

Plūstošo dokumentu tipu šabloni ir mazliet komplicētāki. 4.10.(b) attēlā parādīts vertikāli plūstoša dokumentu tipa piemērs. Šajā šablonā apgabals *Values* ir iekļauts apgabalā *Data*. Apgabals *Values* veido apgabalā *Data* otro rindu. Apgabals *Data* satur apgabalu *Values* un vienu tukšu rindu pirms tās un otru tukšo rindu pēc tās. Katrreiz, kad ir nepieciešams pievienot jaunu rindiņu, tā tiek pievienota apgabalā *Data* pirms pēdējās rindas un tiek piepildīta un noformēta tieši tāpat kā datu elementi apgabalā *Values*.

Horizontāli *plūstošo* dokumentu tipu šabloni ir analogi vertikāli *plūstošo* dokumentu tipu šabloniem, tikai darbs notiek ar kolonnām rindiņu vietā.

Izmantojot tos pašus principus, tiek specificēti un interpretēti arī abos virzienos – horizontāli un vertikāli – *plūstošo* dokumentu tipu šabloni.

Izveidotais ietvars arī atbalsta atskaites ar vairākām datu elementu grupām, kurās ir vēl dažādi papildus aprēķini, piemēram, starpsummas.

Ietvara programmatūra lieto vēl citus apgabalus, piemēram, ar mērķi veikt datu konsistences pārbaudes.

4.4.5. Pielietojumi

Ietvars ir ticis lietots kā pamats divu informācijas sistēmu izstrādē. Abu šo sistēmu lietotāji savāc datus un vēlāk tos izmantot dažādu statistisku apkopojumu un analīžu veidošanai. Abus sistēmu lietotāji ir no valsts pārvaldes finansu institūcijas, kurā strādā vairāk par 500 cilvēku. Organizācijai ir centrālais ofiss un vairākas filiāles dažādās Latvijas vietās.

Pašlaik pirmajā no sistēmām ir izveidoti 205 ievaddokumentu tipu šabloni un 1397 atskaišu šabloni, bet otrajā ir 132 ievaddokumentu šabloni un 185 atskaišu šabloni. Abas sistēmas nemitīgi tiek papildinātas ar jauniem dokumentu šabloniem.

Ietvars ir izstrādāts, izmantojot MS Excel kā izklājlapu programmatūru lietotāju saskarnēm un kā klienta puses programmatūru. Kā datu glabātuve ir izmantota MS SQL servera datu bāze.

Gandrīz visus saskarņu šablonus izstrādāja gala lietotāji. Parasti viņi atsaucās, ka šablonu veidošana nav pārāk grūta.

Visvieglāk izveidojamie bija *fiksēto* dokumentu tipu šabloni. Lai piešķirtu unikālu kodu katram datu elementam, tie tika veidoti pēc šablona *DocumentTypePrefix_RowNo_ColumnNo,* kur *DocumentTypePrefix* ir sistēmas ietvaros unikāls dokumentu tipa apzīmējums *RowNo* un *ColumnNo* ir atbilstoši rindas un kolonnas numurs, kurā skaitlis atrodas. Kad katrs dokumentu tipa šablons izklājlapā bija izveidots, klienta puses programmatūra reģistrēja to datu bāzē. Reģistrācija nozīmēja izveidot ierakstu atbilstošā tabulā par dokumentu tipu, kā arī katram datu elementa tipam izveidot savu ierakstu atbilstošā tabulā.

Visi *fiksēto* dokumentu dati glabātuvē tika glabāti vienās un tajās pašās tabulās. *Plūstošo* dokumentu tipu datiem datu elementiem tika izveidots katram tipam savs tabulu komplekts, bet pēc viena stila. Katram bija divas tabulas – viena datu elementu pēdējo vērtību saglabāšanai, bet otra – datu elementu vērtību maiņas vēstures saglabāšanai. *Plūstošo* dokumentu tipu nebija daudz – zem 20 katrai sistēmai.

Arī vienkāršākās atskaites pilnībā varēja izveidot gala lietotāji. Tām, kad atskaitē bija nepieciešams ievietot datus, MS Excel makro ģenerēja datu atlases vaicājumu, izmantojot informāciju, kas saglabāta šablona apgabalu šūnās.

Taču bija arī ļoti komplicētas *plūstošā* dokumentu tipu atskaites, kuru izstrāde bija izaicinājums arī profesionāliem programmētājiem. Tām bija nepieciešams veidot storētās procedūras datu glabātuvē ar mērķi atlasīt un apkopot datus dažādos griezumos.

Lietotāji bija ļoti apmierināti, ka saņēma atskaites un ievadformas precīzi tādas, kādas tās bija specificējuši. Savukārt, izstrādātāji bija apmierināti, ka mazas izmaiņas un uzlabojumus šablonos lietotāji spēj veikt ar saviem spēkiem. Pozitīvs blakusefekts bija fakts, ka lietotāji bija spiesti paši savā starpā vienoties par lietotāju saskarņu izskatu un skaitu, neiesaistot IT cilvēkus kā starpniekus.

Programmatūras testētāji bija apmierināti, ka saskarņu testēšana bija vienkārša un ātra, izņemot komplicētās atskaites, jo šablonos nebija jauns pirmkods, tie visi izmantoja vienu un to pašu programmatūras kodolu, ko nodrošināja ietvars.

Savukārt ietvara klientu puses programmatūra bija ļoti sarežģīta, bet ieviesušās problēmas tajā parasti ļoti ātri izdevās atklāt, jo visas saskarnes izmantoja vienu un to pašu kodu.

Izveidotā ietvara svarīgākais ieguvums bija laika un spēku ekonomija sistēmu izveidošanā un uzturēšanā. Tika panākta situācija, ka lietotāji specificē saskarnes un saņem precīzi to, ko paši specificējuši. IT cilvēki – sistēmanalītiķi, programmētāji,

testētāji rūpējas par programmatūras un datu kvalitāti, cik tas atkarīgs no storētajām procedūrām, izklājprogrammas makro, kā arī asistēja datu elementu kodu izveidošanā, lai nodrošinātu to unikalitāti katras sistēmas ietvaros.

Pirmo klienta puses programmatūras prototipu izveidoja viens programmētājs-sistēmanalītiķis. Vēlāk tika izveidota izstrādes grupa no diviem cilvēkiem. Viens no tiem uzlaboja un uzturēja klienta puses programmatūru, kamēr otrs apmācīja lietotājus šablonu veidošanā, testēja sistēmu un papildināja tās prasību dokumentāciju. Pirmajā sistēmā pamatā bija *fiksētie* dokumentu tipi gan kā ievaddokumenti, gan kā atskaites, kā arī vienkāršas horizontāli vai vertikāli *plūstošās* atskaites. Pašā sākumā tikai divi ievaddokumenti bija ar *plūstošo* tipu un tikai 5 atskaišu tipi bija komplicēti – ar apakšsummām un *plūstošajiem* datiem – horizontāli un vertikāli. Arī vēlāk lietotāji pievienoja tikai dažas *plūstošā* tipa ievadformas un komplicētas atskaites.

Divu mēnešu laikā tika panākts, ka lietotāji jau var sākt datu ievadu jaunajā sistēmā, izmantojot pirmās 50 saskarnes, no kurā 2 bija sarežģītākas, vēlāk mēneša laikā tika izveidotas ap 100 atskaitēm, kur 5 no tām bija ļoti komplicētas. Tātad 5 IT cilvēku cilvēkmēnešu laikā tika paveikts darbs, kas pēc COCOMO novērtējuma prasītu 244 cilvēkmēnešus 9 mēnešu laikā.

Kad paralēli pirmās sistēmas uzturēšanai bija nepieciešams uzsākt otrās sistēmas izstrādi, komandu papildināja vēl 3 cilvēki. Otrās sistēmas atskaišu sarežģītības dēļ viens no viņiem specializējās atskaitēm nepieciešamo storēto procedūru un SQL vaicājumu izstrādē. Pārējie divi programmēja, ja nepieciešams, palīdzēja lietotājiem, apmācīja viņus, kā arī pārvaldīja datu elementu kodu sistēmu.

Otrā sistēma tika izveidota apmēram 9 mēnešu laikā, bet tajā bija papildus nepieciešams realizēt netriviālas datu kontroles starp avotiem. Uzsākot ekspluatāciju, tajā bija izstrādāti 23 ievaddokumentu šabloni, 39 atskaišu šabloni un 3 ļoti komplicēti datu konsistences pārbaužu aprēķini. Tātad ieguldot 31.5 cilvēkmēnešu ilgu IT cilvēku darbu tika izveidota sistēma, kas pēc COCOMO novērtējuma prasītu 142 cilvēkmēnešus 8 mēnešu laikā.

Abiem projektiem ir grūti novērtēt darba apjomu, ko ieguldīja gala lietotāji sistēmu izstrādē, jo viņi šablonus papildināja ar DSL izteikumiem uzreiz pēc to izskata specificēšanas, kā dēļ nav zināms laiks daudzums, ko viņi ieguldīja specificēšanā un ko – sistēmas šablonu veidošanā.

4.5. Nodaļas secinājumi

Nodaļā apskatītais SUP modelis testēšanas procesa uzlabošanai ir veidojies pakāpeniski, lai palīdzētu risināt atsevišķas praksē eksistējošas problēmas. Svarīgākais šim modelim ir augšējā līmeņa skats uz programmatūru, tās uzvedību un īpašībām no svarīgāko ieinteresēto pušu redzespunktiem. Modelis domāts ne tikai IT speciālistiem, bet arī dažādiem biznesa pārstāvjiem, kuru zināšanas IT tehnoloģijās un datorzinātnēs ir nelielas un sadrumstalotas.

SUP modelis saklasificē daudzās iespējamās situācijas, kādas veidojas starp dažādu Izpildītāju vīzijām, rakstiskajām prasībām un izveidoto programmatūru. Skaidrojot šīs situāciju klases ar saprotamiem piemēriem, ir vieglāk panākt labāku sapratni par programmatūras izstrādes un testēšanas procesu starp daudzajām izstrādes projektā iesaistītām personām. Uzskatāmi var ieraudzīt, vai projektā nenotiek kādas puses ignorēšana, piemēram, netiek pietiekoši iesaistīti topošās programmatūras Lietotāji. Tāpat ir vieglāk ieraudzīt, vai testēšana nav pārāk vienpusīga un šaura.

Praksē laika trūkuma dēļ Lietotājiem un Pasūtītājiem no biznesa puses var piedāvāt arī sašaurinātu modeli, piemēram, tikai SUP-V vai SUP-D skatus ar iegūto sektoru skaidrojumu.

Izmantojot SUP modeli kā papildus līdzekli, iespējams labāk organizēt programmatūras izstrādes procesus visā dzīves cikla laikā. Modelis atbalsta testēšanas aktivitāšu uzsākšanu jau sākotnējo vīziju stadijā, plānošanā un risku analīzē. Vēlākās fāzēs modeli var izmantot programmas testēšanas stratēģijas izveidošanā, testēšanas plānošanā, plānu pārskatīšanā un uzlabošanā, testēšanas metožu un tehniku izvēlē.

SUP modelis ļauj organizēt programmatūras testēšanu plašumā un veikt pārbaudi, vai programmatūras izstrāde vai pats produkts neatrodas kādā nevēlamā stāvoklī.

SUP modelis tiek pielietots reālos projektos, ar to tiek iepazīstināti arī klausītāji testēšanas apmācību kursos. Līdz šim modelis pielietots neformālā un intuitīvā veidā.

SUP modelis var palīdzēt pārvaldīt testēšanas procesus, runājot par testēšanas sistēmu kā kompleksu sistēmu, jo var kalpot kā viens pārvaldības mehānisms. SUP modelis ļauj testētājiem kā aģentiem kompleksā sistēmā autonomi darboties un tajā pašā laikā mijiedarboties ar citiem aģentiem, nododot savas zināšanas par testējamas sistēmas prasībām, piederību ieinteresētajām pusēm un implementācijas kvalitāti.

Turpmāko pētījumu gaitā nepieciešams SUP modeli formalizēt un izstrādāt precīzāku tā izmantošanas metodoloģiju. Vēlams izstrādāt vairākus algoritmus jeb scenārijus modeļa izmatošanai un dot vadlīnijas to īstenošanai. Īpašu uzmanību

nepieciešams pievērst pašreizējā projekta stāvokļa noteikšanai, t.i. problemātiskāko SUP modeļa sektoru identificēšanā, uzrādot projekta kritiskās un vājās vietas un parādot ceļus vai virzienu to uzlabošanai. To varētu veikt analizējot projektā veiktās aktivitātes (piemēram, sanāksmes, tajā iesaistītās personas, izveidotos dokumentus), inspicējot dokumentāciju un veicot iesaistīto personu aptauju.

Nodaļā piedāvāto daudzaģentu sistēmu darbības principu ievērošana palīdz nodrošināt gan komplekso sistēmu darbības principu, gan SUP modeļa principu realizāciju praksē. Kā aģentus autore piedāvā uztvert testētāju prasmes un iemaņas (principu A grupa), kas izmanto aģentūras – testētāja- resursu, lai vieglāk izprast tieši kādas testētāja prasmes tiek reāli izmantotas, vai to vietā tikt izmantoti rīki, kā arī, kādas prasmes nepieciešams apgūt no jauna. Uztverot testēšanas komandas locekļus kā ieinteresētās puses, kas katra realizē kādu testēšanas sistēmas uzdevumu, SUP modelis palīdz aģentūrām saskatīt savstarpējo ieinteresētību, lai kooperētos un koordinētos (principu C un S grupas), veicinot aģentu sinerģiju un koevolūciju. SUP modeļa izpilde darbā un T un O grupu principu ievērošana palīdz nodrošināt, ka lokāli un diezgan autonomi darbojoties, testētāji pietiekamā laika periodā panāk kopējo testēšanas sistēmas uzdevumu izpildi. F principu grupa rūpējas par aģentūru dažādību, par to iespējām vienam otru dublēt.

SUP modelis izskatāmi demonstrē ne-IT cilvēku iesaistīšanas testēšanā un programmatūras izstrādes procesos nepieciešamību. Ne-IT testētājus uzņēmumos izmanto dažādu iemeslu dēļ – profesionālu IT testētāju trūkst, grūti viņus atrast, viņi dārgi izmaksā, viņi pietiekami labi nepārzina uzņēmuma biznesu. No otras puses ne-IT testētāju stiprā puse ir laba uzņēmuma biznesa pārzināšana un daudzas lietas testēšanā viņiem ātri var iemācīties un efektīvi izmantot.

Problēmas ne-IT testētājiem var sagādāt dokumentācijas veidošana, testu projektējuma aprakstīšana, testpiemēru izvēles pamatošana. To pamatā ir bailes parādīt savu nezināšanu, neprasmi vai kļūdas. Rezultātā cieš kompānija – testētāji nevar viens no otra mācīties, ja kāds testētājs aiziet, viņa izveidoto testu sistēmu pārņemt kolēģiem var būt ļoti grūti. Arī pašam testētājam nav vienkārši sekot līdzi savas testēšanas sistemātiskumam, papildināt testu sistēmu programmatūras izmaiņu gadījumos

Vislabāk ne-IT testētājus var izmantot kombinācijā ar profesionāliem IT testētājiem. Tad viņi papildina viens otra kompetences. Ne-IT testētāji veiksmīgi var testēt sistēmas atbilstību biznesa procesiem, programmatūras lietojamību, lietotāja saskarnes – ekrānformas un atskaites.

Vēlams, lai ne-IT testētāju darbu atbalstītu izstrādātāji, palīdzot darbā ar testdatu izveidošanu datu bāzē, SQL vaicājumu veidošanu un citos jautājumos, kur nepieciešamas specifiskas IT zināšanas.

Ne-IT testētāji var izveidot realitātei atbilstošus lietošanas scenārijus, testdatus, ņemot vērā arī robežgadījumu un ekvivalences klašu analīzi, ko izstrādātāji bieži vien nespēj, jo tik labi nepārzina biznesa nianses. Viņi var veidot vienkāršus SQL vaicājumus datu bāzē, lai pārbaudītu datus un pētītu LOG tabulu ierakstus.

Ne-IT testētāji var labi novērtēt sistēmas lietotāju saskarņu lietojamību – vai darba vietas ir ērtas, vai visas nepieciešamās funkcijas var izpildīt, vai sistēma lieki nenoslogo redzi, atmiņu, vai nav grūti lietojama.

Ne-IT testētāju apmācības testēšanā ne vienmēr būtiski uzlabo testētāju darba rezultātus. Piemēram, laika trūkuma apstākļos viņiem nav iespējams izmantot savas zināšanas pietiekami labi. Ne-IT testētāju apmācības nedrīkst būt vispārējas. Tās jāpiemēro katrai organizācijai, izmantojot piemērus no programmatūras, kuru viņi testē. Apmācībās nepieciešams liels daudzums piemēru, lai materiāls auditorijai būtu labāk uztverams.

Ne-IT testētāji apmācībās nelabprāt uztver teoriju. Visā materiālā vēlams izmantot piemērus no viņu ikdienas aktivitātēm, no programmatūras, kuru viņi tobrīd testē vai gatavojas testēt. No otras puses, viņi labvēlīgi uztver informāciju, kas izskaidro situāciju projektā un viņu vietu tajā, piemēram, par viņu projektā izmantoto programmatūras izstrādes metodoloģiju, par veidiem, kā var organizēt testētāju darbu projekta izstrādes komandā.

Informācija par metrikām jādod ļoti uzmanīgi, jo pirmā reakcija parasti ir, ka tās tiks izmantotas tiešā veidā viņu algu ietekmēšanā. Pārsteidzoši, ka šāda reakcija bija arī no IT jomā izglītotu vadītāju puses. Pat viņi uztvēra metrikas drīzāk kā līdzekli cilvēku darba efektivitātes mērīšanai, nevis kā rīku testēšanas procesa kā tāda efektivitātes mērīšanai un kā motivatoru uzlabot šo procesu.

Klausītāji entuziastiski uztvēra iespēju apgūt praktiskas iemaņas, piemēram, prasību analīzē, vienkāršā un efektīvā testdatu izvēlē, testpiemēru skauta samazināšanas metodēs, testpiemēru kompakta pieraksta metodēs.

Vēlams, lai mācībās piedalās arī vadītāji, jo tajās parādās problēmas, par kuru esamību vadītāji bieži vien neko nezina. Īpaši, runājot par ne-IT testētāju psiholoģisko komfortu attiecībās ar programmētājiem, analītiķiem, lietotājiem, viņu vadītājiem vai par tehniskajām problēmām, kuras viņi izjūt, bet par kurām neuzdrīkstas runāt ar kolēģiem vai vadību.

Ir nepieciešams turpināt pētījumus par problēmām, ar kurām saskaras ne-IT testētāji, par zināšanām, kādas viņiem var palīdzēt. Apmācības kursu saturu un izklāsta veidu jāuzlabo, lai ne-IT testētāji tos varētu uztvert vēl efektīvāk.

Nodaļā tika aprakstīts ietvars informācijas sistēmu, kuru saskarnes veido izklājlapas, izstrādei. Ietvara svarīgākā priekšrocība ir gala lietotāju iesaistīšana sistēmas izstrādes procesā.

Gala lietotāju dalība programmēšanas aktivitātēs ir mainījusi izstrādes sarežģītību:

- Klientu puses programmnodrošinājuma sarežģītība ir kļuvusi lielāka. Aizmugursistēmas programmatūras funkcionalitātes programmēšana ir grūta, šī funkcionalitāte nav viegli pavadāma.
- Jaunu saskarņu izveidošana ir viegla un ātra, darbu lielāko daļu spēj veikt gala lietotāji.
- Saskarņu izstrādes un lietojamības testēšanas sarežģītība „pārceļas" no IT departamenta cilvēkiem uz gala lietotājiem.

Ietvarā izstrādātas sistēmas ontoloģija ir sadalīta un glabājas realizēta vairākās daļās – klienta programmatūrā, aizmugursistēmas programmatūrā un saskarņu šablonos – šablonu likumos, datubāzē storētajās procedūrās. Turpmākajā darbā ir nepieciešams uzlabot likumu pierakstīšanu un pārvaldību ontoloģijā.

Piedāvātā ietvara pozitīvie aspekti:

- Ievadformu specificēšana un implementācija ir ātrāka:
  - Lietotāji paši izveido šablonu izklājlapai, nodrošinot, ka tajā tiks rādīta nepieciešamā informācija, tā būs pietiekami ērta efektīvam darbam, tās izdruka būs ar nepieciešamo kvalitāti.
  - Lietotāji un izstrādātāji nedublējas, zīmējot un specificējot ievadformas un atskaites.
  - Izstrādātāji veic tikai tehnisko darbu, sasaistot saskarņu šablonus ar datu bāzi. Viņi neprogrammē programmas vizuālās komponentes, kas parasti prasa daudz resursu.
  - Izstrādātāji neraksta jaunu pirmkodu katrai formai, visas formas tiek darbinātas ar vienu kodu, ietaupot laiku, kas citādi tiktu notērēts programmēšanai un testēšanai.
  - Ja lietotāji vēlas mainīt jebkura dokumenta vizuālo reprezentāciju sistēmā, tad viņi to var izdarīt, neiesaistot IT speciālistus.
- Informācijas sistēma kopumā ir stabilāka:

- Visas lietotāju saskarnes ekspluatē vienu un to pašu kodu, tāpēc kļūdas tiek atklātas ātrāk, bet ieviestās korekcijas uzreiz darbojas visos saskarņu šablonos, kas tiek lietoti.
- Ir vieglāk nodot sistēmas pavadīšanu citam programmētājam, jo koda apjoms ir mazs un tas nav atkarīgs no kādas konkrētas ievadformas vai atskaites.
- Programmatūra ir stabilāka un vieglāk pārnesama uz citu ekspluatācijas vidi vai platformu.

- Lietotāji saņem sistēmu, kas atbilst viņu vajadzībām:
  - Specificēšanai lietotāji lieto rīku, ko viņi labi pārzina (piemēram, MS Excel).
  - Lietotāji savstarpēji izdiskutē un specificē saskarnes, viņiem maz ir nepieciešamas kādas IT cilvēku konsultācijas.
  - Lietotāji saņem precīzi tādas saskarnes, kādas paši ir specificējuši.
  - Jaunas ievadformas un atskaites var tikt ātri izstrādātas, parasti iztiekot bez tradicionālās programmēšanas.
  - Jauno saskarņu testēšanu var tūlīt pēc to izveides veikt paši lietotāji.

Tomēr arī negatīvi aspekti – daļa klientu puses un aizmugursistēmas programmatūras ir diezgan sarežģīta (programmēta augstā abstrakcijas līmenī) un tādēļ nav viegli izprotama. Daļa atskaišu ir pārāk sarežģītas, lai tiktu izteiktas ar doto domēnspecifisko valodu, tādēļ pie tām vajadzēja strādāt IT cilvēkiem, piemēram, dažkārt atskaišu datus bija nepieciešams atlasīt ar storēto procedūru, jo ar vienu vaicājumu tas nebija izdarāms.

Piedāvātais ietvars un metodoloģija ir ļoti noderīgas konkrētajai pielietojuma sfērai. Aprakstītās pieejas rezultāti pat bija pozitīvāki nekā bija gaidīts. Nepieciešams turpināt pētījumus, lai paplašinātu ietvara pielietojuma iespējas un uzlabotu domēnspecifisko valodu darbam ar sarežģītām saskarnēm.

Nodaļā aprakstītie rezultāti ir publicēti [AA09b], [AA09a], [AA08], [AA11], [Arn07], [Arn00] un [Arn11]. Darba autores jaunieguldījums ir SUP modeļa idejas attīstīšana, daudzaģentu sistēmu darbības principu izmantojuma testēšanā ideju attīstīšana un izstrāde, pētījums par ne-IT cilvēku izmantojumu programmatūras testēšanā, kā arī izveidotais ietvars programmatūras izstrādei, iesaistot izstrādē kā programmētājus un testētājus ne-IT cilvēkus.

REZULTĀTI

Darbs veltīts testēšanas metodoloģisko, tehnoloģisko un organizatorisko aspektu izpētei ar mērķi nodrošināt uzticamas programmatūras izveidi. Testēšana, kas balstīta uz testēšanas modeļa izveidi ar sekojošu programmu testēšanu atbilstoši izvēlētajam modelim, sastopas ar nopietnām grūtībām – gan modeļa izveide, gan tai sekojoša testēšana ir sarežģīta un prasa lielus resursus. Tādejādi nereti testētāji atļaujas sistemātiskas testēšanas vietā aprobežoties ar dažu tipisku lietošanas gadījumu pārbaudi, kas neizbēgami noved pie programmu izraisītām informācijas sistēmu kļūdām. Darbam ir sekojoši galvenie rezultāti:

- Darbā testēšanas tehnoloģisko aspektu jomā aplūkots vērtību apgabalu testēšanas modelis un novērtēta testēšanas sarežģītība kā izpildāmo testu skaits dažādiem vērtību apgabalu testēšanas kritērijiem. Identificētas tradicionālās testēšanas, kas balstīta uz vērtību apgabalu testēšanas modeļiem, galvenā problēma – sarežģītība, kas praktiskos lietojumos padara neiespējamu šo testēšanas modeļu lietošanu.
- Lai pārvarētu testēšanas tehnoloģiskās sarežģītības problēmu, tiek piedāvāts testēšanas teorijā oriģināls modelis (SUP modelis) testēšanas efektivitātes paaugstināšanai, kuru raksturo nepieciešamība ņemt vērā sistēmu izstrādē iesaistīto dalībnieku atšķirīgu skatījumu uz veidojamo sistēmu. Šis modelis pierāda ne-IT speciālistu iesaistīšanas testēšanā nepieciešamību.
- Lai pārvarētu testēšanas tehnoloģisko sarežģītību, tiek piedāvāts testēšanas sistēmu uztvert kā kompleksu sistēmu un kompleksu sistēmu darbības principus izmantot, lai sasniegtu labākus testēšanas rezultātus. Papildus oriģinālai pieejai testēšanai, darbā izstrādāti metodiski ieteikumi testēšanas procesu pārvaldībai, pamatojoties uz komplekso sistēmu darbības principiem.
- Tiek piedāvāts testēšanu organizēt pēc daudzaģentu sistēmas parauga, kas ļauj samazināt testēšanas sarežģītību un atrisināt testētāju kvalifikācijas problēmas, iesaistot galalietotājus sistēmas izstrādē un testēšanā.

Programmatūras testēšanu, tās stratēģiju pēc būtības virza tās sarežģītība, kuru ietekmē gan testējamās programmatūras sarežģītība, gan ieinteresēto pušu doto uzdevumu sarežģītība, gan pieejamo resursu apjoms, gan testētāju komandas raksturlielumi – apjoms, kvalifikācija, pieredze, u.c.

Darbā piedāvātā pieeja programmatūras testēšanai ir balstīta galvenokārt datu apstrādes sistēmu izstrādes un testēšanas praksē. Kaut arī programmatūras testēšanas sarežģītība nereti ir grūti nosakāma, piedāvātā testēšanas organizatoriskā struktūra var mazināt tās ietekmi uz testēšanas kvalitāti.

Darbā veikto pētījumu rezultātā ir parādīts, ka izpratne par programmatūras testēšanas sistēmu kā kompleksu sistēmu sniedz virkni priekšrocību, jo ļauj izprast, ar kādiem paņēmieniem testēšanas sistēma ir pārvaldāma. Ne vienmēr programmatūras testēšanas sistēma darbojas kā kompleksa sistēma – tradicionāli testēšana tika veikta, sīki plānojot un cieši turoties pie šiem plāniem izpildes gaitā. Tomēr autore ir pārliecināta, ka modernās programmatūras izstrādes metodoloģijas, piemēram, kā spējā metodoloģija, jau sevī organiski ietver komplekso sistēmu pārvaldes elementus, kas arī testētājiem ļauj darboties kā kompleksas sistēmas sastāvdaļām.

Komplekso sistēmu pieeja testēšanas sistēmu darba pārvaldībā, SUP modelis, pieredze ne-IT cilvēku iesaistīšanā programmatūras izstrādes un testēšanas darbos nākotnē var tikt izmantoti gan testēšanas projektos, gan testēšanas metodoloģiju attīstīšanā.

IZMANTOTĀ LITERATŪRA UN AVOTI

[AA08] **Arnicane, V., Arnicans, G.** Using the Principles of an Agent-Based Modeling for the Evolution of IS Testing Involving Non-IT Testers. ***In:*** *Proceedings of 8th International Baltic Conference on Databases and Information Systems (Baltic DB&IS 2008)* (Haav, H. M., Kalja, A. eds.), June 2-5, Tallinn, Estonia, Tallinn University of Technology Press, 2008, pp. 129-140

[AA09a] **Arnicane, V., Arnicans, G.** Opportunities to Improve Software Testing Processes on the Basis of Multi-Agent Modeling, **In:** Frontiers in Artificial Intelligence and Applications. Databases and Information Systems V - Selected Papers from the Eighth International Baltic Conference, DB&IS 2008 (Haav, H.M., Kalja, A. eds.), vol 187, IOS Press, Amsterdam Berlin Oxford Tokyo Washington, DC, 2009, pp. 143-154

[AA09b] **Arnicane, V., Arnicans, G.** Using the Sponsor-User-Programmer Model to Improve the Testing Process, ***In:*** *Scientific Papers University of Latvia* (Bārzdiņš, J. ed.), vol 751, Computer Science and Information Technologies, University of Latvia, 2009, pp. 65-79

[AA11] **Arnicane, V., Arnicans, G.** Evolutionary Reduction of the Complexity of Software Testing by Using Multi-Agent System Modeling Principles, ***In:*** *Multi-Agent Systems - Modeling, Interactions, Simulations and Case Studies (*Alkhateeb, F., Al Maghayreh, E., Abu Doush, I. eds.), ISBN: 978-953-307-176-3, InTech, 2011, pp. 149-174

[Aar05] **Aart, C.** *Organizational Principles for Multi-Agent Architectures*, Birkhauser Verlag, Basel Boston Berlin, 2005

[ACP87] **Arnborg, S., Corneil, D. G., Proskurowski, A.** Complexity of finding embeddings in a k-tree. *SIAM J. Alg. Disc. Meth*., 8, 1987, pp. 277–284

[And03] **Andriani, E. P.** Evolutionary dynamic of industrial clusters, ***In:*** *Complex Systems and Evolutionary Perspectives on Organisations: The Application of Complexity Theory to Organisations*, (E. Mitleton-Kelly, ed.), Pergamon, Elsevier Science Ltd, Amsterdam Boston, 2003

[Arn00] **Arnicāne, V.** Statistikas sistēmu datu ievades un atskaišu formu izstrādes metodoloģija, *Ekonomikas un vadības zinību attīstības problēmas, Latvijas Universitātes zinātniskie raksti*, 628.sējums, 2000, Rīga, pp. 9-12

[Arn07] **Arnicane, V**. Use of Non-IT Testers in Software Development. **In:** *8th International Conference on Product Focused Software Process Improvement (PROFES 2007), LNCS*, (Münch, J., Abrahamsson P. eds.), vol. 4589, Berlin, Springer, 2007, 175--187

[Arn09] **Arnicane, V.** Complexity of Equivalence Class and Boundary Value Testing Methods, **In:** *Scientific Papers University of Latvia*, (Bārzdiņš, J. ed.),Vol 751, Computer Science and Information Technologies, University of Latvia, 2009, pp. 80-101

[Arn11] **Arnicane, V.** End-User Development Framework with DSL for Spreadsheets, **In:** Perspectives in Business Informatics Research, Local Proceedings, 10th International Conference, BIR 2011 Associated Workshops and Doctoral Consortium, (Niedrite, L. Strazdina, R., Wangler, B. eds.), Riga Technical University, 2011, pp. 437-447

[Ash03] **Ash, L.** The Web Testing Companion—The Insider's Guide to Efficient and Effective Tests. Wiley Publishing, Inc., Indianapolis Indiana, 2003

[AW93] **Agrawal, K., Whittaker, J. A.** Experiences in applying statistical testing to a real-time, embedded software system, ***In:*** *Proceedings of the Pacific Northwest Software Quality Conference*, October 1993, pp. 154-170

[AWZ92] **Afifi, F. H., White, L. J., Zeil, S. J**. Testing for linear errors in nonlinear computer programs. ***In:*** *Proceedings of the 14th international Conference on Software Engineering ICSE '92,* ACM, New York, NY, 1992, pp. 81–91

[Bac96] **Baccarini, D.** The concept of project complexity, *International Journal of Project Management*, vol. 14, no. 4, 1996, pp. 201-204

[Bae10] **Baerisch, S.** *Domain-Specific Model-Driven Testing*, Vieweg+Teuber Research, 2010

[BBK75] **Barzdins, J., Bicevskis, J., Kalnins, A.** Construction of complete sample system for correctness testing. ***In:*** *Proc. MFCS 1975. LNCS*, vol. 32, Berlin / Heidelberg: Springer, 1975, pp. 1–12

[BBS+79] **Bičevskis, J., Borzovs, J., Straujums, U., Zariņš, A., Miller, E. F.** SMOTL – A System to Construct Samples for Data Processing Program Debugging. *IEEE Trans. Softw. Eng.*, 1979, 5, 1, pp. 60–66

[Bec00] **Beckerman, L.P.** Application of complex systems science to systems engineering, *Systems Engineering*, 3, 2, John Wiley & Sons, Inc., 2000, pp. 96–102

[Bei90] **Beizer, B.** *Software Testing Techniques*. 2nd ed. New York: Van Nostrand Reinhold, 1990

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

[Ber07] **Bertolino, A.** Software Testing Research: Achievements, Challenges, Dreams. ***In:*** *2007 Future of Software Engineering (FOSE '07),* IEEE Computer Society, Washington, DC, USA, 2007, pp. 85-103

[BG01] **Baron, M., Girard, P.** Bringing Robustness to End-User Programming. ***In:*** *Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, IEEE Computer Society, Washington, DC, 2001, pp. 142-149

[Bin99] **Binder, R.V.** *Testing Object-Oriented Systems: Models, Patterns, and Tools,* The Addison-Wesley Object Technology Series, Addison-Wesley, 1999

[Bla02] **Black, R.** Managing the testing process: Practical Tools and Techniques for Managing Hardware and Software Testing, Wiley Publishing, 2nd ed., 2002

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

[Bla99] **Black, R.** *Managing the testing process* Microsoft Press, Redmond Washington, 1999

[BN03] **Broekman, B., Notenboom, E.** *Testing Embedded Software,* Addison-Wesley, 2003

[Boo07] **Booch, G.** Object-Oriented Analysis and Design with Applications. Addison-Wesley, 3rd ed., 2007

[BS87] **Basili, V.R., Selby, R.W.** Comparing the Effectiveness of Software Testing Strategies, *Software Engineering, IEEE Transactions on*, 13, 12, 1987, pp. 1278-1296

[Bur03] **Burnstein, I.** *Practical Software Testing. A Process oriented approach*, Springer-Verlag, New York Berlin Heidelberg, 2003

[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

[CA07] **Cheng, B. H. C., Atlee, J. M.** Research Directions in Requirements Engineering, ***In:*** *Future of Software Engineering (FOSE '07)*, IEEE Computer Society, 2007, 285–303

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

[CBG+04] **Cohen C. F., Birkin, S. J., Garfield, M. J., Webb, H. W.** Managing Conflict in Software Testing, *Communications of ACM*, vol. 47, no 1, January 2004, pp. 76-81

[CF00] **Conradi, R., Fuggetta, A.** Improving Software Process Improvement, *IEEE Software*, vol. 17, no. 4, IEEE Computer Society, July/Aug. 2000, pp. 76–78

[CHR82] **Clarke, L. A., Hassell, J., Richardson, D. J.** A Close Look at Domain Testing, *IEEE Trans. Softw. Eng.*, 8, 4, 1982, pp. 380–390

[Cil98] **Cilliers, P.** Complexity and Postmodernism: Understanding Complex Systems, Routledge, London New York, 1998

[CJ02] **Craig, R. D., Jaskiel, S. P.** *Systematic Software Testing,* Artech House Publishers, 2002

[Cla76] **Clarke, L. A.** A System to Generate Test Data and Symbolically Execute Programs, *IEEE Trans. on Softw. Eng.*, vol. 2, no. 3, 1976, pp. 215–222

[Cla81] **Richardson, D. J., Clarke, L. A.** A partition analysis method to increase program reliability. In: Proceedings of the 5th international Conference on Software Engineering, *IEEE Press*, Piscataway, NJ, 1981, pp. 244–253

[Cop03] **Copeland, L.** *A Practitioner's Guide to Software Test Design*, Artech House, Inc., 2003

[CPR04] **Chen, Y., Probert, R.L., Robeson, K.** Effective test metrics for test strategy evolution, ***In:*** *CASCON '04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, IBM Press, Markham Ontario Canada, 2004, pp. 111-123

[CPR+85] **Clarke, L. A., Podgurski, A., Richardson, D., Zeil, S.** A comparison of data flow path selection criteria, **In:** *Proc. 8th ICSE*, 1985, pp. 244–251

[Cra02] **Craig, R. D., Jaskiel, S. P.** *Systematic Software Testing,* Boston / London: Artech House Publishers, 2002

[DPA+95] Datu pārraides un apstrādes sistēmas: Angļu-krievu-latviešu skaidrojošā vārdnīca, SWH, Rīga, 1995

[Dra05] **Dragan, M.** Using Excel as a front-end for MLF. Symbolic and Numeric Algorithms for Scientific Computing, ***In:*** *SYNASC 2005. Seventh International Symposium on* , 2005, pp. 114-117

[Dus02] **Dustin, E.** *Effective Software Testing: 50 Ways to Improve Your Software Testing,* Addison-Wesley Longman Publishing Co., Inc., 2002

[DV10] **Dunin-Kepli, B.M., Verbrugge R.** *Teamwork in Multi-Agent Systems: A Formal Approach*, Wiley, 2010

[EAC+05] **Erwig, M., Abraham, R., Cooperstein, I., Kollmansberger, S.** Automatic generation and maintenance of correct spreadsheets. ***In:*** *Proceedings of the 27th international conference on Software engineering (ICSE '05),* ACM, New York, 2005, pp. 136-145

[EM07] **Everett, G. D., McLeod, R., Jr.** *Software Testing: Testing Across the Entire Software Development Life Cycle,* Wiley-IEEE Computer Society Press, 2007

[Erw09] **Erwig, M.** Software Engineering for Spreadsheets, *Software, IEEE* , vol.26, no.5, 2009, pp. 25-30

[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

[FI98] **Frankl, P., Iakounenko, O.** Further Empirical Studies of Test Effectiveness, *ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering*, Nov. 1998, pp. 153-162

[FW88] **Frankl, P.G., Weyuker, E.J.** An Applicable Family of Data Flow Testing Criteria, *IEEE Trans. Softw. Eng.,* Vol.14, No.10, Oct. 1988, pp. 1483-1498

[FW93a] **Frankl, P.G., Weyuker, E.J.** An analytical comparison of the fault-detecting ability of data flow testing techniques, *Software Engineering, Proceedings., 15th International Conference on*, 1993, 415-424

[FW93b] **Frankl, P. and Weyuker, E. J.** A Formal analysis of the Fault Detecting Ability of Testing Methods, *IEEE Transactions of Software Engineering*, 1993, 19:3, pp. 202-213

[Gee04] **Geels, F.W.** "From sectoral systems of innovation to socio-technical systems Insights about dynamics and change from sociology and institutional theory, *Research Policy*, 33, 2004, pp. 897-920

[GH05] **Gershenson, C., Heylighen, F.** How can we think the complex?, ***In:*** *Managing Organizational Complexity: Philosophy, Theory and Application* (K.A. Richardson, ed.), Information Age Publishing, 2005, Ch. 3

[GH88] **Gelperin, D., Hetzel, B.** The growth of software testing, *Communications of the ACM,* Vol. 31, No. 6, ACM, 1988, pp. 687-695

[GOA05] **Grindal, M., Offutt, J., Andler, S. F.** Combination testing strategies: a survey, *Software Testing, Verification and Reliability*, 15, 3, Wiley and Sons Ltd., Chichester, UK, Sep. 2005, pp 167–199

[Gra86] **Grassberger, P.** Toward a quantitative theory of self-generated complexity, *International Journal of Theoretical Physics,* 25, 1986, pp. 907–938

[GTW+03] **Gao, J. Z., Tsao, J., Wu, Y., Jacob, T. H.** *Testing and Quality Assurance for Component-Based Software*, Artech House, Inc., 2003

[GU07] **Grobbelaar, S., Ulieru, M.** Complex networks as control paradigm for complex systems, ***In:*** *Proceedings of IEEE International Conference on Systems, Man and Cybernetics (SMC)*, Montreal, QC, Canada, 2007, pp. 4069-4074

[Ham89] **Hamlet, R.** Theoretical comparison of testing methods. **In:** Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification, TAV3, ACM, New York, NY, 1989, pp. 28–37

[Hey09] F. Heylighen, „Complexity and Self-organization, *Encyclopedia of Library and Information Sciences* (Bates, M.J., Maack, M.N. eds.), CRC Press, 2009

[HF98] **Hajnal, Á., Forgács, I.** An applicable test data generation algorithm for domain errors. **In:** *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '98* (Tracz, W. ed.) ACM, New York, NY, 1998, pp. 63–72

[Hon96] **Hong, Z.** A Formal Analysis of the Subsume Relation Between Software Test Adequacy Criteria, *IEEE Trans. Softw. Eng.* 22, 4, April 1996, pp. 248-255

[How75] **Howden, W. E.** Methodology for the Generation of Program Test Data, *IEEE Transactions on Computers*, vol. 24, no. 5, 1975, pp. 554–560

[How76] **Howden, W. E.** Reliability of the Path Analysis Testing Strategy, *IEEE Trans. Softw. Eng.* vol. 2, no. 3, 1976, pp. 208–215

[HT90] **Hamlet, D., Taylor, R.** Partition Testing Does Not Inspire Confidence (Program Testing), *IEEE Trans. Softw. Eng.*, 1990, 16, 12, pp. 1402–1411

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

[HFG+94] **Hutchins, M., Foster, H., Goradia, T., Ostrand, T.** Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria, *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on* , 1994, pp. 191-200

[HZJ06] **Huo, M., Zhang, H., Jeffery, R.** An Exploratory Study of Process Enactment as Input to Software Process Improvement, *Proceedings of the 2006 International Workshop on Software Quality,* 2006, Shanghai, pp. 39–44

[Jen00] **Jennings, N. R.** On agent-based software engineering, *Artifical Intelligence*, 117, 2000, pp. 277-296

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

[Jos04] **Jost, J.** External and internal complexity of complex adaptive systems, *Theory in Biosciences*, 123, 1, 2004, pp. 69-88

[JR00] **Joslyn, C., Rocha,** L.M. Towards semiotic agent-based models of socio-technical organizations, *Proceedings of AI, Simulation and Planning in High Autonomy Systems (AIS 2000)*, Tucson, Arizona, USA, 2000, pp. 70–79

[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

[JW89] **Jeng, B., Weyuker, E.** Some observations on partition testing ***In:*** *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*, ACM, New York, NY, 1989, pp. 38–47.

[JW94] **Jeng, B., Weyuker, E. J.** A simplified domain-testing strategy, *ACM Trans. Softw. Eng. Methodol.* 3, 3, 1994, pp. 254–270

[KAB+11] **Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M.B., Rothermel, G., Shaw, M., Wiedenbeck, S.** The state of the art in end-user software engineering, *ACM Comput. Surv.*, 43, 3, 2011, pp. 1-44

[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

[KT05] **Kleinberg, J., Tardos, E.** *Algorithm Design*, Addison-Wesley, Boston, 2005

[Lew05] **Lewis, W. E.** *Software Testing and Continuous Quality Improvement,* Auerbach Publications, 2nd ed., 2005

[LF03] **Link, J., Frolich, P.** *Unit Testing in Java: how Tests Drive the Code,* Morgan Kaufmann Publishers Inc., 2003

[LÖ09] *Encyclopedia of Database Systems* (Liu, L., Özsu, M. T. eds.), Springer-Verlag, 2009

[LPW06] *End-User Development* (Lieberman, H., Paterno, F., Wulf, V. eds.). Kluwer/ Springer, 2006

[LR03] **Lewin, R., Regine, B.** The Core of Adaptive Organisations, *Complex Systems and Evolutionary Perspectives on Organisations: The Application of Complexity Theory to Organisations* (Mitleton-Kelly, E. ed.), Pergamon, Elsevier Science Ltd, Amsterdam Boston, 2003, ch. 8

[May04] **Myers, G. J. The Art of Software Testing** 2nd ed., Hoboken, New Jersey: John Wiley & Sons, Inc., 2004

[MB89] **McCabe, T. J., Butler, C. W.** Design complexity measurement and testing. *Communications of. ACM* 32, 12, December 1989, pp. 1415-1425

[MBB06] **Minai, A.A., Braha, D., Bar-Yam, Y.** Complex Engineered Systems: A New Paradigm, *Complex Engineered Systems: Science Meets Technology (Understanding Complex Systems)*, (Minai, A.A., Braha, D., Bar-Yam, Y. eds.), Springer-Verlag New York, Inc, Secaucus, NJ, USA, 2006, ch. 1.

[MCC+87] **De Millo, R., McCracken, W. M., Martin, R. J., Passafiume, J.** *Software Testing and Evaluation,* Benjamin-Cummings Publishing Co., Inc., 1987

[Mit03] **Mitleton-Kelly, E.** Ten Princples of Complexity and Enabling Infrastructures, *Complex Systems and Evolutionary Perspectives on Organisations: The Application of Complexity Theory to Organisations*, (Mitleton-Kelly, E. ed.), Pergamon, Elsevier Science Ltd, Amsterdam Boston, 2003

[MKB06] **Myers, B., Ko, A.J., Burnett, M.M.** Invited research overview: end-user programming. ***In:*** *CHI '06 extended abstracts on Human factors in computing systems (CHI EA '06)*, ACM, New York, 2006, pp. 75-80

[MS01] **McGregor, J. D., Sykes, D. A.** *A Practical Guide to Testing Object-Oriented Software,* Addison-Wesley Longman Publishing Co., Inc., 2001

[MSC04] **Morasca, S. Serra-Capizzano, S.** On the analytical comparison of testing techniques, *SIGSOFT Softw. Eng. Notes*, 29, 4, July 2004, pp. 154-164

[MW04] **Magee, C. L., de Weck, O. L.** Complex System Classification, Fourteenth Annual International Symposium of the International Council on Systems Engineering (INCOSE), 2004, pp. 1-18

[Mye92] **Myers, J.P. Jr.** The complexity of software testing, *Software Engineering Journal*, vol. 7, no. 1, Jan. 1992, pp. 13-24

[NJH01] **Nguyen, H. Q., Johnsom, B., Hackett, M.** *Testing Applications on the Web: Test Planning for Mobile and Internet-Based Systems*, 2nd ed., John Wiley & Sons, Inc., 2001

[NK06] **Norman, D.O., Kuras, M.L.** Engineering Complex Systems, *Complex Engineered Systems: Science Meets Technology (Understanding Complex Systems)*, (Braha, D., Minai, A.A., Bar-Yam, Y. eds.), Springer-Verlag New York, Inc, Secaucus, NJ, USA, 2006, ch. 10.

[NSV+07] **Neto, A. C. D., Subramanyan, R., Vieira, M., Travassos, G.H.** A survey on model-based testing approaches: a systematic review. ***In:*** *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007* (WEASELTech '07). ACM, New York, NY, USA,2007, pp. 31-36

[Nta88] **Ntafos, S. C.** A comparison of some structural testing strategies, *IEEE Trans. Softw. Eng. SE*, 14, 1988, pp. 868–874

[Off92] **Offutt, A.J.** Investigations of the Software Testing Coupling Effect, *ACM Transactions on Software Engineering Methodology*, 1992, 1:1, pp. 15-20

[OG08] **Obrenovic, Z., Gasevic, D.** End-User Service Computing: Spreadsheets as a Service Composition Tool, *Services Computing, IEEE Transactions on* , vol.1, no.4, 2008, pp. 229-242

[OHa00] **O'Hara, F.** European Experiences with Software Process Improvement. ***In:*** *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, 2000, pp. 635–640

[Paj00] **Pajerek, L.** Processes and organizations as systems: when the processors are people, not pentiums, *Systems Engineering*,3, 2, John Wiley & Sons, Inc., 2000, pp. 103–111

[Par01] **Paradiso, M.** Software Verification & Validation Introduced. ***In:*** *Software Quality Approaches: Testing, Verification, and Validation* (Haug, M., Olsen, E. W., Consolini, L. eds), Springer-Verlag, 2001, pp. 36-45

[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

[PM06] **Palyagar, B., Moisiadis, F.** Validating Requirements Engineering Process Improvements – a Case Study, *First International Workshop on Requirements Engineering Visualization (REV'06 - RE'06 Workshop)*, 2006, pp. 9-18

[Pro03] **Prowell, S.J.** JUMBL: A tool for model-based statistical testing. ***In:*** *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, IEEE, 2003, pp. 331–345

[PV00] **Polacek, G. A., Verma, D.** Requirements Engineering for Complex Adaptive Systems: Principles vs. Rules. ***In:*** *Proceedings of the 7th Annual Conference on Systems Engineering Research CSER 2009* (Kalawsky, R.S., O'Brien, J., Goonetilleke, T., Grocott, C. eds.),

Loughborough University, UK, April 2009, Research School of Systems Engineering, Loughborough University, Loughborough, UK

[Rab08] **Rabinovich, R**. *Complexity Measures of Directed Graphs,* Diploma thesis, RWTH-Aachen, 2008

[Rac01] **Raccoon, L. B. S.** Definitions and demographics, *Software Engineering Notes*, Volume 26, Number 1. ACM Press, New York, 2001, pp. 82-91

[RC85] **Richardson, D. J., Clarke, L. A.** Partition analysis: a method combining testing and verification, *IEEE Trans. Softw. Eng. SE-11*, 12, 1985, pp. 1477–1490

[RR00] **Rosaria, S., Robinson, H.** Applying models in your testing process. *Information and Software Technology*, 42(12), September 2000, pp. 815-824

[RW85] **Rapps, S., Weyuker, E. J.** Selecting software test data using data flow information, *IEEE Trans. Softw. Eng. SE*, 11, 4, 1985, pp. 367–375

[Sky05] **Skyttner, L.** General Systems Theory: Problems, Perspectives, Practice, World Scientific, 2005

[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

[SM09] **Sheard, S.A., Mostashari, A.** Principles of complex systems for systems engineering, *Systems Engineering*, 12, 4, John Wiley and Sons Ltd, Chichester, UK, 2009, pp. 295-311

[Sol00] **Van Solingen, R.** Product Focused Software Process Improvement. SPI in the Embedded Software Domain, Eindhoven University of Technology, Eindhoven, 2000

[Tai80] **Tai, K. C.** Program Testing Complexity and Test Criteria, *Software Engineering, IEEE Transactions on* , **6,** 1980, pp. 531- 538

[TBW06] **Thorsten, P., Bose, F., Windt, K.** Autonomously controlled processes – characterisation of complex production systems, **In:** *Proceedings of 3rd International CIRP Conference on Digital Enterprise Technology (DET)* (Huang, G.Q., Mak, K.L., Maropoulos, P.G. eds.), Setubal, Portugal, 2006, ch. 28

[TC93] **Turner, J.R., Cochrane, R.A.** Goals-and-Methods Matrix: Coping with Projects with illDefined Goals and/or Methods of Achieving Them. *International Journal of ProjectManagement*, 11, 1993, pp. 93-102

[Tho98] **Thorup, M.** Structured programs have small tree-width and good register allocation, *Information and Computation*, 142, 1998, pp. 159–181

[TI11] **Tsui, F., Iriele, S.** Analysis of software cohesion attribute and test case development complexity, ***In:*** *Proceedings of the 49th Annual Southeast Regional Conference (ACM-SE '11)*, 2011, pp. 237-242

[UL07] **Utting, M., Legeard, B.** *Practical Model-based Testing: A Tools Approach*, Elsevier, 2007

[UPL05] **Utting, M., Pretschner, A., Legeard, B.** A Taxonomy of Model-Based Testing, *Technical report*, Department of Computer Science The University of Waikato, 2005

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

[VKB03] **Vilkomir, S. A., Kapoor, K., Bowen, J. B.** Tolerance of Control-Flow Testing Criteria. ***In:*** *Proceedings of the 27th Annual International Conference on Computer Software and Applications (COMPSAC '03),* IEEE Computer Society, Washington, DC, USA, 2003, 182-187

[Wat01] **Watkins, J.** *Testing IT: An Off-the-Shelf Software Testing Process*, Cambridge University Press, 2001

[WC80] **White, L. J., Cohen, E. I.** A Domain Strategy for Computer Program Testing, *IEEE Trans. Softw. Eng.*, 6, 3, 1980, pp. 247–257

[Wei89] **Weiss, S. N.** Comparing test data adequacy criteria, *SIGSOFT Softw. Eng. Notes* 14, 6, 1989, pp. 42–49

[WGM85] **Weiser, M. D., Gannon, J. D., McMullin, P. R.** Comparison of structured test coverage metrics, *IEEE Software,* 2, 2, 1985, pp. 80–85

[Wil99] **Williams, T.M.** The Need for New Paradigms for Complex Projects. *International Journal of Project Management*, 17, 5, 1999, pp. 269-273

[WJ91] **Weyuker, E. J., Jeng, B.** Analyzing Partition Testing Strategies, *IEEE Trans. Softw. Eng.* 17, 7, 1991, pp. 703–711

[WWH91] **Weyuker, E.J., Weiss, S.N.**, Hamlet, D. Comparison of program testing strategies, ***In:*** *TAV4: Proceedings of the Symposium on Testing, Analysis, and Verification,*. ACM, New York, British Columbia, Canada, Oct. 08-10, 1991, pp. 1-10

[XL04] **Xia, W., Lee, G.** Grasping the complexity of IS development projects, *Commun. ACM,* vol. 47, no. 5, ACM, New York, NY, USA, May 2004, pp. 68-74

[XL05] **Xia, W., Lee, G.** Complexity of Information Systems Development Projects: Conceptualization and Measurement Development. *J. Manage. Inf. Syst.* 22, 1, 2005, pp. 45-83

[YC02] **Yoder, A.G., Cohn, D.L.** *Domain-specific and general-purpose aspects of spreadsheet languages*, Distributed Computing Research Lab, University of Notre Dame, 2002

[ZA92] **Zeil, S. J., Afifi, F. H., White, L. J.** Detection of linear errors via domain testing. ACM Trans. Softw. Eng. Methodol. 1, 4, 1992, pp. 422–451

[ZHM97] **Zhu, H., Hall, P. A., May, J. H.** Software unit test coverage and adequacy, *ACM Comput. Surv.* 29, 4, 1997, pp. 366–427