VILNIAUS UNIVERSITETAS

MATEMATIKOS IR INFORMATIKOS FAKULTETAS

KOMPIUTERIJOS KATEDRA

Defektų analizės sistema

Magistrinis darbas

Atliko: 1M kurso 8 grupės studentė

Virginija Andrijauskaitė

Vadovas: Asist. Linas Būtėnas

Vilnius

2007

Turinys

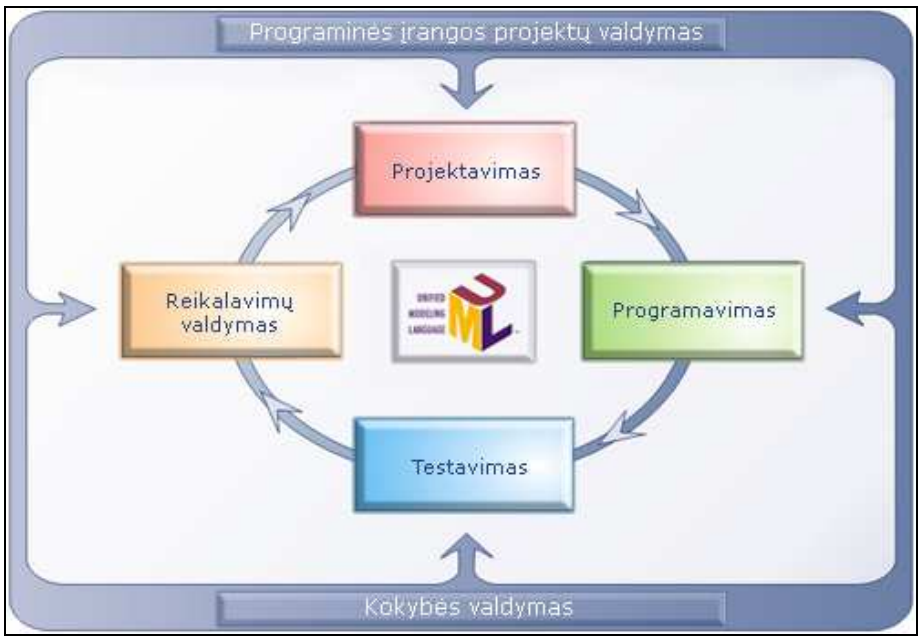
1. Įvadas

Nepaisant sparčios informacinių technologijų plėtros, programinės įrangos kūrimas yra palyginti nauja industrijos šaka, kurios praktikos nėra nusistovėjusios. Tiek kūrėjai, tiek užsakovai ieško sprendimų, kaip optimizuoti programinės įrangos kūrimo procesus, kad minimaliomis lėšomis būtų sukurtas kokybiškas, vartotojo poreikius atitinkantis produktas [SIL07].

Dažniausiai išskiriami keturi pagrindiniai programinės įrangos kūrimo etapai: **reikalavimų analizė**, **projektavimas**, **programavimas** ir **testavimas**. Šios veiklos formaliai ar neformaliai atliekamos visuose programinės įrangos projektuose:

- pradžioje analizuojama rinka, vartotojų poreikiai, trumpai tariant, atliekama vertinamoji rinkos(konkretaus kliento poreikių) analizė;
- vėliau remiantis gautais rezultatais(vertinamosios analizės išvadomis), sudaromas projektas, numatomas programos funkcionalumas, savybės, programos vystymo galimybės. Tai vadinamasis projektavimo etapas. Projektavimo etapui pasibaigus toliau darbai perduodami sekančiam etapui – produkto kūrimui;
- Atliekamas užsakytų darbų(papildomo funkcionalumo) programavimas;
- Sukurta programinė įranga ar papildomas programos funkcionalumas yra tikrinamas: ar jis tikrai atitinka projekto reikalavimus, jei funkcionalumas yra tinkamas, tikrinama ar nėra situacijų, kai programa jau nebeatitinka reikalaujamo funkcionalumo. Suprantama, jei šiame etape randama defektų, tuomet vėl grįžtame prie trečiojo etapo. Rasti netikslumai yra koreguojami pagal pateiktas pastabas. Atlikus pataisymus, vėl pakartotinai patikrinama, ar programa neturi trūkumų, jei testavimo etape vėl randama defektų prieš tai aprašytas žingsnis yra kartojamas tol, kol tikrintojų grupė nuspręs, jog visos joje esančios funkcijos veikia, taip kaip ir turi veikti.

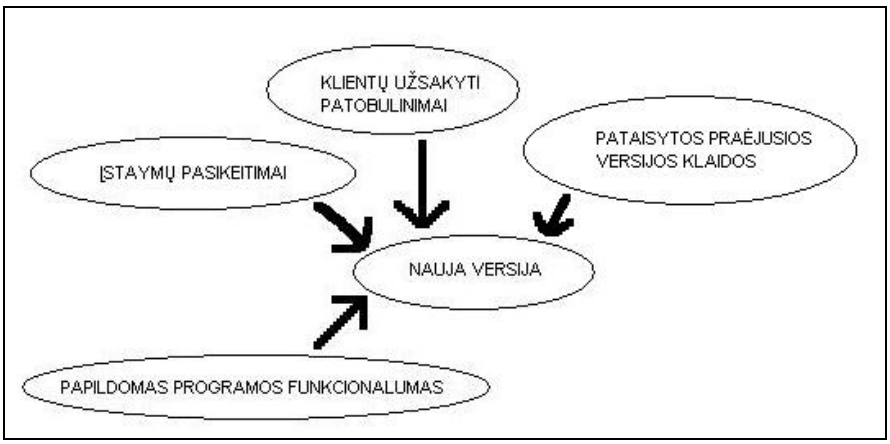
Perėjus visas keturias kūrimo pakopas programinis produktas ar papildomas jo funkcionalumas gali būti pateiktas, rinkai arba klientui užsisakiusiam jį. Trumpai visą šį procesą galime pamatyti paveikslėlyje Nr. 1



Paveikslas Nr. 1 – Programinės įrangos kūrimo procesas

Tokia pati arba labai panaši veiksmų seka, naudojama tuo atveju, kai programinis produktas yra tobulinamas, t.y. planuojama išleisti nauja programos versija, turinti papildomą funkcionalumą. Vėlgi, pirmiausia, analizuojama, kokių funkcijų vartotojams reikėtų ar kokios funkcijos palengvintų darbą su pačia programa. Taip pat atsižvelgiama ir į vyriausybės nutarimų pakeitimus, jei pvz. Ataskaitos pateikiamos programoje arba skaičiavimai vykdomi programoje yra reglamentuojami valstybės nutarimais. Atlikus vertinamąją analizę vyksta projektavimo etapas. Šiuo metu siekiama, kuo mažiau keičiant programos struktūrą atlikti suplanuotus pakeitimus. Suprojektavus toliau vykdomi programavimo darbai. O po jų seka programos tikrinimas, kaip ir prieš tai buvusiu atveju, taip ir tobulinant programą, jei patikrinimo metu randama klaidų, neatitikimų suprojektuotiems ir aprašytiems reikalavimams klaidos yra dokumentuojamos ir perduodamos programuotojams taisyti. Atlikus nurodytus pataisymus vėl tikrinama ir jei randama neatitikčių vėl atliekami pataisymai. Kaip ir pradiniu atveju paskutinis žingsnis **„Programavimas“ – „Testavimas“** kartojamas, kol tikrinimo etape bus nuspręsta, jog daugiau nebereikia programos grąžinti taisymui.

Pagrindiniai kriterijai, kuriais yra remiamasi, planuojant naujos versijos funkcionalumą pavaizduoti paveikslėlyje Nr. 2



Paveikslas Nr. 2 – Naujos versijos planavimo kriterijai

Tolesniame darbe labiau gilinsiuosi ir nagrinėsiu būtent testavimo procesą. Testavimas apima tiek galutinio programinės įrangos produkto, tiek tarpinių artefaktų tikrinimą, siekiant įvertinti, ar pasiekti rezultatai tenkina lūkesčius. Testavimas gali daug kartų padidinti programinės įrangos kokybiškumą. Nors dažnai yra galvojamas, jog testavimas nedaro didelės įtakos produkto sėkmei.

2. Tikslai

Darbo tikslas yra suprojektuoti sistemą. Suteikiančią vykdomajai programinės įrangos kūrimo bei diegimo grupei galimybę pažiūrėti informaciją apie projekto eigą apie projekto vykdymo laiką, trukmes apie dar programose rastų defektų būsenas. Taip pat testavimo sąlygų valdymo įrankį, kuris būtų plačiai pritaikomas(ne tik vienos srities programinių produktų testavimui). Pagrindiniai darbo tikslai būtų tokie:

- Susipažinti su esamais testavimo modeliai, metodais bei testavimo strategijomis;
- Sukurti įrankį, kurio dėka atsakius į pateiktus klausimus, testuotojui ar kitam darbo komandos dalyviui būtų lengviau parinkti efektyviausią testavimo metodą(strategiją).
- Susipažinti su šiuo metu rinkoje esančiomis defektų bei testavimo sąlygų valdymo sistemomis. Išanalizuoti jų funkcijas, privalumus bei trūkumus.
- Sumodeliuoti sistemą, turinčią galimybę kurti įvairius testavimo šablonus, jiems priskirti testavimo sąlygas, o testuotojams galimybę pažymėti ar tam tikroje programos versijoje testavimo sąlyga yra patenkinama ar ne.

- Padaryti galimybę sistemoje gauti įvairią informaciją pagal pateiktas testavimo išvadas(pvz. versijoje esančių klaidų skaičius procentais, dviejų versijų palyginimas - kas patobulinta, o kokie nauji programos defektai buvo rasti, ar viršytas numatytas projekto įvykdymo valandų skaičius ir k.t.);

3. Testavimas

Gerai žinoma tiesa - programinės įrangos be defektų nebūna [SIL07]. Todėl natūralu, jog atliekant programavimo darbus neišvengiamai susiduriama su problema, kad ne visada pavyksta išvengti klaidų arba projektavimo etape ne visada yra numatomos visos įmanomos situacijos, taip pat pasitaiko programos klaidų, kurias galima pastebėti tik tam tikra tvarka atlikus veiksmų seką. Net, jei programa yra nedidelė turinti vos keletą funkcijų neįmanoma iš pirmo karto numatyti visų situacijų ar vartotojo atliekamų veiksmų, o ką jau kalbėti kai programinis produktas yra didelis, sudarytas iš kelių modulių, kurių kiekvieną programuoja keletas žmonių. Todėl ir sudėjus visas programos dalis į vieną reikalingas dar vienas svarbus etapas, kol programinis produktas ar nauja jo versija bus pateikti vartotojams. Šis etapas vadinamas testavimu.

Testavimas – tai procesas, kai dirbama su programa tikintis rasti klaidų. Radus klaidų šis procesas vadinamas sėkmingu, nes randama problema, vadinasi, ji bus pašalinta. Kitaip tariant testavimo metu neradus klaidų – laikoma, jog testavimo procesas buvo nesėkmingas, nes tai, jog nebuvo rasta klaidų visai nereiškia, jog programoje klaidų nėra. Klaidų būna visada, nes klaida yra ne tik tada, kai programa daro ne taip kaip turėtų daryti, bet ir kai daro tai ko neturėtų daryti [MYE04]. Pavyzdžiui paimkime paprasčiausią pajamų mokesčio apskaičiavimo algoritmą. Paveikslėlyje Nr. 3 matome procedūrą, kuri nuo darbuotojui priskaitytos sumos paskaičiuoja pajamų mokestį.

```
ProcPajMok(PriskPajMok :real){

        Jei [Darboviete = Pagrindine] Tada
        {
                ApskSuma = (PriskPajMok - NeapmokMin_DARB - NeapmokMin_PAJMOK) * 0.33
        }

        Jei [Darboviete <> Pagrindine] Tada
        {
                ApskSuma = PriskPajMok  * 0.33
        }

        return ApskSuma
}
```

Paveikslas Nr. 3 – Pajamų mokesčio paskaičiavimo procedūra.

Šiuo konkrečiu atveju procedūra veiktų klaidingai, jei darbuotojui paskaičiuotų ne 33% pajamų mokesčio, kaip iš jos tikimasi. Taip pat procedūroje yra klaida kadangi nėra numatytas tas atvejis, kai paskaičiuojamas neigiamas pajamų mokestis. Sumodeliavus tokia situacija procedūra atliks veiksmus, kurių neturėtų daryti – paskaičiuos neigiamą pajamų mokestį.

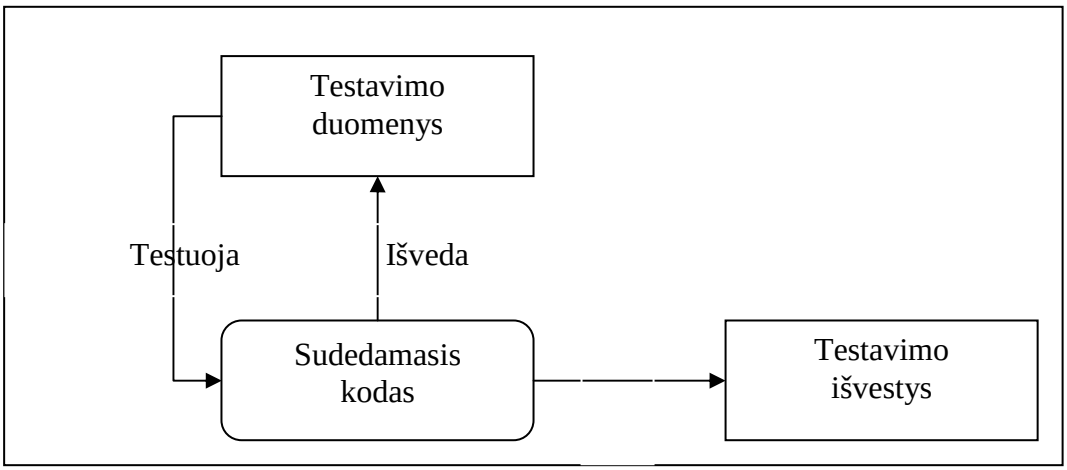
Visų programinės įrangos testavimo metu yra naudojamos dvi pagrindinės – „Baltos dėžės“ ir „Juodos dėžės“ – testavimo technologijos.

Dažniausiai siūloma kombinuoti jų abiejų siūlomus testus. „Baltos dėžės“ testavimo technologija yra orientuota į patį programos kodą, o „Juodos dėžės“ testavimo technologija tikrina, ar programos išeities duomenys atitinka įeities duomenis ir jau nebesigilina į pačios programos struktūrą. Būtent dėl šios priežasties, „Baltos dėžės“ testavimo technologijų siūlomus testus rekomenduojama atlikti patiems programinės įrangos kūrėjams, o „Juodos dėžės“ testą – patyrusiems testuotojams. Dabar į kiekvieną minėtųjų technologijų pažvelgsime šiek tiek įdėmiau [BUR02].

3.1 „Baltos dėžės“ testavimo technologija

Kaip jau minėta, „Baltos dėžės“ testavimo technologija gilinasi į pačios programos struktūrą. Dėl šios priežasties ji dar kitaip yra vadinama struktūrinio testavimo technologija.

Baltos dėžės“ testavimo technologija yra labiau paplitusi ir daug dažniau naudojama nei „Juodos dėžės“ testavimo technologija. Taip yra todėl, kad „Baltos dėžės“ siūlomus testus dažniausiai atlieka patys programuotojai, o testavimo komandos, turinčios pasirūpinti „Juodos dėžės“ testavimo technologijų siūlomų testų atlikimu, vis dar nėra sudaromos kiekvienoje šios dienos įmonėje.



Paveikslas Nr. 4 – „Baltos dėžės“ testavimo technologija

Kaip matome paveikslėlyje Nr. 4, taikant „Baltos dėžės“ testavimo technologiją, testiniai atvejai išvedami iš pačios programos struktūros. Šiuo metodu atliktas testavimas yra tikslus tik tada, kai testą atliekantis darbuotojas tiksliai žino, ką programa turi daryti ir kaip ji turi veikti. Taigi labai svarbi tampa reikalavimų specifikacija. Klaidos joje ar neteisinga jos interpretacija gali lemti abejotiną programinės įrangos testavimą kokybę.

Svarbu paminėti ir tai, kad „Baltos dėžės“ testavimo pavyzdžiai negali būti apibrėžti, kol nebus galutinai baigtas rašyti programų kodas. Šio testavimo tikslas yra išbandyti visus galimus programos operatorius.

Pagrindiniai „Baltos dėžės“ testavimo technologijos pranašumai:

- testuojamos visos programos dalys;
- testiniai atvejai išvedami iš pačios programos struktūros.

„Baltos dėžės“ testavimo technologijos trūkumai:

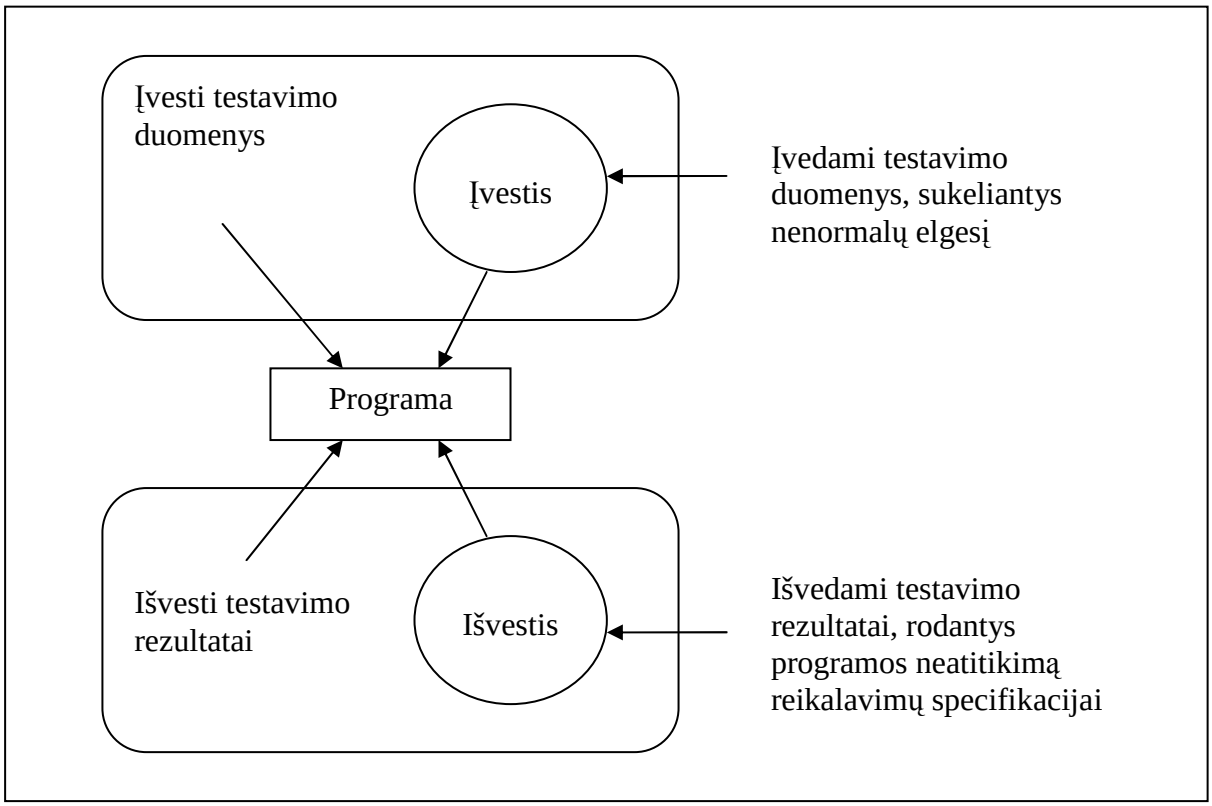
- dažniausiai „Baltos dėžės“ testavimo technologijos testą gali atlikti tik patys programinės įrangos kūrėjai;
- „Baltos dėžės“ testavimo metodu atliktas testavimas tikslus tik tada, kai testą atliekantis darbuotojas tiksliai žino, ką programa turi daryti ir kaip ji turi veikti;
- „Baltos dėžės“ testavimo pavyzdžiai negali būti apibrėžti, kol nebus galutinai baigtas rašyti programų kodas.

„Baltos dėžės“ testavimo technologijų siūlomus testus rekomenduojama kombinuoti su „Juodos dėžės“ testavimo technologijų testais.

3.2 „Juodos dėžės“ testavimo technologija

„Juodos dėžės“ testavimo technologija priešingai nei „Baltos dėžės“ testavimo technologija neatsižvelgia į testuojamos programos vidinę struktūrą. Ji remiasi tik programos reikalavimų specifikacija, todėl testą atliekantis darbuotojas tikrina, ar išeities duomenys atitinka įeities duomenis, nesigilindamas į programos kodą. Iš čia kilęs ir šio metodo pavadinimas: programa įsivaizduojama kaip „juoda dėžė“, kaip matome paveikslėlyje Nr. 5. Dar labai svarbu paminėti, kad „Juodos dėžės“ testavimas gali būti pradėtas labai ankstyvose programinės įrangos kūrimo stadijose.

Vienas labiausiai paplitusių „Juodos dėžės“ testavimo technologijos testų yra ekvivalentinis dalinimas. Jo metu įvesties duomenys ir išvesties rezultatai paskirstomi į atskiras klases, kuriose visų klasių nariai yra tarpusavyje susieti. Programos elgesys su kiekvienu nariu yra lygiai toks pats. Testiniai atvejai pasirenkami iš kiekvienos klasės.



Paveikslas Nr. 5 - „Juodos dėžės“ testavimo technologija

„Juodos dėžės“ testavimo technologijos pranašumai:

- technologija labiau tinkama tikrinant ilgus programų kodus;
- testą atliekantiems darbuotojams nebūtina turėti specifinių žinių apie pačią programos struktūrą ir mokėti atitinkamas programavimo kalbas;
- testuotojas ir programuotojas gali dirbti nepriklausomi vienas nuo kito;
- testas atliekamas, remiantis vartotojišku požiūriu;

- testavimo pavyzdžiai apibrėžiami jau paruošus reikalavimų specifikaciją.

„Juodos dėžės“ testavimo technologijos trūkumai:

- patikrinti galima tik apibrėžtą įvedamų duomenų kiekį (norint patikrinti kiekvieną įmanomą įvesties duomenį (atvejį), procesas tęsis amžinai);
- labai sunku suprojektuoti testavimo pavyzdį neturint trumpos ir aiškios reikalavimų specifikacijos;
- įvesties duomenų bandymai būtini net jei testuotojas žino, kad programuotojas jau išbandė visus testavimo pavyzdžius;
- dauguma programos dalių iš viso nėra testuojamos;
- jei programos kodas išties yra labai sudėtingas, klaidų tikimybė taip pat išlieka labai didelė;
- atlikti „Baltos dėžės“ testavimo technologijos testus dažniausiai vis tiek yra būtina.

Taigi, remdamiesi „Juodos dėžės“ testavimo strategija tikriname, ar įeities duomenys atitinka išeities duomenis. Kyla klausimas, kaip tinkamai parinkti įeities duomenis iš visų galimų jų variantų, kad aptiktumėme kuo daugiau programinės įrangos defektų? Sumanaus testuotojo užduotis ir yra suprojektuoti tokias testavimo sąlygas. Jai (šiai užduočiai) įgyvendinti galime pasirinkti iš „Juodos dėžės“ strategijos siūlomų metodų. Skirtingos defektų rūšys aptinkamos pasirinkus tinkamą siūlomų metodų kombinaciją. Vieni jų (metodų) yra praktiškesni ir dažniau taikomi nei kiti.

Juodos dėžės“ metodai:

- ekvivalentinis dalinimas;
- ribinių reikšmių analizė;
- perėjimų tarp būsenų testavimas;
- priežasčių ir pasekmių analizė;
- klaidų spėjimai.

3.2.1 Atsitiktinis generavimas

Atsitiktinai generuojant įeities duomenis pasirenkamos konkrečios reikšmės iš visų galimų kintamųjų kategorijų. Pavyzdžiui, jei leistinos programinės įrangos modulio įeities reikšmės yra visi sveikieji skaičiai nuo 1 iki 100, pasirenkamos trys atsitiktinės reikšmės: 55, 24 ir 3. Kyla klausimai:

- ar šios trys atsitiktinės reikšmės tikrai padės nuspręsti, ar programinės įrangos modulis atitinka savo specifikaciją (gal reikia pasirinkti daugiau, o gal mažiau įeities reikšmių);
- ar yra kitų reikšmių, kurios būtų labiau linkusios atskleisti programinės įrangos modulio defektus (gal reikia tikrinti ribines reikšmes);
- ar reikia įtraukti reikšmių, kurios nepatenka į sveikųjų skaičių nuo 1 iki 100 kategoriją (t.y. trupmenų, neigiamų reikšmių nuo 1 iki 100, reikšmių daugiau nei 100).

Atsitiktinis įeities reikšmių generavimas, žinoma, gali padėti sutaupyti nemažai laiko, tačiau programinės įrangos testavimo specialistai mano, kad labai maža tikimybė, kad atsitiktinai sugeneruotos įeities reikšmės atskleis tikruosius programinės įrangos defektus. Rekomenduojama naudoti labiau struktūrizuotus „Juodosios dėžės“ strategijos metodus.

3.2.2 Ekvivalentinis dalinimas

Ekvivalentinio dalinimo metodu įvesties duomenys paskirstomi į atskiras klases. Lygiai taip pat gali būti paskirstomi ir išvesties duomenys, tačiau tai jau nėra taip dažnai daroma. Klasių skaičius yra baigtinis. Ekvivalentinis dalinimas leidžia pasirinkti po vieną narį iš kiekvienos klasės kaip ją reprezentuojantį atstovą. Laikoma, kad programos elgesys su kiekvienu nariu yra lygiai toks pats (ekvivalentiškas).

Jei pasirinktas klasės narys neatskleidžia jokių programinės įrangos defektų, sakoma, kad visa klasė neturi defektų. Ir atvirkščiai, jei pasirinktas narys atskleidžia programinės įrangos defektus, to galime tikėtis ir iš kitų tos klasės narių. Žinoma, labai svarbu, kad klasės būtų tinkamai identifikuotos, nes jei viena klasė apima dvi klases, rezultatai nebus tikslūs.

Šio metodo pranašumai:

- išsamus testavimas netenka prasmės, nes užtenka patikrinti po vieną įvesties duomenų klasės narį;
- ekvivalentinis dalinimas padeda testuotojui rasti tas įvesties duomenų klases, kurios turi didžiausią tikimybę atskleisti programinės įrangos defektus;
- testuotojui suteikiamos galimybės patikrinti programinę įrangą turinčią didelį įvesties duomenų skaičių.

Įvesties duomenys dažniausiai paskirstomi į klases remiantis programinės įrangos reikalavimų specifikacija. Taigi testuotojas gali pradėti ruoštis „Juodos dėžės“ strategijos ekvivalentinio dalinimo testams labai anksti, vos tik pradėjus analizuoti vartotojų poreikius.

Svarbu įsidėmėti, kad:

1. įvesties duomenis paskirstydamas į atskiras klases, testuotojas turi nepamiršti ir tų klasių, kurios reprezentuotų klaidingus ir nenumatytus įvesties duomenis;
2. išvesties duomenys taip pat gali būti paskirstomi į atitinkamas klases;
3. skirtingi testuotojai pagal tą pačią reikalavimų specifikaciją tos pačios programos įvesties duomenis gali paskirstyti į skirtingas klases: nėra bendrų taisyklių;
4. tam, kad būtų tinkamai išskirtos duomenų klasės, labai svarbi programinės įrangos reikalavimų specifikacija. Būtina bendradarbiauti su ją rašančiais sistemos analitikais ir kilus neaiškumams, susisiekti su klientais. Kartais įvesties duomenis paskirstyti į klases gali būti itin sudėtinga.

Kaip galima būtų įvesties duomenis suskirstyti į atskiras klases?

1. jei įvesties duomenimis gali būti tam tikra kintamųjų sritis, viena klasė turėtų būti sudaroma iš jos, o kitos dvi už tos srities ribų – pradžios ir pabaigos. Pavyzdžiui, jei tam tikro programinės įrangos modulio kintamųjų sritis yra nuo 1 iki 499;
2. jei įvesties duomenimis gali būti tam tikras kintamųjų skaičius, viena klasė turėtų būti sudaroma iš leistinų kintamųjų, o kitos dvi už leistinų kintamųjų ribų. Pavyzdžiui, jei namo savininkais gali būti 1, 2, 3 arba 4 žmonės;
3. jei įvesties duomenimis gali būti tam tikras kintamųjų rinkinys, viena klasė turėtų būti sudaroma iš to rinkinio reikšmių, o kita – iš visų kintamųjų reikšmių nepatenkančių į minėtąjį rinkinį;
4. jei programinės įrangos modulio kažkurios įvedamos reikšmės turi tam tikrus apribojimus, o kitos – ne, viena klasė turėtų apimti visas reikšmes su tam tikrais apribojimais, o kita – tų apribojimų neatitinkančias. Pavyzdžiui, jei viename programos lauke įvedamas simbolis būtinai turi prasidėti raide;
5. jei tikima, kad įvesties duomenys yra paskirstyti į klases, kuriose jos nariai nėra traktuojami lygiai taip pat (programos elgesys su kiekvienu

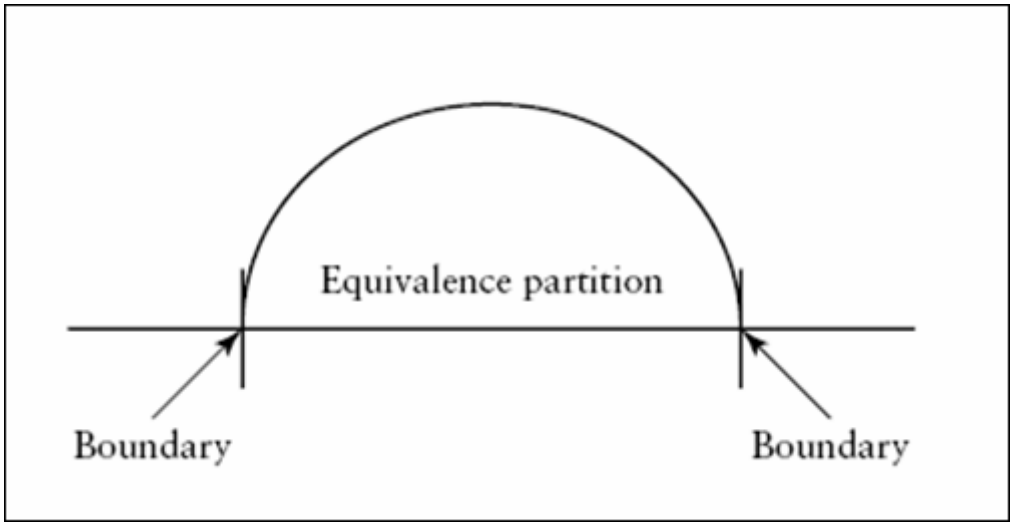
nariu toje pačioje klasėje turi būti ekvivalentiškas), būtina tokias klases dar skaidyti.

Pavyzdžiui, kokias klases išskirsime kvadratinę šaknį traukiančios funkcijos įvesties duomenims?

1. Įvesties duomuo yra realus skaičius.
2. Įvesties duomuo nėra realus skaičius.
3. Įvesties duomuo yra daugiau nei nulis.
4. Įvesties duomuo yra mažiau nei nulis.

3.2.3 Ribinių reikšmių analizė

Ekvivalentinio dalinimo metodo pagalba suprojektuotos testavimo sąlygos gali būti papildomos atliekant ribinių reikšmių analizę. Testuotojų patirtis rodo, kad labai dažnai programinės įrangos defektų priežastys glūdi jau išskirtų klasių sandūroje.



Paveikslas Nr. 6 – Ekvivalentinio dalinimo ribiniai atvejai

Kaip žinome, ekvivalentinio dalinimo pagalba įvesties ir išvesties duomenys suskirstomi į klases, o vėliau testiniai atvejai sudaromi iš jų pasirinkus po vieną narį. Ribinių reikšmių analizės atveju pasirenkamos tos reikšmės, kurios yra arti ribinių arba ribinės skirtingoms įvesties duomenų klasėms.

Kaip atliekama ribinių reikšmių analizė?

1. jei įvesties duomenimis gali būti tam tikra kintamųjų sritis, rekomenduojama suprojektuoti testavimo sąlygas tos srities galimoms ribinėms reikšmėms ir reikšmėms, kurios yra arti minėtųjų, bet jau

nebėra galimos. Pavyzdžiui, jei įvesties duomenys apibrėžiami sritimi nuo -1 iki 1, rekomenduojama testuoti tokias ribines reikšmes kaip -1.1, -1, 1, 1.1;

2. jei įvesties duomenimis gali būti tam tikras kintamųjų skaičius, rekomenduojama suprojektuoti testavimo sąlygas, kurios apimtų mažiausią ir didžiausią galimas reikšmes bei testavimo sąlygas, kurios tikrintų vienu vienetu mažesnę reikšmę nei mažiausia ir vienu vienetu didesnę nei didžiausia;
3. jei įvesties ir išvesties duomenys yra tam tikras reikšmių rinkinys, rekomenduojama koncentruoti dėmesį į sąrašo pradžią ir pabaigą.

Ekvivalentinis dalinimas ir ribinių reikšmių analizė yra labai populiari, taikoma tiek procedūrine, tiek objektine programavimo kalbomis parašytai programinei įrangai. Visgi, tik taikydami kelis metodus, galime tikėtis, kad aptiksime daugiau programinės įrangos defektų. Kiti „Juodos dėžės“ strategijos metodai yra: priežasčių ir pasekmių analizė, perėjimų tarp būsenų testavimas, klaidų spėjimai.

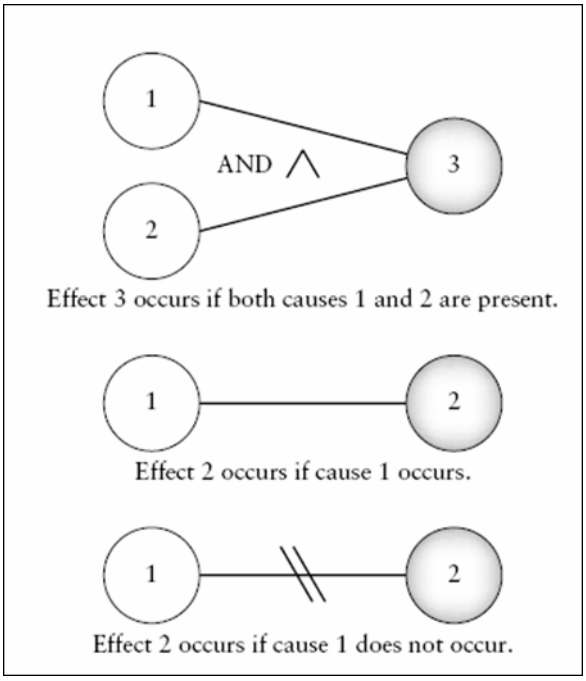
3.2.4 Priežasčių ir pasekmių analizė

Priežasčių ir pasekmių analizė leidžia kombinuoti skirtingų klasių narius tarpusavyje (tai, kad to nebuvo galima daryti ekvivalentinio dalinimo metu, buvo vienas didžiausių šio metodo trūkumų) ir taip suprojektuoti tokias testavimo sąlygas, kurios atskleistų neatitikimus reikalavimų specifikacijai. Tam reikalavimų specifikacija turėtų būti perbraižoma į loginį duomenų grafą.

Kaip atliekama priežasčių ir pasekmių analizė?

1. sudėtingos programinės įrangos reikalavimų specifikacija turėtų būti skaidoma į mažesnes jos dalis;
2. kiekvienam reikalavimų specifikacijos vienetui turėtų būti nustatomos jo priežastys ir pasekmės, jų priklausomybės;
3. sudaromi priežasčių ir pasekmių grafai. Naudojami standartiniai operatoriai: ir, arba, ne;
4. nurodoma, kokios priežastys ir pasekmės dėl vienų ar kitų aplinkybių yra negalimos;
5. priežasčių ir pasekmių grafas perbraižomas į sprendimų lentelę;

6. sprendimų lentelės stulpeliai transformuojami į konkrečias testavimo sąlygas.

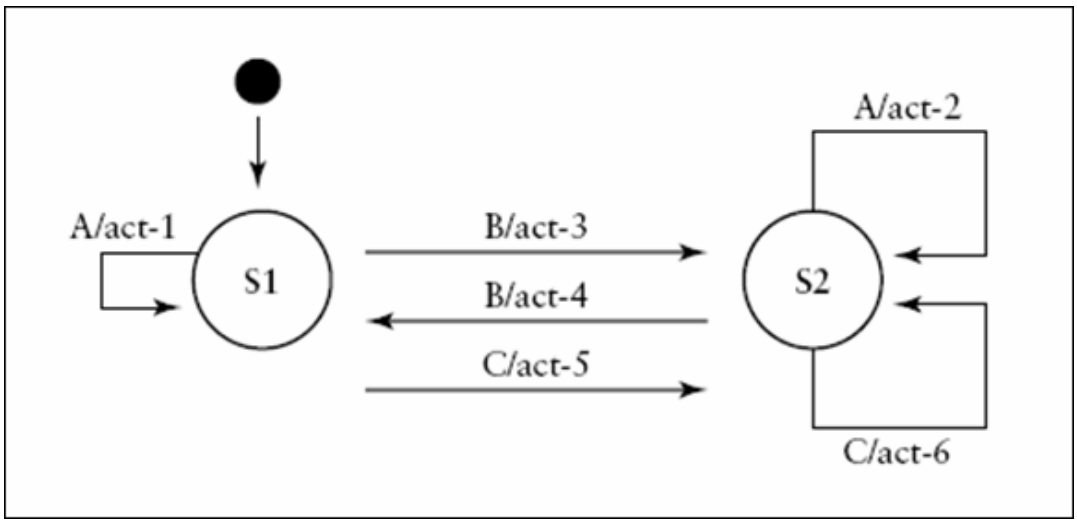


Paveikslas Nr. 7 – priežasčių grafai

3.2.5 Perėjimų tarp būsenų testavimas

Perėjimų tarp būsenų testavimas taip pat puikiai tinka tiek procedūrine, tiek objektine programavimo kalba parašytai programinei įrangai. Šis metodas leidžia testuotojams žiūrėti į programinės įrangos kūrimą, kaip į tam tikras būsenas, perėjimą tarp jų ir jas sužadinančius įvesties duomenis bei įvykius. To neleidžia daryti nei vienas anksčiau minėtųjų metodų.

Braižomi perėjimų tarp būsenų grafai.



Paveikslas Nr. 8 - Paprastas perėjimo tarp būsenų testavimo grafas

Testavimo metu nagrinėjami visi perėjimai tarp būsenų. Būsenų gali būti labai daug.

3.2.6 Klaidų spėjimai

Kaip jau sufleruoja pats metodo pavadinimas, klaidų spėjimai remiasi testuotojo patirtimi testuojant panašias į esamą programas ir jųjų intuicija. Testuotojų protingi spėjimai gali padėti atskleisti tuos programinės įrangos defektus, kurių neatskleidė kiti „Juodos dėžės“ strategijos testavimo metodai. Šis metodas gali būti itin naudingas, jei prieš tai testuotos programinės įrangos defektai buvo kruopščiai dokumentuojami.

Klaidų spėjimo metodo pavyzdžiu galėtų būti paprasčiausia dalyba iš nulio.

4 Defektų valdymas

Testavimas – tai įrankis kokybei pamatuoti. Jo metu mes siekiame įvertinti, ar programinė įranga atitinka apibrėžtus kokybės kriterijus ir nustatyti problemas(**defektus**), dėl kurių sistema negali atitikti vartotojo lūkesčių. Testavimo metu naudojant įvairias testavimo metodikas bei technologijas, pastebimi įvairūs programos trūkumai. Todėl kyla poreikis atrastus defektus ne tik komunikuoti, bet ir valdyti juos. Defektų valdymui visų pirma reikalingas jų dokumentavimas. Tokiu būdu defektų komunikavimas ir valdymas tampa būtina testavimo proceso dalimi, o **defekto ataskaita** – technine dokumentacija, aprašančia defekto simptomus, kai sistema elgiasi ne taip, kaip turėtų.

Kodėl defektų komunikavimo ir raportavimo procesas yra toks svarbus? Defektų ataskaita yra ne tik aprašomi klaidos simptomai, bet ir:

1. Įvertinama **rizika**, kad klaidą pastebės tipinis sistemos vartotojas;
2. Nurodoma **žala**, kurią klaida daro vartotojui ir sistemai;
3. Izoliuojant defektą padedama tiksliau nustatyti **klaidos ištaisymo kaštus** ir riziką.

Ši defektų ataskaitose pateikta informacija padeda tiksliau nustatyti testuojamo produkto kokybės lygį, įvertinti reikalingų pataisymų kiekį ir su jais susijusius kaštus.

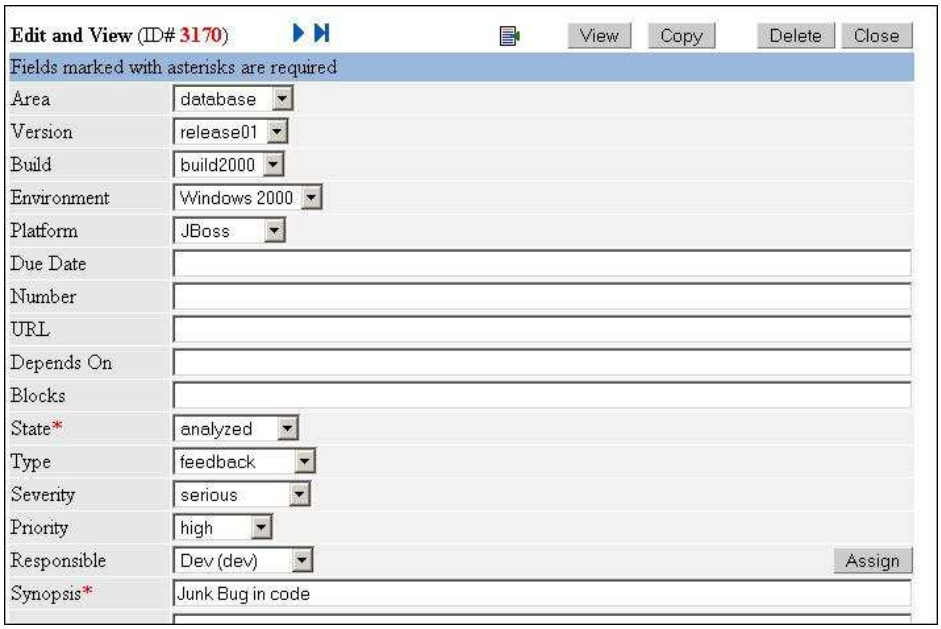
Kadangi visų pastebėtų defektų iškart ištaisyti neįmanoma, todėl juos visus reikia užregistruoti. Defektai turi būti grupuojami pagal **prioritetą**, kuris nurodo, kaip dažnai defektas pasitaiko vartotojui, ir **žalingumą**, kuris nusako defekto poveikį sistemai. Defekto aprašymas turi būti aiškus, kad defektą būtų galima pakartoti ir ištaisyti. Neužtenka tik aprašyti ir pataisyti defektus – reikalingas nuoseklus defektų valdymas. Kiekvienas defektas turi būti **išsprendžiamas**, tačiau nebūtinai turi būti ištaisymas – kai kurie defektai atidedami, kiti yra nepataisomi, dar kiti atmetami, jeigu jie užregistruojami dėl klaidingo testuojančio žmogaus supratimo apie sistemą. Ištaisius defektą, reikia patikrinti, ar jis buvo tinkamai ištaisytas ir ar nesąlygojo naujų defektų atsiradimo. Todėl efektyviam testavimui užtikrinti būtina naudoti **defektų registravimo** ir **valdymo sistemą** bei nustatyti defektų užregistravimo, taisymo, patikrinimo ir jų gyvavimo užbaigimo procedūras.

4.1 Defektų valdymo sistemos

Programinės įrangos defektų valdymo sistemos ne tik padeda sutrumpinti ir valdyti defektų gyvavimo ciklą. Duomenų apie defektus analizė gali mums padėti įvertinti testuojamos sistemos kokybės lygį ar jos tinkamumą atiduoti vartotojui. Vienas iš testavimo tikslų yra **defektų prevencija** – priemonių, padedančių sumažinti klaidų kiekį vėlesniuose programinės įrangos etapuose, paieška ir taikymas. Analizuodami defektų valdymo sistemoje užregistruotus defektus, mes galime suprasti klaidų atsiradimo priežastis, o tai leidžia sumažinti tokių pačių klaidų pasikartojimą ateityje [MAC07].

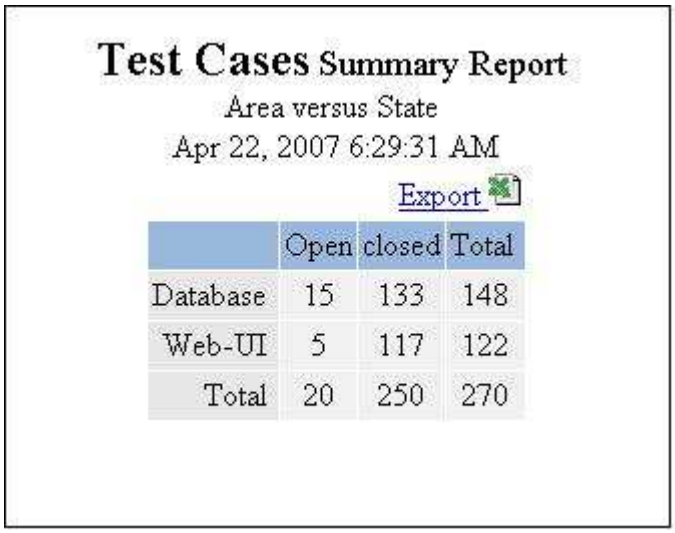
Šiuo metu rinkoje siūloma nemažai įvairių defekto valdymo sistemų – BugZero, BugZilla, ForBugz, JIRA, PR-Tracker, Census ir k.t. praktiškai visose defektų valdymo sistemose yra pamašios funkcijos:

- Visos jos suteikia vartotojui galimybę dirbti tiek Windows tiek Linux aplinkose
- Webinės defektų valdymo sistemos yra pritaikytos dirbti su įvairiomis naršyklėmis;
- Turi lanksčias vartotojų teisių sistemas, kurios suteikia galimybę riboti kai kurių vartotojų priėjimą prie tam tikrų duomenų;
- Leidžia vienu metu dirbti daugeliui vartotojų;
- Turi defektų paieškos mechanizmus;
- Siunčia vartotojui pranešimą elektroniniu paštu, kai registruotas defektas pakeičia būseną ar statusą;
- Suteikia galimybė registruoti defektus, perdavinėti juos taisyti reikiamiems asmenims, gauti informaciją apie defekto būseną;
- Kai kurios galingesnės ir daugiau funkcijų turinčios defektų valdymo sistemos, tokios kaip BugZilla, ForBugz, Census ir k.t. seka defektų pataisymo terminus. T.y. jei buvo registruotas defektas ir nurodyta, jog jis turi būti ištaisytas iki konkrečios datos ir tai nėra įvykdyta, tuomet vartotojas yra apie tai informuojamas;
- Daugumoje iš aukščiau paminėtų programų yra galimybė grupuoti defektus pagal tam tikrus projektus, kurti naujus projektus, redaguoti esančius;
- Naujas defektas registruojamas pildant beveik standartinę formą – paveikslėlis Nr. 9



Paveikslas Nr. 9 – Defekto registravimo forma

- Kai kurios sistemos(BugZilla, BugZero ir k.t.) – turi paprastas grafines ataskaitas, vaizduojančias informaciją apie tai kiek defektų yra užregistruota, kiek jų yra taisoma, kiek jau pataisyta. Paveikslėlis Nr. 10.



	Open	closed	Total
Database	15	133	148
Web-UI	5	117	122
Total	20	250	270

Paveikslas Nr. 10 – Testavimo sąlygų suvestinė

- Ir kitos funkcijos – savo vartotojo sąsajos sukūrimas, patogesnis mygtuku išdėstymas pagal individualius vartotojo pageidavimus ir k.t.

Esančios programos turi nemažą funkcionalumą ir atrodytų viską, ko reikia bet kuriai įmonei, kuriančiai programinę įrangą ir norinčiai sėkmingai kovoti su programoje randamomis klaidomis bei užtikrinti jų pataisymą. Tačiau visos minėtos programos vaizduoja informaciją:

- Vieno vartotojo pjūvyje – kai kiekvienas vartotojas atskirai mato tik jo įvestus defektus ir informaciją apie tų defektų būsenas
- Mato bendrą informaciją apie visus defektus, kokia dalis jų yra pataisyta, kiek taisoma, kiek liko pataisyti.
- Gauta informacija gali būti pateikiama įvairiais būdais: grafikais, kreivėmis, diagramomis

4.2 Testavimo sąlygų valdymo sistemos

Testavimo sąlygų valdymo sistema(TSVS) yra įrankis testavimo darbams organizuoti. TSVS palengvina darbą, nes suteikia galimybę registruoti testavimo sąlygas į vieningą sistemą ir stebėti jų būseną esant skirtingoms testuojamo produkto versijoms. Vieną kartą aprašius testavimo sąlygas testuojant jas galima panaudoti, kiek norima daug kartų. Testuojant naują programos versiją nereikia galvoti bei kurti testavimo situacijų iš naujo – visos jos yra aprašytos. Laikui bėgant produktui keičiantis tereikia tik atnaujinti testavimo sąlygų žinyną papildant naujomis situacijomis, naujais testavimo scenarijais. Sudėtingesnės TSVS, tokios kaip TestLog, TestLink turi galimybes, kurti projektus, testavimo planus, kuriems galima priskirti testavimo sąlygas. Šios sistemos skirtos sudėtingų produktų, sudarytų iš daugelio modulių ir turinčių didelį funkcionalumo, testavimo proceso organizavimui.

Pagrindinės TSVS galimybės:

- Sistemos suteikia galimybę vienu metu dirbti keliems vartotojams, kas yra patogu dirbant darbo grupei. Sistemos turi griežtą teisių priskyrimą ir valdymą, tuo apibrėžiamos, kiekvieno vartotojo teisės pagal jo pareigybes;
- Galimybė kurti projektus, jiems priskirti testavimo planus, o pastariesiems priskirti testavimo sąlygas. Ši funkcija leidžia testuojant didelį funkcionalumą turintį programinį produktą, testuoti naudojant įvairias testavimo strategijas(įvairiais pjūviais) ir gauti detalias kiekvieno pjūvio ataskaitas apie produkto patikimumą;
- Naudojant TSVS užfiksuojami įvairūs testavimo scenarijai, tai sumažina projekto kaštus: kartojant testavimo ciklą nereikia iš naujo kurti defektų paieškos scenarijų, o pasikeitus programos struktūrai ar pasipildžius funkcionalumui tereikia tik papildyti testavimo sąlygų žinyną reikiamais scenarijais;
- Turi platų ataskaitų pasirinkimą įvairiais pjūviais;

4.3 Defektų valdymo ir testavimo sąlygų valdymo sistemų trūkumai

Problema atsiranda tuomet, kai projekte, kuriant programinę įranga arba kuriant naują programos versiją su papildomu funkcionalumu, testavimo procese dalyvauja nevienas darbuotojas, o darbuotojų grupė. Tuo atveju, kai programinis produktas turi didelį funkcionalumą natūralu, jog tobulinant programą pasitaiko ir didesnis kiekis klaidų, kurios atsiranda bandant pritaikyti programą didesniems vartotojų poreikiams. Šiuo atveju dirbant testuotojų grupei prie naujos versijos kokybės tikrinimo kiekvienas iš komandos narių

registruoja nemažą kiekį klaidų kiekvieną dieną. Programavimo grandis savo ruožtu taiso klaidas, atideda jas tolimesniam laikotarpiui arba nusprendžia, jog jos bus pataisytos, tik kitose versijose.

Esant tokiai situacijai nesunku pastebėti, jog sunku suvaldyti visos komandos darbą bei visų registruotų klaidų perėjimą iš vienos būsenos į kitą. Žinoma, tvarkingai įvedus informaciją aukščiau aprašytos sistemos gali suteikti informaciją, apie projekte rastų ir pataisytų defektų skaičių – šiuo atveju naujai kuriama versija gali būti laikoma nauju projektu. Tačiau tai tik vienas iš variantų, kokia informacija reikalinga, kuriant programinį produktą.

Iš praktikos esu pastebėjusi, jog dažniausiai kuriant naują programos versiją visą darbo komandą dominanti informacija yra:

- Kurie programos moduliai, kiek turi klaidų?
- Ar yra programoje kritinių klaidų, dėl kuriu negalima jos eksploatuoti?
- Kokios praėjusios versijos klaidos, kurios buvo rastos ir užregistruotos po versijos uždarymo yra ištaisytos naujoje versijoje?
- Kas atsakingas už modulyje esančias klaidas?
- Kada galima programą/naują jos versiją atiduoti vartotojui?

Asmenis, kurie tiesiogiai bendrauja su klientais dažniausiai domina sekantys klausimai.

- Ar yra atlikti pataisymai, patobulinimai, nauji funkcionalumai konkrečiam klientui? Jei taip, kurioje versijoje?
- Ar klientų pastebėti ir registruoti defektai yra pataisyti? Jei taip, kurioje programos versijoje?
- Kurioje versijoje bus atlikti pakeitimai, kurie reikalauja, jog programa atitiktų Lietuvos Respublikos įstatymus?
- Kada planuojamas likusių defektų versijoje pašalinimas?

Vadovus dažniausiai domina kaštai bei atsakomybės:

- Kuris darbuotojas ar kuri darbo grupė(grandis) atsakinga dėl atsiradusių defektų?
- Kiek viršytas projekto laikas?
- Dėl kokių priežasčių viršytas projekto uždarymo laikas, limitas ir pan.?
- Nuostolingas ar pelningas projektas yra įmonei
- Prie kurio modulio dirbant sugaišta daugiausiai laiko?
- Kuriame modulyje dažniausiai kartojasi klaidos?

Įmonėje skirtingų pareigų darbuotojus domina skirtinga informacija apie vieną ir ta patį produktą. Deja aukščiau aprašyti produktai negali suteikti informacijos į visus šiuos klausimus.

Atsakymams į šiuos klausimus gauti reikalingas produktas, sugebantis pateikti informaciją analizuodamas įvairias situacijas susijusias su tam tikrais projektais, produktais ir jų versijomis bei užregistruotomis problemomis susijusiomis su projektu. Taip pat turi būti pateikiama informacija vadovams apie ekonominę projekto pusę susijusią su išlaidomis, nuostoliais, statistiniais duomenimis apie modulius, turinčius daugiausiai defektų, ir kurių skaičius nemažėja.

Natūraliai kyla klausimas ar tikrai ir kam reikalinga informacija, kuriame modulyje pastoviai randamos klaidos ir atliekant pakeitimus jos vis kartojasi arba kokią įtaką testavimui turi žinojimas, jog suplanuota versijos išleidimo data buvo sausio mėnuo, o išleista ji buvo pora savaičių vėliau. Ši informacija leidžia efektyviau organizuoti darbą ir neleisti atsirasti naujoms klaidoms, užbėgti už akių jų atsiradimui:

- Jei viename modulyje su kiekviena versija iš naujo vis randamos ir randamos klaidos arba funkcionalumas nuolat veikia neteisingai problema gali būti ne tame, jog programuotojas blogai atlieka savo darbą, o dėl to, kad pradinėje produkto kūrimo stadijoje ši sritis buvo blogai suprojektuota. Turint tokią informaciją darbo komanda gali atidžiau pažvelgti ir išanalizuoti šią probleminę sritį ir pasirinkti tinkamiausią sprendimo variantą pvz.: pakeisti bazės struktūrą arba pritaikyti kitą algoritmą.
- Antroji paminėta situacija taip pat savyje gali slėpti problemą, jog nespėjama laiku atiduoti versijos dėl to, jog nebuvo realiai įvertintos darbų apimtys ir ateityje planuojant produkto, projekto ar versijos išleidimo laikus reikėtų papildomai pridėti laiko nenumatytų situacijų šalinimui. Kita problema susijusi su vėlavimu, gali būtį išaugęs klientų skaičius ir per mažas darbuotojų skaičius ir k.t..

Kadangi produkto kūrimo, tobulinimo procese ar vykdant projektą darbo komandą sudaro įvairių pareigybių darbuotojai, todėl ir reikalingas produktas galintis suteikti informaciją visų sričių specialistams apie kuriamo produkto ar vykdomo projekto eigą. Produkto tikslas atsakyti į klausimus ir suteikti informaciją, kuri padėtų ne tik informuoti vartotojus apie jų užsakytų darbų būseną, bet ir išanalizavus nepavykusių darbų priežastis stengtis išvengti jų ir gerinti darbų kokybę.

5. Defektų analizės sistema

Magistrinio darbo metu buvo suprojektuota defektų analizės sistema(DAS). Sistemos pagrindinė funkcija suteikti informacija į ankstesniame skyriuje aprašytus klausimus, su kuriais programinio produkto kūrimo bei aptarnavimo komandos susiduria kiekvieną dieną.

Kadangi kiekvienas programinis produktas turi klaidų ir atliekant testavimo darbus neįmanoma patikrinti visų galimų variantų, o ir toks darbas neretai yra nereikalingas, nes užtenka patikrinti tik keletą variantų apimančių visų įmanomų variantų sritis. Taip pat įmonė kurianti programinę įrangą neturi galimybės taisyti produkto ir nepateikti jo klientui tol, kol programoje nebebus klaidų(Nebebus randama klaidų, nes tikrojo programoje esančių klaidų skaičius niekada nėra žinomas). Praktikoje paprastai yra taikomi keli programinės įrangos testavimo baigimo scenarijai:

- Testavimas ir klaidų taisymas baigiamas konkrečią iš anksto numatytą dieną. Šis metodas turi trūkumą, nes atėjus numatytai datai programinė įranga dar gali turėti esminių defektų, dėl kurių ji negali būti atiduota eksploatuoti vartotojui.
- Sekantis būdas baigti testavimo darbus, kai per tam tikrą laiko tarpą pvz. per 3-5 dienas nerandama naujų defektų ar papildomai atsiradusių programos trūkumų dėl kitų defektų taisymo. Šis testavimo pabaigos scenarijus tam tikrais atvejais pasiteisina ir veikia efektyviai.
- Taip pat testavimo procesą galima laikyti užbaigtų, o programinį produktą paruoštą pristatyti klientui, kai randami tik nedideli kosmetinio pobūdžio trūkumai, kurie neturi įtakos pagrindiniams programos algoritmams bei struktūroms. Naudojant šį metodą patartina išsiaiškinti konkrečius vartotojo poreikius, tam kad nebūtų nesusipratimų klientui gavus eksploatuoti produktą, su neesminiu programos trūkumu(taip atrodė programinio produkto kūrimo komandai), kuris vartotojui pasirodo yra esminis.
- Praktikoje išbandžius didžiausias efektyvumas ir programinio produkto kokybė gaunama naudojant mišrų testavimo pabaigos scenarijų. Šiam scenarijui sudaryti įtraukiama visa programinio produkto kūrimo komanda nuo analitikų, programuotojų iki testuotojų bei programinio produkto diegėjų ir vartotojų. Metodo esmė: programinės įrangos kūrimo proceso pradžioje, įvertinus klientų pageidavimus bei specifinius prašymus ir darbo specifiką, apsibrėžti programos funkcijas bei savybes, be kurių programa negali funkcionuoti, prioriteto tvarka nuo svarbiausio iki nesvarbaus. Susirašius tokiu metodu programos funkcionalumą, be kurio programa negali būti eksploatuojama, testuojama iki tada, kol per tam tikrą

laikotarpį neberandama jokių defektų, kurių prioritetas yra numatyto lygio pvz. „nelabai svarbus“. Arba testavimas baigiamas tik tuomet kai nebebūna jokių defektų, kurių prioritetas „svarbus“ ir aukštesnis.

Pasirinkus testavimo pabaigos sąlygą galima sudaryti testavimo scenarijus pagal pasirinktą testavimo pabaigos bei efektyviausią testavimo metodą ir pradėti programinės įrangos kokybės kontrolę.

Testavimo sąlygos surašomos į DAS ir viso programinio produkto kūrimo/ar tobulinimo metu atlikus produkto kokybės patikrinimą, defekto radimą pažymėjus DAS tam tikrą informaciją bet kurioje produkto kūrimo stadijoje bus galima gauti informaciją: pvz. apie versijoje esančių klaidų skaičių procentais, dviejų versijų palyginimas - kas patobulinta ir kokie nauji defektai atsirado programoje ir k.t.);

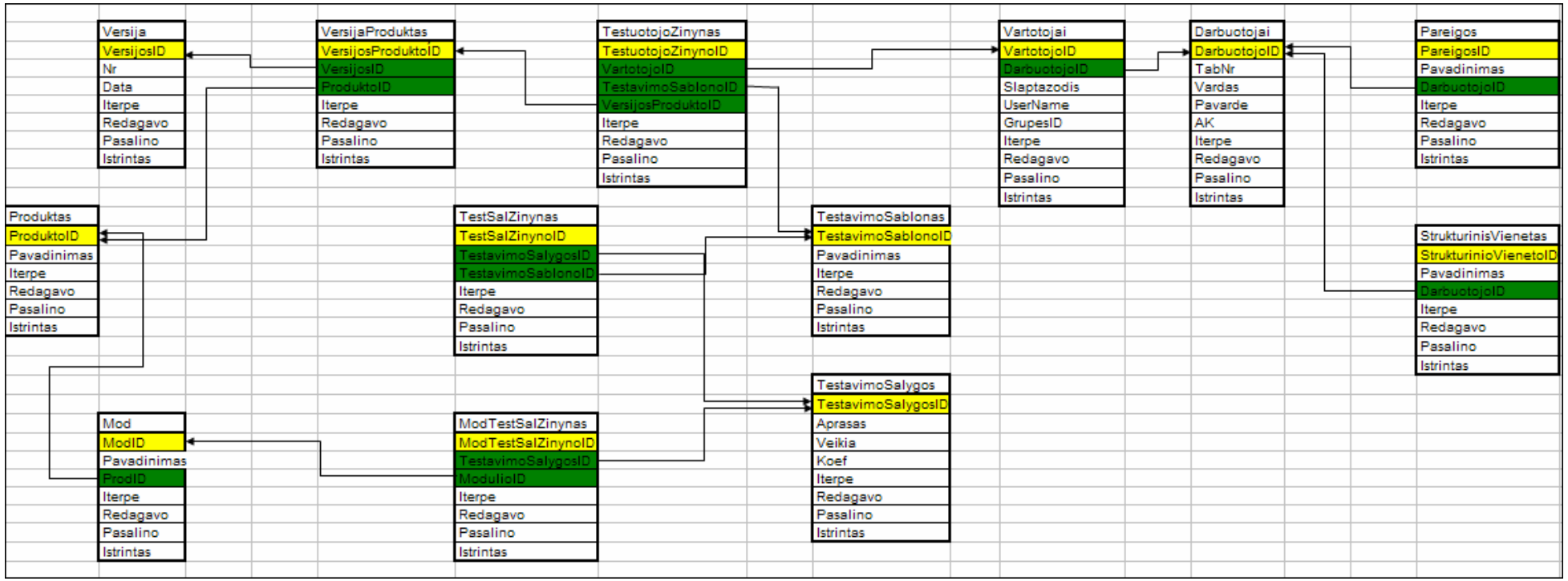
5.1 DAS funkcijos

Suprojektuota DAS yra „webinis“ sprendimas. Pagrindinis DAS principas toks:

1. kaip ir testavimo sąlygų valdymo sistemose administratorius sukuria vartotojus su skirtingomis teisėmis skirtingoms darbuotojų grupėms. pvz.: testuotojai gali pildyti žinynus pagal savo poreikius, žymėti ar pasirinktas scenarijus yra įvykdomas ar ne, o pvz.: projektų vadovo teises turintis vartotojas negali to daryti, jis gali tik peržiūrėti tam tikras ataskaitas.
2. toliau pagal poreikį yra kuriami testavimo šablonai ir jiems priskiriamos testavimo sąlygos. Testavimo šablonas gali būti tiek produkto modulis, tiek projekto pavadinimas ar tiesiog nedidelio funkcionalumo patikrinimas. Testavimo sąlygos papildomos pagal poreikį, testavimo šablonų kūrimas taip pat nėra ribojamas.
3. testuotojas gavęs užduotį testuoti, pasirenka testavimo šabloną bei produktą ir tikrina funkcionalumą, pagal šablone nurodytus testavimo scenarijus, prie kiekvieno scenarijaus nurodydamas ar jis atitinka reikalaujamą funkcionalumą ar ne. Atitinkamai, jei scenarijus netenkina funkcionalumo pažymi pastabą, kas neveikia ir kodėl.
4. testuotojams atlikus testavimus ir pažymėjus sistemoje jų rezultatus sistema pagal įvestus programos trūkumus ir defektus pateikia statistinius duomenis: ataskaitas, diagramas, procentinius įvertinimus ir t.t.

5.2 DAS struktūra

Toliau pateikiama koncepcinė defektų analizės sistemos reliacinė diagrama bei lentelių aprašymai. Sistemos lentelės gali būti papildytos papildomais stulpeliais. Atsiradus naujiems poreikiams gauti informaciją arba atlikti analizę pagal naują pjūvį turint programos defektų duomenis.



Paveikslas Nr. 11 – Reliacinė diagrama

Žalia – spalva pažymėti lentelių išoriniai raktai.

Geltona – spalva pažymėti lentelių pirminiai raktai.

Produktas – lentelė, kurioje yra saugomas testuojamo produkto pavadinimas. Kai testavimo sistema yra testuojami keli produktai, produktai turi būti įvesti i žinyną. Praktikoje produktu vadinamas testuojamos programos pavadinimas.

Lentelė turi pagrindinius laukus:

- **ProduktoID** – produkto identifikavimo numeris. Jis vienareikšmiškai identifikuoja produktą. Šis laukas yra lentelės pirminis raktas. Jis generuojamas automatiškai, kiekvieną kartą įvedus naują produktą jam suteikiamas vienetu didesnis identifikavimo numeris.
- **Pavadinimas** – produkto pavadinimas. T.y. testuojamos programos pavadinimas.

Likę keturi lentelės laukeliai yra vienodi visoms lentelėms. Jie skirti informacijos sekimui apie įvestą įrašą. Laukeliuose **„Iterpe“, „Redagavo“, „Pasalino“, „Istrintas“** saugomas vartotojo identifikavimo numeris, kuris įterpė įrašą, paskutinis jį redagavo ar pašalino jį iš programos. Laukelis **„Ištrintas“** – naudojamas kaip požymis pašalintiems įrašams iš programos, pažymėti. Jei pasirenkama pašalinti įrašą iš programos, jis nėra ištrinamas iš duomenų bazės, tiesiog šiame laukelyje nurodomas atitinkamas požymis ir įrašas tampa vartotojui nematomas, tačiau bazėje egzistuoja.

Versija – lentelė, kurioje saugomi produkto versijų numeriai. Versijų žinynas papildomas, kiekvieną kartą, kai sukuriama nauja programos versija.

Lentelės pagrindiniai laukai:

- **VersijosID** – versijos identifikavimo numeris. Laukelis vienareikšmiškai identifikuoja versijos numerį. Laukelis yra lentelės pirminis raktas ir generuojamas automatiškai įvedus naują įrašą į lentelę.
- **Nr** – versijos numeris, laukelis pildomas vartotojo.
- **Data** – versijos sukūrimo data. Laukelis pildomas automatiškai įrašo įvedimo metu yra įrašoma šios dienos data.

Likę lentelės laukeliai yra standartiniai.

Kadangi kiekviena testuojama programa turi ne vieną versiją, o versijų numeriai skirtingoms programoms gali kartotis, todėl reikalinga papildoma lentelė ***VersijaProduktas***, kurioje saugomas produktai ir su jų versijų numeriais.

Pagrindiniai lentelės laukai:

- **VersijosProduktoID** – lentelės pirminis raktas, vienareikšmiškai identifikuojantis lentelės įrašą. Generuojamas automatiškai įvedus naują įrašą į lentelę.
- **VersijosID** – versijos identifikavimo numeris. Lentelės išorinis raktas į lentelę *Versija*.
- **ProduktoID** – produkto identifikavimo numeris. Lentelės išorinis raktas į lentelę *Produktas*.

Likę lentelės laukeliai yra standartiniai.

Dažnai testuojant programas, jos yra skirstomos dalimis – taip vadinamais moduliais. Todėl lentelėje ***Moduliai*** – šioje lentelėje saugomi, testuojamų produktų moduliai.

Lentelės pagrindiniai laukai:

- **ModulioID** – lentelės pirminis raktas, generuojamas automatiškai įterpiant į lentelę naują įrašą. Laukas vienareikšmiškai identifikuoja lentelės įrašus.
- **Pavadinimas** – laukelyje saugomas vartotojo įvestas modulio pavadinimas.
- **ProduktoID** – kadangi kiekvienas produktas gali turėti kelis modulius todėl reikalingas požymis, kuris priskirtų modulį konkrečiam produktui. Lentelės išorinis raktas į lentelę *Produktas*.

Likę lentelės laukeliai yra standartiniai.

Testuojant programas būtinos testavimo sąlygos, kurios turi būti patenkintos tam, kad būtų galima sakyti, jog programa veikia teisingai. Šios sąlygos yra saugomos lentelėje ***TestavimoSalygos***.

Lentelės pagrindiniai laukai:

- **TestavimoSalygosID** – lentelės pirminis raktas, generuojamas automatiškai, įterpiant naują testavimo sąlyga į lentelę. Laukas vienareikšmiškai identifikuoja lentelės įrašus.

- **Aprašas** – laukelyje saugomas testavimo sąlygos aprašymas įvedamas vartotojo. Pvz.: „Ar leidžiama įterpti įrašą į žinyną“, „ar galima redaguoti pasirinktą įrašą“ ir pan.
- **Veikia** – požymis, kuris pažymi ar testavimo sąlyga yra patenkinta, t.y. ar ji veikia ar ji yra nepatenkinta – neišpildyta. Laukelis pildomas testuotojo, patikrinus ar sąlyga išpildyta ar ne.
- **Koef** – laukelis skirtas įvesti testavimo sąlygos koeficientą. Laukelis pildomas vartotojo įvedant naujas testavimo sąlygas. Šis koeficientas naudojamas ataskaitose, skaičiuojant versijos ar modulio kokybę procentais. Taip pat palyginant vienos ar kitos versijos stabilumą.

Likę lentelės laukeliai yra standartiniai.

Visos testavimo sąlygos turi būti priskirtos, vienam ar kitam testavimo šablonui. Nepasirinkus testavimo šablono negalima testuoti jokio produkto. Testavimo šablonai įvedami ir saugomi lentelėje ***TestavimoSablonas***.

Lentelės pagrindiniai laukai:

- **TestavimoSablonoID** – lentelės pirminis raktas, vienareikšmiškai identifikuojantis lentelės įrašus. Generuojamas automatiškai įterpiant naują testavimo šabloną į lentelę.
- **Pavadinimas** – laukelis saugantis testavimo šablono pavadinimą. Laukas pildomas vartotojo įterpiant naują įrašą.

Likę lentelės laukeliai yra standartiniai.

Kiekvienas testavimo šablonas turi begalę testavimo sąlygų, tačiau viena ir ta pati testavimo sąlyga gali būti naudojama testuojant vieną ar kitą testavimo scenarijų, todėl testavimo šablonų ir testavimo sąlygų susiejimas yra saugomas lentelėje ***TestSalZinynas***.

Pagrindiniai lentelės laukai:

- **TestSalZinynoID** – laukas vienareikšmiškai identifikuojantis lentelės įrašus. Lauko reikšmė generuojama automatiškai, kiekvieną kartą įterpiant naują įrašą į lentelę.

- **TestavimoSalygosID** – išorinis lentelės raktas į lentelę *TestavimoSalygos*.
- **TestavimoSablonasID** – išorinis lentelės raktas į lentelę *TestavimoSablonas*.

Likę lentelės laukai yra standartiniai.

Lygiai taip pat, kaip kiekvienas testavimo šablonas gali turėti begalę testavimo sąlygų, lygiai taip pat ir kiekvienam moduliui gali priklausyti ne viena testavimo sąlyga ir taip pat ta pati testavimo sąlyga gali priklausyti keliems moduliams. Modulio ir testavimo sąlygų susiejimas yra saugomas lentelėje ***ModTestSalZinynas***.

Pagrindiniai lentelės laukai:

- **ModTestSalZinynoID** – laukas vienareikšmiškai identifikuojantis lentelės įrašus. Lauko reikšmė generuojama automatiškai, kiekvieną kartą įterpiant naują įrašą į lentelę.
- **TestavimoSalygosID** – išorinis lentelės raktas į lentelę *TestavimoSalygos*.
- **ModulioID** – išorinis lentelės raktas į lentelę *Modulis*.

Likę lentelės laukai yra standartiniai.

Testuotojai prisijungę prie programos testuoja tam tikras versijas, pasirinkę norimą/reikiamą testavimo šabloną bei versiją. Todėl reikalinga lentelė sauganti šią informaciją: kuris vartotojas, testavo versiją, pagal reikiamą testavimo šabloną. Lentelėje ***TestuotojoZininynas*** ir saugoma ši informacija.

Lentelės pagrindiniai laukai:

- **TestuotojoZinynoID** – lentelės pirminis raktas, vienareikšmiškai identifikuojantis lentelės įrašus. Lauko reikšmė generuojama automatiškai įterpiant įrašą į lentelę.
- **VartojoID** – lentelės išorinis raktas į lentelę *Vartotojai*.
- **TestavimoSablonoID** – lentelės išorinis raktas į lentelę *TestavimoSablonai*.
- **VersijosProduktoID** – lentelės išorinis raktas į lentelę *VersijaProduktas*.

Likę lentelės laukai yra standartiniai.

Vartotojai dirbantys su testavimo sistema yra identifikuojami pagal jiems suteiktą vartotojo vardą. Kiekvienas iš vartotojų yra žmogus dirbantis įmonėje, tam tikrose pareigose,

tam tikrame padalinyje. Visa informacija apie vartotojus, darbuotojus, pareigas bei padalinius saugomi šiose lentelėse:

Vartotojai – lentelėje saugomi vartotojai, kuriems suteikta arba jau panaikinta teisė dirbti su sistema.

Lentelės pagrindiniai laukai:

- **VartotojoID** – lentelės pirminis raktas, vienareikšmiškai identifikuojantis sukurtus vartotojus. Lauko reikšmė generuojama automatiškai įterpiant naują vartotoją į lentelę.
- **DarbuotojoID** – lentelės išorinis raktas į lentelę *Darbuotojai*. Parodo kuriam darbuotojui yra suteiktas šis prisijungimo vardas.
- **Slaptazodis** – vartotojo slaptažodis, pasirenkamas kiekvieno vartotojo, gali būti keičiamas.
- **UserName** – konkretaus vartotojo prisijungimo vardas, norint dirbti su sistema.
- **GrupesID** – vartotojų grupės identifikavimo numeris. Šiuo metu šis laukelis jokios reikšmės neturi, bet ateityje suteikiant teises atlikti tam tikrus veiksmus dirbant su testavimo sistema bus reikalingas tam, kad nereiktu kiekvienam vartotojui suteikti tų pačių teisių kiekvieną kartą, o būtų galima jas išsaugoti ir vienu mygtuko paspaudimu jas suteikti.

Likę lentelės laukai yra standartiniai.

Darbuotojai – lentelėje saugoma informacija apie darbuotojus.

Pagrindiniai lentelės laukai:

- **DarbuotojoID** – priminis lentelės raktas, generuojamas automatiškai įvedant naują darbuotoją į programą.
- **TabNr** – unikalus indeksas, darbuotojui suteikiantis unikalų jo numerį įmonėje. Lauko reikšmė yra įvedama vartotojo, tačiau keli vartotojai negali turėti vienodų šio laukelio reikšmių.
- **Vardas** – darbuotojo vardas.
- **Pavarde** – darbuotojo pavardė.
- **AK** – darbuotojo asmens kodas.

Likę lentelės laukai yra standartiniai.

Pareigos – lentelėje saugoma informacija apie darbuotojui galimas priskirti pareigas. Jei reikiamų pareigų joje nėra, jas galima įvesti.

Pagrindiniai lentelės laukai:

- **PareigosID** – lentelės pirminis raktas vienareikšmiškai identifikuojantis pareigas. Lauko reikšmė automatiškai generuojama įterpiant naujas pereigas į lentelę.
- **Pavadinimas** – pareigų pavadinimas. Laukelio reikšmė įvedama vartotojo.
- **DarbuotojoID** – lentelės išorinis raktas į lentelę *Darbuotojai*.

Likę lentelės laukai yra standartiniai.

StrukturinisVienetas – lentelėje saugoma informacija apie darbuotojui galimą priskirti padalinį. Jei reikiamų padalinių joje nėra, jį galima įvesti.

Pagrindiniai lentelės laukai:

- **StrukturinioVienetoID** – lentelės pirminis raktas vienareikšmiškai identifikuojantis padalinį. Lauko reikšmė automatiškai generuojama įterpiant naują padalinį į lentelę.
- **Pavadinimas** – padalinio pavadinimas. Laukelio reikšmė įvedama vartotojo.
- **DarbuotojoID** – lentelės išorinis raktas į lentelę *Darbuotojai*.

Pagrindinės programos funkcijos ir pranašumai yra tai, kad:

- Galimybė kurti neribojamą kiekį testavimo šablonų, kuriant naują šabloną galima kopijuoti informaciją iš jau sukurtų šablonų
- Kurti testavimo sąlygas ir priskirti jas atitinkamam testavimo šablonui
- Kiekvieną programinio produkto versiją testuoti pagal tą patį testavimo šabloną ir tą pačią programos versiją pagal skirtingus testavimo šablonus.
- Pagal užfiksuotas klaidas gauti ataskaitas įvairias pjūviais: produkto patikimumas procentine išraiška; dviejų versijų palyginimas – ar produkte mažėja klaidų skaičius su kiekvienu nauju pataisymu ar taisant senas klaidas daromos kitos; produkto kaštai; užtęstas laikas projekto(versijos užbaigimui) ir t.t.

6 Tinkamiausio testavimo įrankio(strategijos) pasirinkimo metodika

Nesvarbu, kaip stengsimės išvengti programinės įrangos defektų, jų vis tiek bus. Defektas (trūkumas) – tai programos veikimas ne pagal specifikaciją dėl klaidos (klaida – programų autoriaus apsirikimas, klaidingas požiūris, supratimas). Dėl šios priežasties mes ir testuojame programinę įrangą.

Testuotojų atsakomybė yra suprojektuoti testus, kurie: 1) atskleistų defektus; 2) galėtų būti naudojami programinės įrangos kokybės įvertinimui. Šių uždavinių tikslams pasiekti būtina projektuoti reikiamą testavimo sąlygų skaičių (jis turi būti baigtinis). Nepatyrę testuotojai nori testuoti absoliučiai viską ir tikisi, kad tai padės išvengti visų galimų defektų. Deja, tai niekada nebuvo ir nebus įmanoma. Testuotojai visada yra spaudžiami laiko ir projektui įgyvendinti skirtų lėšų. [BUR02]

Sumanus(geras) testuotojas – asmuo, kuris atsakingas už programinės įrangos testavimą – turintis tam tikrą žinių bagažą. Daugeliu atveju būti geru testuotoju yra daug sunkiau nei geru kūrėju, nes testavimas reikalauja ne tik labai gerų programinės įrangos kūrimo žinių, bet taip pat reikalauja tam tikros nuojautos, kur programoje galėtų būti nenumatytos situacijos ir dėl to atsirastų defektai. Pavyzdžiui, programuotojas turi sugalvoti algoritmą, kuris leistų sukurti šokinėjančio paveiksliuko animaciją aplink kvadratą kompiuterio ekrane. Tuo tarpu testuotojas, turi numanyti galimas klaidas bei trūkumus, kuriuos programuotojas gali padaryti, bei sukurti efektyvius būdus programavimo klaidoms rasti. [MCD01]

Dažnai testuotojui tenka pažvelgti į testuojamą produktą iš įvairių prizmių:

- tenka tikrinti vartotojo sąsają – ar ji patogi vartotojui, ar yra apsaugota nuo netinkamų vartotojo veiksmu, kaip pvz. išsaugoti formą, kai nėra užpildyti privalomi laukai.
- Ar apdorotos retai pasitaikančios situacijos, kurios algoritmuose turi didelę reikšmę: pvz. vasario 29 d., kuri būna tik kas ketverius metus.
- Taip pat testuotojams tenka patikrinti ir programinės įrangos galimybes, atsparumą sistemos „lūžiams“ ir k.t.

Suprantama, jog testuojant programinį produktą visų variantų ir visų gyvenimo atveju patikrinti neįmanoma, todėl sumanaus testuotojo privalumas – gebėjimas parinkti tinkamą testavimo strategiją bei testavimo būseną arba jų derinius testuojamai programinei įrangai.

Dažnai praktikoje kuriant testavimo sąlygas didelį funkcionalumą turinčiai programiniai įrangai, testuotojams tenka atsižvelgi ne tik į programinio produkto savybes bei funkcijas(programos modulių skaičių algoritmų sudėtingumas ir t.t.), bet ir į įmonės turimus resursus(testavimui skirtą laiką, testuotojų skaičių, testuotojų patirtį).

Toliau magistriniame darbe remdamasi 2 skyriuje aptartais populiariausias testavimo metodais bei strategijomis išskyriau kelias pagrindines testavimo metodų grupes:

1. juodosios dėžės testavimo strategijos:
    - atsitiktinis generavimas;
    - ekvivalentinis dalinimas;
    - ribinių reikšmių analizė;
    - priežasčių ir pasekmių analizė;
    - sprendimų lentelė;
    - klaidų spėjimas;
2. programinės įrangos testavimo būsenos:
    - funkcinis testavimas – testuojama ar programinė įranga veikia taip, kaip iš jos tikimasi, kad ji turi veikti;
    - „stress“ testavimas – testavimas naudojant didelius duomenų kiekius, tikrinami sudėtingiausi atvejai;
    - „recovery“ testavimas – programos atsparumo sistemos „lūžiams“ testavimas;
    - „Usability“ testavimas – vartotojo sąsajos patogumo testavimas;
    - „volume“ testavimas – testavimas, nustatant programos galimybių ribas pvz.: su kokiomis naršyklėmis veikia produktas, su kuriomis duomenų bazių valdymo sistemomis yra suderinta programa ir t.t.
    - Struktūrinis testavimas – numato visų programos modulių korektišką veikimą ir visų programos kelių vykdymą;
    - Modulinis testavimas – atskiras kiekvieno modulio testavimas;
3. testavimas vartotojo lygyje, kai į testavimo procesą įtraukiamas vartotojas ir programinio produkto kūrimo metu reaguojama į jo pateiktas pastabas:
    - „user acceptance“ testavimas – programinė įranga parodoma vartotojui su tikslu įsitikinti, jog ji atitinka vartotojo lūkesčius;

- „alpha“ testavimas – vartotojas atlieka programinės įrangos testavimą gamintojo patalpose. Praneša apie pastebėtus defektus, kurie privalo būti ištaisyti;
- „beta“ testavimas – vartotojas atlieka programinio produkto testavimą, su realiais duomenimis, bet jau ne pas gamintoja. Pastebėtus trūkumus praneša gamintojui;

Pasinaudodama šiomis grupėmis sukūriau schemą, kuri testuotojui suteikia galimybę pasirinkti testavimo strategiją ar kelias iš jų efektyviam testavimo sąlygų sudarymui. Kitaip tariant, testuotojas ar kitas testavimo sąlygas kuriantis asmuo pasinaudodamas šia schema gali parinkti vieną ar kelias tinkamiausias testavimo strategijas konkrečiam produktui testuoti atsižvelgiant tiek į jo funkcionalumą ir ypatybes bei į įmonės turimus resursus.
Schemos principas labai paprastas testuotojas atsakinėja į klausimus taip arba ne ir priklausomai nuo atsakymų į klausimus jam pasiūloma vienas ar kitas testavimo metodas.

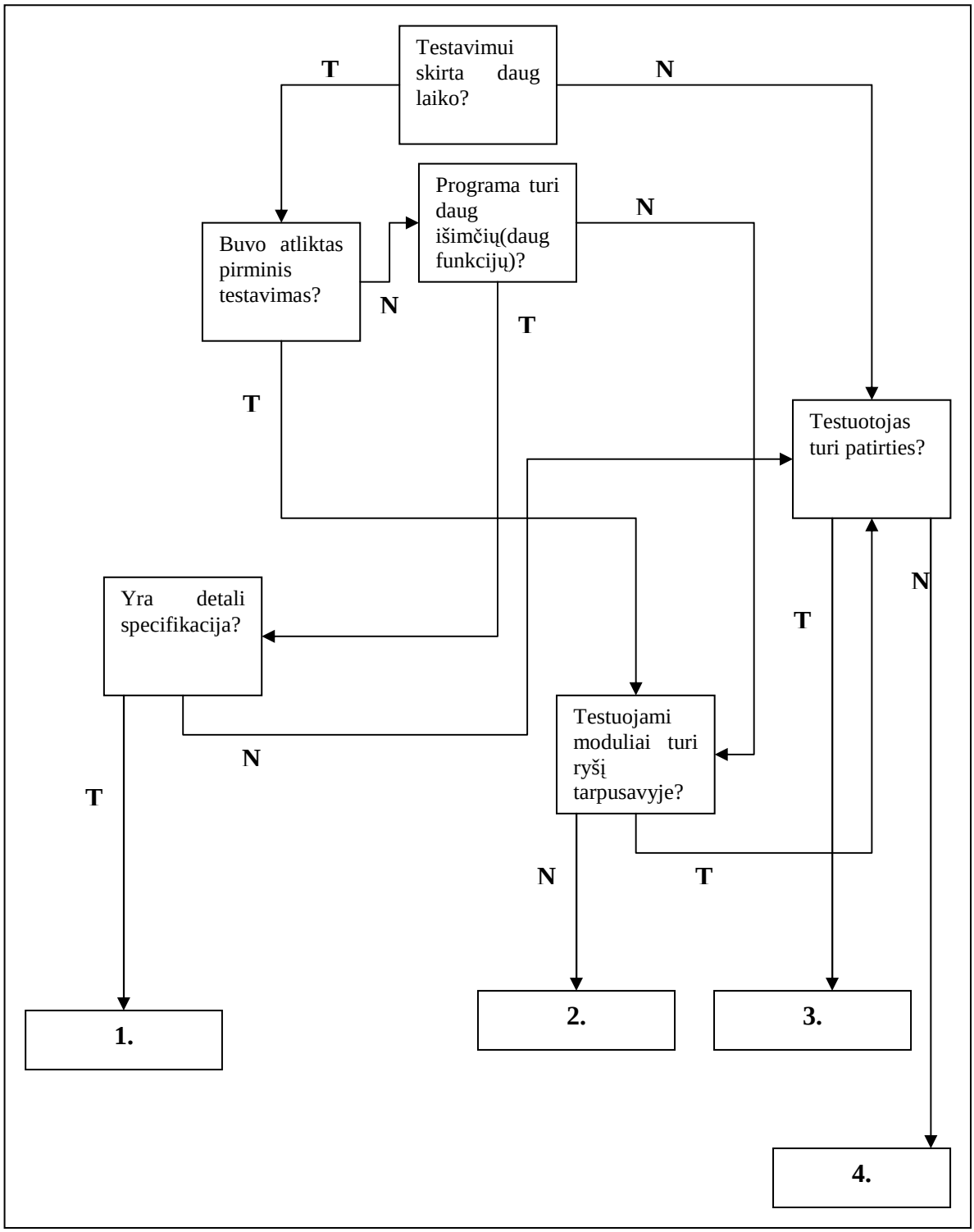
Remdamasi praktine patirtimi sudariau du modelius efektyvaus testavimo metodo parinkimui, nes prieš atliekant pradedant testavimą ir susipažįstant su programiniu produktu testuotojui visada kyla du pagrindiniai klausimai: kiek laiko turime? Kokios apimties ir funkcionalumo yra programinė įranga?

Praktikoje atliekant programinės įrangos defektų paiešką testuotojams dažniausiai trūksta laiko, todėl pagrindinis pirmos efektyvios testavimo strategijos parinkimo schemos klausimas – Ar daug laiko skirta testavimui? Schema pavaizduota paveikslėlyje Nr. 12. Ši schema pateikia keturis testavimo sąlygų kūrimo scenarijus su pasiūlomosiomis testavimo strategijomis bei būdais:

1. funkcinis testavimas, priežasčių ir pasekmių analizė, sprendimų lentelė;
2. ekvivalentinis dalinimas, modulinis testavimas;
3. ribinių pasekmių analizė, struktūrinis testavimas, klaidų spėjimas;
4. atsitiktinis generavimas, „volume“, ir „stress“ testavimas;

Diagramos pateikiami rezultatai gana tikslūs, nes pvz.: turintiems mažai patirties testuotojams dažniausiai patikimos nedidelės atsakomybės ir specifinių srities žinių reikalaujančios užduotys(atsitiktinis generavimas, „stress“ testavimas – reikalaujantis mažai specifinių srities žinių, kurioje programa bus taikoma, pvz.: draudimas, buhalterija, logistika ir t.t.). Ribinių pasekmių analizės modelis taikomas, tuomet kai turima nedaug laiko skirto testavimui ir norima minimizuoti testavimo variantų skaičių, tačiau rasti maksimalų programoje esančių defektų skaičių. Žinoma, tai reikalauja testuotojo darbo patirties bei

specifinės srities išmanymo. Turint detaliai aprašytą programos specifikaciją patogu naudoti priežasčių ir pasekmių analizės testavimo strategiją, o sprendimų lentelės sudarymui reikia daugiau laiko, tačiau ilgainiui šis testavimo būdas labai padeda – ypač kai reikalingas pakartotinis testavimas.

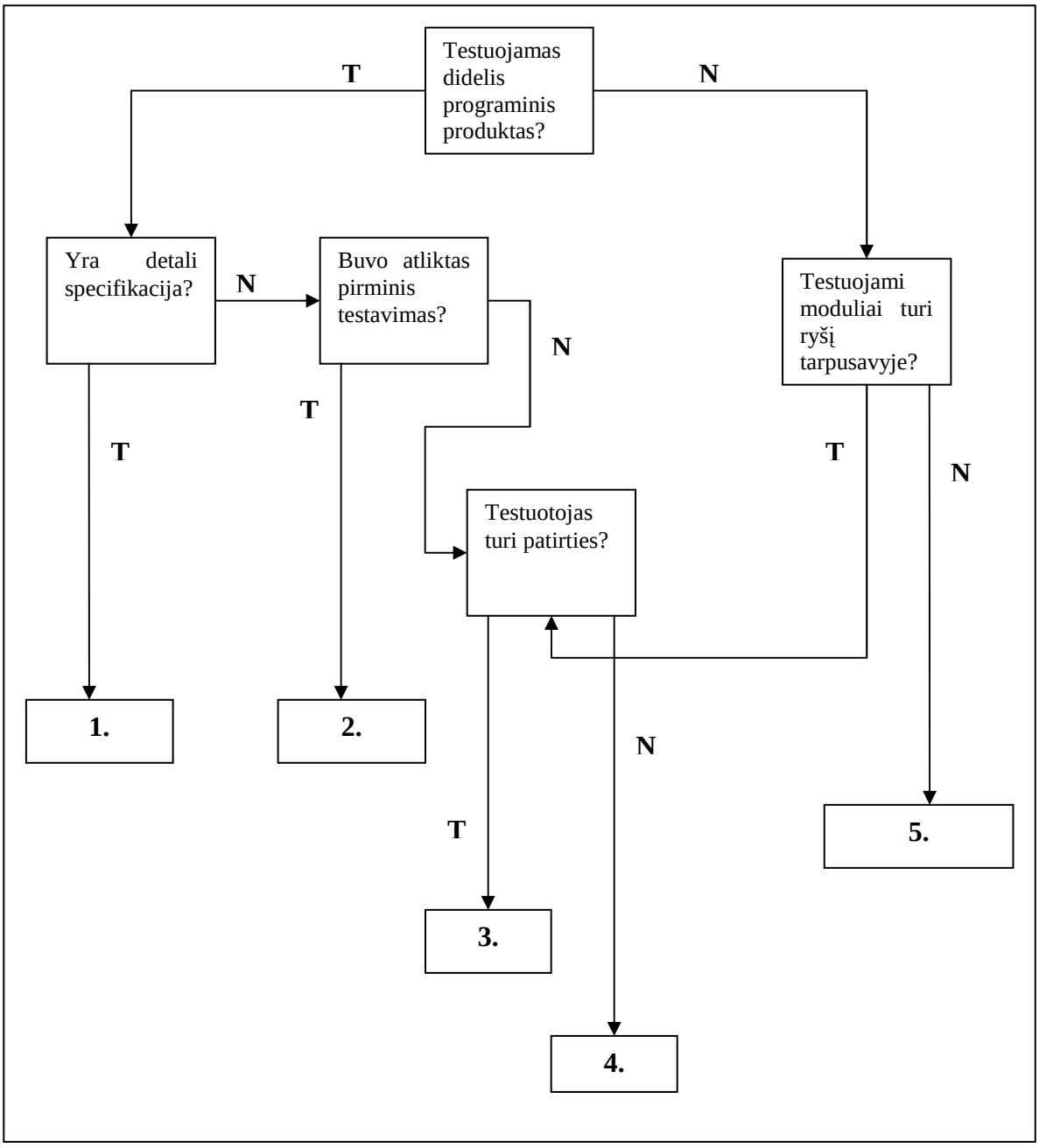


Paveikslas Nr. 12 – Testavimo sąlygų parinkimo schema_1

T – teigiamas atsakymas;

N – neigiamas atsakymas;

Antroji testavimo strategijos parinkimo schema pateikia testavimo sąlygų parinkimo scenarijus atsižvelgiant ne į skirtą testavimui laiko kiekį, o atsižvelgiant į programinio produkto ir jo funkcionalumo dydį. Schema pavaizduota paveikslėlyje Nr. 13.



Paveikslas Nr. 13 – Testavimo sąlygų parinkimo schema_2

T – teigiamas atsakymas;

N – neigiamas atsakymas;

Ši schema pateikia penkis testavimo sąlygų kūrimo scenarijus su pasiūlomosiomis testavimo strategijomis bei būdais:

1. funkcinis testavimas, priežasčių ir pasekmių analizė, sprendimų lentelė;
2. klaidų spėjimas, atsitiktinis generavimas;
3. ribinių pasekmių analizė, struktūrinis testavimas, klaidų spėjimas;
4. atsitiktinis generavimas, „volume“, ir „stress“ testavimas;
5. ekvivalentinis dalinimas, modulinis testavimas;

Antruoju atveju pasiūloma vienu testavimo scenarijumi daugiau. Antrasis atvejis nuo pirmojo skiriasi tik tuo, jog atlikus pirminį testavimą siūloma taikyti atsitiktinio generavimo bei klaidų spėjimo testavimo metodus. Natūralu, nes jau atlikus priminį testavimą, pagrindiniai programos defektai yra rasti ir tuomet galima atkreipti dėmesį į įvairias netikėtas, nestandartines situacijas.

Atidžiau pažvelgę į abi schemas taip pat matysime, jog nei vienoje iš jų nėra paminėtas „usability“ testavimo metodas, taip pat nėra aptartas nei vienas vartotojo lygmens testavimas. Gali kilti klausimas, ar šios testavimo būsenos nėra svarbios ir neturi jokios įtakos galutinei programinės įrangos kokybei? Atsakymas labai paprastas: šios testavimo būsenos bei metodai nėra paminėti schemose, nes jų taikymas tiesiog privalomas, kuriant programinį produktą ir jie negali būti tarp pasirenkamų. Geras programinis produktas, visų pirma, yra toks, kuris patogus vartotojui, neapkrauna dirbančio vartotojo įvairiais sudėtingais veiksmais, funkcijos yra išdėstytos ir pavadintos taip, kad dirbančiam su programa asmeniui nekyla dviprasmiškų klausimų, kaip programa pasielgs pasirinkus šį mechanizmą. Tam kad programa tenkintų visus patogumo, aiškumo bei paprastumo reikalavimus, programinės įrangos kūrimo ir testavimo etapuose yra būtinas testavimas vartotojo lygyje.

Abi aukščiau pavaizduotos schemos yra puikus įrankis testavimo parinkimo strategijoms, jos nėra universalios ir tinkamos bet kuriuo atveju, tačiau gali būti lengvai papildomos naujais klausimais ir pritaikomas testavimo metodo parinkimui sudėtingesnėms ar daugiau specifinės srities žinių reikalaujančioms sistemoms

7 Išvados

Mokslinio tiriamojo darbo metu teko susipažinti su tradicinėmis „juodosios“ bei „baltosios“ dėžės testavimo strategijomis. Kiekvienos iš šių strategijų turi savo privalumus ir trūkumus bei esant tam tikromis aplinkybėmis taikant vieną ar kitą testavimo strategiją yra

pasiekiami maksimaliai efektyviausi testavimo darbai – randamas maksimalus defektų skaičius sugaištant mažiausiai laiko.

Susipažinus su egzistuojančiomis testavimo strategijomis bei išanalizavus dažniausiai pasitaikančias problemas, su kuriomis susiduria įmonės programinio produkto kūrimo procese, sukūriau pora tinkamiausio testavimo įrankio(strategijos) parinkimo mechanizmų. Jų dėka atsakius į klausimus, susijusius tiek su testuotino produkto funkcinėmis savybėmis, tiek su įmonės taikoma politika bei resursais, yra pasiūloma viena ar kelios testavimo strategijos(metodai), kuriais vadovaujantis testuojant programinę įrangą galima maksimaliai minimalizmuoti programoje esančių defektų skaičių.

Taip pat tiriamojo darbo metu teko susipažinti su rinkoje esančiomis defektų valdymo bei testavimo sąlygų sistemomis bei jų turimomis funkcijomis ir galimybėmis. Išanalizavus esamų programų privalumus ir trūkumus nesunku buvo sumodeliuoti defektų valdymo sistemą(DAS), kuri pagal pažymėtus programoje esančius defektus suteiktų informaciją programinio produkto kūrimo komandai apie: pvz. Kurie programos moduliai, kiek turi klaidų; Ar yra programoje kritinių klaidų, dėl kuriu negalima jos eksploatuoti; Dėl kokių priežasčių viršytas projekto uždarymo laikas, limitas ir pan.; Nuostolingas ar pelningas projektas yra įmonei; Prie kurio modulio dirbant sugaišta daugiausiai laiko; Ar yra atlikti pataisymai, patobulinimai, nauji funkcionalumai konkrečiam klientui; Jei taip, kurioje versijoje; ir pan.

Mokslinio tiriamojo darbo metu buvo sukurtas koncepcinis defektų valdymo sistemos modelis, kuris bus pritaikytas įmonėje programinio produkto tobulinimo procese. Modelis yra universalus ir gali būti pritaikomas, bet kokio programinio produkto kūrimo(projekto vykdymo) ar tobulinimo procese. Defektų analizės sistemai pritaikius testavimo sąlygų parinkimo schemas, turimas pakankamai galingas testavimo darbą palengvinantis įrankis.

8 Literatūra

[SIL07] Darius Šilingas „Programinės įrangos kūrimas“
URL: http://www.bpi.lt/text.php?lang=1&item=221&arg=199 2007.05.16

[BUR02] Ilene Burstein „Praktical Software Testing“ 2002m.

[MYE04] Glenford J. Myers „The Art of Software testing“ 2004m.

[MAC07] Jolita Mackienė „ PĮ Defektų valdymas“
URL: http://www.bpi.lt/text.php?lang=1&item=224&arg=202 2007.05.16

[MS06] Edita Milevičienė, Darius Šilingas „PROGRAMINĖS ĮRANGOS REIKALAVIMŲ PASIKEITIMAI IR JŲ VALDYMAS“
URL:
http://www.ktu.lt/lt/apie_renginius/konferencijos/2006/k6_02/IT2005/Sekc09.pdf 2007.03.03

[MCD01] John D. McGregor, David A. “A practical guide to testing object-oriented software Sykes” 2001.