KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

Kristina Draudvilaitė

GHERKIN TESTŲ GENERAVIMO IŠ REIKALAVIMŲ GALIMYBIŲ TYRIMAS

Baigiamojo magistro projektas

Vadovas
prof. dr. R. Butleris

Konsultantė
dokt. K. Aleliūnė

KAUNAS, 2016

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

GHERKIN TESTŲ GENERAVIMO IŠ REIKALAVIMŲ GALIMYBIŲ TYRIMAS

Baigiamasis magistro projektas
Informacinių sistemų inžinerijos studijų programa (kodas 621E15001)

Vadovas
prof. dr. R. Butleris
2016-05-

Konsultantė
dokt. K. Aleliūnė
2016-05-

Recenzentas
prof. dr. E. Sakalauskas
2016-05-

Projektą atliko
Kristina Draudvilaitė
2016-05-

KAUNAS, 2016

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

(Fakultetas)

Kristina Draudvilaitė

(Studento vardas, pavardė)

Informacinių sistemų inžinerijos studijų programa, 621E15001

(Studijų programos pavadinimas, kodas)

Baigiamojo projekto „Pavadinimas“

AKADEMINIO SĄŽININGUMO DEKLARACIJA

20 16 m. gegužės d.

Kaunas

Patvirtinu, kad mano, **Kristinos Draudvilaitės**, baigiamasis projektas tema „*Gherkin* testų generavimo iš reikalavimų galimybių tyrimas“ yra parašytas visiškai savarankiškai ir visi pateikti duomenys ar tyrimų rezultatai yra teisingi ir gauti sąžiningai. Šiame darbe nei viena dalis nėra plagijuota nuo jokių spausdintinių ar internetinių šaltinių, visos kitų šaltinių tiesioginės ir netiesioginės citatos nurodytos literatūros nuorodose. Įstatymų nenumatytų piniginių sumų už šį darbą niekam nesu mokėjęs.

Aš suprantu, kad išaiškėjus nesąžiningumo faktui, man bus taikomos nuobaudos, remiantis Kauno technologijos universitete galiojančia tvarka.

(vardą ir pavardę įrašyti ranka) *(parašas)*

Draudvilaitė, Kristina. *Gherkin* testų generavimo iš reikalavimų galimybių tyrimas. Magistro baigiamasis projektas / vadovas prof. dr. Rimantas Butleris; Kauno technologijos universitetas, Informatikos fakultetas.

Mokslo kryptis ir sritis: Informatikos inžinerija, technologijos mokslai

Reikšminiai žodžiai: *Gherkin, testai, generavimas, reikalavimai, automatinis.*

Kaunas, 2016. 108 p.

SANTRAUKA

Testavimas tampa neatsiejama programinės įrangos kūrimo dalimi ir dažnai savo svarbumu konkuruoja su pačiu programavimu. Norint testavimą atlikti kokybiškai ir greitai, reikia tinkamai paruošti testų scenarijus, kurie sudaromi remiantis sistemai iškeltais reikalavimais, tačiau daugumoje projektų reikalavimai dažnai kinta, todėl tenka atnaujinti jau sukurtus testus. Atnaujinimas reikalauja laiko ir atidumo, nes kyla pavojus prarasti jau turėtus duomenis, todėl testų generavimas iš reikalavimų atrodo geras šių problemų sprendimas. Šiame darbe pasirinkta analizuoti automatinį *Gherkin* testų generavimą, nes ši kalba ne tik padeda aiškiau apibrėžti reikalavimus ir jų scenarijus, bet gali būti naudojama kuriant automatinius testus. Eksperimentinis tyrimas parodė, kad *Gherkin* kalba yra suprantama, bei gali būti naudinga kuriant testus. Atlikus analizę paaiškėjo, kad yra realizuotas tik vienas sprendimas kaip generuoti *Gherkin* kalbą iš sistemos reikalavimų. Darbo rezultatas – pasiūlyti keturi algoritmai, kaip būtų galima generuoti *Gherkin* kalbą iš skirtingų reikalavimų struktūrų: veiklos diagramos, sekų diagramos, lentelių ir tekstinio formato – darbo istorijų. Šie algoritmai turi taikymo apribojimus, tačiau reikalui esant gali būti papildyti ir taikomi sudėtingesnėms reikalavimų struktūroms.

Draudvilaitė, Kristina. *Feasibility Study of Gherkin Tests Generation from Requirements*: Master‘s thesis in Information Systems Engineering / supervisor assoc. prof. Rimantas Butleris. The Faculty of Informatics, Kaunas University of Technology.

Research area and field: Informatics Engineering, Technology Science

Key words: *Gherkin, tests, generation, requirements, automatic.*

Kaunas, 2016. 108 p.

SUMMARY

Testing is becoming a concurrent part of the software development. To ensure the testing quality and rapid testing in the future, testing scenarios need to be prepared properly. These scenarios are made from software requirements, but requirements tend to change often, which cause the need to update the testing scenarios. This process cost time and it is possible to lose the important data.

Because of that the best decision for this problem could be the automated generation of test cases. The Gherkin tests generation was chosen because this language is easily understandable and could be used to create the automated tests. Experimental investigation has shown that the automated tests generation would improve the work for development teams. The analysis has shown that only one solution for Gherkin generation from requirements is realized. As a result of this thesis, four algorithms where suggested to generate Gherkin tests from the requirements: from the activity diagram, from sequences diagram, from tables and job stories. These algorithms have restrictions, however they can be improved for more complex structures of the requirements.

TURINYS

Lentelių sąrašas 8
Paveikslų sąrašas 9
Terminų ir santrumpų žodynas 11
Įvadas 12
1. Reikalavimų generavimo į *Gherkin* testus analizė 14
1.1. Analizės tikslas 14
1.2. Tyrimo objektas, sritis ir problema 14
1.3. Tyrimo objekto, veiklos proceso ir srities analizė 14
1.4. Tyrimo objekto naudotojų analizė 20
1.5. Esamų problemos sprendimo metodų analizė 20
1.5.1. Reikalavimų pavertimas į testavimui tinkamus testus 20
1.5.2. Atsekamumo matricos kūrimas, naudojant UML 23
1.5.3. *Gherkin* kalba ir jos sintaksė 24
1.5.4. *Agile* metodai ir jų sąsaja su *Gherkin* kalba 27
1.5.5. Reikalavimų aprašymas *Agile* metoduose 29
1.5.6. Reikalavimų aprašymas diagramomis 32
1.5.7. Esami sprendimai, kaip generuoti *Gherkin* kodą 38
1.5.8. Sprendimas, kaip generuoti veiklos diagramą iš *Gherkin* kodo 41
1.5.9. Testų automatizavimo įrankiai ir kalbos 44
1.6. Siekiamo sprendimo apibrėžimas 46
1.7. Analizės išvados 47
2. Sprendimo, kaip generuoti *Gherkin* kalbą iš reikalavimų, reikalavimų specifikacija ir projektas, formalus aprašas 48
2.1. Reikalavimų generavimo į *Gherkin* kalbą sprendimo aprašymas 48
2.1.1. *Gherkin* testo generavimas iš teksto 48
2.1.2. *Gherkin* testo generavimas iš sekų diagramos 52
2.1.3. *Gherkin* testo generavimas iš veiklos diagramos 64
2.1.4. *Gherkin* testo generavimas iš lentelės 78
2.2. Reikalavimų apibendrinimas 80
3. Eksperimentinis *Gherkin* generavimo iš reikalavimų metodikos tyrimas 81
3.1. Eksperimento planas 81
3.2. Eksperimento rezultatai 81
3.2.1. Apklausos rezultatai 81
3.2.2. Generavimo iš veiklos diagramų rezultatai 85
3.2.3. Generavimo iš sekų diagramų rezultatai 85
3.3. Generavimo į *Gherkin* kalbą algoritmų veikimo ir savybių analizė, kokybės kriterijų įvertinimas 86
3.3.1. Apklausos analizė 86
3.3.2. Generavimo algoritmų analizė 86
3.4. Sprendimo taikymo rekomendacijos 87
4. Rezultatų apibendrinimas ir išvados 88

5. Naudotos literatūros sąrašas 89
6. Priedai 91
6.1. priedas. Veiklos diagramos generavimas į *Gherkin* testą 1 91
6.2. priedas. Veiklos diagramos generavimas į *Gherkin* testą 2 98
6.3. priedas. Veiklos diagramos generavimas į *Gherkin* testą 3 100
6.4. priedas. Sekų diagramos generavimas į *Gherkin* testą 1 101
6.5. priedas. Sekų diagramos generavimas į *Gherkin* testą 2 103
6.6. priedas. Apklausos anketa 104

LENTELIŲ SĄRAŠAS

Lentelė 1. Veiklos diagramos elementai [21] 34
Lentelė 2. *Hexawise* matricos pliusai ir minusai 39
Lentelė 3. „Darbo istorijos“ žodžių transformavimas į *Gherkin* kalbos žodžius 50

PAVEIKSLŲ SĄRAŠAS

1 pav. Programinės įrangos kūrimas ir testavimas 15
2 pav. Tradicinio ir *Agile* metodų V-modeliai [6] 19
3 pav. Vartotojo pasakojimo struktūra 31
4 pav. „Darbo istorijos“ struktūra 31
5 pav. Sekų diagramos pavyzdys 33
6 pav. Veiklos diagramos pavyzdys 36
7 pav. Panaudojimo atvejų diagramos pavyzdys 37
8 pav. SysML reikalavimų diagramos pavyzdys 37
9 pav. Padengimo matricos pavyzdys [23] 38
10 pav. *Hexawise* matricos pavyzdys [23] 39
11 pav. Programinio kodo pavyzdys [24] 40
12 pav. Iš *Gherkin* kodo sugeneruota panaudos atvejų diagrama [25] 42
13 pav. Veiklos diagrama „Searching“ [25] 43
14 pav. Veiklos diagrama „Adding“ [25] 43
15 pav. Veiklos diagrama „Removing“ [25] 43
16 pav. Veiklos diagrama „Ordering“ [25] 43
17 pav. Darbo istorijos struktūra (anglų kalba) [33] 48
18 pav. Procesas nuo darbo istorijos iki automatinio testo kūrimo 49
19 pav. *Gherkin* kodo generavimo iš darbo istorijos algoritmas 49
20 pav. Sekų diagramos vaizdavimas 53
21 pav. *Gherkin* kodo generavimo iš sekų diagramos algoritmas 54
22 pav. Veiklos diagrama „Operatorių transformacijos“ 55
23 pav. Veiklos diagrama “Veiksmų trasformacijos” 57
24 pav. Sekų diagrama „Prisijungimas prie sistemos“ 59
25 pav. Sekų diagramos tekstinis formatas 59
26 pav. Sekų diagrama su pažymėtomis sritimis 60
27 pav. Į sritis padalintas sekų diagramos tekstas 60
28 pav. Veiklos diagramos generavimo į *Gherkin* kodą algoritmas 65
29 pav. Diagrama „Lygiagretūs keliai – paruošimas generavimui“ 66
30 pav. Diagrama „Sprendimai – paruošimas generavimui“ 68
31 pav. Diagrama „Elementų transformacijos į *Gherkin* kodą“ 69
32 pav. Diagrama „*Gherkin* kodo tvarkymas“ 70

33 pav. Veiklos diagrama „Užsakymo tvarkymas“ 71
34 pav. Sužymėti keliai veiklos diagramoje „Užsakymo tvarkymas“ 1 72
35 pav. Sužymėti keliai veiklos diagramoje „Užsakymo tvarkymas“ 2 73
36 pav. Respondentų pasiskirtymas pagal amžių 82
37 pav. Respondentų pareigos darbe 82
38 pav. Reikalavimų pateikimo forma respondentų darbe 83
39 pav. Reikalavimo pateikimo įrankiai respondentų darbe 83
40 pav. Respondentų pasitenkinimas reikalavimų išsamumu darbe 84
41 pav. Respondentų nuomonė apie testų generavimo naudingumą 84

TERMINŲ IR SANTRUMPŲ ŽODYNAS

Informacinė sistema – tai struktūrizuotas procesų ir procedūrų rinkinys, kuriame yra kaupiami ir organizuojami duomenys ir perduodami vartotojui.

Gherkin [1] – tai kalba naudojama aprašant testinius atvejus programinės įrangos įrankyje „Cucumber“. Ši kalba buvo sukurta, norint skatinti veikla pagrįstą kūrimą (angl. *Behaviour Driven Development*) visoje kūrimo komandoje - tarp programuotojų, testuotojų, analitikų ir projektų vadovų. Šios kalbos sintaksė turi tris pagrindines dalis arba raktinius žodžius: duota (angl. *Given*), kuomet (angl. *When*) ir tuomet (angl. *Then*). Po kiekvieno iš šių raktinių žodžių, rašoma po vieną žingsnį.

Jira [2] – programinės įrangos kūrimo įrankis, naudojamas *Agile* komandose.

Panaudojimo atvejis (PA) – tai panaudos atvejų diagramos elementas, aprašantis vieną konkrečią sistemos arba programos funkciją.

Reikalingų patikrinimų sąrašas (angl. *check list*) – testavimo sąrašas. Naudojamas *Agile* metoduose aprašant testus ir testavimo scenarijus.

Specifikacija – tai dokumentas, kuriame surašyti reikalavimai sistemai. Reikalavimai apibrėžia kaip sistema turėtų veikti, atrodyti, kokį rezultatą duoti ir k.t..

Testavimo atvejis (TA) – tai scenarijus, kurį atlikus galima patikrinti sistemos veikimo teisingumą.

Testavimo scenarijus – keli testai apjungti į vieną, sudarant nuoseklią veiksmų seką.

Vartotojo pasakojimas (angl. *user story*) – reikalavimo pateikimo forma *Agile* metode.

UML (angl. *Unified Modeling Language*) – vieninga modeliavimo kalba.

ĮVADAS

Šis darbas priklauso informacinių sistemų inžinerijos studijų programai. Jame tiriama automatizuotų testų atnaujinimas, pasikeitus sistemos reikalavimams.

Darbo problematika ir aktualumas

Kuriant programinę įrangą trys svarbiausi faktoriai yra laikas, kaina ir kokybė. Programinės įrangos užsakovai siekia gauti kuo kokybiškesnį produktą, kuo greičiau ir už kuo mažesnę kainą. Tuo tarpu programinės įrangos kūrimo komanda siekia pabaigti produkto realizaciją laiku ir neviršijant biudžeto, todėl dažnai skiriama nepakankamai dėmesio kokybei. Norint užtikrinti kokybę, vieni svarbiausių veiksnių yra tinkamų testų sudarymas ir sistemos testavimas. Kartais priimamas sprendimas automatizuoti testus, o ne juos atlikti rankiniu būdu.

Testai yra sudaromi remiantis sistemai iškeltais reikalavimais, tačiau daugumoje projektų reikalavimai dažnai kinta, yra atnaujinami. Pasikeitus sistemos reikalavimams, tenka atnaujinti testavimo atvejus ir jau sukurtus automatinius testus. Tam sugaištama sąlyginai daug laiko. Taip pat kyla pavojus prarasti duomenų atsekamumą ir kitą svarbią informaciją. Šiuo metu trūksta informacijos kaip efektyviai ir automatizuotai atnaujinti testus, pasikeitus reikalavimams, tuo pačiu neprarandant svarbios informacijos.

Šio darbo tyrimo sritis – informacinių sistemų reikalavimai, pritaikyti testavimui, ir testų sudarymo būdai. Tyrimo objektas – *Gherkin* kalba parašytų testų atnaujinimas, pasikeitus sistemos reikalavimams. Šie testai pasirinkti, nes jų naudojimas atneša didelę naudą, tačiau dažnai jų atsisakoma dėl laiko sąnaudų, reikalingų parašyti *Gherkin* testus, bei juos atnaujinti pasikeitus sistemos reikalavimams.

Darbo tikslas ir uždaviniai

Darbo tikslas – pagerinti testų atnaujinimą, pasikeitus sistemos reikalavimams, neprarandant informacijos.

Darbo uždaviniai:

1. Išanalizuoti reikalavimų struktūrą, pritaikytą testų kūrimui.
2. Išanalizuoti esamus reikalavimų ir testų sąsajos būdus.
3. Išanalizuoti testų sudarymo būdus iš apibrėžtų sistemos reikalavimų.
4. Pasiūlyti testų atnaujinimo metodą, kuris leistų atnaujinti testus, pasikeitus sistemos reikalavimams.
5. Sukurti anketą, skirtą analizuoti sukurtam metodui.
6. Atlikti eksperimentinius tyrimus pasiūlytam metodui.

Darbo rezultatai ir jų svarba

Atliekant šį darbą buvo sukurti keturi algoritmai, kurie galėtų generuoti *Gherkin* testus iš keturių skirtingų reikalavimų aprašymo struktūrų: veiklos diagramos, sekų diagramos, lentelės ir tekstinio formato – darbo istorijos. Komanda, naudojanti šiuos algoritmus, galėtų greitai sugeneruoti *Gherkin* testus, o pasikeitus reikalavimams nereikėtų gaišti laiko taisant testus – juos būtų galima sugeneruoti iš naujo. Naudojant šiuos testus būtų lengviau apsvarstyti kuriamus sistemos reikalavimus. Taip pat, automatizavus atskirus šių testų žingsnius, būtų sukurti automatiniai testai.

Darbo struktūra

Pirmame darbo skyriuje atliekama esamų problemos sprendimų analizė, kurios reikia, norint kurti generavimo į *Gherkin* testus algoritmus. Aprašoma *Gherkin* kalba ir jos sintaksė. Išanalizuojamos įvairios reikalavimų pateikimo formos: sekų, veiklos, būsenų, panaudojimo atvejų, *SysML* diagramos. Taip pat ištiriama kaip reikalavimai aprašomi *Agile* metoduose. Randamos reikalavimų struktūros, geriausiai tinkančios generuoti *Gherkin* testus. Taip pat šiame skyriuje atliekama jau esamų sprendimų analizė – randami du būdai generuoti *Gherkin* kodą, tačiau tik vienas iš jų generuoja iš reikalavimų. Apibrėžiamas siekiamas sprendimas.

Antrame skyriuje aprašomi sukurti algoritmai: pateikiamos algoritmų veiklos diagramos, formalūs reikalavimų aprašai, reikalavimai norint naudoti algoritmus, bei algoritmų taikymo pavyzdžiai.

Trečiame skyriuje aprašomas atliktas eksperimentas. Eksperimentas susideda iš dviejų dalių. Pirmoje dalyje buvo atlikta apklausa tarp IT srityje dirbančių žmonių ir ištirti komandų poreikiai, *Gherkin* kalbos žinomumas ir suprantamumas, poreikis generuoti testus. Antroje dalyje sukurti algoritmai išbandyti su įvairiomis sekų ir veiklos diagramų struktūromis. Taip buvo surastos esminės algoritmų vietos, bei algoritmų teisingumas.

1. REIKALAVIMŲ GENERAVIMO Į *GHERKIN* TESTUS ANALIZĖ

1.1. Analizės tikslas

Išanalizuoti galimas reikalavimų (specifikacijų) struktūras ir nustatyti kurios specifikacijos geriausiai pritaikytos kurti *Gherkin* testams. Nustatyti kada *Gherkin* testai būna dažniausiai naudojami. Aprašyti programinės įrangos kūrimo ir testavimo procesą.

1.2. Tyrimo objektas, sritis ir problema

Tyrimo objektas. *Gherkin* testų atnaujinimas, pasikeitus reikalavimams.

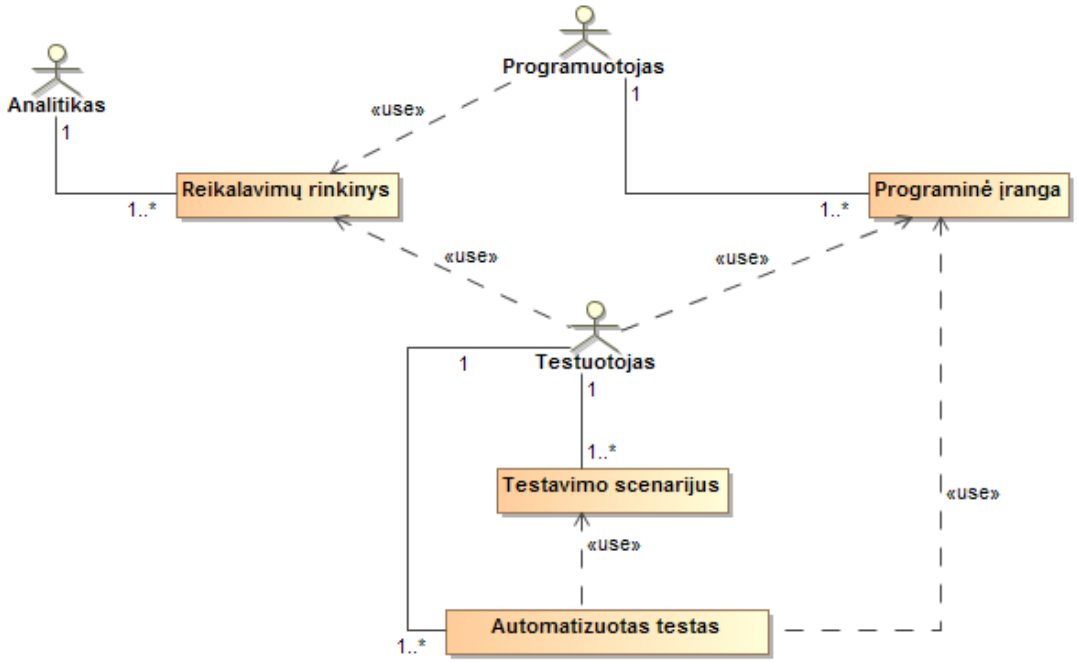
Tyrimo sritis. Informacinių sistemų reikalavimai, pritaikyti testavimui, ir testų sudarymo būdai.

Tyrimo problema. Pasikeitus sistemos reikalavimams, sugaištama sąlyginai daug laiko atnaujinant testavimo atvejus, taip pat tenka atnaujinti duomenų atsekamumą.

1.3. Tyrimo objekto, veiklos proceso ir srities analizė

PĮ kūrimo ir testavimo procesas

Prieš programinės įrangos (PĮ) kūrimo pradžią, analitikas išklauso projekto užsakovų norus ir pagal juos surašo reikalavimus kuriamai PĮ. Pagal gautus reikalavimus, programuotojai kuria PĮ, o testuotojai turi patikrinti ar sukurta sistema atitinka prieš tai iškeltus reikalavimus. Kad būtų lengviau testuoti, testuotojai sukuria testus arba testavimo scenarijus, pagal kuriuos ieškos sistemos klaidų ir tikrins ar ji atitinka anksčiau iškeltus reikalavimus. Kartais testai gali būti atliekami tik vieną kartą – pabaigus kurti PĮ. Tačiau daug dažniau pasitaiko atvejų, kai tuos pačius testus tenka atlikti daug kartų, pavyzdžiui po kiekvieno klaidos pataisymo arba po naujo funkcionalumo pridėjimo. Kad kiekvieną kartą nereikėtų atlikinėti tų pačių veiksmų, kai kurie testų scenarijai gali būti automatizuoti. Tam sukuriami testų projektai, kuriuose laikomi automatizuoti testų scenarijai. Šie scenarijai gali būti daug kartų atliekami testuojamoje sistemoje. PĮ kūrimo komandos veikėjai ir objektai pavaizduoti 1 pav..



1 pav. Programinės įrangos kūrimas ir testavimas

Reikalavimai, testavimo scenarijai ir automatizuotų testų projektai gali būti labai įvairūs.

PĮ reikalavimai skirstomi į dvi pagrindines dalis: jie gali būti funkciniai – aprašantys sistemos veikimą, arba nefunkciniai – aprašantys kitus sistemos parametrus, kaip greitis, patogumas, vartotojo sąsajos vaizdas ir k.t. PĮ reikalavimai gali būti labai įvairūs: tekstinis formatas (struktūrizuotas arba ne), vartotojo pasakojimai, įvairios diagramos, lentelės, būsimos vartotojo sąsajos vaizdas ir k.t..

Taip pat yra daug būdų aprašyti ir testavimo scenarijus. Tai gali būti paprastas sąrašas tekstinėje tvarkyklėje arba gali būti sukurti naudojant tam skirtus įrankius. Vienas iš būdų aprašyti testavimo scenarijus yra *Gherkin* kalba. Plačiau apie ją rašoma skyriuje 1.5.3 „*Gherkin* kalba ir jos sintaksė“.

Testų automatizavimas taip pat gali būti skirtingas. Skiriasi ne tik įrankiai ir programavimo kalbos, kuriais testai gali būti realizuojami, tačiau ir testo lygis. Automatizuoti testai yra išskiriami į tris pagrindines grupes / lygius:

1. Vienetų testai (angl. *unit tests*)
2. API testai
3. Vartotojo sąsajos testai

Norint sukurti pirmo ir antro lygio testus, reikia gerai išmanyti testuojamos sistemos struktūrą, todėl šiuos testus dažniausiai rašo patys programuotojai. Šiame darbe bus orientuojamasi į trečio lygio testus, skirtus testavimui iš vartotojo perspektyvos. Norint sukurti tiek automatizuotus, tiek neautomatizuotus testus, tereikia gerai žinoti sistemos reikalavimus. Be to, šis testavimo būdas apima ir kitus du lygius, nes testuojamas visos sistemos veikimas, o ne atskiros jos dalys.

Norint nustatyti kaip geriausia atnaujinti automatizuotus arba paprastus testus, pasikeitus sistemos reikalavimams, pirmiausia reikia ištirti kokias būdais gali būti pateikti sistemos reikalavimai, bei kaip jie tikrinami – testuojami.

Remiantis [3] šaltiniu, sistemos reikalavimai gali būti pateikti šiais būdais:

1. **Reikalavimai surašyti į dokumentą.** Naudojant šį reikalavimų surašymo būdą, reikalavimai pateikiami paprastai, surašius juos tekstiniu pavidalu.
2. **Modeliavimas.** Naudojant šį reikalavimų surašymo būdą, reikalavimai pateikiami naudojant diagramas, lenteles bei struktūruotą kalbą.

Kaip teigiame [4] šaltinyje, sistemos testavimas gali būti atliekamas naudojant šias pagrindines technikas:

1. **Remiantis specifikacija.**

Naudojant šį testavimo būdą, remiamasi sistemos specifikacija. Sistemos specifikacijoje būna surašyta kaip sistema turėtų ir / arba neturėtų elgtis. Analizuojant specifikaciją, išgaunami ir pasirenkami testavimo pagrindai, kuriuose surašoma kaip sistema turėtų ir / arba neturėtų elgtis, ką ji privalo atlikti ir ko neturėtų daryti. Taigi, kaip matome ši testavimo technika remiasi į sistemos elgseną.

Dažniausiai šioje technikoje naudojamas modelis, kuriuo galėtų būti grafas arba lentelė. Modelio naudojimas padeda nustatyti testavimo atvejų kiekio ir kokybės santykį, t.y. ar yra sukurta pakankamai testavimo atvejų, bei ar jų nėra per daug. Tai labai svarbu, nes sukūrus per mažai testavimo atvejų galima nepastebėti svarbių klaidų, tačiau sukūrus per daug testavimo atvejų, jie perdengs vienas kitą, todėl atsiras nereikalingų testų.

Apibendrinant, specifikacija paremtas testavimas tikrina sistemos reikalavimus, kurie dažniausiai būna funkciniai. Taigi, pagal sistemos reikalavimus būna sukuriami testavimo modeliai, o iš jų – testavimo atvejai.

2. **Remiantis sistemos struktūra.**

Ši testavimo technika remiasi vidine sistemos struktūra, pagal kurią sukuriami dinaminiai testavimo atvejai. Kitaip tariant, testavimo atvejai kuriami naudojant informaciją apie tai, kaip sistema yra sukurta.

Kuriant sudėtingas ir didelės apimties sistemas kyla pavojus, kad sistemos naudotojai įves blogą reikšmę, ištrins reikalingą elementą ar kitaip sugadins sistemą. Atsižvelgiant į tai, programuotojai turi įvertinti galimas klaidas ir į kodą įrašyti papildomų apsaugų. Tokiu būdu sudėtinga sistema tampa dar sudėtingesnė. Dėl šių aplinkybių tampa labai sunku gerai ištestuoti sistemą. Tokiais atvejais gerai pasitarnauja struktūra patemtas testavimas.

Šis testavimas remiasi programiniu kodu, todėl programuotojai turi padėti testuotojams, sukurdami pagalbinį funkcionalumą, kurio dėka testuotojas galėtų patikrinti vidinius sistemos duomenis. Atliekant tokius testavimus naudojamos „juodos dėžės“ (angl. *black-box*), „baltos dėžės“ (angl. *white box*) ir „pilkos dėžės“ (angl. *grey box*) testavimo technikos, kurios bus aptartos vėliau.

3. Remiantis testavimo patirtimi ankstesniuose projektuose ir sistemos defektais.

Ši testavimo technika nėra tokia struktūrizuota ar paremta reikalavimais, kaip prieš tai paminėtos, tačiau yra ne mažiau svarbi.

Paprastai ši testavimo technika pradedama naudoti, kai pastebimas vienas ar daugiau defektų, kurie yra pastebėti anksčiau ir jau buvo surašyti į defektų sąrašą. Dažniausiai tokie sąrašai sudaromi nuolat dirbant su panašiomis sistemomis ir jas stebint.

Kuriant defektais paremtus testus, rašomi scenarijai, kuriuos atliekant būtų galima iššaukti defektus, esančius sąraše. Pagal susitarimą testai rašomi kiekvienam defektui, tačiau visada įvertinama ar defektas yra rizikingas. Jei defektas nerizikingas –jis netestuojamas.

Šis testavimo būdas atkreipia dėmesį į programuotojų daromas klaidas. Dažnai pasitaiko, kad programuotojų komanda, atlikdama projektus yra linkusi darysi panašias klaidas. Defektais paremta testavimo technika leidžia pastebėti šias klaidas, jas struktūrizuoti, bei padeda atkreipti dėmesį į panašių klaidų atsiradimą kituose projektuose.

Knygoje „*Software Testing and Continuous Quality Improvement*“ [5] apibendrinamos anksčiau paminėtos juodos, baltos ir pilkos dėžės testavimo technikos:

1. ***Black – box testing* – juodos dėžės testavimas (funkcinis testavimas)**

Juodos dėžės arba funkciniame testavime testavimo sąlygos apibrėžiamos remiantis programos arba sistemos funkcionalumu. Tai reiškia, kad testuotojas tikrina įeinančius duomenis ir pateiktą rezultatą, visiškai nesigilindamas, kaip programa arba sistema dirba. Panašiai, kaip ir vairuojant automobilį, mums nereikia žinoti kaip jis veikia. Taip ir testuojant šiuo būdu nesigilinama į sistemos vidinę struktūrą. Keli šios kategorijos testavimo pavyzdžiai:

- Sprendimo lentelės;
- Lygiavertiškumo padalijimas;
- Intervalų tikrinimas;
- Ribinės vertės tikrinimas;
- Duomenų bazės integracijos tikrinimas;
- Priežasčių – poveikio grafikas;
- Išimčių testavimas;
- Ribų testavimas;
- Atsitiktinis testavimas.

Pagrindinis funkcinio (anlg. *black-box*) testavimo privalumas tas, kad testai sudaromi remiantis sistemos veikimui iškeltais reikalavimais, todėl juos gana lengva suprasti ir patikrinti. Šio

testavimo būdo trūkumas tas, kad neįmanoma išsamiai patikrinti visų įmanomų įvesties reikšmių. Dar vienas trūkumas tas, kad nėra žinoma vidinė struktūra bei logika, kurioje gali būti klaidų kurios nėra pastebimos testuojant šiuo metodu.

2. ***White – box testing* – baltos dėžės testavimas (struktūrinis)**

Naudojant baltos dėžės testavimo metodą yra tikrinami loginiai keliai (angl. *path*). Testuotojas tikrina vidinę programos arba sistemos struktūrą, neatsižvelgdamas į programai ar sistemai iškeltus reikalavimus. Testuotojas žino vidinę programos struktūrą ir logiką taip, kaip mašinų mechanikas žino kaip veikia automobilis. Specifiniai šio metodo pavyzdžiai galėtų būti:

- Bendra kelio analizė;
- Išsišakojimo ir šakos apimtis;
- Būsenos apimtis.

Šio metodo privalumas: sutelkiamas dėmesys į produkto kodą. Taip žinoma vidinė struktūra ir logika, randamos klaidos bei programuotojų paliktas brokas.

Trūkumai:

- Nepatikrinama realizuota sistema ar programa atitinka specifikaciją.
- Nepatikrinama ar nėra nerealizuotų vietų
- Nepatikrinamos duomenims jautrios klaidos.
- Negali patikrinti visų galimų atvejų, nes norint tai padaryti reikėtų labai didelio testų kiekio.

3. ***Gray-Box Testing* – pilkos dėžės testavimas (funkcinis ir struktūrinis)**

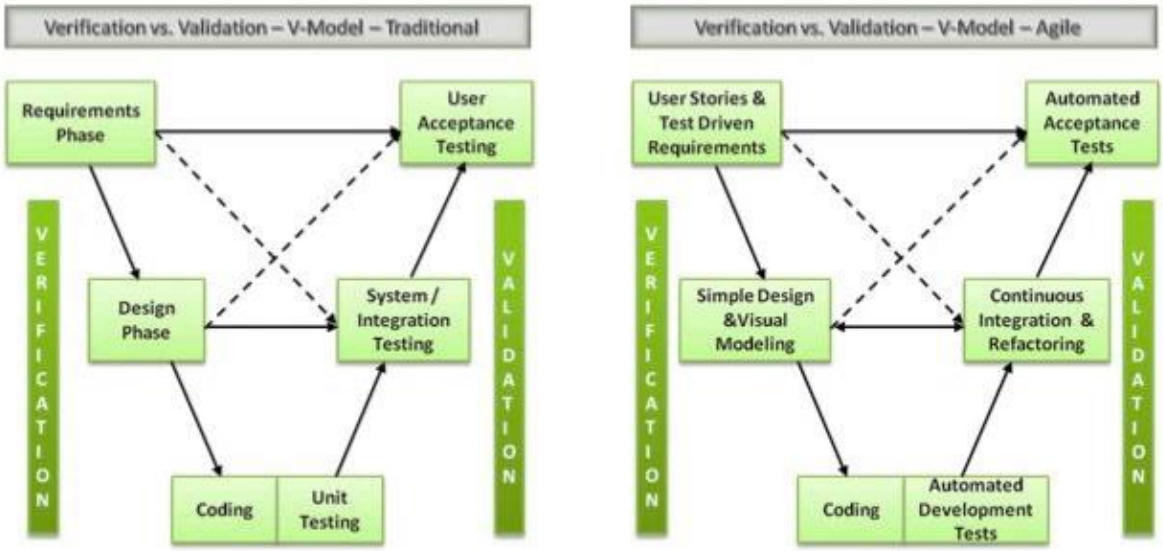
Gray-box testavimas yra *black-box* ir *white-box* kombinacija. Pagal šį metodą testuotojas išstudijuoja specifikacijos reikalavimus ir bendradarbiaudamas su programuotojais įsigilina į vidinę sistemos struktūrą. Viso to tikslas yra išsiaiškinti dviprasmiškus reikalavimus ir sukurti numanomus testus. Šio metodo privalumas tas, kad testuotojas, suprasdamas kaip veikia sistema, gali panaudoti mažiau testų tam, kad patikrintų sistemos funkcionalumą, t.y. jis gali sumažinti funkcinių reikalavimų patikrinimo aibę, išimdamas persidengiančius testus.

***Agile* reikalavimai ir testavimas**

Be tradicinio projekto vykdymo būdo, dar vadinamo krioklio (angl. *waterfall*), egzistuoja ir daug kitokių, kurie yra dinamiškesni ir labiau pritaikyti greitai besikeičiančiai aplinka. Vienas iš jų, ir turbūt populiariausiais yra *Agile* metodas. Nors *Agile* apima daug skirtingų metodikų, jos visos turi bendrus bruožus, todėl dažnai pasitaiko, kad komandos naudoja ne konkretų metodą, bet jų mišinį. Kuriuos metodus naudos, komanda pasirenka pagal savo poreikį. Į visa tai atsižvelgiant, trumpai aptarsime *Agile* reikalavimų aprašymą, projekto vykdymą ir testavimą.

Remiantis šaltiniu [6], galime išskirti pagrindinius skirtumus kuo *Agile* skiriasi nuo tradicinio (krioklio metodika paremto) projekto vykdymo:

1. Tradiciniuose metoduose dažnai manoma, kad testavimo komanda turi būti nepriklausoma ir nepriklausomai pasamdoma tam, kad dirbtų efektyviai. *Agile* praktikuojama, kad testuotojai dirba kartu su programuotojais.
2. Tradiciniuose metoduose dažnai, be atskiros testavimo strategijos ir testavimo plano, sunku įvykdyti testavimą. Kadangi *Agile* projektai yra labai dinaminiai ir nuolat kintantys, neverta apsibrėžti labai tikslaus testavimo plano, ypač tokio, kuris apimtų daugiau nei patį projektą. *Agile* testavimo strategija gali būti apibrėžta konkrečiam projektui ir tik tai daliai kuri yra realizuojama.
3. Manoma, kad patikrinimo ir patvirtinimo V-modelis negali būti pritaikytas *Agile* sprinte. Iš tiesų, *Agile* naudojamas panašus modelis. Žemiau pateikiamas *Agile* V modelis ir palyginamas su tradiciniu V modeliu (žr. 2 pav.).



2 pav. Tradicinio ir *Agile* metodų V-modeliai [6]

4. *Agile*, kaip ir daugelyje kitų metodų, testuotojai dažniausiai koncentruojasi į juodos dėžės testavimą, kuris dažnai yra susijęs su žemo lygio testavimu. Tačiau *Agile* projektuose testuotojai atlieka reikšmingą vaidmenį automatiniame programavime ir priėmimo (angl. *acceptance*) testavime, dažnai rašo automatinius testus.
5. Tradiciniuose metoduose yra paplitusi nuomonė, kad testavimo vertė matoma tik tada kai randama kuo daugiau klaidų. Iš tikrųjų testuotojo darbas yra patikrinti darbų sąrašo įrašo realizavimą ir jį praleisti tik tada, jei viskas veikia gerai.
6. Taip pat *Agile* metoduose testų automatizavimas yra labai svarbus ir būtinas. Ypač tada, kai įmonė plečia savo veiklą. Automatiniai testai padeda užtikrinti produkto kokybę.
7. Kai kuriose komandose testų automatizavimas yra toks svarbus, kad darbas nėra žymimas kaip baigtas, kol nėra parašomi jį padengiantys automatiniai testai.

***Agile* reikalavimų ir testavimo specifika**

Anot Scotto W. Amblerio, Kanados sistemų inžinieriaus [7], esminė *Agile* praktika yra prioriterizuotas reikalavimų sąrašas, vadinamas produkto sprinto darbų sąrašu (angl. *backlog*). Pagrindinė idėja yra ta, kad reikalavimai turi būti įgyvendinami pagal prioritizuotą tvarką. Šiame sąraše gali būti ne tik funkciniai reikalavimai, o prioritetų tvarka gali kisti.

Yra keli svarbūs reikalavimai testavimui, kuriuos iškelia *Agile* reikalavimų strategijos:

1. *Agile* testavimas turi būti iteratyvus. Kadangi *Agile* reikalavimai, dizainas ir kūrimas yra iteratyvūs procesai, toks pat turi būti ir testavimas.
2. *Agile* testuotojai negali sakyti, kad turi pilnas specifikacijas. Kaip žinia *Agile* reikalavimai yra identifikuojami, ištiriami ir patobulinami bėgant laikui. Nėra vienos reikalavimo fazės, kuri pateikia pilną reikalavimo specifikaciją. Dėl šios priežasties neįmanoma turėti tokios testavimo strategijos, kuri galėtų pilnai patikrinti visą specifikaciją.
3. *Agile* testuotojai turi būti lankstūs – kuo geriau pasiruošti savo darbui, naudodami tą informaciją, kurią turi esamu momentu, tuo pačiu suprasdami, kad jau rytoj ta informacija gali pasikeisti.

1.4. Tyrimo objekto naudotojų analizė

Testų ir reikalavimų susiejimas, bei testų atnaujinimo galimybės pasikeistus sistemos reikalavimams, labiausiai aktualios testuotojams ir projektų vadovams.

Testuotojams ši tema aktuali, nes jiems svarbu patikrinti, ar visi reikalavimai tinkamai realizuoti. Pasikeitus reikalavimams, testuotojas turi būti informuotas ir žinoti, kuris jo testas ir kaip susijęs su pasikeitusiu reikalavimu, kad galėtų jį pakeisti ir iš naujo patikrinti.

Projektų vadovams ši tema aktuali, nes jie turi stebėti bendrą projekto veiklą ir visus vykstančius pokyčius. Taip pat projekto vadovas, žinodamas, kaip pasikeitę reikalavimai lemia testų pokyčius, gali prognozuoti, kiek laiko užtruks pakartotinas sistemos ar programos testavimas.

Pasitaiko atvejų, kai patys programuotojai testuoja savo sukurtą produktą. Tokiu atveju ši tema aktuali ir jiems.

1.5. Esamų problemos sprendimo metodų analizė

1.5.1. Reikalavimų pavertimas į testavimui tinkamus testus

Knygoje „*Software Testing and Continuous Quality Improvement*“ [3] pateikiama informacija kaip iš reikalavimų sukuriami testavimo atvejai. Taip pat pabrėžiama, kad norint tai padaryti svarbu turėti kokybiškai apibrėžtus reikalavimus. Reikalavimų kokybės faktoriai:

Suprantamumas - suprantami reikalavimai yra surūšiuoti taip, kad juos būtų lengva peržiūrėti. Keli būdai kaip tai padaryti:

- Suskirstyti reikalavimus pagal jų objektus, pvz. klientas, užsakymas, sąskaita faktūra.
- Naudotojo reikalavimai turi būti surūšiuoti pagal veiklos procesą arba scenarijų.
- Atskirti funkcinius ir nefunkcinius reikalavimus.
- Surūšiuosi reikalavimus pagal detalumo lygį.
- Apibrėžiant reikalavimus naudoti žodį „turėtų“ (angl. *shall*).

Būtinumas – reikalavimai turi būti susiję su sistemos kūrimo užduotimis.

Galimybė patobulinti – turi būti suteikta galimybė reikalavimus pakeisti. Keli patarimai kaip tai padaryti:

- Naudoti reikalavimų kūrimui skirtas programas tokias kaip „CaliberRM“ arba „Doors“ Šios programos suteikia galimybę patogiai modifikuoti reikalavimus, ne taip kaip naudojant „MS Word“ programą. Kita vertus, jei projektas nedidelis, optimaliau būtų naudoti „MS Word“, nes nereikės gaišti laiko mokantis naudotis nauju įrankiu.
- Reikalavimams suteikti unikalius numerius.
- Fiksuoti reikalavimų priklausomumą, pvz. „Reikalavimas Y gali priklausyti nuo reikalavimo X.“

Unikalumas – neturi būti dubliuojančių reikalavimų. Reikalavimų dubliavimas reikalauja didelės priežiūros, nes pakeitus vieną reikalavimą, būtina pakeisti ir jo dublikatą. Taip pat tai gali būti klaidų reikalavimuose atsiradimo priežastimi.

Glaustumas – geras reikalavimas neturi turėti nereikalingos informacijos. Neturi būti naudojami žodžiai: kita vertus, tačiau, iš kitos pusės.

Galimybė patikrinti – turi būti įmanoma patikrinti ir patvirtinti, kad reikalavimas įvykdytas.

Atsekamumas – svarbu žinoti kuris reikalavimas priklauso nuo kitų reikalavimų ir kaip. To nežinant gali nepavykti kokybiškai patikrinti sistemos reikalavimų.

Testavimo atvejų (TA) kūrimo procesas, naudojant kokybiškus reikalavimus

Žemiau pateikiamas algoritmas, kokius žingsnius reikėtų atlikti, norit sukurti testavimo atvejus, naudojant kokybiškus reikalavimus:

1. Peržiūrėti reikalavimus.
2. Parašyti testavimo planą.
3. Identifikuoti testavimo rinkinį.
4. Pavadinti testavimo atvejus.
5. Parašyti testavimo atvejų aprašymus ir tikslus.

6. Sukurti testavimo atvejus.
7. Peržiūrėti testavimo atvejus.

Kartais testavimo atvejai kuriami naudojant panaudojimo atvejus. Geriausia tai padaryti šia seka:

1. Nupiešti panaudojimo atvejų diagramą.
2. Parašyti detalius panaudojimo atvejų aprašymus.
3. Identifikuoti PA scenarijus.
4. Sugeneruoti testavimo atvejus.
5. Sugeneruoti testavimo duomenis.

Testavimo atvejų generavimas

Testavimo atvejus sudaro testavimo įvesties duomenys, įvykdomos sąlygos ir laukiamas rezultatas. Testavimo atvejai kuriami analizuojant scenarijus ir peržiūrint panaudojimo atvejų tekstinius aprašymus. Kiekvienam scenarijui turi būti sukurtas mažiausiai vienas testavimo atvejis.

Šį algoritmą galima naudoti kuriant testavimo atvejus tiek rankiniu būdu, tiek automatiniu būdu. Vis tik siekiama viską generuoti automatiškai, todėl yra sukurtų programų, kurių pagalba galima tai padaryti.

Straipsnyje „*Generating Tests from UML Specifications*“ [8] paaiškinama kaip naudojant UML specifikacijas sugeneruoti testavimo duomenis ir pačius testus.

UML būsenų diagramos yra naudojamos pavaizduoti objekto elgesį. Kadangi šios diagramos yra labiausiai formalizuoti UML aspektai, jos suteikia natūralų pagrindą generuoti testavimo duomenis. Objekto būsena yra visų galimų atributų reikšmių kombinacija. UML moduliacijos sintaksės pavyzdys:

event-name (parameters) [guard] / action list ^ event list

Įvykio pavadinimas (angl. *event-name*) yra moduliacijos etikelė, o *parameters* (parametrai) yra reikšmės susiję su įvykiu. *Guard* (apsauga) yra prielaida, kuri kontroliuoja ar moduliacija priimtas ar ne. *Action list* (veiksmo sąrašas) ir *event list* (įvykių sąrašas) aprašo pokyčius programinėje įrangoje, kurie atsirado kaip moduliacijos rezultatas.

UML skirsto moduliacijas į penkias rūšis:

- Aukšto lygio moduliacija;
- Jungiamoji moduliacija;
- Vidinė moduliacija;
- Pabaigta moduliacija;
- Įgalinta moduliacija.

Toliau bus aprašoma tik įgalinta moduliacija. Ši moduliacija panaši į moduliacijas, kurios pagrįstos predikatų pasitenkinimo sąvoka. Įgalintą moduliaciją įgalina įvykis ir ji tampa aktyvios būsenos. Įgalinta modifikacija yra užfiksuojama kai egzistuoja bent vienas pilnas kelias einantis iš šaltinio būsenos į tikslinę būseną.

UML išskiriami keturi įvykių tipai:

Iškviečiamieji įvykiai. Jie vyksta tada, kai atsiranda užklausa sinchronizuotai taikyti specifinę operaciją.

Signaliniai įvykiai. Atstovauja konkretaus (sinchroninio) signalo priėmimą.

Laiko įvykiai. Atstovauja paskirto laiko periodo praėjimą nuo paskirto įvykio.

Pokyčių įvykiai. Jie modeliuoja įvykius, kurie įvyksta kai gaunama teigiama Būlio reikšmė kaip vieno ar daugiau susietumo atributų pokyčio rezultatas.

1.5.2. Atsekamumo matricos kūrimas, naudojant UML

Straipsnyje „*Requirements Traceability in the Model-Based Testing Process*“ [9] pateikiamas požiūris, kaip galima automatiškai sukurti atsekamumo matricą tarp reikalavimų ir testavimo atvejų. Tai naudojama kaip testavimo kūrimo procesas. Šiame dokumente pristatoma, kaip galima anotuoti UML modelį ir naudoti jį generuojant atsekamumo matricą.

1994 metais Gotelis ir Finkelsteinas taip apibrėžė reikalavimų atsekamumą: „Reikalavimų atsekamumas nurodo galimybę aprašyti ir sekti reikalavimo gyvenimą, tiek į priekį, tiek atgal. T. y. nuo jo originalios versijos iki jo įgyvendinimo ir specifikavimo, iki jo realizavimo ir naudojimo.“

Reikalavimų kintamumo sekimas projekto metu užtikrina, kad visi reikalavimai yra apsvarstomi įvairiuose gyvavimo ciklo etapuose. Žiūrint iš modeliais grindžiamo testavimo perspektyvos, tai reiškia, kad iš modelio sugeneruoti testavimo atvejai yra susieti su reikalavimais. Gerai žinomas būdas sieti reikalavimus su testavimo atvejais – atsekamumo matrica: kiekvienam reikalavimui matrica suteikia sąrašą testavimo atvejų, kurie būtini norint patikrinti reikalavimo įgyvendinimą. Taip pat vienas testavimo atvejis gali būti naudojamas tikrinant kelis reikalavimus. Automatinis atsekamumo matricos generavimas suteikia privalumą keliose programinės įrangos testavimo etapuose:

- Ji sugeneruotiems testavimo atvejams suteikia aiškius funkcinius aprėpties rodiklius iš reikalavimo pusės.
- Žinoma, kurie reikalavimai turi jiems sukurtus testavimo atvejus, o kurie ne. Tai leidžia papildyti testavimo atvejus rankiniu būdu arba patobulinti modelį bei testavimo atvejų generavimo strategiją taip, kad kiekvienas reikalavimas turėtų pilnai jį patikrinančių TA rinkinį.

- Tai suteikia grįžtamąjį ryšį į reikalavimus – kai kurie TA gali būti nesusieti su jokiu reikalavimu, kas galimai reikštų, kad reikalavimai išreikšti netinkamai.

Reikalavimų atsekamumas ir modeliais grindžiamas testavimas su UML

Funkcinių testų automatinis generavimas yra paremtas nedviprasmišku UML modeliu. Tai – abstraktus funkcinis modelis: jis aprašo, kokio elgesio tikimasi iš testuojamos sistemos, bet neįtraukia vykdymo detalių. Šis modelis turi būti pakankamai tikslus, kad leistų sugeneruoti TA, įskaitant laukiamus rezultatus, kuriuos apskaičiuos automatinė UML/OCL modelio multiplikacija. Testavimo atvejai ir jų laukiamų rezultatų poros yra tiesiogiai naudojami bandymo metu ir pagal gautus rezultatus sprendžiamas automatinis nuosprendis – testavimas sėkmingas ar ne.

Modeliuojant su UML klasių diagramos aprašo SUT duomenų modelį ir parodo jo kontrolės taškus bei kaip klasės sąveikauja tarpusavyje.

- Klasių diagramos (angl. *Instance)* – naudojamos norint paruošti sistemą testavimo generavimui: apibrėžiami objektai, jų atributų pradinės reikšmės ir sąveika tarp objektų.
- Būsenų diagramos (angl. *State Machine)* – UML patvirtinimo aparatas, susietas su pagrindine SUT klase, įformina kokio elgesio tikimasi iš tos klasės. Tai atliekama naudojant moduliacijas tarp patvirtinimų, atsakančių į naudotojo įvykius.
- Objekto apribojimo taisyklių kalba (OCL - *The Object Constraint Language*) – formaliai aprašo kokio elgesio tikimasi iš klasių operacijų.

1.5.3. *Gherkin* kalba ir jos sintaksė

Šiame skyriuje apžvelgsime *Gherkin* kalbos paskirtį ir jos sintaksės taisykles.

Gherkin kalba yra naudojama aprašant testinius atvejus programinės įrangos įrankyje „Cucumber“ ir yra pritaikyta projektams, kurie vykdomi veikla pagrįsto programavimo (angl. *Business Driven Development*) principu.

Kaip minima knygoje apie „Cucumber“, pavadinimu „The Cucumber Book“ [1], priežastis, kodėl buvo sukurta *Gherkin* kalba, yra ta, kad kuriant programinę įrangą būdavo išvaistoma daug laiko ir išteklių dėl komunikacijos trūkumo. Pagrindinė to priežastis – nesusikalbėjimas tarp projekto užsakovų ir kūrimo komandos. Norint to išvengti, buvo sukurta technika, kuri palengvina šią komunikaciją. Technikos esmė – aprašyti konkrečius pavyzdžius, remiantis tikro pasaulio situacijomis ir taip perteikiant norimą sistemos elgesį. Toks sprendimas padeda projektų užsakovams įsivaizduoti save naudojančius sistemą, todėl jie gali pateikti atsiliepimus, programuotojams, dar net neparašius nė vienos kodo eilutės.

Galiausiai, šie pavyzdžiai yra išgryninami ir tampa esminėmis gairėmis arba reikalavimais, kaip turi veikti sistema. Vėliau, sukūrus produktą, yra tikrinama ar sistema atitinka anksčiau apibrėžtus reikalavimus. Taigi, be to, kad ši kalba padeda „susikalbėti“ visiems komandos nariams, ji taip pat naudojama ir priėmimo (angl. *acseptance*) testams rašyti. Šie priėmimo testai gali būti atliekami rankiniu būdu arba kaip automatiniai testai, tačiau norint automatizuoti *Gherkin* kalbą, reikia parašyti automatizuotą kodą kiekvienai *Gherkin* testo eilutei.

***Gherkin* kalbos formatas ir sintaksė**

Kaip teigiama knygoje „*The Cucumber Book*“ [1], *Gherkin* failo struktūra ir prasmė sukuriama naudojant tam tikrus raktinius žodžius. Kadangi originali *Gherkin* kalba yra sukurta remiantis anglų kalba, čia pateikiama angliška raktinių žodžių versija su lietuvišku vertimu:

„Feature“ – ypatybė, funkcionalumas. Kiekvienas *Gherkin* failas prasideda šiuo raktiniu žodžiu. Šis žodis neturi jokios įtakos testo veiklai – čia tik pateikiama trumpa dokumentacija apie realizuojamą testų grupę. Pirma eilutė po žodžio „Feature“ yra laikoma ypatybės pavadinimu, o likusios – aprašymu.

„Background“ – prielaida. Kaip rašoma „Cucumber“ *Gherkin* kalbos dokumentacijos puslapyje [10], gali pasitaikyti atvejų, kai keliems testams reikia parašyti tokias pačias pradines sąlygas „Given“. Naudojant **„**Background**“**, šias sąlygas galima iškelti ir priskirti norimai testų grupei. Šios sąlygos turi būti parašytos prieš pirmą scenarijų.

„Scenario“ – tai konkretus scenarijus, kuris iliustruoja veiklos taisyklę [10]. Be to, kad scenarijus atlieka specifikacijos ir dokumentacijos funkciją, taip pat jis atlieka ir testo funciją. Scenarijus susideda iš žingsnių: „Given“, „When“, „Then“.

„Given“ žingsniai yra naudojami aprašant pradinį sistemos kontekstą. Dažniausiai tai kažkas, kas jau įvyko praeityje. Kai „Cucumber“ vykdo šį žingsnį, jis paruošia sistemą, kad ji būtų aprašytoje būsenoje, pavyzdžiui sukuria tam tikrus objektus ar prideda duomenų į duomenų bazę [1].

„When“ žingsniai naudojami aprašant įvykį arba veiksmą. Tai gali būti žmogaus arba kitos sistemos veiksmai su testuojama sistema. Rekomenduojama turėti vieną „When“ žingsnį scenarijuje. Jei jų yra daugiau, gali būti, kad reikėtų kelių skirtingų testų [10].

„Then“ žingsnis naudojamas aprašant baigtį arba rezultatą, kurio tikimasi. Šie žingsniai naudojami kaip patvirtinimas (angl. *assertion*), kad rezultatas, kurio tikimasi sutampa su rezultatu, kuris buvo gautas [10].

Paprasto *Gherkin* testo pavyzdys [10]:

```
Feature: grąžinama prekė
Scenario: Jonas grąžina sugedusį televizorių
Given Jonas pirko televizorių už 2000 eurų
```

```
And Jonas turi pirkimo čekį
When jis grąžina televizorių
Then Jonui turi būti sumokėta 2000 eurų
```

Jei norima pridėti kelis vienodus „Given, When, Then“ žingsnius šalia, tuomet žingsniai atskiriami žodžiais **„And“** (liet. ir) arba **„But“** (liet. bet)

Jei nenorima naudoti anksčiau minėtų žingsnių pavadinimų, scenarijus galima aprašyti ir jų nenaudojant, tačiau prieš kiekvieną žingsnį pridedant „*“ ženklą. Taip prarandama dalis informacijos, tačiau kai kam toks testas atrodo patogiau skaitomas.

Jei reikia sukurti testą sudėtingai veiklos taisyklei, kuri naudoja skirtingus įvedimo ir išvedimo duomenis, gali būti patogu naudoti **„Scenario Outline“** ir **„Examples“** raktinius žodžius. Pirmasis, naudojamas aprašyti testui, kuris bus kartojamas kelis kartus, o antrasis – aprašyti duomenis, kurie bus naudojami vykdant testus. Pavyzdžiui [10]:

```
Scenario: pamaitinti karvę
    Given karvė sveria <svoris> kg
    When apskaičiuojami maitinimo reikalavimai
    Then energija turi būti <energija> MJ
    And proteinų kiekis turėtų būti <proteinai> kg

    Examples:
      | svoris | energija | proteinai |
      |    450 |   26500 |       215 |
      |    500 |   29500 |       245 |
      |    575 |   31500 |       255 |
      |    600 |   37000 |       305 |
```

Taip pat *Gherkin* kalba turi papildomus raktinius ženklus:

@ – skirtas pažymėjimams (angl. *tags*)

– skirtas komentarams

| – naudojamas kuriant duomenų lenteles prie raktinio žodžio „Examples“.

""" – dokumentų kabutės (angl. *doc strings*), naudojamos išskirti žingsnio reikšmei, kai vieno žingsnio aprašymas netelpa vienoje eilutėje.

1.5.4. *Agile* metodai ir jų sąsaja su *Gherkin* kalba

Gherkin kalba yra naudojama aprašant testinius atvejus programinės įrangos įrankyje „Cucumber“. Ši kalba buvo sukurta, norint skatinti veikla pagrįstą kūrimą (angl. *Behaviour Driven Development*) visoje kūrimo komandoje - tarp programuotojų, testuotojų, analitikų ir projektų vadovų. Kadangi šis kūrimo būdas yra *Agile* metodikų dalis, trumpai apžvelgsime populiariausias *Agile* metodikas. Pagrindiniai *Agile* teiginiai:

1. Individai ir sąveikos svarbiau už procesą ir įrankius.
2. Veikianti įranga svarbiau už išsamią dokumentaciją.
3. Kliento bendradarbiavimas svarbiau už kontrakto derybas.
4. Reaguoti ir keistis svarbiau už plano laikymąsi.

Labiausiai paplitę *Agile* metodai [11]:

- *Extreme Programming* (XP);
- *Crystal;*
- *Dynamic Systems Development Method* (DSDM);
- *Behaviour – Driven Development* (BDD);
- *Test – Driven Development* (TDD).

***Extreme Programming* (XP)** turbūt yra labiausiai paplitęs ir daugiausiai dokumentuotas *Agile* metodas. Kaip rašoma knygoje „*Agile software development*“ [12] ši metodika susideda iš kelių skirtingų praktikų, kurių laikantis galima pasiekti geriausių rezultatų. XP praktikos:

- Klientas tampa komandos nariu;
- Reikalavimai aprašomi vartotojo pasakojimais;
- Dirbama trumpais ciklais;
- Naudojami priėmimo testai;
- Programuojama poromis;
- Naudojama testavimo paremtas programavimas (angl. *test driven development*);
- Grupinė atsakomybė;
- Tęstina integracija;
- Atvira darbo aplinka;
- Planavimas;
- Paprastas dizainas;
- Nuolatos palaikomas tvarkingas kodas.

Šiame darbe aktualu apžvelgti kelias iš šių praktikų. Visų pirma reikia atkreipti dėmesį, kad programa kuriama dirbant trumpais ciklais, kas reiškia, kad realizuojama po mažą sistemos dalį. Tai lemia, kad ir testavimas atliekamas tik mažoms sistemoms dalims. Taip pat labai svarbu pastebėti, kaip

bendraujama su klientu – jis įtraukiamas į komandos darbą, o tai reiškia, kad su juo nuolatos palaikomas ryšys, užduodami klausimai klientui, bei gaunami jo pasiūlymai ar, dažnu atveju, pakeitimai. Žmogiškasis faktorius lemia, kad tik pamačius veikiančią sistemos versiją klientams kyla minčių kaip ją reikėtų patobulinti, todėl nors ir labai anksti, reikalavimai gali žymiai pasikeisti ir ne vieną kartą. Iš čia galime daryti išvadą, kad šiame magistro darbe realizuojama programa turi būti adaptyvi besikeičiantiems reikalavimams.

Patys reikalavimai apibrėžiami vartotojo pasakojimais. Kaip rašoma knygoje, vartotojo pasakojimai reikalingi tik tam, kad suprastume užduoties esmę ir galėtume ją planuoti. Tam nereikia išsamių reikalavimų aprašymų, tuo labiau, kad žinoma, jog specifiniai reikalavimai laikui bėgant keisis.

Bendraujant su klientais, vykdomas ir priėmimo testų (angl. *acceptance testing*) parengimas. Tai automatiniai testai, kurie auga, vystantis sistemai. Šie testai vėliau yra paleidžiami kiekvieną dieną, norint įsitikinti ar sistema vis dar veikia. Viena iš galimų tokių testų formų gali būti ir šiame darbe nagrinėjami *Gherkin* testai.

Kitas svarbus XP aspektas yra tai, kad čia naudojamas testavimu grįstas programavimas, kai remiantis reikalavimais, parašomi vienetų (angl. *unit*) testai, o vėliau kodas, kurio dėka testai sėkmingai įvykdomi.

Kitas *Agile* metodas yra ***Crystal***. [11] Jo pagrindinės praktikos: komandinis darbas, komunikacija, paprastumas, dažni išleidimai į realią aplinką (angl. *release*), didelis vartotojų įtraukimas, prisitaikymas.

Crystal prielaidos:

- Kiekvienam projektui reikia šiek tiek kitokių strategijų, susitarimų arba metodologijos.
- Darbų kokybė priklauso nuo juos atliekančių žmonių savybių.
- Geresnis ir dažnesnis bendravimas sumažina tarpinių darbo produktų kiekį.

Dynamic Systems Development Method **(DSDM)** [13] filosofija teigia, kad bet koks projektas turi būti suderintas su aiškia apibrėžtais strateginiais tikslais ir fokusuotis ties išankstinio pristatymo nauda verslui. Ši metodologija nepriklauso nuo pardavėjo ir apima visą kūrimo gyvavimo ciklą. Svarbiausios DSDM praktikos:

- Paprasti seminarai (angl. *workshop'ai*);
- Modeliavimas ir iteratyvus programavimas;
- Prioritizavimas;
- Laiko fiksavimas (angl. *timeboxing*).

Taip pat *Agile* naudojamos jau minėtos praktikos: veikla pagrįstas kūrimas (ang. *Behaviour – Driven Development*) (BDD) ir testavimu pagrįstas kūrimas (angl. *Test – Driven Development*) (TDD). Kaip jau galima suprasti iš pavadinimų, pirmoje praktikoje viskas kuriama, remiantis realizuojama veikla – siekiama, kad sukurtas produktas elgtųsi taip, kaip buvo nuspręsta prieš pradedant jį realizuoti. Tam naudojami reikalavimai. Antruoju atveju, kūrimas remiasi testavimu – prieš pradedant kurti sistemą, parašomi testai, kurie turėtų būti sėkmingai įvykdyti, juos paleidus sukurtoje sistemoje. Po kiekvieno pakeitimo sistemoje, šie testai paleidžiami ir tikrinama ar sukuriant daugiau funkcionalumo nebuvo sugadintas anksčiau realizuotas funkcionalumas. Šis būdas padeda kuo anksčiau pastebėti sistemos klaidas.

Galima daryti išvadą, kad *Gherkin* testai, kurie aprašo sistemos scenarijus ir kurie tampa patikrinimo kriterijais, gali būti labai naudingi tiek pradinėje kūrimo stadijoje, kai reikalavimai yra kuriami kartu su klientu, tiek sukūrus sistemą ir tikrinant ar visos jos vietos veikia taip, kaip tikimasi.

1.5.5. Reikalavimų aprašymas *Agile* metoduose

Nors *Agile* manifestas [14] teigia, kad veikianti programinė įranga yra svarbiau už visapusišką ir plačią dokumentaciją (angl. "*Working software over comprehensive documentation*"), dauguma *Agile* naudotojų teigia, kad bent dalinė dokumentacija yra labai pageidautina. Tai padeda geriau suprasti vartotojų poreikius, suplanuoti ir apibrėžti siekiamą tikslą, bei išsamiau pateikti užduotis programuotojų komandai. Yra keli lygiai ir būdai, kaip yra dokumentuojami reikalavimai *Agile* metoduose. Vis tik šis apibrėžimas nepriklauso nuo naudojamos *Agile* metodikos, nes čia nėra jokių taisyklių. Toliau pateikiami populiariausi ir labiausiai naudojami būdai (šaltinis [15]).

Nėra apibrėžta kaip reikėtų aprašyti reikalavimus *Agile* metodikoje, ten svarbiausia kuo mažiau dokumentuoti. Anot Sheris Mansouro, techninių produktų vadybininko, dirbančio „Atlassian", nėra tokio dalyko kaip teisingas būdas dirbi su *Agile* teisingai.

Kaip kurti supratimą, kuriuo bus galima dalintis (angl. *building a shared understanding*) [16]:

1. Išsiaiškinti klientų pastebėjimus (angl. *communicate observations*);
2. Interpretuoti problemas (angl. *nterpret problems*);
3. Susieti problemas su galimybėmis (angl. *connecting problems with opportunities*)
   a. Modeliuojami klientų reikalavimai: kiekvienas reikalavimas turi atskirą puslapį. Puslapiuose sudedama visa informacija apie reikalavimą, įdedami pavyzdžiai (fotografijos, maketai);
   b. Reikalavimai susiejami (sudedamos nuorodos) su kitais reikalavimais, jau įvykdytais funkcionalumais, pokalbių su klientais aprašais ir viskuo kas aktualu.

Galimo *backlog* įrašo pavyzdys:

As A (User)

I Want (Objective) (feature)

So That (Why) (benefit)

As Alana

I want plan an event

So that my team can run effectively

Šis pavyzdys gana tipinis *Agile* metodikoje, nes trumpai ir aiškiai aprašo situaciją, reikiamą produktą ir priežastį, kodėl jis turi būti realizuotas.

Agile technikos, kurios padeda kurti vartotojo pasakojimą (angl. *user story*) [16] **:**

- Prototipai (angl. *prototypes*)
- Analitika (angl. *analytics*)
- Filmukai, pokalbiai, interviu (angl. *videos, talks and interviews*)
- Apklausos (angl. *surveys*)
- Kelionių žemėlapiai (angl. *journey maps*)
- Reikalavimų dokumentacija (angl. *requirements documents*)

Plačiau apie vieną iš jų – **kelionių žemėlapius** (angl. *journey maps*). Tai grafiškai pavaizduoti įvykiai, planavimai, bei susitikimai, kurie įvyko planuojant ir įgyvendinant užduotį.

Reikalavimų dokumentacija:

Naudojant *Atlassian* po reikalavimais darbuotojai gali rašyti komentarus, todėl kartais išsivysto ilgos diskusijos. Jas gali būti sunku skaityti, tačiau jos naudingos, nes kažkas gali parašyti savo sprendimą, o kitas gali tame sprendime įžvelgti galimų grėsmių arba papildyti sprendimą.

Reikalavimų aprašymas:

1. Aprašyti dokumento atsakomybes / nuosavybes (angl. *properties*). Turi būti nurodyti atsakingi asmenys.
2. Tikslai – trumpi ir aiškūs.
3. Pagrindas ir strateginis tinkamumas (angl. *background and strategic fit*) Kodėl mes tai darome? Kaip tai įsipina į mūsų bendrą viziją?
4. Prielaidos (ang. *assumptions*). Aprašomos techninės, veiklos arba vartotojo prielaidos, kurios gali būti įgyvendintos.
5. Vartotojo pasakojimai (angl. *user stories*). Aprašomi vartotojo pasakojimai, kurie susiję su reikalavimu.
6. Vartotojo indėlis / sąveika ir dizainas (angl. *user interaction and design*). Tai, kas buvo aptarta su vartotojais.

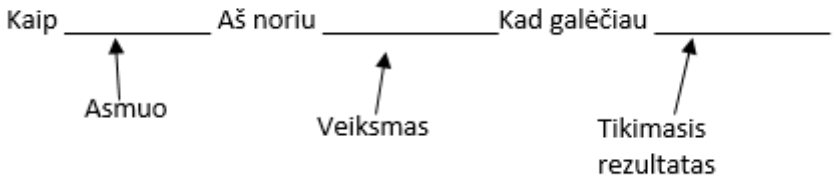
7. Klausimai. Dažnai sprendžiant problemą iškyla neatsakytų klausimų, todėl verta juos užsirašyti.

8. Dalykai, kurie nebus daromi (angl. *not do*). Apibrėžiant tai, kas nebus daroma, padeda komandai susitelkti ties svarbiausiomis užduotimis.

Kaip teigia Gerris Clapsas – *Agile* projektų vadybininkas [16], yra daug būtų dokumentuoti būsimą funkcionalumą, arba klaidą, kurią žadama taisyti. Keli iš jų:

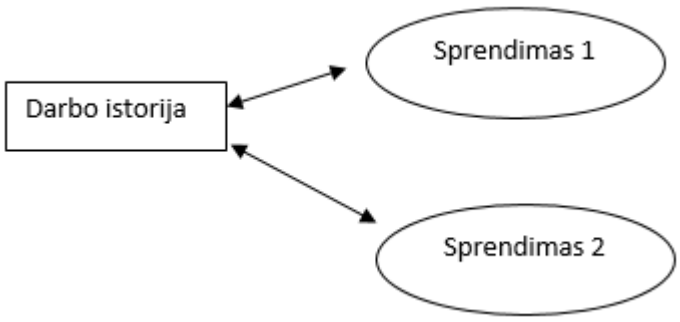
- Vartotojo pasakojimai (ang. *user story*), aprašantys sprendimus.
- Darbo istorijos (angl. *job story*), aprašančios vartotojų tikslus.
- Priėmimo kriterijai (angl. *acceptance criteria*), nustatantys kokybės užtikrinimą.
- Produkto žemėlapis (angl. *product roadmap*), aprašantis strategiją.

Kitais žodžiais, sukuriamas produkto žemėlapis, kuris pagrįstas vartotojų darbo istorijomis. Šios patobulinamos naudojant vartotojų pasakojimus, kurie aprašo kokybę su priėmimo kriterijais.



3 pav. Vartotojo pasakojimo struktūra

Autoriaus manymu pirmos dvi vartotojo pasakojimo dalys suteikia per daug prielaidų. Jo manymu svarbu minimizuoti priklausomybių kiekį tarp vartotojų pasakojimų. Tam gali padėti darbo pasakojimai. Viena darbo istorija gal būti išsprendžiama keliais vartotojo pasakojimais. Tikslas gali būti pasiektas keliais skirtingais sprendimo būdais, tačiau kiekvienas jų gali būti skirtingai efektyvus. Nereikia pasiriboti pirmu rastu sprendimo būdu.



4 pav. „Darbo istorijos“ struktūra

Darbo istorijos formatas anglų kalba:

When <situation>
I want to <motivation>
So I can <expected outcome>

Darbo istorijos formatas lietuvių kalba:

Kai <situacija>
Aš noriu <motyvacija>
Kad galėčiau <rezultatas, kurio tikimasi>

Anot šaltinio [17], *Agile* projektai, ypač *Scrum* metodas, naudoja produkto darbų sąrašą (angl. *backlog*). Tai prioretizuotas funkcionalumų, kuriuos reikia atlikti produkte, sąrašas. Nors įrašai *backlog*'e gali būti bet kokios formos, geriausia ir populiariausia forma yra vartotojų pasakojimai. Nors tradicinė vartotojo pasakojimo forma yra ("*As a user, I want* …"), bet ji nėra užbaigta, kol kyla įvairių klausimų. Todėl patartina šalia vartotojo pasakojimų pridėti reikiamų diagramų, formulių, kaip atlikti skaičiavimus ar kitų reikalingų elementų.

1.5.6. Reikalavimų aprašymas diagramomis

Kartais, norint aiškiau pateikti reikalavimus, jie aprašomi įvairiomis diagramomis. Tam gali būti naudojamos šios UML diagramos:

- Sekų diagrama
- Veiklos diagrama
- Būsenų diagrama
- Panaudojimo atvejų diagrama

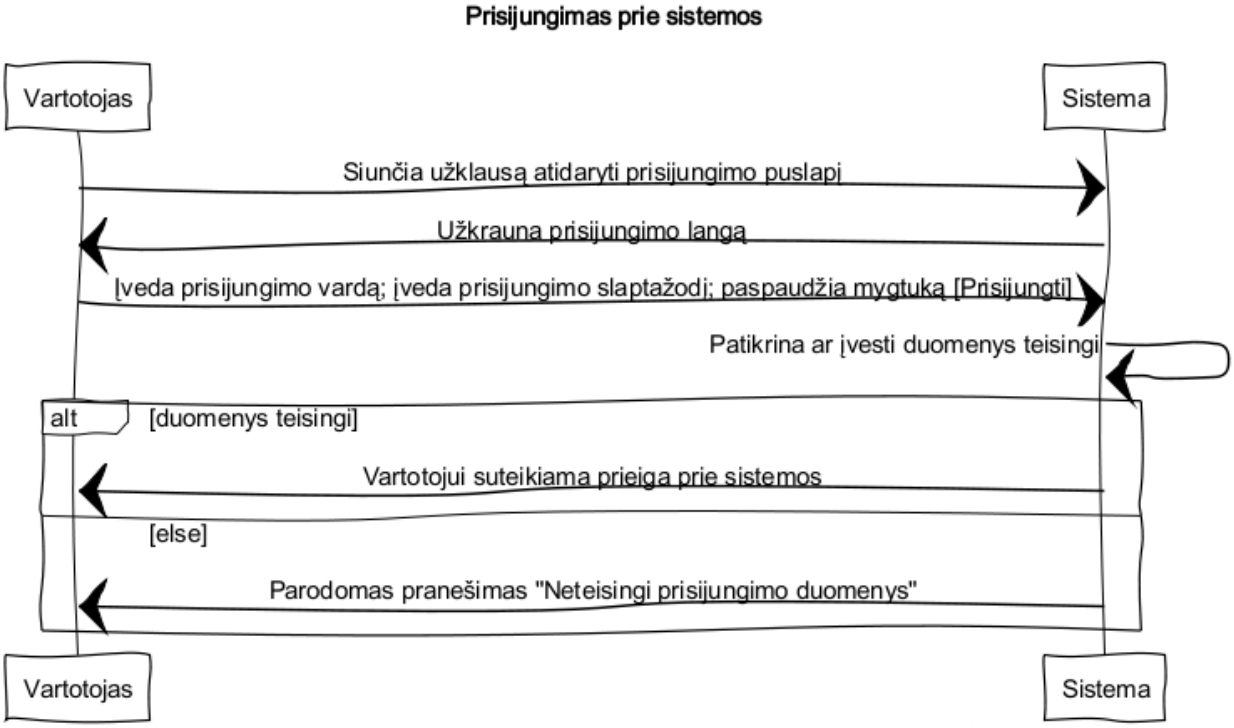
Taip pat gana dažnai, aprašant reikalavimus gali būti naudojama *SysML* diagrama.

1.5.6.1. Reikalavimų aprašymas sekų diagramomis

Sekų diagramos aprašo objektų sąveiką ir jų elgseną nurodytose situacijose. Šioje diagramoje objektai „bendrauja" pranešimų pagalba. Kiekvienas objektas šioje diagramoje turi savo gyvenimo liniją, o pranešimai tarp objektų vaizduojami rodyklėmis tarp linijų.

Aprašant reikalavimus sekų diagrama, būsimi vartotojų, sistemų ir kitų komponentų „veiksmai" pavaizduojami sekų diagramoje. Norint aprašyti reikalavimus iš vartotojo pusės, patartina braižyti aukšto lygio sekų diagramą, nesismulkinant į techninį sistemos veikimo vaizdavimą. Tokioje diagramoje būtų tik du veikėjai – vartotojas ir sistema, o visas sistemos veikimas būtų aprašomas tik tiek kiek reikia vartotojui suprasti, pavyzdžiui: sistema tikrina ar duomenys teisingi, sistema parodo

klaidos pranešimą. Toliau pateikiamas sekų diagramos pavyzdys, sukurtas puslapyje pavadinimu „WebSequenceDiagrams“ [18].



5 pav. Sekų diagramos pavyzdys

Šiame pavyzdyje pateikiamas paprastas atvejis, kai vartotojas, norėdamas prisijungti prie sistemos, turi įvesti prisijungimo vardą, slaptažodį ir paspausti mygtuką [Prisijungti]. Jei įvesti duomenys teisingi, jis gauna prieigą prie sistemos. Norint pavaizduoti galimas skirtingas veiksmų pabaigas, panaudotas alternatyvos operatorius [alt], tačiau sekų diagramos turi ir daugiau operatorių [19]:

Alt – alternatyva. Vykdoma viena iš alternatyvų. Visos alternatyvos užrašomos laužtiniuose skliausteliuose ir turi atskirą regioną. Paskutinė alternatyva visada būna [else] ir yra vykdoma, jei netenkinama nė viena alternatyvos sąlyga.

Loop – ciklas, vykdomas tol, kol galioja operatoriaus viršuje užrašyta sąlyga. Sąlyga tikrinama kiekvieną kartą prieš iš naujo pradedant ciklą.

Opt – pasirenkamasis vykdymas. Vykdoma tik tuomet, jei galioja sąlyga užrašyta laužtiniuose skliausteliuose.

Ref (ang. *Refencing*) – operatorius rodantis, kad toje vietoje įterpiama dar viena sekų diagrama. Šio operatorius viduje užrašomas naudojamos sekų diagramos pavadinimas.

Par – lygiagretus vykdymas. Gana dažnai pasitaiko, kai sistema lygiagrečiai turi vykdyti kelis veiksmus. Tam naudojamas šis operatorius. Jo sritis padalinta į daug regionų, kurių kiekvienas

vaizduoja lygiagretų vykdymą. Vienas toks regionas gali apimtis kelias gyvavimo linijas. Svarbu, kad sąlyginė pranešimų tvarka lygiagrečiuose regionuose būtų nepriklausoma, nes priešingu atveju šis operatorius netiktų.

Break – klaida. Jei išpildoma skliausteliuose nurodyta sąlyga, vykdymas nutraukiamas.

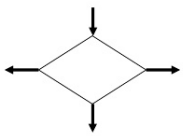
Be šių operatorių, sekų diagramose galima rasti ir kitų: *assert*, *consider*, *critical*, *neg*, *seq*, *strict* ir kitus.

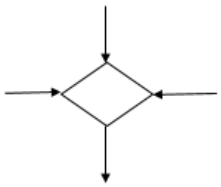
1.5.6.2. Reikalavimų aprašymas veiklos diagramomis

UML veiklos (angl. *activity*) diagramos yra skirtos vizualiai pavaizduoti darbo eigą [20]. Kaip minima IBM išleistame straipsnyje „*Activity Diagrams: What They Are and How to Use Them*“ [20], šios diagramos turi svarbias notacijas, kurios yra būtinos norint suprasti veiklos diagramas.

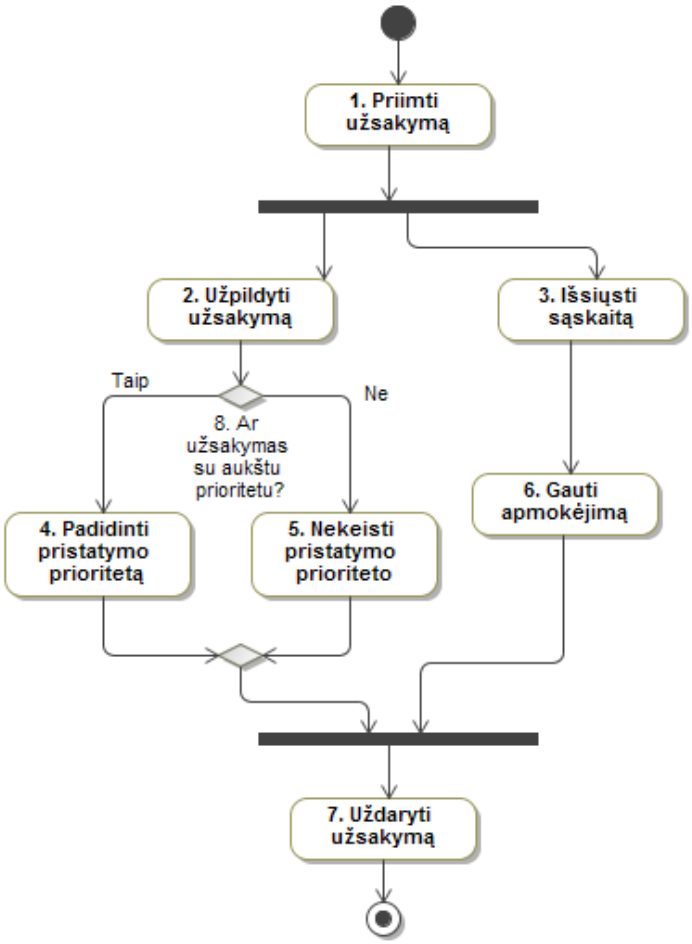
Pagrindinių veiklos diagramos elementai pateikti lentelėje:

Lentelė 1. Veiklos diagramos elementai [21]

Pavadinimas	Grafinis žymėjimas	Aprašymas
Pradžios simbolis (angl. *the start symbol*)		Žymi proceso pradžią veiklos diagramoje. Gali būti naudojamas vienas arba su komentaru.
Veiklos simbolis (angl. *the activity symbol*)		Pagrindinis diagramos elementas, žymintis vieną žingsnį veikloje.
Objekto būsena (angl. *object node*)		Apibūdina objektų srautą veikloje ir žymi objektų judėjimą veiklos diagramoje.
Sujungimo simbolis (angl. *the connector symbol*)		Rodyklė jungianti veiklas. Žymi kryptingą veiklos judėjimą.
Sujungimas (angl. *the join symbol*)		Tai tiesi vertikali arba horizontali linija, į kurią įeina dvi ar daugiau veiklų, o išeina tik viena.
Išsišakojimas (angl. *the fork*)		Tai tiesi vertikali arba horizontali linija, į kurią įeina viena veikla, o išeina kelios.
Sprendimo simbolis (angl. *the descision symbol*)		Deimanto formos, atvaizduoja įvairių srautų apjungimą arba sprendimą/atskyrimą. Apjungimo atveju įeina daug rodyklių, išeina tik viena, sprendimo atveju – įeina viena, išeina kelios.

Pavadinimas	Grafinis žymėjimas	Aprašymas
Apjungimas (angl. *merge*)		Deimanto formos, atvaizduoja įvairių srautų apjungimą arba sprendimą/atskyrimą. Apjungimo atveju įeina daug rodyklių, išeina tik viena, sprendimo atveju – įeina viena, išeina kelios.
Komentaras (angl. *the note*)		Naudojamas papildomai informacijai, kuri negali būti atvaizduota diagramoje, rašyti.
Pasirenkamo ciklo simbolis (angl. *the option loop symbol*)		Leidžia modeliuoti pasikartojantį veiksmą.
Veiklos pabaiga (angl. *flow final node*)		Pabaigia veiklos srautą. Ji pabaigia visus veiksmus atkeliavusius iki veiklos pabaigos, tačiau neturi įtakos kitiems veiklos srautams.
Veiklos srauto pabaiga (angl. *activity final node*)		Vaizduoja proceso arba darbo eigos (angl. *workflow*) pabaigą.

Veiklos diagramos pavyzdys:



6 pav. Veiklos diagramos pavyzdys

1.5.6.3. Reikalavimų aprašymas būsenų diagramomis

Būsenų diagramoje yra patogu pateikti įvairias būsenų kaitas. Jei specifikuojamoje srityje objektai dažnai keičia būsenas arba jei būsenos yra labai svarbus aspektas, patartina naudoti būsenų diagramą, kurioje būtų galima aiškiai matyti kada ir į kokias būsenas gali pereiti specifikuojamas objektas.

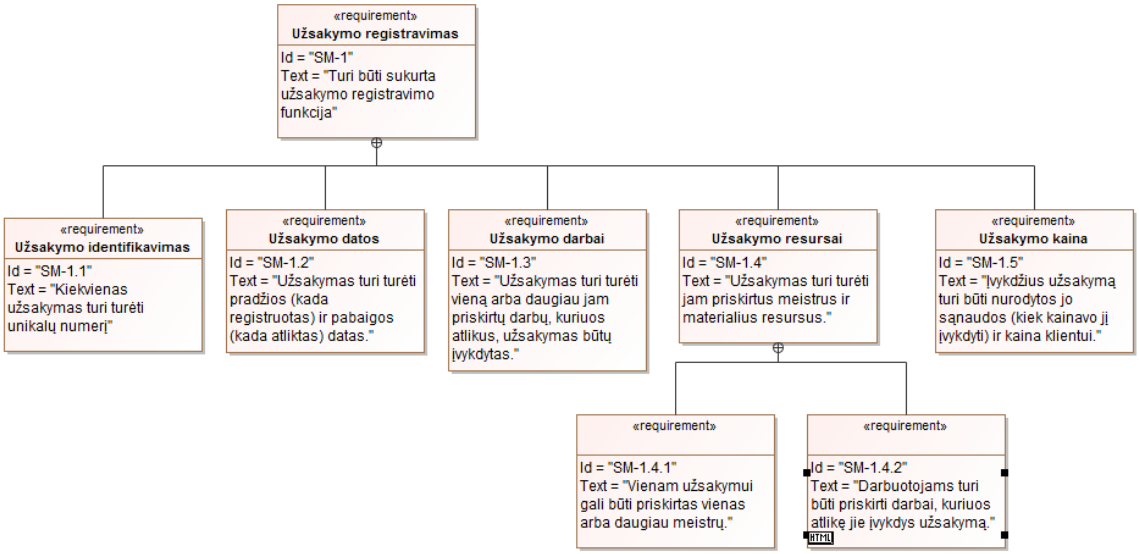
1.5.6.4. Reikalavimų aprašymas panaudojimo atvejų diagramomis

Panaudojimo atvejų diagramoje vaizduojamos sistemos vartotojų rolės (aktoriai) ir veiksmai, kuriuos jie gali atlikti (panaudos atvejus). Visi veiksmai aprašomi iš vartotojo perspektyvos. Nors įmanoma viską surašyti į vieną diagramą, dažnai, aprašant didelės sistemos panaudojimo atvejus, diagramoje pateikiami tik rolių pavadinimai ir veiksmų pavadinimai, kuriuos gali atlikti kiekviena rolė. Kiekvienas veiksmas turi po kodą (identifikacinį numerį), o išsamūs veiksmų reikalavimai pateikiami žemiau, tekstiniu pavidalu.

7 pav. Panaudojimo atvejų diagramos pavyzdys

1.5.6.5. Reikalavimų aprašymas SysML diagramis

SysML – tai sistemų modeliavimo kalba, kurios tikslas vizualiai modeliuoti sistemų inžinerines aplikacijas. *SysML* palaiko techninės ir programinės įrangos, duomenų srautų ir personalo bei procesų sistemų specifikavimą, projektavimą, analizę ir verifikavimą. *SysML* pateikia 9 diagramas sistemoms modeliuoti [22]. Viena iš jų yra reikalavimų diagrama (8 pav.), kuri leidžia aprašyti reikalavimus, juos susieti tarpusavyje, suskirstyti į lygius, sukuriant reikalavimų medžius. Taip pat ši kalba leidžia sugeneruoti reikalavimų lentelę ir reikalavimų išvesties žemėlapį.



8 pav. SysML reikalavimų diagramos pavyzdys

Galime pastebėti, kad panaikinus kiekvieno iš reikalavimų pavadinimus, o jų hierarchiją „perkėlus“ į identifikacinius numerius, pvz. iš pateikto pavyzdžio: SM-1, SM-1.1, SM-1.2, ... , SM-1.5, galime gauti tekstinius reikalavimus.

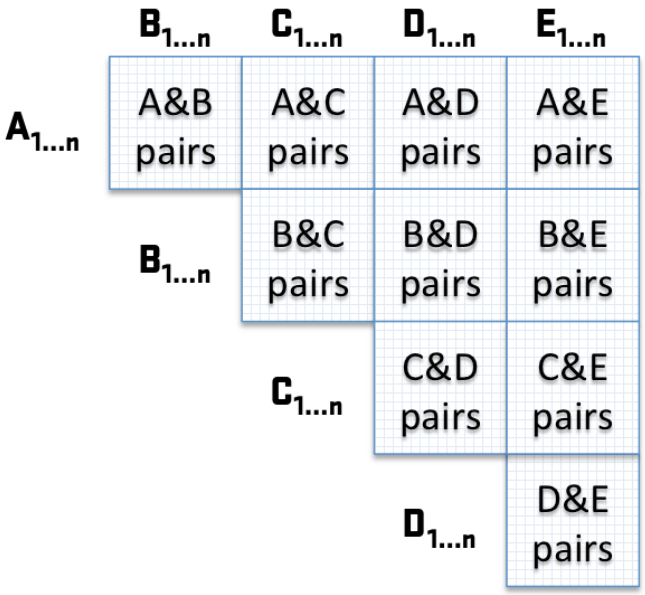
1.5.7. Esami sprendimai, kaip generuoti *Gherkin* kodą

1.5.7.1. Ghekin testų generavimas iš Hexawise matricos

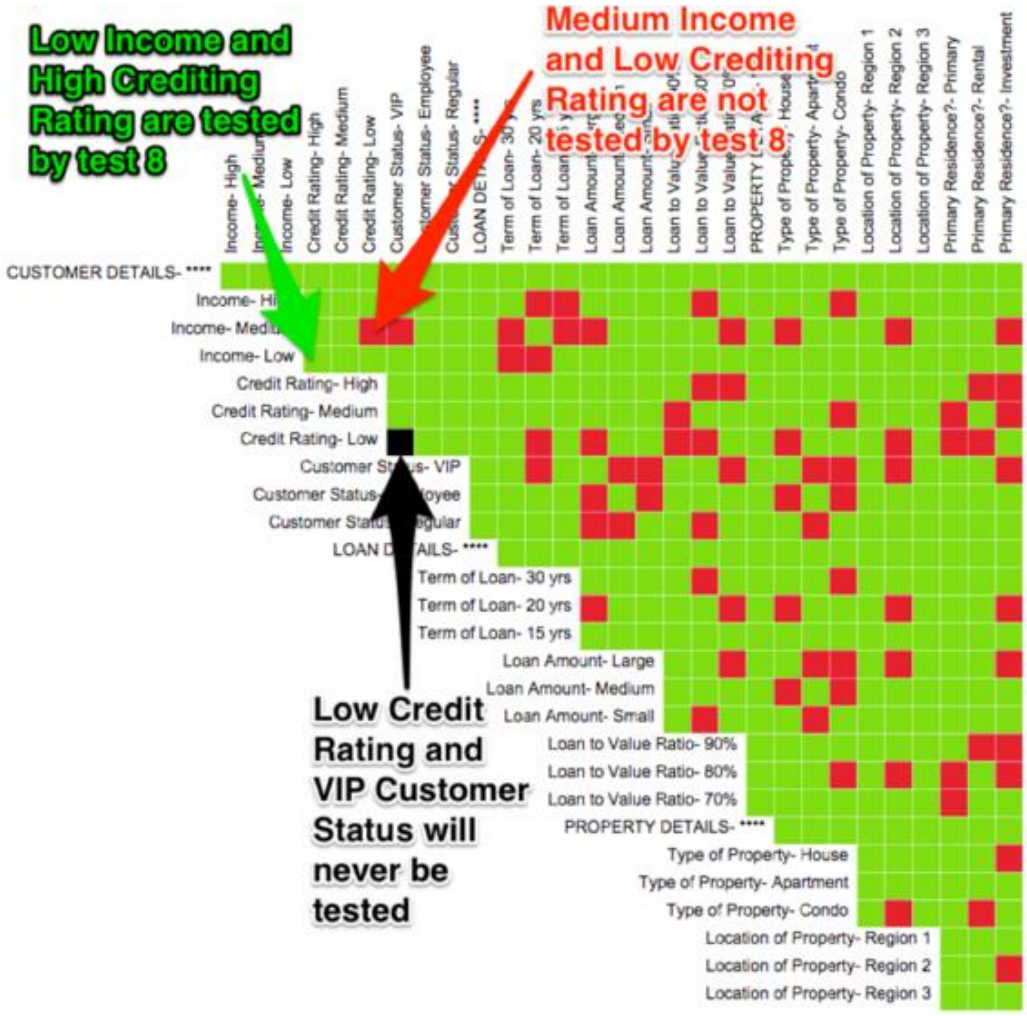
Yra išrastas būdas, kaip generuoti *Gherkin* kodą iš matricos. Ši matrica pavadinta *Hexavise* matrica. Kaip teigiama puslapyje [23], pristatančiame šį sprendimą, testuotojų dažniausiai užduodami klausimai apie programos padengimą testais yra šie:

1. Ar mes testuojame pakankamai?
2. Ar mes testuojame per daug?
3. Kokią dalį sistemos padengia sukurit testai?
4. Kas jei pritrūks laiko? Kokį padengimo lygį galima pasiekti, atliekant pusę numatytų testų?

Hexawise matrica leidžia vizualiai pamatyti padengimą testais. *Hexawise* matricos struktūra: parametrai išdėstyti nuo viršaus žemyn, pradedant pirmu ir baigiant paskutiniu parametru. Viršuje parametrai išdėstyti pradedant antru ir baigiant paskutiniu parametru. Pavyzdžiui, jei plano parametrai būtų „A“, „B“, „C“, „D“ ir „E“, padengimo matrica atrodytų taip:



9 pav. Padengimo matricos pavyzdys [23]



10 pav. *Hexawise* matricos pavyzdys [23]

Hexawise matrica gali būti naudojama exportuojant reikšmes į *Gherkin* kalbą (*Given, When → Then*) [23]. Pateikta idėja: *Hexawise* matricoje pavaizduotos testinių duomenų / parametrų poros yra automatiškai sugeneruojamos į *Gherkin* kodą. Problema, kad nėra apibrėžiamas „Then“ laukas. Jam siūloma uždėti numatytą reikšmę: „elgiasi taip, kaip tikimasi“ (angl. *behaves as expected*), o norint konkrečių rezultatų, juos reikėtų aprašyti patiems. Generuoti kodą nėra taip paprasta, jei reikia naudoti daugiau nei du parametrus. Šiuo atveju tektų kurti atskirą reikalavimą.

Šio sprendimo pliusai ir minusai.

Lentelė 2. *Hexawise* matricos pliusai ir minusai

pliusai	minusai
1. Automatiškai sugeneruojamas Gherkin kodas, todėl pasikeitus testuojamoms reikšmėms, reikėtų tik pergeneruoti kodą, o ne perrašyti. 2. Padengimas matomas matricoje.	1. Neaprašomas laukiamas rezultatas, todėl reikia jį aprašyti ranka. 2. Pergeneravus testus (tuo atveju jei pasikeitė duomenys), prarandami ranka aprašyti laukiami rezultatai. 3. Kiek neaišku ar čia nesusigadina „given“ ir „when“ prasmė, nes vienam teste ta pati reikšmė būtų prie „given“, kitam

pliusai	minusai
	prie „when“ (pagal pirmą paaiškinamąjį pav.) 4. Nepatogu, jei reikia testo su daugiau nei dviem parametrais.

1.5.7.2. Gherkin testų generavimas iš programinio kodo

Yra išrastas būdas kaip generuoti *Gherkin* kodą ir iš programinio kodo [24]. Tam yra naudojami kontraktai. Iš pradžių aprašomi kontraktai ir jų galimos reikšmės lentelės pavidalu. Vėliau parašomas kodas, naudojantis anksčiau parašytus kontraktus.

Kontraktų pavyzdys [24]:

warehouse	product.measurable	result
WH1	false	false
WH1	true	false
WH2	false	false
WH2	true	true

```
"Type::warehouse": {
 "constraint": create_enum('WH1', 'WH2'),
 "cucumber": {
   "type": "Given",
   "match"   : qr/the warehouse is "(WH\d)"/,
   "describe": "the warehouse is \"{0}\"",
 }
},

 "Type::product_measurable": {
 "constraint": "Type::Bool",
 "cucumber": {
     "type": "When",
   "match": [
     qr/the product (is (not )?) measurable/,
     function (m) { return m == 'is ' }
   ],
   "describe": function (m) {
     return "the product is " . ( m ? '' : 'not ' ) . "measurable";
   },
 }
}
```

11 pav. Programinio kodo pavyzdys [24]

Paprastai programinis kodas yra rašomas iš *Gherkin* testų, kurie yra parašyti pagal reikalavimus. Šiame skyriuje aprašomas sprendimas padeda sugeneruoti *Gherkin* testus iš programinio kodo, visiškai nesigilinant į sistemos reikalavimus.

1.5.8. Sprendimas, kaip generuoti veiklos diagramą iš *Gherkin* kodo

Nors kol kas nėra sukurto būdo, kaip generuoti *Gherkin* kodą iš UML veiklos diagramų, yra pasiūlyta idėja kaip atlikti atvirkštinį veiksmą – generuoti UML veiklos diagramas iš *Gherkin* kodo. Šią idėją pasiūlė Taras Kalapunas – IT konsultantas ir projektų vadovas. Informaciją apie šį žmogų ir minėtą idėją galima rasti jo internetiniame puslapyje [25] .Šiame puslapyje pateikiami pavyzdžiai kaip iš skirtingų *Gherkin* kodų sugeneruotos veiklos diagramos. Nors pavyzdžiuose pateiktas originalus *Gherkin* kodas iš kurio buvo generuotos pateiktos veiklos diagramos, pats algoritmas, kaip tai buvo padaryta, neatskleistas.

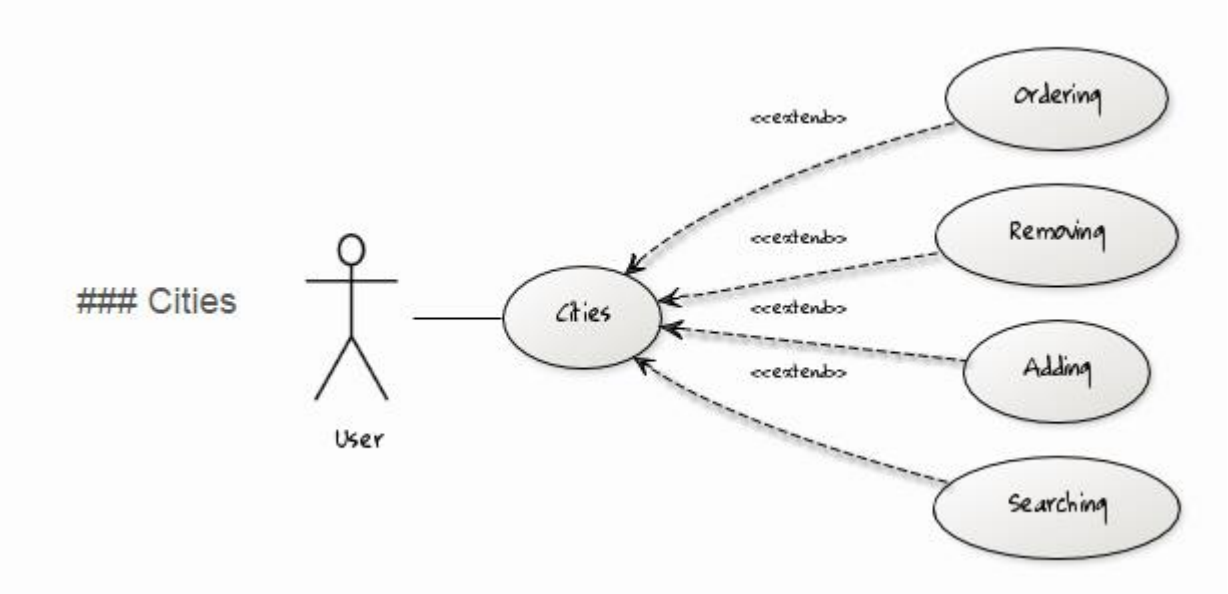
Minėtame puslapyje [25] pateikti pavyzdžiai dviems funkcionalumams (angl. *features*). Pirmasis, pavadinimu „Miestai“ (ang. *cities*) turi keturias funkcijas: užsakyti (angl. *ordering*), pašalinti (angl. *removing*), pridėti (angl. *adding*) ir ieškoti (angl. *searching*). Antrasis pavyzdys labai panašus į pirmąjį, todėl čia nebus pateiktas.

Toliau pateikiamas originalus *Gherkin* kodas anglų kalba, iš kurio buvo generuotos diagramos [25]:

```
Feature: Cities
  In order view weather for different cities,
  As a User,
  I want to have ability to manage cities.

 Scenario: Searching
   Given I am on 'Kiev' city screen
    When I enter cities setting screen
     And I touch Add button
     And I enter 'Amsterdam' in search field
    Then I should see 'Amsterdam' table row

 Scenario: Adding
   Given I am on 'Kiev' city screen
    When I enter cities setting screen
     And I add 'Amsterdam' city
     And I save settings
```

```
    Then I should see 'Amsterdam' city screen

Scenario: Removing
   Given I am on 'Kiev' city screen
    When I enter cities setting screen
    Then I should see 'Kiev' table row
    When I touch Delete on 'Kiev' table row
    Then I should not see 'Kiev' table row
    When I save settings
    Then I should not see 'Kiev' city screen
     And there should not be 'Kiev' city screen

 Scenario: Ordering
   Given there are the existing cities 'Kiev', 'London' and 'Amsterdam'
     And I am on first city screen
    Then I should see 'Kiev' city screen
    When I enter cities setting screen
     And I order cities to 'London', 'Kiev' and 'Amsterdam'
     And I save settings
    Then I should see 'London' city screen
```

Pagal šį tekstą sugeneruotos keturios veiklos diagramos ir viena panaudos atvejų diagrama.



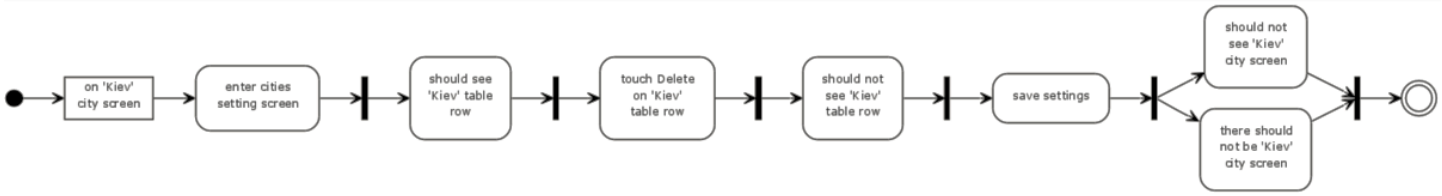
12 pav. Iš *Gherkin* kodo sugeneruota panaudos atvejų diagrama [25]

Kaip matome, panaudos atvejų diagramoje pavaizduotas tik vienas aktorius – vartotojas (angl. *user*), viena jo veikla – miestai (angl. *cities*), kuri buvo bendras testų rinkinio pavadinimas (angl.

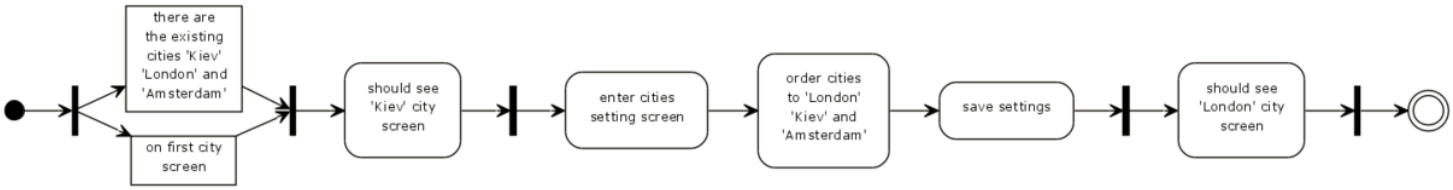
features) ir keturi veiklos „miestai“ plėtiniai. Kaip matome, šie plėtinių pavadinimai sutampa su keturių *Gherkin* testų pavadinimais.

13 pav. Veiklos diagrama „Searching“ [25]

14 pav. Veiklos diagrama „Adding“ [25]



15 pav. Veiklos diagrama „Removing“ [25]



16 pav. Veiklos diagrama „Ordering“ [25]

Pagal turimą *Gherkin* kodą ir sugeneruotas diagramas galima atsekti kelias tendencijas:

1. Visi „Given“ sakiniai pavaizduojami pačiame pirmame veiklos diagramos žingsnyje.
2. Jei yra daugiau nei vienas „Given“ sakinys, jie visi vaizduojami kaip lygiagretūs veiksmai pirmame žingsnyje.
3. Visi „Given“ sakiniai vaizduojami kaip objekto būsenos.
4. „When“ ir „Then“ sakiniai vaizduojami kaip veiklos simboliai.
5. Visi „Then“ sakiniai vaizduojami kaip paskutinis žingsnis.
6. Jei yra daugiau nei vienas „Then“ sakinys, jie visi vaizduojami kaip lygiagretūs veiksmai paskutiniame žingsnyje.
7. Jei *Gherkin* kode yra vienas „When“ arba keli „When“ atskirti žodžiu „And“, jie visi vaizduojami kaip veiksmai einantys vienas po kito. Šiuos veiksmus jungia paprastos rodyklės.

8. Jei *Gherkin* kode „When“ ir „Then“ kaitaliojasi vienas po kito arba jei „Then“ įsiterpia tarp „Given“ ir „When“, tuomet veiklos diagramoje šie veiksmai eina vienas po kito, tačiau jie sujungiami ne paprasta rodykle, bet ir sujungimo / išsišakojimo ženklu.

1.5.9. Testų automatizavimo įrankiai ir kalbos

Šiame skyriuje aprašyta analizės, apie testų rašymo įrankius, rezultatai.

„Cucumber“ [26] – tai atviro kodo įrankis, kuriame galima kurti testus. Testams aprašyti „Cucumber“ naudoja *Gherkin* kalbą, o vėliau šios kalbos sakiniai arba žingsniai aprašomi „JavaScript“ aplinkoje. Plačiau apie *Gherkin* kalbą rašoma sk. 1.5.3 „*Gherkin* kalba ir jos sintaksė“.

„SpecFlow“ [27] – tai atviro kodo įrankis leidžiantis ne techniniams vartotojams paruošti ir automatizuoti testus. Šis įrankis veikia naudojant „Visual Studio“. Rašant testus su „SpecFlow“ naudojama specifine kalba pavadinta *Gherkin*, kurios dėka aprašomi su „SpecFlow“ kuriami testai. Šis įrankis nuo „Cucumber“ skiriasi tuo, kad leidžia rašyti testus „.Net“ aplinkoje.

Apibendrinant šiuos įrankius galima pabrėžti jų privalumus:

- Lengvai struktūrizuojami.
- Aiškiai suprantami net ir daug nemokant programuoti.
- Aprašomi natūralia kalba.
- Gali būti lengvai siejami su reikalavimais (jei reikalavimai buvo aprašyti tinkamai).

Šio sprendimo trūkumai:

- Testo kodas vis tiek turi būti parašytas ranka (nėra auto generavimo)
- Pakeitus *Gherkin* kodą, reikės taip pat pakeisti ir jam parašytą testo kodą.
- Šiuose testuose naudojami duomenys nėra dinaminiai, jie atliekami visada su tokiomis pačiomis reikšmėmis (angl. *mocking*).

„Microsoft Test Manager“ [28]

Kaip rašoma „Microsoft“ puslapyje su šia programa galima:

- Atliekant testavimą įrašyti savo veiksmus.
- Planuoti testus, su galimybe įtraukti jau įrašytus testus.
- Kopijuoti testavimo rinkinius iš vieno projekto į kitą.

„CASE Spec“ [29]

Ši programa labiau pritaikyta testuoti reikalavimams. Pagrindiniai jos privalumai:

- Leidžia skirtingiems naudotojams kartu dirbti testuojant reikalavimus.
- Suteikia automatinę versijų kontrolę.
- Leidžia žymėti grafines tėvo – vaiko sąsajas.
- Leidžia susieti reikalavimus su kitais artefaktais.

„Rational DOORS“ [30]

Tai įmonės IBM produktas. Ši programa skirta optimizuoti reikalavimų valdymą. Ji leidžia surinkti, atsekti, analizuoti ir valdyti reikalavimus bei jų pokyčius, išlaikant atitinkamus reglamentus ir standartus. Ši programa suteikia galimybę:

- Valdyti reikalavimus centralizuotoje vietoje, kurioje jie būtų pasiekiami visai komandai.
- Susieti reikalavimus su testavimo planais, testavimo atvejais bei kitais reikalavimais.
- Nustatyti keičiamo reikalavimo mąstą.
- Naudoti testų atsekamumo įrankį skirtą susieti rankinių testų aplinkas su reikalavimais ir testavimo atvejais.
- Naudoti integracijas, kurios padeda valdyti reikalavimų pokyčius, nepriklausomai ar jie sudėtingi, ar didesni.

„Seapine“ atsekamumo įrankiai [31]

Šie įrankiai sukurti taip, kad leistų kokybiškai susieti informaciją. Keli įmonės siūlomi įrankiai, kurie galėtų būti panaudoti šiame darbe:

- Testų sąsajos įrankis (angl. *TestTrack RM*)

Tai įrankis skirtas susieti visą su testavimu susijusią informaciją: testavimo atvejus, testavimo rezultatus ir testuojamą kodą. Nustatant sąsajas, įrankis leidžia pasirinkti įvairias taisykles, pagal kurias bus susieti objektai. Taip pat įrankis leidžia tam tikrą informaciją pavaizduoti grafiškai.

- Automatinio funkcinio testavimo įrankis (angl. *QA Wizard Pro*)

Šis įrankis skirtas funkciniam ir regresiniam internetinio tinklo, „Windows“, „Java“ aplikacijų, bei interneto apkrovos testavimui. Įrankio pranašumai:

- juo galima paleisti tiek funkcinius, tiek apkrovos testus;
- leidžia apriboti tinklo, atminties ar disko prieigą;
- galima naudoti su skirtingomis naršyklėmis: IE, „Firefox“, „Chrome“;
- palaiko „Adobe Flash“;
- galima testuoti aplikacijas, parašytas įvairiomis kalbomis: „Java“, „Flash“, „HTML 5“, „TinyMCE“, „JavaScript“, „Silverlight“, „C#“, „VB.NET“, „C++“, „Win32“, „Qt“, „AJAX“, „ActiveX“, „DevExpress controls“, „Delphi controls“, „Janus Systems controls“, „Oracle Forms“, „Infragistics Windows Forms controls“ ir t.t.

- Versijų kontrolės įrankis (angl. *Surround SCM*)

Tai įrankis padedantis kontroliuoti kuriamo produkto versijas. Įrankis leidžia sukurti kelis filialus (angl. *branch*), atsekti senesnių versijų istoriją, sujungti kelių filialų turinį (angl. *merge*).

„Rational RequisitePro“

Tai įmonės IBM produktas. Ši programa yra reikalavimų ir panaudojimo atvejų valdymo įrankis, skirtas projektų komandoms, kurios nori patobulinti projekto tikslų komunikaciją, sustiprinti bendradarbiavimo plėtrą, sumažinti projekto riziką ir padidinti programų kokybę prieš dislokavimą.

Ši programa naudinga tuo, kad paprastus reikalavimus, surašytus į „MS Word“ programą, leidžia susieti tarpusavyje, nustatyti įvairius artefaktus ir požymius (pvz. svarbumas, aktualumas). Taip pat ši programa leidžia pasirinkti pagal kokį šabloną surašyti reikalavimai.

„MS visual studio“ [32]

„MS Visual Studio“ siūlo net kelis įrankius, skirtus testavimui. Keli iš jų: „MS Test“, „xUnit.net“, „NUnit“. Su šiais įrankiais galima:

- Automatiškai sugeneruoti testus.
- Importuoti testavimo atvejus iš kitų „Visual“ produktų ir taip sukurti testavimo atvejus, kurie bus susieti su reikalavimais.
- Įvykdyti automatinius testus ir gautus rezultatus fiksuoti „Microsoft Test Manager“ įrankyje.
- Naudoti internetinę vartotojo sąsają, kuriant naujus testavimo planus.

1.6. Siekiamo sprendimo apibrėžimas

Siekiama sukurti metodą, kuris automatiškai generuotų *Gherkin* kodą iš pateiktų reikalavimų. Kadangi reikalavimai gali būti aprašyti įvairiais būdais, bus analizuojamos tik šios reikalavimų formos:

- **Tekstinis formatas**. Kadangi reikalavimai turi turėti bent šiek tiek struktūros, kad iš jų būtų galima generuoti *Gherkin* testus, pasirinkta naudoti tekstinę reikalavimų formą – darbo istorijas.
- **Sekų diagramos**. Šiomis diagramomis galima aiškiai aprašyti vartotojo veiksmus su sistema, todėl jose pateikiama informacija gali būti aprašyta *Gherkin* testais.
- **Veiklos diagramos**. Kaip ir sekų diagramų atveju, veiklos diagramomis galima aiškiai aprašyti sistemos veikimo žingsnius ir scenarijus, todėl ši informacija gali būti aprašyta ir *Gherkin* testuose.
- **Lentelės.** Šis būdas pasirinktas remiantis analizės dalyje aprašyta informacija apie *Hexawise* matricą. Sprendimu norima patobulinti *Hexawise* matricos sprendimą ir pasiūlyti metodą, kurio dėka būtų galima aprašyti ir laukiamą rezultatą, ko nebuvo galima padaryti su *Hexawise* matrica.

Siūlomas sprendimas padėtų informacines sistemas kuriančioms komandoms automatiškai sukurti *Gherkin* testus. Sukurti testai perteiktų reikalavimus į realius scenarijus ir padėtų aiškiau įsivaizduoti kaip veiks sukurta sistema. Taip pat iš sugeneruoto *Gherkin* kodo bus galima rašyti automatinius testus.

1.7. Analizės išvados

1. *Gherkin* kalbos analizė parodė, kad *Gherkin* kalba yra suprantama tiek programuotojams, tiek analitikams ar testuotojams. Ši kalba pritaikyta projektams, kurie dirba veikla pagrįstu programavimu (angl. *Behaviour – Driven Development*), kuris naudojamas *Agile* metodikose.
2. *Agile* analizė parodė, kad *Gherkin* testai gali būti labai naudingi tiek pradinėje kūrimo stadijoje, kai reikalavimai yra kuriami kartu su klientu, tiek sukūrus sistemą ir tikrinant ar visos jos vietos veikia taip, kaip tikimasi.
3. Reikalavimų struktūros analizė parodė, kad *Gherkin* kodo generavimui tinkamos šios reikalavimų struktūros: darbo istorija, sekų diagrama, veiklos diagrama, lentelė.
4. Reikalavimų ir testų sąsajos analizė parodė, kad labiausiai paplitęs sąsajos būdas yra matrica. Joje aiškiai išdėstomi duomenys ir vizualiai patogu matyti padengimus.
5. Testų sudarymo būdų, iš apibrėžtų sistemos reikalavimų, analizė parodė, kad iki šiol yra realizuoti tik du būdai *Gherkin* kodui generuoti, tai – *Hexawise* matrica ir generavimas iš programinio kodo. Taip pat buvo sukurta idėja, kaip generuoti veiklos diagramas iš *Gherkin* kodo. Iš šių sprendimų, vienintelis *Hexawise* matricos sprendimas generuoja *Gherkin* kodą iš reikalavimų.
6. Įvertinus analizės rezultatus ir sprendžiamą problemą, buvo nuspręsta sukurti kelis algoritmus, kurie generuotų *Gherkin* kodą iš šių reikalavimų struktūrų: darbo istorijų, sekų diagramų, veiklos diagramų, lentelių.

2. SPRENDIMO, KAIP GENERUOTI *GHERKIN* KALBĄ IŠ REIKALAVIMŲ, REIKALAVIMŲ SPECIFIKACIJA IR PROJEKTAS, FORMALUS APRAŠAS

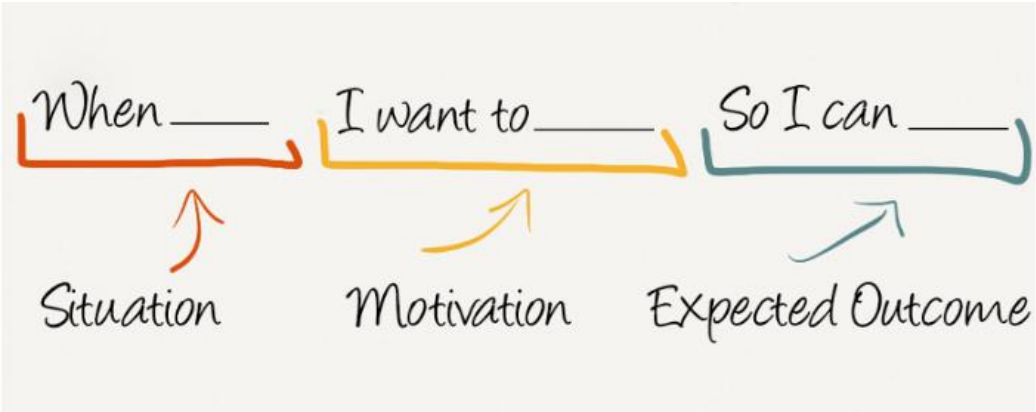
2.1. Reikalavimų generavimo į *Gherkin* kalbą sprendimo aprašymas

Šiame darbe siekiama atrasti geriausius būdus, kaip iš reikalavimų gauti *Gherkin* kodą. Kadangi reikalavimai gali būti pateikiami įvairias būdais, tai atitinkamai čia pateiksime skirtingus būdus, kaip galima gauti *Gherkin* kodą iš įvairių tipų reikalavimų. Šiame darbe apžvelgsime 3 pagrindinius reikalavimų aprašymo būdus:

1. **Tekstinis formatas.** Tai tekstu pateikiami reikalavimai. Šis tekstas gali būti patalpinamas, „MS Word“ programoje, „Jira“ puslapyje, ar net „SysML“ diagramoje. Atsižvelgiame, kad nepriklausomai, kur tokie reikalavimai bus įdėti, jie turės du pagrindinius elementus – identifikacinį numerį ir tekstu aprašytą reikalavimą.
2. **Diagramos.** Reikalavimai gali būti pateikiami ir diagramų pavidalu. Tam gali būti naudojamos šios diagramos: sekų, veiklos, būsenų, panaudojimo atvejų. Šiame darbe pasirinkta kurti algoritmus *Gherkin* kodo generavimui iš **veiklos** ir **sekų** diagramų, nes jos labiausiai atspindi testuojamos veiklos scenarijų, o *Gherkin* testai yra skirti aprašyti testų scenarijams.
3. **Lentelės formatas.** Šiuo formatu dažniausiai būna pateikiami tekstu aprašyti reikalavimai, pvz., „SysML“ reikalavimai, arba jau apibendrinti reikalavimai, iš kurių bus ruošiami testavimo atvejai.

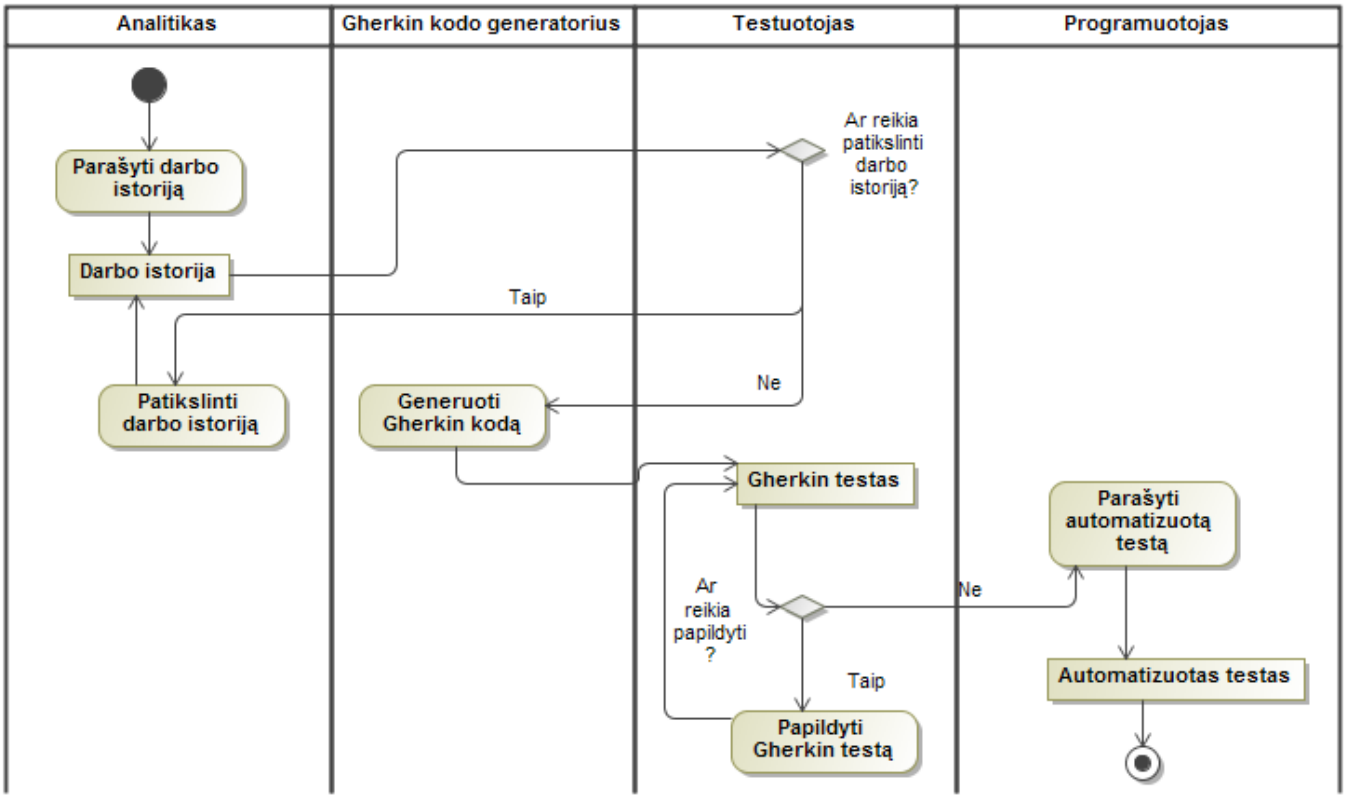
2.1.1. *Gherkin* testo generavimas iš teksto

Aprašant reikalavimus tekstu, vienas patogiausių būdų naudoti darbo istorijas (angl. *job story*).



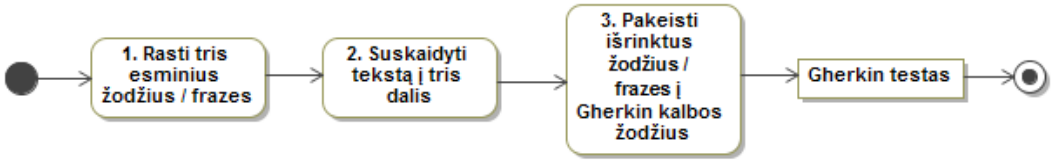
17 pav. Darbo istorijos struktūra (anglų kalba) [33]

Žemiau pateikta veiklos diagrama, kurioje parodyta kaip atrodytų procesas nuo darbo istorijos kūrimo iki automatinių testų kūrimo (žr. 18 pav.).



18 pav. Procesas nuo darbo istorijos iki automatinio testo kūrimo

Žemiau pateikiama veiklos diagrama, kurioje pavaizduoti veiksmai, kuriuos atlieka *Gherkin* kodo algoritmas:



19 pav. *Gherkin* kodo generavimo iš darbo istorijos algoritmas

Reikalavimų formalus aprašas:

1. Pirmame žingsnyje paimamas visas darbo istorijos tekstas ir ieškoma esminių žodžių, kurie pateikti transformacijų lentelėje, pirmame arba antrame stulpeliuose, priklausomai nuo to, kokia kalba parašyta darbo istorija. (žr. Lentelė 3)
2. Antrame žingsnyje turimas darbo istorijos tekstas suskaidomas į tris dalis, pagal rastus esminius žodžius. Tekstas keliamas į naują eilutę, pradedant esminiu žodžiu (žr. pavyzdžius žemiau).
3. Trečiame žingsnyje vykdoma transformacija – rasti esminiai žodžiai pakeičiami į *Gherkin* kalbos žodžius, pateiktus transformacijos lentelėje, trečiame ir ketvirtame stulpeliuose.

4. Gaunamas *Gherkin* testas, kuris gali būti patobulintas rankiniu būdu. To gali prireikti norint pataisyti lietuviškus linksnius ar atlikti kitus nežymius pataisymus.

Reikalavimai, norint taikyti šį algoritmą:

1. Darbo istorija turi būti išsami.
2. Darbo istorijoje turi būti naudojami transformacijų lentelėje įrašyti raktiniai žodžiai. Reikalui esant galima papildyti transformacijų lentelę dažnai naudojamais raktiniais žodžiais.

Lentelė 3. „Darbo istorijos“ žodžių transformavimas į *Gherkin* kalbos žodžius

Angliškas tekstas	Lietuviškas tekstas	Keisti į EN	Keisti į LT
When Given	Kai Kuomet Duota	Given	Duota
I want to When	Noriu Kuomet Noriu turėti galimybę	When	Kai
So that So I can So I could Then	Tuomet Kad galėčiau	Then	Tuomet
And ,	Ir ,	And	Ir

Algoritmo taikymo pavyzdys anglų kalba

Tarkime turime tokią darbo istoriją:

When I’m ready to have estimators bid on my game, I want to create a game in a format estimators can understand, so that the estimators can find my game and know what they are about to bid on.

Remiantis algoritmu, pažingsniui gaunamas *Gherkin* testas:

1. Surandami trys mums svarbūs žodžiai / frazės:

   When I’m ready to have estimators bid on my game, I want to create a game in a format estimators can understand, so that the estimators can find my game and know what they are about to bid on.

2. Tekstas suskaidomas į tris dalis:

   When I’m ready to have estimators bid on my game,

   I want to create a game in a format estimators can understand,

   so that the estimators can find my game and know what they are about to bid on.

3. Pažymėti žodžiai paverčiami į *Gherkin* kodą:

   Given I’m ready to have estimators bid on my game,

When create a game in a format estimators can understand,

Then the estimators can find my game and know what they are about to bid on.

4. Gautas testas dar nėra visiškai „tobulas”, tačiau kodą reikėtų pakoreguoti labai nedaug:

Given user is ready to have estimators bid on my game,

When create a game in a format ~~estimators can understand(~~ particular format)**,**

Then the estimators can find my game in (particular place) and ~~know what they are about to bid on~~ message appears “Now you are able to bid on”.

Iš pateikto pavyzdžio galima pamatyti, kad:

1. Tam tikru būdu aprašius darbo istoriją, nieko nereikėtų keisti rankomis po generavimo.
2. Vietoj „So I can“ gali būti „So that“ ar kiti išsireiškimai, todėl galima būtų sudaryti tam tikrą žodyną, ką į ką galima konvertuoti.
3. Šis sprendimas gali palaikyti ir versijavimą, kas būtų labai naudinga besikeičiant reikalavimams.

Algoritmo taikymo pavyzdys lietuvių kalba

Darbo istorija:

Kuomet turiu sukurtą paskyrą sistemoje, noriu turėti galimybę prisijungti prie sistemos tam, kad galėčiau ja naudotis.

Ši darbo istorija yra gana abstrakti, nes neaišku ką konkrečiai turi atlikti vartotojas, norėdamas prisijungti. Iš jo galima sugeneruoti *Gherkin* kodą, tačiau reikėtų koreguoti testą ir pridėti konkrečius veiksmus. Tai pastebėjus iš anksto, galima prašyti patikslinti reikalavimus, įtraukiant konkrečius veiksmus:

Kuomet turiu sukurtą paskyrą sistemoje, noriu turėti galimybę įvesti vartotojo vardą, slaptažodį ir paspausti prisijungimo mygtuką, kad galėčiau prisijungti ir naudotis sistema.

Šiai darbo istorijai būtų galima sugeneruoti *Gherkin* testą, iš kurio būtų galima sukurti automatizuotą testą:

1. Kuomet turiu sukurtą paskyrą sistemoje, noriu turėti galimybę įvesti vartotojo vardą, slaptažodį ir paspausti prisijungimo mygtuką, kad galėčiau prisijungti ir naudotis sistema.
2. Kuomet turiu sukurtą paskyrą sistemoje,
noriu turėti galimybę įvesti vartotojo vardą
, slaptažodį
ir paspausti prisijungimo mygtuką,
kad galėčiau prisijungti ir naudotis sistema.

3. Angliškai:
   Given turiu sukurtą paskyrą sistemoje,
   When įvesti vartotojo vardą
   And slaptažodį
   And paspausti prisijungimo mygtuką,
   Then prisijungti ir naudotis sistema.

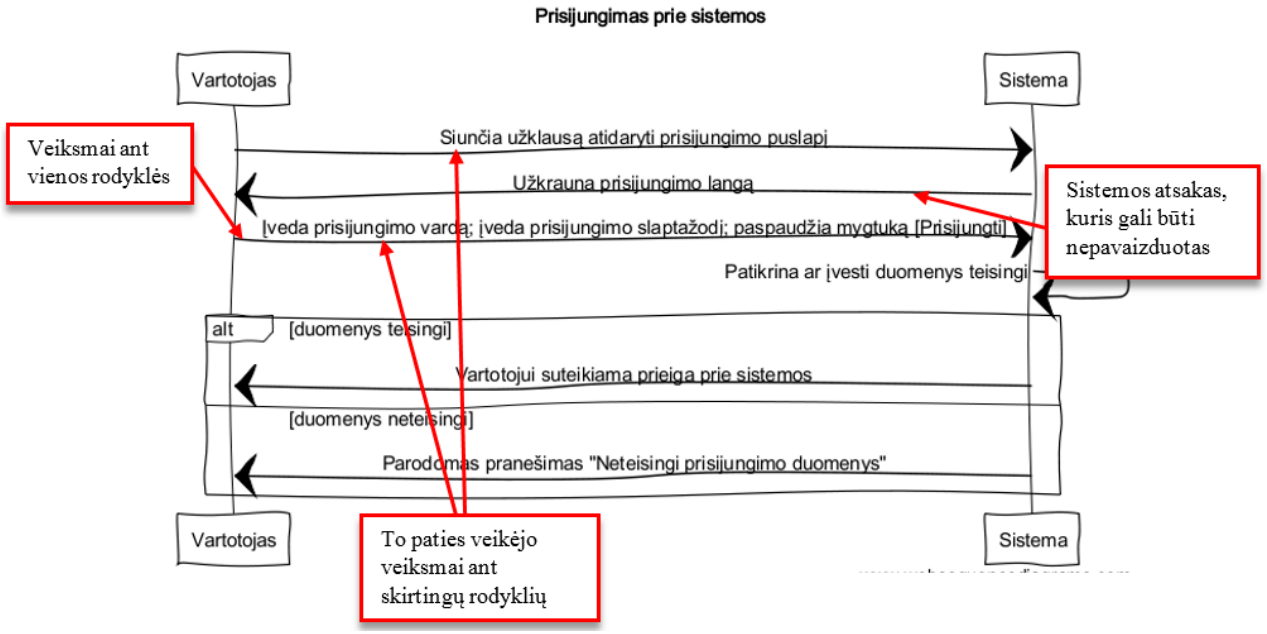
4. Lietuviškai:
   Duota turiu sukurtą paskyrą sistemoje,
   Kai įvesti vartotojo vardą
   Ir slaptažodį
   Ir paspausti prisijungimo mygtuką,
   Tuomet prisijungti ir naudotis sistema.

5. Gautą tekstą galima pakoreguoti:
   Duota turiu sukurtą paskyrą sistemoje,
   Kai įvesti vartotojo vardą
   Ir įvesti vartotojo slaptažodį
   Ir paspausti prisijungimo mygtuką,
   Tuomet prisijungti ir turėti galimybę naudotis sistema.

2.1.2. *Gherkin* testo generavimas iš sekų diagramos

Remiantis analitine dalimi, apsiribosime tik aukšto lygio sekų diagramomis, kurios turi tik du bendraujančius objektus – vartotoją ir sistemą. Iš tokių diagramų paprasčiau kurti vartotojo sąsajos testus, nes bus automatizuojami tik vartotojo veiksmai ir vartotojo laukiami rezultatai, atlikus veiksmus.

Tačiau net ir tokios sekų diagramos gali turėti įvairų išsamumą, bei veiksmų pateikimą. Pavyzdžiui, vienoje diagramoje to paties veikėjo atskiri veiksmai gali būti pateikiami ant vienos rodyklės, o kitoje ant atskirų. Taip pat gali būti, kad vienur bus pavaizduoti sistemos pateikiami duomenys vartotojui, o kitur ši informacija bus praleista (žr. 20 pav.).



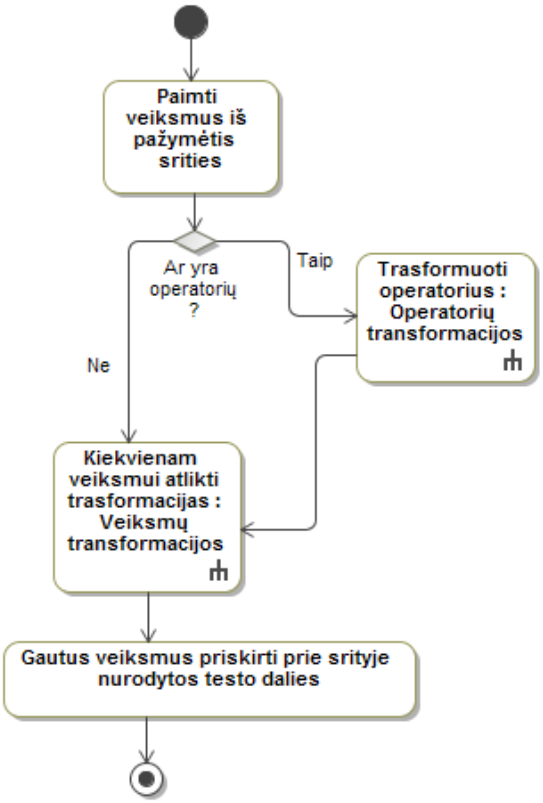
20 pav. Sekų diagramos vaizdavimas

Tai, kad kiekviena sekų diagrama gali būti pavaizduota skirtingais būdais, gali labai apsunkinti *Gherkin* kodo generavimą, nes priešingai nei tekstinio darbo istorijos atveju, algoritmui gali būti sunku atpažinti kur baigiasi viena dalis (*given* arba *when*) ir prasideda kita (*when* arba *then*). Kad algoritmas būtų tikslesnis įvedamas reikalavimas, kad prieš generavimą vartotojas pažymėtų atitinkamas sritis:

- Pirma sritis - duomenų pateikimas,
- Antra sritis - atliekami veiksmai,
- Trečia sritis - laukiamas rezultatas.

Kiekvienoje iš pasirinktų sričių gali būti po vieną ar kelis operatorius (žr. sk. 1.5.6.1 „Reikalavimų aprašymas sekų diagramomis“). Operatorius gali būti interpretuojamas kaip vienas sudėtinis veiksmas, todėl įtraukiant operatorių į žymimą sritį reikia pažymėti visą operatorių, o ne tik jo dalį. Priešingu atveju galima prarasti operatoriaus prasmę. Reikalavimams skirtose sekų diagramose dažniausiai gali pasitaikyti šie operatoriai: *alt, opt, par*. Kitų operatorių neanalizuosime, nes jie labiau techniniai ir yra mažai tikėtina, kad bus naudojami aprašant paprastus veiksmus tarp sistemos ir vartotojo.

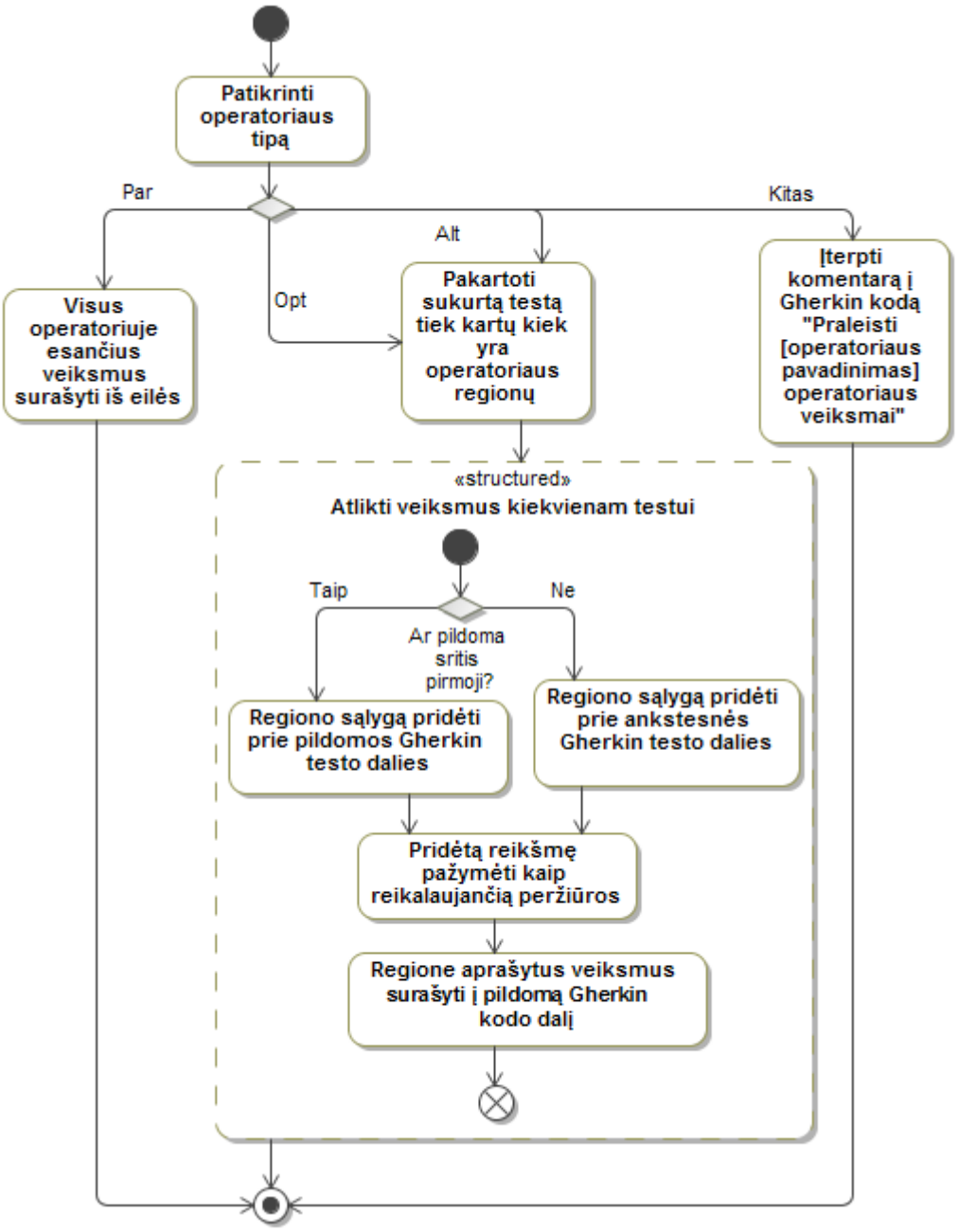
Atsižvelgiant į šią informaciją, sukurtas algoritmas, kuris automatizuotų testo, parašyto *Gherkin* kalba, generavimą iš sekų diagramos:



21 pav. *Gherkin* kodo generavimo iš sekų diagramos algoritmas

Čia pavaizduotas algoritmas turi būti atliekamas kiekvienai vartotojo pažymėtai sričiai atskirai, t.y. paimami duomenys iš pirmosios pažymėtos srities, atliekami veiksmai su ja, tuomet viskas pakartojama su antrąja ir trečiąja sritimis.

Šis algoritmas turi dvi papildomas veiklos diagramas – tai „Operatorių transformacijos“ (žr. 22 pav.) ir „Veiksmų transformacijos“ (žr. 23 pav.).



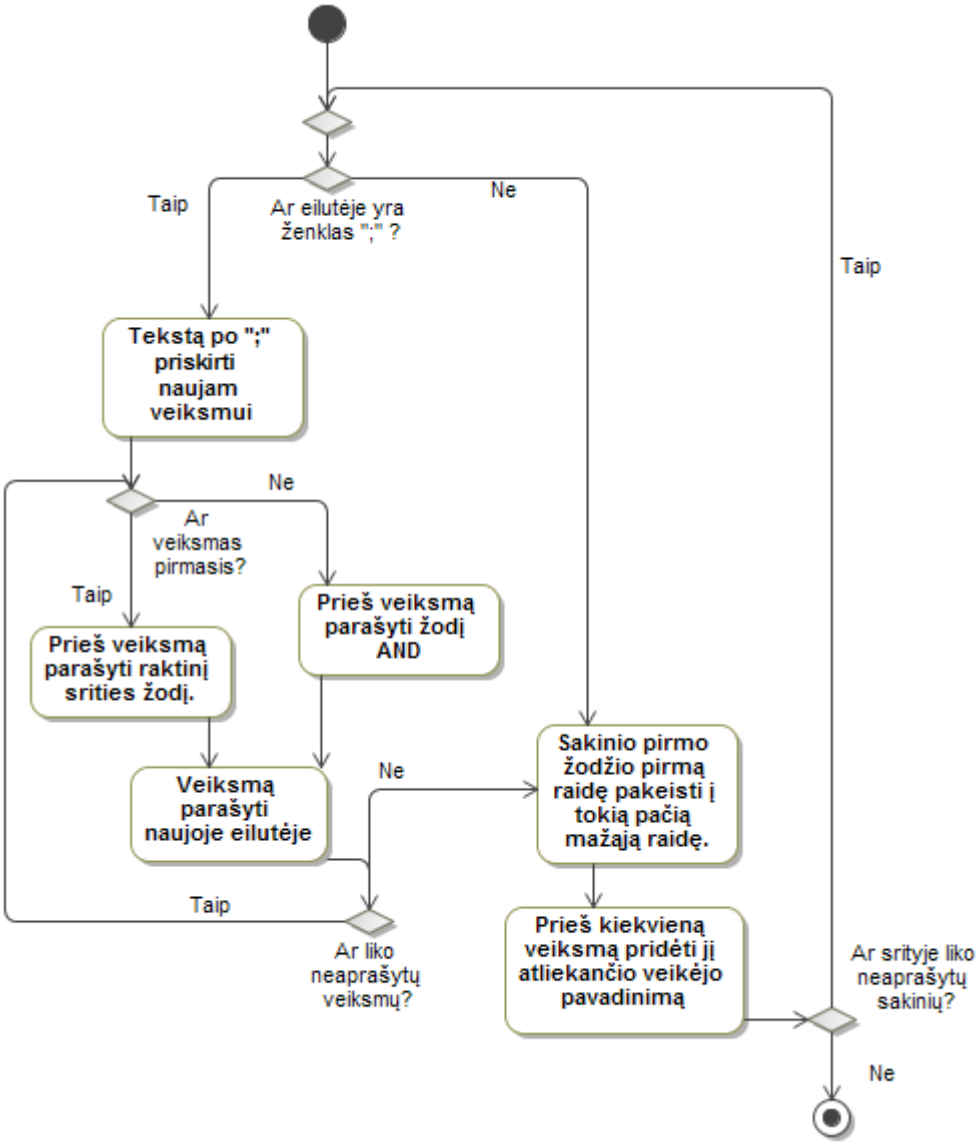
22 pav. Veiklos diagrama „Operatorių transformacijos“

Šioje diagramoje pavaizduoti algoritmo veiksmai atliekami kiekvienam sekų diagramos operatoriui. Pirmiausia patikrinama kokio tipo operatorius panaudotas. Jei operatorius nėra *par*, *opt* arba *alt*, tuomet į *Gherkin* kodą įterpiamas komentaras su tekstu „Praleisti [operatoriaus pavadinimas] operatoriaus veiksmai”. Kitiems operatoriams atliekamos tam tikros transformacijos:

- ***Par*** – lygiagretus vykdymas. Kadangi šis operatorius parodo, kad veiksmų seka nėra svarbi ir juos galima atlikti lygiagrečiai, tai reiškia, kad nesvarbu, kokia tvarka regionai bus vykdomi atliekant testą. Todėl algoritmas visus regionus ir veiksmus juose vykdo iš eilės.

- ***Opt*** – pasirenkamasis vykdymas. Kadangi naudojant šį operatorių regionuose esantys veiksmai atliekami tik tuomet, jei išpildoma regiono sąlyga, kuriant testus svarbu ištestuoti visus galimus atvejus, todėl kiekvienam regionui sukuriama po testą. Tai darant, nukopijuojama iki šio momento sugeneruota testo dalis ir pakartojama tiek kartų, kiek yra operatoriaus regionų. Tuomet kiekvieno regiono testui sugeneruojama:
    - Jie pildoma pirmoji testo dalis, regiono sąlyga pridedama prie pildomos *Gherkin* testo dalies, o jei antroji arba trečioji, regiono sąlyga pridedama prie ankstesnės testo dalies.
    - Pridėta dalis pažymima kaip reikalaujanti peržiūros, nes gali būti, kad pagal logiką jai priklauso būti ankstesnėje *Gherkin* kodo dalyje.
    - Regione aprašyti veiksmai surašomi į pildomą *Gherkin* kodo dalį.

  Nors šiame operatoriuje gali būti vykdoma ir daugiau nei po vieną regioną, testuojant svarbu patikrinti bent tas variacijas, kai išpildoma tik po vieną regionų sąlygą. Jei norima testuoti daugiau variacijų, pavyzdžiui, kai išpildomos dvi ar daugiau regionų sąlygų, galima rankiniu būdu sukurti norimos variacijos testą, tiesiog perkopijuojant *Gherkin* kodo dalį iš jau sugeneruotų testų.
- ***Alt*** – alternatyvus vykdymas. Šis operatorius labai panašus į pasirenkamojo vykdymo *Opt* operatorių. Vienintelis skirtumas tas, kad čia vykdomas tik vienas regionas iš visų, o *Opt* operatoriuje gali būti įvykdyta ir daugiau nei vienas veiksmas. Į tai atsižvelgiant, abu operatoriai vykdomi vienodai.



23 pav. Veiklos diagrama "Veiksmų trasformacijos"

Veiksmų transformacijos atliekamos visiems sekų diagramos veiksmams, pateikiamiems ant rodyklių. Veiksmai, kuriuos atlieka ši transformacija:

- Jei ant rodyklės parašytame tekste randamas ženklas „ ; “, jis interpretuojamas kaip veiksmų „skirtukas“, todėl *Gherkin* kode šio ženklo pusėse esantis tekstas suprantamas kaip skirtingi veiksmai.
- Jei yra daugiau nei vienas veiksmas, prieš visus veiksmus, išskyrus pirmąjį pridedamas žodis AND.
- Sakinio pirmojo žodžio prima raidė pakeičiama į tokią pačią mažąją.
- Prieš kiekvieną veiksmą pridedama jį atliekančio veikėjo pavadinimas.

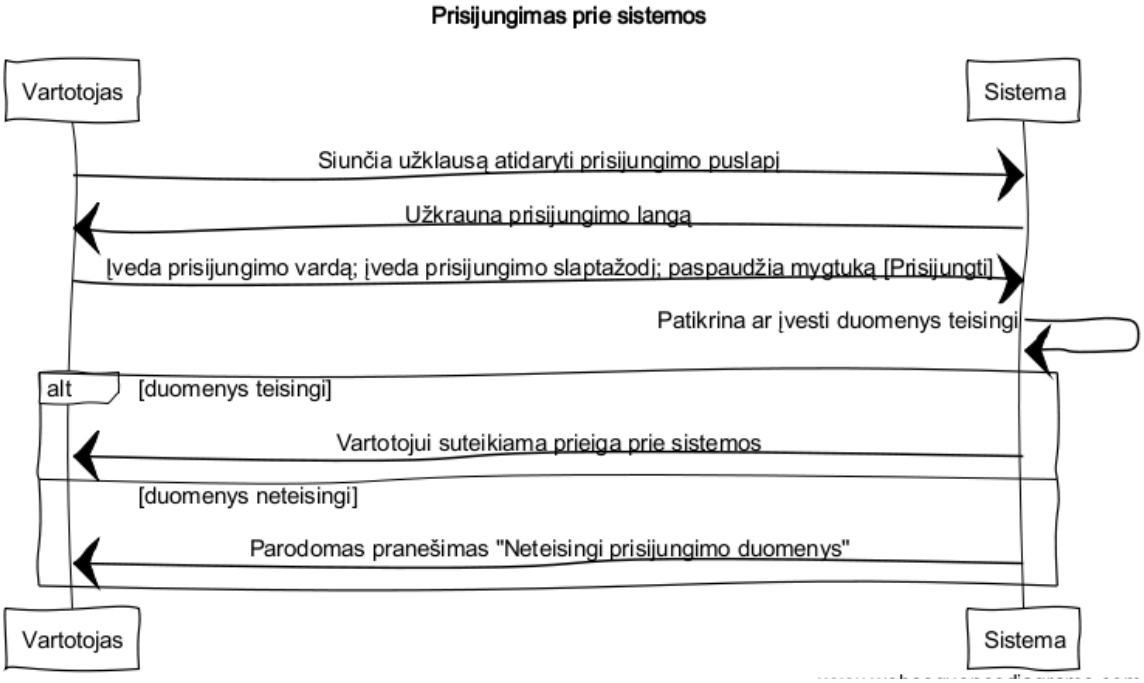
Taigi, apibendrinant galima teigti, kad norit pasiūlytu algoritmu generuoti *Gherkin* kodą iš sekų diagramos, reikia laikytis žemiau pateiktų reikalavimų.

Reikalavimai, norint taikyti šį algoritmą:

1. Generuoti tik iš aukšto lygio sekų diagramų, kurios turi tik du sąveikaujančius objektus.
2. Sekų diagramoje to paties veikėjo iš eilės einantys veiksmai turi būti surašyti ant vienos rodyklės ir atskirti „ ; “ ženklu.
3. Jei reikia, sekų diagramose naudoti tik šiuos operatorius: *alt* (alternatyva), *opt* (neprivalomas), *par* (paralelinis).
4. Prieš generavimą pažymėti sritis:
    a. Duomenų pateikimas
    b. Veiksmai
    c. Laukiamas rezultatas
5. Jei į sritį įtraukiamas operatorius, reikia įtraukti visą operatorių, o ne tik jo dalį.
6. Jei algoritmas parodo, kad yra vietų, į kurias reikėtų atkreipti dėmesį, jas peržvelgti ir reikalui esant pataisyti / papildyti.
7. Visi sekų diagramos žingsniai turi būti įtraukti į pažymėtas sritis, priešingu atveju nepažymėtos vietos nebus įtrauktos į sugeneruotą *Gherkin* testą.

Algoritmo taikymo pavyzdys

Toliau pateikiamas *Gherkin* kodo generavimo iš paprastos sekų diagramos pavyzdys. Pavyzdžiui naudojama jau anksčiau pateiktas sekų diagramos pavyzdys (žr. 5 pav.). Šioje diagramoje antroji alternatyva iš *else* pakeista į konkrečią reikšmę „duomenys neteisingi“, nes tokiu atveju nereikės taisyti sugeneruoto *Gherkin* testo.



24 pav. Sekų diagrama „Prisijungimas prie sistemos“

Šiame pavyzdyje pateikiamas paprastas atvejis, kai vartotojas, norėdamas prisijungti prie sistemos, turi įvesti prisijungimo vardą, slaptažodį ir paspausti mygtuką [Prisijungti]. Jei įvesti duomenys teisingi, jis gauna prieigą prie sistemos.

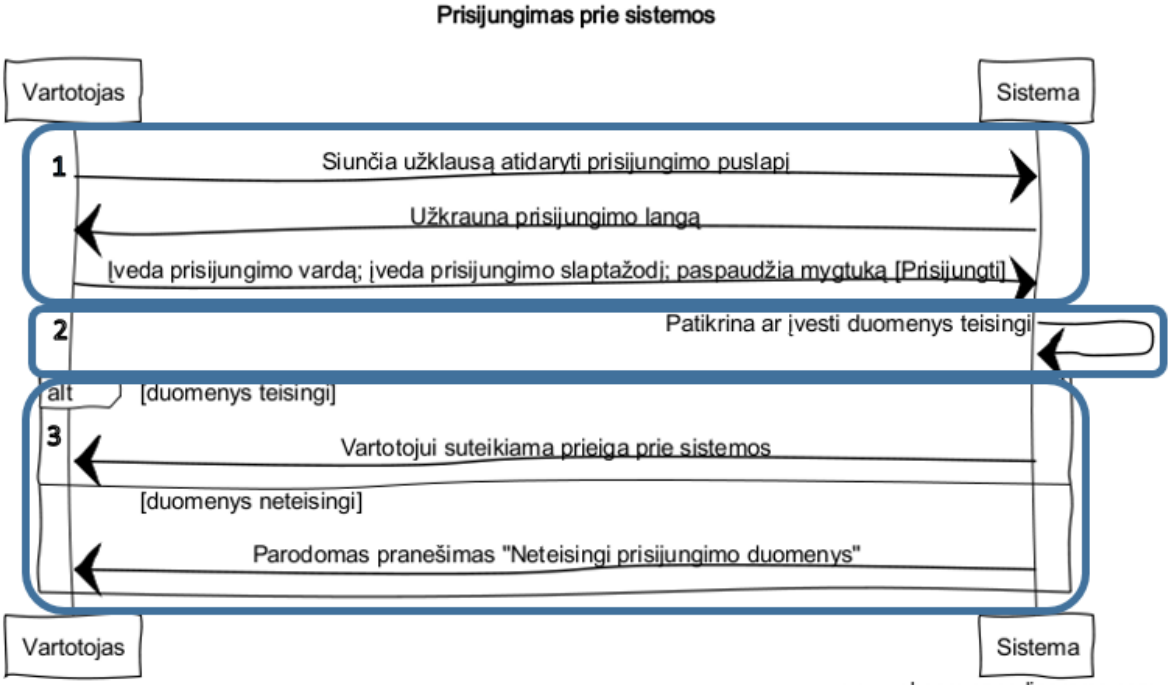
„*WebSequenceDiagrams*“ [18] puslapyje ši sekų diagrama aprašoma tokiu tekstiniu pavidalu:

```
title Prisijungimas prie sistemos


Vartotojas->Sistema: Siunčia užklausą atidaryti prisijungimo puslapį
Sistema->Vartotojas: Užkrauna prisijungimo langą
Vartotojas->Sistema: Įveda prisijungimo vardą; įveda prisijungimo slaptažodį; paspaudžia mygtuką [Prisijungti]
Sistema -> Sistema: Patikrina ar įvesti duomenys teisingi
alt duomenys teisingi
Sistema->Vartotojas: Vartotojui suteikiama prieiga prie sistemos
else duomenys neteisingi
Sistema->Vartotojas: Parodomas pranešimas "Neteisingi prisijungimo duomenys"
end
```

25 pav. Sekų diagramos tekstinis formatas

Turint tokį tekstą, galima iš jo generuoti *Gherkin* kodą ir taip sukurti testą. Tačiau galima naudoti ir kitus būdus, norint išgauti tekstinį diagramos formatą.

Taigi, pirmiausia pažymime tris sritis sekų diagramoje:



26 pav. Sekų diagrama su pažymėtomis sritimis

Atitinkamai į sritis suskirstomas diagramos tekstinis formatas:

```
        title Prisijungimas prie sistemos

1       Vartotojas->Sistema: Siunčia užklausą atidaryti prisijungimo puslapį
        Sistema->Vartotojas: Užkrauna prisijungimo langą
        Vartotojas->Sistema: Įveda prisijungimo vardą; įveda prisijungimo slaptažodį;
paspaudžia mygtuką [Prisijungti]
2       Sistema -> Sistema: Patikrina ar įvesti duomenys teisingi
3       alt duomenys teisingi
        Sistema->Vartotojas: Vartotojui suteikiama prieiga prie sistemos
        else duomenys neteisingi
        Sistema->Vartotojas: Parodomas pranešimas "Neteisingi prisijungimo
duomenys"
        end
```

27 pav. Į sritis padalintas sekų diagramos tekstas

Tuomet, pagal algoritmą paimamas pirmosios srities tekstas ir patikrinama, ar jame yra operatorių. Kadangi operatorių nėra, pradedama teksto transformacija. Pirmoje eilutėje nerandama „;“ ženklo, todėl ji tiesiog priskiriama prie GIVEN dalies, prieš veiksmą prirašant veikėjo pavadinimą. Gaunama:

GIVEN Vartotojas siunčia užklausą atidaryti prisijungimo puslapį

Paimamas antras sakinys. Jame taip pat nerandama ženklo „;“. Jis prirašomas kitoje eilutėje prieš tai pridedant AND ir pridedant veikėjo pavadinimą:

GIVEN Vartotojas siunčia užklausą atidaryti prisijungimo puslapį
AND Sistema užkrauna prisijungimo langą

Paimamas trečias sakinys „Įveda prisijungimo vardą; įveda prisijungimo slaptažodį; paspaudžia mygtuką [Prisijungti]“. Jame randami du ženklai „;“, todėl vienas sakinys suskaldomas į tris: „Įveda prisijungimo vardą“, „įveda prisijungimo slaptažodį“, „paspaudžia mygtuką [Prisijungti]“. Visi šie sakiniai parašomi iš naujos eilutės ir kadangi nė vienas iš šių veiksmų nėra pirmas GIVEN dalyje, prieš visus juos pridedamas žodis AND. Pirmojo sakinio raidė pakeičiama į mažąją ir prieš visus veiksmus parašomas veikėjo pavadinimas:

GIVEN Vartotojas siunčia užklausą atidaryti prisijungimo puslapį
AND Sistema užkrauna prisijungimo langą
AND Vartotojas įveda prisijungimo vardą
AND Vartotojas įveda prisijungimo slaptažodį
AND Vartotojas paspaudžia mygtuką [Prisijungti]

Paimamas tekstas iš antros pažymėtos srities. Šis tekstas bus pildomas į *Gherkin* testo sritį WHEN. Paimtame tekste „Sistema -> Sistema: Patikrina ar įvesti duomenys teisingi“ nerandama nei operatorių nei „;“ ženklų, todėl atliekamos paskutinės transformacijos: sakinio pirma raidė pakeičiama į mažąją ir pradžioje pridedamas veikėjo pavadinimas. Gaunama:

GIVEN Vartotojas siunčia užklausą atidaryti prisijungimo puslapį
AND Sistema užkrauna prisijungimo langą
AND Vartotojas įveda prisijungimo vardą
AND Vartotojas įveda prisijungimo slaptažodį
AND Vartotojas paspaudžia mygtuką [Prisijungti]
WHEN Sistema patikrina ar įvesti duomenys teisingi

Paimamas tekstas iš trečios pažymėtos srities.

„alt duomenys teisingi
Sistema->Vartotojas: Vartotojui suteikiama prieiga prie sistemos
else duomenys neteisingi
Sistema->Vartotojas: Parodomas pranešimas "Neteisingi prisijungimo duomenys"
end”

Randamas operatorius *alt* ir du jo regionai, todėl jau sugeneruota testo dalis pakartojama du kartus:

GIVEN Vartotojas siunčia užklausą atidaryti prisijungimo puslapį
AND Sistema užkrauna prisijungimo langą
AND Vartotojas įveda prisijungimo vardą
AND Vartotojas įveda prisijungimo slaptažodį
AND Vartotojas paspaudžia mygtuką [Prisijungti]
WHEN Sistema patikrina ar įvesti duomenys teisingi

GIVEN Vartotojas siunčia užklausą atidaryti prisijungimo puslapį
AND Sistema užkrauna prisijungimo langą
AND Vartotojas įveda prisijungimo vardą
AND Vartotojas įveda prisijungimo slaptažodį
AND Vartotojas paspaudžia mygtuką [Prisijungti]
WHEN Sistema patikrina ar įvesti duomenys teisingi

Kadangi pildoma trečioji testo dalis, regiono sąlygos pridedamos prie prieš tai buvusios testo dalies WHEN ir pažymima kaip reikalaujanti peržiūros (šiame darbe tokį tekstą pajuodinsime).

GIVEN Vartotojas siunčia užklausą atidaryti prisijungimo puslapį
AND Sistema užkrauna prisijungimo langą
AND Vartotojas įveda prisijungimo vardą
AND Vartotojas įveda prisijungimo slaptažodį
AND Vartotojas paspaudžia mygtuką [Prisijungti]
WHEN Sistema patikrina ar įvesti duomenys teisingi
duomenys teisingi

GIVEN Vartotojas siunčia užklausą atidaryti prisijungimo puslapį
AND Sistema užkrauna prisijungimo langą
AND Vartotojas įveda prisijungimo vardą
AND Vartotojas įveda prisijungimo slaptažodį
AND Vartotojas paspaudžia mygtuką [Prisijungti]
WHEN Sistema patikrina ar įvesti duomenys teisingi
duomenys neteisingi

Regione atliekami veiksmai surašomi į trečiąją testo dalį:

GIVEN Vartotojas siunčia užklausą atidaryti prisijungimo puslapį
AND Sistema užkrauna prisijungimo langą
AND Vartotojas įveda prisijungimo vardą
AND Vartotojas įveda prisijungimo slaptažodį
AND Vartotojas paspaudžia mygtuką [Prisijungti]
WHEN Sistema patikrina ar įvesti duomenys teisingi
duomenys teisingi
THEN Vartotojui suteikiama prieiga prie sistemos

GIVEN Vartotojas siunčia užklausą atidaryti prisijungimo puslapį
AND Sistema užkrauna prisijungimo langą
AND Vartotojas įveda prisijungimo vardą
AND Vartotojas įveda prisijungimo slaptažodį
AND Vartotojas paspaudžia mygtuką [Prisijungti]
WHEN Sistema patikrina ar įvesti duomenys teisingi
duomenys neteisingi
THEN Parodomas pranešimas "Neteisingi prisijungimo duomenys"

Kiekvienam pridėtam tekstui atliekamos veiksmų transformacijos: pridedamas žodis AND, prirašomas veikėjo pavadinimas, pirma sakinio raidė pakeičiama į mažąją. Gaunamas galutinis rezultatas:

GIVEN Vartotojas siunčia užklausą atidaryti prisijungimo puslapį
AND Sistema užkrauna prisijungimo langą
AND Vartotojas įveda prisijungimo vardą
AND Vartotojas įveda prisijungimo slaptažodį
AND Vartotojas paspaudžia mygtuką [Prisijungti]
WHEN Sistema patikrina ar įvesti duomenys teisingi
AND duomenys teisingi
THEN Sistema vartotojui suteikiama prieiga prie sistemos

GIVEN Vartotojas siunčia užklausą atidaryti prisijungimo puslapį
AND Sistema užkrauna prisijungimo langą
AND Vartotojas įveda prisijungimo vardą

AND Vartotojas įveda prisijungimo slaptažodį
AND Vartotojas paspaudžia mygtuką [Prisijungti]
WHEN Sistema patikrina ar įvesti duomenys teisingi
AND duomenys neteisingi
THEN Sistema parodomas pranešimas "Neteisingi prisijungimo duomenys"

Gautą rezultatą dar gali peržiūrėti testuotojas ir atlikti nežymius koregavimus, jei jų reikia. Testuotojui jau pažymėtos vietos, į kurias jam reikėtų atkreipti dėmesį. Šiuo atveju teksto prasmė sudėta teisingai, nebent galima pakeisti raidžių dydį. Taip pat peržiūros metu galima pastebėti, kad THEN sakiniuose nebūtinas žodis „Sistema“, tačiau tai tik perteklinė informacija, kuri netrukdytų naudojant sugeneruotą testą. Pridėjus testuotojo koregavimus gaunamas toks tekstas:

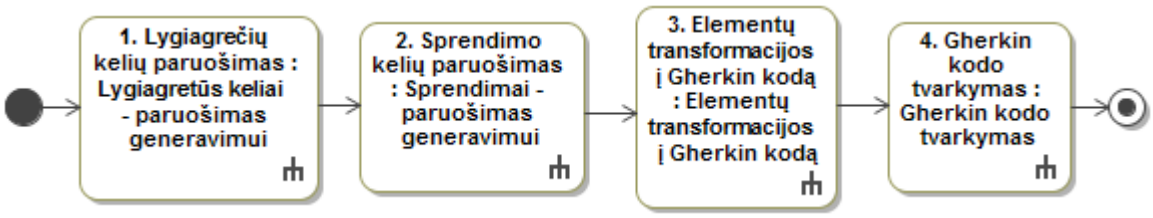
GIVEN Vartotojas siunčia užklausą atidaryti prisijungimo puslapį
AND Sistema užkrauna prisijungimo langą
AND Vartotojas įveda prisijungimo vardą
AND Vartotojas įveda prisijungimo slaptažodį
AND Vartotojas paspaudžia mygtuką [Prisijungti]
WHEN Sistema patikrina ar įvesti duomenys teisingi
AND Duomenys teisingi
THEN Vartotojui suteikiama prieiga prie sistemos

GIVEN Vartotojas siunčia užklausą atidaryti prisijungimo puslapį
AND Sistema užkrauna prisijungimo langą
AND Vartotojas įveda prisijungimo vardą
AND Vartotojas įveda prisijungimo slaptažodį
AND Vartotojas paspaudžia mygtuką [Prisijungti]
WHEN Sistema patikrina ar įvesti duomenys teisingi
AND Duomenys neteisingi
THEN Parodomas pranešimas "Neteisingi prisijungimo duomenys"

2.1.3. *Gherkin* testo generavimas iš veiklos diagramos

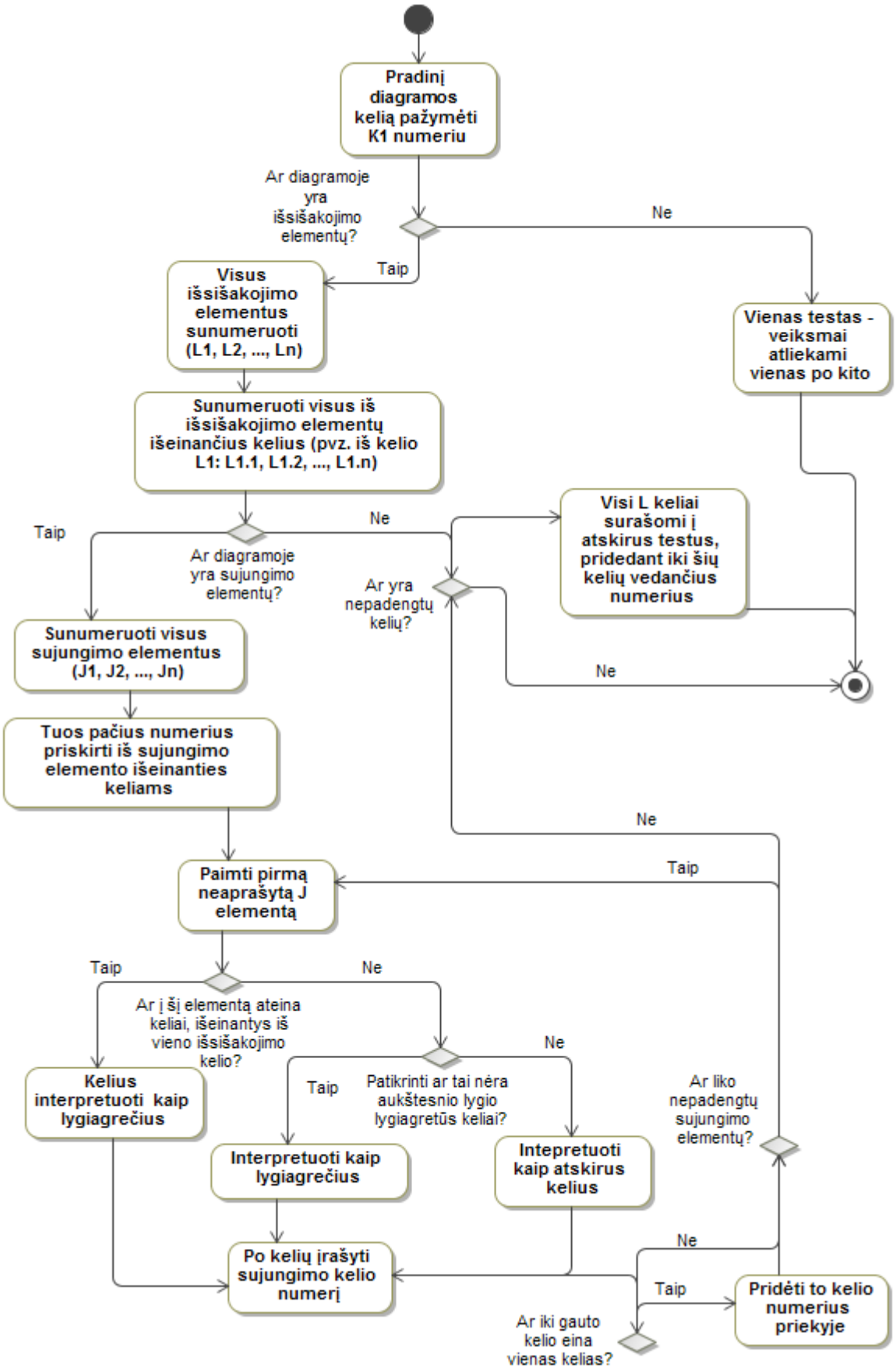
Analizės metu (žr. sk. 1.5.8 „Sprendimas, kaip generuoti veiklos diagramą iš *Gherkin* kodo“) buvo rastas jau egzistuojantis sprendimas su pavyzdžiu kaip iš *Gherkin* kodo sugeneruotos veiklos diagramos. Nors autorius nepateikė algoritmo kaip tai padaryti, buvo galima pastebėti kelias

tendencijas. Šiame darbe buvo sukurtas atvirkštinis algoritmas – kaip iš veiklos diagramos sugeneruoti *Gherkin* kodą.



28 pav. Veiklos diagramos generavimo į *Gherkin* kodą algoritmas

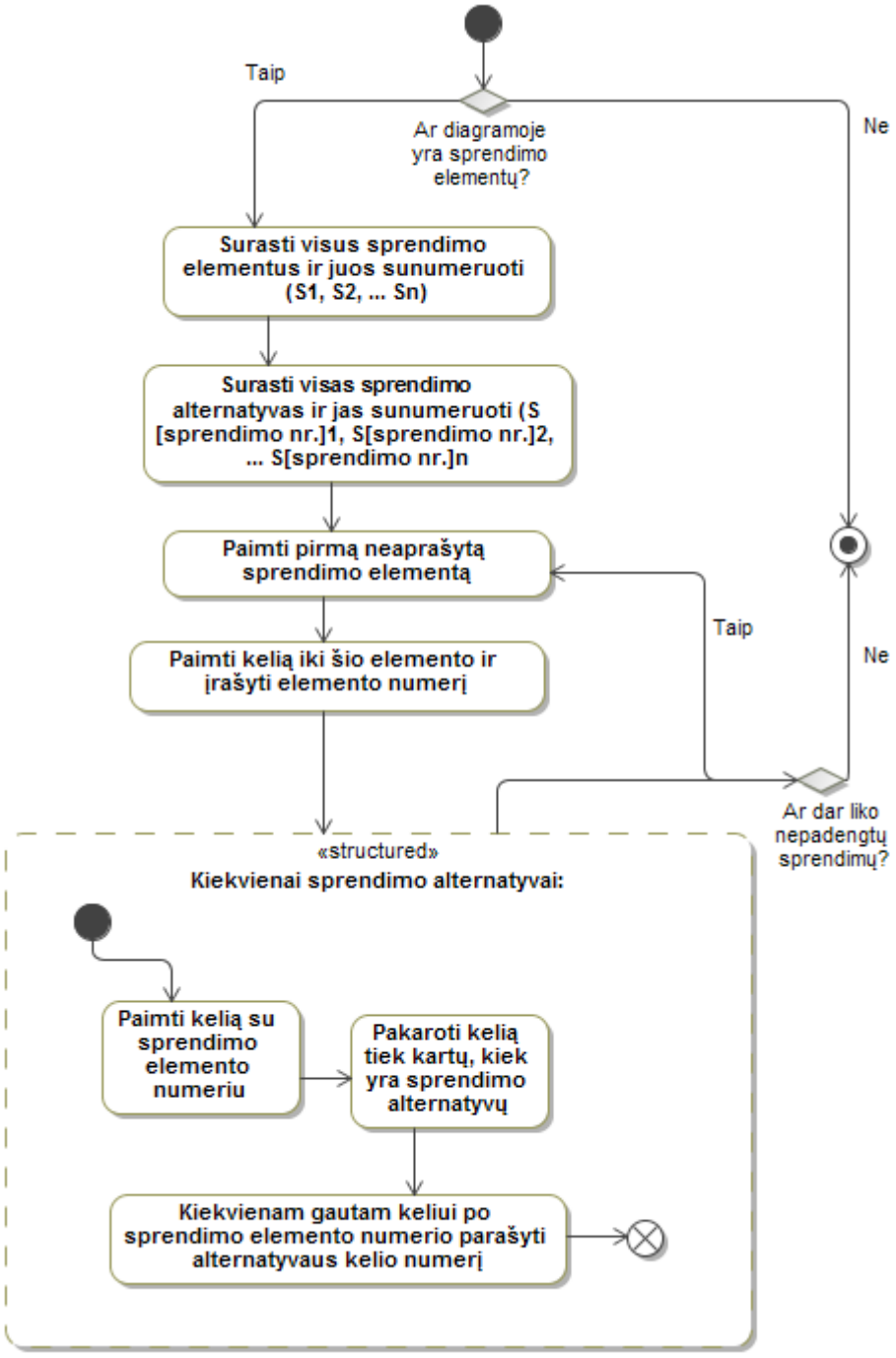
Paveikslėlyje 28 pav. pateikiamas veiklos diagramos generavimo į *Gherkin* kodą algoritmas. Šis algoritmas sudarytas iš keturių didelių žingsnių, kurie yra atvaizduoti kitose diagramose. Pirmi du žingsniai yra paruošiamieji darbai – sunumeruojami diagramos keliai, tikrinama ar diagramoje yra lygiagrečių kelių arba sprendimų ir pagal tai sudėliojami būsimų testų scenarijai – kelių numeriai sudėti eilės tvarka. Trečiame žingsnyje, einant anksčiau sudėliotais keliais, tikrinami diagramos elementai ir transformuojami į *Gherkin* kodą. Ketvirtame žingsnyje, paimami sugeneruoti *Gherkin* testai ir šiek tiek pakoreguojamas kodas. Toliau pateikiamos šios keturios diagramos.



29 pav. Diagrama „Lygiagretūs keliai – paruošimas generavimui“

Reikalavimų formalus aprašas diagramai „Lygiagretūs keliai – paruošimas generavimui“:

1. Patį pirmą diagramos kelią pažymėti K1 numeriu. Čia ir visur kitur, žymint kelią, pažymima tik jo pradžia, bet interpretuojama kaip visas kelias iki pirmo išsišakojimo.
2. Randami išsišakojimo elementai ir jiems suteikiami numeriai, prasidedantys „L“ raide. Atitinkamai iš išsišakojimo elemento išeinantys keliai pažymimi elemento numeriu + „.“ + kelio numeriu, pvz. pirmo išsišakojimo trečias kelias bus pažymėtas L1.3. Jei nerandama išsišakojimo kelių, taip ir paliekamas kelias su numeriu K1.
3. Ieškoma sujungimo elementų ir jie sunumeruojami, pridedant raides „J“. Tie patys numeriai suteikiami ir iš šių elementų išeinantiems keliams.
4. Tikrinama, kurie keliai įeina į sujungimo elementą. Jei nustatoma, kad keliai išėjo iš to pačio išsišakojimo elemento, jie interpretuojami kaip lygiagretūs. Čia ir kitur, interpretuojant kelius kaip lygiagrečius, kelių numeriai surašomi iš eilės į vieną testą.
5. Pagal algoritmą, gavus lygiagrečių kelių numerius, gale numerių pridedamas ir sujungimo elemento / kelio numeris.
6. Pradžioje gauto kelio pridedamas kelio, vedančio iki pirmų elementų numeris.
7. Tikrinant, ar tai nėra aukštesnio lygio lygiagretūs keliai, tikrinami keliai į viršų, kol bus rastas taškas, iš kurio keliai išsišakojo (išsišakojimo elemente) arba iki veiklos diagramos pradžios K1. Tikrinimo rezultatai yra teigiami, jei randama, kad keliai kurioj nors vietoj išeina iš to paties išsišakojimo elemento.

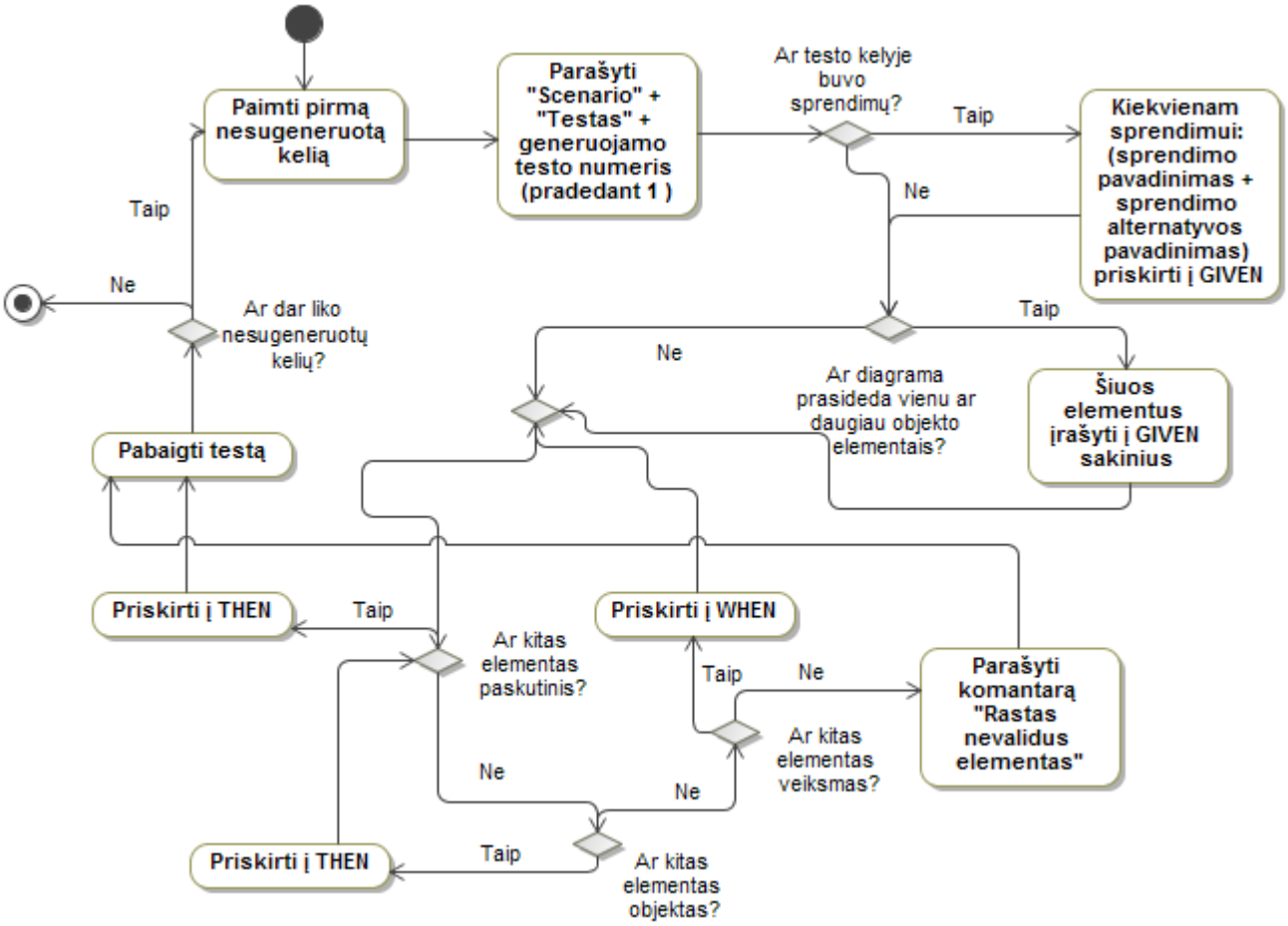


30 pav. Diagrama „Sprendimai – paruošimas generavimui“

Reikalavimų formalus aprašas diagramai „Sprendimai – paruošimas generavimui“:

1. Patikrinama ar diagramoje yra sprendimo elementų. Jei nėra, daugiau veiksmų šiame žingsnyje neatliekama. Jei yra, visi sprendimo elementai sunumeruojami, priekyje prirašant raidę „S“.
2. Keliai, išeinantys iš sprendimo elementų sunumeruojami: sprendimo numeris + „.“ + alternatyvos numeris. Pvz. Antro sprendimo pirma alternatyva būtų žymima S2.1.
3. Iš eilės imami sprendimo elementai, paimamas kelias iki šio elemento ir reikiamoje vietoje įrašomas elemento numeris.

4. Sukuriama tiek turimo kelio versijų, kiek sprendimo elementas turi alternatyvų. Atitinkamai, kiekvienam gautam keliui, po sprendimo elemento numerio parašomas alternatyvos numeris.

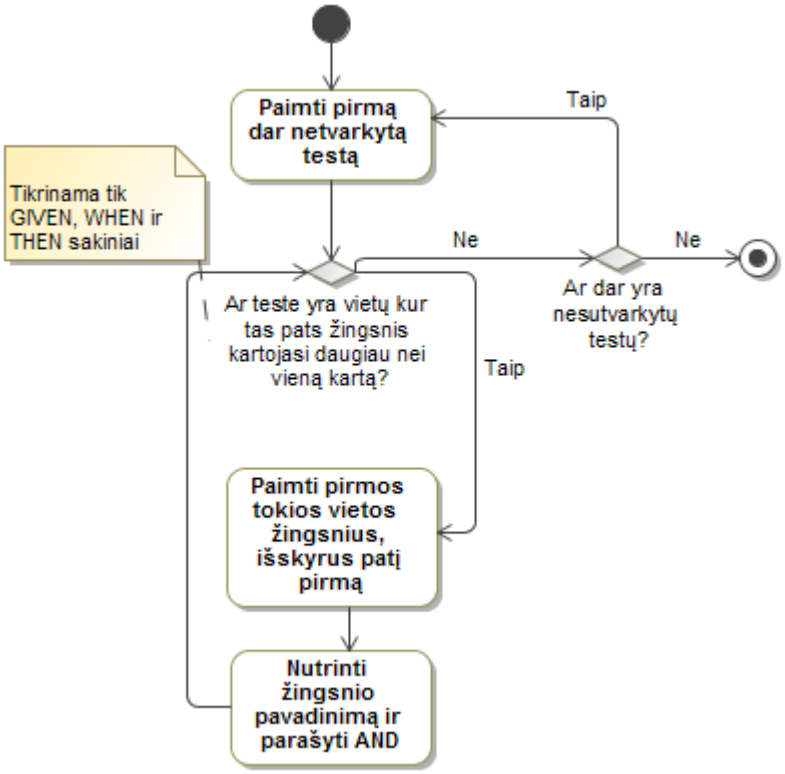


31 pav. Diagrama „Elementų transformacijos į *Gherkin* kodą“

Reikalavimų formalus aprašas diagramai „Elementų transformacijos į *Gherkin* kodą“:

1. Kiekvienam ankstesniuose žingsniuose sugeneruotam keliui, sukuriama po testą, pradžioje parašant „Scenario Testas (testo numeris)“. Numeracija pradedama nuo 1. Esant poreikiui, galima pridėti ir kitokius testų pavadinimus.
2. Tikrinant ar teste buvo sprendimų, patikrinama ar kelio kode yra S raidžių.
3. Jei buvo, sprendimo elemento tekstas + alternatyvos tekstas įrašomas į *Gherkin* testo GIVEN dalį.
4. Jei diagrama prasideda objekto elementais, jie priskiriami į GIVEN dalį, kiekvienam elementui teste parašant „GIVEN“ + „objekto elemento tekstas“. Analogiškai atliekami ir kiti priskyrimai į *Gherkin* testo dalis.

5. Tikrinant diagramos elementus tikrinami visi elementai, išskyrus rodykles, sujungimo elementus, išsišakojimo elementus, sprendimo elementus ir apjungimo elementus. Kelyje radus šiuos elementus, jie yra ignoruojami ir einama prie kito elemento.



32 pav. Diagrama „*Gherkin* kodo tvarkymas“

Reikalavimų formalus aprašas diagramai „*Gherkin* kodo tvarkymas“:

1. Paimamas pirmas dar nesutvarkytas testas.
2. Patikrinama ar jame yra vietų, kur iš eilės kartojasi tas pats žingsnis: GIVEN, WHEN arba THEN.
3. Jei tokių vietų yra, paimami patys pirmieji besikartojantys sakiniai, išskyrus patį pirmą iš jų. Paimtiems sakiniams žingsnio žodis pakeičiamas žodžiu AND.
4. Taip kartojama visoms besikartojančioms vietoms, visuose ankstesniame žingsnyje sugeneruotuose testuose.

Reikalavimai, norint taikyti šį algoritmą:

1. Algoritmas gali būti taikomas tik toms veiklos diagramoms, kurios turi tik šiuos išvardintus elementus:
    - Veiksmas
    - Objektas
    - Sujungimas

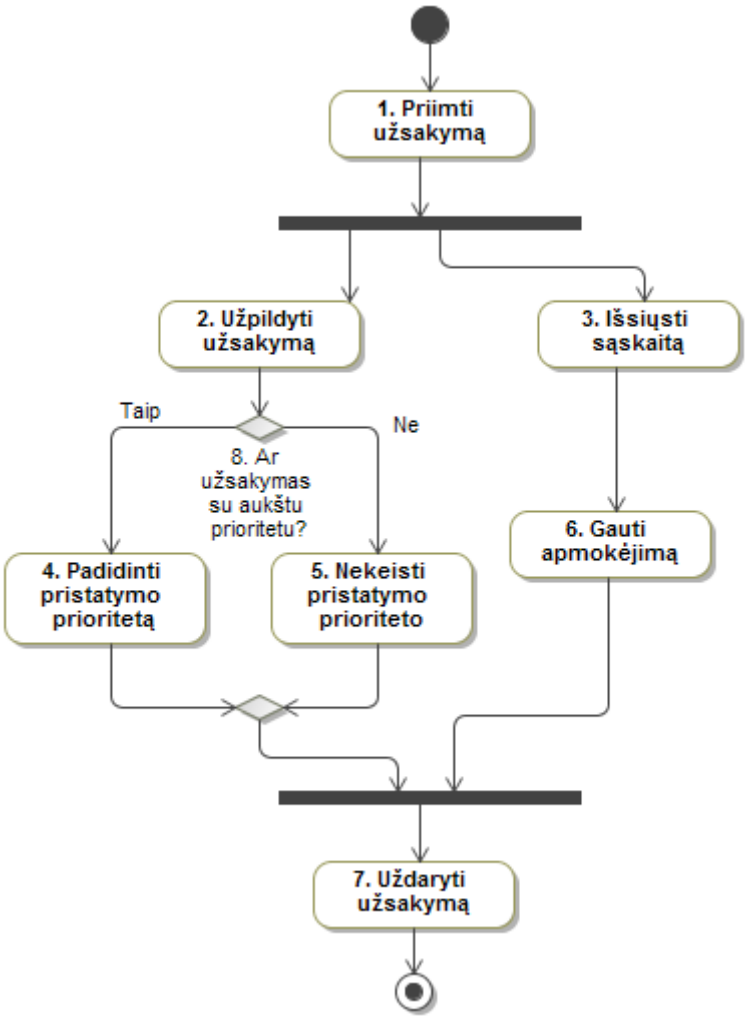
- Išsišakojimas
- Sprendimas
- Veiklos arba objekto rodyklė

Jei diagramos kelyje algoritmas aptiks elementą ne iš šio sąrašo, bus parašomas pranešimas „Rastas nevalidus elementas“ ir pabaigiamas testo generavimas tam keliui.

2. Diagrama turi turėti tik vieną pradžią.
3. Jei diagramoje yra takeliai su veikėjais, algoritmas tai ignoruos, todėl pateikiant diagramą generavimui, reikia į tai atsižvelgti.
4. Pateikiamoje diagramoje neturi būti ciklų.

Algoritmo taikymo pavyzdys

Tarkime turime tokią veiklos diagramą:



33 pav. Veiklos diagrama „Užsakymo tvarkymas“

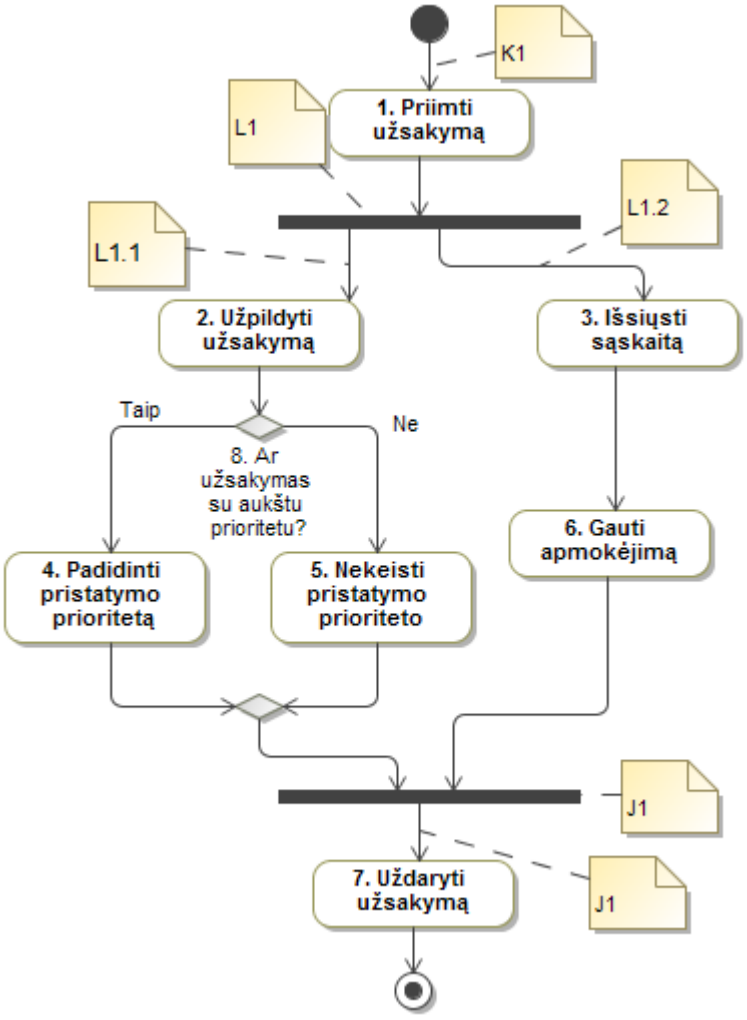
Šioje veiklos diagramoje pavaizduota veikla, kai gaunamas užsakymas. Ši diagrama turi du lygiagrečius kelius – užsakymo tvarkymą ir apmokėjimo tvarkymą. Taip pat, šioje diagramoje yra ir

sprendimo elementas, skirtas patikrinti užsakymo prioritetui. Priklausomai nuo to, koks gauto užsakymo prioritetas, atliekami skirtingi veiksmai su užsakymu.

Pateikus šią diagramą algoritmui, pirmiausia atliekamas **lygiagrečių kelių tikrinimo algoritmas**.

1. Pradinis diagramos kelias pažymimas K1 numeriu.
2. Randamas vienas išsišakojimo elementas ir pažymimas L1.
3. Sunumeruojami iš šio elemento išeinantys keliai: L1.1, L1.2.
4. Randamas vienas sujungimo elementas ir pažymimas J1.
5. Iš sujungimo elemento išeinantis kelias pažymimas J1.

Diagramos kelių žymėjimai:



34 pav. Sužymėti keliai veiklos diagramoje „Užsakymo tvarkymas“ 1

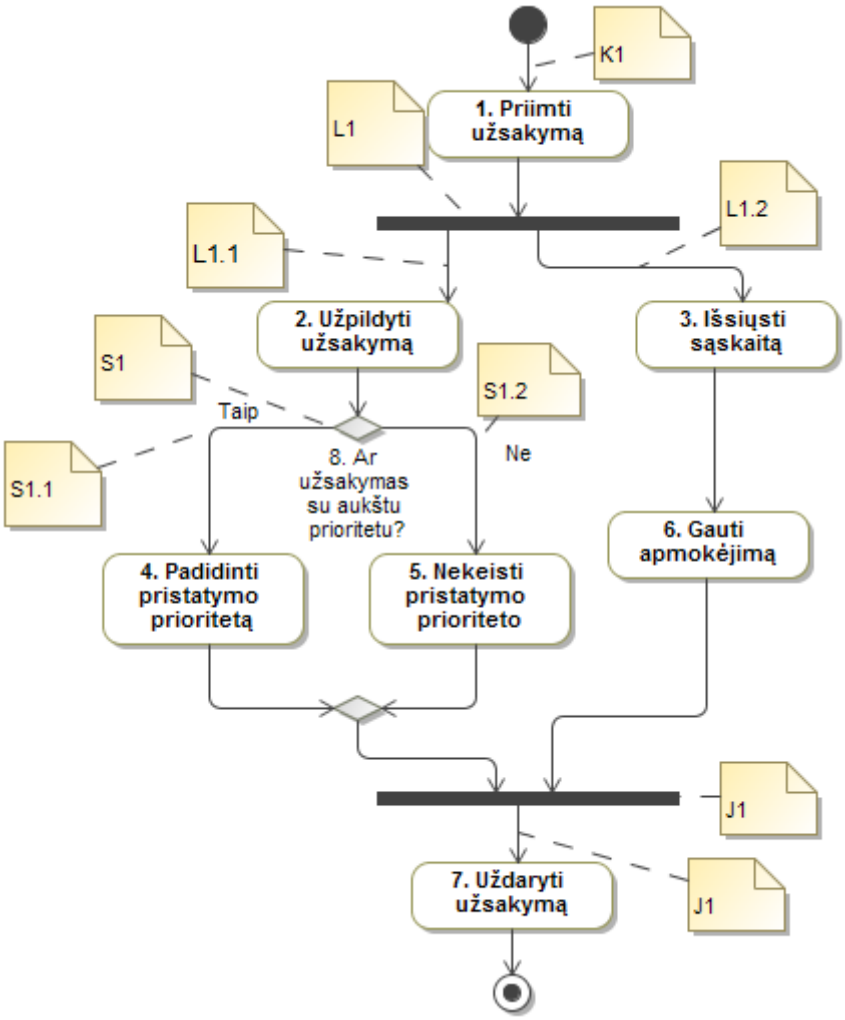
6. Patikrinama ir randama, kad į elementą J1 įeina abu iš išsišakojimo L1 išeinantys keliai.

7. Šie keliai suprantami kaip lygiagretūs, todėl jie bus vykdomi vienas po kito, o kelių numeriai surašomi į vieną testą: K1, L1.1, L1.2.
8. Prie testo prirašomas po sujungimo elemento einančio kelio numeris, gaunama: K1, L1.1, L1.2, J1.

Po lygiagrečių kelių paruošimo atliekamas **sprendimo kelių paruošimas**:

1. Diagramoje randamas vienas sprendimo elementas ir pažymimas S1.
2. Randamos dvi sprendimo alternatyvos ir pažymimos S1.1 ir S1.2.

Sužymėta diagrama atrodo taip:



35 pav. Sužymėti keliai veiklos diagramoje „Užsakymo tvarkymas“ 2

3. Paimamas jau sugeneruotas kelių kodas: K1, L1.1, L1.2, J1.
4. Kelyje, kur yra sprendimo elementas, įrašomas sprendimo elemento numeris (pajuodinta): K1, L1.1, **S1**, L1.2, J1.
5. Kadangi sprendimas turi tik dvi alternatyvas, šis kelias padauginamas du kartus:

   K1, L1.1, S1, L1.2, J1

   K1, L1.1, S1, L1.2, J1

6. Kiekvienam iš šių kelių po sprendimo elemento numerio parašomas vis kitos alternatyvos numeris:

   K1, L1.1, S1, **S1.1**, L1.2, J1

   K1, L1.1, S1, **S1.2**, L1.2, J1

7. Gaunami du keliai, kuriems bus kuriamas testas.

Po šių paruošiamųjų darbų atliekamas elementų transformacijos į *Gherkin* kodą algoritmas:

1. Paimamas pirmas nesugeneruotas kelias: K1, L1.1, S1, S1.1, L1.2, J1.
2. Pradedamas testas – parašomas jo pavadinimas – „Scenario: Testas [numeris]“:

Scenario: Testas 1

3. Kadangi testo kelyje K1, L1.1, S1, S1.1, L1.2, J1 buvo sprendimas – jo pavadinimas ir alternatyvos pavadinimas pridedami į GIVEN dalį. Algoritmas tik nukopijuoja tai, kas yra sąlygoje, kad nebūtų prarasta prasmė. Vėliau bus galima šį tekstą koreguoti ranka.

Scenario: Testas 1 **GIVEN Ar užsakymas su aukštu prioritetu? Taip**

4. Tikrinama ar diagrama prasideda objekto elementais. Kadangi taip nėra, jokie veiksmai neatliekami.
5. Tikrinamas pirmas diagramos elementas nr. 1:
   a. Elementas kelyje nėra paskutinis, todėl tikrinama ar tai objektas.
   b. Kadangi tai ne objektas, tikrinama ar tai veiksmas.
   c. Kadangi tai veiksmas, jis prirašomas kaip WHEN. Gaunama:

Scenario: Testas 1 GIVEN Ar užsakymas su aukštu prioritetu? Taip **WHEN Priimti užsakymą**

6. Tikrinamas kitas diagramos elementas – nr. 2:
   a. Elementas kelyje nėra paskutinis, todėl tikrinama ar tai objektas.
   b. Kadangi tai ne objektas, tikrinama ar tai veiksmas.
   c. Kadangi tai veiksmas, jis prirašomas kaip WHEN. Gaunama:

Scenario: Testas 1 GIVEN Ar užsakymas su aukštu prioritetu? Taip WHEN Priimti užsakymą **WHEN Užpildyti užsakymą**

7. a, b ir c veiksmai atliekami ir likusiems diagramos elementams: 4, 3 ir 6. Gaunama:

Scenario: Testas 1

GIVEN Ar užsakymas su aukštu prioritetu? Taip
WHEN Priimti užsakymą
WHEN Užpildyti užsakymą
WHEN Padidinti pristatymo prioritetą
WHEN Išsiųsti sąskaitą
WHEN Gauti apmokėjimą

8. Tikrinamas diagramos elementas nr. 7:
    a. Patikrinus ar elementas yra paskutinis diagramoje, gaunamas teigiamas atsakymas, todėl jis priskiriamas į THEN. Gaunama:

Scenario: Testas 1
GIVEN Ar užsakymas su aukštu prioritetu? Taip
WHEN Priimti užsakymą
WHEN Užpildyti užsakymą
WHEN Padidinti pristatymo prioritetą
WHEN Išsiųsti sąskaitą
WHEN Gauti apmokėjimą
THEN Uždaryti užsakymą

9. Kadangi tai buvo paskutinis diagramos elementas pabaigiamas testas.
10. Patikrinama ar dar liko nesugeneruotų kelių.
11. Kadangi liko, paimamas kitas nesugeneruotas kelias: K1, L1.1, S1, S1.2, L1.2, J1.
12. Pradedamas testas – parašomas jo pavadinimas – „Scenario: Testas [numeris]“:

Scenario: Testas 2

13. Kadangi testo kelyje K1, L1.1, S1, S1.2, L1.2, J1 buvo sprendimas – jo pavadinimas ir alternatyvos pavadinimas pridedami į GIVEN dalį. Algoritmas tik nukopijuoja tai, kas yra sąlygoje, kad nebūtų prarasta prasmė. Vėliau bus galima šį tekstą koreguoti ranka.

Scenario: Testas 2
GIVEN Ar užsakymas su aukštu prioritetu? Ne

14. Tikrinama ar diagrama prasideda objekto elementais. Kadangi taip nėra, jokie veiksmai neatliekami.
15. Tikrinamas pirmas diagramos elementas nr. 1:
    a. Elementas kelyje nėra paskutinis, todėl tikrinama ar tai objektas.
    b. Kadangi tai ne objektas, tikrinama ar tai veiksmas.

c. Kadangi tai veiksmas, jis prirašomas kaip WHEN. Gaunama:

Scenario: Testas 2 GIVEN Ar užsakymas su aukštu prioritetu? Taip **WHEN Priimti užsakymą**

16. a, b ir c veiksmai atliekami ir likusiems diagramos elementams:2, 5, 3 ir 6. Gaunama:

Scenario: Testas 2 GIVEN Ar užsakymas su aukštu prioritetu? Ne WHEN Priimti užsakymą WHEN Užpildyti užsakymą **WHEN Nekeisti pristatymo prioriteto** **WHEN Išsiųsti sąskaitą** **WHEN Gauti apmokėjimą**

17. Tikrinamas diagramos elementas nr. 7:
    a. Patikrinus ar elementas yra paskutinis diagramoje, gaunamas teigiamas atsakymas, todėl jis priskiriamas į THEN. Gaunama:

Scenario: Testas 2 GIVEN Ar užsakymas su aukštu prioritetu? Ne WHEN Priimti užsakymą WHEN Užpildyti užsakymą WHEN Nekeisti pristatymo prioriteto WHEN Išsiųsti sąskaitą WHEN Gauti apmokėjimą **THEN Uždaryti užsakymą**

18. Kadangi tai buvo paskutinis diagramos elementas, pabaigiamas testas.
19. Patikrinama, ar dar liko nesugeneruotų kelių.
20. Kadangi tokių kelių neliko, pabaigiamas elementų transformacijos į *Gherkin* kodą algoritmas. Rezultatas – gautas dviejų testų kodas.

Atlikus šį algoritmą atliekamas *Gherkin* kodo tvarkymo algoritmas:

1. Paimamas dar netvarkytas testas – Testas 1.

Scenario: Testas 1 GIVEN Ar užsakymas su aukštu prioritetu? Taip WHEN Priimti užsakymą

WHEN Užpildyti užsakymą
WHEN Padidinti pristatymo prioritetą
WHEN Išsiųsti sąskaitą
WHEN Gauti apmokėjimą
THEN Uždaryti užsakymą

2. Patikrinamas ir randama, kad teste yra vietų, kur tas pats žingsnis kartojasi daugiau nei vieną kartą.
3. Paimami visi WHEN žingsniai, išskyrus patį pirmą WHEN žingsnį.
4. Šiems žingsniams nutrinamas žodis WHEN ir vietoj jo parašoma AND. Gaunama:

Scenario: Testas 1
GIVEN Ar užsakymas su aukštu prioritetu? Taip
WHEN Priimti užsakymą
AND Užpildyti užsakymą
AND Padidinti pristatymo prioritetą
AND Išsiųsti sąskaitą
AND Gauti apmokėjimą
THEN Uždaryti užsakymą

5. Tikrinama, ar Teste 1 yra daugiau vietų kur kartojasi tas pats žingsnis iš eilės. Daugiau tokių vietų nerandama.
6. Patikrinus ar dar yra nesutvarkytų randama, kad dar nesutvarkytas Testas 2.
7. Paimamas Testas 2:

Scenario: Testas 2
GIVEN Ar užsakymas su aukštu prioritetu? Ne
WHEN Priimti užsakymą
WHEN Užpildyti užsakymą
WHEN Nekeisti pristatymo prioriteto
WHEN Išsiųsti sąskaitą
WHEN Gauti apmokėjimą
THEN Uždaryti užsakymą

8. Atliekami šio algoritmo veiksmai 2, 3, 4 ir gaunamas sutvarkytas testas:

Scenario: Testas 2

GIVEN Ar užsakymas su aukštu prioritetu? Ne WHEN Priimti užsakymą AND Užpildyti užsakymą AND Nekeisti pristatymo prioriteto AND Išsiųsti sąskaitą AND Gauti apmokėjimą AND Uždaryti užsakymą

9. Patikrinus, ar yra daugiau nesutvarkytų testų, tokių nerandama, todėl algoritmas baigiamas.
10. Taip pat baigiamas ir didysis algoritmas. Rezultatas – paruošti du *Gherkin* testai.

2.1.4. *Gherkin* testo generavimas iš lentelės

Kartais patogu apibendrinti reikalavimus lentelės pavidalu. Pavyzdžiui, kai specifikuojama daug panašių veiksmų su panašiais duomenimis, tarkim, skaičiuotuvo veiksmai, kai įvedami du skaičiai, tačiau su jais galima atlikti daug įvairių operacijų: sudėti, atimti, dalinti, dauginti ir kitus. Taip pat, lentelė yra patogus variantas, jei reikia ištestuoti įvairias veiksmų su duomenimis variacijas, nes lentelė gali parodyti testų padengimą ir kraštines reikšmes. Lentelėmis pateikus reikalavimus, galima labai paprasta sugeneruoti *Gherkin* testus.

Testų paruošimo, lentelės pavidalu, algoritmas:

1. Į lentelės stulpelių pavadinimus surašyti duomenis. Priklausomai nuo poreikio, tai gali būti konkrečios reikšmės (nebūtinai skaičiai), arba generuojami atsitiktiniai skaičiai x, $x \in R(x_1, x_2, ..., x_n)$ tam tikruose intervaluose: $0<x\leq100$, $1000\leq x\leq9999$ ir pan.
2. Į pirmą lentelės stulpelį surašyti norimus veiksmus.

Taip turėtų atrodyti paruošta lentelė:

Veiksmai/ duota	Duomenys 1	Duomenys 2	Duomenys 3	Duomenys 4
Veiksmas 1	Rezultatai 1	Rezultatai 2	Rezultatai 3	Rezultatai 4
Veiksmas 2	Rezultatai 5	Rezultatai 6	Rezultatai 6	Rezultatai 7
Veiksmas 3	Rezultatai 8	Rezultatai 9	Rezultatai 10	Rezultatai 11
Veiksmas 4	Rezultatai 12	Rezultatai 13	Rezultatai 14	Rezultatai 15

Dabar galima pasirinkti kokius duomenis ir veiksmus norima ištestuoti:

3. Kiekvienam testui parenkama kokius duomenis ir veiksmus norima panaudoti. Pasirinkti duomenys pažymimi lentelėje (pilka spalva):

Testas 1

Veiksmai/ duota	Duomenys 1	Duomenys 2	Duomenys 3	Duomenys 4
Veiksmas 1	Rezultatai 1	Rezultatai 2	Rezultatai 3	Rezultatai 4
Veiksmas 2	Rezultatai 5	Rezultatai 6	Rezultatai 6	Rezultatai 7
Veiksmas 3	Rezultatai 8	Rezultatai 9	Rezultatai 10	Rezultatai 11
Veiksmas 4	Rezultatai 12	Rezultatai 13	Rezultatai 14	Rezultatai 15

Taip gaunama vieno testo padengimo matrica. Tokiu būdu galima gauti ir daugiau testų.

Testas 2

Veiksmai/ duota	Duomenys 1	Duomenys 2	Duomenys 3	Duomenys 4
Veiksmas 1	Rezultatai 1	Rezultatai 2	Rezultatai 3	Rezultatai 4
Veiksmas 2	Rezultatai 5	Rezultatai 6	Rezultatai 6	Rezultatai 7
Veiksmas 3	Rezultatai 8	Rezultatai 9	Rezultatai 10	Rezultatai 11
Veiksmas 4	Rezultatai 12	Rezultatai 13	Rezultatai 14	Rezultatai 15

Apjungus visas šias lenteles, galima pamatyti bendrą duomenų ir veiksmų padengimą:

Bendras padengimas – tamsiausiomis spalvomis pažymėti dažniausiai naudoti deriniai, šviesesniais atspalviais – rečiausiai. Balta – derinys nenaudotas nė karto.

Veiksmai/ duota	Duomenys 1	Duomenys 2	Duomenys 3	Duomenys 4
Veiksmas 1	Rezultatai 1	Rezultatai 2	Rezultatai 3	Rezultatai 4
Veiksmas 2	Rezultatai 5	Rezultatai 6	Rezultatai 6	Rezultatai 7
Veiksmas 3	Rezultatai 8	Rezultatai 9	Rezultatai 10	Rezultatai 11
Veiksmas 4	Rezultatai 12	Rezultatai 13	Rezultatai 14	Rezultatai 15

Bendroje lentelėje sudedami visi stulpeliai ir visos eilutės iš atskirų testų lentelių, suskaičiuojama, kurie deriniai dažniausiai kartojasi ir pagal dažnumą nuspalvinami atitinkami langeliai: tamsiausia spalva – dažniausiai pasikartojantys deriniai, šviesiausia spalva – rečiausiai. Toks bendro testų padengimo pamatymas gali būti labai naudingas, kadangi žmogaus smegenys geriau supranta vaizdinę informaciją, nei tekstą. Šiuo būdu gali greičiau išryškėti, kad tam tikros vietos turi per daug testų, o kitos yra visai nepadengtos.

PASTABA, šiuo atveju apvedus visus užtušuotus langelius, turi būti gaunamas taisyklingas stačiakampis, nes jei parinktas bent vienas veiksmas – jis galioja visoms teste naudojamoms sąlygoms, ir atvirkščiai – jei parinkta bent viena sąlyga, ji galioja visiems teste parinktiems veiksmams.

Kitas galimas reikalavimų aprašymo būdas, iš kurio būtų galima tiesiogiai generuoti *Gherkin* kodą:

Testas	Given	When	Then
1	Sąlyga 1	Veiksmas 1, Veiksmas 2	Resultatas 1
2	Sąlyga 1, Sąlyga 2	Veiksmas 3	Resultatas 2
3	Sąlyga 3	Veiksmas 3, Veiksmas 4	Resultatas 3

Galutinio rezultato pavyzdys pagal paskutinės lentelės pirmo testo aprašymą:

GIVEN Sąlyga 1

WHEN Veiksmas 1 AND Veiksmas 2

THEN Rezultatas 1

Galutinio laukiamo rezultato realus pavyzdys:

GIVEN klientas turi galiojančią bako kortelę AND jo sąskaitoje yra 100 eurų WHEN jis įdeda savo banko kortelę į bakomatą AND įveda teisingą PIN kodą AND nurodo, kad nori išsiimti 45 eurus THEN bakomatas jam išduoda 45 eurus AND jo banko sąskaitoje liko 55 eurai AND bakomatas grąžino banko kortelę.

Šis sprendimas yra geresnis už jau egzistuojantį – *Hexawise* matricą, nes čia duomenys ir veiksmai su jais yra visiškai atskiriami, todėl tas pats komponentas negali viename teste būti interpretuojamas kaip *given*, o kitame teste kaip *when*.

2.2. Reikalavimų apibendrinimas

Reikalavimų analizės metu buvo sukurti keturi algoritmai, kurie generuoja *Gherkin* kodą iš šių reikalavimų struktūrų: tekstinis formatas, sekų diagramos, veiklos diagramos, lentelės. Kadangi yra daug variantų, kaip naudojant šias struktūras gali būti pateikti reikalavimai, kiekvienai struktūrai buvo iškelti apribojimai:

- **Tekstinis formatas** – *Gherkin* kodas generuojamas tik iš darbo istorijų.
- **Sekų diagramos** – diagramoje turi būti tik du veikėjai – vartotojas ir sistema. Taip pat algoritmas veiks tik su šiais operatoriais: *alt* (alternatyva), *opt* (neprivalomas), *par* (paralelinis).
- **Veiklos diagramos** – buvo apibrėžta, kad algoritmas veiks tik tuo atveju, jei veiklos diagrama turės tik tam tikrus, dažniausiai naudojamus, elementus.
- **Lentelės** – buvo pateikti konkretūs nurodymai kaip turi būti užpildytos lentelės, kad algoritmas atpažintų *Gherkin* testo dalis.

3. EKSPERIMENTINIS *GHERKIN* GENERAVIMO IŠ REIKALAVIMŲ METODIKOS TYRIMAS

3.1. Eksperimento planas

Eksperimento metu planuojama atlikti du tyrimus:

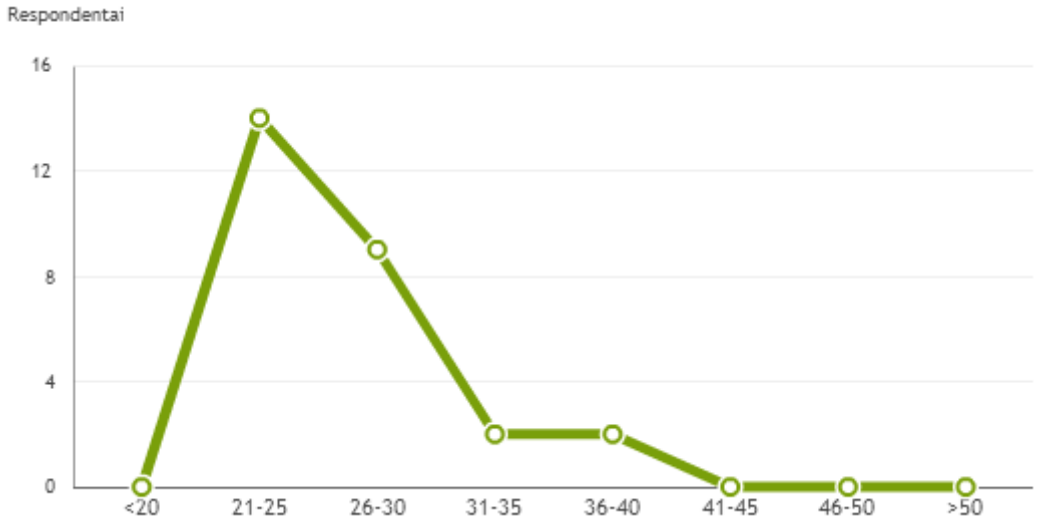
1. Atlikti, žmonių, dirbančių IT srityje apklausą. Apklausos tikslas išsiaiškinti:
    - Informaciją apie atsakantį žmogų: amžius, lytis, profesija, patirtis IT srityje.
    - Informacija apie reikalavimus atsakančiojo darbe: formatas, naudojami įrankiai reikalavimams, pasitenkinimas reikalavimų išsamumu.
    - Respondentų žinios ir patirtis su *Gherkin*.
    - Respondentų nuomonė apie *Gherkin*: suprantamumas, naudingumas atsakančiojo darbe, sudėtingumas rašant.
    - Poreikis automatiškai generuoti testus.
    - Kaip apklausiamieji įsivaizduoja *Gherkin* kodą, duotiems reikalavimų pavyzdžiams.
2. Ištirti, kaip sukurtas sekų diagramos algoritmas generuoja *Gherkin* kodą esant įvairioms sekų diagramų struktūroms. Patikrinti atvejus, kai diagrama turi skirtingus leistinus operatorius: *alt* (alternatyva), *opt* (neprivalomas), *par* (paralelinis).
3. Ištirti, kaip sukurtas veiklos diagramos algoritmas generuoja *Gherkin* kodą esant įvairioms veiklos diagramų struktūroms:
    - Lygiagretūs keliai, esantys vienas kitame.
    - Lygiagretūs keliai, kurie taip ir nesusijungia (tik išsišakojimo elementas).
    - Keli sprendimai vienoje diagramoje.
    - Diagrama, turinti objektų.

3.2. Eksperimento rezultatai

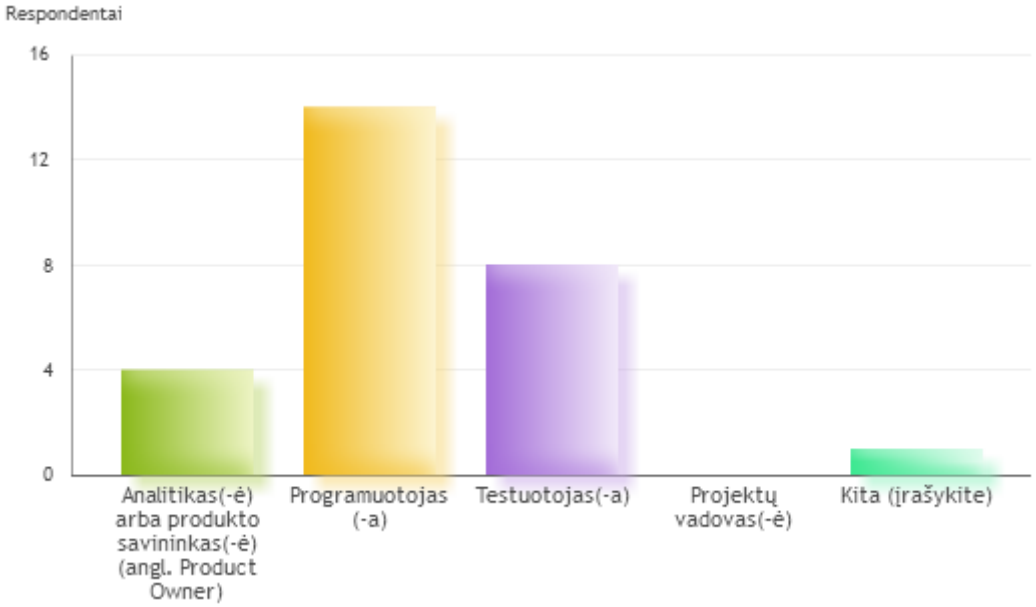
3.2.1. Apklausos rezultatai

Apklausa buvo išplatinta tarp IT srityje dirbančių žmonių ir buvo paprašyta atsakyti į priede pateiktus klausimus.

Anketą užpildė 27-i žmonės, iš jų 26-i nurodė, kad turi patirties IT srityje. Daugiausia atsakymų pateikė 21-25-erių metų žmonės, iš kurių 70,4 % buvo vyrai, o 29,6 % - moterys.

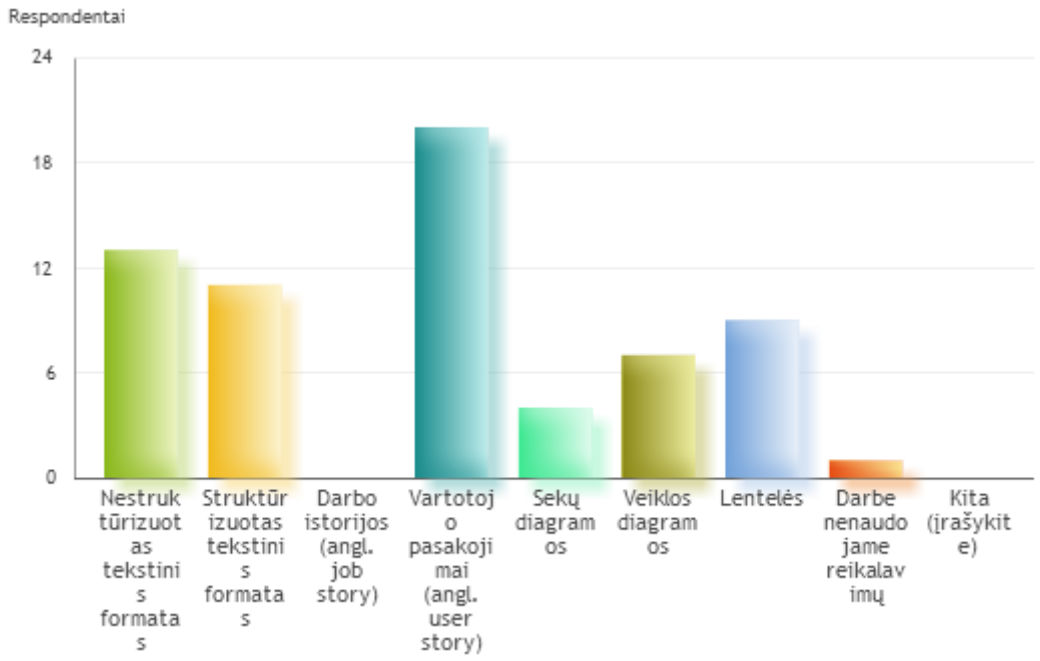


36 pav. Respondentų pasiskirtymas pagal amžių



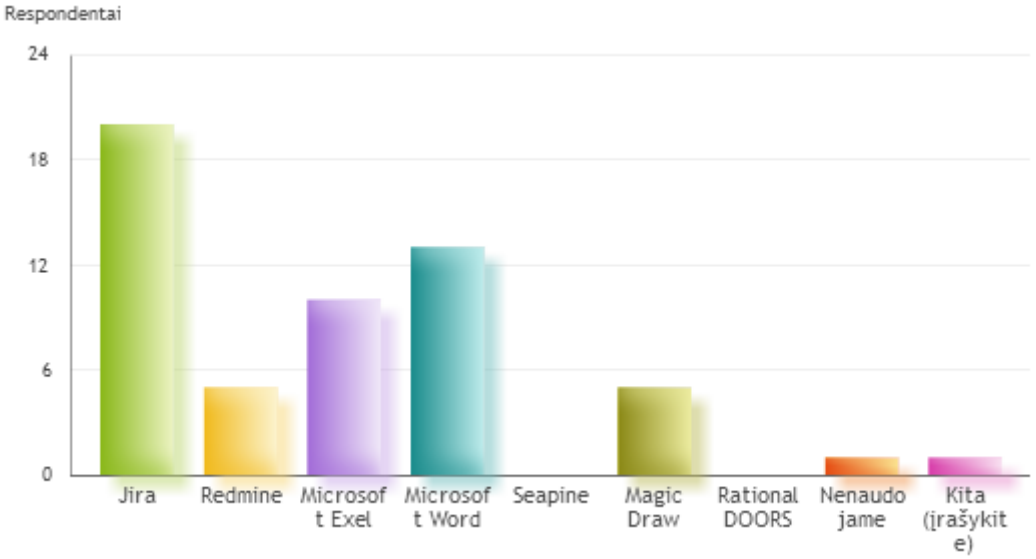
37 pav. Respondentų pareigos darbe

Daugiausia atsakiusiųjų buvo programuotojai – 51.9 %. Testuotojų buvo 29.6 % iš visų atsakiusiųjų, o analitikų – 14.8 %. Taip pat vienas respondentas buvo iš klientų aptarnavimo srities.



38 pav. Reikalavimų pateikimo forma respondentų darbe

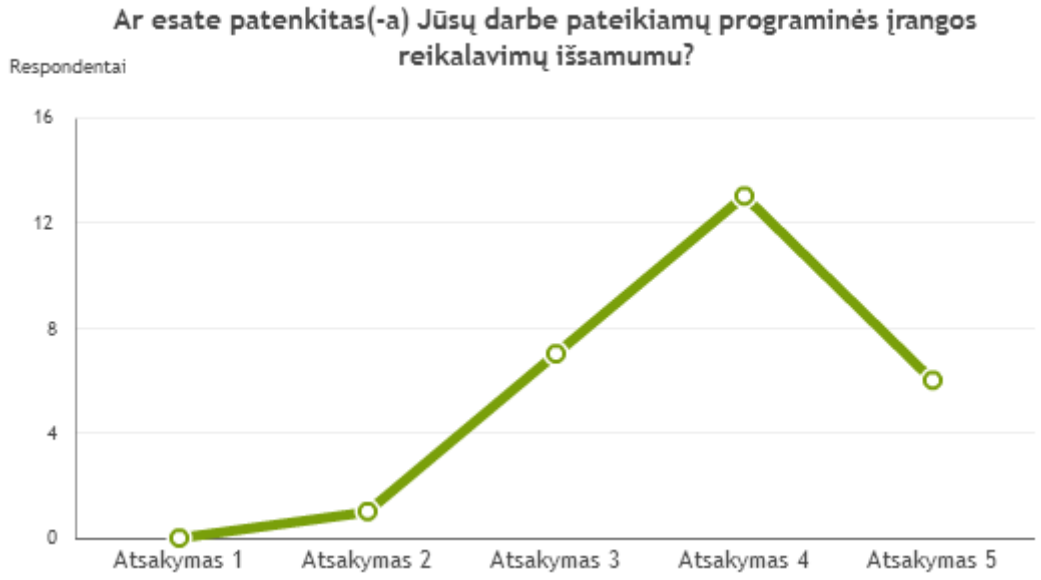
Apklausos rezultatai parodė, kad dažniausiai respondentų darbe naudojama reikalavimų forma yra vartotojo pasakojimai. Taip atsakė, net 30.8 % apklausiamųjų. Nestruktūrizuotą tekstinį formatą pasirinko 20% respondentų, o struktūrizuotą kiek mažiau – 16,9 % respondentų. Taip pat buvo įvardintos lentelės – 13,8 %, veiklos diagramos – 10,8 % ir sekų diagramos – 6,2 %. Darbo istorijos nebuvo pasirinktos.



39 pav. Reikalavimo pateikimo įrankiai respondentų darbe

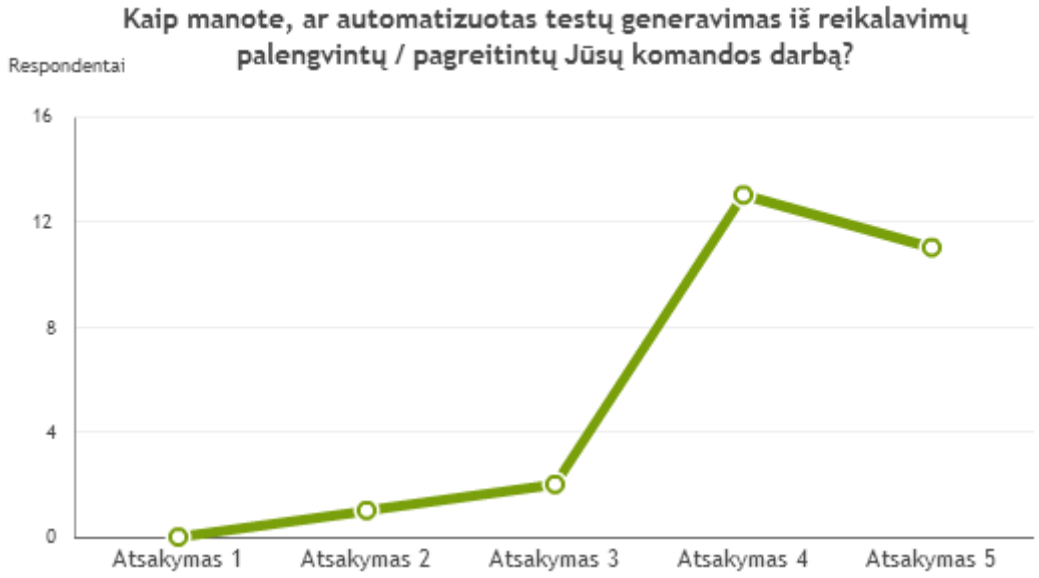
Apklausos rezultatai parodė, kad respondentų darbe reikalavimai dažniausiai pateikiami „Jira“ įrankyje. Šį įrankį pasirinko 36,4 % respondentų, Taip pat populiarūs buvo „Microsoft Word“ – 23,6 % ir „Microsoft Excel“ – 18,2% įrankiai. Ne tokie populiarūs buvo „Redmine“ – 9,1% ir „Magic

Draw“ – 9,1 %. Po vieną atsakymą surinko „Leankit“ įrankis ir variantas, kai reikalavimai visai nenaudojami. Reikėtų atkreipti dėmesį, kad šiam klausimui buvo galima pasirinkti daugiau nei vieną atsakymą.



40 pav. Respondentų pasitenkinimas reikalavimų išsamumu darbe

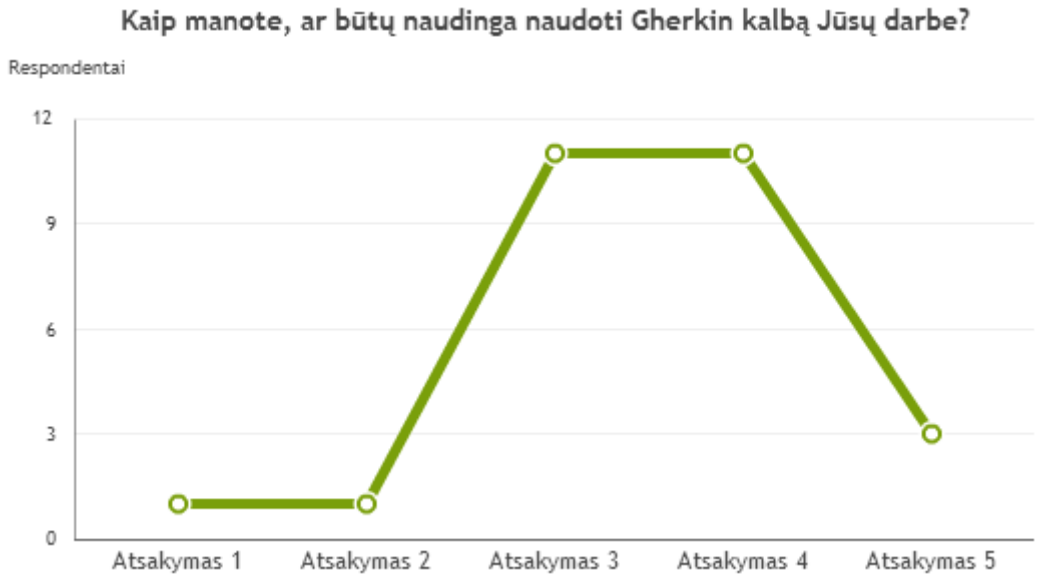
Į klausimą „Ar esate patenkintas(-a) Jūsų darbe pateikiamų programinės įrangos reikalavimų išsamumu?“ dauguma respondentų pasirinko 4 balus iš 5. Bendras vidurkis – 3,89. 5 šiuo atveju rodė didžiausią pasitenkinimą, o 1 – mažiausią.



41 pav. Respondentų nuomonė apie testų generavimo naudingumą

Į klausimą, ar automatizuotas testų generavimas iš reikalavimų palengvintų darbą, daugiausia buvo pasirinkta 4 ir 5 balai. Bendras vidurkis – 4,26. 5 balai reiškia, kad generavimas būtų labai naudingas.

Paklausus ar yra girdėję, ar naudoję *Gherkin* kalbą, dauguma respondentų atsakė neigiamai. Nepaisant to, net 64,3 % nurodė, kad *Gherkin* kalba yra labai suprantama.



Į klausimą „Ar būtų naudinga naudoti *Gherkin* kalbą Jūsų darbe?“, dauguma rinkosi 3 -is ir 4-is balus kai 5-i balai rodė labai didelį naudingumą. Bendras vidurkis – 3,52.

Atsakant į klausimą "Ar būtų sudėtinga parašyti *Gherkin* testus, pagal Jūsų darbe naudojamą reikalavimų aprašymo formą(-as)?“, dauguma pasirinko 3-is balus iš 5-ių galimų, o bendras vidurkis buvo 2,89 balai.

Paprašius pagal pateiktas diagramas parašyti *Gherkin* kodą, pabandė tik keli respondentai. Kai kurie nurodė, kad rašyti kodą atrodo sudėtinga. Tačiau beveik visi atsakiusieji kodą parašė teisingai, kas rodo, kad yra gana paprasta perprasti *Gherkin* kalbą.

3.2.2. Generavimo iš veiklos diagramų rezultatai

Atliekant analizę, algoritmas buvo išbandytas ant kelių skirtingų veiklos diagramų. Visi bandymo žingsniai pateikiami prieduose:

- 6.1. priedas – algoritmas taikomas veiklos diagramai, kurioje yra objekto elementų.
- 6.2. priedas - algoritmas taikomas veiklos diagramai, kurioje yra kelių lygių lygiagretūs keliai.
- 6.3. priedas – algoritmas taikomas veiklos diagramai, kurioje yra kelių lygių sprendimai.

3.2.3. Generavimo iš sekų diagramų rezultatai

Atliekant analizę, algoritmas buvo išbandytas ant dviejų skirtingų sekų diagramų. Visi bandymo žingsniai pateikiami prieduose:

- 6.4. priedas – algoritmas taikomas sekų diagramai, kurioje yra operatorius “Opt”.

- 6.5. priedas – algoritmas taikomas sekų diagramai, kurioje yra operatorius “Par”.

3.3. Generavimo į *Gherkin* kalbą algoritmų veikimo ir savybių analizė, kokybės kriterijų įvertinimas

3.3.1. Apklausos analizė

Pagal gautus anketos atsakymus galima daryti išvadą, kad apklaustųjų daugiausiai naudojamos reikalavimų formos yra vartotojo pasakojimai, bei tekstinis formatas – struktūrizuotas ir nestruktūrizuotas. Sekų ir veiklos diagramos nebuvo tokios populiarios, tačiau rezultatai rodo, kad kartais yra naudojamos.

Populiariausi reikalavimų pateikimo įrankiai buvo „Jira“, „MS Word“ ir „MS Excel“. Šie rezultatai koreliuoja su atsakymais apie reikalavimų formą ir galima daryti išvadą, kad dažniausiai reikalavimai pateikiami tekstiniu formatu.

Nors dauguma respondentų yra patenkinti reikalavimų išsamumu savo darbe – vidurkis apie 4 balai iš 5-ių galimų – atsakymai rodo, kad automatizuotas testų generavimas iš reikalavimų palengvintų komandos darbą.

Dauguma respondentų iki šiol nebuvo girdėję apie *Gherkin* kalbą, tačiau, perskaitę pateiktą pavyzdį, nurodė, kad ši kalba jiems yra suprantama. Vis tik ne visiems ši kalba būtų naudinga jų darbe – naudingumas buvo įvertintas vidutiniškai – 3,52 balai iš 5-ių galimų, kur 5 reiškia labai didelį naudingumą. Galbūt tai lėmė respondentų darbe naudojamų reikalavimų forma, nes paklausus, ar būtų sudėtinga parašyti *Gherkin* testus pagal respondentų darbe naudojamus reikalavimus, dauguma rinkosi tris balus iš penkių galimų. Bendras vidurkis – 2,89 balai.

Paprašius parašyti testus pagal pateiktus reikalavimų pavyzdžius, dauguma respondentų atsisakė tai padaryti. Kai kurie argumentavo, kad pateikta diagrama atrodo sudėtinga. Vis tik tie, kurie pabandė parašyti *Gherkin* testus, teisingai įvertino diagramų elementus ir diagramose pateiktų įvykių eigą bei prasmę. Šie rezultatai parodo, kad šiame darbe sukurti algoritmai teisingai įvertina diagramas ir sukuria *Gherkin* testus taip, lyg juos parašytų žmogus.

3.3.2. Generavimo algoritmų analizė

Veiklos diagramos algoritmas:

1. Svarbiausios algoritmo vietos yra pirmi du žingsniai – lygiagrečių kelių ir sprendimų tikrinimai. Šios vietos esminės, nes nuo jų priklauso kaip bus sudėliotas testo kelias. Norint gauti teisingą kelią, reikia laikytis reikalavimų, iškeltų veiklos diagramai iš kurios bus generuojamas *Gherkin* kodas.
2. Trečiame žingsnyje, veiksmų ir objektų elementus algoritmas interpretuoja teisingai.
3. Ketvirtas žingsnis įvykdomas teisingai.

4. Jei veiklos diagramoje yra daugiau nei vienas sprendimo elementas, o sprendimai eina vienas po kito tame pačiame kelyje (žr. Priedas 3), tuomet algoritmas sugeneruoja papildomų kelių.

Sekų diagramos algoritmas:

1. Nors sugeneruotas testas yra geras, jis nepadengia atvejo, kai operatoriaus sąlyga nėra tenkinama. Tam turėtų būti sukurtas atskiras testas. Galima daryti išvadą, kad operatoriai „Alt“ ir „Opt“, nors ir yra panašus, tačiau jiems netinka tas pats algoritmas.
2. Atlikus algoritmo veiksmus, gautas sekų diagramos logiką atitinkantis testas, todėl priimta išvada, kad sekų diagramai su operatoriumi „Par“ algoritmas veikia be klaidų.

3.4. Sprendimo taikymo rekomendacijos

Įvertinus anketos duomenis buvo nustatyta, kad dažniausiai naudojami reikalavimai, pateikiami tekstiniu formatu, o šio formato populiariausia forma buvo vartotojo pasakojimai. Norint naudotis šiame darbe sukurtais algoritmais, rekomenduojama vietoj vartotojo pasakojimų naudoti darbo istorijas, kurios turi panašią struktūrą, tačiau labiau aprašo situaciją ir veiksmus, o ne informaciją apie vartotoją.

Taip pat rekomenduojama aktualiausius sistemos scenarijus pavaizduoti diagramų pavidalu (sekų arba veiklos diagrama), o tuomet, pritaikius šiame darbe sukurtus algoritmus, sugeneruoti *Gherkin* kodą.

4. REZULTATŲ APIBENDRINIMAS IR IŠVADOS

1. Atlikus analizę paaiškėjo, kad *Gherkin* kalba sukurta taip, kad būtų suprantama tiek programuojantiems, tiek neprogramuojantiems žmonės. Ji labai naudinga aprašant sistemos veikimo scenarijus. Šie scenarijai gali būti naudingi kuriant sistemos reikalavimus, kai sistema dar nerealizuota, arba tikrinant sistemos veikimą po jos realizacijos.
2. Atlikus esamų sprendimų analizę, paaiškėjo, kad egzistuoja tik du būdai generuoti į *Gherkin* kalbą ir tik vienas jų generuoja iš reikalavimų, tai – *Hexawise* matrica. Deja, šis būdas nėra universalus ir tinka tik specifiniais atvejais.
3. Atlikus analizę buvo nuspręsta kurti *Gherkin* kalbos generavimo algoritmus šioms reikalavimų struktūroms:
    - Darbo istorijoms
    - Sekų diagramoms
    - Veiklos diagramoms
    - Lentelėms
4. Sukurtas algoritmas, generuojantis darbo istorijas į *Gherkin* kalba parašytus testus. Šis algoritmas pagrįstas raktinių žodžių paieška ir pakeitimu į *Gherkin* kalbos raktinius žodžius. Raktiniai žodžiai pateikiami transformacijų lentelėje, kuri reikalui esant gali būti pildoma ir atnaujinama.
5. Sukurtas algoritmas, generuojantis *Gherkin* kalba parašytus testus iš sekų diagramos. Šis algoritmas gali būti taikomas tik toms sekų diagramoms, kurios turi tik du veikėjus – vartotoją ir sistemą. Sukurtas algoritmas remiasi vartotojo pažymėtomis sritimis, taip atskirdamas, kurią diagramos dalį, kuriam *Gherkin* kalbos testui priskirti.
6. Sukurtas algoritmas, generuojantis *Gherkin* kalba parašytus testus iš veiklos diagramos. Šis algoritmas pirmiausia sugeneruoja veiklos diagramos kelius, kurie atitiks skirtingus *Gherkin* testus, o tuomet kiekvienam keliui sugeneruoja kodą pagal diagramos elementus.
7. Patobulintas *Hexawise* matricos sprendimas ir pasiūlyta, kaip patogiau generuoti *Gherkin* testus iš lentelių.
8. Sukurti algoritmai pateikia esminę generavimo į *Gherkin* kalbą idėją, tačiau gali būti taikomi tik toms diagramoms, kurios naudoja tik populiariausius elementus arba turi tik du veikėjus (sekų diagramos atveju). Ateity galima patobulinti algoritmus, praplečiant jų galimybes.
9. Eksperimento rezultatai rodo, kad automatinis testų generavimas iš reikalavimų palengvintų informacinių sistemų kūrimo komandos darbą.

5. NAUDOTOS LITERATŪROS SĄRAŠAS

[1] M. A. H. WYNNE, The Cucumber Book, 2012.

[2] „Jira software," Atlassian, [Tinkle]. Žiūrėta: https://www.atlassian.com/software/jira. [Kreiptasi 25 04 2016].

[3] „Specification-based Testing," [Tinkle]. Žiūrėta: https://www.cs.drexel.edu/~spiros/teaching/CS576/slides/10.spec-testing.pdf. [Kreiptasi 2014].

[4] R. Black ir J. Mitshell, Advanced Software Testing, Santa Barbara, CA, 2011.

[5] W. E. Levis, Software Testing and Continuous Quality Improvement, 3 mont., 2009.

[6] scrumalliance.org, 06 2015. [Tinkle]. Žiūrėta: https://www.scrumalliance.org/community/articles/2012/january/agile-testing-key-points-for-unlearning.

[7] S. W. Ambler, 06 2015. [Tinkle]. Žiūrėta: http://www.ambysoft.com/essays/agileTesting.html.

[8] J. Offutt ir A. Abdurazik, Generating Tests from UML Specifications, 1999.

[9] E. Bernard ir B. Legeard, Requirements Traceability in the Model-Based Testing, 2004.

[10] „cucumber.io," [Tinkle]. Žiūrėta: https://cucumber.io/docs/reference#gherkin. [Kreiptasi 15 04 2016].

[11] Versionone, „What is the Agile Methodology?," Versionone, [Tinkle]. Žiūrėta: https://www.versionone.com/agile-101/agile-methodologies/. [Kreiptasi 15 11 2015].

[12] R. C. Martin, Agile software development.

[13] „dsdm.org," 01 2016. [Tinkle]. Žiūrėta: http://www.dsdm.org/content/what-dsdm.

[14] M. J. H. Fowler, The Agile Manifesto, 2001.

[15] „quora.com," [Tinkle]. Žiūrėta: https://www.quora.com/Where-can-I-find-examples-of-good-agile-requirements-documents. [Kreiptasi 12 2015].

[16] „Do Agile Right - Lessons Learned from an Atlassian Product Manager," youtube.com, [Tinkle]. Žiūrėta: www.youtube.com/watch?v=aT5HPU1IRi0. [Kreiptasi 07 11 2015].

[17] „mountaingoatsoftware.com," [Tinkle]. Žiūrėta: https://www.mountaingoatsoftware.com/agile/user-stories. [Kreiptasi 09 2015].

[18] „WebSequenceDiagrams," [Tinkle]. Žiūrėta: websequencediagrams.com. [Kreiptasi 07 04 2016].

[19] IBM, „ibm.com," IBM, [Tinkle]. Žiūrėta: www.ibm.com/developerworks/rational/library/3101.html.

[Kreiptasi 12 04 2016].

[20] M. Ericsson, „Activity Diagrams: What They Are and How to Use,“ IBM, 2014.

[21] Lucidchart, „UML - Activity Diagram Symbols' Meaning,“ [Tinkle]. Žiūrėta: https://www.lucidchart.com/pages/uml-activity-diagram-symbols-meaning. [Kreiptasi 16 04 2016].

[22] „SysML Open Source Specification Project,“ SysML, [Tinkle]. Žiūrėta: http://sysml.org/. [Kreiptasi 05 04 2016].

[23] „Introducing the Hexawise Coverage Matrix!,“ Hexawise , 29 01 2015. [Tinkle]. Žiūrėta: https://hexawise.com/posts/introducing-the-hexawise-coverage-matrix. [Kreiptasi 22 01 2016].

[24] „Automatic Generation of Cucumber from Code,“ 24 12 2011. [Tinkle]. Žiūrėta: http://www.writemoretests.com/2011/12/torturing-your-code-until-it-gives-up.html. [Kreiptasi 20 02 2016].

[25] T. Kalapun, 19 02 2013. [Tinkle]. Žiūrėta: http://kalapun.com/portfolio/tmp/gherkin/. [Kreiptasi 05 04 2016].

[26] B. Meacham, „Humble Programmers’ Reflections by Example on Unit Tests, TDD and BDD,“ 10 2015. [Tinkle]. Žiūrėta: http://www.infoq.com/presentations/unit-test-specflow-cucumber?utm_source=infoqEmail&utm_medium=WeeklyNL_EditorialContentDevelopment&utm_campaign=10282014news#downloadPdf.

[27] „specflow.org,“ SpecFlow, [Tinkle]. Žiūrėta: http://www.specflow.org/. [Kreiptasi 20 04 2016].

[28] Microsoft, 12 2014. [Tinkle]. Žiūrėta: http://msdn.microsoft.com/en-us/library/jj635157.aspx.

[29] „Affordable Advanced Requirements Management Software,“ 12 2014. [Tinkle]. Žiūrėta: http://www.analysttool.com/features/index/.

[30] IBM, 01 2015. [Tinkle]. Žiūrėta: http://www-03.ibm.com/software/products/en/ratidoor.

[31] „Seapine Software,“ 01 2015. [Tinkle]. Žiūrėta: http://www.seapine.com/.

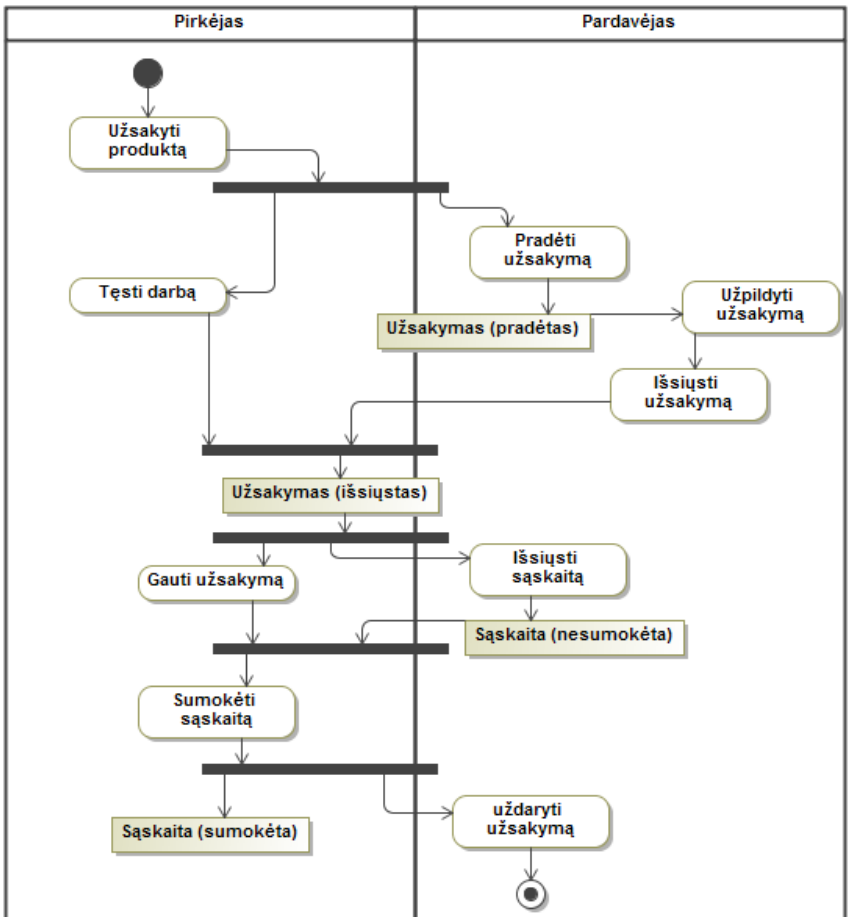
[32] visualstudio, 05 2015. [Tinkle]. Žiūrėta: https://www.visualstudio.com/en-us/features/testing-tools-vs.aspx.

[33] A. Klement, „Replacing The User Story With The Job Story,“ 12 11 2013. [Tinkle]. Žiūrėta: https://jtbd.info/replacing-the-user-story-with-the-job-story-af7cdee10c27#.u2dqj166m. [Kreiptasi 10 02 2016].

6. PRIEDAI

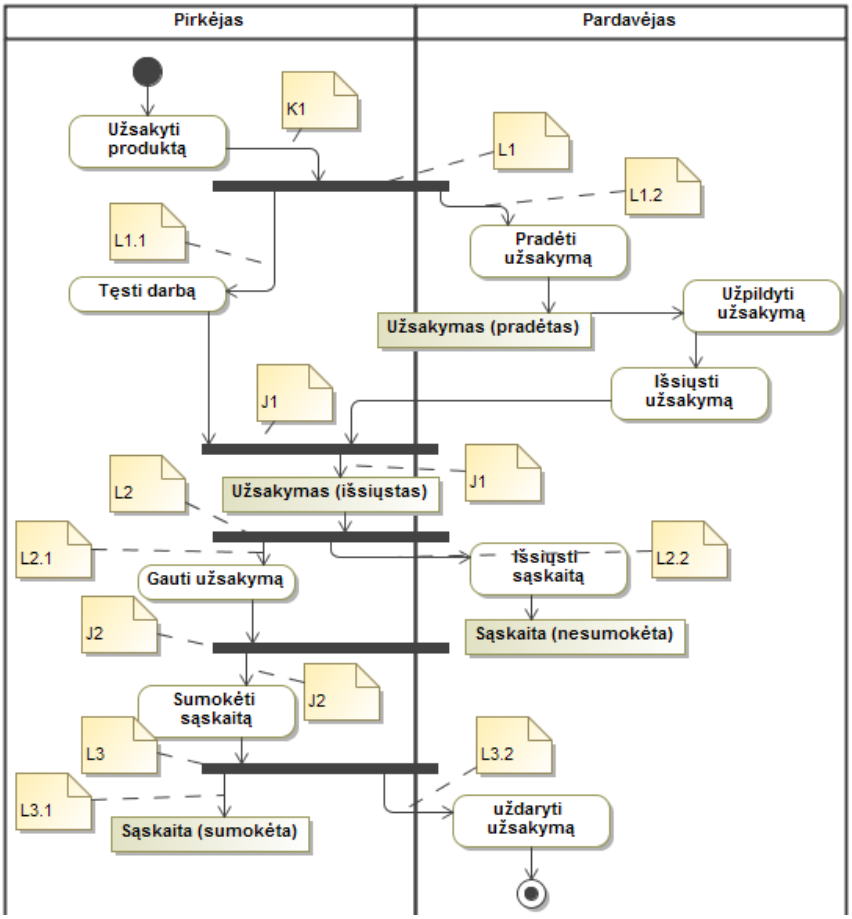
6.1. priedas. Veiklos diagramos generavimas į *Gherkin* testą 1

Turima tokia veiklos diagrama:



Pateikus šią diagramą algoritmui, pirmiausia atliekamas **lygiagrečių kelių tikrinimo algoritmas**.

1. Pradinis diagramos kelias pažymimas K1 numeriu.
2. Randami ir sunumeruojami išsišakojimo elementai: L1, L2, L3.
3. Sunumeruojami iš šio elemento išeinantys keliai: L1.1, L1.2, L2.1, L2.2, L3.1, L3.2.
4. Randami sujungimo elementai ir pažymimi: J1, J2.
5. Iš sujungimo elementų išeinantys keliai pažymimi: J1, J2.



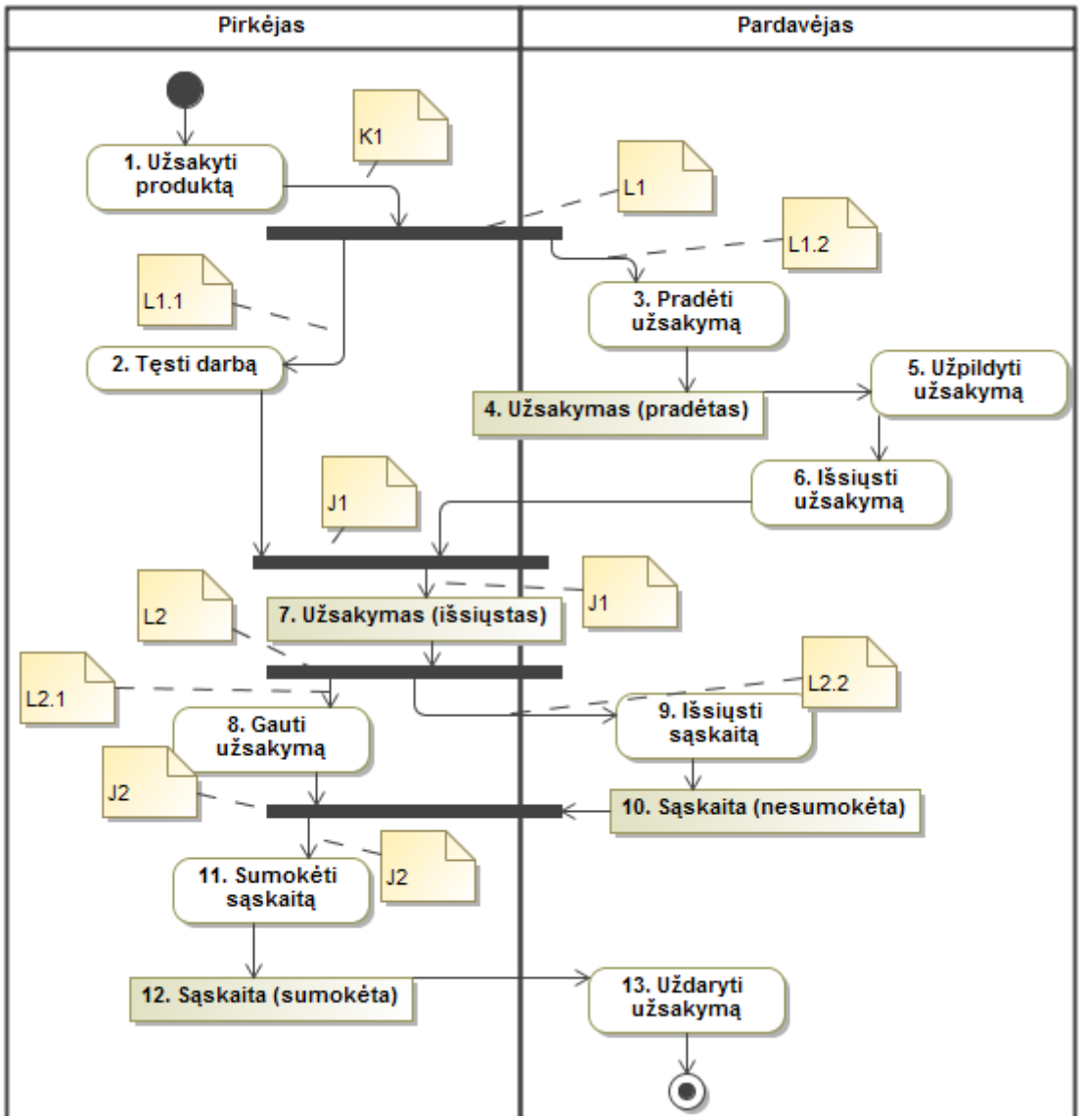
6. Šiai diagramai pagal pažymėtus kelius parašomi testo keliai:

K1, L1.1, L1.2, J1, L2.1, J2, L3.1

K1, L1.1, L1.2, J1, L2.1, J2, L3.2

K1, L1.1, L1.2, J1, L2.2

Iš gautų kelių galima pastebėti, kad algoritmas blogai interpretavo diagramą. Taip nutiko todėl, kad nors ir iš pažiūros diagrama atrodo teisinga, bet ji turi elementų, kurie niekur neveda. Algoritmas buvo sukurtas taip, kad veiktų tuo atveju, kai visi elementai yra sujungti ir kai veikla baigiasi pabaigos elemente. Kad būtų galima taikyti algoritmą, pakoreguojama diagrama. Jos sužymėti keliai atrodo taip:



Šiai diagramai sugeneruojamas toks kelias: K1, L1.1, L1.2, J1, L2.1, L2.2, J2. Šis kelias teisingai padengia visas diagramos vietas.

Patikrinama ar diagramoje nėra sprendimo elementų. Kadangi nėra, einama prie trečios algoritmo dalies – elementų transformacijos į *Gherkin* kodą:

1. Paimamas kelias K1, L1.1, L1.2, J1, L2.1, L2.2, J2.
2. Pradedamas testas. Parašomas jo pavadinimas:

Scenario: Testas1

3. Patikrinus, ar kelyje buvo sprendimų, jų nerandama.
4. Patikrinus, ar diagrama prasideda vienu objektu ar daugiau, gaunamas neigiamas atsakymas.
5. Tikrinamas diagramos elementas nr. 1:
    a. Patikrinus, ar elementas paskutinis, gaunamas neigiamas atsakymas.

b. Patikrinus, ar elementas objektas, gaunamas neigiamas atsakymas.
c. Patikrinus, ar elementas veiksmas, gaunamas teigiamas atsakymas. Veiksmas priskiriamas į WHEN:

Scenario: Testas1 **WHEN Užsakyti produktą**

6. Tikrinami diagramos elementai nr. 2 ir nr. 3. Jiems atliekami tokie patys veiksmai a, b ir c, kaip ir pirmajam elementui. Gaunama:

Scenario: Testas1 WHEN Užsakyti produktą **WHEN Tęsti darbą** **WHEN Pradėti užsakymą**

7. Tikrinamas diagramos elementas nr. 4.
   a. Patikrinus, ar elementas paskutinis, gaunamas neigiamas atsakymas.
   b. Patikrinus, ar kitas elementas objektas, gaunamas teigiamas atsakymas. Elementas priskiriamas į THEN:

Scenario: Testas1 WHEN Užsakyti produktą WHEN Tęsti darbą WHEN Pradėti užsakymą **THEN Užsakymas (pradėtas)**

8. Patikrinami diagramos elementai nr. 5 ir nr. 6. Jiems atliekami tokie patys veiksmai kaip ir diagramos elementui nr.1. Gaunama:

Scenario: Testas1 WHEN Užsakyti produktą WHEN Tęsti darbą WHEN Pradėti užsakymą THEN Užsakymas (pradėtas) **WHEN Užpildyti užsakymą** **WHEN Išsiųsti užsakymą**

9. Patikrinamas diagramos elementas nr. 7:
   a. Patikrinus, ar elementas paskutinis, gaunamas neigiamas atsakymas.

b. Patikrinus, ar elementas objektas, gaunamas teigiamas atsakymas. Elementas priskiriamas į THEN:

Scenario: Testas1
WHEN Užsakyti produktą
WHEN Tęsti darbą
WHEN Pradėti užsakymą
THEN Užsakymas (pradėtas)
WHEN Užpildyti užsakymą
WHEN Išsiųsti užsakymą
THEN Užsakymas (išsiųstas)

10. Patikrinami diagramos elementai nr. 8 ir nr. 9. Atliekami a , b ir c veiksmai kaip ir elementui nr. 1. Gaunama:

Scenario: Testas1
WHEN Užsakyti produktą
WHEN Tęsti darbą
WHEN Pradėti užsakymą
THEN Užsakymas (pradėtas)
WHEN Užpildyti užsakymą
WHEN Išsiųsti užsakymą
THEN Užsakymas (išsiųstas)
WHEN Gauti užsakymą
WHEN Išsiųsti sąskaitą

11. Patikrinamas diagramos elementas nr. 10:
    a. Patikrinus, ar elementas paskutinis, gaunamas neigiamas atsakymas.
    b. Patikrinus ar elementas objektas, gaunamas teigiamas atsakymas. Elementas priskiriamas į THEN:

Scenario: Testas1
WHEN Užsakyti produktą
WHEN Tęsti darbą
WHEN Pradėti užsakymą
THEN Užsakymas (pradėtas)
WHEN Užpildyti užsakymą

WHEN Išsiųsti užsakymą
THEN Užsakymas (išsiųstas)
WHEN Gauti užsakymą
WHEN Išsiųsti sąskaitą
THEN Sąskaita (nesumokėta)

12. Patikrinami diagramos elementas nr. 11. Atliekami a , b ir c veiksmai kaip ir elementui nr. 1. Gaunama:

Scenario: Testas1
WHEN Užsakyti produktą
WHEN Tęsti darbą
WHEN Pradėti užsakymą
THEN Užsakymas (pradėtas)
WHEN Užpildyti užsakymą
WHEN Išsiųsti užsakymą
THEN Užsakymas (išsiųstas)
WHEN Gauti užsakymą
WHEN Išsiųsti sąskaitą
THEN Sąskaita (nesumokėta)
WHEN Sumokėti sąskaitą

13. Patikrinamas diagramos elementas nr. 12:
    a. Patikrinus, ar elementas paskutinis, gaunamas neigiamas atsakymas.
    b. Patikrinus, ar elementas objektas, gaunamas teigiamas atsakymas. Elementas priskiriamas į THEN.
14. Patikrinamas diagramos elementas nr. 13.
    a. Patikrinus, ar elementas paskutinis, gaunamas teigiamas atsakymas, todėl šis elementas priskiriamas į THEN ir pabaigiamas testas:

Scenario: Testas1
WHEN Užsakyti produktą
WHEN Tęsti darbą
WHEN Pradėti užsakymą
THEN Užsakymas (pradėtas)
WHEN Užpildyti užsakymą

```
WHEN Išsiųsti užsakymą
THEN Užsakymas (išsiųstas)
WHEN Gauti užsakymą
WHEN Išsiųsti sąskaitą
THEN Sąskaita (nesumokėta)
WHEN Sumokėti sąskaitą
THEN Sąskaita (sumokėta)
THEN Uždaryti užsakymą
```

15. Kadangi daugiau nesugeneruotų kelių neliko, baigiamas trečias algoritmo žingsnis.

Pradedamas paskutinis algoritmo žingsnis - "*Gherkin* kodo tvarkymas":

1. Paimamas sugeneruotas testas.
2. Patikrinus, ar yra vietų, kur tas pats žingsnis kartojasi daugiau nei vieną kartą, randama, kad tokių vietų yra.
3. Paimami šios vietos žingsniai, išskyrus patį pirmą, ir jų pradžia pakeičiama į žodį AND:

```
Scenario: Testas1
WHEN Užsakyti produktą
AND Tęsti darbą
AND Pradėti užsakymą
THEN Užsakymas (pradėtas)
WHEN Užpildyti užsakymą
WHEN Išsiųsti užsakymą
THEN Užsakymas (išsiųstas)
WHEN Gauti užsakymą
WHEN Išsiųsti sąskaitą
THEN Sąskaita (nesumokėta)
WHEN Sumokėti sąskaitą
THEN Sąskaita (sumokėta)
THEN Uždaryti užsakymą
```

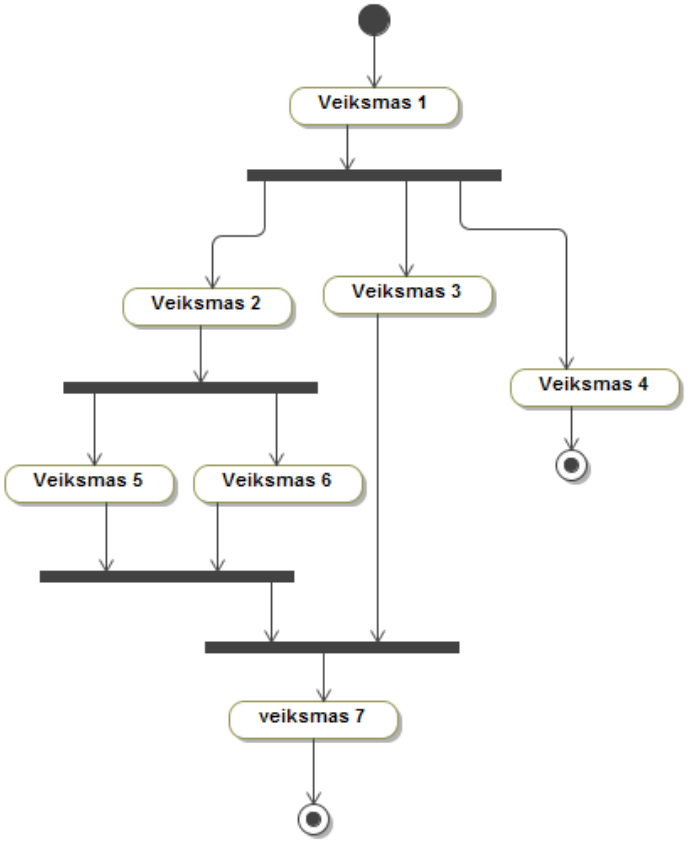
4. Šie veiksmai kartojami tol, kol nelieka nesutvarkytų vietų. Galutinis rezultatas:

```
Scenario: Testas1
```

WHEN Užsakyti produktą
AND Tęsti darbą
AND Pradėti užsakymą
THEN Užsakymas (pradėtas)
WHEN Užpildyti užsakymą
AND Išsiųsti užsakymą
THEN Užsakymas (išsiųstas)
WHEN Gauti užsakymą
AND Išsiųsti sąskaitą
THEN Sąskaita (nesumokėta)
WHEN Sumokėti sąskaitą
THEN Sąskaita (sumokėta)
AND Uždaryti užsakymą

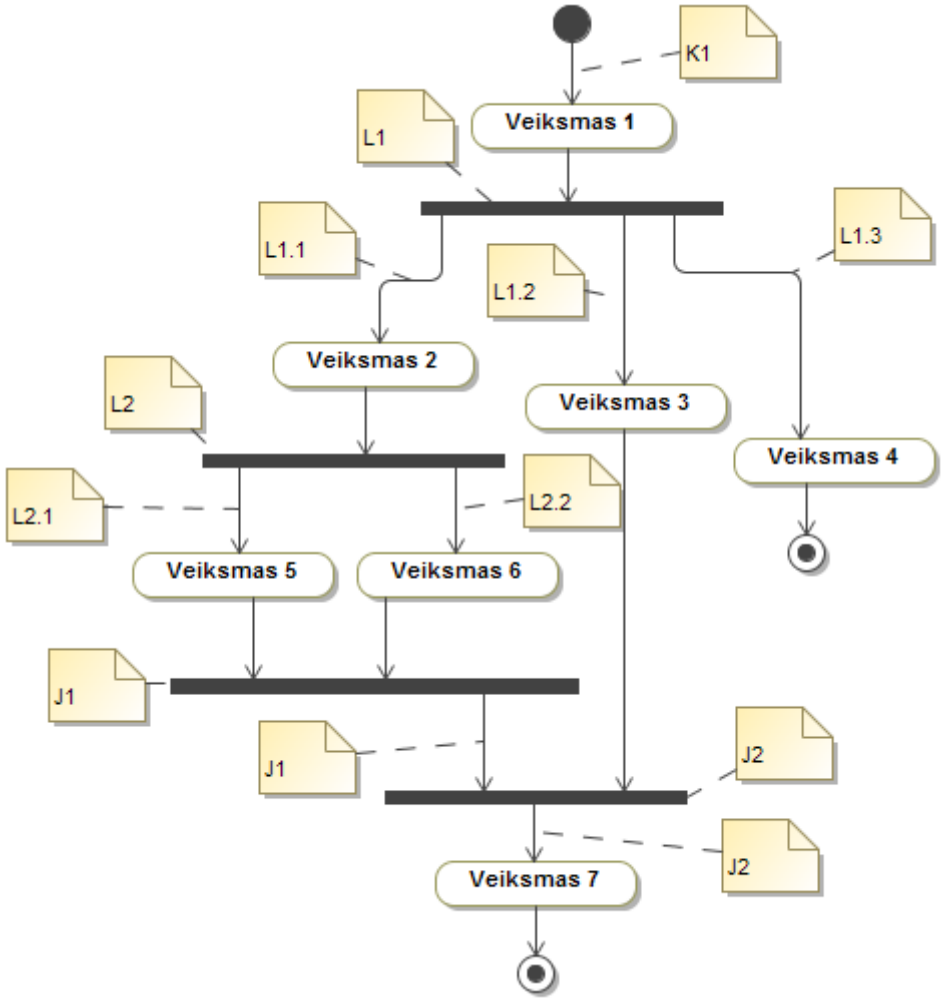
6.2. priedas. Veiklos diagramos generavimas į *Gherkin* testą 2

Turima veiklos diagrama, turinti kelis lygiagrečius kelius, bei vieną kelią, kuris atsiskiria per išsišakojimo elementą, bet nebesusijungia su kitais keliais:



Pradedamas pirmas algoritmo žingsnis – **lygiagrečių kelių tikrinimas.**

1. Pradinis diagramos kelias pažymimas K1.
2. Diagramoje randami du išsišakojimo elementai ir pažymimi L1, L2.
3. Iš šių elementų išeinantys keliai pažymimi L1.1, L1.2, L1.3, L2.1, L2.2.
4. Diagramoje randami du sujungimo elementai ir pažymimi J1,J2.
5. Iš šių elementų išeinantys keliai pažymimi J1, J2. Sužymėta diagrama:



6. Tikrinama, ar į pirmą sujungimo elementą J1 įeina iš išsišakojimo elemento išeinantys keliai. Gaunamas teigiamas atsakymas, todėl šie keliai interpretuojami kaip lygiagretūs – vienas kelias L2.1, L2.2. Prie šių kelių pridedamas sujungimo kelio numeris ir gaunama: L2.1, L2.2, J1. Prie gauto kelio pridedama jo pradžia: L1.1, L2.1, L2.2, J1.
7. Dar liko nepadengtų sujungimo elementų, todėl tikrinamas antrasis – J2. Randama, kad į jį ateina du keliai: J1 ir L1.2. Patikrinus, ar šie keliai išeina iš išsišakojimo elemento, randama, kad kelias L1.2 ir kelias L1.1, L2.1, L2.2, J1 ateina iš to pačio išsišakojimo L1, todėl šie keliai interpretuojami kaip lygiagretūs, gaunamas vienas kelias: L1.1, L2.1, L2.2, J1, L1.2. Jo pabaigoje prirašomas sujungimo elemento

numeris, o pradžioje pridedamas į juos vedančio kelio numeris, gaunama: K1, L1.1, L2.1, L2.2, J1, L1.2, J2.

8. Patikrinus randama dar niekur nepanaudotas kelias – L1.3. Šis kelias priskiriamas atskiram testui, gaunama: K1, L1.3.
9. Baigiamas pirmasis algoritmo žingsnis – lygiagrečių kelių tikrinimas.

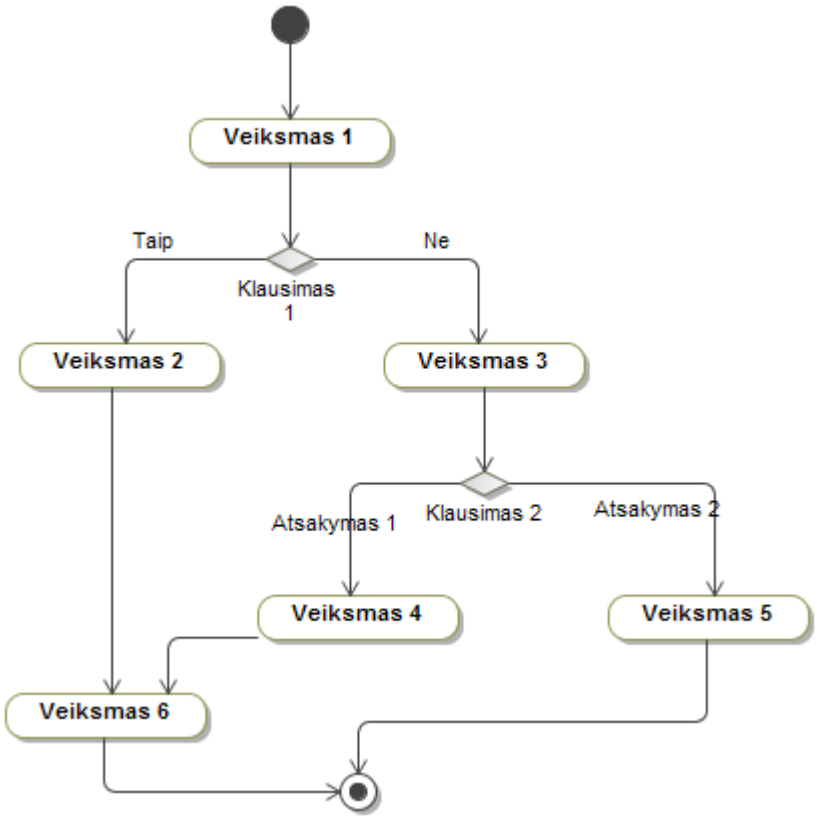
Atliekant antrą algoritmo žingsnį – „Sprendimai – paruošimas generavimui“, nebuvo rastas nė vienas sprendimo elementas, todėl liko tie patys keliai: K1, L1.1, L2.1, L2.2, J1, L1.2, J2 ir K1, L1.3.

Šie keliai, atitinka veiklos diagramos kelių logiką.

Kadangi, atlikus ankstesnius bandymus buvo išsiaiškinta, kad veiklos ir objektų elementai interpretuojami teisingai, tolimesnio algoritmo tyrimas šiai diagramai nebus atliekamas.

6.3. priedas. Veiklos diagramos generavimas į *Gherkin* testą 3

Turima veiklos diagrama, turinti kelis sprendimus:



Šiai diagramai atlikus pirmą žingsnį, lygiagrečių kelių tikrinimą, nebuvo rastas nė vienas lygiagretus kelias, todėl visa diagrama pažymėta K1 numeriu.

Atliekamas antrasis – sprendimų tikrinimo žingsnis:

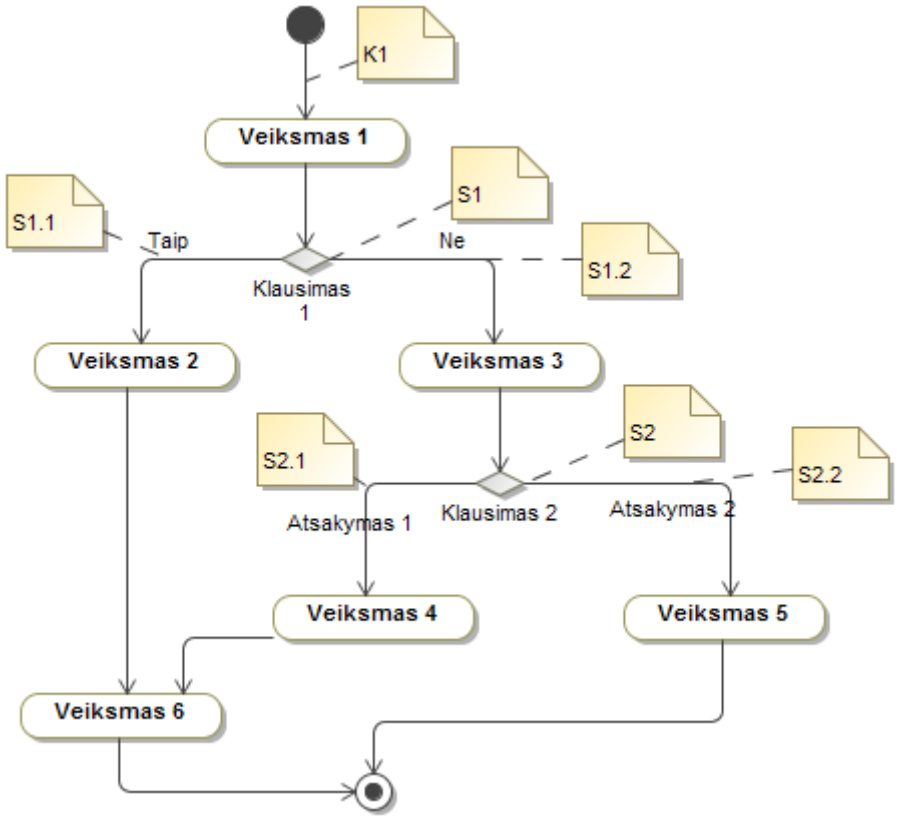
1. Diagramoje rasti du sprendimo elementai ir sunumeruoti S1 ir S2.
2. Sprendimo alternatyvos sunumeruojamos S1.1., S1.2, S2.1, S2.2.
3. Paimami jau sugeneruotų kelių numeriai: K1.
4. Keliui K1 pridedamas sprendimo elemento numeris S1, gaunama K1,S1.

5. Gautas kelias pakartojamas du kartus, nes sprendimas turi dvi alternatyvas. Kiekvienai alternatyvai prie gauto kelio pridedamas alternatyvos numeris. Gaunama: K1, S1, S1.1; K1, S1, S1.2.
6. Atliekami veiksmai su antruoju sprendimo elementu. Paimamas kelias iki šio elemento ir šio kelio pabaigoje įrašomas sprendimo elemento numeris. Gaunama: K1, S1, S1.2, S2.
7. Kadangi sprendimas S2 turi dvi alternatyvas, gautas kelias pakartojamas du kartus ir prie gautų kelių prirašomi alternatyvų numeriai, gaunama: K1, S1, S1.2, S2, S2.1 ir K1, S1, S1.2, S2, S2.2.

Po dviejų algoritmų žingsnių, gaunami keturi testai:

1. K1, S1, S1.1
2. K1, S1, S1.2
3. K1, S1, S1.2, S2, S2.1
4. K1, S1, S1.2, S2, S2.2

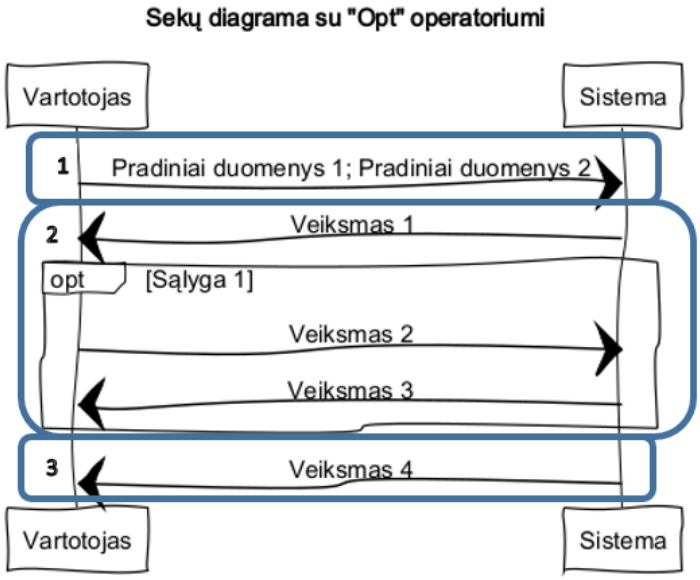
Sužymėti diagramos veiksmai:



Visi testai atitinka veiklos diagramos logiką, išskyrus antrąjį testą, kuris sutampa su trečio ir ketvirto testų pradžia. Šis testas yra perteklinis ir turėtų būti pašalintas.

6.4. priedas. Sekų diagramos generavimas į *Gherkin* testą 1

Turima sekų diagrama su operatoriumi „Opt“. Joje jau sužymėtos trys sritys:



Šios diagramos tekstinis formatas, pagal pažymėtas sritis:

```
title Sekų diagrama su "Opt" operatoriumi
1
Vartotojas->Sistema: Pradiniai duomenys 1; Pradiniai duomenys 2
2
Sistema -> Vartotojas: Veiksmas 1
opt Sąlyga 1
   Vartotojas-> Sistema: Veiksmas 2
   Sistema -> Vartotojas: Veiksmas 3
end
3
Sistema->Vartotojas: Veiksmas 4
```

Paimamas tekstas iš pirmos pažymėtos srities ir patikrinama, ar nėra operatorių. Kadangi operatorių nėra, pradedama teksto transformacija. Sakinyje randamas ženklas „;“, todėl vienas sakinys suskaldomas į du: „Pradiniai duomenys 1“, „Pradiniai duomenys 2“. Kadangi tai pirmasis pirmos srities sakinys, jam priskiriamas žodis GIVEN, pridedamas veikėjo pavadinimas. Gaunama:

GIVEN Vartotojas pradiniai duomenys 1

Kadangi dar liko neaprašytų veiksmų, kartojamas algoritmas kitai sakinio daliai, gaunama:

GIVEN Vartotojas pradiniai duomenys 1 **AND Vartotojas pradiniai duomenys 2**

Paimamas tekstas iš antros pažymėtos srities. Patikrinus randamas „Opt“ operatorius, todėl atliekamas operatorių transformacijų algoritmas. Kadangi operatorius turi tik vieną regioną, testas nėra kartojamas. Kadangi pildoma sritis nėra pirmoji, operatoriaus sąlyga priskiriama ankstesnei sričiai ir pažymima, kaip reikalaujanti peržiūros. Regiono veiksmai surašomi į pildomą sritį. Atliekamos veiksmų transformacijos šiems elementams ir gaunama:

GIVEN Vartotojas pradiniai duomenys 1 AND Vartotojas pradiniai duomenys 2 **#AND Sąlyga 1**

WHEN Sistema veiksmas 1 **AND Vartotojas veiksmas 2** **AND Sistema veiksmas 3**

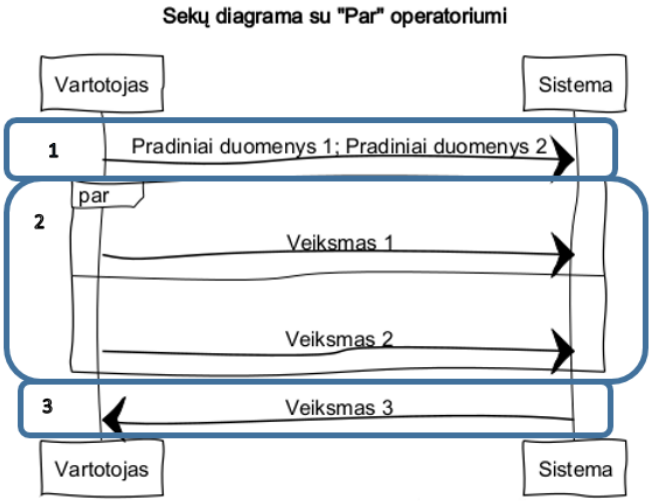
Paimamas tekstas iš trečios pažymėtos srities. Operatorių nerandama, todėl atlikus veiksmų transformacijas, gaunamas rezultatas:

GIVEN Vartotojas pradiniai duomenys 1 AND Vartotojas pradiniai duomenys 2 AND Sąlyga 1 WHEN Sistema veiksmas 1 AND Vartotojas veiksmas 2 AND Sistema veiksmas 3 **THEN Sistema veiksmas 4**

Galima pastebėti, kad, nors ir sugeneruotas testas yra geras, bet jis nepadengia atvejo, kai operatoriaus sąlyga nėra tenkinama. Tam reikėtų atskiro testo.

6.5. priedas. Sekų diagramos generavimas į *Gherkin* testą 2

Turima sekų diagrama su operatoriumi „Par“. Joje jau sužymėtos trys sritys:



Šios diagramos tekstinis formatas, pagal pažymėtas sritis:

```
title Seku diagrama su "Par" operatoriumi
1
Vartotojas->Sistema: Pradiniai duomenys 1; Pradiniai duomenys 2
2
par
   Vartotojas-> Sistema: Veiksmas 1
else
   Vartotojas-> Sistema: Veiksmas 2
end
3
Sistema->Vartotojas: Veiksmas 3
```

Paimamas tekstas iš pirmos pažymėtos srities ir patikrinama, ar nėra operatorių. Kadangi operatorių nėra, pradedama teksto transformacija. Sakinyje randamas ženklas „;“, todėl vienas sakinys suskaldomas į du: „Pradiniai duomenys 1“, „Pradiniai duomenys 2“. Kadangi tai pirmasis srities pirmos srities sakinys, jam priskiriamas žodis GIVEN, pridedamas veikėjo pavadinimas. Gaunama:

GIVEN Vartotojas pradiniai duomenys 1

Kadangi dar liko neaprašytų veiksmų, kartojamas algoritmas kitai sakinio daliai, gaunama:

GIVEN Vartotojas pradiniai duomenys 1 AND Vartotojas pradiniai duomenys 2

Paimamas tekstas iš antros pažymėtos srities. Patikrinus randamas „Par“ operatorius, todėl atliekamas operatorių transformacijų algoritmas. Šio operatoriaus veiksmai interpretuojami, kaip veiksmai einantys iš eilės. Atliekamos veiksmų transformacijos šiems elementams ir gaunama:

GIVEN Vartotojas pradiniai duomenys 1 AND Vartotojas pradiniai duomenys 2 **WHEN Vartotojas veiksmas 1** **AND Vartotojas veiksmas 2**

Paimamas tekstas iš trečios pažymėtos srities. Operatorių nerandama, todėl atlikus veiksmų transformacijas, gaunamas rezultatas:

GIVEN Vartotojas pradiniai duomenys 1 AND Vartotojas pradiniai duomenys 2 WHEN Vartotojas veiksmas 1 AND Vartotojas veiksmas 2 **THEN Sistema veiksmas 3**

Gautas testas atitinka diagramos logiką, todėl galima teigti, kad sekų diagramai su operatoriumi „Par“ algoritmas veikia be klaidų.

6.6. priedas. Apklausos anketa

Sveiki, esu KTU informacinių sistemų inžinerijos magistrantė. Savo diplominiame darbe norėčiau ištirti žmonių nuomonę apie *Gherkin* kalbą, jos suprantamumą ir poreikį automatizuoti šia kalba parašytų testų generavimą. Siekdama gauti tikslią ir objektyvią informaciją šia tema, aš Jūsų maloniai prašau atsakinėti nuoširdžiai.

Ši apklausa yra anoniminė. Atsakymai nebus išskirti atskirai. Susumuoti kiekvieno klausimo atsakymai bus skaičiuojami ir paminėti diplominiame darbe. Šią apklausą atlikti užtruks apie 5 - 10 minučių.

Dar nieko negirdėjote apie *Gherkin* kalbą? Nieko tokio! Perskaitykite šį trumpą pristatymą ir būsite pasirengę atsakyti į visus šios anketos klausimus. Trumpai apie *Gherkin*:

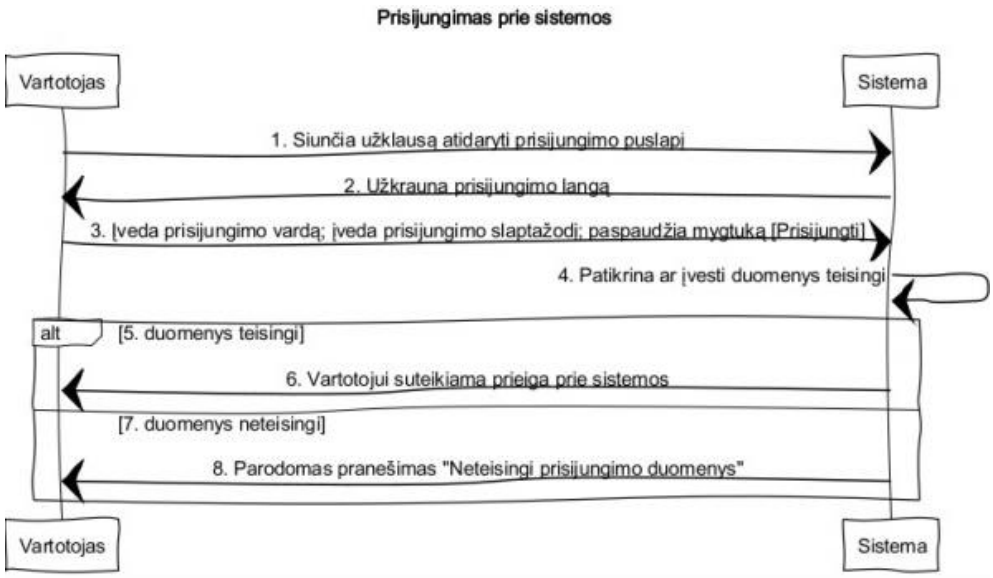
Gherkin kalba yra naudojama aprašant testinius atvejus programinės įrangos įrankyje „Cucumber“. Ši kalba buvo sukurta, norint skatinti veikla pagrįstą programavimą (angl. *Business Driven Development*) visoje kūrimo komandoje - tarp programuotojų, testuotojų, analitikų ir projektų vadovų. Šios kalbos sintaksė turi tris pagrindines dalis: duota (angl. *Given*), kuomet (angl. *When*) ir tuomet (angl. *Then*). Pavyzdžiui *Gherkin* testas bankomato funkcijai, kai klientas iš jo išsiima pinigus galėtų atrodyti taip:

```
GIVEN klientas turi galiojančią banko kortelę
AND jo sąskaitoje yra 100 eurų
WHEN jis įdeda savo banko kortelę į bankomatą
AND įveda teisingą PIN kodą
AND nurodo, kad nori išsiimti 45 eurus
THEN bankomatas jam išduoda 45 eurus
AND jo banko sąskaitoje liko 55 eurai
AND bankomatas grąžino banko kortelę.
```

1. Jūsų amžius:
    - <20
    - 21-25
    - 26-30
    - 31-35
    - 36-40
    - 41-45
    - 46-50
    - >50
2. Jūsų lytis:
    - Vyras
    - Moteris
3. Ar turite darbo patirties informacinių technologijų srityje?
    - Taip
    - Ne
4. Kokios Jūsų pareigos darbe šiuo metu?
    - Analitikas(-ė) arba produkto savininkas(-ė) (angl. *Product Owner*)
    - Programuotojas(-a)
    - Testuotojas(-a)
    - Projektų vadovas(–ė)
    - Kita (įrašykite)
5. Kokia forma Jūsų darbe aprašomi reikalavimai? (keli galimi atsakymai)
    - Nestruktūrizuotas tekstinis formatas
    - Struktūrizuotas tekstinis formatas
    - Darbo istorijos (angl. *job story*)
    - Vartotojo pasakojimai (angl. *user story*)
    - Sekų diagramos
    - Veiklos diagramos

- Lentelės
- Darbe nenaudojame reikalavimų
- Kita (įrašykite)

6. Kokius reikalavimų pateikimo įrankius naudojate darbe? (keli galimi atsakymai)
    - Jira
    - Redmine
    - Microsoft Excel
    - Microsoft Word
    - Seapine
    - Magic Draw
    - Rational DOORS
    - Nenaudojame
    - Kita (įrašykite)
7. Ar esate patenkintas(-a) Jūsų darbe pateikiamų programinės įrangos reikalavimų išsamumu? Įvertinimas nuo 1-o iki 5-ių, kai 1- Ne, nepatenkintas(-a), 5- Taip, patenkinta(-a).
8. Kaip manote, ar automatizuotas testų generavimas iš reikalavimų palengvintų / pagreitintų Jūsų komandos darbą? Įvertinimas nuo 1-o iki 5-ių, kai 1- Ne, visai nepalengvintų, 5- Taip, būtų labai naudinga.
9. Ar iki šiol esate ką nors girdėję apie *Gherkin* kalbą? Įvertinimas nuo 1-o iki 5-ių, kai 1- Ne, nieko negirdėjau, 5- Taip, puikiai žinau kas tai.
10. Ar esate naudoję *Gherkin* kalbą? Įvertinimas nuo 1-o iki 5-ių, kai 1- Ne, niekada neteko naudoti, 5- Taip, naudoju kasdien.
11. Kaip manote, ar *Gherkin* kalba suprantama? (*Gherkin* kalbos pavyzdys pateiktas anketos pradžioje) Įvertinimas nuo 1-o iki 5-ių, kai 1- Visiškai nesuprantama, 5- Puikiai suprantama.
12. Kaip manote, ar būtų naudinga naudoti *Gherkin* kalbą Jūsų darbe? Įvertinimas nuo 1-o iki 5-ių, kai 1- Visai nenaudinga, 5- Labai naudinga.
13. Kaip manote, ar būtų sudėtinga parašyti *Gherkin* testus, pagal Jūsų darbe naudojamą reikalavimų aprašymo formą(-as)? Įvertinimas nuo 1-o iki 5-ių, kai 1- Labai sudėtinga, 5- Labai paprasta.
14. Sekų diagramos pavyzdys:

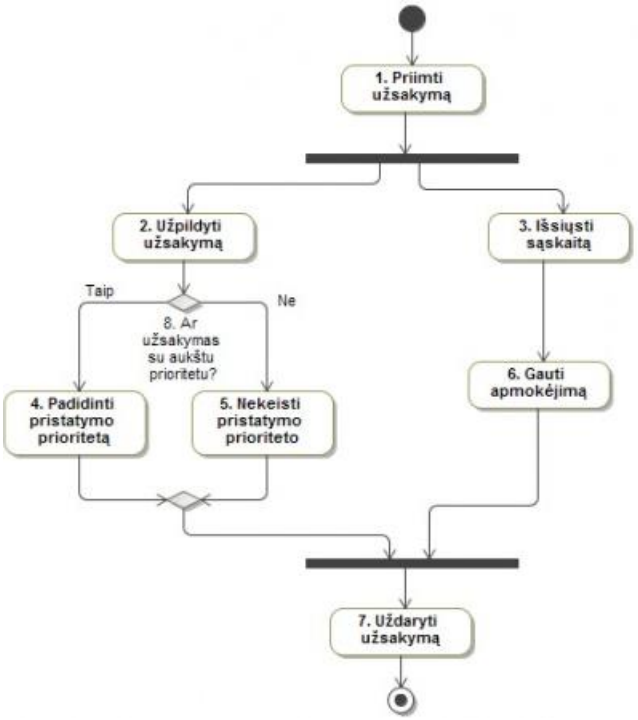


Remdamiesi anketos pradžioje pateiktu pavyzdžiu, pabandykite parašyti *Gherkin* testą 14-ame punkte pateiktai sekų diagramai. (Kad būtų paprasčiau galite parašyti tik reikiamų sakinių numerius, pvz. *Given* 1, *When* 2, *Then* 3)

15. Darbo istorijos pavyzdys:

    "Kuomet turiu sukurtą paskyrą sistemoje, noriu turėti galimybę įvesti vartotojo vardą, slaptažodį ir paspausti prisijungimo mygtuką, kad galėčiau prisijungti ir naudotis sistema." Remdamiesi anketos pradžioje pateiktu *Gherkin* testo pavyzdžiu, pabandykite parašyti *Gherkin* testą šiai darbo istorijai.

16. Veiklos diagramos pavyzdys:



Remdamiesi anketos pradžioje pateiktu pavyzdžiu, pabandykite parašyti *Gherkin* testą 17-ame punkte pateiktai veiklos diagramai. (Kad būtų paprasčiau galite parašyti tik reikiamų sakinių numerius, pvz. *Given* 1, *When* 2, *Then* 3)