KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Robertas Jasaitis

Programų vartotojo sąsajos automatinis testavimas pagrįstas UML modeliais

Magistro darbas

Darbo vadovas prof. dr. Eduardas Bareiša

Kaunas, 2010

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Robertas Jasaitis

Programų vartotojo sąsajos automatinis testavimas pagrįstas UML modeliais

Magistro darbas

Recenzentė
prof. dr. Lina Nemuraitė
2010 - 05 - 28

Vadovas
prof. dr. Eduardas Bareiša
2010 - 05 - 28

Magistrantas
Robertas Jasaitis, IFM-4/2 gr.
2010 - 05 - 28

Kaunas, 2010

Turinys

Paveikslų sąrašas

Lentelių sąrašas

ĮVADAS

Modeliu paremtas testavimas tapo populiaria priemone ne tik programinės įrangos architektūros kūrime ar programinės įrangos kūrimo procesuose, bet ir plačiai naudojamas programinės įrangos testavimui vykdyti. Tokio tipo testavimas gali būti ypač naudingas vartotojo sąsajos testavimo proceso automatizavimui. Tokiu testavimo modeliu paremtas testavimas nėra pilnai automatinis, tik automatizuotas. Taikant šią metodiką, yra būtina sudaryti testuojamą sistemą atitinkantį modelį ir analizuoti rezultatus rankiniu būdu. Kiti testavimo proceso veiksmai vykdomi automatiškai.

Temos aktualumas. Vartotojo sąsajos kūrėjai įprastai naudoja žmogiškuosius resursus testuodami savo kuriamą sistemą. Viena iš priežasčių, kodėl šiam darbui reikia daug resursų, yra ta, kad rinkoje egzistuoja aibė įvairių įrenginių, kurie skiriasi savo ekrano raiška, operacine sistema, architektūra ir pan. Kuriamą programą siekiama testuoti kuo platesnėje įrenginių aibėje, o testavimą dažniausiai atlieka žmogus. Testavimas naudojant žmogiškuosius resursus užima daugiau laiko ir yra mažiau patikimas klaidų atžvilgiu negu automatinis testavimas.

Vartotojo sąsajos testavimo automatizavimui dažniausiai naudojami įrašymo - pragrojimo įrankiai. Tokie įrankiai teikia galimybę programą vykdyti įrašymo rėžimu. T.y., visi testuotojo veiksmai vykdant testuojamą programą yra įrašomi iš išsaugojami. Vėliau šiuos veiksmus testavimo sistema gali automatiškai atkartoti. Tokios sistemos yra nepajėgios užtikrinti aukšto sistemos vartotojo sąsajos ištestuojamumo lygio. Be to pasikeitus sistemai testinius atvejus reikia pakartotinai įrašyti. Įrašymas yra rankinis veiksmas, be to ne visada užtikrina, kad įrašinėjama sistema veikia teisingai, tad galima teigti, kad tuo atveju yra sudaromi klaidingi testiniai rinkiniai.

Tyrimo objektas - vartotojo sąsajos testavimo automatizavimas.

Tyrimo tikslas - išnagrinėti vartotojo sąsajos testavimo automatizavimo būdus. Realizuoti programinę įrangą gebančią automatiškai testuoti vartotojo sąsają ir gebančią testinius atvejus generuoti iš duotų UML modelių. Kuriama programinė įranga turėtų būti realizuota naudojant Java SE technologiją [1]. Kuriama programinė įranga turi būti pagrįsta „testuok anksti – testuok dažnai“ [2] principu. Ištirti sukurtos programinės įrangos kokybę ir, jei reikia, ją pagerinti.

Tyrimo problema - vartotojo sąsajos testavimas dažniausiai vis dar vykdomas rankiniu būdu. Rinkoje nėra kokybiškų priemonių gebančių automatiškai testuoti vartotojo

sąsają. Egzistuojančių priemonių neužtenka kokybiškam vartotojo sąsajos testavimo automatizavimui užtikrinti.

Tyrimo uždaviniai:

- Išnagrinėti egzistuojančias sistemas gebančias bent dalinai automatizuoti vartotojo sąsajos testavimo procesą.
- Išnagrinėti vartotojo sąsajos testavimo testinio atvejo sąvoką.
- Realizuoti automatinį, spartų, mažai atminties reikalaujantį bei tikslų vartotojo sąsajos testavimo būdą paremtą testinių atvejų generavimu iš UML diagramų.
- Ištirti realizuotos sistemos kokybę ir, jei reikia, sistemą patobulinti.

Darbas sudarytas iš šių dalių:

Vartotojo sąsajos automatinio testavimo sistemų analizė. Aprašoma tyrimo sritis, objektas bei, remiantis literatūros analize ir nagrinėjant egzistuojančias sistemas, atskleidžiamas temos aktualumas.

Projektavimas. Ši dalis apima komponentų projektavimą bei programavimą. Projektavimui naudojami UML įrankiai.

Eksperimentinis tyrimas. Realizuotos vartotojo sąsajos automatinio testavimo sistemos kokybės tyrimas remiantis pasirinktos konkrečios programos programų mutantų [3] generavimu ir vykdymu.

Išvados. Šioje dalyje pateikta apibendrinta informacija apie sukurtą vartotojo sąsajos automatinio testavimo sistemos funkcionalumą ir išskirtinumą, o taip pat ir tobulinimo galimybes.

Tyrimo metodika:

- mokslinės literatūros analizė;
- publikacijų ir pranešimų analizė.

Darbo rezultatas - sukurtos vartotojo sąsajos automatinio testavimo sistemos reikšmė jos vartotojams. Rezultatų palyginimas.

1. VARTOTOJO SĄSAJOS AUTOMATINIO TESTAVIMO SISTEMŲ ANALIZĖ

1.1. Tyrimo sritis, objektas ir problema

Pagrindinis **darbo tikslas - realizuoti programinę įrangą, gebančią automatiškai testuoti vartotojo sąsają ir testinius atvejus generuojančią iš duotų UML modelių. Kuriama programinė įranga turėtų būti realizuota naudojant Java technologiją.** Taigi šio darbo tyrimo sritis apims vartotojo sąsajos testavimo automatizavimo būdų analizę.

UML modeliai tapo ypač populiaria priemone modeliuojant programinės įrangos architektūrą. UML modeliai šiais laikais naudojami ne tik įprastoms klasių diagramoms, veiklos diagramoms, sekų diagramoms modeliuoti ir pan., bet tampa vis populiaresnia priemone ir yra taikomi daugelio kitų projektavimo uždavinių sprendimui. Vartotojo sąsajos atvaizdavimas UML diagramomis vis dar nėra populiarus būdas, nors literatūroje vis dažniau sutinkame siūlymų naudoti šias technologijas. Taigi literatūroje atsirandantys straipsniai, vykdomos konferencijos ir panašaus pobūdžio įvykiai byloja, kad ateityje ši technologija neaplenks ir vartojo sąsajos modeliavimo proceso.

Rinkoje jaučiamas vartotojo sąsajos automatinio testavimo sistemos, kuri gebėtų testinius atvejus generuoti iš UML modelių ir automatiškai vykdyti testavimą, trūkumas. Tokia sistema galėtų būti naudojama efektyvesniam testavimui atlikti. Tokia sistema gebėtų testuoti vartotojo sąsają greičiau ir tiksliau. Greitis pasiekiamas tuo, kad automatinio testavimo procese nedalyvauja žmogus, o visą testavimą atlieka įrenginio procesorius. Testuojant sistemą skirtingose platformose testavimą galima atlikti lygiagrečiai (testuoti skirtingus įrenginius tuo pačiu metu). Šie procesai nereikalautų žmogiškųjų resursų ir vyktų lygiagrečiai, o tai ir yra testavimo atlikimo greičio esmė. Tikslumas būtų pasiekiamas tuo, kad sistema tikrintų kiekvieną ekrano grafinį elementą ir, nesutapus bent vienam elementui ar bent vienam iš jo atributų, fiksuotų testuojamos programos klaidą. Tikrinami būtų visi ekrano vaizdai viso testavimo proceso metu. Testiniai atvejai būtų generuojami automatiškai naudojant UML arba specifinius testinius atvejus, kurie būtų sudaromi programuotojo ir įrašomi į atmintį. Vieno testavimo procese būtų įvykdomi visi testiniai atvejai. Taigi tokia sistema yra šio darbo objektas. Tokios sistemos trūkumas ir sprendimo nebūvimas yra tyrimo problema.

1.2. Pasaulinio lygio analogiškų sistemų lyginamoji analizė

- **„Marathon" – įrašymo/pergrojimo testavimo įrankis**

„Marathon" paleidžia programą „maratono" rėžimu [4]. Programa įrašo visus vartotojo veiksmus ir programos esamą būseną. Po tokio įrašymo programą galima pakartotinai testuoti. Tokios programos trūkumas yra tas, kad pakeitus bent menką dalį programos reikia iš naujo įrašinėti visus veiksmus, o tai reiklus laikui veiksmas. Verta paminėti, kad ši programa tinka testuoti Java SE programavimo kalba sukurtas programas. Taigi ši programa netiktų mobiliųjų telefonų programoms testuoti.

- **Vienetų (angl. Unit) testavimo sistema**

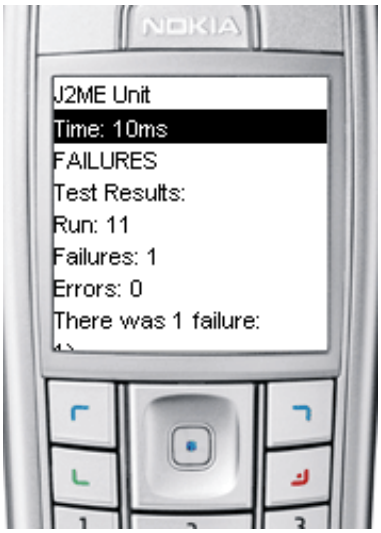
Kitas plačiai paplitęs taikomųjų programų testavimo būdas – vienetų testavimas [5]. Vienetų testavimas taip pat gali būti taikomas ir vartotojo sąsajai testuoti. Vienas iš vienetų testavimo sistemų pavyzdžių – „JUnit" [6]. „JUnit" yra Java 2 Standart Edition biblioteka, turinti vienetų testų karkasą skirtą Java platformos programoms testuoti.

Keletas „JUnit" panaudojimo pavyzdžių:

„JUnit" galima integruoti į vieną iš populiariausių Java kalbos programavimo aplinkų „Eclipse". Integravimo pavyzdys pateiktas 1 paveiksle.

1 pav. JUnit integravimasis į „Eclipse" programavimo aplinką

„JUnit" gali būti vykdomas ir mobiliuosiuose telefonuose. Vykdymas telefone pateiktas 2 paveiksle.



2 pav. JUnit vykdymas mobiliajame telefone

„JUnit” puikiai tinka testuoti programų sistemų logikai – įvairaus pobūdžio funkcijoms, kurios gali būti pilnai ištestuotos parinkus tam tikrą kiekį pradinių duomenų ir laukiamų rezultatų rinkinių. Tačiau „JUnit” tinka ir vartotojo sąsajai testuoti, nors tai ir nėra šios sistemos pagrindinė paskirtis. Galima tikrinti kai kuriuos grafinių objektų parametrus, šių objektų būsenas, metodus, kurie atvaizduoja šiuos objektus, ir pan. Tačiau yra labai komplikuota testuoti būtent tai kas vaizduojama ekrane konkrečiu laiko momentu. Testinių atvejų parengimas yra vykdomas sistemos programuotojų ir neteikia jokių testinių atvejų automatinio generavimo galimybių. Taigi „JUnit” sistema šiuo požiūriu nepatenkina automatinio vartotojo sąsajos testavimo įrankio reikalavimų.

1.3. Aptartų sistemų lyginamoji analizė

Lyginant aptartas sistemas yra akivaizdu, kad naudojant „Marathon” įrankį testinių atvejų kūrimas yra žymiai paprastesnis lyginant su „JUnit” įrankiu. Pirmuoju atveju testinius rinkinius gali kurti testuotojas, neturintis žinių ir kompetencijos programavimo srityje, kai tuo tarpu antruoju atveju testuotoju dažniausiai būna testuojamos sistemos programuotojas. Automatinio testinių atvejų generavimo šie įrankiai neturi. Galimybės naudoti UML priemones testinių atvejų kūrimo palengvinimui šie įrankiai taip pat neturi.

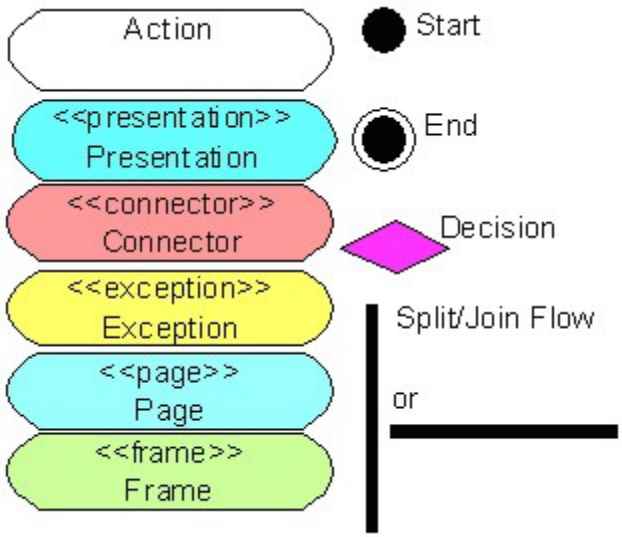
Kitame skyrelyje bus aptarti įvairūs literatūroje minimi automatinio vartotojo sąsajos testavimo metodai, bei automatinio testinių atvejų generavimo iš UML modelių metodai. Taip pat vartotojo sąsajos atvaizdavimas UML diagramomis.

1.4. Literatūros šaltiniuose pateiktų sprendimų lyginamoji analizė

1.4.1. Vartotojo sąsajos atvaizdavimas UML modeliais

UML modeliai dėl savo populiarumo siūlomi naudoti ir vartotojo sąsajos atvaizdavimui. Literatūroje vis dažniau sutinkami tokio tipo problemų sprendimo pasiūlymai. Vienas iš šių pasiūlymų yra pateiktas IBM kompanijos, skatinantis naudoti UML veiklos diagramas vartotojo sąsajos modeliavimui [7].

Vartotojo sąsajos atvaizdavimui naudojamos UML veiklos diagramos šiek tiek skiriasi nuo standartinių UML veiklos diagramų. Šiose diagramose naudojami elementai pateikti 3 paveiksle. Verta atkreipti dėmesį kad elementų stereotipai skiriasi nuo standartinėse veiklos diagramose naudojamų stereotipų.



3 pav. UML veiklos diagramose (vartotojo sąsajos modeliavimui) naudojami elementai

- **Diagramos elementų analizė [7]**

Action yra pagrindinis diagramos elementas. Šis elementas atstovauja veiksmus vykdomus sistemos arba aktoriaus (bet ne vartotojo). Kadangi šis elementas yra plačiausiai naudojamas, jis paprastai yra arba baltos spalvos arba visai bespalvis.

Presentation stereotipas atstovauja ryšį tarp vartotojo aktorių ir sistemos. Šis elementas yra specialus *Action* elemento atvejis, kuris yra aktyvuojamas tik vartotojo bet ne sistemos. Verta paminėti, kad naudojant *Presentation* elementą *Action* elemento tam pačiam įvykiui aprašyti naudoti nebereikia.

Connector stereotipas yra jungtis tarp dviejų veiklos diagramų. Naudojant veiklos diagramas jos dažnai tampa didelės ir sunkiai skaitomos, todėl yra naudinga dalį diagramos išskaidyti į mažesnes diagramos dalis ir jas tarpusavyje jungti šiuo elementu.

Exception paprastai naudojamas klaidos atvejui atvaizduoti, tačiau gali būti naudojamas ir atvaizduoti neįprastą ar netikėtą sistemos elgseną. Šis elementas dažnai naudojamas diagramos šakai veiksmų po klaidos atsiradimo atvaizdavimui.

Frame ir *Page* stereotipai yra naudojami tiek web puslapiams tiek ir taikomųjų programų langams atvaizduoti. Šis elementas taip pat gali būti naudojamas logiškai ar fiziškai atskirtoms sistemos dalims atvaizduoti. *Frame* paprastai yra suprantamas kaip puslapis, bet tai nebūtinai turi būti laikoma kaip taisyklė. Pavyzdžiui *Frame* gali būti universalios vartotojo sąsajos dalis, bet nebūtinai atskiras puslapis.

Kiti diagramos elementai nesiskiria nuo standartinių veiklos diagramų elementų ir jų naudojimas identiškas standartinių veiklos diagramų elementų naudojimui.

Šis būdas nereikalauja jokių papildomų UML komponentų, todėl taikant šį būda galima naudoti visas standartines UML savybes. Taip pat gali būti naudojami visi standartiniai įrankiai UML diagramoms modeliuoti. Šis būdas yra patogus būdas vartojo sąsajai modeliuoti ir nereikalauja jokių papildomų įrankių. UML diagramos gali būti konvertuojamos į XML formato failą – XMI failą [8]. Ši savybė yra patogi tuo, kad XML formatą yra nesunku analizuoti programavimo lygyje. Rinkoje yra aibė atviro kodo optimizuotų XML analizatorių realizuotų įvairiomis programavimo kalbomis.

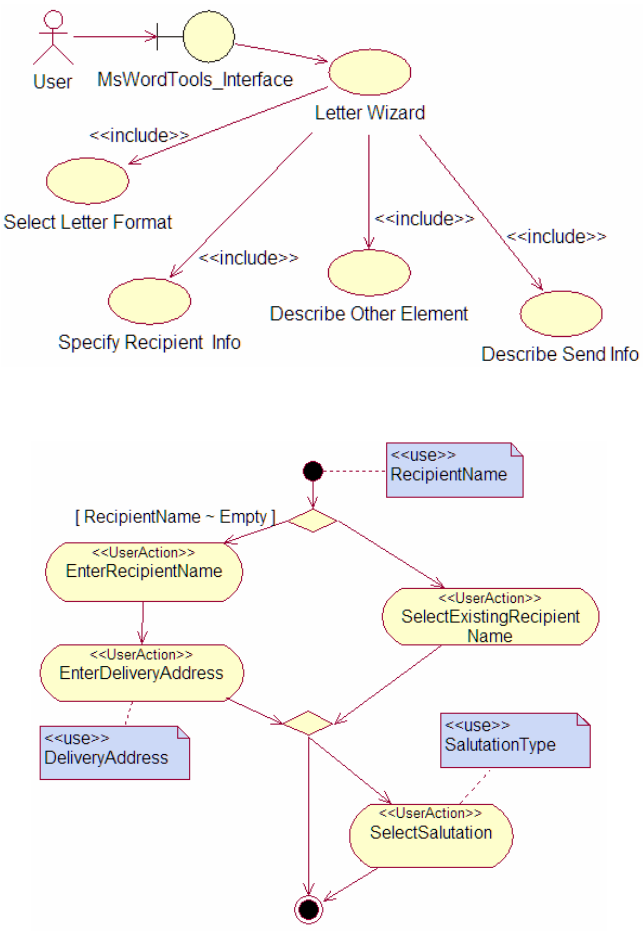
1.4.2. Vartotojo sąsajos testavimo automatizavimas naudojant modeliu paremto testavimo metodiką

Nagrinėjant vartotojo sąsajos testavimo automatizavimo galimybes neretai susiduriame su siūlymais naudoti modeliu paremto testavimo metodikas. Modeliu paremto testavimo proceso metu yra vykdomas sistemą atitinkantis modelis lygiagrečiai su testuojama sistema ir yra tikrinama, ar testuojama sistema atitinka modelį [9]. Modelis gali būti sudarytas visai sistemai ar tik testuotoją dominančiai sistemos daliai, kurią siekiama ištestuoti. Sistemos atvaizdavimą modeliu galima vykdyti įvairiais metodais, nes ši metodika nenurodo kokio tipo modelius yra privalu naudoti.

Literatūroje egzistuoja teoriniai metodai vartotojo sąsajos testavimo automatizavimui įgyvendinti naudojantys UML panaudos atvejų diagramas kartu su UML veiklos diagramomis [10]. Kiekvienam panaudos atvejui yra sudaroma atskira UML veiklos diagrama. T.y. testuojamai sistemai sudaroma panaudos atvejų diagrama. Visi į diagramą įtraukti panaudos atvejai - tai lyg nuorodos į tas testuojamos sistemos funkcionalumo dalis kurias reikia ištestuoti. Tai nebūtinai turi būti visi sistemos panaudos atvejai. Kiekvienam panaudos atvejui yra sudaroma po atskirą UML veiklos diagramą. Ši diagrama testavimo metu yra naudojama testuojamos sistemos funkcionalumo eigai nustatyti. Sistemos funkcionalumo eiga čia suprantama kaip sistemos naudojimo atvejis. Tai tarsi veiksmų sąrašas kuriuos reikia atlikto norint ištestuoti tam tikrą sistemos funkcionalumą. Į šį aprašą gali būti įtraukti sąlygos sakiniai, ciklai ir pan. Taigi ši metodika reikalauja, kad kiekvienos sistemos panaudos atvejis būtų aprašytas detaliais žingsniais, išreikštais veiklomis UML veiklos diagramoje. Taigi didelės apimties sistemoms, turinčioms aibę panaudos atvejų, tektų sudaryti tokią pat didelę aibę UML veiklos diagramų norint pilnai ištestuoti sistemą. Be to, tokio tipo sistemos testuoja ne pačią vartotojo sąsają, o sistemos valdymą per vartotojo sąsają. Taigi testuojamas objektas yra ne visiškai aiškus. Tokio tipo sistemos netikrina ar vartotojo sąsajos grafiniai komponentai

yra teisingai atvaizduojami ekrane: ar jų koordinatės yra teisingos, ar jų spalva, dydis ir kiti parametrai yra teisingi. Taigi tokio tipo sistemos naudingos testuojant tik tam tikrą specifinį sistemos funkcionalumą, nekreipiant dėmesio į visą kitą sistemos funkcionalumą.

Šiuo metodu sumodeliuoto elektroninio laiško išsiuntimo sistemos modelių pavyzdžiai pateikti 4 paveiksle.



4 pav. Modeliu paremto testavimo būdu sudaryti UML panaudos atvejų ir UML veiklos modeliai

Anksčiau aptartos modelių dydžio problemos bei testuojamo objekto abstraktumo problemos lemia, kad ši metodika praktikoje yra retai naudojama vartotojo sąsajos testavimui.

1.5. Automatinio vartotojo sąsajos testavimo įgyvendinimo problemos

Sprendžiama problema yra mažai išnagrinėta ir neturi tikslių analogų. Kyla klausimas kokios problemos lėmė, kad vartotojo sąsajos testavimas vis dar yra mažai automatizuotas, reikalaujantis nemažų žmogiškųjų resursų veiksmas. Yra žinoma aibė programų padedančių automatizuoti vartotojo sąsajos testavimo procesą, tačiau tokios programos padengia mažesniąją dalį viso testavimo proceso.

Šiame skyrelyje aptartos problemos, lemiančios šią situaciją.

- **Ribotas prieinamumas prie testuojamo objekto**

Sunku sukurti universalų testavimo įrankį, nes testuojamas objektas yra vartotojo sąsaja, kuri gali būti realizuota įvairiausiais būdais. Norint testuoti ją yra būtina žinoti testuojamos vartotojo sąsajos struktūrą, klasių hierarchiją ir pan. Nagrinėjamų programų vartotojo sąsają kuriama nenaudojant standartinių komponentų. Taigi programų kūrėjai kuria savo grafinius komponentus arba naudoja komercinius vartotojo sąsajos karkasus.

Taip pat susiduriama su vartotojo sąsajos klasių prieinamumo problema. Nežinant kokios klasės egzistuoja yra neįmanoma jų pasiekti iš programos kodo. O kurti testavimo karkasą kiekvienai programai tiesiog neapsimoka laiko ir išlaidų atžvilgiu. Verta paminėti, kad Java programavimo kalba turi „java.awt.Robot“ klasę kuri geba valdyti vartotojo sąsajos įvykius, tačiau dažniausiai ši klasė nėra naudojama kuriant nestandartinę vartotojo sąsają.

- **Testavimo spektras yra labai platus**

Testuojant vartotojo sąsają ir norint ją pilnai ištestuoti, testinių atvejų aibė gali siekti net kelis tūkstančius testinių atvejų. Norint ištestuoti vien paprastą vartotojo sąsajos komponentą, reikia turėti bent keletą testinių atvejų kiekvienai komponento būsenai. Vartotojo sąsajos komponento būsenų skaičius paprastai siekia net keliasdešimt būsenų. Tarkime turime paprastą mygtuką. Norint ištestuoti šį komponentą tektų tikrinti kaip jis atvaizduojamas ekrane – tikrinti jo koordinates, dydžius, spalvas, sufokusavimo sąlygą ir pan. Įvykdžius mygtuko įvykį, tarkime mygtuko paspaudimą, tektų šiuos parametrus pakartotinai patikrinti, o taip pat patikrinti ir įvykio atlikimo rezultatą. Būtina tikrinti šio komponento sąveiką su kitais komponentais ir pan. Taigi panagrinėję, iš pažiūros, paprastą komponentą matome, kad testinių atvejų yra kur kas daugiau negu atrodo iš pirmo žvilgsnio.

- **Sunku nustatyti testuojamos programos būseną**

Dažniausiai yra neįmanoma nustatyti ar programa yra teisingoje būsenoje vien žiūrint į vartotojo sąsają. Kaip pavyzdys, gali būti klaida atsitikusi kitoje (ne vartotojo sąsajos) gijoje, kuri nėra atvaizduojama vartotojo sąsajoje. Tokioje situacijoje yra neįmanoma nustatyti šios klaidos, kuri vėliau gali smarkiai įtakoti vartotojo sąsajos elgseną ir teisingumą.

- **Programinio kodo pakeitimai kūrimo stadijoje gali smarkiai įtakoti testavimo rezultatus**

Keičiant vartotojo sąsają, pavyzdžiui papildant komponentą naujomis savybėmis, pakeitimas gali įtakoti jau sukurtų testinių atvejų teisingumą. Kad ir mažiausias pakeitimas, pavyzdžiui mygtuko perkėlimas į kitą vietą ar į kitą konteinerį, gali įtakoti visus susijusius testinius atvejus, tad tektų visus susijusius testinius rinkinius pakeisti.

- **Sunku nustatyti ištestavimo kriterijų**

Automatizavus vartotojo sąsajos testavimo procesą susiduriama su testinių atvejų kiekio pilnumo, persipildymo problema. Būtina nagrinėti testinius atvejus, jų persidengimą, pilnumą. Kuriant automatinio testavimo priemones yra sunku nepersistengti, bei kuo pilniau ištestuoti testuojamą programą su minimaliais laiko ir užimamos atminties resursais.

1.6. Analizės išvados

- Išanalizavus literatūroje siūlomus problemos sprendimo būdus neretai susiduriame su siūlymais naudoti modeliu paremto testavimo metodikas. Tokio tipo metodikos ypač tinka automatizuojant vartotojo sąsajos testavimo procesą. Taigi vartotojo sąsajos automatinio testavimo karkasas turėtų būti realizuotas remiantis modeliu paremto testavimo metodiką.
- Vartotojo sąsajos atvaizdavimui tinka UML modeliai. Literatūroje yra keletas pasiūlymų naudoti UML veiklos modelius vartotojo sąsajos atvaizdavimui. UML veiklos modeliai gali būti pritaikomi vartotojo sąsajos navigacijai atvaizduoti naudojant egzistuojančius elementus su tam tikrais specifiniais stereotipais. Taigi UML veiklos modeliai gali būti naudojami realizuojant vartotojo sąsajos automatinio testavimo karkasą.
- Vartotojo sąsajos testavimui dažnai naudojami įrašymo pergrojimo įrankiai, todėl ši funkcija turėtų būti įtraukta į vartotojo sąsajos automatinio testavimo karkaso funkcionalumą.

2. VARTOTOJO SĄSAJOS AUTOMATINIO TESTAVIMO METODAS

Apžvelgus egzistuojančias sistemas ir literatūroje siūlomus problemos sprendimo būdus, šiame skyriuje bus aptartas kuriamos sistemos sprendimų pasirinkimas. Dalis sprendimų pavaizduoti UML modeliais.

Atskleidžiant sprendimo pasirinkimą yra būtina apibrėžti sprendimo metodo aprašyme naudojamas sąvokas.

2.1. Vartotojo sąsajos įvykis

Vartotojo sąsajos įvykis - tai įvykis kurį įprasto sistemos naudojimo metu atlieka vartotojas, bendravimui su sistema per vartotojo sąsają. Tokių įvykių pavyzdžiai: klaviatūros mygtuko paspaudimo įvykis, pelės klavišo paspaudimo įvykis, pelės judesio įvykis, liečiamosios lazdelės pagalba aktyvuoto ekrano įvykis ir pan.

2.2. Vartotojo sąsajos būsena

Vartotojo sąsajos būsena nėra tokia akivaizdi ir suprantama sąvoka kaip programoje egzistuojančios bet kurios klasės būsenos sąvoka ar visos sistemos būsenos sąvoka. Taip yra todėl, kad vartotojo sąsaja sudaryta iš daugelio klasių ir yra labai lanksti lyginant su bet kuria atskira klase, kuri turi tam tikrą pritaikymą. Vartotojo sąsaja kaip visuma yra kur kas lankstesnė, tad vienareikšmiškai apibrėžti vartotojo sąsajos būseną yra sudėtingas uždavinys. Šiame kontekste, kuriant vartotojo sąsajos automatinio testavimo sistemą, vartotojo sąsajos būsena aprašoma kaip aibė žemiau pateiktų elementų:

- Grafinių objektų hierarchija. Bet kuriuo sistemos vykdymo metu yra įmanoma ekrane atvaizduojamą vartojo sąsają išskaidyti į tam tikras dalis. Šias dalis pavadinkime grafiniais komponentais. Šių grafinių komponentų pavyzdžiai yra mygtukas, konteineris, teksto eilutė, sąrašas ir pan. Paprastai šiuos komponentus galima sudėlioti į hierarchiją. T.y., kiekvienas komponentas gali turėti savyje komponentus (vaiko komponentus) ar būti dalis kokio nors aukštesnio lygio komponento (tėvo komponento). Taigi apibendrinus, vartotojo sąsają galima aprašyti kaip hierarchiją, turinčią vieną tėvinį komponentą, kuri turi aibę vaiko komponentų, o šie savo ruožtu gali turėti savo aibę vaiko komponentų ir t.t. Taigi tokią hierarchiją patogu atvaizduoti XML formatu, kuris šiuo metu yra plačiai paplitęs, ne tik duomenų atvaizdavimui, bet ir vartotojo sąsajos aprašymui ar modeliavimui.
- Kiekvieno grafinio objekto (komponento) stebimi atributai. Vien suskaidžius grafinę vartotojo sąsają į komponentų hierarchiją dar pilnai neaprašome vartotojo sąsajos. Aprašome tik komponentų kiekį ir jų tarpusavio ryšius (tėvo - vaiko ryšius). Tačiau tai pilnai neatsako į klausimą kur vienas ar kitas komponentas yra ekrano koordinačių pradžios atžvilgiu, kokia jo fono spalva, jo tekstas, tipas ir pan. Taigi kiekvienas grafinės vartotojo sąsajos komponentas turi jį aprašančių atributų aibę. Ši aibė gali skirtis tarp skirtingo tipo komponentų. Pavyzdžiui mygtuko komponentas gali turėti teksto ar etiketės atributą, o paveikslo komponentas jų gali neturėti, jei tai paprasto tipo paveikslo komponentas. Komponentų atributų aibės skiriasi tarp skirtingų programų priklausomai nuo naudojamos vartotojo sąsajos, tačiau programos lygyje to pačio tipo komponentas turi tokią pačią jį aprašančių atributų aibę. Jei grafinės vartotojo sąsajos komponentus aprašėme XML formatu, tai prie šito aprašo galime pridėti ir komponentų

atributus. XML struktūra leidžia apibrėžti ne tik objektų ryšius, bet ir saugoti kiekvieno komponento atributų reikšmes.

2.3. Testinis atvejis

Testinis atvejis paprastai suprantamas kaip aibė sąlygų ar kintamųjų, kuriuos testuotojas patikrindamas galėtų atsakyti į klausimą, ar sistema šioje vietoje veikia korektiškai ar klaidingai [11]. Automatinio vartotojo sąsajos testavimo sistemos kontekste šios sąlygos ir kintamieji susideda iš dviejų žemiau pateiktų kintamojo ir sąlygos.

- Teisinga (laukiama) vartotojo sąsajos būsena. Tai ankščiau aprašytu būdu pateikta vartotojo sąsajos būsena tam tikru laiko momentu. Laiko momentai čia skaičiuojami tarp vartotojo sąsajos įvykių. Pirma vartotojo sąsajos būsena visada yra ta kurią vartotojas mato programai pilnai startavus. Laikoma kad ši būsena nesikeičia iki bet kokio vartotojo sąsajos įvykio. Taigi antra vartotojo sąsajos būsena yra vartotojo sąsajos būsena po pirmo vartotojo sąsajos įvykio ir laikoma, kad ji nesikeičia iki antro vartotojo sąsajos įvykio ir t.t.
- Perėjimo sąlyga prie kitos vartotojo sąsajos būsenos. Vartotojo sąsajos būsenos pilnai užtenka patikrinimui ar sistema veikia korektiškai tam tikru laiko momentu. Tačiau turėdami vien teisingą vartotojo sąsajos būseną nežinosime iki kurio momento ši būsena yra teisinga. Šiam trūkumui užpildyti yra būtina įvesti perėjimo prie kitos vartotojo sąsajos būsenos sąlygos kintamąjį. Perėjimo sąlygos struktūra: <aktyvavimo sąlyga>;<vykdymo sąlyga>:<rezultatas>.
  - Aktyvavimo sąlygos galimos reikšmės:
    - n0 – nereikia jokių veiksmų norint aktyvuoti kitą komponentą. Tai reiškia kad šis komponentas yra aktyvuotas pagal nutylėjimą
    - n<k><j> – reiškia navigaciją k kartų (sveikasis skaičius) j kryptimi. „j“ čia gali įgauti reikšmes:
      - d – žemyn (angl. down)
      - u – aukštyn (angl. up)
      - l – kairėn (angl. left)
      - r – dešinėn (angl. right)
  - Vykdymo sąlyga gali būti tik „f“ reikšmė reiškianti paspaudimą (angl. fire) arba tuščia eilutė „“. Vykdymo sąlyga yra išskirta į atskirą grupę dėl tolimesnio funkcionalumo įdėjimo (pavyzdžiui aktyvavimo iš liečiamojo ekrano).

- Rezultato galimos reikšmės:
  - „next“ – pereina į kitą puslapį
  - „back“ – grįžta į prieš tai buvusį puslapį
  - „exit“ – išeina iš sistemos
  - „“ – vykdo veiksmus nesusijusius su navigacija.

Rezultatas reikalingas tik sistemos vidiniams veiksmams ir aiškumui užtikrinti testuotojui sudarant testinius atvejus.

Pavyzdžiui, sąlyga „nr2;f:next” reiškia, kad reikia du kartus spustelėti mygtuką „dešinėn”, tada spragtelėti mygtuką „enter”, tada sistema pereis į kitą puslapį. Jei reiktų eiti du kartus „žemyn” ir vieną kartą „dešinėn”, tada „enter” ir tada sistema turi pereiti į kitą puslapį, tokią seką užrašytume dvejomis sąlygomis: „nd2;:” ir „nr1;f:next”.

Taigi perėjimo sąlyga yra laikoma įvykdyta tada kai yra įgyvendinamos aktyvavimo sąlyga ir vykdymo sąlyga.

2.4. UML veiklos modeliai vartotojo sąsajos atvaizdavimui

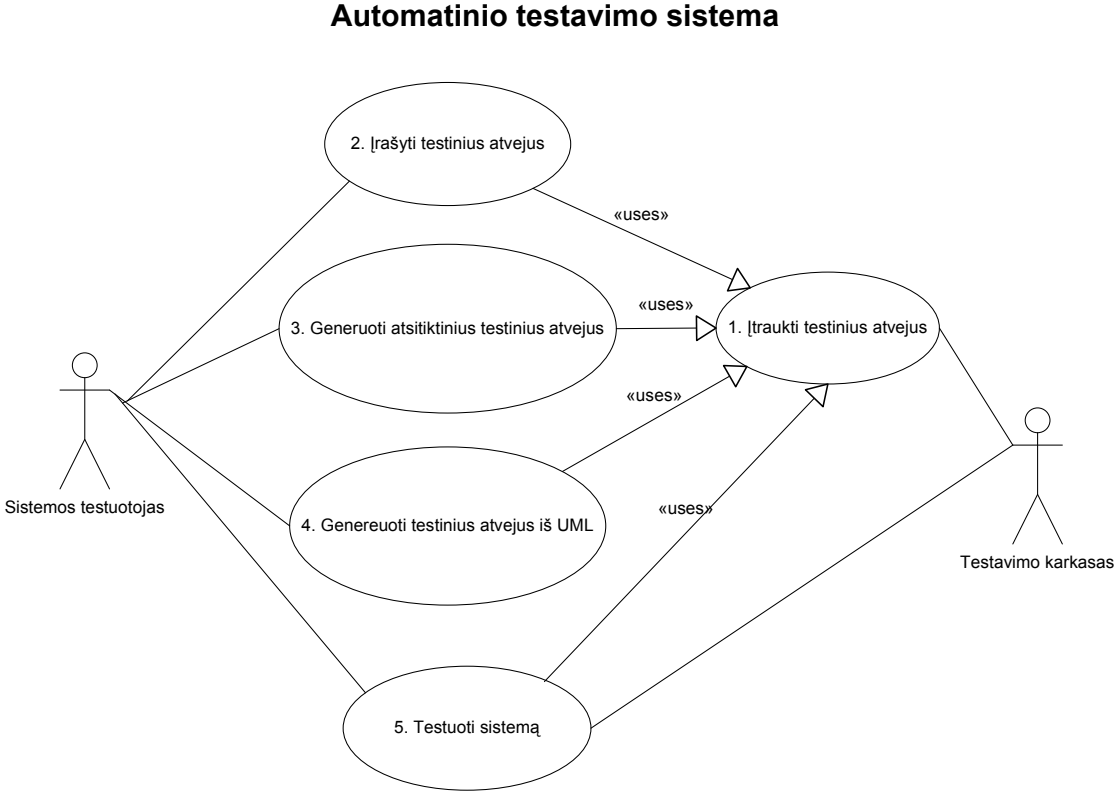
UML veiklos modeliai gali būti naudojamos ir vartotojo sąsajos atvaizdavimui. UML modeliai yra plačiai naudojami ir gerai žinomi. Rinkoje egzistuoja aibė priemonių leidžiančių patogiai ir greitai modeliuoti įvairias UML diagramas. Vartotojo sąsajos atvaizdavimo UML veiklos diagramomis metodika pateikta literatūros analizėje 1.4.1 punkte. Tokia metodika pilnai apibrėžia vartotojo sąsajos navigaciją. T.y., iš kurių objektų kuriuos ekrano vienetus galima pasiekti. Ekrano vienetu čia laikomas puslapis, forma ar langas priklausomai nuo to kokia platforma yra analizuojama. Tačiau, kad sudaryti testinius atvejus vien šios informacijos nepakanka. Reikia papildyti UML veiklos diagramoje naudojamus elementus papildoma informacija.

Šiame kontekste apibrėžtas testinis atvejis reikalauja teisingos vartotojo sąsajos būsenos. Teisinga vartotojo sąsajos būsena yra integruojama į *Page* stereotipo komponentus komentaro komponento pavidalu. Taigi kiekvienas *Page* elementas savo viduje privalo turėti savo vidinės vartotojo sąsajos elementų hierarchijos aprašą. Aprašo struktūra pateikta sąvokų sąraše punkte *vartotojo sąsajos būsena*. Kaip buvo minėta anksčiau, teisinga vartotojo sąsajos būsena pateikiama XML formatu. Ateityje šį formatą galima pakeisti į bet kokią kitą formą, pavyzdžiui, tam tikrą grafinę formą.

Šiame kontekste apibrėžtas testinis atvejis taip pat reikalauja perėjimo sąlygos. Perėjimo sąlyga yra integruojama į *Presentation* stereotipo komponentus komentaro elemento pavidalu. Kiekvienas *Presentation* elementas privalo turėti savo aktyvavimo sąlygos aprašą pateiktą testinio atvejo apibrėžime. T.y., sistemai yra svarbu žinoti prie kokių sąlygų šis įvykis yra aktyvuojamas ir kokius veiksmus šis komponentas iššaukia.

2.5. Vartotojo sąsajos automatinio testavimo metodo aprašymas

Apibrėžus šiame kontekste naudojamas sąvokas galima aptarti sistemoje realizuojamas funkcijas pateiktas UML panaudos atvejų diagrama kuri pateikta 5 paveiksle. Analizuojant panaudos atvejus bus galima lengviau įsisavinti siūlomo metodo esmę.



5 pav. Sistemos panaudos atvejų modelis

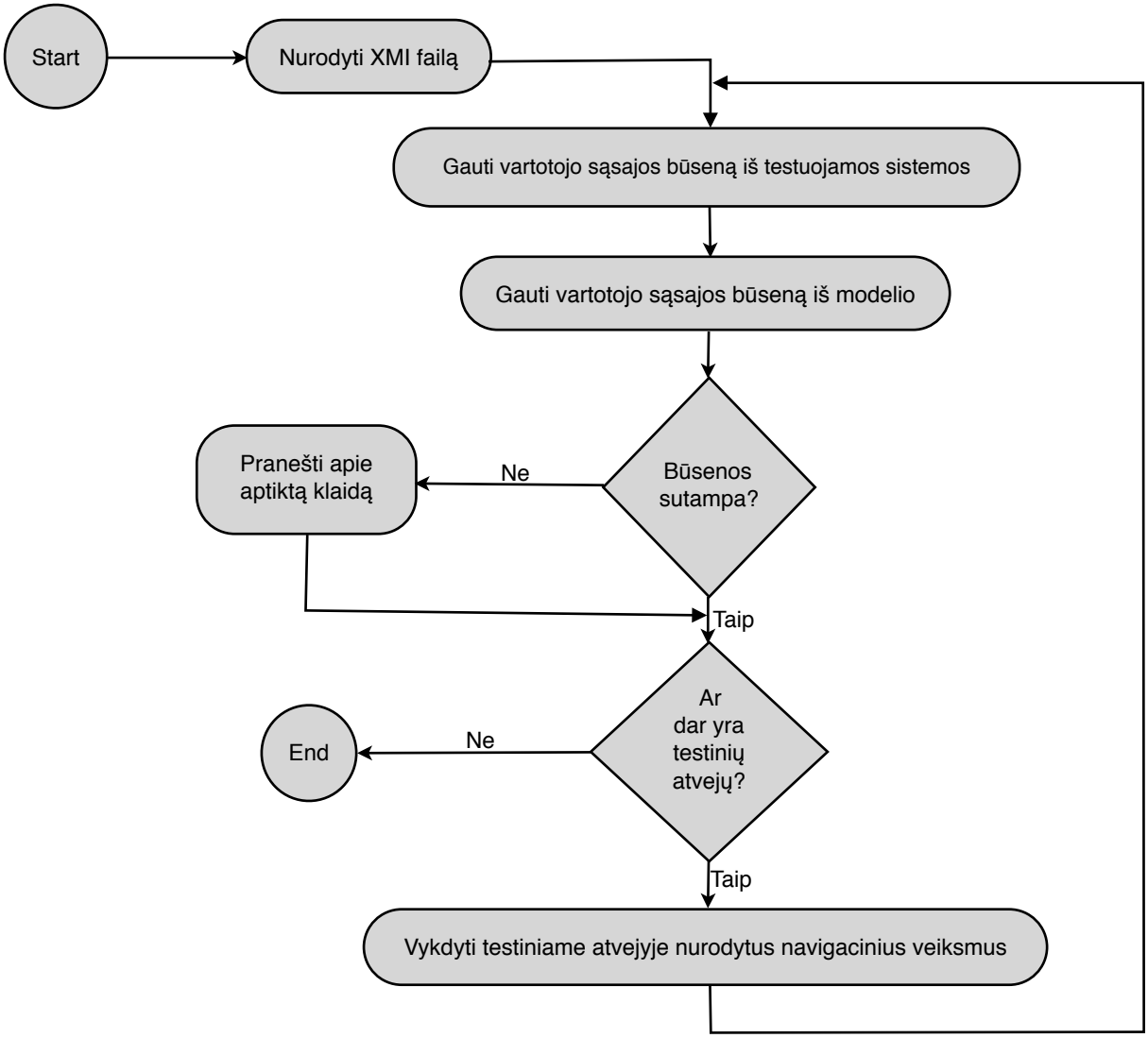
Vartotojo sąsajos automatinio testavimo karkasas atlieka žemiau pateiktas pagrindines funkcijas pavaizduotas 5 paveiksle:

- Atsitiktinio testavimo funkcija – tai funkcionalumas, kurio metu testavimo karkasas nepertraukiamai generuoja atsitiktinius vartotojo sąsajos įvykius ir imituoja, kad juos atlieka vartotojas, paduodamas vartotojo sąsajos įvykius testuojamai sistemai. Šis funkcionalumas nepatikrina sistemos vartotojo sąsajos teisingumo, tik tikrina sistemos vartotojo sąsajos reakciją į nepertraukiamą ir dirbtinai dažną vartotojo sąsajos įvykių

aktyvavimą. Ši funkcija naudinga sistemos kūrimo išskaidymo į atskiras dalis tikslui įgyvendinti. Sistemos dirbtinis vartotojo sąsajos įvykių generavimas, tokiu būdu kaip tai generuoja vartotojas, yra pirmas žingsnis automatinio testavimo karkaso įgyvendinimo procese. Šių vartotojo sąsajos įvykių generavimas yra techninis uždavinys apsiribojantis tikslinės sistemos analize, kad būtų žinoma kaip galima dirbtinai generuoti tokius vartotojo sąsajos įvykius.

- Rankinio testinių atvejų generavimo funkcija – tai funkcionalumas leidžiantis testuotojui įrašyti savo norimus specifinius testinius atvejus paprastu ir ilgai netrunkančiu būdu. Aktyvavus šią funkciją sistema klausosi kiekvieno testuotojo vartotojo sąsajos įvykio ir kiekvieno vartotojo sąsajos įvykio metu sugeneruoja testuojamos sistemos vartotojo sąsajos būseną, kurią laiko teisinga sistemos būsena. Sistemos būsena apibrėžta šio skyrelio sąvokų sąraše. Vėliau galima šiuos testinius atvejus pakartotinai įvykdyti ir sistema automatiškai patikrina ar testuojamoje sistemoje vykdant tuos pačius veiksmus neatsirado naujų klaidų. Tai ypač patogu vykdant testavimą sistemos kūrimo metu. Realizavus tam tikrą sistemos dalį, galima sukurti tokį rankinį testinių atvejų rinkinį, kuris patikrina tik realizuotą sistemos dalį. Vėliau realizavus kitas dalis, galima patikrinti ar anksčiau realizuota dalis nebuvo įtakota ir ar joje neatsirado klaidų. Šiam veiksmui atlikti jau nereikėtų jokių testinių atvejų generavimo veiksmų, būtų apsiribota tik anksčiau sugeneruotų testinių atvejų įvykdymu.
- Automatinio testinių atvejų generavimo iš UML funkcija – tai funkcionalumas leidžiantis testuotojui iš sistemos vartotojo sąsajos, aprašytos UML formatu, generuoti vartotojo sąsajos testavimo testinius atvejus. Tokiu būdu testavimo procesas gali prasidėti labai ankstyvoje stadijoje ir vystytis kartu su visa sistema. Evoliucionuojant sistemai yra pildomi ir UML modeliai. Bet kuriuo momentu sugeneruoti testiniai atvejai vėliau gali būti pakartotinai įvykdyti. UML braižymo taisyklės aprašytos šio skyrelio sąvokų sąraše, UML veiklos diagramų naudojimo vartotojo sąsajos atvaizdavimui skirsnyje. Sistemai paduodamos UML diagramos pateiktos XMI formatu. Sistema išanalizuoja duotą XMI failą ir iš jo struktūros sugeneruoja testinius atvejus, bei vartotojo sąsajos įvykių seką. Vartotojo sąsajos įvykiai čia išreikšti unikaliais sveikojo tipo skaičiais. Kiekvieno tipo įvykis turi jam priskirtą identifikavimo numerį sistemoje.
- Automatinio testavimo funkcija – tai funkcionalumas leidžiantis anksčiau sugeneruotus testinius atvejus įvykdyti automatiškai ir gauti testavimo rezultatus patogioje formoje. Toks testavimo būdas yra efektyvus, nes trunka kur kas mažiau laiko ir naudoja kur

kas mažiau žmogiškųjų resursų. Automatinio testavimo proceso diagrama pateikta 6 paveiksle.



6 pav. Automatinio testavimo proceso modelis

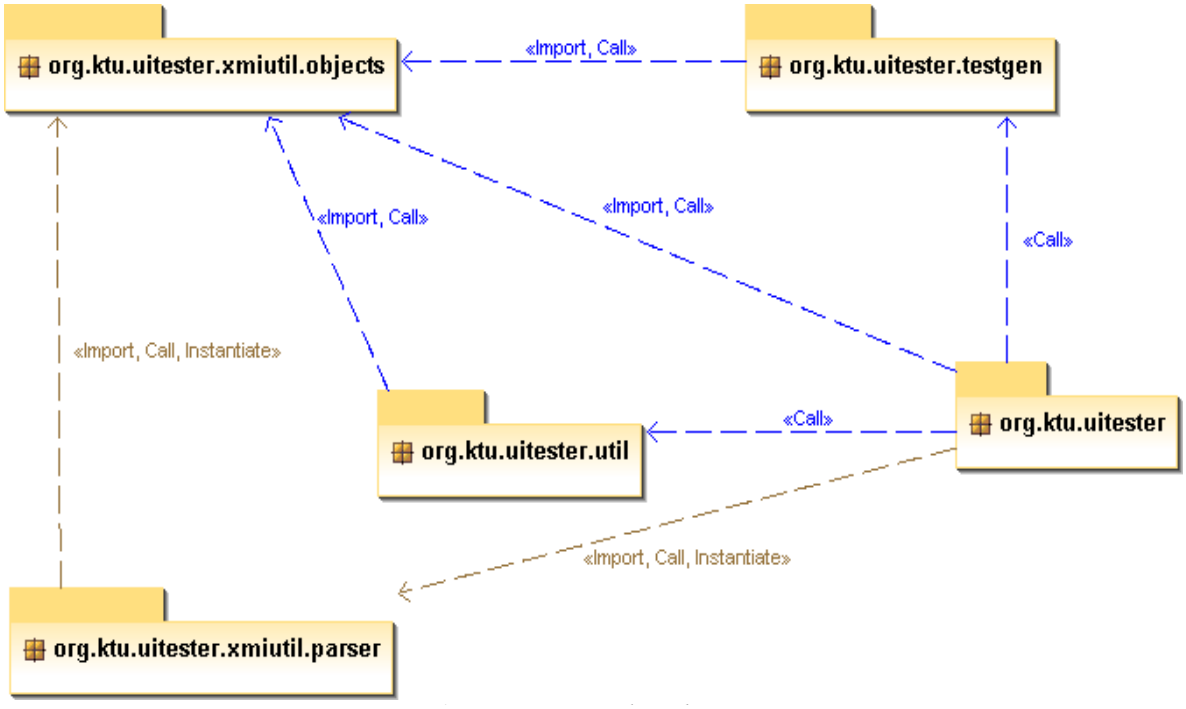
Automatinio testavimo metu sistema paeiliui vykdo (imituoja) vartotojo sąsajos įvykius, išsaugotus testinių atvejų generavimo metu. Prieš ir po kiekvieno vartotojo sąsajos įvykio, sistema paima vartotojo sąsajos būseną iš testuojamos sistemos ir palygina ją su laukiama vartotojo sąsajos būsena, paimta iš anksčiau sugeneruotų testinių atvejų. Laukiamos vartotojo sąsajos būsenos turi savo identifikavimo numerį susietą su vartotojo sąsajos įvykiu. Šis numeris reikalingas tam, kad būtų žinoma kuri laukiama vartotojo sąsajos būsena turi būti naudojama konkrečiu laiko momentu. Laiko vienetas čia yra susietas su įvykdytų vartotojo sąsajos įvykių skaičiumi. Pirmu laiko vienetu yra laikomas laiko tarpas tarp sistemos pilno startavimo iki pirmo vartotojo sąsajos įvykio vykdymo pradžios. Antru laiko vienetu yra laikomas laiko tarpas tarp pirmo vartotojo sąsajos įvykio vykdymo pabaigos iki antro

vartotojo sąsajos įvykio vykdymo pradžios ir t.t. Patikrinus einamą laukiamą vartotojo sąsajos būseną su vartotojo sąsajos būsena, paimta iš testuojamos sistemos, gautas lyginimo rezultatas išsaugomas faile ir atspausdinamas teksto išvestyje (angl. console) (jei testuojamoje platformoje teksto išvestis yra prieinama).

Sistemos realizacijai reikalinga paketų struktūra pavaizduota 7 paveiksle.

Iš 7 paveiksle pateiktos paketų diagramos matome, kad pagrindinis paketas yra org.ktu.uitester. Šis paketas valdo visas pagrindines sistemos funkcijas:

- XML analizavimą
- XML objektų valdymą
- Testinių atvejų generavimą
- Bendravimą su testuojama sistema



7 pav. Sistemos paketų diagrama

3. VARTOTOJO SĄSAJOS AUTOMATINIO TESTAVIMO KARKASO PROJEKTAVIMAS

3.1. Projektavimo tikslas

Projektavimas yra vienas sudėtingiausių sistemos kūrimo etapų. Nuo projektavimo kokybės priklauso tai, kaip lengva bus sistemą realizuoti, kaip sudėtinga bus sistemą pakeisti pasikeitus kliento reikalavimams bei kaip sudėtinga bus sistemą prižiūrėti.

Įgyvendinant vartotojo sąsajos automatinio testavimo panaudojant UML modelius sistemą bus atlikti žemiau pateikti projektavimo uždaviniai:

- Sudarytos komponentų klasių diagramos
- Sudarytos komponentų sekų diagramos
- Sudarytos sistemos būsenų diagramos

3.2. Reikalavimų modelis

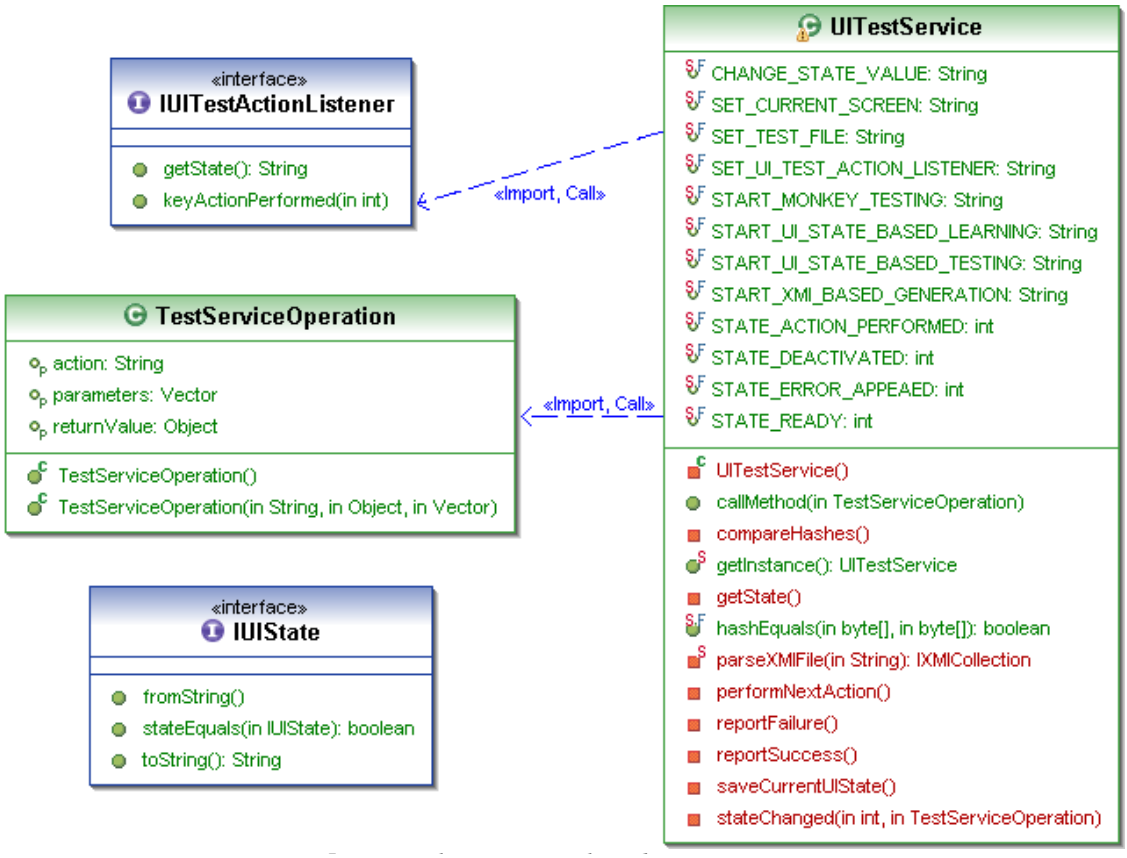
Sistemos analizės metu buvo nustatyta kad sistema bus sudaryta iš žemiau išvardintų paketų:

- Testavimo eigos valdymo paketas
- Testinių atvejų generavimo paketas
- XML analizavimo paketas
- XML objektų paketas
- Bendrojo pobūdžio funkcijų paketas

Projektavimo metu buvo sudarytos šių paketų klasių diagramos.

3.2.1. Testavimo eigos modelis

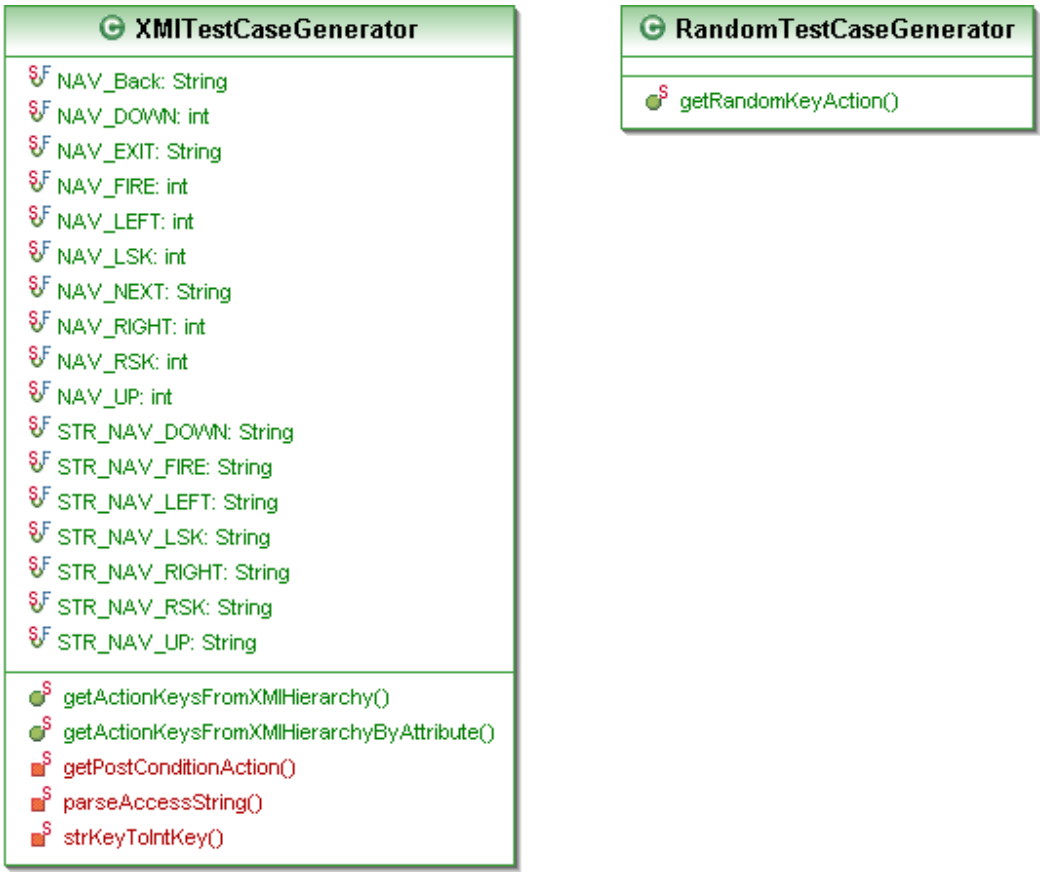
Testavimo eigos valdymo paketas yra testavimo karkasą valdantis paketas. Šis paketas pasiekiamas vartotojui per IUITestActionListener sąsają. Naudodamasis šia sąsaja vartotojas gali valdyti visą testavimo procesą. Valdymas atliekamas paduodant karkasui TestServiceOperation klasės objektus. Testavimo eigos valdymo paketo klasių diagrama pateikta 8 paveiksle.



8 pav. Testavimo eigos valdymo paketo klasių diagrama

3.2.2. Testinių atvejų generavimo modelis

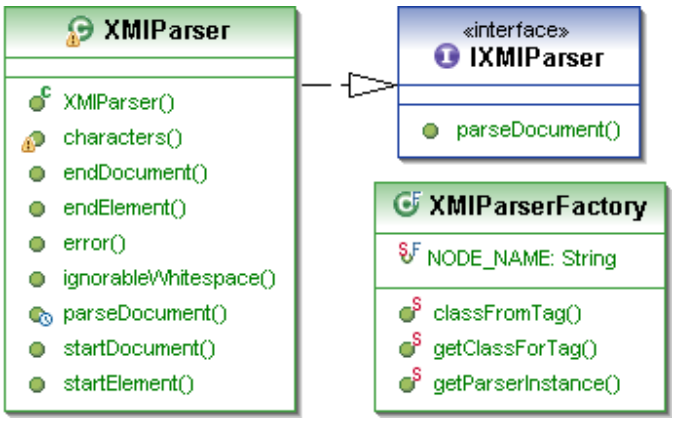
Testinių atvejų generavimo paketas sudarytas tik iš dviejų klasių. Viena iš šių klasių atsakinga už atsitiktinių testinių atvejų generavimą, kita už testinių atvejų generavimą iš UML modelių. Prireikus šis paketas gali būti praplėstas kito tipo klasėmis gebančiomis generuoti testinius atvejus. Testinių atvejų generavimo paketo klasių diagrama pateikta 9 paveiksle.



9 pav. Testinių atvejų generavimo paketo klasių diagrama

3.2.3. XML analizavimo modelis

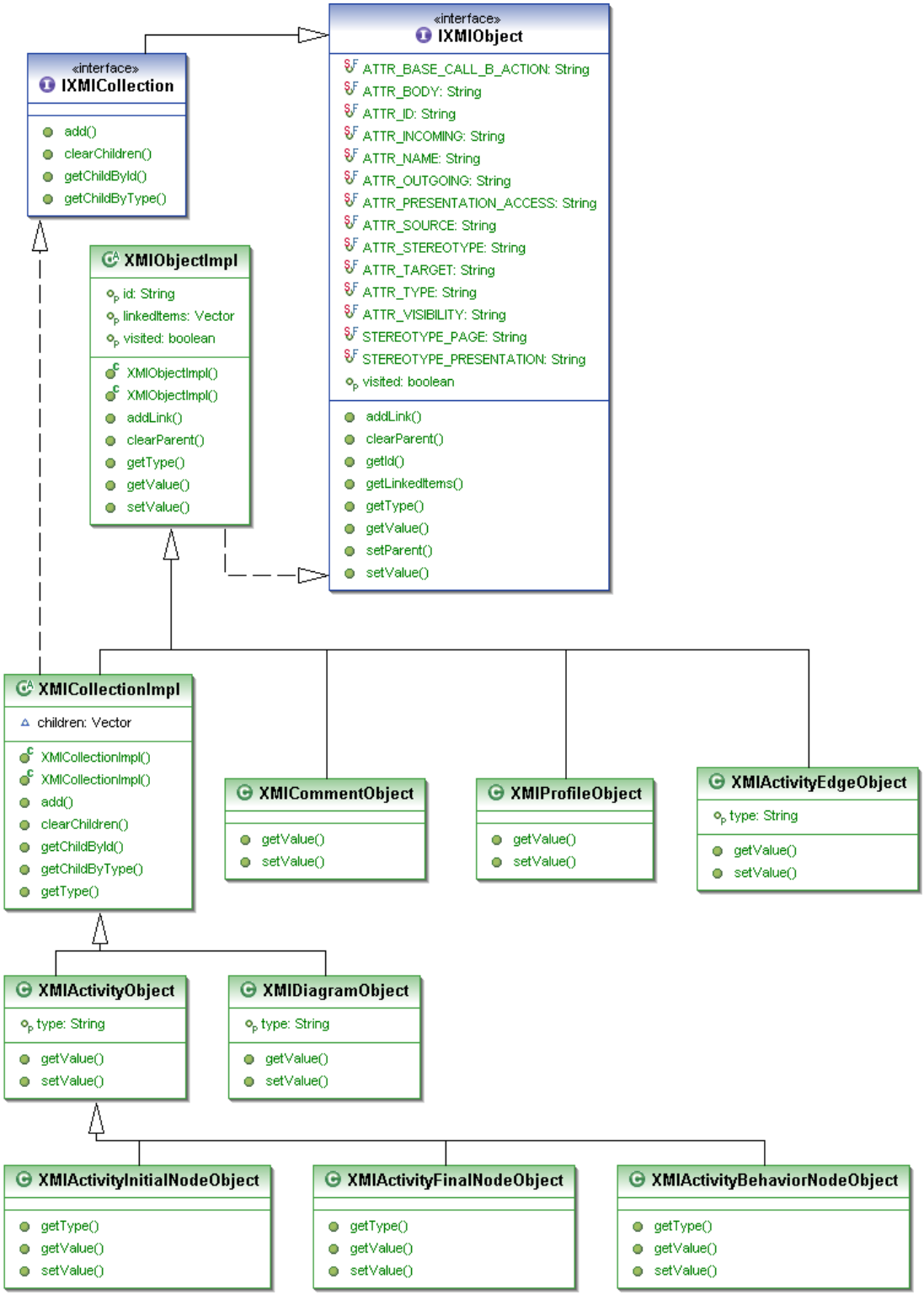
XML analizavimo paketas yra skirtas UML diagramų pateiktų XMI formatu analizatorių realizavimui. Šiame pakete pateikta viena XML analizatoriaus realizacija, tačiau realizuotas abstraktaus fabriko šablonas įgalinantis bet kuriuo sistemos naudojimo metu įtraukti naujo XML analizatoriaus realizaciją skirtą kito tipo įrenginiams. Pavyzdžiui mobiliems telefonams skirto XML analizatoriaus realizacija. XML analizavimo paketo klasių diagrama pateikta 10 paveiksle.



10 pav. XML analizavimo paketo klasių diagrama

3.2.4. XML objektų modelis

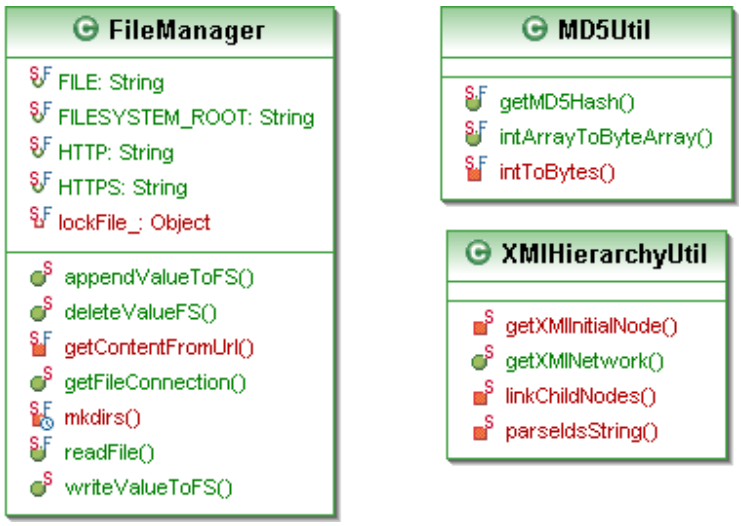
XML objektų pakete yra realizuotos klasės apibrėžiančios XMI faile egzistuojančius objektus, jų savybes ir tarpusavio ryšius. XML objektų paketo klasių diagrama pateikta 11 paveiksle. XML objektų klasės realizuotos panaudojant kompozicijos šabloną.



11 pav. XML objektų paketo klasių diagrama

3.2.5. Bendrojo pobūdžio funkcijų modelis

Bendrojo pobūdžio funkcijų paketas yra skirtas klasėms kurios atlieka bendrojo pobūdžio funkcijas. Tai gali būti failų valdymo funkcijos, užkodavimo-dekodavimo funkcijos, XMI hierarchijos pagalbinės funkcijos ar panašaus pobūdžio funkcijos. Bendrojo pobūdžio funkcijų paketo klasių diagrama pateikta 12 paveiksle.



12 pav. Bendrojo pobūdžio paketo klasių diagrama

Sistemos projekto metu buvo analizuojami ryšiai tarp anksčiau aprašytų komponentų taip sujungiant juos i vieną sistemą. Sistemos apjungimui naudojamos UML sekų ir veiklos diagramos. Sekų veiklos diagramos pateiktos po vieną kiekvienam panaudojimo atvejui. Minėtos diagramos pateiktos prieduose paveiksluose 1-8.

3.3. Vartotojo sąsajos automatinio testavimo karkaso kokybės užtikrinimas

Kokybės užtikrinimo tikslas yra įsitikinti kad sistema veikia korektiškai, t.y. sistemos veikimas atitinka jai keliamus funkcinius ir nefunkcinius reikalavimus. Kad tai atlikti programuotojas ir testuotojai privalo naudoti kuo daugiau priemonių sistemos kokybės užtikrinimui. Norint užtikrinti automatinio vartotojo sąsajos testavimo sistemos kokybę proceso metu atliekami žemiau pateikti veiksmai:

- statinė kodo analizė atliekama programavimo stadijoje;
- komponentų (vienetų) testavimas atliekamas testavimo stadijoje;
- sistemos integracijos testavimas atliekamas testavimo stadijoje.

Statinės kodo analizės metu aptinkamos potencialios sistemos klaidos jau programavimo fazės metu. Paprastai statinė kodo analizė aptinka potencialias nenumatytų atvejų klaidas, amžinų ciklų klaidas ir panašiai. Tokio tipo klaidos gali ir būti neaptiktos normalaus sistemos vartojimo atveju, tačiau tokių klaidų atsiradimas dažnai įtakoja sistemos funkcionalumo pabaigą, o tai gali iššaukti vykdomo darbo nutraukimą be galimybės sugrąžinti prieš tai vykdytą žingsnį.

Programuojant komponentus siekiama išanalizuoti kiekvieno komponento metodo silpnąsias savybes, kraštutinumus ir pan. T.y. analizuojama kiekvieno komponento struktūra. Toks testavimas literatūroje yra vadinamas „baltos dėžės“ testavimu. „Baltos dėžės“ testavimas apibrėžiamas: „Testiniai atvejai gaunami iš programos struktūros. Žinios apie programą naudojamos nustatyti papildomus testinius atvejus. Tikslas yra išbandyti visus programos operatorius ir galimus skaičiavimo kelius.“ [12].

Sistemos integracijos testavimas atliekamas integruojant atskirus komponentus į sistemą. Čia jau testuojami ne atskiri komponentai, o jų sąveika tarpusavyje ir bendras visos sistemos ar jos dalies veikimas. Sistemos integracijos testavimas taip pat atliekamas „baltos dėžės“ principu.

4. VARTOTOJO SĄSAJOS AUTOMATINIO TESTAVIMO KARKASO EKSPERIMENTINIS TYRIMAS

4.1. Eksperimento aprašyme naudojamos sąvokos

- **Programa mutantas**. Programa mutantas yra tam tikros originalios programos kopija turinti vieną pasikeitimą programos išeities tekste [3]. Kitaip tariant tai programa kuri skiriasi nuo savo originalios programos bent viena klaida (vienas pakeitimas programos išeities tekste turi įtakoti bent vieną klaidą). Paprastai tai būna kiek įmanoma smulkesnis pasikeitimas. Programų mutantų originali programa turi apribojimą, kad visos išeities kodo eilutės turi būti reikalingos ir neturėti perteklinio, niekur nenaudojamo kodo, ar kodo, nuo kurio pasikeitimo nepriklauso programos elgsena. Pakeitimu paprastai yra laikomas bet kokio kintamojo priskyrimo pakeitimas, lyginimo operatoriaus pakeitimas kitu ar pan.
- **Mutavimo testavimas.** Programinės įrangos testavimo srityje egzistuoja testavimo metodas, vadinamas mutavimo testavimu (arba mutavimo analize) [13]. Mutavimo testavimo tikslas yra patikrinti testavimo metodo kokybę. Mutavimo testavimo esmė yra sukurti aibę programų mutantų ir juos naudoti kaip testuojamą sistemą. Jei testavimo

sistema tikrindama programą mutantą neaptinka klaidos, galima teigti, kad testavimo sistema yra netobula.

- **Programų mutantų generavimo operatorius.** Tai atributas naudojamas programų mutantų generavimo procese. Programų mutantų generavimo operatorius susideda iš išeities teksto dalies, kurią tikimasi aptikti analizuojant originalios programos išeities tekstą ir išeities teksto dalies, kurią reikia naudoti vietoj ieškomos išeities teksto dalies ją aptikus analizuojant originalios programos išeities tekstą generuojamoje programoje mutante.

4.2. Testinės programos programų mutantų generavimui pasirinkimas

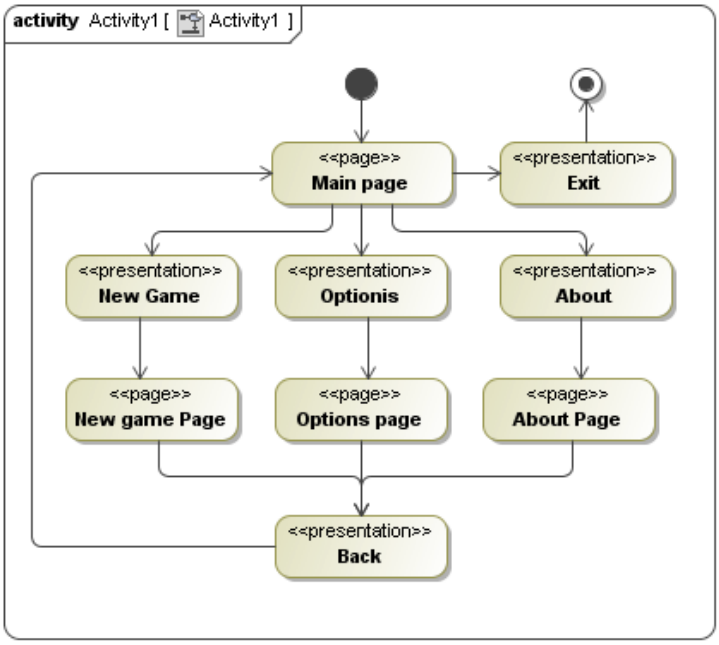
Eksperimentinio tyrimo metu pasirinkta pavyzdinė „Chess King Demo" programa. Ši programa realizuota Java programavimo kalba. Programos išeities teksto eilučių kiekis yra apie 1800. Programa turi 8 klases atsakingas už vartotojo sąsajos atvaizdavimą ir valdymą, kurios sudaro apie 1500 kodo eilučių. Vartotojo sąsajos atvaizdavimui ir valdymui skirtas išeities tekstas šioje programoje sudaro didžiąją dalį programos išeities teksto, nes programos logika yra apribota. Taip yra todėl, kad tai yra pavyzdinė programa. Ši programa eksperimentui pasirinkta dėl kelių pasirinkimą lemiančių veiksnių.

- Ši programa realizuota Java programavimo kalba.
- Ši programa yra atviro kodo pavyzdinė programa.
- Ši programa realizuota nenaudojant standartinių Java kalbos bibliotekose egzistuojančių grafinės vartotojo sąsajos komponentų. Tai komponentai iš paketų javax.swing ir java.awt. Programos, kurios naudoja šiuos grafinius komponentus, nerealizuoja savo vartotojo sąsajos komponentų, tad testavimo apimtis ženkliai sumažėja. Taip atsitinka nes vartotojo sąsajos testavimo karkasas tikrina tik tas klases kurios yra atsakingos už vartotojo sąsajos atvaizdavimą ir valdymą. „Chess King Demo" programos atveju visi grafiniai vartotojo sąsajos komponentai realizuoti atskirose klasėse programoje, tad programų mutantų generavimo galimybės ženkliai padidėja.

Pasirinktos programos keletas ekrano vaizdų kopijų pateikta 13 paveiksle.

13 pav. Programos "Chess King Demo" ekrano vaizdų kopijos

Testuojama sistema „Chess King Demo" buvo adaptuota veikimui su vartotojo sąsajos automatiniu testavimo karkasu. Taip pat testuojamai sistemai buvo realizuotas sistemos UML veiklos modelis pagal anksčiau apibrėžtas taisykles. Testuojamos sistemos veiklos diagrama pateikta 14 paveiksle. Naudojant realizuotą UML veiklos diagramą automatinio vartotojo sąsajos testavimo karkaso pagalba buvo sugeneruoti testuojamos sistemos testiniai atvejai. Šie testiniai atvejai buvo išsaugoti vėlesniam naudojimui testuojant programas mutantus.



14 pav. Testuojamos sistemos „Chess King Demo" vartotojo sąsajos veiklos modelis

4.3. Programų mutantų generavimas

Siekiant ištirti sukurto automatinio testavimo karkaso testavimo kokybę yra stengiamasi patikrinti sistemą su kuo didesne aibe programų mutantų. Programų mutantų generavimui buvo sukurtas automatinis programų mutantų generatorius. Programų mutantų generatorius dirba pagal jam paduotus programų mutantų generavimo operatorius. Programų mutantų generavimo operatorių sąvoka aprašyta šio skyriaus sąvokų sąraše. Programų mutantų generavimui naudojami operatoriai pateikti 1 lentelėje.

1 lentelė. **Programų mutantų generavimui naudojami operatoriai**

Ieškoma išeities teksto dalis	Keičiama išeities teksto dalis
“!=”	“==”
“==”	“!=”
“<=”	“>=”
“>=”	“<=”
“>”	“<“
“<“	“>”
“++”	“--”
“--”	“++”
“||”	“&&”
“&&”	“||”
“-1”	“-2”
“0”	“1”

Programų mutantų generatorius sugeneravo 102 programas mutantas pagal duotus operatorius, analizuojant programą „Chess King Demo”. Verta paminėti, kad programos mutantai buvo generuojami keičiant tik klases, atsakingas už vartotojo sąsajos atvaizdavimą ir valdymą. Šios programos mutantai bus naudojamos eksperimento metu.

4.4. Pirminiai rezultatai

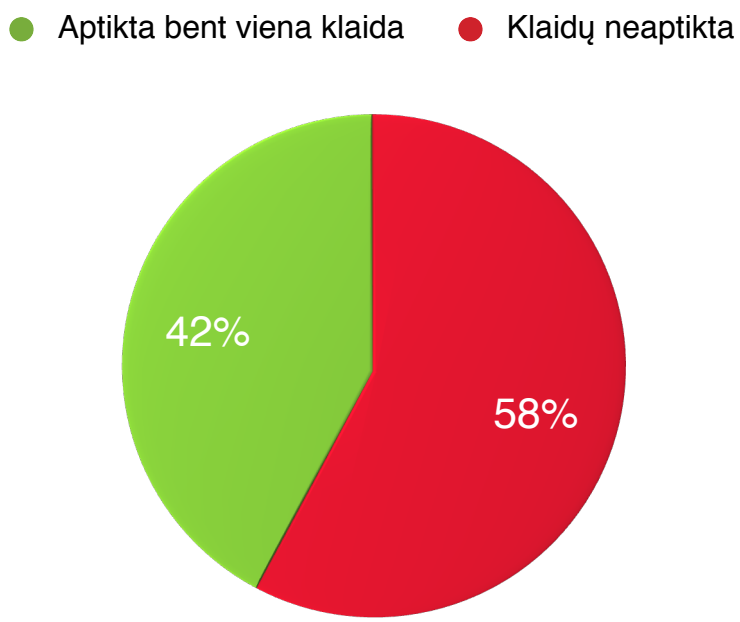
Sumodeliavus pasirinktos programos vartotojo sąsają UML veiklos diagramą, sugeneravus testinius atvejus iš šio modelio ir sugeneravus aibę programų mutantų iš

pasirinktos programos, buvo pradėtas eksperimento vykdymas. Apibendrinti eksperimentui naudojamų programų mutantų kiekiai ir testinių atvejų kiekiai pateikti 2 lentelėje.

2 lentelė. **Eksperimento parametrai**

Eksperimento parametras	Parametro reikšmė
Programų mutantų kiekis	102
Sugeneruotų navigacinių veiksmų kiekis	13
Sugeneruotų testinių atvejų kiekis	598

Visos sugeneruotos programos mutantai buvo testuojamos, naudojant vartotojo sąsajos automatinio testavimo karkasą, vykdant testinius atvejus sugeneruotus iš duoto UML veiklos modelio. Šio eksperimento rezultatai pateikti 15 paveiksle.



15 pav. Pirminiai eksperimento rezultatai

Pirminiai rezultatai yra vidutiniai. Beveik 60-yje% visų sugeneruotų programų mutantų klaidų nebuvo aptikta. Tai yra prastas ir netikėtas rezultatas, tad buvo nagrinėjamos priežastys, kurios lėmė tokį rezultatą. Nustatytos žemiau pateiktos priežastys įtakojančios tokius rezultatus.

- **Klaidos piešimo metoduose.** Buvo nustatyta, kad vartotojo sąsajos automatinio testavimo karkasas neužtikrina programinio kodo, vykdomo piešimo metoduose,

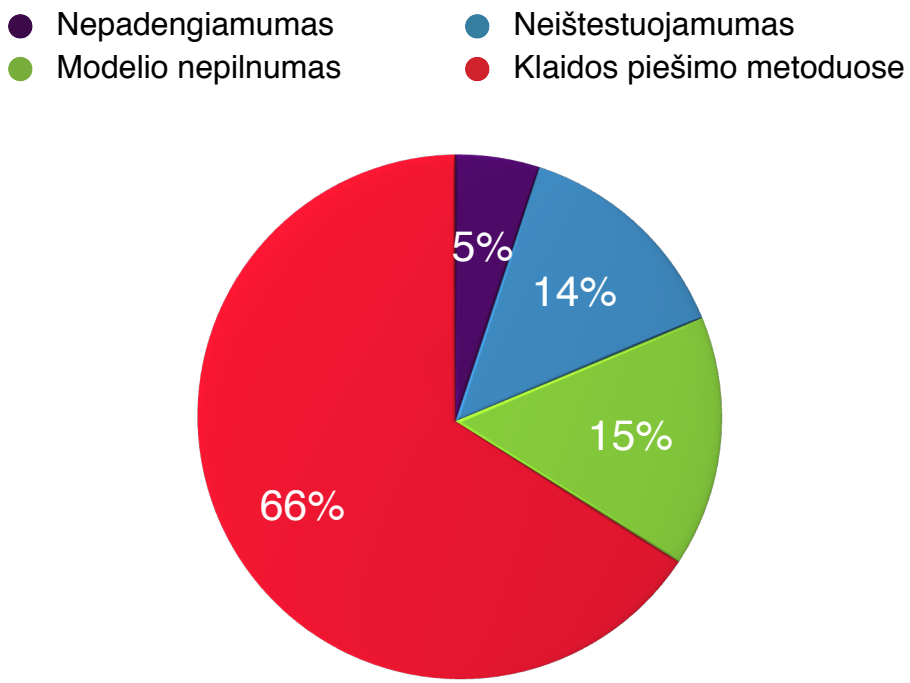
korektiškumo. Karkasas patikrina vartotojo sąsajos komponento atributus, tačiau neatsižvelgia kokie atributai yra naudojami atvaizduojant komponentą ekrane.

- **Modelio nepilnumas.** Buvo nustatyta, kad sudarytas modelis pilnai neapibrėžia testuojamos sistemos. Modelis neatsižvelgia į kai kuriuos tam tikrų grafinių komponentų atributus.

- **Neištestuojamumas [14].** Dėl testuojamos sistemos programinio kodo savybių, kai kurių programos kodo vietų yra neįmanoma ištestuoti naudojant mutavimo testavimo metodą. Tai prieštarauja mutavimo testavimo metodui, tad sistema yra nepajėgi aptikti šiose kodo dalyse atliktų pakeitimų.

- **Nepadengiamumas [15].** Buvo nustatyta kad sistema neatsižvelgia į tas programinio kodo dalis kurios yra vykdomos atsitikus tam tikroms klaidoms kurios yra suvaldomos programos kodo lygyje. Vartotojo sąsajos automatinio testavimo karkasas nepalaiko tokio tipo klaidų imitavimo.

Kiekybinis anksčiau minėtų veiksnių pasiskirstymas procentais pateiktas 16 paveiksle.



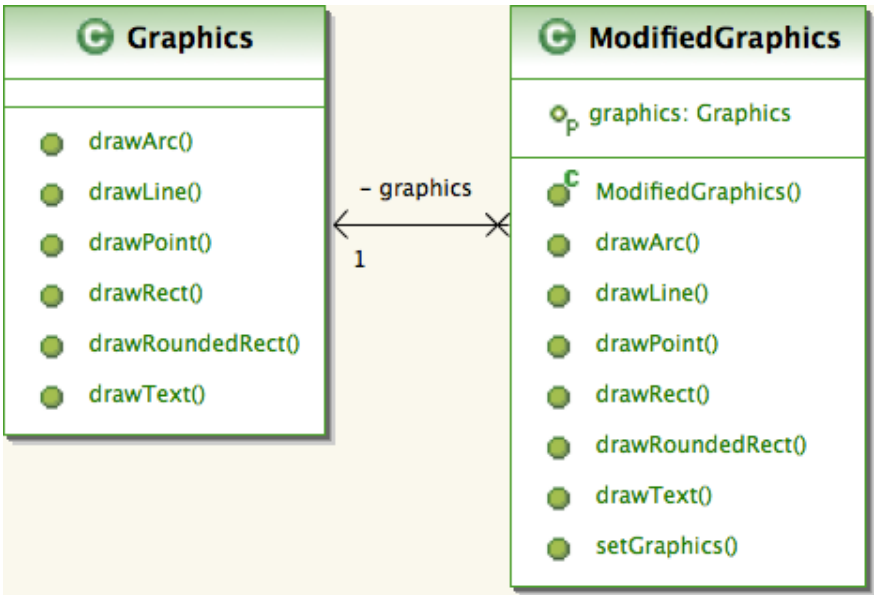
16 pav. Klaidų neaptikimo programų mutantuose eksperimento metu priežastys

4.5. Sistemos tobulinimas

Sistemos tobulinimo etape buvo nagrinėjami galimi anksčiau minėto rezultato priežasčių sprendimai. Šiame skyriuje bus aptartos šios priežastys ir jų sprendimai.

- **Klaidos piešimo metoduose.** Šis trūkumas yra dažniausiai pasitaikantis trūkumas, tad jo išsprendimas yra kritinis. Tai yra automatinio vartotojo sąsajos testavimo karkaso trūkumas. Karkaso struktūra nepritaikyta tikrinti tai kas yra atvaizduojama įrenginio ekrane. Sistema užtikrina, kad kiekvienas grafinis komponentas turi teisingus atributus, tačiau neužtikrina, kad šie atributai yra teisingai naudojami piešimo metoduose. Šiam trūkumui užpildyti buvo pasiūlyti keli karkaso patobulinimo būdai:
    - Papildyti vartotojo sąsajos būseną ekrano vaizdo kopija [16]. Ekrano vaizdo kopija yra vienas iš paprasčiausių ir, atrodytų, tiksliausių piešimo metodo patikrinimo būdų. Tačiau analizuojant šį būdą buvo pastebėta keletas jo trūkumų. Pirma, kad galėtume išsaugoti ekrano vaizdo kopiją testiniame atvejyje ją turime paimti iš UML veiklos modelio. O tai reiškia, kad modeliuojant sistemos vartojo sąsają jau turėtume turėti veikiančią programą, iš kurios galėtume paimti ekrano vaizdo kopijas. Tai pavėlintų testavimo pradžią programinės įrangos kūrimo proceso atžvilgiu. Antra, būtų sunku įsitikinti, kad visi ekrano kopijos taškai yra teisingi. T.y. sunku užtikrinti, kad sistema dabartinėje būsenoje veikia teisingai. O jei ir galime tai užtikrinti, kam tada testuoti? Trečia, programos vaizdų kopijos teisingumas tiesiogiai priklauso nuo testuojamos sistemos vykdymo aplinkos. T.y., priklauso nuo to kokie aplinkos ekrano ar lango dydis, raiška ir pan. Taigi, iš pažiūros patrauklus sprendimo būdas, panagrinėjus atidžiau, tampa netinkamu.
    - Piešimui naudoti modifikuotą piešimo objektą. Piešimas Java programavimo kalboje (bei daugelyje kitų programavimo kalbų) vykdomas naudojant tam tikrus piešimo objektus [17]. Šie objektai paprastai turi realizuotus primityvius piešimo metodus. Tokių metodų pavyzdžiais gali būti taško piešimo metodas, linijos piešimo metodas, stačiakampio piešimo metodas, apskritimo piešimo metodas ir pan. Kad ir koks sudėtingas grafine prasme bebūtų nagrinėjamas grafinis komponentas, jo atvaizdavimas ekrane visada vykdomas naudojant šiuos primityvius piešimo objekto metodus. Siekiant patikrinti kas yra atvaizduojama ekrane užtenka įsitikinti, kad testuojama sistema kviečia reikiamus primityvius piešimo objekto metodus ir, kad jiems perduoda teisingas parametrus. Standartinis piešimo objektas šios užduoties įvykdyti negali, o patikrinti visus šio objekto metodo kvietinius gali būti sudėtinga užduotis, nes šių kvietinių skaičius gali būti ženkliai didesnis už piešimo objekto metodų skaičių. Taigi būtų patogiau vykdyti patikrinimus piešimo objekto metoduose. Kad tai įgyvendinti reikia naudoti modifikuotą piešimo objektą. T.y. naudoti kitą piešimo

objektą kuris naudoja standartinį sistemos piešimo objektą. Taigi šis modifikuotas piešimo objektas būtų pajėgus ne tik vykdyti piešimą, tačiau ir vykdyti vartotojo sąsajos atvaizdavimo korektiškumą. Modifikuoto piešimo objekto klasių diagrama pateikta 17 paveiksle. Kyla klausimas kaip turi būti vykdomi tokie patikrinimai ir kaip tai susieti su vartotojo sąsajos automatinio testavimo karkasu? Visi grafiniai komponentai naudoja tą patį modifikuotą piešimo objektą. Tad galime rinkti informaciją apie viso perpiešimo metu kviestus objektus ir jų parametrus. Surinkę šią informaciją, pateiktą nustatyta forma (tai gali būti bet kokio tipo forma, pvz. teksto pavidalo forma išvardinti metodai ir jų parametrų reikšmės) galime ją konvertuoti į MD5 baito kodų eilutę [18]. Kaip žinia MD5 yra kodavimo algoritmas leidžiantis viena kryptimi užkoduoti duotą įvestį. MD5 algoritmo rezultatas yra baito kodų eilutė iš 16 narių. Dvi skirtingos įvesties reikšmės visada duoda skirtingą rezultatą. Taigi turint šį rezultatą galime jį gražinti kartu su vartotojo sąsajos būsena. Telieka šį papildomą parametrą įtraukti į vartotojo sąsajos UML veiklos modelį ir ‚automatinio vartotojo sąsajos karkaso pagalba, šios dvi reikšmės bus palygintos kartu su visais kitais egzistuojančių grafinių komponentų atributais. Tai nėra paprastas uždavinys testuotojui, nes MD5 baito kodų eilutės generavimas nėra patogus. Tad papildomai reiktų sukurti įrankį padedantį sugeneruoti laukiamą MD5 baito kodų eilutės reikšmę kiekvienam ekrano puslapiui ar formai.



17 pav. Modifikuoto piešimo objekto klasių diagrama

- Naudoti papildomą (alternatyvią) testavimo metodiką piešimo metodams patikrinti. Piešimo metodai nebūtinai turi būti patikrinti automatinio vartotojo sąsajos testavimo karkaso. Šie metodai gali būti testuojami naudojant alternatyvią testavimo

metodiką. Pavyzdžiui, tai gali būti vienetų testavimo pagalba atliekamas šių metodų testavimas. Alternatyvios testavimo metodikos uždavinys būtų patikrinti piešimo metodų teisingumą ir rezultatą išsaugoti kaip grafinės vartotojo sąsajos komponento atributą (tai gali būti paprastas “drawTest=true/false” atributas). Tokiu atveju vartotojo sąsajos automatinio testavimo karkasas patikrintų ar ši reikšmė yra teisinga. Taigi vartotojo sąsajos automatinio testavimo karkaso nereikėtų keisti. Kad ši metodika veiktų reiktų patobulinti vartotojo sąsajos UML veiklos diagramą prie kiekvieno grafinio vartotojo sąsajos komponento pridėjus minėtą atributą (konkrečiu atveju “drawTest=true”). Pergeneravus testinius atvejus iš naujo testuojamą sistemą butų galima pakartotinai testuoti. Dėl šio sprendimo įgyvendinimo paprastumo, šis būdas buvo pasirinktas kaip patobulinimas pakartotiniam eksperimentui.

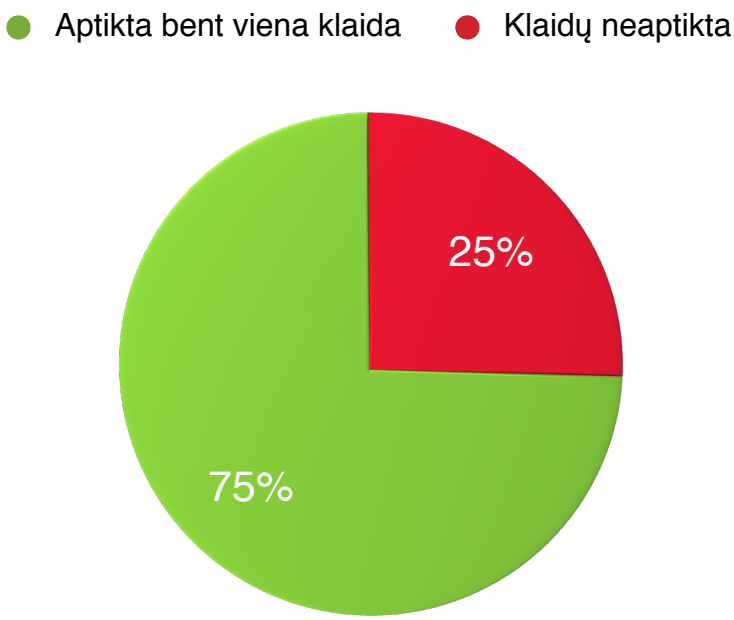
- **Modelio nepilnumas.** Šis atvejis nesusijęs su nagrinėjamu vartotojo sąsajos automatinio testavimo karkasu. Tai konkrečiu atveju sudaryto vartotojo sąsajos UML veiklos modelio klaida, tad ši klaida nebus nagrinėjama kaip vartotojo sąsajos automatino testavimo karkaso patobulinimas.

- **Neištestuojamumas [14].** Šis atvejis nesusijęs su nagrinėjamu vartotojo sąsajos automatiniu testavimo karkasu. Tai konkrečios programos neatitikimo mutavimo testavimo metodikai klaida, tad ši klaida nebus nagrinėjama kaip vartotojo sąsajos automatinio testavimo karkaso patobulinimas.

- **Nepadengiamumas.** Kaip buvo aptarta anksčiau tokio tipo klaidų vartotojo sąsajos automatinio testavimo karkasas neaptinka. Taip yra todėl kad šių klaidų aktyvavimas susijęs su sistemos logika, o ne su vartotojo sąsaja. Jei testuotojas nori patikrinti tokias kodo dalis ir jei šios dalys susijusios su vartotojo sąsaja, tokių kodo dalių iškvietimas turi būti dirbtinai imituojamas. T.y. į testuojamą sistemą turi būti įvedamas tokio kodo aktyvavimo grafinis komponentas. Pavyzdžiui tai gali būti papildomas mygtukas iššaukiantis klaidos apie tam tikrus veiksmus (susisiekimą su serveriu, failo valdymu ar pan.) pranešimą. Dėl minėtų priežasčių ši klaida nebus nagrinėjama kaip vartotojo sąsajos automatinio testavimo karkaso patobulinimas.

Atlikus anksčiau minėtus pakeitimus ir pakeitus pasirinktos programos vartotojo sąsajos UML veiklos diagramos modelį, pakartotinai sugeneravus testinius atvejus iš šio patobulinto modelio ir pakartotinai sugeneravus aibę programų mutantų iš pasirinktos programos buvo pradėtas pakartotinis eksperimento vykdymas. Visos sugeneruotos programos mutantai buvo testuojamos naudojant automatinį vartotojo sąsajos testavimo

karkasą vykdant testinius atvejus sugeneruotus iš duoto UML veiklos modelio. Šio pakartotinio eksperimento rezultatai pateikti 18 paveiksle.



18 pav. Pakartotinio eksperimento rezultatai

4.6. Vartotojo sąsajos automatinio testavimo karkaso apribojimai

Kiekviena kuriama sistema turi savo taikymo sritį ir yra naudojama priimant tam tikrus tos sistemos apribojimus. Realizuotas vartotojo sąsajos automatinio testavimo karkasas yra ne išimtis.

Automatinio vartotojo sąsajos testavimo karkaso trūkumai:

- Karkasas geba testuoti tik tas sistemas, iš kurių galime gauti informaciją apie sistemoje naudojamus grafinius komponentus ir jų atributus. Tai yra pagrindinis sistemos trūkumas.
- Karkasas, testinių atvejų generavimo metu, išrenka visus galimus navigacinius kelius programoje. Šių kelių kiekis didelėse programose gali būti ypač didelis, tad testavimas, tokiais atvejais, užtruktų kur kas ilgiau negu įprastu atveju. Taigi sistema turėtų pasiūlyti galimybę sužymėti kiekvieno elemento prioritetus ir šiems elementams teikti pirmenybę. Vartotojas turėtų turėti galimybę pasirinkti testinių atvejų generavimo funkciją atrenkant tik nurodyto prioriteto elementus.
- Kaip trūkumą galima paminėti ir tai, kad testuojamą sistemą reikia adaptuoti darbui su automatiniu vartotojo sąsajos testavimu nors ir šie adaptavimo veiksmai yra paprasti ir dokumentuoti.

- Automatinio vartotojo sąsajos testavimo karkasas geba pilnai padengti testuojamą vartotojo sąsają, tačiau nesuteikia jokių galimybių nagrinėti sistemą giliau. Šis funkcionalumas nėra susijęs su vartotojo sąsajos testavimu, tačiau būtent tokio funkcionalumo dažnai tikisi vartotojo sąsajos testuotojai.

5. IŠVADOS

1. Vartotojo sąsajos testavimas vis dar yra brangus ir lėtas procesas. Egzistuojančios vartotojo sąsajos testavimo automatizavimo priemonės yra retai pritaikomos praktikoje, todėl dažnai testavimą atlieka žmogus. Iš to seka, kad rinkoje yra jaučiamas pritaikomų vartotojo sąsajos testavimo automatizavimo įrankių poreikis.
2. Darbo metu buvo pasiūlyta naudoti UML veiklos modelius vartotojo sąsajos atvaizdavimui. Siūlomų modelių elementų stereotipai skiriasi nuo standartinių veiklos modelių, tačiau šie modeliai gali būti kuriami naudojant bet kuriuos egzistuojančius UML įrankius. Darbo metu atlikta analizė parodė, kad naudojant šiuos modelius galima pilnai aprašyti vartotojo sąsajos navigaciją ir vartotojo sąsajos elementų struktūrą.
3. Darbo metu buvo sukurtas ir aprobuotas vartotojo sąsajos automatinio testavimo karkasas, naudojantis anksčiau minėtus UML veiklos modelius. Ši sistema vadinama karkasu nes yra integruojama į testuojamą sistemą. Karkasas geba automatiškai generuoti testinius atvejus iš duotų UML veiklos modelių ir geba automatiškai vykdyti sugeneruotus testinius atvejus.
4. Atliktas eksperimentas parodė, kad vartotojo sąsajos automatinio testavimo sistema turi keletą trukumų. Šie trūkumai buvo išanalizuoti ir pasiūlyti jų sprendimo būdai. Pritaikius vieną iš pasiūlytų sprendimo būdų buvo pakartotinai atliktas eksperimentas. Pakartotinis eksperimentas parodė, kad pasiūlytas sprendimas buvo naudingas ir sistemos klaidų aptinkamumas padidėjo apie 35% ir siekia 75%.
5. Sukurtą sistemą galima naudoti tolimesniuose eksperimentuose bei naudoti testuojant kuriamos sistemos vartotojo sąsają. Sistemos praplėtimo galimybės pateiktos 4.6 skyrelyje.
6. Šio darbo pagrindu buvo paruoštas apžvalginis straipsnis ir skaitytas pranešimas 14-ojioje magistrantų ir doktorantų konferencijoje „Informacinės technologijos” Vilniaus universiteto Kauno humanitariniame fakultete. Taip pat paruoštas detalus siūlomo metodo analizės straipsnis ir pristatymas 16-ojioje tarptautinėje informacijos ir programinės įrangos technologijų konferencijoje (angl. 16th International Conference on Information and Software Technologies) Kauno technologijos universitete.

LITERATŪRA

1. Java SE Technologies at Glance [Žiūrėta 2009 11 08], prieiga per internetą <http://java.sun.com/javase/technologies/>
2. Journal of Object Technology - Test early, Test often Dr. John D. McGregor [Žiūrėta 2009 03 16], prieiga per internetą <http://www.jot.fm/issues/issue_2007_05/column1/>
3. Roger T. Alexander, James M. Bieman - Mutation of Java Objects. IEEE Int. Symp. Software Reliability Engineering (ISSRE) 2002
4. Marathon Integrated Testing Environment [Žiūrėta 2008 11 08], prieiga per internetą <http:// www.marathontesting.com/Home.html>
5. Unit testing [Žiūrėta 2008 11 08], prieiga per internetą <http://en.wikipedia.org/wiki/Unit_test>
6. JUnit [Žiūrėta 2008 12 03], prieiga per internetą <http://junit.sourceforge.net/>
7. UML Activity Diagrams: Detailing User Interface Navigation [Žiūrėta 2008 11 11], prieiga per internetą <http://www.ibm.com/developerworks/rational/library/4697.html>
8. XMI Metadata Interchange Specification [Žiūrėta 2008 11 10], prieiga per internetą <http://www.omg.org/spec/XMI/2.1.1/>
9. Mark Utting and Bruno Legeard - Practical Model-Based Testing: A Tools Approach. ISBN 978-0-12-372501-1, Morgan-Kaufmann 2007
10. Marlon Vieira, Johanne Leduc, Bill Hasling, Rajesh Subramanyan, Juergen Kazmeier - Automation of GUI Testing Using a Model-driven Approach. International Conference on Software Engineering 2006
11. Test Case [Žiūrėta 2009 06 17], prieiga per internetą <http://en.wikipedia.org/wiki/Test_case>
12. British Computer Society Specialist Interest Group in Software Testing - Standard for Software Component Testing. British Computer Society, SIGIST, 2001
13. Mattias Bybro - A Mutation Testing Tool for Java Programs. Stockholm, 2003
14. Software testability [Žiūrėta 2010 03 12], prieiga per internetą http://en.wikipedia.org/wiki/Software_testability
15. Lasse Koskela - Introduction to Code Coverage [Žiūrėta 2010 03 12], prieiga per internetą http://www.javaranch.com/journal/2004/01/IntroToCodeCoverage.html
16. GUI software testing [Žiūrėta 2009 01 19], prieiga per internetą <http://en.wikipedia.org/wiki/GUI_software_testing#Running_the_test_cases>

17. Java 2D: Graphics in Java 2 [Žiūrėta 2010 03 06], prieiga per internetą <http:// www.coreweb programming.com/PDF/ch10.pdf>
18. The MD5 (Message-Digest algorithm 5) [Žiūrėta 2008 11 10], prieiga per internetą <http:// www.bullzip.com/md5/md5.htm>

Program User Interface Automated Testing Based on UML Models

SUMMARY

In many cases, testing is an essential, but time and resource consuming activity in the software development process. In the case of model-based development, test construction and test execution can be partially automated. As the application size is constantly growing, the need for automated testing frameworks comes into place, particularly frameworks for automated testing of user interaction and graphical user interface.

This document describes an implementation of the GUI test generator framework based on UML models where specific UML activity diagrams are used for test case generation. It is not a usual case to use UML activity diagrams for UI modeling. However the existing stereotypes of activity diagram elements are not suitable for UI modeling. With usual activity diagram it is complicated to define buttons, containers, pages and other UI elements in the diagram and find differences between them. Even more complicated is to model the navigation of the testing application. Using this approach the UI can be defined in a set of UI elements along with a set of UI navigation elements. This is an optimal and suitable approach in most cases.

This document describes an implementation of the automated GUI tests runner framework as well. This framework is able to run the given application in test mode using the previously generated test cases. The framework collects all the information about each test case results and provides it to the tester.

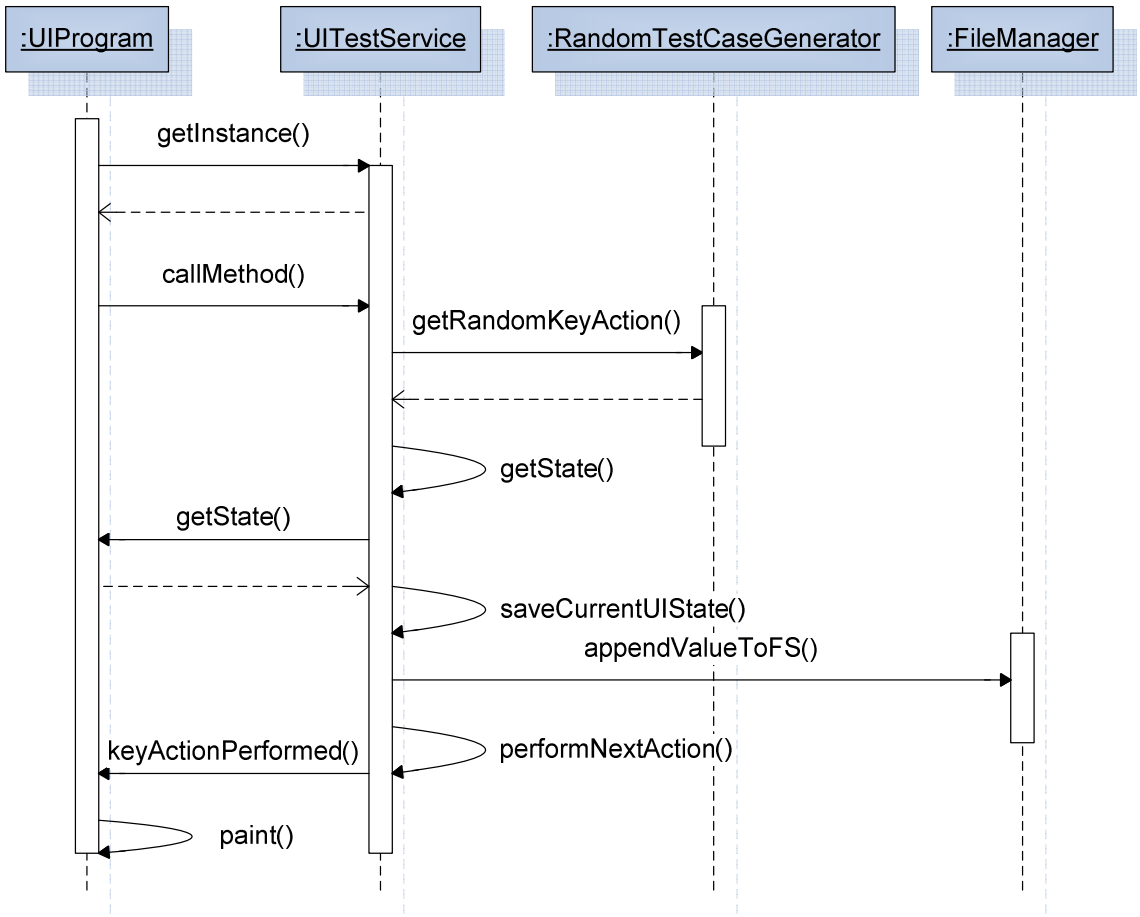
Future improvements:

- Find the solution to identify a set of optimal paths covering all elements in the diagram
- Implement mechanism that allows user to set priorities to the pages in diagram and inspect the most important pages only while generating the test cases
- Improve test case generator by implementing the automated notification of input fields and ability to generate user specific inputs to that input field.

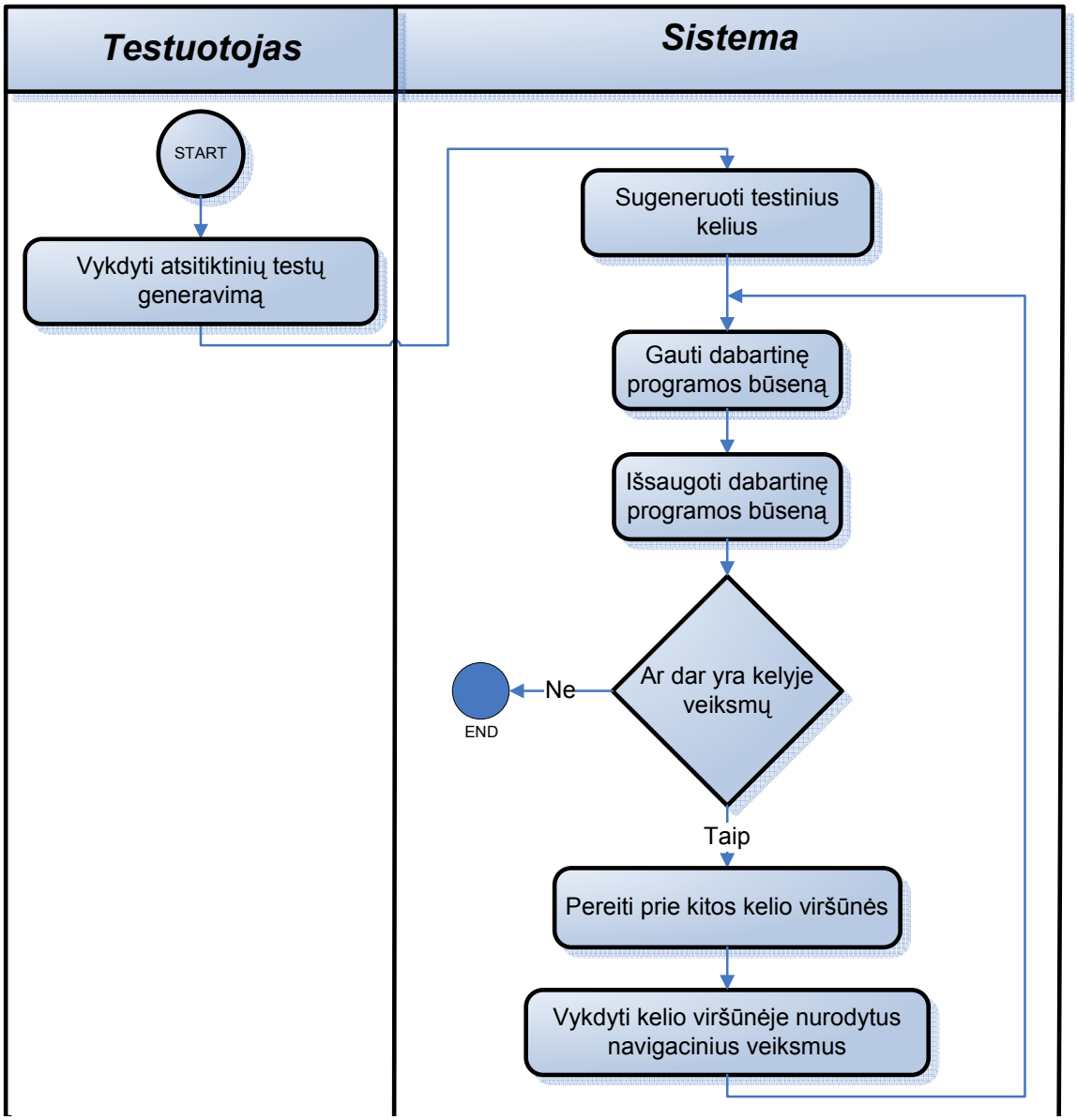
TERMINŲ IR SANTRUMPŲ ŽODYNAS

Java SE (J2SE)	Standartinis Java programavimo kalbos leidimas
MD5	Žinutės santraukos algoritmas plačiai naudojamas kriptografijoje
UML	Modeliavimo ir specifikacijų kūrimo kalba, skirta specifikuoti, atvaizduoti ir konstruoti objektiškai orientuotų programų dokumentus
XML	Bendros paskirties duomenų struktūrų bei jų turinio aprašomoji kalba
XMI	UML objektų aprašomoji kalba pagrįsta XML kalbos struktūra
Programa mutantas	Programa mutantas yra tam tikros originalios programos kopija turinti vieną pasikeitimą programos išeities tekste [3]
Mutavimo testavimas	Programinės įrangos testavimo srityje egzistuoja testavimo metodas, vadinamas mutavimo testavimu (arba mutavimo analize) [13]. Mutavimo testavimo esmė yra sukurti aibę programų mutantų ir juos naudoti kaip testuojamą sistemą.

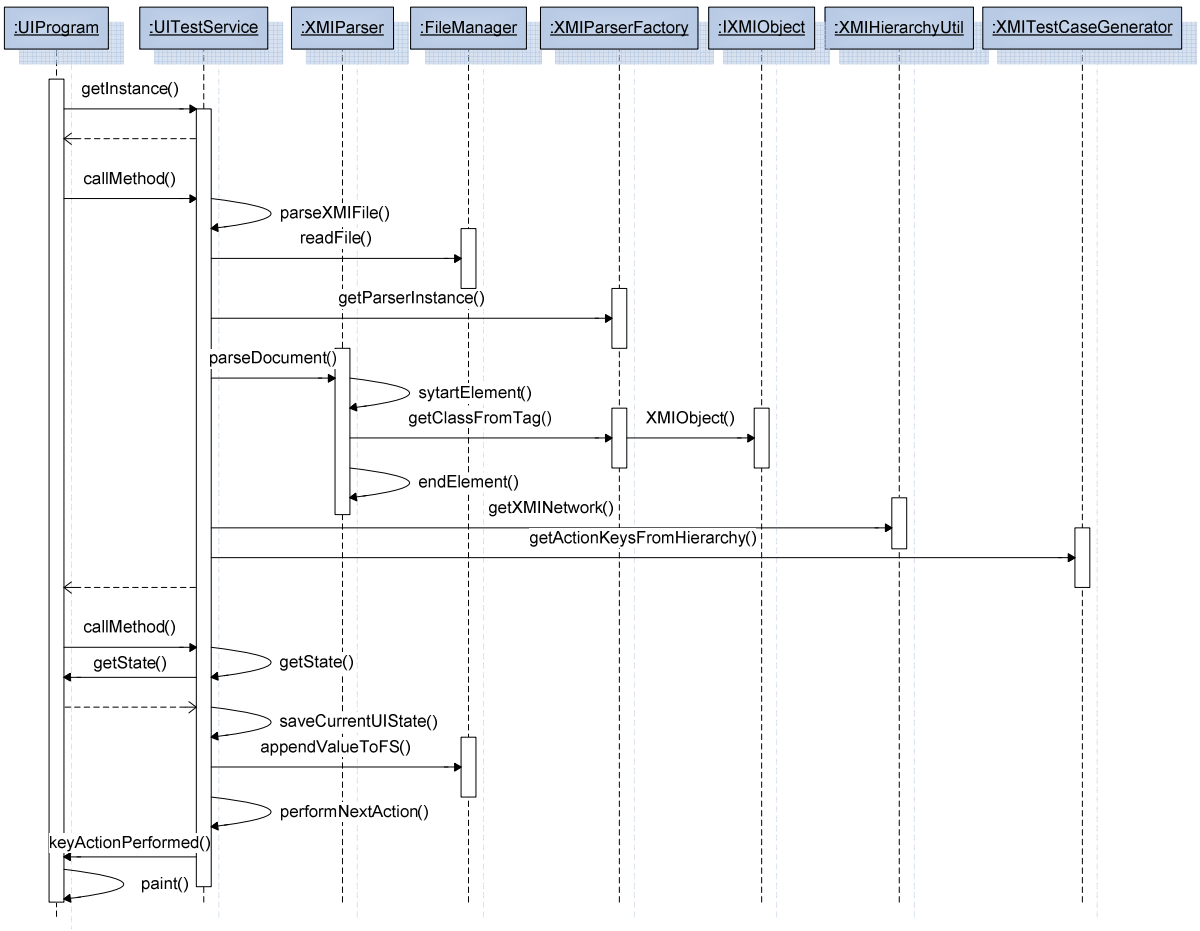
PRIEDAI



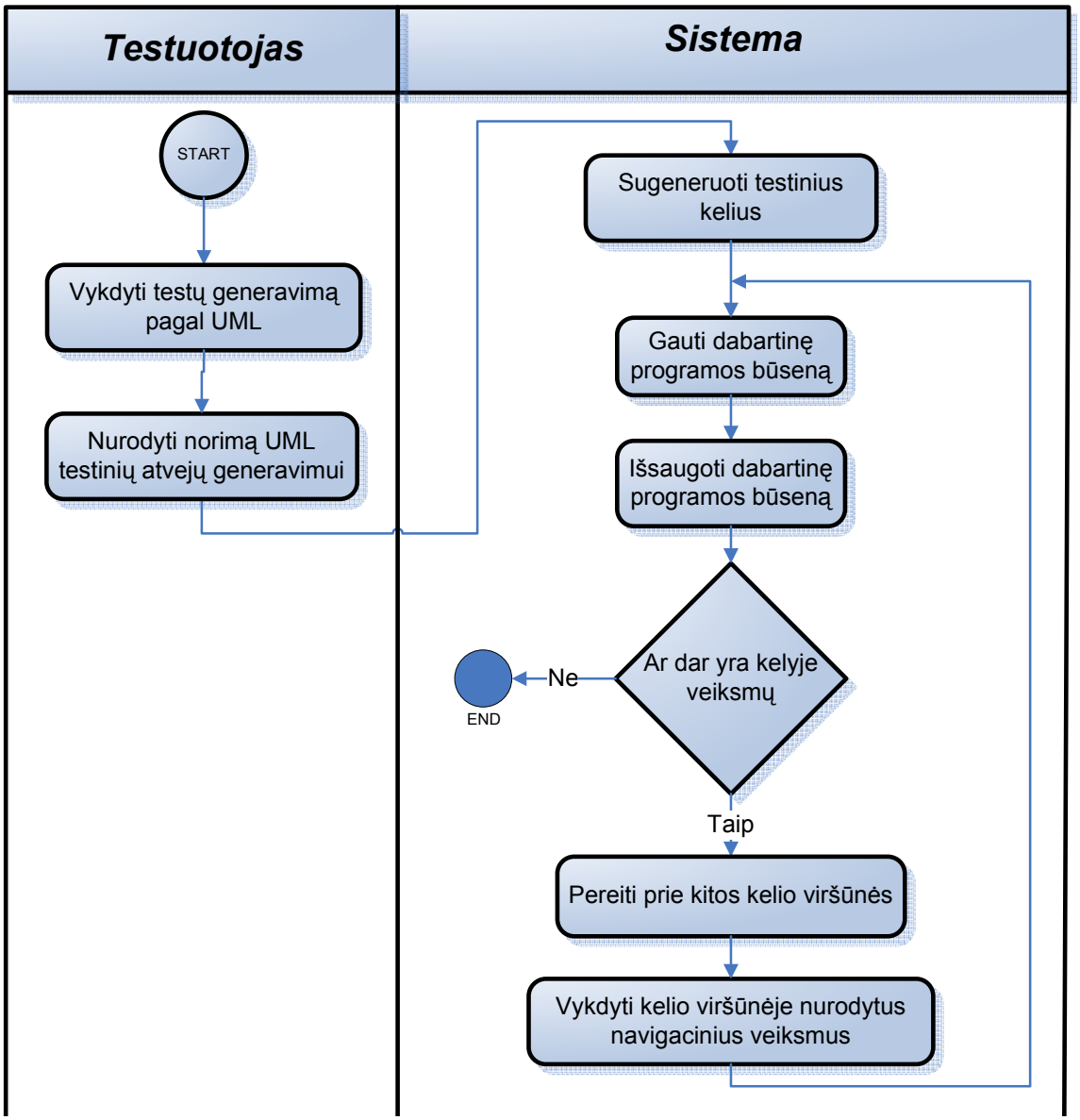
1 pav. Atsitiktinių testinių atvejų generavimo sekų diagrama



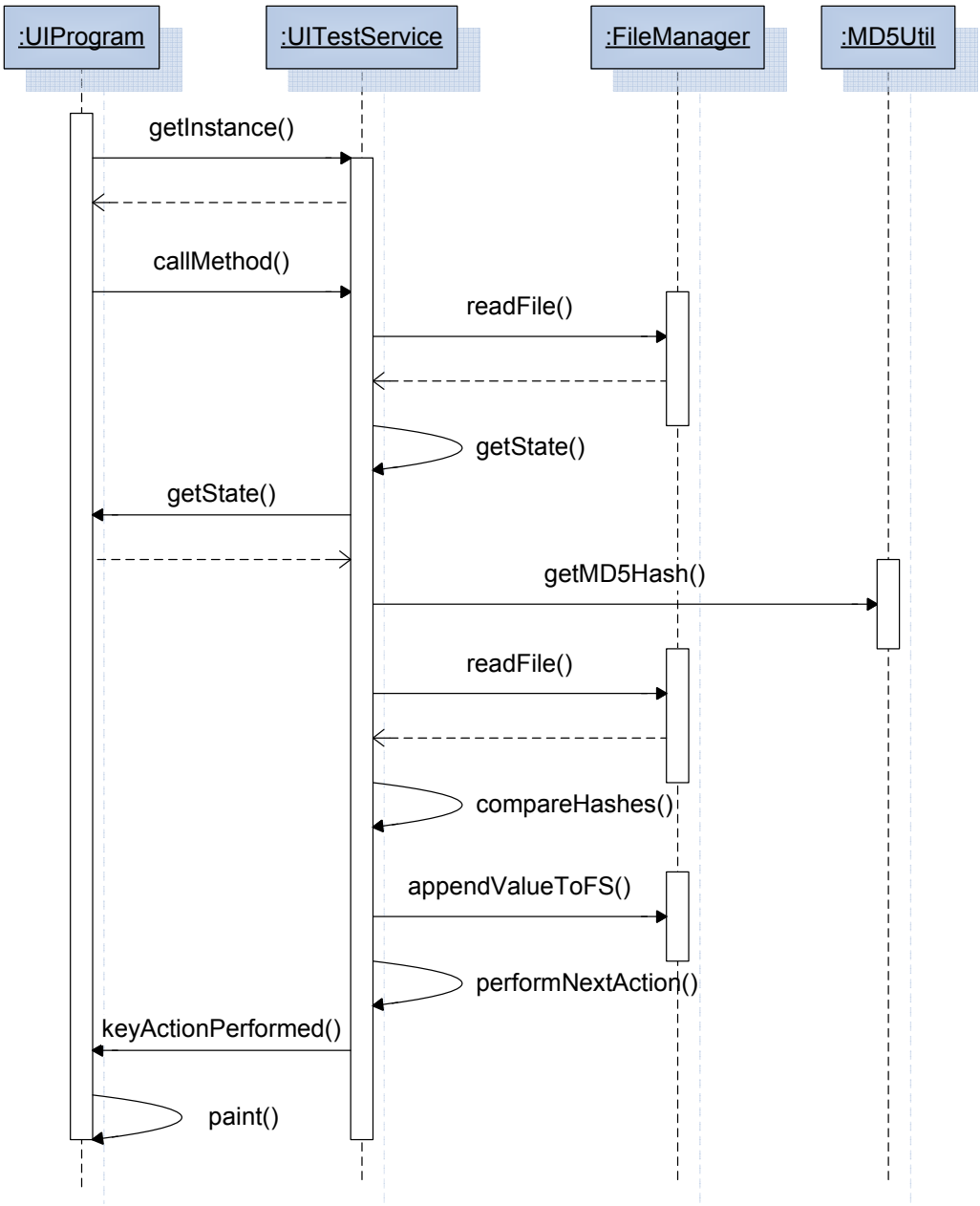
2 pav. Atsitiktinių testinių atvejų generavimo veiklos diagrama



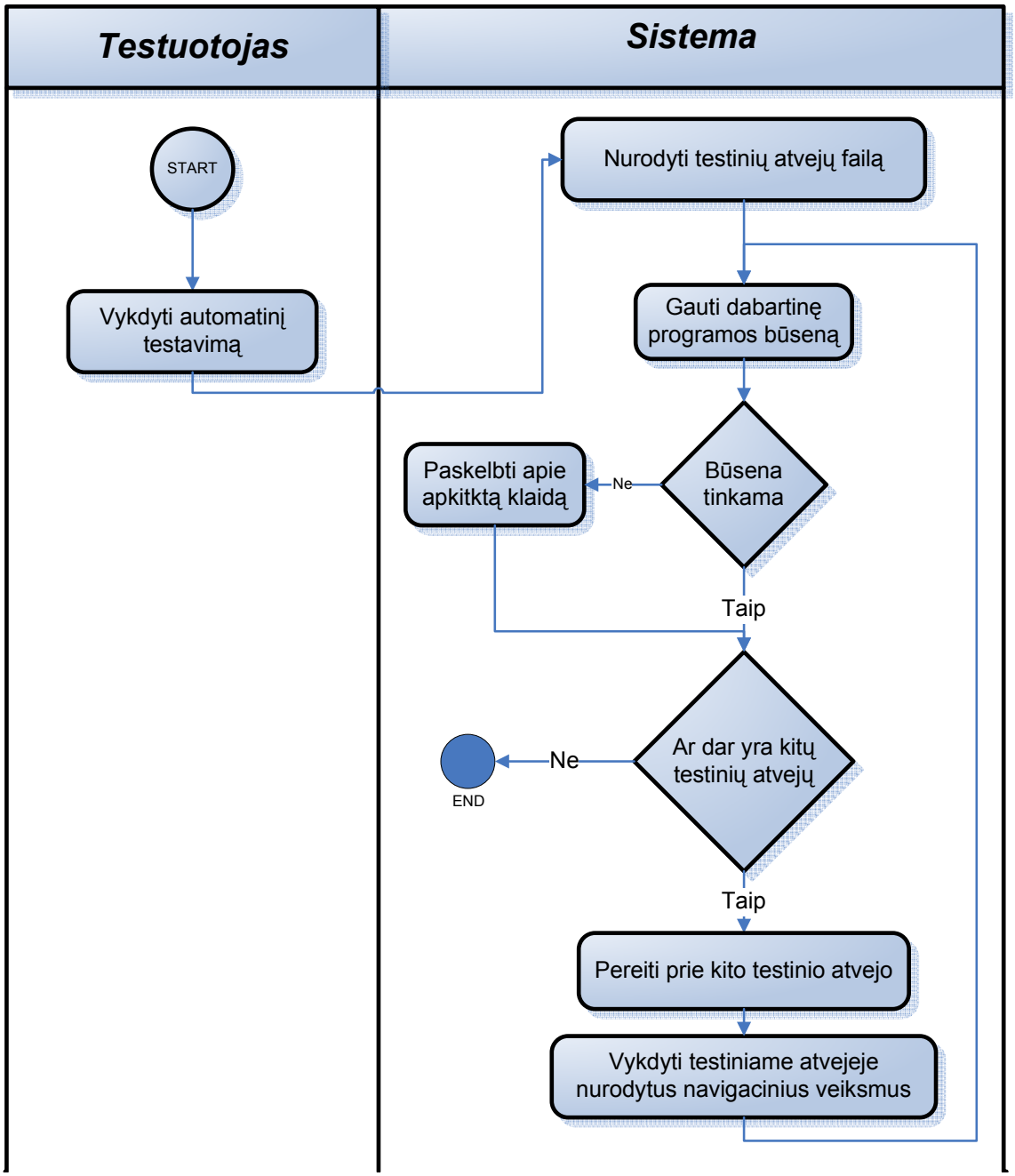
3 pav. Testinių atvejų generavimo iš UML modelių sekų diagrama



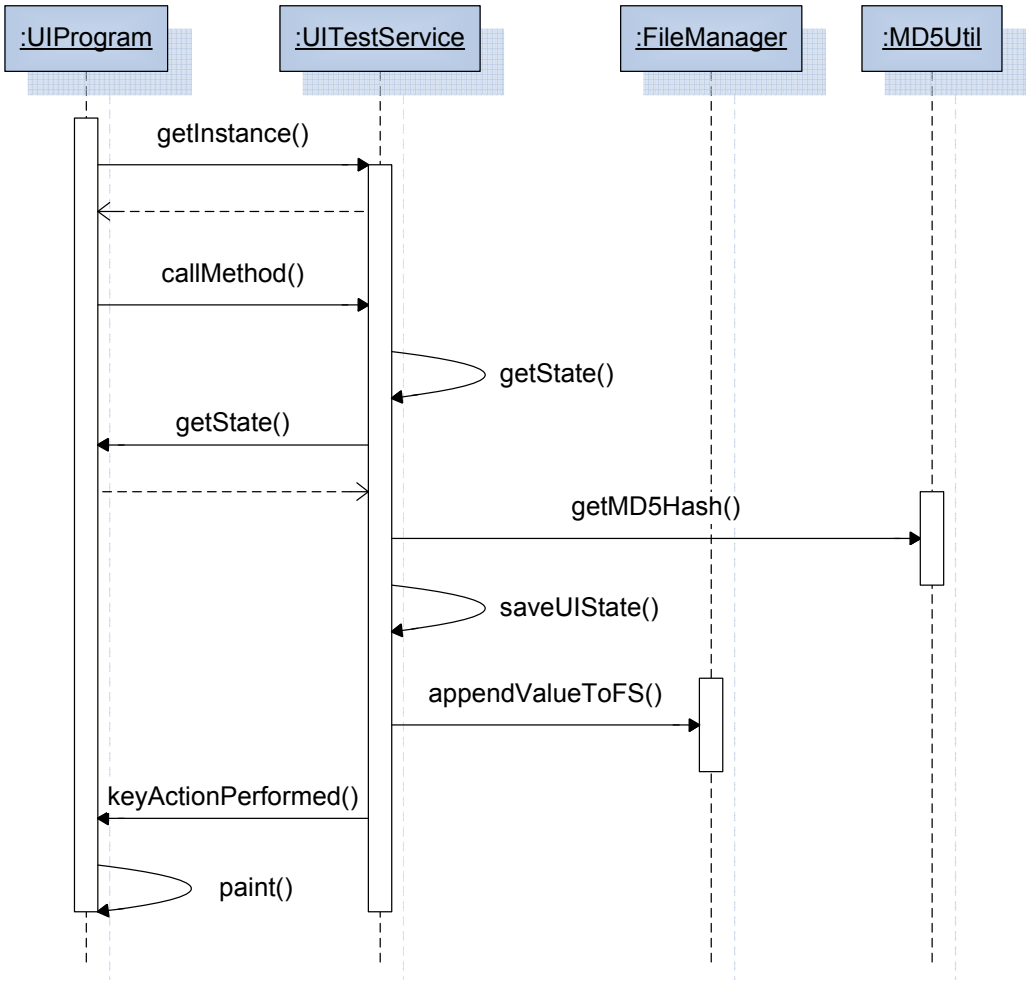
4 pav. Testinių atvejų generavimo iš UML modelių veiklos diagrama



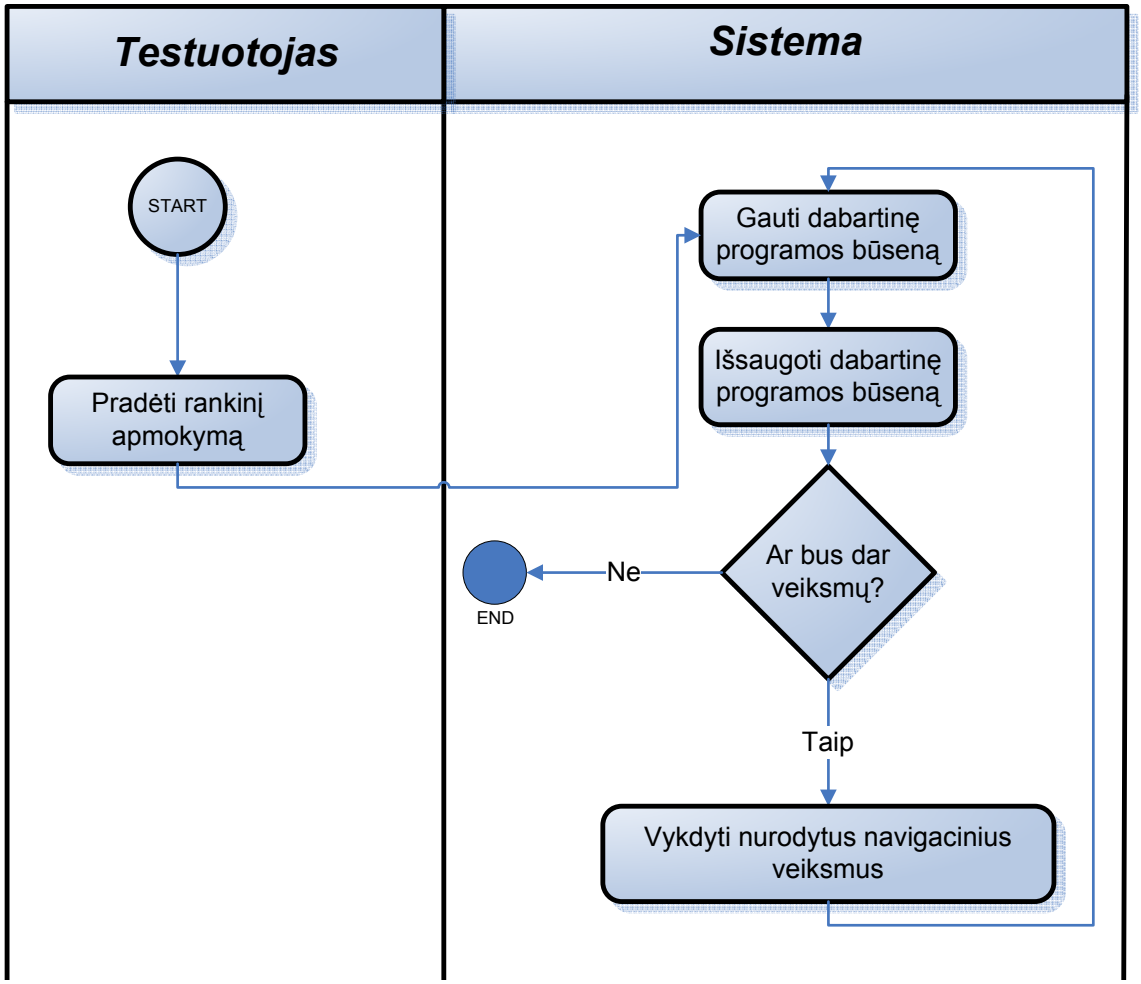
5 pav. Automatinio testavimo sekų diagrama



6 pav. Automatinio testavimo veiklos diagrama



7 pav. Rankinio apmokymo sekų diagrama



8 pav. Rankinio apmokymo veiklos diagrama