VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMOS KATEDRA

# Klaidų valdymo modelio taikymo tyrimas ir vertinimas

An Analysis And Evaluation Of Error Handling Framework‘s Practical Application

Magistro baigiamasis darbas

| | | |
|---|---|---|
| Atliko: | Marius Ašmonas | (parašas) |
| Darbo vadovas: | j.m.d. Mindaugas Plukas | (parašas) |
| Darbo recenzentas: | a. T. Savičius | (parašas) |

Vilnius
2008

# SANTRAUKA LIETUVIŲ KALBA

Niekam jau nebe paslaptis, kad šiai dienai rašyti klaidoms atsparias programas darosi vis sunkiau ir sunkiau, nes programų apimtys didėja, o realizacija sudėtingėja. Naudojamų programavimo kalbų įrankiai nesuteikia pakankamo pajėgumo, kurio taip trūksta, norint turėti greitą, efektyvų ir automatizuotą būdą tvarkytis su klaidomis, atstatyti sistemos darbą ir atlikti kitus reikalingus veiksmus. Dėl to pagrindinis šio darbo tikslas yra pristatyti tam tikrą priemonę, kuri būtų naudinga, projektuojant, kuriant ir testuojant didesniu atsparumu klaidoms pasižyminčias programų sistemas. Toji priemonė – tai klaidų valdymo modelis („*Error-Handling Framework*“), kurio pagrindinė paskirtis yra orientuota į efektyvesnį su klaidų valdymu susijusių sprendimų projektavimą ir realizavimą. Praktiniam tokio modelio panaudojamumui iliustruoti yra naudojama pasirinkta *ATM* bankomatų funkcionalumą simuliuojanti sistema („*ATM Simulation System*“), kuri darbo eigoje yra transformuojama į nuosavą klaidų valdymo posistemę turinčią sistemą. Kiekviename darbo žingsnyje gauti rezultatai yra atitinkamai įvertinami, įvardinant stipriąsias ir silpnąsias jų puses. Darbo pabaigoje yra pateikiamas būdas, kuris leidžia bent empiriškai nustatyti naujosios sistemos atsparumo klaidoms laipsnį.

# SANTRAUKA ANGLŲ KALBA
# (SUMMARY)

*Nowadays it‘s becomming more and more difficult to write programs that behave correctly in the presence of run-time errors. Existing programming language features often provide poor support for executing clean-up code and for restoring normal state in exceptional situations. The main aim of this work is to establish a certain classification that can serve as a tool for understanding how to develop more efficient and more robust software systems. It presents a unified pattern called the Error-Handling Framework, which is targeted directly at different error handling design and implementation issues. Additionally, in order to show how this pattern can be achieved in practical application, an ATM Simulation System is analyzed and transformed into a system that is able to handle the load of various planned and unplanned exceptional situations. Each step of the way results are carefully evaluated by providing the list of their stronger and weeker sides. Finally, an evaluation method showing how error-safe the new ATM simulation system is will be provided along with the concluding results.*

# TURINYS

# ĮVADAS

Prieš pradedant kalbėti apie patį darbą, pirmiausia, manau, būtų tikslinga pristatyti bent jau pagrindinius šiam darbui keliamus tikslus, kurie ir turėtų atskleisti, ką yra ruošiamasi daryti ir ko bus siekiama. Taigi darbo tikslai šiuo atveju būtų tokie:

1. Ištirti klaidų valdymo modelio praktinio taikymo galimybes, kiekvienai jį sudarančiai probleminei sričiai konkrečiame dalykinės srities kontekste pateikiant po projektavimo sprendimą.

2. Iš pasirinktos pavyzdinės[1] darbo analizuojamosios sistemos gauti naująją klaidoms atsparią tos sistemos versiją.

3. Rasti būdą, kaip įvertinti pasiekto analizuojamosios sistemos architektūros atsparumo klaidoms laipsnį.

Joks klaidų valdymo modelis pats savaime paprastai nepateikia jokių konkrečių būdų, idėjų, priemonių ar realizacijų, kaip turėtų būti tiksliai projektuojama klaidoms atspari programų sistema, nes tokių sprendimų priėmimas dažniausiai priklauso nuo kiekvieno atvejo individualiai. Vis dėlto toks modelis leidžia susidaryti bendrą vaizdą, kas turėtų būti numatyta projektuojamoje programų sistemos architektūroje. Taigi šiame darbe į klaidų valdymo modelį bus žiūrima labiau kaip į gerą praktiką, kurią yra siūloma taikyti, siekiant programų sistemoje turėti efektyvų, nepriklausomą ir pakartotinai panaudojamą į klaidų valdymą orientuotą funkcionalumą.

Kas šiame darbe galėtų būti inovatyvaus? Na, pirmiausia tai turbūt tikrai ne pati idėja programų sistemų kūrime naudoti klaidų valdymą. Teisingesnis atsakymas pirmiausia būtų sudaryti ir pasiūlyti kiek įmanoma bendresnį, pilnesnį ir išbaigtesnį klaidų valdymo modelį, kurį būtų galima naudoti kaip pagrindą įvairiose programų sistemų kūrimo praktikose ir kuris ateityje galbūt netgi galėtų pretenduoti ir į sudedamąsias kokio nors *klaidų valdymo kūrimo proceso* dokumentacijos dalis. Visų antra, kartu su šiuo modeliu pasiūlyti ir vieną iš galimų individualiai parengtų klaidų atsparumo vertinimo metodikų.

[1] Sistemos architektūra yra vadinama pavyzdine dėl to, kad ji yra tik kaip konkretus apčiuopiamas gyvenimiškas pavyzdys, leidžiantis tolimesnius svarstymus sieti su praktiniu jų pritaikymu.

Kadangi šis darbas yra rašomas dviejų žmonių (mano kolegos, Lino Šimkaus, ir manęs), tai bendroji darbo rengimo strategija ir konkrečių darbų pasiskirstymas yra gana aiškūs. Idėja paprasta – abu kaip bendrojo darbo rezultatą pristatome klaidų valdymo modelį, pasirenkame vieną pavyzdinę analizuojamąją sistemos architektūrą, pritaikome jai siūlomąjį modelį ir gautus rezultatus atitinkamai įvertiname. Individualus kiekvieno indėlis atsiranda sulig paskutiniais dviem punktais, t.y. pavyzdinei analizuojamajai sistemos architektūrai siūlomąjį modelį kiekvienas pritaikome nepriklausomai vienas nuo kito, skirtingiems jo etapams parinkdami skirtingus įgyvendinimo būdus (klaidų apdorojimo strategijas, su tuo susijusius algoritmus arba pseudo-algoritmus ir t.t.).

Taigi pirmasis skyrius yra skirtas trumpai apžvelgti tris potencialius klaidų valdymo modelius, kurie galėtų būti taikomi ne tik šiame darbe, bet ir būti adaptuojami ir naudojami įvairiose programų sistemų kūrimo praktikose.

Antrajame skyriuje yra pateikiamas klaidų valdymo modelis, kuriuo bus toliau darbe remiamasi, pateikiami tokio pasirinkimo argumentai, aprašoma modelio struktūra ir aptariami su tokio modelio realizavimu susiję klausimai.

Trečiajame skyriuje yra pristatoma pasirinktoji pavyzdinė analizuojamoji sistemos architektūra, trumpai apibūdinama jos dalykinė sritis, probleminė sritis, pateikiamas architektūrą realizuojantis kodas.

Ketvirtas skyrius yra skirtas klaidų valdymo modelio praktiniam tinkamumui duotos architektūros atžvilgiu pagrįsti. Čia yra pateikiami reikalavimai, kuriuos turėtų tenkinti naujoji analizuojamos sistemos versija. Taip pat kiekvienam modelio pritaikymo žingsniui yra parenkami konkretūs projektavimo sprendimai (apžvelgiama kylanti problema, pristatomas jai spręsti žinomas sprendimo būdas, suformuluojama galimas jo realizacijos variantas ir numatomos galimos pasekmės bei įtaka visam sistemos darbui ([LW04])).

Penktajame skyriuje yra aprašomos apžvelgtos ir konkrečiam taikymui siūlomos vertinimo metodikos bei atliekamas pasiekto atsparumo klaidoms laipsnio pasirinktoje sistemos architektūroje įvertinimas.

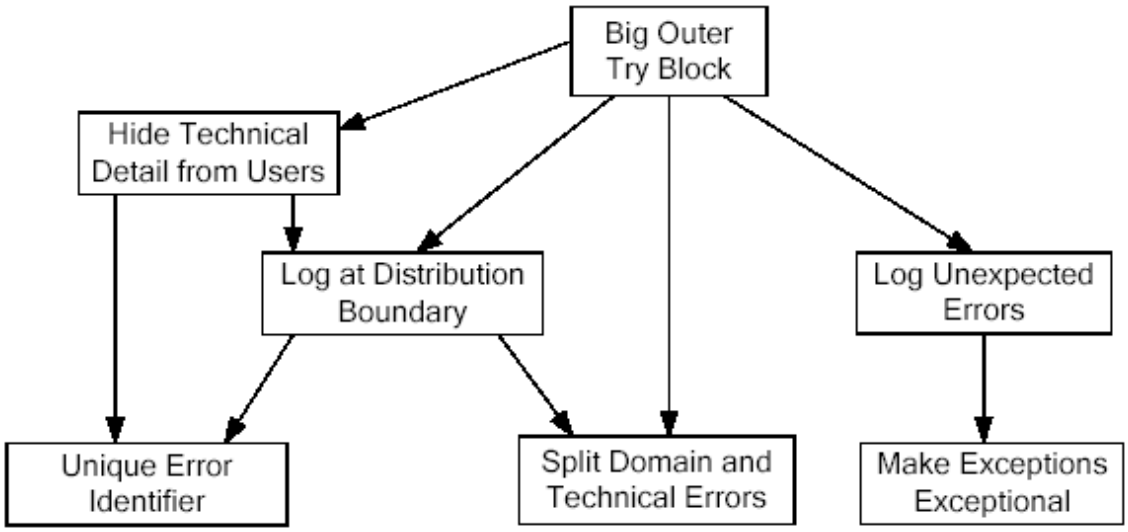
Šeštasis skyrius yra dedikuotas viso atlikto darbo ir gautų rezultatų galutinėms išvadoms pateikti. Čia bus taikoma lyginamoji metodika, skirta atitinkamai įvertinti gautus rezultatus ir juos paaiškinti.

Septintajame skyriuje yra aprašomos bendros darbo išvados ir galutiniai pastebėjimai.

# 1. KLAIDŲ VALDYMO MODELIŲ APŽVALGA

Šiame skyriuje palyginimui trumpai apžvelgsiu tris klaidų valdymo modelius, kurie potencialiai galėtų būti naudojami tolimesnėse darbo temos dėstymo dalyse. Iš šių modelių galiausiai bus pasirinktas vienas modelis, kuriuo ir bus remiamasi darbo metu.

Taigi pirmasis toks į klaidų valdymą orientuotas modelis būtų [LW04] darbe siūlomas klaidų valdymo modelis, skirtas nusakyti programų sistemose galinčių kilti klaidų tam tikrus administravimo ir jų apdorojimo būdus. Jį vaizduojanti schema yra pateikta 1 paveikslėlyje.




**1 pav.** Vienas iš galimų klaidoms apdoroti skirtų klaidų valdymo modelių ([LW04]).

Pavaizduotą klaidų valdymo modelį sudaro 7 dalys, iš kurių kiekviena kaip tik ir nusako tam tikrą būdą, kuriuo remiantis gali būti administruojamos ir / arba apdorojamos programų sistemoje aptinkamos klaidos. Bendru atveju rodyklės reiškia galimą kelių klaidų valdymo būdų mišrų panaudojimą arba atskiru atveju sąryšius, kurie nurodo, koks būdas kuriais kitais klaidų valdymo būdais gali remtis. Toliau ir apžvelgsiu, į ką yra orientuotas kiekvienas į pateiktąjį modelį įeinantis klaidų valdymo būdas.

1. Dalykinės srities ir techninio tipo klaidų atskyrimas („*Split Domain and Technical Errors*“) (žr. 2 paveikslėlį).

   - Dalykinės srities klaidomis gali būti laikomos bendrinės klaidos, susijusios, pavyzdžiui, kad ir su vartotojo atliekamomis užduotimis. Tokio tipo klaidos

pavyzdys galėtų būti situacija, kai darbo su grafiniu redaktoriumi metu vartotojas staiga nebegali pažymėti kurio nors nupiešto grafinio objekto, nors tai iš tikrųjų turėtų būti leidžiama.

- Techninės srities klaidomis gali būti laikomos bet kurios žemo lygio klaidos, kylančios sistemos viduje bei galinčios atsirasti ir vartotojui inicijavus jokių konkrečių veiksmų. Tokio tipo klaidų pavyzdžiai galėtų būti tinklo klaidos, steko perpildymas, aparatūrinis įrangos gedimas ir pan. Be abejo, techninės srities paprastai daugiau ar mažiau įtakoja ir dalykinės srities klaidas, nors to vartotojas gali ir nepajausti (pavyzdžiui, jei buvo neteisingai atliktas duomenų mainų tinklu inicializavimas, nors vartotojo tokio veiksmo vienos darbo su sistema sesijos metu ir nereikėjo).

- Toks klaidų suskaidymas yra daugiausia paremtas loginiu požiūriu, kuris tuo pačiu pasako ir klaidų apdorojimo metu galimą santykį su vartotoju. Šis aspektas gali būti svarbus tokioms probleminėms sritims kaip klaidų apdorojimo strategijų parinkimas, dialogo būtinas ir detalumas, veiksmų funkcionalumo enkapsuliavimas ir pan.

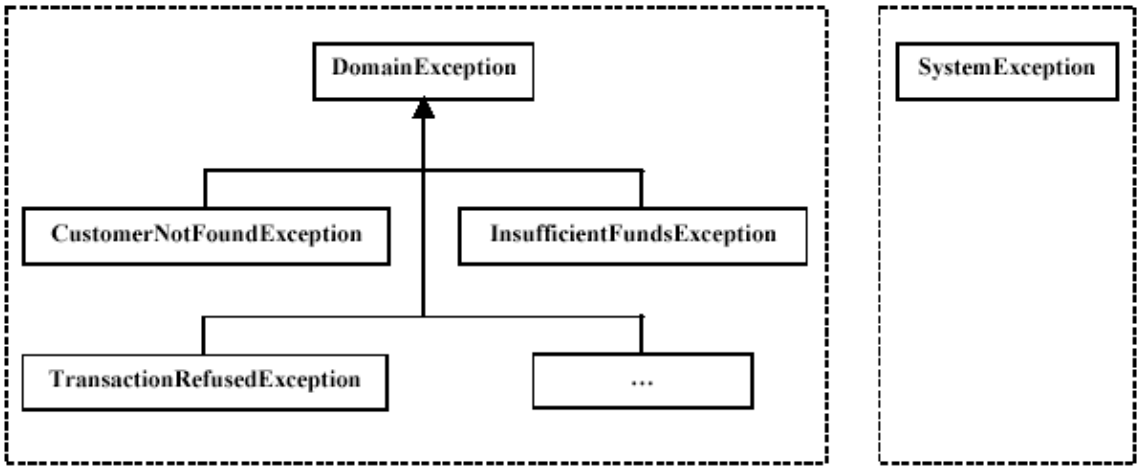
- Šaltinio autorių teigimu, šių dviejų tipų klaidų apjungimas į vieną visumą tik apsunkina ir komplikuoja visą darbą. Dėl techninio tipo klaidų dalykinės srities darbas gali apskritai tapti nebeaktyvus, todėl yra reikalinga, kad egzistuotų sistemos dalis, kuri bandytų viską išanalizuoti ir atstatyti.

- Kai kurioms techninio tipo klaidoms yra prasminga takyti operaciją „*Retry*“, nors dalykinio srities klaidoms tai neturės daug prasmės ar reikšmės.

- Kiekvienam iš tipų yra rekomenduojama sukurti atskirą galimų klaidų ir situacijų hierarchiją bei numatyti individualias darbo su jomis priemones (pavyzdžiui, turėti nepriklausomas funkcijas, resursus ir pan.).

- Dalykinės srities klaidos visada turėtų prasidėti nuo dalykinės srities problemos, kuriai išspręsti būtų paskirtas atitinkamas dalykinės srities kodas.

- Jei yra dirbama su išskirstyta komponentine programų sistema, reikėtų rimtai atkreipti dėmesį į tai, kiek informacijos turėtų su savimi teikti dalykinės srities klaida, atėjusi iš nutolusio sistemos komponento ir kiek ją nutoliniu būdu verta tvarkyti.




**2 pav.** Schematinis dalykinės ir techninės srities tipo klaidų atskyrimo iliustravimas ([LW04]).

Paveikslėlyje yra pavaizduotos 5 klaidas atitinkančios klasės, kurių pavadinimai yra: „*DomainException*“ (nurodo bendrinę dalykinės srities klaidų klasę), „*CustomerNotFoundException*“ (nurodo klasę, skirtą klaidoms, kurios gali kilti, kai reikalingi duomenys apie vartotoją (pavyzdžiui, kokios nors užklausos vykdymo metu) yra nerandami), „*InsufficientFundsException*“ (nurodo klasę, skirtą klaidoms, kurios gali kilti, kai atsiranda lėšų stygius), „*TransactionRefusedException*“ (nurodo klasę, skirtą klaidoms, kurios gali kilti, kai kokios nors transakcijos vykdymas yra atmetamas) ir „*SystemException*“ (nurodo bendrinę techninės srities klaidų klasę).

2. Susitelkimas į klaidų pasiskirstymo zoną („*Log at Distribution Boundary*“).

- Klaidos gali sukelti kitas klaidas. Jei taip nutinka išskirstytų komponentinių programų sistemų atveju su techninio tipo klaidomis, tuomet jų pataisymas su kiekviena tokia klaida gali vis labiau komplikuotis, juolab jei taisyti jas reikės nuotoliniu būdu. Kylus klaidai, turima informacija galėtų būti perduodama kuriam nors centriniam vykdomajam sistemos komponentui, kur būtų tinkamai išanalizuota, prieš jai sukeliant bet kokias tolimesnes problemas.

- Techninio tipo problemas paprastai analizuoja atitinkami specialistai, dėl to visa informacija, susijusi su tokiomis klaidomis, turėtų būti registruojama, išsaugoma ir lengvai prieinama iš kurios nors darbo stoties, kuri nepriklausomai nuo problemų būtų funkcionali.

- Paprastai yra pageidautina, kad klaidų taisymas būtų atliekamas kiek įmanoma labiau nesutrikdant įprasto sistemos darbo.

3. Unikalių klaidų identifikatorių panaudojimas („*Unique Error Identifier*“).

- Jei išskirstytoje komponentinėje programų sistemoje vieno tipo klaida skirtinguose komponentuose sukelia kitas to paties tipo klaidas, tai analizuojant sistemos būklę, gali susidaryti klaidingas įspūdis apie tai, kiek ir kokių klaidų yra einamuoju momentu. Kilus klaidai, išeitis yra, sugeneruoti jos unikalų identifikatorių ir nusiųsti atitinkamam sistemos komponentui.

- Klaidų identifikatoriai turi būti unikalūs visuose sistemos komponentuose.

- Identifikatorių unikalumui užtikrinti galima naudoti vadinamuosius *UUID* („*Universally Unique ID*“) arba *GUID* („*Globally Unique ID*“) reikšmių generavimo principus.

- Taip pat svarbu yra užtikrinti, kad identifikatoriai būtų teisingai interpretuoti (nes jie yra siunčiami tik kaip tam tikra dalis pranešimo apie įvykusią klaidą).

4. Didelio išorinio *Try* bloko panaudojimas („*Big Outer Try Block*“).

- Tikslas yra aprašyti tam tikrą mechanizmą, kuris gaudytų bei apdorotų visas klaidas ir situacijas, kurių kiti sistemos komponentai apdoroti negalėjo.

- Visoms klaidoms ir situacijoms, pasiekusioms tokį mechanizmą, turėtų būti suformuojami atitinkami pranešimai, skirti vartotojams ir pateikti jiems suprantama forma.

- Jei mechanizmą pasiekia dalykinės srities klaida, tai tada gali nutikti taip, kad gali būti interfeiso projektavimo problema, o jei visiškai nežinoma klaida – tuomet tai yra įprasta traktuoti kaip vidinį programos veikimo trikį.

5. Techninio tipo klaidų ypatumų nuslėpimas nuo vartotojų („*Hide Technical Error Detail From Users*“).

- Vartotojams dažniausiai nėra būtina žinoti techninio tipo klaidų detales, nes jos jiems tiesiog bus neinformatyvios.

- Siūlymas yra sukurti standartinį techninio tipo klaidų pranešimų pateikimo vartotojams mechanizmą. Toks mechanizmas turėtų pasižymėti dviem savybėmis. Pirma, jis tam tikru būdu turėtų enkapsuliuoti visą detalią su įvykusia klaida susijusią informaciją. Antra, suformuoti aiškų standartinį pranešimą vartotojui su kokiu nors unikaliu klaidos identifikatoriumi, pagal kurį, reikalui esant, vartotojas galėtų apie tai kam nors pranešti.

6. Nežinomų klaidų registravimas („*Log Unexpected Errors*“).

- Nuspėjamoms ir nežinomoms klaidoms apdoroti yra siūloma sukurti atskiras tokios paskirties priemones.

- Klaidų pasireiškimo aplinkybės, kurios yra nuspėjamos arba žinomos, neturėtų būti įtraukiamos į klaidos aprašymą. Jame turėtų būti tik tai, ką yra būtina išanalizuoti. Apskritai net ir visoms nuspėjamoms klaidoms yra siūloma nekurti atskirų aprašymų, o tiesiog atitinkamai jas apdoroti kuriame nors operacijos kodo kontekste.

- Jei yra būtinumas registruoti ir nuspėjamas klaidas su jų aprašymais, tai galima kurti atskirus registrus pagal jų paskirtį (pavyzdžiui, vieną, skirtą prisijungusiems per dieną prie sistemos vartotojams žymėti, o kitą – su sistemos apsauga susijusiems duomenims fiksuoti) ir / arba saugoti juos skirtingose sistemos vietose tam, kad darbas su jais netrikdytų įprasto darbo su sistema.

7. Išimtinis situacijų panaudojimas („*Make Exceptions Exceptional*“).

- Jei sistemoje yra naudojamos ir tam tikros situacijos, skirtos nuspėjamoms klaidoms ir jų pasireiškimo aplinkybėms fiksuoti, tuomet klaidų apdorojimo kodas tampa gana komplikuotas.

- Nuspėjamoms dalykinės srities klaidoms identifikuoti yra rekomenduojama naudoti grįžimo kodus („*return codes*“), o situacijas naudoti tik „*runtime*“ tipo klaidoms.

Antrojo klaidų valdymo modelio pagrindą sudaro 3 etapai: aptikimas („*detection*“), analizė („*diagnosis*“) ir apdorojimas („*handling*“). Klaidų aptikimo dalyje klaidų situacijoms aprašyti modelyje yra remiamasi kokia nors situacijų / įvykių specifikavimo kalba, pavyzdžiui, *JECA* („*Justified Event–Condition–Action*“) ([LSK+00]). Šią kalbą sudaro specialių taisyklių rinkiniai. Kiekviena taisyklė (paprastai žymima „*r* (*j*, *e*, *c*, *a*)“) yra sudaroma iš 5 dalių:

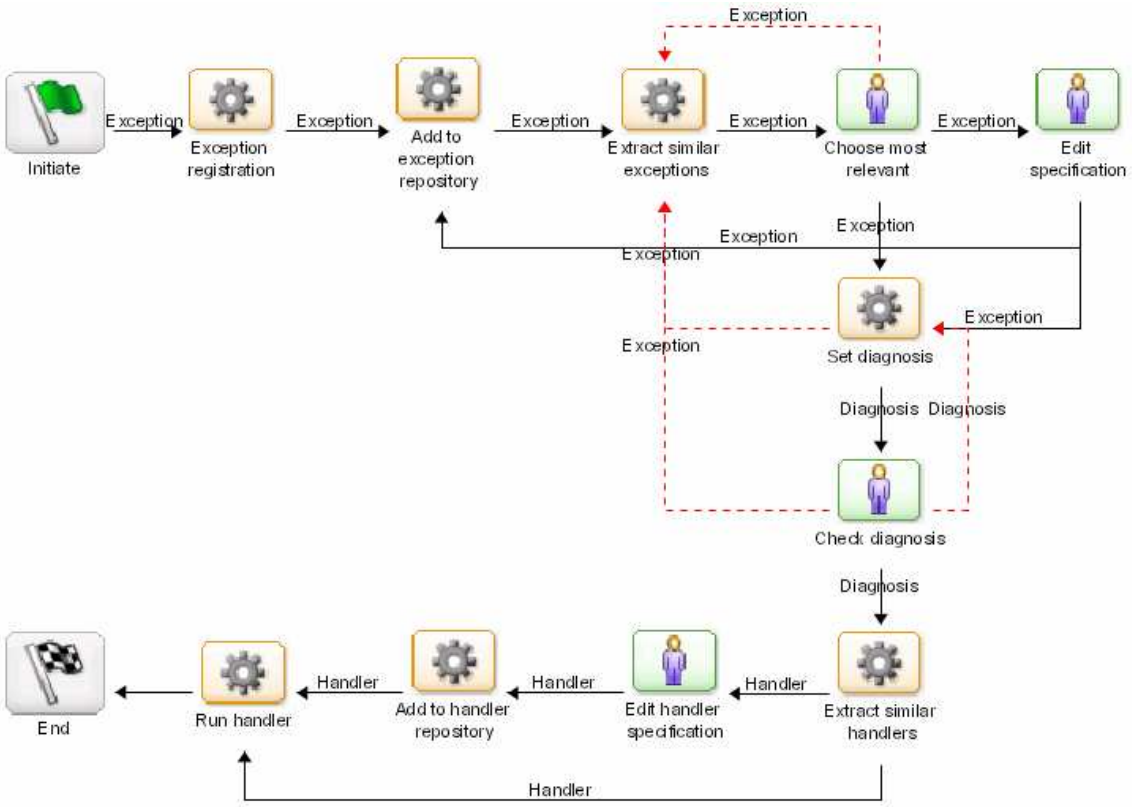
- Patikslinimo („*Justification* (*j*)“), kuris nusako konkretų kontekstą, kuriame toji taisyklė bus taikoma.
- Įvykio („*Event* (*e*)“), kuriam įvykus, taisyklė taps galiojančia („*triggered*“).
- Salygos („*Condition* (*c*)“), kurią sudaro loginiai apribojimai, kurie turi būti patenkinti tam, kad būtų galima taikyti taisykle aprašomą veiksmą.
- Veiksmo („*Action* (*a*)“), kuris bus įvykdytas tik tada, jei bus patenkinta taisyklė.

Anot šaltinio [Voj05] autoriaus, šios taisyklės klaidų valdymo praktikoje yra labai naudingos, nes jomis pagal iš anksto nustatytą formą galima derinti visą renkamą, saugomą ir operuojamą su klaidomis susijusią informaciją. Taisyklės taip pat padeda lengviau atlikti klaidos lokalizavimą, klaidos būvio įvertinimą (t.y. nustatymą, ar situacija iš tikrųjų turi būti laikoma klaida), ir, be abejo, klaidos apdorojimą. Klaidų aptikimo etape pagrindinė sistemos užduotis yra kuo tiksliau, detaliau ir kiek galima greičiau parengti klaidos specifikaciją, kuria toliau remsis kiti du modelio etapai. Be to, klaidų aptikimo etape sistema taip pat turėtų informuoti visas suinteresuotas vartotojų grupes apie susidariusią klaidos situaciją ir suteikti jiems atitinkamą prieigą prie parengtos specifikacijos. Darbui su specifikacijomis turėtų būti skirtos specialios programinės priemonės.

*JECA* yra ne vienintelė specifikavimo kalba, kuri gali būti taikoma klaidų valdymo modelio panaudojimo kontekste. Kitų specifikavimo kalbų pavyzdžiai galėtų būti *PDDL*

(„*Planning Domain and Definition Language*“) ir *AML* („*Aspen Modeling Language*“) ([AH05]). Jomis gali būti specifikuojamos ir modeliuojamos ne tik situacijos ar įvykiai, bet ir, reikalui esant, visos darbo stotys (ypač *PDDL* specifikavimo kalbos atveju). Tam tikslui yra naudojamos specialios tų kalbų apibrėžiamos esybės („*entities*“) ir atitinkami ryšiai, kuriais galima modeliuoti išvestines situacijas ir esybes.

Oficiali analizės etapo pabaiga ateina tada, kai yra baigiama generuoti klaidų apdorojimo specifikacija, kuri idealiu atveju tolimesniam darbui turėtų būti tiesiogiai perduodama klaidų apdorojimo posistemei. O visas klaidų valdymo procesas yra baigiamas tada, kai yra baigiamas vykdyti klaidų apdorojimo etapas. Detali tokio modelio schema yra pateikta 3 paveikslėlyje.




**3 pav.** Vienas iš galimų klaidoms apdoroti skirtų klaidų valdymo modelių ([Voj05]).

Paveikslėlyje pateiktąjį modelį sudaro 13 komponentų, iš kurių kiekvienas atitinka tam tikrą su klaidų valdymu susijusią užduotį. Šios užduotys yra:

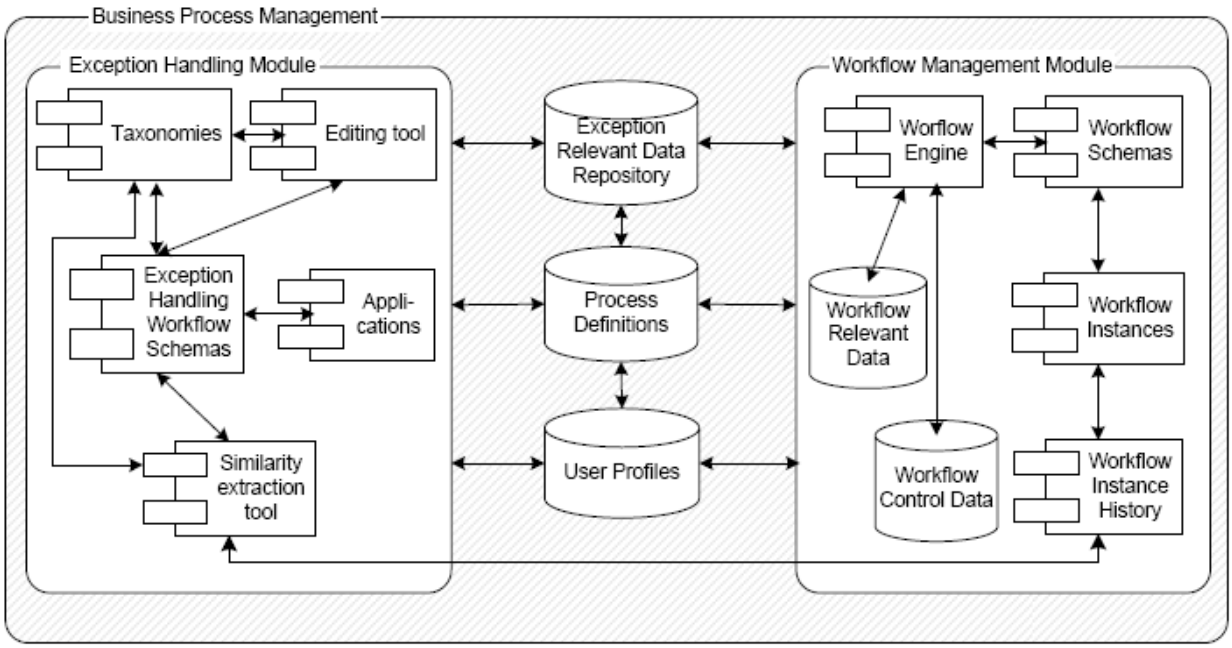
1. Inicijavimas („*Initiate*“).
2. Klaidos situacijų registravimas („*Exception registration*“).
3. Klaidos situacijos patalpinimas į aptiktų klaidos situacijų saugyklą („*Add to exception repository* “).

4. Panašių klaidos situacijų suradimas („*Extract similar exceptions*“).
5. Labiausiai susijusių klaidos situacijų parinkimas („*Choose most relevant*“).
6. Klaidų specifikacijos pakoregavimas („*Edit specification*“).
7. Klaidų analizės („*Set diagnosis*“).
8. Klaidų analizės metu gautų rezultatų patikrinimas („*Check diagnosis*“).
9. Panašiai veikiančių klaidų tvarkyklių suradimas ir parinkimas („*Extract similar handlers*“).
10. Klaidų apdorojimo specifikacijos pakoregavimas („*Edit handler specification*“).
11. Klaidų tvarkyklės patalpinimas į klaidų tvarkyklių saugyklą („*Add to handler repository*“).
12. Parinktos klaidų tvarkyklės įvykdymas („*Run handler*“).
13. Užbaigimas („*End*“).

Šis modelis yra daugiausia skirtas verslo procesų valdymo sistemoms („*Business Process Management Systems (BPMS)*“). O kadangi verslo procesai paprastai yra modeliuojami naudojant vadinamuosius *workflow* metodus, tai modelyje pateikta užduočių seka atrodytų gana įprasta tokiam kontekstui. Dažniausiai taikant tokį klaidų valdymo modelį yra pageidaujama, kad visos paveikslėlyje pateiktos užduotys turėtų būti kiek įmanoma labiau automatizuotos. Dėl to vartotojo dalyvavimo paprastai teprireikia tik kiekvieno iš trijų minėtų etapų pabaigoje, kuomet atsiranda būtinybė įsitikinti, ar visi automatiškai priimti sprendimai buvo teisingi ir pageidaujami.

Taigi vienas iš teigiamų tokio modelio aspektų būtų tai, kad *workflow* metodų taikymo principais paremtas klaidų valdymas leidžia keisti patį klaidų valdymo procesą, nekeičiant naudojamų klaidų valdymo metodų.

Nors šiai dienai verslo proceso valdymo sistemų yra tikrai visokių, tačiau bazinė tokių sistemų architektūra, naudojanti paršaus tipo klaidų valdymo modelius, vis tiek daugiau ar mažiau išlieka tokios pati. Tokios architektūros pavyzdys yra pateiktas 4 paveikslėlyje.




**4 pav.** Bazinė verslo procesų valdymo sistemų architektūra, naudojanti 3 pav. pateiktą klaidų valdymo modelį.

Kaip matyti iš paveikslėlio, tokią architektūrą sudaro 3 pagrindinė dalys, t.y. *workflow* valdymo posistemė („*Workflow Management Module*“), klaidų valdymo posistemė („*Exception Handling Module*“) ir saugyklos („*Repositories*“).

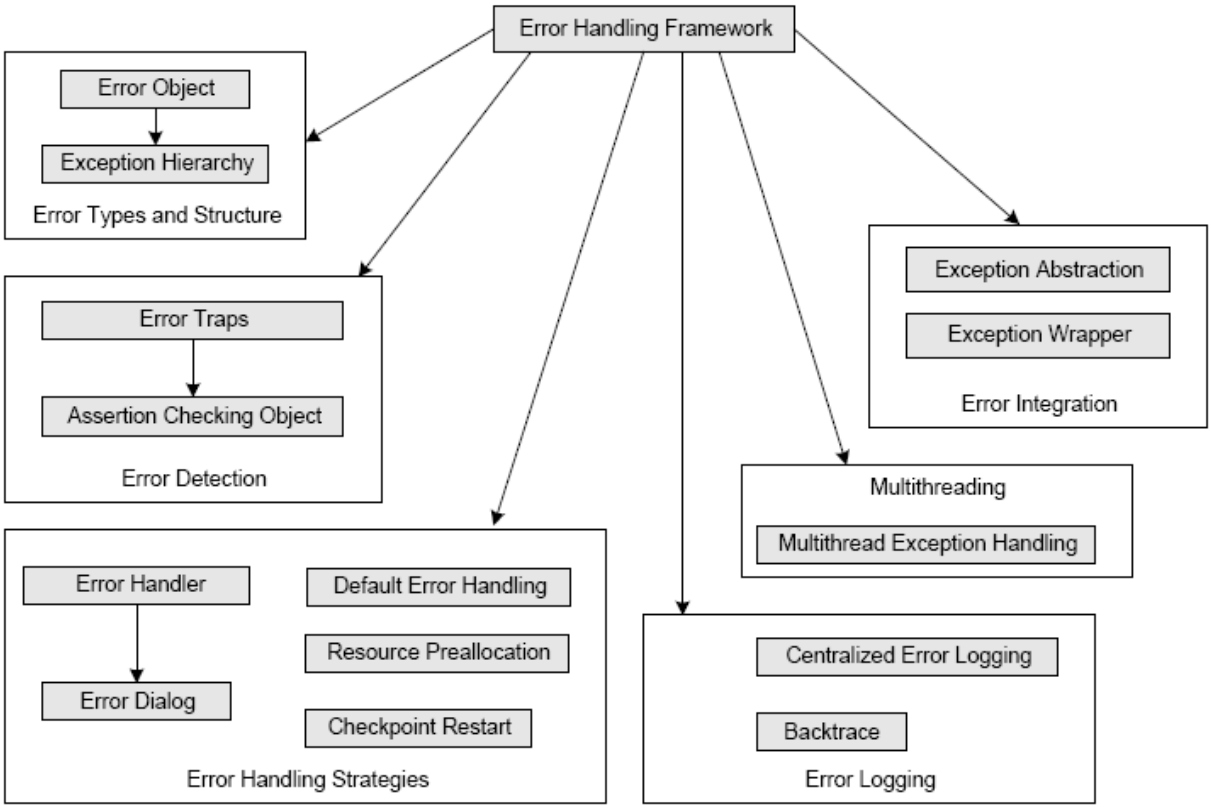
Pirmosios posistemės darbas yra paremtas vadinamosiomis *workflow* schemomis („*Workflow Schemas*“), kur informacijos apie jas surinkimas kaip tik ir yra vienas iš tipinių paruošiamųjų darbų, reikalingų efektyviam klaidų aptikimui atlikti. Papildoma informacija yra gaunama iš darbo metu sugeneruotų specifikacijų ir kontrolinių duomenų saugyklų. Surinktos medžiagos analizę palengvina tam dedikuoti klaidų valdymo posistemės įrankiai (kaip, pavyzdžiui, „*Similarity extraction tool*“ ir kt.).

Antrosios posistemės darbas yra paremtas netiesiogine sąveika su pirmąja posisteme, kur jas jungianti tarpinė grandis ir yra vienas iš dedikuotų verslo procesų valdymo sistemos įrankių „*Similarity extraction tool*“. Pagrindinė jo užduotis yra nustatyti, ar einamuoju momentu aktyvioje *workflow* schemoje neatsirado kokių nepageidaujamų klaidų. Tam tikslui yra lyginami visi su ta schema susiję sukaupti duomenys su duomenimis, kuriuos pasako atitinkama sistemoje saugoma susisteminta informacija („*Taxonomies*“). Jei lyginimo metu atsiranda panašumų, tuomet minėtasis įrankis kreipiasi į klaidų valdymo *workflow* schemų parinkimo komponentą su prašymu nustatyti, ar egzistuoja kokia nors klaidų valdymo *workflow* schema, galinti padėti pašalinti turimą problemą. Jei tokia schema atsiranda, tuomet ji yra perduodama pirmosios posistemės „*Workflow Engine*“ komponentui, kad ją tinkamai apdorotų ir pritaiktų. Jei

konkrečios turimam atvejui tinkančios schemos nėra, tuomet administratorius ar kuris kitas suinteresuotas sistemos vartotojas gali pasirinkti kokią nors arčiausiai tiesos esančią schemą ir ją adaptuoti problemos sprendimui.

Pateiktoje architektūroje yra iliustruojamos trys pagrindinės saugyklos, t.y. su klaidomis susijusių duomenų saugykla („*Exception Relevant Data Repository*“), procesų apibrėžčių saugykla („*Process Definitions*“) ir vartotojo duomenų saugykla („*User Profiles*“).

Na, ir paskutinysis trečiasis klaidų valdymo modelis būtų [Boo03] darbe siūloma 5 paveikslėlyje pavaizduota schema.



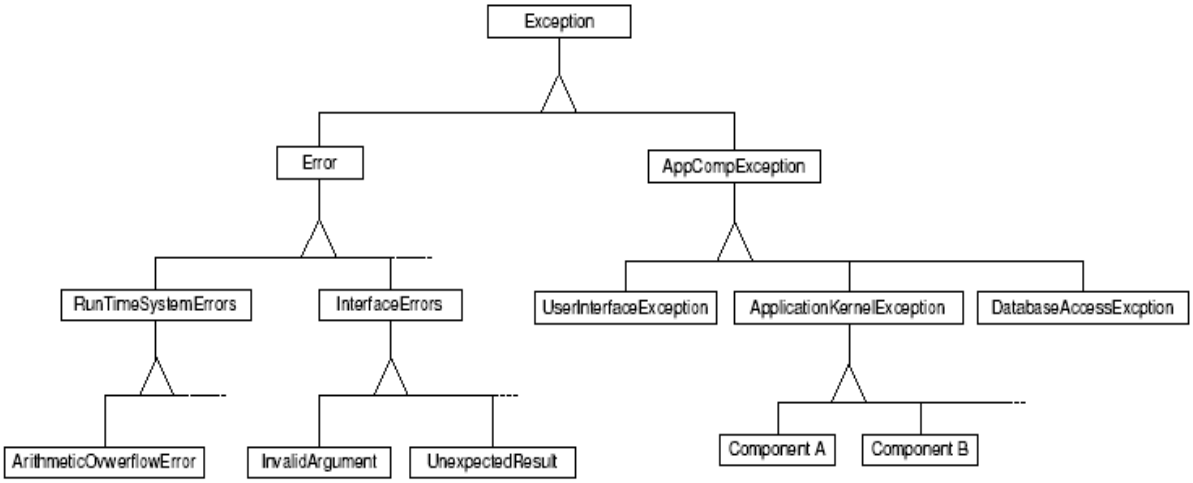

**5 pav.** Vienas iš galimų klaidoms apdoroti skirtų klaidų valdymo modelių ([Boo03]).

Bendru atveju pateiktąjį klaidų valdymo modelį galima suvokti kaip tam tikrą teorinį karkasą, sudarytą iš aibės aspektų, kurie pasako, kas turi būti padaryta ir realizuota, norint programų sistemoje turėti efektyviai sukurtą į klaidų valdymą orientuotą funkcionalumą.

Visas klaidų valdymo modelis yra suskaidytas į 6 stambias problemines sritis, su kuriomis dažnai tenka susidurti, sprendžiant su klaidų valdymu susijusius uždavinius. Šios probleminės sritys yra:

1. Klaidų identifikavimas ir struktūrizavimas („*Error Types and Structure*“).
2. Klaidų aptikimas („*Error Detection*“).
3. Klaidų registravimas („*Error Logging*“).
4. Klaidų apdorojimo strategijų parinkimas („*Error Handling Strategies*“).
5. Klaidų lygiai („*Error Integration*“).
6. Daugiagijis klaidų apdorojimas („*Multithreading*“).

Klaidų identifikavimas ir struktūrizavimas apibrėžia, kas sistemoje turi būti laikoma klaida („*Error Object*“), ir nusakančią bendrą klaidas tarpusavyje siejančią struktūrą („*Exception Hierarchy*“). Tokios struktūros pavyzdys galėtų būti [Boo03] darbe pateikiama situacijų hierarchijos medžio struktūra (žr. 6 paveikslėlį):




**6 pav.** Situacijų hierarchijos pavyzdys ([Boo03]).

Klaidų aptikimas nusako priemones, kurios susideda iš klaidų atsiradimo nustatymui atlikti reikalingų būdų („*Error Traps*“) (pavyzdžiui, atitinkamose kodo vietose atlikti loginį kokios nors išraiškos arba algoritminį kokios nors situacijos įvertinimą ([Pet01])). Šiuo atveju svarbu yra tinkamai nustatyti klaidos mastą. Pavyzdžiui, komponentinėje programų sistemoje pagal mastą klaidos gali būti identifikuojamos taip ([Chro05]):

- Lokalios klaidos („*Local Errors*“), kurios pasireiškia tik vieno kurio nors sistemos komponento viduje.

- Gretutinės klaidos („*Adjacent Errors*“), kai klaidos pasireiškimas turi būti tikrinamas keliose gretimuose tarpusavyje komunikuojančiuose ar kitaip susijusiuose komponentuose.
- Išorinės klaidos („*Outer Errors*“), kai yra tikrinama, ar išoriniai tiriamojo komponento atžvilgiu komponentai (pavyzdžiui, kurie visi yra skirtingose posistemėse) galėjo įtakoti klaidos jame pasireiškimą.
- Globalios klaidos („*Global Errors*“), kurios sutrikdo visą arba bent jau didžiąją dalį sistemos.

Klaidų registravimas yra orientuotas į centralizuotą informacijos valdymą (saugojimą ir pan.) („*Centralized Error Logging*“), bei į klaidos situaciją sukėlusių įvykiam fiksuoti reikalingo steko sudarymą, analizę ir panaudojimą tos situacijos apdorojimo metu („*Backtrace*“).

Klaidų apdorojimo strategijų parinkimas apima tokius klausimus kaip klaidų apdorojimo procedūrų („*Error Handler*“) sukūrimas, prieigos būdų (interfeisų) prie atitinkamų sistemos vietų, kuriose buvo aptiktos klaidos, apibrėžimas, pranešimų parinkimas ir / arba generavimas („*Error Dialog*“), standartinių (t.y. pagal nutylėjimą veikiančių) klaidų apdorojimo procedūrų apibrėžimas ir atvejų, kada jos turi būti taikomos, numatymas („*Default Error Handling*“), klaidų apdorojimui reikalingų resursų paskirstymas („*Resource Reallocation*“), ir klausimų, susijusių su sistemos dalies ar net viso jos darbo pristabdymu dėl klaidų apdorojimo būtinybės, sprendimų apžvalga („*Checkpoint Restart*“).

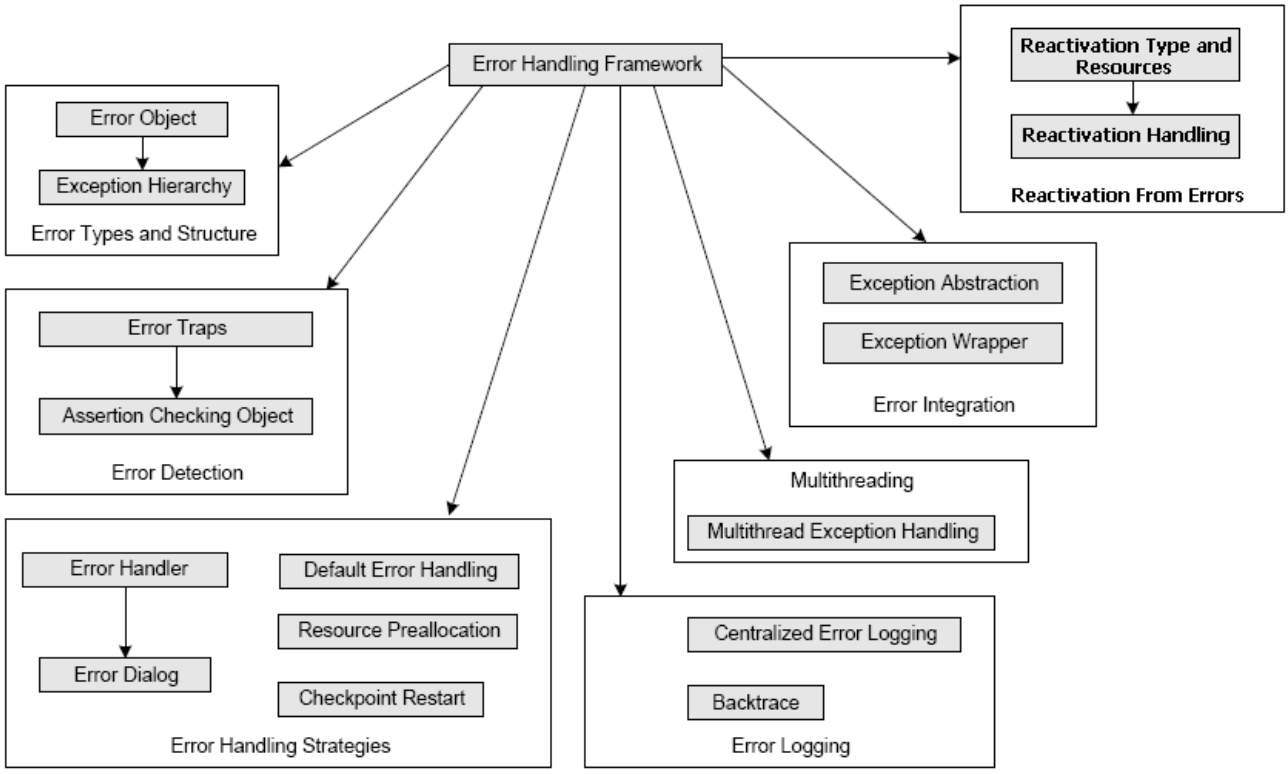
Klaidų lygiai nusako klaidas nusakančios informacijos, kuri bus analizuojama ir kuria bus operuojama, sugeneravimą ir enkapsuliavimą („*Exception Abstraction*) į aukštesnio lygmens situacijas („*Exception Wrapper*“). Čia turėtų būti nagrinėjami ne tik klausimai, susiję su tuo, kokia informacija, kiek jos ir kam turės kuriuo nors konkrečiu atveju būti pateikiama, bet ir apsprendžiami tam tikri, pavyzdžiui, pranešimų internacionalizavimo ir kitokių problemų keliami reikalavimai pranešimams vartotojui sudaryti ir pateikti.

Daugiagijis klaidų apdorojimas („*Multithreading*“) galėtų leisti nenutraukti einamuoju momentu aktyvios sistemos darbo bei galbūt lygiagrečiai išskirstyti tam tikras specifines klaidų apdorojimo operacijas („*Multithreadng Exception Handling*“).

# 2. KLAIDŲ VALDYMO MODELIO PASIRINKIMAS

Iš pirmame skyriuje pateiktų trijų klaidų valdymo modelių vis tik buvo nutarta pasirinkti trečiąjį. Tokio pasirinkimo argumentai galėtų būti tokie, kad jis yra gana išsamus, nes iš visų trijų modelių apie save net ir iš pirmo žvilgsnio teikia daugiausia informacijos, aiškiai suskaidytas į atskiras nepriklausomas, suprantamas logines dalis, kurios apima beveik visus aukščiausiame bendrumo lygyje su klaidų valdymu susijusius aspektus, nepriklausomas nuo jokios konkrečios realizacijos konteksto, ir pakankamai bendras, kad galėtų būti adaptuojamas įvairiose programų sistemų kūrimo praktikose.

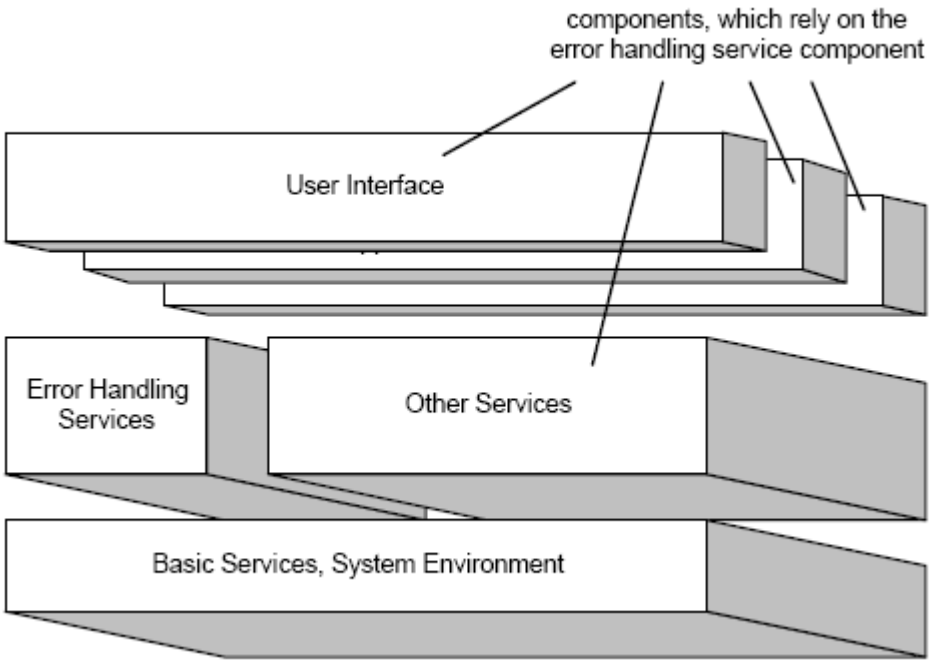
Vis dėlto, mano nuomone, vienos probleminės srities pasirinktame klaidų valdymo modelyje vis dėlto trūksta, t.y. sistemos reaktyvavimo. Šis aspektas svarbus tuo, kad pirmiausia tai yra neatsiejamas nuo klaidų valdymo eigos etapas, kuomet automatiniu būdu arba su vartotojo pagalba yra siekiama grąžinti sistemą į normalų jos veikimo režimą ir / arba atstatyti nebaigtų vykdyti užduočių būklę arba klaidos atsiradimo metu prarastus duomenis. Kadangi su šiuo aspektu susiję klausimai yra specifiniai, tai juo ir bus papildytas pasirinktasis klaidų valdymo modelis (žr. 7 paveikslėlį).




**7 pav.** Klaidų valdymo modelis.

Nagrinėjant sistemos reaktyvavimą („*Reactivation From Errors*“), svarbu yra apibrėžti patį reaktyvavimo tipą, kurių bendru atveju priklausomai nuo sudarytų sistemos veikimo scenarijų gali būti ir keli, taip pat nustatyti tam reikalingus resursus, jų valdymo (pavyzdžiui, priėjimo prie jų kritiniu metu ir pan.) būdus bei apibrėžti ir sukurti patį sistemos reaktyvavimą realizuosiantį funkcionalumą.

Kiekviena iš klaidų valdymo modelį sudarančių probleminių sričių bus kiek detaliau aptariama ketvirtame šio darbo skyriuje. O kol kas, manau, naudinga būtų pateikti bent šiokį tokį įsivaizdavimą apie tai, kokią vietą visas šis klaidų valdymo modelis gali užimti bendrojoje projektuojamoje kurios nors sistemos architektūroje. Tai atspindi 8 paveikslėlyje pateiktas fragmentas.



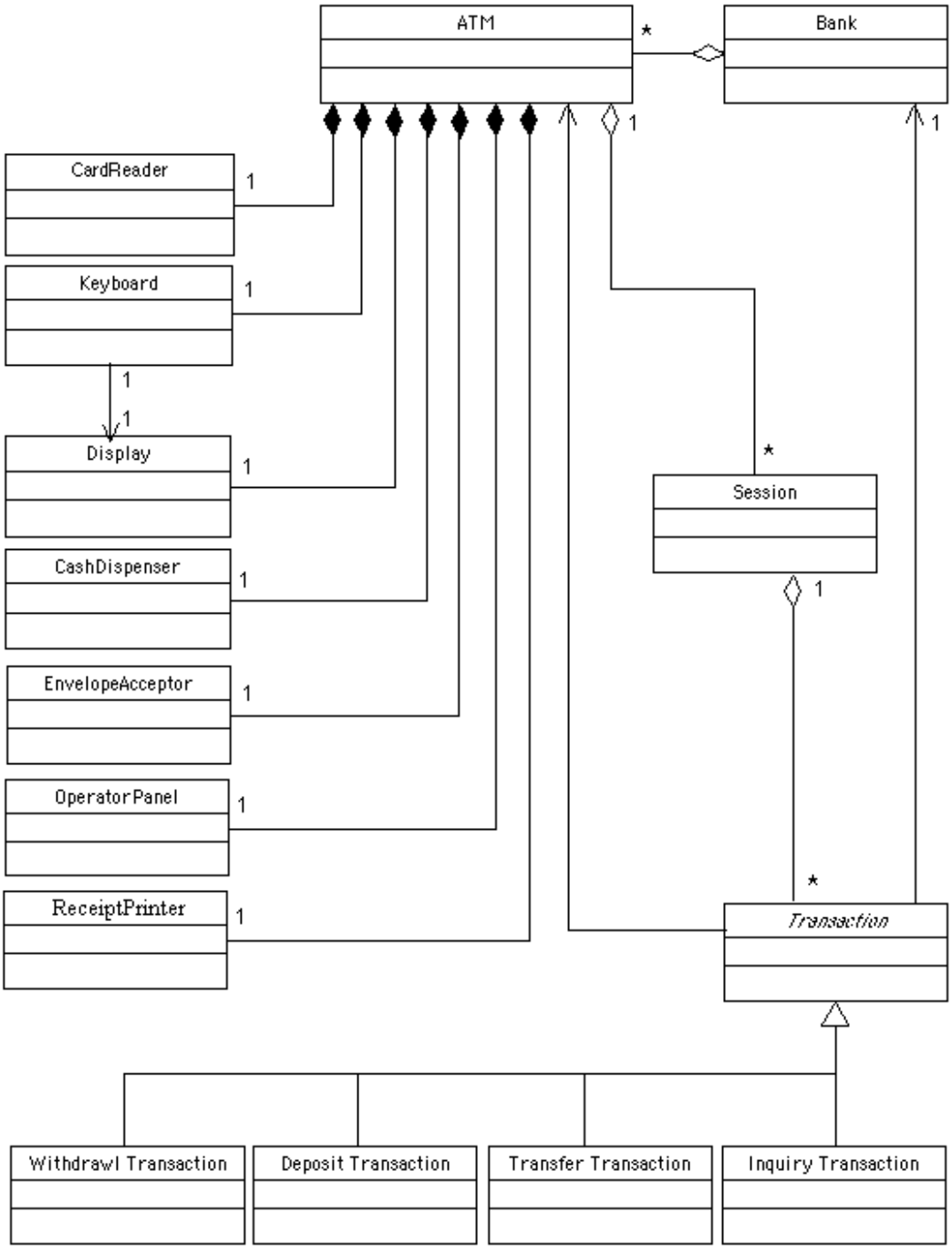

**8 pav.** Klaidų valdymo modelio vaidmuo apibendrintoje sistemos architektūroje ([Ren03]).

Paveikslėlyje nurodyti stačiakampiai blokai atitinka visas pagrindines tipinę programų sistemą sudarančias dalis, t.y. vartojo interfeisą („*User Interface*“), bazinius servisus („*Basic Services*“), sisteminę aplinką („*System Environment*“), bet kuriuos kitus specifinius kiekvienai programų sistemai atskirai servisus („*Other Services*“), na ir, be abejo, klaidų valdymo servisus („*Error Handling Services*“).

Tokioje apibendrintoje sistemos architektūroje klaidų valdymo komponentai turi būti kiek įmanoma labiau nepriklausomi nuo bet kokių sistemoje esančių komponentų. Tai yra reikalinga, siekiant klaidos atveju išvengti ciklišku ryšių. Bendru atveju tokį pateiktąjį klaidų valdymo komponentų vaidmenį sistemoje galima įsivaizduoti kaip tam tikrą vientisą klaidų magistralę, prie kurios yra prisijungę visi kiti sistemos komponentai. Bazinė klaidos klasė sistemoje apibrėžia tam tikrą tipą duomenų, kurie klaidos atveju keliauja klaidų magistrale nuo vieno komponento prie kito, perduodami su tuo susijusią informaciją. Čia svarbu yra pabrėžti tai, kad klaidų magistralė sistemoje yra visiškai nepriklausoma nuo bendro komunikacijos kanalo, kuriuo tarpusavyje bendrauja sistemos komponentai.

# 3. PAVYZDINĖ ANALIZUOJAMOJI SISTEMOS ARCHITEKTŪRA

Tam, kad dėstomų minčių aiškumas išliktų skaidrus ir lengvai suprantamas, pasirinktoji pavyzdinė analizuojamoji sistemos architektūra bus gana paprasta, tačiau vis tiek glaudžiai susijusi su viena iš mūsų visuomeniniame gyvenime egzistuojančių praktikų, t.y. bankomatų teikiamų paslaugų. Bankomatą imituojančios sistemos architektūra yra pateikta 9 paveikslėlyje.




**9 pav.** Bankomatą imituojančios sistemos architektūra ([Bjo97]).

Apskritai šiai dienai bankomatai, kurie veikia kaip autonominės sistemos ir kuriuose yra realizuotos visos pagrindinės banko operacijos (tokios kaip pinigų išėmimas iš sąskaitos, pinigų padėjimas į sąskaitą, pinigų pervedimas tarp sąskaitų ir vartotojo teikiamų informacinių užklausų priėmimas), yra daugiau ar mažiau paplitę visur, daugiausiai, be abejo, užsienyje, tačiau pamažu jau vis populiarėja ir Lietuvoje. Paprastai jie yra žinomi santrumpa *ATM („Automated Teller Machine“).*

Analizuojamąją sistemos architektūrą sudaro 15 pagrindinių komponentų (klasių). Trumpai kiekvieną architektūros komponentą galima apibūdinti taip:

**1 lentelė.** *ATM* architektūros komponentai ir jų aprašymas.

| Nr. | Komponento pavadinimas | Komponento apibūdinimas |
|---|---|---|
| 1. | *ATM* | Pagrindinė *ATM* bankomatą atspindinti klasė. Joje yra pateikiamos visos bankomato darbui pradėti ir palaikyti reikalingos operacijos (*startupOperation, serviceCustomers, getPIN, getMenuChoice, getAmountEntry, checkIfCashAvailable, dispenseCash, acceptEnvelope, issueReceipt, reEnterPIN, reportTransactionFailure, ejectCard, retainCard*). |
| 2. | *Bank* | Pagrindinė banko, kuriam priklauso ir su kuriuo per tinklą bendrauja *ATM* bankomatas, klasė. Joje yra pateikiamos visos bankui sėkmingai funkcionuoti ir *ATM* bankomatų siunčiamoms užklausoms apdoroti reikalingos operacijos (*initiateWithdrawl, finishWithdrawl, initiateDeposit, finishDeposit, doTransfer, doInquiry, chooseAccountType, accountName, rejectionExplanation*). |
| 3. | *CardReader* | *ATM* bankomato įrenginį (banko kortelių skaitytuvą) atstojanti klasė. Pagrindinės jos operacijos yra *ejectCard, retainCard, checkForCardInserted, cardNumber, action*. |
| 4. | *Keyboard* | *ATM* bankomato įrenginį (klaviatūrą) atstojanti klasė. Pagrindinės jos operacijos yra *readPIN, readMenuChoice, readAmountEntry, inKey, action*. |

| | | |
|---|---|---|
| 5. | *Display* | *ATM* bankomato įrenginį (vaizduoklį) atstojanti klasė. Pagrindinės jos operacijos yra *requestCard, requestPIN, displayMenu, requestAmountEntry, requestDepositEnvelope, reportCardUnreadable, reportTransactionFailure, requestReEnterPIN, reportCardRetained, echoInput, clearDisplay, write*. |
| 6. | *CashDispenser* | *ATM* bankomato įrenginį (pinigų išdavimo įrenginį) atstojanti klasė. Pagrindinės jos operacijos yra *setCash, dispenseCash, currentCash*. |
| 7. | *EnvelopeAcceptor* | *ATM* bankomato įrenginį (pinigų priėmimo įrenginį) atstojanti klasė. Pagrindinės jos operacijos yra *acceptEnvelope, action*. |
| 8. | *OperatorPanel* | *ATM* bankomato įrenginį (*ATM* operatoriaus terminalą) atstojanti klasė. Pagrindinės jos operacijos yra *switchOn, getInitialCash.* |
| 9. | *ReceiptPrinter* | *ATM* bankomato įrenginį (kvitų spausdintuvą) atstojanti klasė. Pagrindinė jos operacija yra *printReceipt*. |
| 10. | *Session* | Vieną darbo su *ATM* bankomatu sesiją atspindinti klasė. Pagrindinės jos operacijos yra *doSessionUseCase, doInvalidPINExtention, doFailedTransactionExtention, cardNumber, PIN*. |
| 11. | *Transaction* | Abstrakti klasė, atspindinti bendrinę darbo su *ATM* bankomato metu atliekamą operaciją. Pagrindinės jos operacijos yra *chooseTransaction, doTransactionUseCase* ir abstrakčios operacijos – *getTransactionSpecificsFromCustomer, sendToBank, finishApprovedTransaction*. |
| 12. | *WithdrawTransaction* | Klasė, atspindinti darbo su *ATM* bankomatu metu atliekamą pinigų išėmimo iš sąskaitos operaciją. |
| 13. | *DepositTransaction* | Klasė, atspindinti darbo su *ATM* bankomatu metu atliekamą pinigų padėjimo į sąskaitą operaciją. |
| 14. | *TransferTransaction* | Klasė, atspindinti darbo su *ATM* bankomatu metu atliekamą pinigų pervedimo tarp sąskaitų operaciją. |
| 15. | *InquiryTransaction* | Klasė, atspindinti darbo su *ATM* bankomatu metu atliekamą vartotojo informacinių užklausų priėmimo operaciją. |

Analizuojamosios sistemos architektūros realizacijos (kodo) originalai yra pateikti kaip atitinkami šio darbo priedai. Nors iš pirmo žvilgsnio jie ir teikia vartotojams *ATM* bankomatams būdingą imitacinę elgseną, ir turi savyje numatytą tam tikrų klaidingų situacijų apdorojimą, tačiau, kaip ir patys autoriai teigia, tai yra labai paprasta realizacija ir ne visai atspindinti tikrovę (pavyzdžiui, kad ir dėl akivaizdaus fakto, jog banko kortelių numeriai ir PIN kodai yra įkoduoti ir pan.).

# 4. KLAIDŲ VALDYMO MODELIO TAIKYMAS

Šiame skyriuje bus nagrinėjami ir priimami konkretūs su klaidų valdymo modelio taikymu susiję architektūriniai ir projektavimo sprendimai ([Soi04]). Apie tai kalbėdamas laikysiuosi objektinės paradigmos principų, kurie jau gana seniai yra tapę populiarūs, visuotinai pripažinti ir plačiai naudojami, tiek kuriant, tiek ir projektuojant įvairias programas ir programų sistemas.

## 4.1 REIKALAVIMAI NAUJAJAI ANALIZUOJAMAI SISTEMAI

Tam, kad būtų aišku, už ką mes kovojame, pirmiausia reikia nusistatyti reikalavimus analizuojamai sistemai. Tokiu būdu bus aišku, kokio funkcionalumo galima tikėtis naujojoje analizuojamos sistemos versijoje, panaudojant pasirinktąjį klaidų valdymo modelį.

**2 lentelė.** Naujosios analizuojamos sistemos reikalavimų suvestinė.

| Nr. | Reikalavimo formuluotė |
|---|---|
| 1. | Klaidų valdymo modelis analizuojamoje sistemoje turi būti realizuotas kaip nepriklausoma klaidų valdymo posistemė taip, kad naująją analizuojamosios sistemos versiją iš viso sudarytų dvi posistemės – dalykinės srities (su realizuotu banko operacijų vykdymu) ir klaidų valdymo. |
| 2. | Klaidų valdymo posistemė turi būti sudaryta iš komponentų, iš kurių kiekvienas atitinka kurią nors vieną iš 7 klaidų valdymo modelio probleminių sričių. |
| 3. | Bet kuriam dalykinės srities posistemės komponentui turi būti prieinamas galimų klaidų aptikimo tikrinimo funkcionalumas. |
| 4. | Bet kuriam klaidų valdymo posistemės komponentui turi būti prieinamas galimų klaidų aptikimo funkcionalumas. |
| 5. | Klaidų aptikimas naujojoje analizuojamos sistemos versijoje turi būti orientuotas į klasių metodų ir standartinių bibliotekų naudojamų funkcijų parametrų, „*pre-*“ sąlygų, invariantų ir „*post-*“ sąlygų tikrinimą. |

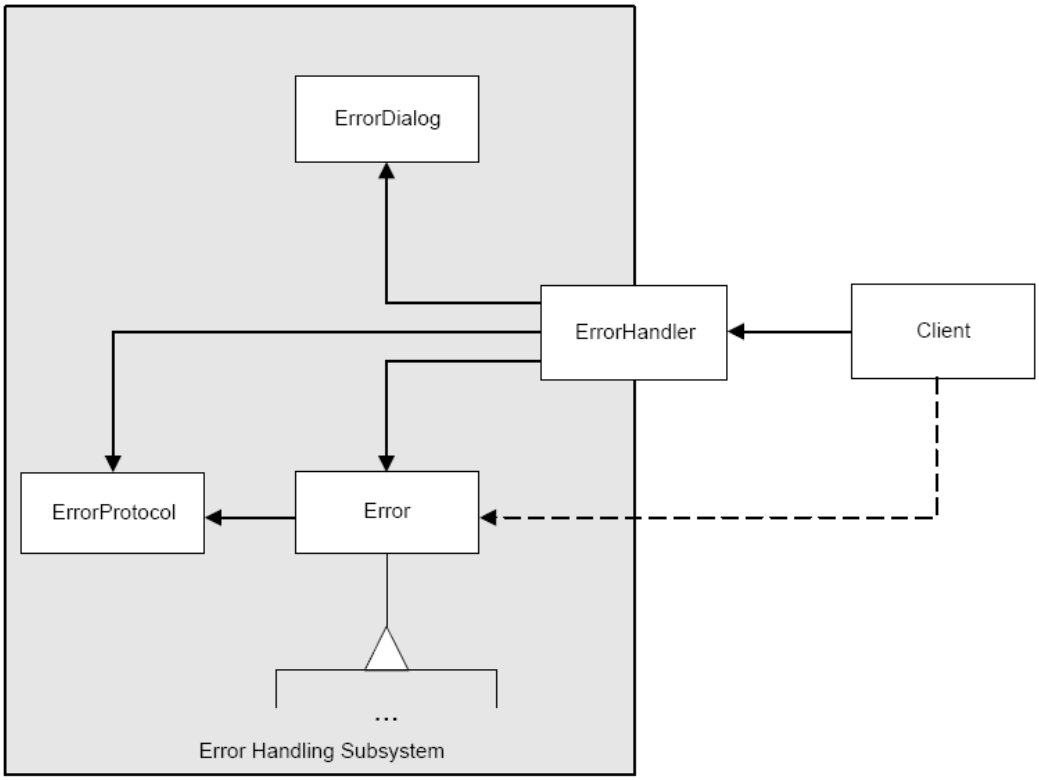
| | |
|---|---|
| 6. | Naujojoje analizuojamos sistemos versijoje turi būti apibrėžta galimų klaidų hierarchija, kurios kiekvienas lygmuo enkapsuliuotų ne tik bendrą, bet ir tam lygmeniui specifinę informaciją apie klaidos įvykį. |
| 7. | Kiekvienam klaidų hierarchijos lygmeniui turi būti apibrėžti ir realizuoti veiksmai, nurodantys, kas turi būti daroma, atsiradus numatytai to lygmens klaidai ar kitokiai susidariusiai nepageidaujamai situacijai. |
| 8. | Naujojoje analizuojamos sistemos versijoje priėjimas prie apibrėžtos klaidų hierarchijos turi būti centralizuotas ir prieinamas tik klaidų valdymo posistemei. |
| 9. | Visa informacija apie naujojoje analizuojamos sistemos versijoje aptiktas numatytas ir nenumatytas klaidas turi būti saugoma tam skirtuose failuose. |
| 10. | Naujojoje analizuojamos sistemos versijoje turi būti užtikrinta, kad atsiradus klaidai, apie jos įvykį bus atitinkamai informuota ir tą sistemą aptarnaujanti centrinė banko sistema. |
| 11. | Bet kuriam dalykinės srities posistemės komponentui turi būti prieinamas klaidų apdorojimo funkcionalumas. |
| 12. | Bet kuriam klaidų valdymo posistemės komponentui turi būti prieinamas klaidų apdorojimo funkcionalumas. |
| 13. | Naujojoje analizuojamos sistemos versijoje turi būti realizuotas būdas kas fiksuotą laiko periodą išsaugoti sistemos arba jos dalies būklę, kuri galės būti panaudojama pakartotiniam darbo nuo atitinkamo jo momento paleidimui įvykdyti. |
| 14. | Visa informacija apie naujosios analizuojamos sistemos arba jos dalies būklę turi būti saugomi tam atitinkamoje duomenų bazėje. |
| 15. | Naujojoje analizuojamos sistemos versijoje turi būti realizuotos specializuotos resursų tvarkymo priemonės, užtikrinančios nepriklausomą disponuojamų programinių resursų administravimą ir teikimą paslaugų, orientuotų į naudojimosi reikalingais resursais teisės suteikimą. |
| 16. | Specializuotos resursų tvarkymo priemonės turi būti prieinamos bet kuriam naujosios analizuojamos sistemos (tiek dalykinės srities, tiek ir klaidų valdymo posistemės) komponentui. |

| | |
|---|---|
| 17. | Naujojoje analizuojamoje sistemos versijoje turi būti realizuota centralizuota visų galimų pranešimų parinkimo ir generavimo vieta. |
| 18. | Naujojoje analizuojamos sistemos versijoje turi būti realizuoti atitinkami lygiai, kurie leistų bet kokiems galimiems (tiek klaidų, tiek ir kitokiems) pranešimams sugeneruoti teisingą, aiškią ir suprantamą jų turinio pateikimo formą. |
| 19. | Naujojoje analizuojamos sistemos versijoje turi būti realizuotas sistemos reaktivavimo iš normalią darbo eigą sutrikdžiusios klaidos ar kokios kitos nenumatytos situacijos mechanizmas, kuris geriausiu atveju užtikrintų nuo tam tikro kontrolinio taško pakartotinai aktyvuoto sistemos darbo palaikymą, vidutiniu atveju – leistų pabaigti paskutinę vykdytą operaciją vartotojo naudai, o blogiausiu atveju – bent jau suteiktų kokį nors aktyvų priėjimą *ATM* bankomato operatoriui. |
| 20. | Visų naujojoje analizuojamos sistemos versijoje esančių dalykinės srities posistemės komponentų realizacijos turi būti atitinkamai pakeistos ir / arba papildytos taip, kad atitiktų padarytus projektinius sprendimus. |

## 4.2 PIRMINĖ KLAIDŲ VALDYMO POSISTEMĖS ARCHITEKTŪRINĖ SCHEMA

Prieš pradedant kalbėti apie tai, koks galėtų būti kiekvienos klaidų valdymo modelio probleminės srities sprendimas analizuojamosios sistemos architektūros atžvilgiu, pirmiausia reikėtų pasirinkti ir bent trumpai aptarti bendrą (aukšto lygio) architektūrinę modelio pritaikymo analizuojamoje sistemoje schemą. Viena iš galimų tokių aukšto lygio pirminių[2] klaidų valdymo posistemės architektūrinių schemų galėtų būti 10 paveikslėlyje pateiktas fragmentas.

[2] Aprašomoji klaidų valdymo posistemės architektūrinė schema yra vadinama pirmine dėl to, kad ji yra bazinė ir kurią galima įsivaizduoti kaip tam tikrą pradinę idėją ar atspirties tašką, nuo kurio ji toliau šio darbo eigoje bus vystoma ir tobulinama.




**10 pav.** Pirminės klaidų valdymo posistemės architektūrinė schema ([Ren03]).

Pateiktajame paveikslėlyje yra pavaizduoti 5 klaidų valdymo posistemės architektūros komponentai: *Error*, *ErrorProtocol*, *ErrorHandler*, *ErrorDialog* ir *Client*. Kiekvieno jų apibūdinimas yra pateiktas 3 lentelėje.

**3 lentelė.** Klaidų valdymo posistemės architektūros komponentai ir jų aprašymas.

| Nr. | Komponento pavadinimas | Komponento apibūdinimas |
|---|---|---|
| 1. | *Error*<br>(„Bazinė klaidos klasė“) | Tai specializuota klasė, enkapsuliuojanti visą su įvykusia klaida ir tokio įvykio kontekstu susijusią informaciją. Bendru atveju tai yra abstrakti bazinė klasė, iš kurios gali būti išvedamos kitos konkretesnio tipo klaidos klasės, tuo pačiu gaunant ir atitinkamą klaidų hierarchiją. Tokių klasių egzemplioriai yra eiliniai objektai, kuriuos kaip ir bet kuriuos kitus objektinius duomenų tipus susikuria ir naudoja klaidų valdymo posistemės klientai. Be to, svarbu paminėti tai, kad tokios klasės objektas (tam, kad liktų nepriklausomas) turi sugebėti pats save įrašyti į klaidų protokolą *ErrorProtocol*. |

| | | |
|---|---|---|
| 2. | *ErrorProtocol*<br>(„Klaidų protokolas“) | Tai pagal *Singleton* ([GHJ+95]) projektavimo šabloną realizuojama klasė, kuri yra atsakinga už atitinkamų duomenų rašymą į duomenų registro failą („*log file*“). Bendru atveju klaidų protokolo turinį sudaro su įvykusia klaida ir tokio įvykio kontekstu susiję duomenys (pavyzdžiui, dinaminė funkcijų / procedūrų vykdymo seka, kuri gali atvaizduoti tam tikrą sistemoje vykstančių įvykių trasavimą iki pat klaidos atsiradimo momento). Paprastai klaidų protokolas turi galimybę veikti pagal konkrečius jo konfigūracinių parametrų nustatymus (pavyzdžiui, gali būti nurodoma, ar duomenys po įrašymo turi būti iškart atlaisvinti („*flushed*“), ar pildomi į specialų buferį ir pan.). |
| 3. | *ErrorHandler*<br>(“Klaidų tvarkyklė”) | Tai tiek pagal *Singleton*, tiek ir paprastai pagal *Façade* ([GHJ+95]) projektavimo šablonus realizuojama klasė, kuri enkapsuliuoja klaidų apdorojimą ir, reikalui esant, padeda suderinti skirtingus tam naudojamus klaidų apdorojimo būdus. Klaidų tvarkyklę paprastai naudoja klaidų valdymo posistemės klientai, o ji pati naudojasi klaidų protokolo ir klaidų pranešėjas paslaugomis, kad sugeneruotų reikalingą klaidos protokolą ir pavaizduotų atitinkamą klaidos pranešimą. |
| 4. | *ErrorDialog*<br>(„Klaidų pranešėjas“) | Tai klasė, kuri atlieka klaidų pranešimų išvedimą į ekraną. Dažniausiai tokie pranešimai vartotojui yra pateikiami *Modal* tipo dialogo languose. O pats klaidų pranešėjas naudojasi žemesnio lygio servisais (pavyzdžiui, *GUI-framework* ar pan.) |
| 5. | *Client*<br>(„Klaidų valdymo posistemės klientas“) | Tai bet koks sistemos, kurios dalimi yra klaidų valdymo posistemė, komponentas. Aptikęs klaidą, klientas sukuria naują bazinės klaidos klasės *Error* objektą ir užpildo jį informaciniais duomenimis. Po to atitinkamu metodu praneša jį iškvietusiam komponentui apie susidariusią klaidos situaciją ir perduoda jam sukurtą klaidos objektą. Jei apie klaidą informuotas komponentas veikia vartotojo interfeiso lygmenyje, tuomet jis gali pasinaudoti klaidų pranešėjo paslaugomis, kad šis pateiktų vartotojui atitinkamą klaidos pranešimą. |

Tokia iš pirmo žvilgsnio paprasta klaidų valdymo posistemės architektūrinė schema, be abejo, turi savo privalumų ir trūkumų, kuriuos dabar 4 lentelėje trumpai ir apžvelgsiu.

**4 lentelė.** Architektūrinės schemos privalumų ir trūkumų aprašymas.

| **Nr.** | **Privalumų / trūkumų aprašymas** |
|---|---|
| **Privalumai** | |

| | |
|---|---|
| 1. | Kadangi klaidų valdymo posistemė yra atsieta nuo visos sistemos veikimo, tai reikalui esant, pateiktoji architektūrinė schema leidžia užbaigti sistemos darbą (atlikti „*system shutdown*“) net ir iškilus rimtai klaidos situacijai. Atitinkamais klaidos pranešimais vartotojas tuomet gali būti informuojamas apie tokios baigties būtinumą, o duomenų registro faile įrašyta informacija gali būti naudinga jau detalesnei analizei atlikti. |
| 2. | Tam, kad būtų išvengta klaidų, galinčių atsirasti dėl ciklišku ryšių, pačiame klaidų apdorojimo lygmenyje (klaidų tvarkyklėse), pateiktoji architektūrinė schema neleidžia klaidų tvarkyklėms priklausyti nuo jokio sistemos funkcionalumo, naudojančio vėlgi klaidų tvarkyklių teikiamas paslaugas. |
| 3. | Taikant architektūrinę schemą, labai lengvai galima kurti ir plėsti klaidų hierarchiją. Tai leidžia atlikti enkapsuliuota bazinė klaidos klasė *Error*. |
| 4. | Architektūrinė schema pateikia gana paprastą ir lankstų būdą perduoti klaidas (kaip objektus) į aukštesnius lygius, taip perduodant reikalingą informaciją iki tos sistemos vietos, kurioje ji bus panaudojama. |
| **Trūkumai** | |
| 1. | Kadangi pavyzdinė analizuojamoji sistema ir nėra realaus laiko ar kritinio veikimo (kai klaida arba jos pašalinimo greitis sistemoje gali kainuoti gyvybę), tai su klaidų valdymu susijusių operacijų atlikimo sparta šiuo atveju nėra labai svarbus akcentas (nes pačios klaidos, lyginant jas su normaliu tikėtinu sistemos veikimu, bet kuriuo atveju yra išimtinės, o joms atsiradus, didesnis dėmesys vis tiek labiau bus skiriamas sistemos patikimumo ir sistemos veikimo korektiškumo atstatymui, t.y. jų kokybei, o ne greičiui). Dėl to operacijoms, kurios yra reikalingos klaidų registravimui, apdorojimui ir raportavimui, didelio spartos optimizavimo nereikia. Vis dėlto, prieš atliekant klaidų apdorojimą, yra būtina tas klaidas aptikti. Tam tikslui sistemos komponentų kodas turi būti „apkraunamas“ įvariomis operacijomis, kurios leidžia tai atlikti, bet lėtina visos sistemos darbą. Dėl to, galvojant apie architektūrinės schemos praktinį, reikia ypatingai atkreipti dėmesį ir derinti pageidaujamą balansą tarp sistemos darbo našumo ir jos patikimumo. |
| 2. | Nors architektūrinėje schemoje visas klaidų valdymo kompleksiškumas ir yra enkapsuliuotas į atskirus klaidų valdymo posistemės komponentus, vis dėlto kodas, kuris yra reikalingas atlikti klaidų aptikimą bei atitinkamus klaidų tvarkyklių iškvietimus, turi būti įtrauktas į kiekvieną klaidų valdymo posistemės kliento metodą. |
| 3. | Kadangi klaidų tvarkyklių komponentus paprastai yra stengiamasi padaryti kiek įmanoma paprastesnius, nepriklausomus ir tik tam dedikuotus, tai kartais gali kilti problemų su realizacijos, reikalingos sistemoje palaikyti tokią architektūrinę schemą, kiekio padidėjimu ar net sąlyginiu dubliavimu. Pavyzdžiui, jei reikia, kad sistemoje turimi standartiniai pranešimų apdorojimo servisai teiktų vartotojui pranešimus ir apie klaidas, be to, jei taip pat yra reikalinga, kad kažkas (ne klaidų tvarkyklėse) būtų atsakingas už vartotojo informavimą apie kokias nors ypatingas situacijas (pavyzdžiui, galimus pagalbos iškvietimus), tuomet gali reikėti suprojektuoti ir realizuoti dvi atskiras pranešimų apdorojimo posistemes (jei nenorime, kad pranešimų apdorojimo kodas būtų įtrauktas į klaidų tvarkyklių komponentus). |

| | |
|---|---|
| 4. | Bendru atveju egzistuoja viena problema, su kuria tikrai gali tekti susidurti architektūrinės schemos realizavimo sistemoje etape, tai jos suderinamumas su kitais programiniais komponentais (pavyzdžiui, bibliotekomis, nutolusiomis programomis, kitokios nei ši infrastruktūros sistemomis ir pan.). Taigi pagrindinis tikslas tokiu atveju yra tiesiog surasti tinkamą (ar net tinkamiausią) būdą, kuris leistų suderinti visų tarpusavio sėkmingą komunikavimą ir veikimą. |
| 5. | Kai kuriais atvejais architektūrinę schemą gali tekti tobulinti dėl lygiagretaus arba išskirstyto sistemos(-ų) veikimo, nes pateiktosios schemos realizacija yra paremta procedūriniu (klaidas apdorojančių tvarkyklių) iškvietimu. Jei toks veikimo principas konkrečiam atvejui yra netinkamas, gali tekti jį keisti (pavyzdžiui, naudoti visų sistemos komponentų būklės stebėtojo („*observer*“) komponentą(-us) arba taikyti kitus dinaminių struktūrų projektavimo šablonus (kaip „*Strategized Decorator*“, kuriuo galima sukurti pakankamai lanksčią kelių lygių („*layers*“) architektūrą ([ERR+01]).) ir pan.). |

Lentelėje yra pateikti tik patys bendriausi ir akivaizdžiausi klaidų valdymo posistemės architektūrinės schemos privalumai ir trūkumai. Be abejonės, sulig kiekvienu bandymu nagrinėti jos praktinį panaudojamumą skirtinguose kontekstuose, panašių privalumų ir trūkumų išryškėtų dar daugiau, tačiau tikslas yra ne juos visus išrašyti, o apibendrintai susipažinti su pasirinktosios schemos stipresnėm ir silpnesnėm pusėm. Ir nors einamuoju momentu lentelėje kiekybiniu atžvilgiu pirmenybę turi trūkumai, tačiau vėlgi pirminis tikslas yra ne surasti tobulą klaidų valdymo modelio pritaikymo analizuojamos sistemos architektūros atžvilgiu sprendimą, o apskritai parodyti, kad tikrai egzistuoja bent vienas toks sprendimas.Visa kita yra sugebėjimo įvertinti gautus rezultatus galioje.

## 4.3 KLAIDŲ IDENTIFIKAVIMAS IR STRUKTŪRIZAVIMAS

### 4.3.1 Problema

Bandant geriau suvokti, nuo ko čia būtų galima pradėti, pravartu būtų iškelti tokius 2 pirminius klausimus: kas galėtų charakterizuoti klaidą ir pagal ką galima kategorizuoti skirtingas klaidų grupes?

### 4.3.2 Sprendimas

Į projektuojamą klaidos klasę bendru atveju pagal poreikius galima įtraukti tokius duomenis:

1. Klaidos lokalizavimo duomenys („*localization information*“) – tai duomenys, kurie padeda nustatyti tikslią klaidos atsiradimo vietą. Analizuojamosios sistemos architektūros atveju šie duomenys galėtų būti tokie:

    - Pirminio kodo failo vardas („*source_ file_name*“)
    - Klasės pavadinimas („*class_name*“)
    - Metodo pavadinimas („*method_name*“)
    - Kodo eilutės numeris („*line_number*“).

2. Vartotojo vardas („*user_name*“) – tai vartotojo, kuris klaidos atsiradimo metu dirbo su sistema, arba kuris yra atsakingas už tam tikrą sistemos funkcionalumo realizaciją, vardas. Jis gali būti naudingas, atliekant klaidos analizę arba net automatizuotą klaidos apdorojimą (pavyzdžiui, užduodant vartotojui su susiklosčiusia situacija susijusius klausimus ir pan.).

3. Vartotojo sąskaitos numeris („*user_account*“) – tai tai vartotojo, kuris klaidos atsiradimo metu dirbo su sistema, banko sąskaitos numeris.

4. Klaidos numeris („*error_number*“) – tai unikalus vartotojo darbo su sistema metu kilusios klaidos identifikatorius.

5. Klaidos kategorija („*error_category*“) – tai kategorija arba grupė, kuriai galima būtų priskirti vartotojo darbo su sistema metu kilusią klaidą.

6. Klaidos aptikimo laikas („*error_detection_time*“) – tai laikas, kada sistemoje buvo aptikta klaida. Tai vėlgi gali būti naudinga, atliekant klaidos analizę (ypač, jei yra manoma, jog klaida yra susijusi su laikiniais parametrais).

7. Klaidos aprašas („*error_description*“) – laisvesnės paskirties aprašas, kuriame yra nurodoma išsamesnė informacija, paaiškinanti klaidos aplinkybes. Siekiant didesnio tikslumo ir lankstumo, klaidos aprašas gali būti parametrizuotas, suskaidant bendrą aprašo turinį į atitinkamus laukus. Be to, jei sistemoje yra numatyta galimybė keisti šiuos aprašus, tada taip pat reikia nuspręsti, kur tie aprašai bus saugomi (pavyzdžiui, pačiuose klaidos objektuose kaip dinaminiai duomenys, duomenų bazėje, failuose ar pan.), nes tai lemia ir jų pasiekiamumą klaidos atveju,

pakeitimų apdorojimo greitį bei sudėtingumą. Kadangi vis tik gera praktika yra klaidų aprašus saugoti kokiame nors faile, tai taip ir darysime – klaidų aprašas nurodys failo vardą, kuriame bus saugomas klaidos aprašymas. Paties klaidų aprašo turinio kol kas į detalesnius laukus neskaidysime.

Visi šie duomenys ir gali sudaryti bazinę klaidos klasę, t.y. būti jos atributais. Tolimesnis žingsnis būtų išskirti galimas klaidas ir jas struktūrizuoti. Kartais literatūroje klaidų struktūrizavimas yra traktuojamas kaip visos sistemos struktūrizavimas, nes tai reiškia, kad pirmiausia klaidos bus skirstomos į 2 stambias grupes: pačių klaidų apdorojimo komponentų (standartinių, nestandartinių) klaidos ir specifinės likusiems sistemos komponentams klaidos. Mūsų atveju visos klaidos taip pat bus skirstomos į 2 stambias grupes, tačiau kiek kitokias, būtent, į dalykinės srities klaidas ir techninės srities klaidas (žr. 5 lentelę).

**5 lentelė.** Dalykinės ir techninės srities klaidos bei jų aprašymas.

| Nr. | Klaidos pavadinimas | Klaidos aprašymas | Ar yra tokia klaida numatyta ATM sistemos realizacijoje? |
|---|---|---|---|
| **Dalykinės srities klaidos** | | | |
| 1. | **ATM_UNABLE_TO_SWITCH_ON** | Klaida, kylanti tuomet, kai pats bankomatas negali įjungti nei vienos iš savo sistemų (įrenginių). | — |
| 2. | **ATM_UNABLE_TO_SWITCH_OFF** | Klaida, kylanti tuomet, kai visos bankomato sistemos (įrenginiai) yra išjungtos, bet pats bankomatas dėl kažkokios priežastis negali išsijungti. | — |
| 3. | **CARD_READER_INVALID_PIN** | Klaida, kylanti tuomet, kai banko kortelių skaitytuvas nuskaito neteisingą vartotojo suvestą PIN kodą. | √ |
| 4. | **CARD_READER_CARD_UNREADABLE** | Klaida, kylanti tuomet, kai banko kortelių skaitytuvas (pavyzdžiui, dėl apgadinto paviršiaus) negali nuskaityti kortelės ir jos atpažinti. | √ |
| 5. | **CARD_READER_CARD_BLACK_LISTED** | Klaida, kylanti tuomet, kai banko kortelių skaitytuvo nuskaityta kortelė yra juodųjų sąraše (pavyzdžiui, yra | — |

| | | | |
|---|---|---|---|
| | | vogta). | |
| 6. | **CARD_READER_CARD_JAMMED** | Klaida, kylanti tuomet, kai banko kortelių skaitytuve kortelės priėmimo / grąžinimo metu ji netikėtai užstringa. | — |
| 7. | **CARD_READER_CARD_EXPIRED** | Klaida, kylanti tuomet, kai banko kortelių skaitytuvas nuskaitytą kortelę atpažino kaip nebegaliojančią. | — |
| 8. | **CARD_READER_CARD_UNSUPPORTED** | Klaida, kylanti tuomet, kai banko kortelių skaitytuvas nepalaiko tokios rūšies kortelių (pavyzdžiui, kito banko kortelių). | √ |
| 9. | **CASH_DISPENSER_CASH_JAMMED** | Klaida, kylanti tuomet, kai pinigų išdavimo įrenginys pinigų grąžinimo metu jie netikėtai užstringa. | — |
| 10. | **CASH_DISPENSER_OUT_OF_CASH** | Klaida, kylanti tuomet, kai pinigų išdavimo įrenginys nebeturi daugiau iš ko išmokėti pinigų (yra tuščias). | — |
| 11. | **ENVELOPE_ACCEPTOR_ENVELOPE_SIZE_UNRECOGNIZED** | Klaida, kylanti tuomet, kai pinigų priėmimo įrenginys negali atpažinti priimamos valiutos dydžio. | — |
| 12 | **ENVELOPE_ACCEPTOR_ENVELOPE_FORMAT_UNRECOGNIZED** | Klaida, kylanti tuomet, kai pinigų priėmimo įrenginys atpažino priimamos valiutos tipą, tačiau neatpažino jos formato (pavyzdžiui, jei valiuta padirbta). | — |
| 13. | **ENVELOPE_ACCEPTOR_CURRENCY_NOT_IN_USE** | Klaida, kylanti tuomet, kai pinigų priėmimo įrenginys atpažino valiutą, kurią jis anksčiau priimdavo, bet daugiau nebepriima (pavyzdžiui, ką tik valstybiniu mastu pasikeitus valiutai). | — |
| 14. | **ENVELOPE_ACCEPTOR_CURRENCY_UNSUPPORTED** | Klaida, kylanti tuomet, kai pinigų priėmimo įrenginys atpažįsta, kad priimama valiuta egzistuoja, tačiau jos einamuoju momentu nepalaiko (pavyzdžiui, kad ir pasibaigus bankomate vienos kurios nors valiutos kupiūroms (doleriams ar pan.)). | — |
| 15. | **ENVELOPE_ACCEPTOR_NO_SPACE_LEFT_FOR_CASH** | Klaida, kylanti tuomet, kai pinigų priėmimo įrenginys fiziškai nebepajėgia priimti daugiau pinigų, nes tam skirta vieta yra jau pilna. | — |

| | | | |
|---|---|---|---|
| 16. | **ENVELOPE_ACCEPTOR_ENVELOPE_DEPOSIT_TIMED_OUT** | Klaida, kylanti tuomet, kai pinigų priėmimo įrenginys nesulaukia pinigų pateikimo iš vartotojo. | √ |
| 17. | **RECEIPT_PRINTER_OUT_OF_PAPER** | Klaida, kylanti tuomet, kai kvitų spausdintuvas nebeturi daugiau popieriaus, ant kurio galėtų atspausdinti kvitą. | — |
| 18. | **RECEIPT_PRINTER_OUT_OF_INK** | Klaida, kylanti tuomet, kai kvitų spausdintuvas nebeturi daugiau kasetėje rašalo, kad galėtų atspausdinti kvitą. | — |
| 19. | **RECEIPT_PRINTER_RECEIPT_JAMMED** | Klaida, kylanti tuomet, kai kvitų spausdintuve kvito spausdinimo metu jis netikėtai užstringa. | — |
| 20. | **WITHDRAW_TRANSACTION_ACCOUNT_BLOCKED** | Klaida, kylanti tuomet, kai pinigų išėmimo operacijos vykdymo metu paaiškėja, kad vartotojo sąskaita yra užblokuota. | — |
| 21. | **WITHDRAW_TRANSACTION_INVALID_AMOUNT_SPECIFIED** | Klaida, kylanti tuomet, kai pinigų išėmimo operacijos vykdymo metu paaiškėja, kad vartotojas neteisingai nurodė pageidaujamą išsiimti pinigų sumą. | √ |
| 22. | **WITHDRAW_TRANSACTION_INSUFFICIENT_AVAILABLE_BALANCE** | Klaida, kylanti tuomet, kai pinigų išėmimo operacijos vykdymo metu paaiškėja, kad tokiai pinigų sumai, kokią nurodė vartotojas, išduoti neužtenka sąskaitoje pinigų (per mažas balansas). | √ |
| 23. | **WITHDRAW_TRANSACTION_DAILY_WITHDRAWL_LIMIT_EXCEEDED** | Klaida, kylanti tuomet, kai pinigų išėmimo operacijos vykdymo metu paaiškėja, kad maksimali leistina išsiimti dienos pinigų suma jau yra išeikvota. | √ |
| 24. | **WITHDRAW_TRANSACTION_WITHDRAWL_LIMIT_EXCEEDED** | Klaida, kylanti tuomet, kai pinigų išėmimo operacijos vykdymo metu paaiškėja, kad maksimali išdavimui nustatyta pinigų suma jau yra išeikvota. | — |
| 25. | **DEPOSIT_TRANSACTION_ACCOUNT_BLOCKED** | Klaida, kylanti tuomet, kai pinigų padėjimo į sąskaitą operacijos vykdymo metu paaiškėja, kad vartotojo sąskaita yra užblokuota. | — |

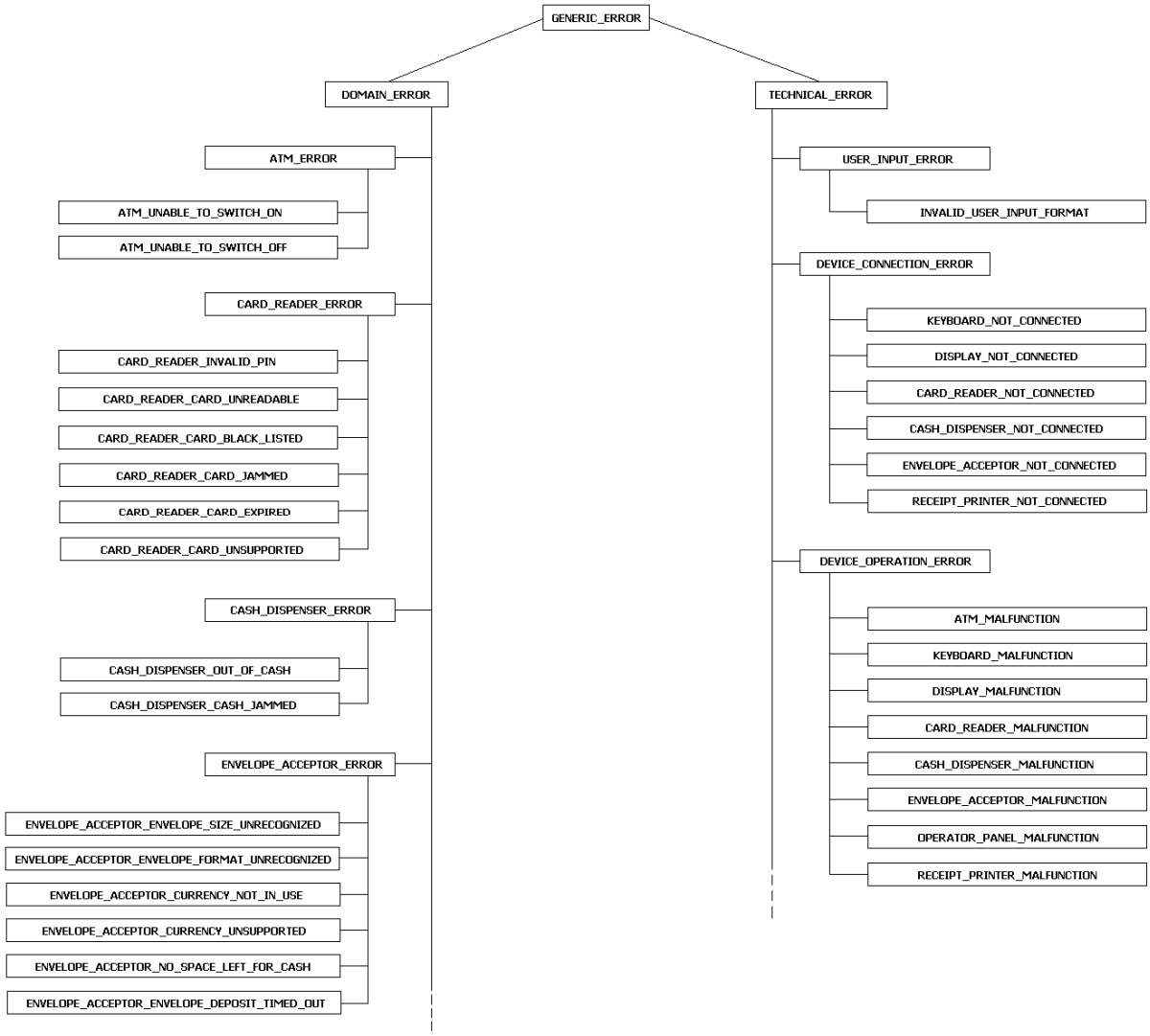
| | | | |
|---|---|---|---|
| 26. | **DEPOSIT_TRANSACTION_INVALID_AMOUNT_SPECIFIED** | Klaida, kylanti tuomet, kai pinigų padėjimo į sąskaitą operacijos vykdymo metu paaiškėja, kad vartotojas neteisingai nurodė pinigų sumą. | √ |
| 27 | **DEPOSIT_TRANSACTION_DEPOSIT_LIMIT_EXCEEDED** | Klaida, kylanti tuomet, kai pinigų padėjimo į sąskaitą operacijos vykdymo metu paaiškėja, kad maksimali priėmimui nustatyta pinigų suma yra viršijama. | — |
| 28. | **TRANSFER_TRANSACTION_ACCOUNT_BLOCKED** | Klaida, kylanti tuomet, kai pinigų pervedimo tarp sąskaitų operacijos vykdymo metu paaiškėja, kad kurio nors vartotojo sąskaita yra užblokuota. | — |
| 29. | **TRANSFER_TRANSACTION_INVALID_ACCOUNT_NUMBER** | Klaida, kylanti tuomet, kai pinigų pervedimo tarp sąskaitų operacijos vykdymo metu paaiškėja, kad nurodytas gavėjo sąskaitos numeris yra neteisingas (pavyzdžiui, tokios sąskaitos nėra ar pan.). | √ |
| 30. | **TRANSFER_TRANSACTION_INSUFFICIENT_AVAILABLE_BALANCE** | Klaida, kylanti tuomet, kai pinigų pervedimo tarp sąskaitų operacijos vykdymo metu paaiškėja, kad tokiai pinigų sumai, kokią nurodė vartotojas, pervesti neužtenka sąskaitoje pinigų (per mažas balansas). | √ |
| 31. | **TRANSFER_TRANSACTION_DAILY_TRANSFER_LIMIT_EXCEEDED** | Klaida, kylanti tuomet, kai pinigų pervedimo tarp sąskaitų operacijos vykdymo metu paaiškėja, kad maksimali leistina pervedimui skirta dienos pinigų suma jau yra išeikvota. | √ |
| 32. | **TRANSFER_TRANSACTION_TRANSFER_LIMIT_EXCEEDED** | Klaida, kylanti tuomet, kai pinigų pervedimo tarp sąskaitų operacijos vykdymo metu paaiškėja, kad maksimali pervedimui skirta pinigų suma yra viršijama. | — |
| 33. | **TRANSFER_TRANSACTION_INVALID_RECIPIENT_DATA_SPECIFIED** | Klaida, kylanti tuomet, kai pinigų pervedimo tarp sąskaitų operacijos vykdymo metu paaiškėja, kad vartotojas klaidingai nurodė gavėjo duomenis (pavyzdžiui, laukelyje, kur turi būti skaitinė reikšmė, vartotojas nurodė simbolių eilutę ar pan.). | — |
| **Techninės srities klaidos** | | | |

| | | | |
|---|---|---|---|
| 1. | **INVALID_USER_INPUT_FORMAT** | Klaida, kylanti tuomet, kai duomenų įvedimo metu vartotojas neteisingai nurodo duomenis (ne pagal formatą ar reikalaujamą duomenų tipą). | √ |
| 2. | **KEYBOARD_NOT_CONNECTED** | Klaida, kylanti tuomet, kai bankomato krovimosi arba darbo metu paaiškėja, kad su klaviatūra nėra fizinio kontakto (ryšio). | — |
| 3. | **DISPLAY_NOT_CONNECTED** | Klaida, kylanti tuomet, kai bankomato krovimosi arba darbo metu paaiškėja, kad su vaizduokliu nėra fizinio kontakto (ryšio). | — |
| 4. | **CARD_READER_ NOT_CONNECTED** | Klaida, kylanti tuomet, kai bankomato krovimosi arba darbo metu paaiškėja, kad su banko kortelių skaitytuvu nėra fizinio kontakto (ryšio). | — |
| 5. | **CASH_DISPENSER_ NOT_CONNECTED** | Klaida, kylanti tuomet, kai bankomato krovimosi arba darbo metu paaiškėja, kad su pinigų išdavimo įrenginiu nėra fizinio kontakto (ryšio). | — |
| 6. | **ENVELOPE_ACCEPTOR_ NOT_CONNECTED** | Klaida, kylanti tuomet, kai bankomato krovimosi arba darbo metu paaiškėja, kad su pinigų priėmimo įrenginiu nėra fizinio kontakto (ryšio). | — |
| 7. | **RECEIPT_PRINTER_NOT_CONNECTED** | Klaida, kylanti tuomet, kai bankomato krovimosi arba darbo metu paaiškėja, kad su kvitų spausdintuvu nėra fizinio kontakto (ryšio). | — |
| 8. | **KEYBOARD_MALFUNCTION** | Klaida, kylanti tuomet, kai bankomato krovimosi arba darbo metu klaviatūros veikimas pradeda striginėti (nors fizinis ryšys gali ir egzistuoti). | — |
| 9. | **DISPLAY_MALFUNCTION** | Klaida, kylanti tuomet, kai bankomato krovimosi arba darbo metu vaizduoklio veikimas pradeda striginėti (nors fizinis ryšys gali ir egzistuoti). | — |
| 10. | **CARD_READER_MALFUNCTION** | Klaida, kylanti tuomet, kai bankomato krovimosi arba darbo metu banko kortelių skaitytuvo veikimas pradeda striginėti (nors fizinis ryšys gali ir egzistuoti). | — |

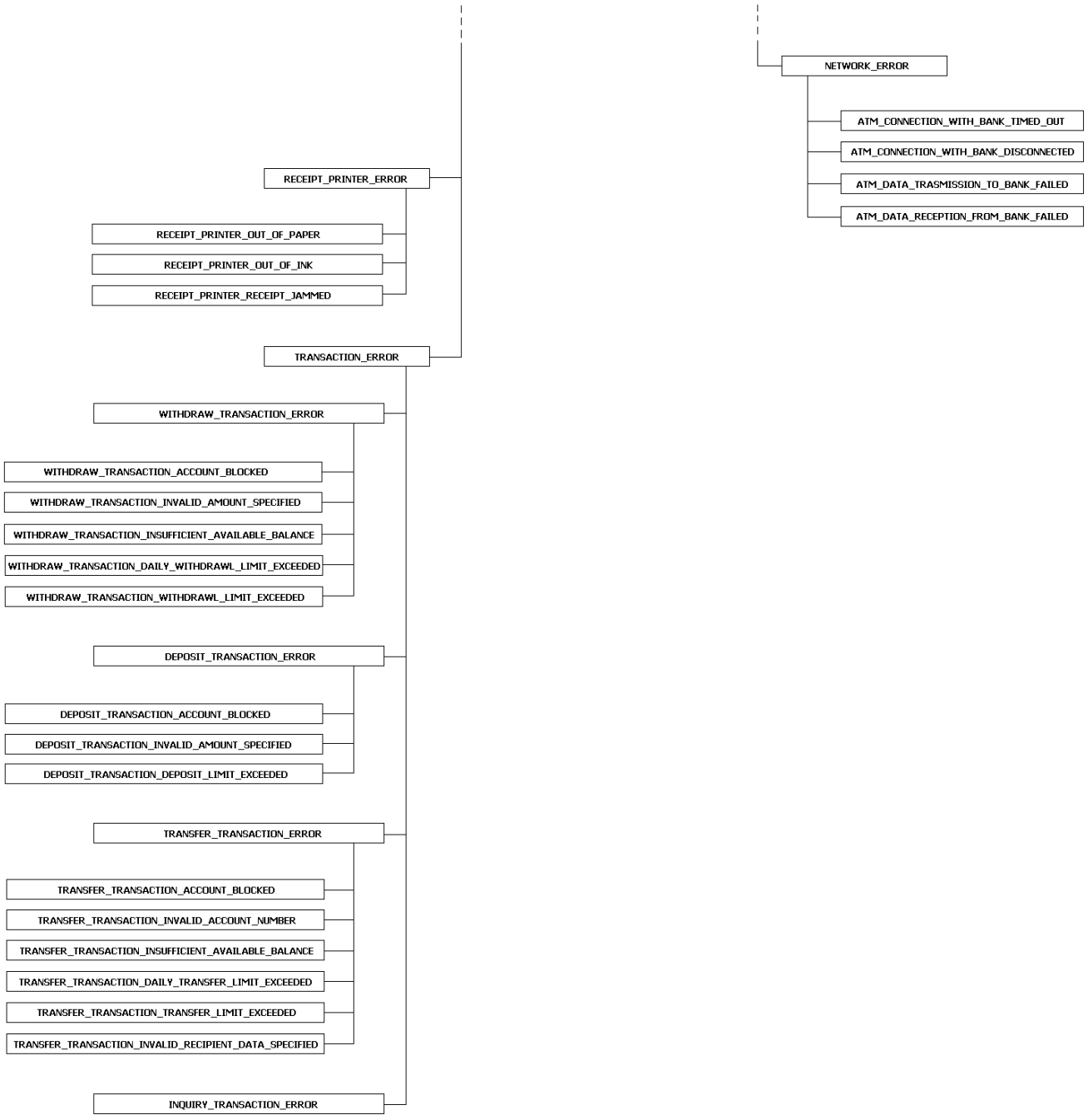
| | | | |
|---|---|---|---|
| 11. | **CASH_DISPENSER_MALFUNCTION** | Klaida, kylanti tuomet, kai bankomato krovimosi arba darbo metu pinigų išdavimo įrenginio veikimas pradeda striginėti (nors fizinis ryšys gali ir egzistuoti). | — |
| 12. | **ENVELOPE_ACCEPTOR_MALFUNCTION** | Klaida, kylanti tuomet, kai bankomato krovimosi arba darbo metu pinigų priėmimo įrenginio veikimas pradeda striginėti (nors fizinis ryšys gali ir egzistuoti). | — |
| 13. | **RECEIPT_PRINTER_MALFUNCTION** | Klaida, kylanti tuomet, kai bankomato krovimosi arba darbo metu kvitų spausdintuvo veikimas pradeda striginėti (nors fizinis ryšys gali ir egzistuoti). | — |
| 14. | **OPERATOR_PANEL_MALFUNCTION** | Klaida, kylanti tuomet, kai bankomato krovimosi arba darbo metu *ATM* operatoriaus terminalo veikimas pradeda striginėti (nors fizinis ryšys gali ir egzistuoti). | — |
| 15. | **ATM_MALFUNCTION** | Klaida, kylanti tuomet, kai bankomato krovimosi arba darbo metu jo veikimas pradeda striginėti. | — |
| 16. | **ATM_CONNECTION_WITH_BANK_TIMED_OUT** | Klaida, kylanti tuomet, kai bankomatas krovimosi arba darbo metu nesulaukia prisijungimo prie banko. | — |
| 17. | **ATM_CONNECTION_WITH_BANK_DISCONNECTED** | Klaida, kylanti tuomet, kai bankomato krovimosi arba darbo metu dėl kažkokios priežasties nutrūksta tinklo ryšys su banku. | — |
| 18. | **ATM_DATA_TRANSMISSION_TO_BANK_FAILED** | Klaida, kylanti tuomet, kai bankomato krovimosi arba darbo metu duomenų siuntimo tinklu bankui operacija buvo atlikta nesėkmingai. | — |
| 19. | **ATM_DATA_RECEPTION_FROM_BANK_FAILED** | Klaida, kylanti tuomet, kai bankomato krovimosi arba darbo metu duomenų priėmimo tinklu iš banko operacija buvo atlikta nesėkmingai. | — |

Kaip jau turbūt pastebėjote, lentelėje yra pateiktos tik tos klaidos, kurios yra susijusios tik su pačiu bankomato ir jo veikimu, bet ne su banku, nes pradinis iškeltas tikslas ir buvo nagrinėti tik pasirinktąją pavyzdinę analizuojamosios sistemos architektūrą.

Kitas žingsnis yra gana paprastas – lentelėje surašytas klaidas tereikia tik struktūrizuoti į atitinkamą klaidų klasių hierarchiją. Šis darbas jau yra padarytas ir gautas rezultatas yra pateiktas 11, 12 paveikslėliuose (diagrama yra padalinta į dvi dalis dėl įskaitomumo).




**11 pav.** Pirmoji klaidų klasių hierarchijos dalis.




**12 pav.** Antroji klaidų klasių hierarchijos dalis.

Taigi iš pateiktųjų diagramų matyti, kad be anksčiau šiame darbe jau pristatytų dalykų galime išskirti šiuos dar neminėtus aspektus:

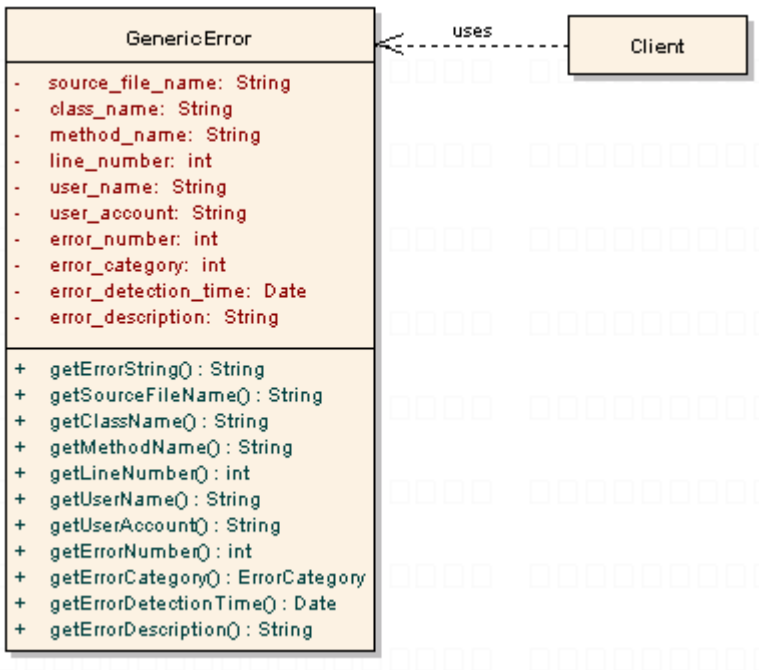
- Aiškumo vardan bazinė klaidos klasė *Error* yra pervadinta *GenericError* vardu.
- Dalykinės ir techninės srities klaidų klases atspindi klasės atitinkamai *DomainError* ir *TechnicalError*.
- *DomainError* išvestinės klasės yra:
    - *ATMError*

- *CardReaderError*
- *CashDispenserError*
- *EnvelopeAcceptorError*
- *ReceiptPrinterError*
- *TransactionError* (ši turi dar ir savo išvestines klases *WithdrawTransactionError, DepositTransactionError, TransferTransactionError* ir *InquiryTransactionError*)

- *TechnicalError* išvestinės klasės yra:
  - *UserInputError*
  - *DeviceConnectionError*
  - *DeviceOperationError*
  - *NetworkError*

### 4.3.3 Realizacija

Bazinę klaidos klasę struktūriškai galima pavaizduoti taip (žr. 13 paveikslėlyje):




**13 pav.** Bazinės klaidos klasės struktūrinis pavaizdavimas.

### 4.3.4 Galimos pasekmės

Galimas pasekmes suformuluosiu kaip iki šiol 3.2 poskyryje pateiktų sprendimų apibendrinamuosius privalumus ir trūkumus (žr. 6 lentelę), nes visi padaryti samprotavimai kol kas egzistuoja tik planiniam lygmenyje, o tai reiškia, kad norint išvysti kokias nors realias pasekmes, visa tai reikėtų pilnai realizuoti ir užsiimti testavimu, o tai nėra šio darbo tikslas.

**6 lentelė.** 3.2 poskyryje priimtų sprendimų privalumų ir trūkumų aprašymas.

| Nr. | Privalumų / trūkumų aprašymas |
|---|---|
| **Privalumai** | |
| 1. | Visų apibrėžtų su klaidomis susijusių duomenų enkapsuliavimas į vieną bendrinę klaidos klasę, padidina tokio duomenų rinkinio priežiūros ir eksploatavimo laipsnį („*maintainability*“) bei leidžia išvengti globalių duomenų (kintamųjų ir pan.). |
| 2. | Apibrėžus bendrinę klaidos klasę, ją kaip duomenų tipą galės naudoti bet kuris tai pasiekiantis sistemos komponentas, o taip pat ir, reikalui esant, perduoti vis kitam komponentui ir t.t. |
| 3. | Sistemos komponentai yra atsieti nuo konkrečios klaidos kaip duomenų tipo realizacijos, o tai nesulieja dviejų skirtingų probleminių sričių kodus į vieną masę ir leidžia lokalizuoti kylančias problemas individualiuose kontekstuose. |
| 4. | Klaidų perdavimo į aukštesnius sistemos lygmenis galimybė nereikalauja, kad apie atsiradusią klaidą būtų pranešama arba į ją būtų reaguojama iš karto, kai tik ji įvyksta. Tai kartais leidžia prieš klaidų apdorojimą dar atlikti reikalingus paruošiamuosius darbus, nusiųsti surinktą informaciją apie klaidą svarbiam jos gavėjui ir pan. |
| **Trūkumai** | |
| 1. | Bene pagrindinė bėda yra ta, kad realizuojant didelę klaidų klasių hierarchiją (kad ir tokią, kokia buvo gauta mūsų atveju), bus gaunamas labai didelis klasių skaičius. O tai reiškia, kad smarkiai padidėja ir bendras sistemos resursų suvaldymo sudėtingumas (atsiranda daugiau kodo, didesnė sistemos vykdomųjų dalių apimtis, lėtėja operavimo sparta ir pan.). |

## 4.4 KLAIDŲ APTIKIMAS

### 4.4.1 Problema

Prieš pradedant kalbėti apie klaidų apdorojimą procesą, pirmiausia reikėtų turbūt pradėti nuo srities, vadinamos klaidų aptikimu. Iš pradžių reikia bent pabandyti atsakyti į šiuos klausimus: o kokie yra tie požymiai, kurie gali būti naudojami aptinkant klaidas? Kurioje

analizuojamosios sistemos architektūros vietoje pagal pasirinktą klaidų valdymo posistemės architektūrinę schemą galėtų būti realizuotas klaidų izoliavimo funkcionalumas („*error traps*“)?

### 4.4.2 Sprendimas

Viena iš idėjų, kuria galima remtis, sprendžiant klaidų aptikimo problemą, yra lyginti analizuojamosios sistemos būklę ir elgseną su kokia nors iš anksto paruošta jos specifikacija, kuri pagal pradinę prielaidą būtų laikoma teisinga. Kitaip tariant, bet koks tikrinamas nukrypimas nuo sutartinai teisingos sistemos specifikacijos bus laikoma klaida. Siekiant, kad ši idėja aprėptų kuo platesnį įvairių klaidų sistemoje aptikimo spektrą, pageidautina, kad būtų tikrinamos visos vietos, kur jos gali kilti. Tokios tinkamiausios vietos būtų metodai, todėl į juos toliau ir bus orientuojamasi. Taigi konkretizuojant pasiūlytą idėją, pagrindiniai jos įgyvendinimo uždaviniai būtų parašyti specifikaciją, kuri būtų naudojama aptikti klaidoms kiekviename klaidų valdymo posistemės kliento metode. Pati specifikacija gali būti sudaroma kiekvienam analizuojamosios sistemos metodui atskirai, naudojant vadinamąsias „*pre-*“ ir „*post-*“ sąlygas ir invariantus. Kas ir kur konkrečiai galėtų būti tikrinama metode yra nurodyta 7 lentelėje.

**7 lentelė.** Patikrinimui metode skirtos klaidos situacijos.

| Nr. | Specifikuojama klaidos situacija | Tikrinimo momentas |
|---|---|---|
| 1. | Neteisingai nurodyti metodo parametrai („*invalid parameters*“). | Metodo iškvietimo metu. |
| 2. | „*Pre-*“ sąlygos netenkinimas („*violation of the precondition*“). | Metodo iškvietimo metu. |
| 3. | Nepageidaujami į specifikuojamą metodą įeinančių metodų grąžinami rezultatai ar kitoks nepriimtinas jų elgesys („*unexpected results or failures of methods called by this method*“). | Iškart po metodo iškvietimo. Be to, jei naudojama programavimo kalba palaiko klaidos situacijų „gaudymą“ ir jos pakanka įvardinti galimą nepageidaujamą atvejį, tuomet jokio papildomo kodo tam galima ir nenaudoti. |

| | | |
|---|---|---|
| 4. | Metodui nurodytų invariantų netenkinimas („*violation of a method's invariant*“). | Šiuo atveju reikėtų išskirti dvi invariantų rūšis. Kai kurie invariantai gali būti susiję su visa klase ir jie paprastai yra tikrinami kiekvieno tos klasės metodo darbo pabaigoje (pavyzdžiui, prieš pat sakinį *return* ar pan.). To visai pakanka, jei tos klasės duomenys (atributai) yra naudojami tik jos pačios metodų. Jei invariantas aprėpia „*shared*“ tipo klasių duomenis („*aliasing*“ atveju) arba jei invariantas ne visiems vienos klasės metodams apima vienodai galiojančių salygų rinkinį, tuomet būtina, kad invariantas būtų tikrinamas tiek metodo iškvietimo metu, tiek ir jo darbo pabaigoje. Pastaruoju atveju invariantas taipogi gali būti išreiškiamas kaip dalis „*pre-*“ arba „*post-*“ sąlygos. |
| 5. | „*Post-*“ sąlygos netenkinimas („*violation of the postcondition*“). | Metodo darbo pabaigoje. |

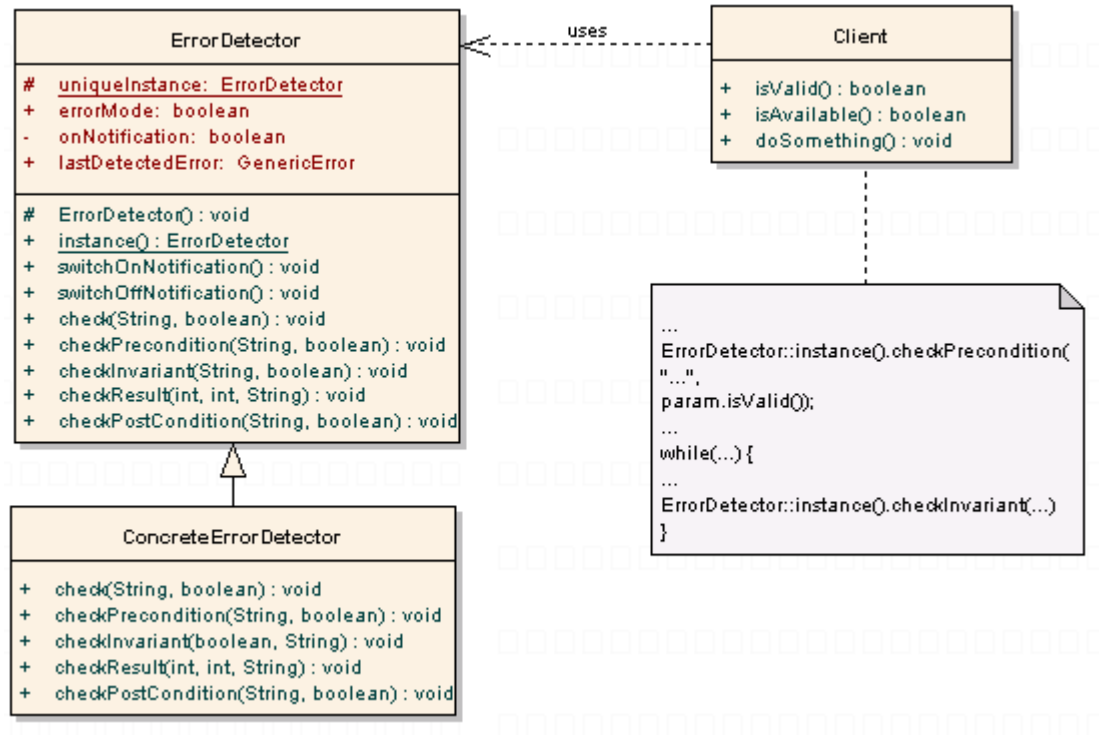
Bendru atveju priklausomai nuo individualių poreikių ir turimos sistemos sudėtingumo lentelėje nurodyta specifikuojamų klaidos situacijų aibė gali būti plečiama. O konkrečiau bendrąją tikrinimo metoduose schemą galima būtų pavaizduoti taip (žr. 1 pavyzdį):

**1 pavyzdys.** Bendroji tikrinimo metoduose schema.

```
METHOD AnyMethod(aType1 aParam1, aType2 aParam2, …) : aReturnType
BEGIN
------- error detection header -----------------------

[ aParam1 valid ? raise exception for invalid parameter ];
[ aParam2 valid ? raise exception for invalid parameter ];
[ invariant holds ? raise exception for violated invariant ];
[ precondition holds ? raise exception for violated precondition ];

-------- normal method body ---------------------------
…
-- do something

[ special test ? raise exception ];

Result = aClass.OtherMethod(aValue);
[ exptected Result ? raise exception ];
…
-------- error detection footer -------------------------

[ invariant holds ? raise exception for violated invariant ];
[ postcondition holds ? raise exception for violated postondition ];
[ return value valid ? raise exception for invalid result value];

RETURN aValue;
END
```

### 4.4.3 Realizacija

Vienas iš pateiktosios klaidų aptikimo idėjos realizavimo būdų būtų sukurti tam skirtą klasę (kad ir dėl to, jog naudojama programavimo kalba ir pats kontekstas yra objektiškai orientuoti). Tokios klasės metodų pirminė paskirtis būtų patikrinti, ar duota loginė išraiška yra teisinga („*true*“) ar klaidinga („*false*“). Taip pat gali būti ir realizuota galimybė atlikti algoritminį kokios nors situacijos įvertinimą (pavyzdžiui, įvertinti, ar visos klasės, iš kurių reikės surinkti kokius nors duomenis, galės juos suteikti ir ar jie bus korektiški ir pan.). Bet kuriuo atveju, jei įvertinimas bus neigiamas, tuomet klaidų aptikimo metodas sukels klaidos situaciją ir turimus duomenis surašys atitinkamą duomenų registro failą („*error log*“). Taipogi iškart galima numatyti ir galimybę analizuojamoje sistemoje turėti ne vieną klaidų aptikimui skirtą klasę, o kelias, nes iškart visiems metodams tinkančių sąlygų rinkinio specifikuoti negalima. Tam tikslui galima naudoti bazinę ir išvestines klases. Be to, kadangi klaidų aptikimo paslaugomis naudosis tikrai ne viena sistemos klasė, tai iškart galima galvoti apie projektavimo šabloną *Singleton*. Struktūriškai tai vaizduoja 14 paveikslėlyje pateiktas fragmentas.




**14 pav.** Bazinės klaidos aptikimo klasės ir susijusių elementų struktūrinis pavaizdavimas.

Pateiktame paveikslėlyje klasė *ErrorDetector* šiuo atveju turi 5 metodus, kuriuos gali perrašyti išvestinės klasės (jei tokių bus). Visuose metoduose, išskyrus *checkResult*, yra

naudojami tie patys parametrai, iš kurių pirmasis klaidos atveju leidžia pateikti tekstinį aprašą, o antrasis – nurodyti loginę išraišką, kurią reikia įvertinti. Siekiant paprastumo, situacijų algoritminio įvertinimo metodų klasėje *ErrorDetector* nėra nurodyta, tačiau juos bet kada galima prijungti, atsiradus tokiems poreikiams. Metodas *checkResult* turi 3 parametrus, iš kurių pirmas reiškia tikėtiną reikšmę, antras – vertinamą reikšmę, o trečias – tekstinį aprašą klaidos atveju. Klasė taip pat turi globalų *boolean* tipo kintamąjį *errorMode*, kurio reikšmę klaidos atveju nustato *true*. Tai tiesiog tam tikras žymeklis („*flag*“), kuris nurodo, kad paskutinio tikrinimo metu buvo arba nebuvo gauta klaida.

Kiekvienoje klaidų valdymo posistemės klientinėje klasėje yra numatyti du metodai: *isValid* ir *isAvailable*, iš kurių pirmasis turi būti realizuotas visuose klientinėse klasėse, o antrasis tik tose klasėse, kur jo reikia pagal paskirtį. Abu šie metodai yra kaip priemonės, leidžiančios objektams pasitikrinti save patiems. Pirmasis metodas yra orientuotas į objekto saugomų duomenų korektiškumo patikrinimą (t.y. ar jie neprieštaringi, ar leistinos jų reikšmės ir pan.), o antrasis – į pačių objektų ir jų teikiamų paslaugų (ypač, jei yra bendraujama, pavyzdžiui, per tinklą ar pan.) prieinamumą. Pastarojo pavyzdys galėtų būti situacija, kuomet vienas objektas kreipiasi į kitą objektą gauti duomenų arba leidimą atlikti kokias nors operacijas, o šis dar nėra pasiruošęs jų suteikti (pavyzdžiui, dėl klaidos arba per didelės atliekamų užduočių apkrovos). Tokiu atveju yra įvertinama situacija ir nustatoma, ar gautas rezultatas turi būti traktuojamas kaip klaida, ar ne. Be abejo, reikalui esant, šių metodų variantų gali būti net ir ne vienas.

Kol kas analizuojamoje sistemoje bent preliminariai pakanka numatyti tik vieną pagrindinę klaidų aptikimui skirtą klasę *ErrorDetector*, nenaudojant jokių papildomų išvestinių jos klasių. O patį ryšį tarp klasių *ErrorDetector* ir *Client* galima iliustruoti konkrečiu analizuojamosios sistemos kodo fragmentu, palyginant originalią rašytą jo versiją su naująja, papildyta klasės *ErrorDetector* panaudojimu klasėje *Client* (žr. 2, 3 pavyzdžius). Visi skirtumai, išskyrus matomumą ir tipą žyminčius bazinius *Java* kalbos žodžius, bus akcentuojami pajuodintu („*bold*“) šriftu.

**2 pavyzdys.** Originali klasės *DepositTransaction* metodo *finishApprovedTransaction* kodo versija.

```
…
protected Session _session;
protected ATM     _atm;
protected Bank    _bank;
protected Money   _newBalance;
protected Money   _availableBalance;
protected Money   _amount;
protected Bank    _bank;
protected int     _serialNumber;
```

```
protected int     _toAccount;
…

public int finishApprovedTransaction()
{
    boolean envelopeAccepted = _atm.acceptEnvelope();
    _bank.finishDeposit(_atm.number(), _serialNumber, envelopeAccepted);

    if (envelopeAccepted)
    {
        _atm.issueReceipt(_session.cardNumber(),
                          _serialNumber,
                          "DEPOSIT TO " + _bank.accountName(_toAccount),
                          _amount,
                          _newBalance,
                          _availableBalance);
        return Status.SUCCESS;
    }
    else
        return Status.ENVELOPE_DEPOSIT_TIMED_OUT;
}
```

**3 pavyzdys.** Klaidų aptikimo taikymo iliustravimas metodo *finishApprovedTransaction* kontekste.

```
…
protected Session _session;
protected ATM     _atm;
protected Bank    _bank;
protected Money   _newBalance;
protected Money   _availableBalance;
protected Money   _amount;
protected Bank    _bank;
protected int     _serialNumber;
protected int     _toAccount;
…

public int finishApprovedTransaction()
{
    // create new instance of ErrorDetector class
    ErrorDetector errorDetector = ErrorDetector::instance();

    // method has no invariants
    // ---

    String myInfo = “|”+ this.toString() +“|”+
        “public int finishApprovedTransaction()” +“|”;

    errorDetector.switchOnNotification();

    // check to see if parameters are not empty, if they are available (e.g.
    // reachable and not in an error state) and can provide us information

    errorDetector.checkPrecondition (“_atm parameter is empty” + myInfo,
        (_atm == 0));

    errorDetector.checkPrecondition (“_bank parameter is empty” + myInfo,
        (_bank == 0));

    errorDetector.checkPrecondition (“_session parameter is empty” + myInfo,
        (_session == 0));
```

```
errorDetector.checkPrecondition ("_atm parameter is invalid" + myInfo,
    _atm.isValid ());

errorDetector.checkPrecondition ("_bank parameter is invalid" + myInfo,
    _bank.isValid ());

errorDetector.checkPrecondition ("_session parameter is invalid" +
    myInfo, _session.isValid ());

errorDetector.checkPrecondition ("_newBalance parameter is invalid" +
    myInfo, _newBalance.isValid ());

errorDetector.checkPrecondition ("_availableBalance parameter is
    invalid" + myInfo, _availableBalance.isValid ());

errorDetector.checkPrecondition ("_amount parameter is invalid" + myInfo,
    _amount.isValid ());

errorDetector.checkPrecondition ("_serialNumber parameter is invalid" +
    myInfo, _bank.isSerialNumberValid (_serialNumber));

errorDetector.checkPrecondition ("_toAccount parameter is invalid" +
    myInfo, _atm.isAccountValid (_toAccount));

errorDetector.checkPrecondition ("_atm component is unavailable" +
    myInfo, _atm.isAvailable ());

errorDetector.checkPrecondition ("_bank component is unavailable" +
    myInfo, _bank.isAvailable ());

errorDetector.check (myInfo, errorDetector.errorMode) ?
    return Status.FAILURE;

boolean envelopeAccepted = _atm.acceptEnvelope();

myInfo = "|"+ this.toString() +"|"+
    "public int finishApprovedTransaction()" +"|";

errorDetector.check (myInfo, errorDetector.errorMode) ?
    return Status.FAILURE;

_bank.finishDeposit(_atm.number(), _serialNumber, envelopeAccepted);

myInfo = "|"+ this.toString() +"|"+
    "public int finishApprovedTransaction()" +"|";

errorDetector.check (myInfo, errorDetector.errorMode) ?
    return Status.FAILURE;

if (envelopeAccepted)
{
    _atm.issueReceipt(_session.cardNumber(),
                      _serialNumber,
                      "DEPOSIT TO " + _bank.accountName(_toAccount),
                      _amount,
                      _newBalance,
                      _availableBalance);

    myInfo = "|"+ this.toString() +"|"+
        "public int finishApprovedTransaction()" +"|";

    errorDetector.check (myInfo, errorDetector.errorMode) ?
        return Status.FAILURE : return Status.SUCCESS;
}
```

```
    else
        return Status.ENVELOPE_DEPOSIT_TIMED_OUT;


    // method has no postconditions
    // ---
}
```

Iš pavyzdžio matyti, kad naujajame metode nėra jokių invariantų ir „*post*-“ sąlygų, nes visi pagrindiniai įvertinimai susiveda į tikrinimą „*pre*-“ sąlygų, kurios turi užtikrinti duomenų korektiškumą, prieš panaudojant juos tolimesnėse operacijose, tuo pačiu užkertant kelią klaidos nepageidaujama pasklidimui. Ketvirtajame pavyzdyje yra kaip tik panaudoti 2 klasei *Bank* priklausantys metodo *isValid* variantai, t.y. *isSerialNumberValid* ir *isAccountValid*, kurie patikrina vartotojo banko kortelės serijinio numerio ir sąskaitos numerio teisingumą.

### 4.4.4 Galimos pasekmės

Galimas pasekmes suformuluosiu kaip iki šiol 3.3 poskyryje pateiktų sprendimų apibendrinamuosius privalumus ir trūkumus (žr. 8 lentelę).

**8 lentelė.** 3.3 poskyryje priimtų sprendimų privalumų ir trūkumų aprašymas.

| Nr. | Privalumų / trūkumų aprašymas |
|---|---|
| **Privalumai** | |
| 1. | Pateiktos klaidų aptikimo idėjos veiksmingumas aptikti klaidas kiek įmanoma anksčiau (nei jos spėja išplisti ir pridaryti daug žalos) labai priklauso nuo parašytų metodų dydžio. Kuo metodo dydis mažesnis, tuo klaidų tikrinimo dažnumas yra didesnis ir jis yra arčiau tikrojo galimo klaidos židinio. Šiaip metodo dydis dažniausiai priklauso nuo naudojamo programavimo stiliaus ir kalbos. Kadangi objektiškai orientuotose programavimo kalbose metodai, kaip praktika rodo, nebūna labai dideli, tai šiuo aspektu mūsų atveju tokia idėja pasiteisina kaip privalumas (nes naudojama programavimo kalba yra *Java*). |
| 2. | Klaidų aptikimui atlikti specifikacijos su „*pre*-“ ir „*post*-“ sąlygomis bei invariantais panaudojimas yra palyginti paprastas, nesudėtingas ir net pačiam kode informatyvus būdas išreikštinai apibrėžti, ko galima tikėtis ir kokios gali kilti bėdos. |
| 3. | Įtraukiant į klaidos aptikimo klasę keletą skirtingų klaidų tikrinimo būdų, padaro ją pakankamai lanksčią, nes praplečia jos galimybes, kaip ji galėtų elgtis įvairiomis kylančių klaidų situacijomis. |
| **Trūkumai** | |

| | |
|---|---|
| 1. | Kadangi klaidų aptikiko idėja remiasi prielaida, kad metodams sudaroma jų specifikacija yra teisinga, tai akivaizdžių problemų gali kilti, jei ji nebus kokybiškai ir išbaigtai parengta. Be to, tokia idėja yra netinkama klaidų aptikimą vykdyti projektavimo stadijoje. Dar dvi problemos gali kilti ir tuomet, jei šią idėja bus bandoma taikyti ciklinio programos vykdymo atveju ir jei patys klaidas aptinkantys programiniai vienetai bus realizuoti neteisingai. Taigi iki pilno ir veiksmingo idėjos įgyvendinimo yra nemažai rimtų rizikų. |
| 2. | Kadangi klaidų aptikimo idėjos esmė yra tam dedikuotu kodu praturtinti kiekvieno sistemos komponento (arba bent jau didžiosios jų dalies) kiekvieną metodą, tai akivaizdi tokios realizacijos galima pasekmė yra kažkiek sumažėjęs sistemos darbo našumas („*performance*“), kurį realiai pakelti turbūt būtų galima tik pašalinus patį klaidas aptinkantį kodą. |
| 3. | Visa klaidų aptikimo klasės realizacija ne tik apkrauna sistemos komponentų kodą, bet ir padidina komunikacinių ryšių ir netiesioginių sąsajų tarp objektų skaičių, nes klaidos tikrinimo metu turi būti iškviečiamas atitinkamas metodas, priklausantis kažkuriai tai bazinės klasės *ErrorDetector* išvestinei klasei. |

## 4.5 KLAIDŲ REGISTRAVIMAS

### 4.5.1 Problema

Nagrinėjant šią probleminę sritį, galima pastebėti, kad paprastai standartiniai programavimo įrankiai nesuteikia efektyvių, naudingų ar net apskritai jokių klaidų registravimo įrankių, tinkamų konkrečiam programų sistemos kūrimo atvejui. Dėl to tokį funkcionalumą reikia suprojektuoti ir sukurti patiems. O čia jau galima pradėti formuluoti ir aktualius su šia problemine sritimi susijusius klausimus: kaip surinkti klaidų analizei ir trasavimui svarbią informaciją (ypač, jei priėjimas prie programinio vykdymo steko (*„execution stack*“) yra ribotas)? Kokia yra ta informacija? Kaip centralizuoti klaidų valdymą ir kam to gali reikėti?

### 4.5.2 Sprendimas

Pirmiausia turbūt reikėtų pradėti nuo to, kad klaidų trasavimui reikalingo kodo gavimas turėtų būti kuriamas ne rankiniu būdu, o bent jau dalinai sugeneruojamas. Taip būtų taupomas laikas ir kiti resursai. Vadinasi, galima kalbėti apie kažkokios schematinės sprendimo idėjos konstravimą.

Jau žinome, kad klaidų trasavimui yra reikalingas koks nors programinis vykdymo stekas. Standartinis sistemos programinis vykdymo stekas gali būti ir neprieinamas arba steko

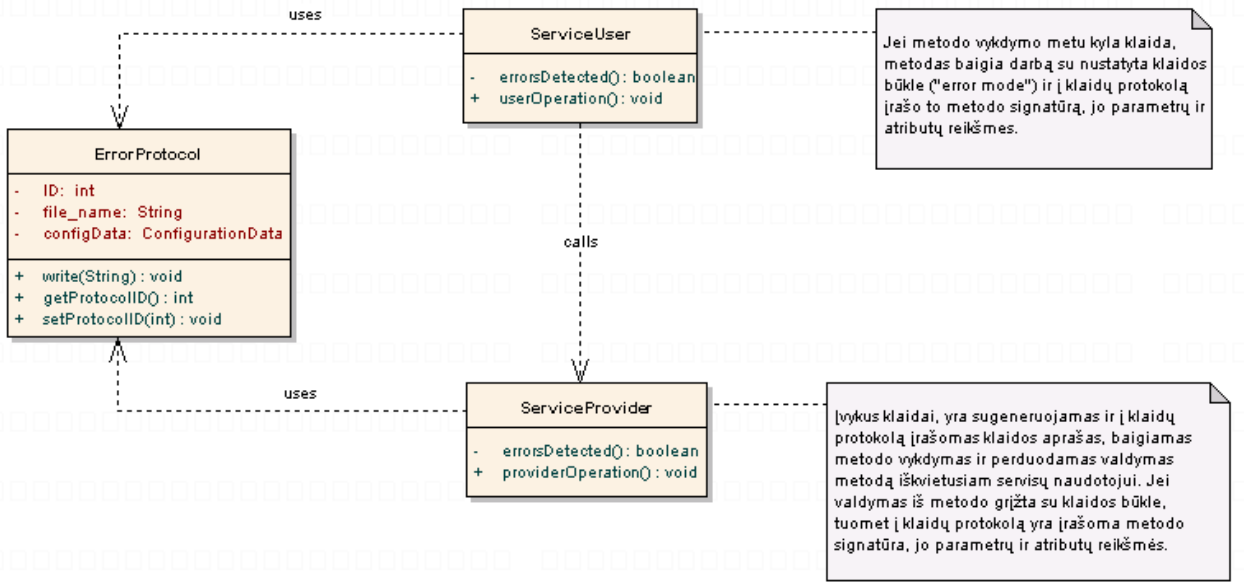
informaciją nuskaityti, atfiltruoti ir struktūrizuoti pagal savo poreikius gali būti per sudėtinga (pavyzdžiui, jei sistemoje egzistuoja tik žemo lygio standartinis mikroprocesorine programavimo kalba grįstas steko veikimas). Be abejo, analizuojamosios sistemos atžvilgiu visas kodas yra parašytas *Java* programavimo kalba, o tai reiškia, kad be daugumos standartinių išvestinių klaidų klasių joje yra ir bazinė bet kokiai klaidai apibrėžti naudinga klasė *Throwable*, kuri savyje jau turi dalį reikalingo funkcionalumo (kaip su programiniu vykdymo steku susijusios informacijos išsaugojimas ir pan.). Vis dėlto, siekiant parodyti, kad ir be standartinių priemonių ši klaidų valdymo modelio probleminė sritis yra realizuojama, galima pabandyti susikurti kokį nors pageidaujamą programinį vykdymo steką ir patiems.

Visa objektiškai orientuota analizuojamosios sistemos architektūra veikia tokiu principu, kad vienos klasės metodas kviečia kitos klasės metodus, ši dar kitos ir t.t. Metodas taip pat gali kviesti ir savo pačios klasės metodus. Taigi pagal tokį principą galima išskirti dvi klasių roles: servisų naudotojas („*ServiceUser*“) ir servisų tiekėjas („*ServiceProvider*“). Bet kuri klaidų valdymo posistemės klasė gali naudotis klaidų protokolo („*ErrorProtocol*“) klasės paslaugomis, kad klaidos atveju į atitinkamą duomenų registro failą galėtų įrašyti svarbią informaciją. Jos dalis, susijusi su klaidų trasavimui reikalinga informacija, gali būti gauta iš metodą sudarančio kodo. Kitaip tariant, klasių metodai, kviesdami kitų klasių metodus, sudaro tam tikrą metodų iškvietimų seką („*chain of calls*“), kurio kopiją įrašyti klaidų protokolą yra pagrindinis klaidų trasavimo kodo uždavinys.

Kitas ne ką mažiau svarbus aspektas yra centralizuotas klaidų valdymas. Kam jo reikia? Ogi tam, kad turint kokią nors didelę išskirstytą sistemą (kad ir tokią kaip analizuojamosios sistemos atveju, t.y. banko sistemą kaip centrinę valdymo stotį ir *ATM* bankomatus kaip nepriklausomus paslaugų terminalus), būtų galima nuotoliniu būdu atlikti bet kurio klaidos ištikto sistemos segmento klaidų analizę, įvertinti būklę ir, jei galima, reaktyvuoti visą darbą. Taigi vienas iš paprastesnių, bet nebūtinai efektyviausių, būdų užtikrinti centralizuotą klaidų valdymą analizuojamoje sistemoje būtų tiesiog turėti joje tam tikrą kodo dalį, skirtą kaupti visiems sistemoje sudaromiems klaidų protokolams ir siųsti juos centrinei banko sistemai. Tam tikslui yra reikalingos 2 klasės: „*CentralLogManager*“ ir „*CentralLog*“. Pirmoji klasė yra atsakinga už klaidų protokolų kaupimą, perdavimą per tinklą ir turi būti realizuota *ATM* bankomato sistemoje. Antroji yra atsakinga už klaidų protokolų priėmimą per tinklą iš ir turi būti realizuota centrinėje banko sistemoje.

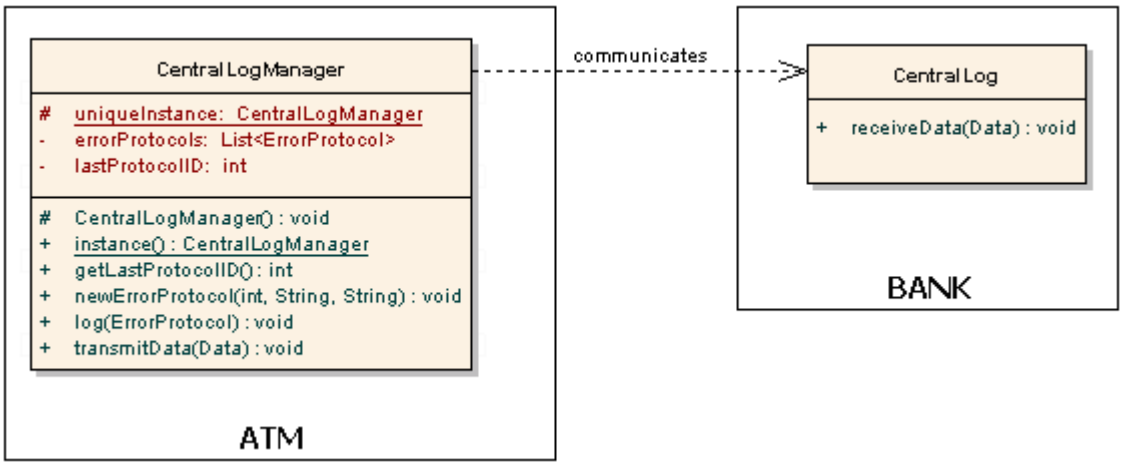
### 4.5.3 Realizacija

Struktūrinis klaidų trasavimo veikimo principas yra pavaizduotas 15 paveikslėlyje.




**15 pav.** Klaidų trasavimo principo struktūrinis pavaizdavimas.

Struktūrinis centralizuoto klaidų valdymo veikimo principas yra pavaizduotas 16 paveikslėlyje.



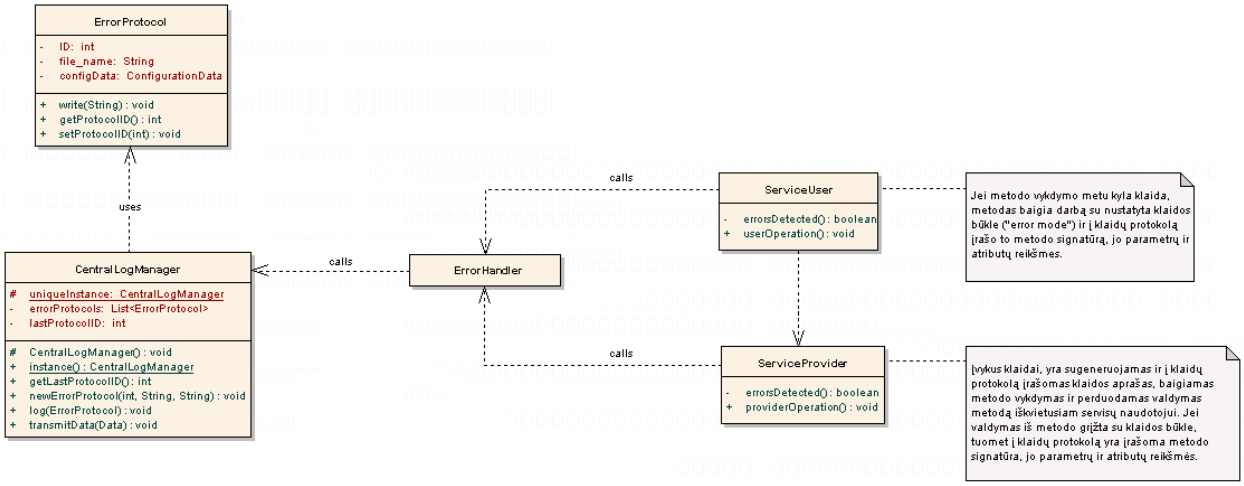

**16 pav.** Centralizuoto klaidų valdymo struktūrinis pavaizdavimas.

Tam, kad centrinė banko sistema gautų tik svarbių ir ją dominančių klaidų protokolus (o ne visų, įskaitant ir smulkių bei tų klaidų, su kuriomis lengvai gali susitvarkyti ir lokalios pačių bankomatų automatinės klaidų šalinimo priemonės), galima į kokį nors (pavyzdžiui, „*.ini*“) failą surašyti tam dedikuotą konfigūraciją. Tokiu atveju tada papildomai reikės atlikti 3 dalykus:

apibrėžti bendrą konfigūracijos struktūrą, užtikrinti jos palaikymo sistemoje priemones ir tarpusavyje suderinti veikimo pagal nustatytą konfigūraciją tvarką, o metodas *transmitData* – klaidų protokoluose esančios informacijos persiuntimui centrinei banko sistemai (šiuo atveju tiesiog klasei *CentralLog*, kuri tokių duomenų priėmimui turi atitinkamą metodą *receiveData*).

Klasė *CentralLogManager* darbui su klaidų protokolais turi 2 atributus, t.y. klaidų protokolų sąrašą (*errorProtocols*) ir paskutinio naujausio sąraše sukurto protokolo identifikatoriaus reikšmę (*lastProtocolID*), kurią grąžina metodas *getLastProtocolID*. Metodas *newProtocolID* yra skirtas naujo protokolo sukūrimui, nurodant jo identifikatorių, failo, į kurį bus įrašoma protokole esanti informacija, vardą ir protokolo duomenis (pavyzdžiui, klaidų trasavimo atveju yra rašoma metodo signatūra, parametrų ir atributų reikšmės ir pan.). Metodas *log* yra skirtas jau sukurto klaidų protokolo patalpinimui į sąrašą.

Taigi turint klaidų trasavimo ir centralizuoto klaidų valdymo idėjas, galima sudaryti ir bendrą struktūrinę schemą, apjungiančią šiuos du aspektus į vieningą klasių junginį (žr. 17 paveikslėlį).




**17 pav.** Galutinio į klaidų registravimą orientuoto struktūrinio sąryšio pavaizdavimas.

Tai, kas ir kaip galėtų būti daroma metode *checkPrecondition*, kaip vienas iš galimų variantų, yra pateikiama kitame kodo fragmente (žr. 4 pavyzdį).

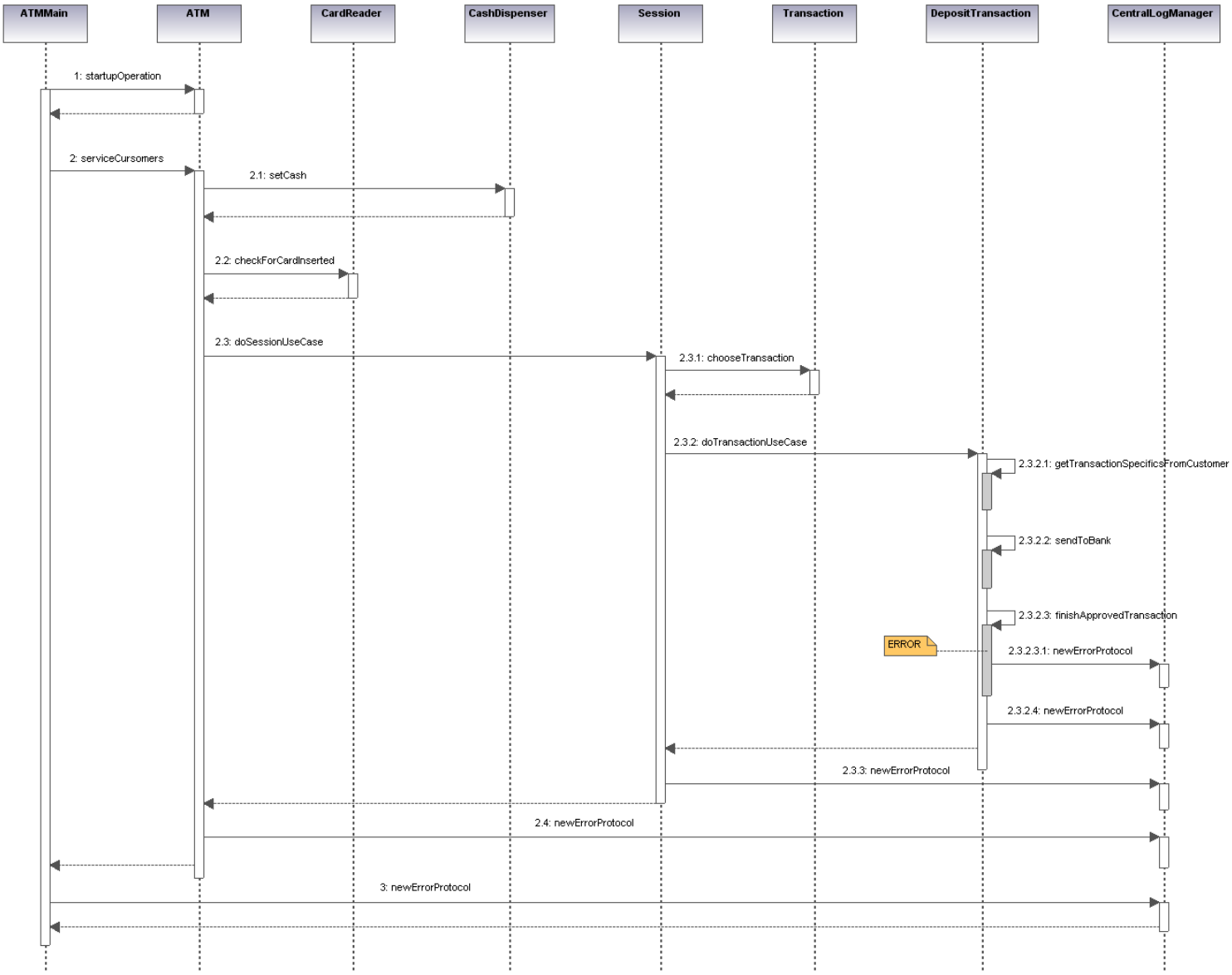
**4 pavyzdys.** Bendrinė klasės *ErrorDetector* kodo realizacija.

```
class ErrorDetector
{
    protected static ErrorDetector uniqueInstance = new ErrorDetector();
    public boolean errorMode = false;
    private boolean onNotification = true;
```

```
    private GenericError lastDetectedError;

    protected ErrorDetector() { uniqueInstance = this; }
    public static ErrorDetector instance() { return instance; }

    public switchOnNotification() { onNotification = true; }
    public switchOffNotification() { onNotification = false; }

    public void checkPrecondition (String descr, boolean condition)
    {
        if (!condition)
        {
            lastDetectedError = new ErrViolatedCondition (descr);
            throw lastDetectedError;
        }
        else if (errorMode && onNotification)
        {
            throw new GenericError (“Previous error state not
                eliminated” +“|”+ this.toString());
        }
    }
}

class ErrViolatedCondition extends GenericError
{
    // create new instance of ErrorHandler class
    ErrorHandler errorHandler = ErrorHandler::instance();

    public ErrViolatedCondition() { super(); }

    public ErrViolatedCondition (String s)
    {
        super(s);
        int epID = errorHandler.centralLogManager.getLastProtocolID()++;
        errorHandler.centralLogManager.newErrorProtocol (epID,
            “_ERRLOG” + epID + “.txt”, s);
    }
}
```

Iš pavyzdžio matyti, kad jame yra laikomasi minėtų klaidų registravimo principų. Šiame pavyzdyje analizuojamosios sistemos klasė *Status* yra papildyta viena nauja skaitine konstanta *FAILURE*, kuri reiškia, kad metodo vykdymo metu kilo klaida. Šiame kontekste centrinė banko sistema nėra ir neturi būti informuojama, nes tai yra klaidų tvarkyklės *ErrorHandler* atsakomybė, o pavyzdyje naudojamas klaidų registravimo funkcionalumas yra skirtas klaidų trasavimui atlikti.

Remiantis šiuo pavyzdžiu, pateiksiu sekų diagramą, kuri pavaizduos veiksmų eiliškumą nuo pat *ATM* bankomato sistemos startavimo pradžios iki momento, kuomet šio metodo vykdymo metu kils klaida. Nuo tada tolimesni veiksmai atspindės klaidų trasavimo eigą (žr. 18 paveikslėlį).




**18 pav.** Klaidų registravimą iliustruojanti sekų diagrama.

Taigi tiek iš paveikslėlio, tiek ir iš pateiktų kodo fragmentų matyti, kad net jei ir kurios nors klasės metodas būtų vykdomas labai ilgą laiko tarpą, vis tiek aprašytasis arba kuris nors standartinis klaidų aptikimo būdas (pavyzdžiui, *try-catch* blokas kaip papildoma saugumo priemonė šalia klaidų aptikimo klasės teikiamo funkcionalumo) galėtų sėkmingai suveikti, nes visi metodai yra atitinkamai tarpusavyje „surišti“ pagal 15 paveikslėlyje pateiktą schemą.

### 4.5.4 Galimos pasekmės

Galimas pasekmes suformuluosiu kaip iki šiol 3.4 poskyryje pateiktų sprendimų apibendrinamuosius privalumus ir trūkumus (žr. 9 lentelę).

**9 lentelė.** 3.4 poskyryje priimtų sprendimų privalumų ir trūkumų aprašymas.

| Nr. | Privalumų / trūkumų aprašymas |
|---|---|
| **Privalumai** | |
| 1. | Nuosavo klaidų trasavimui reikalingo programinio vykdymo steko konstravimas suteikia daugiau lankstumo su informacija susijusio valdymo atžvilgiu, o tai yra gana dinamiška, nes bet kuriame sistemos veikimo žingsnyje (priklausomai nuo to, kokio kritiškumo ir svarbos kuri dalis yra vykdoma), visada galime manipuliuoti su tokiu informacijos kiekiu, koks tame žingsnyje yra reikalingas. |
| 2. | Centralizuoto klaidų valdymo idėja leidžia atlikti tam tikrą sugeneruotų klaidų protokolų filtravimą (pavyzdžiui, skirstyti juos į tam tikras grupes pagal kritiškumą, būtinumą apie tai pranešti ir pan.), o tai gali būti naudinga ne tik klaidų analizei, bet ir pačiam komunikavimui su centrine banko sistema, nes filtravimo dėka komunikacijos operacijų skaičius gali minimaliai ar net ženkliai sumažėti. |
| 3. | Centralizuotas klaidų valdymas, teisingai suprojektavus, gali pasitarnauti ir tuomet, jei dėl kokios nors klaidos sutrinka ir pats bendras klaidų valdymo mechanizmas (bet, pavyzdžiui, ne komunikacijos ryšio kanalas). Kadangi centralizuotą klaidų valdymą pačiu paprasčiausiu atveju sudaro tik vienas komponentas, tai jis teisingai įvertinęs sistemos būklę gali būti pats vienas pajėgus atitinkamai informuoti centrinę banko sistemą. |
| **Trūkumai** | |
| 1. | Konstruojant nuosavą klaidų trasavimui skirtą programinį vykdymo steką, atsiranda papildomų jo palaikymo ir administravimo darbų. Tai ne tik didina sistemos komponentų realizacijos kodą, bet ir komplikuoja komunikacijos ryšius (pavyzdžiui, klaidos atveju šalia įprastinio judėjimo pirmyn ir gilyn ryšio atsiranda dar grįžtamasis ryšis ir pan.). |
| 2. | Naudojant aprašytąją centralizuoto klaidų valdymo idėją, atsiranda papildomų komunikacijos ryšių, o tai ne tik kažkiek apkrauna, bet ir sulėtina sistemos veikimą. Be to, sutrikus bendram komunikacijos kanalui, centrinė banko sistema apie kilusias klaidas gali likti taip ir neinformuota. |
| 3. | Centralizuoto klaidų valdymo idėja reikalauja papildomų atminties resursų bei tuo pačiu atlieka ir šiokį tokį surinktų duomenų dubliavimą, nes sugeneruoti klaidų protokolai yra papildomai kopijuojami dar ir į atskirą vietą, skirtą išsiuntimui centrinei banko sistemai. |

## 4.6 KLAIDŲ APDOROJIMO STRATEGIJŲ PARINKIMAS

### 4.6.1 Problema

Vienas didžiausių sunkumų, su kuriuo tenka susidurti, projektuojant programų sistemas, yra iš anksto numatyti kiek įmanoma daugiau galimų klaidos situacijų kiekviename sistemos

komponente. Svarbu yra, kad sistema gebėtų susitvarkyti ne tik su klaidomis, kurios yra numatytos, bet su tomis, kurių dar niekas nenumatė. Bendru atveju tam galima dedikuoti dviejų rūšių klaidų apdorojimo programinius mechanizmus: standartinį („*default*“) ir specializuotą („*domain oriented*“). Pastarasis atitinkamą jų apdorojimo būdą leidžia parinkti kiekvienai planuotai sistemos klaidai ar jų grupei, o antrasis suteikia galimybę visoms nenumatytoms klaidoms pritaikyti kažkokį bendrinį jų apdorojimo (standartinį sistemos reagavimo į jas) būdą. Su visa šia problemine sritimi galima kelti daugybę įvairių klausimų, tačiau pagrindiniai jų galėtų būti tokie: koks galėtų būti standartinis klaidų apdorojimo su minimalia žala sistemai būdas? Kaip vartotojams pranešti apie susidariusią klaidos situaciją? Kaip išvengti einamuoju momentu atliekamos klaidos sutrikdytos užduoties visiško perkrovimo („*restart*“)? Kurioje vietoje ir kaip klaidų apdorojimas turėtų vykti? Kaip užtikrinti, kad klaidų apdorojimas būtų prieinamas net ir tada, kai yra resursų stygius arba apribojimas?

### 4.6.2 Sprendimas

Standartinis klaidų apdorojimo būdas turėtų būti suprojektuotas taip, kad nereikėtų panašaus funkcionalumo kodo rašyti kiekvienam sistemos komponentų metodui atskirai, tačiau tuo pačiu turėti galimybę, reikalui esant, ir pasinaudoti specializuotu klaidų apdorojimo būdu, t.y. kad jis būtų pasiekiamas. Be to, taip pat gerai būtų, jei sistemos komponentų funkcionalumo kodo pakeitimai neįtakotų klaidų apdorojimo būdų pritaikymo taisykles, t.y. nekeistų architektūrinių su klaidų apdorojimu susijusių reikalavimų.

Taigi panašiai kaip ir *switch* sakiniuose esančio *default* lauko atveju standartinio klaidų apdorojimo būdo panaudojimo principas galėtų būti analogiškas. Su juo taip pat galima sieti ir kokius nors įprastus (trivialius) klaidų aptikimo, registravimo ir perdavimo būdus, o naudoti tiesiog visuose sistemos komponentų metoduose, kuriuose gali kilti klaidos. Kitaip tariant, galima standartinio klaidų apdorojimo veikimo schema tiesiog galėtų būti tokia:

- ***if*** *any (other) error* ***then***
- *record error*
- *signal unexpected error to caller*
- ***endif***

Norint vartotojui pranešti apie klaidos situaciją (ypač, jei automatinis jos pašalinimas yra negalimas), reikėtų pamąstyti ir apie kriterijus, kurie čia gali būti svarbūs. Pirmiausia klaidų pranešimų turinys ir grafinis dialogo lango, kuriame jis yra pateikiamas, apipavidalinimas turėtų

būti atsietas (iškeltas kitur) nuo kodo konteksto, kuriame jis yra naudojamas. Tas pats galioja ir pačiai klaidų valdymo posistemei bei jos klientams. Klaidų pranešimai turi būti tarpusavyje suderinti ir atitikti vartotojo sąsajos projektavimo principus (tokius kaip aiškumas, suprantamumas, sklandumas, informatyvumas ir pan.). Be to, taip pat reikėtų atkreipti dėmesį ir į tokius aspektus, kaip diferencijavimas tarp klaidų, apie kurias vartotojui reikia pranešti iš karto, ir klaidų, kurios gali būti iš pradžių surinktos ir tik po kiek laiko pateiktos atitinkamoje suvestinėje.

Siekiant išvengti atliekamos klaidos sutrikdytos užduoties visiško perkrovimo, galima naudoti vadinamąją pakopinio perkrovimo (arba pakartotinio darbo paleidimo nuo atitinkamo kontrolinio taško) strategiją („*checkpoint restart*“). Jos idėja yra paprasta – kas fiksuotą laiko periodą išsaugoti sistemos arba jos dalies būklę, kuri galės būti panaudojama pakartotiniam darbo nuo atitinkamo jo momento paleidimui įvykdyti. Tokiai strategijai paprastai yra reikalingas priėjimas prie kokios nors duomenų bazės. Patiems pakopinio perkrovimo proceso ypatumams (pavyzdžiui, kaip skaičiaus apdorojamų žingsnių per fiksuotą laiko periodą („*checkpoint rate*“) nustatymas) turi būti sudaryta galimybė juos konfigūruoti. Analizuojamosios sistemos atveju pakopiniam perkrovimui gali būti dedikuota atitinkama klasė *CheckpointRestart*, o jos atributai galėtų būti tokie:

- Pirminio kodo failo vardas („*source_ file_name*“)
- Klasės pavadinimas („*class_name*“)
- Metodo pavadinimas („*method_name*“)
- Kodo eilutės numeris („*line_number*“)
- Vartotojo vardas („*user_name*“)
- Apdorojamų žingsnių dažnis („*checkpoint_rate*“)
- Darbiniai duomenys („*activity_data*“)
- Būklė („*status*“)

Iš sąrašo matyti, kad pirmos keturios jo eilutės sutampa su klaidų lokalizavimą nusakančiais duomenimis. Be abejo, sąrašas gali būti ir ilgesnis bei daug sudėtingesnis (pavyzdžiui, darbiniai duomenys ir / arba būklė gali būti apibrėžiami kaip kokios nors duomenų struktūros ir pan.).

Na, o kalbant apie pačią klaidų tvarkyklę (t.t. klasę *ErrorHandler*), atsižvelgus į jos tikrąją paskirtį, galima atkreipti dėmesį į tai, kad jos teikiamas funkcionalumas pirmiausia turėtų būti

suderintas. O tai reiškia, kad jei šioje klasėje bus enkapsuliuoti skirtingi klaidų apdorojimo būdai, tai jie turi būti parinkti taip, kad vienas kitam neprieštarautų, nekeltų konfliktinių situacijų ir kritiniu momentu nepridarytų dar daugiau klaidų. Kitaip tariant, klasės teikiamas funkcionalumas turi būti tinkamai suderintas. Skirtingos naudojamos programavimo kalbos teikia savitas klaidų apdorojimui skirtas priemones (pavyzdžiui, vėlgi tokius sakinius kaip *try-catch* ar pan.), tačiau patį klaidų apdorojimą dažnai tenka atlikti patiems. O kartais, kai net klaidų apdorojimui skirtos priemonės nėra prieinamos ar apskritai realizuotos, tai tuo taip pat tenka pasirūpinti patiems. Patį klaidų apdorojimą geriausia yra realizuoti kiek įmanoma aukštesniame (pavyzdžiui, vartotojo sąsajos su sistema arba pagrindinio metodo *main*) lygmenyje, kad jis būtų prieinamas bet kam iš bet kur.

Žinant faktą, kad klaidų apdorojimui taip pat yra reikalingi resursai, dažnai yra susiduriama su keblia problema, kad klaidų apdorojimo neretai prireikia būtent tada, kai sutrinka resursų prieinamumas arba jų paskirstymas. Turėti nuosavus resursų valdymo būdus taip pat nelabai apsimoka, nes tada ne tik dubliuojasi su tuo susijęs funkcionalumas ir jam reikalingas kodas, bet ir pats klaidų apdorojimo darbas (nes atsiranda jau keli resursų valdymo mechanizmai). Išankstinis atminties ar kitų resursų rezervavimas gali sukelti bereikalingų operacijų (tokių, kaip *swapping* ir pan.) vykdymą, o tai gali lėtinti sistemos veikimą, apsunkinti klaidų analizę ir patį apdorojimą. Dėl to, norint išvengti įvairių galimų nepageidaujamų šalutinių poveikių, galima bandyti taikyti šiuos patarimus:

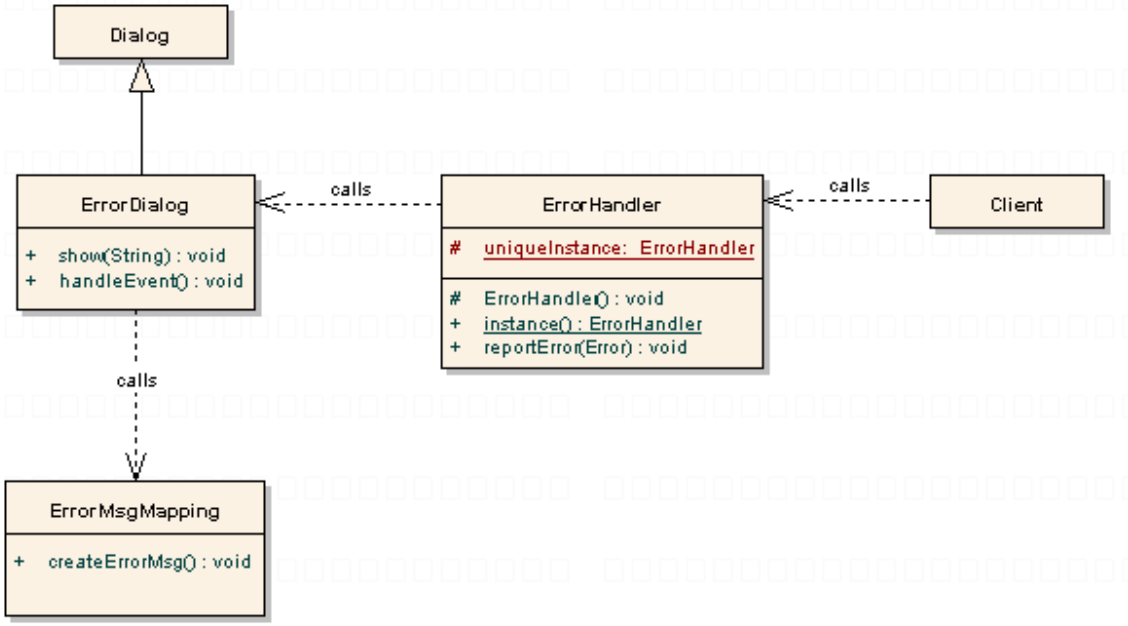
1. Kiek įmanoma labiau sumažinti dinaminės atminties išskyrimą.
2. Iš anksto numatyti, apibrėžti ir rezervuoti failus, dialogo langus ir kitus būtinas resursus (pavyzdžiui, saugoti kaip objektų kolekciją („*collection of preallocated objects*“)).
3. Darbo pradžioje išsiskirti rezervinę atminties dalį, kurią būtų galima naudoti, sumažėjus prieinamos atminties kiekiui.

Be abejo, tai yra tik keletas bendrinių aspektų, o atskiriems jų atvejams būtų galima sugalvoti ir daugiau, tačiau teisingas jų pritaikymas, kaip praktika rodo, išties leidžia išvengti nemalonių probleminių situacijų, susijusių su resursų valdymu. Analizuojamoje sistemoje pastarajam darbui gali būti sukurta speciali klasė *ResourcePreallocator*, kuri būtų tarsi centralizuotas priėjimas prie disponuojamų programinių resursų. Taigi pagrindinė jos paskirtis būtų administruoti rinkinį užrezervuotų resursų ir teikti atitinkamas paslaugas, orientuotas į resursų naudojimo teisės suteikimą.

### 4.6.3 Realizacija

Dirbant su tokiomis programavimo kalbomis kaip *C++* ar *Java*, klaidų gaudymui galima laisvai naudoti jų teikiamas standartines priemones – *try-catch* sakinius, kurie yra skirti bet kokių klaidų situacijų gaudymui. Tuo galima pasinaudoti ir, pavyzdžiui, klasėje *ErrorProtocol* turėti metodą *getErrorCopy()*, kuris leistų užfiksuoti ir užregistruoti kilusią klaidos situaciją. Pats metodas būtų naudojamas anksčiau paminėtų klasių *ServiceUser* ir *ServiceProvider*.

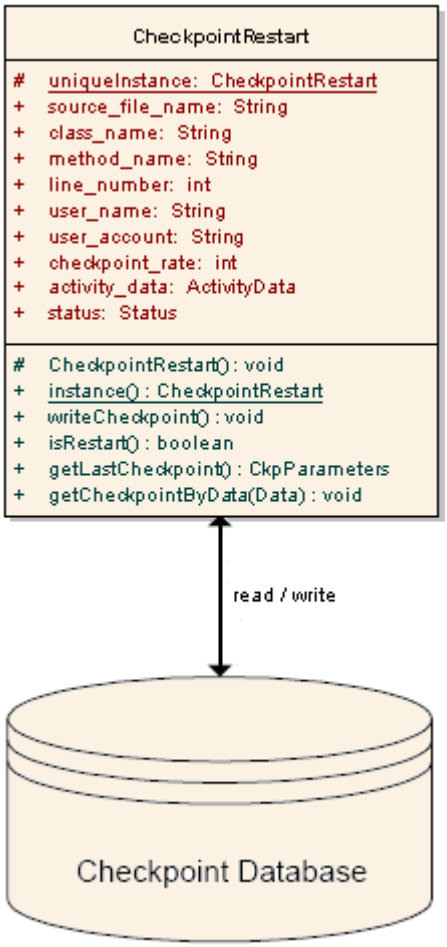
Kalbant apie klaidų pranešimų realizaciją analizuojamoje sistemoje, pagal turimą klaidų valdymo posistemės architektūrinę schemą galima panaudoti klasę *ErrorHandler*, kurioje gali būti metodas *reportError()*, skirtas pranešti vartotojui apie susidariusią klaidos situaciją (žr. 19 paveiksliuką).




**19 pav.** Klaidų pranešimų realizacijos struktūrinis pavaizdavimas.

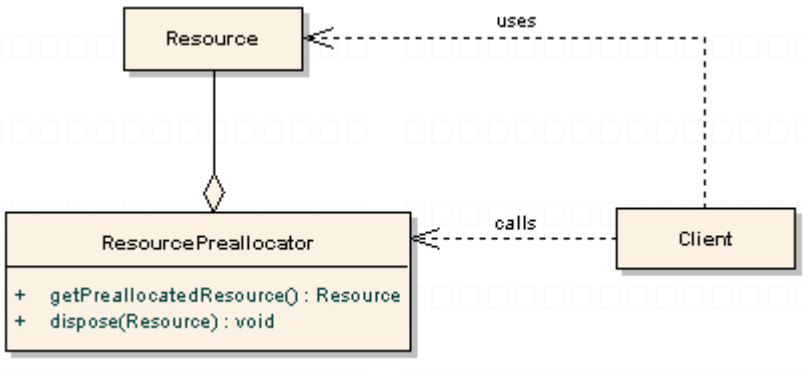
Tipiškai klasė *ErrorDialog* gali turėti visus klaidų pranešimų pateikimui vartotojui reikalingus duomenis (pavyzdžiui, dialogo lango dydis, spalva, antraštė, mygtukų aibė ir pan.). Jei kurioms nors klaidoms yra reikalinga turėti skirtingus jų vizualinio pateikimo būdus, tuomet klasė gali turėti ir keletą savo atmainų (t.y. išvestinių klasių).

Klasės, atsakingos už pakopinį darbo perkrovimą, struktūrinis vaizdas yra pateiktas 20 paveikslėlyje.



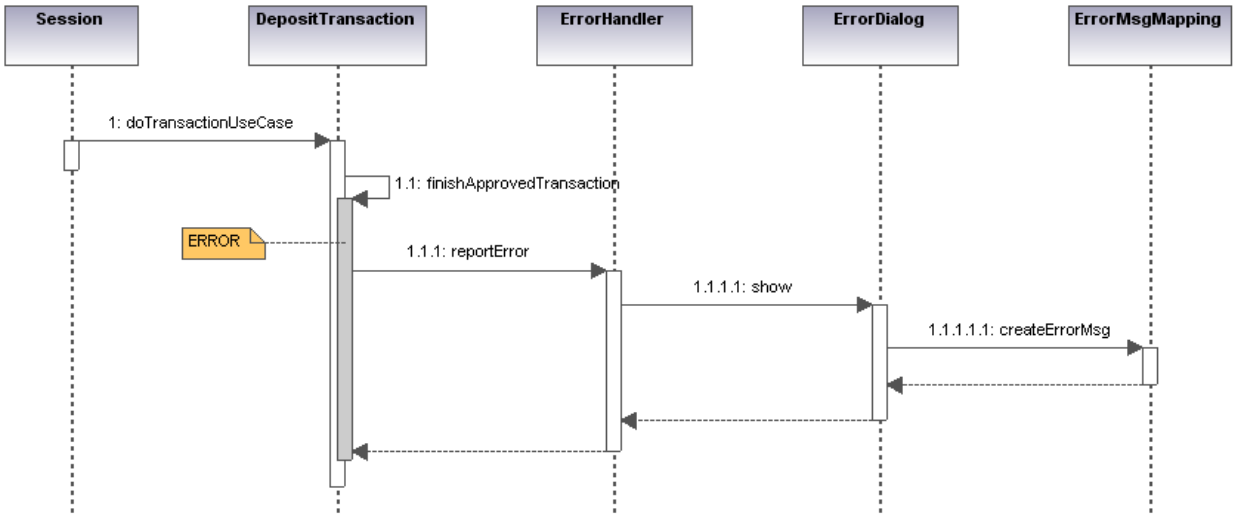

**20 pav.** Pakopinio perkrovimo klasės struktūrinis pavaizdavimas.

Efektyvesniam resursų analizuojamoje sistemoje valdymui galima sudaryti kad ir tokią nesudėtingą funkcionavimo schemą, kokia yra pateikta 21 paveikslėlyje.




**21 pav.** Resursų valdymo klasės struktūrinis pavaizdavimas.

Turint šias schemas, jau galima pailiustruoti ir konkretesnį jų panaudojimą pasirinkto klasės *DepositTransaction* metodo *finishApprovedTransaction* kontekste. 22 paveikslėlyje yra pateikiama klasės *ErrorDialog* panaudojimą iliustruojanti sekų diagrama.




**22 pav.** *ErrorDialog* klasės panaudojimą iliustruojanti sekų diagrama.

Klasės *CheckpointRestart* panaudojimą iliustruojantis kodo fragmentas yra pateiktas 5 pavyzdyje.

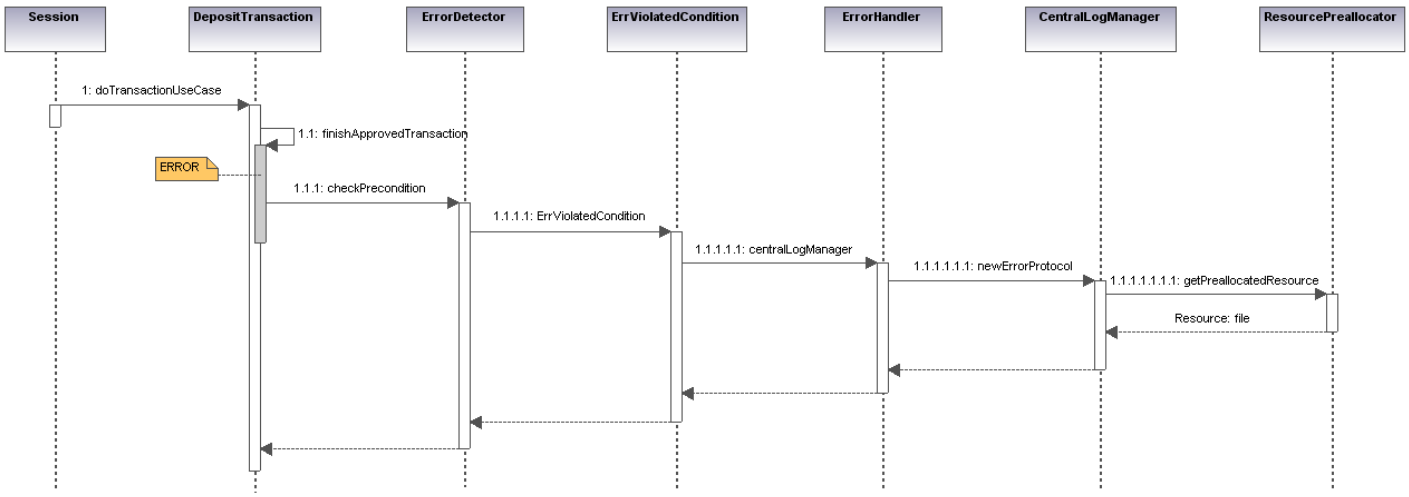
**5 pavyzdys.** Klasės *CheckpointRestart* panaudojimas.

```
…
protected Session _session;
protected ATM     _atm;
protected Bank    _bank;
protected Money   _newBalance;
protected Money   _availableBalance;
protected Money   _amount;
protected Bank    _bank;
protected int     _serialNumber;
protected int     _toAccount;
…

public int finishApprovedTransaction()
{
    // create new instance of ErrorDetector class
    ErrorDetector errorDetector = ErrorDetector::instance();

    // method has no invariants
    // ---

    String myInfo = “|”+ this.toString() +“|”+
        “public int finishApprovedTransaction()” +“|”;

    errorDetector.switchOnNotification();

    // check to see if parameters are not empty, if they are available (e.g.
    // reachable and not in an error state) and can provide us information

    //
    // …
    //
```

```
    CheckpointRestart ckpRestart = CheckpointRestart::instance();

    ckpRestart.ckpParameters.source_file_name  = __FILE__;
    ckpRestart.ckpParameters.class_name        = "DepositTransaction";
    ckpRestart.ckpParameters.method_name       = "finishApprovedTransaction";
    ckpRestart.ckpParameters.line_number       = __LINE__;
    ckpRestart.ckpParameters.user_name         = _bank.userName();
    ckpRestart.ckpParameters.user_account      = _bank.currentAccountName();
    ckpRestart.ckpParameters.activity_data     = getActivityData();
    ckpRestart.ckpParameters.status            = getStatus();

    ckpRestart.writeCheckpoint();

    _bank.finishDeposit(_atm.number(), _serialNumber, envelopeAccepted);

    myInfo = "|"+ this.toString() +"|"+
        "public int finishApprovedTransaction()" +"|";

    errorDetector.check (myInfo, errorDetector.errorMode) ?
        return Status.FAILURE;

    if (envelopeAccepted)
    {
        _atm.issueReceipt(_session.cardNumber(),
                          _serialNumber,
                          "DEPOSIT TO " + _bank.accountName(_toAccount),
                          _amount,
                          _newBalance,
                          _availableBalance);

        myInfo = "|"+ this.toString() +"|"+
            "public int finishApprovedTransaction()" +"|";

        errorDetector.check (myInfo, errorDetector.errorMode) ?
            return Status.FAILURE : return Status.SUCCESS;
    }
    else
        return Status.ENVELOPE_DEPOSIT_TIMED_OUT;


    // method has no postconditions
    // ---
}
```

Bendru atveju sistemos būklė gali būti išsaugoma tiek rankiniu, tiek ir automatiniu būdu. Pateiktame pavyzdyje yra iliustruojamas pirmasis variantas. Tam, kad būtų naudojamas antrasis variantas, reikėtų, kad kažkas kas nustatytą laiko intervalą pastoviai vis kreiptųsi į klasę *CheckpointRestart*, arba tiesiog šioje klasėje būtų galima realizuoti gijos veikimo mechanizmą.

Resursų gavimas yra iliustruojamas 23 paveikslėlyje pateiktoje sekų diagramoje. Joje yra vaizduojamas failo resurso gavimas, panaudojant klasę *ResourcePreallocator*.




**23 pav.** Failo resurso gavimą klaidos protokolui įrašyti iliustruojanti sekų diagrama.

### 4.6.4 Galimos pasekmės

Galimas pasekmes suformuluosiu kaip iki šiol 3.5 poskyryje pateiktų sprendimų apibendrinamuosius privalumus ir trūkumus (žr. 10 lentelę).

**10 lentelė.** 3.5 poskyryje priimtų sprendimų privalumų ir trūkumų aprašymas.

| Nr. | Privalumų / trūkumų aprašymas |
|---|---|
| **Privalumai** | |
| 1. | Pateikta standartinio klaidų apdorojimo būdo idėja yra gera tuo, kad techninės veiksmų detalės yra paslėptos (t.y. nereikia kaskart rūpintis tuo, kaip klaidų aptikimas ar kitos priemonės yra realizuotos). |
| 2. | Standartinis klaidų apdorojimo būdas pagal aprašytąją idėją ne tik užtikrina, kad visomis ar bent jau didžiąja dalimi nenumatytų klaidų bus atitinkamai pasirūpinta, bet ir turi standartizuotą formą, nusakančią, kokiais žingsniais tai bus atliekama. |
| 3. | Jei analizuojamoje sistemoje atsirastų keletas grafinių vartotojo interfeisų (ir, pavyzdžiui, juos atitinkančių klasių), tuomet klaidų pranešimų vartotojui pateikimui kiekvienai iš jų pakaktų naudoti tik vieną klasę *ErrorHandler*. |
| 4. | Klaidų pranešimų pateikimas vartotojui, naudojant klasę *ErrorHandler*, neapriboja to vien grafiniam režimui. Toks būdas puikiai tinka ir darbui tekstiniame režime. |
| 5. | Klasė *ErrorHandler* yra patogi tuo, kad ji gali būti praplečiama, adaptuojant jos teikiamą funkcionalumą ir tokiems dalykams, kaip klaidų talpinimas į buferį („*error buffering*“) ir pan. |

| | |
|---|---|
| 6. | Klaidų apdorojimo funkcionalumo enkapsuliavimas į vieną klaidų tvarkyklės klasę suteikia galimybę jį keisti, nekeičiant su tuo nesusijusių sistemos komponentų (klaidų valdymo posistemės klientų) kodą. |
| 7. | Pristatyta resursų valdymo idėja leidžia klaidų apdorojimo darbams vykti net ir tuomet, kai probleminė situacija yra „*out-of-memory*“ tipo. Kodėl? Visų pirma, todėl, kad tokio tipo klaidai aptikti gali būti naudojamos ne standartinės programavimo kalbos priemonės (kurios gali būti šiuo atveju net ir neveiksmingos), o kad ir klaidų valdymo modelio resursų valdymo klasės *ResourcePreallocator* kokia nors išvestinė klasė, pavyzdžiui, *ResourceAnalyzer*, kuri būtų aktyvi tik kaip gija. Turėdama tam reikalingas priemones, ji galėtų pastoviai dinamiškai tikrinti prieinamos atminties būklę. Jei situacija pasidarytų kritinė, tada būtų keturi galimi keliai: 1) bandyti pasinaudoti atsarginiais, t.y. iš anksto dar darbo pradžioje klasės *ResourcePreallocator* užrezervuotais atminties resursais; arba 2) visą sistemos valdymą perduoti pačiai klasei *ResourcePreallocator*, kuri iškart nutrauktų gijos veikimą, perskirstytų / „apkarpytų“ einamuoju momentu naudojamus atminties resursus ir toliau mėgintų reaktyvuoti sistemos darbą taupymo režime; arba 3) perduoti valdymą klaidų tvarkyklei *ErrorHandler*, kuri nuosekliai užbaigtų sistemos darbą dar prieš jam pačiam nutrūkstant; 4) arba perduoti laikinai valdymą *ATM* operatoriaus terminalui, kuris turi nedidelį kiekį nuosavų visų būtiniausių resursų (įskaitant maitinimą, atmintį ir pan.), kuriais būtų galima pasinaudoti. |
| **Trūkumai** | |
| 1. | Kadangi standartinis klaidų apdorojimo būdas šiuo atveju bus naudojamas visuose sistemos komponentų metoduose, kuriuose gali kilti klaidos, tai jį taikant, akivaizdžiai padidėja bendroji kodo apimtis. |
| 2. | Pakopinio darbo perkrovimo realizacija net ir bendru atveju turi neigiamos įtakos visos sistemos darbo našumui (t.y. lėtina veikimą). Dėl to, taikant tokią strategiją, reikėtų tinkamai suprojektuoti ne tik patį pakopinio perkrovimo veikimo principą (nustatyti ir suderinti komunikacijos ryšius, optimizuoti kodą, t.t.), bet parinkti ir atitinkamas konfigūracinių parametrų reikšmes (ypač per fiksuotą laiko periodą apdorojamų žingsnių skaičių). |
| 3. | Pakopinio darbo perkrovimo realizacija reikalauja papildomų vietos resursų duomenims administruoti. |
| 4. | Klaidų trasavimas bendru atveju gali parodyti tik vietą, nuo kurios veikusi programa nebeteko prieinamos atminties („*ran out of memory*“), tačiau nei jis, nei pateikta resursų valdymo idėja nesuteikia funkcionalumo, kuris būtų naudingas, atliekant atminties (o ypač dinaminės atminties) analizę visokių nutekėjimų („*memory leaks*“), konkuravimo ir kitų probleminių situacijų atžvilgiu. Tam dažnai prireikia papildomų programinių priemonių. |
| 5. | Išankstinis atminties rezervavimas dažnai ne tik komplikuoja pačius programavimo darbus, bet ir sukelia papildomas rizikas palikti klaidų, nes sistema arba jos dalis tiesiog nėra į tai orientuota (kaip, pavyzdžiui, operacinės sistemos), o naudoja tą atminties resursų valdymą tik kaip tam tikrą paslaugą. Dėl to, realizavus atminties resursų valdymą, yra patariama atlikti stresinį testavimą („*stress testing*“). |

## 4.7 KLAIDŲ LYGIAI

### 4.7.1 Problema

Bet kurioje programų sistemos architektūroje gali būti išskirti tam tikri lygiai („*layers*“) ([BMR+96]), kurie bendru atveju nusako, su kokios rūšies informacija konkrečiame lygyje yra operuojama. Kitaip tariant, kiekvienas toks lygis leidžia pasakyti, kiek bendra (o ne konkreti, techniniu žargonu suformuluota ir tik tame lygyje suprantama) yra naudojama informacija. Pavyzdžiui, jei yra rašomas kodas darbui su duomenų baze, tai *SQL* komandų vykdymo metu, jei duomenų bazė dėl kažkokių priežasčių (pavyzdžiui, trikių ar pan.) netikėtai tampa neprieinama, gali būti gautos klaidos, kurios tiesiog informuos, kad reikalinga lentelė yra nerasta („*table is missing*“), indeksas yra sunaikintas („*index is destroyed*“) ar pan., bet nepraneš, kad pati duomenų bazė galbūt „nulūžo“. Iš čia matyti, kad *SQL* komandų lygmuo yra aukštesnis už duomenų bazės lygmenį ir dėl to yra operuojama su skirtinga informacija, kuri pateiktame pavyzdyje buvo klaidų pranešimai. Taigi galima suformuluoti su klaidų valdymu susijusią problemą: kaip teisingai sugeneruoti klaidų pranešimus, nepažeidžiant konkretaus sistemos architektūrinio lygmens?

### 4.7.2 Sprendimas

Visų pirma, dėl ko pateiktoji problema galėtų būti aktuali? Priežasčių tikrai yra ne viena, pavyzdžiui, kad ir dėl to, jog žemesnio lygmens klaidų pranešimai gali būti absoliučiai nenaudingi kuriame nors aukštesniame lygmenyje, nes juos tiesiog gali būti sunku arba net visiškai neįmanoma reikalingu būdu teisingai suinterpretuoti. Be to, perduodant klaidas ar klaidų pranešimus skirtingais lygmenimis be jokio jų derinimo, gali būti pažeidžiamas informacijos slėpimo („*information hiding*“) principas. Iš kitos pusės, vis dėlto persistengti labai nereikėtų su atitikimo kuriam nors lygmeniui reikalavimais, nes tai gali neigiamai įtakoti visos sistemos darbo našumą. Taigi pirmiausia, ką reikėtų padaryti, būtų identifikuoti sistemos logines dalis, kurioms gali reikėti, kad informacija būtų atitinkamai suderinta. Tos dalys ir galėtų būti, pavyzdžiui, skirtingi sistemos lygiai, posistemės ir pan. Na, o toliau tiesiog belieka kiekvienam lygmeniui suformuluoti galimas informacijos pateikimo formas ir realizuoti patį klaidų pranešimų transformavimo iš vieno lygmens į kitą procesą. Šiam tikslui gali būti naudojama ir atitinkama klasė *ErrorAbstractor*.

Analizuojamos sistemos kontekste, kaip jau buvo minėta, pačiame aukščiausiame (taigi ir bendriausiame) lygmenyje egzistuoja dviejų rūšių klaidos: dalykinės ir techninės srities. Kadangi dalykinės srities klaidos yra orientuotos į trikius, susijusius su vartotojo atliekamomis užduotimis, tai klaidų pranešimai ir jų pobūdis jau patys savaime yra formuluojami vartotojui suprantama ir priimtina forma. Tokios formos pavyzdys klaidos *CARD_READER_CARD_JAMMED* atveju galėtų būti pateikiamas kad ir tokiu būdu:

> „*The program execution will now be suspended due to a severe system error.*
> *The message is:*
> ***Your bank card is jammed and cannot be ejected by the Card Reader.***
> *Please inform an operator at your local bank.*“

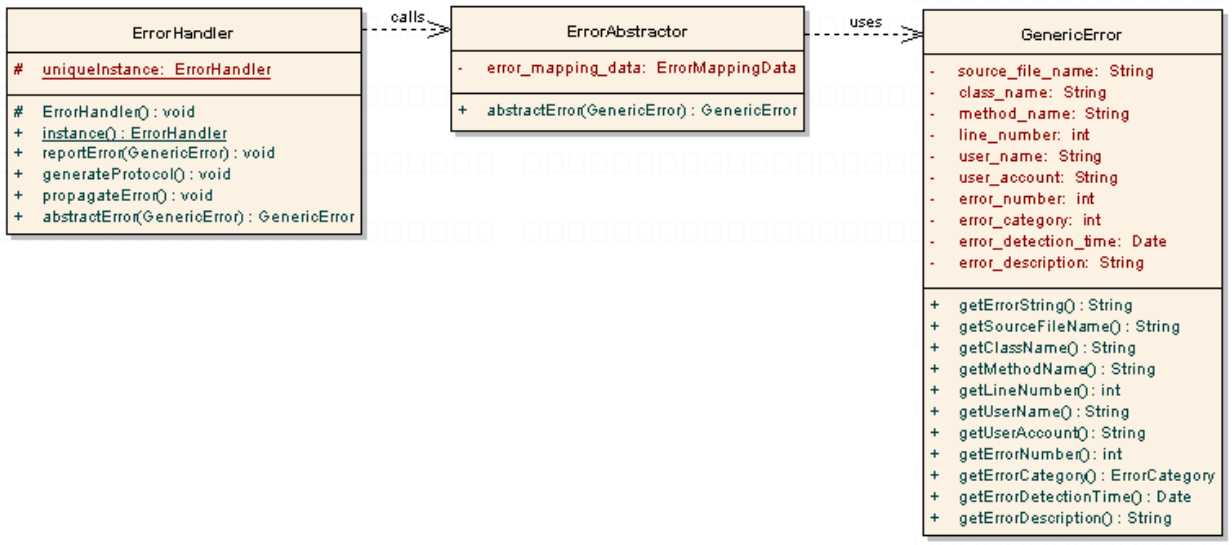
Kilus bet kokios rūšies numatytai ar nenumatytai techninės srities klaidai, kuri paprastai iššaukia ir vienos ar net kelių dalykinės srities klaidų atsiradimą, vartotojas taip pat turėtų būti informuojamas jam suprantama ir aiškia forma apie sistemoje susidariusią situciją. Tam tikslui gali pasitarnauti klaidų lygiai, t.y. kiekviena aptikta techninės srities klaida (o bendru atveju ir bet kokia klaida) galėtų būti asocijuojama su kokia nors konkrečia vykdymo baigtimi („*completion*“). Vienas galimų tokių baigčių identifikatorių pavyzdžių yra pateiktas 11 lentelėje.

**11 lentelė.** Apibendrintų operacijų vykdymo baigčių identifikatorių pavyzdys.

| Nr. | Operacijos baigties identifikatorius | Reikšmė |
|---|---|---|
| 1. | **`EXECUTION_SUCCESS`** | Operacija, užklausa ar veiksmas buvo įvykdyti sėkmingai (t.y. be jokių aptiktų ar neaptiktų trikių buvo gautas kažkoks rezultatas). |
| 2. | **`EXECUTION_NO_EFFECT`** | Operacija, užklausa ar veiksmas nebuvo įvykdyti iki galo, bet jokių papildomų trikių ar klaidų neatsirado, arba buvo įvykdyti sėkmingai, tačiau nebuvo gautas joks planuotas rezultatas. |
| 3. | **`EXECUTION_SEVERE_EFFECT`** | Operacija, užklausa ar veiksmas nebuvo įvykdyti iki galo, o eigoje atsirado aptiktų arba neaptiktų trikių arba klaidų. |

### 4.7.3 Realizacija

Kadangi su kylančiomis klaidomis dirba klaidų tvarkyklės, tai būtų visai logiškai, jei jų perdavimą iš žemesnio lygmens į aukštesnį atliktų, būtent, jos, pasinaudodamos klasės *ErrorAbstractor* teikiamomis paslaugomis (žr. 24 paveikslėlį).




**24 pav.** Klaidų lygių realizavimo struktūrinis pavaizdavimas.

Klaidų lygių klasės *ErrorAbstractor* panaudojimą iliustruosiu metodo *finishApprocedTransaction* kontekste (žr. 6 pavyzdį), kuriame aptikta klaida yra apibendrinama ir pateikiama, o vartotojui yra pateikiamas atitinkamas pranešimas.

**6 pavyzdys.** Klasės *ErrorAbstractor* panaudojimas.

```
…
protected Session _session;
protected ATM     _atm;
protected Bank    _bank;
protected Money   _newBalance;
protected Money   _availableBalance;
protected Money   _amount;
protected Bank    _bank;
protected int     _serialNumber;
protected int     _toAccount;
…

public int finishApprovedTransaction()
{
    // create new instance of ErrorDetector class
    ErrorDetector errorDetector = ErrorDetector::instance();

    // create new instance of ErrorDetector class
    ErrorHandler errorHandler = ErrorHandler::instance();

    // method has no invariants
    // ---

    String myInfo = "|"+ this.toString() +"|"+
```

```
        “public int finishApprovedTransaction()” +“|”;

    errorDetector.switchOnNotification();

    // check to see if parameters are not empty, if they are available (e.g.
    // reachable and not in an error state) and can provide us information

    //
    // …
    //

    CheckpointRestart ckpRestart = CheckpointRestart::instance();

    ckpRestart.ckpParameters.source_file_name  = __FILE__;
    ckpRestart.ckpParameters.class_name        = “DepositTransaction”;
    ckpRestart.ckpParameters.method_name       = “finishApprovedTransaction”;
    ckpRestart.ckpParameters.line_number       = __LINE__;
    ckpRestart.ckpParameters.user_name         = _bank.userName();
    ckpRestart.ckpParameters.user_account      = _bank.currentAccountName();
    ckpRestart.ckpParameters.activity_data     = getActivityData();
    ckpRestart.ckpParameters.status            = getStatus();

    ckpRestart.writeCheckpoint();

    _bank.finishDeposit(_atm.number(), _serialNumber, envelopeAccepted);

    if (errorDetector.errorMode)
    {
        // to hide the tangling with errors, this can be done
        // in some other class as well

        errorHandler.reportError (errorHandler.errorAbstractor.
           abstractError (errorDetector.lastDetectedError));

        return Status.FAILURE;
    }

    if (envelopeAccepted)
    {
        _atm.issueReceipt(_session.cardNumber(),
                          _serialNumber,
                          "DEPOSIT TO " + _bank.accountName(_toAccount),
                          _amount,
                          _newBalance,
                          _availableBalance);

        myInfo = “|”+ this.toString() +“|”+
            “public int finishApprovedTransaction()” +“|”;

        errorDetector.check (myInfo, errorDetector.errorMode) ?
            return Status.FAILURE : return Status.SUCCESS;
    }
    else
        return Status.ENVELOPE_DEPOSIT_TIMED_OUT;


    // method has no postconditions
    // ---
}
```

### 4.7.4 Galimos pasekmės

Galimas pasekmes suformuluosiu kaip iki šiol 3.6 poskyryje pateiktų sprendimų apibendrinamuosius privalumus ir trūkumus (žr. 12 lentelę).

**12 lentelė.** 3.6 poskyryje priimtų sprendimų privalumų ir trūkumų aprašymas.

| Nr. | Privalumų / trūkumų aprašymas |
|---|---|
| **Privalumai** | |
| 1. | Klaidų lygių realizavimas suteikia galimybę diferencijuoti operuojamą informaciją (pavyzdžiui, klaidų pranešimus) pagal skirtingus poreikius, atsirandančius skirtinguose sistemos lygiuose. Tokiu būdu yra išlaikomas informacijos slėpimo principas, pereinant į aukštesnį lygmenį, yra enkapsuliuojamas žemesnio lygmens kompleksiškumas ir suvienodinama informacijos pateikimo vartotojui forma. |
| **Trūkumai** | |
| 1. | Jei pereinant iš vieno lygio į kitą tos pačios apibendrintos informacijos prisireiks kelioms klasėms iškart (pavyzdžiui, paleistoms veikiančioms gijoms), tuomet klasėje *ErrorAbstractor* ar atskiroje tam dedikuotoje klasėje papildomai bus reikalingas efektyvus sinchronizavimo mechanizmas, atitinkamai koordinuojantis informacijos perdavimo procesą. Esant neefektyviam sinchronizavimo mechanizmui, tai gali neigiamai atsiliepti sistemos darbo našumui. |
| 2. | Kadangi klasė *ErrorAbstractor* yra tik viena (t.y. realizuota pagal *Singleton* projektavimo šabloną), tai palikus pervedimo iš vieno lygio į kitą procese klaidų, sistemos elgsena gali tapti nebekontroliuojama ir grandinine reakcija prasidėti kylančių klaidų antplūdis (pavyzdžiui, jei bent kelioms skirtingoms klasėms buvo perduota ne to formato (neatpažįstama) informacija ir pan.). |

## 4.8 DAUGIAGIJIS KLAIDŲ APDOROJIMAS

### 4.8.1 Problema

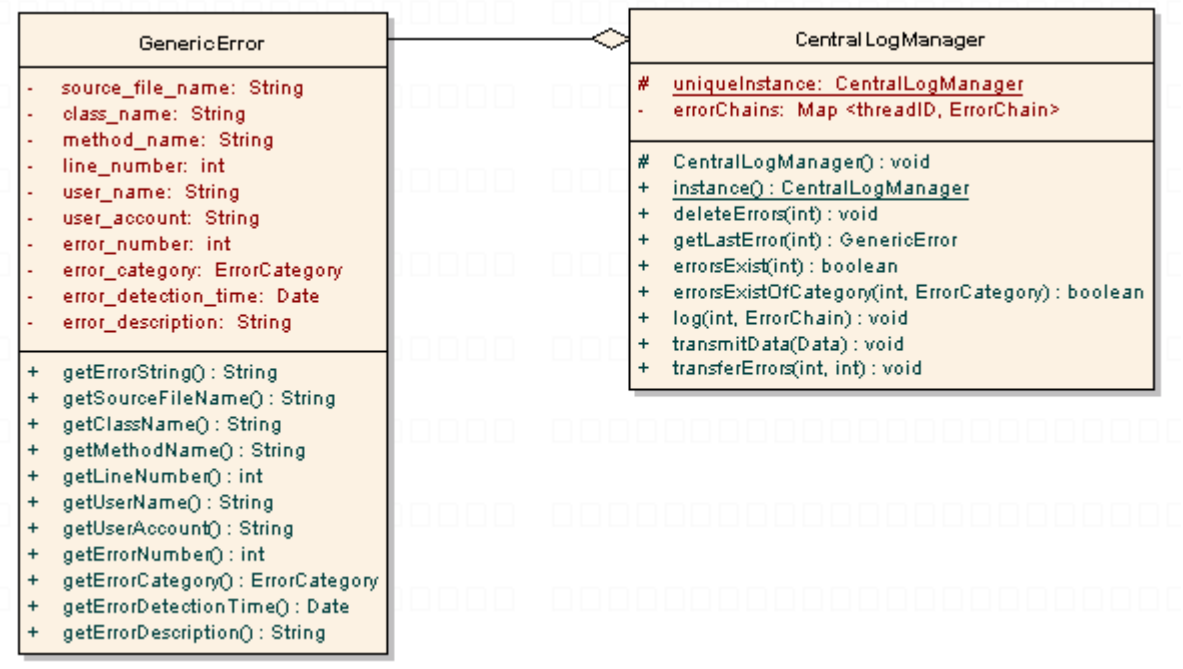
Ši probleminė sritis, kaip ir pats jos pavadinimas sako, daugiausia yra orientuota į sistemos darbą, kurį atlieka daugiau nei viena gija, arba, kitaip tariant, į daugiagiję aplinką („*multithread environment*“). Sunkumai prasideda tada, kai lygiagrečiai veikiančių gijų darbas yra netikėtai sutrikdomas atsiradusių klaidų. Tokiu atveju aptiktos klaidos turi būti perduotos pagrindinei gijai, kurioje ir turėtų įvykti pats apdorojimas. Taigi problema yra tokia: kaip suvaldyti klaidas daugiagijoje aplinkoje?

### 4.8.2 Sprendimas

Šiame darbe pateiktoji klaidų valdymo posistemės architektūrinė schema ir visi jos papildymai puikiai tinka taip pat ir daugiagijos aplinkos atvejui. Vienintelis esminis skirtumas būtų tik toks, kad centralizuoto klaidų valdymo klasė *CentralLogManager* turėtų būti realizuota taip, kad saugotų kiekvienos aktyvios gijos klaidų protokolus individualiai. Kitaip tariant, naujajam šios klasės egzemplioriuje reikės tik turimus metodus papildyti gijų identifikatoriaus parametru *threadID* ir į pačią klasę įtraukti vieną naują metodą *transferErrors()*, kuris leistų likusiai gyvai gijai iš jau nebegyvų gijų susirinkti jų klaidų protokolus ir perduoti juos klaidų apdorojimui. Ir nors pačioje analizuojamoje sistemoje šis būdas nebus naudojamas, nes ji yra realizuota taip, kad joje veikia tik viena gija, tačiau galimą tokio sprendimo realizacijos struktūrinį fragmentą vis dėlto pateiksiu kaip smulkų pavyzdėlį, iliustruojantį kaip tai galėtų atrodyti (žr. 3.7.3 poskyrį).

### 4.8.3 Realizacija

Daugiagijai aplinkai adaptuota centralizuota klaidų valdymo klasė *CentralLogManager* galėtų būti panaši į klasę, pateiktą 25 paveikslėlyje.




**25 pav.** Daugiagijai aplinkai adaptuotos centralizuoto klaidų valdymo klasės struktūrinis pavaizdavimas.

### 4.8.4 Galimos pasekmės

Galimas pasekmes suformuluosiu kaip iki šiol 3.7 poskyryje pateiktų sprendimų apibendrinamuosius privalumus ir trūkumus (žr. 13 lentelę).

**13 lentelė.** 3.7 poskyryje priimtų sprendimų privalumų ir trūkumų aprašymas.

| Nr. | Privalumų / trūkumų aprašymas |
|---|---|
| **Privalumai** | |
| 1. | Pateiktasis daugiagijo klaidų apdorojimo variantas yra geras tuo, kad jis yra saugus pačių gijų atžvilgiu („*thread safe*“) ir kad informacija apie klaidas gali būti surenkama iš neveiksnių gijų ir perduodama apdorojimui, padidinant tikimybę, kad sistemos darbas bus atstatytas. |
| **Trūkumai** | |
| 1. | Bėdų gali kilti su klaidų pranešimų suderinamumu, ypač jei kuriai nors likusiai gyvai gijai, dirbančiai aukštesniame lygmenyje, reikia ne tik perimti iš neveiksnių gijų, esančių žemesniame lygmenyje, kokią nors su klaidomis susijusią informaciją, bet ir atitinkamai į ją sureaguoti (pavyzdžiui, nuskaityti kitų gijų paskutinę saugomą klaidą tam, kad nuspręsti ar likusi gyva gija dar gali dirbti ir taip nepridarys dar daugiau bėdos). Tokiu atveju problemos sprendimui gali prireikti net ir nedidelių architektūrinių pakeitimų (pavyzdžiui, klasės vertėjo („*interpreter*“), kurios paskirtis būtų ne grąžinti apibendrintos klaidos objektą, o tiesiog leisti vienam lygiui, „šnekančiam“ viena kalba „paskaityti“ kito lygio „užrašus“ ir pan.). |

## 4.9 SISTEMOS REAKTYVAVIMAS

### 4.9.1 Problema

Bet koks klaidų apdorojimas programų sistemoje visada yra orientuotas į tos sistemos teikiamo funkcionalumo atstatymą. Vis dėlto kartais nepakanka vien tik pranešti vartotojui apie susidariusią klaidos situaciją ar panaudoti standartinius sistemos klaidų apdorojimo būdus. Priklausomai nuo klaidos kritiškumo sistemos funkcionalumo atstatymui gali tekti naudoti ne tik numatytas klaidų tvarkykles, bet ir specialius sistemos reaktyvavimo būdus. Tokių būdų realizacija dažnai yra susijusi su struktūriniu adaptyvumu („*structural adaptation*“) arba dar kitaip dinaminė rekonfigūracija („*dynamic reconfiguration*“), kuri reikalui esant, leidžia dinamiškai koreguoti net ir pačią sistemos architektūrą. Taigi galima iškelti klausimą: koks gali būti sistemos reaktyvavimo modelis ir ko jam gali reikėti?

### 4.9.2 Sprendimas

Paprastai dinaminė rekonfigūracija ([BJC05]) yra realizuojama taip, kad jos procesas būtų kiek įmanoma labiau automatizuotas. Dėl to jos svarba ir yra didžiausia itin kritinio veikimo programų sistemų kūrimo procesuose. Kalbant apie dinaminės rekonfigūracijos procesą, svarbu yra numatyti galimus rekonfigūracijos scenarijus. Kadangi pastarųjų dažnai egzistuoja bent keletas ir jie nėra taip jau lengvai realizuojami, kaip gali atrodyti iš pirmo žvilgsnio, tai siekiant pailiustruoti tik pačią dinaminės rekonfigūracijos pritaikymo idėją, analizuojamosios sistemos kontekste pateiksiu tik vieną scenarijų:

- Jei vartotojo darbo su analizuojamąja sistema metu susidaro kritinė situacija, dėl kurios visa bankomato sistema tampa nebeveiksni (pavyzdžiui, dėl elektros srovės dingimo, trumpo sujungimo ar kitų priežasčių) arba kyla kritinė nežinomos rūšies klaida, tai *ATM* operatoriaus terminalas gali laikinai (pavyzdžiui, vieną sesiją) suimituoti bankomato darbą (kad ir, pavyzdžiui, nebe grafiniame, o tekstiniame režime), kurio metu vartotojas bus informuojamas apie susidariusią klaidos situaciją, bet jam bus leidžiama pabandyti pakartoti / pabaigti paskutinę sėkmingai inicijuotą užduotį ar bent jau išsiimti savo banko kortelę. Jei nei vienas iš dviejų variantų nepavyksta, vartotojui gali būti pabandoma pranešti apie klaidą arba vaizduoklio ekrane, arba atspausdinant atitinkamą pranešimą kvito lapelyje. Jei nepavyksta net tai, tuomet operatoriaus terminalas pabando perkrauti sistemą (išjungti / įjungti). Jei nepavyksta, terminalas nutraukia bankomato sistemos darbą ir pasiruošia avariniam aktyvavimuisi, kurio metu aktyvuojasi tik operatoriaus terminalas, kuris leidžia atvykusiam operatoriui peržiūrėti paskutinius atmintyje išsaugotus duomenis (pavyzdžiui, duomenų registro failus, klaidų protokolus ir t.t.), vykdytas operacijas ir t.t.

Tokio pobūdžio procesas galėtų būti laikomas automatine architektūrine dinamine rekonfigūracija ([NBS00]). Jokių papildomų klasių šiuo atveju nereikės, nes pagrindinė *ATM* klasė jau turi ryšį su klase *OperatorPanel*. Taigi viskas, kas belieka, tai tik tarp turimų klasių sukurti reikalingus architektūrinius ryšius.

### 4.9.3 Realizacija

Vienas iš galimų būdų realizuoti pateiktąjį dinaminės rekonfigūracijos scenarijų būtų tiesiog susieti *ATM* operatoriaus terminalo klasę su *ATM* klase ir atskirai dar su keliomis kitomis klaidų valdymo posistemės klasėmis (tokiomis kaip *CheckpointRestart* ir kt.). Tokiu atveju klasė *OperatorPanel*, nepaisant jos galimybės nepriklausomai nuo visos bankomato sistemos paleisti ir save pačią, taptų antrąja (avarine) bankomato sistemos pagrindine paleidžiamąja dalimi *main* (žr. 17 paveikslėlį).

### 4.9.4 Galimos pasekmės

Galimas pasekmes suformuluosiu kaip iki šiol 3.8 poskyryje pateiktų sprendimų apibendrinamuosius privalumus ir trūkumus (žr. 14 lentelę).

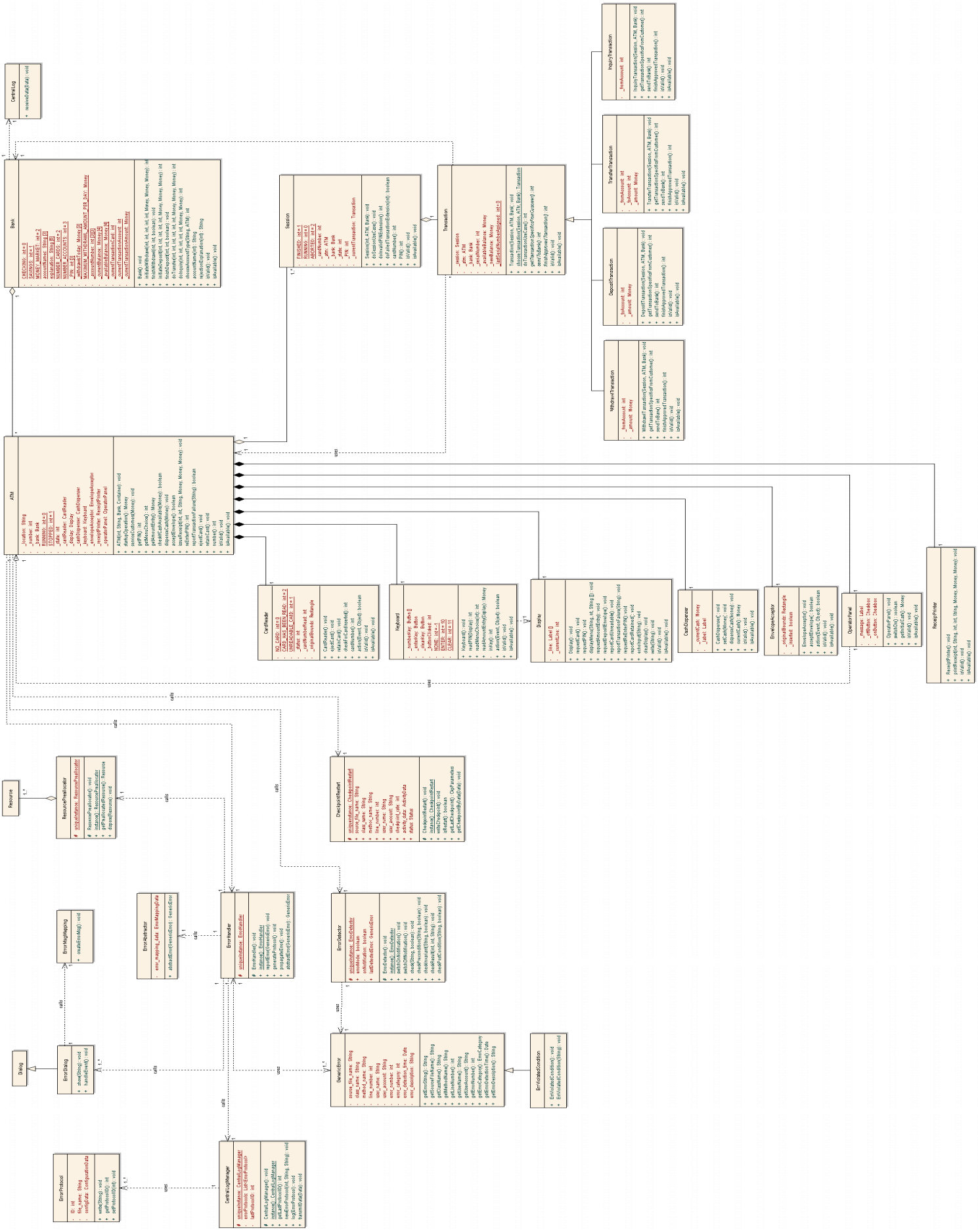
**14 lentelė.** 3.8 poskyryje priimtų sprendimų privalumų ir trūkumų aprašymas.

| Nr. | Privalumų / trūkumų aprašymas |
|---|---|
| **Privalumai** | |
| 1. | Pateiktas analizuojamosios sistemos reaktyvavimo scenarijus yra paprastas ir leidžia bankomate turėti 2 kontrolės punktus, orientuotus į darbą su vartotoju, ir 1 avarinį kontrolės punktą, orientuotą į darbą su bankomato operatoriumi. |
| **Trūkumai** | |
| 1. | Aprašytasis analizuojamosios sistemos reaktyvavimo funkcionalumas bus nepasiekiamas, jei bankomate nebus dubliuota elektros tiekimo linija (viena, skirta visai bankomato sistemai, kita – operatoriaus terminalui). Be to, operatoriaus terminalas turėtų kažkokiu būdu nepriklausomai kas fiksuotą laiko periodą sekti pagrindinės bankomato sistemos darbo būklę ir gebėti pakrauti sistemą ir nustatyti jos darbinę būseną taip, kad ji pereitų prie paskutinės sėkmingai vartotojo inicijuotos užduoties vykdymo, o tai nėra lengvai realizuojamas iššūkis nei techniškai, nei programiškai. |

## 4.10 PILNA KLAIDOMS ATSPARI NAUJOSIOS ANALIZUOJAMOSIOS SISTEMOS ARCHITEKTŪRA

Apžvelgus visas šiame darbe pristatyto klaidų valdymo modelio problemines sritis, galima sudaryti jau ir bendrą patobulintą pavyzdinės analizuojamosios sistemos architektūrą (šiuo atveju klasių diagramą), kuri būtų atspari klaidoms ir pakankamai imli įvairiems pakeitimams ar papildymams (žr. 26 paveikslėlį). Dabar tokią architektūrą sudaro jau dvi dalys: pagrindinė

dalykinės srities (t.y. *ATM* bankomato funkcionalumą realizuojanti) posistemė ir klaidų valdymo posistemė. Atsakymas į klausimą, kiek iš tikrųjų tokia sistema yra atspari klaidoms, yra trumpai apžvelgta kitame 4.2 poskyryje.



**26 pav.** Klaidoms atsparios pavyzdinės analizuojamosios sistemos architektūra.

## 4.11 KLAIDŲ VALDYMO MODELIO PRIVALUMAI IR TRŪKUMAI

Apibendrinamuosius klaidų valdymo modelio privalumus ir trūkumus pateikiu 21 lentelėje.

**21 lentelė.** Klaidų valdymo modelio privalumų ir trūkumų aprašymas.

| Nr. | Privalumų / trūkumų aprašymas |
|---|---|
| **Privalumai** | |
| 1. | Pristatytas klaidų valdymo modelis yra bendrojo pobūdžio, specifikuojantis visas pagrindines problemines sritis, kurios įeina į su klaidų valdymu susijusių sprendimų analizės, projektavimo ir pritaikymo nagrinėjamoje sistemoje sferas. O tai reiškia, kad jis gali būti naudojamas kaip gera praktika, paremta šiandieninėje industrijoje vyraujančių tendencijų susisteminta žinių baze. |
| 2. | Klaidų valdymo modelis numato, kas turi būti atlikta, o ne kaip, dėl to programų sistemų kūrimo projektuose už atsakingi žmonės gali nevaržomai ieškoti ir patys priimti konkrečius juos tenkinančius sprendimus. |
| 3. | Klaidų valdymo modelį sudaro daugiau ar mažiau viena nuo kitos nepriklausomos probleminės sritys, o tai leidžia aiškiai pasiskirstyti darbais / darbo grupėmis ir toliau dirbti individualiai, lygiagrečiai atliekant visus darbus. Tai projektuose yra pakankamai svarbu, nes lygiagretus darbų vykdymas leidžia taupyti laiką ir didintį produktyvumo laipsnį. |
| 4. | Kadangi klaidų valdymo modelis yra bendrinis, tai jo praktinis panaudojimas gali apimti ne tik programų sistemų kūrimo industriją, bet ir akademinę tokios veiklos sritį (nuo mokyklinio lygio ir universitetinio). |
| **Trūkumai** | |
| 1. | Klaidų valdymo modelis vis dėlto stokoja konkretumo, nes kiekvienai jį sudarančiai probleminei sričiai egzistuoja gana nemaža sprendimų aibė, su kuria susipažinti, išnagrinėti, įvertinti ir pasirinkti joje tinkamą būdą kokiame nors programų sistemų kūrimo projekte gali būti per sudėtinga tiek laiko, tiek ir reikalingą kompetenciją turinčių žmogikųjų išteklių atžvilgiais. Dėl to toks modelis galėtų būti dar labiau detalizuotas ir standartizuotas (pavyzdžiui, nusakantis kokie yra efektyvūs kiekvienos probleminės srities sprendimų paieškos kriterijai, kokie konkrečiai turėtų būti rezultatai, kokių darbų ar resursų tam gali reikėti ir pan.). |
| 2. | Klaidų valdymo modelis nepateikia jokių sistemos atsparumo klaidoms vertinimo ar testavimo kriterijų, kurie leistų nustatyti, kiek taikytas modelis buvo realizuotas, koks atsparumo klaidoms laipsnis, kokie gali būti kiekvienos probleminės srities testavimo reikalavimai ir pan. Kitaip tariant, trūksta informacijos kiekybinei ir kokybei analizei atlikti. |

| | |
|---|---|
| 3. | Klaidų valdymo modelis jokios su projektų valdymu susijusios informacijos (pavyzdžiui, modelio pritaikymo vykdomojo projekto kontekste strateginis planavimas, reikalingos infrastruktūros palaikymas, galimos veiklos, rizikos ir pan.). |

# 5. VERTINIMO METODIKOS

## 5.1 TRUMPA SUSIJUSIŲ VERTINIMO METODIKŲ APŽVALGA

Atlikus literatūros apžvalgą, paaiškėjo, kad konkrečių vertinimo metodikų, kurios būtų skirtos arba tiktų panašių evoliucionuojančių sistemų klaidų atsparumui vertinti, deja, nepavyko rasti. Dėl to šiame darbe yra pasiūlyta individualiai parengta vertinimo metodika, kuri yra vadinama *PCMB*. Detaliau ji yra aprašoma tolimesniame poskyryje, o kol kas pateiksiu trumpą apžvalgą 3 vertinimų metodikų, kurios buvo didžiausios pretendentės į šiame darbe pateiktosios naujosios analizuojamos sistemos klaidų atsparumo vertinimą.

Pirmoji vertinimo metodija buvo viena iš airių organizacijos „*Software Quality Research Laboratory*“ siūlomų vertinimo metodikų pavadinimu „*Availability evaluation of hardware / software systems with several recovery procedures*“ ([VPM+03]). Šios metodikos taikymas yra paremtas formaliu klaidų modelio („*Segregated failure model*“) sudarymu ir įvertinimu. Tai atliekant yra daroma prielaida, kad sistema kiekvienu laiko momentu gali būti vienoje iš dviejų būsenų: normalioje darbinėje būsenoje („*normal (working) state*“) ir klaidos būsenoje („*failed state*“). Su kiekviena būsena yra susieta tam tikra aibė parametrų, kurių reikšmės yra tiksliai randamos arba apytiksliai įvertinamos taikant nustatytus formalius būdus. Ši vertinimo metodika daugiausia yra orientuota į sistemos atstatymo iš klaidos būsenos į normalią darbinę būseną įvertinimui. Ji netiko, nes pirmiausia yra orientuota tik į vieną iš klaidų valdymo modelį sudarančių probleminių sričių, t.y. sistemos reaktyvavimą, ir dėl to, kad reikalauja duomenų, kurių šiame darbe nebuvo galima paskaičiuoti.

Antroji vertinimo metodika yra Didžiosios Britanijos universiteto „*University Of Newcastle*“ kompiuterių laboratorijos darbuotojų sudaryta vertinimo metodika pavadinimu „*Definition And Evaluation Of Adaptive Architectures In A Distributed Computing Environment*“ ([GBX+00]). Joje pirmiausia yra formaliai apibrėžiama nagrinėjamos sistemos architektūra. Tai yra padaroma, naudojant atitinkamas duomenų grupes. Bendra architektūrą nusakanti išraiška yra $X(F, N, H_b, H_{\max}, \ldots)$, kurioje žymuo $X$ reiškia pasirinktą nagrinėti architektūrą, $F$ – leistiną klaidų skaičių, kurį sistema dar $X$ gali toleruoti jos veikimo metu, $H_b$ – minimalių bazinių architektūrinių reikalavimų, orientuotų į norimą pasiekti klaidų atsparumą, aibė ir $H_{\max}$ – maksimalių architektūrinių reikalavimų, orientuotų į norimą pasiekti klaidų

atsparumą, aibė. Kiekviena šių duomenų grupė yra dar labiau detalizuojama atitinkamais nustatytų parametrų duomenimis. Toliau yra išskiriamos statinės ir dinaminės nagrinėjamos architektūros dalys, kiekvienai iš grupių yra apskaičiuojamas nustatytas tikimybinis klaidų pasireiškimo tose dalyse modelis ir atitinkamai įvertinami tokie parametrai kaip vidutinis tikėtinas sunaudojamų resursų kiekis („*Average resource consumption*“), vidutinis tikėtinas sistemos reakcijos laikas („*Average response time*“) ir pan. Visoms gautoms reikšmėms yra nubraižomi atitinkami (pavyzdžiui, pasiskirstymo laike) grafikai. Ši vertinimo metodika netiko dėl to, kad jai yra reikalinga veikianti sistema arba veikiantis jos prototipas, o taip pat ir dėl to, kad jos įvertinime nedalyvauja joks kodas, kuris šiame darbe yra svarbi vieta nagrinėti klaidų identifikavimą, aptikimą, apdorojimą ir t.t.

Na, ir trečioji vertinimo metodika būtų Oksfordo universiteto kompiuterių laboratorijos darbuotojos sudaryta vertinimo metodika pavadinimu „*Quantitative Verification: Models, Techniques and Tools*“ ([Mar07]). Ši metodika yra taip pat paremta formalia sistemos analize, kurioje dalyvauja formaliai apibrėžiamos galimos sistemos veikimo būsenos, nagrinėjami patekimo į juos keliai (operacijų sekos), galimoms klaidoms yra sudaroma formali specifikacija, konkretizuojamas specifikacijos vertinimo modelis ir paskaičiuojami atitinkamų tikimybinių parametrų reikšmės, kurios atspindi klaidų kilimo priežastingumą, pasklidimą, kritiškumą ir kitus aspektus. Šis modelis netiko dėl to, kad jis yra per daug formalus, reikalauja nemenkos darbo su formaliais metodais kompetencijos, nemažų rezultatų interpretavimo įgūdžių ir kitų teorinių žinių, apie kurių įsisavinimą ir pritaikymą būtų galima parengti atskirą darbą.

## 5.2 ĮVERTINIMAS PAGAL *PCMB* VERTINIMO METODIKĄ

### 5.2.1 Prielaidos ir reikalavimai

Prieš pateikiant detalų empirine analize paremtos *PCMB* vertinimo metodikos aprašymą, pirmiausia reikėtų nustatyti, kokioms prielaidoms esant ši metodika gali būti apkritai taikoma praktinėje veikloje. Šios prielaidos yra tokios:

1. Nagrinėjama programų sistema yra komponentinė (t.y. visas jos funkcionalumas gali būti išreikštas panaudojant atitinkamus komponentus ir ryšius tarp jų).

2. Senosios ir naujosios kuriamos programų sistemos versijų[3] projektinės diagramos (pavyzdžiui, eskizinis projektas, detalus projektas, klasių diagrama ar pan.) yra teisingos (bent jau tiek, kad iš jų būtų galima gauti kokią nors veikiančią realizaciją).

3. Sprendžiamos dalykinės srities (šiuo atveju banko operacijų vykdymas ne banko patalpose, o *ATM* bankomatuose) realizacija senojoje ir naujojoje programų sistemos versijose turi išlikti tokia pati (tai yra reikalinga tam, kad būtų vertinami tik su klaidų valdymu susiję aspektai).

4. Visas vertinimo procesas yra grindžiamas taip, kad pirmiausia yra padaromos teisingos prielaidos, paremtos stebėjimais, praktika, tendencijomis, asmenine patirtimi ir pan., o po to iš jų yra išvedamos (bent jau teoriškai) teisingos išvados.

Toliau yra svarbu atsakyti į klausimą, o kokiais resursais minėtoji vertinimo metodika gali remtis, kad ją būtų galima taikyti. Tam tikslui pačiu paprasčiausiu atveju yra reikalinga tenkinti bent jau šį reikalavimą:

1. Praktinis vertinimo metodikos pagrindas yra lyginamoji analizė, dėl to yra būtina turėti senąją ir naująją kuriamos programų sistemos versijas (minimaliu atveju yra reikalinga, kad būtų baigti bent senosios ir naujosios programų sistemos projektavimo etapai, o geriausiu atveju gali egzistuoti jau ir pilnai veikiančios sistemos ar kurios nors jų realizacijos versijos).

Na, ir paskutinis dalykas, kurį reikia paminėti – tai atsakymas į klausimą, kokie yra *PCMB* vertinimo metodikos įeities duomenys? Šiuos duomenis galima įvardinti taip:

1. Senosios programų sistemos architektūrinis modelis, kuriame atsispindi visi pagrindiniai tos sistemos komponentai (pavyzdžiui, klasės) ir jų tarpusavio ryšiai.

2. Naujosios programų sistemos architektūrinis modelis, kuriame atsispindi visi pagrindiniai tos sistemos komponentai (pavyzdžiui, klasės) ir jų tarpusavio ryšiai.

[3] Šiame kontekste senoji programų sistemos versija yra traktuojama kaip sistema iki kokio nors klaidų valdymo modelio duotojoje sistemoje suprojektavimo, o naujoji – jau turinti pilnai suprojektuotą klaidų valdymo mechanizmą.

3. Senąją programų sistemą realizuojantis kodas arba kitoks aprašas, kuriame yra pateikti tos sistemos komponentus sudarantys metodai ir kiti duomenys.

4. Naująją programų sistemą realizuojantis kodas arba kitoks aprašas, kuriame yra pateikti tos sistemos komponentus sudarantys metodai ir kiti duomenys.

### 5.2.2 Vertinimo metodikos aprašymas

*PCMB* („*Plan, Check, Manage, Boost*“) – tai empirine lyginamąja analize paremta vertinimo metodika, kuri nustatytais kiekybiniais aspektais nagrinėja keturis su klaidų valdymu tiriamojoje programų sistemoje susijusius aspektus: planavimą, tikrinimą, pajėgumą ir pakankamumą.

#### 5.2.2.1 *Planinis koeficientas*

Planavimo aspektas yra skirtas atsakyti į klausimą, kiek pasikeitė senojoje ir naujojoje programų sistemos versijose apdorojamų klaidų skaičius. Jis remiasi principu, teigiančiu: kuo daugiau klaidų sužinosi anksčiau, tuo labiau būsi joms pasiruošęs. Kadangi klaidas pačiu bendriausiu atveju galima skirstyti į planuotas (nuspėjamas) ir neplanuotas (nenuspėjamas), o einamuoju momentu dėmesys yra orientuotas būtent į planuotas klaidas, tai su planavimo aspektu susijęs kiekybinis matas taip ir vadinasi: planinis koeficientas (žymimas *PC* („*Planning Coeficient*“)).

Tarkime, kad senojoje programų sistemos versijoje, sudarytoje iš $C_1, C_2, \ldots, C_N$ komponentų (šiuo atveju klasių), iš viso gali kilti $TE_B$ planuotų ir neplanuotų loginių (bet ne sintaksinių ar semantinių) klaidų be pasikartojimų. Taip pat tarkime, kad komponente $C_1$ gali kilti $E_1$ klaidų, komponente $C_2$ - $E_2$ klaidų ir t.t. Tada $TE_B = E_1 + E_2 + \ldots + E_N$. Be to, sistemoje iš viso buvo numatyta apdoroti $P_B$ klaidų.

Realizavus tokioje sistemoje kokį nors klaidų valdymo modelį, buvo gauta naujoji programų sistemos versija, kurios architektūra buvo papildyta *K* naujais komponentais: $C_1, C_2, \ldots, C_N, C_{N+1}, C_{N+2}, \ldots, C_{N+K}$. Tada logiška tikėtis, kad sulig kiekvienu nauju komponentu padidėjo ir galimų kilti klaidų kiekis: $TE_A = TE_B + E_{N+1} + E_{N+2} + \ldots + E_{N+K} = TE_B + \Delta TE$. Iš

kitos pusės, naujoji programų sistema buvo realizuota tam, kad galėtų apdoroti daugiau įvairesnių klaidų. Iš čia turime, kad $P_A = P_B + \Delta P$. Taigi bendrą planinį koeficientą galima įsivaizduoti kaip tam tikrą santykį, kuris parodytų, kiek daugiau naujojoje programų sistemos versijoje yra numatyta apdoroti klaidų negu senojoje arba, kitaip tariant, kiek mažiau neplanuotų klaidų liko naujojoje programų sistemos versijoje negu senojoje.

$$PC = \frac{TE_B - P_B}{TE_A - P_A} = \frac{TE_B - P_B}{TE_B + \Delta TE - P_B - \Delta P} = \frac{TE_B - P_B}{(TE_B - P_B) + (\Delta TE - \Delta P)}.$$

Nežinomiems išraiškos dydžiams įvertinti gali būti taikomi bet kurie būdai, kurie leidžia nustatyti bent apytikslį ne analizuojamoje sistemoje (pavyzdžiui, operacinėje sistemoje, standartinių funkcijų viduje ir pan.) galinčių kilti klaidų skaičių. Taip pat galima bandyti skaičiuoti gautos išraiškos ribą ar taikyti kitokius įvertinimo metodus. Bendru atveju vertinimo tendencija būtų tokia: kuo santykio reikšmė didesnė, tuo nenumatytų klaidų skaičius naujojoje analizuojamoje sistemoje yra mažesnis ir tuo labiau iš anksto galima pasirūpinti tų klaidų apdorojimu.

Pateiktosios formulės taikymą iliustruosiu konkrečia situacija (žr. 7 pavyzdį[4]). Analogišku būdu ši formulė yra taikoma ir skyrelyje 5.2.2.5 „*Atlikti skaičiavimai*“.

**7 pavyzdys.** Planinio koeficiento išraišką randančios formulės taikymą iliustruojanti situacija.

Tarkime, kad turime paprastą sistemėlę, susidedančią iš vienos klasės *Calculus*. Joje yra metodas, skirtas suskačiuoti dviejų sveikųjų skaičių sumą (žr. toliau).

```
class Calculus
{
    int lastUsedNumber1 = 0;
    int lastUsedNumber2 = 0;
    int lastFoundResult = 0;

    public Calculus() {};
    public int add (int num1, int num2)
    {
        number1         = num1;
        number2         = num2;
        lastFoundResult = num1 + num2;

        return lastFoundResult;
    }
}
```

[4] Šiame pavyzdyje pateiktų sistemėlių versijos nėra labai geri *PCMB* vertinimo metodikos taikymo kontekstai, nes į jas nėra įtraukti jokie klaidų valdymo būdai. Vis dėlto jis yra tikrai paprastas, kad būtų galima geriau paaiškinti pačią aprašomos vertinimo metodijos taikymo eigos esmę.

Pažiūrėję į šios klasės aprašą, galime identifikuoti bent keletą klaidų, kurios gali įvykti vykdant metodą *add*. Šios klaidos gali būti, pavyzdžiui, dvi: *argument_specification_error* ir *arithmetic_overflow_error*.

Dabar tarkime, kad iš tokios sistemos norime padaryti naują jos versiją, įtraukdami į ją dar vieną klasę *DataOutput*, kurią sudaro du metodai, skirti formatuotam ir neformatuotam duomenų išvedimui į ekraną. Patobulinę klasę *Calculus* nauju funkcionalumu, turėsime tokius klasių aprašus (žr. toliau).

```
class Calculus
{
    int lastUsedNumber1 = 0;
    int lastUsedNumber2 = 0;
    int lastFoundResult = 0;

    DataOutput out;

    public Calculus() { out = new DataOutput() };
    public int add (int num1, int num2)
    {
        number1         = num1;
        number2         = num2;
        lastFoundResult = num1 + num2;

        out.formatedPrint (String (lastFoundResult), “framed”);

        return lastFoundResult;
    }
}

class DataOutput
{
    public DataOutput() {};
    public void simplePrint (String s) { System.out.println (s); }
    public void formatedPrint (String s, String type)
    {
       if (type == “framed”)
       {
          for (int i = 0; i < (s.length() – 1) + 4; i++)
             System.out.print (“-”);
          System.out.println ();
          System.out.println (“| ” + s + “ |”);
          for (int i = 0; i < (s.length() – 1) + 4; i++)
             System.out.print (“-”);
          System.out.println ();
       }
       else simplePrint (s);
    }
}
```

Kaip matome, papildžius senąją sistemos versiją vienu komponentu, padaugėjo ir naujų klasėse galinčių kilti nepageidaujamų situacijų. Identifikuokime jas taip: *object_creation_error*, *string_line_too_long_error*, *string_line_empty_error*.

Turint šiuos duomenis, jau galime rasti ir planinio koeficiento išraišką, kuri yra gaunama imant senosios ir naujosios sistemos versijų turimų duomenų santykį. Šis leidžia teoriškai įvertinti, kurioje sistemos versijoje yra likę mažiau nežinomų klaidų.

Tegul iš viso numatytų ir nenumatytų klaidų senojoje sistemos versijoje yra $TE_B$. Kadangi joje buvo numatytos 2 klaidos, tai nenumatytų liko $(TE_B - 2)$.

Tegul iš viso numatytų ir nenumatytų klaidų naujojoje sistemos versijoje yra $TE_B + \Delta TE = TE_A$. Kadangi joje be turimų 2 dar buvo numatytos 3 klaidos, tai iš viso numatytų klaidų dabar yra 5, o nenumatytų liko $(TE_A - 5)$. Taigi planinio koeficiento išraiška bus tokia:

$$PC = \frac{TE_B - 2}{TE_A - 5} = \frac{TE_B - 2}{TE_B + \Delta TE - 2 - 3} = \frac{TE_B - 2}{(TE_B - 2) + (\Delta TE - 3)}.$$

Kaip matome, pagrindinis narys, lemiantis santykio reikšmės didumą yra $(\Delta TE - 3)$. Bendru atveju kuo didesnė ir sudėtingesnė sistema, tuo yra labiau tikėtina, kad nenumatytų klaidų skaičius joje bus didesnis. Tas pats ir šiuo atveju – abi sistemos versijos yra mažos, visi potencialūs klaidų židiniai yra aiškiai matomi, todėl tikėtina, kad klaidų atsiradimą daugiausia šiuo atveju gali įtakoti tik vidinės standartinių komandų vykdymo arba sistemos palaikymo operacinės aplinkos trikiai. *Šis koeficientas daugiausia yra skirtas lyginti bent kelias naujas tos pačios sistemos versijas.*

#### 5.2.2.2 *Tikrinimo vertinimo koeficientas*

Tikrinimo aspektas yra skirtas atsakyti į klausimą, kiek naujoji programų sistemos versija klaidų atžvilgiu yra budresnė nei senoji. Kitaip tariant, nors bet kokie papildomi patikrinimai kode dažniausiai neigiamai atsiliepia sistemos darbo našumui ir realizacijos apimčiai, vis dėlto klaidų atžvilgiu šiokių tokių privalumų egzistuoja. Dėl to čia yra remiamasi principu: kuo daugiau yra tikrinama, tuo didesnė tikimybė, kad laiku ir vietoje klaida bus aptikta. Su šiuo aspektu susijęs kiekybinis matas yra vadinamas tikrinimo vertinimo koeficientu (žymimas *EC* („*Evaluation Coeficient*“)), kuris leidžia nustatyti, kiek kiekvienoje programų sistemos versijoje klaidų tikrinimo apimtis skiriasi nuo pilno tikrinimo plano ir kurioje versijoje tai yra ryškiau.

Pilno tikrinimo planas („*Full Inspection Plan*“) nenusako tam reikalingų priemonių ar kokių konkrečių realizacijos būdų. Jis tik įvardina visus esminius akcentus, kurie galėtų būti tikrinami. Be abejo, tokių pilno tikrinimo planų galima sudaryti ne vieną, tačiau šiuo atveju remsiuosi tokiu jo variantu:

1. Sistemoje turi būti tikrinami visi jos sudedamieji komponentai.
2. Komponente turi būti tikrinami kiekvienos operacijos įeities duomenys (parametrų reikšmės).
3. Komponente turi būti tikrinami kiekvienos operacijos grąžinami rezultatai.
4. Komponente turi būti tikrinami kiekvienos operacijos invariantai (operacijai galiojančios sąlygos) kiekviename operacijų vykdymo žingsnyje.

Turint tokią schemą, prieš tolimesnę analizę reikia nustatyti, kokias operacijas minimi patikrinimai gali apimti. Šiuo atveju tai gali būti klasių metodai ir standartinių bibliotekų naudojamos funkcijos.

Tarkime, kad programų sistemos komponentą $C_1$ sudaro $M_1$ metodai, komponentą $C_2$ – $M_2$ metodai ir t.t. Be to, analogiškai tarkime, kad metodą $M_1$ sudaro $O_1$ operacijos, metodą $M_2$ – $O_2$ operacijos ir t.t. Dabar įveskime sąvoką „tikrinimo taškai“ („*evaluation points (EVP)*“), kuri nusakytų, kad konkrečioje kodo vietoje tiesiogiai arba netiesiogiai yra naudojama arba galioja kažkokia tikrinimo operacija. Tegul operacijos invariantų tikrinimas bus žymimas $EVP^{INV}$, įeities duomenų tikrinimas – $EVP^{PRE}$ (nes paprastai tai yra atliekama prieš operacijos iškvietimą), o grąžinamo rezultato tikrinimas – $EVP^{POST}$ (nes tai yra atliekama po operacijos iškvietimo). Tada, jei pirmojo sistemos komponento $C_1$ pirmąjį metodą $M_{11}$ sudaro operacijos $O_{11}, O_{12}, \ldots, O_{1s}$, tai iliustruojant pilno tikrinimo planą tenkinančią realizaciją, metodo $M_{11}$ turinys turėtų atrodyti maždaug taip:

$$EVP^{INV}, EVP_{11}^{PRE}, O_{11}, EVP_{11}^{POST}, EVP^{INV}, EVP_{12}^{PRE}, O_{12},$$
$$EVP_{12}^{POST}, EVP^{INV}, \ldots, EVP_{1s}^{PRE}, O_{1s}, EVP_{1s}^{POST}, EVP^{INV}$$

Iš čia aiškiai matyti, kad 1 operacijai tenka 3 tikrinimai. Vadinasi, visoms metodo *s* operacijoms reikės $3s+1$ tikrinimų. Turint šiuos duomenis, jau galima sudaryti ir galutinę tikrinimo vertinimo koeficiento išraišką:

$$EC = \frac{EVR_B}{EVR_A},$$

$$EVR = \frac{\sum_{i=1}^{N}\sum_{j=1}^{M_k} EVOPC(i,j)}{\sum_{i=1}^{N}\sum_{j=1}^{M_k} 3 * SOPC(i,j) + 1}.$$

*EVR* („*Evaluation Ratio*“) yra žymuo, nusakantis atitinkamą (trupmenos skaitiklyje arba vardiklyje) esančią išraišką, $EVOPC(i,j)$ („*Evaluation Operations Coverage*“) yra funkcija, grąžinanti *i*-ojo komponento *j*-ajame metode esančių tikrinimo operacijų skaičių, $SOPC(i,j)$

(„*Selected Operations Coverage*“) yra funkcija, grąžinanti visų įmanomų (pagal pilno tikrinimo planą) *i*-ojo komponento *j*-ajame metode galimų tikrinimo operacijų skaičių.

Bendru atveju pateiktosios *EVR* išraiškos vertinimo tendencija būtų tokia: kuo santykio reikšmė didesnė, tuo analizuojamoje sistemoje yra numatyta daugiau tikrinimo operacijų, skirtų aktualioms probleminėms sritims tikrinti, ir tuo yra labiau tikėtina, kad kilusios klaidos bus laiku ir vietoje aptiktos ir apdorotos.

Pateiktosios formulės taikymą iliustruosiu konkrečia situacija (žr. 8 pavyzdį). Analogišku būdu ši formulė yra taikoma ir skyrelyje 5.2.2.5 „*Atlikti skaičiavimai*“.

**8 pavyzdys.** Tikrinimo vertinimo koeficiento reikšmę skaičiuojančios formulės taikymą iliustruojanti situacija.

Imkime tas pačias 7 pavyzdyje aprašytąsias dvi sistemėles. Iš pradžių susitarkime, koks šiuo atveju galėtų būti pilnas tikrinimo planas, t.y. nustatykime, kas tose klasėse geriausiu atveju galėtų būti tikrinama. Tai galėtų būti, pavyzdžiui, visi metodai ir standartinės operacijos. Iš viso operacijos būtų tokios: *add(), Calculus(), DataOutput(), formatedPrint(), print(), println(), length()* ir *simplePrint()*. Galime dar patikslinti: kiekvienos operacijos atveju galėtų būti tikrinamas jų argumentų, grąžinamos reikšmės (rezultato) teisingumas ir operacijos invariantų tenkinimas. Tegul tokios tikrinimo vietos kode atitinkamai būna žymimos $EVP^{PRE}$, $EVP^{POST}$ ir $EVP^{INV}$. Siekiant paprastumo, tarkime, kad kiekviena *EVP* atitinka vieną kokią nors tikrinimo operaciją. Tuomet pagal pilną (idealistinį) tikrinimo planą kodas atrodytų taip:

<u>*Senoji sistema*</u>:

```
class Calculus
{
    int lastUsedNumber1 = 0;
    int lastUsedNumber2 = 0;
    int lastFoundResult = 0;

    public Calculus() {};
    public int add (int num1, int num2)
    {
        number1         = num1;
        number2         = num2;
        lastFoundResult = num1 + num2;

        return lastFoundResult;
    }
}
```

<u>*Naujoji sistema*</u>:

```
class Calculus
{
    int lastUsedNumber1 = 0;
    int lastUsedNumber2 = 0;
    int lastFoundResult = 0;

    DataOutput out;

    public Calculus()
    {
        EVP^INV
```

```
            EVP^PRE
            out = new DataOutput()
            EVP^POST
            EVP^INV
        };

        public int add (int num1, int num2)
        {
            number1         = num1;
            number2         = num2;
            lastFoundResult = num1 + num2;

            EVP^INV
            EVP^PRE
            out.formatedPrint (String (lastFoundResult), “framed”);
            EVP^POST
            EVP^INV

            return lastFoundResult;
        }
    }

    class DataOutput
    {
        public DataOutput() {};
        public void simplePrint (String s)
        {
            EVP^INV
            EVP^PRE
            System.out.println (s);
            EVP^POST
            EVP^INV
        }

        public void formatedPrint (String s, String type)
        {
           if (type == “framed”)
           {
              EVP^INV
              EVP^PRE
              int s_length = s.length();
              EVP^POST
              EVP^INV
              for (int i = 0; i < (s_length – 1) + 4; i++)
              {
                 EVP^INV
                 EVP^PRE
                 System.out.print (“-”);
                 EVP^POST
                 EVP^INV
              }
              EVP^INV
              EVP^PRE
              System.out.println ();
              EVP^POST
```

```
                EVP^INV
                EVP^PRE
                System.out.println ("| " + s + " |");
                EVP^POST
                for (int i = 0; i < (s_length – 1) + 4; i++)
                {
                EVP^INV
                EVP^PRE
                   System.out.print ("-");
                EVP^POST
                EVP^INV
                }
                EVP^INV
                EVP^PRE
                System.out.println ();
                EVP^POST
                EVP^INV
            }
            else
            {
                 EVP^INV
                 EVP^PRE
                 simplePrint (s);
                 EVP^POST
                 EVP^INV
            }
        }
    }
```

Iš kodo matyti, kad kiekvienai metode esančiai tikrinamai operacijai tenka po 3 *EVP* tikrinimo operacijas. Iš čia seka, kad jei metode yra s tikrinamų operacijų, tai iš viso *EVP* tikrinimo operacijų bus 3s + 1 (čia vienetą atitinka viena $EVP^{INV}$, einanti metodo pradžioje).

Dabar galime paskaičiuoti, kiek klasių metoduose iš tikrųjų yra tikrinimo operacijų, lyginant su tikrinimo operacijų skaičiumi, gautu pagal pilną tikrinimo planą. Tegul šis santykis senojoje sistemoje būna žymimas $EVR_B$ ir jis bus lygus:

$$EVR_B = \frac{0}{3*0+1} = 0.$$

Skaitiklyje ir vardiklyje yra reikšmės nulis, nes *Calculus()* ir *add()* metoduose nėra ne tik jokių tikrinimo, bet ir tikrinamų operacijų. Tegul analogiškas santykis naujojoje sistemoje būna žymimas $EVR_A$. Tuomet jis bus lygus:

$$EVR_A = \frac{0+0}{\left[(3*1+1)+(3*1+1)\right]+\left[(3*1+1)+(3*7+1)\right]} = 0$$

Pirmosiose laužtiniuose skliaustuose yra pateikti klasės *Calculus* dėmenys, o antruosiuose – klasės *DataOutput* dėmenys.

Tikrinimo vertinimo koeficientas yra skaičiuojamas, imant santykį tarp senosios sistemos $EVR_B$ ir naujosios sistemos $EVR_A$. O kadangi abi šios reikšmės yra lygios nuliui, tai ir tikrinimo vertinimo koeficientas *EC* bus lygus nuliui.

Bendru atveju tikrinimo vertinimo koeficiento reikšmė pasako, kiek kartų senojoje sistemoje realus atitikimas pilnam tikrinimo planui yra mažesnis negu naujosios. Ką tai reiškia? Ogi tik tai, kad kuo daugiau yra sistemoje atliekama nustatytiems šaltiniams tikrinimų, tuo yra labiau tikėtina, kad sistema yra labiau pasiruošusi nenumatytiems atvejams ir galimoms klaidoms. Taigi šiuo atveju abi sistemos yra vienodo pasiruošimo klaidos laipsnio be jokių klaidas galinčių aptikti tikrinimo operacijų. *Šis koeficientas yra taip pat skirtas lyginti bent kelias naujas tos pačios sistemos versijas.*

#### 5.2.2.3 *Pajėgumo koeficientas*

Pajėgumo aspektas yra skirtas atsakyti į klausimą, kiek naujoji programų sistemos versija yra labiau pajėgi darbo metu pakelti klaidų apgultį nei senoji. Čia yra remiamasi tokiu principu: kuo didesnį spaudimą sistema gali pakelti, tuo ji yra labiau atsparesnė. Su šiuo aspektu susijęs kiekybinis matas yra vadinamas pajėgumo koeficientu (žymimas *CC* („*Capability Coeficient*“)), kuris leidžia nustatyti, kiek naujojoje programų sistemos versijoje klaidų apgulties metu gali išlikti aktyvių komponentų daugiau negu senojoje ir kurioje versijoje tai yra ryškiau.

Tokio pobūdžio vertinimas yra susijęs su vartotojo užduotimis. Kitaip tariant, čia yra bandoma žvelgti iš vartotojo perspektyvos, nes jam yra svarbu tik atlikti numatytas savo užduotis visai nesigilinant į tai, ar tokio proceso metu sistemoje funkcionuoja visi komponentai, ar tik jų dalis. Dėl to tarkime, kad tiriamojoje sistemoje vartotojo užduočių aibė yra $T = \{T_1, T_2, \ldots, T_L\}$. Tada galima suformuluoti tokį teiginį:

$$\forall T_i \left(i \in \{1, \ldots, L\}\right) \exists S(T_i) = \{C_{t_1}, C_{t_2}, \ldots, C_{t_h}\} : USER \rightarrow\!\!\oplus\ T_i (t_j \in \{1, \ldots, N\}).$$

Šis teiginys sako, kad su kiekviena vartotojo užduotimi $T_i$ gali būti susieta atitinkama sistemos komponentų, reikalingų tai užduočiai atlikti, aibė $S(T_i)$ ($S(T_i) \subset \{C_1, C_2, \ldots, C_N\}$). Žymuo *USER* yra vartotojo rolę atspindinti esybė. Simbolis „$\rightarrow\!\!\oplus$“ reiškia gebėjimą / galėjimą atlikti / įvykdyti / susidoroti. Atitinkamas ženklas „$\rightarrow\!\!\ominus$“ reiškia pastarojo priešingybę.

Toliau pateikiu bendrą eigą, kuria turėtų vykti tolimesnis vertinimo pagal pajėgumo aspektą procesas:

1. Sudaryti vartotojo užduočių sąrašą.
2. Nustatyti kiekvienos vartotojo užduoties komponentų aibę.

3. Sudaryti mažiausios rizikos komponentų aibę $S_{MRKA}(T)$, vidutinės rizikos komponentų aibę $S_{VRKA}(T)$ ir didžiausios rizikos komponentų aibę $S_{DRKA}(T)$.
4. Kiekvienai užduočiai nustatyti funkcinių vertinimo kriterijų aibę $CR(T)$.
5. Atlikti globalų ir / arba lokalų sistemos operacinio darbo pajėgumo klaidų apgulties metu įvertinimą.

Didžiausios rizikos komponentų aibė (DRKA) nusako sistemos komponentus, kurie dalyvauja visuose vartotojo arba daugiau nei viename užduočių atlikimo procesuose. Formaliai DRKA aibę galima būtų išreikšti taip:

$$S_{DRKA}(T) = S(T_1) \cap S(T_2) \cap \ldots \cap S(T_L).$$

Jei $S_{DRKA}(T) \neq \phi$, tuomet reikia pasirinkti tokį $i$ ($i \in \{1, \ldots, L\}$), kad

$$S_{DRKA}(T \setminus \{T_i\}) = S(T_1) \cap S(T_2) \cap \ldots \cap S(T_{i-1}) \cap S(T_{i+1}) \cap \ldots \cap S(T_L) \neq \phi .$$

Jei ir tokiu atveju bus gauta tuščia aibė, tai šį veiksmą reikia kartoti tol, kol bus gauta netuščia aibė. Jei netuščia aibė yra gaunama su mažesniu nei 50% visų sistemos komponentų kiekiu, tai didžiausios rizikos komponentų aibės taikyti negalima.

Vidutinės rizikos komponentų aibė (VRKA) nusako sistemos komponentus, iš kurių kiekvienas dalyvauja vienos ir tik vienos kurios nors vartotojo užduoties atlikimo procese. Formaliai VRKA aibę galima būtų išreikšti taip:

$$\begin{aligned} S_{VRKA}(T) = {} & (S(T_1) \setminus (S(T_2) \cup S(T_3) \cup \ldots \cup S(T_L))) \cup \\ & (S(T_2) \setminus (S(T_1) \cup S(T_3) \cup \ldots \cup S(T_L))) \cup \\ & \ldots \\ & (S(T_L) \setminus (S(T_1) \cup S(T_2) \cup \ldots \cup S(T_{L-1}))) \end{aligned}$$

Mažiausios rizikos komponentų aibė (MRKA) nusako sistemos komponentus, kurie neįeina nei į DRKA, nei į VRKA. Kitaip tariant, jie dalyvauja maksimum keleto vartotojų užduočių atlikimo procesuose ir yra naudojami tik kaip pagalbiniai komponentai (pavyzdžiui, tekstinių pranešimų formatavimo komponentai ar pan.), kurių funkcionalumui sutrikus, užduotis vis tiek gali būti atlikta.

$$S_{MRKA}(T) = S(T) \setminus (S_{DRKA}(T) \cup S_{VRKA}(T)).$$

Funkciniai vartotojo užduoties vertinimo kriterijai – tai aspektai, kurie leidžia identifikuoti visas pagrindines skirtingas vartotojo užduoties funkcionalumo grupes. Sudarant tokių funkcinių vartotojo užduoties vertinimo kriterijų sąrašą, reikia pasirinkti ir / arba detalizuoti kelis (bet ne daugiau 5) galimus kriterijus iš šio sąrašo:

1. Įvedimas (pavyzdžiui, duomenų iš klaviatūros, failo, DB, laikmenos ir pan.).
2. Išvedimas (pavyzdžiui, duomenų į ekraną, failą, DB, laikmeną ir pan.).
3. Užduoties funkcinė kategorija.

Nurodant užduoties funkcinę kategoriją, reikia pasirinkti ir / arba detalizuoti iki 3 galimų aspektų iš sąrašo:

1. Duomenų priėmimas / perdavimas tinklu.
2. Sudėtingi procesiniai / išskirstyti skaičiavimai.
3. Darbas su DB.
4. Duomenų mainai.
5. Duomenų apdorojimas (pavyzdžiui, transformacijos, savybių nustatymai ir pan.).
6. Kitas funkcionalumas.

Pasirinkimų kiekis yra ribojamas dėl to, kad užduotys programų sistemose dažnai yra dekomponuotos į nedidelius, lengvai valdomus ir logiškai išgrynintus „paketėlius“. Dėl to, jei skirtingoms užduoties funkcionalumo grupėms identifikuoti reikia daugiau nei 5 aspektų, tai gali reikšti vieną iš dviejų dalykų: arba įvardinta užduotis yra per daug bendrinė ir turi smulkesnių ją sudarančių použduočių, arba programų sistema yra neefektyviai suprojektuota.

Na, o pats vertinimo procesas susiveda į galutinės pajėgumo koeficiento reikšmės paskaičiavimą, kuris gali būti atliktas, remiantis šiomis išraiškomis:

$$CC = \frac{OPUS_B}{OPUS_A},$$

$$OPUS = \sum_{i=1}^{L} \left( \sum_{m=1}^{|S_{MRS}(T)|} crit(i,m) + \sum_{v=1}^{|S_{VRS}(T)|} crit(i,v) + \sum_{d=1}^{|S_{DRS}(T)|} crit(i,d) \right).$$

*OPUS* („*Operation Under Siege*“) yra žymuo, nusakantis, kiek iš viso funkcinių vertinimo kriterijų yra tenkinama kiekvienos vartotojo užduoties atžvilgiu, $crit(i, j)$ yra funkcija, grąžinanti *i*-osios vartotojo užduoties funkcinių vertinimo kriterijų skaičių, kuris yra nustatomas pagal tai, kiek kriterijų yra tenkinama, jei atitinkamos rizikos komponentų aibėje esantis *j*-asis komponentas tampa neveiksnus.

Bendru atveju pateiktosios *OPUS* išraiškos vertinimo tendencija būtų tokia: kuo reiškinio reikšmė didesnė (lyginant su 0), tuo didesnė tikimybė, kad vartotojo užduotis analizuojamoje sistemoje bus galima atlikti net ir klaidų atveju.

Pateiktosios formulės taikymą iliustruosiu konkrečia situacija (žr. 9 pavyzdį). Analogišku būdu ši formulė yra taikoma ir skyrelyje 5.2.2.5 „*Atlikti skaičiavimai*“.

**9 pavyzdys.** Pajėgumo koeficiento reikšmę skaičiuojančios formulės taikymą iliustruojanti situacija.

Imkime tas pačias 7 pavyzdyje aprašytąsias dvi sistemėles. Pagal pajėgumo koeficiento skaičiavimo eigos planą, iš pradžių reikia sudaryti vartotojo užduočių sąrašą *T*, kuris šiuo atveju bus toks: $T = \{Add\_Two\_Integer\_Numbers, Output\_Data\_To\_Screen\}$, t.y. turime dvi vartotojo užduotis (dviejų sveikųjų skaičių sumavimą ir duomenų išvedimą į ekraną).

Toliau reikia nustatyti, kokios sistemos klasės yra susijusios su identifikuotų vartotojo užduočių realizavimu. Nagrinėjamu atveju kiekvienai užduočiai turėsime po vieną klasę:

$$S(Add\_Two\_Integer\_Numbers) = \{Calculus\},$$
$$S(Output\_Data\_To\_Screen) = \{DataOutput\}.$$

Toliau reikia sudaryti mažiausios, vidutinės ir didžiausios rizikos komponentų aibes. Didžiausios rizikos komponentų aibė (*DRKA*) nusako klases, kurios dalyvauja visų vartotojo užduočių atlikime (t.y. bent vienoje tokioje klasėje kyla klaida ir jos funkcionalumas sutrinka, tai tikėtina, kad nei vieną vartotojo užduotį nebebus galima įvykdyti). Vidutinės rizikos komponentų aibė (*VRKA*) nusako klases, iš kurių kiekviena dalyvauja vienos ir tik vienos kurios nors užduoties atlikime (t.y. jei tokioje klasėje kyla klaida ir jos funkcionalumas sutrinka, tai tikėtina, kad tik kurios nors vienos užduoties nebebus galima įvykdyti). Mažiausios rizikos komponentų aibė (*MRKA*) nusako klases, kurios neįeina nei į *DRKA*, nei į *VRKA* (t.y. jei tokioje klasėje kyla klaida ir jos funkcionalumas sutrinka, tai geriausiu atveju tikėtina, kad visas užduotis bus galima įvykdyti, tik galbūt bus duodami kokie nors neteisingi rezultatai arba pasireikš kiti netikslumai, tačiau sistema iš esmės veiks). Taigi šiuo atveju užduočių komponentų aibės bus tokios:

$$S_{DRKA}(T) = S(Add\_Two\_Integer\_Numbers) \cap S(Output\_Data\_To\_Screen) = \phi$$

$$S_{VRKA}(T) = (S(Add\_Two\_Integer\_Numbers) \setminus S(Output\_Data\_To\_Screen)) \cup$$
$$(S(Output\_Data\_To\_Screen) \setminus S(Add\_Two\_Integer\_Numbers)) =$$
$$= \{Calculus, DataOutput\}$$

$$S_{MRKA}(T) = S(T) \setminus (S_{DRKA}(T) \cup S_{VRKA}(T)) = \phi$$

Taigi turime tik vidutinės rizikos komponentų aibės atvejį (t.y. be klasės *Calculus* neveiks sumavimas, o be klasės *DataOutput* – duomenų išvedimas į ekraną).

Toliau kiekvienai užduočiai reikia nustatyti funkcinių vertinimo kriterijų aibę $CR(T)$. Tai reiškia, kad reikia nustatyti, kokios yra pagrindinės skirtingos tų užduočių funkcinės grupės. Jų sąrašas yra pateiktas prie pajėgumo koeficiento teorinės aprašymo dalies, o šiuo atveju turimų užduočių funkciniai vertinimo kriterijai būtų tokie:

$$CR(Add\_Two\_Integer\_Numbers) = \{Duomenu\_Apdorojimas\}$$

$$CR(Output\_Data\_To\_Screen) = \{Isve\,\mathrm{dim}\,as\}$$

Dabar galima teoriškai įvertinti, kiek kiekvienos užduoties atveju bus tenkinama kriterijų, jei su tos užduoties atlikimu susijusi jos klasė netikėtai taptų neveiksni. Tegul tokių kriterijų skaičių žymi funkcija *crit()*. Šios funkcijos reikšmę reikia paskaičiuoti atskirai senajai ir atskirai naujajai sistemos versijai visų rizikų komponentų aibių atžvilgiu. Kadangi šiame pavyzdyje netuščią turime tik vidutinės rizikos komponentų aibę, tai jos atžvilgiu ir bus atliekamas įvertinimas.

Jei tiek senojoje, tiek ir naujojoje sistemoje klasė *Calculus* taptų neveiksni (t.y. nebegalėtų atlikti savo funkcijų), tuomet abiem atvejais pirmosios užduoties funkcinis vertinimo kriterijus *Duomenu_Apdorojimas* nebūtų tenkinamas, nes klasės metodas *add()* būtų neveiksnus, o sistemose nėra aprašyta jokių priemonių, kurios galėtų padaryti jį veiksniu. Dėl to abiem atvejais funkcijos *crit()* reikšmė yra lygi 0.

Analogiškai yra ir su antrąja užduotimi, t.y. jei klasė *DataOutput* taptų neveiksni, tuomet tiek senojoje, tiek ir naujojoje sistemoje tos užduoties funkcinis vertinimo kriterijus *Isvedimas* nebūtų tenkinamas. Dėl to funkcijos *crit()* reikšmė vėlgi abiem atvejais yra lygi 0.

Taigi turint visus reikalingus duomenis, galima paskaičiuoti ir galutinę pajėgumo koeficiento reikšmę. Tai yra daroma imant santykį tarp funkcijos *crit()* reikšmių sumos senosios sistemos atžvilgiu ir funkcijos *crit()* reikšmių sumos naujosios sistemos atžvilgiu. Suma yra skaičiuojama, imant visų rizikų komponentų aibių funkcijos *crit()* reikšmes. Ši suma senosios sistemos atžvilgiu yra žymima $OPUS_B$, o naujosios – $OPUS_A$. Taigi iš turimų duomenų matyti, kad

$$OPUS_B = 0$$
$$OPUS_A = 0$$

O iš čia seka, kad ir galutinė pajėgumo koeficiento *CC* reikšmė bus lygi 0. Tai parodo, kad abiejose sistemose nėra jokių priemonių ar kitų sugalvotų būdų, kurie klaidų atveju padėtų jas pašalinti arba bent jau padaryti taip, kad sistema kiek įmanoma išliktų veiksmingesnė.

Bendru atveju pajėgumo koeficiento santykio reikšmė parodo, kiek kartų senosios sistemos sugebėjimas klaidų atveju išlikti veiksmingai yra mažesnis negu naujosios. *Šis koeficientas yra taip pat skirtas lyginti bent kelias naujas tos pačios sistemos versijas.*

#### 5.2.2.4 *Pakankamumo koeficientas*

Pakankamumo aspektas yra skirtas atsakyti į klausimą, kiek pakankamas yra naujosios programų sistemos versijos į klaidų valdymą orientuotas funkcionalumas negu senosios. Čia yra remiamasi principu: iš klaidų mokomės. Kitaip tariant, yra laikomasi to, kad kiekviena silpnoji vieta programų sistemoje turi veikti kaip akstinas jos kūrėjus ją pašalinti, o viskas, ko tereikia, tai

tik kuo anksčiau pradėti tokias silpnąsias vietas identifikuoti ir atitinkamai įvertinti. Su šiuo aspektu susijęs kiekybinis matas yra vadinamas pakankamumo koeficientu (žymimas *OC* („*Opportunity Coeficient*“)), kuris leidžia nustatyti, kiek naujojoje programų sistemos versijoje yra į klaidų valdymą orientuotų priemonių, kurios yra stipresnės negu tos, kurios yra senojoje programų sistemos versijoje. Kitaip tariant, pakankamumo aspektas yra orientuotas būtent į su klaidų valdymu susijusį funkcionalumą.

Kas yra į klaidų valdymą orientuota priemonė? Bendrąja prasme tai yra tam tikras funkcionalumas (pavyzdžiui, suformuluotas atitinkamų užduočių terminais ar pan.), nusakomas keturnariu $D = \langle AR, FR, GR, GE \rangle$, kurioje *AR* reiškia architektūrinę priemonės realizaciją sistemoje, *FR* – funkcinę priemonės realizaciją sistemoje, *GR* – algoritminę priemonės realizaciją sistemoje, o *GE* („*Global Extent*“) – globalų priemonės panaudojimo sistemoje mastą.

Architektūrinė realizacija – tai struktūriniai sistemos architektūros vienetai (komponentai, klasės) ir atitinkami ryšiai, skirti konkrečios priemonės sistemoje realizavimui. Pastarieji bendru atveju gali būti dvejopi: tie, kurie tarpusavyje sieja pačius struktūrinius sistemos architektūros vienetus, ir tie, kurie jungia juos su likusia sistemos dalimi.

Funkcinė realizacija – tai struktūriniuose klaidų valdymo priemonei realizuoti skirtuose sistemos architektūros vienetuose numatytų funkcijų karkasai, kuriuos nusako pati funkcija (t.y. jos buvimas), interfeisas ir tos funkcijos prieinamumas kitiems struktūriniams sistemos architektūros vienetams, arba funkcinis pavidalas (pavyzdžiui, numatytos kokios standartinės priemonės).

Algoritminė realizacija – tai konkrečios struktūriniuose klaidų valdymo priemonėms realizuoti skirtuose sistemos architektūros vienetuose esančių funkcijų realizacijos pasirinktu būdu. Jas nusako jų pačių realizacijos buvimas, jų sėkmingam veikimui reikalingi resursai ir jas sudarantys loginiai žingsniai.

Globalus priemonės panaudojimo mastas – tai dydis, kuris nusako, kiek toji priemonė yra plačiai naudojama visos sistemos kontekste. Išreiškiamas santykiu ir skaičiuojamas taip:

$$GE = \frac{R_{ACTUAL}}{R_{TOTAL}}.$$

Čia $R_{TOTAL}$ – kiekis sistemos komponentų, kuriuose nagrinėjamoji priemonė turėtų būti naudojama, siekiant maksimalios naudos, o $R_{ACTUAL}$ – kiekis komponentų, kuriuose nagrinėjamoji priemonė yra iš tikrųjų naudojama.

Taip pat gali būti naudojamas ir lokalus priemonės panaudojimo mastas $LE$ („*Local Extent*“), kuris nurodo, kiek toji priemonė yra plačiai naudojama kiekviename konkrečiame jos tikslinio panaudojimo kontekste funkcinės realizacijos atžvilgiu. Tam tikslui funkcinės realizacijos aspektus reikėtų skaidyti pagal tai, kiek skirtingų konkrečios priemonės funkcinių realizacijų yra naudojama (pavyzdžiui, klaidų aptikimui yra naudojamos standartinis *try-catch* blokas, prieinamas bet kam, ir keletas individualiai rašytų funkcijų specifinėms situacijoms analizuoti). Tai atlikus, lokalus konkrečios priemonės panaudojimo mastas toliau bus skaičiuojamas kiekvienam suskaidytam funkcinės realizacijos aspektui individualiai. Jis taip pat išreiškiamas santykiu ir šiuo atveju galėtų būti skaičiuojamas taip:

$$LE = \sum_{i=1}^{R_{ACTUAL}} \left( \sum_{j=1}^{frac()} \left( \sum_{k=1}^{mec(i)} \frac{uopc(j,i,k)}{copc(i,k)} \right) \right).$$

Čia $frac(\ )$ („*functional realization aspect count*“) yra funkcija, grąžinanti skirtingų funkcinės realizacijos aspektų skaičių, $mec(i)$ („*method count*“) yra funkcija, grąžinanti *i*-ojo komponento metodų skaičių, $copc(i,k)$ („*complete operation count*“) yra funkcija, grąžinanti *i*-ojo komponento *k*-ajame metode esančių operacijų skaičių, o $uopc(j,i,k)$ („*used operation count*“) yra funkcija, grąžinanti skaičių operacijų, kurioms atitinkamas funkcinės realizacijos aspektas yra taikomas. Taigi iš viso kiekvieną konkrečią į klaidų valdymą orientuotą priemonę jau nusakys nebe aprašytasis keturnaris, o penkianaris $D = \langle AR, FR, GR, GE, LE \rangle$.

Vis tik tolimesniuose skaičiavimuose lokalaus priemonės panaudojimo masto nenaudosiu, siekiant kuo mažiau atitolinti būsimus rezultatus nuo realybės kad ir dėl to, jog naujosios analizuojamos sistemos atveju jokia nauja kodo realizacija nėra sukurta, o remtis teoriniais samprotavimais šįkart būtų neprognozuojamai sudėtinga.

Tarkime, kad tiriamojoje programų sistemoje į klaidų valdymą yra orientuota ši priemonių aibė: $I = \{I_1, I_2, \ldots, I_f\}$. Be to, analogiškai tarkime, kad turime klaidų aibę $E = \{e_1, e_2, \ldots, e_g\}$.

Tuomet pakankamai akivaizdu, kad klaida $e_i\left(i \in \{1,\ldots,g\}\right)$ kyla sistemoje tada, kai galioja šie teiginiai:

$$\forall I_j\left(j \in \{1,\ldots,f\}\right) \rightarrow\otimes\, e_i,$$

$$I_j \rightarrow\otimes\, e_i : \exists W = \{w_1, w_2, \ldots, w_v\} \forall w_l \in W\left(l \in \{1,\ldots,v\}\right): SYSTEM \rightarrow\ominus\, w_l.$$

Pirmasis teiginys sako, kad sistemoje kyla klaida tada, kai nei viena iš turimų į klaidų valdymą orientuotų priemonių nesugeba jos pašalinti. Antrasis teiginys sako, kad priemonė nesugeba pašalinti klaidos tada, kai egzistuoja tokia silpnųjų vietų aibė *W*, kad sistema nėra pajėgi tai atlikti (pavyzdžiui, joje trūksta kažkokio funkcionalumo ar pan.). Žymuo *SYSTEM* yra sistemos rolę atspindinti esybė. Simbolis „$\rightarrow\otimes$“ reiškia nesugebėjimą pašalinti / eliminuoti / sutvarkyti. Atitinkamas simbolis „$\rightarrow\odot$“ reiškia pastarojo priešingybę.

Taigi pagrindinė vertinimo proceso idėja yra ta, kad pirmiausia reikia sudaryti į klaidų valdymą orientuotų sistemoje realizuotų priemonių sąrašą, po to sudaryti klaidų sąrašą ir atlikti įvertinimą pagal tai, kiek kiekviena priemonė yra realizuota taip, kad pajėgtų susidoroti su atitinkamomis klaidomis. Tai galima padaryti, remiantis šiomis išraiškomis:

$$OC = \frac{IMPCOV_B}{IMPCOV_A},$$

$$IMPCOV = \sum_{i=1}^{f}\left(\mathrm{cov}(I_i) * GE_{I_i}\right),$$

$$\mathrm{cov}(I) = \frac{reach(AR) + reach(FR) + reach(GR)}{9}.$$

*IMPCOV* („*Implementation Coverage*“) yra žymuo, nusakantis, kiek kiekvienos priemonės turima realizacija yra pajėgi ir tinkama klaidų valdymui, $\mathrm{cov}(I)$ yra funkcija, grąžinanti skaičių, kuris parodo, kiek konkrečios priemonės *I* bendroji realizacija yra pilna, $reach(R)$ yra funkcija, nurodanti, kiek kiekvienos rūšies realizacija (pagal apibrėžtus jos aspektus) yra įgyvendinta.

Bendru atveju pateiktosios *IMPCOV* išraiškos vertinimo tendencija būtų tokia: kuo reiškinio reikšmė didesnė (lyginant su 0), tuo analizuojamoje sistemoje į klaidų valdymą

orientuotos priemonės yra labiau išvystytos ir labiau realizuotos kaip nepriklausomi atitinkamą su klaidų valdymu susijusį funkcionalumą (pavyzdžiui, klaidų aptikimą, apdorojimą ir pan.) teikiantys komponentai.

Pateiktosios formulės taikymą iliustruosiu konkrečia situacija (žr. 10 pavyzdį). Analogišku būdu ši formulė yra taikoma ir skyrelyje 5.2.2.5 „*Atlikti skaičiavimai*“.

**10 pavyzdys.** Pakankamumo koeficiento reikšmę skaičiuojančios formulės taikymą iliustruojanti situacija.

Kadangi pakankamumo koeficientas yra orientuotas senojoje ir naujojoje sistemos versijose esančias į klaidų valdymą orientuotas priemones, o jų nagrinėjamu atveju tiesiog nėra, tai padarykime turimose sistemėlese keletą pakeitimų – senojoje sistemoje klaidų tikrinimui panaudokime standartinius sąlyginius sakinius ir *try-catch* bloką, o naujojoje aprašykime dar vieną klasę *ErrorMessenger*, skirtą atitinkamoms klaidoms pranešti vartotojui.

*Senoji sisema*:

```
class Calculus
{
    int lastUsedNumber1 = 0;
    int lastUsedNumber2 = 0;
    int lastFoundResult = 0;

    public Calculus() {};
    public int add (int num1, int num2)
    {
        try
        {
            number1         = num1;
            number2         = num2;
            lastFoundResult = num1 + num2;
        }
        catch (Exception e)
        {
            System.out.println (“Error occurred during the execution of
                                 operation add()!”);
        }

        return lastFoundResult;
    }
}
```

*Naujoji sistema*:

```
class ErrorMessenger
{
    private String possibleErrors [] =
    {“argument_specification _error”,
     “arithmetic_overflow_error”,
     “object_creation_error”,
     “string_line_empty_error”,
     “string_line_too_long_error”,
     “unknown_error”};

    private int errorOccurenceCount [6];

    public ErrorMessenger(); {}
    public printErrorMessage(int error_type)
    {
        switch (error_type)
```

```
            {
                case 0: System.out.println ("The specified argument value
                                             type or format is incorrect!");
                         break;
                case 1: System.out.println ("Arithmetic operation cannot be
                                             completed because the argument
                                             values are too big!");
                         break;
                case 2: System.out.println ("The object cannot be created
                                             due to errors!");
                         break;
                case 3: System.out.println ("The specified string is
                                             empty!");
                         break;
                case 4: System.out.println ("The specified string is
                                             too long!");
                         break;
            }
            errorOccurenceCount [error_type]++;
        }
    }

    class Calculus
    {
        int lastUsedNumber1 = 0;
        int lastUsedNumber2 = 0;
        int lastFoundResult = 0;

        DataOutput out;
        ErrorMessenger eMess;

        public Calculus()
        {
            try { out = new DataOutput(); }
            catch (NullPointerException e) {}
            { eMess.printErrorMessage (2); }
        }

        public int add (int num1, int num2)
        {
            number1         = num1;
            number2         = num2;
            lastFoundResult = num1 + num2;

            String line_to_print = String (lastFoundResult);

            if (line_to_print == "") eMess.printErrorMessage (3);
            else if (line_to_print.length() > 76)
                     eMess.printErrorMessage (4);

            try { out.formatedPrint (line_to_print, "framed"); }
            catch (Exception e) { eMess.printErrorMessage (5); }

            return lastFoundResult;
        }
    }

    class DataOutput
    {
        public DataOutput() {};
        public void simplePrint (String s)
        {
            System.out.println (s);
        }
```

```
        public void formatedPrint (String s, String type)
        {
           if (type == “framed”)
           {
              int s_length = s.length();
              for (int i = 0; i < (s_length – 1) + 4; i++)
                 System.out.print (“-”);
              System.out.println ();
              System.out.println (“| ” + s + “ |”);
              for (int i = 0; i < (s_length – 1) + 4; i++)
                 System.out.print (“-”);
              System.out.println ();
           }
           else simplePrint (s);
        }
     }
```

Turėdami šiuos pakeitimus, dabar jau galime įvertinti pakankamumo koeficiento reikšmę. Pirmiausia reikia sudaryti senojoje ir naujojoje sistemoje į klaidų valdymą orientuotų priemonių aibę. Priemonių pavadinimai bendru atveju gali sutapti su klaidų valdymo modelio probleminių sričių pavadinimais. Taigi tegul priemonių aibė senojoje sistemoje bus žymima $I_B$, o naujojoje – $I_A$. Tuomet į klaidų valdymą orientuotos priemonės bus tokios:

$$I_B = \begin{Bmatrix} Klaidu\_Aptikimas, \\ \Pr anesimu\_Generavimas \end{Bmatrix}$$

$$I_A = \begin{Bmatrix} Klaidu\_Identifikavimas, \\ Klaidu\_Aptikimas, \\ Klaidu\_\mathrm{Re}gistravimas, \\ \Pr anesimu\_Generavimas \end{Bmatrix}$$

Dabar kiekvieną priemonę reikia detalizuoti pagal jos apibrėžimą ( $D = \langle AR, FR, GR, GE \rangle$ ) arba, kitaip tariant, nustatyti, kiek kriterijų bus tenkinama kiekvienos realizacijos (*AR*, *FR* ir *GR*) atžvilgiu bei paskaičiuoti *GE* reikšmę. Kiekvieno tipo realizacija turi po tris ją glaustai apibūdinančius aspektus, kurie yra išvardinti teorinėje pakankamumo koeficiento dalyje.

GE reikšmė nusako, kiek kartų klaidų valdymo priemonė yra naudojama sistemoje mažesniu mastu negu turėtų būti naudojama iš tikrųjų. Santykyje bus imamas skaičius klasių, kuriose priemonė turėtų būti naudojama (vardiklis) ir skaičius klasių, kuriose priemonė yra realiai naudojama (skaitiklis).

Prie kiekvienos priemonės aprašymo iškart bus apskaičiuojama ir funkcijos *cov()* bei reiškinio *(cov()*GE)* reikšmės. Funkcijos *cov()* reikšmė nusako, kiek kartų priemonė yra mažiau realizuota kaip nepriklausomas vienetas negu iš tikrųjų galėtų tokia būti. Santykio skaitiklyje yra tiesiog susumuojama, kiek yra tenkinama kiekvienos realizacijos aspektų, o vardiklyje esantis skaičius 9 yra maksimalus tokių aspektų, kuriuos priemonė galėtų tenkinti, kiekis. Reiškinio *(cov()*GE)* reikšmė nusako priemonės išvystymo lygį visame per jos globalaus panaudojimo mastą.

<u>*Senoji sistema*</u>:

Klaidų aptikimo priemonės architektūrinės realizacijos senojoje sistemoje nėra (nėra tam skirtų klasių, nėra tas klases jungiančių (vidinių) ryšių ir nėra (išorinių) ryšių, jungiančių tas klases su kitomis sistemos klasėmis). Yra naudojama standartinė funkcinė realizacija – *try-catch* blokas. Jokių tam dedikuotų (klaidų aptikimą realizuojančių) metodų nėra, interfeisų (pavyzdžiui, abstrakčių metodų) nėra. Jokių algoritminių priemonės realizacijų nėra, dedikuotų tam naudojamų resursų nėra, loginės klaidų aptikimo schemos nėra.

$$GE = \frac{1}{1} = 1,$$

$$\text{cov}(Klaidu\_Aptikimas) = \frac{0+1+0}{9} = 0{,}1111,$$

$$(\text{cov}(Klaidu\_Aptikimas) * GE) = 0{,}1111.$$

Pranešimų generavimo priemonės architektūrinės realizacijos nėra. Naudojama standartinė pranešimų išvedimo į ekraną funkcija *System.out.println()*. Dedikuotų metodų nėra, interfeisų nėra. Algoritminės priemonės realizacijos nėra, dedikuotų naudojamų resursų nėra, loginės klaidų aptikimo schemos nėra.

$$GE = \frac{1}{1} = 1,$$

$$\text{cov}(\text{Pr}anesimu\_Generavimas) = \frac{0+1+0}{9} = 0{,}1111,$$

$$(\text{cov}(\text{Pr}anesimu\_Generavimas) * GE) = 0{,}1111.$$

<u>*Naujoji sistema*</u>:

Klaidų identifikavimas yra atliekamas atskiroje *ErrorMessenger* klasėje. Vidinių struktūrinių ryšių nėra. Yra išorinis vienas ryšys, jungiantis klasę *ErrorMessenger* su klase *Calculus*. Naudojamų standartinių klaidų identifikavimo funkcijų nėra, dedikuotų naudojamų metodų nėra (identifikavimas yra atliekamas pranešimų generavimui skirtame metode), interfeisų nėra. Yra algoritminė realizacija (apjungta kartu su klaidų pranešimų parinkimu), dedikuotų resursų nėra, yra naudojama primityvi loginė klaidų aptikimo schema (parinkimas pagal nurodytą indekso reikšmę).

$$GE = \frac{1}{3} = 0{,}3333,$$

$$\text{cov}(Klaidu\_Identifikavimas) = \frac{2+0+2}{9} = 0{,}4444,$$

$$(\text{cov}(Klaidu\_Identifikavimas) * GE) = 0{,}1481.$$

Klaidų aptikimas yra atliekamas atskiroje *ErrorMessenger* klasėje. Vidinių struktūrinių ryšių nėra. Yra išorinis vienas ryšys, jungiantis klasę *ErrorMessenger* su klase *Calculus*. Yra naudojama standartinė funkcinė realizacija – *try-catch* blokas ir sąlyginiai sakiniai. Dedikuotų naudojamų metodų nėra, interfeisų nėra. Aiškios algoritminės realizacijos nėra, dedikuotų naudojamų resursų nėra, loginės klaidų aptikimo schemos nėra.

$$GE = \frac{1}{3} = 0{,}3333,$$

$$\text{cov}(Klaidu\_Aptikimas) = \frac{2+1+0}{9} = 0{,}3333,$$

$$(\text{cov}(Klaidu\_Aptikimas) * GE) = 0{,}1111.$$

Klaidų registravimas yra atliekamas atskiroje *ErrorMessenger* klasėje. Vidinių struktūrinių ryšių nėra. Yra išorinis vienas ryšys, jungiantis klasę *ErrorMessenger* su klase *Calculus*. Naudojamų standartinių funkcijų nėra, dedikuotų naudojamų metodų nėra, interfeisų nėra. Yra algoritminė realizacija (naudojamas masyvas, kurio kiekvienas elementas nusako atitinkamos klaidos atsiradimų skaičių), dedikuotų naudojamų resursų nėra, loginės schemos nėra.

$$GE = \frac{1}{3} = 0{,}3333,$$

$$\text{cov}(Klaidu\_\text{Re}gistravimas) = \frac{2+0+1}{9} = 0{,}3333,$$

$$(\text{cov}(Klaidu\_\text{Re}gistravimas) * GE) = 0{,}1111.$$

Pranešimų generavimas yra atliekamas atskiroje *ErrorMessenger* klasėje. Vidinių struktūrinių ryšių nėra. Yra išorinis vienas ryšys, jungiantis klasę *ErrorMessenger* su klase *Calculus*. Naudojama standartinė pranešimų išvedimo į ekraną funkcija *System.out.println()* ir sąlyginio parinkimo sakinys *switch*. Dedikuotų metodų nėra, interfeisų nėra. Algoritminės realizacijos nėra, dedikuotų naudojamų resursų nėra, loginės klaidų aptikimo schemos nėra.

$$GE = \frac{1}{3} = 0{,}3333,$$

$$\text{cov}(\text{Pr}anesimu\_Generavimas) = \frac{2+1+0}{9} = 0{,}3333,$$

$$(\text{cov}(\text{Pr}anesimu\_Generavimas) * GE) = 0{,}1111.$$

Remiantis gautais duomenimis, tegul visų į klaidų valdymą orientuotų priemonių išraiškų *(cov()*GE)* suma senojoje sistemoje bus žymima $IMPCOV_B$, o naujojoje sistemoje – $IMPCOV_A$. Tada, suskaičiavę šias sumas, galime paskaičiuoti ir galutinę pakankamumo koeficiento (*OC*) reikšmę:

$$OC = \frac{0{,}1111 + 0{,}1111}{0{,}1481 + 0{,}1111 + 0{,}1111 + 0{,}1111} = 0{,}4814.$$

Ši reikšmė rodo, kiek kartų apytiksliai senojoje sistemoje į klaidų valdymą orientuotos priemonės yra silpniau išvystytos ir mažiau panaudojamos negu naujojoje. *Šis koeficientas yra taip pat skirtas lyginti bent kelias naujas tos pačios sistemos versijas.*

#### 5.2.2.5 *Atlikti skaičiavimai*

Atlikus pavyzdinės analizuojamosios sistemos analizę pagal pristatytąją *PCMB* vertinimo metodiką, pirmiausia buvo paskaičiuotas planinis koeficientas. Senąją analizuojamos sistemos versiją sudaro 15 komponentų (klasių), o naująją - 27 komponentai. Remiantis 3.2 poskyryje pateikta 4 lentele, senojoje analizuojamos sistemos versijoje yra numatyta apdoroti iš viso $P_B = 12$ skirtingų klaidų, o naujojoje - $P_A = 52$ klaidos. Iš čia, $\Delta P = 52 - 12 = 40$. Kadangi nėra būdo, kuris leistų sužinoti dydžių $TE_B$, $TE_A$, $\Delta TE$ reikšmes, nes į juos įeina tiek nuspėjamos, tiek ir nenuspėjamos klaidos, tai planinis koeficientas bus lygus išraiškai:

$$PC = \frac{TE_B - 12}{(TE_B - 12) + (\Delta TE - 40)}.$$

Kadangi pokyčio $\Delta P$ reikšmė yra gana nemaža (naujojoje analizuojamos sistemos versijoje numatytų apdoroti klaidų skaičius yra net ~4,3 karto didesnis už klaidų skaičių senojoje analizuojamos sistemos versijoje), tai visai tikėtina, kad reiškinio $\Delta TE - 40$ reikšmė, kuri reiškia dar nežinomų klaidų skaičių, nebus labai didelė, nes kuo daugiau klaidų mes sužinome, tuo mažiau jų lieka nežinomų. Taigi bendroji išvada būtų tokia, kad yra tikėtina, jog planinio koeficiento reikšmė turėtų būti pakankamai nemaža, o tai reiškia, kad naujoji analizuojamos sistemos versija yra tikrai labiau pasirengusi klaidų antpuoliui nei senoji.

Skaičiuojant tikrinimo vertinimo koeficientą, reikia nustatyti funkcijų *EVOPC* ir *SOPC* reikšmes kiekvienam senosios ir naujosios analizuojamosios sistemos versijos komponentui (žr. 15 lentelę). Kadangi naujosios analizuojamosios sistemos versijoje nėra sukurta jokios naujos kodo realizacijos, tai funkcijų įvertinimai bus empiriniai ir pateikiami tik kaip teoriniai samprotavimai, remiantis 3.3 poskyryje pristatyto klaidų aptikimo metodo aprašymu.

**15 lentelė.** *EVOPC* ir *SOPC* funkcijų reikšmių įvertinimų suvestinė.

| **Nr.** | **Metodo pavadinimas** | ***SOPC* reikšmė prieš** | ***EVOPC* reikšmė prieš** | ***SOPC* reikšmė po** | ***EVOPC* reikšmė po** |
|---|---|---|---|---|---|
| ***ATM*** | | | | | |
| 1. | *startupOperation* | 3 | 1 | 5 | 3 |
| 2. | *serviceCustomers* | 10 | 3 | 14 | 7 |
| 3. | *getPIN* | 3 | 0 | 4 | 1 |
| 4. | *getMenuChoice* | 3 | 0 | 5 | 2 |
| 5. | *getAmountEntry* | 3 | 0 | 4 | 1 |
| 6. | *checkIfCashAvailable* | 1 | 0 | 3 | 2 |
| 7. | *dispenseCash* | 1 | 0 | 3 | 2 |
| 8. | *acceptEnvelope* | 1 | 0 | 2 | 1 |
| 9. | *issueReceipt* | 1 | 0 | 2 | 1 |
| 10. | *reEnterPIN* | 3 | 0 | 4 | 1 |

| | | | | | |
|---|---|---|---|---|---|
| 11. | *reportTransactionFailure* | 3 | 0 | 5 | 2 |
| 12. | *ejectCard* | 1 | 0 | 2 | 1 |
| 13. | *retainCard* | 3 | 0 | 4 | 1 |
| ***Bank*** | | | | | |
| 1. | *initiateWithdrawl* | 4 | 2 | 9 | 7 |
| 2. | *finishWithdrawl* | 3 | 0 | 4 | 1 |
| 3. | *initiateDeposit* | 2 | 0 | 6 | 4 |
| 4. | *finishDeposit* | 1 | 0 | 2 | 1 |
| 5. | *doTransfer* | 7 | 1 | 13 | 7 |
| 6. | *doInquiry* | 2 | 0 | 6 | 4 |
| 7. | *chooseAccountType* | 1 | 0 | 4 | 3 |
| 8. | *accountName* | 0 | 0 | 2 | 2 |
| 9. | *rejectionExplanation* | 0 | 0 | 2 | 2 |
| ***CardReader*** | | | | | |
| 1. | *ejectCard* | 7 | 2 | 10 | 5 |
| 2. | *retainCard* | 1 | 1 | 1 | 1 |
| 3. | *checkForCardInserted* | 16 | 6 | 18 | 8 |
| 4. | *cardNumber* | 0 | 0 | 1 | 1 |
| 5. | *action* | 1 | 0 | 1 | 0 |
| ***Keybord*** | | | | | |
| 1. | *readPIN* | 9 | 1 | 10 | 2 |
| 2. | *readMenuChoice* | 1 | 0 | 1 | 0 |
| 3. | *readAmountEntry* | 20 | 1 | 21 | 2 |
| 4. | *inKey* | 2 | 1 | 3 | 1 |
| 5. | *action* | 1 | 0 | 2 | 1 |
| ***Display*** | | | | | |
| 1. | *requestCard* | 1 | 0 | 1 | 0 |

| | | | | | |
|---|---|---|---|---|---|
| 2. | *requestPIN* | 1 | 0 | 1 | 0 |
| 3. | *displayMenu* | 2 | 0 | 3 | 1 |
| 4. | *requestAmountEntry* | 1 | 0 | 1 | 0 |
| 5. | *requestDepositEnvelope* | 1 | 0 | 1 | 0 |
| 6. | *reportCardUnreadable* | 1 | 0 | 1 | 0 |
| 7. | *reportTransactionFailure* | 2 | 0 | 3 | 1 |
| 8. | *requestReEnterPIN* | 1 | 0 | 1 | 0 |
| 9. | *reportCardRetained* | 2 | 1 | 2 | 1 |
| 10. | *echoInput* | 1 | 0 | 3 | 2 |
| 11. | *clearDisplay* | 1 | 0 | 2 | 1 |
| 12. | *write* | 4 | 2 | 6 | 4 |
| ***CashDispenser*** | | | | | |
| 1. | *setCash* | 0 | 0 | 1 | 1 |
| 2. | *dispenseCash* | 7 | 1 | 9 | 3 |
| 3. | *currentCash* | 0 | 0 | 0 | 0 |
| ***EnvelopeAcceptor*** | | | | | |
| 1. | *acceptEnvelope* | 12 | 5 | 14 | 7 |
| 2. | *action* | 1 | 0 | 2 | 1 |
| ***OperatorPanel*** | | | | | |
| 1. | *switchOn* | 5 | 2 | 6 | 3 |
| 2. | *getInitialCash* | 4 | 2 | 5 | 3 |
| ***ReceiptPrinter*** | | | | | |
| 1. | *printReceipt* | 11 | 1 | 13 | 3 |
| ***Session*** | | | | | |
| 1. | *doSessionUseCase* | 6 | 4 | 7 | 5 |
| 2. | *doInvalidPINExtention* | 3 | 1 | 4 | 2 |
| 3. | *doFailedTransactionExtention* | 4 | 1 | 5 | 2 |

| | | | | | |
|---|---|---|---|---|---|
| 4. | *cardNumber* | 0 | 0 | 0 | 0 |
| 5. | *PIN* | 0 | 0 | 0 | 0 |
| ***Transaction*** | | | | | |
| 1. | *chooseTransaction* | 5 | 0 | 6 | 1 |
| 2. | *doTransactionUseCase* | 4 | 4 | 5 | 5 |
| 3. | *getTransactionSpecificsFromCustomer* | 0 | 0 | 0 | 0 |
| 4. | *sendToBank* | 0 | 0 | 0 | 0 |
| 5. | *finishApprovedTransaction* | 0 | 0 | 0 | 0 |
| ***WithdrawTransaction*** | | | | | |
| 1. | *getTransactionSpecificsFromCustomer* | 10 | 1 | 11 | 2 |
| 2. | *sendToBank* | 4 | 0 | 6 | 2 |
| 3. | *finishApprovedTransaction* | 6 | 0 | 8 | 2 |
| ***DepositTransaction*** | | | | | |
| 1. | *getTransactionSpecificsFromCustomer* | 2 | 0 | 3 | 1 |
| 2. | *sendToBank* | 5 | 1 | 7 | 3 |
| 3. | *finishApprovedTransaction* | 6 | 1 | 8 | 3 |
| ***TransferTransaction*** | | | | | |
| 1. | *getTransactionSpecificsFromCustomer* | 3 | 0 | 4 | 1 |
| 2. | *sendToBank* | 4 | 0 | 6 | 2 |
| 3. | *finishApprovedTransaction* | 4 | 0 | 6 | 2 |
| ***InquiryTransaction*** | | | | | |
| 1. | *getTransactionSpecificsFromCustomer* | 1 | 0 | 2 | 1 |
| 2. | *sendToBank* | 4 | 0 | 6 | 2 |
| 3. | *finishApprovedTransaction* | 4 | 0 | 6 | 2 |

Remiantis šioje lentelėje pateiktais įverčiais, jau galima paskaičiuoti ir galutinį tikrinimo vertinimo koeficientą.

$$EVR_B = \frac{46}{805} = 0{,}05714,$$

$$EVR_A = \frac{146}{1112} = 0{,}13129,$$

$$EC = \frac{EVR_B}{EVR_A} = \frac{0{,}05714}{0{,}13129} = 0{,}43522.$$

Iš gautų rezultatų matyti, kad naujojoje analizuojamos sistemos versijoje klaidų tikrinimo metoduose atitikimas pilnam tikrinimo planui yra ~2,3 karto didesnis nei senojoje, o tai įrodo, kad naujoji analizuojamos sistemos versija yra labiau pasirengusi laiku ir vietoje aptikti klaidas prieš tai, kol jos pradės plisti.

Kitas žingsnis yra pajėgumo koeficiento reikšmės skaičiavimas. Atsižvelgiant į anksčiau aprašytą pajėgumo koeficiento radimo eigą, pirmiausia reikia nustatyti visas pagrindines vartotojo užduotis. Jos analizuojamoje bankomato sistemoje yra keturios: $T = \{Withdraw\_Transaction, Deposit\_Transaction, Transfer\_Transaction, Inquiry\_Transaction\}$ Dabar kiekvienai vartotojo užduočiai reikia nustatyti komponentų aibes.

$$S(Withdraw\_Transaction) = \{ATM, Bank, Session, Transaction, WithdrawlTransaction\}$$
$$S(Deposit\_Transaction) = \{ATM, Bank, Session, Transaction, DepositTransaction\}$$
$$S(Transfer\_Transaction) = \{ATM, Bank, Session, Transaction, TransferTransaction\}$$
$$S(Inquiry\_Transaction) = \{ATM, Bank, Session, Transaction, InquiryTransaction\}$$

Kaip matome, visoms keturioms užduotims atlikti yra reikalinga 80% tų pačių komponentų. Kadangi toliau reikia sudaryti mažiausios, vidutinės ir didžiausios rizikų komponentų aibes, tai situacija bus tokia:

$$S_{DRS}(T) = S(Withdraw\_Transaction) \cap S(Deposit\_Transaction) \cap$$
$$S(Transaction\_Transaction) \cap S(Inquiry\_Transaction) = \{ATM, Bank, Session, Transaction\}$$

$$S_{VRS}(T) = \left( S(Withdraw\_Transaction) \setminus \begin{pmatrix} S(Deposit\_Transaction) \cup \\ S(Transfer\_Transaction) \cup \\ S(Inquiry\_Transaction) \end{pmatrix} \right) \cup$$
$$\left( S(Deposit\_Transaction) \setminus \begin{pmatrix} S(Withdraw\_Transaction) \cup \\ S(Transfer\_Transaction) \cup \\ S(Inquiry\_Transaction) \end{pmatrix} \right) \cup$$
$$\left( S(Transfer\_Transaction) \setminus \begin{pmatrix} S(Withdraw\_Transaction) \cup \\ S(Deposit\_Transaction) \cup \\ S(Inquiry\_Transaction) \end{pmatrix} \right) \cup$$
$$\left( S(Inquiry\_Transaction) \setminus \begin{pmatrix} S(Withdraw\_Transaction) \cup \\ S(Deposit\_Transaction) \cup \\ S(Transfer\_Transaction) \end{pmatrix} \right) =$$
$$= \begin{Bmatrix} WithdrawlTransaction, \\ DepositTransaction, \\ TransferTransaction, \\ InquiryTransaction \end{Bmatrix}$$

$$S_{MRS}(T) = S(T) \setminus (S_{DRS}(T) \cup S_{VRS}(T)) = \phi$$

Kaip matome, analizuojamosios sistemos atveju turime tik didžiausios ir vidutinės rizikos komponentų aibes ir dar blogiau – aibę, sudarytą beveik iš visų kiekvienai vartotojo užduočiai atlikti reikalingų komponentų. Kitaip tariant, jei bent vieno iš keturių komponentų *ATM*, *Bank*, *Session* ir *Transaction* veikimas sutriks, tai nebebus galima atlikti nei vienos vartotojo užduoties.

Toliau kiekvienai užduočiai reikia nustatyti funkcinius vertinimo kriterijus. Taigi vartotojo užduočių funkcinių vertinimo kriterijų aibės bus tokios:

$$CR(Withdraw\_Transaction) = \begin{Bmatrix} Įve\,\dim as, \\ Išve\,\dim as, \\ Duomenų\_Mainai, \\ Duomenų\_\Pr iemimas / Perdavimas\_Tinklu \end{Bmatrix}$$

$$CR(Deposit\_Transaction) = \begin{Bmatrix} Įve\,\dim as, \\ Išve\,\dim as, \\ Duomenų\_Mainai, \\ Duomenų\_\Pr iemimas / Perdavimas\_Tinklu \end{Bmatrix}$$

$$CR(Transfer\_Transaction) = \begin{Bmatrix} Įve\,\text{dim}\,as, \\ Išve\,\text{dim}\,as, \\ Duomenų\_Mainai, \\ Duomenų\_\text{Pr}iemimas / Perdavimas\_Tinklu \end{Bmatrix}$$

$$CR(Inquiry\_Transaction) = \begin{Bmatrix} Įve\,\text{dim}\,as, \\ Išve\,\text{dim}\,as, \\ Duomenų\_Mainai, \\ Duomenų\_\text{Pr}iemimas / Perdavimas\_Tinklu \end{Bmatrix}$$

Šiuo atveju kiekvienos vartotojo užduoties funkciniai vertinimo kriterijai sutampa. Taigi lieka tik paskutinis šio vertinimo proceso eigos etapas - globalaus sistemos operacinio darbo pajėgumo klaidų apgulties metu įvertinimas, remiantis anksčiau pateiktomis formulėmis. Lokalaus sistemos operacinio darbo pajėgumas klaidų apgulties metu nebus nagrinėjamas, nes tam tikslui geriausia yra turėti jau daugiau ar mažiau aiškią ir išbaigtą sistemos funkcionalumo realizaciją bei dėl to, kad jis labiau tinka atvejui, kai vartotojo užduočių funkciniai vertinimo kriterijai yra skirtingi. Dėl to prieš galutinės pajėgumo koeficiento reikšmės paskaičiavimą reikia rasti visas funkcijos *crit* reikšmes. Šie duomenys yra pateikti 16 lentelėje.

**16 lentelė.** *Crit* funkcijos reikšmių įvertinimų suvestinė.

| **Nr.** | **Vartotojo užduotis** | **Neveiksnus komponentas** | **Funkcijos *crit* reikšmė prieš** | **Funkcijos *crit* reikšmė po** |
|---|---|---|---|---|
| **DRKA** | | | | |
| 1. | *Withdraw_Transaction* | *ATM* | 0 | 5 |
| 2. | *Withdraw_Transaction* | *Bank* | 3 | 3 |
| 3. | *Withdraw_Transaction* | *Session* | 4 | 4 |
| 4. | *Withdraw_Transaction* | *Transaction* | 0 | 0 |
| 5. | *Withdraw_Transaction* | *ATM, Bank* | 0 | 3 |
| 6. | *Withdraw_Transaction* | *ATM, Session* | 0 | 4 |
| 7 | *Withdraw_Transaction* | *ATM,Transaction* | 0 | 0 |
| 8. | *Withdraw_Transaction* | *ATM, Bank, Session* | 0 | 2 |
| 9. | *Withdraw_Transaction* | *ATM, Bank, Transaction* | 0 | 0 |
| 10. | *Withdraw_Transaction* | *ATM, Bank, Session, Transation* | 0 | 0 |

| | | | | |
|---|---|---|---|---|
| 1. | *Deposit_Transaction* | *ATM* | 0 | 5 |
| 2. | *Deposit_Transaction* | *Bank* | 3 | 3 |
| 3. | *Deposit_Transaction* | *Session* | 4 | 4 |
| 4. | *Deposit_Transaction* | *Transaction* | 0 | 0 |
| 5. | *Deposit_Transaction* | *ATM, Bank* | 0 | 3 |
| 6. | *Deposit_Transaction* | *ATM, Session* | 0 | 4 |
| 7 | *Deposit_Transaction* | *ATM,Transaction* | 0 | 0 |
| 8. | *Deposit_Transaction* | *ATM, Bank, Session* | 0 | 2 |
| 9. | *Deposit_Transaction* | *ATM, Bank, Transaction* | 0 | 0 |
| 10. | *Deposit_Transaction* | *ATM, Bank, Session, Transation* | 0 | 0 |
| | | | | |
| 1. | *Transfer_Transaction* | *ATM* | 0 | 5 |
| 2. | *Transfer_Transaction* | *Bank* | 3 | 3 |
| 3. | *Transfer_Transaction* | *Session* | 4 | 4 |
| 4. | *Transfer_Transaction* | *Transaction* | 0 | 0 |
| 5. | *Transfer_Transaction* | *ATM, Bank* | 0 | 3 |
| 6. | *Transfer_Transaction* | *ATM, Session* | 0 | 4 |
| 7 | *Transfer_Transaction* | *ATM,Transaction* | 0 | 0 |
| 8. | *Transfer_Transaction* | *ATM, Bank, Session* | 0 | 2 |
| 9. | *Transfer_Transaction* | *ATM, Bank, Transaction* | 0 | 0 |
| 10. | *Transfer_Transaction* | *ATM, Bank, Session, Transation* | 0 | 0 |
| | | | | |
| 1. | *Inquiry_Transaction* | *ATM* | 0 | 5 |
| 2. | *Inquiry_Transaction* | *Bank* | 3 | 3 |
| 3. | *Inquiry_Transaction* | *Session* | 4 | 4 |
| 4. | *Inquiry_Transaction* | *Transaction* | 0 | 0 |
| 5. | *Inquiry_Transaction* | *ATM, Bank* | 0 | 3 |
| 6. | *Inquiry_Transaction* | *ATM, Session* | 0 | 4 |
| 7 | *Inquiry_Transaction* | *ATM,Transaction* | 0 | 0 |
| 8. | *Inquiry_Transaction* | *ATM, Bank, Session* | 0 | 2 |
| 9. | *Inquiry_Transaction* | *ATM, Bank, Transaction* | 0 | 0 |
| 10. | *Inquiry_Transaction* | *ATM, Bank, Session, Transation* | 0 | 0 |

| **VRKA** | | | | |
|---|---|---|---|---|
| 1. | *Withdraw_Transaction* | *WithdrawTransaction* | 0 | 5 |
| 2. | *Deposit_Transaction* | *DepositTransaction* | 0 | 5 |
| 3. | *Transfer_Transaction* | *TransferTransaction* | 0 | 5 |
| 4. | *Inquiry_Transaction* | *InquiryTransaction* | 0 | 5 |

Kadangi vartotojo užduotys remiasi vienodomis *ATM*, *Bank* ir *Session* paslaugomis tiems patiems tikslams pasiekti, tai ir funkcijų *crit* reikšmės kiekvienu atveju sutampa. Toliau remiantis šioje lentelėje pateiktais įverčiais, jau galima paskaičiuoti ir galutinį pajėgumo koeficientą.

$$OPUS_B = 28,$$

$$OPUS_A = 104,$$

$$CC = \frac{OPUS_B}{OPUS_A} = \frac{28}{104} = 0{,}26923.$$

Iš gautų rezultatų matyti, kad naujoji analizuojama sistema yra ~3,7 karto pajėgesnė palaikyti ir užtikrinti operacinį darbą klaidų apgulties atveju nei senoji. Vienas pagrindinių tai lemiančių veiksnių yra sistemos reaktyvavimo idėja ir jos realizavimas naujojoje analizuojamoje sistemoje. Pavyzdžiui, jei koks nors vartotojo užduoties atlikimą lemiantis komponentas tampa neveiksnus, tai kritiniu atveju galima restartuoti sistemos darbą taip, kad jį palaikytų (bent iki paskutinės nebaigtos vykdyti sesijos pabaigos) ne pats *ATM* bankomatas, o jo operatoriaus terminalas.

Na, ir paskutinysis *PCMB* vertinimo metodikoje apibrėžtas žingsnis yra pakankamumo koeficiento reikšmės skaičiavimas. Tam tikslui pirmiausia reikia sudaryti į klaidų valdymą orientuotų priemonių sąrašą. Tokios priemonių aibės senojoje ir naujojoje analizuojamos sistemos versijose būtų tokios:

$$I_B = \begin{Bmatrix} Klaidų\_Identifikavimas, \\ Klaidų\_Aptikimas, \\ Klaidų\_Apdorojimas, \\ \Pr anešimų\_Generavimas \end{Bmatrix}$$

$$I_A = \begin{Bmatrix} Klaidų\_Identifikavimas, \\ Klaidų\_Aptikimas, \\ Klaidų\_\mathrm{Re} gistravimas, \\ Klaidų\_Apdorojimas, \\ \Pr anešimų\_Generavimas, \\ Sistemos\_\mathrm{Re} aktyvavimas \end{Bmatrix}$$

Dabar kiekvienos aibės kiekvieną priemonę reikia detalizuoti, naudojant anksčiau aprašytąsias triadas. Tai yra pateikta 17 lentelėje.

**17 lentelė.** Į klaidų valdymą orientuotų priemonių detalizavimas.

| **Nr.** | **Priemonės pavadinimas** | **Architektūrinė realizacija (*AR*)** | **Funkcinė realizacija (*FR*)** | **Algoritminė realizacija (*GR*)** | **Globalus Priemonės panaudojimo mastas (*GE*)** |
|---|---|---|---|---|---|
| **Senoji analizuojamos sistemos versija** | | | | | |
| 1. | *Klaidų_Identifikavimas* | Atskiro klaidų klasifikavimo nėra; Dedikuotų komponentų, struktūrinių ryšių nėra. | Klaidos apibrėžtos kaip konstantiniai *public static final int* tipo kintamieji; Centralizuotos jų saugojimo vietos nėra; Pasiekiamumas tarp klasių egzistuoja; Dedikuotų funkcijų nėra. | Dedikuotų funkcijų realizacijų nėra; Dedikuotų resursų panaudojimo nėra; loginės klaidų identifikavimo schemos nėra. | $GE = \frac{5}{15} = 0{,}33333$ |

| | | | | | |
|---|---|---|---|---|---|
| 2. | *Klaidų_Aptikimas* | Atskirų į klaidų aptikimą orientuotų komponentų, struktūrinių ryšių nėra. | Klaidų aptikimas yra realizuotas, panaudojant sąlyginius sakinius, tikrinant nepageidaujamas situacijas, ir standartiniai *try-catch* blokai; Dedikuotų funkcijų nėra. | Dedikuotų funkcijų realizacijų nėra; Naudojamos standartinės (*Exception* tipo) išvestinės klaidų klasės; loginės klaidų aptikimo schemos nėra. | $GE = \frac{12}{15} = 0{,}8$ |
| 3. | *Klaidų_Apdorojimas* | Atskirų į klaidų apdorojimą orientuotų komponentų, struktūrinių ryšių nėra. | Klaidų apdorojimui naudojamas standartinis *try-catch* blokas, klaidų pranešimų vartotojui pateikimo funkcijos, koregavimo veiksmų inicijavimo laukimo režimas; Dedikuotų funkcijų nėra; *try-catch* blokas yra pasiekiamas visose klasėse. | Dedikuotų funkcijų realizacijų nėra; Naudojamos standartinės (*Exception* tipo) išvestinės klaidų klasės; loginės klaidų apdorojimo schemos nėra. | $GE = \frac{12}{15} = 0{,}8$ |
| 4. | *Pranešimų_Generavimas* | Atskirų į klaidų pranešimų generavimą orientuotų komponentų, struktūrinių ryšių nėra. | Pranešimų generavimo funkcijų nėra; Egzistuoja tik pranešimų parinkimo funkcijos, realizuotos 2 klasėse; jų pasiekiamumas egzistuoja, tačiau ribotai. | Naudojamos individualiai rašytos pranešimų parinkimo funkcijos; Dedikuotų resursų panaudojimo nėra; loginės pranešimų generavimo ir parinkimo schemos nėra. | $GE = \frac{4}{15} = 0{,}26667$ |
| **Naujoji analizuojamos sistemos versija** | | | | | |
| 1. | *Klaidų_Identifikavimas* | Sistemoje yra realizuota atskira klaidų klasifikacija; Kiekvienam klaidos tipui egzistuoja atskira dedikuota klasė, sujungta su bendra sistemos klaidų valdymo posisteme. | Kiekvienoje klaidos klasėje yra atitinkami klaidą nusakantys atributai, kurių reikšmės pasiekiamos atitinkamomis funkcijomis; Realizuotas centralizuotas klaidų klasių pasiekiamumas; | Kiekvienoje klaidų klasėje yra numatytų funkcijų realizacijos; Dedikuotų resursų panaudojimo nėra; Klaidų hierarchija yra gauta, naudojant loginį jų tarpusavio susiejimo ryšį pagal probleminę sritį. | $GE = \frac{15}{27} = 0{,}55556$ |

| | | | | | |
|---|---|---|---|---|---|
| 2. | *Klaidų_Aptikimas* | Klaidų aptikimui yra realizuota atskira klasė, sujungta su bendra sistemos klaidų valdymo posisteme. | Klaidų aptikimui atlikti tam skirtoje klasėje yra numatytos dedikuotos funkcijos, kurios yra pasiekiamos kiekvienai dalykinės srities, bet ne pačiai klaidų valdymo posistemei. | Klaidų aptikimui skirtoje klasėje yra numatytų funkcijų realizacijos; Dedikuotų resursų panaudojimo nėra; Pageidaujamam funkcionalumui pasiekti yra naudojama loginė klaidų aptikimo schema. | $GE = \frac{15}{27} = 0{,}55556$ |
| 3. | *Klaidų_Registravimas* | Klaidų registravimui yra realizuotos 2 atskiros klasės, sujungtos su bendra sistemos klaidų valdymo posisteme. | Klaidų registravimui atlikti tam skirtose klasėse yra numatytos dedikuotos funkcijos, kurios yra pasiekiamos klaidų valdymo posistemės klaidų apdorojimo komponentui. | Klaidų registravimui skirtose klasėse yra numatytų funkcijų realizacijos; Klaidų registravimui yra reikalinga atmintis, kurioje bus saugomi klaidų protokolai; Pageidaujamam funkcionalumui pasiekti yra naudojama loginė klaidų registravimo schema. | $GE = \frac{15}{27} = 0{,}55556$ |
| 4. | *Klaidų_Apdorojimas* | Klaidų apdorojimui yra realizuota atskira klasė, sujungta su bendra sistemos klaidų valdymo posisteme. | Klaidų apdorojimui atlikti tam skirtoje klasėje yra numatytos dedikuotos funkcijos, kurios yra pasiekiamos kiekvienai dalykinės srities, bet ne pačiai klaidų valdymo posistemei. | Klaidų apdorojimui skirtoje klasėje yra numatytų funkcijų realizacijos; Dedikuotų resursų panaudojimo nėra; Pageidaujamam funkcionalumui pasiekti yra naudojama loginė klaidų apdorojimo schema. | $GE = \frac{15}{27} = 0{,}55556$ |

| | | | | | |
|---|---|---|---|---|---|
| 5. | *Pranešimų_Generavimas* | Klaidų pranešimų generavimui yra realizuotos 4 atskiros klasės, sujungtos su bendra sistemos klaidų valdymo posisteme. | Klaidų pranešimų generavimui atlikti tam skirtose klasėse yra numatytos dedikuotos funkcijos, kurios yra pasiekiamos klaidų valdymo posistemės klaidų apdorojimo komponentui. | Klaidų pranešimų generavimui skirtose klasėse yra numatytų funkcijų realizacijos; Dedikuotų resursų panaudojimo nėra; Pageidaujamam funkcionalumui pasiekti yra naudojama loginė klaidų pranešimų generavimo schema, paremta skirtingais klaidų lygiais. | $GE = \frac{15}{27} = 0{,}55556$ |
| 6. | *Sistemos_Reaktyvavimas* | Sistemos reaktyvavimui dedikuotų komponentų nėra, tačiau egzistuoja specialus tokį funkcionalumą leidžiantis pasiekti ryšys, jungiantis operatoriaus terminalą su pagrindine bankomato sistemos *ATM* klase. | Sistemos reaktyvavimui atlikti *ATM* operatoriaus terminalo klasėje yra numatytos dedikuotos funkcijos, skirtos privačiam naudojimui ir pasiekiamos tik tam komponentui. | Sistemos reaktyvavimui *ATM* operatoriaus terminalo klasėje yra numatytų funkcijų realizacijos; Sistemos reaktyvavimui yra reikalinga papildoma atmintis ir DB, kurioje bus saugomi duomenys apie konkrečiu laiko momentu užfiksuotą sistemos būklę; Pageidaujamam funkcionalumui pasiekti yra naudojama loginė sistemos reaktyvavimo schema, paremta dinamine rekonfigūracija. | $GE = \frac{15}{27} = 0{,}55556$ |

Toliau, prieš pereinant prie galutinio rezultato skaičiavimo, kiekvienai priemonei liko paskaičiuoti funkcijos *cov* ir išraiškos (*cov* * *GE*) reikšmes, kurios yra pateiktos 18 lentelėje.

**18 lentelė.** Funkcijos *cov* ir išraiškos (*cov* * *GE*) reikšmių įvertinimų suvestinė.

| **Nr.** | **Priemonės pavadinimas** | **Funkcijos *cov* reikšmė** | **Išraiškos (*cov* * *GE*) reikšmė** |
|---|---|---|---|

| | | | |
|---|---|---|---|
| **Senoji analizuojamos sistemos versija** | | | |
| 1. | *Klaidų_Identifikavimas* | $\frac{0+2+0}{9}=0,22222$ | 0,07407 |
| 2. | *Klaidų_Aptikimas* | $\frac{0+2+1}{9}=0,33333$ | 0,26666 |
| 3. | *Klaidų_Apdorojimas* | $\frac{0+2+1}{9}=0,33333$ | 0,26666 |
| 4. | *Pranešimų_Generavimas* | $\frac{0+3+1}{9}=0,44444$ | 0,11852 |
| **Naujoji analizuojamos sistemos versija** | | | |
| 1. | *Klaidų_Identifikavimas* | $\frac{3+3+2}{9}=0,88889$ | 0,49383 |
| 2. | *Klaidų_Aptikimas* | $\frac{3+3+2}{9}=0,88889$ | 0,49383 |
| 3. | *Klaidų_Registravimas* | $\frac{3+3+3}{9}=1$ | 0,55556 |
| 4. | *Klaidų_Apdorojimas* | $\frac{3+3+2}{9}=0,88889$ | 0,49383 |
| 5. | *Pranešimų_Generavimas* | $\frac{3+3+2}{9}=0,88889$ | 0,49383 |
| 6. | *Sistemos_Reaktyvavimas* | $\frac{3+3+3}{9}=1$ | 0,55556 |

Turint jau visus reikalingus duomenis, galima rasti ir galutinę pakankamumo koeficiento reikšmę.

$$IMPCOV_B = 0,72591,$$

$$IMPCOV_A = 3,08644,$$

$$OC = \frac{IMPCOV_B}{IMPCOV_A} = \frac{0{,}72591}{3{,}08644} = 0{,}23519\,.$$

Iš gautų rezultatų matyti, kad naujojoje analizuojamos sistemos versijoje į klaidų valdymą orientuotos priemonės yra ~4,3 karto labiau, stipriau ir plačiau išvystytos nei senojoje.

Visuose *PCMB* vertinimo metodikos koeficientų skaičiavimo etapuose skirtumai tarp gautų rezultatų, išryškinantys ir esminius atsparumo klaidoms skirtumus tarp naujosios bei senosios analizuojamos sistemos versijų, galėtų būti dar didesni ir ryškesni, tačiau naujojoje analizuojamos sistemos versijoje klaidų valdymo modelis nėra net pilnai išbaigtas. Taip yra, pavyzdžiui, kad ir dėl to, jog klaidų valdymo posistemė neturi galimybės suvaldyti joje pačioje kylančias klaidas. Be to, dauguma rezultatų buvo gauti, remiantis teoriniais samprotavimais, nes naujosios analizuojamos sistemos atveju ne tik nebuvo sukurta jokia nauja kodo realizacija, bet ir dėl to, kad ir pats projektavimas nebuvo iki galo pilnai detalizuotas, išbaigtas ir ištestuotas. Tam tikslui reikėtų ne tik nemažai laiko ir praktinių įgūdžių, bet ir atitinkamų sričių specialistų, kurie galėtų užtikrinti bent jau pirminių rezultatų teisingumą konkretesnei analizei su visais turimais ir reikalingais duomenimis.

## 5.3 ĮVERTINIMAS PAGAL *RCEM* VERTINIMO METODIKĄ

Pirmasis *RCEM* vertinimo metodikos skaičiavimo etapas prasideda nuo teorinės bazės poreikių vertinimo skaičiavimo. Tam tikslui pirmiausia reikia nustatyti šiame darbe naudotos teorinės bazės (t.y. klaidų valdymo modelio) loginę struktūrą, kurią sudarys visos pristatyto modelio probleminės sritys kartu su jų smulkiosiomis sudedamosiomis dalimis. Galutinės reikšmės skaičiavimui taip pat reikės dviejų koeficientų: teorinės bazės pakankamumo ir pritaikomumo vertinimo, kuris bus atliktas pagal 19 lentelėje pateiktus duomenis.

**19 lentelė.** Teorinės bazės loginės struktūros dalių įvertinimas.

| **Nr.** | **Sritis** | $nsc(P_i)$ | $fsc(P_i)$ | $atac(P_i)$ | $ctac(P_i)$ | $atoc(P_i)$ | $ctoc(P_i)$ |
|---|---|---|---|---|---|---|---|
| 1. | Klaidos objektas | 3 | 1 | 0 | 0 | 2 | 3 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2. | Klaidų hierarchija | 3 | 1 | 1 | 1 | 2 | 3 |
| 3. | Klaidų izoliavimo priemonės | 3 | 1 | 0 | 0 | 0 | 3 |
| 4. | Sąlygomis grįsto išraiškų patikrinimo priemonės | 4 | 1 | 0 | 0 | 0 | 3 |
| 5. | Atgalinis trasavimas | 3 | 1 | 0 | 1 | 2 | 3 |
| 6. | Centralizuotas klaidų registravimas | 3 | 1 | 0 | 0 | 2 | 3 |
| 7. | Klaidų tvarkyklė | 2 | 1 | 0 | 0 | 2 | 3 |
| 8. | Klaidų valdymas pagal nutylėjimą | 3 | 1 | 0 | 0 | 2 | 3 |
| 9. | Pranešimai apie klaidas | 3 | 1 | 0 | 0 | 2 | 3 |
| 10. | Išankstinis resursų išskyrimas | 4 | 1 | 0 | 0 | 2 | 3 |
| 11. | Pakopinis perkrovimas | 3 | 1 | 0 | 0 | 2 | 3 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 12. | Klaidų lygiai | 2 | 1 | 0 | 0 | 2 | 3 |
| 13. | Klaidų adaptavimas | 3 | 1 | 0 | 0 | 0 | 3 |
| 14. | Daugiagijis klaidų apdorojimas | 4 | 1 | 0 | 0 | 2 | 3 |
| 15. | Reaktyvavimo tipas ir reikalingi resursai | 4 | 1 | 0 | 0 | 2 | 3 |
| 16. | Reaktyvavimo valdymas | 3 | 1 | 0 | 0 | 2 | 3 |
| **Suma:** | | **50** | **16** | **1** | **2** | **26** | **48** |

Lentelėje iš viso yra skaičiuojamos 6 funkcijų reikšmės kiekvienai klaidų valdymo modelį sudarančiai probleminei sričiai. Šios funkcijos yra:

- $nsc(P_i)$ („*Needed Source Count*“) – skirta nustatyti, kiek iš viso skirtingos tematikos literatūros šaltinių kiekvienai klaidų valdymo modelio probleminei sričiai reikia, norint pradėti ieškoti jai sprendimo būdų.
- $fsc(P_i)$ („*Found Source Count*“) – skirta nustatyti, kiek iš tikrųjų prireikė literatūros šaltinių kiekvienai klaidų valdymo modelio probleminei sričiai, kol buvo surastas arba sugalvotas jos sprendimo būdas.
- $ctac(P_i)$ („*Complete Task Count*“) – skirta nustatyti, kiek iš viso skirtingų paruošiamųjų darbelių reikėjo atlikti, kad kiekvienai klaidų valdymo modelio probleminei sričiai surastas arba sugalvotas jos sprendimo būdas galėtų būti pradėtas taikyti konkrečiam atvejui.

- $atac(P_i)$ („*Actual Task Count*“) – skirta nustatyti, kiek iš tikrųjų prireikė atlikti paruošiamųjų darbelių, kol kiekvienai klaidų valdymo modelio probleminei sričiai surastas arba sugalvotas jos sprendimo būdas buvo pradėtas taikyti.
- $ctoc(P_i)$ („*Complete Tool Count*“) – skirta nustatyti, kiek iš viso skirtingų įrankių reikia kiekvienos klaidų valdymo modelio probleminės srities paruošiamiesiems darbeliams atlikti.
- $atoc(P_i)$ („*Actual Tool Count*“) – skirta nustatyti, kiek iš tikrųjų prireikė įrankių kiekvienos klaidų valdymo modelio probleminės srities paruošiamiesiems darbeliams atlikti.

Remiantis lentelėje pateiktais duomenimis, galima suskaičiuoti jau ir teorinės bazės pakankamumo ir pritaikomumo vertinimo koeficientus (atitinkamai *SC* („*Sufficiency Coeficient*“) ir *AC* („*Application Coeficient*“)), kurių reikšmės bus lygios:

$$SC = \frac{aka[TB]}{cka[TB]} = \frac{\sum_{i=1}^{n} fsc(P_i)}{\sum_{i=1}^{n} nsc(P_i)} = \frac{16}{50} = 0{,}32,$$

$$AC = \frac{\sum_{i=1}^{n} atac(P_i) + \sum_{i=1}^{n} atoc(P_i)}{\sum_{i=1}^{n} ctac(P_i) + \sum_{i=1}^{n} ctoc(P_i)} = \frac{1+26}{2+48} = \frac{27}{50} = 0{,}54.$$

Iš čia, galutinė teorinės bazės poreikių vertinimo (*PPL* („*Practical Preparation Level*“)) reikšmė bus lygi:

$$PPL = AC \times SC = 0{,}32 * 0{,}54 = 0{,}1728.$$

Gautieji rezultatai tik patvirtina faktą, kad iš tikrųjų daugiau su teorine baze buvo dirbama techninių ir programinių priemonių (t.y. kompiuterio ir *MS Word* teksto redaktoriaus) naudojimo metu. Jokių paruošiamųjų darbų, nagrinėjant kiekvieną klaidų valdymo modelio probleminę sritį, beveik nebuvo, o tiksliau vienas paruošiamasis buvo atliktas tik pradžioje. Dėl to ir galutinė teorinės bazės poreikių vertinimo reikšmė gavosi gana maža.

Reikalavimų išpildymo vertinimo skaičiavimui pirmiausia reikia įvardinti reikalavimus, kuriuos naujoji analizuojamosios sistemos versija turi tenkinti. Po to tame sąraše reikia pažymėti tuos reikalavimus, kurie buvo įgyvendinti. Tokių nedekomponuotų ir nenuleistų žemyn aukšto lygio reikalavimų, tinkančių nagrinėjamam atvejui, pavyzdys yra pateiktas 20 lentelėje.

**20 lentelė.** Naujosios analizuojamos sistemos reikalavimų suvestinė.

| Nr. | Reikalavimo formuluotė | Ar reikalavimas buvo įgyvendintas? | Komentaras |
|---|---|---|---|
| 1. | Klaidų valdymo modelis analizuojamoje sistemoje turi būti realizuotas kaip nepriklausoma klaidų valdymo posistemė taip, kad naująją analizuojamosios sistemos versiją iš viso sudarytų dvi posistemės – dalykinės srities (su realizuotu banko operacijų vykdymu) ir klaidų valdymo. | — | Abi posistemės naujojoje analizuojamos sistemos versijoje yra realizuotos nepilnai ir tik projektavimo lygmenyje. |
| 2. | Klaidų valdymo posistemė turi būti sudaryta iš komponentų, iš kurių kiekvienas atitinka kurią nors vieną iš 7 klaidų valdymo modelio probleminių sričių. | √ | — |
| 3. | Bet kuriam dalykinės srities posistemės komponentui turi būti prieinamas galimų klaidų aptikimo tikrinimo funkcionalumas. | √ | — |
| 4. | Bet kuriam klaidų valdymo posistemės komponentui turi būti prieinamas galimų klaidų aptikimo funkcionalumas. | — | Pačiai klaidų valdymo posistemei klaidų aptikimo funkcionalumas nebuvo numatytas. |
| 5. | Klaidų aptikimas naujojoje analizuojamos sistemos versijoje turi būti orientuotas į klasių metodų ir standartinių bibliotekų naudojamų funkcijų parametrų, „*pre*-“ sąlygų, invariantų ir „*post*-“ salygų tikrinimą. | √ | — |
| 6. | Naujojoje analizuojamos sistemos versijoje turi būti apibrėžta galimų klaidų hierarchija, kurios kiekvienas lygmuo enkapsuliuotų ne tik bendrą, bet ir tam lygmeniui specifinę informaciją apie klaidos įvykį. | √ | — |

| | | | |
|---|---|---|---|
| 7. | Kiekvienam klaidų hierarchijos lygmeniui turi būti apibrėžti ir realizuoti veiksmai, nurodantys, kas turi būti daroma, atsiradus numatytai to lygmens klaidai ar kitokiai susidariusiai nepageidaujamai situacijai. | — | Naujojoje analizuojamos sistemos versijoje tai yra realizuota nepilnai ir tik projektavimo lygmenyje. |
| 8. | Naujojoje analizuojamos sistemos versijoje priėjimas prie apibrėžtos klaidų hierarchijos turi būti centralizuotas ir prieinamas tik klaidų valdymo posistemei. | √ | — |
| 9. | Visa informacija apie naujojoje analizuojamos sistemos versijoje aptiktas numatytas ir nenumatytas klaidas turi būti saugoma tam skirtuose failuose. | √ | — |
| 10. | Naujojoje analizuojamos sistemos versijoje turi būti užtikrinta, kad atsiradus klaidai, apie jos įvykį bus atitinkamai informuota ir tą sistemą aptarnaujanti centrinė banko sistema. | √ | — |
| 11. | Bet kuriam dalykinės srities posistemės komponentui turi būti prieinamas klaidų apdorojimo funkcionalumas. | √ | — |
| 12. | Bet kuriam klaidų valdymo posistemės komponentui turi būti prieinamas klaidų apdorojimo funkcionalumas. | — | Pačiai klaidų valdymo posistemei klaidų apdorojimo funkcionalumas nebuvo numatytas. |
| 13. | Naujojoje analizuojamos sistemos versijoje turi būti realizuotas būdas kas fiksuotą laiko periodą išsaugoti sistemos arba jos dalies būklę, kuri galės būti panaudojama pakartotiniam darbo nuo atitinkamo jo momento paleidimui įvykdyti. | — | Naujojoje analizuojamos sistemos versijoje tai yra realizuota nepilnai ir tik projektavimo lygmenyje. |
| 14. | Visa informacija apie naujosios analizuojamos sistemos arba jos dalies būklę turi būti saugomi tam atitinkamoje duomenų bazėje. | √ | — |

| | | | |
|---|---|---|---|
| 15. | Naujojoje analizuojamos sistemos versijoje turi būti realizuotos specializuotos resursų tvarkymo priemonės, užtikrinančios nepriklausomą disponuojamų programinių resursų administravimą ir teikimą paslaugų, orientuotų į naudojimosi reikalingais resursais teisės suteikimą. | — | Naujojoje analizuojamos sistemos versijoje tai yra realizuota nepilnai ir tik projektavimo lygmenyje. |
| 16. | Specializuotos resursų tvarkymo priemonės turi būti prieinamos bet kuriam naujosios analizuojamos sistemos (tiek dalykinės srities, tiek ir klaidų valdymo posistemės) komponentui. | — | Naujojoje analizuojamos sistemos versijoje tai yra realizuota nepilnai ir tik projektavimo lygmenyje. |
| 17. | Naujojoje analizuojamoje sistemos versijoje turi būti realizuota centralizuota visų galimų pranešimų parinkimo ir generavimo vieta. | — | Naujojoje analizuojamos sistemos versijoje tai yra realizuota nepilnai ir tik projektavimo lygmenyje. |
| 18. | Naujojoje analizuojamos sistemos versijoje turi būti realizuoti atitinkami lygiai, kurie leistų bet kokiems galimiems (tiek klaidų, tiek ir kitokiems) pranešimams sugeneruoti teisingą, aiškią ir suprantamą jų turinio pateikimo formą. | — | Naujojoje analizuojamos sistemos versijoje tai yra realizuota nepilnai ir tik projektavimo lygmenyje. |
| 19. | Naujojoje analizuojamos sistemos versijoje turi būti realizuotas sistemos reaktivavimo iš normalią darbo eigą sutrikdžiusios klaidos ar kokios kitos nenumatytos situacijos mechanizmas, kuris geriausiu atveju užtikrintų nuo tam tikro kontrolinio taško pakartotinai aktyvuoto sistemos darbo palaikymą, vidutiniu atveju – leistų pabaigti paskutinę vykdytą operaciją vartotojo naudai, o blogiausiu atveju – bent jau suteiktų kokį nors aktyvų priėjimą *ATM* bankomato operatoriui. | — | Naujojoje analizuojamos sistemos versijoje tai yra realizuota nepilnai ir tik projektavimo lygmenyje. |
| 20. | Visų naujojoje analizuojamos sistemos versijoje esančių dalykinės srities posistemės komponentų realizacijos turi būti atitinkamai pakeistos ir / arba papildytos taip, kad atitiktų padarytus projektinius sprendimus. | — | Dalykinės srities posistemės komponentų realizacijos nebuvo keičiamos nei projektiniame, nei kodo lygmenyje. |

Turint šiuos duomenis, jau galima rasti ir reikalavimų išpildymo įverčio (*IRIL* („*Intermediate Requirement Implementation Level*“)) reikšmę, kuri yra skaičiuojama taip:

$$IRIL = \frac{arc(\ )}{trc(\ )} = \frac{9}{20} = 0{,}45\,.$$

Šioje formulėje funkcija $arc(\ )$ („*Actual Requirement Count*“) atitinka reikšmę, nusakančią išpildytų reikalavimų kiekį, o funkcija $trc(\ )$ („*Total Requirement Count*“) – reikšmę, nusakančią bendrą reikalingų išpildyti reikalavimų kiekį.

Ši reikšmė rodo, kad šiame darbe atliktos analizės ir patobulinimų metu nebuvo įgyvendinta net pusės 19 lentelėje pateiktų reikalavimų. Vis dėlto nesunku pastebėti, kad toks reikalavimų išpildymo vertinimo skaičiavimo būdas yra gana griežtas ir remiasi tik absoliučiu reikalavimų įgyvendinimo vertinimu ir neaprėpia situacijų, kai reikalavimai yra įgyvendinti dalinai. O pastarųjų atvejų vertinimas gali būti ne tik svarbus, bet ir naudingas, nes kartais net ir patį kurio nors konkretaus reikalavimo įgyvendinimą gali sudaryti keli aiškiai išskiriami etapai (pavyzdžiui, skaidant užduotimis, použdotimis ir pan.), iš kurių ne visi dar gali būti atlikti vertinimo metu (kaip, pavyzdžiui, šiame darbe). Tuomet ne toks griežtas vertinimas leistų parodyti, kiek kiekvienas konkretus reikalavimas buvo įgyvendintas pagal apibrėžtus jos tarpinius veiksmus.

Toliau, norint suskaičiuoti projektinių sprendimų įverčio reikšmę, reikia rasti tris tarpines reikšmes: architektūrinio lygmens vertinimas (funkcijos $arev(\ )$ („*Architecture Level Evaluation*“) skaičiavimas), funkcinio lygmens vertinimas (funkcijos $flev(\ )$ („*Function Level Evaluation*“) skaičiavimas) ir algoritminio lygmens vertinimas (funkcijos $glev(\ )$ („*Algorithm Level Evaluation*“)).

Pirmuoju architektūrinio lygmens vertinimo atveju reikia nustatyti reikalingų ir sukurtų (suprojektuotų / realizuotų) patobulintos dalykinės srities ir klaidų valdymo posistemėse esančių komponentų bei vidinių ir išorinių tuos komponentus jungiančių ryšių skaičius. Atsižvelgiant į 19 lentelėje pateiktus reikalavimus ir 17 paveikslėlyje pavaizduotą naujosios analizuojamos sistemos architektūrą, galima padaryti tokius pastebėjimus:

- Iš viso reikalingų komponentų turėtų būti bent jau 12.

- Iš viso vidinių komponentus jungiančių ryšių turėtų būti bent jau 7.
- Iš viso išorinių komponentus jungiančių ryšių turėtų būti bent jau 18.
- Turimu atveju komponentų skaičius yra lygus 12.
- Turimu atveju vidinių komponentus jungiančių ryšių skaičius yra lygus 10.
- Turimu atveju išorinių komponentus jungiančių ryšių skaičius yra lygus 4.

Turint šiuos duomenis, galima paskaičiuoti ir funkcijos $arev(\ )$ reikšmę.

$$arev(\ ) = \frac{acc(\ ) + aicc(\ ) + aocc(\ )}{ccc(\ ) + cicc(\ ) + cocc(\ )} = \frac{12 + 10 + 4}{12 + 7 + 18} = \frac{26}{37} = 0{,}7027\,.$$

Pateiktoje formulėje naudojamų funkcijų prasmės yra tokios:

- *acc()* („*Actual Component Count*“) – esamų analizuojamoje sistemoje komponentų kiekis.
- *aicc()* („*Actual Inter-Connection Count*“) – esamų analizuojamoje sistemoje komponentus siejančių ryšių kiekis.
- *aocc()* („*Actual Outer-Connection Count*“) – esamų dalykinę ir klaidų valdymo posistemes siejančių ryšių kiekis.
- *ccc()* („*Complete Component Count*“) – reikalingų analizuojamoje sistemoje komponentų kiekis.
- *cicc()* („*Complete Inter-Connection Count*“) – reikalingų analizuojamoje sistemoje komponentus siejančių ryšių kiekis.
- *cocc()* („*Complete Outer-Connection Count*“) – reikalingų dalykinę ir klaidų valdymo posistemes siejančių ryšių kiekis.

Antruoju funkcinio lygmens vertinimo atveju reikia nustatyti reikalingų ir sukurtų (suprojektuotų / realizuotų) patobulintos dalykinės srities ir klaidų valdymo posistemėse esantiems komponentams priklausančių funkcijų interfeisų, jų realizacijų ir esminių duomenų skaičių. Kadangi ne tik jokiems numatytiems analizuojamosios sistemos patobulinimams šiame darbe nebuvo sukurtos jokios kodo realizacijos, bet ir pats tų patobulinimų projektavimas nėra pilnai išbaigtas, tai reikalingas reikšmes įvertinsiu preliminariai ir ne tiksliau nei vidutiniškai, paliekant realią galimybę egzistuojančiai paklaidai. Kaip bebūtų, šiuo atveju ieškomos reikšmės yra tokios:

- Iš viso komponentuose reikalingų funkcinių interfeisų turėtų būti bent jau 3.
- Iš viso komponentuose reikalingų funkcinių realizacijų turėtų būti bent jau $39 + 37 + 41 * 2 = 158$.
- Iš viso komponentuose reikalingų esminių duomenų skaičius turėtų būti bent jau $27 + 41 = 68$.
- Turimu atveju numatytų funkcinių interfeisų skaičius komponentuose yra lygus 39.
- Turimu atveju numatytų funkcinių realizacijų skaičius komponentuose yra lygus 0.
- Turimu atveju numatytų esminių duomenų skaičius komponentuose yra lygus 27.

Turint šiuos duomenis, galima paskaičiuoti ir funkcijos $flev(\ )$ reikšmę.

$$flev(\ ) = \frac{\sum_{i=1}^{acc()} (afic(i) + afrc(i) + ada(i))}{\sum_{i=1}^{ccc()} (cfic(i) + cfrc(i) + cda(i))} = \frac{39 + 0 + 27}{3 + 158 + 68} = \frac{66}{229} = 0{,}28821.$$

Pateiktoje formulėje naudojamų funkcijų prasmės yra tokios:

- *afic()* („*Actual Functional Interface Count*“) – esamų analizuojamosios sistemos klasėse numatytų, bet nerealizuotų funkcinių interfeisų kiekis.
- *afrc()* („*Actual Functional Realization Count*“) – esamų analizuojamosios sistemos klasėse numatytų ir / arba realizuotų metodų kiekis.
- *ada()* („*Actual Data Amount*“) – esamų analizuojamosios sistemos klasėse numatytų ir / arba realizuotų duomenų kiekis.
- *cfic()* („*Complete Functional Interface Count*“) – reikalingų analizuojamosios sistemos klasėse nerealizuotų funkcinių interfeisų kiekis.
- *cfrc()* („*Complete Functional Realization Count*“) – reikalingų analizuojamosios sistemos klasėse numatytų ir / arba realizuotų metodų kiekis.
- *cda()* („*Complete Data Amount*“) – reikalingų analizuojamosios sistemos klasėse numatytų ir / arba realizuotų duomenų kiekis.

Trečiuoju algoritminio lygmens vertinimo atveju reikia nustatyti reikalingų ir sukurtų (suprojektuotų / realizuotų) komunikuojančių komponentų aibių skaičių ir įvertinti į jas įeinančių komponentų pasiekiamumo laipsnį. Vienas iš būdų skaičiuoti komunikuojančių komponentų

aibes yra pagal vartotojo ir sistemines užduotis, kurių nedetalizuotus (iki použduočių ir pan.) aprašymus pirmiausia ir pateiksiu.

Vartotojo užduotys:

1. Pinigų išėmimo iš banko sąskaitos operacija.

   Reikalingos komunikuojančių komponentų aibės:

   $$\begin{Bmatrix} ATM, Bank, Card\ \mathrm{Re}ader, Keyboard, Display, CashDispenser, \mathrm{Re}ceipt\ \mathrm{Print}er, \\ Session, Transaction, WithdrawlTransaction \end{Bmatrix}$$

   Alternatyvios komunikuojančių komponentų aibės klaidos atveju:

   $$\begin{Bmatrix} OperatorPanel, ATM, Bank, Card\ \mathrm{Re}ader, Keyboard, Display, CashDispenser, \\ \mathrm{Re}ceipt\ \mathrm{Print}er, Session, Transaction, WithdrawlTransaction \end{Bmatrix}$$

2. Pinigų padėjimo į banko sąskaitą operacija.

   Reikalingos komunikuojančių komponentų aibės:

   $$\begin{Bmatrix} ATM, Bank, Card\ \mathrm{Re}ader, Keyboard, Display, EnvelopeAcceptor, \mathrm{Re}ceipt\ \mathrm{Print}er, \\ Session, Transaction, DepositTransaction \end{Bmatrix}$$

   Alternatyvios komunikuojančių komponentų aibės klaidos atveju:

   $$\begin{Bmatrix} OperatorPanel, ATM, Bank, Card\ \mathrm{Re}ader, Keyboard, Display, EnvelopeAcceptor, \\ \mathrm{Re}ceipt\ \mathrm{Print}er, Session, Transaction, DepositTransaction \end{Bmatrix}$$

3. Pinigų pervedimo tarp sąskaitų operacija.

   Reikalingos komunikuojančių komponentų aibės:

   $$\begin{Bmatrix} ATM, Bank, Card\ \mathrm{Re}ader, Keyboard, Display, \mathrm{Re}ceipt\ \mathrm{Print}er, \\ Session, Transaction, TransferTransaction \end{Bmatrix}$$

   Alternatyvios komunikuojančių komponentų aibės klaidos atveju:

   $$\begin{Bmatrix} OperatorPanel, ATM, Bank, Card\ \mathrm{Re}ader, Keyboard, Display, \\ \mathrm{Re}ceipt\ \mathrm{Print}er, Session, Transaction, TransferTransaction \end{Bmatrix}$$

4. Informacijos apie banko sąskaitos balansą suteikimas.

   Reikalingos komunikuojančių komponentų aibės:

   $\left\{ \begin{array}{l} ATM, Bank, Card\ \mathrm{Re}ader, Keyboard, Display, \mathrm{Re}ceipt\ \mathrm{Pr}inter, \\ Session, Transaction, InquiryTransaction \end{array} \right\}$

   Alternatyvios komunikuojančių komponentų aibės klaidos atveju:

   $\left\{ \begin{array}{l} OperatorPanel, ATM, Bank, Card\ \mathrm{Re}ader, Keyboard, Display, \\ \mathrm{Re}ceipt\ \mathrm{Pr}inter, Session, Transaction, InquiryTransaction \end{array} \right\}$

Sisteminės užduotys:

1. Klaidų identifikavimas.

   Reikalingos komunikuojančių komponentų aibės:

   $\{DalykinesSritiesPosistemesKlase, ErrorDetector, GenericError\}$

   $\{KlaiduValdymoPosistemesKlase, ErrorDetector, GenericError\}$

   $\{Bank, ErrorDetector, GenericError\}$

   Alternatyvių komunikuojančių komponentų aibių klaidos atveju nėra.

2. Klaidų aptikimas.

   Reikalingos komunikuojančių komponentų aibės:

   $\{DalykinesSritiesPosistemesKlase, ErrorDetector\}$

   $\{KlaiduValdymoPosistemesKlase, ErrorDetector\}$

   $\{Bank, ErrorDetector\}$

   Alternatyvių komunikuojančių komponentų aibių klaidos atveju nėra.

3. Klaidų registravimas.

   Reikalingos komunikuojančių komponentų aibės:

   $\left\{ \begin{array}{l} DalykinesSritiesPosistemesKlase, ErrorHandler, CentralLogManager, \\ Error\ \mathrm{Pr}otocol, \mathrm{Re}source\ \mathrm{Pr}eallocator \end{array} \right\}$

   $\left\{ \begin{array}{l} KlaiduValdymoPosistemesKlase, ErrorHandler, CentralLogManager, \\ Error\ \mathrm{Pr}otocol, \mathrm{Re}source\ \mathrm{Pr}eallocator \end{array} \right\}$

$\left\{\begin{matrix} Bank, ErrorHandler, CentralLogManager, \\ Error\ \mathrm{Pr}\ otocol, \mathrm{Re}\ source\ \mathrm{Pr}\ eallocator \end{matrix}\right\}$

$\{Bank, CentralLog, \mathrm{Re}\ source\ \mathrm{Pr}\ eallocator\}$

Alternatyvių komunikuojančių komponentų aibių klaidos atveju nėra.

4. Klaidų apdorojimas.

   Reikalingos komunikuojančių komponentų aibės:

   $\{DalykinesSritiesPosistemesKlase, ErrorHandler\}$

   $\{KlaiduValdymoPosistemesKlase, ErrorHandler\}$

   $\{Bank, ErrorHandler\}$

   Alternatyvių komunikuojančių komponentų aibių klaidos atveju nėra.

5. Resursų paskirstymas.

   Reikalingos komunikuojančių komponentų aibės:

   $\{DalykinesSritiesPosistemesKlase, \mathrm{Re}\ source\ \mathrm{Pr}\ eallocator\}$

   $\{KlaiduValdymoPosistemesKlase, \mathrm{Re}\ source\ \mathrm{Pr}\ eallocator\}$

   $\{Bank, KlaiduValdymoPosistemesKlase, \mathrm{Re}\ source\ \mathrm{Pr}\ eallocator\}$

   Alternatyvių komunikuojančių komponentų aibių klaidos atveju nėra.

6. Klaidų lygiai.

   Reikalingos komunikuojančių komponentų aibės:

   $\{DalykinesSritiesPosistemesKlase, ErrorHandler, ErrorAbstractor\}$

   $\{KlaiduValdymoPosistemesKlase, ErrorHandler, ErrorAbstractor\}$

   $\{Bank, ErrorHandler, ErrorAbstractor\}$

   Alternatyvių komunikuojančių komponentų aibių klaidos atveju nėra.

7. Pranešimų parinkimas.

   Reikalingos komunikuojančių komponentų aibės:

   $\{DalykinesSritiesPosistemesKlase, \mathrm{Pr}\ anesimuValdymoKlase\}$

$\{DalykinesSritiesPosistemesKlase, ErrorHandler, ErrorDia\log, ErrorMsgMapping\}$

$\{KlaiduValdymoPosistemesKlase, \Pr anesimuValdymoKlase\}$

$\{KlaiduValdymoPosistemesKlase, ErrorHandler, ErrorDia\log, ErrorMsgMapping\}$

$\{Bank, \Pr anesimuValdymoKlase\}$

$\{Bank, ErrorHandler, ErrorDia\log, ErrorMsgMapping\}$

Alternatyvių komunikuojančių komponentų aibių klaidos atveju nėra.

8. Pranešimų generavimas.

Reikalingos komunikuojančių komponentų aibės:

$\{DalykinesSritiesPosistemesKlase, \Pr anesimuValdymoKlase\}$

$\{DalykinesSritiesPosistemesKlase, ErrorHandler, ErrorDia\log, ErrorMsgMapping\}$

$\{KlaiduValdymoPosistemesKlase, \Pr anesimuValdymoKlase\}$

$\{KlaiduValdymoPosistemesKlase, ErrorHandler, ErrorDia\log, ErrorMsgMapping\}$

$\{Bank, \Pr anesimuValdymoKlase\}$

$\{Bank, ErrorHandler, ErrorDia\log, ErrorMsgMapping\}$

Alternatyvių komunikuojančių komponentų aibių klaidos atveju nėra.

9. Centrinės banko sistemos informavimas apie klaidos situaciją.

Reikalingos komunikuojančių komponentų aibės:

$$\left\{\begin{matrix} DalykinesSritiesPosistemesKlase, ErrorHandler, CentralLogManager, Error\Pr otocol, \\ Bank, CentralLog \end{matrix}\right\}$$

Alternatyvių komunikuojančių komponentų aibių klaidos atveju nėra.

10. Sistemos reaktyvavimas.

Reikalingos komunikuojančių komponentų aibės:

$\{ATM, DalykinesSritiesPosistemesKlase, OperatorPanel\}$

$\{ATM, KlaiduValdymoPosistemesKlase, OperatorPanel\}$

Alternatyvių komunikuojančių komponentų aibių klaidos atveju nėra.

Įvardinus numatytas komunikuojančių komponentų aibes, galima nustatyti ir likusius trūkstamus duomenis.

- Iš viso komunikuojančių komponentų aibių turėtų būti bent jau 38 ($reach() = 160{,}8$).
- Iš viso alternatyvių komunikuojančių komponentų aibių turėtų būti bent jau 4 ($areach() = 71{,}6$).
- Turimu atveju komunikuojančių komponentų aibių skaičius lygus 18 ($reach() = 96$).
- Turimu atveju komunikuojančių komponentų aibių skaičius lygus 4 ($areach() = 71{,}6$).

Dabar jau galime suskaičiuoti ieškomą funkcijos $glev(\ )$ reikšmę.

$$glev(\ ) = \frac{\sum_{i=1}^{arc()} (a\csc(i) \times reach(i) + aa\csc(i) \times areach(i))}{\sum_{i=1}^{trc()} (c\csc(i) \times reach(i) + ca\csc(i) \times areach(i))} = \frac{18*96 + 4*71{,}6}{38*160{,}8 + 4*71{,}6} = \frac{2014{,}14}{6396{,}8} = 0{,}31487$$

Pateiktoje formulėje naudojamų funkcijų prasmės yra tokios:

- *acsc()* („*Actual Communication Scenario Count*“) – realizuotų analizuojamoje sistemoje klasių komunikuojančių komponentų aibių kiekis.
- *aacsc()* („*Actual Alternative Communication Scenario Count*“) – realizuotų analizuojamoje sistemoje alternatyvių komunikuojančių komponentų aibių kiekis.
- *ccsc()* („*Complete Communication Scenario Count*“) – reikalingų analizuojamoje sistemoje komunikuojančių komponentų aibių kiekis.
- *cacsc()* („*Complete Alternative Communication Scenario Count*“) – reikalingų analizuojamoje sistemoje alternatyvių komunikuojančių komponentų aibių kiekis.
- *reach(), areach()* – į komunikuojančių komponentų aibes įeinančių klasių panaudojamumo vienas kito atžvilgiu lygis.

Turint visus reikalingus duomenis, galima rasti ir galutinę projektinių sprendimų vertinimo (*DFL* („*Design Fulfilment Level*“)) reikšmę.

$$DFL = arev(\ ) + flev(\ ) + glev(\ ) = 0{,}7027 + 0{,}28821 + 0{,}31487 = 1{,}30578\,.$$

Gautieji rezultatai ne tik patvirtina anksčiau minėtą argumentą, kad naujosios analizuojamosios sistemos versijos projektavimas nėra pilnai išbaigtas, bet ir tai, jog net ir pati klaidų valdymo posistemės projektinė realizacija nėra pilnai išdirbta.

Kadangi realizacijos vertinimo (*IL* („*Implementation Level*“)) skaičiavimui nereikia jokių papildomų paruošiamųjų darbų, tai jo reikšmę galima paskaičiuoti iš karto ir ji bus lygi:

$$IL = \frac{arsl()}{crsl()} = \frac{6}{15} = 0{,}4\,.$$

Formulėje naudojama funkcija *arsl*() („*Actual Real State Level*“) yra realus parinktas programų sistemos įgyvendinimo lygis, funkcija *crsl*() („*Complete Real State Level*“) – galutinis programų sistemos įgyvendinimo lygis.

Gautoji reikšmė atspindi lygį, kuris labiausiai tinka praktiniam naujosios analizuojamos sistemos versijos kūrimo išbaigtumui nusakyti pagal šiame darbe atliktą darbą. Be abejo, tokio vertinimo skalė apima visą programų sistemų gyvavimo ciklą, o šiame darbe atlikti darbai buvo orientuoti tik į projektavimo etapą.

Atlikus keturių pagrindinių *RCEM* vertinimo metodikos aspektų vertinimą, galima paskaičiuoti ir galutinę šiame darbe pasiūlytų patobulinimų realizavimo laipsnio (*RFL* („*Refinement Fulfilment Level*“)) reikšmę.

# 6. APIBENDRINIMAS IR REZULTATAI

Šiame skyrelyje bus pateiktos bendros jungtinio darbo išvados, kurios bus padarytos, lyginant Lino Šimkaus ir šiame darbe gautus rezultatus, pritaikius kiekvieno pasiūlytą *PCMB* arba *RCEM* vertinimo metodiką. Tam tikslui pirmiausia pateiksiu individualiai gautų rezultatų suvestinę, nurodytą 22 lentelėje.

**22 lentelė.** Bendra individualiai gautų rezultatų suvestinė.

| Nr. | Parametras | Mariaus Ašmono gautoji rezultato reikšmė | Lino Šimkaus gautoji rezultato reikšmė |
|---|---|---|---|
| ***PCMB* vertinimo metodika** | | | |
| 1. | $PC$ | $\frac{TE_B - 12}{(TE_B - 12) + (\Delta TE - 40)}$ | $\frac{TE_B - 12}{(TE_B - 12) + (\Delta TE - 40)}$ |
| 2. | $EVR_B$ | 0,05714 | 0,06208 |
| 3. | $EVR_A$ | 0,13129 | 0,13527 |
| 4. | $EC$ | 0,43522 | 0,43893 |
| 5. | $OPUS_B$ | 28 | 28 |
| 6. | $OPUS_A$ | 104 | 83 |
| 7. | $CC$ | 0,26923 | 0,33735 |
| 8. | $IMPCOV_B$ | 0,72591 | 0,72591 |
| 9. | $IMPCOV_A$ | 3,08644 | 2,37153 |
| 10. | $OC$ | 0,23519 | 0,30609 |
| ***RCEM* vertinimo metodika** | | | |
| 1. | $SC$ | 0,32 | 0,51852 |
| 2. | $AC$ | 0,54 | 0,83333 |
| 3. | $PPL$ | 0,1728 | 0,4321 |
| 4. | $IRIL$ | 0,45 | 0,6 |
| 5. | $arev(\ )$ | 0,7027 | 0,89326 |
| 6. | $flev(\ )$ | 0,28821 | 0,20112 |

| | | | |
|---|---|---|---|
| 7. | $glev(\ )$ | 0,31487 | 0,5507 |
| 8. | *DFL* | 1,30578 | 1,64508 |
| 9. | *IL* | 0,4 | 0,4 |
| 10. | *RFL* | 2,32858 | 3,07718 |

Atsižvelgiant į *PCMB* vertinimo metodikos taikymo metu gautus rezultatus, matyti, kad planinio koeficiento ( *PC* ) reikšmės abiejuose darbuose yra vienodos. Tai yra teisinga, nes darbo metu buvo remiamasi ta pačia identifikuotų klaidų aibe. Bet kuriuo atveju gauta reikšmė patvirtina, kad naujoji analizuojamos sistemos versija abiejuose darbuose yra labiau pasiruošusi įvairioms galimoms klaidų situacijoms dėl didesnio apdorojimui numatyto jų kiekio.

Iš $EVR_B$ ir $EVR_A$ reikšmių matyti, kad Lino Šimkaus darbe nagrinėtoje naujojoje analizuojamos sistemos versijoje (lyginant su senąja versija) klaidų tikrinimo atitikimas pilnam tikrinimo planui yra ~0,95652 karto mažesnis nei šiame darbe (pirmuoju atveju klaidų tikrinimo atitikimo pilnam tikrinimo planui skirtumas tarp versijų yra ~2,2 karto, o antruoju atveju – ~2,3 karto). O viena iš galimų priežasčių, dėl ko Lino Šimkaus darbe $EVR_B$, $EVR_A$ ir $EC$ reikšmės yra didesnės už šiame darbe gautąsias reikšmes, gali būti ne tik dėl skirtingų priimtų architektūrinių ir funkcinių sprendimų, bet ir dėl kitokio apsibrėžto pilno tikrinimo metodo ypatumų.

Iš $OPUS_B$ ir $OPUS_A$ reikšmių matyti, kad Lino Šimkaus darbe nagrinėtos naujosios analizuojamos sistemos versijos (lyginant su senąja versija) pajėgumas palaikyti ir užtikrinti operacinį darbą klaidų apgulties atveju yra~1,25 karto mažesnis nei šiame darbe (pirmuoju atveju minėtojo pajėgumo skirtumas tarp versijų yra lygus ~2,96 karto, o antruoju atveju – ~3,7 karto). Dėl to atitinkamai ir pati pajėgumo koeficiento *CC* reikšmė Lino Šimkaus darbo atžvilgiu yra didesnė už šiame darbe gauto pajėgumo koeficiento reikšmę. Tikslias konkrečias priežastis, dėl ko taip atsitiko, įvardinti yra gana sunku, tačiau bet kuriuo atveju viskas susiveda į pasirinktų projektavimo sprendimų (tokių kaip architektūros planavimas, funkcinių galimybių nustatymas, sąveikos našumas ir pan.) efektyvumą.

Na, o pakankamumo koeficiento reikšmės skaičiavimo rezultatai rodo, kad Lino Šimkaus darbe nagrinėtoje naujojoje analizuojamos sistemos versijoje (lyginant su senąja versija) į klaidų

valdymą orientuotos priemonės yra ~0,76744 karto mažiau išvystytos (ir dėl to tikėtina, kad silpnesnės) nei šiame darbe (pirmuoju atveju į klaidų valdymą orientuotų priemonių našumo skirtumas tarp versijų yra lygus ~3,3 karto, o antruoju atveju – ~4,3 karto). O pati didesnė galutinė pakankamumo koeficiento reikšmė Lino Šimkaus darbo atžvilgiu kaip tik ir parodo, kad jo suformuotas į klaidų valdymą orientuotas priemones dar reikia patobulinti ir tai pasiekti jis turi daugiau galimybių, nes jo priemonės dar nėra pilnai išvystytos ir išbaigtai adaptuotos jo naujosios analizuojamos sistemos architektūros variantui.

Atsižvelgiant į *RCEM* vertinimo metodikos taikymo metu gautus rezultatus, matyti, kad klaidų valdymo modelio pritaikymui ir išvystymui Lino Šimkaus darbe reikėjo didesnio tiek teorinio, tiek ir praktinio pasiruošimo nei šio darbo atžvilgiu. Reali situacija šį faktą iš tikrųjų patvirtina. Gautasis skirtumas (skaičiuojant pagal nurodytas teorinės bazės poreikių vertinimo reikšmes) yra lygus ~2,5.

Iš reikalavimų išpildymo vertinimo reikšmės $IRIL$ matyti, kad Lino Šimkaus darbe atliktos analizės ir padarytų senosios analizuojamos sistemos versijos patobulinimų metu buvo įgyvendinta daugiau iškeltų reikalavimų nei šiame darbe. Gautasis skirtumas yra lygus ~0,75 karto.

Iš $erev(\ )$, $flev(\ )$ ir $glev(\ )$ reikšmių matyti, kad Lino Šimkaus darbe architektūrinė realizacija yra pilnesnė nei šiame darbe (skirtumas pagal pirmosios funkcijos reikšmes yra lygus ~0,78667 karto). Vis dėlto ta pati architektūrinė realizacija funkciniu požiūriu yra silpniau suplanuota ir dėl to analizuotame projektavimo lygmenyje mažiau išvystyta (skirtumas pagal antrosios funkcijos reikšmes yra lygus ~0,69782 karto). Algoritminė realizacija, iš kitos pusės, yra labiau išpildyta, t.y. iš visų pagal iškeltus reikalavimus reikalingų komunikuojančių komponentų didžioji dalis naujojoje analizuojamoje sistemos architektūroje yra numatyta ir turinti didesnį sudėtingumo (ryšių atžvilgiu) laipsnį (skirtumas pagal trečiosios funkcijos reikšmes yra lygus ~0,57176 karto). Dėl to ir galutinė projektinių sprendimų vertinimo koeficiento $DFL$ reikšmė Lino Šimkaus atveju yra ~0,79375 karto didesnė nei šiame darbe.

Realizacijos vertinimo koeficiento $IL$ reikšmė abiejų darbų atžvilgiu yra vienoda ir teigianti, kad lygis, kuris labiausiai tinka praktiniam naujosios analizuojamos sistemos versijos kūrimo išbaigtumui nusakyti, galėtų būti įvardijamas kaip „*Nebaigtas projektavimas*“ (*INCOMPLETE_DESIGN*), kuris iš penkiolikos apibrėžtų lygių yra šeštoje vietoje. O tai reiškia, kad pagal programų sistemų industrijoje vyraujančias tokių produktų kūrimo tendencijas

naujosios analizuojamos sistemos versijos tobulinimas nepraėjo nei 50% visų savo gyvavimo ciklo etapų.

Na, o suminė *RCEM* vertinimo metodikos koeficiento *RFL* reikšmė rodo, kad reziumuojant visus gautus rezultatus, galima padaryti išvadą, kad Lino Šimkaus darbas, orientuotas į senosios analizuojamos sistemos versijos klaidų valdymo tobulinimą, yra ~0,75673 karto labiau išbaigtas projektavimo atžvilgiu, tačiau minimum ~0,86813 karto (pagal *PCMB* vertinimo metodikos gautus duomenis) silpnesnis atsparumo klaidoms atžvilgiu.

# IŠVADOS

Visų pirma, tiek pats atlikto darbo vertinimas, tiek ir visi gauti rezultatai dar kartą patvirtino faktą, kad panašią kompetenciją nagrinėjamoje dalykinėje ir probleminėje srityje turinčių žmonių gaunamų rezultatų skirtumas dažniausiai nebūna labai didelis. Šiuo atveju tai įrodė mūsų su kolega numintas darbo kelias, davęs rezultatus, kurie nei vienu iš tirtų probleminių aspektų nebuvo tarpusavyje labai skirtingi.

Pasirinkto klaidų valdymo modelio taikymas šiame darbe pateisino jo pasirinkimo argumentų teisingumą. Didžiausias jo privalumas yra tas, kad jis tikrai yra pakankamai bendras bent jau tiek, kad galėtų būti naudojamas įvairiose programų sistemų kūrimo praktikose kaip medžiaga, pasakanti, kas turi būti padaryta ir realizuota, norint programų sistemoje turėti efektyviai sukurtą į klaidų valdymą orientuotą funkcionalumą. Vis dėlto bene didžiausias jo minusas kol kas vis tik būtų per menkas modelio detalumas, kuris einamuoju momentu nepasako nieko nei apie jokius konkrečius reikalavimus, kurie galėtų būti suformuluoti kiekvienai modelį sudarančiai probleminei sričiai, nei apie kokias nors darbo metodikas ar priemones, kuriomis galima būtų remtis, nei apie patį darbo procesą ar jo struktūrą. Atsiradus tokiems papildymams, klaidų valdymo modelis būtų jau nebe vien teorinis karkasas, bet ir vedlys, lydintis kūrėjus nuo programų sistemų kūrimo projekto startavimo pradžios iki pat programinio produkto išleidimo, eksploatavimo ar net demontavimo momento.

Darbe pristatyta programų sistemos architektūros atsparumo klaidoms vertinimo metodika yra paremta empirine analize, o tai reiškia, kad vertinimo metodikos taikymo kontekste visos išvados gali būti ir yra tik tikėtinos. Dėl to pats vertinimo procesas yra daugiau ar mažiau panašus tik į savotišką prognozavimą, bet ne į tikslių faktų konstatavimą, ir remiasi grynai vertintojo (eksperto) individualia kompetencija.

# ŠALTINIŲ SĄRAŠAS


[AH05] N. Arshad, D. Heimbigner. A Comparison of Planning Based Models for Component Reconfiguration, URL: http://www.cs.colorado.edu/department/publications/reports/docs/CU-CS-995-05.pdf, 127 KB, 2005.

[BJC05] T. Batista, A. Joolia, G. Coulson. Managing Dynamic Reconfiguration in Component Based Systems, URL: http://www.comp.lancs.ac.uk/~geoff/Publications/EWSA05.pdf, 97,2 KB, 2005.

[Bjo97] R. C. Bjork. ATM Simulation Code, žiūrėta: 2008-03-20, URL: http://www.math-cs.gordon.edu/courses/cs320/ATM_Example, 1997.

[Boo03] L. Boonard. Error Handling Strategies, URL: http://users.iptelecom.net.ua/~agp1/arts/errhdl_2.pdf, 191 KB, 2003.

[BMR+96] F. Buschmann, R. Meunier, H. Ronhert, P. Sommerland, M. Stal. Pattern Oriented Software Architecture, A System of Patterns, Wiley 1996.

[Chro05] D. B. Chromek. Fault Tolerance and Error Handling Fault Tolerance and Error Handling in the TDAQ & Detectors in the TDAQ & Detectors system, URL: http://atlas.web.cern.ch/Atlas/GROUPS/DAQTRIG/EHFT/docs/FaultTolErrorHandlingDec02.pdf, 1.38 MB, 2002.

[ERR+01] E. Eide, A. Reid, J. Regehr, J. Lepreau. Static and Dynamic Structure in Design Patterns. Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), November 1, 2001.

[GBX+00] F. Di Giandomenico, A. Bondavalli, J. Xu, S. Chiaradonna. Hardware And Software Fault Tolerance: Definition And Evaluation Of Adaptive Architectures In A Distributed Computing Environment, URL: http://bonda.cnuce.cnr.it/Documentation/Papers/file-DGBXC97-ESREL-110.pdf, 73,2 KB, 2000.

[GHJ+95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[LSK+00] Z. Luo, A. Sheth, K. Kochut, J. Miller. Exception Handling in Workflow Systems. Applied Intelligence, Volume 13, Number 2. September/October, 2000: pp. 125-147.

[LW04] A. Longshaw, E. Woods. Patterns for Generation, Handling and Management of Errors, URL: http://www.blueskyline.com/ErrorPatterns/A2-LongshawWoods6.pdf, 307 KB, 2004.

[Mar07] M. Kwiatkowska. Quantitative Verification: Models, Techniques and Tools, URL: http://qav.comlab.ox.ac.uk/papers/esec-fse07.pdf, 314 KB, 2007.

[NBS00] S. Neema, T. Bapty, J. Scott. Adaptive Computing and Runtime Reconfiguration, URL: http://klabs.org/richcontent/MAPLDCon99/Papers/E6_Neema_P.pdf, 1,64 MB, 2000.

[Pet01] B. Pettichord. Design for Testability, URL: http://www.io.com/~wazmo/papers/design_for_testability.pdf, 92 KB, 2001.

[Ren03] K. Renzel. Error Handling for Business Information Systems. A Pattern Language, URL: http://www.eso.org/projects/alma/develop/acs/OnlineDocs/ARCUSErrorHandling.pdf, 815 KB, 2003.

[Soi04] J. P. Soininen. Architecture Design Methods for Application Domain-Specific Integrated Computer Systems, VTT Publications 523, 2004.

[Voj05] D. Vojevodina. Exception Handling Automation in E-business Workflow Processes, URL: http://lbd.epfl.ch/e/conferences/caise05dc/Final_version/Caise05dovile.pdf, 225 KB, 2005.

[VPM+03] S. A. Vilkomir, D. L. Parnas, V. B. Mendiratta, E. Murphy. Availability evaluation of hardware/software systems with several recovery procedures, URL: http://iex.org/iel5/10076/32335/01510068.pdf, 92 KB, 2003.

# SĄVOKŲ APIBRĖŽIMAI

1. „*Automated Teller Machine (ATM) – An unmanned electronic device that performs basic teller functions, such as accepting deposits, cash withdrawls, and account balance inquiries*“.
   Automatiškai valdomas įrenginys, kuris atlieka pagrindines banko kasininko funkcijas, tokias kaip piniginių indėlių priėmimas, pinigų išėmimas, užklausų apie sąskaitos balansą suteikimas ir pan.
   (www.agriculturefcu.org/glossary.html)

2. Struktūrinis adaptyvumo sistemos („*Structurally Adaptive Systems*“) – tai programų sistemos, kurios veikimo metu, nenutraukdamos darbo, gali koreguoti nuosavas architektūrines struktūras ([NBS00]).

3. Dinaminė rekonfigūracija („*Dynamic Reconfiguration*“) – tai dinamiškai atliekamų komponentų įkėlimo, pozicijos pakeitimo arba iškėlimo veiksmų procesas, kurio metu programų sistema gali greitai prisitaikyti prie kintančių sąlygų be jokių jos korektiškam veikimui žalingų pasekmių ([NBS00]).

# PRIEDAI

Pavyzdinės analizuojamosios *ATM* bankomato sistemos programinis kodas yra pateiktas prie šio darbo prisegtame kompaktiniame diske. Originalią programinio kodo kopiją galima parsisiųsti adresu, nurodytu šaltinių sąraše ties nuoroda [Bjo97].