KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

VERSLO INFORMATIKOS KATEDRA

Tadas Tauginas

FORMALIŲ METODŲ PANAUDOJIMAS KURIANT VERSLO VALDYMO SISTEMAS MSBS-NAVISION TERPĖJE

Magistro darbas

Darbo vadovas:
prof. habil. dr. Henrikas Pranevičius

KAUNAS
2004

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

VERSLO INFORMATIKOS KATEDRA

TVIRTINU
Katedros vedėjas
prof. habil. dr. Henrikas Pranevičius
2004 05 25

FORMALIŲ METODŲ PANAUDOJIMAS KURIANT VERSLO VALDYMO SISTEMAS MSBS-NAVISION TERPĖJE

Informatikos mokslo magistro baigiamasis darbas

Lietuvių kalbos konsultantė
dr. J. Mikelionienė
2004 05 20

Vadovas
prof. habil. dr. Henrikas Pranevičius
2004 05 20

Recenzentas
doc. dr. Bronius Paradauskas
2004 05 24

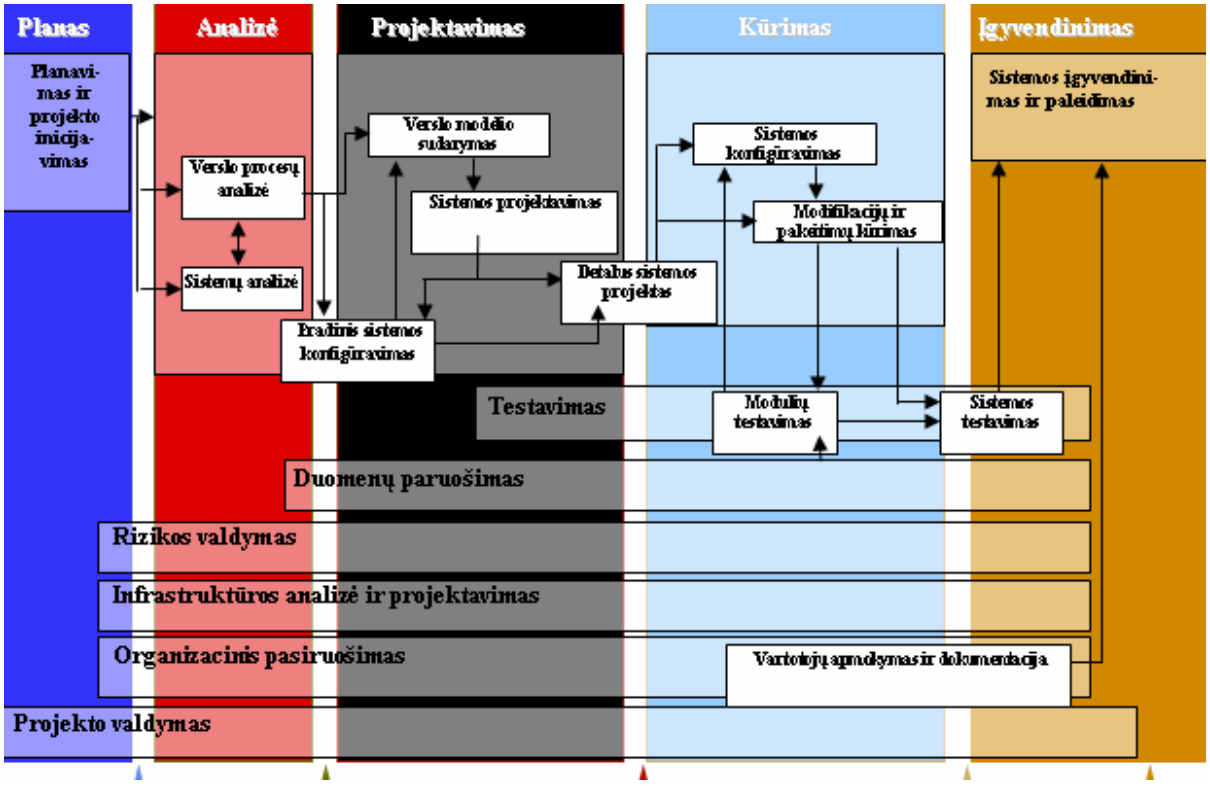
Atliko
IFM 8-1 gr. stud. T. Tauginas
2004 05 25

KAUNAS
2004

TURINYS

1. ĮVADAS

MS Navision – verslo valdymo sistema, skirta stambiam ir vidutiniam verslui. Šioje sistemoje pagrindinės verslo ir apskaitos funkcijos yra jau sukurtos, tačiau sistema yra lengvai pritaikoma konkrečios įmonės poreikiams. Adaptuojant galima keisti standartines verslo funkcijas dėl įmonės veiklos specifikos, taip pat įdiegti naujas. Retai įmonei tinka standartinis verslo valdymo funkcijų paketas. Todėl kuriant ir diegiant sistemą yra pereinama per visus tokios programinės įrangos kūrimo etapus: analizę, projektavimą, programavimą, ir testavimą.



1 pav. MSBS Navision siūloma sistemos kūrimo eiga

1 pav. yra pavaizduota visa projekto eiga. Analizės etape sistema yra specifikuojama natūralia kalba. Paruošiamas funkcinis reikalavimų dokumentas (FRD). Projektavimo etape paruošiama sistemos programos architektūra, sudaromas verslo modelis. Tam tikslui yra naudojama UML (*Unified Modeling Language*) notacija. Verslo valdymo sistemų diegimo praktikoje UML realiai naudojama tik procesams aprašyti. Formalūs metodai praktikoje nenaudojami. Dėl to, kad dažniausiai yra pritaikoma vienas ar kitas prototipas, likusi dalis specifikacijos yra paliekama žodinėj formoj. Tokiu būdu sistema lieka pilnai neapibrėžta.

Programuojant kyla daug neaiškumų. Programuotojas turi pats interpretuoti sąlygą, tuo galbūt pakenkdamas bendram sistemos darnumui. Dažnai tik programavimo etape būna pastebimos loginės sistemos klaidos. Visa tai sąlygoja išaugusias pradžioj numatytas verslo valdymo sistemos išlaidas bei diegimo laiką. Siekiant to išvengti reikia tiksliau specifikuoti sistemą. Norint tai padaryti reiktų taikyti formalius specifikavimo metodus.

Egzistuoja daug formalių specifikavimo kalbų, pvz., VDM, Z, B, OCL. Konkrečiai šiam darbui bus pasirinkta Z (tariasi Zet) kalba.

Tačiau specifikuoti sistemą vien Z kalba nėra tikslinga. Nes šis formalus metodas negali grafiškai parodyti procesų eigos. Taip pat UML leidžia grafiškai pavaizduoti daugelį sistemos savybių bei būsenų. O grafiškas žymėjimas yra daug lengviau suprantamas. Taigi visiškai atmesti UML negalima.

MSBS Navision kūrėjai siūlo visai nenaudoti formalių metodų. Jie siūlo dažniau komunikuoti; projektui priskirti bent vieną patyrusi konsultantą, kuris galėtų spręsti neaiškumus bei nesutarimus. Tačiau praktika rodo, kad Lietuvoje retas konsultantas gali padėti spręsti tokias problemas. Taigi geriausia būtų iš karto paruošti kuo tikslesnę verslo valdymo sistemos ar jos priedo specifikaciją, kad vėliau būtų išvengta nereikalingų laiko ir resursų sąnaudų.

Šio darbo tikslas yra palyginti UML ir Z specifikavimo kalbas. Ištirti jų teorinį pagrindą. Išsiaiškinti jų pranašumus ir silpnąsias vietas. Pagrindiniai palyginimo kriterijai bus aiškumas ir tikslumas, bei realaus praktinio panaudojimo galimybės. Taip pat palyginti verslo funkcijų realizavimą MSBS Navision terpėje, remiantis skirtingomis specifikacijomis. Įrodyti formalių metodų naudą, siekiant kuo tiksliau ir aiškiau aprašyti verslo valdymo sistemų modelius.

2. VERSLO VALDYMO SISTEMŲ SPECIFIKAVIMO METODŲ ANALIZĖ

2 pav. Analizės eiga

MSBS Navision rekomenduojama metodologija siūlo analizės etape identifikuoti kliento verslo funkcijas ir jas aprašyti paprasta kalba. Žemiau yra pateikta pavyzdinės įmonės ilgalaikio turto apskaitos tekstinė specifikacija:

Ilgalaikis turtas

Reikalavimas: kaupti pagrindinę ilgalaikio turto informaciją.

IT kortelė

Sprendimas:

Visa pagrindinė informacija susijusi su ilgalaikiu turtu Navision Attain turi registruoti IT kortelėje.

Reikalavimas:

IT kortelės istorija

Sprendimas:

Navision Attain matyti ilgalaikio turto kortelės istorija (kokie veiksmai buvo atlikti su IT). IT įsigijimo, nusidėvėjimo, perkainavimo istorija atsispindės DK ir IT žurnaluose. IT perdavimo istorijai peržiūrėti bus programuojamas atskiras sprendimas.

Reikalavimas:

IT nusidėvėjimo sąnaudų paskirstymas

Sprendimas:

Galimybė Navision Attain IT nusidėvėjimo sąnaudas paskirstyti pagal skirtingus padalinius procentais arba rankiniu būdu.

Reikalavimas:

Galimybė keisti nusidėvėjimo normatyvus

Sprendimas:

Navision Attain galimybė keisti nusidėvėjimo normatyvus (normos keitimas, kurį laiką nudėvėjus turtą taikant kitą normą).

Reikalavimas:

Galimybė skaičiuoti nusidėvėjimą pagal IT grupes

Sprendimas:

Navision Attain nusidėvėjimo skaičiavimo galimybė ne tik pagal registravimo grupes, bet ir pogrupius.

Stambesnius procesus galima atvaizduoti grafiškai naudojant UML notaciją. Smulkesnė specifikacija yra nukeliama į projektavimą. Tačiau projekto kaina turi būti paskelbiama jau analizės etape. Todėl teoriškai analitikas turi įsivaizduoti visas projekto darbo sąnaudas. Vien iš tekstinės specifikacijos tai padaryti yra be galo sunku, todėl dažnai apsirinkama. Ribojant sąnaudas netgi projektavimo etape nėra sukuriamas visiškai aiškus ir tikslus būsimų funkcijų algoritmas. Esant tokioms sąlygoms klientai dažnai keičia reikalavimus ir tuo pačiu kenkia ne tik projekto sėkmingam įgyvendinimui, bet ir patys sau. Prarandama nustatyta tvarka. Kol informacija susisteminama ir pasiekia programuotojus, kliento noras gali pasikeisti. Taigi siekiant to išvengti reikėtų svarbesnes specifikacijos dalis kiek galima detaliau ir tiksliau aprašyti. Nors verslo valdymo sistema ir nereikalauja kritinio saugumo, bet formalizuojant tekstines specifikacijas būtų galima išvengti daug iki šiol atsirandančių nesklandumų. Svarbiausia sistemos dalis paprastai daugiausia ir kainuoja. Taigi ją pastoviai modifikuojant ir kaitaliojant, po truputį prarandamas numatytas projekto valdymas. Sudarius formalią specifikaciją, būtų galima

patikrinti sistemos veikimą ankstyvose jos kūrimo stadijose, realiai įvertinti kaštus, įgyvendinimo galimybes (1).

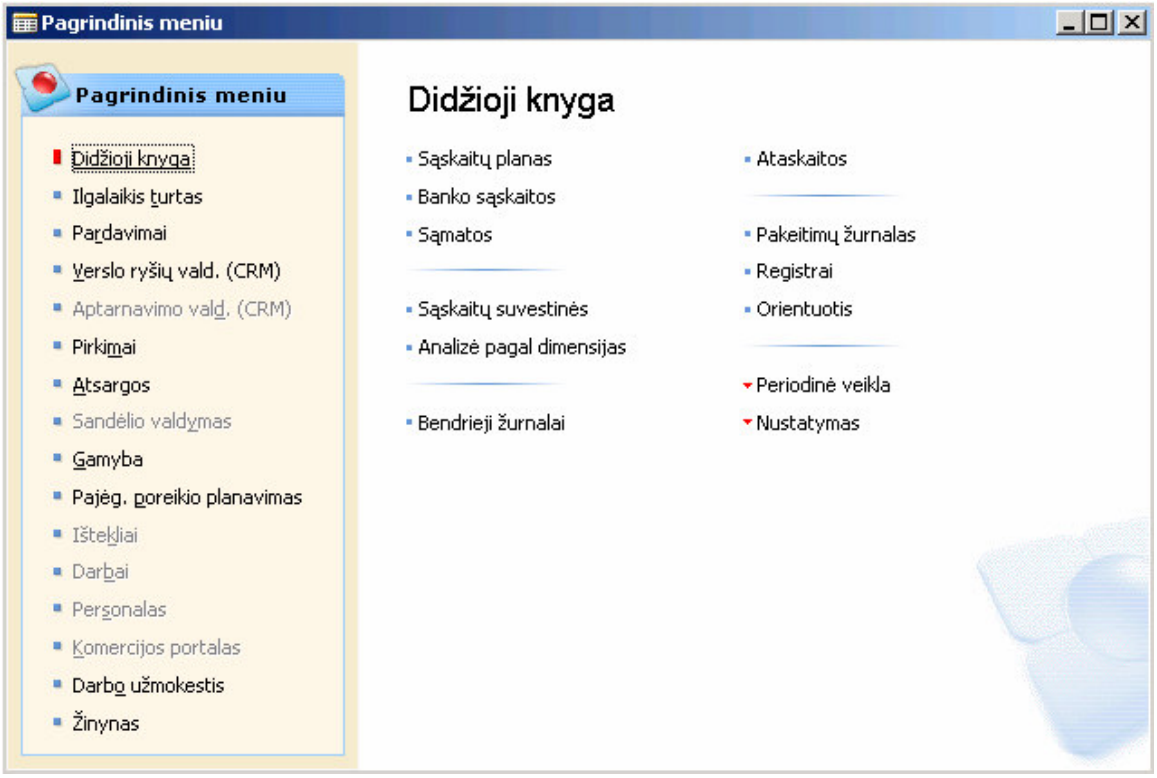
Konkrečiu atveju yra pasirinkta formalaus specifikavimo "Z" kalba. Ji sukurta remiantis aibių teorija ir predikatų logika. Matematiniai duomenų tipai puikiai tinka aprašyti duomenims sistemoje, o predikatų logika yra naudojama nusakyti operacijoms, vykdomoms su tais duomenimis. "Z" kalba nespecifikuoja, "kaip" turi būti padaryta sistema, bet "kas" turi būti padaryta (1). Stambūs procesai gali būti skaidomi į mažesnes funkcijas, kurios savo ruožtu į dar smulkesnes. Taigi yra palaikoma ir struktūrinė schema. Funkcijos gali būti verifikuojamos.

MSBS Navision nesiūlo sprendimų nuo nulio. Klientui susipažinti yra pateikiamas verslo valdymo funkcijų paketo prototipas. Aplink jį klientas „lipdo" savo pageidavimus ir vizijas. Todėl reikalinga formaliai specifikuoti jau paruoštas verslo valdymo funkcijas taip pat kaip ir naujas. Formaliai specifikavus jau suprogramuotas VVS funkcijas, būtų galima daug lengviau ir greičiau atlikti kliento pageidaujamas modifikacijas. Taip pat būtų iš karto matoma, ar naujos funkcijos nepersikirs su senomis funkcine ir darnumo prasmėmis.

Konkrečiam atvejui tirti yra pasirinkta MSBS Navision "Ilgalaikio Turto" funkcinė sritis. Įvertinus MSBS Navision "Ilgalaikio Turto" srities siūlomas jau suprogramuotas verslo valdymo funkcijas ir iš anksčiau pateiktos pavyzdinės įmonės ilgalaikio turto apkaitos neformalią specifikaciją yra matoma, kad naujas funkcionalumas yra: "IT mokesčio paskaičiavimas", "IT įsigijimo sumos paskaičiavimas", "IT informacijos keitimas (nenusižengiant bendram sistemos darnumui), "Keitimo informacijos kaupimas". Visa kita yra padaroma jau papildant egzistuojančias funkcijas ir duomenų modelius. Taigi norit gerai ir greitai specifikuoti papildymą, ir ypač jei jis yra sudėtingas, reikia turėti jau formaliai specifikuotą standartą. Formali standarto specifikacija būtų naudojama diegiant vis naują sistemą. Taigi formalios standarto specifikacijos sąnaudos būtų vienkartinės.

Šiuo atveju reikia specifikuoti MSBS Navision "Ilgalaikio Turto" funkcinę sritį, t. y., koks duomenų modelis yra pasirinktas ilgalaikio turto informacijai saugoti. Taip pat reikia formaliai specifikuoti IT apskaitos funkcijas: registravimą iš žurnalų į didžiąją knygą, registravimą iš žurnalų į ilgalaikio turto knygą, nusidėvėjimo skaičiavimą. Kliento reikalavimus specifikuoti atsižvelgiat į jau sudarytą ilgalaikio turto funkcinės srities specifikaciją.

Standartinės MSBS Navision verslo valdymo funkcijos yra suskirstytos į tokias funkcines sritis: „Didžioji knyga", „Ilgalaikis turtas", „Pirkimai", „Pardavimai" ir kt.

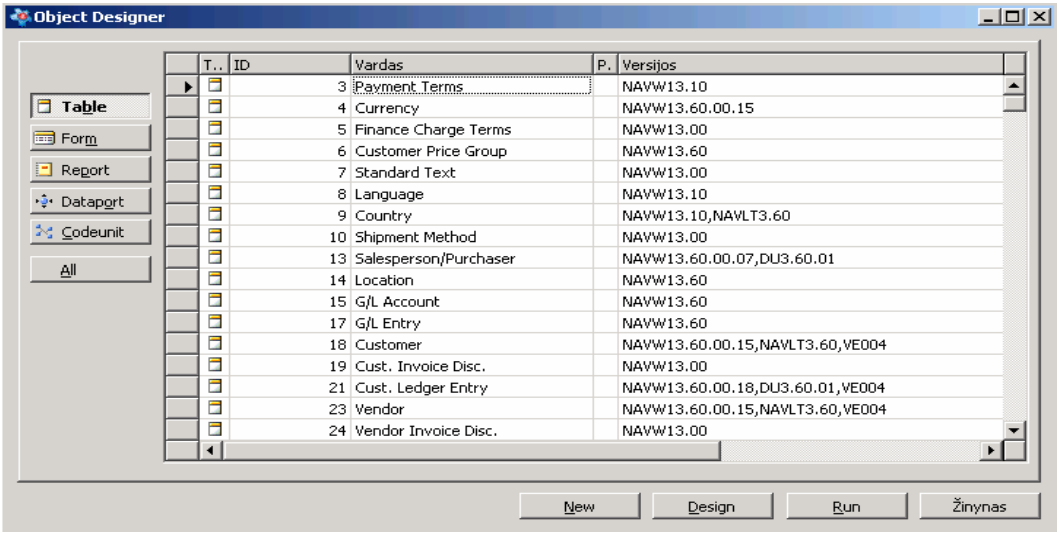


3 pav. MSBS Navision pagrindinis meniu

Kiekviena sritis turi specifines funkcijas. Siekiant išlaikyti vientisumą sistemoje visų funkcinių sričių meniu ir struktūra yra panašios. Pvz., visose srityse yra ataskaitų, periodinės veiklos bei nustatymų skyriai.

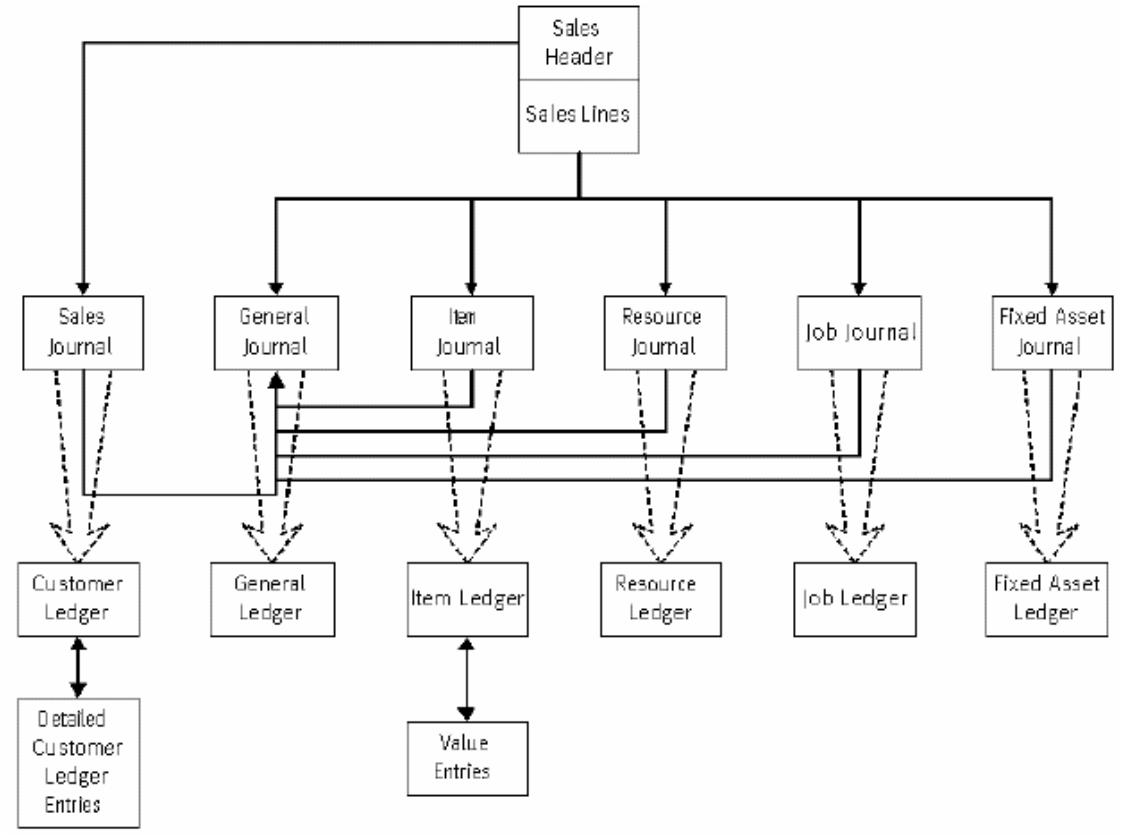
Navision sistemoje visi duomenys yra saugomi Navision duomenų bazėje, kurios duomenų struktūra yra panaši į kitų reliacinių duomenų bazių. Funkcionalumas yra realizuojamas specifiniais objektais. Yra penki objektų tipai:

Lentelės (kuriose ir saugomi duomenys), formos (vartotojo grafinė sąsaja), ataskaitos, duomenlaidės (skirtos duomenų importui ir eksportui iš/į sistemą), kodiniai (įvairūs papildomi programų tekstai).



4 pav. Objektų redakotrius

Visa apskaita realizuojama per žurnalus, t. y. į visas apskaitos knygas informacija anksčiau ar vėliau patenka registruojant konkrečios apskaitos srities žurnalą. Sakykim vartotojas užpildė pardavimo sąskaitą faktūrą ir ją registruoja. Informacija neina tiesiai į pirkėjo ir didžiąją knygas. Iš tiesų programa performuoja informaciją į pardavimų žurnalą ir tik tada jį registruoja. Sekančioj schemoj yra parodyta “Pardavimų “ srities principinė veikimo schema. Kiekvienas struktūrinis elementas yra lentelė. Taigi registruojant, realiai informacija yra imama tik iš žurnalų lentelių. Knygų lentelėse informacija neredaguojama. Klaidos taisomos registruojant atvirkščius įrašus.



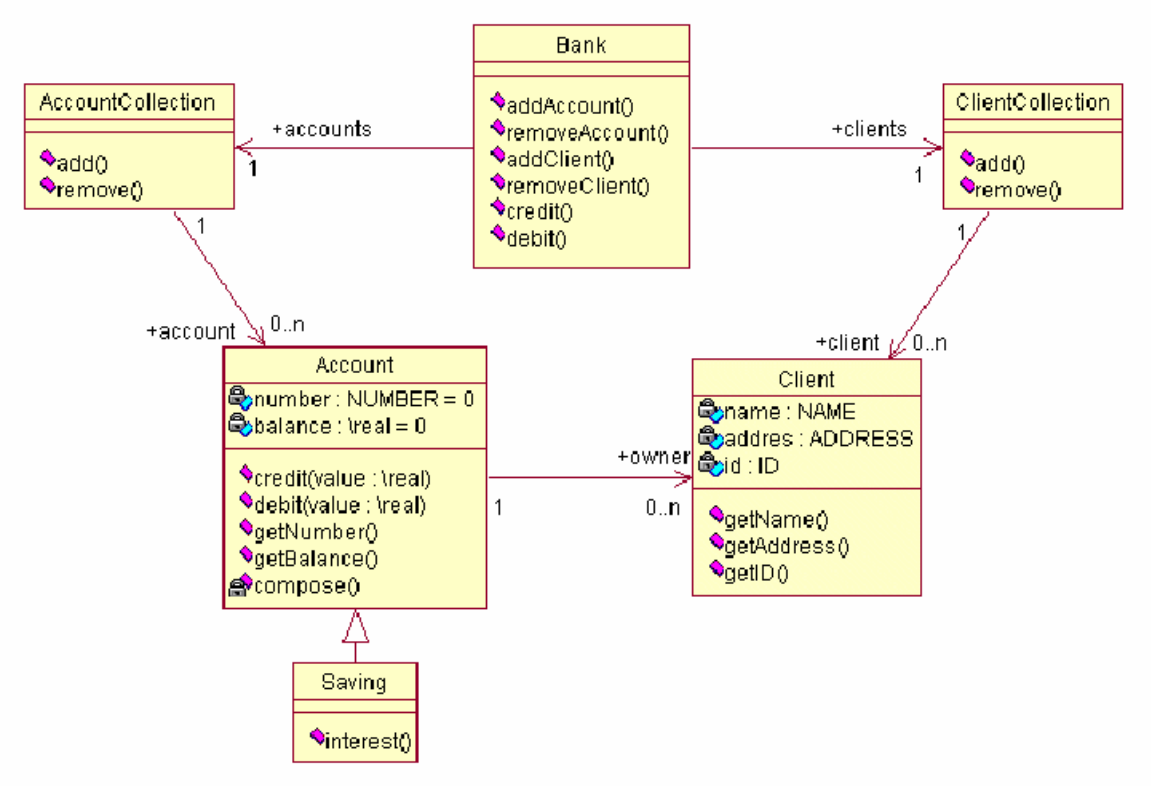
5 pav. Registravimo strukūra

Formuojant ataskaitas informacija yra imama iš lentelių atitinkamai ją apdorojus. Galima uždėti registravimo datos rėžį arba registravimo grupės filtrą. Bet informacija gali būti atvaizduojama ir naudojant formas.

Funkcionalumas yra suprogramuotas ketvirtos kartos iš dalies objektine programavimo kalba C/AL ir specialiomis objektų savybėmis (*properties*). Pvz., naudojant savybių nustatymą, yra nustatomi sąryšiai tarp lentelių, ar laukas vartotojo gali būti redaguojamas ir pan. Fundamentaliosios duomenų bazių funkcijos yra jau sukurtos. Pvz., jei pirminis raktas lentelėje yra ID , tai norint išgauti eilutę iš lentelės, kurios ID = ‘ABC123’, užtenka panaudoti funkciją „lentelė.GET(‘ABC123’)” ir objekte ‘lentelė’, kuris yra įrašo (lentelės eilutė) tipas, mes turėsime tos eilutės informaciją iš duomenų bazės. Duomenų sinchronizacija nereikia rūpintis, nes ji jau yra sistemiškai įdiegta.

Kaip jau minėta ankstesniuose skyriuose, dažniausiai Navison programinė įranga yra specifikuojama UML kalba. Tačiau konkretesni veiksmai, atliekami su duomenimis, nėra visiškai specifikuojami. Galima naudoti ir UML būsenų diagramą, tačiau jai verifikuoti kol kas nėra sąlygų. Taip pat galima naudoti ir OCL (*object constraint language*), tačiau ji nėra pilnai

apibrėžta ir nėra nusakyta jos sintaksė. OCL neturi kol kas jokio automatinio verifikavimo įrankio. Viena iš UML rūšių yra klasių diagrama. Ji patogi nusakyti lentelių reliacinius ryšius, atributus bei operacijas.



6 pav. UML klasių diagramos pavyzdys

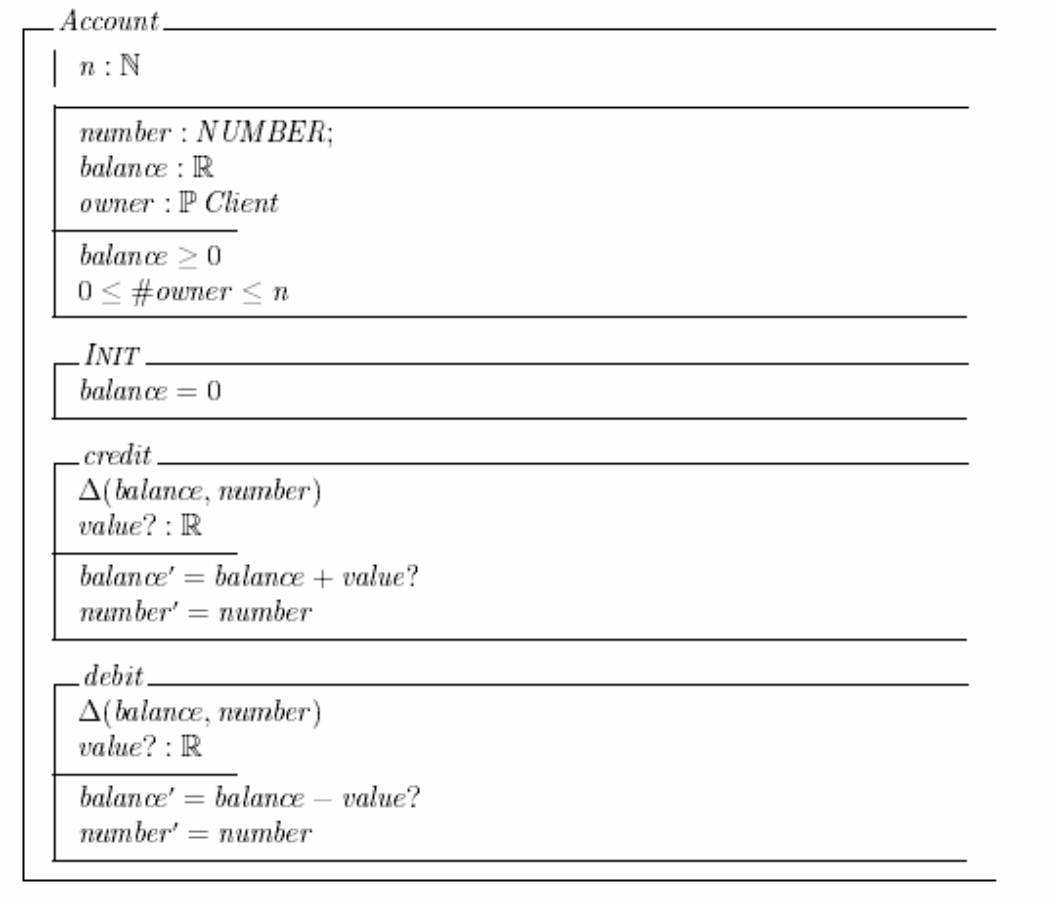
Paveiksle Nr. 6 yra klasių diagrama, nusakanti supaprastintą bankininkystės modelį. Šioje diagramoje yra beveik visi UML klasių diagramoj naudojami elementai: klasės, asociacijos ir paveldėjimas.

Klasė yra aibė objektų, kurie naudoja tuos pačius atributus, operacijas, sąryšius ir semantiką. Klasė turi vardą, atributus ir operacijas. Paveiksle Nr.3 yra klasė vardu „Account". Ši klasė turi du atributus „number", kurio tipas „NUMBER", ir „balance", kurio tipas yra R (realių skaičių aibė). Su šiais atributais naudojami keturiose operacijose: „credit", „debit", „getNumber", „getBalance".

Nusakyti ryšiams tarp objektų UML diagramose naudojamos asociacijos. Asociacija yra struktūrinis sąryšis nusakantis kaip klasės sąveikauja tarpusavyje. Paveiksle Nr. 3 yra matoma daug asociacijų pavyzdžių. „Bank" klasė turi sąryšius su „AccountCollection" ir „ClientCollection" klasėmis.

Paveldėjimas yra sąryšis tarp bendresnės klasės ir labiau nusakytos klasės. Pavyzdys: Klasė „Account" yra superklasė, o klasė „Saving" yra subklasė.

Pav. 7. yra parodyta kaip atrodo klasės "Account" specifikuota Z kalba.



7 pav. Z pavyzdys

Reikia pastebėti, kad asociacija „owner" yra jau kaip klasės atributas. Taip pat parodyta objekto inicializacijos operacija „Init".

Z galima specifikuoti tuos pačius specifikacijos elementus kaip ir UML klasių diagrama. Taip pat Z turi pranašumą prieš UML formalizmo prasme. Z yra pilnai formalizuota, kai UML - ne. Tačiau grafinės specifikacijos nėra būtina atsisakyti. Šiuo metu yra keli automatizuoti CASE įrankiai (RoZe), kurie padeda UML klasių diagramas automatiškai sutransliuoti į Object-Z

specifikacijas. Tačiau Object-Z kalbai kol kas nėra sukurta pakankamai galingo verifikavimo įrankio. Nėra būtina tiksliai specifikuoti kiekvieną operaciją, nes jos paprastai lengvai aprašomos ir suprantamos žodžiu, o atsisakius, UML mes negalėtume pavaizduoti tos operacijos be vidinių jos veiksmų. Galima daryti prielaidą, kad UML kalba ir Z gerai papildo viena kitą.

Todėl palyginimui darbe naudosime Z ir UML specifikavimo kalbas.

2.1. Specifikavimas UML

UML (*unified modeling language*) yra modeliavimo kalba, kuri dabartiniame išvystymo lygyje, apibrėžia žymėjimus ir meta-modeliavimą. UML nėra vien tik specifikacijų žymenų identifikavimas bei jų standartizavimas. UML taip pat apima daugelį kitų sričių, kurios pagrinde nėra objektiškai orientuotos technologijos kūrėjų tyrimų sritys:

- kaip modeliavimo kalboje identifikuoti ir panaudoti šablonus
- kaip panaudoti stereotipų koncepcijas išplečiant ir adaptuojant kalbą
- kaip užtikrinti tikslų perėjimą nuo specifikacijos iki implementuotos sistemos.

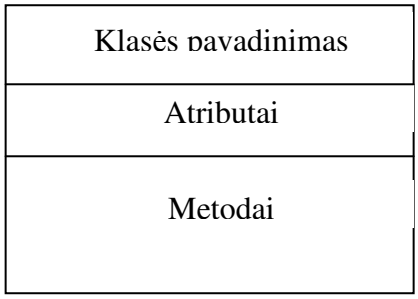
Viena iš UML pagrindinių savybių yra būti nepriklausomai nuo pasirinktos konkrečiu atveju programavimo kalbos ar terpės. Ja galima nesunkiai aprašyti įvairius kūrimo metodus. Kaip bebūtų, UML supratimas nereiškia vien tik žymenų ar jų prasmės mokymąsi. Tai yra modernus objektiškai orientuoto modeliavimo supratimas.

UML palaiko sekančias diagramas:

- panaudojimo atvejų diagramas (*use case diagrams*)
- klasių diagramas (*class diagrams*)
- elgsenos diagramas (*behavioral diagrams*)
- būsenų diagramas (*statechart diagrams*)
- veiklos diagramas (*activity diagrams*)
- tarpininkavimo diagramas (*interaction diagrams*)
- sekų diagramas (*sequence diagrams*)
- bendradarbiavimo diagramas(*collaboration diagrams*)
- sistemos kūrimo diagramas (*implementation diagrams*)
- sistemos komponentų diagramas (*component diagrams*)
- sistemos išsidėstymo diagramas (*deployment diagrams*)

2.1.1. Klasių diagramos

Klasių diagrama yra įrankis, leidžiantis parodyti sistemos klases ir sąryšius tarp jų. UML siūlo tokį klasės žymėjimą. Pav. nr. 8 viršutinėje skiltyje yra klasės pavadinimas. Viduriniojoje jos atributai. Žemiausioje skiltyje yra klasės metodai.



8 pav. UML klasė

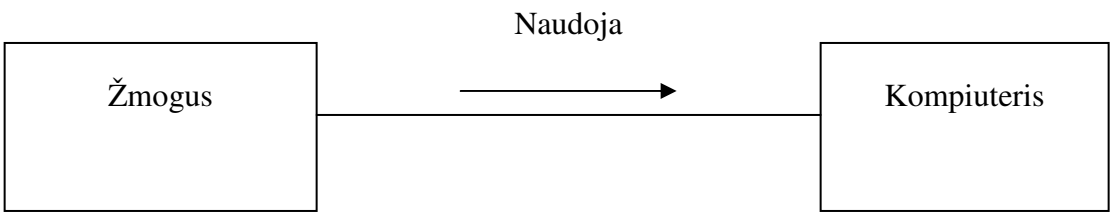
Klasių diagrama apibrėžia sistemos objektų tipus ir įvairius statinius ryšius tarp jų. Klasių diagrama parodo klasių statininę sistemos klasių struktūrą.

2.1.2. Sąryšiai

Klasių diagramos yra sudarytos iš klasių ir sąryšių tarp jų. Egzistuoja įvairių sąryšių rūšių. Tai asociacijos, paveldėjimo, priklausomybės, agregacijos, apibendrinimas ir kt.

2.1.3. Asociacijos

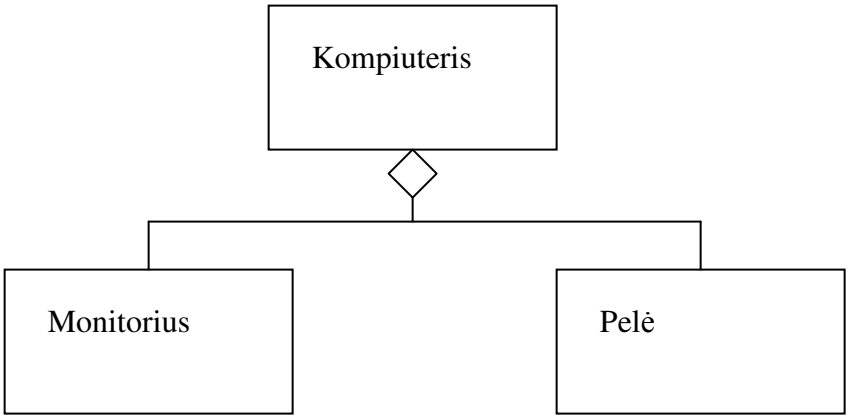
Asociacija yra semantinis ryšys tarp dviejų klasių, kuris sujungia dvi klases kai jos turi asociaciją. Asociacija dažnai yra dvipusis ryšys tarp dviejų klasių, kas reiškia, kad abi klasės turi įtaką viena kitai. Pav. Nr. 9 parodytas asociacijos pavyzdys:



9 pav. Asociacija

2.1.4. Agregacija

Agregacija yra situacinis ryšys, kai vienas sistemos objektas turi savyje kelis kitus sistemos objektus. T. y. kai objektas gali būti suskaidomas į dar kelis sistemos objektus, kurie savo ruožtu turi savo kitas asociacijas. Pav. Nr. 10 parodytas agregacijos pavyzdys:



10 pav. Agregacija

2.1.5. Paveldėjimas

Kartais klasės turi sutampančių atributų bei metodų. Pav. Nr. 11 parodytos dvi panašios klasės.

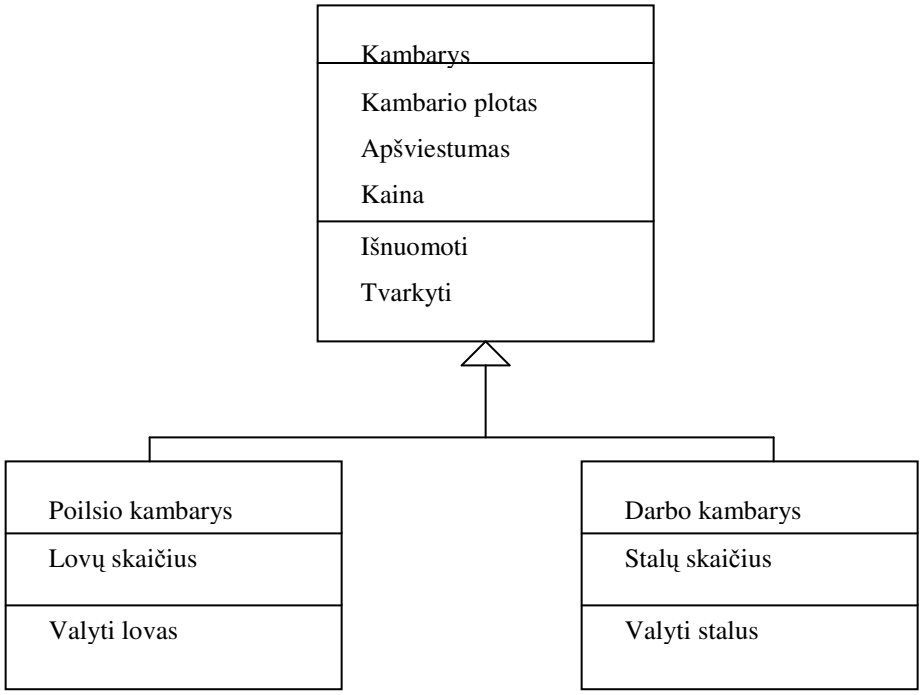
Poilsio kambarys
Kambario plotas Apšviestumas Kaina Lovų skaičius
Išnuomoti Valyti lovas Tvarkyti

Darbo kambarys
Kambario plotas Apšviestumas Kaina Stalų skaičius
Išnuomoti Valyti stalus Tvarkyti

11 pav. Dvi panašios klasės

Viešbutis nuomoja ir tvarko ir poilsio kambarius, ir darbo kambarius. Objektų duomenys ir metodai yra beveik vienodi. Tačiau yra ir skirtumų. Poilsio kambaryje yra valomos lovos, o

darbo – stalai. Siekiant išvengti pasikartojamumo naudojamas paveldėjimo ryšys tarp klasių. Yra sukuriama viena bendra klasė, kuri apima bendrus kelių klasių atributus bei metodus. Toliau yra taikoma hierarchinė struktūra. Bendroji klasė yra tėvinė, o anksčiau minėtos dvi klasės tampa vaikais. Pav. Nr. 12 yra parodyta klasių paveldėjimo hierarchija ankstesniam panašių klasių atvejui:

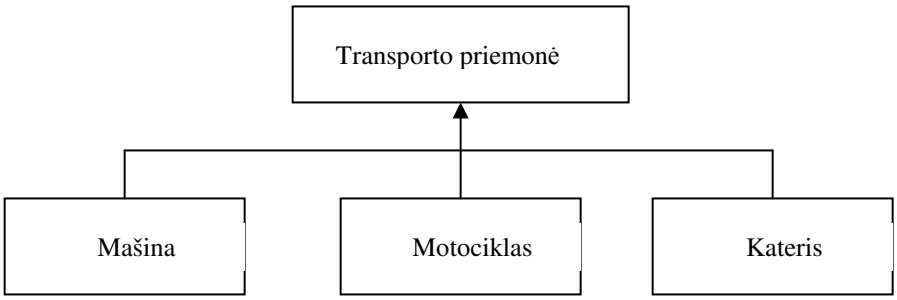


12 pav. Paveldėjimas

Duomenys ir metodai kurie yra bendri poilsio ir darbo kambariui yra iškeliami į atskirą bendrą klasę „Kambarys“. Metodai ir atributai kurie skiriasi, lieka atskirose klasėse. Klasė „Kambarys“ yra vadinama superklase, o „Poilsio kambarys“ ir „Darbo kambarys“ subklasėmis. Rodyklė Pav. Nr. 12 rodo paveldėjimą.

2.1.6. Apibendrinimas

Apibendrinimas yra sąryšis tarp apibendrinančiosios klasės ir atskirų specifinių klasių. Konkreti specifinė klasė, vadinama subklase, paveldi viską iš apibendrinančiosios (superklasės). Ir atributus, ir metodus. Pav. Nr. 13 parodyta superklasė„Transporto priemonė“ ir subklasės „Mašina“, „Motociklas“, „Kateris“.



13 pav. Apibendrinimas

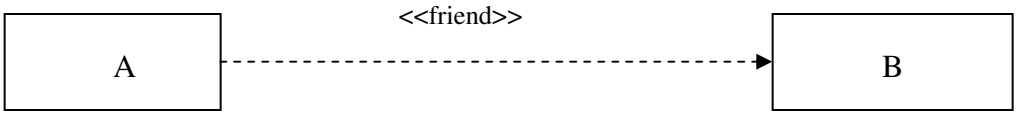
2.1.7. Priklausomybės

Priklausomybės sąryšis yra semantinis sąryšis tarp dviejų modelinių elementų. Vienas iš jų yra nepriklausomas, kai kitas – priklausomas. Pokytis nepriklausomam elemente iššaukia pokyčius priklausomam elemente. Modelio elementas gali būti klasė, paketas, ar panaudojimo atvejis ir kt.

Priklausomybės gali egzistuoti dėl daugelio priežasčių. Keletas iš jų:

- viena klasė siunčia pranešimą kitai
- klasė turi kitą klasę kaip duomenis
- klasė laiko kitą klasę taip parametrą savo metodui

Pav. Nr. 14 yra parodyta „friend“ tipo priklausomybė, kuri reiškia, kad elementas A turi pilną priėjimą prie elemento B vidinių duomenų.



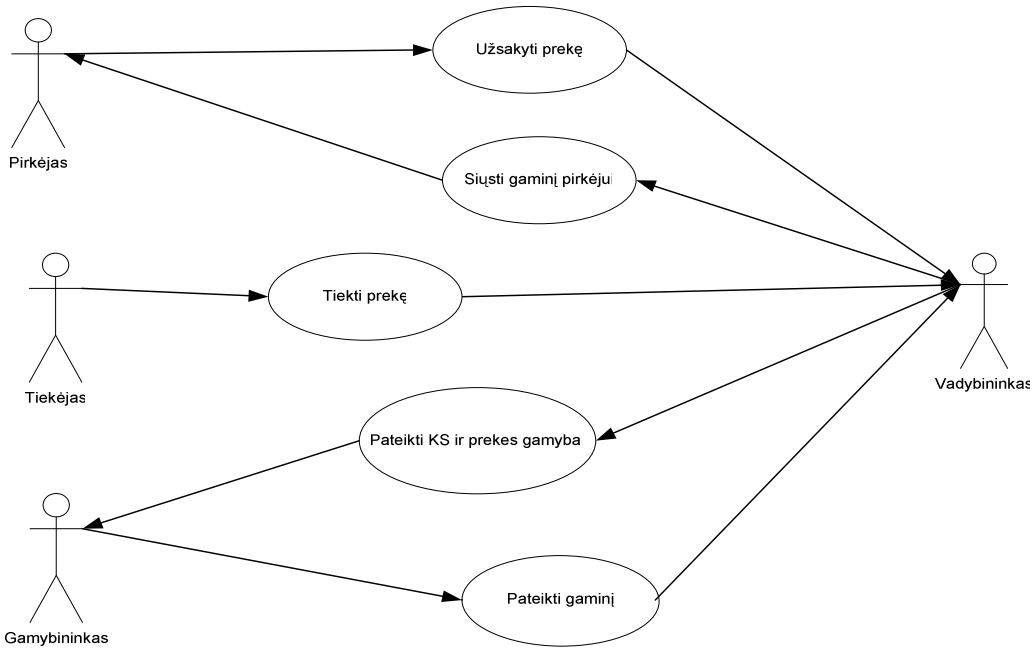
14 pav. Priklausomybė

Praktikoje, dideliuose projektuose, yra priimta naudoti kuo mažiau priklausomybių. Jeigu egzistuoja priklausomybė tarp skirtinguose paketuose esančių klasių, tai ir tarp paketų turi būti priklausomybė.

2.1.8. Panaudojimo atvejų diagramos

Perėjimas nuo reikalavimų aprašymo prie reikalavimų specifikacijos yra dažnai pasikartojantis procesas. Šiam procesui palengvinti yra naudojamos panaudojimo atvejų diagramos. Kuriant panaudojimo atvejų diagramas yra nustatoma kaip klientas dirbs su sistema vienu ar kitu atveju. Tai padeda identifikuoti objektus, operacijas bei objektų klases, kurie bus naudojami analizės procese.

Panaudojimo atvejo modelis UML yra aprašomas kaip Panaudojimo atvejo diagrama. Vienas panaudojimo atvejo modelis gali būti suskaidomas į kelias panaudojimo atvejų diagramas. Panaudojimo atvejo diagrama vaizduoja sistemos modelio elementus kaip aktorius (kurie veikia diagramoje) ir pasiūlytas operacijas, kurios buvo identifikuotos panaudojimo atvejų analizės metu. Panaudojimo atvejai yra atvaizduojami kaip elipsės apibrėžtoje sistemoje. Jie turi sąryšius su aktoriais kurie veikia sistemoje. Pav. Nr. 15 yra parodytas panaudojimo atvejo pavyzdys:



15 pav. Panaudojimo atvejų diagrama

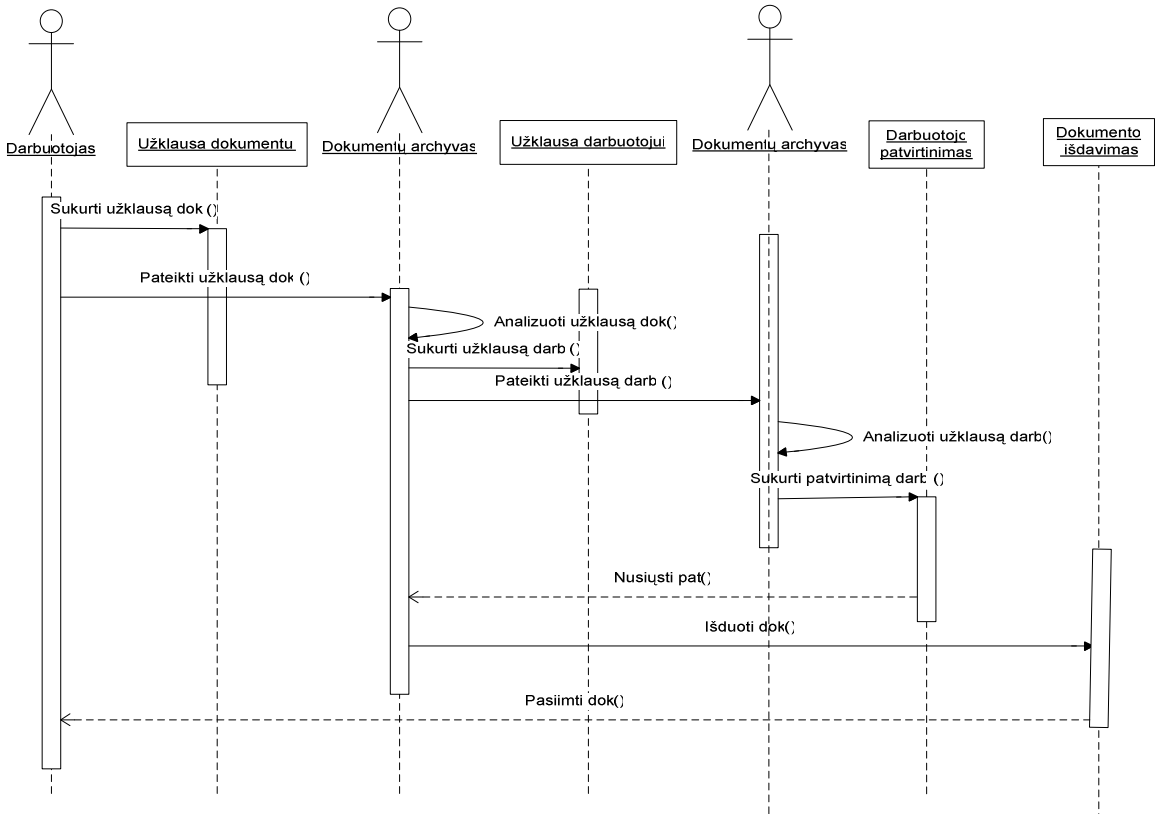
2.1.9. Sąveikos diagramos

Sąveikos diagramos yra modeliai, kurie nusako kaip bendradarbiauja grupė objektų elgsenos prasme. Sąveikos diagrama apima vieno panaudojimo atvejo elgsenas. Diagrama parodo kaip objektai bendrauja pranešimais konkrečiame panaudojimo atvejyje. Sąveikavimo diagrama

yra skirta objektų bendradarbiavimo analizei. Yra dvi rūšys sąveikos diagramų: sekų diagrama, bei bendradarbiavimo diagrama.

2.1.10. Sekų diagramos

Sekų diagramos taip pat yra vadinamos pranešimų eiliškumo diagramomis. Sekų diagramos parodo dinaminį bendradarbiavimą tarp grupės objektų. Sąveika tarp konkrečių objektų yra parodoma tam tikram sistemos vykdymo eigos vietoje. Objektai čia vaizduojami kaip vertikalios linijos su pavadinimais. Laiko ašis yra iš viršaus žemyn. Diagrama parodo kaip keičiasi objektai pranešimais laike. Pranešimai yra vaizduojami kaip rodyklės siejančios objektus laike. Pav. Nr. 16 yra pavaizduota sekų diagrama:

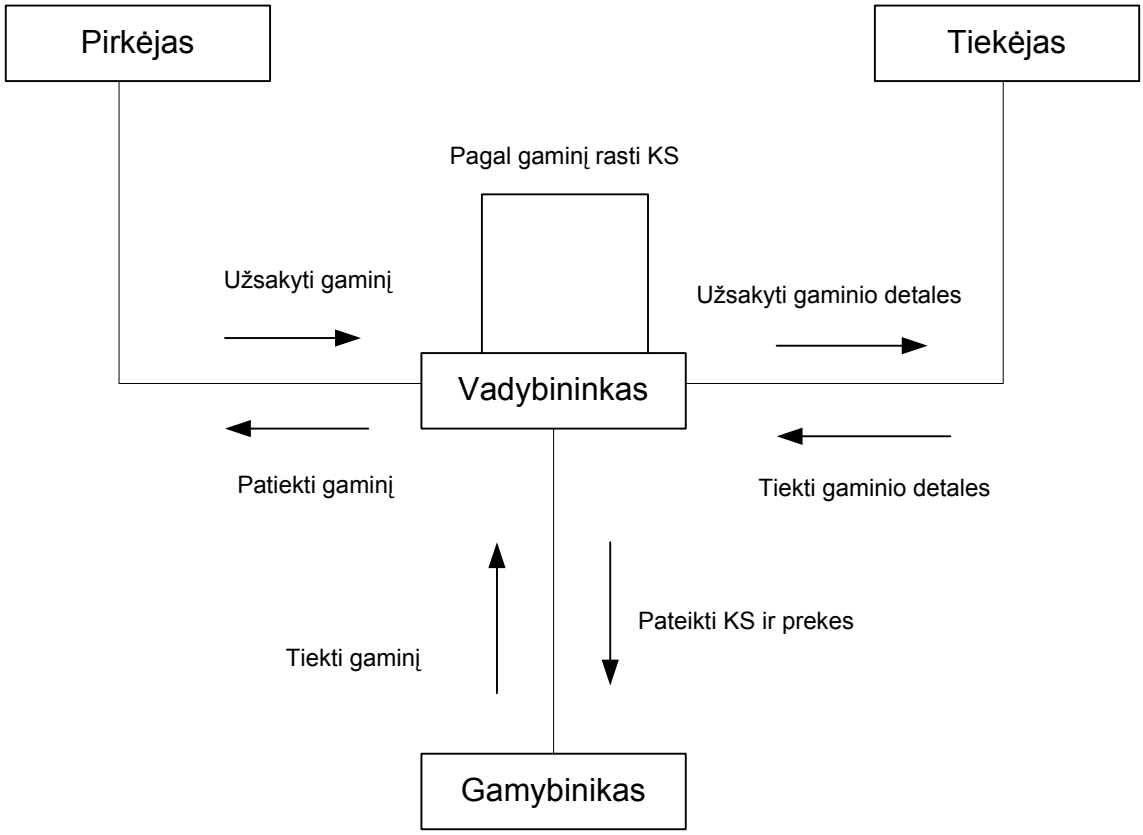


16 pav. Sekų diagrama

2.1.11. Bendradarbiavimo diagrama

Bendradarbiavimo diagrama parodo dinaminį bendradarbiavimą tarp objektų. Ji yra piešiama kaip objektų diagrama kartu su sąryšiais tarp jų. Pranešimų rodyklės parodo pranešimų

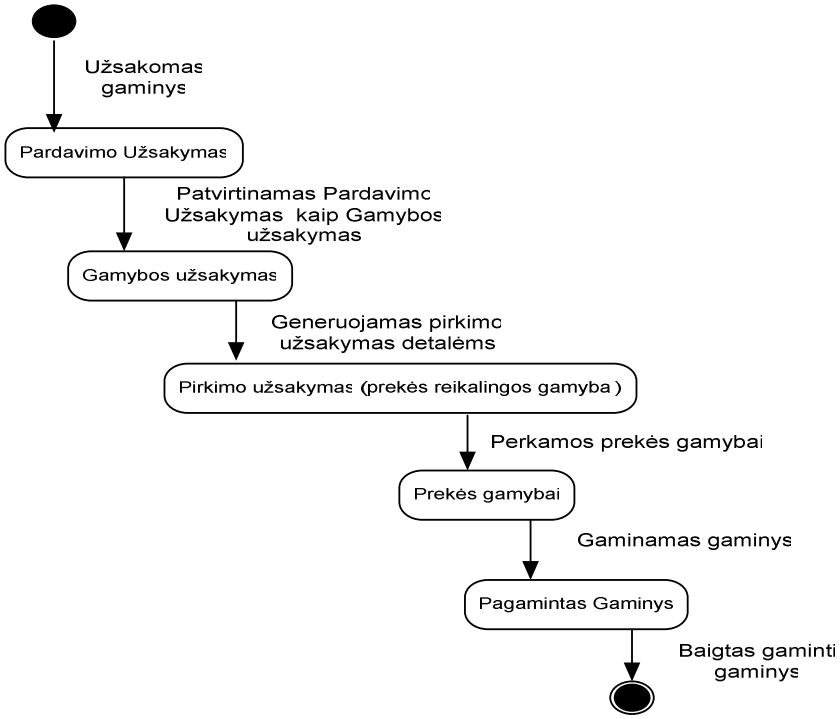
srautą tarp objektų. Rodyklių antraštės identifikuoja pranešimus. Pranešimai taip pat gali būti numeruojami eilės tvarka. Pav. Nr. 17 yra parodytas bendradarbiavimo diagramos pavyzdys:



17 pav. Bendradarbiavimo diagrama

2.1.12. Būsenų diagrama

Būsenų diagrama apibrėžia konkretaus objekto būsenas ir kaip jos keičiasi, kai juos paliečia tam tikri įvykiai. Paprastai būsenų diagrama vaizduoja klasės aprašą. Būsenų diagramos yra paišomos klasėms, kurioms būna tiksliai nustatytos būsenos į kurias ji gali patekti. Pav. Nr. 18 yra būsenų diagramos pavyzdys:



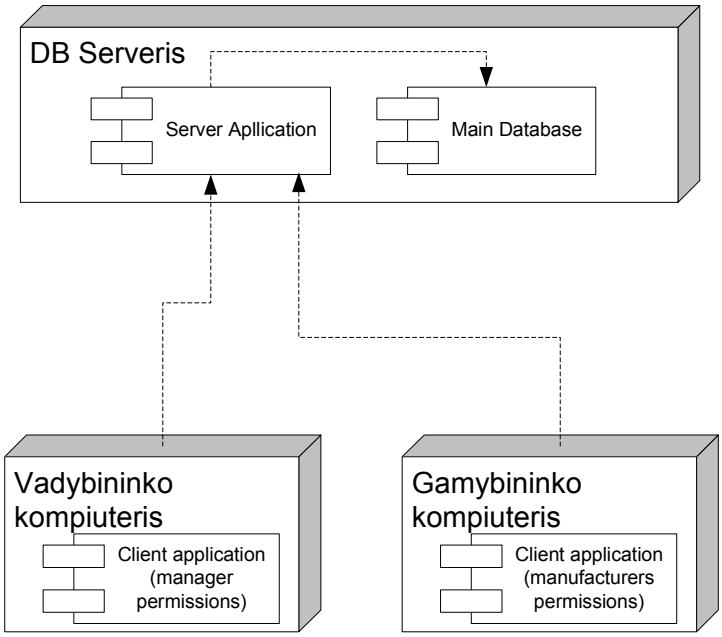
18 pav. Būsenų diagrama

2.1.13. Komponentų diagrama

Komponentų diagrama parodo programinio kodo komponentų architektūrą. Komponentas gali būti išeities teksto komponentas arba sutransliuoto kodo komponentas. Komponentas turi savyje informaciją apie klases iš kurių jis yra sudarytas, o tai leidžia iš loginio lygio pereiti prie komponentinio sistemos vaizdavimo lygio. Yra parodomos priklausomybės tarp komponentų, kurios apsprendžia kaip komponentai sąveikauja tarpusavy.

2.1.14. Sistemos išsidėstymo diagrama

Sistemos išsidėstymo diagrama parodo ryšius tarp programinės įrangos komponentų ir aparatūros komponentų. Programos komponentai yra vaizduojami aparatūros komponentų viduje, siekiant parodyti, kuriuose taškuose veiks tam tikros programos. Išoriniai sąryšiai vaizduoja kaip sąveikauja programos aparatūriniame lygyje. Pav. Nr. 19 yra parodytas sistemos išsidėstymo diagramos pavyzdys:



19 pav. Sistemos išsidėstymo diagrama

2.2. FORMALIOS SPECIFIKACIJOS

Rašant formalias specifikacijas yra naudojamas matematinis žymėjimas, siekiant aprašyti tikslias sistemos savybes. Jomis nesiekiama apibrėžti būdo kaip pasiekti tas sistemos savybes. Formalios specifikacijos aiškina ką sistema turi daryti, o ne kaip sistema turi būti padaryta. Ši abstrakcija daro formalias specifikacijas naudingas sistemos kūrimo procese, nes ji leidžia atsakyti į abstrakčius klausimus, nesigilinant į būsimo išeities teksto detales. Taip pat užkerta kelią įvairioms interpretacijoms, kurios neišvengiamai kyla sistemą specifikavus paprasta žodine kalba. Neformali sistemos specifikacija gali turėti savyje prieštaravimų, dviprasmiškumų, neaiškumų, nepilnų tvirtinimų bei sumaišytų abstrakcijos lygių (*mixed levels of abstraction*).

Formali specifikacija gali tarnauti kaip vieningas atspirties taškas žmonėms, kurie nagrinėja užsakovo poreikius. Taip pat tiems, kurie kuria sistemą, ją testuoja bei rašo vartotojo instrukcijas. Kadangi specifikacija yra nepriklausoma nuo programavimo kalbos, ji gali būti užbaigta ankstyvose projekto stadijose. Ji gali būti keičiama ir vėlesniuose etapuose, jei pvz., kliento norai keičiasi arba projekto komanda, įsigilinusi į specifikaciją, nusprendžia atsisakyti ar pridėti naujų savybių.

Formalios specifikacijos kalba paprastai susideda iš trijų pagrindinių komponenčių :

- sintaksės , kuri apibrėžia specifinį žymėjimą kurio pagalba specifikacija yra atvaizduojama

- semantikos, kuri padeda apibrėžti objektų visumą, kuri bus naudojama aprašant sistemą

- aibės ryšių (*set of relation*) , kuri apibrėžia taisyklės, parodančias kokie objektai tenkina specifikaciją.

Formalios specifikacijos sintaksinė sritis (syntactic domain) dažnai pagrįsta sintakse paveldėta iš standartinės aibių teorijos ir remiasi skaičiavimais (*calculus*).

Formalios specifikacijos semantinė sritis (semantic domain) parodo kaip kalba atvaizduoja sistemos reikalavimus. Pavyzdžiui programavimo kalba turi formalią semantiką, kuri leidžia programuotojui nurodyti algoritmus transformavimui įvedimo srauto į išvedimo srautą. Formali gramatika gali būti panaudojama aprašant programavimo kalbos sintaksę, bet programavimo kalba nėra gera specifikavimo kalba, nes ji gali atvaizduoti tik apskaičiuojamas funkcijas. Specifikavimo kalbos semantinė sfera turi būti platesnė, ji turi būti pajėgi išreikšti panašias idėjas: "Kiekvienam *x* begalinėj aibėj *A*, egzistuoja toks *y* begalinėj aibėj *B*, kad ir *x*, ir *y* tenkinama savybė *P*." Kitos specifikavimo kalbos paprastai naudoja semantiką, kurios pagalba specifikuojamos sistemos elgsenos. Pavyzdžiui sintaksė ir semantika gali būti išvystyta tam, kad specifikuoti būsenas ir būsenų perėjimus, įvykius ir jų efektus būsenų perėjimui.

2.2.1. Z kalba

Z kalba yra turbūt plačiausiai sėkmingai naudojama formali kalba. Ypač dideliuose sudėtinguose projektuose. Specifikuotojas turi būti ypač matematiškai tikslus, kad rezultate gautųsi kuo mažiau dviprasmybių, prieštarų bei neaiškumų. Z kalba parašyta specifikacija leidžia sistemos kūrėjui bet kuriuo metu įvertinti specifikacijos teisingumą. Z specifikacijų panaudojimas dažnai sumažindavo sistemų kūrimo kaštus.

Z kalba turi griežtai apibrėžtą semantiką, todėl kiekviena atskira kalbos konstrukcija turi tiksliai apibrėžtą interpretaciją. Z naudoja aibę integruotų matematinių duomenų tipų, siekiant apibrėžti sistemoje naudojamus duomenis. Šie duomenų tipai nėra orientuoti į specifinę programavimo kalbą. Jie reikalingi matematinėms taisyklėms nusakyti, kurios savo ruožtu, apibrėžia sistemos savybes ir jos veikimą. Vienareikšmiška interpretacija padeda sistemos kūrėjams bendrauti. Leidžia išvengti nenumatytų atvejų.

Z specifikacija susideda iš schemų. Kiekviena schema susideda iš kintamųjų deklaravimo (aprašymo) ir predikatų, kurie apibrėžia tų kintamųjų galimas reikšmes. Z yra labai lanksti specifikacija, nes schemos leidžia lanksčiai aprašyti įvairius objektus. Taip pat statinius bei dinaminius sistemos aspektus.

Statiniai sistemos aspektai:

- Būsenos, kurias sistema gali turėti
- Būsenų invariantiniai sąryšiai, kurie turi būti apibrėžti, kai sistema iš vienos būsenos pereina į kitą.

Dinaminiai sistemos aspektai:

- Sistemoje vykdomos operacijos
- Sąryšiai tarp operacijų duomenų ir rezultatų.
- Būsenos pokyčiai, kurie gali įvykti arba jiems gali būti leista įvykti.

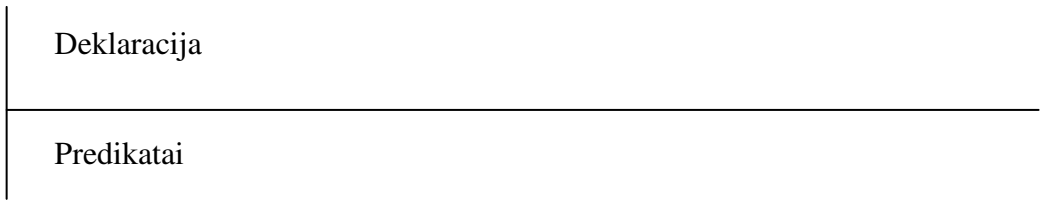
Būsenos invariantas yra formuluotė, kuri apibrėžia sistemos būsenos teisingumą, prieš ir po kiekvienos operacijos įvykdymą. Tai padaroma detaliai ir tiksliai apibrėžiant duomenis schemos deklaravimo srityje bei predikatų srityje. Abstraktus duomenų tipas yra aibė reikšmių, su kuriomis duomenų tipas gali turėti operacijas.

Z specifikacijos turi keletą rūšių schemų. Tai būsenų, operatorių schemos, bei aksiomatiniai aprašai.

Daugiau informacijos apie Z galima rasti J.M. Spivey knygoje (1).

2.2.1.1. Aksiomatiniai aprašai

Globalūs kintamieji Z specifikacijoje yra aprašomi paragrafuose, pavadintuose aksiomatiniais aprašais. Šiuose aprašuose deklaruoti globalus kintamieji galioja visai tolesnei specifikacijai.



Kintamųjų vardai virš horizontalios linijos turi būti nė karto nepanaudoti iki šios deklaracijos. Aksiomatiniai aprašai taip pat gali būti panaudojami nusakyti konstantoms. Sakykime, kad maksimalus automobilių skaičius *max_cars* negali būti didesnis už 50. Tada Z specifikacijoj ši konstanta būtų aprašoma sekančiai:

$$max\_cars : \mathbb{N}$$

$$max\_cars = 50$$

Aksiomatiniai aprašai taip pat gali būti naudojami nusakyti funkcijoms išreikštoms Z kalba. Pvz., jei *iroot(a)* yra kvadratinė šaknis iš *a*, tai kvadratinės šaknies traukimo funkcija gali būti atvaizduojama Z specifikacijoje sekančiai:

$$iroot : \mathbb{N} \rightarrow \mathbb{N}$$

$$\forall\, a : \mathbb{N} \bullet iroot(a) * iroot(a) \leq (iroot(a) + 1) * (iroot(a) + 1)$$

2.2.1.2. Būsenų schemos

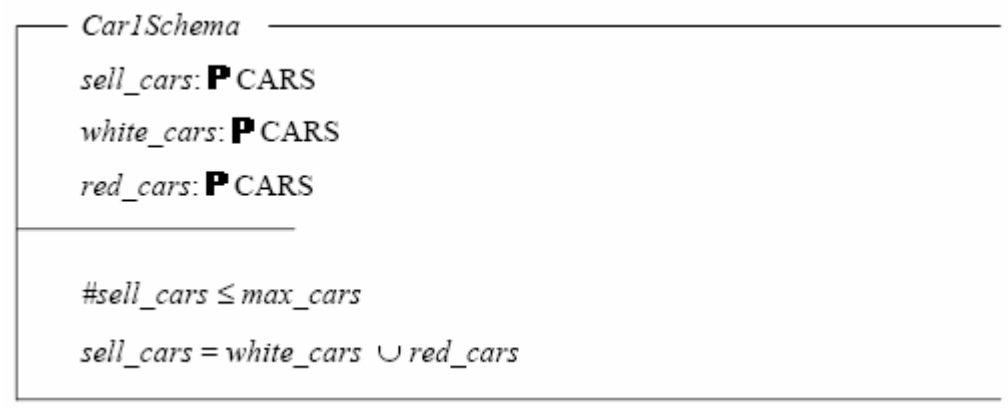
Būsenų schemos apibrėžia kintamuosius ir ryšius tarp jų. Kiekvieną kartą kai kintamasis pakeičia reikšmę yra laikoma, kad kintamasis pakeitė būseną. Sąryšiai tarp kintamųjų yra aprašomi predikatais. Šie predikatai, kitaip sistemos invariantai, apibrėžia kiekvieną teisingą sistemos būseną. Schema turi sekantį žymėjimą:

Schemos pavadinimas

Kintamųjų deklaracijos

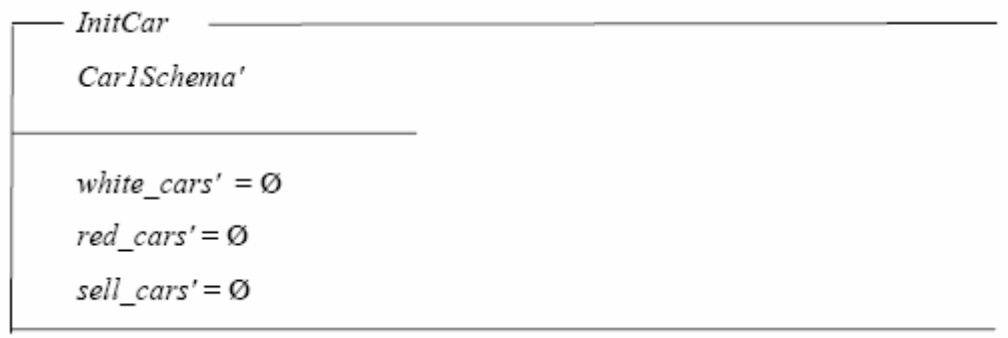
Predikatai

Pavyzdys:



Schemos pavadinimas – *Car1Schema.* Kintamieji *sell_cars*, *white_cars*, *red_cars* turi vieną bendrą tipą ℙ**CARS**. Šis tipas reiškia visų galimų automobilių aibę. Po viduriniosios linijos eina predikatai : *#sell_cars⩽max_cars* ir *sell_cars* = *white_cars* ∪ *red_cars*. Pirmasis predikatas teigia, kad *sell_cars* (parduotų automobilių kiekis) negali būti didesnis už *max_cars* (maksimalus automobilių skaičius). Simbolis „#" reiškia *sell_cars* aibės kardinalumą. Antrasis teigia, kad aibė *sell_cars* yra aibių *white_cars* ir *red_cars* sąjunga.

Pradinė sistemos būsena yra apibrėžiama tokia schema:



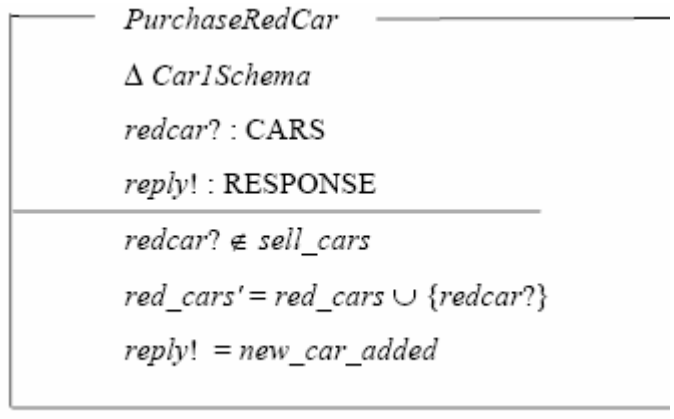
Apostrofai prie kintamųjų reiškia, kad sistemos būsena yra keičiama.

2.2.1.3. Operacijų schemos

Operacijų schemų modelis nurodo, kaip keičiasi sistemos būsena po konkrečios operacijos. Išankstinės sąlygos apibrėžia apribojimus operacijoms. Toliau yra nusakomos sąlygos sistemos būsenai po operacijos. Operacijos schemos būtini atributai:

- Visi kintamieji, kuriems operacija turi įtakos turi būti deklaruoti
- Kintamieji, su kuriais operacija daro veiksmus yra žymimi kintamojo vardu ir gale simboliu „?“. Pvz., *sell_cars?*
- Rezultato kintamieji yra žymimi kintamojo vardu ir gal simboliu „!“. Pvz., *sell_cars!*
- Būsenos schemos kintamieji yra įtraukiami į deklaracijos skyrių ir pažymimi simboliais „Δ“ arba „Ξ“. Simbolis „Δ“ parodo, kad būsenos schemos kintamieji gali keisti savo reikšmes po operacijos. Simbolis „Ξ“ parodo, kad būsenos schemos kintamieji nekeis savo reikšmių operacijos rezultate.
- Pranešimai gali būti aprašomi naudojant laisvuosius duomenų tipus. Pvz., *RESPONSE*::=*car_removed* | *new_car_added*.

Pabandykime formaliai specifikuoti operaciją, kai perkama raudona mašina (*red_card*) salonui (*showroom*). Po operacijos ši mašina bus prijungta prie aibės kitų raudonų mašinų. Taigi kokie turi būti veiksmai norit atlikti šią operaciją? Prieš perkant mašiną turi būti patikrinta ar ji nėra jau nupirkta, t. y. ar ji nėra jau mašinų kolekcijoje. Jei mašina nėra nupirkta, tai ją galima pirkti ir prijungti prie kitų raudonų mašinų aibės. Kintamaso *red_cars* reikšmė bus atnaujinta. Nauja raudona mašina bus prijungta prie kitų raudonų mašinų aibės. Kintamasis schema Δ*Car1Schema* parodo žemiau nurodytoje operacijos schemoje *PurchaseRedCar* , kad jos kintamųjų būsena gali keistis šioje operacijų schemoje:



- Predikatas *redcar?∉sell_cars* teigia išankstinę sąlygą, kad perkamoji mašina negali būti parduodamų mašinų aibėje.

- Predikatas *red_cars'=red_cars∪{redcar?}* nurodo rezultato sąlygą, kad nauja raudona mašina yra prijungiama prie jau esančių raudonų mašinų aibės.
- Paskutinis predikatas suteikia reikšmę pranešimui, kad nauja mašina nupirkta sėkmingai

2.2.1.4. Verifikavimas

Prieš pereinant į projektavimo bei programavimo etapus Z specifikacijos gali būti validuojamos ir verifikuojamos. Kiekvienam sistemos kūrimo etape galima patikrinti rezultatus, ar jie atitinka ankstesnėj fazėj numatytas užduotis. Tai yra verifikavimas.

Verifikavimas apima tokias sritis kaip patikrinimą ar specifikacija pilnai nusako trokštamas sistemos savybes, ar priešingai, viskas ko nebūtina sistemai yra atmesta. Vienas iš stipriųjų formalių metodų pusių yra tai, kad jos remiasi matematiniu pagrindu, kas suteikia galimybę verifikuoti ir patikrinti specifikuotą sistemą. Sistemos specifikacijos vientisumo bei suderinamumo faktoriai yra sekantys:

- Globalių apibrėžčių suderinamumas
- Sistemos būsenų suderinamumas
- Operacijų suderinamumas

Z specifikacija yra suderinta, jei joje nėra logikos klaidų ir gali būti realizuojama kaip matematinis objektas.

2.2.1.4.1. Verifikavimo metodai

Egzistuoja keletas specifikacijų verifikavimo būdų:

- Kūrimo grupės peržiūra. Specifikacija parašyta vieno žmogaus ar jų grupės yra peržiūrima kitų asmenų, kurie turi atitinkamą kvalifikaciją ir žinias. Grupės peržiūra gali nustatyti klaidingai vartojamus specifikavimo kalbos niuansus, taip pat gali pastebėti neatitikimų su numatyta sistema. Grupės peržiūra užtikrina, kad specifikacija bus aiški, prasminga ir patogi naudojimui projekto kūrimo nariams.
- Tipų ir sintaksės tikrinimas. Z specifikavimo kalba turi griežtai nusakytą sintaksę, taigi specifikacijos kūrėjas gali naudoti automatizuotus įrankius, kurie tiksliai sutikrina tipus ir sintaksę, bei automatiškai randa klaidas. Pvz. CaDiZ.

- Specifikacijos korektiškumo pagrindimas. Šis pagrindimas leidžia surasti konfliktines specifikacijos vietas bei nustatyti specifikuotų veiksmų skirtumus.

2.2.1.4.2. Globalių apibrėžimų verifikavimas

Sakykime, kad turime aksiomatinį aprašymą:

GlobaliosDeklaracijos

GlobalūsPredikatai

Ši schema gali būti perrašoma horizontaliai: *GlobaliosDeklaracijos* | *GlobalūsPredikatai.* Taigi turi būti paskelbta, kad ⊢ ∃ *GlobaliosDeklaracijos* • *GlobalūsPredikatai*, kas reiškia, jog egzistuoja tokios *GlobaliosDeklaracijos* reikšmės, kad tenkina predikatą *GlobalūsPredikatai.* Pavyzdžiui, imkime sekantį (klaidingą) aksiomatinį aprašą:

$$sign : \mathbb{Z} \rightarrow \mathbb{Z}$$

$$\forall n: \mathbb{Z} \bullet$$
$$(n \leq 0 \Rightarrow sign\ n = -1) \wedge$$
$$(n \geq 0 \Rightarrow sign\ n = 1)$$

Šis aprašas yra klaidingas todėl, kad tenkina dvi sąlygas: kai *sign* 0 = -1 ir kai *sign* 0 = 1. O *sign* juk yra (pilna) funkcija. Ji gali turėti tik vieną reikšmę. Norint verifikuoti šį atvejį, sekanti teorema gali būti paskelbta:

$$\vdash \exists\ sign : \mathbb{Z} \rightarrow \mathbb{Z} \bullet \forall n: \mathbb{Z} \bullet n <= 0 \Rightarrow sign\ n = -1 \wedge n >= 0 \Rightarrow sign\ n = 1$$

Eliminuojama „<“ ir „>“:

$$\vdash \exists\ sign : \mathbb{Z} \rightarrow \mathbb{Z} \bullet \forall n: \mathbb{Z} \bullet n = 0 \Rightarrow sign\ n = -1 \wedge n = 0 \Rightarrow sign\ n = 1$$

Pagal $\mathbb{Z}$ savybes:

$$\vdash \exists\, sign : \mathbb{Z} \rightarrow \mathbb{Z} \bullet \forall n: \mathbb{Z} \bullet n = 0 \Rightarrow sign\ n = -1 \wedge \neg (sign\ n = -1)$$

Iš aukščiau išplaukia, kad:

$$\vdash \exists\, sign : \mathbb{Z} \rightarrow \mathbb{Z} \bullet false$$

Reiškia:

$$\vdash false$$

Taigi šis predikatas yra prieštara, kadangi *sign*$:\mathbb{Z} \rightarrow \mathbb{Z}$ yra su juo nesuderinama.

2.2.1.4.3. Būsenos modelių verifikavimas

Gali būti vykdomas ir būsenos modelio patikrinimas, ar jis logiškai teisingas. Patikrinimas gali būti išreikštas kaip teorema, kuri turi sekančią formą:

$\vdash \exists$ *Būsena‘* • *PradinėBūsenosSchema*

Kuri gali būti išplėsta į:

$\vdash \exists$ *Būsena‘;įvestis?* • *PradinėBūsenosSchema*

jeigu yra įvesties duomenų inicializavimo schemoje *PradinėBūsenosSchema*.

Ši teorema skelbia, kad egzistuoja bendro modelio būsena (ir įvestys schemoje *PradinėBūsenosSchema*), kuri tenkina inicializavimo būsenos aprašą. Įrodant šią teoremą yra tuo pačiu įrodomas suderinamumas tarp *Būsena* ir *PradinėBūsenosSchema*. T. y. *Būsena* predikatai nėra išvedami iki *false* reikšmės.

2.2.1.4.4. Operacijų verifikavimas

Operacijai kuri aprašoma *OperacijosDeklaracijos* | *OperacijosPredikatai*, suderinamumo teorema atrodo sekančiai:

$\vdash \exists$ *OperacijosDeklaracijos* • *OperacijosPredikatai*

Prieš-sąlygų patikrinimas leidžia nustatyti operacijos loginį teisingumą. Jeigu operacija yra logiškai neteisinga, tai prieš-sąlygos turės reikšmę *false*. Klaidingos prieš-sąlygos dažniausiai reiškia operacijos aprašo klaidą. Kaip pavyzdys, žemiau yra įrodyta operacija *Exchange*.

$$Schema1 \triangleq [x, y : \mathbb{Z} \mid x > y]$$

$$Exchange \triangleq [\Delta Schema1 \mid x' = y \wedge y' = x]$$

Suderinamumo teorema šiai operacijai bus:

$$\vdash \exists \Delta Schema1 \bullet Exchange$$

Sukeliame schemų deklaracijas ir predikatus:

$$\vdash \exists x, y, x', y' : \mathbb{Z} \bullet x > y \wedge x' > y' \wedge x' = y \wedge y' = x$$

Vieno taško taisyklė taikoma x‘ ir y‘:

$$\vdash \exists x, y : \mathbb{Z} \bullet x > y \wedge y > x$$

Pagal $\mathbb{Z}$ savybes:

$$\vdash \exists x, y : \mathbb{Z} \bullet x > y \wedge \neg(y > x)$$

Iš ankstesnės išraiškos išplaukia, kad:

$$\vdash \exists x, y : \mathbb{Z} \bullet false$$

Rezultatas:

$$\vdash false$$

Tai reiškia, kad šis predikatas nesiderina su būsenos schema *Schema1*.

2.2.1.5. VALIDAVIMAS

Specifikacija gali būti teisinga ir neturėti savyje prieštaraujančių teiginių, bet tai dar nereiškia, kad tiksliai apibūdina užsakovo sistemos viziją. Kitais žodžiais tariant, specifikacija gali būti pakankama, kad sukurti teisingą nekonfliktuojančią tarpusavio srityse sistemą, tačiau ji gali neatitikti visų pradžioje užsakytų reikalavimų.

Neegzistuoja įrodymų, kurie patvirtintų teisingą natūralios kalbos formalizavimą. Neįmanomas yra formalus perėjimas nuo neformalios kalbos prie formalios kalbos. Formalūs metodai gali įrodyti, kad sukurta sistema atitinka specifikaciją, bet jais negalima pasikliauti kalbant apie tai, kad jie pilnai aprėpia kliento įsivaizdavimą apie sistemą.

Priešingai nei sistemos suderinamumas, validumas negali būti griežtai įrodomas, kadangi šiam procesui reikalingas užsakovas, sistemos srities analizė bei reikalavimai sistemai.

Tinkamumo vertinimai gali būti vidiniai ir išoriniai:

- Vidiniai vertinimai gali būti vykdomi pačių specifikuotojų, kad padidinti jų supratimą ar specifikacija yra tinkama. Pvz., Gali būti patikrinamos būsenos operatorių savybės. Tikrinimo rezultate turi būti prieita išvados, kad specifikacija atitinka reikalavimus.
- Išoriniai vertinimai turėtų būti inicijuojami kliento. Turi būti parodoma, kad būsenos modelis yra tam tikros realybės abstrakcija, o operacijos nusako funkcionalumą, kurio klientui ir reikia.

Specifikacijų validavimo būdai apima peržiūras, prototipų konstravimą ir netgi specifikacijos virtualų vykdymą su tam tikrais automatizuotais įrankiais. Teisingas specifikacijos validavimas reiškia, kad:

- bus gauta nauda iš visų teigiamų formalaus specifikavimo savybių
- formali specifikacija bus visomis prasmėmis teisinga

Jeigu sistemos reikalavimai yra aiškūs ir formali specifikacija yra struktūriškai gerai sudaryta, tada yra įmanomas validavimas vien peržiūrint sistemą.

2.3. Apibendrintas palyginimas: Z ir UML

Kad specifikuoti duomenis ir operacijas, Z kalba naudoja matematines koncepcijas, pagrinde aibių teoriją. Tai leidžia pagrįsti sistemą suderinamumo prasme ir operacijų bei būsenų loginio teisingumo prasme. Taip pat leidžia verifikuoti pačią specifikaciją. Sistemos statinių

būsenų bei operacijų specifikacija yra griežta ir neturi dviprasmybių. Pagrindinis būdas rašyti Z specifikacijas yra statinių būsenų identifikavimas ir operacijų jų pokyčiams aprašymas.

UML yra pagrįsta grafiniu žymėjimu, apimantį platų srities problemų spektrą. Grafinis žymėjimas bei modeliavimas yra gera priemonė kovojant prieš sistemos aprašymo sudėtingumą. Šiuo atveju UML ypač gerai tinka. Diagraminių struktūrų analizė jau įrodė esanti gera komunikavimo priemonė nepatyrusiems sistemų kūrime žmonėms, netgi po to kai sistema būna formaliai specifikuota. Nors UML žymenys yra intuityvūs ir lengvai suprantami vartotojams, jiems vis tiek trūksta pilnumo bei tikslumo.

Nei Z, nei UML nėra specifikavimo metodai. Jie yra specifikavimo kalbos.

Z apjungimas su grafiniu žymėjimu galėtų būti vienas iš sprendimų, siekiant visiems suprantamos ir aiškios specifikacijos. UML kalboje nėra lengva patikrinti duomenų suderinamumą kaip Z.

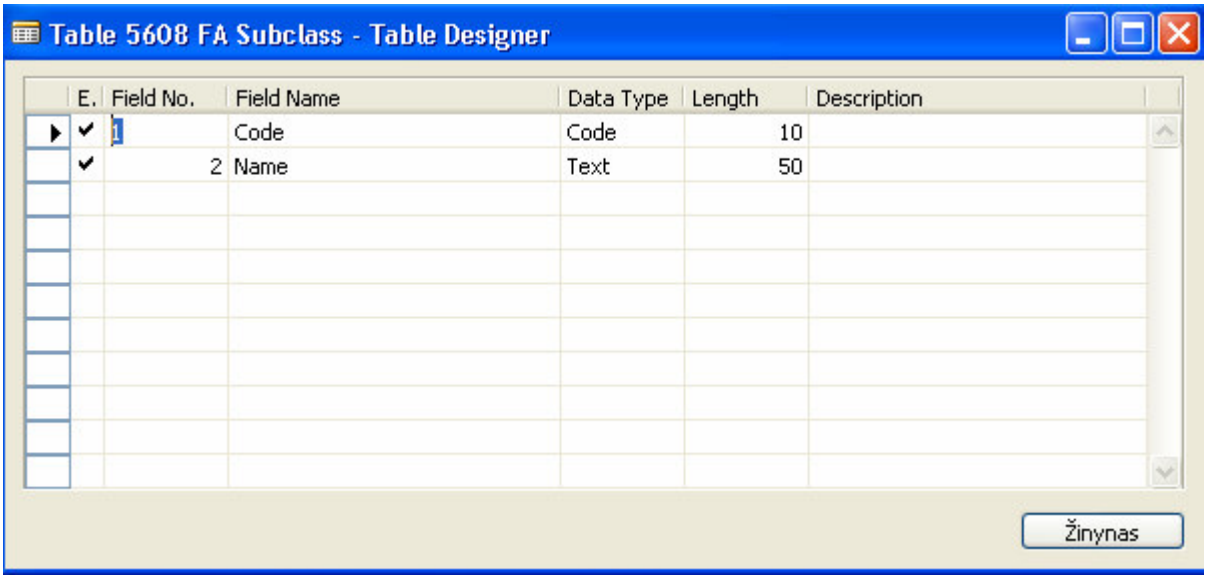
2.4. C/SIDE programavimo terpė

Siekiant įsitikinti sistemos formalizavimo prasmingumu, reikia išnagrinėti MSBS Navision programavimo terpę C/SIDE (*client server intergrated development environment*). Išnagrinėjus terpės savybes, bus galima patikrinti ar formalios specifikacijos lemia tik sistemos supratimą, ar ir pagreitina programavimą.

C/SIDE – tai kliento serverio integruota programinės įrangos vystymo terpė. Ji kažkiek panaši į Visual Basic. Daugelį sistemos savybių galima implementuoti nerašant kodo. T. y. galima sakyti jog tai grafinė programavimo terpė. Ši grafinė terpė palaiko eilę redaktorių: lentelių redakotrius, formų redaktorius, ataskaitų redaktorius, duomenlaidžių redaktorius, kodinių redaktorius.

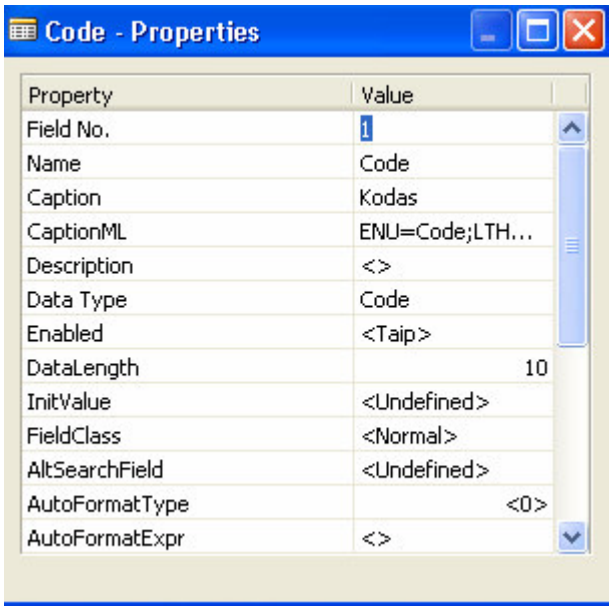
2.4.1. Lentelių redaktorius

Lentelių redaktorius yra skirtas lentelėms redaguoti. Juo galima nustatyti lentelės atributų savybes, užprogramuoti įvedamų reikšmių tikrinimą ir t. t. Pav. Nr. 20 yra parodyta lentelių grafinio redaktoriaus išvaizda:



20 pav. Lentelių redaktorius

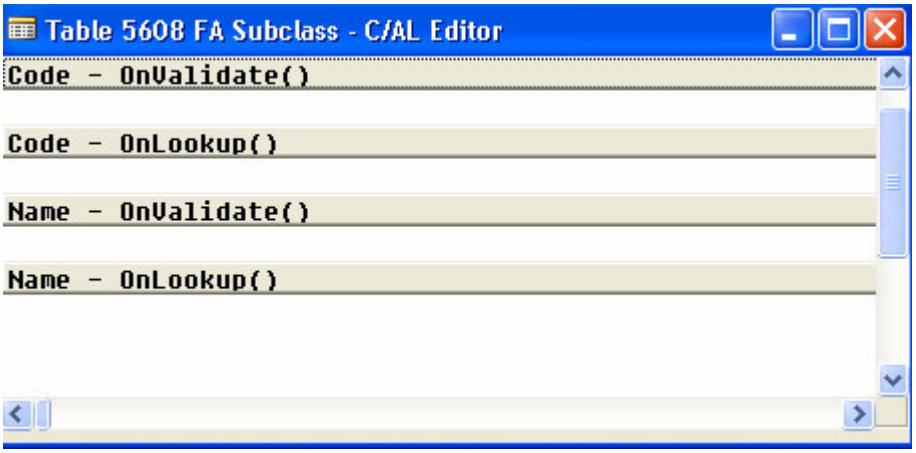
Lentelės lauko savybės yra nustatinėjamos atskirame lauko savybių (*properties*) lange. Jos apsprendžia ar laukas yra redaguojamas, kokie sąryšiai jį sieja su kitos lentelės lauku ir kt. Pav. Nr. 21 yra parodytas lauko savybių nustatymo lango dalis:



21 pav. Lauko savybių nustatymo langas

Taip pat lentelių redaktorius leidžia programuoti vadinamuose trigeriuose (*triggers*), kuriuose užprogramuotas kodas yra vykdomas atsitikus tam tikram įvykiui. Programuojama yra C/AL kalba, kuri yra panaši į Paskalį. Tai kad C/SIDE naudoja trigerius parodo, kad šioje terpėje

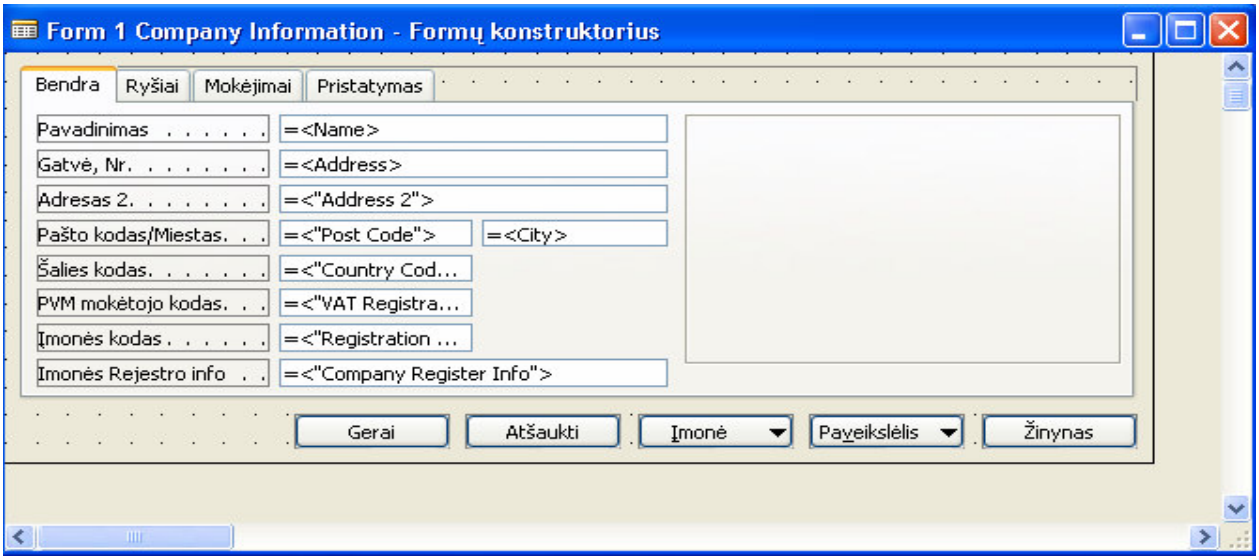
programuojamos funkcijos yra iškviečiamos arba kitų funkcijų, arba įvykių kuriuos sąlygoja vartotojas. Žemiau parodytame pav. Nr. 22 yra pateiktas langas, kuriame yra programuojama, įvykus tam tikram įvykiui kodas:



22 pav. Trigerių programavimo terpė

2.4.2. Formų redaktorius

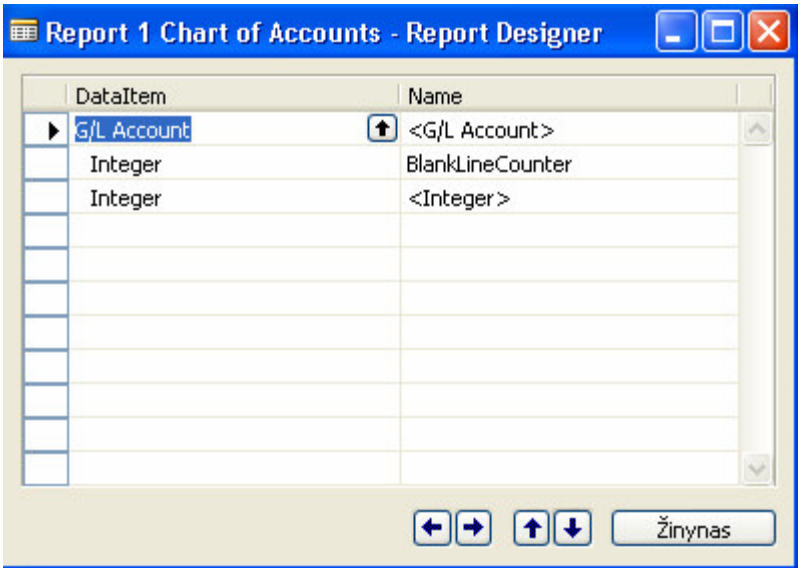
Formų redaktorius yra skirtas redaguoti formoms per kurias vartotojas valdo sistemos duomenis. Formų laukai taip pat kaip ir lentelių laukai turi savo savybes ir situacijų kodo iškvietimo trigerius. Formos būna keleto rūšių: lentelinės formos (vienas duomenų įrašas rodomas vienoj eilutėj), kortelinės formos (vienas įrašas rodomas per visą ekraną), užklausų formos (jose vartotojas nustato parametrus pvz., ataskaitai). Pav. Nr. 23 yra pateiktas kortelinės formos redagavimo būsenoje pavyzdys:



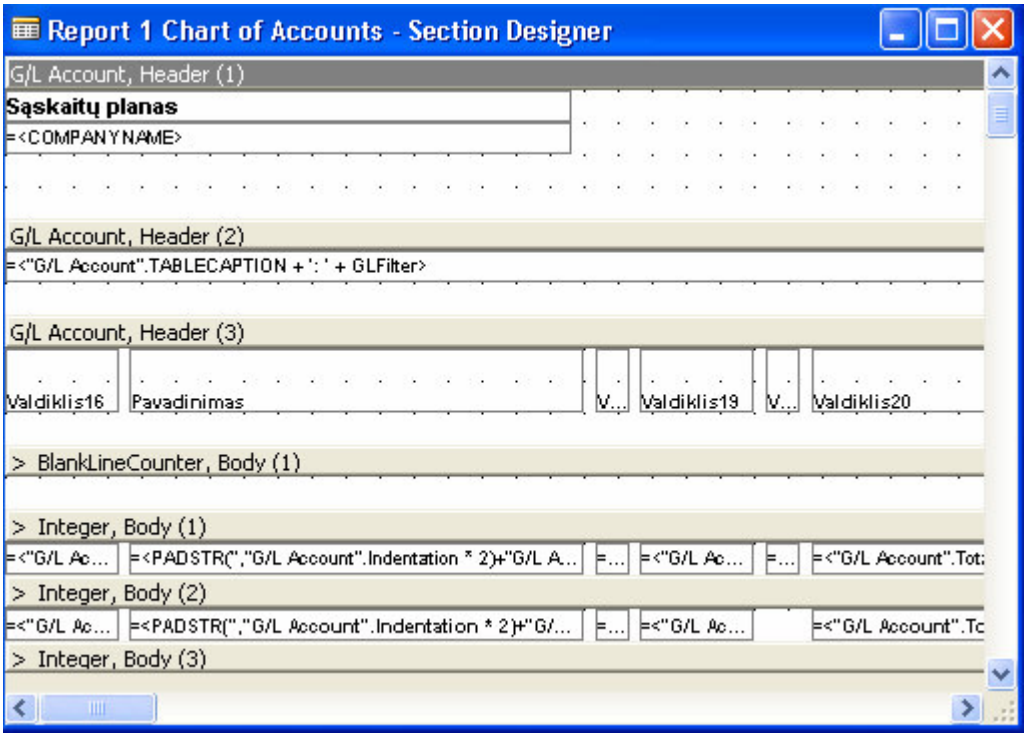
23 pav. Formų redaktorius

2.4.3. Ataskaitų redaktorius

Ataskaitų redaktorius yra skirtas redaguoti ataskaitoms. Ataskaitų redaktorius leidžia kurti ataskaitas grafinio modeliavimo būdu. Duomenys ataskaitoje yra organizuojami duomenų lygių hierarchijos pagalba. Ataskaitoje taip pat yra trigeriai kaip ir kituose sistemos objektuose, kurie leidžia formuojant ataskaitą tam tikroje vietoje atlikti laikinus duomenų pakeitimus, kaupti sumas, keisti lenteles ir t. t. Pav. Nr.24 ir pav. Nr. 25 yra ataskaitos redaktoriaus pavyzdys:



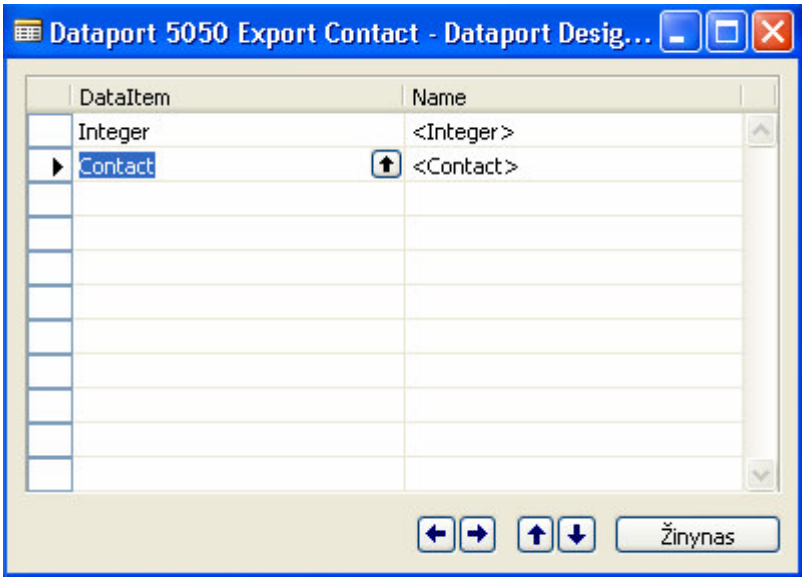
24 pav. Ataskaitos redaktoriaus langas (duomenų hierarchija)



25 pav. Ataskaitų redaktoriaus langas (sekcijų redagavimas)

2.4.4. Duomenlaidžių redaktorius

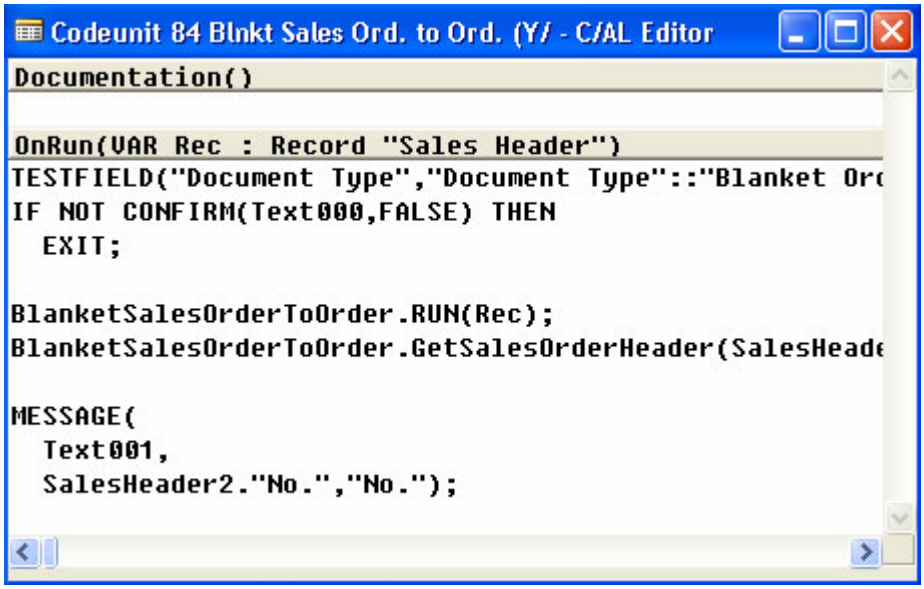
Duomenlaidės yra skirtos duomenims iš sistemos eksportuoti ir importuoti. Duomenlaidžių redaktorius leidžia panašiai kaip ir ataskaitų redaktorius nustatyti norimą duomenų hierarchiją, bei suprogramuoti duomenų pokyčius importuojant juos ar eksportuojant. Pvz., galima suprogramuoti, kad įrašo lauko reikšmė, kuri yra „false“ būtų konvertuojama į reikšmę „0“. Taip pat redaktorius leidžia nurodyti, kurie įrašo laukai bus eksportuojami ar importuojami. Pav. Nr. 26 parodytas duomenlaidžių redaktorius:



26 pav. Duomenlaidės redaktorius

2.4.5. Kodinių redaktorius

Kodiniai yra skirti didelės apimties funkcijoms rašyti ir saugoti. Kodiniai turi tik vieną trigerį „On Run“, kuris suveikia, jei kodinys yra paleidžiamas nenurodant funkcijos. Kodinių redaktorius yra paprasčiausias tekstinis redaktorius, kuris leidžia skirstyti funkcijas į atskirus kodo segmentus. Kodinio redaktoriaus pavyzdys yra pav. nr. 27.



27 pav. Kodinių redaktorius

3. SPECIFIKAVIMAS IR REALIZAVIMAS C/SIDE TERPĖJE

Šiame skyriuje bus išnagrinėti teoriniai specifikacijų rašymo aspektai Z ir UML kalbomis bei jų realizacija C/SIDE terpėje. Taip pat bus nustatyti pagrindiniai skirtumai tarp šių specifikavimo kalbų. Skyriuje bus nagrinėjami šie specifikacijų aspektai:

- UML klasės (schemos, atvaizduojančios klases Z kalboje)
- Sąryšiai (kitaip vadinamos sąsajomis):
  - „vienas su vienu“ sąryšis
  - „vienas su daug“ ir „daug su vienu“ sąryšiai
  - „daug su daug“ sąryšio problemos sprendimas

Specifikacijos bus realizuotos C/SIDE terpėje. Palyginimui yra paimta keletas pavyzdžių. Dėl ribotos darbo apimties neįmanoma panagrinėti visų įmanomų skirtumu bei panašumų tarp Z ir UML. Pagrindinis tikslas yra principialiai palyginti specifikavimo eigą, kuri vienaip ar kitaip turėtų įtakos realizuojant algoritmus.

Realizuojant specifikacijas į kodą retai reikia formalizuoti visą sistemą, kadangi daugelis dalykų gali būti intuityviai suprantami. Pernelyg didelė formali specifikacija gali pati tapti našta, nes nepatyrusiems kūrėjams ją yra sunku skaityti, bei suvokti. Taigi formalizuoti galima tik sudėtingą algoritmą arba sistemos dalį, kuri sunkiai pasiduoda aprašoma žodžiu.

Prieš pradedant specifikavimo būdų palyginimą reikia susipažinti su tam tikrais terminais naudosimais lyginime:

- klasių diagrama apibrėžia sistemos objekto tipus, bei įvarius sąryšius tarp jųjų
- klasių invariantas yra sąlyga, kuri turi būti tenkinama, kai objektas yra statinėj būsenoj
- būsenos schema apibrėžia kintamuosius ir sąryšius tarp jų
- abstraktus duomenų tipas apibrėžia kintamųjų priklausomybes aibei bei veiksmų su jomis struktūrą.

3.1. Klasės

Kiekviena UML klasė yra realizuojama kaip lentelė C/SIDE terpėje. Lentelės pavadinimas tampa lentelės vardu. Lentelės atributai tampa lentelės laukais. Klasės metodai gali būti realizuojamai kaip lentelės funkcijos, nes C/Side terpėje lentelės gali turėti savo funkcijas. Pav. Nr. 28 yra parodytas klasės „knyga“ pavyzdys. Specifikacija aprašo, kad knyga turi numerį, ISBN numerį, pavadinimą bei autorių.

Book
Book_nos ISBN_nos title author

28 pav. Klasės specifikacijos pavyzdys

Z specifikacijoje atitikmuo gali būti sugrupuoti į objektus duomenų tipų aprašai. Pirminis raktas turi būti kintamasis, kurio tipas yra susietas su kitu kintamųjų tipais injekciniu ryšiu ir jų domainai yra vienodi. Sekančioj schemoj yra parodytas objekto pavyzdys, kuri galima realizuoti kaip lentelę:

book_details

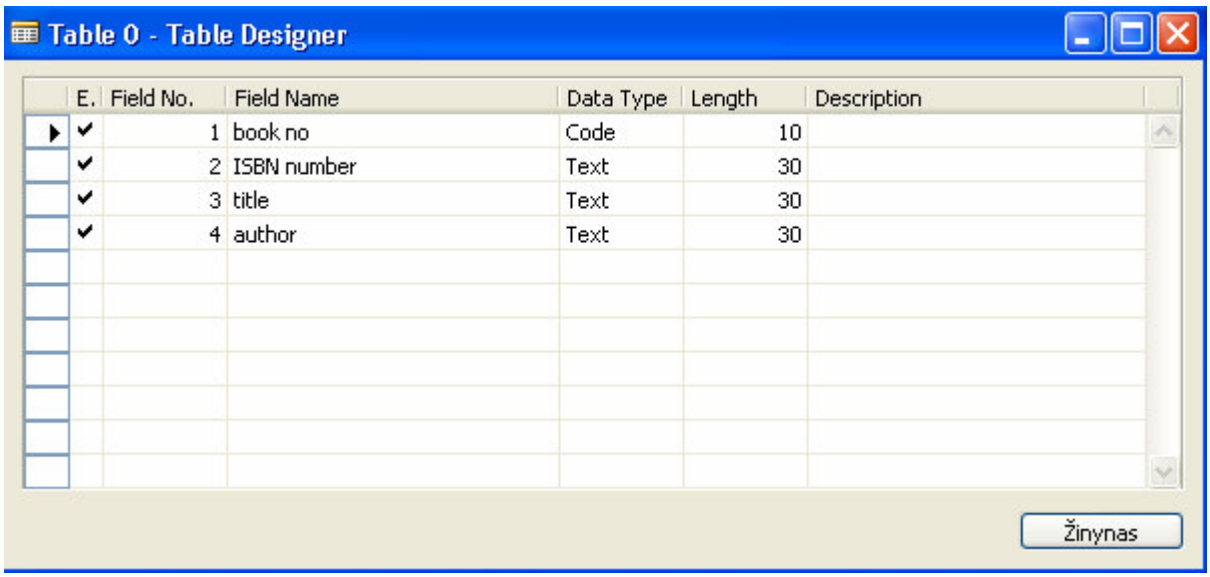
$has\_ISBN : book\_nos \rightarrowtail ISBN\_nos$

$has\_title : book\_nos \rightarrowtail titles$

$has\_author : book\_nos \rightarrowtail author\_names$

$\mathrm{dom}\ has\_ISBN = \mathrm{dom}\ has\_title = \mathrm{dom}\ has\_author$

Pav. Nr. 29 yra parodyta realizacija C/Side terpėje. Sąryšių tipas apsprendžia duomenų susietumą į vieną objektą t. y. lentelę.



29 pav. Klasės realizacija C/Side

Palyginimas:

- UML klasė ir būsenos schema Z kalboje gali būti realizuojama kaip lentelė
- UML kalboje klasėje aprašytas atributas tampa lauku lentelėje. Z kalboje statiniai kintamieji jei jie aprašomi bijekciniais sąryšio tipais gali būti apjungiami į vieną lentelę C/Side terpėje.

3.3. Z būsenų invariantai ir UML klasių invariantai

UML klasės invariantai turi būti tenkinami viso proceso vykdymo metu. Sąlygos ir invariantai yra aprašomi klasėje ir turi būti tenkinami visų susijusių klasių. Jeigu jie būna pažeisti sistemos vykdymo metu, yra iššaukiamas nenumatytas atvejis. Šis nenumatytas atvejis gali būti apdorojamas kaip pranešimas apie klaidą, jei programuotojas jį numatė. Jeigu nėra implementuotas nenumatytų atvejų valdymo mechanizmas, programa stringa.

Panašiai ir Z. Būsenos invariantas turi būti tenkinamas ir prieš operaciją ir po jos. Ši sąlyga turi būti tenkinama ir tolesniuose sistemos vystymo etapuose. To užtikrinimui yra sukuriama funkcija, kuri rezultate pateikia reikšmę „true“, jei invariantas tenkinamas ir „false“, jei netenkinamas.

Palyginimas:

- Z kalboje būsenos invariantas yra realizuojamas kaip funkcija, kuri parodo, ar operacija nepažeidė sistemos suderinamumo
- UML kalboje klasės invariantas yra aprašomas kaip klasės dalis ir jis turi būti tenkinamas visų likusių sistemos klasių.

3.4. Asociacijos (sąryšiai)

UML klasių diagramos apima klases ir sąryšius tarp jų. Sąryšiai egzistuoja įvairų tipų. Tai asociacijos, apibendrinimai, paveldėjimo, priklausomybės, tobulinimo. Dažniausiai yra naudojama asociacija , žymima vientisa linija. Kardinalumas yra rėžis, kuris nustato objektų kiekį, kurie yra susiję. Šie rėžiai gali būti įvairūs. Gali būti nulis su vienu (0 .. 1), nulis su daug (0 .. *), vienas su penkiolika (1 .. 15) ir t. t. Taip pat galima išreikšti sekas pvz., (1, 4, 6, 8 .. 12). Jeigu daugialypiškumas nėra specifikuotas, tada laikoma, kad rėžis yra (1 .. 1). Daugialypiškumas yra vaizduojamas asociacijos linijos gale, kur pažymimi rėžiai.

Z specifikavimo kalboje sąryšiai yra modeliuojami kaip matematiniai ryšiai ar funkcijos tarp esybių tipų. Nepriklausomas sąryšis Z kalboje yra modeliuojamas aibės priklausymu, kuris apibrėžiamas predikatu. Pavyzdžiui, šioms deklaracijoms :

voters : ℙ VOTER

constituents : ℙ CONSTITUENT

is_registered_with : VOTER ⇸ CONSTITUENT

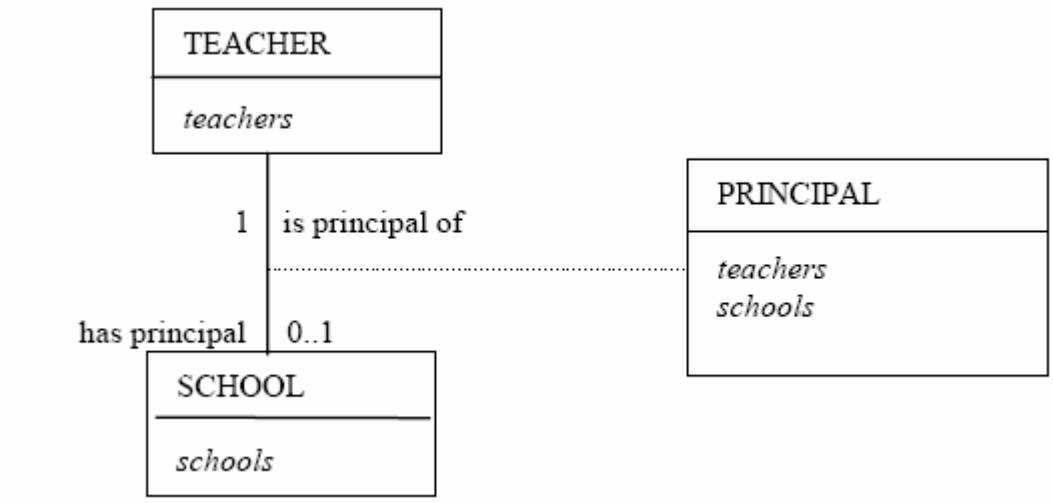
esantis sąryšis reiškia, kad

dom is_registered_with ⊆ voters

Taigi potencialus rinkėjas (*voter*) gali ir nebūti registruotas kaip tikras rinkėjas.

3.4.1. Sąryšis „vienas su vienu“

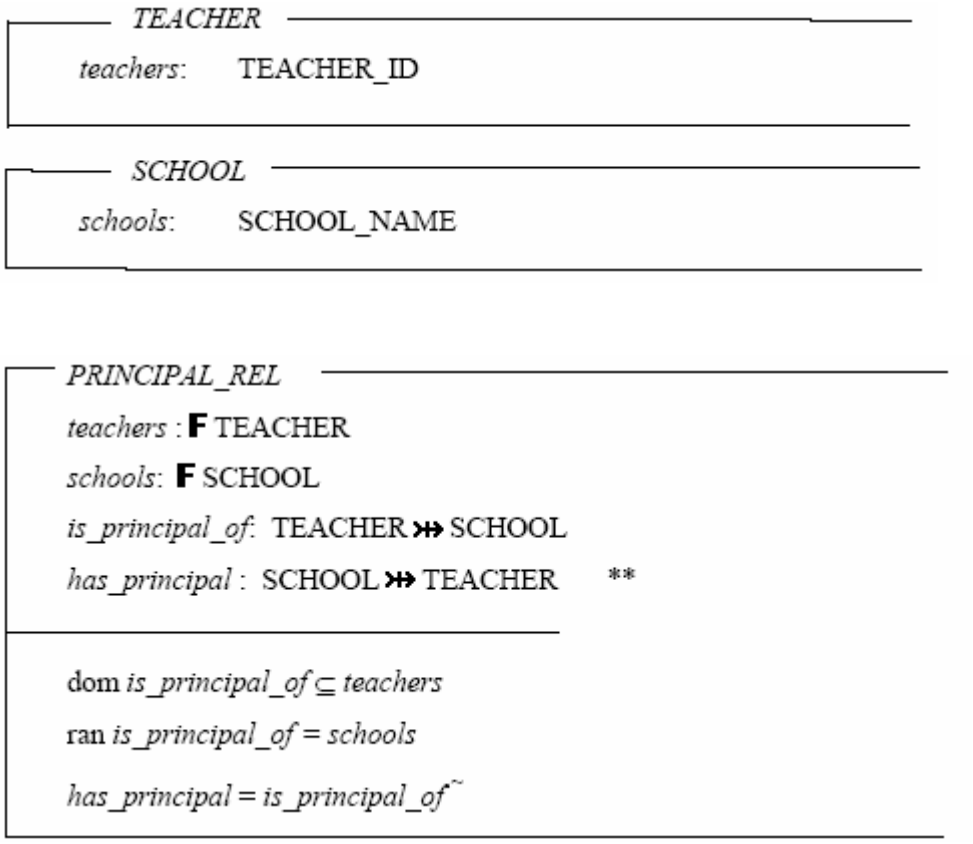
Imkime sekantį pavyzdį. Egzistuoja kelios mokyklos (*school*). Mokykloje yra tam tikras kiekis mokytojų (*teacher*). Kiekvienos mokyklos, kažkuris iš mokytojų yra direktorius (*principal*). Yra du mokytojai, kurie nėra jokios mokyklos direktoriai. UML klasių diagrama yra parodyta pav. nr. 30. Sąryšis „vienas su vienu“ pavadintas „yra mokyklos direktorius“ (*is principal of*).



30 pav. Sąryšio „vienas su vienu“ klasių diagrama

Specifikacija Z kalboje parodyta sekančiose schemose:

Pagrindiniai tipai yra [TEACHER_ID, SCHOOL_NAME]



** Pastaba. Nors kintamasis *has_principal* turėtų iš tikrųjų būti pilna injekcija (kiekviena mokykla (*schools*) turi direktorių (*is_principal_of*)), bet šiuo atveju žymėsime daline injekcija. Nes aibė visų funkcijų iš vienos aibės į kitą aibę yra poaibis visų dalinių funkcijų aibės. Z kalboje sąryšis vienas su vienu gali būti modeliuojamas kaip dalinė injekcija „⤔“. Papildomas predikatas apibrėžia santykį kaip atvirkštinį (~).

Realizuojant šį sąryšį C/SIDE galima pastebėti, kad „viens su vienu“ yra atskiras atvejis sąryšio „vienas su daug“. Taigi šio atvejo realizacijos C/Side nenagrinėsim.

Palyginimas.:

- Z specifikacijoj ryšys „vienas su vienu“ gali būti modeliuojamas kaip kaip dalinė injekcija. Papildomas predikatas apjungia injekcijas į abipusį „vienas su vienu“ sąryšį. UML specifikacijoje sąryšis „vienas su vienu“ yra vaizduojamas kaip pav. nr. 30.
- Konkrečiu atveju UML specifikacija yra aiškesnė, nes visos sistemos savybės matomos iš karto pažiūrėjus į diagramą. Z kalboje reikia nagrinėti kintamuosius ir sąsajas tarp jų.

- Z ir UML kalboje verifikavimas atliekamas atitinkamai aksiomatiniais aprašais ir instancijavimu.

3.4.2. Sąryšis „Vienas su daug“

Imkime sekantį pavyzdį. Yra keletas mokyklų (*schools*). Kiekviena mokykla turi keletą mokytojų (*teacher*). Mokytojas gali dirbti tik vienoje mokykloje. Žemiau esančioje specifikacijoje yra matyti, kad papildomas predikatas yra prieš tai esančio inversija „~“, kuris reprezentuoja sąryšį „vienas su daug“. Alternatyvus būdas Z kalboje specifikuoti sąryšį „vienas su daug“ yra dalinė funkcija, kuri parodo ryši tarp kelių objektų aibių.

Pagrindiniai tipai yra [TEACHER_ID, SCHOOL_NAME]

TEACHER

$teachers$: TEACHER_ID

SCHOOL

$schools$: SCHOOL_NAME

EMPLOYMENT

$teachers : \mathbb{F}$ TEACHER

$schools : \mathbb{F}$ SCHOOL

$works\_in$: TEACHER $\nrightarrow$ SCHOOL

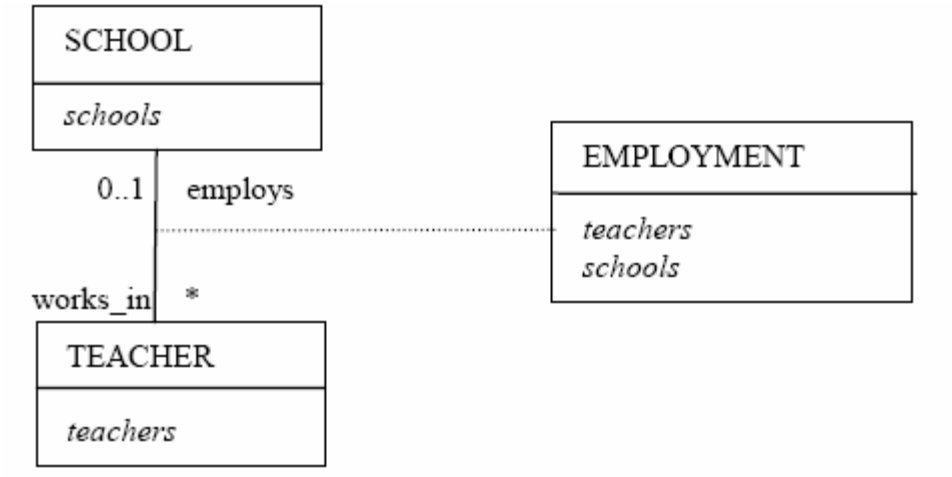
$employs$: SCHOOL $\leftrightarrow$ TEACHER

---

dom $works\_in = teachers$

ran $works\_in = schools$

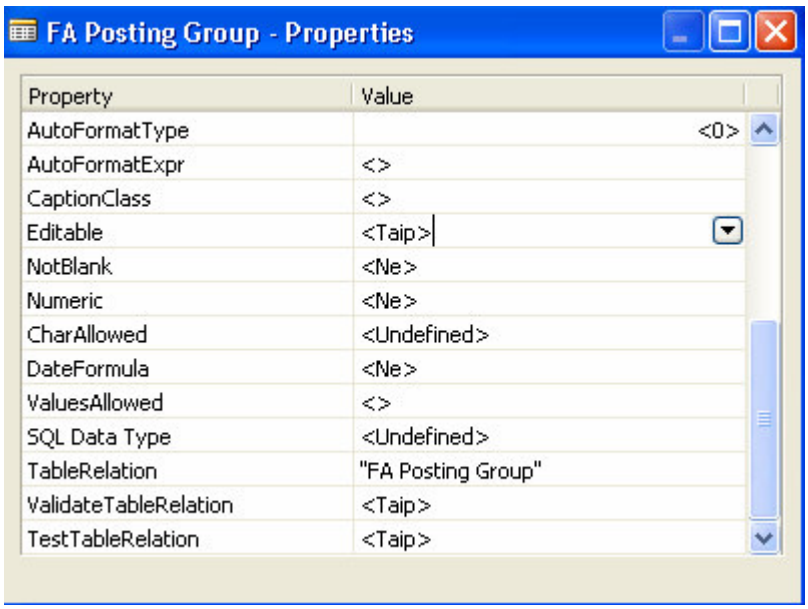
$employs = works\_in^{\sim}$



31 pav. UML klasių diagrama (sąryšis „vienas su daug“)

Pav. Nr. 31 yra parodyta UML klasių diagrama su sąryšio „vienas su daug“ atveju. Sąryšis *employment* (dirba) apibrėžia santykį tarp mokyklos (*school*) ir mokytojo (*teacher*). Viena mokykla gali turėti daug mokytojų, bet mokytojas gali dirbti tik vienoje mokykloje arba būti bedarbis.

Sąryšio realizacija C/SIDE terpėje vykdoma sekančiu būdu. Klasė, kuri yra toje asociacijos pusėj, kur nurodytas kardinalumas „*” yra realizuojama kaip lentelė, o jos siejamojo lauko su kita lentele savybėse yra nurodomas ryšys į kitą lentelę. Pav. Nr. 32 iliustruoja sąryšio realizavimą C/SIDE.



32 pav. Laukelyje “TableRelation” nurodomas sąryšis

Palyginimas:

- Z specifikacijoje sąryšis „vienas su daug“ (*employs*) yra indikuojamas ryšiu „↔“. Šio ryšio inversija yra ryšys „daug su vienu“ (*works_in*). Žiūrėti į predikatą *employs = works_in~*.
- Alternatyvus sąryšio „vienas su daug“ specifikavimo modelis yra employs = SCHOOL→↠TEACHER. UML notacijoje tai žymima asociacija (1 .. *).
- Z kalboje sąryšis „daug su vienu“ dar gali būti modeliuojamas kaip dalinė surjekcija, arba dalinė funkcija. UML kalboje šis sąryšis žymimas kaip asociacija (* .. 1).

UML ir Z specifikacijose sistemos suderinamumas yra verifikuojamas atitinkamai instancijavimu bei aksiomatinių aprašų įrodymais. Realizuojant programinį kodą ir Z, ir UML specifikacijos aprašo daugmaž vienodai sistemos detalių. Taigi modeliuojant duomenų objektus galima teigti, kad UML kalba šiuo atveju yra pranašesnė už Z. Tačiau nereikia pamiršti, kad Z specifikacija leidžia tikslų sistemos verifikavimą.

3.5. UML ir Z palyginimas

Darant išvadas galima pažymėti, kad realizacijos prasme C/SIDE terpėje UML specifikacija yra vaizdingesnė, aiškesnė bei lengviau skaitoma. Tačiau Z specifikacija gali pateikti daugiau detalių apie specifinę sritį, kas savo ruožtu lemia tikslesnę realizaciją. . Z specifikaciją galima tobulinti iki tol, kol praktiškai visos jos operacijos galėtų būti tiesiai perkeliamos į programinį kodą. Tačiau tada specifikacija taptų per daug sudėtinga skaityti. Teoriniais samprotavimais remiantis, galima teigti, , kad ir UML, ir Z specifikacijos turi pakankamą aiškumo lygį. Tik sistemos modeliai yra vaizduojami skirtingais būdais.

3.6. IŠVADOS

Šiame darbo skyriuje buvo palyginti įvairūs Z ir UML specifikavimo kalbų aspektai. Išnagrinėtas teorinis formalių bei neformalių kalbų pagrindas. Susipažinta su neformalia specifikavimo kalba UML bei formalia kalba Z. Buvo ištirta kaip skiriasi skirtingų specifikacijų realizavimas į C/SIDE programavimo terpę. Atlikus teorinį tyrimą buvo prieita prie išvadų, kad

kiekviena kalba atskirai (UML ir Z) turi ir privalumų, ir trūkumų. Palyginimui žiūrėti į lentelę nr. 1.

Lentelė nr. 1 Z ir UML kalbų palyginimas

	Z	UML
Stipriosios pusės	1. Matematiškai tiksli. 2. Gali sumažinti specifikacijos klaidas. 3. Gali sumažinti projekto kaštus, bei realizacijos klaidas. 4. Leidžia korektiškumo patikrinimą. 5. Draudžia dviprasmybes.	1. Lengvai suprantama klientų. 2. Daug tikslesnė nei kitos neformalios specifikacijos. 3. Daug kūrėjų jas naudoja. 4. Specifikacija yra grafinė, todėl lengviau suprantama. 5. Gera komunikavimo priemonė tarp nepatyrusių diegėjų.
Silpnosios pusės	1. Sunkiai išmokstama 2. Sudėtinga naudoti 3. Dažnai specifikacija sunkiai suprantama klientams 4. Formalūs metodai patys savaime yra sunkiau išmokstami ir naudojami nei realizacijos terpė	1. Specifikacija nėra griežtai tiksli. 2. Procesų bei duomenų suderinamumą yra sunku patikrinti.

Darant tolesnes išvadas, galima pasakyti, kad Z specifikacija naudingesnė yra programinio kodo kūrėjui. T. y. Z specifikacijos tinka aprašyti sistemos funkcijoms, kurių realizacijos vieta yra kodiniuose. Tačiau negalima atmesti teiginio, kad stambiam abstrakcijos lygyje galima naudingai specifikuoti Z kalboje tam tikras sistemos būsenas. Procesų bei duomenų modelių projektavimui

ir specifikavimui geriau naudoti UML. Dar geriau būtų, jei teisingai kombinuojant šias dvi kalbas būtų galima pasiekti ir aiškią, ir detalią sistemos specifikaciją.

4. PRAKTINIS TYRIMAS: „ILGALAIKIO TURTO APSKAITA“ SISTEMOS SPECIFIKACIJŲ TYRIMAS

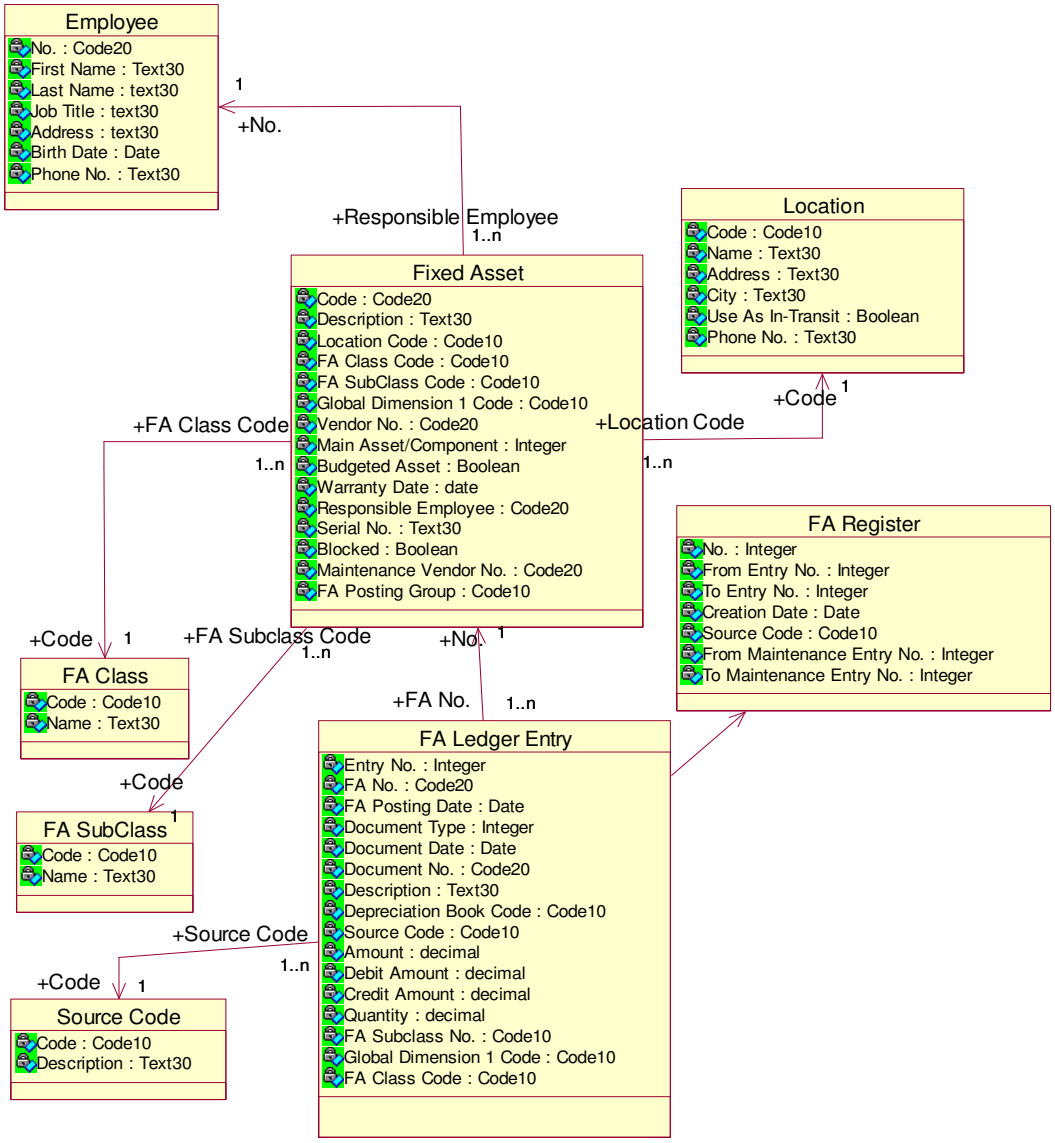
Praktiniam tyrimui atlikti pasirinkta „Ilgalaikio turto apskaitos“ sistema. Sistemos duomenų modelis specifikuojamas UML ir Z kalbomis. Palyginama skirtingų specifikacijų realizacijos į C/SIDE terpę. Palyginamas duomenų modelio aiškumas, suprantamumas. Padaromos išvados. Praktinės išvados palyginamos su teorinėmis. Pateikiamos rekomendacijos. UML specifikacijos yra pateiktos priede (žr. 2 priedą).

4.1. Sistemos reikalavimai

Įmonei reikalinga ilgalaikio turto apskaitos sistema. Kiekvienas ilgalaikis turtas turi atsakingą asmenį. Ilgalaikis turtas skirstomas pagal grupes, pogrupius ir vietas. Ilgalaikio turto istorija yra kaupiama. Kaupiant informaciją yra būtina nurodyti šaltinį, kuris parodo istorijos įrašo kilmę. Įvedinėjant naują ilgalaikį turtą reikia tikrinti ar jis nėra jau apskaitomas. Trinant iš apskaitos ilgalaikį turtą, reikia tikrinti ar jis neturi istorijos įrašų. Trinant ar šalinant grupes, pogrupius ir vietas tikrinti ar nėra jiems priskirtų ilgalaikių turtų. Ilgalaikis turtas gali būti apdraudžiamas. Sistema realizuojama MSBS Navision terpėje. Įterpinėjant įrašus reikia tikrinti, kad kai įvedamas įsigijimo tipo įrašas tam ilgalaikiui turtui sistemoje nebūtų įrašų. Kai įvedinėjamas nuvertėjimo įrašas reikia stebėti, kad būtų tam ilgalaikiui turtui įsigijimo įrašas.

4.2. Duomenų objektų modelis

Pradžioje specifikuojant sistemą yra svarbu nustatyti sistemos duomenų modelį. Konkrečiu atvejų atskiri sistemos objektai bus: Ilgalaikis turtas, Ilgalaikio turto pogrupis, Ilgalaikio turto grupė, ilgalaikio turto vieta, ilgalaikio turto istorijos įrašas ir kt. Pav. nr. 33 yra pateikta UML klasių diagrama sistemos duomenų modeliui. Ilgalaikio turto grupė gali turėti daug ilgalaikio turto vienetų. Taigi šiuo atveju yra vaizduojama asociacija (1 .. *). Analogiškai pogrupiui, bei vietai. Ilgalaikis turtas gali turėti daug istorijos įrašų, todėl asociacijos kardinalumai yra vaizduojami atvirkščiai ilgalaikio turto atžvilgiu nei IT vieneto ir pvz., grupės.



33 pav. Ilgalaikio turto duomenų modelio UML klasių diagrama

Sistemos duomenų modelio Z specifikacija yra pateikta žemiau esančioje ištraukoje iš priedo nr. 1.

Pradžioje yra apibrėžiami sistemos abstraktūs tipai.

[*FA_NOS*, *EMPL_NOS*, *LOC_NOS*, *SUBC_NOS*, *C_NOS*, *LEDGER_NOS*]

[*NAME*, *JOB*, *ADDRESS*, *DATE*, *DESCR*]

Sekančioje schemoje yra specifikuoti sistemos objektų sąryšiai. UML kalboje tai būtų asociacijos.

Rel_Schema

$$
\begin{aligned}
&\mathit{employee}: \mathbb{P}\ \mathit{EMPL\_NOS}\\
&\mathit{fa}: \mathbb{P}\ \mathit{FA\_NOS}\\
&\mathit{loc}: \mathbb{P}\ \mathit{LOC\_NOS}\\
&\mathit{subclass}: \mathbb{P}\ \mathit{SUBC\_NOS}\\
&\mathit{class}: \mathbb{P}\ \mathit{C\_NOS}\\
&\mathit{ledger}: \mathbb{P}\ \mathit{LEDGER\_NOS}\\
&\mathit{rel\_fa\_empl}: \mathit{FA\_NOS} \pfun \mathit{EMPL\_NOS}\\
&\mathit{rel\_fa\_loc}: \mathit{FA\_NOS} \pfun \mathit{LOC\_NOS}\\
&\mathit{rel\_fa\_subclass}: \mathit{FA\_NOS} \pfun \mathit{SUBC\_NOS}\\
&\mathit{rel\_fa\_class}: \mathit{FA\_NOS} \pfun \mathit{C\_NOS}\\
&\mathit{rel\_fa\_ledger}: \mathit{FA\_NOS} \pfun \mathit{LEDGER\_NOS}\\
&\mathit{rel\_empl\_fa}: \mathit{EMPL\_NOS} \rightarrow \mathit{FA\_NOS}\\
&\mathit{rel\_loc\_fa}: \mathit{LOC\_NOS} \rightarrow \mathit{FA\_NOS}\\
&\mathit{rel\_subclass\_fa}: \mathit{SUBC\_NOS} \rightarrow \mathit{FA\_NOS}\\
&\mathit{rel\_class\_fa}: \mathit{C\_NOS} \rightarrow \mathit{FA\_NOS}\\
&\mathit{rel\_ledger\_fa}: \mathit{LEDGER\_NOS} \rightarrow \mathit{FA\_NOS}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{dom}\ \mathit{rel\_fa\_empl} = \mathit{fa}\\
&\mathrm{dom}\ \mathit{rel\_fa\_loc} = \mathit{fa}\\
&\mathrm{dom}\ \mathit{rel\_fa\_subclass} = \mathit{fa}\\
&\mathrm{dom}\ \mathit{rel\_fa\_class} = \mathit{fa}\\
&\mathrm{dom}\ \mathit{rel\_fa\_ledger} = \mathit{fa}\\
&\mathrm{ran}\ \mathit{rel\_fa\_empl} = \mathit{employee}\\
&\mathrm{ran}\ \mathit{rel\_fa\_loc} = \mathit{loc}\\
&\mathrm{ran}\ \mathit{rel\_fa\_subclass} = \mathit{subclass}\\
&\mathrm{ran}\ \mathit{rel\_fa\_class} = \mathit{class}\\
&\mathrm{ran}\ \mathit{rel\_fa\_ledger} = \mathit{ledger}\\
&\mathit{rel\_empl\_fa} = \mathit{rel\_fa\_empl}^{\sim}\\
&\mathit{rel\_loc\_fa} = \mathit{rel\_fa\_loc}^{\sim}\\
&\mathit{rel\_subclass\_fa} = \mathit{rel\_fa\_subclass}^{\sim}\\
&\mathit{rel\_class\_fa} = \mathit{rel\_fa\_class}^{\sim}\\
&\mathit{rel\_ledger\_fa} = \mathit{rel\_fa\_ledger}^{\sim}
\end{aligned}
$$

Schemoje *Employee_Schema* aprašomas UML klasės *Employee* atitikmuo. T .y. darbuotojas ir jo atributai.

Employee_Schema

ΔRel_Schema

$e\_name, e\_name' : EMPL\_NOS \rightarrowtail\!\!\!\!\!\!\rightarrow NAME$

$e\_job : EMPL\_NOS \rightarrowtail\!\!\!\!\!\!\rightarrow JOB$

$e\_address : EMPL\_NOS \rightarrowtail\!\!\!\!\!\!\rightarrow ADDRESS$

$e\_birth : EMPL\_NOS \rightarrowtail\!\!\!\!\!\!\rightarrow DATE$

$\mathrm{dom}\ e\_name = employee$

$\mathrm{dom}\ e\_job = employee$

$\mathrm{dom}\ e\_address = employee$

$\mathrm{dom}\ e\_birth = employee$

Apibrėžiamas tipas *BOOL*, kuris gali turėti „*true*" arba „*false*" reikšmes.

$BOOL ::= TRUE \mid FALSE$

Schemoje Fa_Schema apibrėžtas ilgalaikio turto objektas ir su juo susiję atributai.

Fa_Schema

ΞRel_Schema

$fa\_descr : FA\_NOS \rightarrowtail\!\!\!\!\!\!\rightarrow DESCR$

$fa\_blocked : FA\_NOS \rightarrowtail\!\!\!\!\!\!\rightarrow BOOL$

$fa\_insured : FA\_NOS \rightarrowtail\!\!\!\!\!\!\rightarrow BOOL$

$fa\_war\_date : FA\_NOS \rightarrowtail\!\!\!\!\!\!\rightarrow DATE$

$\mathrm{dom}\ fa\_descr = fa$

$\mathrm{dom}\ fa\_blocked = fa$

$\mathrm{dom}\ fa\_insured = fa$

$\mathrm{dom}\ fa\_war\_date = fa$

Schema *Fa_Ledger_Schema* aprašo ilgalaikio turto istorijos įrašo objekto atributus ir sąryšį su pačiu objektu.

Fa_Ledger_Schema

Ξ*Rel_Schema*
Posting_Date: LEDGER_NOS $\rightarrowtail$ *DATE*
Doc_No: LEDGER_NOS $\rightarrowtail$ *DOCNO*
Doc_Date: LEDGER_NOS $\rightarrowtail$ *DATE*
Entry_Descr: LEDGER_NOS $\rightarrowtail$ *DESCR*
Quantity: LEDGER_NOS $\rightarrowtail$ $\mathbb{N}$
Amount: LEDGER_NOS $\rightarrowtail$ $\mathbb{Z}$
EntryType: LEDGER_NOS $\rightarrowtail$ *ENTRYTYPE*

---

dom *Doc_No* = *ledger*
dom *Doc_Date* = *ledger*
dom *Posting_Date* = *ledger*
dom *Entry_Descr* = *ledger*
dom *Quantity* = *ledger*
dom *Amount* = *ledger*

Iš pavyzdžio matome, kad Z specifikacija didelės apimties ir sunkiai skaitoma. Specifikuojant reikia apibrėžti kiekvieną abstraktų duomenų tipą bei sąryšį su pačiu objektu. Akivaizdu, kad realizuojant duomenų modelį į MSBS NAVISION C/SIDE terpę UML specifikacija yra tinkamesnė, nes yra lengviau skaitoma ir suprantama. Tačiau Z specifikacijoje apibrėžti duomenų modelį yra būtina, kadangi jie vėliau naudojami apibrėžiant įvairias operacijas, kurios savo ruožtu gali būti verifikuojamos. Duomenų modelio realizacija yra pateikiama priede nr. 3.

4.3. OPERACIJOS

Operacijos UML kalboje gali būti specifikuojamos įvairiomis diagramomis. Tai gali būti sekų diagrama, bendradarbiavimo diagrama, būsenų diagrama ir kt. Įvairūs diagramų pavyzdžiai yra pateikti 2 skyriuje. Kadangi specifikacijos, projektuojant MSBS Navision verslo funkcijas, dažniausiai yra pateikiamos UML diagramomis, tai praktiniame tyrime atskirai jų detaliau nenagrinėsim. Užduoties UML specifikacijos yra 2 priede. Žemiau pateiktame pavyzdyje yra Z kalba specifikuota nusidėvėjimo įrašo įterpimo operacija.:

add_depr_entry

ΔFa_Ledger_Schema
a_ledger?: LEDGER_NOS
a_fa?: FA_NOS
a_Posting_Date?: DATE
a_Doc_No?: DOCNO
a_Doc_Date?: DATE
a_Entry_Descr?: DESCR
a_Quantity?: $\mathbb{N}$
a_Amount?: $\mathbb{Z}$
a_EntryType?: ENTRYTYPE

$\exists x: LEDGER\_NOS \bullet x = rel\_fa\_ledger\ a\_fa?$
$ledger' = ledger \cup \{a\_ledger?\}$
$Posting\_Date' = Posting\_Date \cup \{(a\_ledger? \mapsto a\_Posting\_Date?)\}$
$Doc\_No' = Doc\_No \cup \{(a\_ledger? \mapsto a\_Doc\_No?)\}$
$Doc\_Date' = Doc\_Date \cup \{(a\_ledger? \mapsto a\_Doc\_Date?)\}$
$Entry\_Descr' = Entry\_Descr \cup \{(a\_ledger? \mapsto a\_Entry\_Descr?)\}$
$Quantity' = Quantity \cup \{(a\_ledger? \mapsto a\_Quantity?)\}$
$Amount' = Amount \cup \{(a\_ledger? \mapsto a\_Amount?)\}$
$EntryType' = EntryType \cup \{(a\_ledger? \mapsto a\_EntryType?)\}$

Specifikacija nusako kaip įrašas yra įterpiamas į įrašų aibę, su jam priklausančiais atributais. Predikatas $\exists x: LEDGER\_NOS \bullet x = rel\_fa\_ledger\ a\_fa?$ reiškia, kad įterpiant įrašą į istorijos įrašų aibę, turi būti nors vienas atitinkamo ilgalaikio turto įrašas. Predikatas ΔFa_Ledger_Schema parodo, kad įrašų sistemos būsena apibrėžta pradžioje gali būti keičiama. T. y. gali atsirasti sistemoje papildomas įrašas. Siekiant specifikaciją padaryti tikslesnę galima aprašyti klaidų pranešimų būsenas, kurios yra tenkinamos, kai operacija nepavyksta. Pvz.,

Depreciation_Check

ΞFa_Ledger_Schema
$error!: MESSAGE$
$a\_fa?: FA\_NOS$

$\forall x: LEDGER\_NOS \bullet x \neq rel\_fa\_ledger\ a\_fa?$
$error! = Acquisition\_Must\_Exsist$

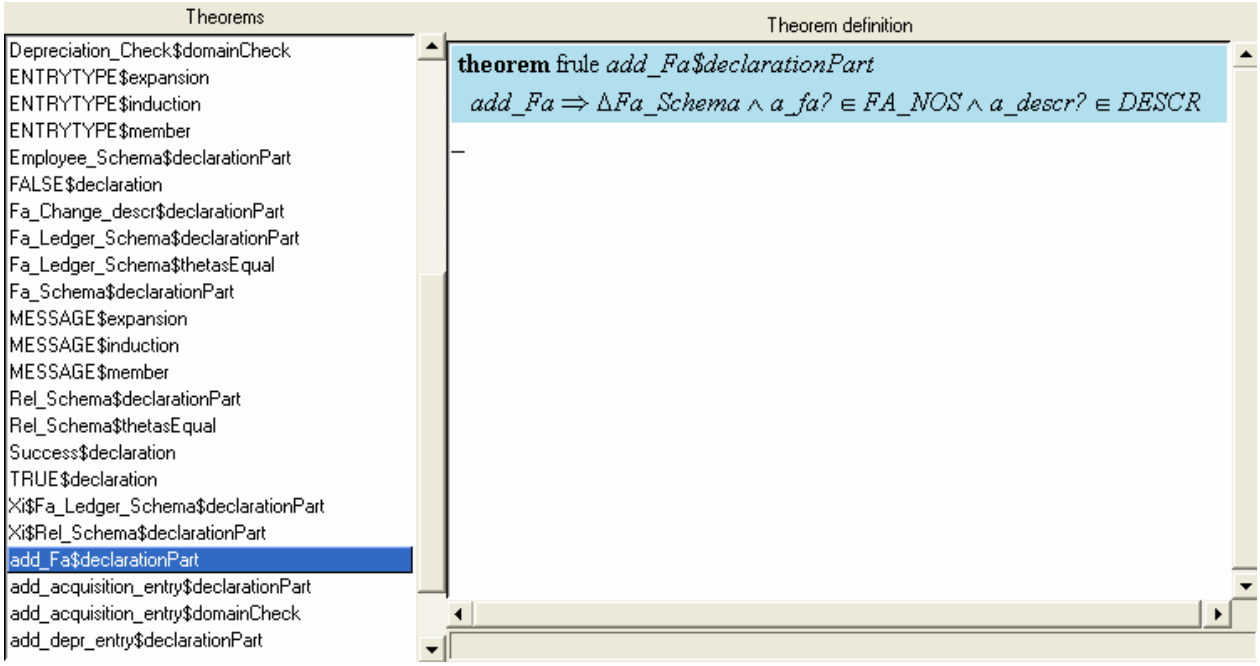
Depreciation_Check schema apibrėžia būseną, kai vykdant operaciją *add_depr_entry* įvyksta klaida. Tada gaunamas pranešimas apie klaidą. T. y. vykdant operaciją nėra tenkinami schemos *add_depr_entry* prieš-sąlygos, apibrėžiamos predikatu $\exists x: LEDGER\_NOS \bullet x = rel\_fa\_ledger\ a\_fa?$. Operacijos ir pranešimų schemos yra apjungiamos į vieną sekančiu būdu:

$$InsertEntry \mathrel{\widehat=} (add\_depr\_entry \lor Depreciation\_Check)$$

Schema *InsertEntry* visada privalo turėti loginę reikšmę „t*rue*“.

4.4. Verifikavimas

Operacijų schemos gali būti verifikuojamos. Naudojantis Z/Eves įrankį galima generuoti teoremas, kuriuos gali būti įrodomos Z/Eves skriptų pagalba. Pav. Nr. 34. yra parodyta Z/Eves sugeneruota pavyzdinė teorema *add_Fa$declarationPart*:



34. pav. Z/Eves sugeneruota teorema add_Fa$declarationPart

Įvairių operacijų realizacijų C/SIDE terpėje pavyzdžiai yra pateikti priede nr. 3. Operacijas galima specifikuoti Z kalboje įvairiais detalumo lygiais. Nuo labai abstrakčių struktūrų iki beveik programinio kodo lygio. Be to šis iteracinis procesas gali būti verifikuojamas. C/Side terpės atžvilgiu nėra būtinas labai detalus operacijų specifikavimas, nes galų gale pats programinis kodas gaunasi aiškesnis už specifikaciją. UML specifikavimo kalba taip pat užtikrina įvairius detalumo lygius. Tačiau UML specifikacijų tikslinimo proceso neįmanoma verifikuoti.

4.5. Formalių metodų panaudojimo galimybės

Siekiant į specifikacijų tobulinimą bei detalizavimą įnešti tikslumo, galima kombinuotai naudoti Z specifikacijas. T. y. nėra būtina kiekvieną procesą pilnai ir detaliai formalizuoti. Užtenka formalizuoti tam tikras sudėtingas situacijas arba vietas, kur UML specifikacija pati savaime sunkiai užtikrina suderinamumą. Taip pat Z specifikacijas galima panaudoti detalizuojant UML diagramas. T.y. būtų galima abstrakčią UML specifikaciją sutransliuoti į Z. Ir joje verifikuojant pereiti prie detalesnio sistemos modelio. Po to galima vėl konvertuoti į UML. Tokiu atveju būtų užtikrinamas tikslus UML specifikacijos detalizavimas. Šiam procesui nereikėtų daug žmonių išmanančių Z specifikavimo kalbą, kuri yra sunkiai skaitoma ir išmokstama. Praktikoje yra keli automatizuoti įrankiai, tokie kaip „Roze“, kurie transliuoja UML

specifikacijas į Z kalba, tačiau jie nėra pilnai ištestuoti, apima ne visus įmanomus sistemos modeliavimo elementus ir t. t.

Tam tikram abstraktumo lygyje Z specifikacija gali pakankamai aiškiai ir tiksliai aprašyti sistemos būsenas, kad jas galėtų skaityti vidutiniškai patyręs programuotojas. Kai kuriais atvejais Z kalbos struktūra galėtų netgi padėti įsivaizduoti, kur kokiam įvykio trigeryje turi būti vykdoma funkcija, kokios sąlygos apriboja funkciją ir t. t. Pvz., Duomenlaidės trigeriai: „ON BEFORE EXPORT“, „ON AFTER EXPORT“. Galima būtų Z kalboje aprašyti sąlygas, kurios turi būti tenkinamos šiuose trigeriuose.

Baigiant praktinės dalies aprašymą galima konstatuoti, kad UML yra tikrai patogus įrankis sistemos objektiniam modeliui bei stambioms operacijoms aprašyti. Z specifikacija yra naudinga sistemos modelio derinimo ir tikslinimo iteraciniame procese, bei specifikuojant tam tikras atskiras sistemos modelio vietas, kurios negali būti vaizduojamos grafiškai, arba kur šis vaizdavimas sukeltų dviprasmybių. Pilnas formalus verslo valdymo funkcijų specifikavimas nėra būtinas. Formalūs metodai labiau tinka specifikuoti, pvz., kritinėms sistemoms.

5. IŠVADOS

1. Abejomis specifikavimo kalbomis (UML ir Z) galima specifikuoti verslo valdymo sistemas, realizuojamas MSBS Navision terpėje.
2. UML kalba yra intuityvesnė, vaizdingesnė, lengviau suprantama ir išmokstama. Jos pagrindinis trūkumas yra tai, kad nėra galimybės tiksliai ir vienareikšmiškai verifikuoti sistemos modelio. Negalima užtikrinti iteracinio specifikacijos detalizavo proceso suderinamumo.
3. Z specifikacija yra žymiai sunkiau skaitoma bei išmokstama nei UML. Z specifikaciją galima tiksliai verifikuoti. Panaudojant formalius metodus yra įmanoma užtikrinti suderinamumą, pereinant nuo abstraktaus specifikavimo lygio iki detalaus. Kuriant verslo valdymo sistemas MSBS Navision terpėje, tikslinga formalizuoti ir verifikuoti tik dalį funkcijų, kurių neformalus aprašymas gali sukelti dviprasmybių. Vienareikšmiška formali specifikacija gali padėti spręsti ginčus tarp užsakovo ir rangovo.
4. Pilna pradinės sistemos formali specifikacija leistų patikrinti, ar sistemos papildymas yra suderinamas su pačia pradine sistema. Tai galima padaryti verifikuojant apjungtas formalias specifikacijas. Tačiau pilnas sistemos specifikavimas reikalauja daug laiko ir resursų. Tikslinga Z kalba specifikuoti tas sistemos vietas, kurios realizuojamos nenaudojant objektinio programavimo.
5. Siekiant specifikaciją padaryti aiškią ir tikslią, galima kombinuotai naudoti UML ir Z specifikavimo kalbas.
6. Tobulinant ir detalizuojant UML specifikacijas galima panaudoti formalius metodus. Taip būtų užtikrinama skirtingų abstrakcijos lygių specifikacijų suderinamumas.
7. Šio tyrimo rezultatų patikimumas nėra didelis. Praktiniam ir teoriniam tyrime nebuvo išanalizuoti visi formalių bei neformalių specifikacijų taikymo aspektai. Reikėtų atlikti tyrimą bent keliuose stambiuose projektuose, kad būtų galima tiksliai įvardinti formalių metodų panaudojimo naudą, kuriant verslo valdymo sistemas MSBS Navision terpėje.

6. LITERATŪRA

1. Spivey J. M. The Z Notation, Second Edition,1992. Žiūrėta [2003-12-10].Prieiga per internetą: www.oxford.co.uk/spivey/zrm.pdf.
2. INCE D.C. An Introduction to Discrete Mathematics and Formal System Specification, CLARENDON PRESS, OXFORD,1988.
3. Spivey J. M. An introduction to Z and formal specifications. Software Engineering Journal, January 1992.
4. Toyn Ian Formal Specification – Z Notation – Syntax, Type and Semantics, 2000. Žiūrėta [2004-02-10]. Prieiga per internetą: http://wwwold.dkuug.dk/jtc1/sc22/open/n3187.pdf
5. Introduction to OMG‘s Unified Modelling Language, 2004. Žiūrėta [2004-03-24]. Prieiga per internetą: http://www.omg.org/gettingstarted/what_is_uml.htm
6. Navision OnTarget methodology, MSBS Navision documents, 2002

7. PRIEDAI

1. PRIEDAS. Ilgalaikio turto apskaitos sistemos Z specifikacija

[*FA_NOS*, *EMPL_NOS*, *LOC_NOS*, *SUBC_NOS*, *C_NOS*, *LEDGER_NOS*]

[*NAME*, *JOB*, *ADDRESS*, *DATE*, *DESCR*]

Rel_Schema

$$
\begin{array}{l}
employee: \mathbb{P}\ EMPL\_NOS \\
fa: \mathbb{P}\ FA\_NOS \\
loc: \mathbb{P}\ LOC\_NOS \\
subclass: \mathbb{P}\ SUBC\_NOS \\
class: \mathbb{P}\ C\_NOS \\
ledger: \mathbb{P}\ LEDGER\_NOS \\
rel\_fa\_empl: FA\_NOS \pfun EMPL\_NOS \\
rel\_fa\_loc: FA\_NOS \pfun LOC\_NOS \\
rel\_fa\_subclass: FA\_NOS \pfun SUBC\_NOS \\
rel\_fa\_class: FA\_NOS \pfun C\_NOS \\
rel\_fa\_ledger: FA\_NOS \pfun LEDGER\_NOS \\
rel\_empl\_fa: EMPL\_NOS \rel FA\_NOS \\
rel\_loc\_fa: LOC\_NOS \rel FA\_NOS \\
rel\_subclass\_fa: SUBC\_NOS \rel FA\_NOS \\
rel\_class\_fa: C\_NOS \rel FA\_NOS \\
rel\_ledger\_fa: LEDGER\_NOS \rel FA\_NOS \\
\hline
\mathrm{dom}\ rel\_fa\_empl = fa \\
\mathrm{dom}\ rel\_fa\_loc = fa \\
\mathrm{dom}\ rel\_fa\_subclass = fa \\
\mathrm{dom}\ rel\_fa\_class = fa \\
\mathrm{dom}\ rel\_fa\_ledger = fa \\
\mathrm{ran}\ rel\_fa\_empl = employee \\
\mathrm{ran}\ rel\_fa\_loc = loc \\
\mathrm{ran}\ rel\_fa\_subclass = subclass \\
\mathrm{ran}\ rel\_fa\_class = class \\
\mathrm{ran}\ rel\_fa\_ledger = ledger \\
rel\_empl\_fa = rel\_fa\_empl^{\sim} \\
rel\_loc\_fa = rel\_fa\_loc^{\sim} \\
rel\_subclass\_fa = rel\_fa\_subclass^{\sim} \\
rel\_class\_fa = rel\_fa\_class^{\sim} \\
rel\_ledger\_fa = rel\_fa\_ledger^{\sim}
\end{array}
$$

Employee_Schema

ΔRel_Schema

$e\_name, e\_name' : EMPL\_NOS \rightarrowtail\!\!\!\!\!\!\rightarrow NAME$

$e\_job : EMPL\_NOS \rightarrowtail\!\!\!\!\!\!\rightarrow JOB$

$e\_address : EMPL\_NOS \rightarrowtail\!\!\!\!\!\!\rightarrow ADDRESS$

$e\_birth : EMPL\_NOS \rightarrowtail\!\!\!\!\!\!\rightarrow DATE$

$\mathrm{dom}\ e\_name = employee$

$\mathrm{dom}\ e\_job = employee$

$\mathrm{dom}\ e\_address = employee$

$\mathrm{dom}\ e\_birth = employee$

AddEmployee

$\Delta Employee\_Schema$

$a\_employee? : EMPL\_NOS$

$a\_name? : NAME$

$a\_job? : JOB$

$a\_address? : ADDRESS$

$a\_birth? : DATE$

$a\_employee? \notin employee$

$employee' = employee \cup \{a\_employee?\}$

$e\_name' = e\_name \cup \{(a\_employee? \mapsto a\_name?)\}$

$e\_job' = e\_job \cup \{(a\_employee? \mapsto a\_job?)\}$

$e\_address' = e\_address \cup \{(a\_employee? \mapsto a\_address?)\}$

$e\_birth' = e\_birth \cup \{(a\_employee? \mapsto a\_birth?)\}$

remove_employee

$\Delta Employee\_Schema$

$rem\_employee? : EMPL\_NOS$

$rel\_empl\_fa = \emptyset$

$rem\_employee? \in employee$

$employee' = employee \setminus \{rem\_employee?\}$

$e\_name' = \{rem\_employee?\} \lhd e\_name$

$e\_job' = \{rem\_employee?\} \lhd e\_job$

$e\_address' = \{rem\_employee?\} \lhd e\_address$

$e\_birth' = \{rem\_employee?\} \lhd e\_birth$

$BOOL ::= TRUE \mid FALSE$

Fa_Schema

ΞRel_Schema
$fa\_descr: FA\_NOS \rightarrowtail\!\!\!\!\!\!\rightarrow DESCR$
$fa\_blocked: FA\_NOS \rightarrowtail\!\!\!\!\!\!\rightarrow BOOL$
$fa\_insured: FA\_NOS \rightarrowtail\!\!\!\!\!\!\rightarrow BOOL$
$fa\_war\_date: FA\_NOS \rightarrowtail\!\!\!\!\!\!\rightarrow DATE$

$\mathrm{dom}\ fa\_descr = fa$
$\mathrm{dom}\ fa\_blocked = fa$
$\mathrm{dom}\ fa\_insured = fa$
$\mathrm{dom}\ fa\_war\_date = fa$

add_Fa

ΔFa_Schema
$a\_fa?: FA\_NOS$
$a\_descr?: DESCR$

$a\_fa? \notin fa$
$fa' = fa \cup \{a\_fa?\}$
$fa\_descr' = fa\_descr \cup \{(a\_fa? \mapsto a\_descr?)\}$
$fa\_blocked' = \{\}$
$fa\_insured' = \{\}$
$fa\_war\_date = \{\}$

remove_fa

ΔFa_Schema
$r\_fa?: FA\_NOS$

$r\_fa? \in fa$
$\forall x: FA\_NOS \bullet x \notin \mathrm{dom}\ rel\_fa\_ledger$
$fa' = fa \setminus \{r\_fa?\}$
$fa\_blocked' = \{r\_fa?\} \lhd fa\_blocked$
$fa\_descr' = \{r\_fa?\} \lhd fa\_descr$
$fa\_insured' = \{r\_fa?\} \lhd fa\_insured$
$fa\_war\_date' = \{r\_fa?\} \lhd fa\_war\_date$

Fa_Change_descr

ΔFa_Schema
$fa\_descr?: DESCR$
$fa?: FA\_NOS$

$fa\_descr' = fa\_descr \oplus \{(fa? \mapsto fa\_descr?)\}$

ENTRYTYPE ::= *Acquisition* | *Depreciation*
[*DOCNO*]

Fa_Ledger_Schema

ΞRel_Schema
$Posting\_Date: LEDGER\_NOS \rightarrowtail DATE$
$Doc\_No: LEDGER\_NOS \rightarrowtail DOCNO$
$Doc\_Date: LEDGER\_NOS \rightarrowtail DATE$
$Entry\_Descr: LEDGER\_NOS \rightarrowtail DESCR$
$Quantity: LEDGER\_NOS \rightarrowtail \mathbb{N}$
$Amount: LEDGER\_NOS \rightarrowtail \mathbb{Z}$
$EntryType: LEDGER\_NOS \rightarrowtail ENTRYTYPE$

---

dom *Doc_No* = *ledger*
dom *Doc_Date* = *ledger*
dom *Posting_Date* = *ledger*
dom *Entry_Descr* = *ledger*
dom *Quantity* = *ledger*
dom *Amount* = *ledger*

add_acquisition_entry

ΔFa_Ledger_Schema
$a\_ledger?: LEDGER\_NOS$
$a\_fa?: FA\_NOS$
$a\_Posting\_Date?: DATE$
$a\_Doc\_No?: DOCNO$
$a\_Doc\_Date?: DATE$
$a\_Entry\_Descr?: DESCR$
$a\_Quantity?: \mathbb{N}$
$a\_Amount?: \mathbb{Z}$
$a\_EntryType?: ENTRYTYPE$

$\forall x: LEDGER\_NOS \bullet x \neq rel\_fa\_ledger\ a\_fa?$
$ledger' = ledger \cup \{a\_ledger?\}$
$Posting\_Date' = Posting\_Date \cup \{(a\_ledger? \mapsto a\_Posting\_Date?)\}$
$Doc\_No' = Doc\_No \cup \{(a\_ledger? \mapsto a\_Doc\_No?)\}$
$Doc\_Date' = Doc\_Date \cup \{(a\_ledger? \mapsto a\_Doc\_Date?)\}$
$Entry\_Descr' = Entry\_Descr \cup \{(a\_ledger? \mapsto a\_Entry\_Descr?)\}$
$Quantity' = Quantity \cup \{(a\_ledger? \mapsto a\_Quantity?)\}$
$Amount' = Amount \cup \{(a\_ledger? \mapsto a\_Amount?)\}$
$EntryType' = EntryType \cup \{(a\_ledger? \mapsto a\_EntryType?)\}$

add_depr_entry

ΔFa_Ledger_Schema
$a\_ledger?: LEDGER\_NOS$
$a\_fa?: FA\_NOS$
$a\_Posting\_Date?: DATE$
$a\_Doc\_No?: DOCNO$
$a\_Doc\_Date?: DATE$
$a\_Entry\_Descr?: DESCR$
$a\_Quantity?: \mathbb{N}$
$a\_Amount?: \mathbb{Z}$
$a\_EntryType?: ENTRYTYPE$

$\exists x: LEDGER\_NOS \bullet x = rel\_fa\_ledger\ a\_fa?$
$ledger' = ledger \cup \{a\_ledger?\}$
$Posting\_Date' = Posting\_Date \cup \{(a\_ledger? \mapsto a\_Posting\_Date?)\}$
$Doc\_No' = Doc\_No \cup \{(a\_ledger? \mapsto a\_Doc\_No?)\}$
$Doc\_Date' = Doc\_Date \cup \{(a\_ledger? \mapsto a\_Doc\_Date?)\}$
$Entry\_Descr' = Entry\_Descr \cup \{(a\_ledger? \mapsto a\_Entry\_Descr?)\}$
$Quantity' = Quantity \cup \{(a\_ledger? \mapsto a\_Quantity?)\}$
$Amount' = Amount \cup \{(a\_ledger? \mapsto a\_Amount?)\}$
$EntryType' = EntryType \cup \{(a\_ledger? \mapsto a\_EntryType?)\}$

$MESSAGE ::= Acquisition\_Already\_Exists \mid Acquisition\_Must\_Exsist \mid Success$

Acquistion_check

ΞFa_Ledger_Schema
$error!: MESSAGE$
$a\_ledger?: LEDGER\_NOS$
$a\_fa?: FA\_NOS$

$\exists x: LEDGER\_NOS \bullet x = rel\_fa\_ledger\ a\_fa?$
$error! = Acquisition\_Already\_Exists$

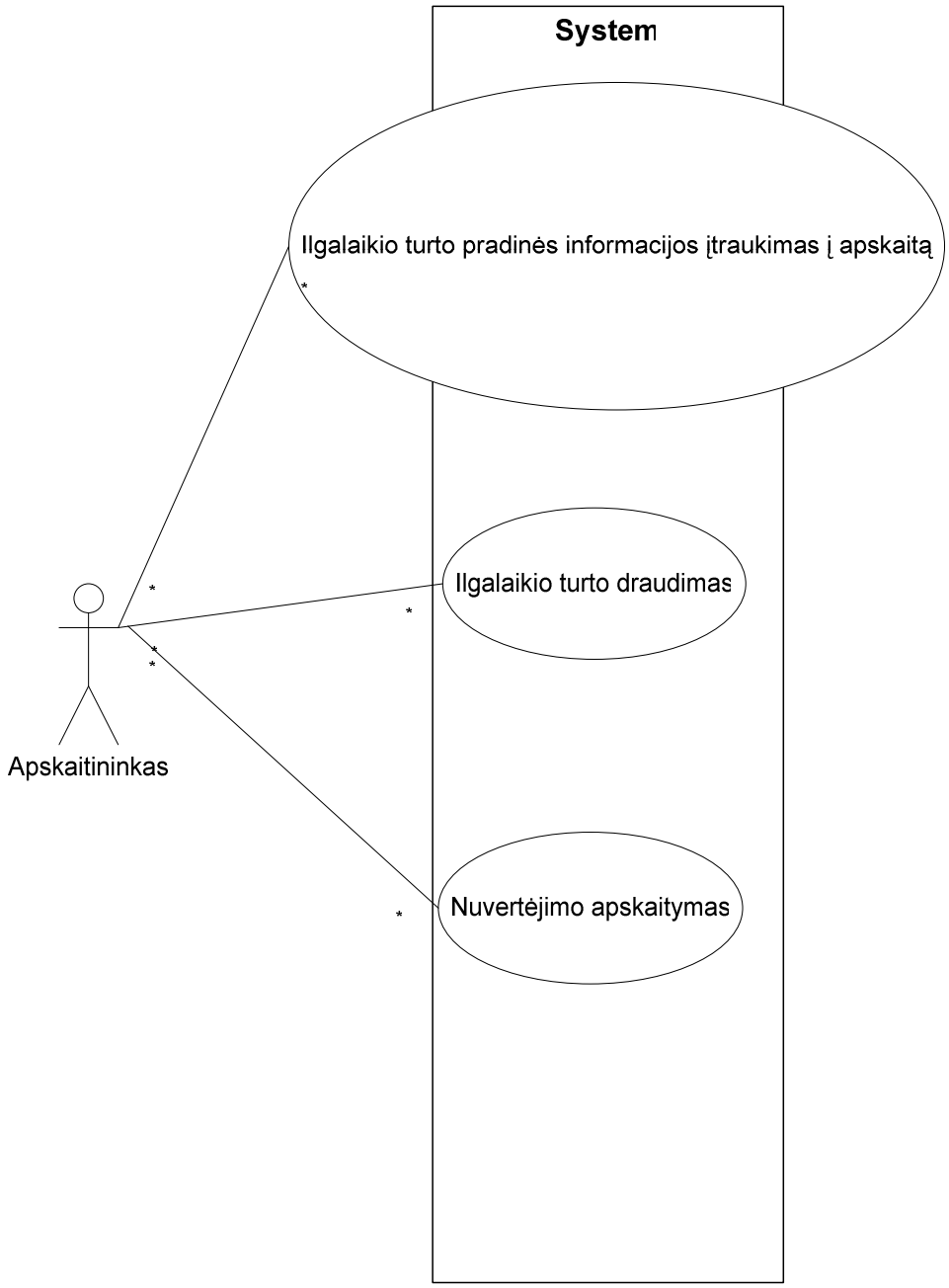
Depreciation_Check

ΞFa_Ledger_Schema
$error!: MESSAGE$
$a\_fa?: FA\_NOS$

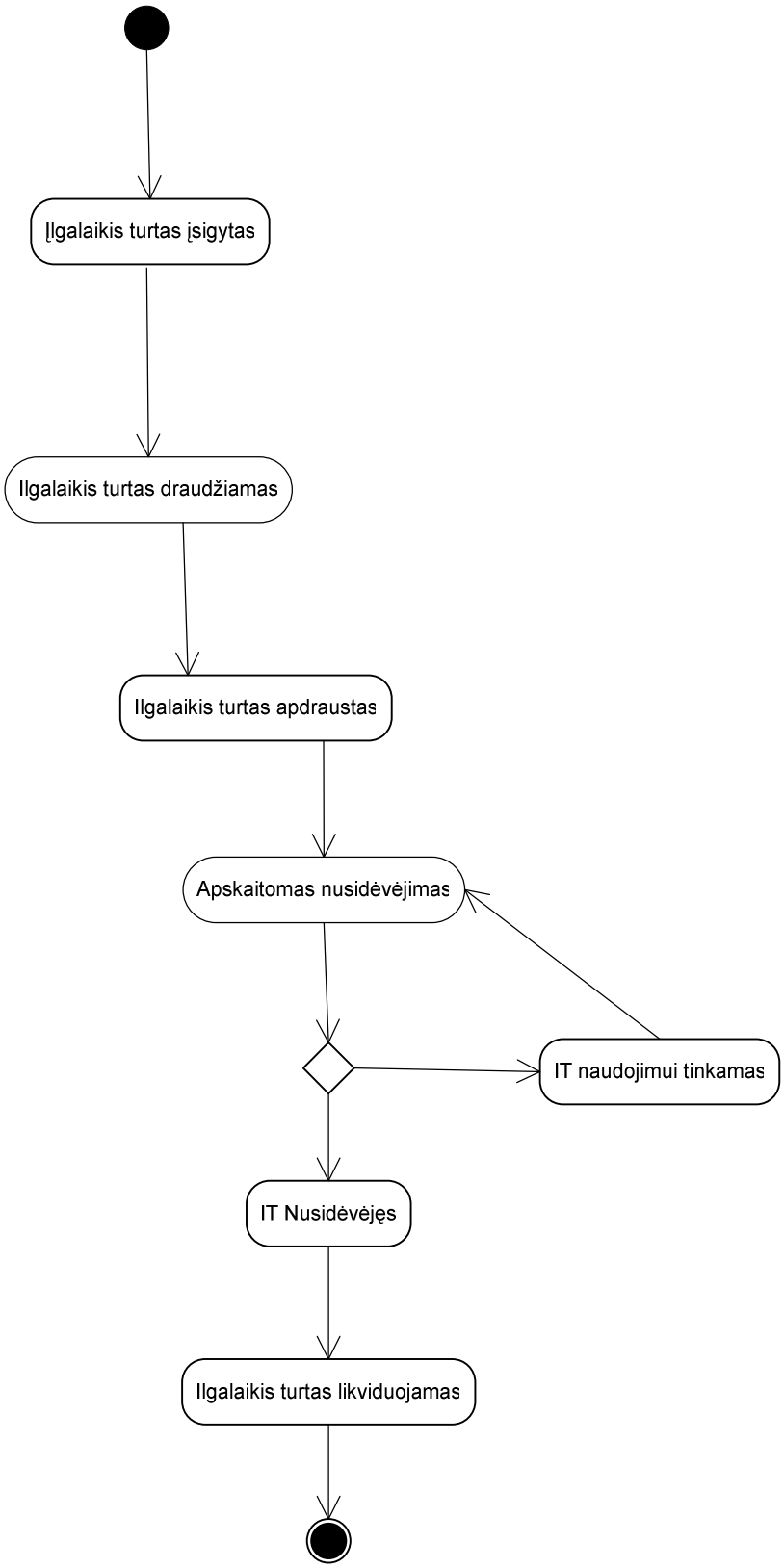
$\forall x: LEDGER\_NOS \bullet x \neq rel\_fa\_ledger\ a\_fa?$
$error! = Acquisition\_Must\_Exsist$

$$InsertEntry \;\widehat{=}\; (add\_depr\_entry \vee Depreciation\_Check) \wedge (add\_acquisition\_entry \vee Acquistion\_check)$$

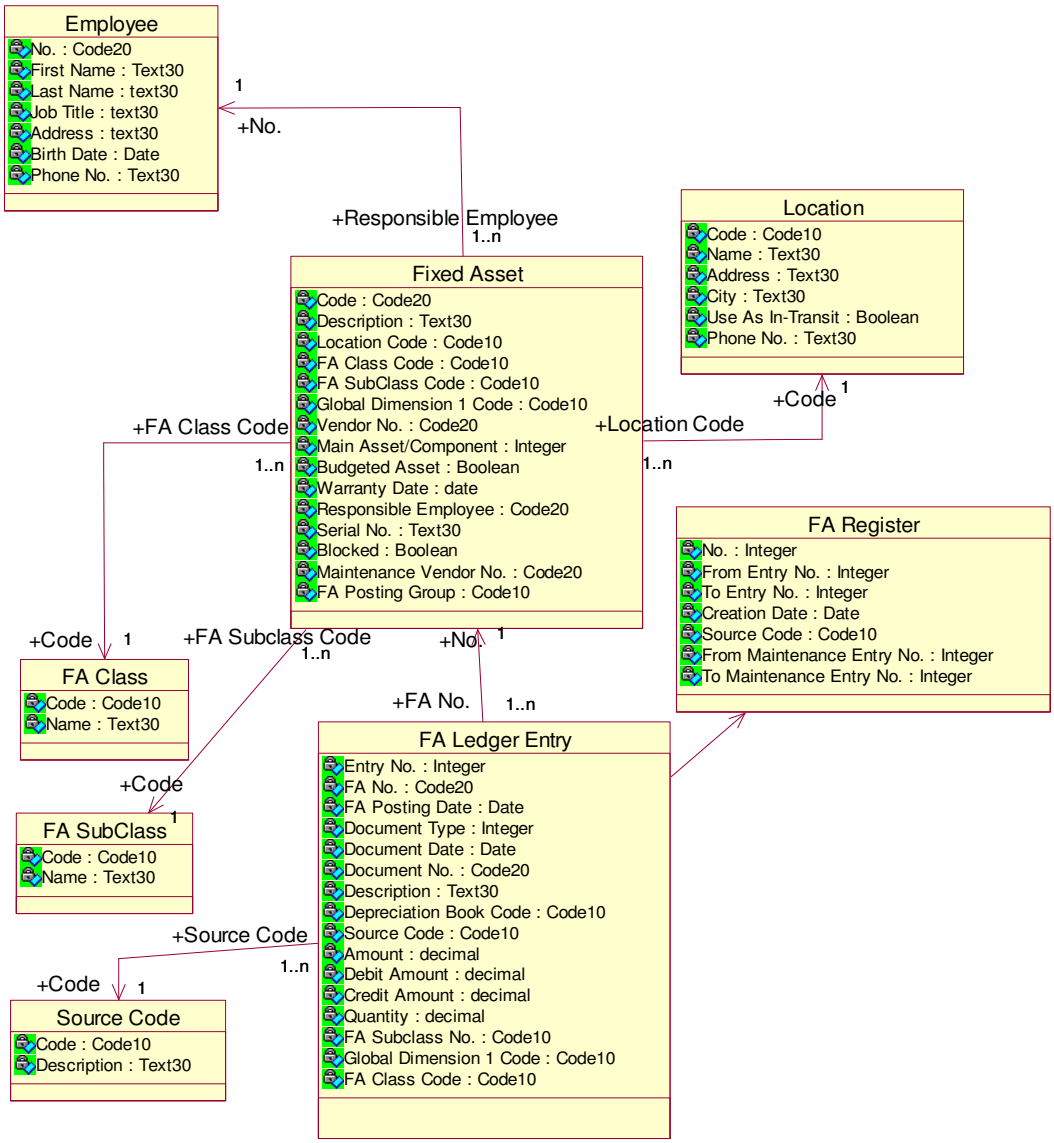
2. PRIEDAS. Ilgalaikio turto UML specifikacijos



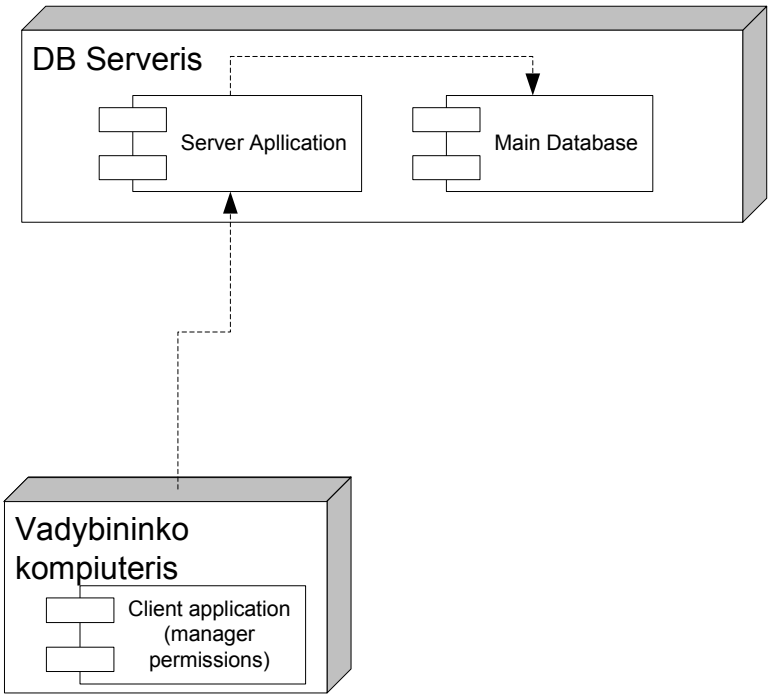
1 pav. Panaudojimo atvejų diagrama



2 pav. Būsenų diagrama



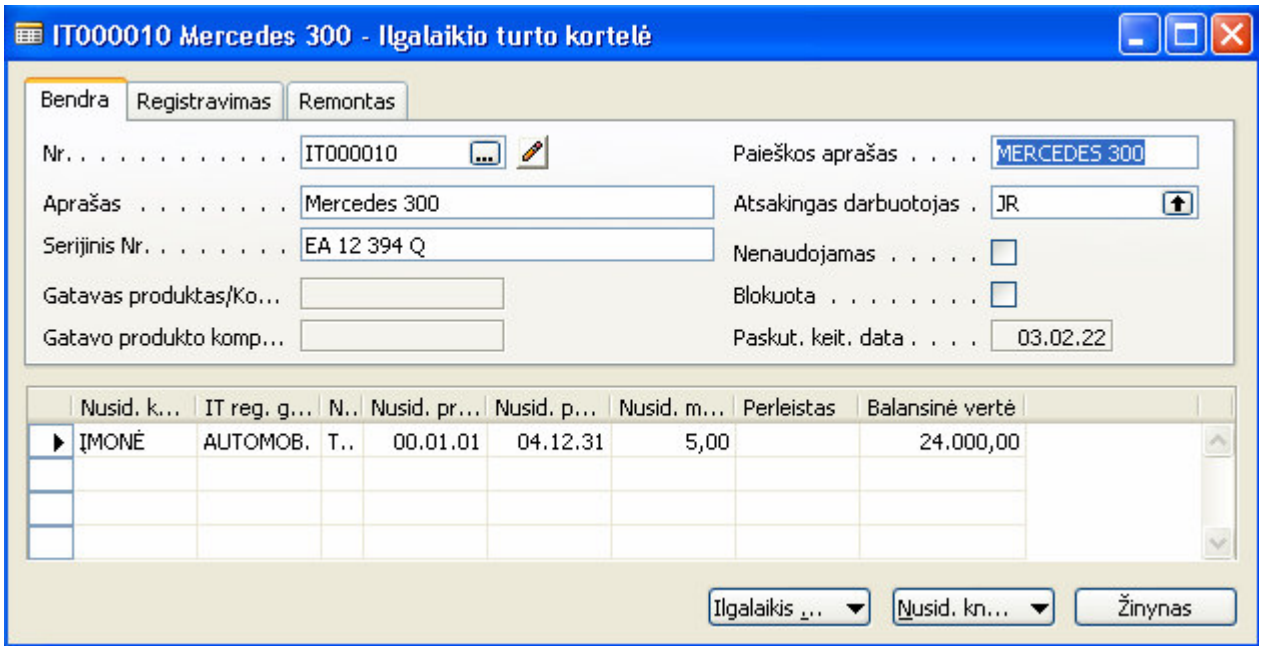
3 pav. Klasių diagrama



4. pav. Sistemos komponentų diagrama

3. PRIEDAS

Ilgalaikio turto sistemos realizacija



1 pav. Ilgalaikio turto kortelė