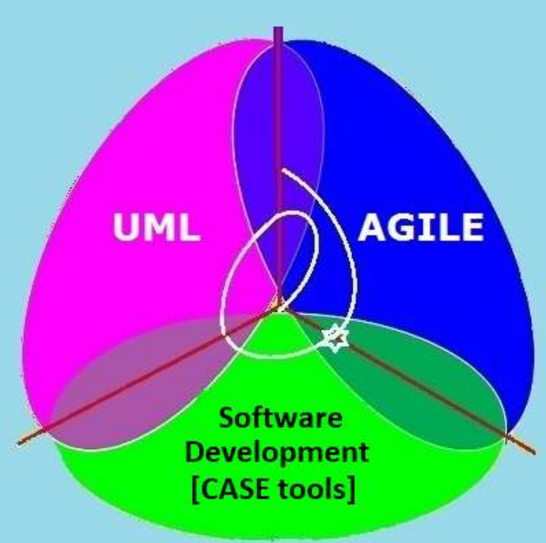
გია სურგულაძე, ლილი პეტრიაშვილი,
მარინე ბიტარაშვილი, ხატია ხატიაშვილი

პროგრამული სისტემების ავტომატიზებული დაპროექტების და ტესტირების ტექნოლოგიები

(CASE, UML, AGILE)



სტუ-ის „IT კონსალტინგის სამეცნიერო ცენტრი“

გია სურგულაძე, ლილი პეტრიაშვილი,
მარინე ბიტარაშვილი, ხატია ხატიაშვილი

პროგრამული სისტემების ავტომატიზებული დაპროექტების და ტესტირების ტექნოლოგიები (CASE, UML, Agile)

დამტკიცებულია:

სტუ-ს „IT კონსალტინგის სამეცნიერო ცენტრის" სარედაქციო კოლეგიის მიერ, 2024 წ. 25 აპრილი, ოქმი N7

თბილისი- 2024

უაკ 002.5

განხილულია გამოყენებითი პროგრამული სისტემების ავტომატიზებული დაპროექ-ტების და დაპროგრამების მეთოდები და თანამედროვე ინსტრუმენტული საშუალებები (CASE-ტექნოლოგიები), ობიექტ- და პროცეს-ორიენტირებული მიდგომებით და სერვის-ორიენტირებული არქიტექტურით. წარმოდგენილია ბიზნესპროცესების მოდელირების ნოტაციის (BPMN), კონცეპტუალური სქემის ავტომატიზებული დაპროექტების ობიექტ-როლური მოდელირების (ORM), გამოყენებითი პროგრამული აპლიკაციების შექმნის უნიფიცირებული მოდელირების ენის (UML) და მოქნილი დეველოპმენტის (Agile) მეთოდოლოგიები. ასახულია როგორც UML/Agile მეთოდო-ლოგიების მეთოდები: ექსტრემალური პროგრამირების, Scrum და Kanban/Lean მაგალითებზე, ასევე C# კოდის ავტომატიზებული გენერაციის საშუალებები. ნაშრომის ორიგინალური გადაწყვეტები რეალიზებულია პროგრამულად სხვადასხვა პრობლე-მური სფეროს მაგალითებზე. განსაკუთრებით გამოკვეთილია პროგრამული სისტემების სასიცოცხლო ციკლის ანალიზის, პროექტირების, პროგრამული დეველოპმენტის და ტესტირების ეტაპები. *მონოგრაფია* შეიძლება გამოყენებულ იქნას დამხმარე სახელ-მძღვანელოდ კომპიუტერული მეცნიერების და პროგრამული ინჟინერიის სპეციალობის სტუდენტებისა და ამ საკითხებით დაინტერესებული მკითხველისათვის.

რეცენზენტები:

- პროფ. თეიმურაზ სუხიაშვილი (სტუ, ტექნიკის მეცნიერებათა კანდიდატი),
- ეკატერინე თურქია (ტექნიკის მეცნიერებათა კანდიდატი ინფორმატიკაში, საქართველოს ეროვნული ბანკის განყოფილების გამგე)

რედკოლეგია:

ა. ფრანგიშვილი (თავმჯდომარე), მ. ახობაძე, ზ. ბოსიკაშვილი, ზ. გასიტაშვილი, გ. გოგიჩაიშვილი, მ. თევდორაძე, ე. თურქია, თ. კაიშაური, რ. კაკუბავა, დ. კაპანაძე, თ. ლომინაძე, ნ. ლომინაძე, თ. ჟვანია, ლ. პეტრიაშვილი, გ. სურგულაძე (რეადაქტორი), ი. ქართველიშვილი, ო. შონია, ა. ცინცაძე, ზ. წვერაიძე

© სტუ-ს „IT-კონსალტინგის სამეცნიერო ცენტრი", 2024

ISBN 978-9941-8-6334-9

ყველა უფლება დაცულია, ამ წიგნის არც ერთი ნაწილის (იქნება ეს ტექსტი, ფოტო, ილუსტრაცია თუ სხვა) გამოყენება არანაირი ფორმითა და საშუალებით (იქნება ეს ელექ¬ტრონული თუ მექანიკური), არ შეიძლება გამომცემლის წერილობითი ნებართვის გარეშე. საავტორო უფლებების დარღვევა ისჯება კანონით.

Georgian Technical University

Gia Surguladze, Lily Petriashvili,
Marine Bitarashvili, Khatia Khatiashvili

SOFTWARE DEVELOPMENT AND TESTING TECHNOLOGIES (CASE, UML, AGILE)

Methods of automated design and programming of applied software systems and modern tools (CASE-technologies) with object- and process-oriented approaches and service-oriented architecture are discussed. Business process modeling notation (BPMN), Unified modeling language (UML) and Agile Development and Testing methodologies for creating applied software are presented. The methods of UML/Agile methodologies: extreme programming, Scrum and Kanban/Lean examples, as well as C# (or other) automated code generation tools are described. The original solutions of the paper are implemented programmatically on examples of different problem areas. The stages of software systems life cycle analysis, design, software development and testing are especially highlighted. The monograph can be used as a guide for software engineering majors in informatics and readers interested in these issues.

© „IT-Consulting scientific center" of GTU, Tbilisi, 2024
ISBN 978-9941-8-6334-9

ავტორთა შესახებ:

გია სურგულაძე – ტექნიკის მეცნიერებათა დოქტორი, გაეროსთან არსებული „ინფორმატიზაციის საერთაშორისო აკადემიის“ ნამდვილი წევრი (1994 წლიდან, IIA), საქ. ტექნიკური უნივერსიტეტის პროფესორი. ინფორმატიკის ფაკულტეტის „პროგრამული ინჟინერიის“ აკადემიური დეპარტამენტის უფროსი. 100 წიგნის ავტორი (მათ შორის 27 მონოგრაფია, 20 სახელმძღვანელო, 53 დამხმარე და მეთოდური სახელმძღვანელო), 300-ზე მეტი სამეცნიერო ნაშრომის ავტორი მართვის საინფორმაციო სისტემების პროგრამული ინჟინერიის, მონაცემთა საცავების, დაპროგრამების ჰიბრიდული და მობილური ტექნოლოგიების, ექსპერტული სისტემებისა და ვირტუალური რეალობის, იმიტაციური მოდელირების, ინფორმატიკის დიდაქტიკის და სხვ. სფეროებში. სტუ-ის „ინფორმატიკის“ სადოქტორო პროგრამის ხელმძღვანელი. .მსოფლიო ბანკის, USAID-ის და NATO-ს პროექტების მონაწილე. *წვლილი:* წინამდებარე ნაშრომში შეიმუშავა მართვის საინფორმაციო სისტემების განვითარების მეთოდოლოგია საპრობლემო სფეროს ბიზნესპროცესების ობიექტ-როლური და უნიფიცირებული მოდელების აგების ავტომატიზაციის საფუძველზე.

ლილი პეტრიაშვილი – სტუ-ის ინფორმატიკისა და მართვის სისტემების ფაკულტეტის დეკანის მოადგილე სამეცნიერო დარგში, საქართველოს განათლების მეცნიერებათა აკადემიის წევრი, დოქტორანტურის საგანმანათლებლო პროგრამის „ინფორმატიკა" და საბაკალავრო პროგრამის „კომპიუტერული მეცნიერება" ხელმძღვანელი. ტექნიკის და ტექნოლოგიის მსოფლიო აკადემიის სამეცნიერო სარედაქციო კომიტეტის წევრი. DAAD-ის ელჩი – სტუ-ში. Volkswagen ის ფონდისა და DAAD-ის მრავალგზის სტიპენდიანტი. აქვს 100-ზე მეტი სამეცნიერო ნაშრომი, მათ შორის 5 მონოგრაფია და 23 სახელმძღვანელო. მისი ხელმძღვანელობით დაცულია 10 სადოქტორო დისერტაცია. *წვლილი:* შეიმუშავა მონაცემთა საცავების დაპროექტების და ბიზნეს-ანალიტიკის მეთოდები, რიგების თეორიის და NoSQL ბაზების საფუძველზე.

მარინე ბიტარაშვილი – ინფორმატიკის აკად. დოქტორი (2013). სტუ-ის ბაკალავრი და მაგისტრი, სტუ-ს „ინფორმატიკის“ პროგრამის დოქტორანტი (2010-2013) /მეცნ. ხელმძღ, გ. სურგულაძე). დისერტაციის თემა „პროგრამული სისტემების მენეჯმენტის ბიზნესპროცესების კვლევა Agile მოდელირების მეთოდების და ინსტრუმენტების საფუძველზე“, არის 1 მონოგრაფიის, 2 დამხმარე სახელმძღვანელოს, 12 სამეცნიერო სტატიის ავტორი, პროგრამისტ-დეველოპერი. *წვლილი:* შეასრულა საგადასახადო სფეროს შემოსავლების სამსახურის დავების განხილვის სისტემის ბიზნესპროცესების პროგრამული აპლიკაციის დეველოპმენტი.

ხატია ხატიაშვილი – ინფორმატიკის აკად. დოქტორი (2023). სტუ-ის „ინფორმატიკის“ პროგრამის დოქტორანტი (2020-2023) /მეცნ. ხელმძღ, გ. სურგულაძე), დისერტაციის თემა „Agile დეველოპმენტი და ტესტირების მეთოდების შემუშავება და კვლევა მართვის საინფორმაციო სისტემის ასაგებად“. არის 1 მონოგრაფიის და 5 სამეცნიერო სტატიის ავტორი. *წვლილი:* შეასრულა საინფორმაციო სისტემის დაპროექტება სუფთა არქიტექტურის მეთოდით, Agile Development-ის Scrum ფრეიმვორკის და Agile Testing-ის საფუძველზე. არის საქართველოს ფინანსებისა და ჯანმრთელობის დაცვის საინფორმაციო პროექტების მონაწილე – პროგრამისტ-დეველოპერი.

შინაარსი

შესავალი …………............................………………………………………………........ 9
ნაწილი I. უნიფიცირებული და მოქნილი მეთოდოლოგიების საფუძვლები 13
თავი 1. ობიექტორიენტირებული პროგრამირების მეთოდი ………………………… 13
1.1. ობიექტ-ორიენტირებული დაპროგრამების არსი …….. 13
1.2. ობიექტები და კლასები. მონაცემთა აბსტრაქტული ტიპები …………………….. 16
1.3. კლასების იერარქია, მემკვიდრეობითობა ……….……......................................., 19
1.4. პოლიმორფიზმი ………………………………………….. 20
1.5. ობიექტ-ორიენტირებული დიაგრამების აგება სისტემების დაპროექტების ეტაპზე ……………….. 21
თავი 2. ბიზნეს-პროცესების მოდელირების CASE ინსტრუმენტები და მეთოდოლოგიები 30
2.1. ბიზნეს-პროცესებზე ორიენტირებული მოდელირების ენა (BPMN) 31
2.2. ობიექტ-ორიენტირებული, უნიფიცირებული მოდელირების ენა (UML) 37
2.2.1. UML-ის კლასიკური დიაგრამები 40
2.2.2. კლასთაშორის კავშირების სისტემა 43
2.2.3. რევერსული ტექნოლოგია: „კლასი-კოდი-კლასი“ 44
თავი 3. მოქნილი პროგრამირების მეთოდოლოგია და მეთოდები 52
3.1. დაპროგრამების Agile მეთოდოლოგია და აპლიკაციების დეველოპმენტის მეთოდები ....……..... 52
3.1.1. ექსტრემალური დაპროგრამების პრინციპები და ინსტრუმენტული საშუალებანი ....….. 52
3.1.2. პროგრამების მოქნილი დეველოპმენტის მანიფესტი და პრინციპები 53
3.1.3. მოქნილი (სწრაფი) მოდელირება (Agile Modeling) 55
3.1.4. მოქნილი მოდელირების ფასეულობანი 56
3.1.5. Scrum - მოქნილი მეთოდის ფრეიმვორკი ....…..... 61
3.1.6. Kanban - ეკონომიური მოქნილი მეთოდი ……….…... 64
3.1.7. Scrum და Kanban მეთოდების შედარება 69
3.2. UML და Agile მეთოდოლოგიების გამოყენების კომპრომისული გადაწყვეტა .. 70
3.3. Agile და DevOps მეთოდოლოგიები და მათი ინსტრუმენტული საშუალებები.. 73
- პირველი ნაწილის დასკვნა 75
ნაწილი II. ბიზნესპროცესების მოდელირების ტექნოლოგიები 77
თავი 4. კორპორაცული მენეჯმენტის სისტემების ბიზნეს–პროცესების მოდელირების პრობლემები და ახალი ინფორმაცული ტექნოლოგები 77
4.1. კორპორაციული მენეჯმენტის ბიზნესპროცესები - მართვის რთული და განაწილებული სისტემა ....…...... 77
4.2. საინფორმაციო სისტემების დაპროექტების და მოდელირების თანამედროვე

მეთოდოლოგია ………........................……….……….................................... 79
4.2.1. UML/2 ტექნოლოგიის არსი და მისი გამოყენება …... 79
4.2.2. UML/2-ის ახალი დიაგრამები 81
4.3. საინფორმაციო სისტემების პროგრამული უზრუნველყოფის აგების 100
თანამედროვე ინსტრუმენტული საშუალებები 111
4.3.1. Ms Visio, Ratonal Rose და სხვა პაკეტები 111
4.3.2. Enterprise Architect და Visual Studio.NET ………………………….................... 114
4.3.3. ბიზნესპროცესების დაპროგრამების ახალი ტექნოლოგია Workflow
Foundation in Visual Studio.NET 115
4.4. საინფორმაციო სისტემების ინფორმაციული უზრუნველყოფა Visual Studio
.NET გარემოში 116
4.4.1. MsSQL Server პაკეტი 117
4.4.2. XML მონაცემთა ბაზები 118
4.5. მათემატიკური მოდელირება ორგანიზაციული მართვის საინფორმაცო
სისტემებში 120
4.5.1. რიგების თეორია 120
4.5.2. WinPepsy ინსტრუმენტული საშუალება 122
**თავი 5. კორპორაციული მენეჯმენტის ბიზნესპროცესების საინფორმაციო
სისტემის ვიზუალური და მათემატიკური მოდელირება 123**
5.1. კორპორაციული მენეჯმენტის ბიზნესპროცესების მოდელირება
UML/2 ტექნოლოგიით 123
5.1.1. საპრობლემო სფეროს ბიზნესპროცესების და ბიზნესწესების აღწერა 123
5.1.2. საპრობლემო სფეროს Use Case და Activity დიაგრამები 127
5.1.3. საპრობლემო სფეროს ფუნქციური ამოცანის კომუნიკაციის დიაგრამა 130
5.1.4. საგადასახადო დავების გადაწყვეტის მხარდამჭერი სისტემის
UML/2 დიაგრამები 131
5.1.5. საგადასახადო დავების წარმოების Sequence და Timing დიაგრამები 137
5.1.6. საგადასახადო დავების წარმოების Information_Flow და Class დიაგრამები 139
5.2. საგადასახადო დავის წარმოების ბიზნესპროცესის მათემატიკური მოდელი
რიგების თეორიის მიხედვით 140
5.2.1. რიგების სახეები მასობრივი მომსახურების სისტემებში 140
5.2.2. დავების განხილვის ბიზნესპროცესების მასობრივი მომსახურების
მოდელები 149
5.2.2.1. ერთარხიანი სისტემის მოდელი M/M/1 151
5.2.2.2. მრავალარხიანი სისტემის მოდელი M/M/m 152
5.4. ორგანიზაციულ სისტემდათა ქსელების მათემატიკური მოდელების
გამოკვლევა WinPepsy პროგრამული პაკეტის ბაზაზე 153

5.4.1. „კლიენტ-სერვერ“ ჩაკეტილი ქსელის მოდელირება და ანალიზი 156
5.4.2. „კლიენტ-სერვერ“ ღია ქსელის მოდელირება და ანალიზი 160
5.4.3. ჰიბრიდული ქსელის მოდელირება და ანალიზი 163
5.4.4. WinPepsy - დავების გადაწყვეტის სისტემისთვის 165
5.5. საპრობლემო სფეროს ობიექტ-როლური კონცეპტუალური სქემის ავტომატიზებული დაპროექტება CASE NORMA ინსტრუმენტით 166
თავი 6. საგადასახადო სფეროს ბიზნესპროცესების მოდელირება და პროგრამული რეალიზაცია ახალი ინსტრუმენტული საშუალებებით 171
6.1. ბიზნეს-პროცესების მოდელირება UML/2 ტექნოლოგიის Enterprise Architect ინსტრუმენტით 171
6.1.1. საგადასახადო დავების წარმოების ბიზნესპროცესების მოდელირება 175
6.1.2. კლასების დიაგრამიდან პროგრამული კოდის გენერირება 178
6.2. პროექტების აგება Workflow Foundation ტექნოლოგიით .NET პლატფორმაზე .. 185
6.3. საგადასახადო დავების განხილვის სისტემის პროგრამული რეალიზაცია Workflow Foundation ტექნოლოგიით 191
6.4. მონაცემთა ბაზის ფიზიკური სტრუქტურის ავტომატიზებული აგება DDL ფაილებით Ms SQL Server - პაკეტისთვის 197
6.5. სისტემის ინფორმაცული ბაზა 199
6.6. მომხმარებელთა ინტერფეისების პროგრამული რეალიზაცია 202
6.7. ინფორმაციის კომბინირებული დამუშავების მეთოდი 210
- მეორე ნაწილის დასკვნა 212
ნაწილი III. Agile Software Development და Agile Testing 213
თავი 7. პროგრამული სისტემების აგება Agile Development და Agile Testing მეთოდოლოგიის საფუძველზე 216
7.1 მართვის საინფორმაციო სისტემები (ზოგადი მიმოხილვა) 216
7.2. Agile მეთოდოლოგია vs „ჩანჩქერის“ მოდელი 218
7.3. Agile მეთოდოლოგიის Scrum მეთოდი 222
7.3.1. Scrum: როლები და ფუნქციები 222
7.3.2. სპრინტი 225
7.3.3. Scrum ფრეიმვორკის დადებითი და უარყოფითი მხარეები 226
7.4. Agile ტესტირება 228
7.5. QA გამოწვევები Agile პროგრამული უზრუნველყოფის განვითარების პროცესში 231
7.5.1. ავტომატიზაციის რისკი Agile პროცესში 231
7.5.2. Agile მეთოდოლოგია პროგრამული უზრუნველყოფის ტესტირებაში 233
7.5.3. ტესტირების გეგმა Agile QA-სთვის 235

თავი 8. ობიექტის კვლევის ამოცანა და ტექნოლოგიები 239
8.1. ამოცანის დასმა 240
8.2. პროექტის მოდელი 243
8.2.1. სუფთა არქიტექტურის დაპროექტყების მეთოდი 244
8.2.2. CQRS დიზაინ პატერნი 248
8.2.3. მედიატორ დიზაინ პატერნი 249
8.2.4. UML დიაგრამა 250
თავი 9. პროექტის რეალიზაცია 251
9.1. თანამშრომლის მოდელი პროექტში - Domain ფენა (დასატესტი პროგრამული კოდი) 251
9.2. პროექტის მომზადება ტესტირებისთვის 266
9.4. Agile ტესტირება პროექტის ფარგლებში 267
9.4.1 მოდულური ტესტები 267
9.4.2 მუტაციური ტესტები 280
9.4. მუტაციური ტესტირების ავტომატიზაცია 282
9.5. მოდულური და მუტაციის შედეგების შედარება და რეკომენდაციები 290
- მესამე ნქწილის დასკვნა 292
- გამოყენებული ლიტერატურა 294

შესავალი

პროგრამული ინჟინერია – ინფორმატიკის (კომპიუტინგის) მეცნიერების მიმართულებაა, რომელიც შეისწავლის სისტემური და გამოყენებითი პროგრამული პროდუქტების (აპლიკაციების) დაპროექტების (დიზაინის), განვითარების (დეველოპმენტის), ტესტირების, დანერგვის (იმპლემენტაციის) და მხარდაჭერის (თანხლებისა და გაფართოების) საკითხებს [1].

პროგრამული აპლიკაციების (უზრუნველყოფის) ინჟინრები იყენებენ საინჟინრო პრინციპებს, პროგრამირების ენების ცოდნას, დაპროგრამების მეთოდოლოგიებს და მეთოდებს – საბოლოო მომხმარებლებისთვის (პროდუქ-ტის დამკვეთისათვის) პროგრამული გადაწყვეტილებების (ვალიდური აპლიკაციების) შესაქმნელად [2,3].

XXI საუკუნიდან *პროგრამული ინჟინერიის* მეცნიერული მიმართულება და მისი საუკეთესო პრაქტიკები მნიშვნელოვნად განვითარდა და კვლავაც განიცდის უწყვეტ სრულყოფას. 2020 წლისათვის მკვეთრად გაიზარდა უახლესი კომპიუტერული და მობილური ტექნიკისა და ინფორმაციული ტექნოლო-გიების გამოყენება სოციალური, ეკონომიკური, სამრეწველო, საინჟინრო და სხვა სფეროებში, მათი შესაბამისი ახალი თაობის პროგრამული აპლიკაციების შექმნისა და დანერგვის მიზნით [4-8].

ახალი პარადიგმების საფუძველზე პროგრამული ინდუსტრიის განვითარებას, მათი მეთოდებისა და მეთოდოლოგიების შექმნას ან არსებულის სრულყოფას დიდი ყურადღება ექცევა მსოფლიო ბაზრის ლიდერების მხრიდან, როგორებიცაა Microsoft, Microsystem, HP, Oracle და მრავალი სხვ.

საერთაშორისო საუნივერსიტეტო საგანმანათლებლო პროგრამები კომპი-უტინგის და კომპიუტერული მეცნიერების მიმართულებით მნიშვნელოვნად შეიცვალა ბოლო ათწლეულში ახლი კონცეფციების, მეთოდოლოგიებისა და ტექნოლოგიების მიმართულებით. განსაკუთრებით მნიშვნელოვანია აქ გამოყენებითი პროგრამული ინჟინერიის (Applied Software Engineering) სფერო, რომლის ძირითადი მიზანი მომხმარებლებზე ორიენტირებული კომპიუ-ტერული სისტემების დიზაინი და დეველოპმენტია, უახლესი ჰიბრიდული და მობილური პროგრამირების ინტეგრაციის საფუძველზე. მათი ხარისხის მართვის და მწარმოებლურობის ამაღლების თვალსაზრისით მნიშვნელოვანი ფუნქცია დაეკისრა დაპროგრამების ავტომატიზაციის (CASE – Computer-Aided Software Engineering) და სწრაფი დეველოპმენტის (Agile) მეთოდოლოგიებს [1,3].

წიგნის პირველ ნაწილში გადმოცემულია კომპიუტერული პროგრამირების კონცეფცია და ძირითადი ცნებები, მეთოდებისა და მეთოდოლოგიების, მათი სტილებისა და ენების კლასიკური, ტრადიციული მიღწევები და პერსპექტივები [2]. წარმოდეგენილია საპრობლემო სფეროს კონცეპტუალური მოდელების დაპროექტების (ORM) და აპლიკაციების დაპროგრამების პროცესების ავტომატიზაციის საკითხები, როგორც CASE ტექნოლოგიის ნიმუშები [25,26].

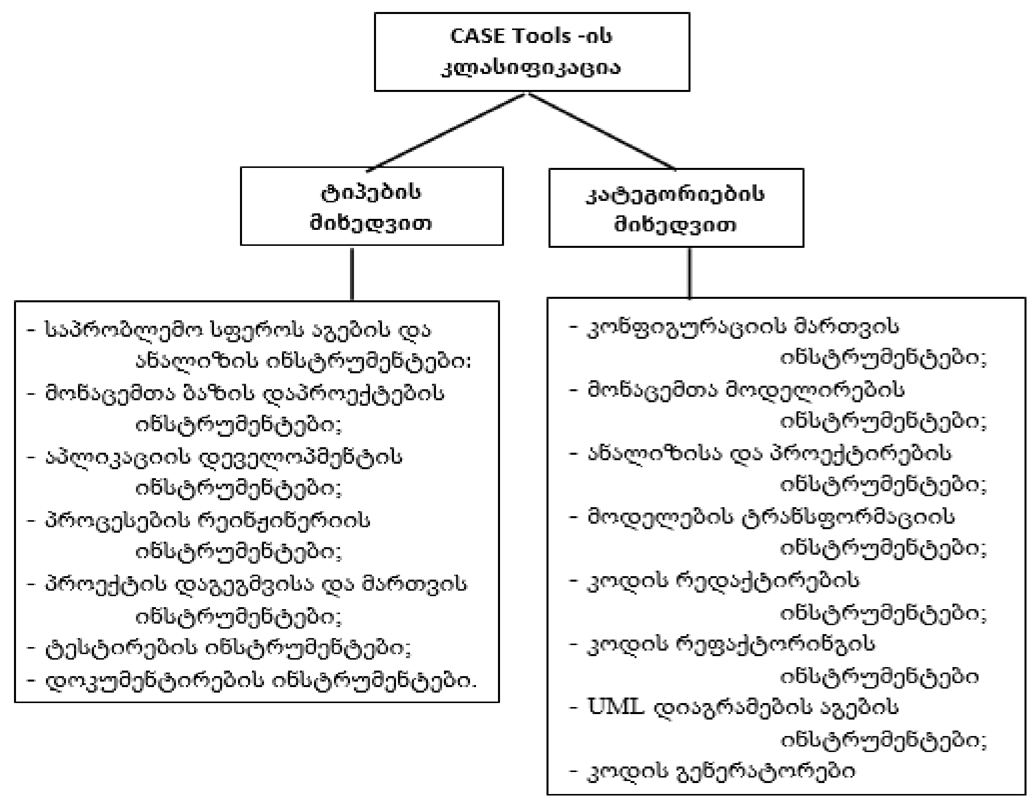
პროგრამული აპლიკაციის დამუშავების მეთოდოლოგია, ზოგადად, არის დეველოპმენტის სამუშაოს დაყოფის პროცესი ცალკეულ ფაზებად – დიზაინის, პროდუქტებისა და პროექტების მართვის სრულყოფის მიზნით. იგი ცნობილია აგრეთვე როგორც პროგრამების დეველოპმენტის სასიცოცხლო ციკლი (Software Development Life Cycle - SDLC) [9].

პროგრამული დეველპომენტის პროცესების მეთოდოლოგიებია, მაგალითად, ჩანჩქერის (Waterfall), იტერაციულ-ინკრემენტალური (Iterative and Incremental), სპირალური (Spiral), აპლკაციების სწრაფი დამუშავების (Rapid) და სხვ. 2000 წლიდან მნიშვნელოვანი ყურადღება მიექცა და განვითარდა უნიფიცირებული მოდელირების ენის (UML) და მოქნილი (Agile) პროგრამირების მეთოდოლოგიები. განსაკუთრებით გამახვილებულია ყურადღება ექსტრემალური პროგრამირების, Scrum და Kanban მეთოდებზე. ეს საკითხები *წიგნის მეორე და მესამე* ნაწილებშია გადმოცემული.

პროგრამული აპლიკაციების დეველოპმენტის ავტომატიზაციის ხელსაწყოები (CASE tools) არის ინსტრუმენტები პროგრამული უზრუნველყოფის დიზაინისა და განვითარების პროცესების რობოტიზაციისთვის (RPA - Robotic Process Automation) [10,11]. ისინი გამიზნულია შრომატევადი პროცესების გასამარტივებლად, პირველ რიგში, სისტემების ანალიტიკოსების, პროგრამული უზრუნველყოფის დიზაინერებისა და პროგრამისტ-დეველოპერებისთვის. შემდეგ კი, ISO/IEC 14102 სტანდარტის მიღებასთან ერთად, CASE-tools განისაზღვრა როგორც პროგრამული აპლიკაციების შექმნის ინსტრუმენტები, მათი სასიცოცხლო ციკლის პროცესების მხარდასაჭერად [12].

CASE-ინსტრუმენტები, კლასიფიკაციის თვალსაზრისით იყოფა ტიპებისა და კატეგორიების მიხედვით (ნახ.1).

– *ტიპების მიხედვით კლასიფიკაცია* ასახავს ინსტრუმენტთა ფუნქციონალურ ორიენტაციას პროგრამული აპლიკაციის შექმნის სასიცოცხლო ციკლის სხვადასხვა პროცესის შესაბამისად;



ნახ. 1. CASE ინსტრუმენტების კლასიფიკაცია

– *კატეგორიების მიხედვით კლასიფიკაცია* განსაზღვრავს ინტეგრაციის ხარისხს შესრულებული ფუნქციების მიხედვით. იგი მოიცავს: ლოკალურ საშუალებებს ცალკეული ავტონომიური ამოცანების გადასაწყვეტად; ნაწილობრივ ინტეგრირებულ საშუალებებს სასიცოცხლო ციკლის უმრავლესი ეტაპებისთვის და სრულად ინტეგრირებულ საშუალებებს - ინფორმაციული სისტემების აგების მთლიანი სასიცოცხლო ციკლისა და მონაცემთა საცავისათვის.

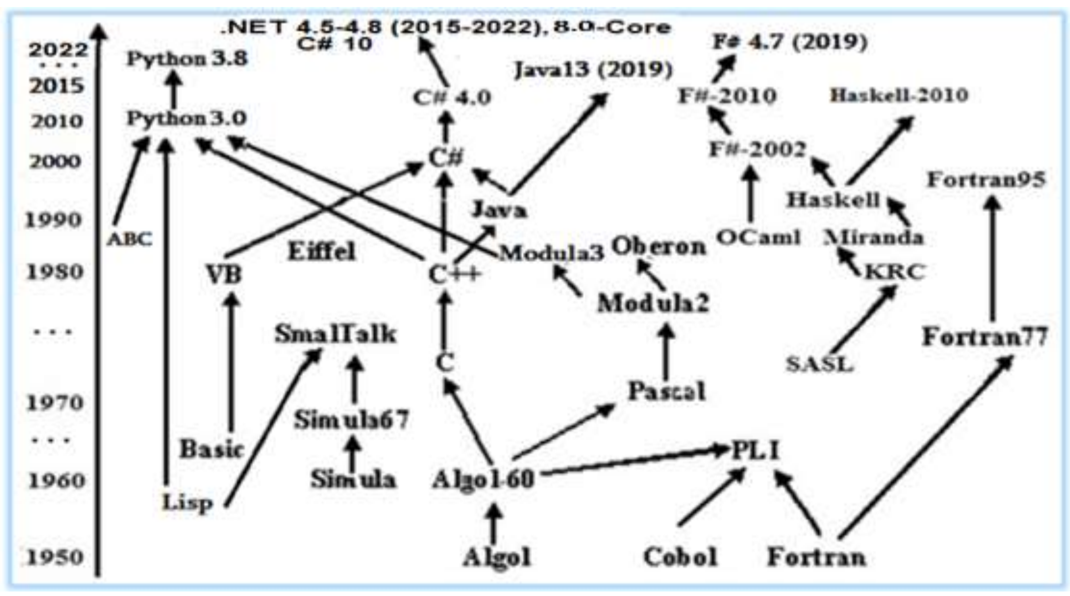
CASE ტექნოლოგიები განსაკუთრებით გამოყენებადია იარპი-სისტემების (ERP – Enterprise Resource Planning) ასაგებად, სადაც ისინი ერთდროულად ასრულებს ბიზნესპროცესების და ინფორმაციის ნაკადების ანალიზის, მოდელირებისა და ოპტიმიზაციის ინსტრუმენტთა ფუნქციებს.

საინჟინრო მეთოდლოგიებზე და ჩანჩქერულ მოდელზე გადასვლა ნამდვილად იყო წინგადადგმული ნაბიჯი. მან შემოიტანა განსაზღვრული მოწესრიგება და ორგანიზებულობა დამუშავების პროცესში. მაგრამ საინჟინრო

მეთოდოლოგიებმა, საბოლოოდ მაინც ვერ შეძლეს პროგრამული პროექტების ყველა პრობლემის გადაწყვეტა [1].

XX და XXI საუკუნეების მიჯნაზე შეიქმნა ახალი ალტერნატიული მიდგომები, რომელთა შორის განსაკუთრებით სწრაფად განვითარდა და დიდი პოპულარობა მოიპოვა მსუბუქმა (ან მსუბუქი წონის) მეთოდებმა (ავტორები, მ. ფოულერი, კ. ბეკი, ა. კოუბერნი და სხვ. ამ მიმართულებას Agile (სწრაფი, მოქნილი) უწოდეს, რომლის მეთოდებიცაა ექსტრემალური პროგრამირება, Scrum, Kanban და სხვ. [2,20]. დღეისათვის Agile Software Development და Agile Testing - ყველაზე აქტუალური და ეფექტური მიმართულებაა პროგრამულ ინჟინერიაში, რაც აშკარად ჩანს ინტერნეტული წყაროებიდან და საერთაშორისო მაღალრეიტინგული საუნივერსიტეტო პროგრამებიდან.

შესავლის ბოლოს წარმოგიდგენთ დაპროგრამების კლასიკური ენების განვითარების დღევანდელი მდგომარეობის ფრაგმენტს (ნახ.2). წიგნში მრავლადაა ასახული ჩვენი პროექტების საილუსტერაციო მაგალითები.



ნახ.2. დაპროგრამების კლასიკური ენების განვითარების ისტორია

წიგნში წარმოდგენილი ორიგინალური შედეგები მიღებულია ავტორთა უშუალო მონაწილეობით სტუ-ის მართვის ავტომატიზებული სისტემების (პროგრამული ინჟინერიის) დეპარტამენტში, აგრეთვე გერმანელ კოლეგებთან თანამშრომლობით (ბერლინის ჰუმბოლდტის და ნიურნბერგ-ერლანგენის უნივერსიტეტები).

ნაწილი I

უნიფიცირებული და მოქნილი მეთოდოლოგიების საფუძვლები

თავი 1: ობიექტ-ორიენტირებული პროგრამირების მეთოდი

1.1. ობიექტ-ორიენტირებული დაპროგრამების არსი

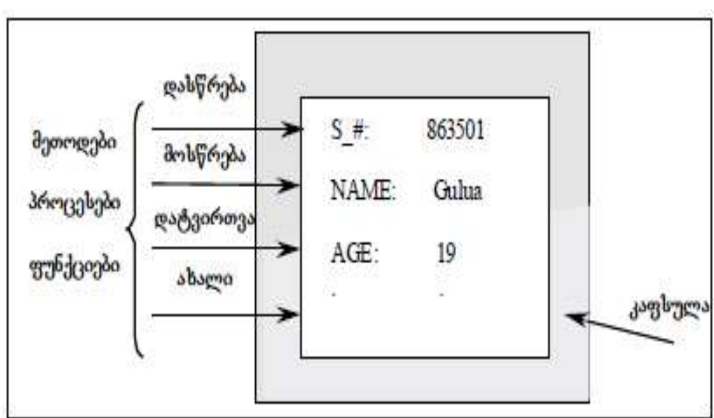
კომპიუტერული მეცნიერების პროფესორის, დანიელი ბრიან სტრაუსტრუპის (Bjarne Stroustrup, Aarhus-ის უნივ.) მიერ C++ ენის შექმნამ, რომელიც C ენაში ობიექტზე ორიენტირებული პარადიგმის ჩაშენებით იყო განპირობრბული, საფუძველი დაუდო თანამედროვე პროგრამული ინჟინერიის და ინფორმაციული ტექნოლოგიების ევოლუციური განვითარების ახალ ერას. ოო-პროგრამირების პარადიგმა (კლასებისა და ობიექტების საფუძველზე) ის კატალიზატორი აღმოჩნდა, რომელმაც კარდინალურად შეცვალა პროგრამული დეველოპმენტის მსოფლმხედველობა და თითქმის ყველა არსებული თუ ახლადშექმნილი კომპიუტერული ენების ძირითადი მეთოდი გახდა. მის ბაზაზე განვითარდა უნიფიცირებული და მოქნილი პროგრამირების მეთოდოლოგიები (UML, Agile და სხვ.), საგრძნობლად გაუმჯობესდა CASE-ტექნოლოგიები და საბოლოო ჯამში, სრულყოფის ახალ საფეხურზე ავიდა დაპროგრამების ავტომატიზაციის მეცნიერული მიმართულება [1].

ოო-მეთოდის მიზანია დიდი, რთული პროგრამული სისტემების კონსტრუირება (Software Engineering). იგი თვისებრივად ახალი კონცეფციების მატარებელი დაპროგრამების ტექნოლოგიაა. სისტემების ობიექტორიენტი-რებული ანალიზისა და ობიექტორიენტირებული დაპროექტების მეთოდებითა და რეალიზაციის მოქნილი ინსტრუმენტული საშუალებებით, მთლიანად მოიცავს სტრუქტურული დაპროგრამების იდეოლოგიასაც [2].

ობიექტი (Object) განიხილება, როგორც გარკვეული არსი (Entity), რომელიც ხასიათდება მდგომარეობით (მონაცემთა ერთობლიობა) და ქცევით (ფუნქციური პროგრამები). ობიექტის ქცევა ანუ რეაქცია, რომლის დროსაც მისი ახალი მდგომარეობა განისაზღვრება, დამოკიდებულია გარედან მოსულ ინფორმაციაზე, შეტყობინებებზე. ვინაიდან ობიექტი უმთავრესი ცნებაა, იგი საწყისია ობიექტორიენტირებული პროგრამებისა, ამიტომ დიდი მნიშვნელობა

აქვს მის სწორად გაგებას. 1.1 და 1.2 ნახაზებზე განხილულია ბიოლოგიური და ინფორმაციული უჯრედების მოდელები [13]. საინტერესოა მათი „მსგავსება“.

ნახ.1.1. ბიოლოგიური უჯრედი



ნახ.1.2. ინფორმაციული უჯრედი

ბიოლოგიური უჯრედი კაფსულირებულია მემბრანის (გარსის) საშუალებით, რომლითაც ის გამოიყოფა სხვა უჯრედებისა და გარემოსაგან. მას უნარი აქვს გარემოსგან მიიღოს ქიმიური სახის ინფორმაცია და გადასცეს უჯრედს შიგნით. შუაგულში მოთავსებულია ბირთვი, რომელიც უჯრედის ძირითადი ინფორმაციის მატარებელია. იგი შედგება ქრომოსომებისგან, რომლებიც გარკვეული გენეტიკის მქონეა. უჯრედის დაყოფის (გამრავლების) დროს ხდება მემკვიდრული თვისებების გადაცემა. ბირთვის გარშემო

დაჯგუფებულია სხვადასხვა ფუნქციის ელემენტები. მაგალითად, ენდოპლაზმური ბადე – ცილების წარმოქმნის ფუნქცია, მიტოქონდრები – ენერგიის გარდაქმნის ფუნქცია, ციტოპლაზმა (თხევადი მოძრავი გარემო) – ტრანსპორტირების ფუნქცია და ა.შ.

როგორია „ინფორმაციული უჯრედის" აგებულება და რა ანალოგიაა ბიოლოგიურთან ?

1.3 ნახაზზე განიხილება ობიექტი-სტუდენტი, რომელიც რეალური სამყაროს ნაწილია. იგი კაფსულირებულია, რომლის შიგნითაც მოთავსებულია ბირთვი ობიექტის თვისებების აღმწერ მონაცემთა ელემენტები

S_# (სტუდენტის ნომერი), NAME (გვარი, სახელი), AGE (ასაკი)

და ა.შ.

კაფსულაში შეღწევა და მონაცემების დამუშავება გარედან ყველა ფუნქციას არ შეუძლია, არსებობს წინასწარ განსაზღვრული მეთოდები (პროცესები ან ფუნქციები), რომელთაც ზოგადად სერვისული პროგრამები შეიძლება ვუწოდოთ. მათ აქვს უნარი შეაღწიოს კაფსულაში და გადაამუშაოს მონაცემები. ამგვარად, ინფორმაციული ობიექტი კაფსულირებული მონაცემე-ბისა და მათი დასამუშავებელი მეთოდებისაგან შედგება.

მონაცემები განსაზღვრავს ობიექტის სტატუსს, ანუ მდგომარეობას, ხოლო მეთოდები - ობიექტის ქცევას, მის რეაქციას გარედან მოსულ შეტყობინებაზე.

სტუდენტის მონაცემების დამუშავება შეუძლია მხოლოდ სამ ფუნქციას, როგორებიცაა მოსწრება, დასწრება, დატვირთვა. თუ მოვიდა შეტყობინება სწორედ ამ ინფორმაციის მისაღებად, მაშინ ობიექტი (კაფსულა) ცნობს მათ. სხვა შეტყობინებებისათვის მონაცემები დამალულია. თუ საჭიროა ახალი ფუნქციის დამატება, ის წინასწარ უნდა მოთავსდეს „კაფსულაში".

კლასის ცნება ამერიკელმა მეცნიერმა გრადი ბუჩმა (1990-იანი წლები) დაუკავშირა ობიექტთა სიმრავლის ისეთ სტრუქტურას, რომლის იერარქია გარკვეული მემკვიდრეობით და ქცევით განისაზღვრება. ამრიგად, ობიექტი კლასის კონკრეტულ გამოვლინებად, ეგზემპლარად განიხილება [14].

მემკვიდრეობითი კავშირები ობიექტ-ორიენტირებული მიდგომის აუცი-ლებელი კომპონენტია და იგი ნიშნავს ობიექტებს (ან ფუნქციებს) შორის მემკვიდრული თვისებების გადაცემას, თუ ისინი ერთ კლასს მიეკუთვნება.

კლასისა და ობიექტის მოდელირების და დაპროგრამების საკითხები ობიექტ-ორიენტირებული ენების (მაგალითად, C++, Java, C#, Python და სხვ.) შესწავლისას განხილება მრავალ ნაშრომში [15-17].

1.2. ობიექტები და კლასები. მონაცემთა აბსტრაქტული ტიპები

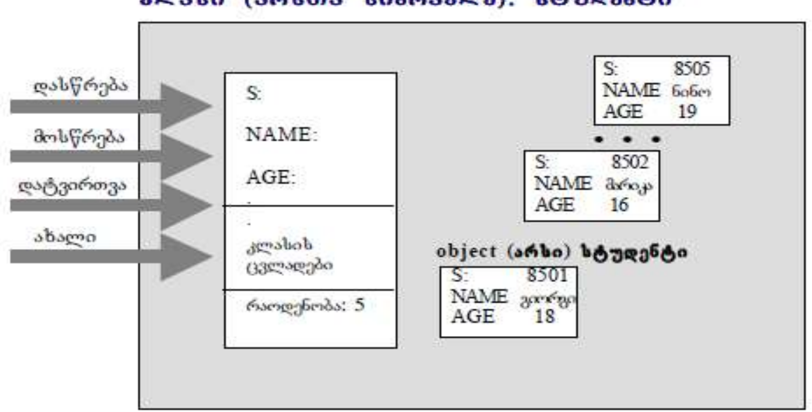
ობიექტები და კლასები ობიექტორიენტირებული დაპროგრამების ძირითადი კომპონენტებია. დავაზუსტოთ მისი განსაზღვრება და ავხსნათ ამ ცნების მიმართება კლასის ცნებასთან.

ობიექტი არის რეალური ან წარმოსახვითი სამყაროს ნაწილი, ინდივიდუალური ეგზემპლარია. ის არსია (Entity) და აქვს უნიკალური იდენტიფიკატორი, რომლითაც სხვა არსებისაგან განსხვავდება. ობიქტს აქვს ინდივიდუალური *თვისებები,* რომლებიც გამოიხატება მონაცემებით (ატრიბუტები, ცვლადები) და *ქცევით* (მეთოდები, ფუნქციები) გარემოში.

ობიექტებს შორის კომუნიკაციები ხორციელდება *შეტყობინებების* გადაცემით. მიღებული შეტყობინების საფუძველზე ობიექტში აქტიურდება შესაბამისი მეთოდი, რომელიც მის ქცევას განსაზღვრავს და შეუძლია გადაიყვანოს იგი სხვა მდგომარეობაში (იცვლება ატრიბუტების და ცვლადების მნიშვნელობები). ობიექტის მდგომარეობა ხასიათდება *სტატიკური* კომპონენტით (ატრიბუტები) და *დინამიკური* კომპონენტით (ატრიბუტთა მნიშვნელობები). ობიექტის ქცევა მიუთითებს იმაზე, თუ როგორ იცვლება მისი მდგომარეობები და სხვა ობიექტებთან ურთიერთმიმართებანი.

ობიექტის ცნება დაპროგრამების ენაში პირველად გამოყენებულ იქნა Simiula-ში, ხოლო მისი ზემოაღნიშნული კლასიკური განმარტება მოგვცა გრადი ბუჩმა [14]. მან ასევე ჩამოაყალიბა კლასის განმარტება.

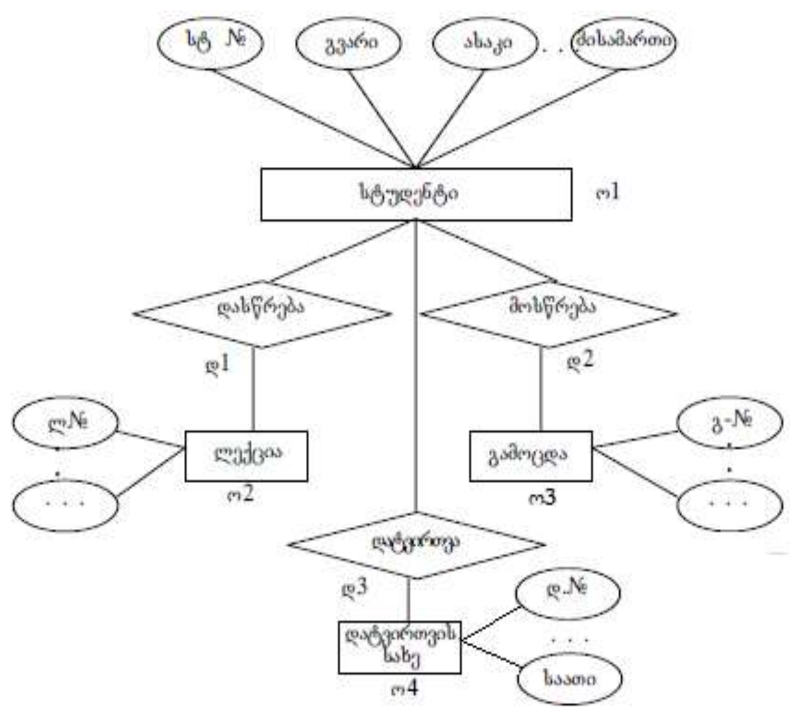
კლასი არის ერთი ტიპის ობიექტების სიმრავლე, რომელთაც აქვს მსგავსი სტრუქტურა და ქცევა. 1.3 ნახაზზე ნაჩვენებია ერთი კლასის მაგალითი.



ნახ.1.4. კლასი და ობიექტები

მონაცემთა რელაციური ბაზების და არსთა-დამოკიდებულებათა (Entity-Relationhip) მოდელის სქემის განხილვისას, ადვილად შესამჩნევია გარკვეული ანალოგია საპრობლემო სფეროს კონცეპტუალურ მოდელსა, მის ლოგიკურ სტრუქტურასა და ფიზიკურ ორგანიზაციასთან.

მსგავსება შემდეგი თვალსაზრისით შეიძლება იქნეს განხილული: კლასი ***სტუდენტი*** ეთანადება 1.4 ნახაზზე წარმოდგენილი კონცეპტუალური სქემის ფრაგმენტს.



ნახ.1.5. საპრობლემო გარემოს კონცეპტუალური (ER) მოდელი
ო-ობიექტი, დ - დამოკიდებულება

ამ კონცეპტუალური სქემის გადატანით მონაცემთა ბაზების ლოგიკურ სტრუქტურაში მივიღებთ სქემას, რომელიც ახლოა კლასის მოდელთან. ლოგიკური სტრუქტურა ატრიბუტებისგან შემდგარი სქემაა, რომელიც ობიექტების სტატიკურ კომპონენტებად მოვიხსენიეთ ზემოთ. კლასის ობიექტები კი ეთანადება ამ ლოგიკური სტრუქტურის ქვეშ მდგარ ფიზიკურ ჩანაწერებს (ჩანაწერის უნიკალური ნომრითა და ველების მნიშვნელობებით).

მონაცემთა რელაციური ბაზების თეორიაში გამოიყენება მონაცემებისა და პროგრამების ერთმანეთისაგან იზოლირების პრინციპი, რათა განხორციელდეს მათ შორის სრული დამოუკიდებლობა. მონაცემთა აბსტრაქტული სტრუქტურების გამოყენებით ეს საკითხი დადებითად იქნა გადაჭრილი.

ობიექტ-ორიენტირებული დაპროგრამების ენა ფლობს კლასების, ობიექტების, მათი მნიშვნელობების დამუშავების მეთოდების რეალიზაციის საშუალებებს. როგორც აღვნიშნეთ, ძირითადი პრინციპი მონაცემების *ინკაფსულირებაშია. კლასი კი თავისი ბუნებით მონაცემთა ახალი ტიპია, რომელიც იქმნება თვით მომხმარებლის მიერ.* როგორ უნდა გვესმოდეს ეს საკითხი?

დაპროგრამების ენებში არის მონაცემთა სტანდარტული ტიპები (მაგალითად, int a), რომელიც აცხადებს a ცვლადს მთელი ტიპით. ეს a პროგრამისათვის ობიექტია. თუ ენას აქვს მონაცემთა ახალი ტიპის შექმნის შესაძლებლობა, ეს მის სიმძლავრეზე მიუთითებს, მაგალითად, C++ ენის ფრაგმენტის მოშველიებით გამოვაცხადოთ Magistrant როგორც ახალი კლასი:

```
class Magistrant {
    private:
        char Name [20];
        int Age;
        char Specification [30];
    public:
        Input-Name (NAME);
        Input-Age (AGE);
        Input-Spec (SPEC); };
    void main(void) {
         Magistrant M1,M2,M1.....; // da misi obieqtebic:
    . . . }
```

ამგვარად, Magistrant ისეთივე ტიპია, როგორც int, char და ა.შ. ყოველი ობიექტი M1, M2, M4..... არის კონკრეტული მაგისტრანტი, რომელიც სტრუქტურას იღებს class Mgistrant(....)-იდან private და public ოპერატორები განსაზღვრავს მონაცემებს (Name, Age, Specifikation) და ფუნქციებზე (Input-Name, Input-Age, Input-Spec) მიმართვის შესაძლებლობას. პირველი ლოკალურია და მალავს ამ მონაცემებს სხვა ობიექტებისათვის. მათთან მიმართვა შეუძლია მხოლოდ მოცემული ობიექტის ფუნქციებს და ზოგჯერ მათ „მეგობრებსაც"-friend). Public იძლევა ნებართვას, რათა მის შემდეგ მდგომი ფუნქციები

გამოყენებულ იქნას ობიექტის გარედან. განხილული მაგალითის განზოგა-დებით შეიძლება დავასკვნათ, რომ *კლასების საფუძველი მონაცემთა აბსტრაქტული ტიპებია.*

განასხვავებენ *პროცედურულ და მონაცემთა აბსტრაქციებს.* პირველი ცალ-ცალკე განიხილავს პროცედურის მიზანს, მის შინაგან რეალიზაციას და მონაცემთა აბსტრაქციას. ეს უკანასკნელი ნიშნავს, რომ საჭიროა მხოლოდ იმის ცოდნა, თუ რა ოპერაციებს ასრულებს მოცემული პროგრამული მოდული. არაა აუცილებელი ვიცოდეთ, თუ რომელ მონაცემებს ამუშავებს (ისინი დამალულია) და როგორ სრულდება ეს ოპერაციები.

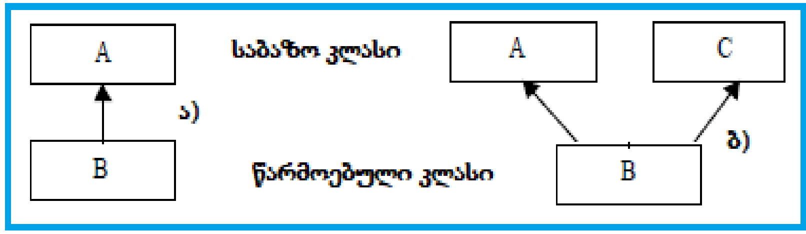
1.4. კლასების იერარქია, მემკვიდრეობითობა

სამი ძირითადი პრინციპი: *ინკაფსულაცია, მემკვიდრეობითობა და პოლი-მორფიზმი* არის ობიექტორიენტირებული პროგრამირების საბაზო სტილი. პირველი მათგანი წინა პარაგრაფში განვიხილეთ და კლასის ცნებამდე მივედით. კლასებს შორისაც არსებობს გარკვეული დამოკიდებულებანი.

რას ნიშნავს ეს ?

თუ არსებობს გარკვეულ კლასთა ბიბლიოთეკები, რომლებიც შექმნილია ამ მომენტამდე, მაშინ სასურველია მოხდეს მათი გამოყენება ახალი ამოცანების გადასაწყვეტად.

არსებული კლასების ბაზაზე უნდა განისაზღვროს ახალი კლასები, მოხდეს მათი გაფართოება და მოდიფიკაცია. ყოველივე ეს მნიშვნელოვნად ამცირებს ახალი სისტემების დაპროექტებისა და რეალიზაციის ვადებს. სწორედ ამაშია ობიექტორიენტირებული პროგრამირების მეთოდის გამოყენების ეფექტურობის საიდუმლოება. ორი კლასიდან ერთი ბაზისური, ხოლო მეორე წარმოებულია. 1.5 ნახაზზე ნაჩვენებია მარტივ-მემკვიდრეობითი (single inheritance) და მრავალ-მემკვიდრეობითი (multiple inheritance) იერარქიული კავშირები.



ნახ.1.6. მემკვიდრეობითობა

საბაზო კლასი შეიძლება იყოს აბსტრაქტული, რომელსაც თვითონ არ აქვს კონკრეტული ეგზემპლარი, მაგრამ გამოიყენება სხვა წარმოებული კლასების მისაღებად.

იერარქიული კავშირების აღწერა შეიძლება გრაფის საშუალებით, ორიენტირებული ხის სახით, ციკლების გარეშე. გრაფის წვეროებს კლასები შეესაბამება. ფესვური წვერო ის კლასია, რომელიც აღწერს ყველაზე ზოგად თვისებებს, ისინი კი გადაეცემა ქვედა იერარქიის წარმოებულ კლასებს.

ამგვარად შეიძლება დავასკვნათ, რომ ფუნქციური შესაძლებლობების თვალსაზრისით *წარმოებული კლასები, უფრო მძლავრია, ვიდრე საბაზო კლასები.* ეს იმიტომ, რომ წარმოებულ კლასს შეუძლია თავისი ფუნქციების შესრულებაც და საბაზო კლასისაც, საბაზოს კი - მხოლოდ თავისი.

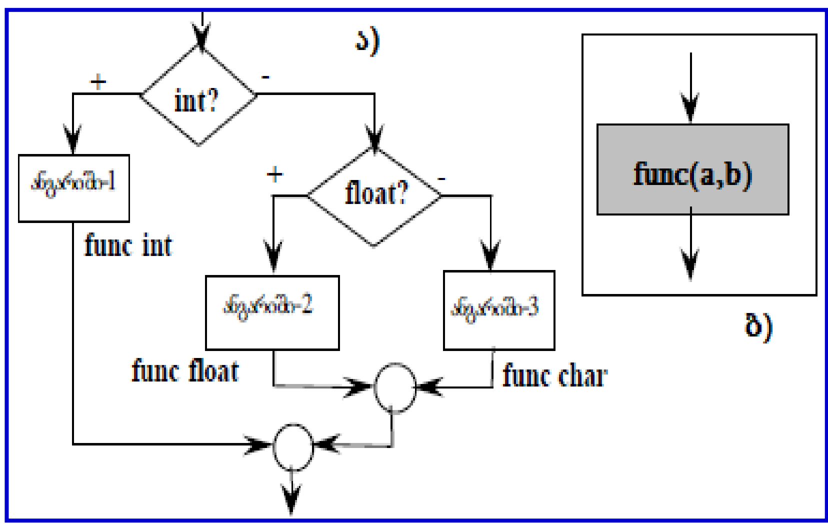
წარმოებულ კლასს შეუძლია გამოიყენოს საბაზო მონაცემებიც (თუ ისინი არაა დამალული სპეციალური ატრიბუტებით, მაგალითად, private) და ა.შ.

1.5. პოლიმორფიზმი

ობიექტორიენტირებულ პროგრამირებაში ინკაფსულაციისა და მემკვიდ-რეობითობის გვერდით მნიშვნელოვანი ადგილი უკავია *პოლიმორფიზმის* ცნებას. ის ბერძნული სიტყვაა polymorphism და „მრავალფორმიანობას" ნიშნავს.

პოლიმორფიზმი ობიექტის თვისებაა. იგი უზრუნველყოფს ერთსა და იმავე ფუნქციების გამოყენებას სხვადასხვა ამოცანათა გადასაწყვეტად. ეს ამოცანები შეიძლება მოითხოვდეს ფუნქციების და მათი არგუმენტების სხვადა-სხვა ტიპებს, რასაც პოლიმორფიზმი ადვილად წყვეტს ე.წ. *ვირტუალური ფუნქციების* რეალიზაციით (virtual function) ან *ფუნქციათა გადატვირთვით* (overloaded function).

მონომორფულ სისტემებში ყოველი ფუნქცია და მისი არგუმენტები მხოლოდ ერთი ტიპითაა შეზღუდული. მაგალითად, ჩვეულებისამებრ, C ენაში საჭიროა დაიწეროს ორი სხვადასხვა ფუნქცია int - func(int, int) და float – func(float, float), თუკი გვინდა ორ მთელ რიცხვზე ან ორ ნამდვილ რიცხვზე არითმეტიკული ოპერაციების ჩატარება (ნახ. 1.6-ა). პოლიმორფიზმის იდეა C++ ენაში საშუალებას გვაძლევს დავწეროთ მხოლოდ ერთი func(a,b) ფუნქცია, შემდეგ კი, არგუმენტების ტიპების ანალიზის საფუძველზე, ენის კომპილატორი თვითონ აირჩევს, თუ რომელი ოპერაციები შეასრულოს (ნახ.1.6-ბ).



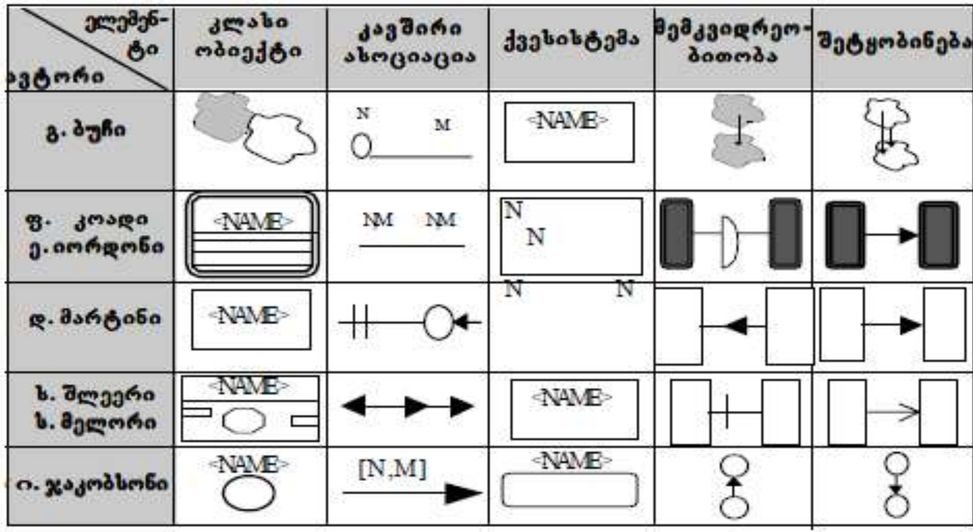
ნახ.1.7. პოლიმორფიზმი (ბ) C++

1.6. ობიექტ-ორიენტირებული დიაგრამების აგება სისტემების დაპროექტების ეტაპზე

დიდი სისტემების პროცესების მართვისათვის პროგრამული პაკეტების შექმნა და მათი თანხლება ფუნქციონირების ეტაპზე, საკმაოდ რთული საკითხია. იგი ბევრადაა დამოკიდებული იმაზე, თუ როგორ, რა მეთოდებითა და საშუალებებით იქნა ეს სისტემა შექმნილი. განსაკუთრებული ყურადღება არის გასამახვილებელი სისტემის დაპროექტების ეტაპზე.

ეს საკითხი უნდა განვიხილოთ ობიექტ-ორიენტირებული მიდგომის თვალსაზრისით; კერძოდ, ობიექტორიენტირებული დაპროექტების ისეთი მნიშვნელოვანი საკითხი, როგორიცაა ასაგები სისტემის ობიექტ-ორიენტირებული დიაგრამის (ოო-დიაგრამის) გრაფიკული გამოსახვა.

1.7 ნახაზზე წარმოდგენილია ობიექტ-ორიენტირებული ანალიზისა და დაპროექტების ცნობილი კლასიკოსების ბუჩის, კოად-იორდონის, ჯაკობსონის, მარტინისა და შლეერ-მელორის გრაფიკული აღწერის საშუალებანი [18]. რა თქმა უნდა, არსებობს სხვა სისტემებიც, მაგრამ ამ სფეროში ძირითადად ზემოაღნიშნული ინსტრუმენტული საშუალებები დომინირებს.



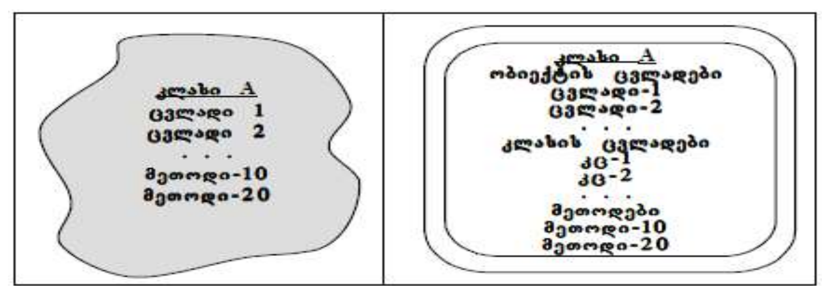
ნახ.1.8. ოო-დიაგრამების აგების აღნიშვნათა სისტემა

ზოგიერთი კომენტარი ნახაზთან დაკავშირებით.

– <name> – კლასის ან ობიექტის სახელია. გრადი ბუჩს კლასი წყვეტილი, ხოლო ობიექტი კონტურული ღრუბლისმაგვარი ფიგურით აქვს მოცემული. კოად-იორდონი მათ ორმაგი მრგვალკუთხედებით წარმოადგენს და ა.შ.

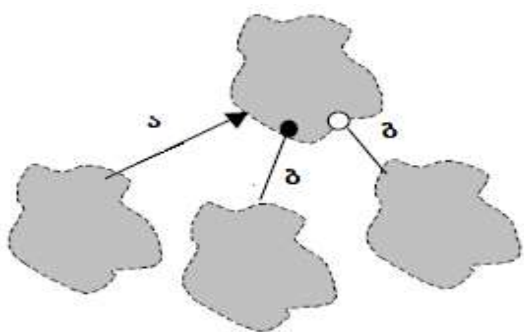
1.8 ნახაზზე გადმოცემულია კლასის წარმოდგენა ოო - დიაგრამისთვის ბუჩისა და კოად-იორდონის აღნიშვნათა სისტემების მაგალითზე.

როგორც ვხედავთ, კლასი არის მონაცემებისა და მეთოდების ერთობლიობა, აბსტრაქცია, ხოლო ობიექტი – რეალობა, კონკრეტული ეგზემპლარები.



ნახ.1.9. კლასის წარმოდგენა ოო-დიაგრამით გ. ბუჩისა (ა) და
კოად-იორდონის (ბ) მიხედვით

– ***კავშირის ასოციაციები კლასებს ან ობიექტებს შორის*** მრავალი სახისაა. ამიტომაც 1.9 ნახაზზე მოცემული გვაქვს მათი ძირითადი ტიპები. ესაა ტიპური მათემატიკური ასახვის ფუნქციები, დამოკიდებულებანი კლასებსა და მის ელემენტებს შორის.



ნახ.1.9. ძირიად კავშირთა ასოციაციები

შეიძლება განვიხილოთ აგრეთვე ისეთი დამოკიდებულებები, რომლებიც სტრუქტურულ მოწესრიგებას ემსახურება, მაგალითად, *კლასიფიკაცია* და *აგრეგაცია.* პირველი მათგანი აერთიანებს ობიექტებს გარკვეული კრიტე-რიუმებით, რომლებიც მსგავს, მაგრამ არაიდენტურ თვისებებს ეყრდნობა. მეორე კი მთელისა და შემადგენელი ნაწილის მიმართების ტიპური ასახვაა. 1.10 ნახაზზე ნაჩვენებია სქემატურად კლასებს შორის მემკვიდრეობითი, აგრეგირე-ბული და რელაციური კავშირების გამოსახვა.

A კლასი — 1 — B კლასი	A→B	A კლასის ყოველ ობიექტს შეესაბამება B კლასის მხოლოდ 1 ობიექტი
A კლასი — 0.1 — B კლასი	A_\B	A კლასის ყოველ ობიექტს შეესაბამება B კლასის არც ერთი ან ერთი ობიექტი
A კლასი — M — B კლასი	A→→B	A კლასის ყოველ ობიექტს შეესაბამება B კლასის ერთი ან რამდენიმე ობიექტი
A კლასი — 0M — B კლასი	A_\\B	A კლასის ყოველ ობიექტს შეესაბამება B კლასის 0, 1 ან რამდენიმე ობიექტი

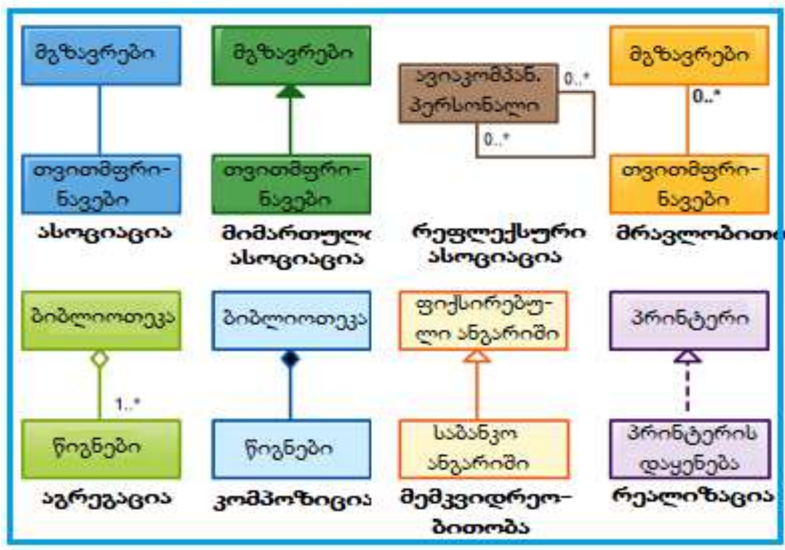
ნახ.1.10. კლასთაშორის ასოციაციები: მემკვიდრეობითი (ა), აგრეგირებული (ბ) და რელაციური (გ) /გ. ბუჩი, 1996/

შენიშვნა: UML მეთოდოლოგია, მისი ინსტრუმენტული საშუალებები საგრძნობლად განვითარდა. 2018 წლისთვის UML/2-ის ბოლო ვერსია არის 2.6.1 [19]. განვითარებაში იგულისხმება ამ მეთოდოლოგიის შემადგენელი კომპონენ-ტებისსა და კლასთა ასოციაციური კავშირების გაფართოება და სრულყოფა საკვლევი ობიექტის ასახვის სემანტიკური სიმძლავრის ამაღლების მიზნით.

ეს, ერთი მხრივ, კი სრულყოფს საბოლოო მოდელის სიზუსტეს, მაგრამ, მეორე მხრივ, ართულებს მისი გამოყენების სიმარტივეს, რაც მნიშვნელოვანი მომენტია მომხმარებელთა მიზიდვის თვალსაზრისით.

ბოლო პერიოდში მეტი პრიორიტეტი შეიძინა UML-ის კონკურენტმა მეთოდოლოგიამ, რომელიც Agile სახელწოდებითაა ცნობილი (მაგალითად, ექსტრემალური პროგრამირება (XP), Scrum, Kanban და სხვ.) [20].

UML/2-ის კლასთა ასოციაციის კავშირების საილუსტრაციო მაგალითები მოყვანილია 1.11 ნახაზზე, მისი დეტალური ანალიზი კი მრავლად არის წარმოდგენილი ინტერნეტ-წყაროებში [33].



ნახ.1.11. კლასთა ასოციაციური კავშირები (UML/2.6.1,2018)

– *ქვესისტემა* კლასების გარკვეული სახელმინიჭებული ჯგუფია, რომელიც ამოცანის პრაგმატული (მიზნობრივი) გადაწყვეტის თვალსაზრისით განიხილება. კოად-იორდონის შესაბამის მართკუთხედში ასო N სწორედ ამ კლასთა სუბიექტური ჯგუფის იდენტიფიკატორის როლს ასრულებს. იგი ნატურალური რიცხვია 1,2,....და ა.შ. მაგალითად,

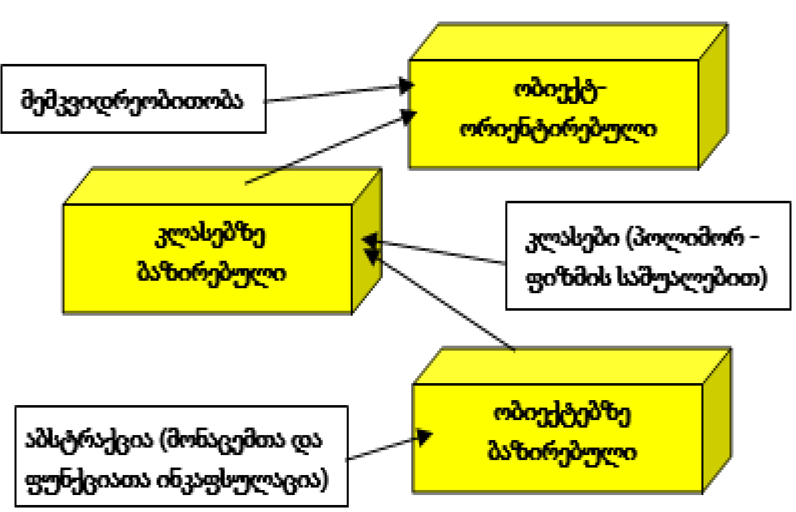
1-ელი სუბიექტური ჯგუფი შეიძლება იყოს ***სასწავლო აუდიტორიები, ლაბორატორიები, კომპიუტერული კლასები*** და სხვ.;

მე-2 სუბიექტური კლასი შეიძლება იყოს ***პროფესორ-მასწავლებლები, ლაბორანტები, ადმინისტრაციული და საინჟინრო-ტექნიკური პერსონალი.;***

მე-3 სუბიექტური კლასი ***სტუდენტები, მსმენელები, მოსწავლეები*** და ა.შ.

არსებობს, რა თქმა უნდა, ისეთი კლასებიც, რომლებიც ერთდროულად რამდენიმე სუბიექტური ჯგუფის წევრი შეიძლება იყოს. მაგალითად, ***ლაბორატორიული მეცადინეობა კონკრეტულ საგანში, კონკრეტულ ლექტორთან, კონკრეტულ სასწავლო ოთახში, დაწყების დრო და ხანგრძლივობა.***

მემკვიდრეობითობა ობიექტორიენტირებული მიდგომის ერთ-ერთი ძირითადი და აუცილებელი კომპონენტია. 1.12 ნახაზზე მოცემულია ოო-მიდგომის ერთ-ერთი ძირითადი პრინციპის მოდელი, რომლის „მწვერვალზეც“ სწორედ მემკვიდრეობითობის კონცეფციის რეალიზებით (ენაში) ვაღწევთ ასვლას.

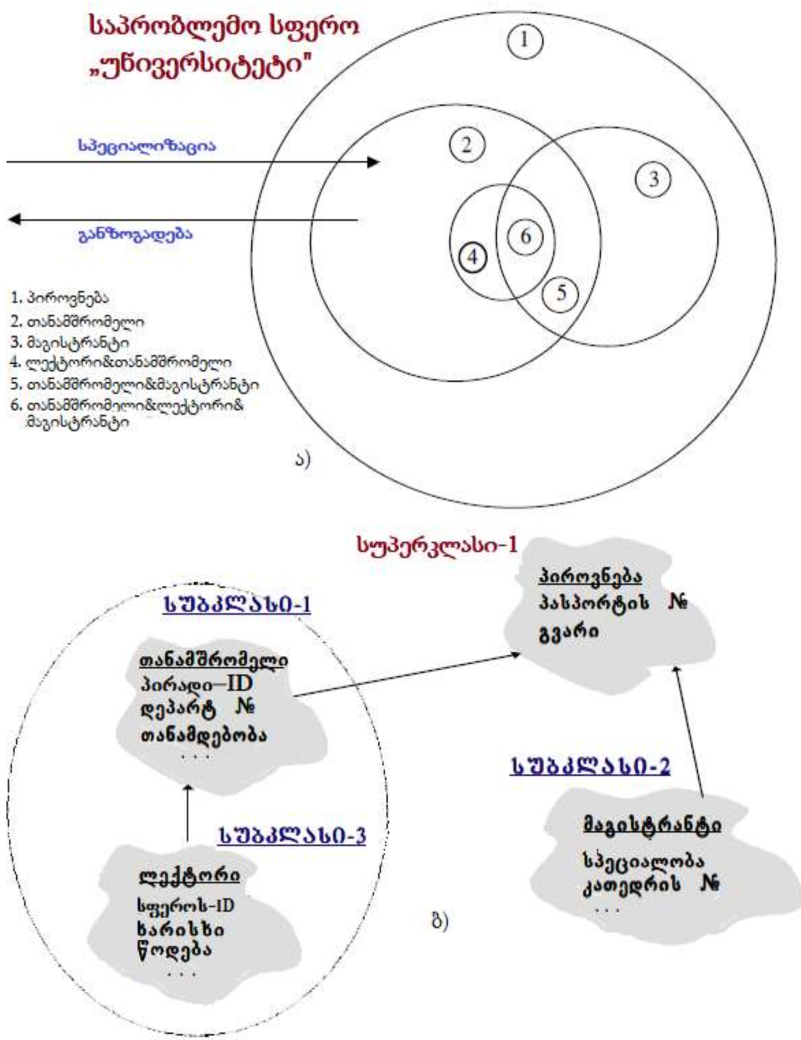


ნახ.1.12. ოო-მიდგომის პრინციპის სქემა

განვიხილოთ საპრობლემო გარემო უნივერსიტეტი (ნახ.1.13). მემკვიდრეობითი სტრუქტურები გამოიკვეთება სპეციალიზაციითა და განზოგადებით, რომლებიც ურთიერთსაპირისპირო პროცედურებია.

მაგალითად, უნივერსიტეტში არსებული ადამიანები განსაზღვრული დროის მომენტში შეიძლება დახარისხდეს სტატუსის მიხედვით: ***პიროვნება, თანამშრომელი, მაგისტრანტი, ლექტორი*** და ა.შ.

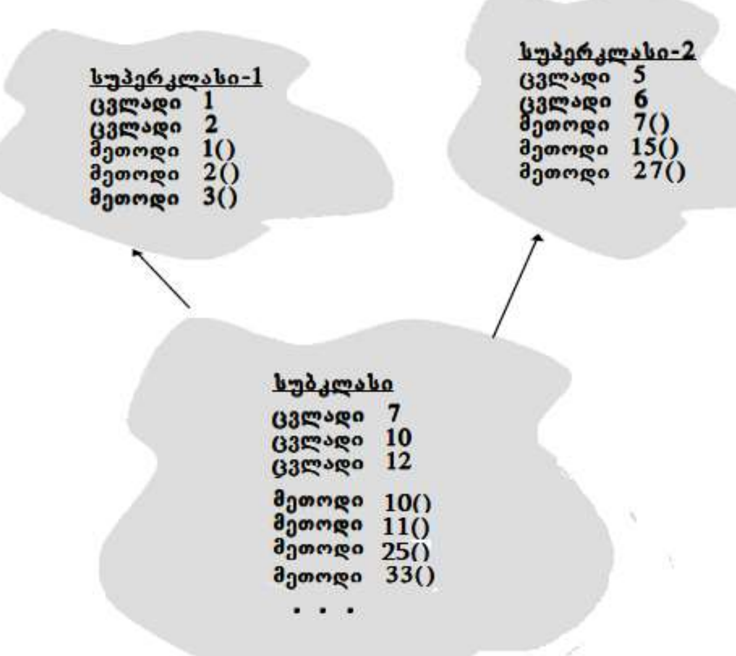
შეიძლება ადგილი ჰქონდეს მათ გადაკვეთასაც.



ნახ.1.14. მემკვიდრეობითობის სტრუქტურა: სპეციალიზაცია და განზოგადება (ა), მათიწარმოდგენა ოო-დიაგრამით (ბ)

ხშირად საჭიროა გამოისახოს *მრავალჯერადი მემკვიდრეობითობის კავშირი.* ამ დროს „შვილი" (სუბკლასი) იღებს მემკვიდრეობის „გენებს" (ცვლადებს და მეთოდებს) თავისი „მშობლებიდან" (სუპერკლასებიდან). ამავე

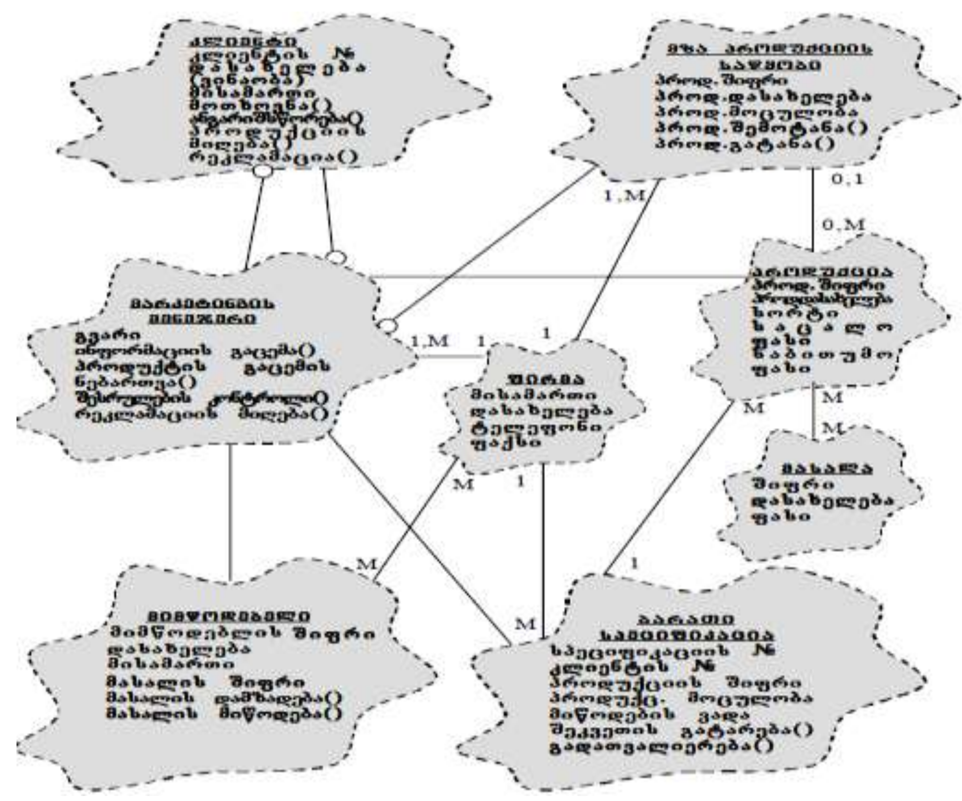
დროს მათ თვითონაც ექნება განსხვავებული, ახალი თვისებებიც (სხვა ცვლადები და მეთოდები), ზოგადი მაგალითი მოცემულია 1.14 ნახაზზე.



ნახ.1.15. მრავალჯერადი მემკვიდრეობითობის მაგალითი
ორი სუპერკლასით

– *შეტყობინება* კლასებსა და ობიექტებს შორის ასევე განმსაზღვრელია მათი სხვადასხვა პროცესის ინიციალიზაციისათვის. შეტყობინებაში ჩადებულია კონკრეტული მოთხოვნის არსი, რომელმაც უნდა გამოიწვიოს კლასში შესაბამისი მეთოდის პროვოცირება, მონაცემთა გადამუშავება და შემდგომი გადაადგილება სხვა კლასების ან ობიექტებისაკენ. პროცესი მთავრდება შესაბამისი შედეგების მომზადებითა და უკან დაბრუნებით (შეტყობინების მოსვლის მისამართით). კონკრეტულ შეტყობინებათა გადამუშავებისათვის გამოიყენება შესაბამისი სცენარები.

1.15 ნახაზზე წარმოდგენილია ოო-დიაგრამის ფრაგმენტი საპრობლემო სფეროსათვის „მარკეტინგის მართვა საწარმოო ფირმაში“. დიაგრამა აგებულია გ. ბუჩის აღწერის სისტემაში. ნახაზზე ნაჩვენებია სისტემის ძირითადი კლასები და მათი ურთიერთკავშირები, კლასთა ცვლადები (მონაცემები) და მათი ფუნქციები (მეთოდები).



ნახ.1.16. ოო-დიაგრამა საპრობლემო სფეროსათვის „მარკეტინგი“

ამ საბაზო ოო-დიაგრამის საფუძველზე შესაძლებელია მოთხოვნების შესრულება. 1.16 ნახაზზე მოცემულია ორი შეტყობინების მაგალითი.

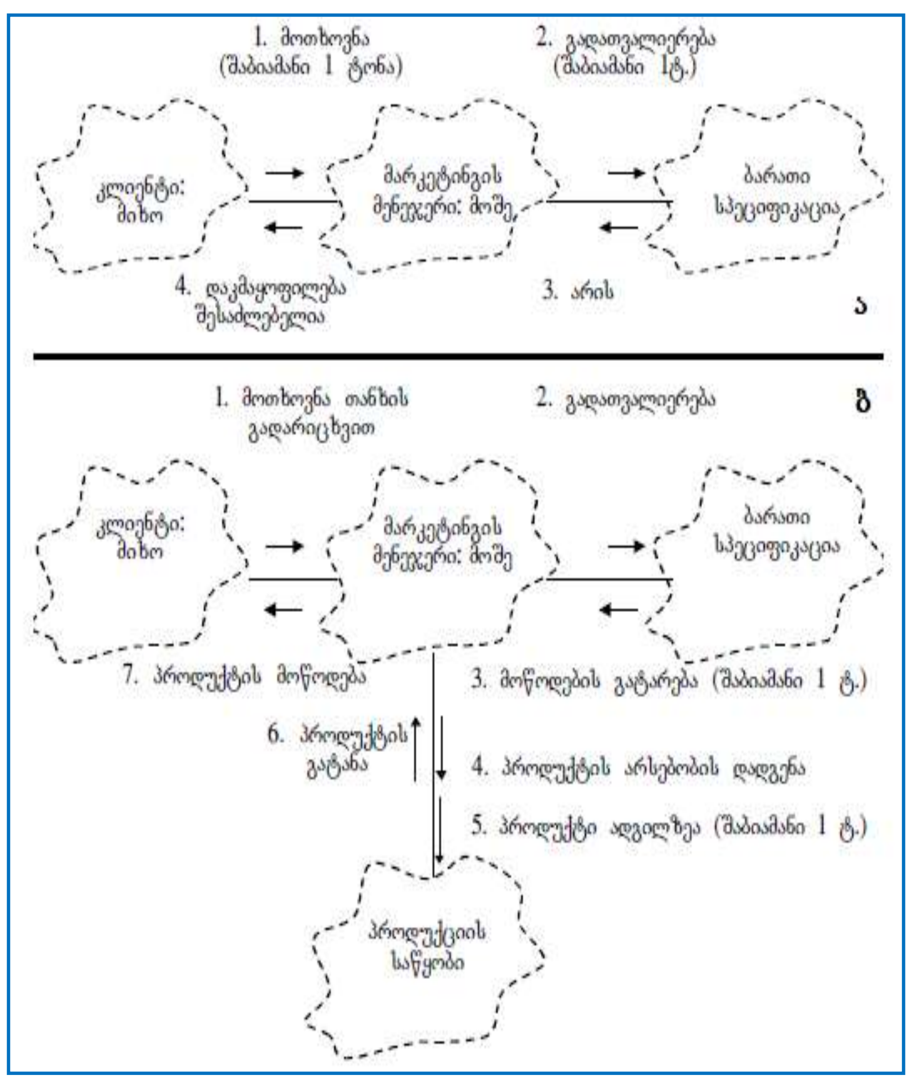
პირველი – კლიენტს აინტერესებს ინფორმაცია, არის თუ არა ფირმაში მისთვის საჭირო პროდუქცია (ა);

მეორე – კლიენტმა უნდა გაიტანოს ფირმიდან მისი კუთვნილი პროდუქცია (ბ).

თუ შევაჯამებთ განხილულ საკითხებს, ადვილად დავრწმუნდებით, რომ გამოკვეთილია ორი სახის პრობლემა:

1) ობიექტთა მდგომარეობის მოდელირება (პროცესები შეტყობინების დამუშავებამდე) და

2) ობიექტთა ქცევის მოდელირება (პროცესები შეტყობინების დამუშავებით და შედეგებით).



ნახ.1.17. შეტყობინების დამუშავების სცენარი
ოო-დიაგრამით

თავი 2

ბიზნეს-პროცესების მოდელირების CASE ინსტრუმენტები და მეთოდოლოგიები

ვიზუალური პროგრამირების ენა (VPL) იყენებს გრაფიკულ ელემენტებს და ფიგურებს პროგრამების ასაგებად. VPL-ის ბაზაზე იქმნება 2D და 3D (განზომილებიანი) პროგრამული აპლიკაციები, რისთვისაც მის რესურსებში წარმოდგენილია გრაფიკული ელემენტები, ტექსტები, სიმბოლოები და სურათები (ნახატები) [21].

ვიზუალური ენის იდეალური მაგალითებია:

1) BPMN/BPEL. Business Process Model and Notation – ესაა ბიზნესპროცესების დიზაინის და სრულყოფის ვიზუალური ინსტრუმენტი (იყენებენ ბიზნეს ანალიტიკოსები), ხოლო Business Process Execution Language კი – მისი პროგრამული რეალიზაციისთვის (იყენებენ პროგრამისტ-დეველოპერები) [22];

2) UML მეთოდოლოგია თავისი დიაგრამებით და CASE ინსტრუმენტული საშუალებები, რა თქმა უნდა, პროგრამული კოდის ავტომატიზებული გენერაციის შესაძლებლობებით [23, 32].

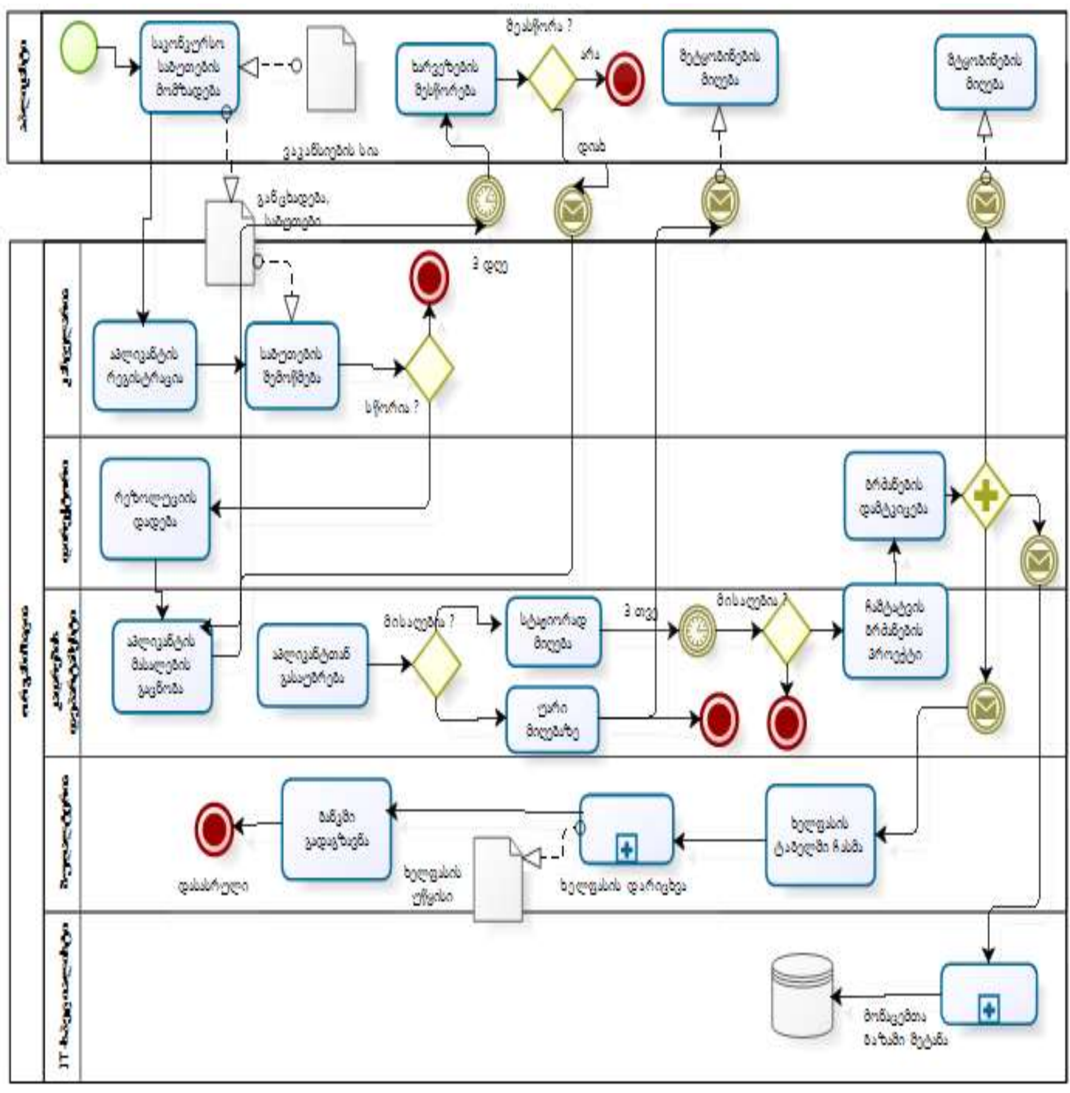
მთავარი განსხვავება BPMN-სა და UML-ს შორის პრაგმატული მიზანია: BPMN არის *პროცეს-ორიენტირებული* CASE-ინსტრუმენტი, ხოლო UML კი – *ობიექტ-ორიენტირებული.* BPMN ფართოდ გამოყენებადია როგორც IT-სთვის, ასევე ბიზნესისთვის, ხოლო UML უფრო შესაფერისია IT სისტემების განვითარებაზე (დეველოპმენტისთვის) და ნაკლებად შესაფერისი ბიზნეს-პროცესების სრულყოფისთვის.

მომდევნო პარაგრაფებში ჩვენ შევეხებით ამ CASE-ინსტრუმენტებს უფრო დეტალურად.

შენიშვნის სახით შეიძლება აღვნიშნოთ, რომ, მაგალითად, BPMN მოდელების აგება შესაძლებელია სხვადასხვა კონკრეტული ინსტრუმენტის გამოყენებით, როგორიცაა: Bizag (Bizagi Limited, Freeware), Enterprise Architect (Sparx Systems) ან სხვ. მათი აღნიშვნების სისტემა (Notation) მსგავსია და ბიზნეს-პროცესების მოდელებიც ერთნაირ სტილში კეთდება.

2.1. ბიზნეს-პროცესებზე ორიენტირებული მოდელირების ენა (BPMN)

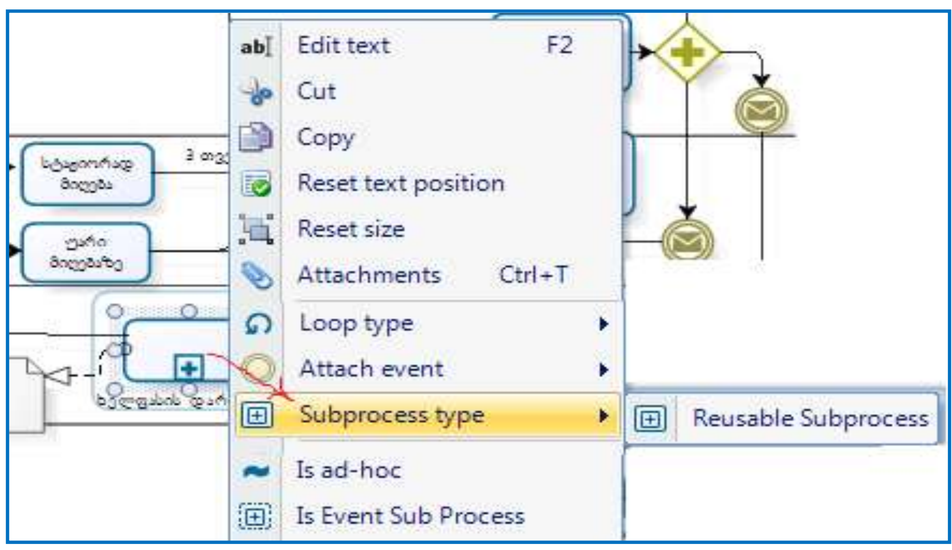
განვიხილოთ ბიზნეს-პროცესის მოდელირების კონკრეტული მაგალითი BPMN-ის, როგორც CASE-ინსტრუმენტის ბაზაზე. 2.1 ნახაზზე წარმოდგენილია ვირტუალურ ორგანიზაციაში ახალი თანამშრომლის მიღების ამოცანის ბიზნეს-პროცესის BPMN დიაგრამა.



ნახ. 2.1. ახალი თანამშრომლის მიღების BPMN მოდელის ფრაგმენტი

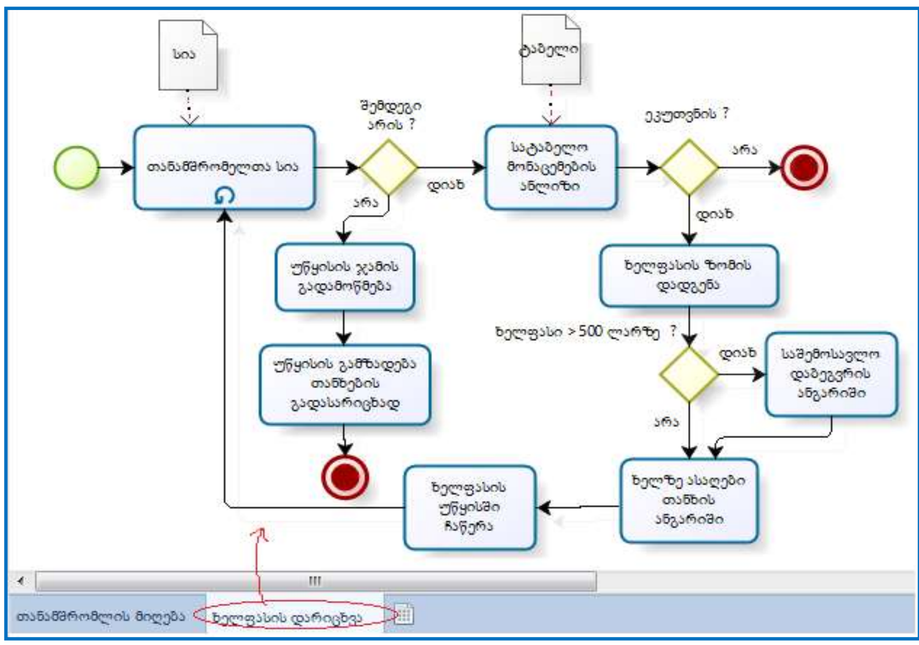
მთლიანი ბიზნესპროცესი დაყოფილია ქმედებებად (Activities), რომლებიც გამოსახულია მრგვალკუთხედებით, გადასასვლელები ქმედებებს შორის კი – ისრებით. დოკუმენტები, რომლებსაც საწყისად იყენებს ეს ქმედებები ან თვითონ ქმნის როგორც საშედეგოს, ნაჩვენებია კუთხეჩამოკეცილი ფურცლის სახით. ეს ფურცლები შეერთებულია წყვეტილი ისრებით იმ ქმედებებთან, რომლებიც მათ ქმნის (ქმედებიდან გამომავალი ისარი) ან გამოიყენებს შესასვლელზე (ქმედებისკენ მიმართული ისარი) [24].

სქემაზე „+"-ით აღნიშნულია ქმედება, რომელიც ქვეპროცესია. მისთვის ცალკე აიგება დიაგრამა, მაგალითად, ბლოკზე „ხელფასის დარიცხვა" მარჯვენა ღილაკით ავირჩევთ Subprocess პუნქტს (ნახ.2.2).



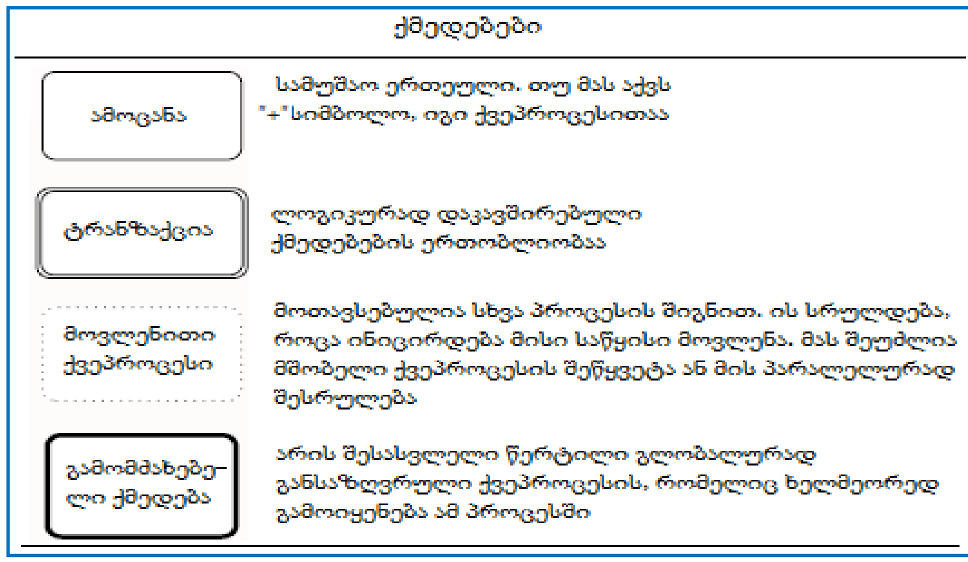
ნახ.2.2. ქვეპროცესის შექმნის ინიცირება

2.3 ნახაზზე ნაჩვენებია ახლად აგებული ქვეპროცესის BPMN სქემა. ძირითადი პროცესის და ქვეპროცესების დიაგრამათა ასარჩევად გამოიყენება გადამრთველი სათაურებით (ქვედა მარცხენა კუთხეში).



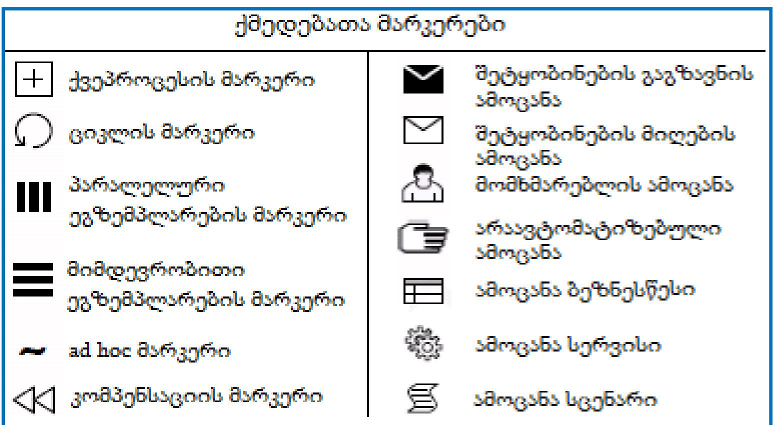
ნახ. 2.4. ხელფასის დარიცხვის ქვეპროცესის სქემა

ბიზნესპროცესების მოდელის ასაგებად გამოყენებულია BPMN ნოტაციის ელემენტები, რომლებიც ასახულია 2.4-2.7 ნახაზებზე მოცემულ ცხრილებში.



ქმედებები	
ამოცანა	სამუშაო ერთეული. თუ მას აქვს "+"სიმბოლო, იგი ქვეპროცესითაა
ტრანზაქცია	ლოგიკურად დაკავშირებული ქმედებების ერთობლიობაა
მოვლენითი ქვეპროცესი	მოთავსებულია სხვა პროცესის შიგნით. ის სრულდება, როცა ინიცირდება მისი საწყისი მოვლენა. მას შეუძლია მშობელი ქვეპროცესის შეწყვეტა ან მის პარალელურად შესრულება
გამომძახებე-ლი ქმედება	არის შესასვლელი წერტილი გლობალურად განსაზღვრული ქვეპროცესის, რომელიც ხელმეორედ გამოიყენება ამ პროცესში

ნახ. 2.5. ქმედებათა ცხრილი



ნახ. 2.6. მარკერების ცხრილი (Bizag გარემოში)

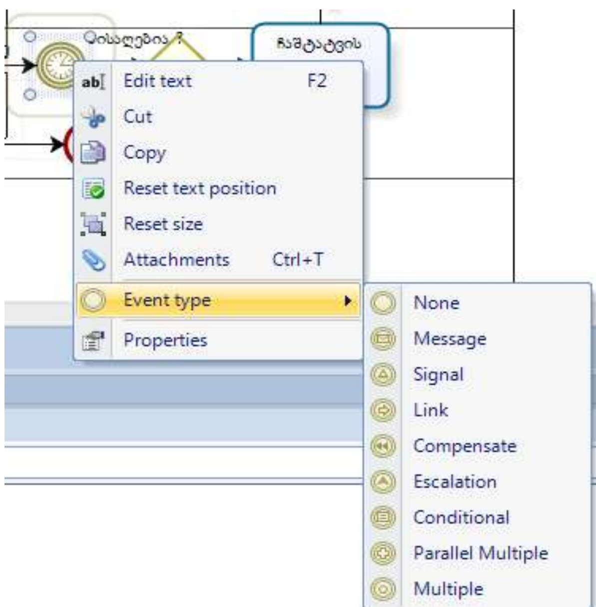
ლოგიკური ოპერატორები	
"ან" ოპერატორის გამორიცხვა მონაცემთა მართვისას (Data XOR)	ან
მოვლენითი "ან" ოპერატორი: ქმნის პროცესის ახალ ეგზემპლარს (Event XOR)	
"და" ოპერატორი (AND)	
"ან" ოპერატორი (OR): განშტოებისას აქტიურდება ერთი ან ყველა შტო, შერწყმისას ყველა მოქმედი შემავალი შტო იხურება	
რთული ოპერატორი, ამოდელირებს განშტოების და შერწყმის რთულ პირობებს	
მოვლენითი "და" ოპერატორი (ქმნის პროცესის ახალ ეგზემპლარს)	

ნახ. 2.7. ლოგიკური ოპერატორების ცხრილი

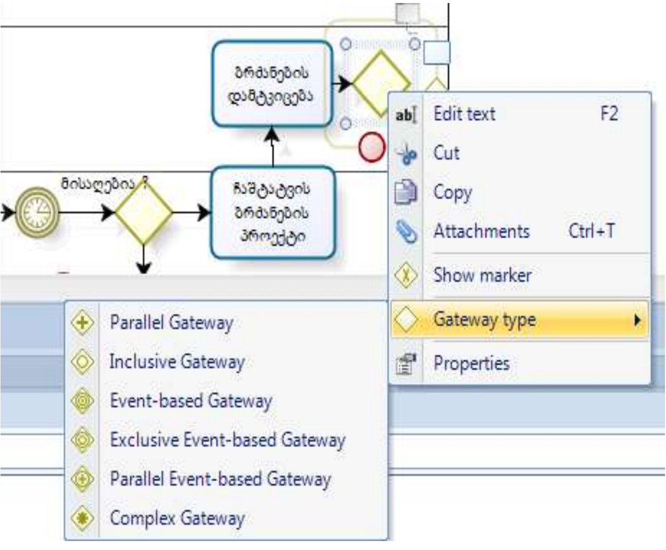
მოვლენები

მოვლენათა ტიპები	დაწყება	შუალედური	დასასრული
ჩვეულებრივი			
შეტყობინება			
წამზომი			
შეცდომა			
შეწყვეტა			
კომპენსაცია			
ბიზნესწესის შესრულება			
ბმული			
სიგნალი			
შედგენილი (სრულდება ერთი)			
პარალელური შედგენილი (სრულდება ყველა)			
ესკალაცია: საკითხის ატანა ორგ–იერარქიის ზედა დონეზე			
შეჩერება			

ნახ. 2.8. მოვლენების ცხრილი



ნახ. 2.9. მოვლენის ტიპის შერჩევა (Bizag გარემოში)



ნახ. 2.9. ლოგიკური ოპერატორის ტიპის შერჩევა
(Bizag გარემოში)

2.2. ობიექტ-ორიენტირებული, უნიფიცირებული მოდელირების ენა (UML)

მოდელირების უნიფიცირებული ენა – UML (Unified Modeling Language) შექმნილია, როგორც უნივერსალური მოდელირების ენა ობიექტორიენტი-რებული პროგრამირების სფეროში და არის სტანდარტული ვიზუალური მოდელირების ენა, რომელიც იძლევა საშუალებას სისტემა აღიწეროს გრაფიკულად და ტექსტურად [13,14].

როგორც აღვნიშნეთ, UML პრაქტიკულად არის გრადი ბუჩის მეთოდის (Booch Method), ობიექტის მოდელირების ტექნიკის (Object-modeling technique – OMT) და ობიექტორიენტირებული პროგრამული უზრუნველყოფის ინჟინერიის (Object-oriented software engineering - OOSE) სინთეზი და გვევლინება როგორც ერთი, საერთო და ფართო გამოყენების მოდელირების ენა [1].

UML არის დიდი პროექტების ფართოდ გამოყენებადი უნიფიცირებული მოდელირების ენა. იგი შექმნილია საერთაშორისო ასოციაციის, ობიექტების მართვის ჯგუფის – OMG (Object Management Group) მიერ [27], ქმნის ღია სტანდარტებს ობიექტორიენტირებული აპლიკაციებისათვის. წარმოადგენს გრაფიკულ ენას და გამოიყენება ოო-მოდელირების აღწერისათვის პროგრამული უზრუნველყოფის სფეროში. აერთიანებს მონაცემთა მოდელირების (Entity Relationship Diagrams), ბიზნესნაკადების (Workflows), ობიექტების და კომპონენტების მოდელირების მეთოდებს. გამოიყენება პროგრამული უზრუნველყოფის და მისი ტექნიკური რეალიზაციის მთელი სასიცოცხლო ციკლის განმავლობაში. UML-ის მიზანია იყოს სტანდარტული მოდელირების ენა, რომლის საშუალებითაც შესაძლებელია განაწილებული სისტემების მოდელების შექმნა.

არის რამდენიმე მიზეზი, რატომ უნდა გამოვიყენოთ UML ენა მოდელი-რებისთვის:

– უნიფიცირებული ტერმინოლოგიის და მნიშვნელობათა სტანდარტიზაციის საშუალებით საგრძნობლად გამარტივებულია ურთიერთობა მოდელირებადი სისტემის სხვადასხვა მხარეს შორის. ეს აადვილებს მოდელების გაცვლას სხვადასხვა დეპარტამენტსა და კომპანიას შორის, განსაკუთრებით პროექტების გადაგზავნას საპროექტო ჯგუფებს შორის;

– მოდელირებაზე მოთხოვნის გაზრდასთან ერთად ვითარდება UML-ენაც. ვინაიდან UML არის ბიზნესპროცესების მოხერხებული ასახვის ენა. ჩვენ

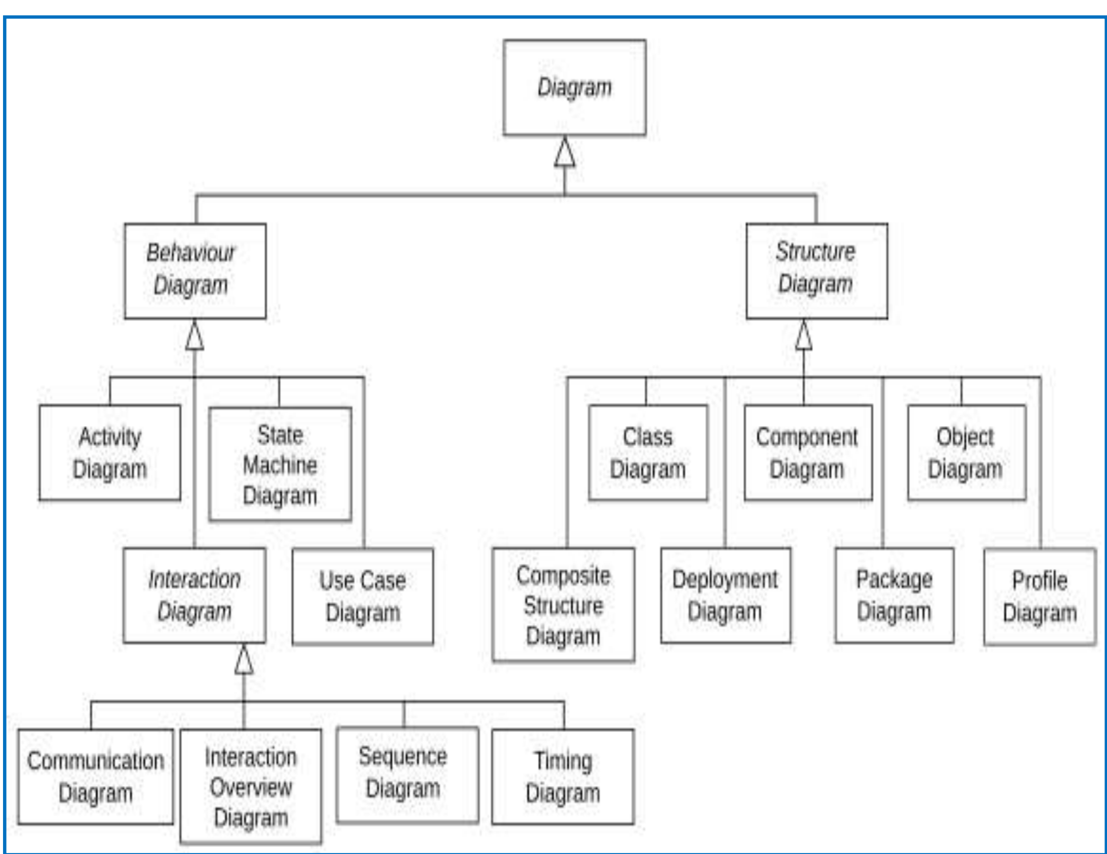
შეგვიძლია მისი საშუალებით შევქმნათ, როგორც მარტივი სისტემის მოდელები, ისე კომპლექსური სისტემების დაწვრილებითი მოდელები და თუ UML–ის ფუნქციური შესაძლებლობები არასაკმარისი იქნება, მაშინ ჩვენ მის გაფართოებას სტერეოტიპების საშუალებით შევძლებთ;

– UML დაფუძნებულია ფართოდ გამოყენებად მიღწევებზე. იგი მოდელირების არსებული ენების გამოყენებით რეალური პრობლემების გადასაჭრელად შემუშავდა, რაც უზრუნველყოფს მის მოხერხებულობას და პრაქტიკაში გამართულ ფუნქციონირებას.

UML-ის განვითარება. UML პირველად 1997 წელს გამოჩნდა. მას შემდეგ UML1-ის სხვადასხვა ვერსიები გამოდიოდა 2005 წლის ჩათვლით. გაფართოვდა და დაიხვეწა აქტიურობისა და მიმდევრობითობის დიაგრამები. კლასები გაფართოებულია შიგა სტრუქტურებით და პორტებით ე.წ. კომპოზიციური სტრუქტურებით. დაემატა ინფორმაციული ნაკადები და სხვ. მას შემდეგ UML2-ის სხვადასხვა ვერსია გამოვიდა: UML2.1-UML2.5, FTF_Beta1 სახელწოდებით [28]. OMG-მ დღესდღეობით UML2.0 ვერსიის სტანდარტიზაცია დაასრულა. ახალი UML2.0 სპეციფიკაცია UML-ისთვის შეიცავს ისეთ სრულყოფილ სიახლეებს, რომელიც რესტრუქტურიზაციას უკეთებს და ხვეწს ენას, რათა ის უფრო ადვილი გამოსაყენებელი, შესასრულებელი და გასაგები გახადოს.

რაც შეეხება მთლიანად UML2-ს, UML1-ისგან განსხვავებით, აქ შემუშვებულია ახალი დიაგრამები [28]: ობიექტების დიაგრამა (object diagrams), პაკეტების დიაგრამა (package diagrams), სტრუქტურის შემადგენლობის დიაგრამა (composite structure diagrams), ურთიერთქმედების მიმოხილვის დიაგრამა (interaction overview diagrams), დროითი (სონქრონიზაციის) დიაგრამა (timing diagrams), პროფილების დიაგრამა (profile diagrams), ხოლო კოოპერაციის დიაგრამა (collaboration diagrams) გვხვდება კომუნიკაციის დიაგრამის (communication diagrams) სახელით. მოხდა აქტიურობათა დიაგრამის (activity diagrams) და მიმდევრობითობის დიაგრამის (sequence diagrams) გაფართოება.

UML/2.6.1-ის დადებითი მხარეებია: ახალი სტრუქტურა; არქიტექტურული მოდელირების კონსტრუქციები; პორტები, კავშირები და ნაწილები; ახალი UML2-დიაგრამები; სტრუქტურის შემადგენლობის დიაგრამა; ქცევის დიაგრამების UML2.5-განახლება. მათ შორის მდგომარეობათა დიაგრამის; ურთიერთქმედების დიაგრამის; აქტიურობათა დიაგრამის და ა.შ. აქვს 15 ტიპის დიაგრამა, რომელიც იყოფა 2 კატეგორიად (ნახ.2.10) [27].



ნახ.2.10. UML/2.6.1-ის სტრუქტურა

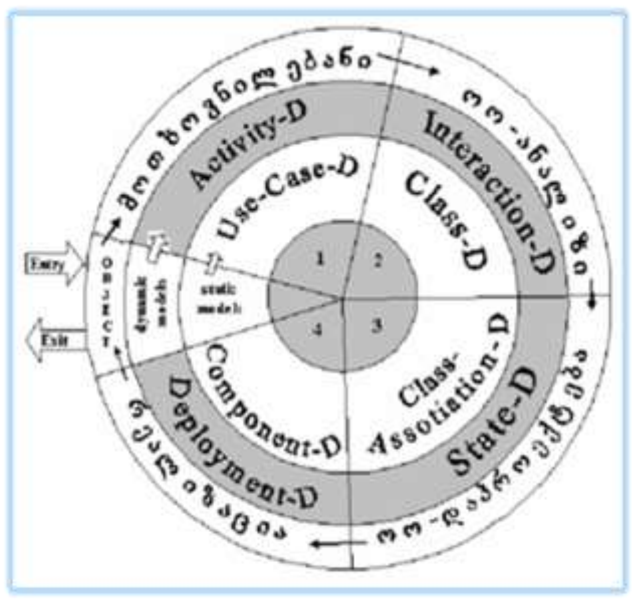
აქედან 7 დიაგრამა გვაძლევს სტრუქტურულ ინფორმაციას, ხოლო დანარჩენი 8 ქცევის საერთო ტიპს. მათ შორის 4 სახის დიაგრამა ასახავს ურთიერთქმედების სხვადასხვა ასპექტს. ამ დიაგრამების სტრუქტურა მოცემულია მე-3 თავში.

ამჯერად განვიხილავთ UML-ის კლასიკურ - პირველ ვარიანტს.

საინჟინრო ხაზვის ტრადიციის მიხედვით UML დიაგრამაში შესაძლებელია გამოყენების შესახებ კომენტარის და შენიშვნის მითითება, ასევე შეზღუდვის ან მიმართულების ჩვენება. დიაგრამების სტრუქტურა ახდენს იმის ხაზგასმას, რაც წამოდგენილი უნდა იყოს სისტემაში, რომლის მოდელირებასაც ვახორციელებთ.

2.2.1. UML-ის კლასიკური დიაგრამები

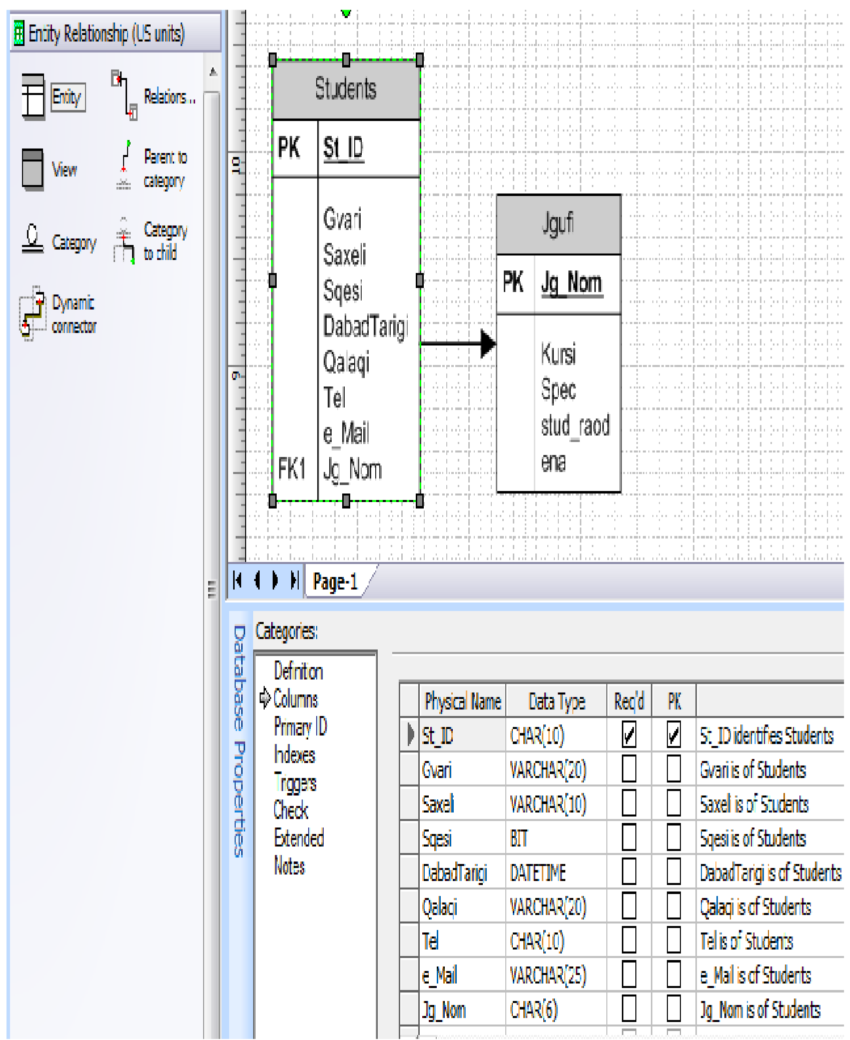
2.11 ნახაზზე ნაჩვეებია UML/1 ტექნოლოგიის კლასიკური მოდელი 4 წყვილი (ეტაპის) დიაგრამით. მათგან 4 სტატიკური მოდელია და 4 - დინამიკური. UML მოდელის ასეთი ინტერპრეტაცია შემუშავებულ იქნა სტუ-ს ინფორმატიკის ფაკულტეტის პროფესორის, გ. სურგულაძის მიერ 2000-2001 წლებში ბერლინის ჰუმბოლდტის უნივერსიტეტში, DAAD-ის გრანტის საფუძველზე [17, 30]. პროექტის თემა იყო „Object-oriented Programming with the UML Methodology, Linux Platform "Suse" and Languages C++/Java.



ნახ. 2.11. UML მეთოდოლოგიის კლასიკური მოდელი [30]

ახლა განვიხილოთ UML ენის ინსტრუმენტული საშუალებები, რომლებიც ფართოდ გამოიყენება დიდი პროგრამული პროექტების განხორციელების მიზნით. შეიძლება რამდენიმე ძირითადი ჩამოვთვალოთ, ვინაიდან CASE ტექნოლოგიების და, განსაკუთრებით, UML-ენის სპექტრი საკმაოდ დიდია. ქვემოთ მოყვანილი მაგალითები აგებულია Ms Visio და Enterprise Architect ინსტრუმენტებით [31, 37].

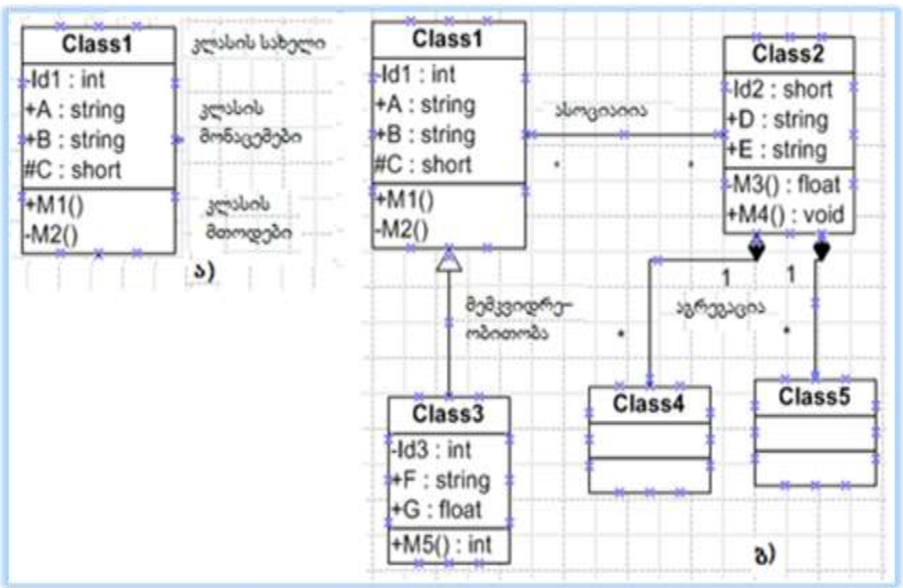
➢ *ER დიაგრამის* ვიზუალური აგება: Database Model Diagram არჩევით ეკრანზე გამოიტანება მონაცემთა ER-მოდელის ასაგები რედაქტორი (Entity-Relationship Model). 2.12 ნახაზზე ნაჩვენებია ინტერფეისი კონკრეტულ „სტუდენტთა-ჯგუფის“ მონაცემთა მოდელისათვის.



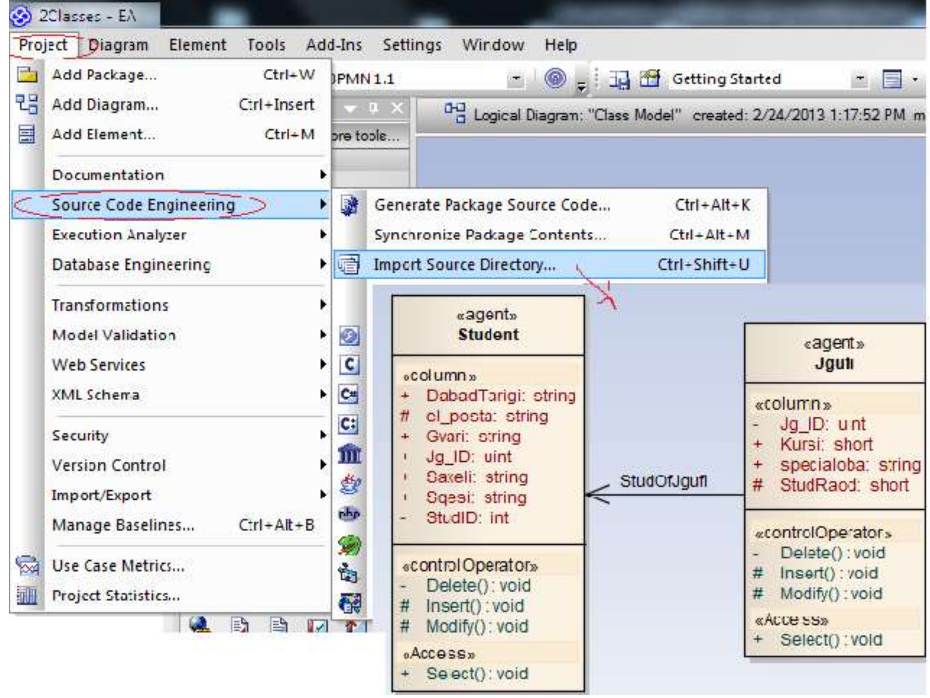
ნახ. 2.12. Database Model Diagram –ის აგება

➢ *კლასთა-ასოციაციის დიაგრამა:* კლასის მეთოდები (ან ფუნქციები) ის პროგრამული მოდულებია, რომლებიც ამუშავებს ამ კლასის მონაცემებს. მათი ინიციალიზაცია ხდება გარედან შემოსული შეტყობინების საფუძველზე.

განსაკუთრებით მნიშვნელოვანია კლასთა-ასოციაციის დიაგრამის აგება და შემდეგ პროგრამული კოდის გენერაცია (ნახ.2.13-15). სხვა დიაგრამებს აქ აღარ შევეხებით, რადგან ისინი ლიტერატურულ წყაროებში მრავლადაა მოცემული.



ნახ.2.14. კლასი (ა) და 2.15. კლასთა დიაგრამა (ბ)



ნახ.2.16. Student და Jgupi კლასების მომზადება „Code Engineering“ პროცესისთვის

2.2.2. კლასთაშორის კავშირების სისტემა

2.14 ნახაზზე ასახული კლასთაშორისი კავშირები შეიძლება ასე დავახასიათოთ:

- მემკვიდრეობითი (Generalization) ასახავს „გენეტიკურ“, კლასებს შორის განზოგადებულ კავშირებს. ასეთ დროს ერთი კლასი („შვილი“) მთლიანად იღებს მეორე კლასის („მშობელი“) ყველა ატრიბუტს, მეთოდსა და კავშირს;
- აგრეგაციული (Aggregation) არის კავშირი „მთელი–ნაწილი“. მაგალითად, „ავტომობილი“ – „ძარა, ძრავი, საბურავები” და ა.შ.;
- ასოციაციური (Assotiation) ნიშნავს სემენტიკურ კავშირს კლასებს შორის. ის შეიძლება გამოისახოს ერთ- ან ორმიმართულებიანი (იგივეა, რაც უისრო) ხაზით. ისარი გვიჩვენებს შეტყობინების გადაცემის მიმართულებას. ასოციაციური კავშირის რეალიზება ხდება ერთ კლასში დამატებით მეორე კლასის ატრიბურტის ჩასმით. ეს ჰგავს პირველადი (Primary) და მეორეული გასაღებური ატრიბუტების შეერთებას;
- რელაციური (Dependency) ნიშნავს ერთი კლასის დამოკიდებულებას მეორეზე. იგი ერთმიმართულებიანი წყვეტილი ისრით გამოიხატება. მასში დამატებითი დამაკავშირებელი ატრიბუტები არ გამოიყენება;

➢ *კლასთა დიაგრამიდან კოდის გენერაცია:* თანამედროვე CASE-ტექნოლოგიები, რომლებიც სისტემების *დაპროგრამების ავტომატიზაციაზეა* ორიენტირებული, მაგალითად, Rational Rose, Visual Paradigm, Enterprise Architect და მრავალი სხვა, ახორციელებს რევერსული დაპროგრამების კონცეფციას [1,31]. ანუ კლასების დიაგრამიდან შესაძლებელია პროგრამული კოდის გენერაცია და პირიქითაც, კოდიდან აიგება ავტომატურად გრაფიკული დიაგრამა [32]. ამ საკითხს დეტალურად შემდეგ პარაგრაფში განვიხილავთ.

2.15 ნახაზზე მოცემულია კოდის გენერირების პროცედურა Enterprise Architect ინსტრუმენტის სამუშაო გარემოში, ხოლო 2.16 ნახაზზე კი ილუსტრირებულია გენერირებული კოდის ფრაგმენტი.

დასასრულ, შეიძლება აღინიშნოს, რომ UML-ტექნოლოგიის გამოყენება თავისი ინსტრუმენტული საშუალებებით აუცილებელი და მეტად ეფექტურია დიდი პროექტების შესასრულებლად, სადაც განსაკუთრებული ყურადღება ექცევა საბოლოო პროგრამული პროდუქტის ხარისხს, შესრულების საიმედოობას და პროექტის ვადები გათვლილია შედარებით ხანგრძლივ პერიოდზე.

2Classes - EA   Student.cs

Student
- DabadTarigi
- Gvari
- Jg_ID
- Saxeli
- Sqesi
- StudID
- el_posta
- Delete()
- Dispose()
- Insert()
- Modify()
- Select()
- Student()
- ~Student()

```
1 ///////////////////////////////////////////////
2 //   Student.cs
3 //   Implementation of the Class Student
4 //   Generated by Enterprise Architect
5 //   Created on:        24-Feb-2013 2:26:10 PM
6 //   Original author: user
7 ///////////////////////////////////////////////
8 public class Student {
9     public string DabadTarigi;
10    protected string el_posta;
11    public string Gvari;
12    public uint Jg_ID;
13    public string Saxeli;
14    public string Sqesi;
15    private int StudID;
16    public Student(){    }  // constructor
17    ~Student(){ }   // destructor
18    public virtual void Dispose(){   }
19    private void Delete(){
20      // . . . code-1
21    }
22    protected void Insert(){
23      // . . . code-2
24    }
25    protected void Modify(){
26     // . . . code-3
27    }
28    public void Select(){
29     // . . . code-4
30    }
31 }//end Student
```

ნახ.2.17. C#-კოდის ლისტინგი Student კლასისათვის (Enterprise Architect)

2.2.4. რევერსული ტექნოლოგია: „კლასი-კოდი-კლასი“

ბოლო ათწლეულში განსაკუთრებული ყურადღება ექცევა მართვის საინფორმაციო სისტემების დაპროექტების და პროგრამული რეალიზაციის პროცესში MVC (Model-View-Controller) კონცეფციის გამოყენებას [33].

ეს იდეა ჯერ კიდევ წინა საუკუნის 80-იანი წლებისთვის იქნა შემოტანილი ნორვეგიელი მეცნიერის, ოსლოს უნივერსიტეტის პროფესორის ტრიგვე რინსკაუგის მიერ (Trygve Reenskaug), SmalTalk ენაზე მუშაობის დროს (იხ. ნახ.2 - შესავალი) [34].

მოკლედ თუ ვიტყვით, ეს იყო პროგრამული უზრუნველყოფის შექმნის გრაფიკული ინტერფეისი მომხმარებლებისთვის (GUI), დეველოპერებისთვის.

შემდგომ, თითქმის ყველა მულტიპარადიგმულ ენებმა (C++, Java,, C#, Python და სხვ.) უზრუნველყვეს ამ იდეის პრაქტიკული რეალიზაცია თავიანთ პლატფორმებზე, რამაც მნიშვნელოვანი გავლენა მოახდინა ვალიდური გამოყენებითი აპლიკაციების სწრაფი და ხარისხიანი შექმნის პროცესების სრულყოფისათვის.

წინამდებარე პარაგრაფში ჩვენ განვიხილავთ კონკრეტულ ამოცანას, რომელიც ეხება „კლასი-კოდი-კლასი" რევერსული მექანიზმის გამოყენებას Visual Studio.NET 2022 პლატფორმაზე [32]. („კლასი" აქ მოდელია, „კოდი" - C#).

გამოყენებითი საპროლემო სფეროს მომხმარებლების (მაგალითად, ბიზნეს ანალიტიკოსები) ან ნაკლებკვალიფიციური პროგრამისტ-დეველოპერების მიერ შესაძლებელი ხდება პროგრამული პროდუქტების სწრაფად რეალიზაცია და ტესტირება. მასალის ექსპერიმენტული ნაწილი შესრულებულია VS.NET Framework პლატფორმაზე *უნივერსიტეტის საპრობლემო სფეროს* მარტივ მაგალითზე, თუმცა შემდეგ შესაძლებელია ამოცანის გართულება მისი გაფართოებით, VS.NET Core პლატფორმაზე გადასვლა და სრული MVC მოდელის რეალიზაცია.

ამოცანა: შევქმნათ დესკტოპ-პროგრამული პროექტი კლასებით: Student, Lector, Group, Acad_course და ა.შ., რა თქმა უნდა კლასთა ურთიერთკავშირებით: Inheritance, Association, Agregation და ა.შ.

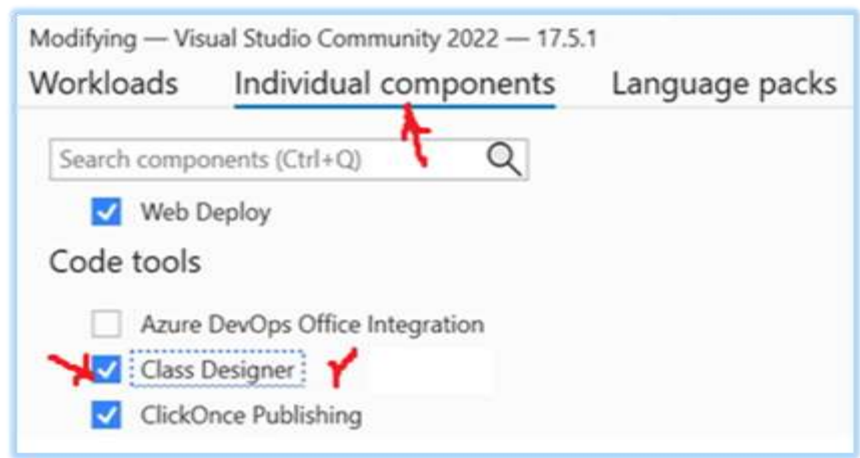
საპროექტო სისტემის საბოლოო მიზანი უნივერსიტეტის ფაკულტეტების, დეპარტამენტების, სტუდენეტებისა და ლექტორების შესაბამისი ინტერფეი-სების აგებაა, რომლებიც დაკავშირებულია მონაცემთა ბაზის კლასიკურ SQL-ცხრილებთან (ან NoSQL - კოლექციებთან, MongoDB Compaas მაგალითზე) [33].

***ბიჯი_1)** VS.NET პლატფორმის Class Designer რესურსის მომზადება:*

ავამუშავოთ Visual Studio.NET 2022 პაკეტი და აპლიკაციის ტიპის არჩევის საწყისი ფანჯრის ბოლოში ავირჩიოთ ლინკი (ნახ.2.17)

ნახ. 2.18. Installer-ში გადასვლა

შედეგად გადავდივართ ახალ გვერდზე (ნახ. 2.18). აქ Class Designer-ის ჩეკბოქსს ჩავრთავთ. მისი დანიშნულებაა, მაგალითად, UML-კლასების დიაგრამის მოდელირება, რომლის დახმარებითაც განვახორციელებთ „მოდელი-კოდის“ (ან პირიქით) რევერსულ პროგრამირებას.



ნახ. 2.19. Class Designer ინდივიდუალური კომპონენტის დაინსტალირება

ბიჯი_2) *VS.NET პლატფორმაზე Console App (.Net Framework) არჩევა და პროექტის შექმნა:*

პროგრამული პროექტის სახელი დავაფიქსირეთ: ConsoleApp_Revers. მიიღება საწყისი სურათი Solution Explorer-ით (ნახ. 2.19).

პროექტის სახელზე კონტექსტური მენიუს გამოტანით ვირჩევთ Add -> Class და ვარქმევთ შესაბამისი კლასის სახელს (მაგალითად, Student) (ნახ. 2.20).

ამ კოდში კლასის ხილვადობა Internal შევცვალოთ public-ით. შემდეგ კლასის კოდში შევიტანოთ პროგრამულად ატრიბუტები:

- St_ID,
- Name,
- first_Name,
- Gender,
- Age და
- Nr_Gr.

კოდი მიიღებს ასეთ სახეს (ნახ. 2.21).

```
Program.cs
ConsoleApp_Revers   ConsoleApp_Revers.Pr   Main(string[] args)
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace ConsoleApp_Revers
8   {
        0 references
9       internal class Program
10      {
            0 references
11          static void Main(string[] args)
12          {
13          }
14      }
15  }
16
```

```
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'ConsoleApp_Revers' (1 of 1 projec
  ConsoleApp_Revers
    Properties
    References
    App.config
    Program.cs
Solution Explorer   Git Changes
Properties
```

ნახ. 2.19. პროექტის საწყისი სტრუქტურის (Solution Explorer) და C# კოდის შექმნა

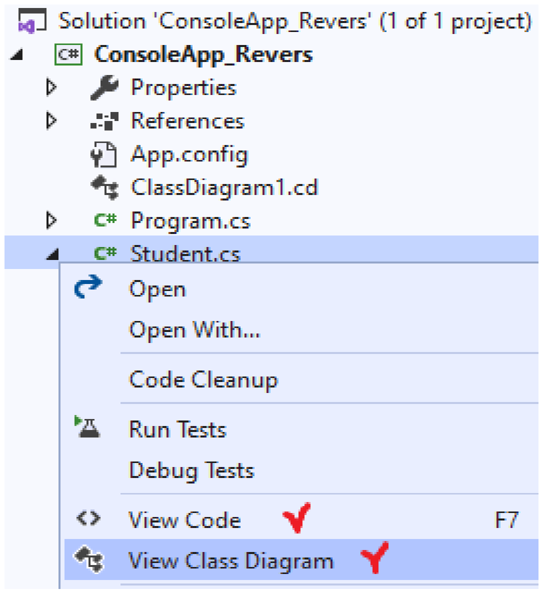
```
Student.cs   Program.cs
ConsoleApp_Revers   ConsoleApp_Revers.Student
1   using System;
7   namespace ConsoleApp_Revers
8   {
        0 references
9       internal class Student
10      {
11      }
12  }
```

ნახ. 2.19. Student კლასის შექმნა

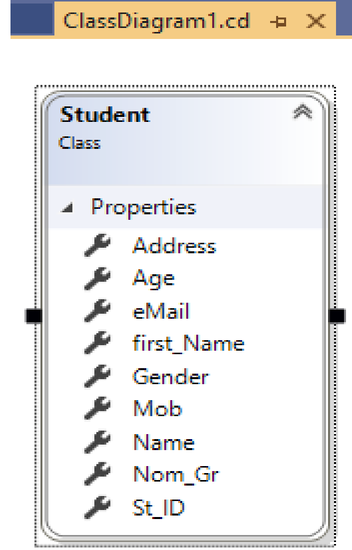
```
7   namespace ConsoleApp_Revers
8   {
        0 references
9       public class Student
10      {
            0 references
11          public int St_ID { get; set; }
            0 references
12          public string Name { get; set; }
            0 references
13          public string first_Name { get; set; }
            0 references
14          public string Gender { get; set; }
            0 references
15          public int Age { get; set; }
            0 references
16          public int Nom_Gr { get; set; }
17      }
18  }
```

ნახ. 2.20. C# კოდი Student კლასისთვის

Solution Explorer-ში Student-კლასის კონტექსტური მენიუდან კლასების დიაგრამის გამოსატანად ვირჩევთ View Class Diagram-ს (ნახ. 2.21, 22).

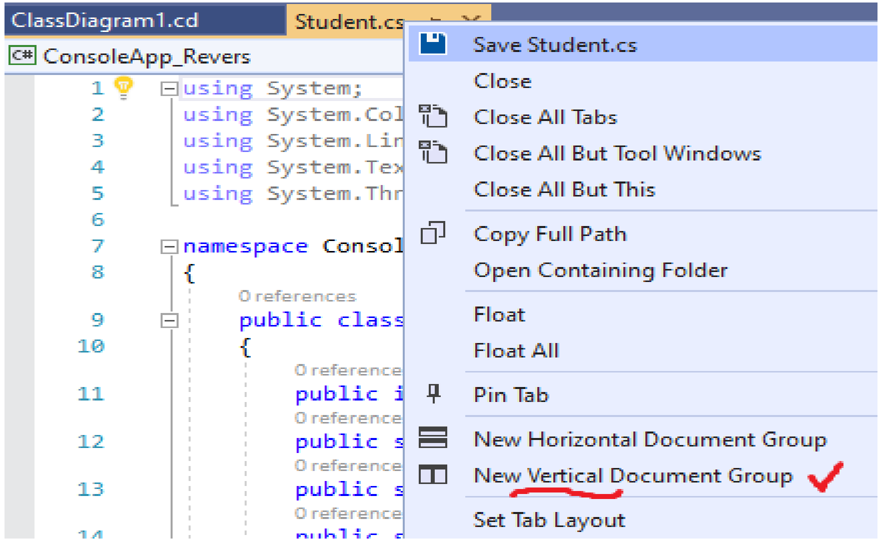


ნახ. 2.21



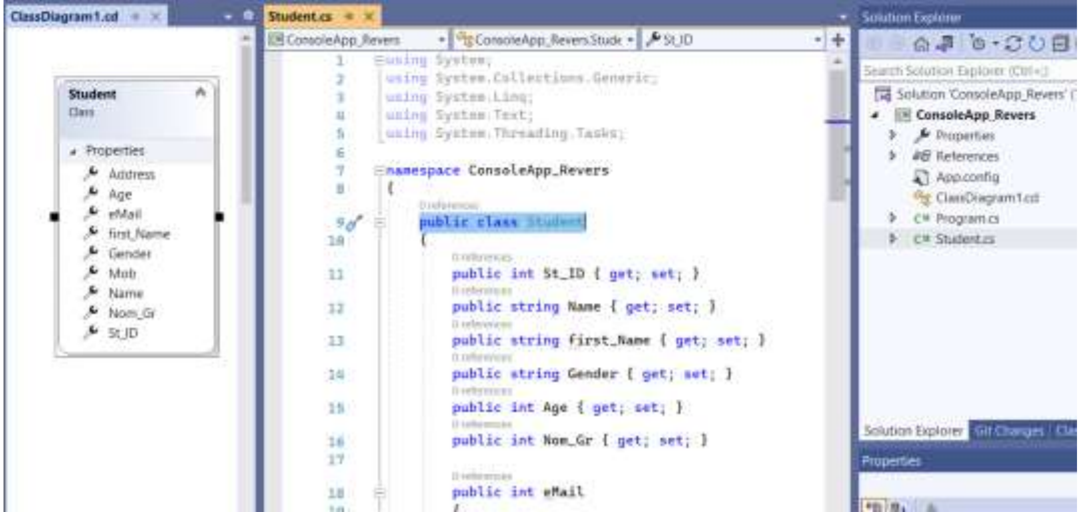
ნახ. 2.22

Student-კლასის დიაგრამის და მისი C# კოდის ერთდროულად ხილვადობისათვის ვიყენებთ კონტექსტ-მენიუს ჩანართს New Vertical Document Group (ნახ. 2.23).



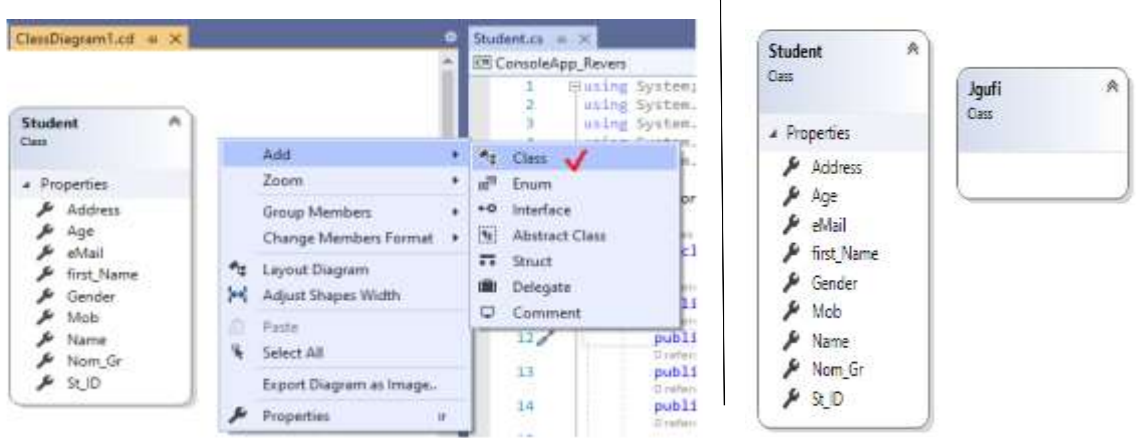
ნახ. 2.23

შედეგად მიიღება 2.24 ნახაზზე ნაჩვენები ფორმა. სწორედ აქ არის შესაძლებელი ჩვენი ამოცანის გაფართოება ახალი კლასებით, ახალი თვისებებით (ატრიბუტებით) და ა.შ., ანუ რევერსული თვალსაზრისის საფუძველზე თუ შევცვლით კლასთა დიაგრამას (მარცხენა ფანჯარაში), მყისიერად შეიცვლება პროგრამა (მარჯვენაში). ასევე, თუ პროგრამულ ნაწილში ჩავამატებთ ახალ კლასს თავისი ატრიბუტებით, ან შევცვლით არსებულს, მაშინვე კლასთა დიაგრამაზე აისახება ეს ცვლილებები.



ნახ. 2.24

მაგალითად, დავამატოთ მოდელურ ნაწილში კლასი Jgufi (ნახ. 2.25).



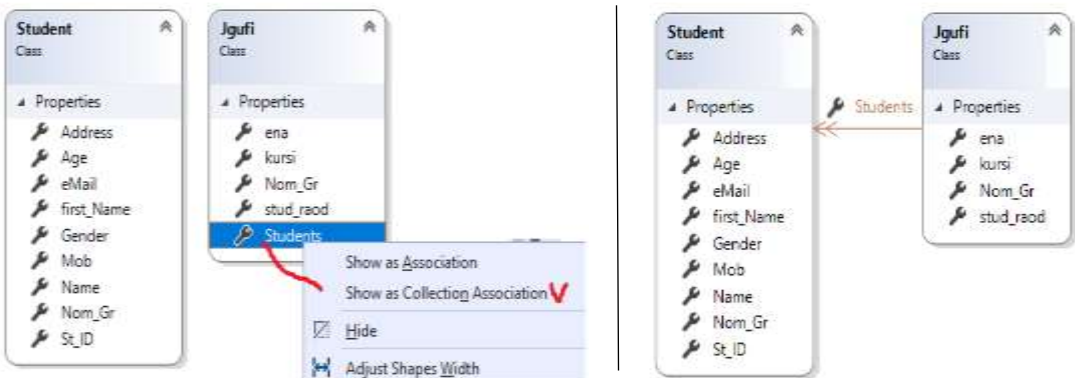
ნახ. 2.26. კლასთა დიაგრამაზე გამოჩნდა ახალი კლასი Jgufi

კლასში Jgufi (დიაგრამაზე) ჩავამატოთ შესაბამისი ატრიბუტები (Properties - თვისებები): Nom_Gr, kursi, ena და stud_raod. კოდის ცვლილება მყისიერად ისახება კლასების დიაგრამაზე და პირიქით, კლასის დიაგრამაზე თვისების ცვლილება ავტომატურად ანახლებს კოდს.

კოდში ჩავამატოთ Student და Jgufi კლასებს შორის ასოციაციური კავშირისთვის სტრიქონი:

```
public ICollection<Student> Students { get; set; }
```

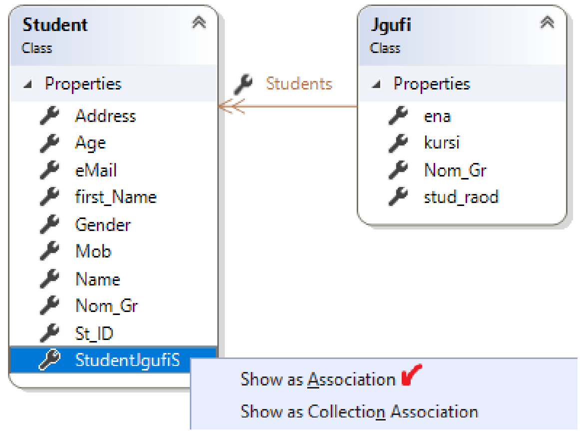
Jguf-ის მოდელში გამოჩნდება თვისება Students. ამ თვისებაზე კონტექ-სტური მენიუს გამოტანით ავირჩევთ Show as Collection Association და კავშირის შედეგიც სახეზეა (ნახ. 2.26).

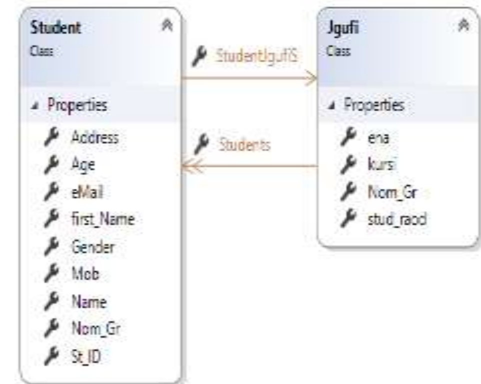


ნახ. 2.27. კლასთაშორის ასოციაციური კავშირის აგება (1)

ახლა Student კლასის კოდში ჩავამატოთ სტრიქონი თვისებისთვის StudentJgufiS (ნახ. 2.27):

```
public Jgufi StudentJgufiS { get; set; }
```





ნახ. 2.28. კლასთაშორის ასოციაციური კავშირის აგება (2)

მარჯვენა ნახაზზე აისახა კლასთაშორის ასოსიაციური კავშირი, რომელც მოგვაგონებს მონაცემთა ბაზის ცხრილებს შორის 1:N დამოკიდებულების შემთხვევას.

კლასის თვისებების შეცვლა შესაძლებელია როგორც დიაგრამიდან (მოდელიდან), ასევე კოდიდან და კლასის დეტალების (Class Details) ფანჯრიდან.

ამგვარად, დასკვნის სახით შეიძლება აღვნიშნოთ, რომ დიდი პროგრამული პროექტების აგების პროცესში ძალზე ეფექტურია CASE ტექნოლოგიების გამოყენება, განსაკუთრებით უნიფიცირებული პროცესების მოდელირებისას ობიექტ-ორიენტირებული მიდგომის საფუძველზე.

ექსტრემალური პროგრამირების ან Scrum მეთოდების გამოყენებისას დროის ფაქტორი მნიშვნელოვანია, ამიტომ აქ UML დიაგრამების სრული გამოყენება არ ხდება.

მხოლოდ ბიზნესპროცესის (Activity-D) ან კლასების დიაგრამის (Class-D) გამოყენებაა რეკომენდებული (სემანტიკური მიზნებისთვის), რაც კომპრომისულ გადაწყვეტილებად ითვლება [2].

მაიკროსოფთის კორპორაცია აქტიურად აფართოებს Visual Studio.NET Core პლატფორმის ფუნქციონალობას ამ მიმართულებით, რაც დესკტოპ- და ვებ-აპლიკაციების სწრაფ და ხარისხიან დეველოპმენტს უწყობს ხელს.

სწორედ .NET Core პლატფორმაზე იქმნება მრავალფუნქციური, სერვისებზე ორიენტირებული არქიტექტუის აპლიკაციები, მათ შორის ინტეგრირებული პროექტებიც, რომლებშიც ეფექტურად გამოიყენება Web-, Mobile- და Cloud-ტექნოლოგიები [35].

ასეთი კომპიუტერული სისტემების დეველოპმ,ენტი ნამდვილად ითვლება დღეს პრიორიტეტულ მიმართულებად.

თავი 3
მოქნილი პროგრამირების მეთოდოლოგია და მეთოდები

3.1. დაპროგრამების Agile მეთოდოლოგია და აპლიკაციების დეველოპმენტის მეთოდები

შედარებით მცირე ზომის და სწრაფად შესასრულებელი პროექტებისათვის, დამკვეთის მოთხოვნების დაკმაყოფილების თვალსაზრისითაც, უფრო მისაღებია ე.წ. „მოქნილი“ (Agile, სწრაფი), მაგალითად, ექსტრემალური პროგრამირების მეთოდის გამოყენება, რაზეც მომდევნო პარაგრაფში გვექნება საუბარი.

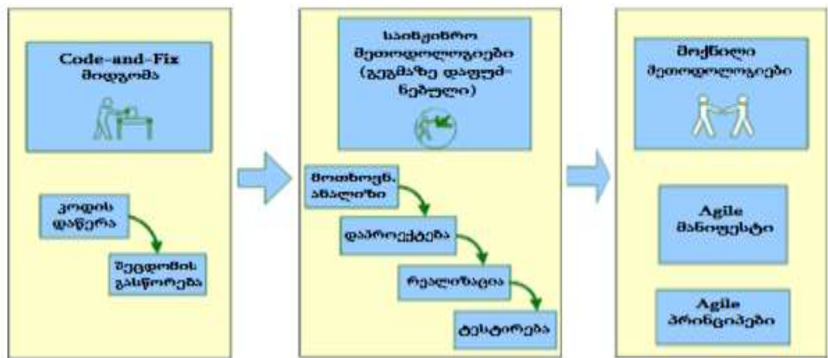
დაპროგრამების მოქნილი მეთოდოლოგია (Agile Software Methodology) განიხილავს განსხვავებულ Agile-მეთოდებს, როგორიცაა მაგალითად, Extreme Programming (XP), Scrum, Kanban/Lean, Agile Unified Process (AUP), Dynamic Systems Development Method (DSDM), Disciplined Agile Delivery (DAD), Feature-Driven Development (FDD), Test-Driven Development (TDD), Rapid Application Development (RAD) და Scaled Agile Framework (SAFe) [38, 39].

მოქნილი პროგრამირების კონცეფციებს, მათ პრინციპებს და ზოგიერთ მეთოდს აქ დეტალურად განვიხილავთ. განსაკუთრებით უფრო ფართოდ გამოყენებად (XP, Scrum) და ვიზუალური თვისებებით აღჭურვილ ეკონომიურ მეთოდებს (Kanban/Lean).

3.1.1. ექსტრემალური დაპროგრამების პრინციპები და ინსტრუმენტული საშუალებანი

2001 წ.ლის თებერვლიდან ოფიციალურად იქნა ჩამოყალიბებული პროგრამული სისტემების დამუშავების ახალი, Agile (მოქნილი) – მეთოდოლოგია [9].

ადრინდელი ქაოსური მიდგომების (code-and-fix) და მკაცრად ფორმალიზებული საინჟინრო მეთოდოლოგიების ნაცვლად განვითარება დაიწყო მოქნილი მეთოდოლოგების ოჯახმა, რომელიც დაფუძნებული იყო პროგრამული უზრუნველყოფის დამუშავების მანიფესტსა და მისი რეალიზაციის პრინციპებზე (ნახ.3.1) [1,40-42].



ნახ. 3.1. Agile მეთოდოლოგიებზე გადასვლა

ფორმალიზაციის ხარისხის მიხედვით მოქნილი მეთოდოლოგიები იკავებს შუალედურ ადგილს წინა ორ განხილულ მიდგომას შორის ანუ ისინი არც ისე მკაცრად ფორმალიზებულია, როგორც საინჟინრო მიდგომები და არც ქაოსური, უსისტემო მიდგომაა, როგორც code-and-fix [40].

Agile მეთოდოლოგიაში ჩადებული იდეები არც თუ ახალია. იტერაციული დამუშავება და ადამიანური ფაქტორის განსაკუთრებული როლი და სხვა იდეები, 2001 წლის თებერვლამდეც იყო ცნობილი, მაგრამ ამ დროიდან პირველად მოხდა პროგრესულ ფასეულობათა და პრინციპების ერთად შეკრება და რეკომენდაციების სახით შემოთავაზება, როგორც პრაქტიკულად შემოწმებული და საკმარისი ალტერნატივა პროგრამული სისტემების დამუშავების ტრადიციული მიდგომებისათვის [42].

3.1.2. პროგრამების მოქნილი დეველოპმენტის მანიფესტი და პრინციპები

მანიფესტი ოთხი პუნქტისაგან შედგება, რომელთაგანაც თითოეული ალტერნატივაა [42]. მის მარცხენა ნაწილში მოთავსებულია ცნებები და ასპექტები, რომლებსაც პროგრამული უზრუნველყოფის დამუშავებისას დიდი ღირებულება აქვს, ვიდრე მარჯვენა ნაწილში მოთავსებულს. მოქნილი მოდელირების ძირითადი კონცეფციები ასეთია:

- ადამიანები და ურთიერთობები უფრო მნიშვნელოვანია, *ვიდრე* პროცესები და ინსტრუმენტები;

- სამუშაო პროდუქტი უფრო მნიშვნელოვანია, *ვიდრე* ვრცელი-ამომწურავი დოკუმენტაცია;
- დამკვეთთან თანამშრომლობა უფრო მნიშვნელოვანია, *ვიდრე* კონტრაქ-ტით შეთანხმებული პირობები;
- მზადყოფნა ცვლილებებისადმი უფრო მნიშვნელოვანია, *ვიდრე* პირველ-საწყისი გეგმის დაცვა.

მოქნილი დამუშავების პრინციპები ბაზირებულია Agile მანიფესტის ფასეულობებზე, დეტალურად ხსნის და განავრცობს მათ მეტი პრაქტიკული თვისებების მქონე ინფორმაციით.

ეს პრინციპებია:

1) კლიენტის დაკმაყოფილება ღირებული პროგრამული უზრუნველყოფის ადრეული და უწყვეტი მიწოდების მეშვეობით;
2) მოთხოვნილებათა ცვლილებების მისალმება სამუშაოს ბოლო ეტაპზეც კი (ეს ზრდის საშედეგო პროდუქტის კონკურენტუნარიანობას);
3) სამუშაო პროგრამული უზრუნველყოფის ხშირი მიწოდება დამკვეთზე (ყოველთვიური, კვირეული ან უფრო ხშირი);
4) დამკვეთის ხშირი, ყოველდღიური კონტაქტი მიმწოდებელთან პროექტის შესრულების მთელ მანძილზე;
5) პროექტზე მუშაობენ მოტივირებული პირები, რომლებიც უზრუნველ-ყოფილია მუშაობის საჭირო პირობებით, მხარდაჭერითა და ნდობით;
6) ინფორმაციის გადაცემის რეკომენდებული მეთოდი – პირადი საუბრები (პირისპირ);
7) მომუშავე პროგრამული უზრუნველყოფა – პროგრესის საუკეთესო საზომია. პროექტების მიზანია პროგრამული სისტემის შექმნა და არა გეგმებისა და დოკუმენტაციის. აპლიკაციის მუშაობისუნარიანობის შეფასებით შეიძლება პროექტის პროგრესის ობიექტური გაზომვა;
8) სპონსორებს, მიმწოდებლებსა და მომხმარებლებს უნდა ჰქონდეთ მხარდაჭერის შესაძლებლობა მუდმივი ტემპის შესანარჩუნებლად გაურკვეველი ვადით;
9) მუდმივი ყურადღება ტექნიკური ოსტატობის (უნარების) სრულყოფას და მოსახერხებელ დიზაინს;

10) სიმარტივე – ხელოვნება ზედმეტი სამუშაოს შესრულების გარეშე. არაა საჭირო რთული უნივერსალური გადაწყვეტების მიღება, თუ ამის ცხადი აუცილებლობა არაა;

11) საუკეთესო ტექნიკური მოთხოვნები, დიზაინი და არქიტექტურა გამოსდის თვითორგანიზებულ გუნდს;

12) მუდმივი ადაპტაცია ცვალებად გარემოებებისადმი. იტერაციული სასიცოცხლო ციკლი ბაზირებულია მართვაზე უკუკავშირით, რომლის მნიშვნელოვანი ელემენტია შედეგების ანალიზი, უკუკავშირის განხორციელება და პროცესის სრულყოფა.

მოქნილი დამუშავების მანიფესტი და პრინციპები მოიცავს მაღალი დონის კონცეფციებს იმის შესახებ თუ როგორ უნდა განხორციელდეს პროგრამული უზრუნველყოფის დამუშავების პროცესი, რათა წარმატებით დასრულდეს პროექტი, შეიქმნას სამუშაო გუნდები, რომლებშიც სასიამოვნო და საინტერესო იქნება მუშაობა. ეს დოკუმენტები აღწერს, რა უნდა გაკეთდეს ამისთვის, მაგრამ არაფერს ამბობს, როგორ უნდა გაკეთდეს.

3.1.3. მოქნილი (სწრაფი) მოდელირება (Agile Modeling)

Agile Modeling (AM) – არის პროგრამული უზრუნველყოფის (Software) შექმნის სპეციალისტების ერთობლივი მუშაობის ეფექტური ორგანიზების ხერხი დამკვეთების მოთხოვნილებათა დასაკმაყოფილებლად.

მოქნილი მეთოდებით პროგრამული სისტემების დამუშავების სპეციალისტები ქმნიან ერთ გუნდს დამკვეთთან ერთად, რომლის წარმომადგენლებიც უშუალოდ და აქტიურად მონაწილეობენ სისტემის ანალიზის, დაპროექტებისა და აგების პროცესებში. AM-გუნდის მუშაობის მთავარი მიზანია ეფექტურობა, დამკვეთის მეტი წვლილის ჩადება საბოლოო პროდუქტში, შეძლებისდაგვარად მარტივი მოდელების აგება, *სამუშაო სისტემის შექმნა და არა თეორიის* ! [41].

ამგვარად, მოქნილი მოდელირება – პროფესიონალთა გუნდის მუშაობის ეფექტურობის ამაღლების მეთოდოლოგიაა პროგრამული უზრუნველყოფის შესაქმნელად.

AM ითვალისწინებს აგრეთვე *CASE-საშუალებების* ზომიერად გამოყენე-ბასაც, თუკი ამით ეფექტურობა მაღლდება. ინსტრუმენტული საშუალებებიდან შეირჩევა უმარტივესი, რომელიც დასმული ამოცანის გადაწყვეტის საშუალებას იძლევა.

3.1.4. მოქნილი მოდელირების ფასეულობანი

AM-ის ფასეულობებია [43-45]: ურთიერთობა (კომუნიკაცია), სიმარტივე, უკუკავშირი, გამბედაობა და თავმდაბლობა. აქედან პირველი ოთხი „ექსტრემალური დაპროგრამების" მამას, კენტ ბეკსაც ჰქონდა მოცემული (2000 წ. XP) [41]. Agile-კონსორციუმის პრეზიდენტმა სკოტ ამბლერმა (Scott W. Ambler) მეხუთე დაამატა. განვიხილოთ თითოეული მათგანი მოქნილი მოდელირების ჭრილში.

- **კომუნიკაცია ურთიერთობა**

ტერმინოლოგიური ლექსიკონის მიხედვით ესაა „პროცესი, რომლის დროსაც ხდება ინფორმაციის გადაცემა ერთი პირიდან მეორეზე, არსებული სიმბოლოების, ნიშნების ან ქმედებების საერთო სისტემის საშუალებით".

პროგრამული პროექტის შესრულების დროს განსაკუთრებული მნიშვნელობა აქვს ურთიერთობას პროექტის მონაწილეებს შორის, კერძოდ, ეფექტურ კომუნიკაციას დამპროექტებლებს, პროგრამისტებსა და დამკვეთებს შორის.

პროექტის პრობლემები ჩნდება იქ, სადაც ურთიერთობა შეწყვეტილია. მაგალითად, როდესაც დეველოპერი არ ეუბნება კოლეგებს, რომ მისი კოდი არ მუშაობს და საჭიროა დახმარება. ან დამკვეთი ვერ ამახვილებს ყურადღებას პროექტის მნიშვნელოვან მახასიათებლებზე და დეველოპერები მეორე-ხარისხოვანი საკითხებითაა დაკავებული.

ასეთი საკითხები ბოლოს მაინც გამოჩნდება უკვე პრობლემების სახით, რაც მოითხოვს დამატებით დროს და სხვა რესურსებს მათ გადასაწყვეტად.

მოდელირების პროცესის სიკეთე ისიცაა, რომ იგი აადვილებს კომუნიკაციას პროექტში მონაწილე პიროვნებებს შორის. მაგალითად, როცა დამკვეთი შინაარსობრივად აღწერს რთულ ბიზნეს-პროცესს, ჩვენ შეგვიძლია იგი ავსახოთ მონაცემთა ნაკადების დიაგრამის სახით, რაც მის ბიზნეს-ლოგიკას შეესაბამება. ამ დროს დამკვეთს უადვილდება პროცესის უკეთ აღქმა და ხშირად

ამ პროცესის სრულყოფის საფუძველიც ხდება (მაგალითად, პროცედურების სიჭარბის აღმოფხვრის თვალსაზრისით).

ამგვარად, *ურთიერთობისას ხუთ წუთში შეიძლება მეტი ინფორმაციის მიღება, ვიდრე კორპორაციული დოკუმენტების 5 საათიანი კითხვისას.*

ასევე შესაძლებელია დეველოპერებს შორის კლასთა სტრუქტურის უკეთ გასაგებად UML დიაგრამების გამოყენება. ეს კი აუცილებელია გუნდის წევრებს შორის ერთიანი კონცეფციის არსებობისა და მეგობრული დამოკიდებულე-ბისათვის.

➢ **სიმარტივე**

პროგრამული უზრუნველყოფის წარმოების ინდუსტრიის მთელი არსებობის მანძილზე მიეთითება, რომ სისტემა იყოს შეძლებისდაგვარად მარტივი, მაგრამ, სამწუხაროდ, ეს პირობა ხშირად ვერ სრულდება. ეს კი იწვევს პროგრამის რეალიზაციის, ტესტირების და ექსპლუატაციის პროცესების გართულებას.

ყველაზე ხშირად პროგრამების გართულებას შემდეგი მოვლენები იწვევს:

- *რთული შაბლონების (Pattern) ხშირი გამოყენება.* საერთოდ არსებული შაბლონის გამოყენება ახალი პროგრამული სისტემის ასგაებად ერთ-ერთი მარტივი და შესაძლებელი ხერხია (პროტოტიპების თვალსაზრისით), მაგრამ დასმული ამოცანისთვის ის უნდა იყოს შესაბამისად მისაღები, არ უნდა იყოს უფრო რთული, ვიდრე ამას ამოცანა მოითხოვს;
- *სისტემის არქიტექტურის გართულება მომავალი შესაძლო გაფართოებების მხარდასაჭერად.* ხშირად ტრადიციული პროგრამული ტექნოლოგიების მიმდევრები, რომელთაც არ სურთ მომავალში ცვლილებების განხორციელება, ცდილობენ წინასწარ გაითვალისწინონ და დააპროექტონ სისტემის ჭარბი, გაფართოებული ვარიანტი (თუმცა, შეიძლება ასეთი მოთხოვნილება მომავალში ნაკლებალბათური იყოს, ან საერთოდ არ იყოს საჭირო).

მოქნილი მოდელირების მიმდევრები არ ქმნიან ჭარბ პროგრამულ სისტემას, მათ განკარგულებაშია შეძლებისდაგვარად მარტივი პროდუქტი, დღეისათვის აუცილებელი ფუნქციობით. თუ საჭირო გახდება სისტემის გაფართოება, მხოლოდ მაშინ დაუმატებენ არსებულ სისტემას ახალ, ასევე შეძლებისდაგვარად მარტივ ფუნქციობას. გამოიყენება პრინციპი *„ხვალის პრობლემა გადაწყდეს ხვალ“.* ამ დროს გაფართოების მიზანი ცალსახად იქნება

გარკვეული, რაც გამორიცხავს სიჭარბის შექმნას და სისტემის ზედმეტად გართულებას;

- *რთული ინფრასტრუქტურის შემუშავება.* ეს ფართოდ გავრცელებული შეცდომაა, რომელსაც გუნდი უშვებს. კერძოდ, გუნდის საწყისი ძალისხმევა მიმართულია პროექტის ინფრასტრუქტურის შექმნაზე (მაგალითად, კომპონენ-ტები, კლასების ბიბლიოთეკა და სხვ.), და არა სისტემის ცალკეული ბლოკების აგებაზე. ნაკლოვანება ისაა, რომ სისტემაში თავიდან ჩაიდება დამკვეთის საკმაო რესურსები და ამავე დროს მათ არ წარედგინებათ არავითარი შედეგი, რომელსაც ისინი გამოიყენებდნენ ოპერატიულად. დამკვეთს უნდა, რომ მიიღოს მიმწოდებლიდან კონკრეტული პროდუქტი, რომელიც მას დაეხმარება სამუშაოს შესრულებაში (და არა ცარიელი მონაცემთა ბაზების სისტემა ან შეცდომების დამუშავების ქვესისტემა).

ვინაიდან ვერ ხერხდება საჭირო ფუნქციობის პროგრამების სწრაფი დამუშავება, ეს წარმოშობს სათანადო რისკს პროექტის შესასრულებლად. საუკეთესო მიდგომაა, შემცირდეს ინფრასტრუქტურის დამუშავების მასშტაბები პროექტის შესრულებისას მანამ, სანამ ის არ გახდება აუცილებელი. მაგალითად, შეცდომების გასწორების ქვესისტემა შეიძლება დამუშავდეს მოგვიანებით, როცა ის აუცილებელი გახდება.

მოქნილი მოდელირების დროს ძირითადი დებულება მდგომარეობს იმაში, რომ მოდელები შეძლებისდაგვარად მარტივი იქნას შენარჩუნებული, დღეს მოხდეს იმის მოდელირება, რაც დღესაა საჭირო, და ხვალისა შესრულდეს ხვალ. მოდელები თამაშობს ძირითად როლს პროგრამებისა და მათი შექმნის პროცესების გასამარტივებლად. *განსაკუთრებით მარტივია იდეის შემუშავება და ამოცანის გაგება ერთი-ორი დიაგრამის დახაზვით, ვიდრე კოდის ასობით სტრიქონის დაწერით.*

➢ **უკუკავშირი**

არის თუ არა სამუშაო შესრულებული სწორად ? ერთადერთი ხერხია მასზე გამოძახილის (შეფასების, რეცენზიის) მიღება. მოდელის სისწორის შეფასებაც უნდა მოხდეს ასეთივე ხერხით, რომელიც განიხილება როგორც უკუკავშირი. მოდელი არის აბსტრაქცია. მაგალითად, UseCase ელემენტების ერთობლიობა – სისტემასთან მუშაობის ხერხების აბსტრაქციაა, ხოლო Component-ების მოდელი – პროგრამული უზრუნველყოფის შინაგანი სტრუქტურის აბსტრაქცია. როგორ უნდა დადგინდეს, არის თუ არა მიღებული

აბსტრაქცია სწორი? არსებობს ხერხების სიმრავლე, რომლებიც უზრუნველყოფს მოდელების უკუკავშირს:

- *მოდელი მუშავდება გუნდში.* აქ გამოძახილი მიიღება ოპერატიულად, სწრაფად;
- *მოდელის განხილვა ხდება მიზნობრივ აუდიტორიასთან.* მოთხოვნილებათა მოდელირება უნდა შესრულდეს დამკვეთის მომხმარებლებთან ერთად, რომლებიც ბოლოს იმუშავებენ ამ სისტემასთან. დაპროექტების დეტალური მოდელები კი უნდა შემუშავდეს დეველოპერ-პროგრამისტებთან ერთად. თუ ეს არ ხერხდება, მაშინ ყოველი შემუშავებული მოდელი სისტემური ანალიტიკოსის მიერ განხილულ უნდა იქნას პროგრამისტებთან ერთად, ასევე სასურველია დაიწეროს სცენარები თითოეულის გამოყენების მიზნით. შესაძლებელია ასევე არაფორმალური განხილვის (დისკუსიის) მოწყობა, მაგალითად სპეციალის-ექსპერტთან, რომელიც შემდეგ მოგვცემს გამოძახილს;
- *მოდელის რეალიზაცია.* ესაა ყველაზე საიმედო ხერხი გამოძახილის მისაღებად ანუ, მოდელი მუშავდება პროგრამის სახით და მიეწოდება სამუშაოდ დამკვეთ-მომხმარებელს;
- *ტარდება მიღება-ჩაბარების ტესტირება.* მოდელი უნდა ასახავდეს სისტემასთან დამკვეთის მოთხოვნებს. მიღება-ჩაბარების ტესტირების დროს დამკვეთი ამოწმებს თავისი მოთხოვნების სისწორეს.

საინტერესოა განხილულ იქნას დროთი დანახარჯები გამოძახილის მიღების თითოეული ვარიანტისათვის. როცა მოდელი მუშავდება გუნდში, გამოძახილი მიიღება მყისიერად, წამებში ან წუთებში. არაფორმალური განხილვისას შედეგი შეიძლება მიღებულ იქნას რამდენიმე წუთის ან საათის განმავლობაში. ფორმალური განხილვები შეიძლება გადაიდოს რამდენიმე დღით, კვირა ან თვით, მონაწილეებისაგან დამოკიდებულებით. მოდელის რეალიზაციის დროს პროგრამის საშუალებით შედეგები მიიღება სწრაფად, რამდენიმე წუთში, საათში ან დღეში. მიღება-ჩაბარების ტესტირება მოითხოვს რამდენიმე კვირას ან თვეს.

დროითი ხარჯები მნიშვნელოვანია, რადგან რაც უფრო სწრაფად მიიღება გამოძახილი მოდელის შესახებ, მით უფრო ადვილია არსებული მოდელის ცვლილება დამკვეთის მოთხოვნილებათა შესაბამისად.

უპირატესობა უნდა მიენიჭოს მოდელების შემუშავებას გუნდურად და მათ პროგრამულ რეალიზაციას, რადგან ქაღალდზე შეიძლება მოდელი ლამაზად გამოიყურებოდეს, მაგრამ მისი რეალიზაციის შედეგი არ მუშაობდეს.

- **გამბედაობა**

მოქნილი მოდელირება ან ზოგადად პროგრამული უზრუნველყოფის მოქნილი დამუშავება შედარებით ახალი მეთოდოლოგიაა და იგი უპირისპირდება ტრადიციულ, უკვე კარგად დამკვიდრებულ მეთოდოლოგიებს და მათ მიმდევრებს. ამიტომაც ამ ახალი მიმართულების გამტარებლებს დიდი გამბედაობა და რთულ წინააღმდეგობათა გადალახვა სჭირდებათ. გამბედაობა არის პროგრამების მოქნილი (სწრაფად) დამუშავების აუცილებელი კომპონენტი.

უპირველეს ყოვლისა, გამბედაობაა საჭირო, რათა მიღებულ იქნას გადაწყვეტილება მოქნილი მიდგომის გამოყენების შესახებ. შემდეგ კი მისი გამოყენების გაგრძელების შესახებ, თუ საქმე ვერ წავიდა წარმატებით (რაც ხშირად მოსალოდნელია). ორგანიზაციაში მოიძებნება სხვა შეხედულების ადამიანები, რომელთა წინააღმდეგობა დასაძლევია. ეს პოლიტიკაა.

მეორე მხრივ, პროგრამული სისტემის დამუშავებისას საჭიროა გამბედაობა მნიშვნელოვანი გადაწყვეტილების მისაღებად, კერძოდ, რომელი არქიტექტურული გადაწყვეტა ან რომელი დაპროგრამების ენა შეირჩეს. მუშაობის პროცესში საჭიროა გამბედაობა, რათა თუ საჭიროა შეცვლილ იქნეს მიმართულება, უარი ითქვას შესრულებულზე ან ჩატარდეს რეფაქტორინგი, თუ ზოგიერთი გადაწყვეტის შედეგი აღმოჩნდა არასწორი.

მესამე მხრივ, საჭიროა გამბედაობა, რათა გაგებულ იქნას, რომ არ ვართ იდეალურები და შეგვიძლია შეცდომების დაშვებაც. გამბედაობაა საჭირო, რათა გვწამდეს, რომ ხვალის ამოცანების გადაწყვეტას ხვალ შევძლებთ;

- **თავმდაბლობა**

პროგრამული უზრუნველყოფის კარგ დეველოპერებს აქვთ საკმარისი თავმდაბლობა, რომ შეიმეცნონ, რომ მათ არაფერი არ იციან. მოქნილი მოდელირების კონცეფციით მომუშავე სპეციალისტებმა კარგად იციან, რომ მათი კოლეგები და დამკვეთები არიან ექსპერტები თავიანთ სფეროებში, აქვთ ცოდნა, ანუ ღირებული ინფორმაცია, რომელიც საჭიროა პროექტისათვის. მაგალითად, არსებობენ დეველოპერები, რომლებიც უკეთესად ქმნიან კოდს ან

ატესტირებენ პროგრამებს, უკეთესად ამოდელირებენ მოთხოვნილებებს ან ქმნიან არქიტექტურებს. მომხმარებლები უკეთესად ერკვევიან თავიანთ ბიზნესპროცესებში. ორგანიზაციის ხელმძღვანელებს უკეთესად ესმით თავიანთი სფეროს განვითარების ტენდენციები, ხოლო თანამშრომლებს კარგად ესმით, რისი გაკეთების უფლება აქვთ და რისი არა საკუთარ პროდუქციასთან.

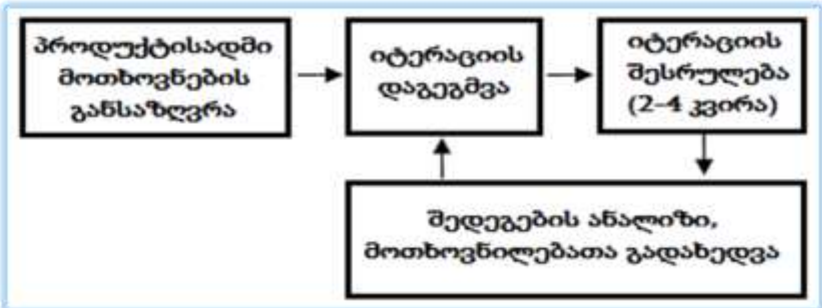
მოქნილი მოდელირების სპეციალისტს უნდა ჰქონდეს საკმარისი თავმდაბლობა თავისი სამუშაოს შესასრულებლად, მას სჭირდება დახმარება და უნდა ითანამშრომლოს ამ ადამიანებთან. თავმდაბლობა ადამიანის ხასისიათის თვისებაა (დიპლომატიურობაა), რომლის წყალობითაც ის კომუნიკაბელურია და მეტ ინფორმაციას იღებს ადამიანებთან ურთიერთობისას, რაც ამაღლებს მის მწარმოებლურობას და, ე.ი. პროექტის შესრულების წარმატებას.

ამგვარად, პროგრამული სისტემის მენეჯერი, კონკრეტული პროექტის ამოცანებისა და მოთხოვნების შესაბამისად, უნდა განსაზღვრავდეს როგორც პროგრამირების მეთოდის, ეტაპთა ფაზებისა და იტერაციათა მოთხოვნების შერჩევა-ფორმირებას, ასევე მუშა გუნდის შემადგენლობას.

3.1.5. Scrum - მოქნილი მეთოდის ფრეიმვორკი

პროგრამული ინდუსტრიის სფეროში Agile მეთოდოლოგიის მნიშვნელოვანი წარმომადგენელია Scrum მეთოდი [46]. იგი პირველად იაპონელებმა მოიხსენიეს, როგორც ინოვაციური მიდგომა ახალი სერვისებისა და პროდუქტების დასამუშავებლად (არა მხოლოდ პროგრამული პროდუქტებისთვის). მეთოდის ძირითადი არსი მდგომარეობდა მცირე ზომის უნივრსალური გუნდის შეკრულ, თანამიმდევრულ მუშაობაში, რომელიც ამუშავებს პროექტის ყველა ფაზას. სიმბოლურ ანალოგიას აკეთებდნენ „რაგბის“-თან, როდესაც ერთიანი გუნდი მოძრაობს წინ და უკან ბურთის გადაცემის შესაბამისად (Scrum-„შეჯიდება“).

ზოგადი აღწერა. Scrum მეთოდი საშუალებას იძლევა პროექტები დამუშავდეს მოქნილად (სწრაფად) მცირე გუნდის მიერ (5-9 კაცი გუნდში). დამუშავების პროცესი იტერაციულია და დიდ თავისუფლებას აძლევს გუნდს. ამასთანავე, მეთოდი ძალზე მარტივია და შესასწავლად ადვილი, ამიტომაც პრაქტიკაში ადვილად გამოყენებადია (ნახ. 3.2).



ნახ. 3.2. Scrum მეთოდის ბიჯები

თავიდან განისაზღვრება მოთხოვნები მთლიანი პროდუქტისათვის. შემდეგ ამოირჩევა მათგან ყველაზე აქტუალურები და შედგება პირველი (მომდევნო) იტერაციის გეგმა. იტერაციის პერიოდში გეგმა არ იცვლება (ეს ხელს უწყობს დამუშავების პროცესის სტაბილობას), მისი ხანგრძლიობა 2-4 კვირაა. იტერაცია სრულდება პროდუქტის სამუშაო ვერსიის შექმნით, რომელიც შეიძლება გადაეცეს დამკვეთს, მოხდეს მისი დემონსტრირება, თუნდაც მინიმა-ლური ფუნქციური შესაძლებლობებით.

ამის შემდეგ ხდება შედეგების განხილვა და პროდუქტისადმი მოთხოვნილებათა კორექტირება. ეს მოსახერხებელია, რადგან არა მხოლოდ ზუსტდება პროდუქტის ფუნქციები, არამედ დამკვეთს შეუძლია მისი გამოყენებაც. შემდეგ იგეგმება ახალი იტერაცია და ყველაფერი მეორდება.

იტერაციის შიგნით მთლიანად მუშაობს გუნდი. Scrum აქ როლებს არ განსაზღვრავს. მენეჯმენტთან და დამკვეთთან სინქრონიზაცია ხდება იტერაციის დასრულების შემდეგ. იტერაცია შეიძლება შეწყდეს მხოლოდ განსაკუთრებულ შემთხვევებში.

როლები. Scrum მეთოდში არის სამი როლი:

- *პროდუქტის მფლობელი (Product Owner)* – ესაა პროექტის მენეჯერი, რომელიც წარმოადგენს დამკვეთის ინტერესებს. მისი მოვალეობაა პროდუქტის საწყისი მოთხოვნების (Product Backlog) განსაზღვრა, მათი დროულად კორექტირება, პრიორიტეტების, ჩაბარების ვადების დადგენა და სხვ. იგი არ მონაწილეობს უშუალოდ იტერაციის შესრულებაში;
- *Scrum-ოსტატი (Scrum Master)* – უზრუნველყოფს გუნდის მაქსიმალურ მწარმოებლურობასა და პროდუქტიულობას, როგორც Scrum-პროცესის შესასრულებლად, ისე სამეურნეო და ადმინისტრაციული ამოცანების გადასაწ-

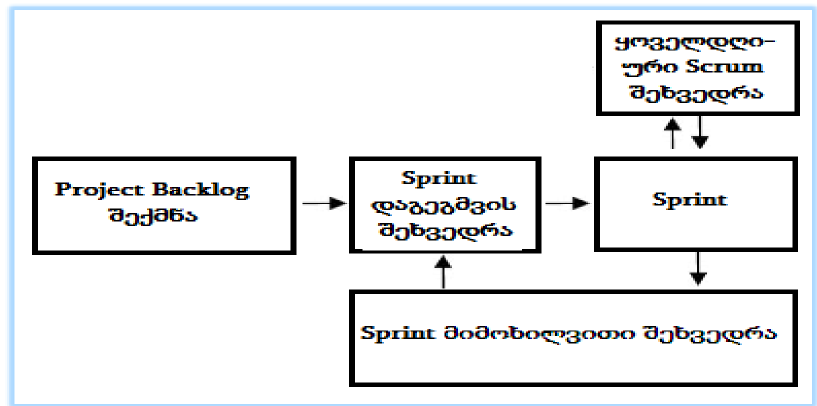
ყვეტად. კერძოდ, მისი ამოცანაა გუნდის დაცვა იტერაციის დროს ყოველგვარი გარე ზემოქმედებიდან;

- *Scrum-გუნდი (Scrum Team)* – ჯგუფია, რომელიც შედგება 5-9 დამოუკიდებელი, ინიციატივიანი პროგრამისტ-დეველოპერებისგან. გუნდის *პირველი ამოცანაა* იტერაციისათვის რეალურად მიღწევადი და პროექტისთვის პრიორიტეტული დავალებების განსაზღვრა (Project Backlog-ის საფუძველზე და პროდუქტის მფლობელისა და Scrum-ოსტატის აქტიური მონაწილეობით). *მეორე ამოცანაა* ამ დავალებების უეჭველი შესრულება დადგენილ ვადებში და მოთხოვნილი ხარისხით. მნიშვნელოვანია, რომ გუნდი თვითონ მონაწილეობს დავალებათა დასმის პროცესში და თვითონ წყვეტს მათ. აქ შეთავსებულია თავისუფლება და პაუხისმგებლობა, დონეზეა მოვალეობათა დისციპლინა.

პრაქტიკები. Scrum-ში განსაზღვრულია პრაქტიკები:

- *Sprint Planning Meeting.* შეხვედრა Sprint-ის დასაგეგმად (ესაა მოკლე დისტანცია, ანუ იტერაცია). თავიდან პროდუქტის მფლობელი, Scrum-ოსტატი, გუნდი, ასევე დამკვეთის წარმომადგენელი და სხვა დაინტერესებული პირები განსაზღვრავენ, თუ რომელი მოთხოვნებია Project Backlog-დან უფრო პრიორიტეტული და რომელი უნდა განხორციელდეს მოცემული სპრინტის ფარგლებში. ფორმირდება Sprint-Backlog. შემდეგ Scrum-ოსტატი და Scrum-გუნდი განსაზღვრავენ, თუ როგორ უნდა იქნას მიღწეული დასმული მიზნები Sprint-Backlog-დან. მისი ყოველი ელემენტისათვის დადგინდება ამოცანათა სია და შეფასდება მათი შრომატევადობა;
- *Daily Scrum Meeting* – ყოველდღიური, 15-წუთიანი Scrum თათბირი, რომლის მიზანია იმის გარკვევა, თუ რა მოხდა წინა თათბირის შემდეგ, კორექტირდეს სამუშაო გეგმა დღევანდელი დღის შესაბამისად და განისაზღვროს არსებული პრობლემების გადაწყვეტის გზები. Scrum-გუნდის ყოველი წევრი პასუხობს სამ კითხვას: რა გააკეთა წინა თათბირის შემდეგ, რა პრობლემები აქვს და რა უნდა გააკეთოს მომდევნო შეხვედრამდე. ამ თათბირზე დასწრება შეუძლია ნებისმიერ დაინტერესებულ პირს, მაგრამ გადაწ-ყვეტილების მიღების უფლება აქვთ მხოლოდ Scrum-გუნდის წევრებს. ეს წესი მართებულია, რადგან ისინი იღებენ ვალდებულებას იტერაციის მიზნის მიღსაღწევად. გარე პირის ჩარევა ასეთ დროს ხსნის მათგან შედეგზე პასუხისმგებლობას;

- *Sprint Review Meeting.* Sprint მიმოხილვის შეხვედრა. იმართება ყოველი სპრინტის დამთავრების შემდეგ (ნახ. 3.3).



ნახ. 3.3. Scrum-მეთოდი Sprint-ბიჯებით

თავიდან Scrum-გუნდი წარმოადგენს პროდუქტის დემონსტრაციას, რომელიც ამ სპრინტის დროს განხორციელდა. აქ მოწვეული იქნება დამკვეთის ყველა დაინტერესებული წარმომადგენელი.

პროდუქტის მფლობელი განსაზღვრავს თუ რომელი მოთხოვნები იქნა შესრულებული Sprint Backlog-დან და განიხილავს გუნდთან და დამკვეთის წარმომადგენლებთან ერთად, თუ როგორ განაწილდეს პრიორიტეტები უკეთესად მომდევნო სპრინტის Sprint Backlog-ში. შეხვედრის მეორე ნაწილი ეხება წინა სპრინტის ანალიზს, რომელსაც წარმართავს Scrum-ოსტატი. Scrum-გუნდი აანალიზებს ბოლო სპრინტის დროს ერთობლივი მუშაობის დადებით და უარყოფით მომენტებს, გამოიტანს დასკვნებს და იღებს მნიშვნელოვან გადაწყვეტილებებს შემდგომი მუშაობისათვის. Scrum-გუნდი ასევე ეძებს გზებს მომავალი სამუშაოს ეფექტურობის ასამაღლებლად. შემდეგ ციკლი მეორდება.

3.1.6. Kanban/Lean - ეკონომიური მოქნილი მეთოდი

Kanban/Lean – პროგრამული უზრუნველყოფის დამუშავების ეკონომიური მეთოდია (Lean method of software development) [47].

პროგრამული უზრუნველყოფის ეკონომიური დეველოპმენტი (Lean Software Development) – არის პროგრამული უზრუნველყოფის *მიწოდების პრინციპების მთელი რიგი,* ეკონომიური წარმოების პრინციპების შესაბამისად.

მომჭირნე გარემოში უნდა გამოირიცხოს ის ქმედებები ან პროცესები, რომლებიც იწვევს მიზნების მისაღწევად ძალისხმევის ან/და რესურსების ისეთ ხარჯებს, რაც კლიენტს არ მოუტანს სარგებელს. სინამდვილეში, მომჭირნეობა ფოკუსირებულია ნაკლები სამუშაოს მქონე ღირებულების შენარჩუნებაზე. ეკონომიურ (Lean) მიდგომებს ხშირად უწოდებენ Six-Sigma ან Just-In-Time (JIT) [48]. აღნიშნული კონცეფცია შემუშავდა Motorola კორპორაციაში 1986 წ. და გამოყენებულ იქნა პირველად General Electric-ში.

„ *6σ* “-კონცეფციის არსი მდგომარეობს წარმოებაში თითოეული პროცესის შედეგების ხარისხის გაუმჯობესების აუცილებლობაში, ოპერაციული საქმიანობის დეფექტებისა და *სტატისტიკური გადახრების* მინიმუმამდე შემცირებაში. იგი იყენებს ხარისხის მართვის მეთოდებს, სტატისტიკური მეთოდების ჩათვლით, მოითხოვს გაზომვადი მიზნებისა და შედეგების გამოყენებას, აგრეთვე მოიცავს საწარმოში სპეციალური სამუშაო ჯგუფების შექმნას, რომლებიც ახორციელებს პროექტებს პრობლემების აღმოსაფხვრელად და პროცესების სრულყოფის მიზნით.

კანბანსაც და სკრამსაც აქვს საკუთარი ფესვები ეკონომიურ წარმოებაში (Lean production), რომლის მიზანიც სამი სახის ნარჩენების მოცილებაა (Lean-პრინციპები იაპონური ავტომრეწველობის სფეროდან) [49]:

- Muda – სამუშაო, რომელიც არ უმატებს პროდუქტს ფასეულობას;
- Muri – თანამშრომელთა და მანქანების გადატვირთვა;
- Mura – პროცესების არარეგულარობა.

სიტყვა kanban იაპონურად არის „სასიგნალო ბარათი“ (kan - სიგნალი, ban - ბარათი). იგი Toyota-ს ავტომანქანების წარმოების ფირმის ტექნოლოგიაა, რომელიც შეიქმნა წარმოების სტაბილური ნაკადის უზრუნველსაყოფად და მარაგების დონის შესამცირებლად. იყენებენ ასეთ ახსნასაც - „დაუმთავრებელი წარმოების მოცულობის შემცირება“.

Kanban-ის გამოყენების ფუძემდებლად ინფორმაციული ტექნოლოგიების სფეროში ითვლება *დევიდ ანდერსონი,* რომელმაც 2007 წელს პირველმა წარმოადგინა ამ მეთოდის ზოგადი კონცეფცია [50]. მან ჩამოაყალიბა *4 საბაზო პრინციპი* და *6 ძირითადი პრაქტიკა,* რომლებსაც თავიანთ საქმიანობაში ინტეგრირებულად იყენებს კომპანიები Kanban-ის საფუძველზე.

- **Kanban-ის პრინციპები:**

- *სპ1. დაიწყეთ იმით, რასაც ამჟერად აკეთებთ*: რომელი აქტუალური სამუშაოც სრულდება, ჯერ ის უნდა დასრულდეს და მხოლოდ ამის შემდეგ დაიწყოს ახალი სამუშაო;
- *სპ2. დათანხმდით, რომ ევოლუციური ცვლილებები მოსალოდნელია*: შემდგომ განვითარებას აქვს განსაკუთრებული მნიშვნელობა, ოღონდაც აქ სრულყოფა მიღწეულ უნდა იქნას ძირითადად მცირე / ევოლუციური ბიჯების ხარჯზე;
- *სპ1. პატივი ეცით პირველარსებულ პროცესებს / როლებს / ვალდებულებებს*: Kanban ადვილად რეალიზებადია, ყველა როლი, პროცესი და ა.შ., რჩება;
- *სპ2. წაახალისეთ ლიდერობა ორგანიზაციის ყველა დონეზე*: სრულყოფა შესაძლებელია მხოლოდ იმ შემთხევაში, თუ ამოქმედებულია ორგანიზაციის ყველა დონე. განსაკუთრებით მნიშვნელოვანია ის, რომ სამუშაოს შემსრულებლები უშუალოდ უკეთებდნენ დემონსტრირებას „ლიდერობის აქტებს“ და ახმოვანებდნენ გაუმჯობესების კონკრეტულ წინადადებებს [50].

- **Kanban-ის ძირითადი პრაქტიკები:**

- *ძპ1. სამუშაოს მსვლელობის (ნაკადის) ვიზუალიზაცია:* ღირებულებათა ჯაჭვი პროცესის სხვადასხვა ეტაპებისათვის (მაგალითად, მოთხოვნილების განსაზღვრა, პროგრამირება, დოკუმენტირება, ტესტირება, დანერგვა) კარგადაა ვიზუალიზირებული ყველა მონაწილისათვის. ეს ხორციელდება Kanban-დაფის (Kanban-Board) დახმარებით, რომელზეც სხვადასხვა კვანძები (Stations) აისახება სვეტების სახით (ნახ. 3.4).

ინდივიდუალური მოთხოვნილებები (ამოცანები, ფუნქციები, მომხმარე-ბელთა ისტორიები, მინიმალური საბაზრო მახასიათებლები და ა.შ.) ჩაიწერება სააღრიცხვო ბარათებში („ბილეთები“, მიმაგრებული დაფის უჯრაზე, Tiket - TK).

მოთხოვნა / დავალება / ინციდენტების პროგრესი					
დავალ_ჟურნ.	დაგეგმილი	მიმდინარე	Developed	ტესტირება	დასრულება
მომხმ. ისტორია	მომხმ. ისტორია TK TK TK	მომხმ. ისტორია	TK TK	მომხმ. ისტორია TK	მომხმ. ისტორია TK TK
მომხმ. ისტორია	IN	მომხმ. ისტორია TK	TK TK IN	TK	IN IN
მომხმ. ისტორია		IN			
მომხმ. ისტორია					
მომხმ. ისტორია					

ნახ. 3.4. Kanban-ის დაფა (იყენებს Software Development Life Cycle-ს)

ისინი დროის და შესრულებული ბიჯის შესაბამისად გადაადგილდება დაფაზე მარცხნიდან მარჯვნივ;

- *ძ32. დაწყებული სამუშაოს მოცულობის შეზღუდვა*: ბილეთების რაოდენობა (Work in Progress - WiP), რომლებიც შეიძლება დამუშავებულ იქნას ერთდროულად ერთ კვანძში, შეზღუდულია. მაგალითად, თუ კვანძი „პროგრამირება“ ამუშავებს ორ ბილეთს და ამ კვანძის ლიმიტი 2-ია, მაშინ მას არ შეუძლია მე-3 ბილეთის მიღება, თუნდაც მოთხოვნის განსაზღვრება ამის უფლებას აძლევდეს. ეს ქმნის ამოთრევის-სისტემას (Pull-System), სადაც ყოველ კვანძს გადააქვს თავისი სამუშაო წინა კვანძში, და არ გადასცემს დასრულებულ სამუშაოს მომდევნო კვანძს;
- *ძ31. ნაკადის მართვა (Manage flow)*: Kanban-პროცესის მონაწილეები ზომავენ ტიპურ მაჩვენებლებს, როგორიცაა რიგის სიგრძე, ციკლის დრო, გამტარუნარიანობა, რათა დაადგინონ თუ რამდენად კარგადაა ორგანიზებული სამუშაო პროცესი, სად შეიძლება მისი სრულყოფა და რა შეიძლება დაპირდეს პარტნიორებს, ვისთვისაც მუშაობენ. ეს აადვილებს დაგეგმვას და ამაღლებს საიმედოობას;
- *ძ32. პროცესისთვის წესები უნდა გაკეთდეს ცხადად:* იმისათვის, რომ პროცესში მონაწილეებმა იცოდნენ თუ რა მოსაზრებებით და კანონებით მუშაობენ, აუცილებელია რაც შეიძლება მეტი ცხადი წესების გაკეთება. მათ შორის:

○ განმარტებულ იქნას ტერმინი „დასრულებულია“, ასევეა განსაზღვრება „მზადაა Scrum-ში“;

○ Kanban-დაფის თითოეული სვეტის მნიშვნელობა;

○ პასუხები შეკითხვებზე: ვინ მოძრაობს, როდის მოძრაობს, როგორ შეირჩეს შემდეგი ბილეთი არსებულებიდან და ა.შ.;

- *ძ36. უკუკავშირის ციკლების რეალიზაცია*: ფიქსირებულ თარიღებში გუნდები ერთმანეთს უკავშირდება. მაგალითად:

○ რეტროსპექტივები: თანამშრომლობის მიმოხილვა;

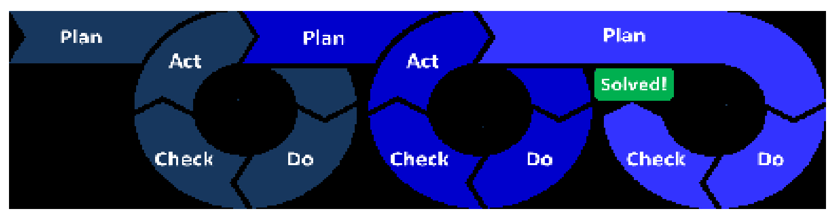
○ მომდევნო შეხვედრა: მომავალი დავალებების შეთანხმება / ბლოკირებების მოხსნა / ნაკადის კოორდინაცია;

○ სამუშაო ოპერაციების რეცენზირება: კომპანიის Kanban გუნდები ხვდება და გამოცდილებას უზიარებს ერთმანეთს;

- *ძ37. მოდელების გამოყენება პროცესის ერთობლივად გაუმჯობესების შესაძლებლობების დასადგენად:* მოდელები ამარტივებს პროცესს. პოპულარული მოდელია, მაგალითად, ღირებულება, ნაკადი, ნარჩენები „ეკონომიური IT“-დან (Lean IT).

სხვა მოდელები ბაზირებულია *ედუარდ დემინგის* იდეებზე (ციკლი PDCA [Plan-Do-Check-Act] „დაგეგმე, გააკეთე, შეამოწმე, იმოქმედე" (ნახ.3.5) – არის სრულყოფის პროცესი, რომელიც ეფუძნება ცოდნის თეორიის გაგებას და გამოიყენება ხარისხის მენეჯმენტში) ან ვიწრო ადგილების თეორიაზე, სისტემურ აზროვნებაზე ან კომპლექსურობის (სირთულის) თეორიაზე [52, 53].

მოდელების დახმარებით შესაძლებელია პროცესის უკეთ გაგება და ექსპერიმენტების მოძიება, რომელთაც მივყავართ პროცესის სრულყოფამდე.



ნახ. 3.5. PDCA ციკლის იტერაციული განმეორება პრობლემის მოგვარებამდე

ვიზუალიზირება და WiP-ის შეზღუდვები მარტივი საშუალებაა, რომლითაც სწრაფად ხდება თვალსაჩინო, თუ რა სისწრაფით მოძრაობს ბილეთები სხვადასხვა კვანძებში და სად გროვდება (იჭედება) ისინი.

იმ კვანძებს, სადაც გროვდება ბილეთები და ამ დროს მომდევნო კვანძი თავისუფალია, უწოდებენ ვიწრო ადგილებს. Kanban-დაფის ანალიზით შესაძლებელია ზომების მიღება მაქსიმალურად თანაბარი ნაკადის მისაღწევად.

მაგალითად, შეზღუდვები შეიძლება შეიცვალოს ცალკეული კვანძებისათვის, შეიძლება შემოტანილ იქნას ბუფერები (განსაკუთრებით ვიწრო ადგილების გაჩენამდე, რაც გამოწვეულია დროებით რესურსების წვდომის გამო), კვანძებზე მომუშავეთა რაოდენობა შეიძლება შეიცვალოს, აღმოიფხვრას ტექნიკური პრობლემები და ა.შ. სრულყოფის ასეთი უწყვეტი პროცესი არის Kanban-ის განუყოფელი ნაწილი [50, 54].

3.1.7. Scrum და Kanban მეთოდების შედარება

Scrum – „სტრუქტურული მიდგომა“. ყოველ პროექტზე მუშაობს სპეციალისტების უნივერსალური გუნდი, რომელსაც უერთდება ორი როლი: პროდუქტის მფლობელი (დამკვეთი) და Scrum-ოსტატი. პირველი აერთიანებს გუნდს დამკვეთთან და აკვირდება პროექტის განვითარებას, ადგენს ამოცანების წონას [56]. იგი არაა გუნდის ფორმალური ხელმძღვანელი, – კურატორია.

მეორე როლი, სკრამ-ოსტატი ეხმარება პირველს ბიზნეს-პროცესების ორგანიზებაში. კერძოდ, ატარებს საერთო შეკრებებს, აგვარებს საყოფაცხოვრებო პრობლემებს, ხელს უწყობს გუნდის მოტივაციას და აკვირდება სკრამ-მიდგომის შესრულების დაცვას. Scrum-მიდგომა ყოფს სამუშაო პროცესს თანაბარ სპრინტებად - ესაა პერიოდი ერთი კვირიდან თვემდე, რაც დამოკიდებულია პროექტსა და გუნდზე. პროცესები იტერაციულია (ნახ.3.6).

სპრინტის წინ ფორმულირდება ამოცანები ამ სპრინტისათვის, ხოლო სპრინტის ბოლოს განიხილება შედეგები. გუნდი იწყებს ახალ სპრინტს. სპრინტების შედარება ერთმანეთთან მოსახერხებელია, რაც შესაძლებელს ხდის ვმართოთ სამუშაოს ეფექტიანობა.

ნახ. 3.6

Scrum-ში ზომავენ სპრინტის დროს შესრულებული დავალებების საერთო წონას. პროექტის ყველა ამოცანის მთლიანი წონის გაყოფით სპრინტის მწარმოებლურობაზე, ღებულობენ პროექტის შესრულების სავარაუდო ვადას. აქედან გამომდინარე, Scrum-ის გუნდის ამოცანაა მწარმოებლურობის გაზრდა და ამით პროექტის შესრულების ვადის შემცირება.

Kanban – „ბალანსური მიდგომაა". მისი ამოცანაა გუნდში სხვადასხვა სპეციალისტების (დიზაინერები, დეველოპერები და სხვ.) სამუშაო დატვირთვის დაბალანსება. გუნდი ერთიანია და მასში არაა როლები. ბიზნეს-პროცესი იყოფა არა „სპრინტებად", არამედ კონკრეტული ამოცანების შესრულების სტადიებად, მაგალითად, „დაგეგმილია", „მუშავდება", „ტესტირდება", „დასრულებულია" და ა.შ.

ეფექტიანობის მთავარი მაჩვენებელია *ამოცანის გავლის საშუალო დრო* კანბანის დაფაზე დასაწყისიდან დასასრულამდე. თუ ამოცანა სწრაფად გავიდა „ფინალში", ე.ი. გუნდი მუშაობდა ნაყოფიერად და ჰარმონიულად. თუ გაჭიანურდა ამოცანის შესრულება, მაშინ საჭიროა გარკვევა, თუ რომელ ეტაპზე და რა მიზეზით მოხდა შეფერხება, ვისი საქმიანობის პროცესი მოითხოვს ოპტიმიზაციას.

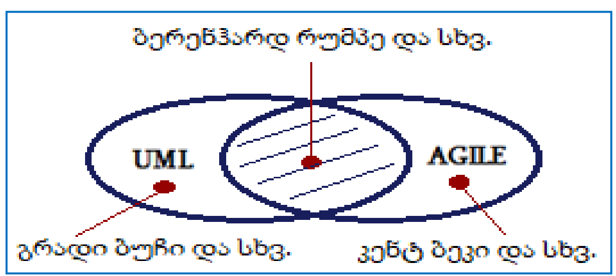
Agile მიდგომები პროცესების ვიზუალიზაციისათვის იყენებს ფიზიკურ და ელექტრონულ დაფებს. ისინი უზრუნველყოფს სამუშაო პროცესები გახადოს გამჭვირვალე და გასაგები ყველა სპეციალისტისთვის, რაც მნიშვნელოვანია, რადგან გუნდს არ ჰყავს ერთი ფორმალური ხელმძღვანელი.

3.2. UML და Agile მეთოდოლოგიების გამოყენების კომპრომისული გადაწყვეტა

გამოყენებითი კომპიუტერული სისტემების მენეჯმენტის თანამედროვე მეთოდოლოგიები, პროგრამული პარადიგმების განვითარების ფონზე, მნიშვნელოვნად განსხვავდება ერთმანეთისაგან, რაც, ერთგვარად, მრავალ ობიექტურ და სუბიექტურ ფაქტორზეა დამოკიდებული.

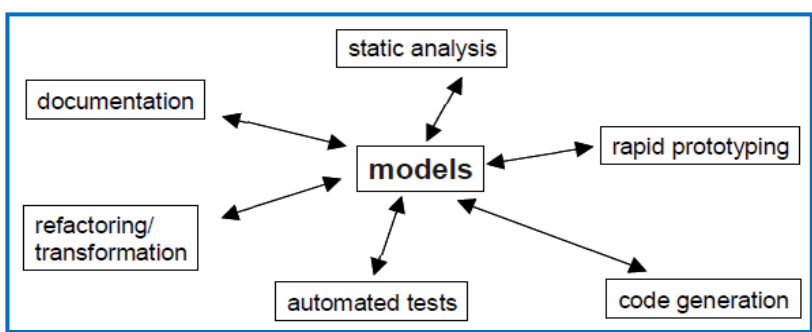
მეცნიერები და პრაქტიკოსი დეველოპერები ცდილობენ დაასაბუთონ თავიანთ მოსაზრებათა ჭეშმარიტება და იდეალურად წარმოადგინონ ესა თუ ის კონცეფცია. მაგალითად, უნიფიცირებული მოდელირების ენა (UML - გ. ბუჩი, ი. ჯაკობსონი, ჯ. რამბო და სხვ.) თუ მოქნილი (Agile) დაპროგრამება, ექსტრემალური პროგრამირების მაგალითზე (კ. ბეკი, დ. მარტინი და სხვ.) [14, 20].

ლიტერატურულ წყაროებში მოიძებნება ისეთ მეცნიერთა ნაშრომებიც პროგრამული ინჟინერიის სფეროში, რომლებიც კომპრომისული მიდგომით ჰიბრიდულ ვარიანტსაც გვთავაზობენ (მაგალითად, კანადელი ს. ამბლერი [45], გერმანელი ბ. რუმპე [46], ქართველი გ. სურგულაძე [58] და სხვ.) (ნახ. 3.7).



ნახ. 3.7. UML და Agile მეთოდოლოგიების ერთობლივი გამოყენების კონცეფცია

მიუნხენის ტექნიკური უნივერსიტეტის პროფესორმა ბ. რუმპემ თავის ფუნდამენტურ კვლევებში ნათლად ჩამოაყალიბა UML მოდელების (დიაგრა-მების) გამოყენების პოტენციალური შესაძლებლობები პროგრამული აპლიკაციების შექმნის სასიცოცხლო ციკლის სხვადასხვა ეტაპებისა და ამოცანებისათვის (ნახ. 3.8) [45].



ნახ. 3.9. UML მოდელების გამოყენების პოტენციალი

განსაკუთრებული ყურადღება ამ ნაშრომში გამახვილებული იყო Agile მეთოდოლოგიის ერთ-ერთი ყველაზე პოპულარულ, ექსტრემალური პროგრა-მირების მეთოდის საფუძველზე პროგრამული სისტემის კოდირების ეფექტუ-

რობის ამაღლების მიზნით – სწორედ უნიფიცირებული მოდელების გამოყენებაზე.

ექსტრემალური პროგრამირება (XP) ხასიათდება შემდეგი მახასიათებ-ლებით:

- *იგი ორიენტირებულია ძირითად მიზანზე* – ეფექტურ პროგრამულ კოდზე. დოკუმენტაცია იგნორირებულია, მაგრამ გამოყენებულია კოდირების სტანდარტები მისი აღწერის მიზნით;
- *ავტომატიზებული ტესტები გამოიყენება ყველა დონეზე.* პრაქტიკული გამოცდილება უჩვენებს, რომ თუ კოდი სწორადაა აგებული, მაშინ წუნის (შეცდომების) პროცენტი საგრძნობლად დაბალია. ამ პროცესის ავტომატიზაცია კი შესაძლებელს ხდის ტესტირების ექსპერიმენტი ჩატარდეს მრავალჯერადად;
- *გამოიყენება ძალზე მცირე იტერაციები უწყვეტი ინტეგრაციით, ხოლო სისტემა მაქსიმალურად მარტივია;*
- *რეფაქტორინგი გამოიყენება კოდის სტრუქტურის გასაუმჯობესებლად,* ხოლო ტესტები უზრუნველყოფს დეფექტების გამოვლენის დაბალ სიხშირეს რეფაქტორინგის შედეგად.

ექსტრემალური პროგრამირების მეთოდის გამოყენების დროს დოკუმენტაციაზე უარის თქმა მოტივირებულია სამუშაო დატვირთვის შემცირების მიზნით, რომ დეველოპერები არ ენდობიან დოკუმენტებს, რადგან ისინი უმეტესად მოძველებულია (შეიძლება არ იყოს მასში ასახული რეალური ობიექტის ბოლო დროინდელი ცვლილებები). ამგავარად, XP უშუალოდ ორიენტირებულია კოდზე. პროექტირების ყველა ქმედება ასახულია კოდში.

პროგრამული სისტემის ხარისხი უზრუნველყოფილია ტესტირების პროცესებით. ტესტები მუშავდება წინასწარ, თვით მუშა პროგრამის შექმნამდე. სისტემის არქიტექტურა ვლინდება უშუალოდ კოდირების დროს.

არქიტექტურული ნაკლოვანებები აღმოიფხვრება *რეფაქტორინგის მეთოდების* გამოყენებით [9,59]. ესაა ტრანსფორმაციული მეთოდები, რომლებითაც შესაძლებელია სისტემის მცირე გარდაქმნების ჩატარება მისი სტრუქტურის სრულყოფის მიზნით.

ამგვარად, *მოქნილი პროგრამირების მეთოდების გამოყენებისას შესაბამისი UML მოდელების გააზრებული ჩართვით პროექტების განვითარება უნდა გახდეს უფრო მეტად ეფექტური.*

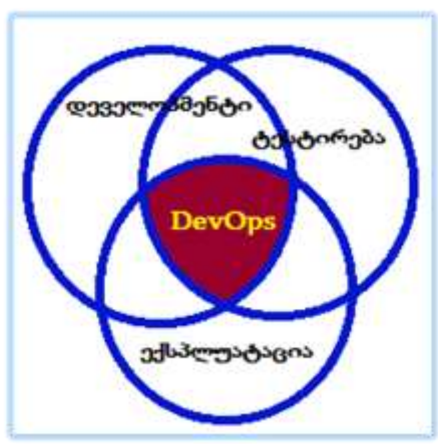
ამ თვალსაზრისით, მოდელებით მართვადი არქიტექტურების გამოყენება, შეიძლება მივიჩნიოთ ძალზე პერსპექტიულ მიმართულებად. მაგალითად, ASP.NET MVC ტექნოლოგა და სხვ. [9, 60, 61].

MVC (Model-View-Controller) – ესაა პროგრამული აპლიკაციის მონაცემების, მომხმარებლის ინტერფეისისა და მმართველი ლოგიკის დაყოფის სქემა სამ კომპონენტად: მოდელი, წარმოდგენა და კონტროლერი.

მოდელი არის მონაცემები და რეაგირებს კონტროლერის ბრძანებებზე, იცვლის თავის მდგომარეობას; *წარმოდგენა* უზრუნველყოფს მოდელის მონაცემთა ასახვას მომხმარებლისთვის, რომელსაც შეუძლია მოდელის შეცვლა; *კონტროლერი* ასრულებს (ინტერპრეტაციას უკეთებს) მომხმარებლის ქმედებებს, აცნობებს მოდელს ცვლილებების შესახებ.

3.3. Agile და DevOps მეთოდოლოგიები და მათი ინსტრუმენტული საშუალებები

DevOps არის პროგრამული უზრუნველყოფის განვითარების (დეველოპმენტის) და ექსპლუატაციის (ოპერაციების) ერთობლიობა. ამ მოდელის მიხედვით, განვითარებისა და ოპერაციების გუნდები ერთად მუშაობენ პროგრამული აპლიკაციის მთელი სასიცოცხლო ციკლის განმავლობაში, შემუშავებიდან და ტესტირებიდან დაწყებული ექსპლუატაციის ჩათვლით (ნახ. 3.10).



ნახ.3.10. DevOps მეთოდოლოგია

რა განსხვავებებაა Agile და DevOps შორის ?

- Agile მეთოდოლოგიის მიზანია პროგრამული აპლიკაციების განვითარება (დეველოპმენტი) და მიწოდება მომხმარებლებზე მათი მოთხოვნილებების გათვალისწინებით;

- DevOps პრაქტიკა აერთიანებს პროგრამული სისტემის უწყვეტი განვითარების (დეველოპმენტის) და ექსპლუატაციის ოპერაციებს, რისთვისაც იყენებს ავტომატიზაციის თანამედროვე CASE ინსტრუმენტებს.

Agile და DevOps მეთოდოლოგიები ასრულებს ურთიერთდამატებით როლებს. მაგალითად, პროგრამული სისტემების ავტომატიზებული კონსტრუირება და ტესტირება, უწყვეტი ინტეგრაცია და უწყვეტი მიწოდება [2, 42, 62].

Agile შეიძლება ჩაითვალოს, როგორც დამკვეთებსა და დეველოპერებს შორის საკომუნიკაციო ხარვეზების გადაჭრის მექანიზმი, ხოლო DevOps კი ორიენტირებულია დეველოპერებსა და IT ოპერაციებს / ინფრასტრუქტურებს შორის არსებული ხარვეზების აღმოფხვრაზე.

კიდევ ერთხელ აღვნიშნავთ, რომ DevOps მეთოდოლოგია აერთიანებს პროგრამული უზრუნველყოფის დეველოპმენტის (Dev) და ინფორმაციული ტექნოლოგიების ოპერაციებს (Ops) (ნახ.3.12). მისი მიზანია პროგრამული სისტემების შექმნის ციკლის დროის შემცირება და მაღალი ხარისხის პროგრამული უზრუნველყოფის უწყვეტი მიწოდება [63]. DevOps ასევე, ყურადღებას ამახვილებს აგებული პროგრამული უზრუნველყოფის განთავსების (deployment) ამოცანებზე,

DevOps არის გუნდური მუშაობის კონცეფცია, ამიტომაც არ არესებობს ერთი კონკრეტული ინსტრუმენტი მის განსახორციელებლად. პირიქით, არსებობს რამდენიმე ინსტრუმენტის ერთობლიობა („DevOps ინსტრუმენტების ჯაჭვი"), რომლებიც კარგად ასახავს პროგრამული სისტემების დეველოპმენტის და დამკვეთებზე მიწოდების ასპექტებს [63]:

1) Coding – კოდის დეველოპმენტი და ანალიზი, საწყისი კოდის მენეჯმენტის (Source Code Management) ინსტრუმენტი, კოდების შერწყმა [64];

2) Building – უწყვეტი ინტეგრაციის (Continuous Integration) ინსტრუმენტი, კონსტრუირების სტატუსი [65];

3) Testing – უწყვეტი ტესტირების (Continuous Testoing) ინსტრუმენტი, რომელიც უზრუნველყოფს სწრაფ და დროულ უკუკავშირს ბიზნეს რისკების მიხედვით [66];

4) Packaging – არტეფაქტების რეპოზიტორია (მონაცეთა საცავი - data warehouse), აპლიკაციის წინასწარი განთავსება. საცავში ინახება პროგრამული პაკეტები და მათი მეტამონაცემები, ცვლილებათა ჟურნალი სისტემის მენეჯერისათვის [67];

5) Releasing – ცვლილებათა მენეჯმენტი, რელიზის (ვერსიის) გამოცემის დამტკიცება, აპლიკაციის ვერსიის ავტომატიზაცია Application-release automation (ARA) [68];

6) Configuring – ინფრასტრუქტურის კონფიგურაცია და მენეჯმენტი, ინფრა-სტრუქტურა, როგორც კოდის ინსტრუმენტი (Infrastructure as code – IaC) [69];

7) მონიტორინგი – აპლიკაციების მწარმოებლურობის (სწრაფქმედების) მონიტორინგი, მუშაობის გამოცდილება საბოლოო მომხმარებელთან (Application Performance Management – APM). APM ცდილობს კომპლექსური პროგრამების შესრულების პრობლემების გამოვლენას.

პირველი ნაწილის დასკვნა

დასასრულ, შეიძლება ვთქვათ, რომ ობიექტ-ორიენტირებული მოდელირების საფუძველზე კომპიუტერული პროგრამირების მოქნილი და უნიფიცირებული მეთოდოლოგიების კომპლექსური გამოყენება სისტემების ეფექტიანი დეველოპმენტის მნიშვნელოვანი საშუალებაა სინერგეტიკული თვისებებიდან გამომდინარე.

წიგნის პირველ ნაწილში წარმოდგენილია ის თეორიული მასალა და პრაქტიკული მაგალითები, რომლებიც საჭიროა მეორე და მესამე ნაწილების მასალის უკეთ გასაგებად, სადაც გადაწყვეტილია კონკრეტული საპრობლემო სფეროების ფუნქციონალური ამოცანები UML და Agile მეთოდოლოგიების ბაზაზე.

Scrum/Kanban ფრეიმვორკებს აქვს გამოყენებითი პროგრამული უზრუნველყოფის განვითარებისთვის დიდი შესაძლებლობები, რაც დღეისათვის ფართოდ გამოიყენება Agile და DevOps-ის ამოცანების გადასაწყვეტად და მეტად აქტუალურია. ისინი ერთმანეთს ავსებენ.

ნაწილი 2
ბიზნესპროცესების მოდელირების ტექნოლოგიები
თავი 4
კორპორაცული მენეჯმენტის სისტემების ბიზნესპროცესების მოდელირების პრობლემები და ახალი ინფორმაცული ტექნოლოგები

4.1. კორპორაციული მენეჯმენტის ბიზნესპროცესები – მართვის რთული და განაწილებული სისტემა

კორპორაცია მრავალპროფილიანი, რთული, ტერიტორიულად განაწილებული სისიტემაა, რომელიც შედგება მრავალი დეპარტამენტის, სამმართველოს თუ სხვა დანაყოფისაგან და ხასიათდება დასაქმებულ თანამშრომელთა მაღალი რიცხვით. კორპორაციული მენეჯმენტი, ფაქტობრივად, მრავალი სხვა მართვის სისტემის ერთობლიობა, მათი ინტეგრაციაა, ეს იქნება რესურსების მართვის, ინფორამციული მართვის, ფინანსების მართვის, შრომითი რესურსების მართვის თუ სხვა სისტემები და ა.შ. რომელთა მდგრადი, ეფექტიანი მუშაობის საწინდარია ინფორმაციული ტექნოლოგიების დანერგვა და განვითარება.

კორპორაციული სისტემების მენეჯმენტი ერთ-ერთ აქტუალური საკითხია, ვინაიდან დღესდღეობით არსებული მრავალი ორგანიზაცია კორპორაციას წარმოადგენს და ასევე მრავალი სახელმწიფო სამსახურიც თამამად შიძლება მივაკუთნოთ ამ ტიპის სისტემას, რომელთა ეფექტური მართვა კორპორაციული სისიტემებისათვის დამახასიათებელ მენჯმენტს მოითხოვს, განსაკუთრებით კი ინფორამციული უზრუნველყოფის საკითხში, ეს იქნება თანამედროვე ტქნოლოგიების დანერგვა, შრომითი რესურსების სწორი გადანაწილება, ფინანსური საკითხები თუ სხვა.

ბიზნეს-კონკურენტულ გარემოში ნებისმიერი კორპორაციის (დიდი თუ მცირე ორგანიზაციის\საწარმოს) წარმატებული ფუნქციონირებისათვის აუცილებელია და მნიშვნელოვან კრიტიკულ წერტილს წარმოადგენს კორპორაციული ინფორმაციის (მმართველობითი, ფინანსური, იურიდიული, მარკეტინგული და სხვ.) და ინტელექტუალური რესურსების (კორპორაციის

ადმიანური, ტექნიკურ-ტექნოლოგიური და პროგრამულ-ინფორმაციული რესურსები) სრული ფლობა და შესაბამისად მათი ოპერატიული მართვა, რაც შესაბამისად კორპორაციაში მიმდინარე მრავალრიცხოვანი ბიზნეს-პროცესის ეფექტურ დაგეგმვას, მართვის საშუალებების შექმნას და განხორციელებას მოიცავს.

კორპორაციული მენეჯმენტის საინფორმაციო სისტემები მიეკუთვნება დიდი და რთული სისტემების კლასს. კორპორაცია მრავალი საწარმოსა და ორგანიზაციისაგან შედგება, რომელთაც გააჩნია დამოუკიდებლობის საკმაოდ მაღალი ხარისხი/დონე და ამავდროულად, ორიენტირებულია კონკრეტული მიზნების შესრულებაზე.

ამდენად, აუცილებელი ხდება კორპორაციული სისტემის ორგანიზაციული მართვის სრულყოფა თანამედროვე ინფორმაციული ტექნოლოგიების გამოყენებით, რომელიც შესაძლებელს გახდის უზრუნველყოფილ იქნას კომპანიების ადამინური რესურსების ოპტიმალური გამოყენება, ნებისმიერი ფუნქციონალური რგოლის ეფექტური მუშაობა, მართვის ახალი მეთოდების, პროცესების ავტომატიზაციის, ვიზუალიზაციის (მოდელირების), დინამიკისა და ოპტიმიზაციის, მონაცემთა ცოდნის ბაზების მართვისა და ქსელური სერვისების კომპლექსური გამოყენებით.

ღრუბლოვანი და მობილური ტექნოლოგიების განვითარებამ მიაღწია ისეთ დონეს, რომ დიდ კორპორაციებს და, განსაკუთრებით საშუალო და მცირე ბიზნესის საწარმოებს უკვე საშუალება აქვს დიდი ინვესტიციების გარეშე მიიღოს სრულყოფილი მხარდაჭერა ახალი ციფრული ტექნიკისა და ტექნოლოგიების გამოყენებით.

შესაძლებელი ხდება მენეჯმენტის პროცესების ინტეგრირებული სისტემების შექმნა და მათი ეფექტიანი გამოყენება თითქმის ყველა დარგის ობიექტებზე. ინტერნეტული რესურსების საფუძველზე და ასეთი ინტეგრირებული სისტემების პროგრამული აპლიკაციების გამოყენებით, შესაძლებელი ხდება მონაცემთა ოპერატიული ანალიზის ბაზებისა და გრძელვადიანი ინფორმაციული საცავების ეკოსისტემის განვითარება, რაც ხელს შეუწყობს კორპორაციათა და საწარმოთა დინამიკური ანალიზის და პროგნოზის ამოცანების გადაწყვეტას.

4.2. საინფორმაციო სისტემების დაპროექტების და მოდელირების თანამედროვე მეთოდოლოგია

4.2.1. UML/2 ტექნოლოგიის არსი და მისი გამოყენება

მოდელირების უნიფიცირებული ენა - UML (Unified Modeling Language) შექმნილია, როგორც უნივერსალური მოდელირების ენა ობიექტ-ორიენტი-რებული პროგრამირების სფეროში და არის სტანდარტული ვიზუალური მოდელირების ენა, რომელიც იძლევა საშუალებას სისტემა აღიწეროს გრაფიკულად და ტექსტურად. იგი გამოიყენება სხვადასხვა სისტემის მოდელირებისთვის, როგორიცაა პროგრამული უზრუნველყოფის სისტემები, ბიზნეს სისტემები და სხვა ნებისმიერი სისტემა, მაგალითად წარმოების სექტორებში, სამხედრო თუ საინჟინრო საქმეში და ა.შ.

UML მეთოდოლოგია, როგორც წინა ნაწილში აღვნიშნეთ, არის ბუჩის მეთოდის (Booch Method), ობიექტის მოდელირების ტექნიკის (Object-modeling technique – OMT) და ობიექტ-ორიენტირებული პროგრამული უზრუნველყოფის ინჟინერიის (Object-oriented software engineering – OOSE) სინთეზით მიღებული და გვევლინება როგორც ერთიანი, ფართოდ გამოყენების მოდელირების ენა [14].

UML აერთიანებს მონაცემთა მოდელირების (entity relationship diagrams) ბიზნეს მოდელირების (work flows), ობიექტების და კომპონენტების მოდელირების მეთოდებს. ის გამოიყენება პროგრამული უზრუნველყოფის და მისი ტექნიკური რეალიზაციის მთელი სასიცოცხლო ციკლის განმავლობაში. UML-ის მიზანია იყოს სტანდარტული მოდელირების ენა, რომლის საშუალებითაც შესაძლებელია პარალელური და განაწილებული სისტემის მოდელების შექმნა [19].

UML მეთოდოლოგია დიდი პროექტების კარგი მხარდაჭერის საშუა-ლებაა. დროთა განმავლობაში იგი ვითარდება და UML1-ის გაფართოებით შეიქმნა UML2-ის ახალი ვერსიები.

როგორც აღვნიშნეთ, UML2-ში დამატებული იქნა ახალი მეტაკლასები: კავშირი, ურთიერთქმედება, კავშირის დასასრული, მექანიზმი (device), განლაგების სპეციფიკაცია, შესრულების გარემო (execution environment), შემთხვევითი მოქმედების მიღება, ობიექტის მოქმედების გაგზავნა, მოქმედების სტრუქტურული მახასიათებელი, აქტიურობის დასასრული, ცენტრალური ბუფერული კვანძი, მონაცემთა საცავები, ნაკადთა დასასრული, წყვეტადი

რეგიონები, პარამეტრი, პორტი, ქცევა, ქცევის კლასიფიკატორი, ხანგძლოვობა, ინტერვალი, დროის განსაზღვრა, კომბინირებული ფრაგმენტი, ინტერაქტიული ფრაგმენტი, ინტერაქტიული გამოყენება, შემთხვევის დაბლოკვა (განადგურება), განზოგადება და ა.შ. ბევრი სტერეოტიპი ამოღებულ იქნა სტანდარტული UML პროფილიდან, მაგალითდ: «destroy», «facade», «friend», «profile», «requirement», «table», «thread» [28].

UML2.0-ის ოთხი უმთავრესი დოკუმენტაციაა:

სუპერსტრუქტურა. სუპერსტრუქტურა იყოფა 6 ძირითად დიაგრამად (3 ქცევის და 4 ურთიერთქმედების დიაგრამა) და მათში შემავალი ელემენტე-ბისაგან.

ინფრასტრუქტურა. UML2.0 – ინფრასტრუქტურა განსაზღვრავს საბაზისო კლასებს, რომელიც ქმნის საფუძველს არამარტო UML2.0-ის სუპერსტრუქ-ტურისთვის, არამედ MOF 2.0 – ისთვისაც.

დიაგრამების ურთიერთქმედება (ურთიერთჩანაცვლება). UML2.0 – ის ურთიერთქმედება – ეს სპეციფიკაცია აფართოებს UML - ის მეტამოდელს დამატებითი პაკეტით - გრაფიკულ-ორიენტირებული ინფორმაციით, რომელიც მოდელებს აძლევს საშუალებას ჩაენაცვლონ ერთმანეთს, შენახულ ან აღდგენილ იქნან და წარმოგვიდგნენ საწყის მდგომარეობაში.

შეზღუდვების ობიექტების ენა (Object Constraint Language). UML2.0 - ის შეზღუდვების ობიექტების ენა - ეს არის ენა, რომელიც მოქმედებების და შესრულებადი კოდების დაწერისათვის კი არა, არამედ შეზღუდვების და მოთხოვნების განსაზღვრისთვის არის განკუთვნილი.

UML2.0-is დადებითი მხარეებია: ახალი სტრუქტურა; არქიტექტურული მოდელირების კონსტრუქციები; პორტები, კავშირები და ნაწილები; ახალი UML2.0-დიაგრამები; სტრუქტურის შემადგენლობის დიაგრამა; ქცევის დიაგრამების UML2.0-განახლება; მდგომარეობის დიაგრამის განახლება; ურთიერთქმედების დიაგრამის განახლება; აქტიურობის დიაგრამის განახლება; UML - პროფილი.

4.2.2. UML/2-ის ახალი დიაგრამები

UML 2.5 - ს აქვს 15 ტიპის დიაგრამა, რომელიც იყოფა 2 კატეგორიად. აქედან 7 დიაგრამა წარმოგვიდგენს სტრუქტურულ ინფორმაციას, ხოლო დანარჩენი 8 ქცევის საერთო ტიპებს, მათ შორის 4 სახის დიაგრამა ასახავს ურთიერთქმედების სხვადასხვა ასპექტს. ამ დიაგრამების სტრუქტურა შეიძლება გამოვსახოთ შემდეგი კლასების UML დიაგრამით (ნახ.4.1).

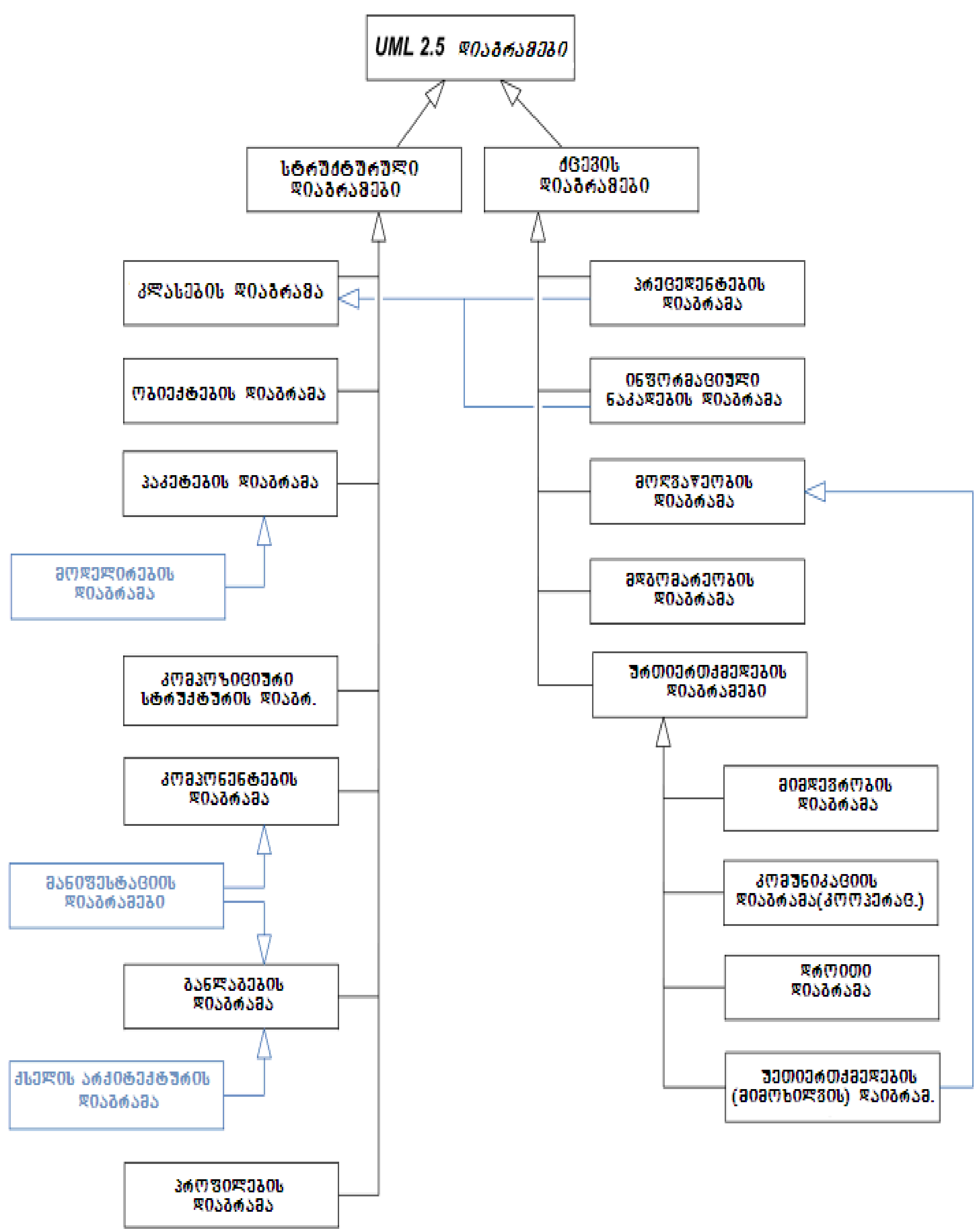
UML არ აწესებს მისი ელემენტების ასახვას რომელიმე ერთი კონკრეტული ტიპის დიაგრამისთვის. ზოგადად, ყველა ელემენტი შეიძლება შეგვხვდეს დიაგრამების თითქმის ყველა ტიპში. ეს მოქნილობა ნაწილობრივ შეზღუდულია UML/2.0-ში. UML-ის პროფილებმა შეიძლება განსაზღვროს დამატებითი დიაგრამის ტიპები ან გააფართოვოს უკვე არსებული დიაგრამები დამატებითი აღნიშვნებით.

საინჟინრო ხაზვის ტრადიციის მიხედვით UML-დიაგრამებში შესაძლებე-ლია კომენტარების და შენიშვნების მითითება, ასევე შეზღუდვის ან მიმართულების ჩვენება.

დიაგრამების სტრუქტურა ახდენს იმის ასახვას, რაც წამოდგენილი უნდა იყოს იმ სისტემაში, რომლის მოდელირებასაც ვახორციელებთ.

ვინაიდან სტრუქტურული დიაგრამა წარმოგვიდგენს სტრუქტურას, იგი ასევე ფართოდ გამოიყენება პროგრამული სისტემების პროგრამული არქიტექტურის ასაგებად. სისტემის პროგრამული არქიტექტურა კი არის იმ საჭირო სტრუქტურების ნაკრები სისტემის აღწერისათვის, რომელიც მოიცავს პროგრამულ ელემენტებს, მათ შორის კავშირებს და მათ თვისებებს.

კლასების დიაგრამა: კლასების დიაგრამა აღწერს სისტემის სტრუქტურას სისტემის კლასების, მისი ატრიბუტების, ოპერაციების და მათ შორის კავშირების ჩვენებით. კლასების დიაგრამა არის ობიექტ-ორიენტირებული პროგრამირების მთავარი სამშენებლო ბლოკი. იგი გამოიყენება, როგორც სისტემური აპლიკაციების კონცეპტუალური მოდელირების ასაგებად, ასევე მოდელების პროგრამულ კოდში გადატანის მიზნით.



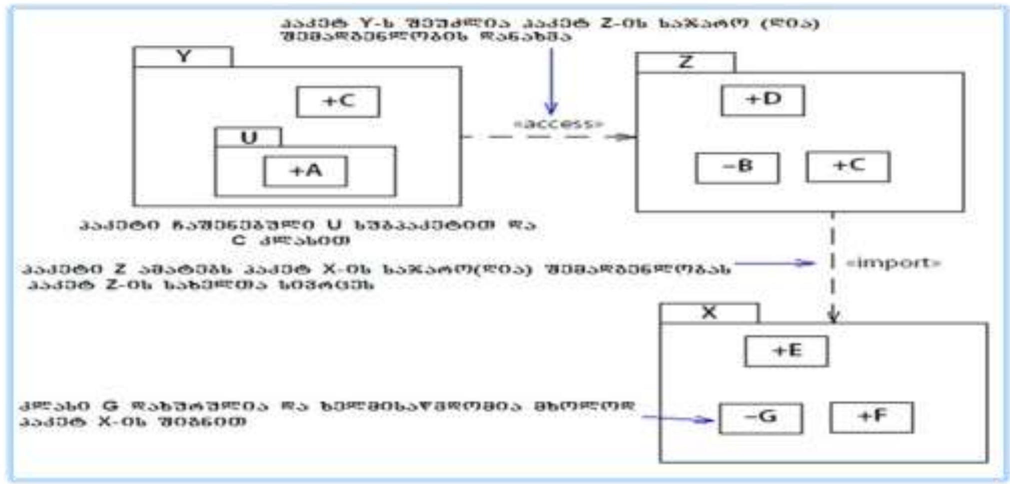
ნახ. 4.1. UML/2.5-ის დიაგრამები

მოდელირება (Modeling). UML დიაგრამები გვიჩვენებს სისტემის 2 მხარეს:

- სტატიკური (ანუ სტრუქტურული) – რომელიც გვიჩვენებს სისტემაში გამოყენებული ობიექტების, ატრიბუტების, ოპერაციების და კავშირების სტრუქტურას. სისტემის სტრუქტურული მხარე მოიცავს კლასების დიაგრამას (Class Diagram) და კომპოზიციური სტრუქტურის დიაგრამას (Composite Structure Diagrams).
- დინამიკური (ანუ ქცევითი) – წარმოგვიდგენს სისტემის ქცევის დინამიკას ობიექტებს შორის ურთიერთქმედების და ობიექტების შიგა მდგომარეობის (შემადგენლობის) ცვლილებების ჩვენებით. დინამიკური მხარე მოიცავს მიმდევრობითობის (Sequence Diagrams), აქტიურობის (Activity Diagrams) და მდგომარეობის (State Machine Diagrams) დიაგრამებს.

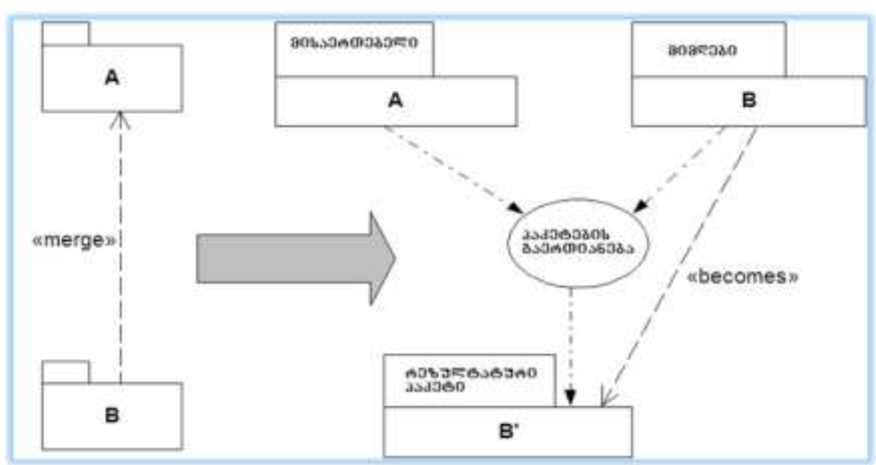
პაკეტების დიაგრამა (package diagram) – UML მოდელში გამოხატავს პაკეტებს შორის დამოკიდებულებას, რომლებიც ქმნის მოდელს. სტანდარტებს შორის, გარდა სტანდარტული დამოკიდებულებისა, არსებობს პაკეტების დამოკიდებულების 2 ტიპი: პაკეტების იმპორტი და პაკეტების გაერთიანება.

პაკეტის იმპორტი არის დამოკიდებულება იმპორტიორ სახელების სიმრავლესა და პაკეტებს შორის, რომელიც მიუთითებს იმაზე, რომ იმპორტიორი სახელთა სივრცე ამატებს იმპორტირებული პაკეტის წევრების სახელებს საკუთარ სახელთა სივრცეში. დუმილის რეჟიმში ურთიერთობა პაკეტებს შორის ინტერპრეტირდება, როგორც დამოკიდებულება „პაკეტის იმპორტი“ (ნახ.4.2).



ნახ. 4.2. პაკეტების წვდომა და იმპორტი

პაკეტების გაერთიანება არის ორ პაკეტს შორის ისეთი დამოკიდებულება, რომელიც უზრუნველყოფს ამ პაკეტების შემადგენლობის გაერთიანებას. ეს მოვლენა წააგავს “განზოგადებას”, იმ გაგებით, რომ ერთი ელემენტი ითავისებს მეორე ელემენტის თვისებებსაც. შედეგად კი ვიღებთ ერთ ელემენტს, რომელსაც ორივე ელემენტის თვისება გააჩნია (ნახ.4.3).



ნახ. 4.1 პაკეტების გაერთიანების შინაარსობრივი წარმოდგენა

გამოყენება. პაკეტები UML-ში გამოიყენება ელემენტების დასაჯგუფებლად, ასევე უზრუნველყოფს საერთო სახელების სიმრავლეს ამ ელემენტებისათვის. ერთი პაკეტი შეიძლება მოიცავდეს სხვა პაკეტებსაც.

თითქმის ყველა UML ელემენტი შეიძლება დაჯგუფდეს პაკეტში ანუ კლასები, ობიექტები, პრეცენდენტები, კომპონენტები, კვანძები და ა.შ. შეიძლება ორგანიზებული იყოს, როგორც ერთი პაკეტი.

ფუნქციური მოდელების ორგანიზებისას (პრეცედენტების, ბიზნეს-პრცესების მოდელი) პაკეტები გამოიყენება იმ სისტემის რეალური მოდულური სტრუქტურის მოდელის ასაგებად, რომლის მოდელირებასაც ვახდენთ. საწყისი კოდის ორგანიზებისას კი პაკეტები გვიჩვენებენ ამ კოდის სხვადასხვა ასპექტებს, (მხარეს, „შრეს“), მაგალითად: პრეზენტაციის შრე (presentation layer); კონტროლერის შრე (controller layer); მონაცემთა ხელმისაწვდომობის შრე (data access layer); ინტეგრირების შრე integration layer); ბიზნეს-მომსახურებების შრე (business services layer).

პაკეტები გამოიყენება კომპონენტების მოდელის აგებისას კომპონენტების დასაჯგუფებლად კუთვნილების და/ან განმეორებითობის მიხედვით. მაგალითად:

- ✓ commercial-off-the-shelf (COTS) products – „გამოსაყენებლად მზა პროდუქტი" არის პროგრამული სისტემა, რომელიც შეიძლება სხვადასხვა მომხმარებლის მოთხოვნილებებზე ისე იქნას ადაპტირებული, რომ არ მოხდეს სისტემის საწყისი კოდის ცვლილება;
- ✓ open-source framework components - ღია კოდის სტრუქტურული კომპონენტები;
- ✓ custom-built framework components - მომხმარებლის moTxovnis შესაბამისი struqturული komponentebi;
- ✓ custom-built application components - მომხმარებლის დაკვეთის შესაბამისი გამოყენებადი (აპლიკაციის) კომპონენტები.

პაკეტებს ვიყენებთ, მაშინ როდესაც განლაგების მოდელებს ვუკეთებთ ორგანიზებას განლაგების მდგომარეობის სხვადასხვა ტიპების გამოსახატავად, მაგალითად: production environment - დამუშავების გარემო (მდგომარეობა); pre-production environment - დამუშავებამდე არსებული გარემო; integration test environment – ინტეგრაციის ტესტირების გარემო; system test environment - სისტემის ტესტირების გარემო; development environment - განვითარების გარემო.

პაკეტების დიაგრამა არის UML-ის სტრუქტურული დიაგრამა, რომელიც გვიჩვენებს დასაპროექტებელი სისტემის სტრუქტურას პაკეტების ეტაპზე. პაკეტების დიაგრამის შესადგენად გამოიყენება შემდეგი ელემენტები: პაკეტი (package), მაპაკეტირებელი ელემენტი packageable element), დამოკიდებულება (dependency), პაკეტების იმპორტი package import) და პაკეტების გაერთიანება (package merge).

მაპაკეტირებელი ელემენტი არის სახელწოდებული ელემენტი (named element), რომელიც შეიძლება უშუალოდ ეკუთვნოდეს პაკეტს.

პაკეტი არის სახელსივრცე, რომელიც გამოიყენება ისეთი ელემენტების დასაჯგუფებლად, რომლებიც შინაარსობრივად მონათესავეებია და შესაბამისად, შესაძლებელია, რომ ერთმა ელემენტმა შეცვალოს მეორე. ეს არის

მთავარი მექანიზმი, რომელიც ელემენტების ერთად დაჯგუფებით გვაძლევს დასაპროექტებელი სისტემის უკეთეს მოდელს.

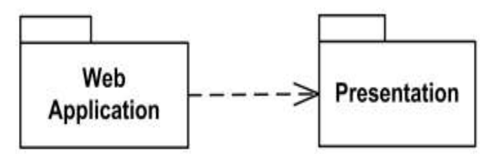
პაკეტის კუთვნილი ყველა ელემენტი უნდა წარმოადგენდეს მაპაკეტირებელ ელემენტს. თუ პაკეტს ამოვიღებთ მოდელიდან, შესაბამისად ამოღებულ იქნება ყველა მისი კუთვნილი ელემენტიც. თითოეული პაკეტი თავად წარმოადგენს მაპაკეტირებელ ელემენტს. ამგვარად, ნებისმიერი პაკეტი შეიძლება იყოს სხვა პაკეტის ერთ-ერთ წევრი.

ვინაიდან პაკეტი არის სახელსივრცე, მონათესავე ელემენტებს ან იგივე ტიპის ელემენტებს უნდა ჰქონდეს უნიკალური სახელები, რომლებიც მათ განასხვავებს სხვა პაკეტებისაგან. სხვადასხვა ტიპის ელემენტებს კი შეიძლება ჰქონდეთ ერთი და იგივე სახელი.

ისევე როგორც სახელსივრცეებს, პაკეტს შეუძლია როგორც პაკეტის ინდივიდუალური წევრების იმპორტირება, ასევე სხვა პაკეტის ყველა წევრის იმპორტიც. პაკეტი ასევე შეიძლება გაერთიანდეს სხვა პაკეტთან.

პაკეტების იმპორტი (package import) არის მიმართება იმპორტიორ სახელსივრცესა და იმპორტირებულ პაკეტს შორის, რომელიც აძლევს პაკეტს სხვა სახელსივრციდან შეუზღუდავი სახელების გამოყენების საშუალებას პაკეტის ელემეტების ასაღწერად. იმპორტიორი სახელსივრცე საკუთარ სახელსივრცეში ამატებს იმპორტირებული პაკეტის წევრების სახელებს. შინაარსობრივად პაკეტის იმპორტი არის ელემენტების იმპორტის ექვი-ვალენტური, თითოეული იმპორტირებული სახელსივრცის ინდივიდუალური წევრისათვის, თუ უკვე არ არსებობს ცალკე აღწერილი ელემენტის იმპორტი.

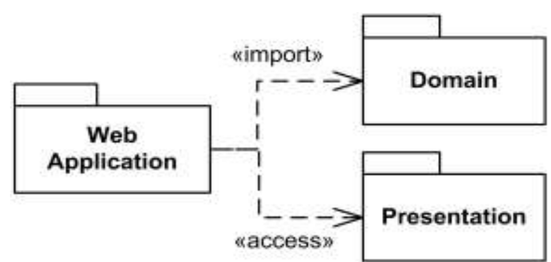
პაკეტის იმპორტი 4.4-ნახაზზე მოცემულია წყვეტილი ისრის საშუალებით, იმპორტიორ პაკეტის სივრციდან იმპორტირებული პაკეტის მიმართულებით.



ნახ.4.4. ვებ-აპლიკაცია აიმპორტებს პრეზენტაციას

ვიზუალურად პაკეტის იმპორტი შეიძლება იყოს ღია (public) ან დახურული (private). თუ პაკეტის იმპორტი ღიაა, მაშინ იმპორტირებული ელემენტები დაემატება სახელთა სივრცეს და იქნება ხილული სახელთა სივრცის გარეთაც, ხოლო თუ ის არის დახურული, მაშინ იმპორტირებული ელემენტები კვლავ დაემატება სახელთა სივრცეს, მაგრამ იქნება უხილავი.

4.5 ნახაზზე ისართან ახლოს მიეთითება სიტყვიერი აღნიშვნა (გასაღებური სიტყვა), თუ რა სახის იმპორტს განეკუთვნება იგი. პაკეტის ღია იმპორტის ასაღნიშნად შემოღებულია სიტყვა - „import“, ხოლო პაკეტის დახურული იმპორტის ასაღნიშნად - „access“. დუმილის რეჟიმში ანუ აღნიშვნის გარეშე იგულისხმება რომ პაკეტების იმპორტი არის საჯარო.



ნახ. 4.5. მოცემულია პაკეტი “Domain”-ის ღია იმპორტი და პაკეტი “Presentation”-ის დახურული იმპორტი

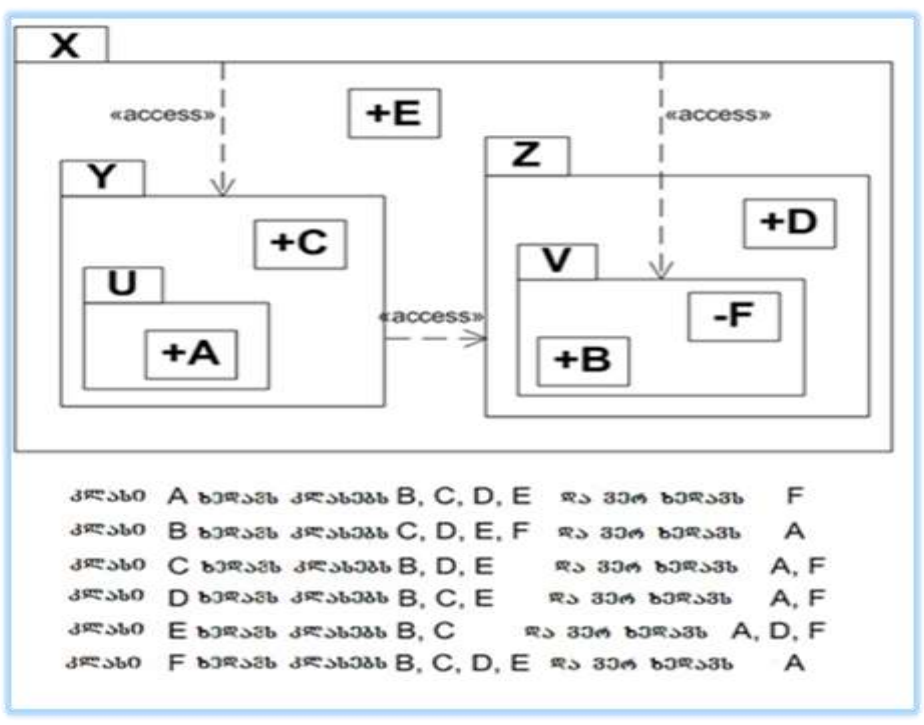
სახელმინიჭებული ელემენტი არის აბსტრაქტული ელემენტი, რომელსაც შეიძლება ქონდეს გარკვეული სახელი. იგი გამოიყენება ზემოხსენებული ელემენტის იდენტიფიცირებისათვის სახელთა სიმრავლეში, რომელშიც აღწერილი და ხელმისაწვდომია. სახელწოდებული ელემენტი საშუალებას იძლევა იყოს ანონიმური ანუ არ ჰქონდეს სახელი. სახელის არქონა შემოღებულია იმისთვის, რომ განასხვავონ ისეთი ელემენტისგან, რომელსაც აქვს ცარიელი (უშინაარსო, empty) სახელი. ამგვარად UML-ს აქვს შემდეგი სახის ვიზუალური მხარეები: Public – საჯარო; Package – შესაფუთი (პაკეტი); Protected – დამცველობითი; Private – კერძო.

თუ სახელწოდებული ელემენტი არ არის საკუთრება რომელიმე სახელსივრცის, მაშინ მას არ აქვს ვიზუალური მხარე.

პაკეტების გაერთიანება (Package merge) არის მიმართება, რომელიც მიუთითებს პაკეტის შემადგენლობის გაფართოებაზე, მეორე პაკეტის შემადგენლობის ხარჯზე. პაკეტების გაერთიანება არის განზოგადების (generalization) იდენტური, იმ მნიშვნელობით, რომ საწყისი ელემენტი კონცეპტუალურად (შინაარსობრივად) უმატებს საკუთარ თვისებებს მიზნობრივი ელემენტის თვისებებს.

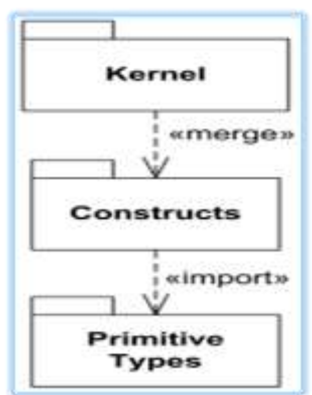
შედეგად კი, ვიღებთ ელემენტს, რომელიც აერთიანებს ორივე ელემენტის თვისებებს. პაკეტების გაერთიანება შეიძლება განვიხილოთ, როგორც ოპერაცია, რომელიც ორი ელემენტის შემადგენლობას აერთიენებს ერთ პაკეტში.

ეს მექანიზმი შეიძლება გამოვიყენოთ, მაშინ როდესაც ელემენტები, რომლებიც ერთიანდებიან ერთ პაკეტში, აქვთ ერთი და იგივე სახელი და განკუთვნილია ერთი და იმავე შინაარსის გამოსახატავად (ნახ.4.6).



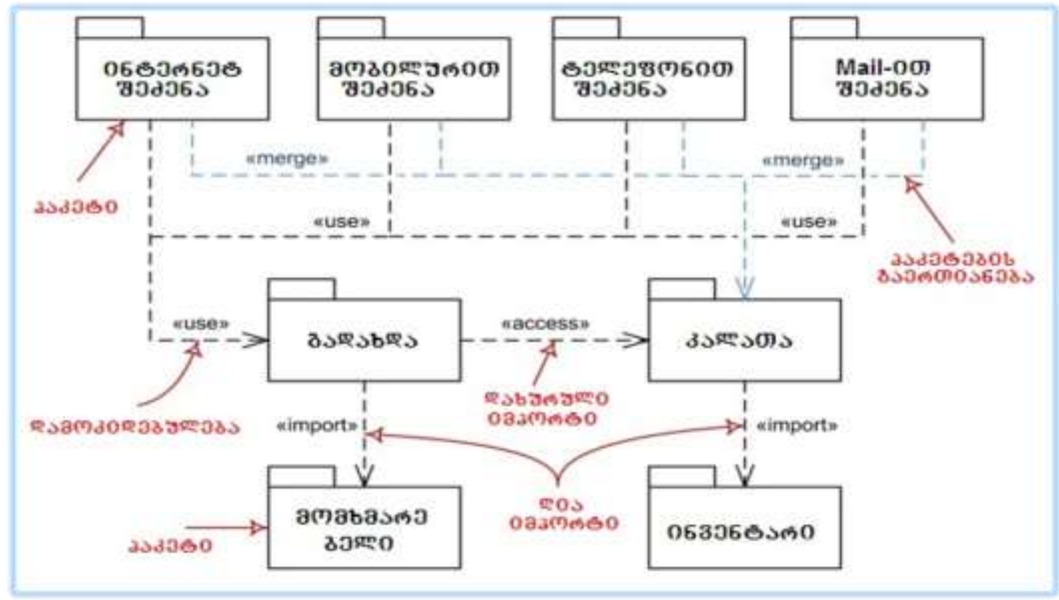
ნახ. 4.6. პაკეტების და კლასების ხილვადობა

პაკეტების გაერთიანება განსაკუთრებით გამოსადეგია მეტა-მოდელირების დროს და ფართოდ გამოიყენება მეტა-მოდელის განსასაზღვრის მიზნით. პაკეტების გაერთიანება გამოისახება წყვეტილი ისრით მიმღები პაკეტიდან გამაერთიანებელი პაკეტისაკენ (ნახ.4.7).

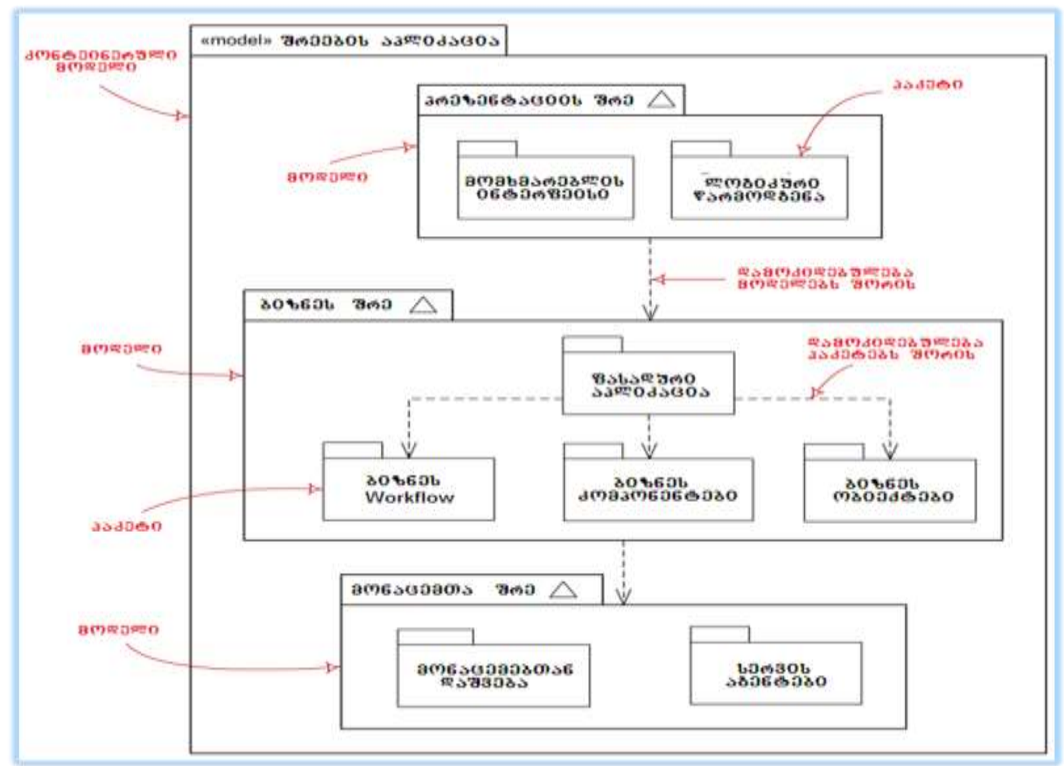


ნახ.4.2 პაკეტი “kernel” ერთიანდება პაკეტ “Constructs”-თან, რომელიც აიმპორტებს პაკეტს “Constructs”

პაკეტების დიაგრამის თვალსაჩინო მაგალითი მოცემულია 4.8 ნახაზზე, ინტერნეტ-შესყიდვების მაგალითზე. UML-ის დამხმარე სტრუქტურული დიაგრამა არის პაკეტებისაგან შემდგარი მოდელის დიაგრამა, რომელიც წარმოგვიდგენს სისტემის განსხვავებულ ან სპეციალურ მხარეებს, სისტემის არქიტექტურული, ლოგიკური ან ქცევითი ასპექტების გამოსახატავად.



ნახ.4.3 პაკეტების იმპორტი და გაერთიანება ინტერნეტ-შესყიდვის მაგალითზე



ნახ. 4.4. მოდელი

პროფილების დიაგრამა (Profile Diagram) - არის სტრუქტურული დიაგრამა, რომელიც აღწერს გაფართოების მარტივ UML მექანიზმს, მომხმარებლის სტერეოტიპების, მონიშნული მნიშველობების და შეზღუდვების საშუალებით. როგორც ვიცით, სტრუქტურული დიაგრამა წარმოგვიდგენს სისტემის სტატიკურ სტრუქტურას, მის ნაწილებს სხვადასხვა აბსტრაქციის და განვითარების დონეზე და ამ ნაწილების ერთმანეთთან კავშირს.

პროფილები UML - მეტამოდელის ადაპტირების საშუალებას იძლევა სხვადასხვა დომენებისთვის (როგორიცაა J2EE ან .NET) და პლატფორმებისთვის (როგირიცაა რეალურ დროში ან ბიზნეს-პროცესების მოდელირება)). მაგალითად, სტანდარტული UML მეტამოდელის ელემენტების სემანტიკა შეიძლება სპეციალიზებულ იქნას პროფილში.

პროფილების მექანიზმი არ არის გაფართოების საუკეთესო მექანიზმი. ის არ იძლევა არსებული მოდელების მოდიფიცირების ან ახალი მეტამოდელის შექმნის საშუალებას. პროფილი მხოლოდ არსებული მეტამოდელის ადაფტირების ან მორგების საშუალებას იძლევა კონსტრუქციებით, რომელიც

არის სპეციფიური კონკრეტული დარგისთვის, პლატფორმისთვის ან მეთოდისთვის. შეუძლებელია მეტამოდელში გამოიყენებული რომელიმე შეზღუდვის ამოღება, მაგრამ შესაძლებელია ახალი შეზღუდვების დამატება, რომელიც სპეციფიური იქნება ამ პროფილისთვის.

მარტივად რომ ვთქვათ, UML-პროფილი უზრუნველყოფს UML-მოდელების აგების საერთო მექანიზმს კონკრეტულ სფეროში. იგი დაფუძ-ნებულია დამატებით სტერეოტიპებზე და მონიშნულ მნიშვნელობებზე (tagged value), რომელიც იყენებს ელემენტებს, ატრიბუტებს, მეთოდებს, ბმულებს, ბმულების დასასრულებს და ა.შ. პროფილი არის გარკვეული შეზღუდვების ნაკრები, რომლებიც ერთიანობაში აღწერს მოდელის რომელიმე კონკრეტულ პრობლემას და აადვილებს ამ ნაწილში მოდელის მშენებლობას.

პროფილში მოცემულია მეტამოდელის თვისებები, რომელიც შემდგომ პაკეტში გამოიყენება. სტერეოტიპები არიან სპეციფიური მეტაკლასები, მონიშნული მნიშვნელობები - სტანდარტული მეტაატრიბუტები, ხოლო პროფილი არის პაკეტის სპეციფიური ტიპი.

პროფილები შეიძლება დინამიკურად იქნას გამოყენებული მოდელში ან უარყოფილი იყოს მოდელისაგან. ისინი ასევე შესაძლებელია დინამიურად იქნას კომბინირებული, ისე რომ რამდენიმე პროფილი ერთსა და იმავე დროს შეიძლება ერთსა და იმავე მოდელში იქნას გამოყენებული.

პროფილები თავიდან UML4.X – ში გამოჩნდა, პროფილების დიაგრამები წარმოდგენილ იქნა UML 2.0 – ში, ხოლო მისი პირველი ოფიციალური ტაქსონომია (სპციფიკაცია) UML 2.2 – ში გაჩნდა.

პროფილის გრაფიკული გამოსახვის საშუალებებია: პროფილი (profile), მეტაკლასი (metaclass), სტერეოტიპი (stereotype), გაფართოება (extension), გზავნილი (reference), პროფილის აპლიკაცია (profile application).

პროფილი (Profile). პროფილი არის პროფილის პაკეტი, რომელიც წარმოდგენილ მეტამოდელს (როგორიცაა UML) აფართოვებს მეტამოდელის ადაპტირებით და კლიენტის მოთხოვნილებაზე მორგების კონსტრუქციების საშუალებით. იგი სპეციფიურია კონკრეტული სფეროსთვის, პლატფორმისთვის თუ პროგრამული უზრუნველყოფის სფეროსთვის. სხვა სიტყვებით რომ ვთქვათ, პროფილი წარმოადგენს UML სტანდარტის გაფართოების მარტივ მექანიზმს [19].

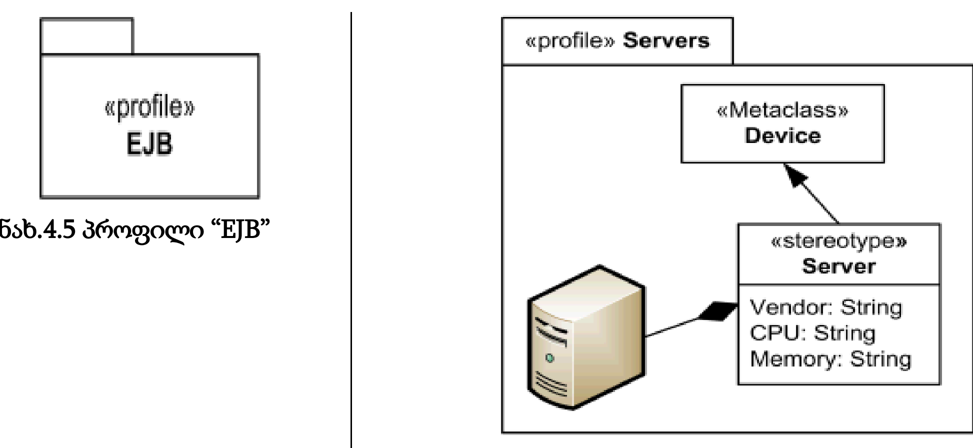
პროფილი წარმოადგენს გარკვეულ აკრძალვებს ან შეზღუდვებს ჩვეულებრივი მეტამოდელირებისას, ამ პაკეტში აღწერილი მეტაკლასების

გამოყენების საშუალებით. ძირითადი გამაფართოებული კონსტრუქცია არის სტერეოტიპი, რომელიც აღწერილია როგორც პროფილის ნაწილი და აფართოვებს გარკვეულ მეტაკლასებს.

პროფილი არის მეტამოდელის განსაზღვრული ფორმა, რომელიც ყოველთვის დაკავშირებული უნდა იყოს გარკვეულ წარმოდგენილ მეტამოდელებთან. დაუშვებელია განცალკევებული პროფილის განსაზღვრა მასთან დაკავშირებული მეტამოდელის გარეშე.

იგი იყენებს იმავე წერილობით აღნიშვნებს, რასაც პაკეტი, გასაღებური სიტყვის - „profile" პაკეტის სახელის ზემოთ ან წინ დამატებით (ნახ.4.10).

პროფილი შეიძლება განსაზღვრავდეს კლასებს, სტერეოტიპებს, მონაცემთა ტიპებს, პრიმიტიულ ტიპებს (primitive types) ან ჩამონათვალს (ნახ. 4.11).



ნახ.4.5 პროფილი "EJB"

ნახ.4.6 პროფილი "servers"

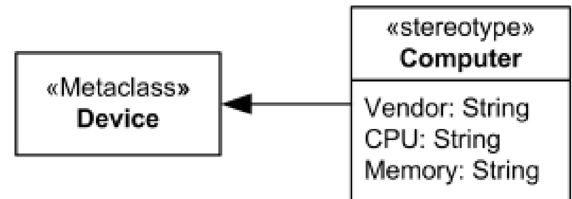
ერთმა პროფილმა შეიძლება მრავალჯერ, სრულად გამოიყენოს სხვა პროფილი ან მისი რაღაც ნაწილი, რათა გააფართოვოს უკვე არსებული პროფილი. ასევე, მრავალი პროფილი შეიძლება გამოყენებულ იქნას ერთსადა-იმავე მოდელში.

მეტაკლასი (ნახ.4.12) არის პროფილის კლასი და მაპაკეტირებელი ელემენტი, რომელიც შეიძლება გაფართოვებულ იქნას ერთი ან რამდენიმე სტერეოტიპის საშუალებით.

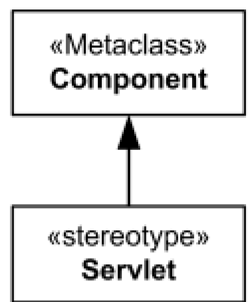
«Metaclass»
Component

ნახ. 4.7. მეტაკლასი "Component"

მეტაკლასი შეიძლება გაფართოვებულ იქნას ერთ ან რამდენიმე სტერეოტიპად, სპეციალური სახეობის ასოციაციის – გაფართოვების გამოყენებით (ნახ.4.13, ნახ.4.14).



ნახ. 4.8. მეტაკლასი "Device" გაფართოებულია სტერეოტიპი "Computer"-ის საშუალებით

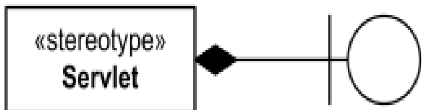


ნახ.4.9 სტერეოტიპი "servlet" აფართოებს "components"

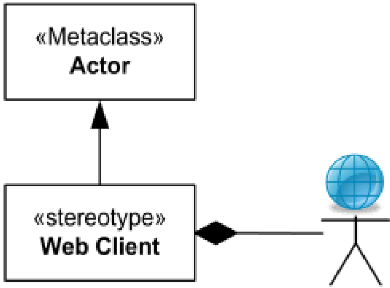
სტერეოტიპი არის პროფილის კლასი, რომელიც გვიჩვენებს, თუ როგორ შეიძლება გაფართოვდეს არსებული მეტაკლასი, როგორც პროფილის ერთ-ერთი ნაწილი. იგი გარკველი სფეროს (დომენის) იმ ტერმინოლოგიის ან აღნიშვნის დამატებით ან ჩანაცვლებით გამოყენების საშუალებას იძლევა, რომელიც გაფართოებულ მეტაკლასში ერთხელ უკვე გამოყენებულია.

სტერეოტიპი არ შეიძლება გამოყენებულ იქნეს დამოუკიდებლად. იგი ყოველთვის უნდა იყოს ერთ მეტაკლასთან, რომელსაც იგი გააფართოვებს, ხოლო სხვა სტერეოტიპის სტერეოტიპით გაფართოვება შეუძლებელია. სტერეოტიპის გამოსახატავად გამოიყენება ისეთივე აღნიშვნა, როგორიც არის კლასი, გასაღებური სიტყვის "stereotype", სტერეოტიპის სახელის წინ ან ზემოთ, ჩვენებით.

გაფართოებული მოდელის ელემენტის გრაფიკული გამოსახვის შეცვლა შესაძლებელია მასთან დაკავშირებული აღნიშვნით, რომელიც შეიძლება პროფილის კლასის ნახატს წარმოადგენდეს (ნახ.4.15, ნახ.4.16).

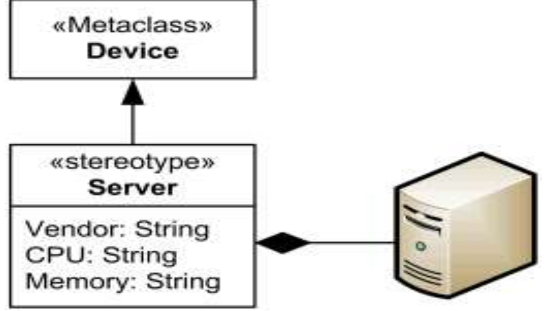


ნახ.4.10 სტერეოტიპ “servlet”-ს უკავშირდება მომხმარებელი,
თავისი აღნიშვნით



ნახ.4.11. როლი გაფართოებულია სტერეოტიპ “Web Client”-ის საშუალებით,
რომელსაც ნახაზზე უკავშირდება მომხმარებლის აღნიშვნით

ვინაიდან სტერეოტიპი არის კლასი, მას შეიძლება ქონდეს საკუთარი თვისებები. სტერეოტიპის თვისებები მოიხსენიება, როგორც მონაცემთა ტიპები, ხოლო როცა სტერეოტიპი მიმართავს მოდელის ელემენტს, თვისების მნიშვნელობები წარმოადგენენ მონაცემთა მნიშვნელობებს. ასეთ შემთხვევაში მიეთითება, ჩანაწერის ტიპები მნიშვნელობებით (ნახ.4.17).



ნახ. 4.12. მეტაკლასი გაშლილია სტერეოტიპ „Server”-ით,
რომლის თვისებებია: *vender, CPU, Memory*

სტერეოტიპის აპლიკაცია. პროფილების დიაგრამა გამოიყენება სტერეოტიპის განსაზღვრის მიზნით შემდეგ დიაგრამებში: პრეცედენტების დიაგრამა (use case diagram), კლასების დიაგრამა (class diagram), განლაგების დიაგრამა (deployment diagram) და ა.შ.

მაშინ, როდესაც სტერეოტიპი განსაზღვრულია (გამოყენებულია) მოდელის ელემენტში სტერეოტიპის ეგზემპლარი უკავშირდება მეტაკლასის ეგზემპლარს. გამოყენებული სტერეოტიპის სახელი მიეთითება მოდელის ელემენტის სახელის ზემოთ ან წინ, წყვილ ბრჭყალებში განთავსებით. მართალია სტერეოტიპის აპლიკაცია, რომლის სახელიც პატარა სიმბოლოთი იწყება ისევეა მოქმედი, როგორც დიდი სიმბოლოთი დაწყებული, მაგრამ ეს მეთოდი უკვე მოძველებულია.

«Servlet»
SearchServlet

ნახ. 4.13 სტერეოტიპი *«Servlet»* მიმართავს (უკავშირდება) მოდელის ელემენტს - *SearchServlet*

მრავალი სტერეოტიპი შეიძლება გამოყენებულ იქნას ერთი და იგივე ელემენტისთვის. გამოყენებული სტერეოტიპების სახელებს, მძიმით გამოყოფილი სიით, წყვილ ბრჭყალებში უთითებენ.

როდესაც, გაფართოვებული მოდელის ელემენტს ახლავს გასაღებური სიტყვა, მაშინ სტერეოტიპის სახელი გასაღებური სიტყვის ბოლოს, ბრჭყალებში შეიძლება მიეთითოს (მაგალითად, «device» «server»).

თუ სტერეოტიპი მოიცავს აღნიშვნის განსაზღვრებას, ეს აღნიშვნა შეიძლება გრაფიკულად იყოს დაკავშირებული მოდელის ელემენტებთან, რომელიც სტერეოტიპის საშუალებით ფართოვდება. ყოველი მოდელის ელემენტს, რომელსაც ახლავს გრაფიკული გამოსახვა, მასთან შეიძლება დაკავშირებული იყოს აღნიშვნაც. როდესაც მოდელის ელემენტი გაფართოებულია ერთი, ცალკეული სტერეოტიპით, აღნიშვნა შეიძლება წარმოდგენილი იყოს მცირე ზომის მართკუთხედით, რომელიც მოდელის ელემენტს, შიგნით, ზედა მხარეს განთავსებით გამოსახავს (ნახ.4.19).

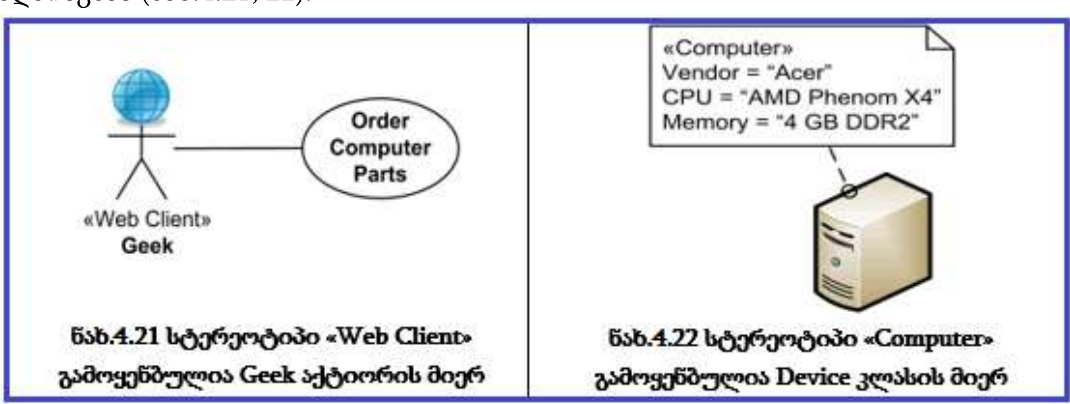
Ю
SearchServlet

ნახ. 4.14 სტერეოტიპი "Servlet" გამოყენებულია კლასის "SearchServlet"-ის მიერ

სტერეოტიპის გამოყენებისას, კლასიფიკატორის მთელი მართკუთხედი შეიძლება ჩანაცვლდეს სტერეოტიპის მოზრდილი გრაფიკული აღნიშვნით (ნახ.4.20).

ნახ.4. 15 სტერეოტიპი “Servlet” გამოყენებულია კლასის “SearchServlet”-ის მიერ

ზოგიერთი მოდელის ელემენტი უკვე დუმილის რეჟიმშიც იყენებს გრაფიკულ აღნიშვნებს საკუთარი თავის წარმოსადგენად. Aმისი ტიპიური მაგალითი არის აქტიორის მოდელის ელემენტი, რომელიც იყენებს „კაცუნას“ აღნიშვნას. იმ შემთხვევაში, როდესაც მოდელის ელემენტი გაფართოვებულია აღნიშვნიანი სტერეოტიპით, სტერეოტიპის გრაფიკული აღნიშვნა დიაგრამის აგებისას ანაცვლებს დუმილის მდგომარეობაში წარმოდგენილ სტერეოტიპის აღნიშვნას (ნახ.4.21, 22).

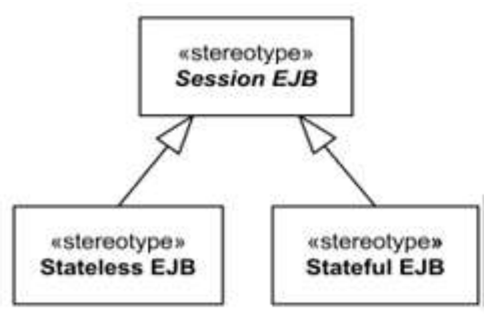


ნახ.4.21 სტერეოტიპი «Web Client» გამოყენებულია Geek აქტიორის მიერ

ნახ.4.22 სტერეოტიპი «Computer» გამოყენებულია Device კლასის მიერ

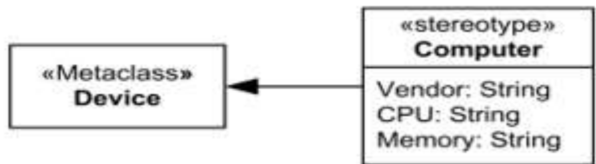
სტერეოტიპის ურთიერთქმედებები (Stereotype Relationship). სტერეოტიპი ყოველთვის უნდა იყოს იმ მეტაკლასთან კავშირში, რომელსაც იგი აფართოებს. მეტაკლასი შეიძლება გაფართოებულ იქნას ერთი ან ერთზე მეტი სტერეოტიპით, ასევე თითოეული სტერეოტიპი შეიძლება გამოყენბულ იქნას ერთი ან ერტზე მეტი მეტაკლასის გასაფართოებლად. სტერეოტიპი ორმაგ ასოციაციაშიც შეიძლება მონაწილეობდეს (binary association) და მოპირდაპირე კლასი, სხვა სტერეოტიპი, არა-სტერეოტიპი (non-sterotype) კლასი შეიძლება იყოს, რომელსაც ფლობს პროფილი ან მეტაკლასი.

სტერეოტიპს უნდა ჰქონდეს საკუთარი თვისება ასოციაციის ბოლოს, რათა მოპირდაპირე კლასზე გადასვლის შესაძლებლობა ქონდეს. თუკი მოპირდაპირე ბოლო არ იქნება სტერეოტიპი, მაშინ თვითონ ასოციაციას უნდა გააჩნდეს მოპირდაპირე თვისება (ნახ.4.23).



ნახ.4.23 აბსტრაქტული სტერეოტიპი "Session EJB" სპეციალიზებულია სტერეოტიპებით "Stateless EJB" და "Steitful EJB"

მონაცემთა ტიპების განსაზღვრა. სტერეოტიპის თვისებები განიხილება, როგორც მონაცემთა ტიპები (Tag Definitions) ან მეტათვისებები (Metaproperties)) (ნახ.4.24).



ნახ.4.24 სტერეოტიპი კომპიუტერი Vendor, CPU, Memory –ის მონაცემთა ტიპებით

სტერეოტიპი გამოიყენება პრეცედენტების, კლასების, განლაგების (deployment) და სხვა დიაგრამებთან. როდესაც სტერეოტიპი გამოიყენება მოდელის ელემენტთან, მისი თვისების მნიშვნელობა განიხილება, როგორც **ტეგის მნიშვნელობა.**

UML1 ტეგს განსაზღვრავს ტეგის მნიშვნელობად, როგორც UML-ის ერთ-ერთ გაფართოების მექანიზმს, რომელსაც მოდელთან დასაკავშირებლად ინფორმაციის დამოუკიდებლად გამოსახვის საშუალება აქვს (რომელიცUML-ის საშუალებით შეიძლება ვერ გამოსახულიყო). ტეგის მნიშვნელობა არის

გასაღებური მნიშვნელობის წყვილი, რომელიც შეიძლება დაკავშირებულ იქნას ნებისმიერი სახის მოდელის ელემენტთან.

საკვანძო სიტყვას ქვია **ტეგი.** თითოეული ტეგი თვისების განსაკუთრებულ სახეა, რომლის გამოყენებაც ერთ ან რამდენიმე სახის მოდელის ელემენტთან შეიძლება. ორივე, ტეგიც და ტეგის მნიშვნელობაც, UML ხელსაწყოს მიერ კოდირებულია, როგორც სტრიქონები (strings), რომელსაც მნიშვნელობებისთვის სხვა მონაცემთა ტიპების გამოყენების ნებას რთავს.

ტეგის მნიშვნელობა, (ისევე როგორც მეტამოდელის ატრიბუტი) გამოისახება მძიმით გამოყოფილი თვისებების მიმდევრობით, რომელიც წყვილ ფიგურულ ფრჩხილებშია მოცემული. მაგ: “{ }” (ნახ.4.25).

«Computer»
{Vendor = “Acer”,
CPU = “AMD Phenom X4”,
Memory = “4 GB DDR2”}
Aspire X1300

ნახ. 4.16 სტერეოტიპი Computer იყენებს “ტრადიციულ” ტეგმნიშვნელობების ჩანაწერებს

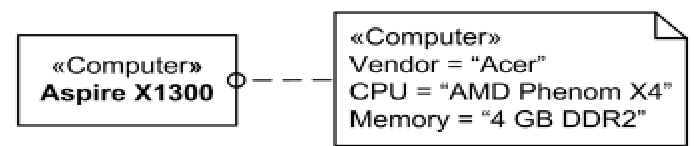
UML/4.3-ში ტეგის მნიშვნელობებს შეეძლოთ მოდელის ელემენტის გაფართოება სტერეოტიპის არარსებობის შემთხვევაშიც კი. UML/4.4-ში ეს შესაძლებლობები, მართალია ისევ არსებობს, მაგრამ უკვე მოძველებულია და შეზღუდულია. გამოიყენება მოკრძალებულად მხოლოდ თავსებადობის მიზნებისთვის.

მას შემდეგ რაც UML/2.0 ვერსია გამოვიდა ტეგის მნიშვნელობა შეიძლება წარმოდგენილი იქნას, როგორც მხოლოდ სტერეოტიპზე განსაზღვრული ატრიბუტი. მაშასადამე, მოდელის ელმენტი გაფართოებული უნდა იყოს სტერეოტიპის მიერ, რათა ტეგის მნიშვნელობების მიერ იქნას გაფართეობული. იმისათვის, რომ UML/4.3 თავსებადი იყოს ახალ ვერსიებთან UML-ის გარკვეულ ხელსაწყოებს (tools) შეუძლიათ ავტომატურად განსაზღვრონ სტერეოტიპი, რომელსაც „დაუკავშირებელი“ („მარტომყოფი“) **ატრიბუტები (ტეგის მნიშვნელობები)** დაუკავშირდებიან (მიემაგრებიან). ტეგის მნიშვნელობები შეიძლება ნაჩვენები იყოს კლასის დანაყოფში სტერეოტიპის სახელის ქვეშ. დამატებითი დანაყოფი კი მოითხოვება ყოველი იმ გამოყენებული სტერეოტიპისთვის, რომლის მნიშვნელობებიც არის ნაჩვენები. ყოველი ასეთი

კომპონენტი დასათაურებულია ბრჭყალებში მოთავსებული სტერეოტიპის სახელით (ნახ.4.26).

ნახ.4.17 სტერეოტიპი «computer» გამოყენებულია დანაყოფში ტეგის მნიშვნელობებთან

ტეგის მნიშვნელობები შეიძლება ნაჩვენები იყოს მიბმული კომენტარით, სტერეოტიპის სახელის ქვეშ (ნახ.4.27).

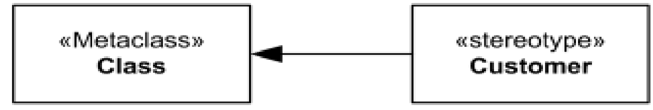


ნახ.4.18 სტერეოტიპი «computer» გამოყენებულია კომენტარის ჩანაწერში ტეგის მნიშვნელობებთან

როდესაც მნიშვნელობის სახელი, კომენტარში ან კომენტარის სიმბოლოთია ნაჩვენები, თითოეული სახელ-მნიშველობის (name-value) წყვილი ახალ განცალკევებულ სტრიქონზე უნდა გამოჩნდეს.

რაც შეეხება **გაფართოვებას,** იგი არის ასოციაციური კავშირი, რომელიც გამოიყენება მეტაკლასის იმ თვისებების საჩვენებლად, რომლებიც სტერეოტიპის საშუალებით არიან გაფართოვებული და იძლევიან სტერეოტიპების კლასზე მოქნილად დამატების და მოგვიანებით მოხსნის საშუალებას, თუ ეს საჭირო იქნება.

გაფართოება აღინიშნება სავსე სამკუთხედიანი ისრით, რომელიც სტერეოტიპიდან მიმართულია გაფართოებული მეტაკლასისაკენ (ნახ.4.28).

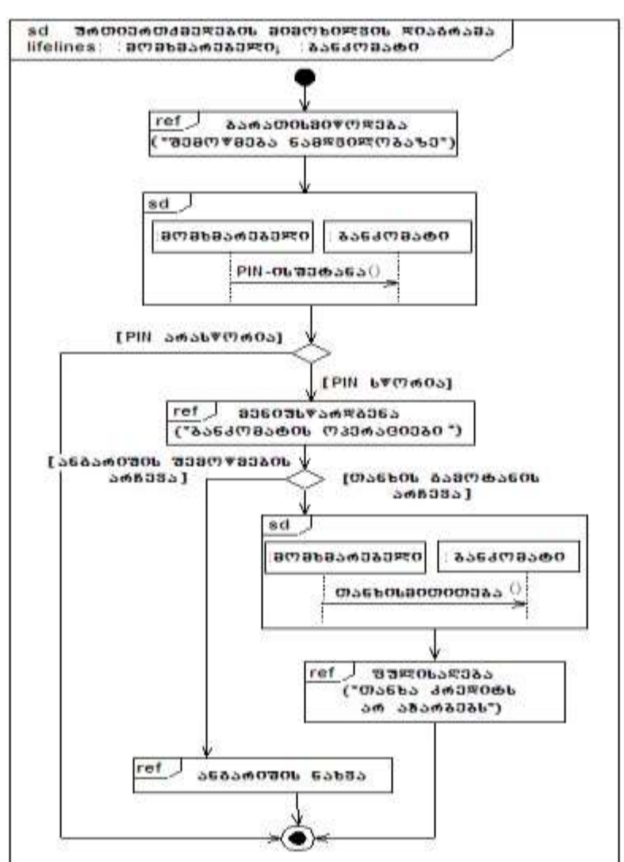


ნახ. 4.19 მეტაკლასი Class გაფართოებულია სტერეოტიპით Customer

რადგან, გაფართოება არის ასოციაციის სუბკლასი, მას შეიძლება ქონდეს ჩვეულებრივი აღნიშვნები (წარწერები), მაგრამ ამ შემთხვევაში არ უნდა იყოს ნაჩვენები მართვადი ისარი. გაფართოების აღნიშვნები სიმბოლურად არის უგულებელყოფილი.

ურთიერთქმედების მიმოხილვის დიაგრამა (Interaction Overview Diagram). დიაგრამის ეს ტიპი ფაქტიურად წარმოადგენს მოღვაწეობის და მიმდევრობის დიაგრამის შერწყმას. მასში მოქმედებების ნაცვლად დიაგრამის კვანძებში წარმოდგენილია მიმდევრობის დიაგრამები (სცენარები). ასეთი მეთოდით მიღწეულია მიზანი, აღიწეროს რთული ქცევა განშტოებების საშუალებით, რისი შესრულებაც მიმდევრობის დიაგრამაზე წინათ შეუძელბელი იყო. იგი განკუთვნილია ურთიერთქმედების წარმოსადგენად მხოლოდ მართვის ნაკადების კონტექსტში.

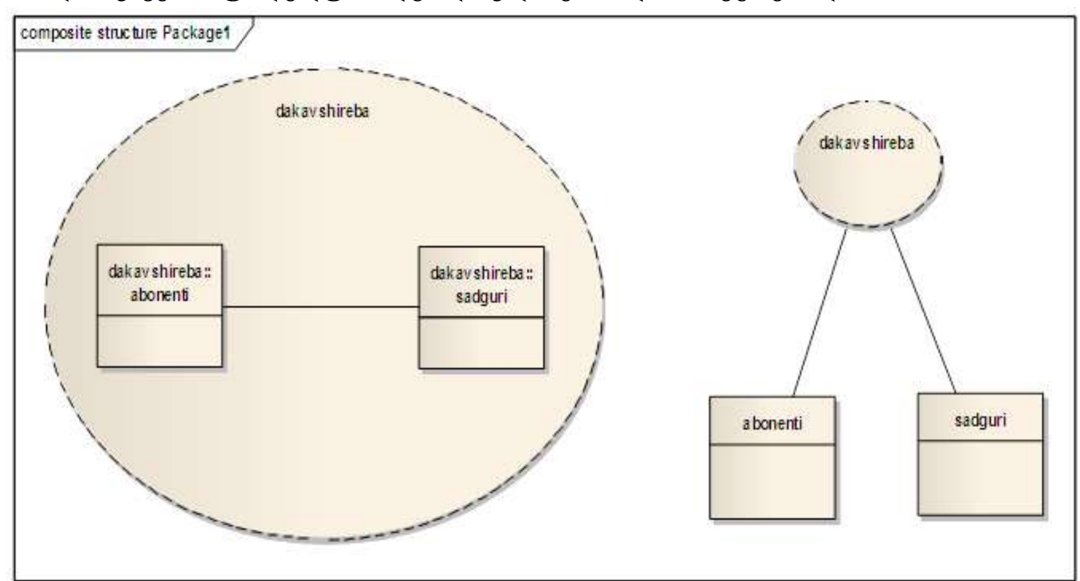
ურთიერთქმდების მიმოხლილვის დიაგრამებს მოქმედების კვანძების და მოღვაწეობის დიაგრამის ობიექტების ნაცვლად, გააჩნია ფრეიმები, რომლებიც ურთიერთქმედების და მისი გამოყენების შესაბამისია. აქ ალტერნატიული კომბინირებული ფრაგმენტები გვევლინება გადაწყვეტილების კვანძებად და კვანძების გაერთიანებების შესაბამისია, ხოლო პარალელური კომბინირებული ფრაგმენტები წარმოადგენენ დაყოფის კვანძებს და გაერთიანების კვანძებს შეესაბამებიან; ციკლის ტიპის კომბინირებული ფრაგმენტები კი წარმოგვიდგებიან მარტივ ციკლებად. ასეთ დიაგრამებზე განშტოებები და განშტოებების გაერთიანებები აუცილებლად აზრობრივად უნდა იყოს დალაგებული. ჩართული ფრეიმი აღინიშნება **sd** - ტეგით, ისე როგორც სხვა სახის ურთიერთქმედების დიაგრამა (ნახ.4.29).



ნახ. 4.20 ურთიერთქმედების მიმოხილვის დიაგრამა: „ბანკომატით მომსახურება"

კომპოზიციური სტრუქტურის დიაგრამით (Composite Structure Diagram) აღიწერება არა ქცევა, არამედ ურთიერთმოქმედი მხარეები და მისი კავშირები. კოოპერაცია გამოისახება სპეციალური ტიპის – კომპოზიციური სტრუქტურის დიაგრამაზე(ნახ.4.30-4.37). ეს დიაგრამები შეიძლება ასევე გამოყენებულ იქნეს კომპოზიციური კომპონენტების მოდელირებისთვის.

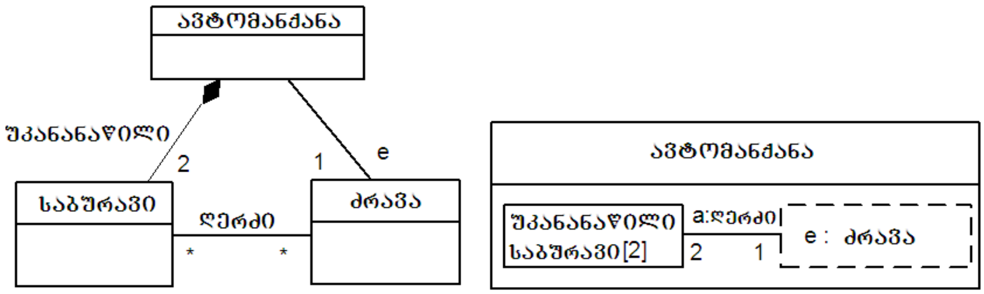
საურთიერთობო მხარეები არიან როლები, რომლებიც კლასის ფუნქციონირების რომელიღაც ნაწილს აღწერენ, რომელიც მთავარ კონტექსტად არის გამოყენებული (ამ შემთხვევაში ასეთი კონტექსტი არის კოოპერაცია). მაგ: ორი კლასისთვის – აბონენტი და სადგური – კოოპერაცია „კავშირი“ განსაზღვრავს კონტექსტს, რაც არის აბონენტსა და სადგურს შორის კავშირის დამყარების პროცედურა, ხოლო ამ კლასის როლები ითავსებენ კლასის იმ ფუნქციას, რომელიც ამ პროცედურის რეალიზებას ახდენს. ვინაიდან ამ ფუნქციის გარდა, მოცემულ კლასს, შეიძლება სხვა მრავალი ფუნქციაც გააჩნდეს, როლი იკავებს შუალედურ ადგილს კლასებსა და მის ეგზემპლარს შორის.



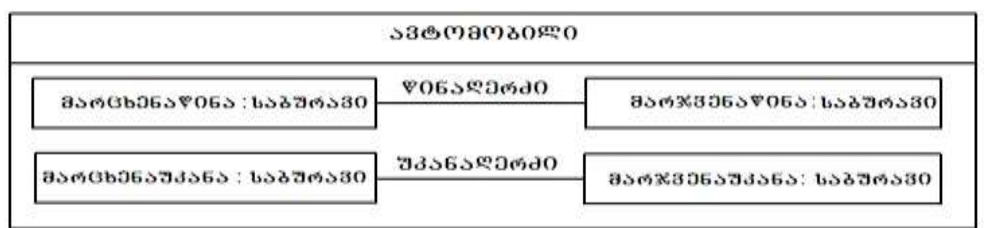
ნახ.4.21 კომპოზიციური სტრუქტურის დიაგრამა
(აბონენტი-სადგური)

იშვიათად ხდება, ისე რომ ერთი სისტემის პროექტირებისას ან აღწერისას ყველა ტიპის დიაგრამა იქნას გამოყენებული. მაგალითად, კლასების დიაგრამა მოსახერხებელია ტიპური ობიექტ-ორიენტირებული პროგრამირებისათვის, ხოლო ამისათვის იშვიათად გამოიყენება მდგომარეობის და გადასვლების

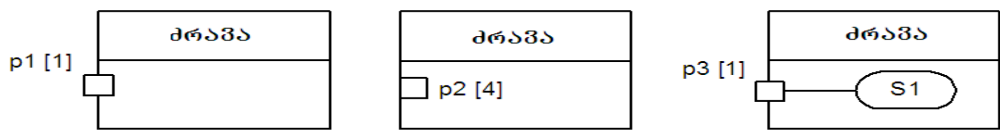
დიაგრამა. სამაგიეროდ კომპოზიციური დიაგრამები აქტიურად გამოიყენება ტელეკომუნიკაციის სისტემების პროგრამული უზრუნველყოფის შემუშავე-ბისას, სადაც კლასების დიაგრამა შეიძლება საერთოდ არ იქნას გამოყენებული და ა.შ. [37].



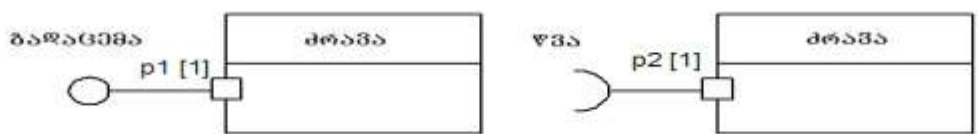
ნახ. 4.22 კომპოზიციური სტრუქტურის დიაგრამა „ავტომანქანა“ 1



ნახ. 4.23 კომპოზიციური სტრუქტურის დიაგრამა „ავტომანქანა“ 2



ნახ. 4.33 პორტები. კომპოზიციური სტრუქტურის დიაგრამა „ძრავა“

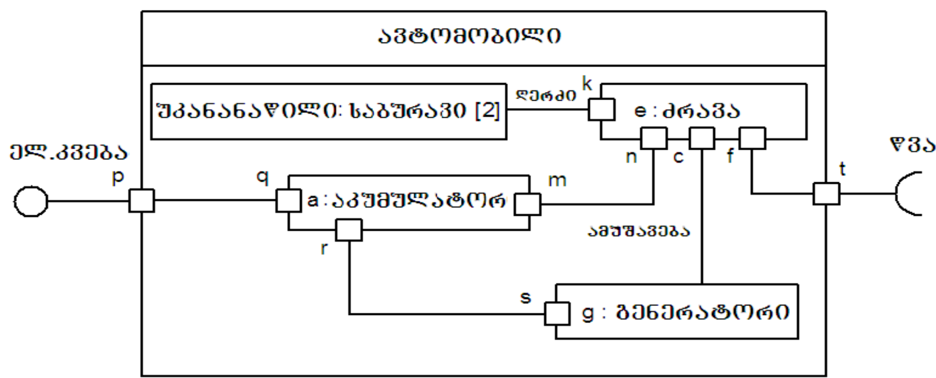


ნახ. 4.24 პორტები, უზრუნველმყოფი და მოთხოვნადი ინტერფეისებით. კომპოზიციური სტრუქტურის დიაგრამა - „ძრავა“

კომპოზიციური სტრუქტურის დიაგრამით შეიძლება ნაჩვენები იყოს:

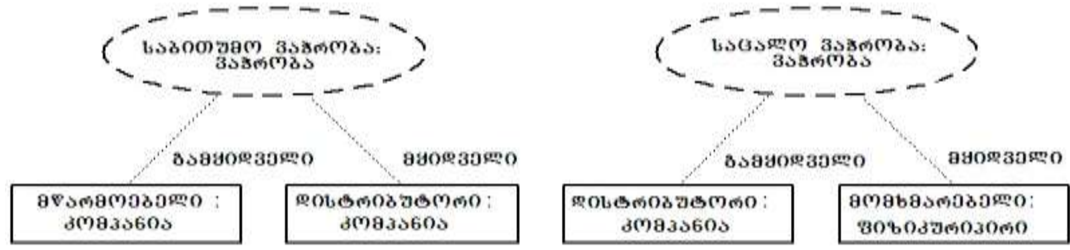
- კლასიფიკატორის შიგა სტრუქტურა - internal structure diagram (ნახ.4.35);

- კლასიფიკატორის გარემოსთან ურთიერთქმედება **პორტების** საშუალებით (ნახ.4.35);

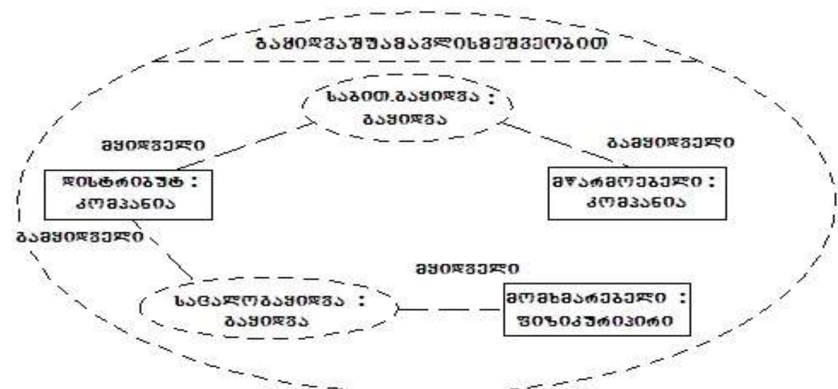


ნახ.4.25 კომპოზიციური სტრუქტურის დიაგრამა: „ავტომანქანა“
შიგა სტრუქტურით და მისი გარემოსთან ურთიერთქმედება

- კოოპერაციის ქცევა - collaboration use diagram. (ნახ.4.36, ნახ.4.37) [37].



ნახ. 4.26 კოოპერაციის ქცევა – „საბითუმო და საცალო გაყიდვა 1“



ნახ.4.27 კოოპერაციის ქცევა - „საბითუმო და საცალო გაყიდვა 2“

ტერმინი „სტრუქტურა“ ისეთი ტიპის დიაგრამებისთვის განისაზღვრა UML-ში, რასაც ურთიერთმოქმედი ელემენტების კომპოზიცია წარმოადგენს და რომელიც განკუთვნილია საერთო მიზნის მისაღწევად, ანუ კომუნიკაციური კავშირების საშუალებით დროში შესრულებადი ეგზემპლარების თანამშრომლობის უზრუნველყოფა (განხორციელება).

კლასიფიკატორი. შიგა სტრუქტურაში და კოოპერაციაში კლასიფიკატორი გაფართოებულია იმ შესაძლებლობით, რასაც კოოპერაციების გამოყენების საშუალება წარმოადგენს. **კოოპერაციის გამოყენებები** კოოპერაციას აკავშირებს კლასიფიკატორთან, რათა აღიწეროს კლასიფიკატორის ქცევა.

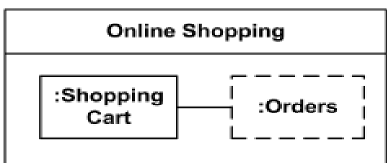
კლასიფიკატორი, რომელიც კოოპარაციას იყენებს, შეიძლება წარმოდგენილ იქნას როგორც კლასიფიკატორის ერთი მთლიანი ქცევა. კლასიფიკატორთან დაკავშირებული კოოპერაცია კოოპერაციის გამოყენებების საშუალებით გვიჩვენებს თუ კლასიფკატორის სტრუქტურული მახასიათებლების შესაბამისი ეგზემპლარები როგორ ურთიერთქმედებს კლასიფიკატორის საერთო ქცევის შესაქმნელად.

წარმოდგენილი კოლაბორაცია შეიძლება გამოყენბულ იქნეს კლასიფიკატორის ქცევის აღწერის უზრუნველსაყოფად აბსტრაქციის სხვადასხვა დონეზე მანამადე, სანამ ის მოთხოვნადი იქნება კლასიფიკატორის შიგა სტრუქტურის მიერ.

სტრუქტურული კლასიფიკატორები. UML უზრუნველყოფს ისეთ მექანიზმებს, რომლებიც აღწერს კლასიფიკატორში შექმნილ ეგზემპლარში ერთმანეთთან დაკავშირებულ ელემენტთა სტრუქტურას. ამ სტრუქტურას ეწოდება (კლასიფიკატორის) შიგა სტრუქტურა და მისაღებია, როგორც სტრუქტურული კლასებისთვის, ისე კოოპერაციისთვის. შიგა სტრუქტურა შედგება **თვისებებისგან, ნაწილების** ჩათვლით, რომლებიც თამაშობს სპეციფიურ როლებს და **კავშირებისგან.**

UML სტრუქტურული კლასიფიკატორი არის კლასიფიკატორი, რომელსაც აქვს შიგა სტრუქტურა და რომლის ქცევაც მთლიანად ან ნაწილობრივ აღწერილია კოოპერაციის საკუთარი ან მიწოდებული ეგზემპლარებით.

კლასიფიკატორები გამოიყენება ელემენტების ან როლების გარკვეული სტრუქტურის აღწერისათვის, რომლებიც სპეციალურ ფუნქციებს ასრულებს და აწარმოებს სასურველ ფუნქციებს.



ნახ. 4.28 სტრუქტურული კლასიფიკატორი Online Shopping
მისი შიგა სტრუქტურით

სიმრავლეები სტრუქტურის მახასიათებლებზე და კავშირის ბოლოებზე მიუთითებს ეგზემპლარების (ობიექტების და კავშირების) რაოდენობაზე, რომლებიც შეიძლება შეიქმნას კლასიფიკატორის შემცველ ეგზემპლარებში, ან როცა კლასიფიკატორის შემცველი ეგზემპლარი შექმნილია ან კავშირების შემთხვევაში, როცა ობიექტი დამატებულია, როგორც როლის მნიშვნელობა.

სიმრავლის დიაპაზონის ქვედა ზღვარი მიუთიეთებს იმ ეგზემპლარის რაოდენობაზე, რომელიც უკვე შექმნილია. სიმრავლის დიაპაზონის ზედა ზღვარი მიუთიეთებს იმ ეგზემპლარის მაქსიმალურ რაოდენობაზე, რომლებიც შეიძლება მომავალში შეიქმნას. სტრუქტურული მახასიათებლების შესაბამისი სლოტები ინიცირდება ამ ეგზემპლარებით.

ამდენად, კომპოზიციური სტრუქტურის დიაგრამა არის დიაგრამა, რომელიც გამოიყენება ისეთი კლასიფიკატორების შიგა სტრუქტურის გამოსახატავად, როგორიცაა კლასი, კომპონენტი თუ კოოპერაცია, სისტემის სხვა ნაწილებთან კლასიფიკატორის ურთიერთქმედების წერტილის ჩათვლით.

შიგა სტრუქტურა (internal structure) არის მოდელის ურთიერთმოქმედი ელემენტების სტრუქტურა, რომელიც იქმნება მათი კლასიფიკატორის შემცველ ეგზემპლარში.

თვისება (property) ეგზემპლარების სიმრავლეა, რომელიც წარმოადგენს მისი ეგზემპლარის შემცველი კლასიფიკატორის საკუთრებას..

ნაწილი - თვისებაა, რომელიც კომპოზიციური კლასიფიკატორის შიგა სტრუქტურის ელემენტია, ამ შემთხვევაში – კლასის ელემენტი.

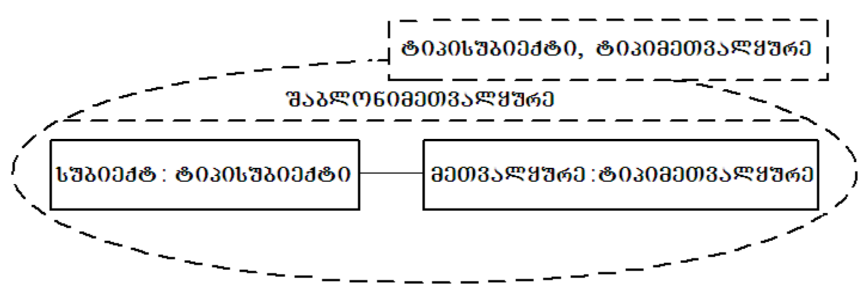
პორტი არის კლასიფიკატორის თვისება, რომელიც განსაზღვრავს (აყალიბებს) ამ კლასიფიკატორებს შორის და მათ გარემოსთან ურთიერთ-ქმდების ან კლასიფიკატორებს შორის და მათი შიგა ნაწილებთან ურთიერთქმედების ცალკეულ მხარეებს. პორტის *უზრუნველმყოფი (პროვაიდერი) ინტერფეისი* (provided interface) აყალიბებს მოთხოვნებს, რომლებიც შეიძლება გარემოდან გადაცემულ იქნეს კლასზე ამ პორტის გავლით,

ხოლო პორტის *მოთხოვნადი ინტერფეისი* (required interface) აყალიბებს მოთხოვნებს, რომლებიც შეიძლება კლასიდან გადაცემულ იქნეს გარემოში ამ პორტის გავლით.

კოოპერაციის როლი (collaboration role) განსაზღვრავს თვისებების საჭირო სიმრავლეს, რომელიც კოოპერაციის შესაბამის მონაწილეებს უნდა გააჩნდეს.

კოოპერაციის შაბლონი არის პარამეტრირებული შაბლონი, რომელიც კოოპერაციის მთლიანი ოჯახის შესაბამისია. შაბლონის პარამეტრები მისი ნაწილების ტიპები ან როლებია, ხოლო ამ ოჯახის ცალკეული კოოპერაცია შეიძლება მიღებულ იქნას კოოპერაციის შაბლონის პარამეტრების დაკავშირებით კონკრეტულ კლასებთან.

კოოპერაციის შაბლონები პოულობს პრაქტიკულ გამოყენებას პროექტი-რების სტანდარტული მოდელების განხილვისას (ნახ.4.39).



ნახ. 4.29. შაბლონი

დროითი დიაგრამა (Timing Diagram) გამოიყენება მდგომარეობის ან ერთი ან მეტი ელემენტის მნიშვნელობის ცვლილების საჩვენებლად დროის განმავლობაში. იგი ასევე შეიძლება გვიჩვენებდეს დროით შემთხვევებსა და დროს შორის ურთიერთქმედებას და ხანგრძლოივობის შეზღუდვებს, რომლებიც მათ მართავს.

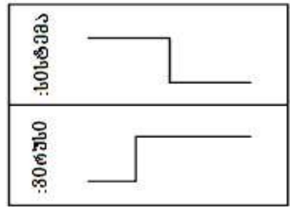
UML დროითი დიაგრამა ურთიერთქმედების დიაგრამაა, რომელიც გვიჩვენებს ურთიერთქმედებებს დროის მიხედვით და გამოიყენება ცალკეული სიცოცხლის ხაზის მდგომარეობის ცვლილებების ან შეტყობინებების სინქრონიზაციის დროში გამოსახატავად. დროითი დიაგრამა ყურადღებას ამახვილებს ცვლილებებზე სიცოცხლის ხაზში და სიცოცხლის ხაზებს შორის ურთიერთქმედებაზე დროითი ღერძის გასწვრივ. დროითი დიაგრამა აღწერს

ქცევას, როგორც ინდივიდუალური კლასიფიკატორების, ისე კლასიფიკატორის ურთიერთქმედებებს შორის, რომელიც სიცოცხლის ხაზის მოდელირებით, აქცენტირებას ახდენს ცვლილებების გამომწვევი მოვლენის მოხდენის დროზე.

UML დროითი დიაგრამა იყენებს შემდეგ აღნიშვნებს (და ჩანაწერებს): lifeline - სიცოცხლის ხაზი; state or condition timeline – მდგომარეობის ან პირობის დროითი ხაზი; destruction event – შეწყვეტის მოვლენა; duration constraint – პერიოდის ხანგრძლივობა; time constraint – დროის ხანგრძლივობა (შეზღუდვა).

სიცოცხლის ხაზი არის სახელობითი ელემენტი, რომელიც გვიჩვენებს ინდივიდუალური მონაწილის ქმედებას, მიუხედავად იმისა, რომ ნაწილები და სტრუქტურული მახასიათებლები შეიძლება იყოს მრავალრიცხოვანი (ანუ 1-ზე მეტი), სიცოცხლის ხაზები წარმოადგენს მხოლოდ ერთი ურთიერთქმედების გაერთიანებას.

სიცოცხლის ხაზი დროით დიაგრამაზე წარმოდგენილია კლასიფიკატორის სახელით, სახელწოდების საშუალებით (ან ეგზემპლარით, რომელსაც იგი წარმოგვიდგენს). ის შეიძლება განთავსდეს, როგორც დიაგრამის ფრეიმის შიგნით, ისე “swimline”-ში (ნახ.4.40).

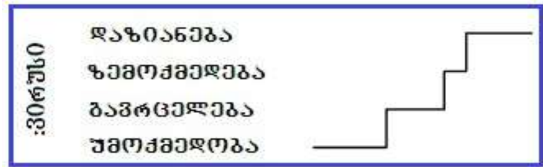


ნახ.4.30 სიცოცხლის ხაზები ეგზემპლარებისთვის: „სისტემა“ და „ვირუსი“

მდგომარეობის სიცოცხლის ხაზი გვიჩვენებს რაიმეს მდგომარეობის ცვლილებას დროში. X-ღერძი გვიჩვენებს გასავლელ დროს, რომელზეც დროის ნებისმიერი ერთეული აირჩევა, ხოლო Y-ღერძი იძლევა მდგომარეობის სიას, ჩამონათვალს (ნახ.41, 42).

დროითმა დიაგრამამ შეიძლება გვიჩვენოს მონაწილე კლასიფიკატორის ან ატრიბუტის მდგომარეობები ან რაიმე შესამოწმებელი (სატესტო) პირობის, როგორიცაა ატრიბუტის დისკრეტული ან რაოდენობრივი მნიშვნელობები.

UML იძლევა მდგომარეობის (გარემოების) უსრალო გაგრძელების ჩვენების საშუალებას. ეს შეიძლება იყოს ისეთი მოვლენები, როგორიცაა სიხშირე, ინტენსიურობა, ტემპერატურა.

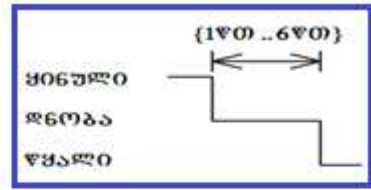


ნახ.4.31 დროითი ხაზი. "ვირუსის" მდგომარეობების ცვლილება დროში

პერიოდის ხანგრძლივობა გვიჩვენებს დროის ინტერვალს, ესა თუ ის მდგომარეობა თუ რამდენ ხანს გრძელდება, ანუ ეს არის დროის შეზღუდვის ჩვენება და იგი აუცილებლად უნდა იყოს დაკმაყოფილებული.

შინაარსობრივად ეს იგივეა, რაც დროის რაღაც პერიოდი (მონაკვეთი). თუ ხანგრძლივობა არის დარღვეული, თანმიმდევრობაც ირღვევა, რაც ნიშნავს იმას, რომ სისტემა მოისაზრება ჩავარდნილად ან განიცდის კრახს.

პერიოდის ხანგრძლივობა ნაჩვენებია გარკვეული გრაფიკული კავშირის სახით ინტერვალს და იმ კონსტრუქციას შორის, რომელსაც ის განსაზღვრავს (ნახ.4.42).



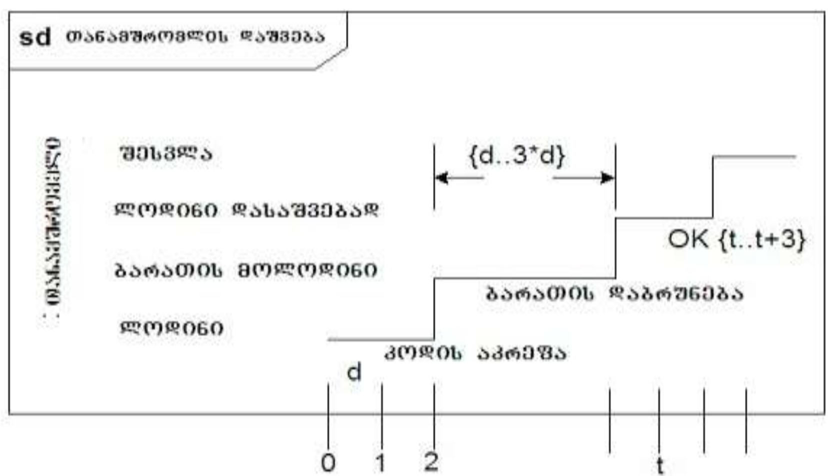
ნახ.4.32 ყინულის დნობის ხანგრძლივობა

დროის შეზღუდვა (ხანგრძლივობა) არის დროის ინტერვალის შეზღუდვა, რომელიც გვიჩვენებს იმ დროის, რა დროსაც მოქმედება უნდა შესრულდეს. დროის შეზღუდვა ნაჩვენებია გრაფიკული კავშირით დროის ინტერვალსა და კონსტრუქციას შორის, რომელსაც ის ზღუდავს (ნახ.4.43).

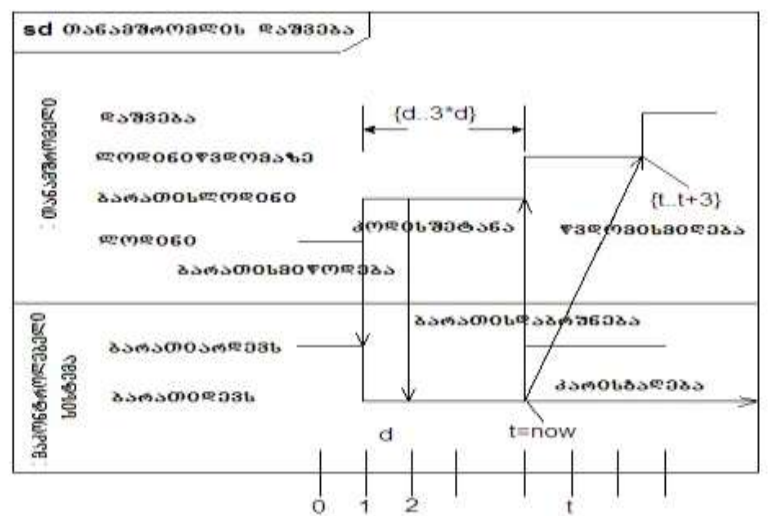
ნახ.4.33 რეისი უნდა დასრულდეს 14:30-14:45 სთ-ზე

შეწყვეტის მოვლენა – არის მესიჯის წარმოქმნა, რომელიც გვიჩვენებს სიცოცხლის ხაზით აღწერილი მაგალითის დასრულებას.

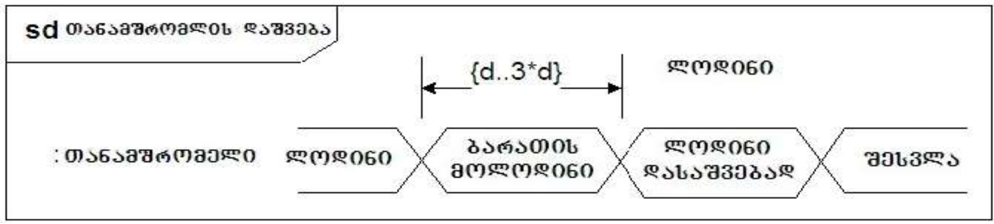
სხვა მოვლენებს ამის შემდეგ ადგილი აღარ ექნება მოცემულ სიცოცხლის ხაზში. UML-ის წინა ვერსიებში ამ მოვლენას შეჩერება (Stop) ეწოდებოდა.



ნახ. 4.34. შენობაში თანამშრომლის დაშვება. სიცოცხლის ხაზი „თანამშრომელი“



ნახ.4.35. შენობაში თანამშრომლის დაშვება. სიცოცხლის ხაზები: „სისტემა“, „თანამშრომელი“



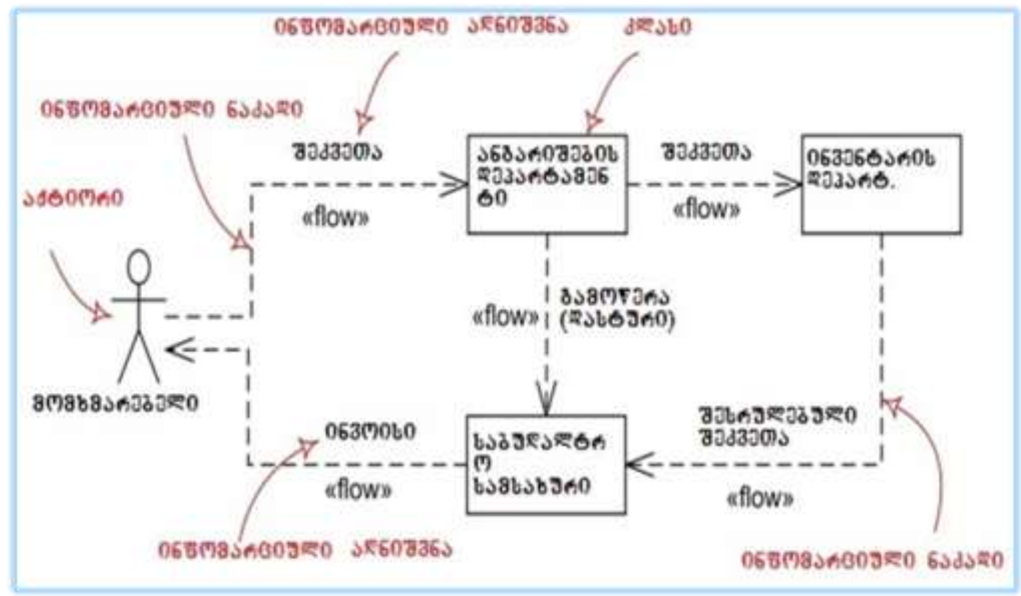
ნახ.4.36 დროითი დიაგრამის კომპაქტური ფორმა.
შენობაში თანამშრომლის დაშვება. სიცოცხლის ხაზი-„თანამშრომელი“

ინფორმაციულ ნაკადთა დიაგრამა (information flow diagrams) არის UML-ის ქცევითი დიაგრამა, რომელიც გვიჩვენებს ინფომაციის გაცვლას სისტემის ნაწილებს შორის აბსტრაქციის მაღალ დონეებზე. ინფომაციული ნაკადები შეიძლება უფრო მოხერხებულად გამოვიყენოთ სისტემაში ინფორმაციის ცირკულაციის ასაღწერად, მოდელის არა მთელის სპეციფიკაციის, არამედ გარკვეული ასპექტების ან ნაკლები რაოდენობის დეტალების წარმოდგენით.

ინფორმაციული ნაკადები არ ახდენს ინფორმაციის ბუნების, მისი გადაცემის მექანიზმის, გაცვლის თანმიმდევრობის, ან სხვა რაიმე მაკონტროლებელ პირობების სპეციფიცირებას.

ინფორმაციული ნიშნები შეიძლება გამოყენებულ იქნას იმ ინფომაციის წარმოსადგენად, რომელიც მიედინება ინფომაციული ნაკადებით სისტემის გავლით, მანამდე სანამ მისი რეალიზაციის დეტალები იქნება შემუშავებული.

UML-ში ინფომაციული ნაკადების დიაგრამა გამოსახვის მწირ შესაძლებლობებს ფლობს, ინფორმაციული ნიშნები არ გვიჩვენებს გადაგზავნილი ინფომაციის დეტალებს და მათ არ გააჩნია მახასიათებლები, განზოგადება, ასოციაცია (ნახ.4.47).



ნახ.4.37 ინფორამციულ ნაკადთა დიაგრამა „შეკვეთა“

კომუნიკაციის დიაგრამა (Communication Diagram) განკუთვნილია სისტემის შიგა სტრუქტურის კონტექსტში ურთიერთქმედების და შეტყობინებების გადაცემის წარმოსადგენად, რომელსაც აქვს გრაფის სახე და რომლის მწვერვალებსაც მართკუთხედის ფორმით გამოსახული კომპოზიციური კლასის ნაწილები ან ურთიერთქმედების როლები წარმოადგენს. ეს მწვერვალები შეესაბამება სიცოცხლის ხაზებს და საკუთარ სტრუქტურულ კონტექსტში გამოისახება. გრაფის წიბოებს კი კავშირები წარმოადგენს, რომელზეც კომუნიკაციის მარშრუტები გადის. სიცოცხლის ხაზებს შორის ხდება შეტყობინებების გაცვლა, რომლებიც პატარა ისრების სახით გამოისახება, ზედ კი სახელებია განთავსებული.

კავშირი და შეტყობინება. კავშირი არის დამოუკიდებელი ასოციაციის კავშირი, რომელიც სიცოცხლის ხაზებს შორის შეტყობინების მიმართული გადაგზავნისათვის არხს განსაზღვრავს.

შეტყობინება ისრით გამოისახება კავშირის ხაზთან ახლოს და მისი გადაგზავნა ისრის მიმართულებით ხდება. ისრის ახლოს ჩანაწერის სპეციალური ფორმით მიეთითება შეტყობინების იდენტიფიკატორი.

4.3. საინფორმაციო სისტემების პროგრამული უზრუნველყოფის აგების თანამედროვე ინსტრუმენტული საშუალებები

4.3.1. Ms Visio, Rational Rose და სხვა პაკეტები

სისტემების ავტომატიზებული დაპროექტებისა და დამუშვების (CASE), მოდელზე ორიენტირებული არქიტექტურისა (MDA) და დანართების სწრაფი დამუშავების (RAD) ტექნოლოგიების ბაზაზე დღესდღეობით შექმნილია და ვითარდება მრავალი ინსტრუმენტული საშუალება. მაგალითად, აღსანიშნავია MS Visio, Visual Paradigm, Altova Umodel, Rational Rose, Enterprise Architect, ასევე ჩაშენებულია Java, Visual Studio პაკეტებში [31].

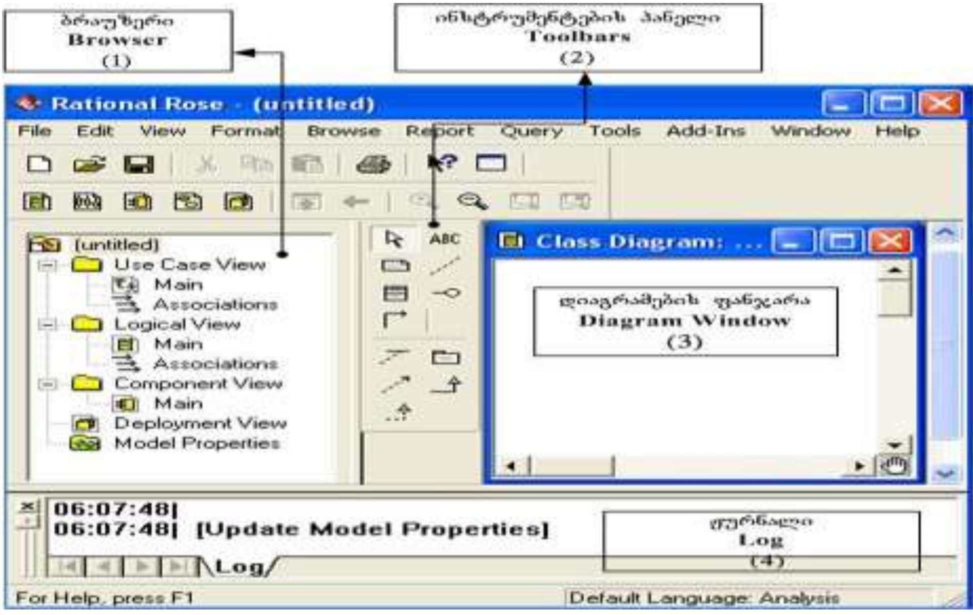
ერთ-ერთი პირველი ინსტრუმენტია Rational Rose (გ.ბუჩი [14]). იგი დაწერილია Javaა-ზე, იძლევა მოდელების თანმიმდევრულად აგების საშუალებას და საკმაოდ მოქნილიცაა [23].

შეუძლია CASE ტექნოლოგიის გამოყენება, კლასთა მოდელიდან პროგრამული კოდის გენერაცია და რევერსული დაპროექტება (პროგრამული კოდიდან კლასთა მოდელის შექმნა). პროგრამულ სისტემებთან კავშირისთვის

გამოყიენებს დაპროგრამების ენების ფართო სპექტრს (Java, Visual Basic, C++, XML და ა.შ.), ხოლო მონაცემთა მოდელების და მონაცემთა ბაზების მართვისთვის რევერსული დაპროექტების საშუალებებს (MSSQLServer, Oracle და ა.შ.) [66].

Rational Rose სისტემაში დიაგრამები დაჯგუფებულია 4 ძირითად ბლოკად: Use Case View, Logical View, Component View და Deployment View. ისინი მოთავსებულია ბრაუზერში (ნახ.4.48).

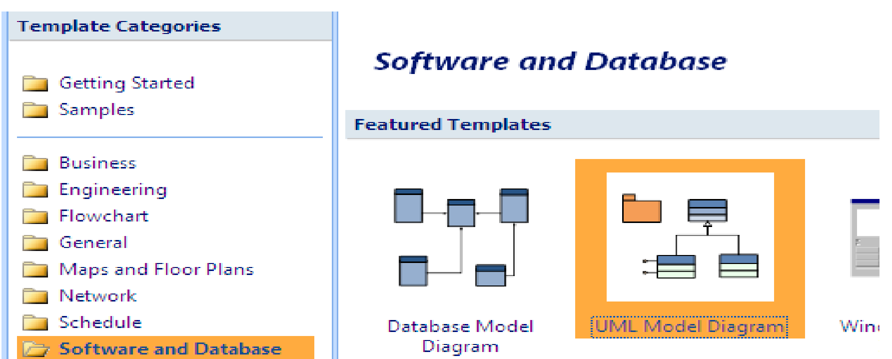
Rational Rose სისტემის ინტერფეისი შეიცავს შემდეგ ელემენტებს: Browser (1), Toolbars (2), Diagram Window (3) და Log (4) [31].



ნახ. 4.38. Rational Rose სისტემის ძირითადი ფანჯარა

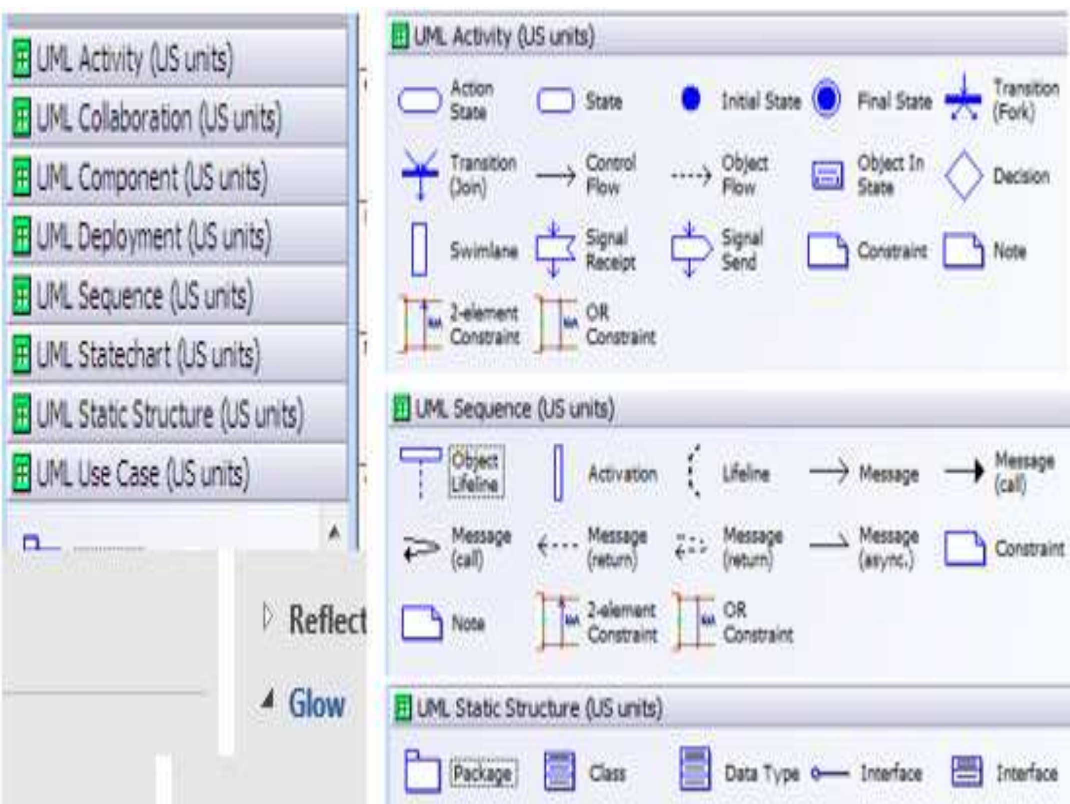
პროგრამული პაკეტი Ms-Visio მაიკროსოფთის ახალი საინფორმაციო ტექნოლოგიაა, რომელიც კომპიუტერული სისიტემებისა და ქსელური ტექნოლოგიების დაპროექტებისა და მოდელირებისათვის გამოიყენება.

გამოყენებითი პროგრამული უზრუნველყოფის (Applled Software) ობიექტ-ორიენტირებული ანალიზის, დაპროექტების და რეალიზაციის ცალკეული ეტაპების მოდელირების მიზნით განიხილება UML მეთოდოლოგიის ინსტრუმენტული საშუალება MsVisio Professional, ეს პაკეტი ძალზე პოპულარული და მრავალფუნქციურიაა. 4.49 ნახაზზე მოცემულია Ms Visio-ს საწყისი გვერდის ფრაგმენტი.



ნახ. 4.49. Ms Visio-ს საწყისი გვერდი

იმისდა მიხედვით თუ რა ტიპის დიაგრამას ვაგებთ, ვირჩევთ შესაბამისი ინსტრუმენტების პანელს, რომელიც ჩანართების სახით არის მოცემული და დიაგრამის ასაგებ ინსტრუმენტებს შეიცავს (ნახ.4.50).



ნახ.4.50. UML-დიაგრამების ინსტრუმენტების პანელების ფრაგმენტი

4.3.2. Enterprise Architect და Visual Studio.NET

Sparx Systems Enterprise Architect არის UML ინსტრუმენტების ერთობლიობის მძლავრი, მრავალმხრივი და მრავალპლატფორმიანი სისტემა. იგი გამოიყენება პროგრამული უზრუნველყოფის დაპროექტებისა და აგების პროცესების ავტომატიზაციის მიზნით და შეიცავს ანალიზის, მოდელების გრაფიკული აგების, ტესტირებისა და მომსახურების ფაზებს.

მასში თავმოყრილია UML1/2, BPMN, Workflow და DocFlow ტექნოლოგიების გამოყენების საშუალებები, პირდაპირი და უკუდაპროექტების მექანიზმები, მონაცემთა ბაზების (SQL Server, MySQL, Oracle 9i და 10g, PostgreSQL, MSDE, Adaptive Sever, MS Access, Firebird) პროგრამული ენების (Java, C#, C++) - მთელი სპექტრის გამოყენებით [31].

სისტემის მომხმარებლები შეიძლება იყვნენ ბიზნეს-ანალიტიკოსები, პროექტ-მენეჯერები, ხარისხის მართვისა და პროგრამული პაკეტების ტესტირების სპეციალისტები, დამპროექტებლები და სხვა.

აღსანიშნავია, რომ სისტემაში Enterprise Architect (EA) გათვალის-წინებულია MDG (Model Driven Generation-გრაფიკული მოდელების გენერაცია) ტექნოლოგია, რის ბაზაზეც სისტემა თანამშრომლობს თანამედროვე პროგრამულ პლატფორმებთან. MDG ტექნოლოგიაში შედის, ისეთი სტანდარტები, როგორიცაა BPMN, BPEL, Data Flow Diagrams, Mind Mapping, SPEM და სხვა.

ამდენად, ამ სისტემის შესაძლებლობაშია, როგორც პირდაპირი და რევერსიული დაპროექტების პროცესების განხორციელება, ისე პროგრამულ პლატფორმებში Enterprise Architect სისტემის ინტეგრაცია.

ინტეგრაცია ხორციელდება სპეციალური პროგრამების (მაგ., .NET პლატfორმისთვის-VSBridge.exe, EMDGVSlink.exe, MDG Link for Visual Studio .NET და ა.შ) ინსტალაციით, რაც ე.წ. ხიდს წარმოადგენს პლატფორმასა და EA სისტემას შორის).

4.3.3. ბიზნესპროცესების დაპროგრამების ახალი ტექნოლოგია Workflow Foundation in Visual Studio.NET

თანამედროვე საინფორმაციო ტექნოლოგიების განვითარება შესაძლებლობას გვაძლევს ვაწარმოოთ ნებისმიერი რთული სტრუქტურული ობიექტის ავტომატიზაცია, როგორც ლოკალური ქსელის ფარგლებში, ისე გლობალური ქსელისათვის და WAP დანართისათვის.

ერთ-ერთ ასეთ მძლავრ თანამედორვე ტექნოლოგიაა Microsoft-ის .NET კონცეფცია. Microsoftt პლატფორმის სივრცეში, რომელიც ბაზირებულია სერვის-ორიენტირებულ მიდგომაზე, სრულყოფილს ხდის მონაცემებთან წვდომის ტექნოლოგიას ADO.NET დრაივერის საშუალებით და მოცავს .NET Framework სისტემას - ინსტრუმენტული საშუალებით Visual Studio.NET, კორპორაციულ .NET სერვერებს, უნიფიცირებული სტანდარტის აგების სერვისებს და ა.შ. [61].

.NET Framework განსაზღვრავს გარემოს, რომელიც უზრუნველყოფს პლატფომრაზე დამოუკიდებელი პროგრამა-დანართების (Application) შემუშავებასა და შესრულებას. იგი, აგრეთვე, უზურნველყოფს პროგრამების მობილობას. შედეგად, Windows - პროგრამები შგვიძლია სხვა პლატფორმებზე ვამუშაოთ.

Visual Studio.NET არის პროგრამული სისტემების დამუშავების ინტეგრირებული გარემო, რომელშიც შესაძლებელია კოდების დაწერა, კომპილირება და გამართვა VB.NET, C++.NET, C#.NET, ASP.NET, ADO.NET და სხვა ტექნოლოგიებით. .NET გარემოში დაპროგრამება დაფუძნებულია ობიექტებზე. ობიექტი (object) – არის პროგრამული კონსტრუქცია, რომელშიც ინკაფსულირებულია ლოგიკურად დაკავშირებული მონაცემებისა და მეთოდების ერთობლიობა. ობიექტი (Object) განიხილება, როგორც გარკვეული არსი (Entity), რომელიც ხასიათდება მდგომარეობით (მონაცემთა ერთობლიობა) და ქცევით (ფუნქციური პროგრამები). ობიექტის ქცევა ანუ რეაქცია, რომლის დროსაც მისი ახალი მდგომარეობა განისაზღვრება, დამოკიდებულია გარედან მოსულ ინფორმაციაზე, შეტყობინებებზე. ვინაიდან ობიექტი უმთავრესი ცნებაა, იგი საწყისია ობიექტ-ორიენტირებული დაპროგრამების [66].

თანამედროვე Workflow Foundation ტექნოლოგია. Workflow Foundation ტექნოლოგია .NET 5.0-ში არის სრულიად ახალი პარადიგმა სამუშაო პროცესებზე (workflow) ბაზირებული აპლიკაციების ასაგებად. იგი ფუნდამენტურად ახლად გააზრებული ტექნოლოგიაა. ჩვენ მას მე-3 თავში პრაქტიკულად განვიხილავთ.

4.4. საინფორმაციო სისტემების ინფორმაციული უზრუნველყოფა Visual Studio.NET გარემოში

საინფორმაციო სისტემა არის სტრუქტურიზებული მონაცემებისა და აპარატურულ-პროგრამული საშუალებების ერთობლიობა, რომელიც განკუთვნილი ინფორმაციის შენახვისა და დამუშავებისათვის.

არსებობს საინფორმაციო სისტემების ორი კლასი: საინფორმაციო-საძიებო და მონაცემების დამუშავების სისტემები. პირველი მათგანი ორიენტირებულია ჩვენთვის საჭირო მონაცემების ძებნასა და მიღებაზე, მეორე კი - მონაცემების დამუშავებასა და დამუშავებული ინფორმაციის მიწოდებაზე. საინფორმაციო სისტემა ასრულებს შემდეგ ფუნქციებს: ინფორმაციის შეტანა, ცვლილება და შენახვა; ინფორმაციის ნახვა და ძებნა; ინფორმაციის ამორჩევა გარკვეული კრიტერიუმის მიხედვთ; ანგარიშების ფორმირება; ინფორმაციის სისწორის შემოწმება.

მონაცემთა განაწილებული ბაზა რამდენიმე ნაწილისაგან შედგება, რომლებიც მოთავსებულია ქსელის სხვადასხვა კომპიუტერზე (სერვერებზე). მონაცემთა ცენტრალიზებული ბაზა მოთავსებულია ერთ კომპიუტერზე, რომელიც შეიძლება იყოს ლოკალურ ქსელში ჩართული და შესაბამისად, შესაძლებელი იქნება სხვა კომპიუტერების მხრიდან ამ ბაზასთან მიმართვა. მონაცემთა ცენტრალიზებული ბაზა არქიტექტურის მიხედვით ორგვარია: ფაილ- სერვერი და კლიენტ-სერვერი [9].

კლიენტ-სერვერული არქიტექტურის შემთხვევაში მონაცემთა ბაზა განთავსებულია სერვერზე და აქვე ხდება მისი დამუშავება. სერვერს ეგზავნება კლიენტის მხრიდან მონაცემების დამუშავების მოთხოვნა. მასზე სრულდება მონაცემების დამუშავება და შედეგები ეგზავნება კლიენტს. სისტემა, რომელიც იყენებს კლიენტ-სერვერულ ტექნოლოგიას ორი ნაწილისაგან შედგება: კლიენტის ნაწილი (front-end) და სერვერის ნაწილი (back-end). კლიენტის ნაწილი უზრუნველყოფს გრაფიკულ ინტერფეისს და იმყოფება მომხმარებლის კომპიუტერზე. სერვერის ნაწილი მოთავსებულია სერვერზე და უზრუნველყოფს მონაცემების მართვას, დანაწილებას, ადმინისტრირებასა და უსაფრთხოებას. კლიენტ-სერვერული მონაცემთა ბაზების მართვის მძლავრი და საერთაშორისო სტანდარტად აღიარებული სისტემებია: Microsoft SQL Server, Oracle, IBM, DB2, Sybase და ა.შ.

ამ არქიტექტურისათვის დამახასიათებელია მოთხოვნების სტრუქტურირებული ენის (SQL, Structured Query Language) გამოყენება, რომელიც საუკეთესო საშუალებაა მონაცემთა მანიპულირებისათვის.

4.4.1. MsSQL Server პაკეტი

MsSQL სერვერი წარმოადგენს მონაცემთა ბაზების მართვის თანამედროვე და ფართოდ გავრცელებულ სისტემას, რომელიც წარმატებით გამოიყენება ქსელში ჩართულ კომპიუტერებთან და მოწყობილობებთან სამუშაოდ.

დღევანდელი საერთაშორისო სტანდარტებით ნებისმიერი ორგანიზაციის მუშაობა განიხილება როგორც ბიზნეს-პროცესების ურთიერთდაკავშირებული ერთობლიობა. ბიზნეს-პროცესი არის ორგანიზაციაში მიმდინარე პროცესების, ერთმანეთთან დაკავშირებული საქმიანობების და შიგა თუ გარე საწარმოო ოპერაციების დეტალური აღწერა. ბიზნეს-პროცესის სრული აღწერა ნიშნავს მისი საზღვრების დადგენას, შიგა ორგანიზაციული ფუნქციების, ეკონომიკურ-ფინანსური მაჩვენებლების, ტექნოლოგიური შესრულების პროცესის, რესურსების, საქმიანი ციკლების, მიმწოდებლებსა და მომხმარებლებს შორის კავშირებისა და ოპერაციების განსაზღვრას. ინფორმაციული უზრუნველყოფა ბიზნესის არსებობის ერთერთი მნიშვნელოვანი რესურსია, რომლის ფლობა და მართვა პრაქტიკულად ნებისმიერი ორგანიზაციის ფუნქციონირების ბირთვია.

Data Base Management System (DBMS) - მონაცემთა ბაზების მართვის სისტემა (მბმს) ზოგადად წარმოადგენს მონაცემთა ბაზების აგების და მისი მართვის ინსტრუმენტულ საშუალებას. კერძოდ, იგი პროგრამული პროდუქტია, რომელშიც რეალიზებულია მონაცემთა აღწერისა ((Data Definition Language - DDL)) და მანიპულირების ენები (Data Manipulation Language - DML), კონცეპტუალური, ლოგიკური და ფიზიკური სტრუქტურების აგების და მოდიფიკაციის პროცედურები, მონაცემების დაცვის და საიმედოობის საშუალებანი და ა.შ.

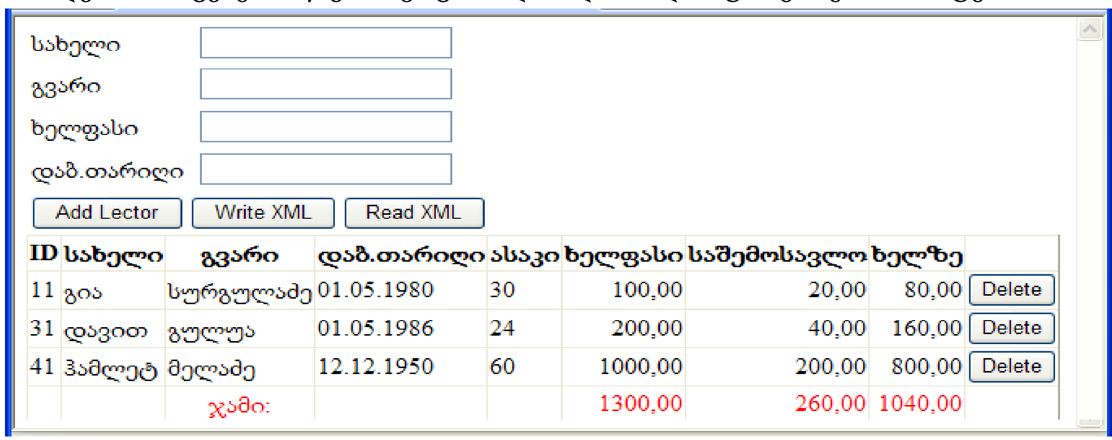
MsSQL Server მონაცემებთან მიმართვისათვის იყენებს ოთხ ძირითად ინტერფეისს: OLE DB, ODBC, DB Library და Transact_SQL, ხოლო მოთხოვნების დამუშავების SQL-ენის ნაცვალდ იყენებს Transact_SQL დიალექტს. ამ ენის ინსტრუქციები დაყოფილია სამ ქვესიმრავლედ: DDL - მონაცემთა ბაზების ცხრილებისა და წარმოდგენების შესაქმნელად, DML- მოთხოვენბის შესაქმნელად და მონაცემთა დასამუშავებლად. DCL (Data Control Language) - მონაცემთა ბაზასთან მიმართვის პროცედურების სამართავად [31]. ჩვენს

შემთხვევაშიც, საინფორმაციო სისტემის სარეალიზაციოდ გათვალისწინებულია ცენტრალიზებული მონაცემთა ბაზა Microsoft SQL Server მონაცემთა მართვის სისტემის ბაზაზე.

4.4.2. XML მონაცემთა ბაზები

Ms Visual Studio.NET გარემოში გარდა SQL Server და სხვა ბაზებისა ერთ-ერთი მნიშვნელოვანი მეტად მოსახერხებელია შედეგების XML ფაილში შენახვა და XML ფაილიდან მათი ამოღების პროცედურების შექმნა. მათ XML ბაზები შეესაბამება, ხდება მის ფორმატში შენახვა [63].

განვიხილოთ მაგალითი, რომელშიც ინტერაქტიული მონაცემები შეიტანება და თავსდება ცხრილში (GridViev) შემდეგ კი სპეც-ღილაკებით გადაიგზავნება XML ფაილში შესაბახად (ნახ.4.51). შემდგომ ამ XML ფაილიდან მოხდება მონაცემების წაკითხვა ცხრილში და ახალი სტრიქონების ჩამატება.



ID	სახელი	გვარი	დაბ.თარიღი	ასაკი	ხელფასი	საშემოსავლო	ხელზე	
11	გია	სურგულაძე	01.05.1980	30	100,00	20,00	80,00	Delete
31	დავით	გულუა	01.05.1986	24	200,00	40,00	160,00	Delete
41	ჰამლეტ	მელაძე	12.12.1950	60	1000,00	200,00	800,00	Delete
		ჯამი:			1300,00	260,00	1040,00	

ნახ. 4.51

ორი ღილაკი: Write_XML (სტრიქონების შესანახად XML ფაილში) და Read_XML (სტრიქონების ამოსაღებად XML ფაილიდან) ასე გამოიყურება:

```
protected void Button2_Click(object sender, EventArgs e)   // Write_XML-------
  {
      DataSet ds = Session["MyDataSet"] as DataSet;
      ds.WriteXml(Request.PhysicalApplicationPath + "\\lectors.xml");
      //Response.Redirect("~/lectors.xml");
  }
protected void Button3_Click(object sender, EventArgs e)  // Read_XML ------------
  {
      DataSet ds = Session["MyDataSet"] as DataSet;
```

```
ds.ReadXml(Request.PhysicalApplicationPath + "\\lectors.xml");
DataTable dtLectors = ds.Tables["Lectors"];
object sumSalary = dtLectors.Compute("SUM(Salary)", "");
object sumTax = dtLectors.Compute("SUM(Tax)", "");
object sumNettoSalary = dtLectors.Compute("SUM(NettoSalary)", "");
//lblSalary.Text = sumSalary.ToString();
GridView4.Columns[5].FooterText = String.Format("{0:F2}", sumSalary);
GridView4.Columns[6].FooterText = String.Format("{0:F2}", sumTax);
GridView4.Columns[7].FooterText = String.Format("{0:F2}", sumNettoSalary);
GridView4.DataSource = ds;
GridView4.DataBind();    }
```

XML ფაილის სტრუქტურას, შეტანილი ჩანაწერებით ექნება ასეთი სახე:

```
<!—— ლისტინგი_4.9 — XML ჩანაწერების შენახვის სტრუქტურა –>
<?xml version="4.0" standalone="yes"?>
<NewDataSet>
 <Lectors>
  <ID>11</ID>
  <FirstName>გია</FirstName>
  <LastName>სურგულაძე</LastName>
  <Salary>100</Salary>
  <BirthDate>1980-05-01T00:00:00+04:00</BirthDate>
 </Lectors>
 <Lectors>
  <ID>31</ID>
  <FirstName>დავით</FirstName>
  <LastName>გულუა</LastName>
  <Salary>200</Salary>
  <BirthDate>1986-05-01T00:00:00+04:00</BirthDate>
 </Lectors>
 <Lectors>
  <ID>41</ID>
  <FirstName>გიორგი</FirstName>
  <LastName>სურგულაძე</LastName>
  <Salary>1900</Salary>
  <BirthDate>1980-12-30T00:00:00+04:00</BirthDate>
 </Lectors>
</NewDataSet>
```

4.5. მათემატიკური მოდელირება ორგანიზაციული მართვის საინფორმაციო სისტემებში

4.5.1. რიგების თეორია

მართვის ავტომატიზებული სისტემის (მას) შემუშავების დროს ფართოდ იყენებენ საინჟინრო პრაქტიკისათვის დამახასიათებელ მათემატიკურ მეთოდებს. დიდ ორგანიზაციებში მმართველი გადაწყვეტილებების პრაქტიკული განხორციელება დაკავშირებულია მატერიალური და შრომითი რესურსების საგრძნობ დანახარჯებთან, რაც შეიძლება კიდევ უფრო გაზარდოს შრომითი რესურსების არასწორმა გადანაწილებამ, რომელიც პროპორციულია ეფექტური მომსახურების ზრდის მაჩვენებელთან, განსაკუთრებით კი მასობრივი მომსახურების სიტემებში. რასაკვირველია, გადაწყვეტილების ამორჩევა შესაძლებელია იმ შემთხვევაში, როდესაც არსებობს ვარიანტების სიმრავლე. ამავე დროს ვარიანტის ხარისხის შეფასებისათვის საჭიროა მმართველი გადაწყვეტილების შედეგის პროგნოზირება. სწორედ ამის საშუალებას იძლევა მათემატიკური მოდელირება.

როგორც ცნობილია, მოდელი არის განსაზღვრულ ობიექტში მიმდინარე პროცესების ან მოვლენების მიახლოებითი ასახვა (წარმოდგენა). მოდელის ძირითადი დანიშნულებაა მისი გამოყენება მართვასა და პროგნოზირებაში. გარდა ამისა, მოდელი საშუალებას იძლევა გამოვიკვლიოთ სამართავი ობიექტის ცალკეული თვისებების ურთიერთგავლენა ისე, რომ არ ჩავატაროთ ობიექტზე რაიმე ექსპერიმენტი.

კორპორაციული სისტემებისათვის კი დამახასიათებელია ორგანიზაციული მართვა, რომლის არსია: წარმოების ორგანიზაცია, მატერიალური და შრომითი რესურსების განაწილება, წარმოების მომარაგება, გამოსაშვები პროდუქციის რაოდენობისა და ასორტიმენტის განსაზღვრა, გადასაზიდი პროდუქციის რაოდენობის და გადაზიდვის მიმართულების განსაზღვრა, საწარმოო პროცესების პარამეტრების ოპერატიული აღრიცხვა და ა.შ. [9].

სწორედ, მასობრივი მომსახურების თეორიის ძირითად ამოცანას წარმოადგენს ოპტიმალური რაოდენობრივი დამოკიდებულების დადგენა მოთხოვნების შემავალ ნაკადს, მომსახურე საშუალებებსა და მოთხოვნების გამომავალ ნაკადს შორის. აქ მისაღებია ინგლისელი მათემატიკოსის, დავიდ

კენდალის (David Kendall, 1918-2007) ნოტაცია რიგების თეორიის სტანდარტიზაციის და კლასიფიკაციის საკითხებზე [70].

მიმდინარე პროცესების ეკონომიკური ანალიზისთვის უნდა იყოს გათვალისწინებული შემდეგი ფაქტორები: მასობრივი მომსახურების სისტემა (მაგალითად, ჩვენ შემთხვევაში შემოსავლების სამსახური; დავების განხილვის საბჭო); განაცხადი (საჩივარი); რიგი; შემოსული განაცხადის (საჩივრების) ინტენსივობა; მომსახურების ინტენსივობა; საშუალო დრო, რომელიც განაცხადს (საჩივარს) ესაჭიროება რიგში დგომისას; რიგის საშუალო სიგრძე; საშუალო დრო, რომელიც ესაჭიროება განაცხადს (საჩივარს) სისტემაში მომსახურებისათვის (მიღება-განხილვა); მომსახურე სისტემაში მომჩივანთა საშუალო რიცხვი; სისტემის მომსახურების ხარჯი; ლოდინის ხარჯი.

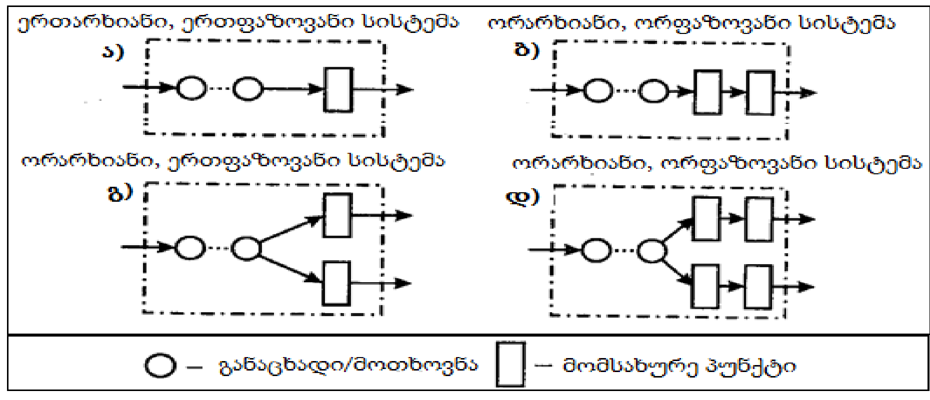
მასობრივი მომსახურების სისტემებში განასხვავებენ სამ ძირითად ეტაპს, რომელსაც გადის ყოველი მოთხოვნა:

1) სისტემის შესასვლელზე მოთხოვნის მიღება;

2) რიგის გავლა;

3) მომსახურების პროცესი, რომლის შემდეგაც მოთხოვნა ტოვებს სისტემას.

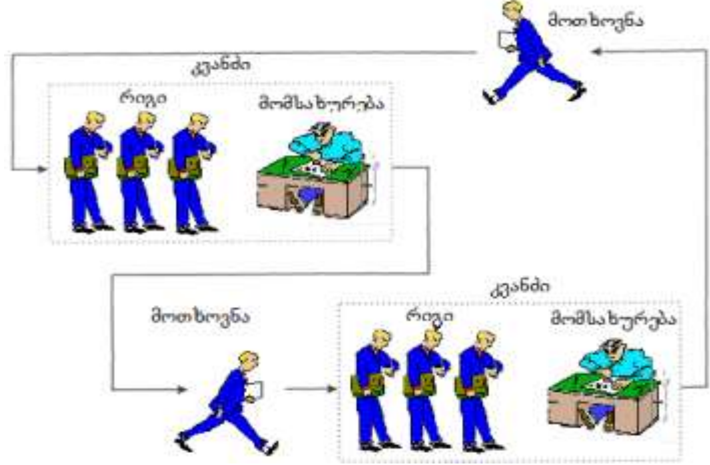
ქვემოთ წარმოდგენილია მომსახურების სისტემის სხვადასხვა ტიპები (ნახ. 4.52) [23]:



ნახ. 4.52. მომსახურები სისტემის ტიპები

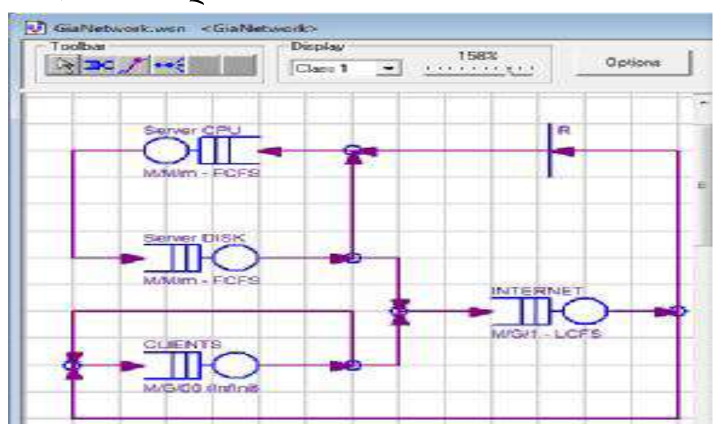
4.5.2. WinPepsy ინსტრუმენტული საშუალება

რიგების ქსელი შედგება ცალკეული რიგებისა და მომსახურე ობიექტებისგან. რიგი, რომელიც ელოდება მომსახურებას, შეიძლება წარმოვადგინოთ როგორც მომლოდინე სისტემა ან კვანძი. 4.53 ნახაზზე მოცემულია რიგების ქსელი და რიგების სისტემა (კვანძი) [23]:



ნახ. 4.53

4.54 ნახაზზე ნაჩენებია საილუსტრაციო მაგალითი WinPepsy გარემოში ქსელის ასაგებად და რიგების თეორიაზე დაყრდნობით ქსელის კვანძის ტიპის შერჩევისა: M/M/1, M/M/m და ა.შ.



ნახ. 4.54

ამ ინსტრუმენტის ფუნქციონირებას და მასზე კონკრეტული ამოცანების გადაწყვეტას განვიხილავთ მომდევნო თავში.

თავი 5
კორპორაციული მენეჯმენტის ბიზნესპროცესების საინფორმაციო სისტემის ვიზუალური და მათემატიკური მოდელირება რეალურ ობიექტზე

5.1. კორპორაციული მენეჯმენტის ბიზნეს-პროცესების მოდელირება UML/2 ტექნოლოგიით

კორპორაციული მართვის ობიექტის სახით ვიხილავთ *ფინანსთა სამინისტროს შემოსავლების სამსახურის საბაჟო დეპარტამენტს.* საპრობლემო სფერო კი წამოადგენს *საბაჟო გამშვები პუნქტისთვის შემოსავლების სამსახურისა და დავების განხილვის საბჭოში შესული საჩივრების განხილვის (საქართველოს საგადასახადო* კოდექსის საფუძველზე) ბიზნეს-პროცესების ოპერატიული მართვის სისტემას [33-35].

მოცემულ თავში განიხილება საბაჟო გამშვებ პუნქტ „თბილისის აეროპორტში" მიმდინარე ბიზნეს-პროცესების მოდელირების საკითხები UML2-ტექნოლოგიის ბაზაზე. აღნიშნული ბიზნეს-პროცესები დაკავშირებულია უშუალოდ მგზავრის მიერ საჰაერო გზით საქონლის, ნივთების ბარგითა და ხელბარგით საქართველოს საბაჟო საზღვარზე გადაადგილებასთან. არსებული ბიზნესპროცესების და ბიზნეს-წესების სრული დაცვით შემოთავაზებულია პრეცედენტების, აქტიურობათა და მიმდევრობითობის დიაგრამები, რომელთა საფუძველზეც განისაზღვრება დასაპროექტებელი სისტემის პროგრამული უზრუნველყოფის მოთხოვნები.

5.1.1. საპრობლემო სფეროს ბიზნესპროცესების და ბიზნესწესების აღწერა

შემოსავლების სამსახურიის საბაჟო დეპარტამენტი მიეკუთვნება იმ ორგანიზაციული მართვის დიდ სისტემებს, რომელიც თავისუფლად შეიძლება განვიხილოთ, როგორც კორპორაცია.

კორპორაცია რთული განაწილებული სისტემაა, რომლის ტოპოლოგია მოიცავს მრავალ განყოფილებასა და დეპარტმენტს. შემოსავლების სამსახურის ეკონომიკური საზღვრის დაცვის დეპარტამენტის მთავარი სამმართველოს

ფუნქციებიდან ერთ-ერთი უმთავრესი, სწორედ საქართველოს საბაჟო საზღვარზე გადაადგილებული საქონლის და სატრანსპორტო საშუალებების აღრიცხვა და სახელმწიფო კონტროლის ორგანიზებაა.

სასაზღვრო კონტროლის ზონა – საბაჟო გამშვები პუნქტი არის საქართველოს საბაჟო საზღვართან მდებარე კონტროლის ზონა, სადაც მგზავრის, საქონლისა და სატრანსპორტო საშუალების მიმართ ხორციელდება საქართველოს საგადასახადო კოდექსით დადგენილი პროცედურები. სწორედ საქართველოს საბაჟო ტერიტორიაზე შემოტანა/ეკონომიკური ტერიტორიიდან გატანა ხორციელდება სგპ-ს გავლით.

ერთ-ერთი სასაზღვრო კონტროლის ზონა არის სგპ „თბილისის აეროპორტი“, რომლის (სხვა სგპ-ების მსგავსად), მნიშვნელოვანი ფუნქციებია:

- საქართველოს საბაჟო საზღვარზე გადაადგილებული საქონლისა და სატრანსპორტო საშუალებების ფიტოსანიტარული და ვეტსანიტარული კონტროლი;

- საქართველოს საბაჟო საზღვარზე გადაადგილებული საქონლისა და სატრანსპორტო საშუალებების სახელმწიფო კონტროლის სხვა სახეების კოორდინაცია და აგრეთვე სამკურნალო საშუალებათა, ნარკოტიკულ საშუალებათა და პრეკურსორების, იარაღის, ორმაგი დანიშნულების საქონლის, სახიფათო ნარჩენების, კულტურულ ფასეულობათა საქართველოს ეკონომიკურ საზღვარზე გადაადგილების კონტროლი, რადიაციული და სავალუტო კონტროლი, საავტორო უფლებების დაცვაზე კონტროლი;

- დანიშნულების საგადასახადო ორგანომდე ან/და გაფორმების ადგილამდე იმ საქონლის, რომლის საიმედო იდენტიფიცირება სგპ-ს ტერიტორიაზე შეუძლებელია, გადაადგილებაზე უშუალო ზედამხედველობის განხორციელება სგპ-ს თანამშრომლის თანხლებით;

- კომპეტენციის ფარგლებში იმპორტის გადასახდელების დარიცხვა-ამოღება და ბიუჯეტში მათი დროული და სრული აკუმულირება;

- კომპეტენციის ფარგლებში, საგადასახადო და ადმინისტრაციულ სამართალდარღვევათა გამოვლენა–აღკვეთა, შესაბამისი სამართალ-დარღვევის ოქმის შედგენა, კანონით გათვალისწინებულ შემთხვევებში და წესით სამართალდარღვევის ადგილზე ჯარიმის დაკისრება, სამართალდარღვევის ოქმის საფუძველზე;

- სამართალდარღვევის ოქმის განხილვის შედეგებზე სგპ-ს უფროსის ბრძანების და „საგადასახადო მოთხოვნის“ პროექტის მომზადება. საგადასა-ხადო/ადმინისტრაციული სამართალდარღვევების აღრიცხვა;

- კანონმდებლობით მინიჭებული სხვა უფლებამოსილების განხორციელება;

- დეპარტამენტის უფროსის/მოადგილის ცალკეული დავალებების და მითითებების შესრულება;

- კომპეტენციის ფარგლებში ფიზიკური პირების მიერ შემოტანილი საქონლის გაფორმება (საჰაერო გზით შემოტანილი საქონლის შემთხვევაში სრულყოფილი დოკუმენტაციის წარმოდგენისას ფორმდება ეკონომიკური დანიშნულების ნებისმიერი ღირებულების საქონელი, გარდა აქციზური საქონლისა და ასევე გაფორმებას ექვემდებარება ფიზიკური პირის მიერ შემოტანილი საქონელი, როდესაც იგი ვერ სარგებლობს საგადასახადო კოდექსის 199 მუხლის „დ“ პუნქტის „დ.ა“, „დ.გ“, „დ.დ“, „დ.ე“, „დ.ზ“ ქვეპუნქტებით გათვალისწინებული შეღავათებით: „დ.ა) კალენდარულ დღეში ერთხელ საგარეო-ეკონომიკური საქმიანობის ეროვნული სასაქონლო ნომენკლატურის 02, 04, 06-12, 15-21 ჯგუფებისა და 0302-0307, 2201-2202 სასაქონლო პოზიციების შესაბამისი ჯამური თანხით 500 ლარამდე ღირებულების, 30კგ-მდე საერთო წონის კვების პროდუქტების იმპორტი, რომელიც არ არის განკუთვნილი ეკონომიკური საქმიანობისათვის“;

- „დ.გ) საჰაერო ტრანსპორტით შემოტანის შემთხვევაში ერთი კალენდარული დღის, ხოლო სხვა შემთხვევაში – 30 კალენდარული დღის განმავლობაში 400 ღერი სიგარეტის ან 50 სიგარის ან 50 სიგარილას ან 250 გრამი თამბაქოს სხვა ნაწარმის (გარდა თამბაქოს ნედლეულისა) ან ამ ზღვრული ოდენობის ფარგლებში თამბაქოს ამ ქვეპუნქტში დასახელებული ნაწარმის საერთო წონით 250 გრამის იმპორტი; ასევე 4 ლიტრი ყველა სახის ალკოჰოლიანი სასმელის იმპორტი“;

- „დ.დ) უცხოეთში ყოველ 6 თვეზე მეტი ხნით ყოფნის შემდეგ საქართველოში შემოსული ფიზიკური პირის მიერ საგარეო-ეკონომიკური საქმიანობის ეროვნული სსაქონლო ნომენკლატურის 28-ე-97-ე ჯგუფების (გარდა 87-ე ჯგუფისა) შესაბამისი 15 000 ლარის ღირებულების საქონლის იმპორტი, რომელიც არ არის განკუთვნილი ეკონომიკური საქმიანობისათვის, ხოლო უცხო ქვეყანაში საქართველოს დიპლომატიური წარმომადგენლობიდან ან საკონსულო დაწესებულებიდან როტაციის წესით სამუშაო მივლინებიდან გამოწვეული დიპლომატიური თანამდებობის პირისათვის – დამატებით მის

მიერ პირადი სარგებლობისათვის გაკუთვნილი საქონლის (ოჯახზე თითო სატრანსპორტო საშუალების, მაცივრის, კომპიუტერისა და ტელევიზორის) იმპორტი“;

- „დ.ე) საქართველოში მუდმივად საცხოვრებლად შემოსვლისას (რაც დასტურდება საქართველოს იუსტიციის სამინისტროს მიერ დადგენილი წესით გაცემული შესაბამისი დოკუმენტით) საქონლის (მათ შორის, ავეჯის, საყოფაცხოვრებო დანიშნულების საქონლის, ოჯახზე ერთი სატრანსპორტო საშუალების) იმპორტი, რომელიც არ არის განკუთვნილი ეკონომიკური საქამიანობისათვის“;

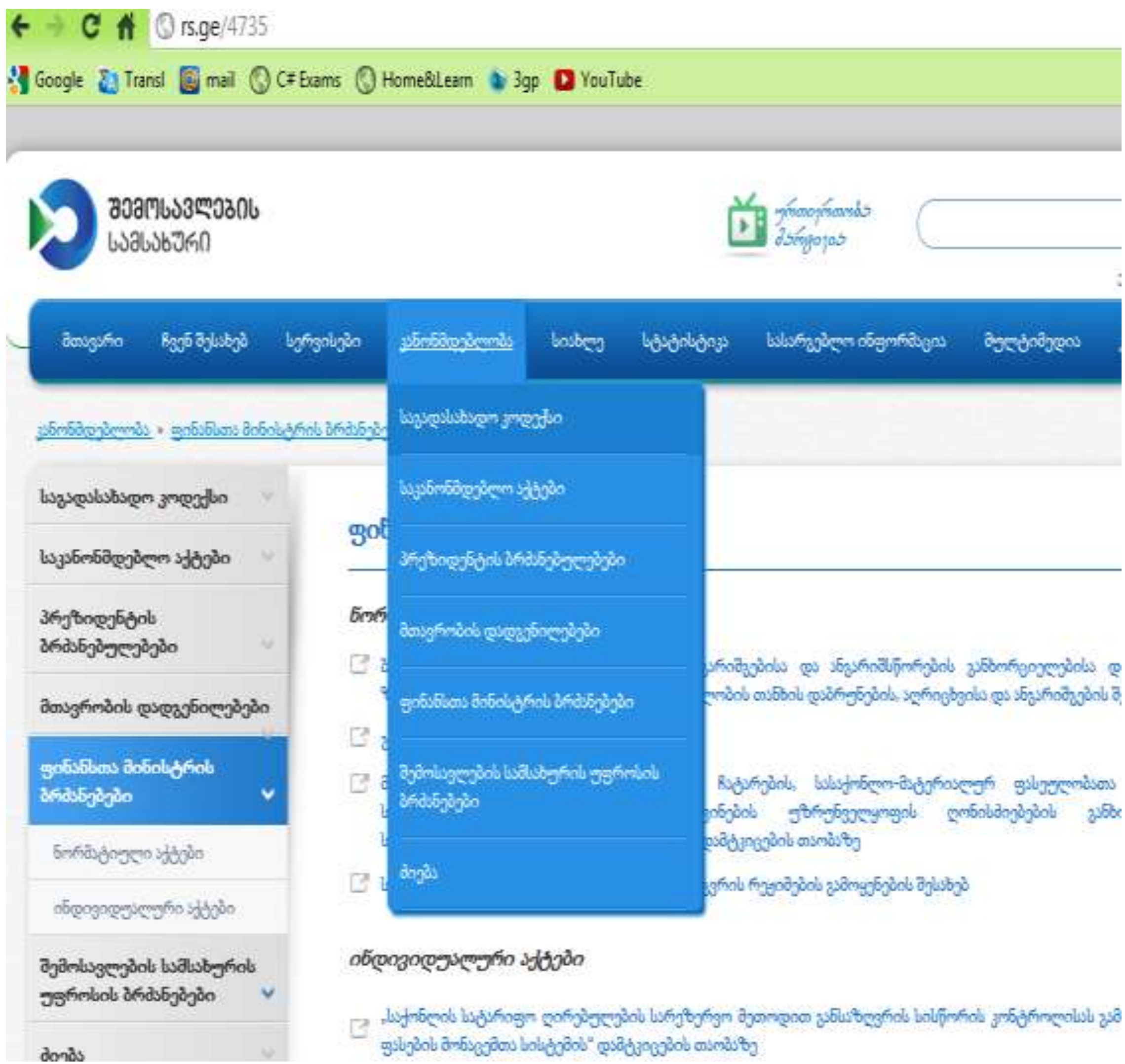
- „დ.ზ) საჰაერო ტრანსპორტით შემოტანის შემთხვევაში საგარეო-ეკონომიკური საქმიანობის ეროვნული სასაქონლო ნომენკლატურის 28-ე-97-ე ჯგუფების შესაბამისი 3 000 ლარამდე ღირებულების, 30 კგ-მდე საერთო წონის საქონლის იმპორტი, რომელიც არ არის განკუთვნილი ეკონომიკური საქმიანობისათვის“.

საქონლის ან/და სატრანსპორტო საშუალების მიერ საბაჟო საზღვრის გადაკვეთის (შემოტანა ან/და გატანა) მიზანსა და წესს საგადასახადო კანონმდებლობის თანახმად ეწოდება სასაქონლო (საბაჟო) ოპერაცია. იგი არის 9 სახის: იმპორტი, ექსპორტი, დროებითი შემოტანა, ტრანზიტი და ა.შ.

აქედან ჩვენს ნაშრომში მოყვანილია სისტემის მოდელის ფრაგმენტი, რომელიც იმპორტს ეხება. საქონლის იმპორტში გაშვება გულისხმობს მის საქართველოში მუდმივ დატოვებას. ამ სასაქონლო ოპერაციის გამოყენება შესაძლებელია საქართველოს საბაჟო ტერიტორიაზე შემოტანილი უცხოური საქონლის მიმართ, რომლის დროსაც საქონელს ენიჭება საქართველოს საქონლის სტატუსი და ასევე გამოიყენება:

1) სავაჭრო პოლიტიკის ღონისძიებები – სატარიფო და არასატარიფო ღონისძიებები (ნებართვები, ლიცენზიები, აკრძალვები, შეზღუდევები და ა.შ);

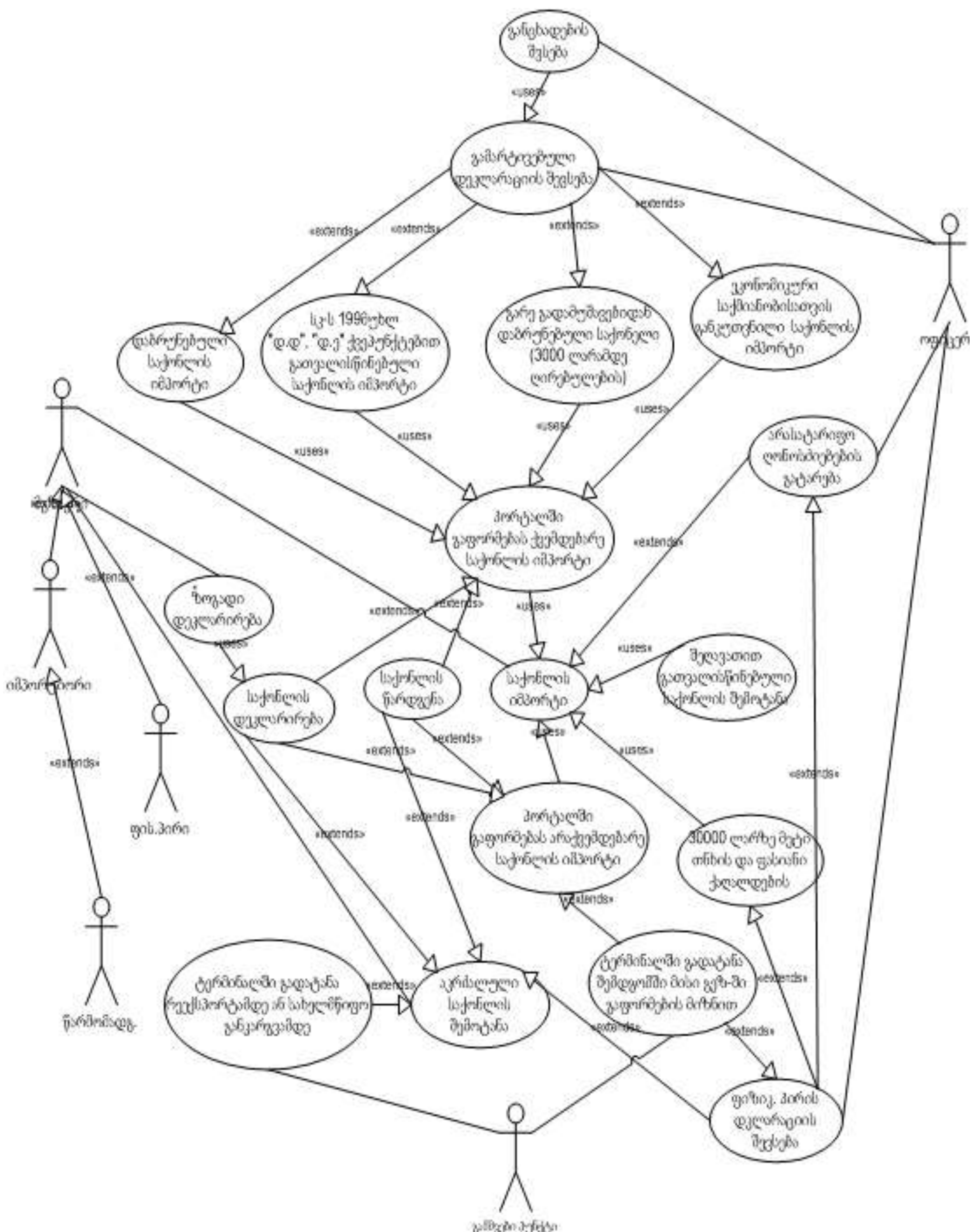
2) ხორციელდება მისი შემოტანისას გათვალისწინებული პროცედურები; გადაიხდევინება იმპორტის გადასახდელები (გარდა ზემოთ აღნიშნული შეღავათებით სარგებლობისას და ასევე გარკვეული შემთხვევებისა). ეს ყოველივე რეგულირდება პრეზიდენტის ბრძანებულებებით, მთავრობის დადგენილებებით, ფინანსთა მინისტრის ბრძანებებით, შემოსავლების სამსახურის უფროსის ბრძანებებით და სხვა საკანონმდებლო აქტებით, რომელიც, რათქმა უნდა, არის საჯარო და მისი ხილვა მრავალ საიტზეა შესაძლებელი (ნახ.5.1).



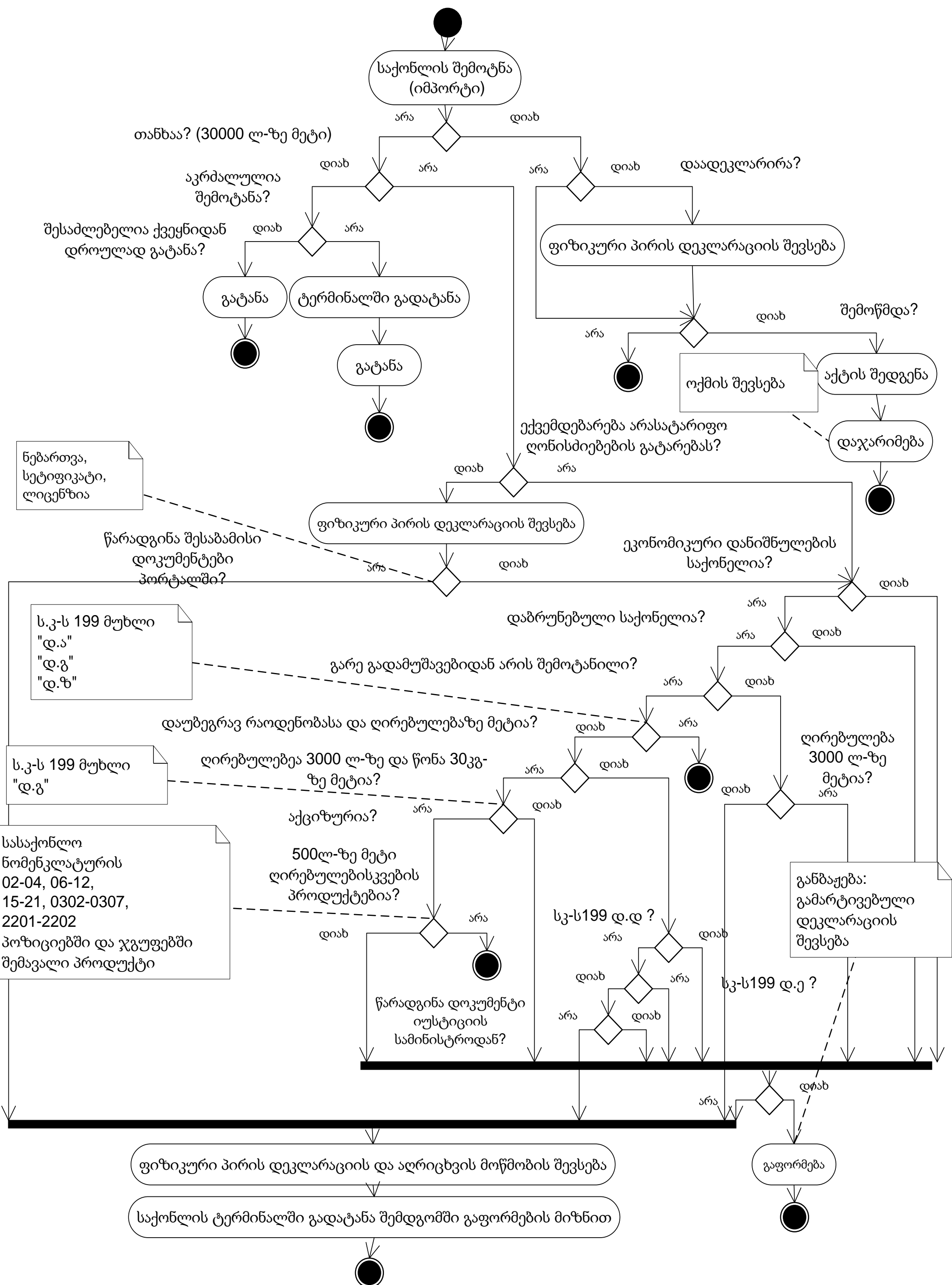
ნახ. 5.1. საიტის ფრაგმენტი

5.1.2. საპრობლემო სფეროს Use Case და Activity დიაგრამები

ზემოთ აღწერილი ბიზნესპროცესებისა და ბიზნესწესების საფუძველზე UML მეთოდოლოგიის გამოყენებით შევიმუშავეთ UseCase (გამოყენებით შემთხვევათა) დიაგრამა (ნახ.5.2) და Activity (აქტიურობათა) დიაგრამა (ნახ.5.3).



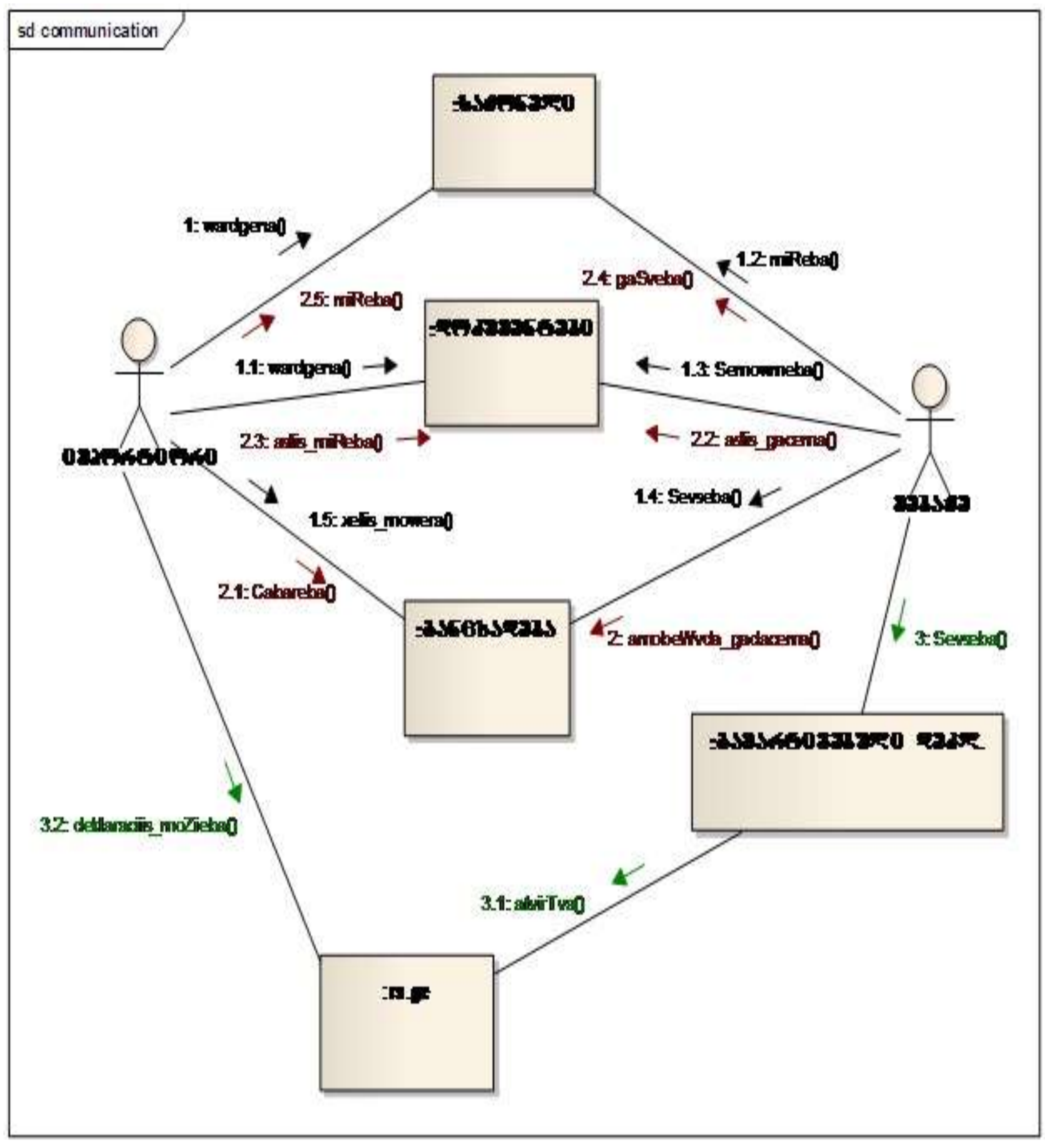
ნახ. 5.2. UseCase დიაგრამა - როლები და ფუნქციები:
„იმპორტი, სგპ „თბილისის აეროპორტი“



ნახ. 5.4. Activity დიაგრამა: „იმპორტი, სგპ „თბილისის აეროპორტი“

5.1.3. საპრობლემო სფეროს ამოცანის კომუნიკაციის დიაგრამა

სგპ-ში საქონლის გაფორმების პროცედურა აღწერილია კომუნიკაციის დიაგრამით, რომელიც განცხადების შევსებიდან უკვე შევსებული გამარტივებული დეკლარაციის საიტზე ატვირთვით დამთავრებული, ოფიცერსა და იმპორტიორს შორის კომუნიკაციას აღწერს (ნახ.5.4).



ნახ.5.4. კომუნიკაციის დიაგრამა: „საქონლის გაფორმება სგპ-ში“

5.1.4. საგადასახადო დავების გადაწყვეტის მხარდამჭერი სისტემის UML/2 -დიაგრამები

განვიხილოთ საგადასახადო სისტემაში მიმდინარე ბიზნეს-პროცესების მოდელირების, კერძოდ საგადასახადო (საბაჟო) სამართალდარღვევის საქმის წარმოების უნიფიცირებული მოდელირების საკითხები UML2-ტექნოლოგიის ბაზაზე. საპრობლემო სფეროსთვის ავაგეთ პრეცედენტების, აქტიურობის და მიმდევრობითობის დიაგრამები. არსებული ბიზნეს-პროცესების საფუძველზე, ბიზნეს-წესების, იურიდიული კანონმდებლობისა და შესრულების რეგლა-მენტის სრული დაცვით წარმოვადგინეთ დროითი დიაგრამის სქემა. შედეგად, კი განვსაზღვრეთ დასაპროექტებელი სისტემის პროგრამული უზრუნველყოფის ფუნქციონალური მოთხოვნები.

შემოსავლების სამსახური თავისი დეპარტამენტებით მიეკუთვნება ორგანიზაციული მართვის დიდ და რთულ სისტემებს. იგი კორპორაციაა, რომელიც მრავალ ბიზნეს-პროცესს მოიცავს. იმისათვის, რომ გამარტივდეს ამ სისტემის პროგრამული უზრუნველყოფის შექმნა (ან სრულყოფა), მისი ობიექტ-ორიენტირებული მოდელირება, ანალიზი და პროექტირებაა საჭირო [14].

კორპორაციული განაწილებული სისტემის ტოპოლოგია მოიცავს მრავალ განყოფილებასა და დეპარტმენტს. შემოსავლების სამსახურის აუდიტის და საბაჟო დეპარტამენტების მრავალი ფუნქციიდან მნიშვნელოვანია საგადასახადო სამართალდარღვევების გამოვლენა და საგადასახადო დავის წარმოება. იმისდა მიხედვით თუ მომჩივანის მიერ საჩივარი დავის განმხილველ რომელ ორგანოში შეიტანება, საგადასახადო სამართალდარღვევის ოქმი და თანდართული მასალები შეიძლება შემოსავლების სამსახურის გარდა ფინანსთა სამინისტროს შემადგენლობაში შემავალი დავების განხილვის საბჭომ ან სასამართლომ განიხილოს.

საგადასახადო სამართალდარღვევა და პასუხისმგებლობა რეგულირდება საგადასახადო კოდექსის XIII კარით, ხოლო საგადასახადო დავის წარმოება XIV თავით [33]. საგადასახადო კოდექსის შინაარსის ობიექტ-ორიენტირებული ანალიზის საფუძველზე, გამოვლენილ იქნა ის მნიშვნელოვანი პუნქტები, რომელთა სემანტიკა აუცილებელია აისახოს საგადასახადო დავის განხილვის ფუნქციონალური ამოცანის ავტომატიზებული საქმისწარმოების სისტემის UML/2 მოდელის დიაგრამებში [35]. ტექსტურ-შინაარსობრივი დებულებები შემდეგია [33]:

„საგადასახადო სამართალდარღვევად ითვლება პირის მართლსაწინააღმდეგო ქმედება (მოქმედება ან უმოქმედობა), რომლისთვისაც ამ კოდექსით გათვალისწინებულია პასუხისმგებლობა. საგადასახადო სამართალდარღვევისათვის პირს პასუხისმგებლობა შეიძლება დაეკისაროს მხოლოდ ამ კოდექსით დადგენილი საფუძვლითა და წესით;

საგადასახადო სანქცია არის პასუხისმგებლობის ზომა ჩადენილი საგადასახადო სამართალდარღვევისათვის;

საგადასახადო სანქცია გამოიყენება გაფრთხილების, საურავის, ფულადი ჯარიმის, საქართველოს საბაჟო საზღვრის გადაკვეთის უფლების შეზღუდვის, სამართალდარღვევის საქონლის ან/და სატრანსპორტო საშუალების უსასყიდლოდ ჩამორთმევის სახით, ამ კოდექსით გათვალისწინებულ შემთხვევებში;

საგადასახადო სამართალდარღვევის ოქმის შედგენაზე უფლებამოსილი პირი სამართალდარღვევის ადგილზე განიხილავს საგადასახადო სამართალდარღვევის საქმეს და სამრთალდამრღვევ პირს ადგილზევე უფარდებს საგადასადო სანქციას. ამ შემთხვევაში პირს შესაბამისი პასუხისმგებლობა დაეკისრება საგადასახადო სამართალდარღვევის ოქმის საფუძველზე, რომელიც ითვლება საგადასახადო მოთხოვნად;

საგადასახადო დავა შესაძლებელია განხილულ იქნეს საქართველოს ფინანსთა სამინისტროს სისტემასა და სასამართლოში; საქართველოს ფინანსთა სამინისტროს სისტემაში საგადასახადო დავის ნებისმიერ ეტაპზე მომჩივანს უფლება აქვს, მიმართოს სასამართლოს; სასამართლოში საგადასახდო დავის წარმოების წესი განისაზღვრება საქართველოს ადმინისტრაციული საპროცესო კანონმდებლობით; ფინანსთა სამინისტროს სისტემაში საგადასახადო დავის განმხილველი ორგანოები არიან შემოსავლების სამსახური და საქართველოს ფინანსთა სამინისტროსთან არსებული დავების განხილვის საბჭო (შემდგომში დავის განმხილველი ორგანოები);

დავების განხილვის საბჭო არის ფინანსთა სამინისტროსთან არსებული დავების განმხილველი ორგანო;

საქართველოს ფინანსთა სამინისტროს სისტემაში საგადასახადო დავა ორეტაპიანია და იწყება საჩივრის შემოსავლების სამსახურში წარდგენით; საგადასახადო ორგანოს მიერ პირის მიმართ ამ კოდექსის საფუძველზე მიღებული გადაწყვეტილება შეიძლება გასაჩივრდეს დავის განმხილველ ორგანოში ამ თავით დადგენილი წესით;

საგადასახადო შემოწმების აქტი და მის საფუძველზე მიღებული გადაწყვეტილება საჩივრდება ამ დოკუმენტების საფუძველზე გამოცემულ საგადასახადო მოთხოვნასთან ერთად. საგადასახადო სამართალდარღვევის ოქმი/ბრძანება საჩივრდება ამ თავით დადგენილი წესით.

მაკონტროლებელი ან სამართალდამცავი ორგანოების მიერ საგადასახადო ორგანოსათვის მიწოდებული ინფორმაციის საფუძველზე გამოცემული საგადასახდო მოთხოვნა საჩივრდება დავების განხილვის საბჭოში;

ამ კოდექსის 273-ე და 281-ე მუხლებით გათვალისწინებულ სამართალ-დარღვევებთან დაკავშირებით გადასახდის გადამხდელის საჩივარზე შემოსავლების სამსახურის მიერ მიღებული გადაწყვეტილება საჩივრდება სასამართლოში;

პირს უფლება აქვს, გაასაჩივროს საგადასახადო ორგანოს გადაწყვეტილება მისი ჩაბარებიდან 30 დღის ვადაში;

თუ საჩივარი არ აკმაყოფილებს პროცედურულ მოთხოვნებს, მომჩივანს წერილობით ეცნობება ამის შესახებ და მიეცემა არანაკლებ 5 დღე საჩივარში არსებული ხარვეზის გამოსასწორებლად. დავის განმხილველ ორგანოს უფლება აქვს, მომჩივნის მოტივირებული მოთხოვნის შემთხვევაში გააგრძელოს ხარვეზის გამოსასწორებლად მიცემული ვადა;

დავის განმხილველი ორგანო უფლებამოსილია ხარვეზის არსებობის მიუხედავად მიიღოს საჩივარი წარმოებაში, თუ იგი არსებითად არ უშლის ხელს საჩივრის განხილვას;

დავის განმხილველი ორგანო საჩივარს განიხილავს 20 დღის ვადაში;

დავის განმხილველი ორგანო უფლებამოსილია საკუთარი ინიციატივით ან მხარის მოტივირებული შუამდგომლობით შეაჩეროს საჩივრის განხილვა დამატებითი ინფომაციის ან/და დოკუმენტაციის მოსაპოვებლად; საჩივრის განხილვის შეჩერებისას დავის განმხილველი ორგანო უფლებამოსილია დაავალოს მომჩივანს ან/და საგადასახადო ორგანოს, წარმოადგინოს დამატებითი ინფომაცია ან დოკუმენტაცია საჩივრის ფარგლებში განსახილველი საკითხების შესახებ; თუ დავის განმხილველი ორგანოს დავალების შესრულება დადგენილ ვადაში ვერ ხერხდება, ამის შესახებ გონივრულ ვადაში უნდა

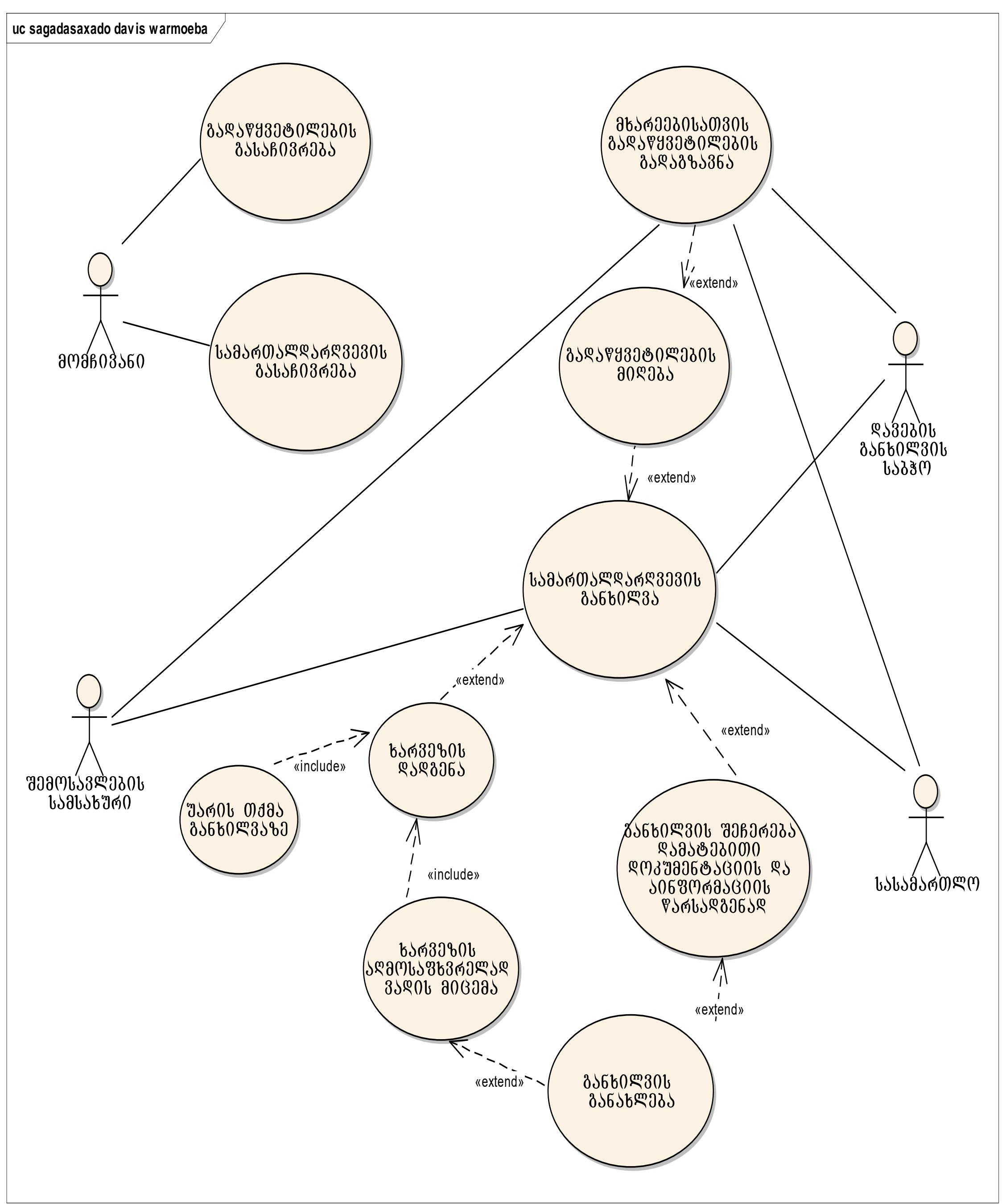
ეცნობოს დავის განმხილველ ორგანოს; დამატებითი ინფორმაციის ან/და დოკუმნტაციის მოპოვების საფუძვლით შეჩერების საერთო ხანგრძლივობა არ უნდა აღემატებოდეს 45 დღეს;

დავის განმხილველი ორგანოს გადაწყვეტილების დამოწმებული ასლი მისი მიღებიდან 5 სამუშაო დღის ვადაში ეგზავნება მხარეებს; შემოსავლების სამსახურის მიერ მომჩივნისთვის არასასურველი გადაწყვეტილების მიღების შემთხვევაში ამ მომჩივანს უფლება აქვს, გადაწყვეტილება, მისი ჩაბარებიდან 20 დღის ვადაში, გაასაჩივროს დავების განხილვის საბჭოში ან სასამართლოში; მომჩივანს უფლება აქვს, დავების განხილვის საბჭოს გადაწყვეტილება მისი ჩაბარებიდან 20 დღის ვადაში გაასაჩივროს სასამართლოში;

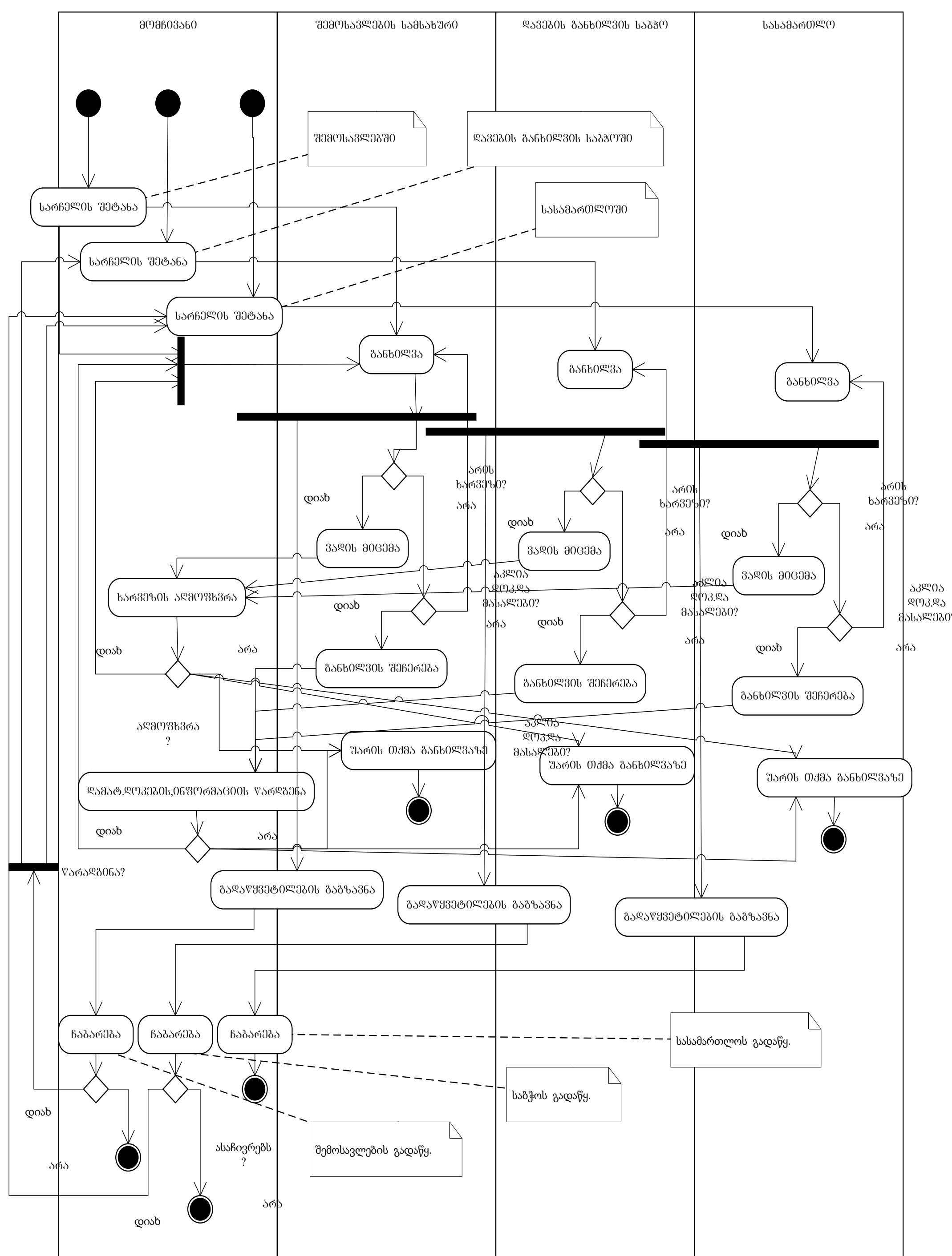
დავის განმხივლელი ორგანოს გადაწყვეტილება ძალაში შედის მომჩვნისათვის ჩაბარებიდან 21-ე დღეს, მისი გაუსაჩივრებლობის შემთხვევაში. სასამართლოს მიერ მიღებული გადაწყვეტილება საბოლოოა და გასაჩივრებას არ ექვემდებარება“.

ქვემოთ მოყვანილია პრეცედენტების (ნახ.5.5), აქტიურობათა (ნახ.5.6), მიმდევრობითობის (ნახ.5.7) და დროითი (ნახ.5.8) დიაგრამების ერთობლიობა UML/2 ენაზე, რომელიც შეესაბამება „საგადასახადო დავის წარმოების“ ზოგად მოდელს. მოდელირების ინსტრუმენტებად ვიყენებდით SparX-Enterprise Architect და Ms Visio პაკეტებს [23,31].

პრეცედენტების დიაგრამაზე ასახულია სისტემაში მონაწილე როლები და მათი ფუნქციები. აქტიურობის დიაგრამა (ნახ.5.6) ოთხი ზოლით (SwimLine: მომჩივანი, შემოსავლების სამსახური, დავების განხილვის საბჭო და სასამართლო) ასახავს ამ პროცესში მონაწილე თითოეული სუბიექტის ფუნქციონალური პროცედურების ერთობლიობას, დალაგებულს დროსა და სივრცეში, ანუ მიმდევრობით-პარალელურ პროცედურებს, მიზეზ-შედეგობრივი დამოკიდებულების თვალსაზრისით.



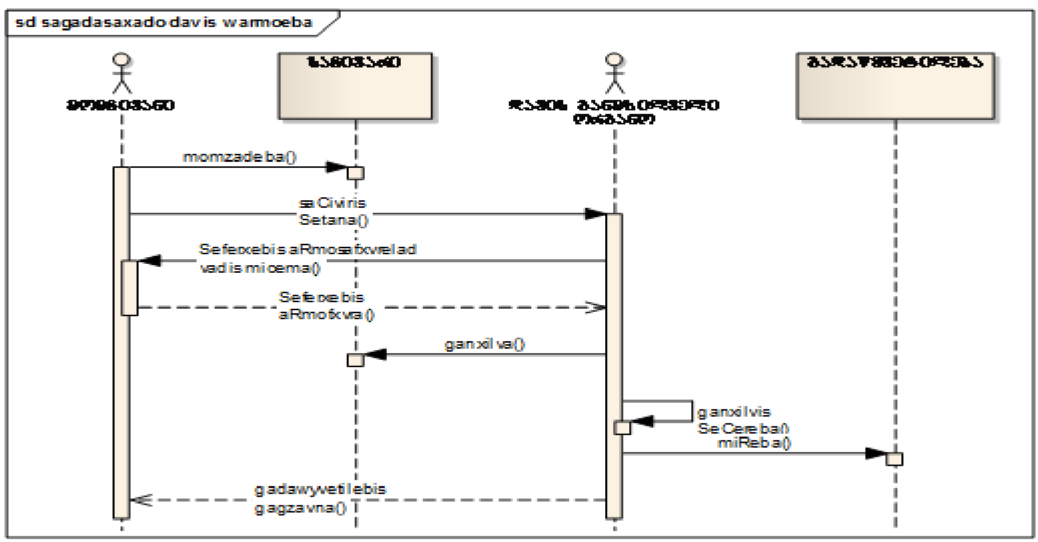
ნახ. 5.5. პრეცედენტების UseCase -დიაგრამა



ნახ. 5.6. Activity -დიაგრამა: „საგადასახადო დავის წარმოება“

5.1.5. საგადასახადო დავების წარმოების Sequence და Timing დიაგრამები

მიმდევრობითობის დიაგრამა ამოცანისათვის „საგადასახადო დავის წარმოება" (ნახ. 5.7) აღწერს იმ პროცესის სცენარს, რომლის მიხედვითაც დროში მიმდევრობითაა დალაგებული პროცესში მონაწილე როლების (მომჩივანი, დავის განმხილველი ორგანო) ფუნქციები.

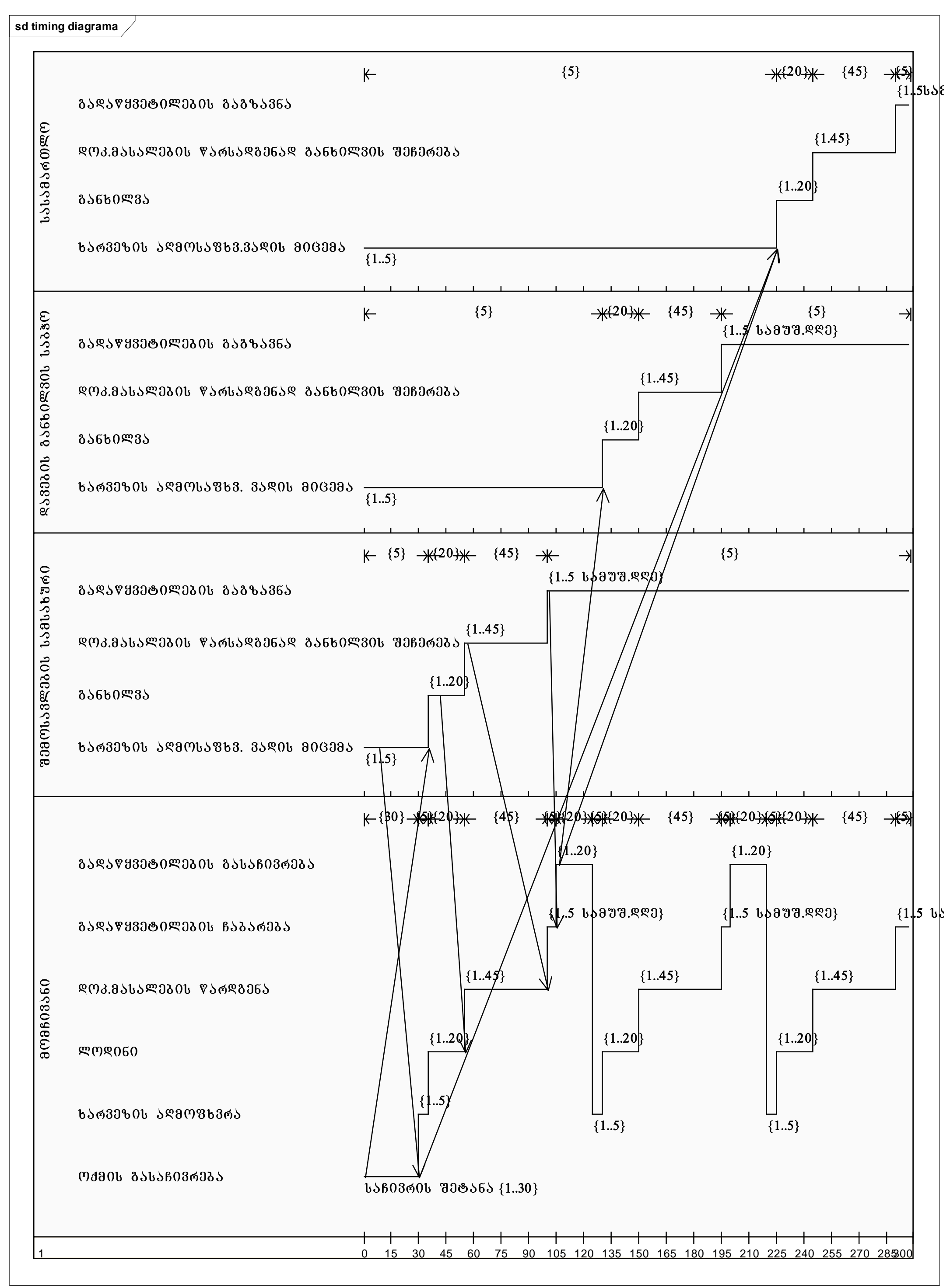


ნახ. 5.7. Sequence -დიაგრამა: „საგადასახადო დავის წარმოება"

განსაკუთრებით საყურადღებოა დროითი (Timing) დიაგრამა, რომელიც UML/2 ენის ერთ-ერთ მნიშვნელოვანი სიახლეა (არ იყო UML/1-ის წინა ვერსიებში), რომელიც გამოიყენება ერთი ან რამდენიმე ელემენტის მდგომარეობის დროში ცვლილების საჩვენებლად. იგი ასევე გვიჩვენებს მოვლენების ურთიერთმიმართებას დროსთან და განგძობითობის საზღვრებს.

5.8 ნახაზზე ნაჩვენებია Timing -დიაგრამა ჩვენი სისტემისათვის.

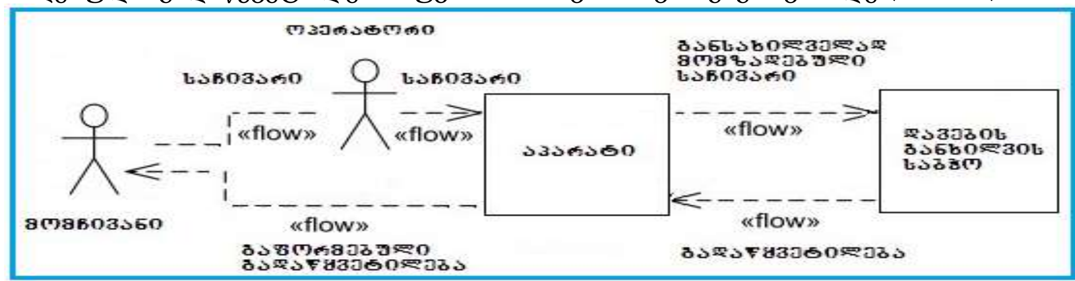
ჩატარებულია საგადასახადო სისტემაში საგადასახადო დავის წარმოების ბიზნეს-პროცესების ობიექტ-ორიენტირებული ანალიზი და შესაბამისი მოდელირება UML/2 ტექნოლოგიის საფუძველზე. ბიზნეს-პროცესების და ბიზნეს-წესების სრული დაცვით შემოთავაზებულია პრეცედენტების, აქტიურობათა, მიმდევრობითობის და დროითი დიაგრამები, რომელთა საფუძველზეც შინაარსობრივად უფრო ზუსტად განისაზღვრება დასაპროექ-ტებელი სისტემის (საგადასახადო დავის წარმოების) პროგრამული უზრუნველყოფის ფუნქციონალური მოთხოვნები.



ნახ. 5.8. Timing-დიაგრამა

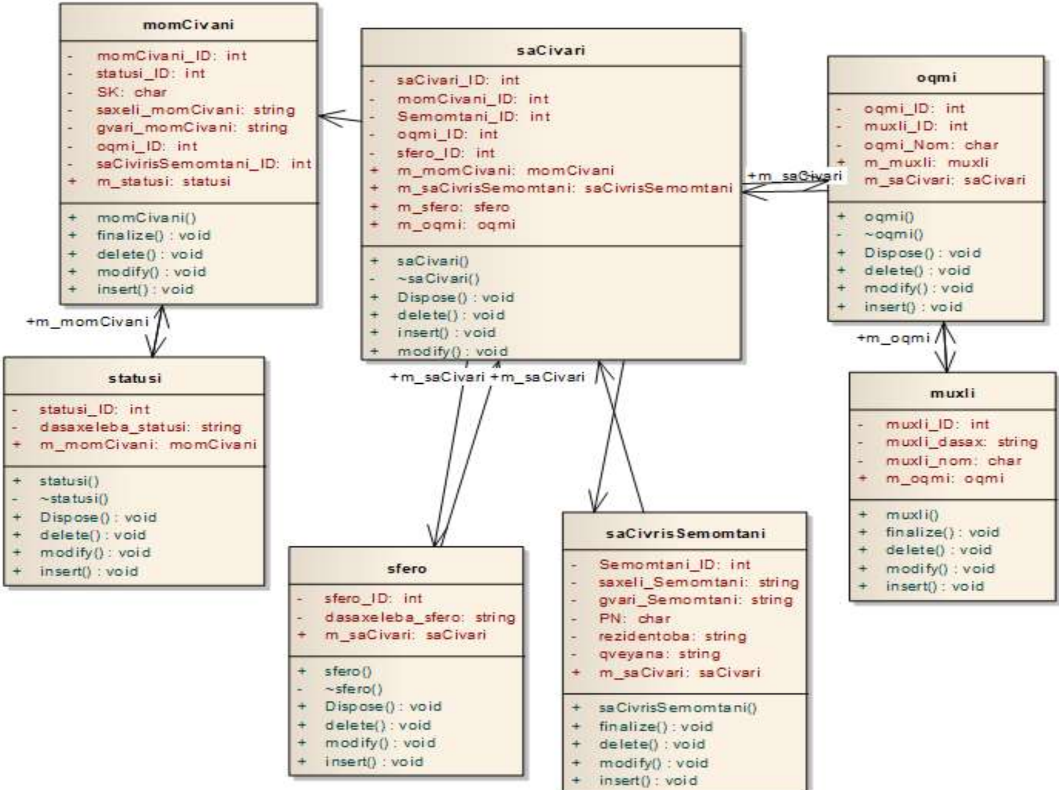
5.1.6. საგადასახადო დავების წარმოების Information_Flow და Class დიაგრამები

საგადასახადო დავის წარმოების პროცესის ინფორმაციული ნაკადის დიაგრამა ასახავს ინფორმაციის გაცვლას სისტემის ნაწილებს შორის, კერძოდ ოპერატორის მიერ მიღებული საჩივრის დავების განხილვის ორგანოსთან არსებული აპარატის გავლით, განმხილველი ორგანოსთვის გადაცემიდან, მიღებული გადაწყვეტილების უკან, მომჩივნისთვის გაგზავნამდე (ნახ. 5.9).



ნახ.5.9. Information flow-დიაგრამა

5.10 ნახაზზე მოცემულია ამ პროცესის კლასთა დიაგრამა.



ნახ.5.10. Class-დიაგრამა

5.2. საგადასახადო დავის წარმოების ბიზნესპროცესის მათემატიკური მოდელი რიგების თეორიის მიხედვით

საჩივრებთან დაკავშირებული საკითხები ცხადყოფს, რომ შემოსავლების სამსახური და მის დაქვემდებარებაში შემავალი თითქმის ყველა დეპარტამენტი და ქვედანაყოფი ფაქტობრივად მასობრივი მომსახურების სისტემებია. მასობრივი მომსახურების სისტემაა ფინანსთა სამინისტროს შემადგენლობაში შემავალი დავების განხილვის საბჭოც.

საინტერესოა მათემატიკური მოდელი ამ სისტემებისთვის, რომელიც პროგრამული უზრუნველყოფის შექმნაში დაგვეხმარება და რომელიც მომსახურების ეფექტურობის გაზრდისკენ და ხარჯების შემცირებისკენ იქნება მიმართული. საჩივრებთან დაკავშირებული ის პროცესები, რომლებიც წინა თავში არის მოცემული შეიძლება აღიწეროს, როგორც „მასობრივი მომსახურების სიტემა“ და საამისოდ გამოყენებულ იქნას რიგების თეორია, კერძოდ მარკოვის პროცესები, პუასონის განაწილება (საჩივრების შემოსვლის ინტენსივობა), ექსპონენციალური განაწილება (საჩივერების მომსახურების განაწილება) და ა.შ. [9,23,71,72].

5.2.1. რიგების სახეები მასობრივი მომსახურების სისტემებში

მასობრივ მომსახურების სისტემებში შემავალ და გამომავალ მოთხოვნათაA ნაკადის ინტენსივობის გამოსათვლელი მეთოდი არის ერთ-ერთი ძირითადი საკითხი. ინგლისელი მათემატიკოსის, დ. კენდალის ნოტაციის მიხედვით, რიგების თეორიის სტანდარტიზაციის და კლასიფიკაციის საკითხებზე, შეიძლება რიგების აღწერის გაფართოებული მოდელის ექვსეულით წარმოდგენა [23]:

$$< A/S/c/K/N/D >,$$ სადაც

A რიგში მოთხოვნების შემოსვლის სტატისტიკური განაწილებაა (M, თუ პროცესი მარკოვულია); S - რიგში მოთხოვნის მომსახურების განაწილება (M-მარკოვული ან ექსპონენციალური, E-ერლანგის განაწილება, G-საერთო განაწილება და ა.შ.); c – იდენტური მომსახურე ობიექტების რაოდენობა ($c \geq 1$); K-კლიენტების max-რაოდენობაა, რომელთა მომსახურება ხდება (თუ K-ზე მეტია, მაშინ კლიენტი არ იცდის რიგში. თუ K არაა მოცემული, მაშინ კლიენტების რაოდენობაზე არაა შეზღუდვა, უსასრულოა); N-კლიენტთა max-

რაოდენობა, რომელიც შეიძლება მოვიდეს სისტემაში (თუ არაა მოცემული, მაშინ ∞). D-მომსახურების დისციპლინა (FIFO, LIFO, SIRO (Servise In Random Order), PS (Priority service)) (23).

გამოკვლევების შედეგად დადგინდა, რომ მარკოვის პროცესები თამაშობს ფუნდამენტურ როლს მასობრივი მომსახურების სისტემების კვლევისას.

M/M/1 სისტემა. სისტემის სტაციონალურ რეჟიმში მუშაობის ერთ-ერთი მნიშვნელოვანი პირობაა, რომ შემავალი და გამომავალი ნაკადი იყოს თანაბარი, ამ პრინციპის გათვალისწინებით არსებობს კლასიკური M/M/1 სახის სისტემა, რომელსაც გამრავლების და გაქრობის სისტემას უწოდებენ. გამრავლებისა და გაქრობის პრცესს აქვს ერთი მეტად მნიშვნელოვანი თვისება: დროის ის შუალედი, რა მომენტშიც ხდება გამრავლება და დროის შუალედი, რა მომენტშიც ხდება გაქრობა (როდესაც სისტემა არ არის თავისუფალი და ვერ ღებულობს მოთხოვნებს) აღიწერება განაწილების მაჩვენებლიანი კანონით (ეს მიუთითებს იმას, რომ პროცესი არის მარკოვული). ეს პროცესი ზოგადად შეიძლება ჩამოვაყალიბოთ შემდეგნაირად: M/M/1 სახის სისტემა არის ისეთი სისტემა, სადაც დროის შუალედი მეზობელ მოთხოვნათა შორის განაწილე-ბულია მაჩვენებლიანი კანონით, აგრეთვე მომსახურების დროც განაწილე-ბულია მაჩვენებლიანი კანონით და სისტემა შეიცავს მხოლოდ ერთ მომსახურე მოწყობილობას.

საგადასახადო დავის წარმოების პროცესების განხილვისას დავინახავთ, რომ უკვე განსახილველად მიღებული საჩივრების რიგში დგომის პროცესი M/M/1 სისტემას განეკუთვნება, რადგანაც დავების განმხილველი ორგანო არის ერთი მომსახურე აპარატი (მიუხედავად იმისა, თუ რამდენი წევრისაგან შედგება იგი). ეს ბიზნეს-პროცესი აღვწერეთ Activity – დიაგრამის საშუალებით (ნახ.5.11).

ეს მიუთითებს, რომ გამრავლების ყველა ინტენსივობა λ არის მუდმივი და თანაბარი და აგრეთვე გაქრობის ყველა ინტენსივობაც μ არის მუდმივი და თანაბარი. ამ შემთხვევაში ორ მეზობელ მოთხოვნას შორის შუალედის საშუალო სიგრძე ტოლია: $\bar{t} = 1/\lambda$ და მომსახურების $\bar{x} = 1/\mu$ საშუალო დრო ტოლია

ეს განპირობებულია იმით, რომ ორივე შემთხვევითი სიდიდე $\bar{t}$ და $\bar{x}$ განაწილებულია მაჩვენებლიანი კანონით. აღსანიშნავია ის გარემოება, რომ

სისტემის მდგომარეობის სივრცე არის უსასრულო და რომ მოთხოვნათა მომსახურება ხორციელდება მათი დადგომის თანამიმდევრობით.

აპარატი

დავების განმხილველი ორგანო

დოკ-ების მომზადება

განმხილველი ორგანოსთვის წარდგენა

გახილვა

გადაწყვეტილების გამოტანა

აპარატისთვის გადაცემა

პასუხის გაგზავნა

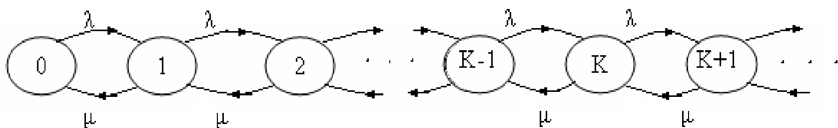
ნახ. 5.11. Activity – დიაგრამა. აპარატის მიერ მომზადებული საჩივრის განხილვა განმხილველი ორგანოს მიერ

თუ ავიღებთ კოეფიციენტებისთვის შემდეგ მნიშვნელობებს:

$$\lambda_k = \lambda, \text{ k=0,1,2, . . .}$$

$$\mu_k = \mu, \text{ k=1,2, 3, . . .}$$

გადასასვლელთა ინტენსივობის დიაგრამა მოცემულია M/M/1 ტიპისთვის (ნახ.5.12).



ნახ. 5.12. გადასასვლელთა ინტენსივობის დიაგრამა M/M/1 ტიპისთვის

ალბათობა იმისა, რომ სისტემაში დაყენებული ყველა მოთხოვნა დადგება მომსახურებაზე ტოლია [23]:

$$p_k = p_0 \prod_{i=0}^{k-1} \frac{\lambda}{\mu} \quad \text{ან} \quad p_k = p_0 \left(\frac{\lambda}{\mu} \right)^k, \qquad k \geq 0. \quad . \; . \; . \quad (5.1)$$

განხილული სისტემისათვის ერგოდიულობის პირობა მდგომარეობს იმაში, რომ $S_1 < \infty$ და $S_2 = \infty$; პირველი პირობა მოცემული შემთხვევისათვის ჩაიწერება შემდეგი სახით:

$$S_1 = \sum_{k=0}^{\infty} \frac{p_k}{p_0} = \sum_{k=0}^{\infty} \left(\frac{\lambda}{\mu} \right)^k < \infty.$$

ტოლობის მარცხენა მხარე სრულდება მხოლოდ მაშინ, როცა $\lambda / \mu < 1$. ერგოდიულობის მეორე პირობა ღებულობს შემდეგ სახეს:

$$S_2 = \sum_{k=0}^{\infty} \frac{1}{\lambda (p_k / p_0)} = \sum_{k=0}^{\infty} \frac{1}{\lambda} \left(\frac{\mu}{\lambda} \right)^k = \infty.$$

ეს უკანასკნელი პირობა სრულდება მაშინ როდესაც $\lambda / \mu < 1$. ე. ი. აუცილებელი და საკმარისი პირობა იმისა, რომ M/M/1 სისტემა არის ერგოდიული არის ის, რომ უნდა სრულდებოდეს უტოლობა: $\lambda < \mu$.

იმისათვის, რომ ვიპოვოთ p_0 უნდა გამოვიყენოთ ფორმულა:

$$p_0 = 1 / \left[1 + \sum_{k=1}^{\infty} \left(\frac{\lambda}{\mu} \right)^k \right].$$, რადგან $\lambda < \mu$ მოვიღებთ:

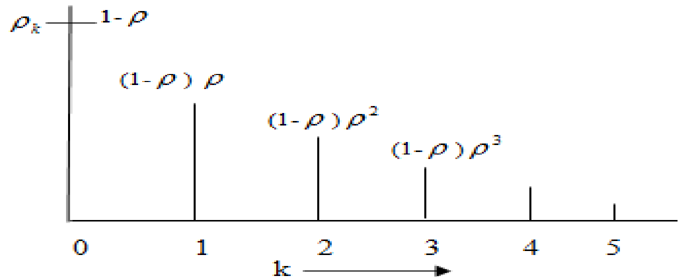
$$p_0 = \frac{1}{1 + \frac{\lambda / \mu}{1 - \lambda / \mu}} = 1 - \frac{\lambda}{\mu} \quad . \; . \; . \qquad (5.2)$$

როგორც ვიცით ρ=λ/μ. სტაციონალურობის პირობის თანახმად უნდა სრულდებოდეს უტოლობა: 0≤ρ<1; ეს პირობა გვაძლევს იმის გარანტიას, რომ შესრულდეს უტოლობა $p_0 > 0$; განტოლება (5.1) - ის თანახმად საბოლოოდ მივიღებთ:

$$p_k = (1 - \rho) \rho^k, k = 0,1,2,... \quad ... \qquad (5.3)$$

(5.3) ტოლობის თანახმად ალბათობა იმისა, რომ სისტემა შეიცავს k მოთხოვნას. მნიშვნელოვანია აღვნიშნოთ, რომ p_k ალბათობა განისაზღვრება λ და μ-s ρ -სთან დამოკიდებულებაზე.

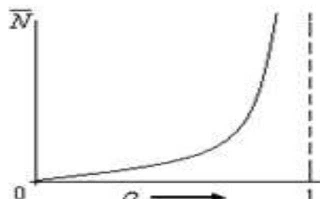
5.13 ნახაზზე ნაჩვენებია განხილული სისტემისათვის p_k ალბათობის მნიშვნელობა, იმ შემთხვევაში როცა $\rho = 1/2$. როგორც ნახაზიდან ჩანს ეს განაწილება არის გეომეტრიული.



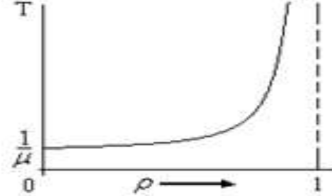
ნახ. 5.13. სტაციონალური p_k ალბათობა მმს-ის M/M/1 ტიპისთვის

M/M/1 სისტემა კვლევისას დავინახავთ, რომ სისტემაში არსებული თითქმის ყველა მნიშვნელოვანი განაწილების ალბათობა განეკუთვნება უშემდეგმოქმედო განაწლების ტიპს. მასობრივი მომსახურების სისტემებში მნიშვნელოვანია მოთხოვნათა საშუალო $\overline{N}$ რიცხვი. ეს სიდიდე მოცემულია ტოლობით:

$$\overline{N} = \sum_{k=0}^{\infty} k p_k = (1-\rho)\sum_{k=0}^{\infty} k\rho^k = \frac{\rho}{1-\rho} \tag{5.4}$$



ნახ. 5.14. M/M/1- ტიპია სისტემაში მოთხოვნათა საშუალო რიცხვი



ნახ. 5.15. M/M/1-ტიპის სისტემაში მოთხოვნათა დაყენების საშუალო დრო, როგორც ρ-ს ფუნქცია

სისტემაში მოთხოვნათა საშუალო რიცხვის გრაფიკი მოცემულია 5.14 ნახაზზე, ანალოგიური მეთოდით ვპოულობთ, რომ სისტემაში მოთხოვნათა რიცხვის დისპერსია ტოლია:

$$\sigma_N^2 = \sum_{k=0}^{\infty}(k-\overline{N})^2 p_k;$$

$$\sigma_N^2 = \frac{\rho}{(1-\rho)^2}. \quad . \; . \; . \tag{5.5}$$

(5.5) არის ჯონ ლიტლის ფორმულა [39], საიდანაც შესაძლებელია მივიღოთ სისტემაში მოთხოვნათა შემოსვლის საშუალო დრო:

$$\begin{aligned} T &= \frac{\overline{N}}{\lambda} \\ T &= \left(\frac{\rho}{1-\rho}\right)\left(\frac{1}{\lambda}\right); \quad . \; . \; . \\ T &= \frac{1/\mu}{1-\rho}. \end{aligned} \tag{5.6}$$

მოთხოვნათა სისტემაში შემოსვლის საშუალო დროის ρ-ზე დამოკიდე-ბულების გრაფიკი ნაჩვენებია 5.15 ნახაზზე. T-სიდიდე, რომელიც შეესაბამება წერტილს ρ=0, ტოლია მოთხოვნის მომსახურების საშუალო მნიშვნელობის; სხვა სიტყვებით, რომ ვთქვათ, ამ შემთხვევაში მოთხოვნა არ ელოდება რიგში და კმაყოფილდება საშუალოდ 1/μ-წამში.

როდესაც სისტემაში ρ მიისწრაფის ერთისკენ, როგორც მოთხოვნათა საშუალო რიცხვი, ასევე მოთხოვნათა შემოსვლის საშუალო დრო უსასრულოდ იზრდება.

ინტერესს მოკლებული არ არის გამოვთვალოთ ალბათობა იმისა, როდესაც სისტემა შეიცავს k-ზე ნაკლებ მოთხოვნებს:

$$P[\geq k \text{ **მოთხოვნა სისტემაში**}] = \sum_{i=k}^{\infty} p_i = \sum_{i=k}^{\infty}(1-\rho)\rho_1^i = \rho^k. \tag{5.7}$$

ამ თვალსაზრისით ალბათობა იმისა, რომ სისტემაში დაყენებული მოთხოვნათა რიცხვი გადააჭარბებს ზღვრულ მნიშვნელობას, აღიწერება გაქრობის გეომეტრიული პროგრესიით, რომელიც დამოკიდებულია ამ ზღვრულ რიცხვზე და მიისწრაფვის ნულისკენ.

M/M/m სისტემა: m მომსახურე მოწყობილობით. 1900-იანი წლების დასაწყისში დანიელი მეცნიერი აგნერ ერლანგი, როგორც მასობრივი

მომსახურების სისტემების ფუძემდებელი, განიხილავდა მას, როგორც სატელეფონო ქსელის მუშაობის ერთ-ერთ მოდელს.

იგი აღნიშნავდა, რომ სისტემაში შემოსული მოთხოვნები უნდა დადგეს რიგში და დაელოდოს მომსახურებას. ერლანგი განიხილავდა აგრეთვე სატელეფონო სისტემის ისეთ მოდელსაც, როგორიცაა მაგალითად, მმს-ის M/M/m ტიპი. ეს ისეთი შემთხვევაა, როდესაც არ ხდება ლოდინი. თუ სისტემაში შემოვა მოთხოვნა და ამ დროს სისტემის მომსახურე მოწყობილობა დაკავებულია, მაშინ მოთხოვნა იკარგება (სისტემა დანაკარგებით).

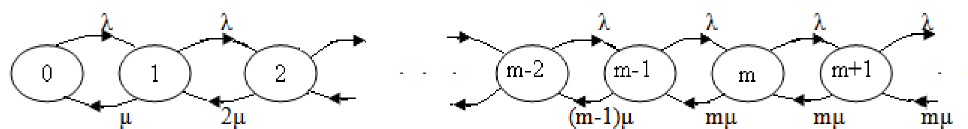
განვიხილოთ M/M/m სისტემა, რომელსაც აქვს მოთხოვნათა განუსაზღვრელი რაოდენობის მიღების საშუალება და დაყენებულ მოთხოვნათა მუდმივი ინტენსივობა. ჩავთვალოთ, რომ სისტემაში შეიძლება გამოყენებულ იქნას მაქსიმუმ m მომსახურე მოწყობილობა. ამ პირობის ფორმულირება შესაძლებელია გამრავლებისა და გაქრობის პროცესის დახმარებით:

$$\lambda_k = \lambda, \qquad k=0,1,2,\ldots;$$

$$\mu_k = \min[k\mu, m\mu] = \begin{Bmatrix} k\mu, 0 \le k \le m; \\ m\mu, m \le k. \end{Bmatrix} \quad \ldots \tag{5.8}$$

$$\frac{\lambda_k}{\mu_{k+1}} < C < 1 \ \ldots \tag{5.9}$$

5.9 უტოლობიდან გამომდინარე იოლად დავრწმუნდებით, რომ მოცემულ შემთხვევაში ერგოდიულობის პირობას აქვს შემდეგი სახე: $\lambda/\mu<1$; გადასასვლელების ინტენსივობის დიაგრამა ამ პროცესისთვის წარმოდგენილია 5.16 ნახაზზე.



ნახ. 5.16. mms-ისთვის გადასასვლელების ინტენსივობის დიაგრამა M/M/m ტიპი

მოცემული დიაგრამა ასახავს მომსახურე მოწყობილობასთან წარმოქმნილი რიგის შემთხვევაში მოთხოვნა როგორ გადადის უახლოეს მომსახურე ხელსაწყოში. ალბათობა იმისა, რომ სისტემაში შემოსული k მოთხოვნა დადგება თუ არა მომსახურებაზე, ტოლია:

$$P_k = P_0 \prod_{i=0}^{k=o} \frac{\lambda_i}{\mu_{i+1}}, \qquad k=0,1,2, \quad \ldots \tag{5.10}$$

ზემოთ მოყვანილი ტოლობის საშუალებით შეგვიძლია განვსაზღვროთ P_k ალბათობის მნიშვნელობა, ხოლო თუ მას გავყოფთ ორ ნაწილად, როგორც დამოკიდებულებას μ_k-სი k-ზე, სადაც შესაბამისად k≤m-ზე, მივიღებთ შემდეგ ტოლობას:

$$P_k = P_0 \prod_{i=0}^{k=o} \frac{\lambda}{(i+1)\mu} = P_0 \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!}. ,\ \ . \ . \ . \qquad (5.11)$$

ანალოგიურად მოვიღებთ, როცა k ≥ m,

$$P_k = P_0 \prod_{i=0}^{k=o} \frac{\lambda}{(i+1)\mu} \prod_{j=m}^{k-1} \frac{\lambda}{m\mu} = P_0 \left(\frac{\lambda}{\mu}\right)^k \frac{1}{m! m^{k-m}} , \dots \qquad (5.12)$$

თუ გავაერთიანებთ (5.11) და (5.12) ტოლობებს, მივიღებთ:

$$P_k = \begin{cases} P_0 \dfrac{(m\rho)^k}{k!}, k \le m; \\ P_0 \dfrac{(\rho)^k m^m}{m!}, k \ge m; \end{cases} \quad \text{sadac } \rho = \frac{\lambda}{m\mu} < 1. \ \dots \qquad (5.13)$$

ρ გამოსახავს დაკავებული მოწყობილობების ლოდინის კოეფიციენტს. λ არის შემოსული ნაკადის ინტენსივობა, ხოლო 1/μ მომსახურების საშუალო დრო. ე.ი. 5.13 განტოლება გამოსახავს სტაციონალურ ალბათობას იმისას, რომ სისტემაში შემოსული k მოთხოვნა დადგება თუ არა მომსახურებაზე.

P_0 - ის განსაზღვრისათვის დავწეროთ შემდეგი ტოლობა:

$$P_0 = \left[1 + \sum_{k=1}^{m-1} \frac{(m\rho)^k}{k!} + \sum_{k=m}^{\infty} \frac{(m\rho)^k}{m!} * \frac{1}{m^{(k-m)}}\right]^{-1},$$

მაშასადამე

$$P_0 = \left[1 + \sum_{k=1}^{m-1} \frac{(m\rho)^k}{k!} + \left(\frac{(m\rho)^k}{m!}\right)\left(\frac{1}{1-\rho}\right)\right]^{-1}. \qquad (5.14)$$

ალბათობა იმისა, რომ დაყენებული მოთხოვნა ჩადგება რიგში მოცემულია ტოლობით:

$$P(lodinisa) = \sum_{k=m}^{\infty} p_k = \sum_{k=m}^{\infty} p_0 \frac{(m\rho)^k}{m!} * \frac{1}{m^{k-m}}.$$

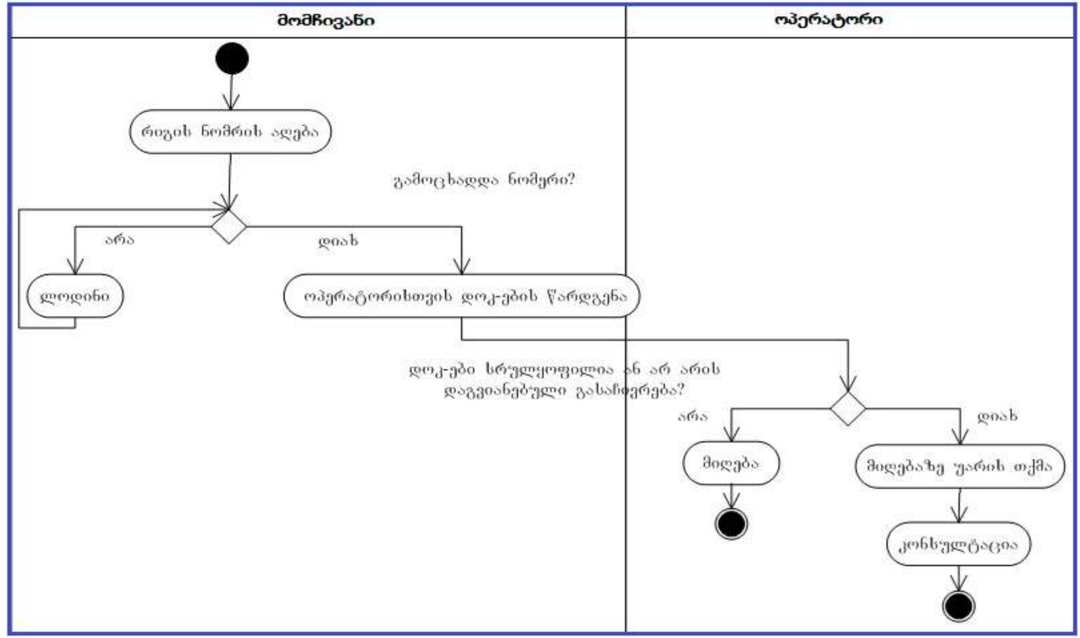
ამ თვალსაზრისით,

$$P(lodinisa) = \frac{\left(\frac{(m\rho)^m}{m!}\right) * \left(\frac{1}{1-\rho}\right)}{\left[\sum_{k=0}^{m-1} \frac{(m\rho)^k}{K!} + \left(\frac{(m\rho)}{m!}\right) * \left(\frac{1}{1-\rho}\right)\right]} \tag{5.15}$$

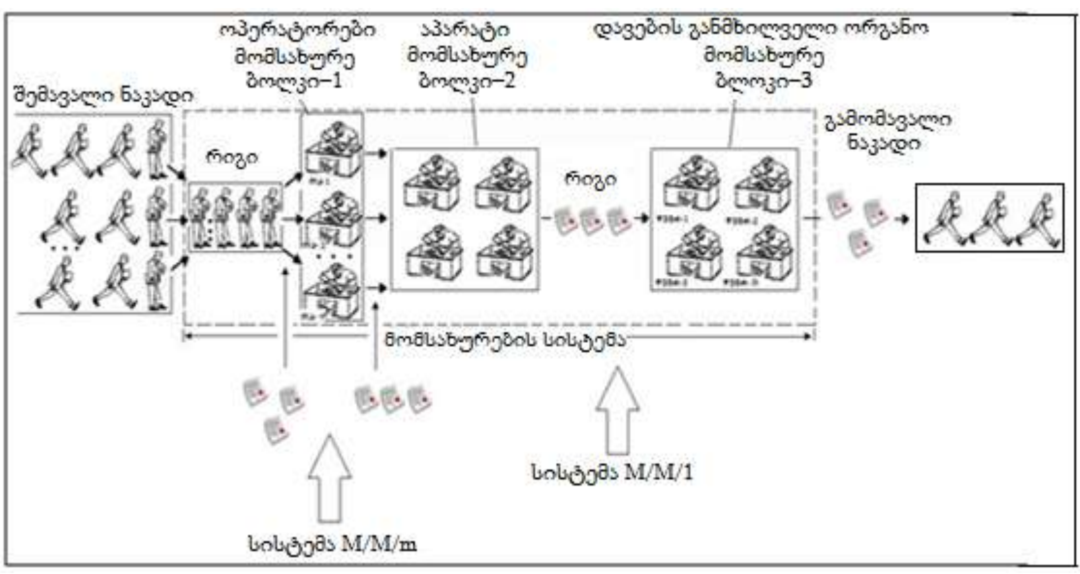
ეს ფორმულა ფართოდ გამოიყენება სატელეფონო სისტემებში; იგი განსაზღვრავს ალბათობას იმისას, რომ m ხაზის მიმართ დაყენებული მოთხოვნათა ნაკადი ვერ დაიკავებს ვერცერთ თავისუფალ ადგილს, ანუ მომსახურე მოწყობილობას, რის შედეგადაც მოხდება მოთხოვნის დაკარგვა.

ე.ი. ალბათობა იმისა რომ მოთხოვნა სისტემაში ვერ დაიკავებს ვერცერთ თავისუფალ ადგილს, გამოითვლება (5.15) ფორმულით. განხილულ ფორმულას უწოდებენ C-ერლანგის ფორმულას, ხოლო შესაბამისი ალბათობა აღინიშნება C(m, λ/μ). ევროპაში მას შემდეგნაირად აღნიშნავენ $E_{2,m}$ (λ/μ).

ჩვენი სისტემის შემთხვევაში M/M/m ტიპის სისტემას განეკუთვნება *მომჩივნის მიერ საჩივრის შეტანის პროცესი,* როდესაც ერთი ოპერატორი დაკავებულია და მას სხვა თავისუფალ ოპერტორთან მიმართვის საშუალება აქვს ან უწევს რიგის დაკავება. ეს პროცესი აღწერილია Activity დიაგრამის ნახ.5.17 და 18 ნახაზების საშუალებით.



ნახ.5.17. Activity – დიაგრამა: მომჩივნის მიერ
საჩივრის შეტანა



ნახ. 5.18. M/M/m და M/M/1 რიგის ტიპები: საჩივრის შეტანა და საჩივრის განხილვა

5.2.2. დავების განხილვის ბიზნესპროცესების მასობრივი მომსახურების მოდელი

ცოდნას რიგების შესახებ უწოდებენ რიგების ან მასობრივი მომსახურების თეორიას. რიგი არის ჩეულებრივი მოვლენა, რომელიც ყოველდღიური ცხოვრების თანამდევია. 5.1 ცხრილში მოცემულია რიგების რამდენიმე მაგალითი:

მასობრივი მომსახურების სისტემები **ცხრ. 5.1**

საპრობლემო სფერო	რიგში მომლოდინე ობიექტები	მომსახურების პროცესი
საგადასახადო დავების განმხილველი ორგანო (შემოსავლების სამსახური	საჩივარი	საჩივრის განხილვა
სუპერმარკეტი	მომხმარებლები	სალაროს აპარატთან მისვლა
ექიმთან ჩაწერა	პაციენტები	მიღება ექიმთან
კომპიუტერი	კომპიუტერული პროგრამები	პროცესორის მიერ პროგრამის შესრულება
სატელეფონო კომპანია	აბონენტები	განაცხადის შესრულება საერთაშორისო მომსახურებაზე

მიუხედავად იმისა, რომ ცხრილში მოყვანილი მაგალითები არსობრივად განსხვავდება ერთმანეთისაგან, მათ აქვთ ბევრი საერთო, პროცესების მართვისა და მომსახურების თვალსაზრისით. ასევეა "*საგადასახადო დავის განმხილველი ორგანოს"* შემთხვევაშიც. ყველა პროცესი იყენებს, როგორც ადამიანურ ასევე ტექნიკურ რესურსებს, კლიენტთა მოთხოვნილების დასაკმაყოფილებლად.

რიგების მოდელის პარამეტრები. მასობრივი მომსახურების სისტემის ანალიზისას გასათვალისწინებელია სისტემის ტექნიკური და ეკონომიკური ასპექტები. ტექნიკური ასპექტების ფაქტორები:

1) საშუალო დრო, რომელსაც კლიენტი (მომჩივანი) ატარებს რიგში;

2) რიგის საშუალო სიგრძე;

3) საშუალო დრო, რომელიც კლიენტებს (მომჩივანს) ესაჭიროებათ მომსახურებისათვის;

4) კლიენტთა (მომჩივანთა) საშუალო რაოდენობა;

5) ალბათობა იმისა, რომ მომსახურე სისტემა (ოპერატორები; დავების განხილვის ორგანო) არ აღმოჩნდება დაკავებული;

6) ალბათობა იმისა, რომ სისტემაში მოხვდება კლიენტთა (საჩივართა) განსაზღვრული რაოდენობა.

ეკონომიკური მაჩვენებლების გარდა საინტერესოა შემდეგი ფაქტორები:

1) რიგში ლოდინის ხარჯები;

2) სისტემაში ლოდინის ხარჯები;

3) ხარჯები მომსაურებაზე.

მასობრივი მომსახურების სისტემის მოდელები. არსებობს რამდენიმე სახის ***მმ*** მოდელი, მაგრამ აღსანიშნავია ის ფაქტი, რომ მათ ყველას აქვს საერთო მახასიათებლები:

ა) განაცხადის დაყენების ალბათობა განისაზღვრება პუასონის განაწილებით;

ბ) კლიენტთა ქცევა არის სტანდარტული წესებით;

გ) მომსახურების წესი ***FIFO*** (პირველი მოვიდა –პირველი წავიდა);

დ) მომსახურების ერთადერთი ფაზა.

5.2.2.1. ერთარხიანი სისტემის მოდელი M/M/1

მასობრივი მომსახურების ერთარხიანი სისტემის მოდელი **M/M/1,** სადაც შემავალი ნაკადი არის პუასონის განაწილებით, ხოლო მომსახურების დრო განაწილებულია ექსპონენციალურად.

იმისთვის, რომ სისტემა ჩავთვალოთ ერთარხიანად უნდა სრულდებოდეს შემდეგი პირობები:

1) სისტემაში შემავალი მოთხოვნა სრულდება პრინციპით: „პირველი მოვიდა, პირველი წავიდა“ *(FIFO)*, ყოველი კლიენტი ელოდება თავის რიგს მიუხედავად რიგის სიდიდისა;

2) მოთხოვნათა (საჩივარების) შემოსვლა რიგში დამოუკიდებული მოვლენებია, ოღონდაც დროის ერთეულში შემოსულ მოთხოვნათა (საჩივართა) საშუალო რიცხვი უცვლელია;

3) მოთხოვნის (საჩივარის) შემოსვლის პროცესი აღიწერება *პუასონის განაწილებით*, რომლის დროსაც მოთხოვნები შემოდის შეუზღუდავი სიმრავლიდან;

4) მომსახურების დრო (საჩივრის განხილვა) აღიწერება ალბათობათა ექსპონენციალური განაწილებით;

5) მომსახურების (საჩივრის განხილვის) სისწრაფე გაცილებით მაღალია, ვიდრე მოთხოვნის (საჩივართა) შემოსვლის სისწრაფე.

დავუშვათ, რომ:

- λ არის დროის ერთეულში შემოსულ მოთხოვნათა (საჩივარების) რაოდენობა;
- μ – კლიენტების (შემოსული საჩივრების) რაოდენობა, რომელთა მომსახურებაც ხდება დროის ერთეულში;
- n – სისტემაში (დავების განხილვის ორგანოში) შემოსულ მოთხოვნათა (საჩივართა) რაოდენობა.

ქვემოთ მოცემულია ფორმულები, რომელთა საშუალებითაც ხდება ***M/M/1*** სისტემის აღწერა:

$$L_s = \frac{\lambda}{\mu - \lambda}$$

- კლიენტთა (საჩივართა) საშუალო რიცხვი სისტემაში (დავების განხილვის ორგანოში);

$W_s = \frac{1}{\mu - \lambda}$ - სისტემაში (დავების განხილვის ორგანოში); მყოფი ერთი კლიენტის (საჩივირის) მომსახურების საშუალო დრო (ლოდინის დროს დამატებული მომსახურების დრო);

$L_q = \frac{\lambda^2}{\mu(\mu - \lambda)}$ - რიგში მყოფი კლიენტების (საჩივრების) საშუალო რიცხვი;

$W_q = \frac{\lambda}{\mu(\mu - \lambda)}$ - რიგში ლოდინის საშუალო დრო;

$r = \frac{\lambda}{\mu}$ - სისტემის დატვირთვა (დროის შუალედი, როდესაც სისტემა ასრულებს მოთხოვნის შესრულებას (დროის შუალედი, როდესაც ორგანო განიხილავს საჩივარს));

$P_0 = 1 - \frac{\lambda}{\mu}$ - სისტემაში მოთხოვნის (საჩივართა) არარსებობის ალბათობა;

$P_{n>k} = \left(\frac{\lambda}{\mu}\right)^{k+1}$ - ალბათობა იმისა, რომ სისტემაში არის k –ზე მეტი მოთხოვნა (განსახილველი საჩივარი).

5.2.2.2. მრავალარხიანი სისტემის მოდელი M/M/m

მრავალარხიან სისტემაში არის ორი ან მეტი მომსახურე არხი. კლიენტები (მომჩივნეები) დგანან ერთ საერთო რიგში და მიმართავენ თავისუფალ არხს (ოპერატორი).

საბანკო ქსელებში ფართოდ გამოიყენება მრავალარხიანი ერთფაზიანი სისტემები: კლიენტთა მომსახურე სექტორებში, სადაც წარმოიქმნება რიგები და კლიენტები მიმართავენ პირველივე თავისუფალ სექტორს.

მრავალარხიან სისტემებში მოთხოვნათა ნაკადი ექვემდებარება პუასონის კანონს, ხოლო მომსახურების დრო განაწილებულია ექსპონენციალურად.

მრავალარხიანი სისტემებისათვის, სადაც რიგი შეუზღუდავი რაოდენობით წარმოიქმნება უნდა სრულდებოდეს პირობა $\frac{r}{n} < 1$, სადაც r არის დატვირთვის პარამეტრი (დაკავებული არხის საშუალო რიცხვი), n კი - არხების მინიმალური რაოდენობა, სადაც რიგის გაზრდა არ ხდება უსასრულოდ.

M/M/m სისტემის აღწერისას გამოიყენება შემდეგი ფორმულები:

$P_0 = \left(1 + \frac{r}{1!} + \frac{r^2}{2!} + \ldots + \frac{r^n}{n!} + \frac{r^{n+1}}{n!(n-r)}\right)^{-1}$ - ალბათობა იმისა, რომ სისტემა (ოპერატორი) თავისუფალია;

$P_n = \frac{r^n}{n!} P_0$ - ალბათობა იმისა, რომ სისტემაში არის n მოთხოვნა (მომჩივანი ან სხვა პირი);

$P_q = \frac{r^{n+1}}{n!(n-r)} P_0$ - ალბათობა იმისა, რომ განაცხადი (მისაღები საჩივარი) არმოჩნდება რიგში;

$r = \frac{\lambda}{\mu}$ - დაკავებული არხების (ოპერატორების) საშუალო რიცხვი;

$L_q = \frac{r^{n+1} P_0}{n \cdot n! \left(1 - \frac{r}{n}\right)^2}$ - რიგში განაცხადის (მისაღები საჩივრების) საშუალო რაოდენობა;

$L_s = L_q + r$ - სისტემაში განაცხადის (მისაღები საჩივრების) საშუალო რაოდენობა;

$W_q = \frac{1}{\lambda} L_q$ - განაცხადის (მისაღები საჩივრების) რიგში ყოფნის დრო;

$W_s = \frac{1}{\lambda} L_s$ - სისტემაში განაცხადის ყოფნის (ოპერატორის მომსახურების) დრო.

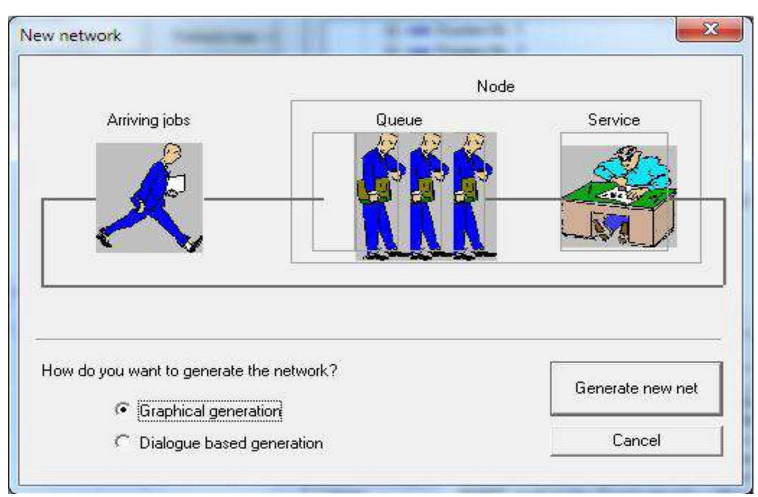
5.4. ორგანიზაციული სისტემის ქსელების მათემატიკური მოდელების გამოკვლევა WinPepsy პროგრამული პაკეტის ბაზაზე

ორგანიზაციული (კორპორაციული) სისტემის კომპიუტერული ქსელი შედგება ცალკეული კვანძების, რიგების და მომსახურე ობიექტებისგან. ყოველ კვანძს გააჩნია გარკვეული სტრატეგია, რომელიც აწესრიგებს თუ როგორი სახით უნდა დადგეს მოთხოვნა რიგში და როგორ უნდა მოხდეს მისი რედაქტირება [23]. მოცემული მოთხოვნის დროებითი შეყოვნებისას, მომსახურე ობიექტი ყოველ კვანძში აღწევს მომსახურების დროის განაწილებას სპეციალური პარამეტრებით. ქსელში, სადაც განთავსებულია სხვადასხვა სახის შეკვეთები, ერთიანდება დავალებათა კლასში, რომელიც იყოფა ორ კლასად: ღია და ჩაკეტილი (მესამე კლასი ჰიბრიდულია).

ღია კლასის ქსელური გრაფი შეიცავს საწყის და სასრულ მოთხოვნათა წყაროს. ამ შემთხვევაში მოთხოვნათა წყარო შეიძლება ვარეგულიროთ. ჩაკეტილი კლასის ქსელური გრაფი კი ვერცერთ ახალ მოთხოვნას ვერ მიიღებს ქსელში და აგრეთვე ვერცერთი მოთხოვნა ვერ დატოვებს რიგს. ქსელი მუდმივად ინარჩუნებს თანაბარ მოთხოვნათა რაოდენობას.

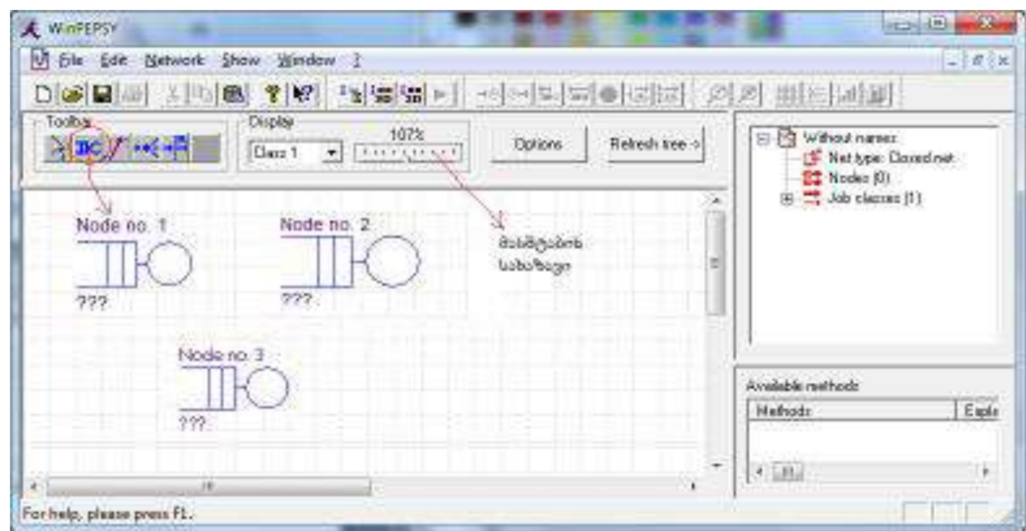
ახალი ქსელის შესაქმნელად WinPepsy რედაქტორში მთავარი მენიუდან (ნახ.5.19) ავირჩევთ File->New და მივიღებთ 5.20 ნახაზზე მოცემულ სქემას.

ნახ. 5.19. WinPepsy რედაქტორის ინსტრუმენტების პანელი



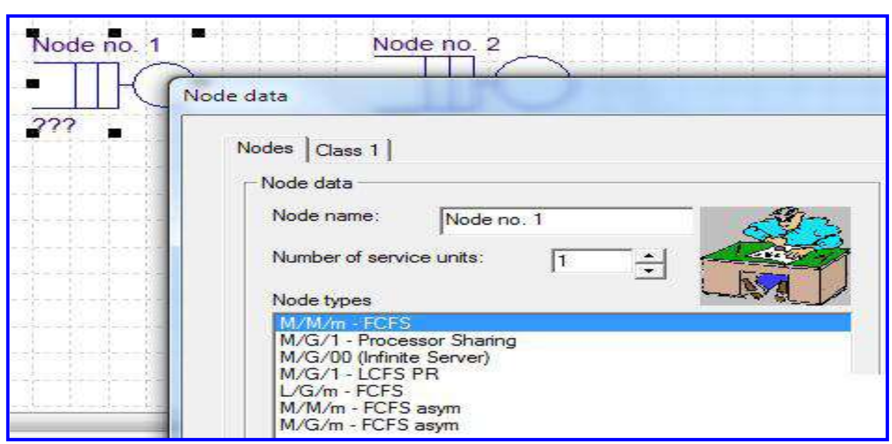
ნახ. 5.20

„გრაფიკული გენერაციის“ არჩევით ეკრანზე გამოვა 5.21 ნახაზზე ნაჩვენები გრაფიკული რედაქტორის ფანჯარა. ვირჩევთ მენიუდან მითითებულ სიმბოლოს („კვანძი“) და გამოგვაქვს მუშა არეში (სამი კვანძი: **no** 1-3). მასშტაბის სახაზავით შეგვიძლია ვცვალოთ სქემის ზომები.



ნახ. 5.21. ქსელის აგების პროცესის ფრაგმენტი

კვანძის ტიპი მაუსის მარჯვენა ღილაკით აირჩევა (ნახ.5.22).



ნახ. 5.22. კვანძის ტიპის (M/M/m) არჩევა

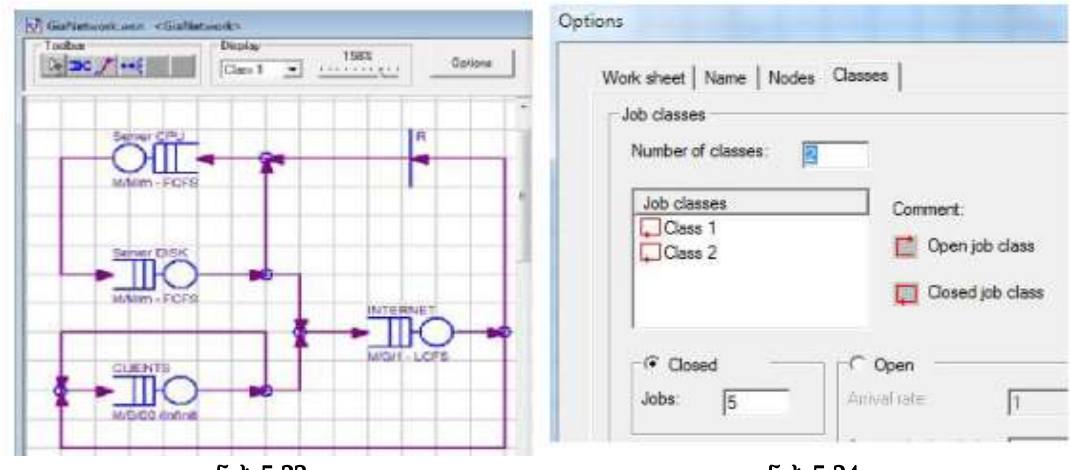
ქსელის გრაფიკულ რედაქტორს აქვს კომფორტული ინტერფეისი, რაც ხელსაყრელს ხდის მონაცემთა ინტერაქტიულ გრაფიკულ შეტანას. ეს პროცესი არსებითად მნიშვნელოვანია იმ მომხმარებელთათვის, რომელთაც აქვთ მხოლოდ მიახლოებითი წარმოდგენა გასაანალიზებელ ქსელზე და სურვილი აქვთ მასზე გარკვეული მანიპულირება მოახდინონ თვალსაჩინოდ. ყველაზე დიდი დადებითი მხარე, რომელიც გააჩნია ასეთ ქსელურ გრაფს არის ის, რომ იგი შედგება დუბლირებული საბაზო ქსელისგან, სადაც შესაძლებელია

ქვექსელებიდან მოხდეს მონაცემთა გადაცემა, წაშლა ან შეცვლა. ამის შედეგად ციკლური ქსელური სტრუქტურა დიდი დანახარჯების გარეშე შეძლებს კვანძების გადასასვლელები, ერთი კლასიდან მეორეში უკავშირდებოდეს ერთმანეთს და მონაწილეობა მიიღოს ქსელის საერთო მუშაობაში.

განვიხილოთ ჩაკეტილი, ღია და ჰიბრიდული ქსელების მოდელირების და ანალიზის ექსპერიმენტული ამოცანები და მათი გადაწყვეტის პროცედურები WinPepsy ინსტრუმენტული პაკეტის გამოყენებით [23].

5.4.1. „კლიენტ-სერვერ“ ჩაკეტილი ქსელის მოდელირება და ანალიზი

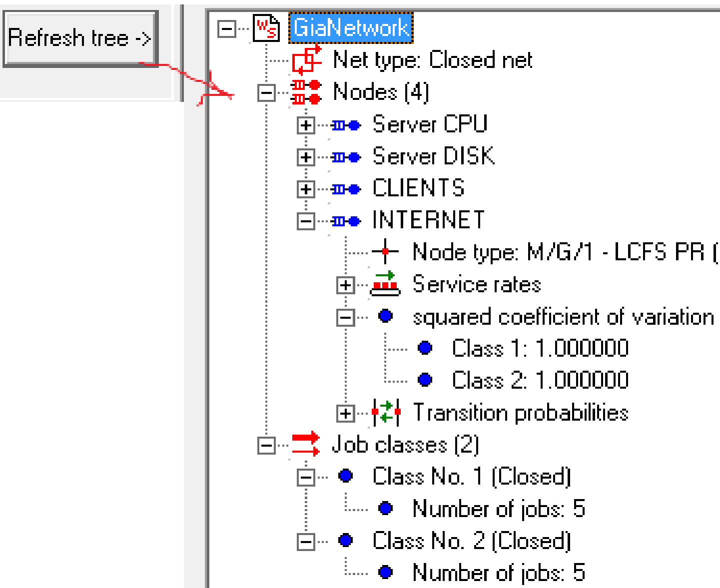
ავაგოთ მარტივი ჩაკეტილი ქსელი კლიენტ-სერვერ არქიტექტურის მაგალითისთვის, რომელთა შორის კავშირი ინტერნეტით ხორციელდება (ნახ.5.23). ღილაკით “**Options**” გამოიტანება დიალოგური ფანჯარა (ნახ.5.24), სადაც “**Classes**” გვერდზე შევცვლით “**Number of classes**” 2-ით და “**Jobs**” 5-ით.



ნახ.5.23 ნახ.5.24

კვანძზე დაკლიკვით შევალთ ფანჯარაში, სადაც “Nodes“ გვერდზე შეიძლება ქსელის კვანძებზე სახელების და ტიპების შერჩევა (მაგალითად, INTERNET, M-G-1). კვანძებისთვის სერვისის ტარიფები (service rates) ასე Server CPU-6/6, Server disk-14/14, Internet-16/18, Clients-10/20 (Class1/Class2-თვის).

კვანძთაშორის გადასასვლელებზე (ისრებზე) ალბათობათა მნიშვნელობები ჩვენი შემთხვევისთვის ასე გადავანაწილეთ: Server CPU-დან Server DISK-კენ ისარზე ავიღეთ 1, ხოლო ყველა დანარჩენზე 0.5; ბოლოს ავამოქმედოთ ღილაკი “Refresh tree->”, მივიღებთ 5.25 ნახაზს.



ნახ.5.25

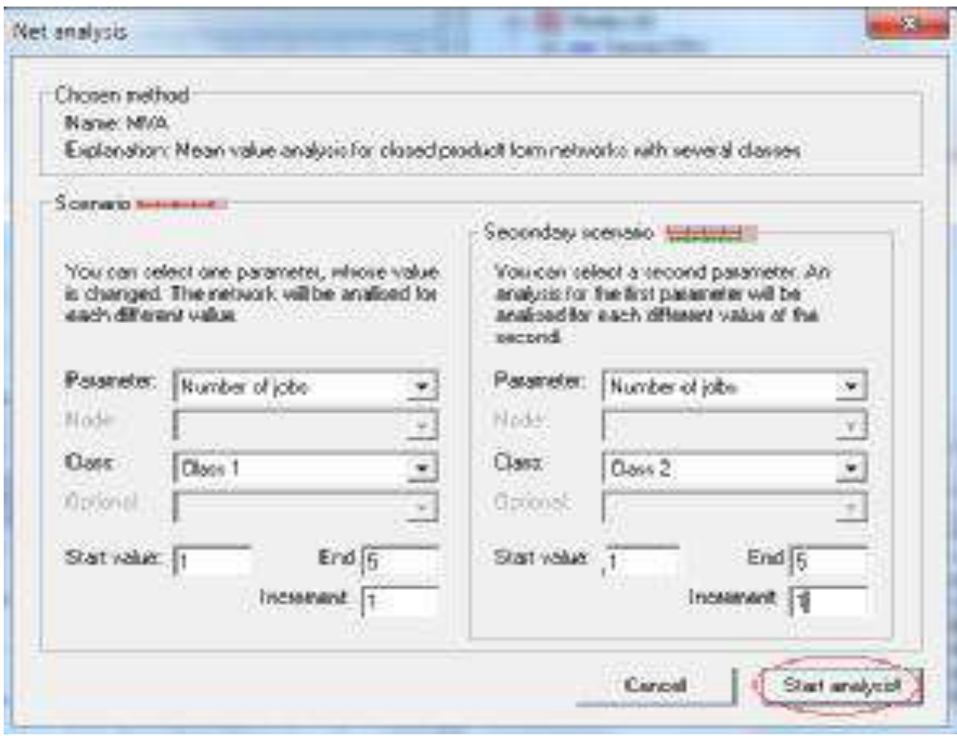
აგებული ჩაკეტილი ქსელის ანალიზისთვის ფანჯრის მარჯვენა ქვედა ნაწილში (**Available methods**) ვირჩევთ **MVA**–მეთოდს (ნახ.5.26).

Available methods

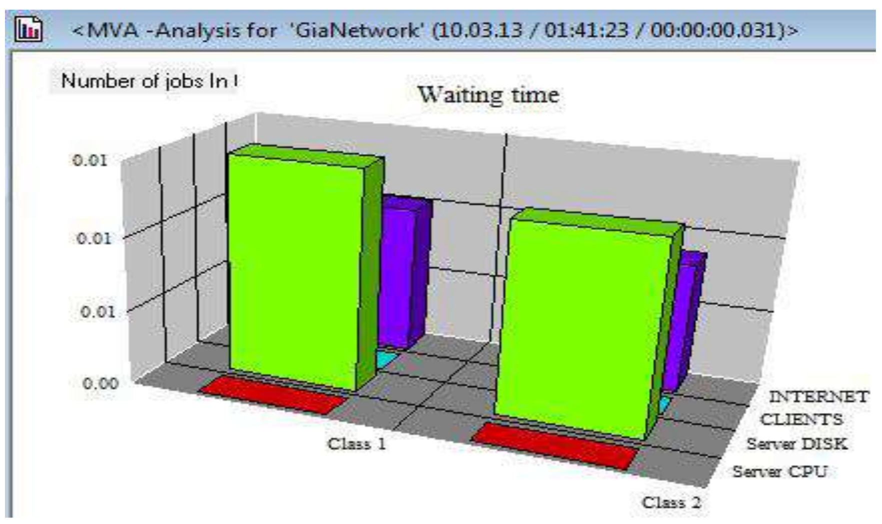
Methods	Explanation
MVA	Mean value analysis for closed product form networks with several classes
BIPHASE	BIPHASE analysis for closed networks without classes
Simulation	Simulation for mixed networks with classes and general service time distributi
OPFN analysis	OPFN analysis for open networks with classes and single server nodes
SOPFN analysis	SOPFN analysis for open networks without classes and with multi server nod
Marie	MARIE analysis for closed networks without classes with general service time
DECOMP	Decomposition analysis for open networks with classes and general service t
STATESP	Statespace analysis for closed networks with classes (not implemented yet: n

ნახ. 5.26. მეთოდის არჩევა

დიალოგურ ფანჯარაში ავირჩევთ პარამეტრებს (ნახ.5.27) და **Start**-ღილაკით მივიღებთ ანალიზის შედეგს (ნახ.5.28).

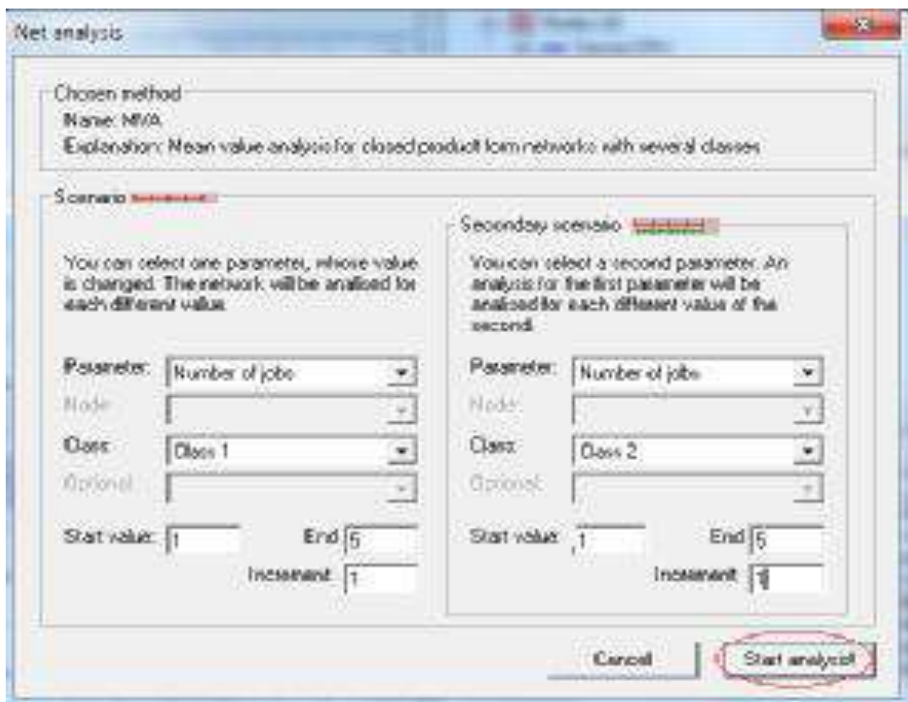


ნახ.5.27. პარამეტრების შეცვლა

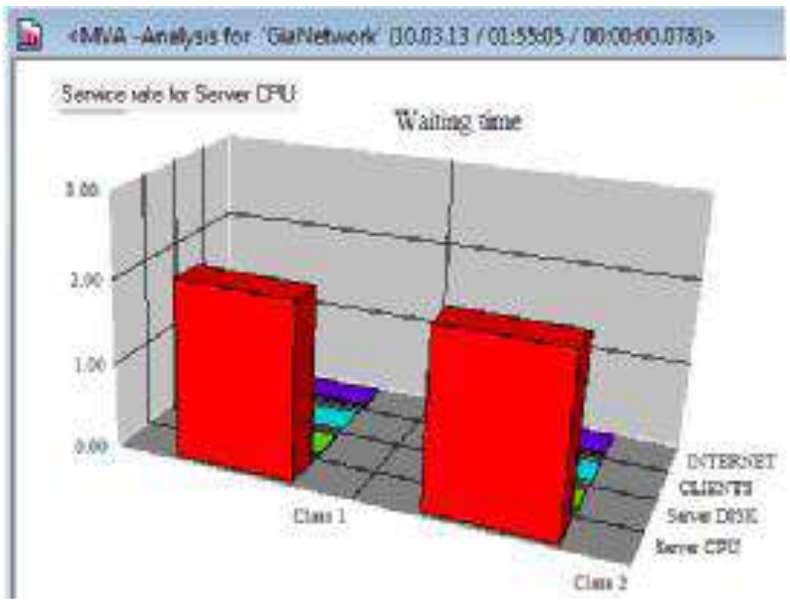


ნახ.5.28. ანალიზის შედეგები

ამის შემდეგ შეიძლება ექსპერიმენტის გაგრძელება სხვადასხვა მახასიათებლის ანალიზის მისაღებად. მაგალითად, მომსახურების ინტენსიუ-რობის პარამეტრის არჩევით (ნახ.5.29, 5.30).



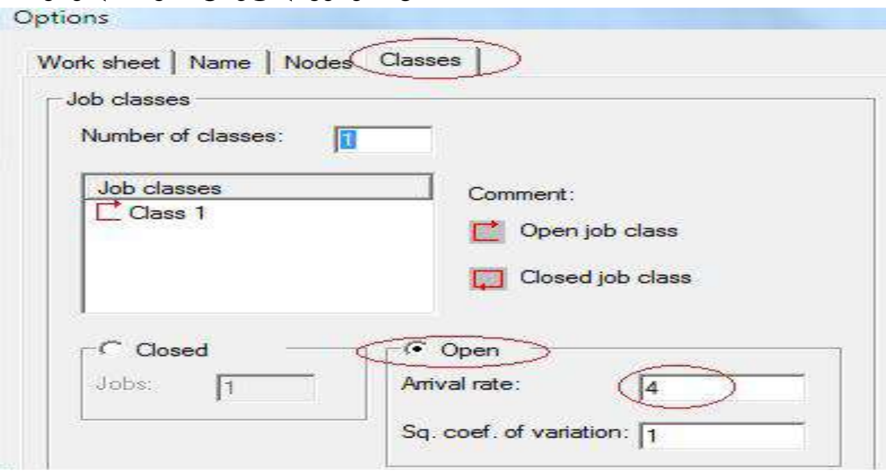
ნახ.5.29. Service Rate პარამეტრის არჩევა Class1/Class2-ისთვის



ნახ.5.30. ანალიზის შედეგები

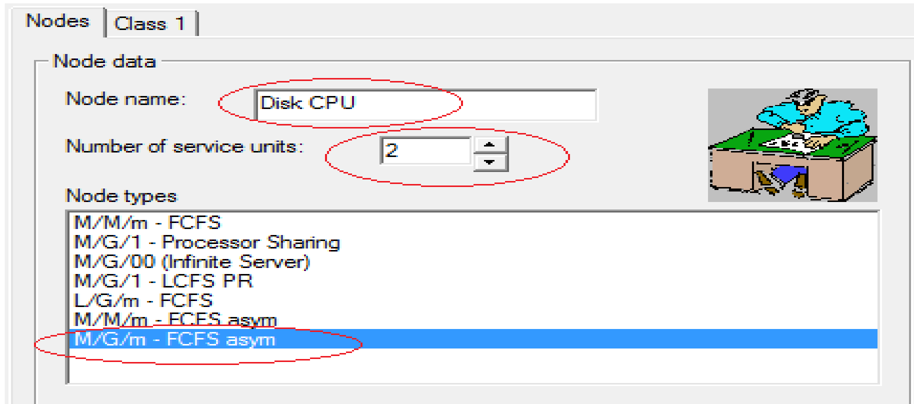
5.4.2. „კლიენტ-სერვერ" ღია ქსელის მოდელირება და ანალიზი

ახლა ავაგოთ და გამოვიკვლიოთ ღია ქსელის შემთხვევა **WinPepsy** –ის გარემოში ქსელის გრაფიკულ გენერაციით (ნახ.5.31).



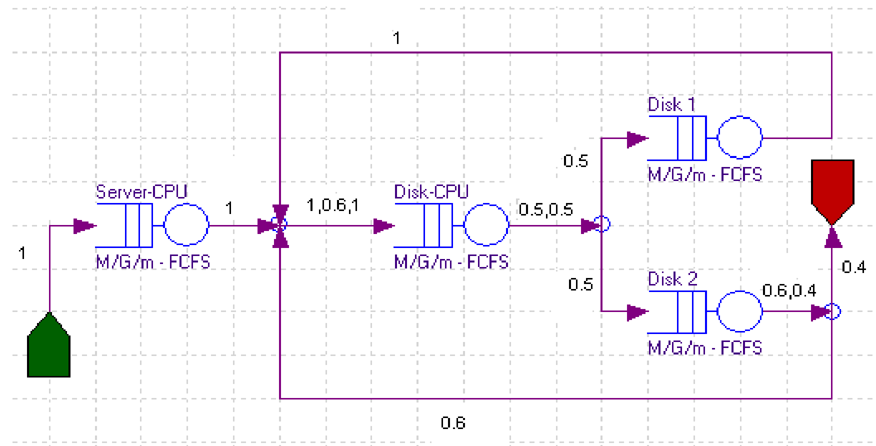
ნახ.5.31. Open ქსელის და Arival rate=4 არჩევა

ქსელის ოთხკვანძიან სქემას დაემატა კვანძთაშორისი გადასასვლელები შესაბამისი ალბათობებით, აგრეთვე ორი ელემენტი: ქსელში შესვლის წერტილი (ხუტთკუთხედი მარცხენა ქვედა ნაწილში) და ქსელიდან გამოსვლის წერტილი (მარჯვნივ ხუთკუთხედი) (ნახ.5.32, 33).



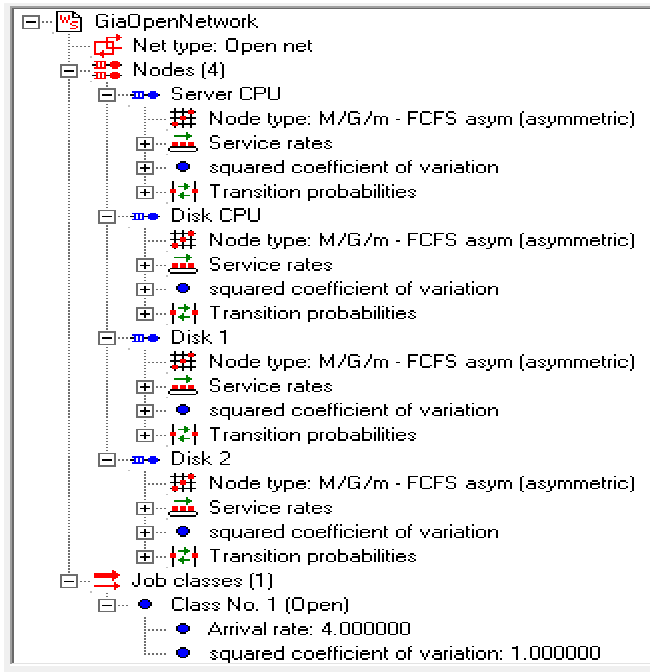
ნახ.5.32. Disk CPU კვანძის პარამეტრების შერჩევა

ეს სიმბოლოები აუცილებელია ღია ქსელებისთვის. შედეგი ასახულია 5.33 ნახაზზე.



ნახ.5.33. ღია ქსელის სქემა გადასასვლელებით და ალბათობებით

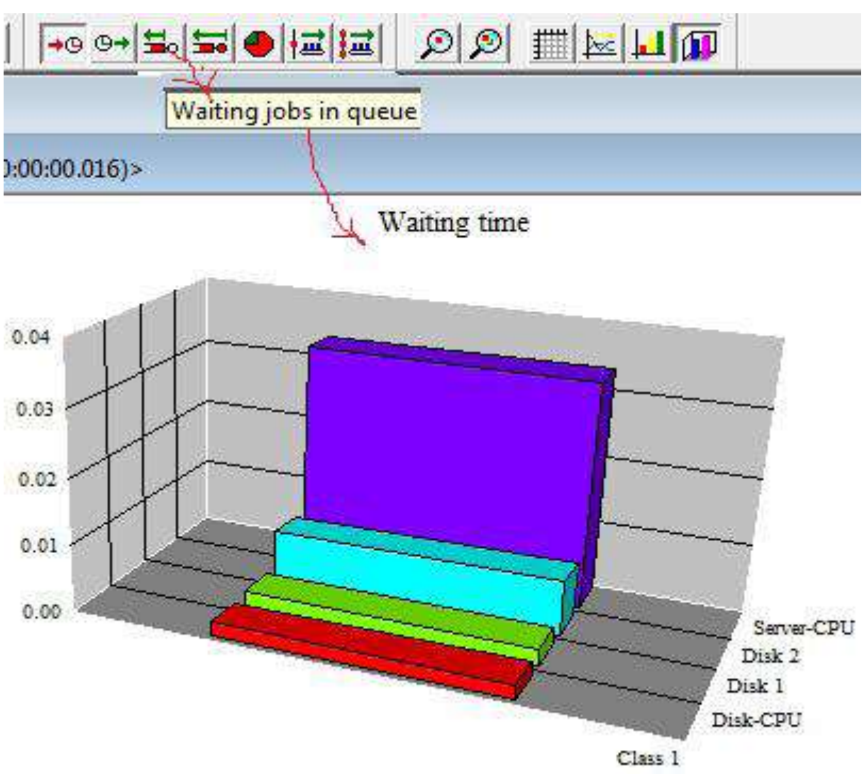
Refresh tree-> ღილაკით მივიღებთ მოდიფიცირებულ ხეს (ნახ.5.34). ახლა მოდელი მზად არის ანალიზის ჩასატარებლად. ეკრანის ქვედა მარჯვენა კუთხეში მოცემულია მეთოდები, რომლებიც გამოიყენება ამისათვის, კერძოდ ღია ქსელისთვის DECOMP-მეთოდი.



ნახ. 5.34

Start analysis ღილაკის ამოქმედების შედეგი მოცემულია 5.35 ნახაზზე.

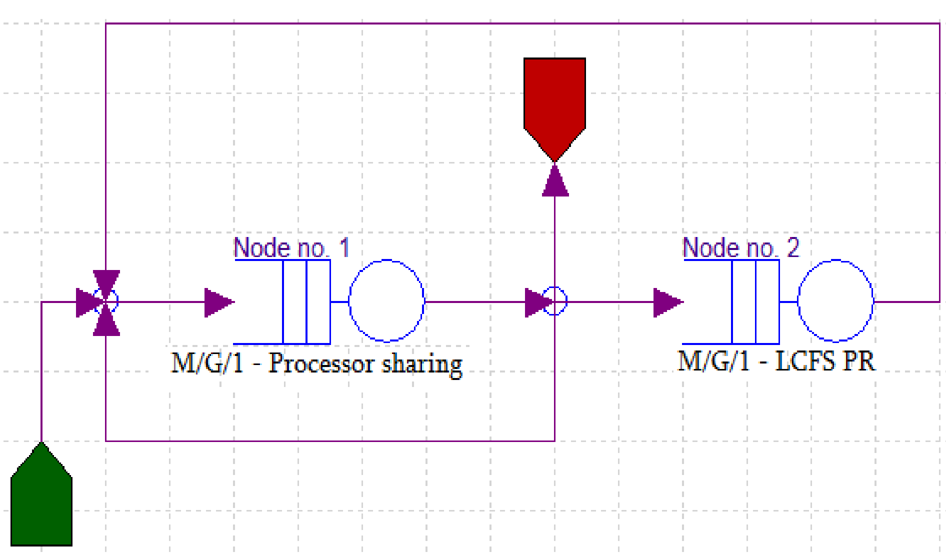
სხვა მახასიათებლების გასაანალიზებლად საჭიროა გამოვიყენოთ პანელის ღილაკები და მენიუში show-პუნქტი.



ნახ.5.35. მოთხოვნათა ლოდინის დრო რიგში

5.4.3. ჰიბრიდული ქსელის მოდელირება და ანალიზი

ჰიბრიდული ანუ შერეული ტიპის ქსელი ისეთი ქსელია, რომელსაც აუცილებლად აქვს მინიმუმ ერთი ღია კლასი და ერთიც ჩაკეტილი კლასი (ანუ ორივე სახეზეა). ქსელის აგების თვალსაზრისით **WinPepsy** რედაქტორში ღია კლასი აიგება ღია ქსელის კლასის წესებით, ხოლო ჩაკეტილი კლასი – ჩაკეტილი ქსელის კლასის წესებით. განვიხილოთ კონკრეტული მაგალითი (ნახ.5.36).



ნახ. 5.36. ჰიბრიდული ქსელის მოდელი

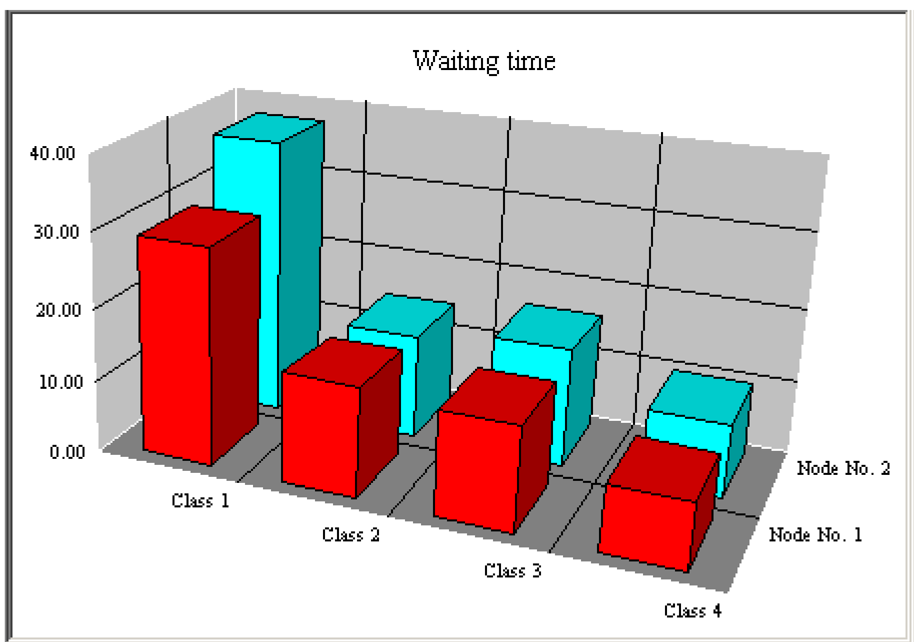
ამ შემთხვევაში ვირჩევთ კვანძის ტიპებს და პარამეტრებს. ვღებულობთ ხის სტრუქტურას და ბოლოს შევარჩევთ გასაანალიზებელ მეთოდს, ამჯერად სახელით Simulation (ნახ.5.37).

Available methods

Methods	Explanation
MVA	Mean value analysis for closed product form networks with several classes
BIPHASE	BIPHASE analysis for closed networks without classes
Simulation	Simulation for mixed networks with classes and general service time distributions
OPFN analysis	OPFN analysis for open networks with classes and single server nodes
SOPFN analysis	SOPFN analysis for open networks without classes and with multi server nodes
Marie	MARIE analysis for closed networks without classes with general service time dist
DECOMP	Decomposition analysis for open networks with classes and general service time (
STATESP	Statespace analysis for closed networks with classes (not implemented yet: mixed

ნახ. 5.37. ჰიბრიდული ქსელის ანალიზის მეთოდი

ანალიზის ერთი შედეგი 2-კვანძისა და 4-კლასისთვის დაყოვნების დროის მიხედვით მოცემულია 5.38 ნახაზზე.



ნახ.5.38. ქსელის ანალიზის შედეგების ფრაგმენტი

რიგების თეორიის და მისი ინსტრუმენტული საშუალებების გამოყენება საინჟინრო ამოცანების, განსაკუთრებით მასობრივი მომსახურების სისტემების მოდელირებისა და ანალიზისათვის მეტად მნიშვნელოვანია.

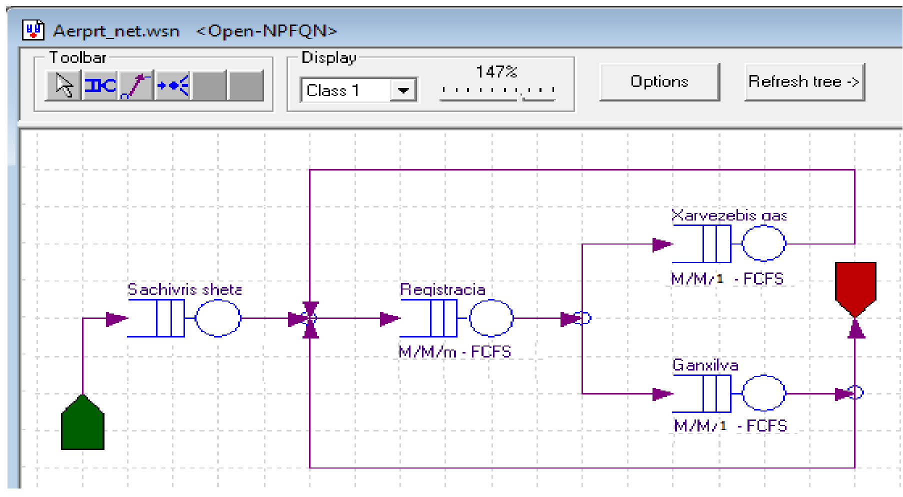
განაწილებულ სისტემებში, კერძოდ კომპიუტერულ ქსელებში რაოდენობრივი ანალიზის ჩასატარებლად, სასურველია სხვადასხვა ტიპის მათემატიკური მოდელების, მაგალითად, **M/M/1**, **M/M/m**, **M/G/1**, **G/M/1**, **G/G/1** და სხვა გამოკვლევა [23].

ჩვენთვის განსაკუთრებით საყურადღებოა M/M/m ტიპის მოდელის ანალიზი, რაც ნიშნავს, რომ შემავალი ნაკადი უმარტივესია (მარკოვული), მომსახურების დრო კი ექსპონენტური კანონით განაწილებული შემთხვევითი სიდიდეა (მარკოვული).

WinPepsy ინსტრუმენტული პაკეტი, საშუალებას იძლევა ავაგოთ და მიზნობრივად გამოვიკვლიოთ სხვადასხვა სახის ქსელები მასობრივი მომსახურების სისტემებში პროცესების შესრულების ეფექტური ორგანიზების თვალსაზრისით [23].

5.4.4. WinPepsy - დავების გადაწყვეტის სისტემისათვის

საშემოსავლო სამსახურის დავების გადაწყვეტის სისტემის მოდელი შეიძლება წარმოვადგინოთ ღია ქსელის სახით (ნახ. 5.39), რომელშიც შემავალი ნაკადი მომჩივანთა რიგის სახითაა მოცემული (მარცხენა ხუთკუთხედი). მათ რეგისტრაციას ემსახურება რამდენიმე ოპერატორი (M/M/m ტიპი).



ნახ. 5.39

საჩივრის გაცნობა და განსახილველად მომზადება ითვალისწინებს ხარვეზების გასწორებას (M/M/1 ტიპი), რომლის შემდეგაც მასალები შედის სარეგისტრაციო განყოფილებაში ხელახლად. თუ ყველაფერი წესრიგშია, მაშინ საჩივარი გადადის განხილვის ბლოკში (M/M/1 ტიპი). პროცესი სრულდება (გამოსასვლელი ხუთკუთხედი).

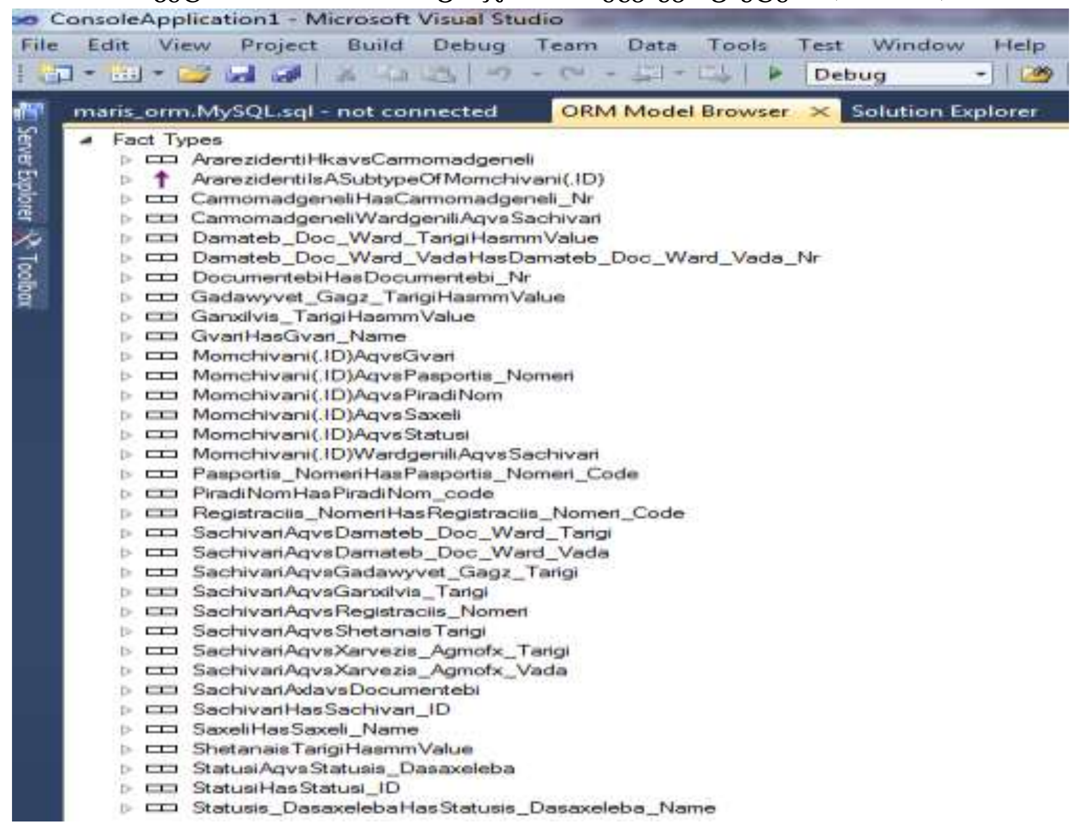
WinPepsy ინსტრუმენტით მიმდინარე პროცესების ანალიზისთვის უნდა იყოს გათვალისწინებული მასობრივი მომსახურების სისტემის შემდეგი ფაქტორები: მაგალითად, ჩვენ შემთხვევაში შემოსავლების სამსახური; დავების განხილვის საბჭო; განაცხადი (საჩივარი); რიგი; შემოსული განაცხადის (საჩივრების) ინტენსივობა; მომსახურების ინტენსივობა; საშუალო დრო, რომელიც განაცხადს (საჩივარს) ესაჭიროება რიგში დგომისას; რიგის საშუალო სიგრძე; საშუალო დრო, რომელიც ესაჭიროება განაცხადს (საჩივარს) სისტემაში მომსახურებისათვის (მიღება-განხილვა); მომსახურე სისტემაში კლიენტთა (მომჩივანთა) საშუალო რიცხვი; სისტემის მომსახურების ხარჯი; ლოდინის ხარჯი.

სქემაზე მოყვანილია ერთ– და მრავალარხიანი მოდელების მაგალთი და ზემოთ ჩამოთვლილი ფაქტორების გათვალისწინებით WinPepsy–ში გაითვლება შესაბამისი მნიშვნელობები, სადაც რიგების M/M/1 და M/M/m სისტემებისთვის დამახასიათებელი პარამეტრების მიხედვით.

მესამე თავში დამუშავებული გვაქვს ჩვენი საკუთარი პროგრამული სისტემა, რომლის შედეგებსაც მოგვიანებით შევეხებით.

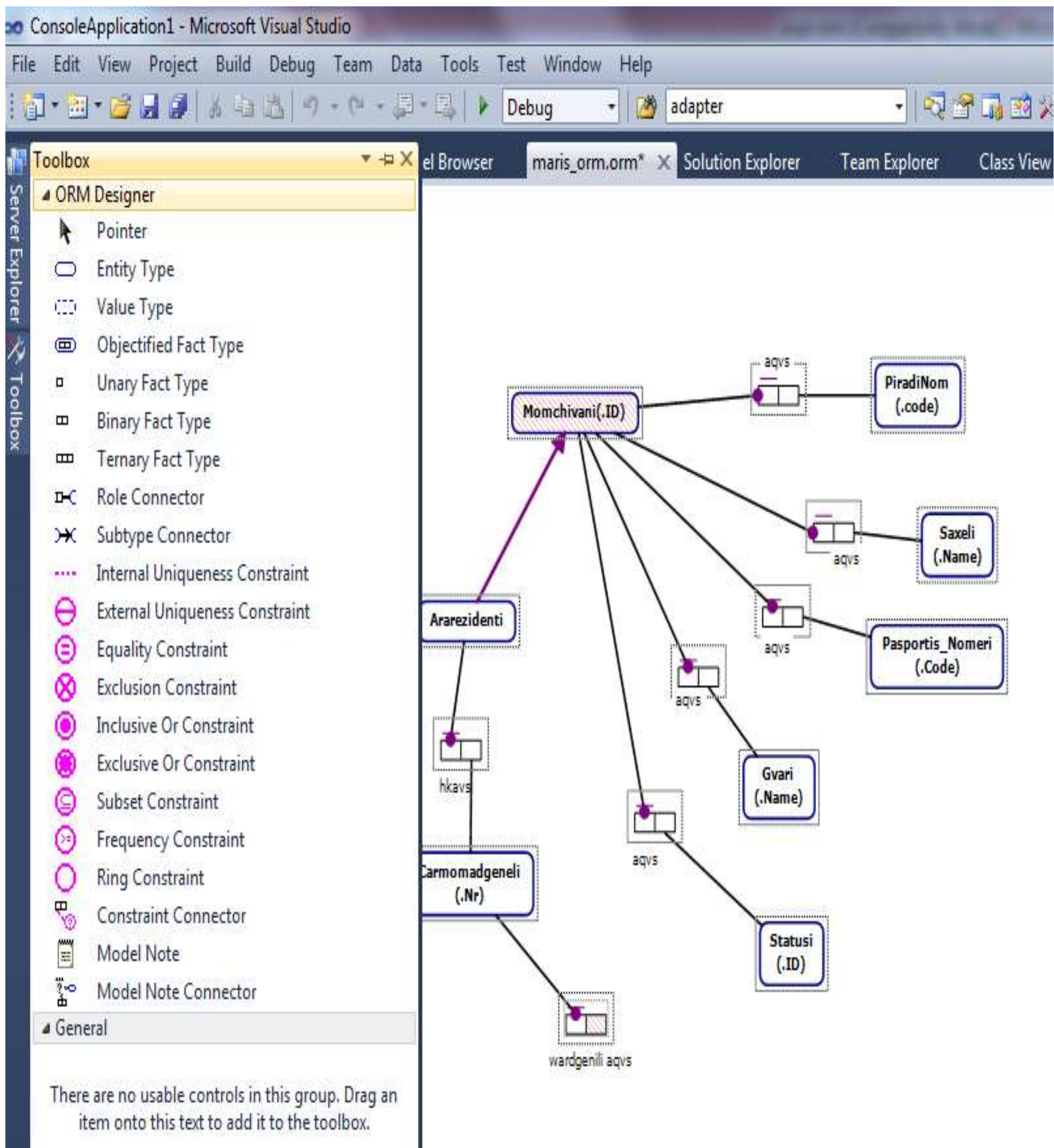
5.5. საპრობლემო სფეროს კონცეპტუალური სქემის ავტომატიზებული დაპროექტება CASE NORMA ინსტრუმენტით

თავდაპირველად ხდება საპრობლემო არის მოთხოვნილებათა ანალიზი, საიდანაც ჩამოყალიბდება კანონზომიერად არსებული ფაქტები. სწორედ ამ ელემენტარული ფაქტების საშუალებით განისაზღვრება ORM-მოდელი, რომელის მიხედვითაც შემდგომ აიგება ORM-დიაგრამა [26,36]. Natural ORM Architect პაკეტის Fact Editor -ის ფანჯარაში შეგვაქვს ფაქტები (ნახ.5.40).



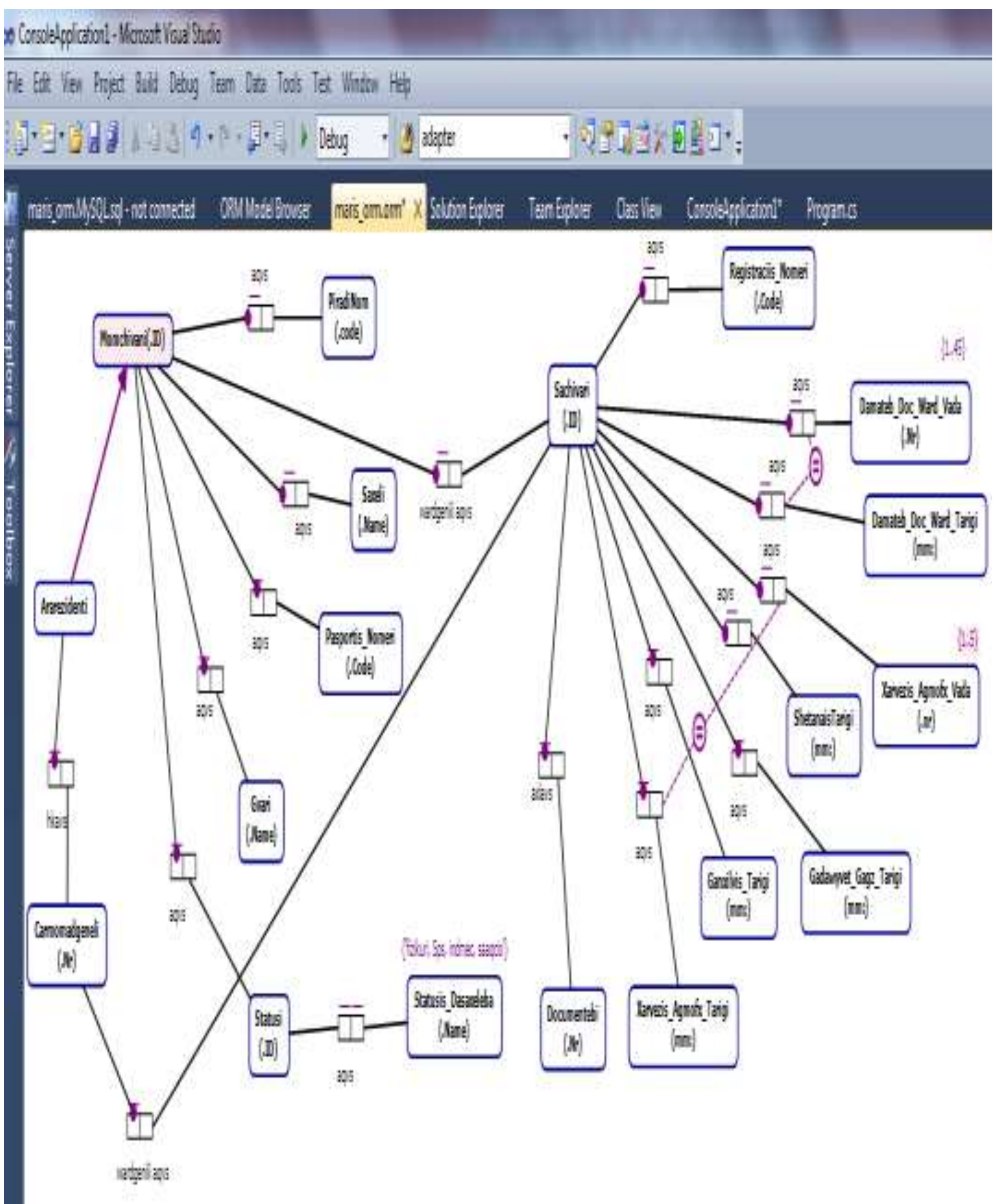
ნახ. 5.40

სქემის დაპროექტება ხდება განსაზღვრული შეზღუდვების დამატებით (ნახ.5.41).



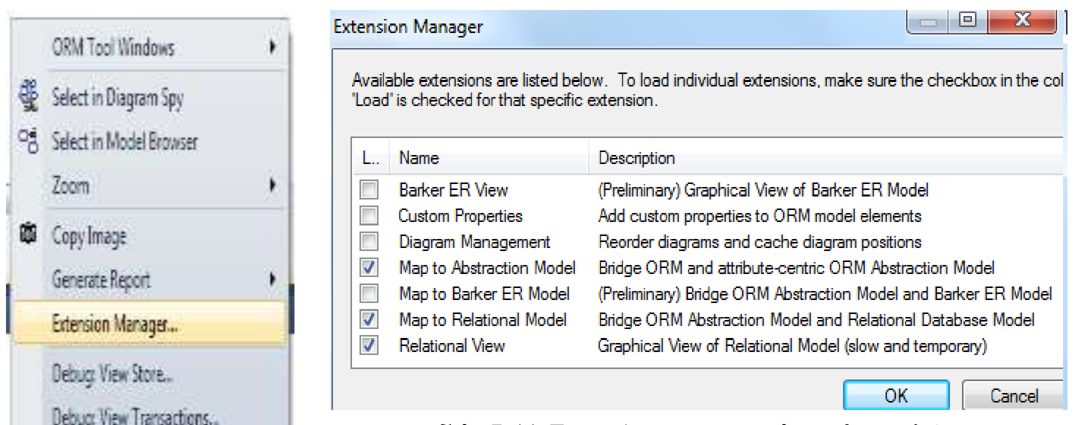
ნახ. 5.41. ORM -კონცეპტუალური მოდელი

ფაქტების ჩამონათვალს ORM Model Browser ფანჯარაში ექნება 5.42 ნახაზზე მოცემული სახე:



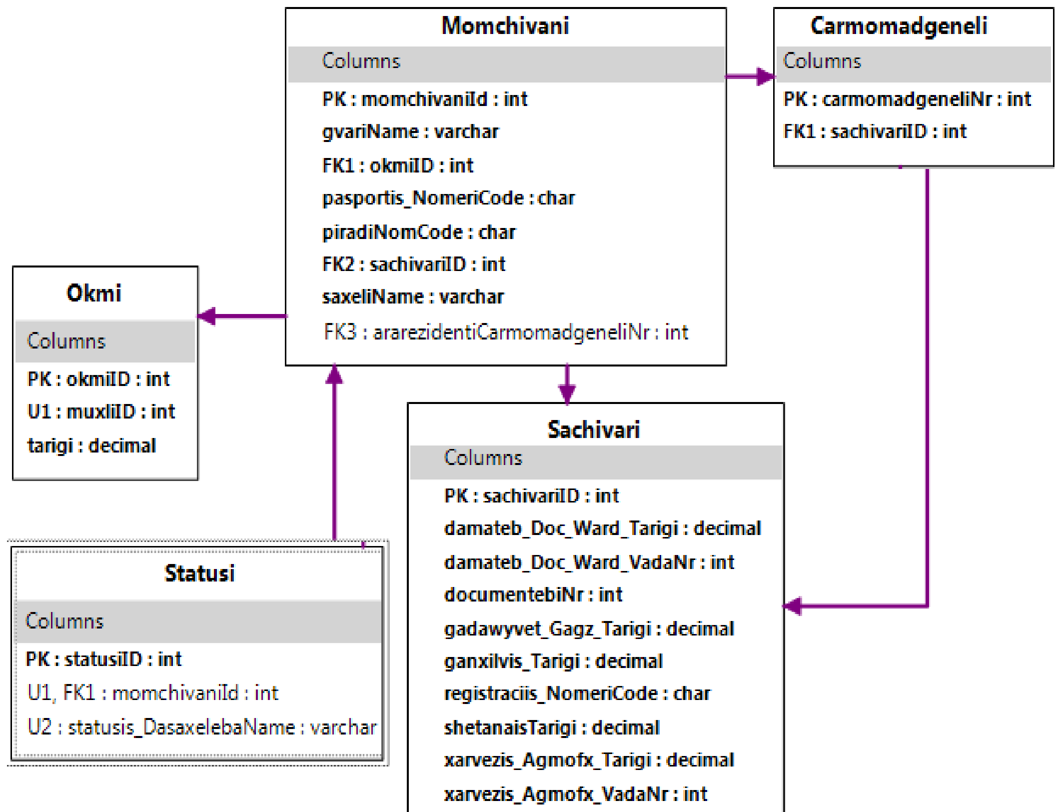
ნახ. 5.42. ORM-დიაგრამის ფრაგმენტი

დიაგრამიდან ავტომატიზებულ რეჟიმში ვღებულობთ ER-მოდელს Extension manager-ის საშუალებით (ნახ.5.43, 44).



ნახ. 5.44. Extension manager-ის გააქტიურება

5.44 ნახაზზე ნაჩვენებია ORM-დიაგრამის (ნახ.5.42) შესაბამისი ERM-სქემა, რომელიც ავტომატურად დაპროექტდა აღნიშნული CASE tool-ით.



ნახ. 5.44. ავტომატიზებულ რეჟიმში მიღებული ER-მოდელი

Natural ORM Architect პაკეტი ახდეს DDL კოდის გენერაციას. Solution Explorer-ის საშუალებით. 5.45 ნახაზზე მოცემულია ასეთი კოდის ფრაგმენტი (სრულ კოდს ჩვენ მომდევნო თავში განვიხილავთ).

სისტემაში შესაძლებელია ORM და ERM მოდელების ავტომატიზებულ რეჟიმში (ადამან-კომპიუტერული კორექტირების პროცესი) განახლება, რასაც მოჰყვება ასევე ავტომატურ რეჟიმში შესაბამისი ახალი კოდის ფორმირება.

```
CREATE TABLE Sachivari
(
        sachivariID INT AUTO_INCREMENT NOT NULL,
        damateb_Doc_Ward_Tarigi DECIMAL(65,65) NOT NULL,
        damateb_Doc_Ward_VadaNr INT NOT NULL,
        documentebiNr INT NOT NULL,
        gadawyvet_Gagz_Tarigi DECIMAL(65,65) NOT NULL,
        ganxilvis_Tarigi DECIMAL(65,65) NOT NULL,
        registraciis_NomeriCode CHAR(63) NOT NULL,
        shetanaisTarigi DECIMAL(65,65) NOT NULL,
        xarvezis_Agmofx_Tarigi DECIMAL(65,65) NOT NULL,
        xarvezis_Agmofx_VadaNr INT NOT NULL,
        CONSTRAINT Sachivari_PK PRIMARY KEY(sachivariID)
);
CREATE TABLE Rezident_ID
(
        `value` INT AUTO_INCREMENT NOT NULL,
        CONSTRAINT Rezident_ID_PK PRIMARY KEY(`value`)
);
```

ნახ. 5.54. Ms SQL Server-ისათვის ავტომატიზებულ რეჟიმში გენერირებული კოდის ფრაგმენტი

თავი 6

საგადასახადო სფეროს ბიზნეს-პროცესების მოდელირება და პროგრამული რეალიზაცია ახალი ინსტრუმენტული საშუალებებით

6.1. ბიზნესპროცესების მოდელირება UML/2 ტექნოლოგიის Enterprise Architect ინსტრუმენტით

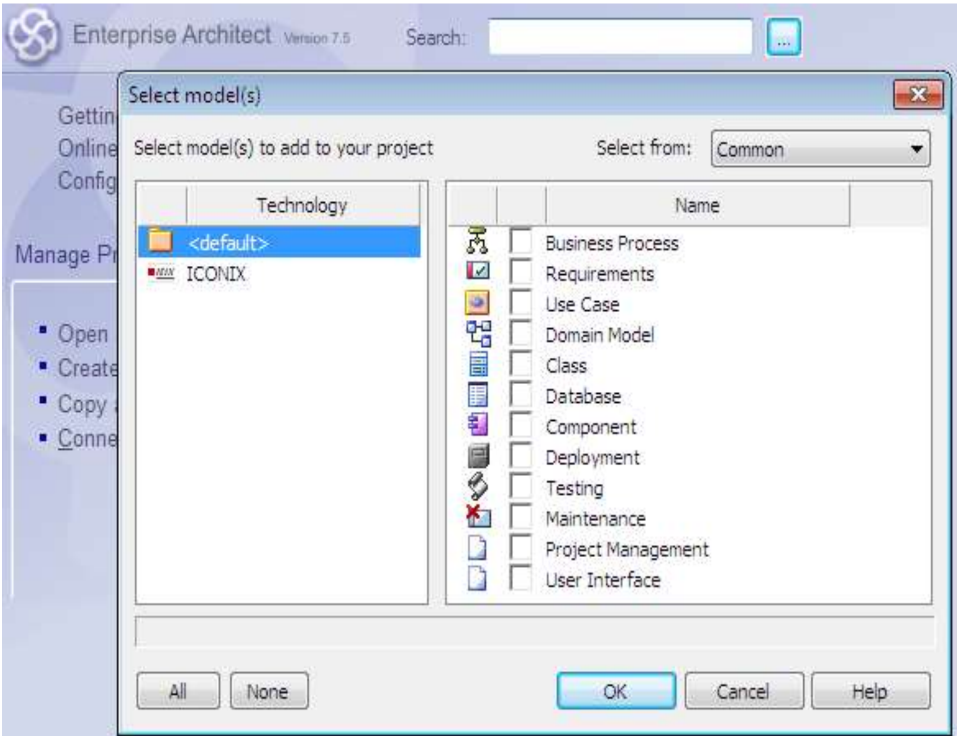
მოცემული თავი ეხება საინფორმაციო სისტემების დაპროექტებისა და მოდელირების საკითხებს. ამ თვალსაზრისით, განხილულია საინფორამაციო სისტემების დაპროექტებისა და დამუშავების თანამედროვე ტექნოლოგია (CASE – Computer Aided System Engineering), რომელსაც საფუძვლად უდევს უნიფიცირებული მოდელირების ენის სხვადასხვა ვერსიები. ჩვენს შემთხვევაში გამოყენებულია UML-ის ახალი UML2 ვერსია. ამ ტექნოლოგიაზე დაყრდნობით დამუშავებულია ბიზნეს-პროექტების მართვის სისტემის ტექნოლოგიური პროცესის მოდელი (ინსტრუმენტული საშუალებების გამოყენებით საპრობლემო არის ვიზუალური მოდელირება და ამ მოდელის ანალიზი სისტემის დამუშავების ყველა ეტაპზე).

მართვის ავტომატიზებული სისტემების სრულყოფილი, საიმედო და მოქნილი პროგრამული უზრუნველყოფის (Software Engineering) სწრაფად დაპროექტება, რეალიზაცია, დანერგვა და შემდგომი თანხლება სისტემის დამკვეთ ორგანიზაციაში მეტად მნიშვნელოვანი ამოცანაა. მისი ეფექტურად გადაწყვეტა ზევრადაა დამოკიდებული როგორც საპროექტო-დეველოპმენტის გუნდის შემადგენლობა გამოცდილებაზე, ასევე IT-ინფრასტრუქტურასა და CASE-ინსტრუმენტებზე [9,31].

Sparx Systems Enterprise Architect არის UML ინსტრუმენტების ერთობლიობის მძლავრი, მრავალმხრივი და მრავალპრალტპორმიანი სისტემა, იგი გამოიყენება პროგრამული უზრუნველყოფის დაპროექტებისა და აგების პროცესების ავტომატიზაციის მიზნით და შეიცავს ანალიზის, მოდელების გრაფიკული აგების, ტესტირებისა და მომსახურების ფაზებს. მასში თავმოყრილია UML, BPMN, Workflow და DocFlow ტექნოლოგიების გამოყენების საშუალებები, პირდაპირი და უკუდაპროექტების მექანიზმები, მონაცემთა ბაზების და პროგრამული ენების (Java, C#, C++) - მთელი სპექტრის გამოყენებით [73].

Enterprise Architect სისტემაში ახალი პროექტის შექმნისას, სასტარტო გვედზე იხსნება მოდელების არჩევის ფანჯარა (ნახ.6.1), რის მიხედვითაც შემდგომში იგება შესაბამისი მოდელები. გამოყენებაშია როგორც UML1 და

UML2 ინსტრუმენტების ერთობლიობა, ისე ბიზნეს-პროცესების მოდელირების თანამედროვე სტანდარტები.

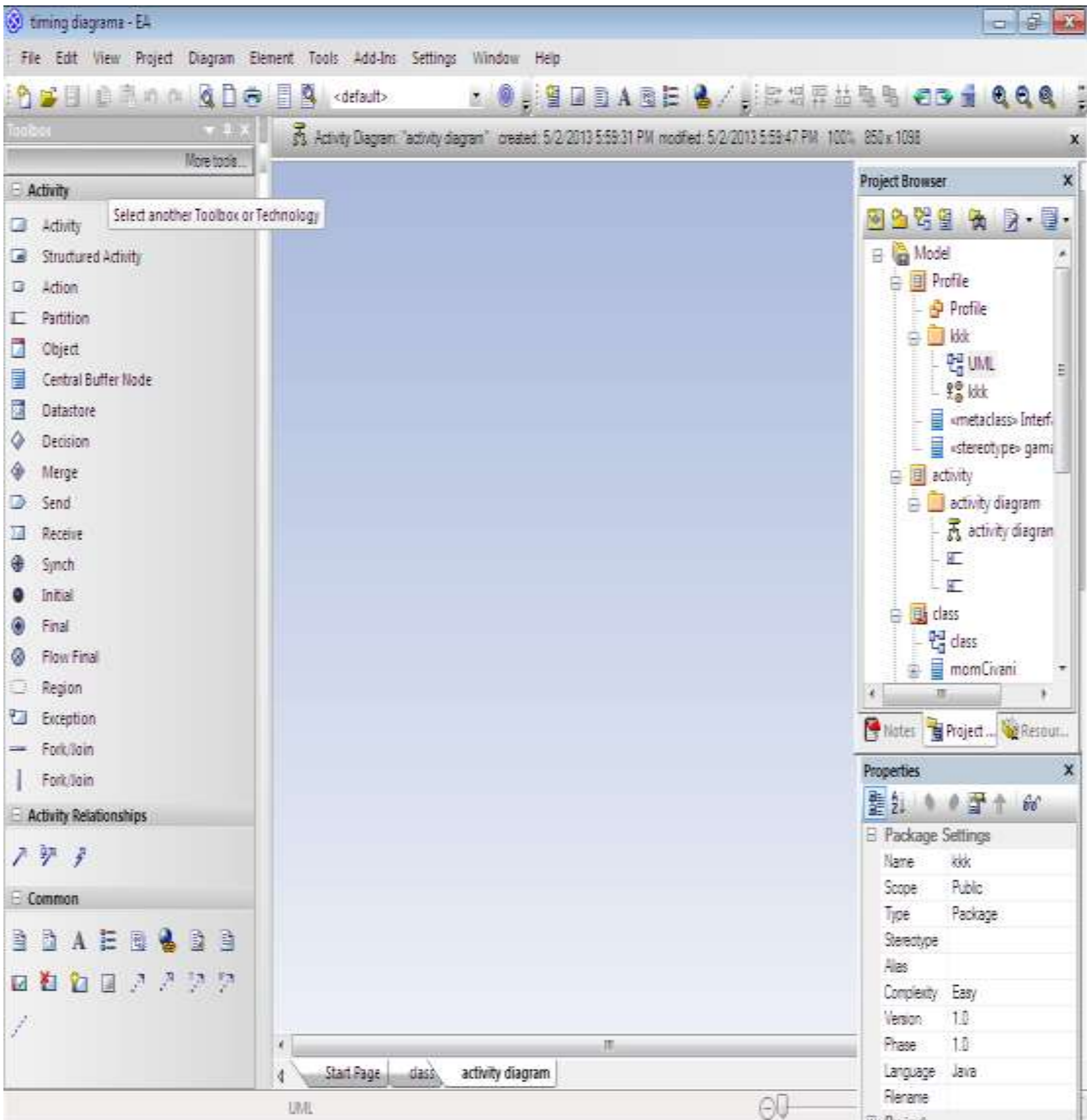


ნახ. 6.1. Enterprise Architect სისტემაში მოდელების არჩევის ფანჯარა

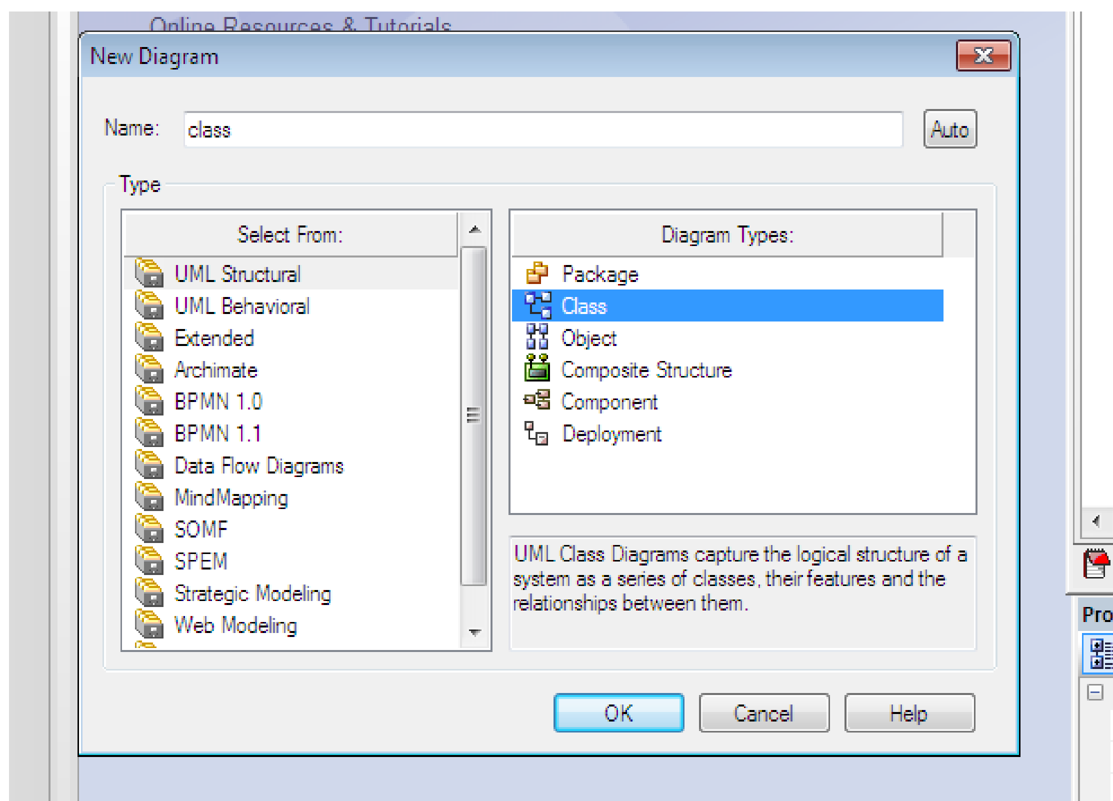
სისტემის სასტარტო გვერდი გაყოფილია სამ ძირითად ნაწილად:

1) ინსტრუმენტების პანელი “Toolbox”,
2) მოდლების პანელი “Project Browser” და
3) სამუშაო გარემო (ნახ.6.2).

თითოეულ მოდელში შესაძლებელია შეიქმნას UML1 და UML2 ენის სტანდარტული დიაგრამები - მენიუდან Project - AddDiagram (Ctrl+Insert). ფანჯარაში “New diagram”, ველში “Name” იწერება “Project Browser” პანელში მონიშნული მოდელის სახელი, დიალოგის მარცხენა ნაწილიდან “Select Form” ხდება მოდელის საჭირო ფორმის არჩევა, რომლის გააქტიურებით მარცხენა ნაწილში “Diagram types” იხსნება მოდელის შესაბამისი დიაგრამის ტიპები (ნახ. 89). ღილაკით “OK” არჩეული დიაგრამა ჯდება მითითებულ მოდელში (“project browser” პანელში) ხოლო, ინსტრუმენტების პანელში “Toolbox”, ჩნდება დიაგრამის შესაბამისი ელემენტები.



ნახ. 6.2. Enterprise Architect სისტემის სასტარტო გვერდი



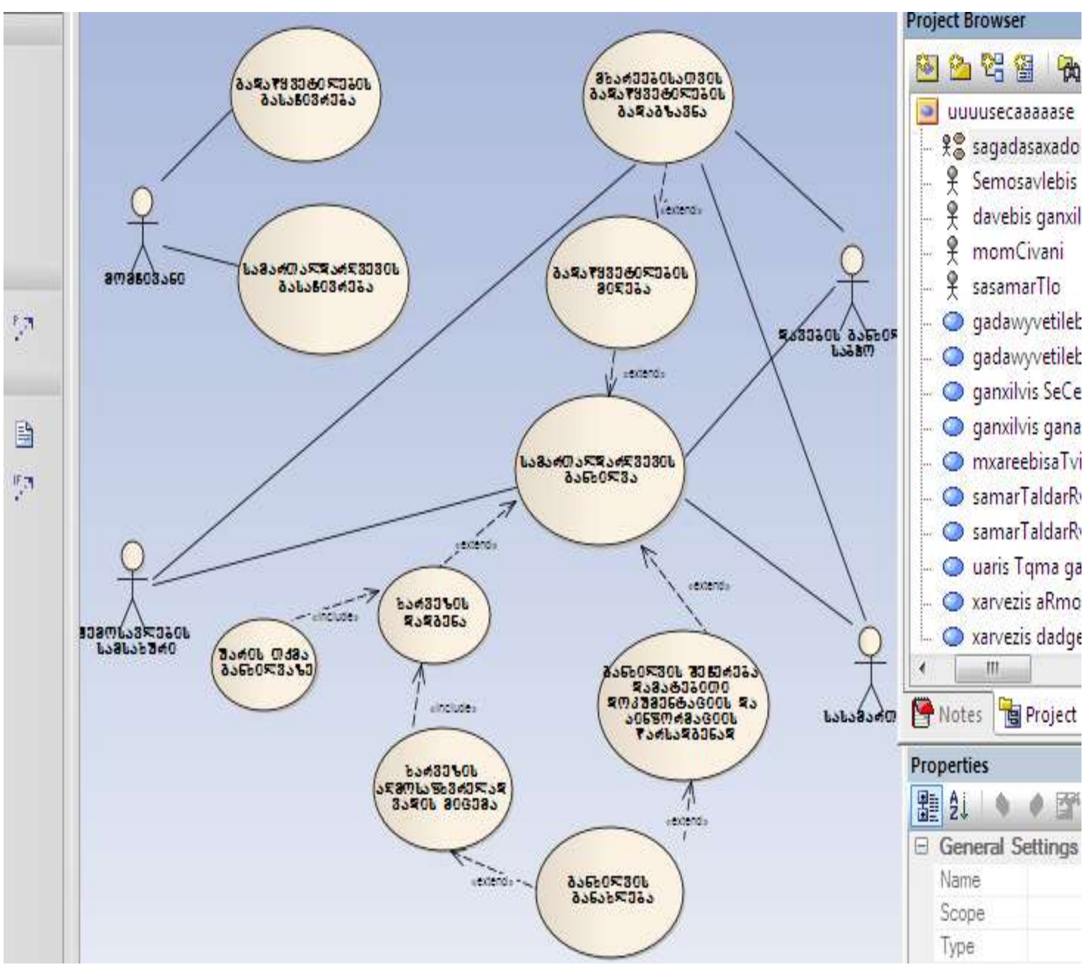
ნახ. 6.3. სტანდარტული დიაგრამების დამატების ფანჯარა

დიდი პროექტისთვის, რომელშიც რესურსები და დროითი ფაქტორები, შედარებით კრიტიკული არაა, ხდება ობიექტ-ორიენტირებული მიდგომის ყველა ეტაპის და ფაზის გამოყენება შესაბამისი საკონტროლო წერტილების აუცილებელი მონიტორინგით და რეპორტებით. ამ დროს სრული მოცულობით ხორციელდება უნიფიცირებული მოდელირების ენის (UML/2) და შესაბამისი ინსტრუმენტული საშუალების, მაგალითად, MsVisio ან Enterprise Architect პაკეტის გამოყენება [31,74].

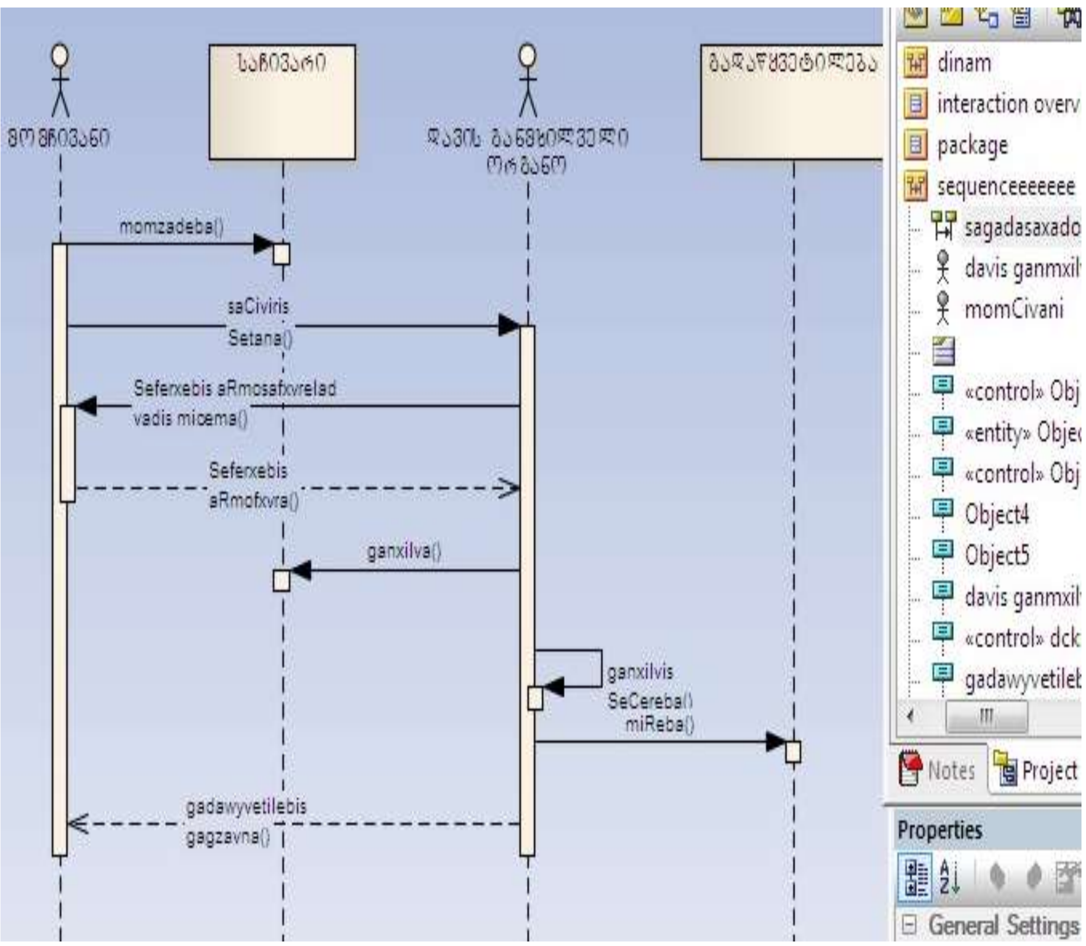
გამოყენებითი პროგრამული უზრუნველყოფის (Applied Software) ობიექტ-ორიენტირებული ანალიზის, დაპროექტების და რეალიზაციის ცალკეული ეტაპების მოდელირების მიზნით განიხილება UML მეთოდოლოგიის ინსტრუმენტული საშუალება Ms Visio Professional. ეს პაკეტი ძალზე პოპულარული და მრავალფუნქციურია.

6.1.1. საგადასახადო დავების წარმოების ბიზნესპროცესების მოდელირება

Enterprise Architect პროგრამული პაკეტის დახმარებით ავაგეთ ის დიაგრამები (UseCase, Sequence), რომლებიც შემდგომში სისტემის (საგადასა-ხადო დავის წარმოების) ფრაგმენტის ინფორამციული უზრუნველყოფის შექმნის წინმსწრებ ეტაპზე დაგვჭირდება (ნახ. 6.4 - 6.5).

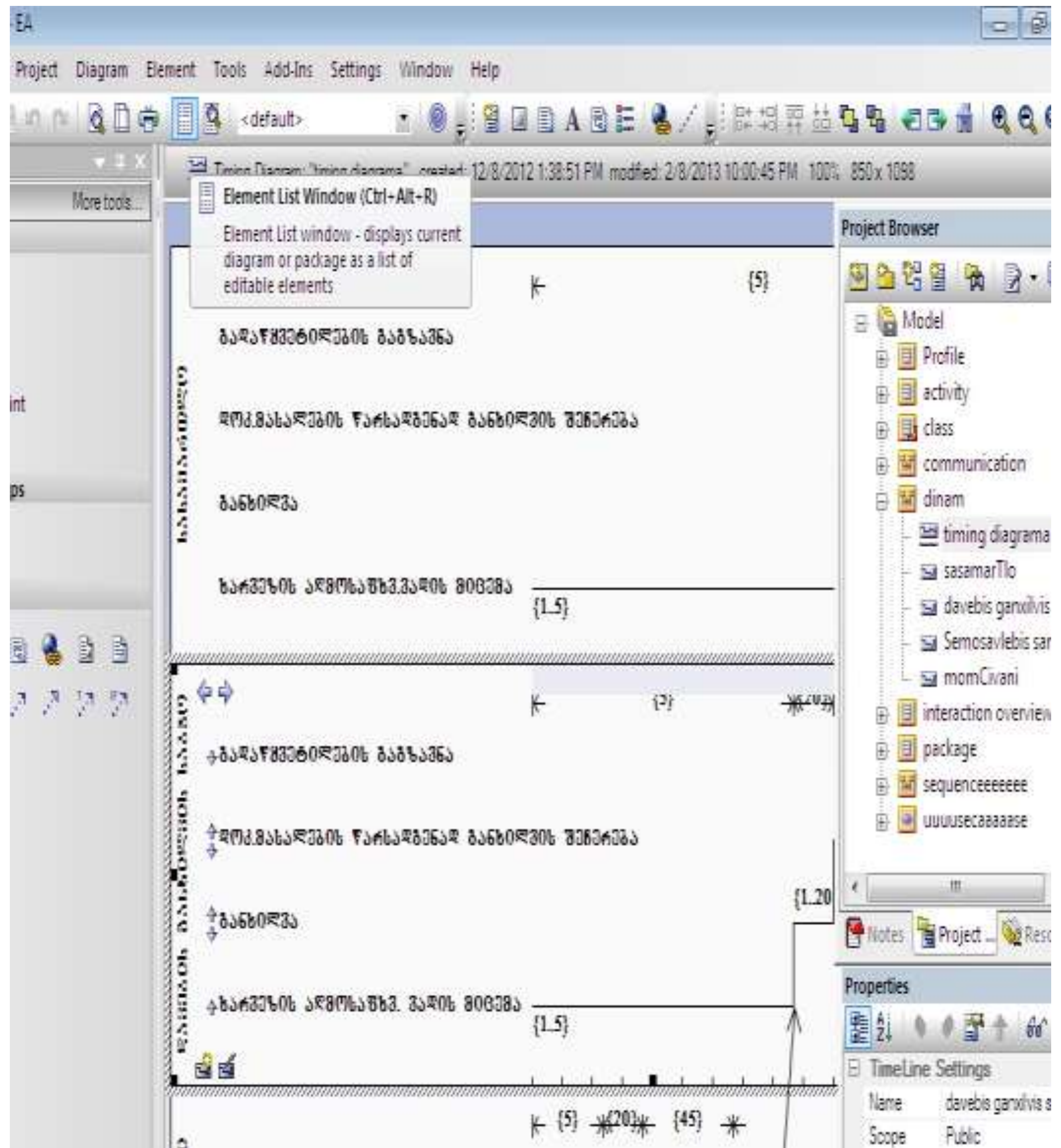


ნახ. 6.4. Use Case - დიაგრამა



ნახ. 6.5. Sequence – დიაგრამა

სინქრონიზაციის დიაგრამა მიმდევრობითობის დიაგრამის ალტერნატი-ული ვარიანტია. დიაგრამა განსაზღვრავს სხვადასხვა მდგომარეობათა ცვლილებებსა და ობიექტების ქცევას დროითი შკალის საზღვრებში. შესაძლებელია გამოყენებულ იქნას, დროის მიხედვით მართვადი ბიზნეს-პროცესებისათვის [73]. მაგალითად საგადასახადო დავის წარმოების პროცესების დროში ასაღწერად (ნახ.6.6).



ნახ. 6.6. სინქრონიზაციის დიაგრამა

სინქრონიზაციის დიაგრამა ძირითადად, შედგება მდგომარეობისა და ცვლადების სასიცოცხლო ციკლის ელემენტებისაგან. მდგომარეობის სასიცოცხლო ციკლი უჩვენებს მოვლენათა მსვლელობას დროსთან მიმართებაში [31].

6.1.2. კლასების დიაგრამიდან პროგრამული კოდის გენერირება

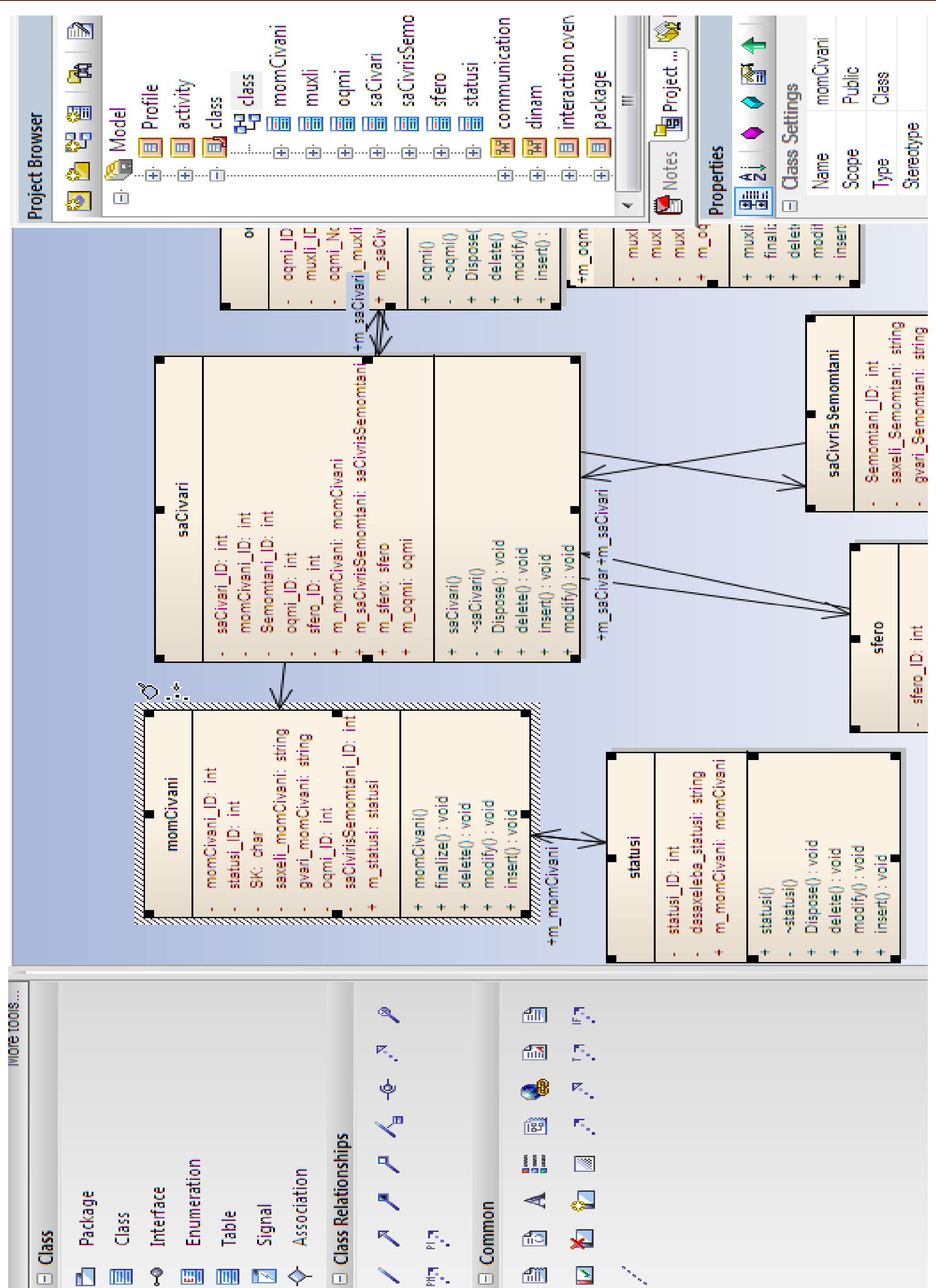
თანამედროვე CASE-ტექნოლოგიები, რომლებიც სისტემების დაპროგრამების ავტომატიზაციაზეა ორიენტირებული, მაგალითად, Rational Rose, Visual Paradigm, Enterprise Architect და მრავალი სხვა [74], ახორციელებს რევერსული დაპროგრამების კონცეფციას. ანუ კლასების დიაგრამიდან შესაძლებელია პროგრამული კოდის გენერაცია და პირიქითაც, კოდიდან აიგება ავტომატურად გრაფიკული დიაგრამა (ნახ. 6.7, 6.8).

MsVisio არ მიეკუთვნება ასეთი სიმძლავრის ინსტრუმენტს. მისი საშუალებით დიალოგურ რეჟიმში იხაზება დიაგრამები (UML-ის სტანდარტულ აღნიშვნათა საერთაშორისო ნორმებით), მაგრამ კოდის გენერაცია არაა შესაძლებელი.

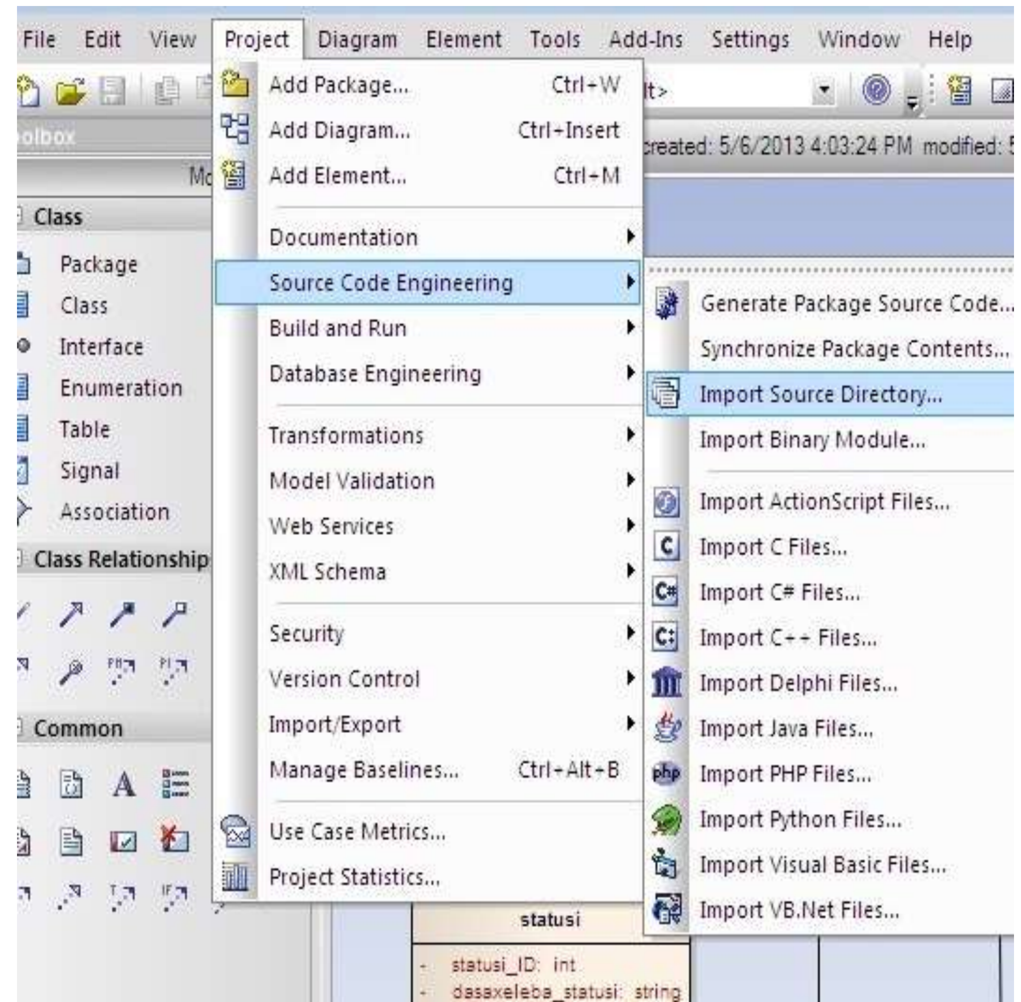
Ms Visual Studio .NET Framework -ისთვის შექმნილია ინსტრუმენტები და მათი ინტეგრაციით .NET გარემოში, შესაძლებელია დიაგრამებიდან კოდის გენერაცია. ასეთი პაკეტები ყოველთვის ფასიანია და ძვირადღირებული. აქ განვიხილავთ SparX ფირმის Enterprise Architect პროდუქტის ამ კონკრეტულ ფუნქციას, კლასების დიაგრამიდან კოდის გენერაციის ამოცანას.

Import Source Directory -ის არჩევის შემდეგ გამოვა ნახაზზე (ნახ. 6.9) ნაჩვენები ფანჯარა, რომელშიც უნდა განისაზღვროს ზოგიერთი მნიშვნელოვანი პარამეტრი, მაგალითად, ენა (C#),), მომავალი კოდის შესანახი ადგილი (დირექტორია) და ა.შ.,

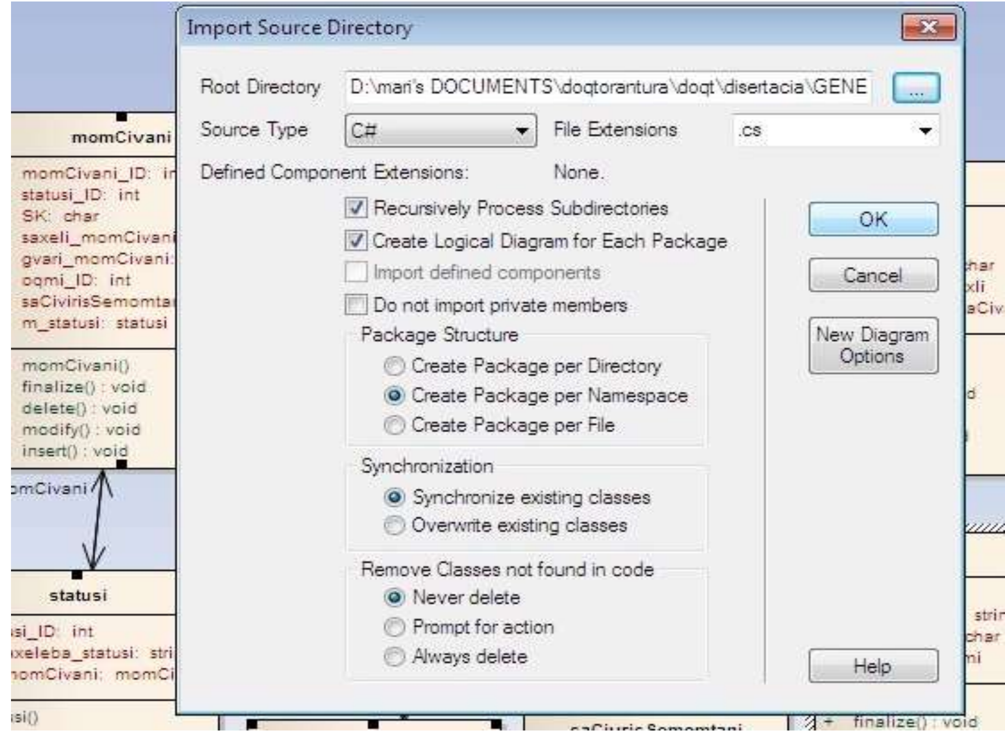
ბოლოს “Ok” და მივიღებთ ნახაზზე (ნახ.6.10) მოცემულ შედეგს.



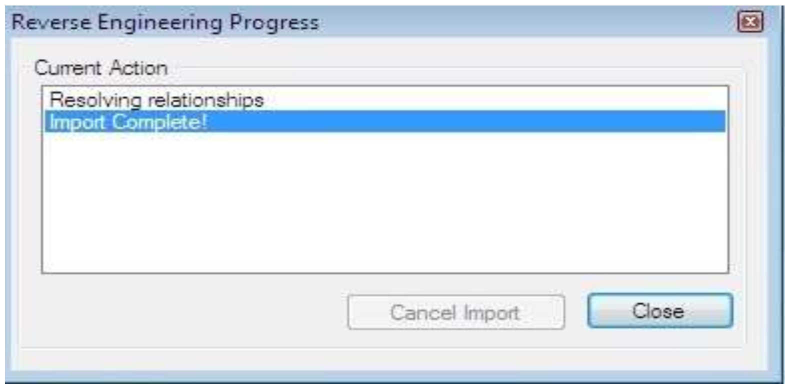
ნახ. 6.7. კლასების დიაგრამის ფრაგმენტი კოდის გენერირებისთვის



ნახ. 6.8. კლასების მომზადება “Code Engineering” პროცესისთვის Enterprise Architect გარემოში

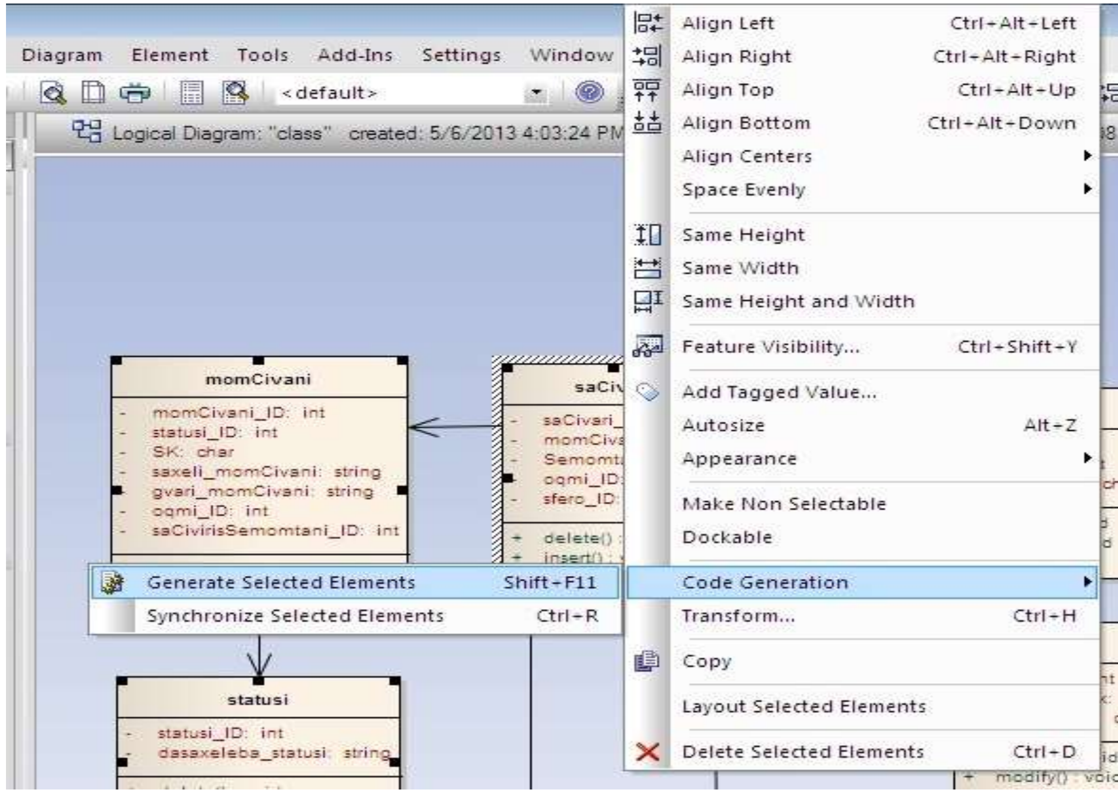


ნახ. 6.9. განისაზღვროს C#-კოდის დირექტორია

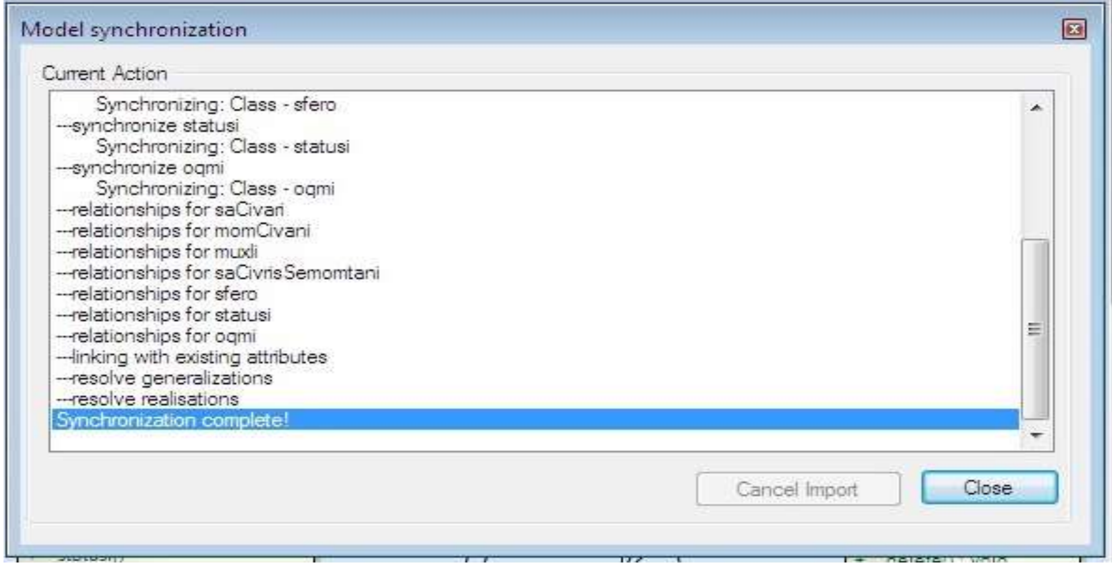


ნახ. 6.10. იმპორტი დასრულებულია

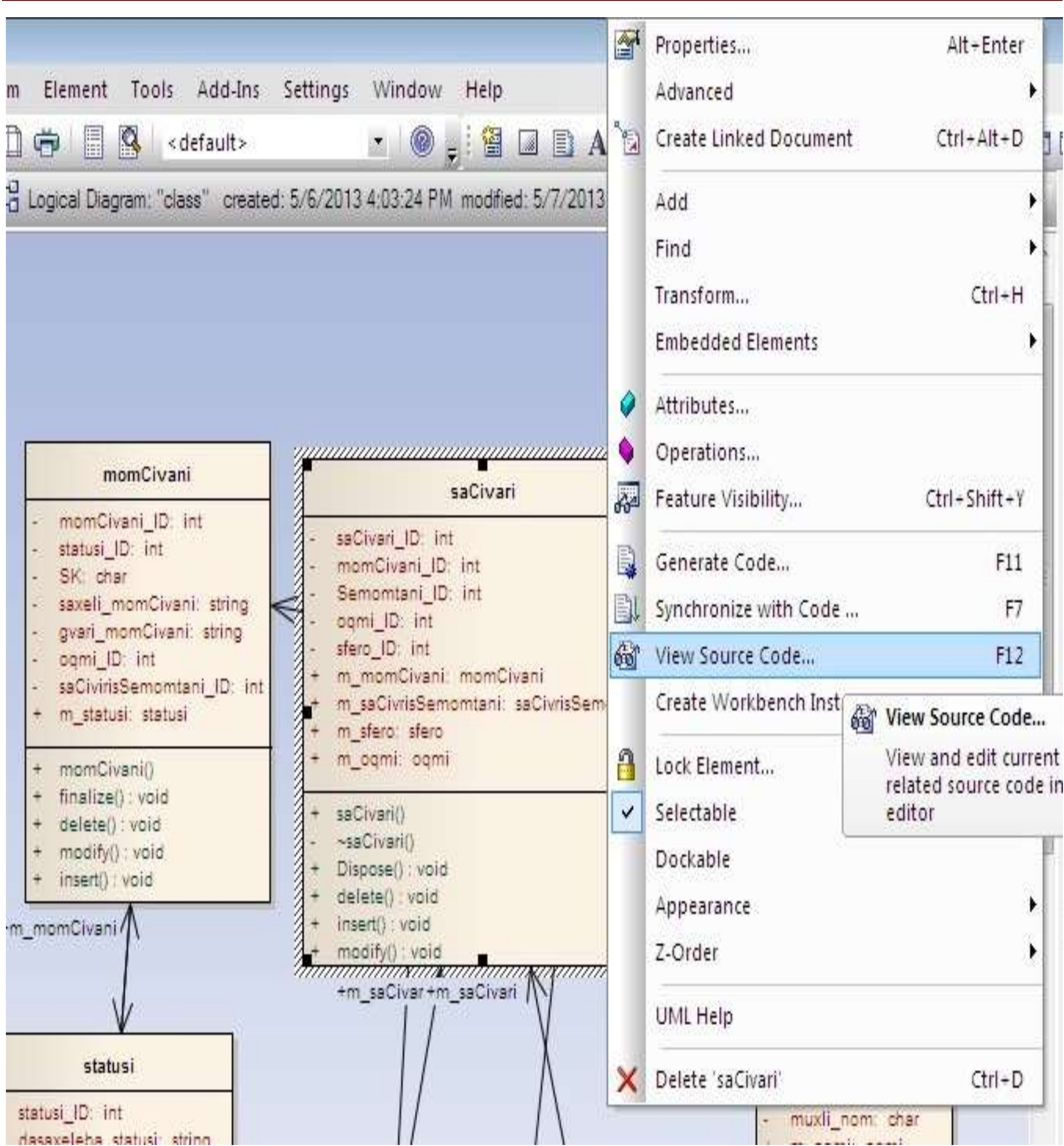
უშუალოდ კოდის გენერაცია წინასწარ მომზადებული (Sachivari) კლასების დიაგრამიდან ხდება. ვააქტიურებთ კლასებს და მაუსის მარჯვენა ღილაკით გამოტანილ კონტექსტური მენიუდან ვირჩევთ “Code Generation” (ნახ.6.11, 6.12). ბოლო ფაზაზე გამოგვაქვს ეკრანზე კლასების ბაზაზე გენერირებული კოდის ლისტინგი (ნახ.6.13).



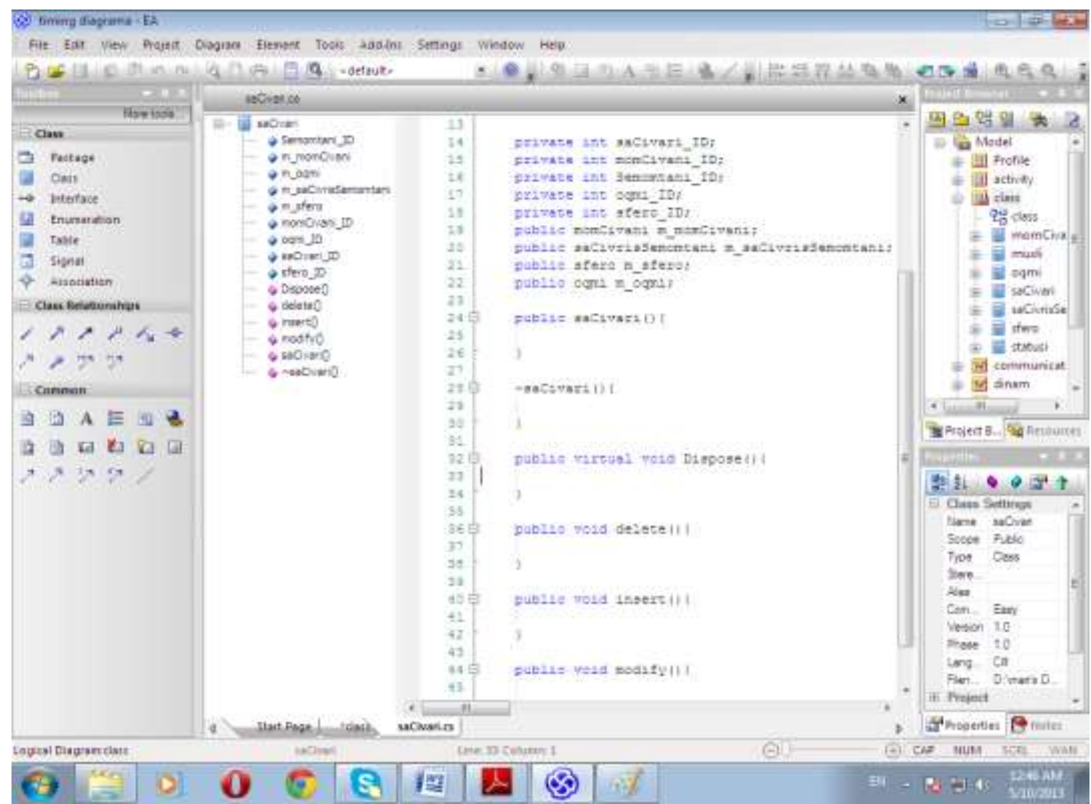
ნახ. 6.11. კოდის გენერაციის დაწყება



ნახ. 6.12. არჩეული ელემენტების სინქრონიზაციის შედეგი



ნახ. 6.13. კოდის გამოტანის პუნქტი მენიუში



ნახ. 6.14. C#-კოდის ლისტინგი კლასისთვის Sachivari

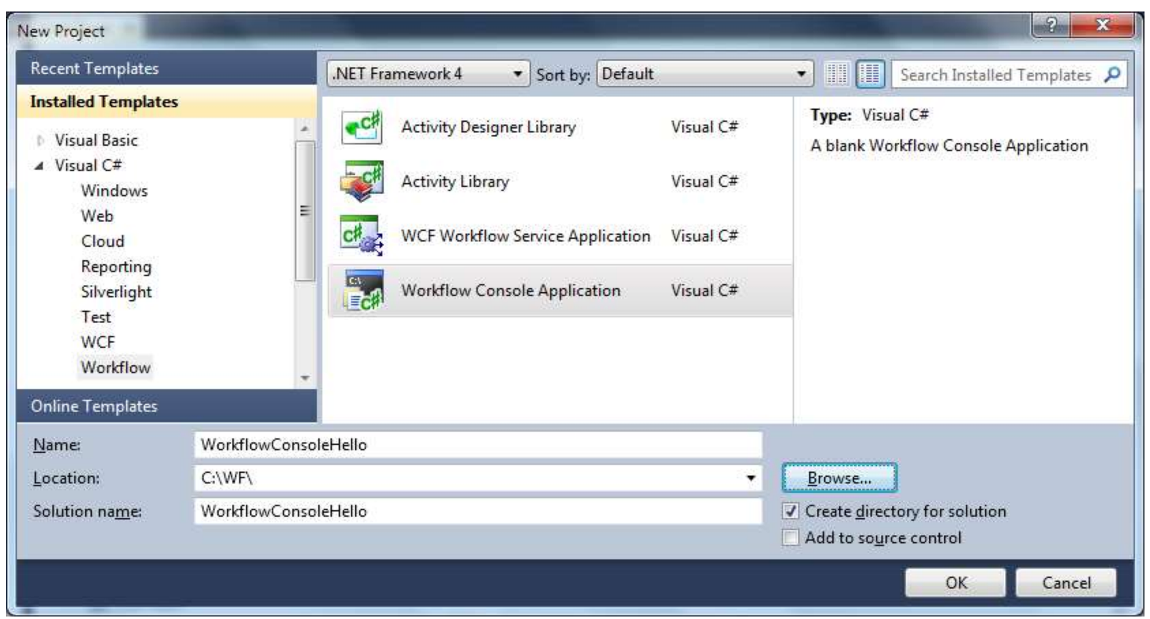
```
///////////////////////////////////////////////////////////
//  saCivari.cs
//  Implementation of the Class saCivari
//  Generated by Enterprise Architect
//  Created on:      10-May-2013 12:40:28 AM
//  Original author: Home
///////////////////////////////////////////////////////////
public class saCivari {
        private int saCivari_ID;
        private int momCivani_ID;
        private int Semomtani_ID;
        private int oqmi_ID;
        private int sfero_ID;
        public momCivani m_momCivani;
        public saCivrisSemomtani m_saCivrisSemomtani;
```

```
        public sfero m_sfero;
        public oqmi m_oqmi;
        public saCivari(){          }
        ~saCivari(){        }

        public virtual void Dispose(){       }
        public void delete(){   }
        public void insert(){   }
        public void modify(){   }
}//end saCivari
```

6.2. პროექტების აგება Workflow Foundation ტექნოლოგიით .NET პლატფორმაზე

workflow-ის შექმნა ხდება Visual Studio.NET გარემოში. Template-ში Visual C#-ს და Workflow-ს, ხოლო შუა ფანჯრიდან Workflow Console Application-ის არჩევით (ნახ.6.15).

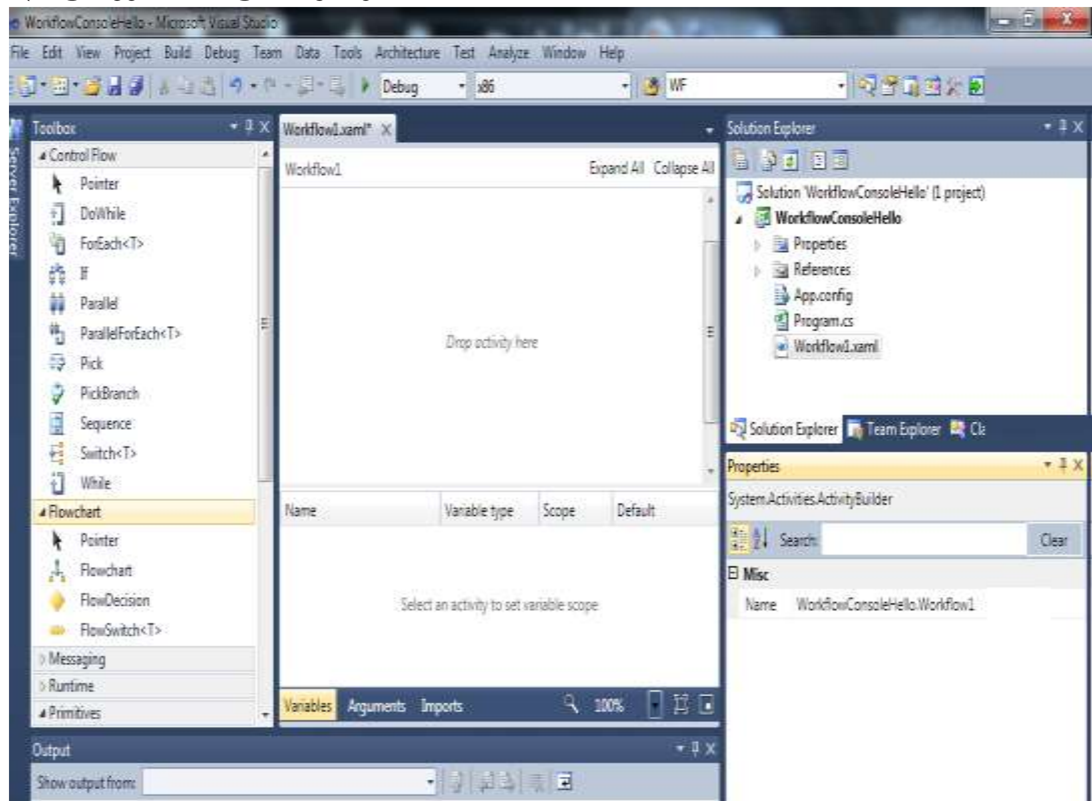


ნახ. 6.15. ახალი workflow პროექტის შექმნა

შაბლონი (Template) აგენერირებს Program.cs ფაილს, რომელიც რეალიზებას უკეთებს კონსოლურ აპლიკაციას (დანართს). იგი ასევე აგენერირებს Workflow4.xaml ფაილს, რომელიც განსაზღვრავს ქმედებას (აქტიურობას) სამუშაო პროცესში (workflow-ში). XAML ენა გამოიყენება პროგრამული ელემენტების გამოსაცხადებლად, ოღონდ ლებელის, ტექსტ-ბოქსის და ბადის ნაცვლად ეს ფაილი შეიცავს წარმოებული ელემენტების აქტიურობებს ჩვენს მიერ განსაზღვრულ სამუშაო პროცესში.

VS აძლევს დიზაინერს ქმედებების (აქტიურობების) გრაფიკულად ნახვის და რედაქტირების საშუალებას.

6.16 ნახაზზე ნაჩვენებია Visual Studio .NET პლატფორმის ინტეგრირებული დამუშავების სამუშაო გარემო (IDE).

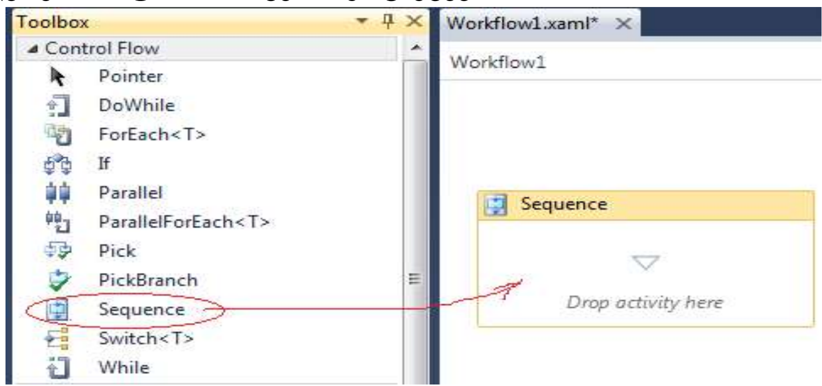


ნახ. 6.16. Visual Studio .NET სამუშაო გარემო

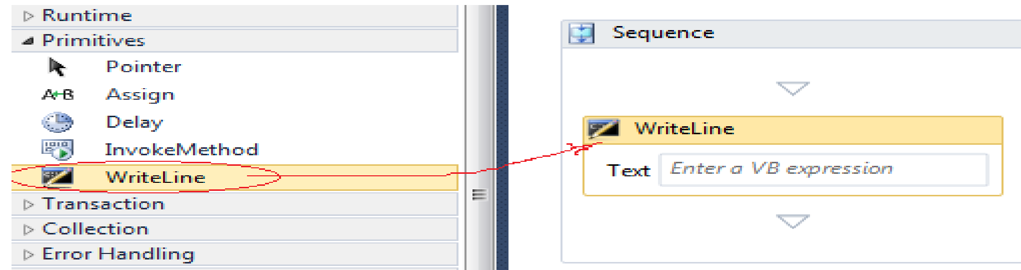
მარცხნივ მოთავსებულია ინსტრუმენტების პანელი, რომელიც მოიცავს ჩადგმულ და მომხმარებლის აქტიურობებს, რომელთა გაფართოება შესაძლებელია სხვა აქტიურობებითაც. Solution Explorer და თვისებათა ფანჯარა

მარჯვნივაა მოთავსებული. ქვემოთ ფანჯარაში გამოიტანება შეცდომების შეტყობინებები, შუალედური შედეგები და სხვა ინფორმაცია.

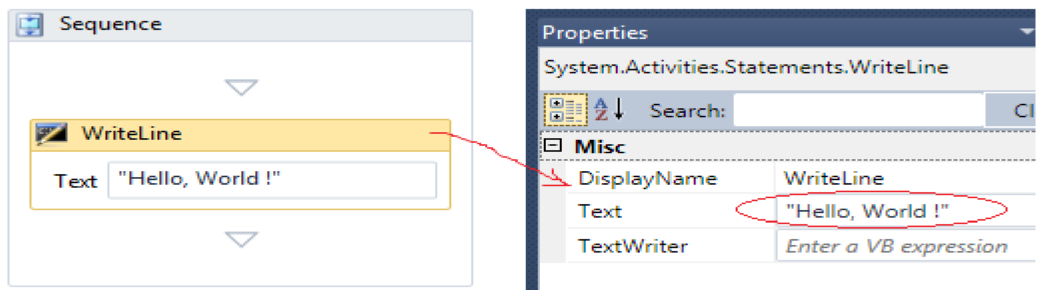
თავიდან სამუშაო ნაკადის კონსტრუქტორი ცარიელია. ინსტრუმენტების პანელიდან საჭირო აქტიურობა გადაიტანება დიზაინერის გარემოში, რითაც განისაზღვრება სამუშაო პროცესის ყოფაქცევა (ნახ.6.17 - 6.20).



ნახ. 39Sequence ქმედების დამატება



ნახ. 6.18. WriteLine ქმედების დამატება



ნახ. 6.19. Text –ის მნიშვნელობის შეტანა Properties-ში

სისტემის ამუშვების შემდეგ შედეგი მიიღება კონსოლის რეჟიმში (ნახ.6.20).

ნახ. 6.20. კონსოლზე გამოსული შედეგი

WF 5.0 აქვს რიგი პროცედურული ელემენტებისა, როგორიცაა, მაგალითად, If, While, Assign, Sequence და სხვა. მუშა ელემენტში გამოყენებული ცვლადები ღილაკზე Variables დაჭერით უნდა გამოცხადდეს (ნახ.6.21).

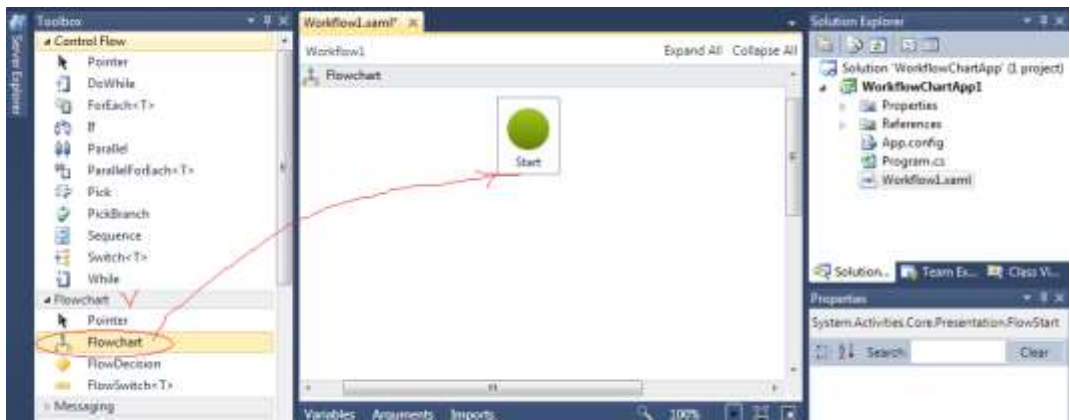
ნახ. 6.21

ბიზნესპროცესის დიაგრამა (Flowchart Workflow).

Workflow Foundation მუშა ბიზნეს-პროცესის აგებისას იყენებს აქტიურობათა დიაგრამას. დიაგრამა მუშაობს როგორც ბლოკ–სქემა, ქმედებათა სახეები დაკავშირებულია ერთმანეთთან გადაწყვეტილების ხეებით (decision trees).

ქმედებათა მიმდევრობითობის გამოყენებით, შვილი–პროცესები სრულდება მიმდევრობით ზემოდან-ქვემოთ (top-down).

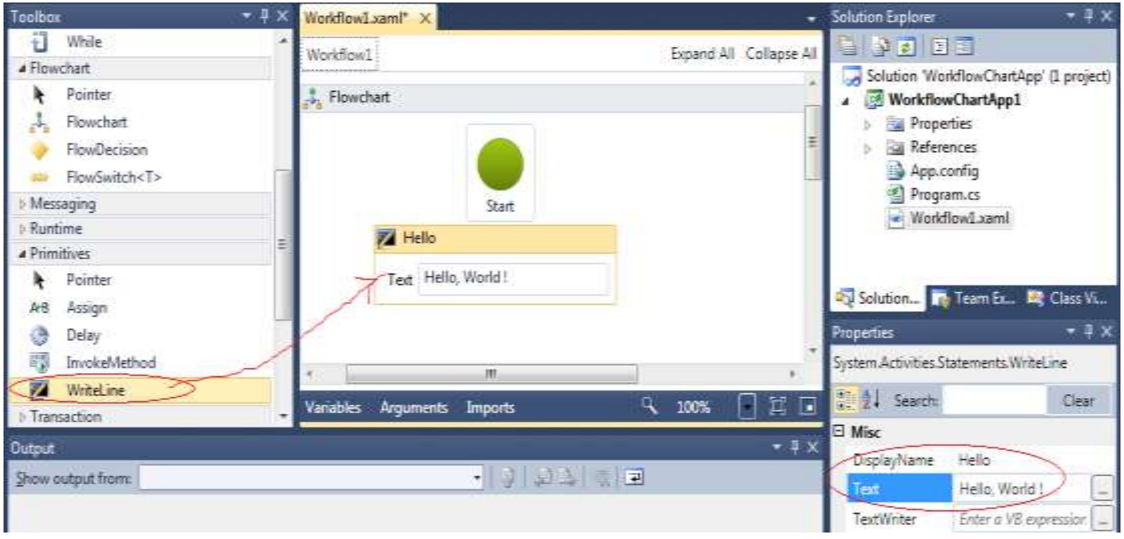
ამისდა მიუხედავად, შვილი-ქმედებები შეიძლება შესრულდეს ნების-მიერი მიმდევრობით, გადაწყვეტილების მიმღები პირის მიერ. მუშა პროცესის დაწყება დიაგრამაზე სასტარტო მწვანე წრით აღინიშნება (ნახ.6.22).



ნახ. 6.22

ძირითადი განსხვავება Flowchart ქმედებასა და Sequence ქმედებას შორის ისაა, თუ როგორაა დაკავშირებული შვილი–აქტიურობები. ქმედებების დამატებისას მიმდევრობითობაში ისინი სრულდება დაღმავალი (top-down) რიგითობით. აქ შესაძლებელია მიმდევრობის კონტროლი (მართვა), ქმედებათა გადაადგილებით, ოღონდ ისინი ყოველთვის გასწორებულია ვერტიკალში და განაწილებულია თანაბრად. ისრები ქმედებებს შორის კი აგებულია ავტომატურად.

Flowchart აქტიურობით შესაძლებელია ქმედებათა განთავსება პალიტრის ნებისმიერ ადგილას. მნიშვნელოვანია ისიც, რომ აქ ისრები ხელით უნდა გაკეთდეს და შეერთება დასაშვებია უკან, წინა ქმედებასთანაც (ნახ.6.23).

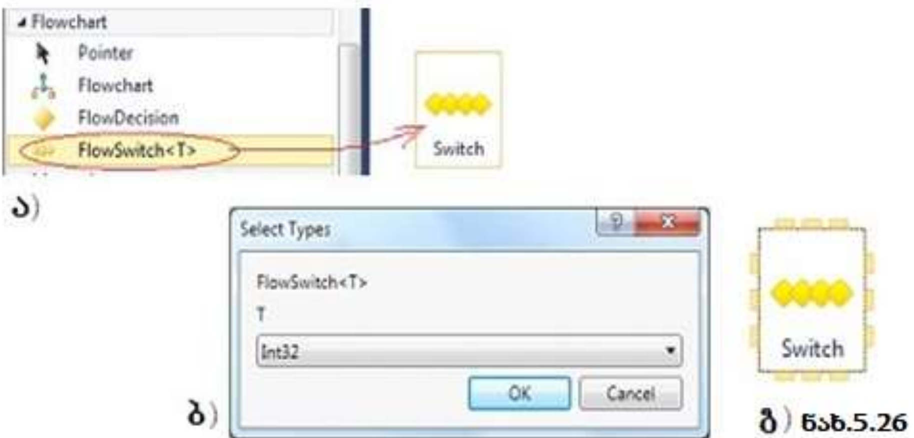


ნახ. 6.23

პირობისა და გადაწყვეტილების ნაკადის სქემაზე ჩვენება FlowDecision ქმედების საშუალებით ხდება. FlowDecision ქმედება გამოიყურება ყვითელი ალმასის სახით, როგორც განშტოების სიმბოლო ნორმალურ ბლოკ–სქემაში, თვისებათა ფანჯარაში შეიტანება პირობა (Condition) - ნახ.6.24.

ნახ. 6.24

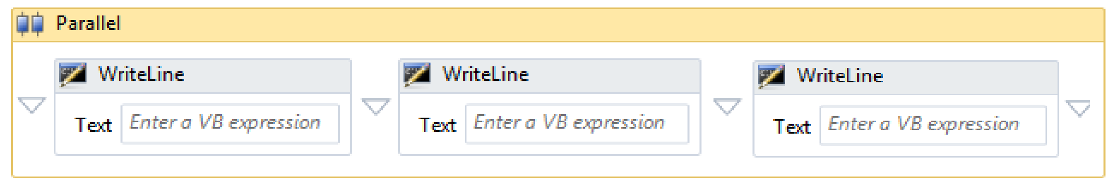
რაც შეეხება FlowSwitch ქმედებას, იგი ფუნქციონირებს, როგორც FlowDecision იმ გამონაკლისით, რომ არაა შეზღუდვა მხოლოდ true/false ორი განშტოებით. შესაძლებელია შტოების ნებისმიერი რაოდენობის განსაზღვრა ისე, როგორც ეს იყო C# ენაში. FlowSwitch არის <T> კლასის შაბლონი და უნდა მიეთითოს ტიპი. 6.25 ნახაზზე ნაჩვენებია FlowSwitch აქტიურობის პიქტოგრამა ინსტრუმენტების პანელზე.



ნახ. 40ა–პიქტოგრამა, ბ–ტიპის განსაზღვრა,
გ–შედეგი ფანჯარაში

ნახ. 6.27. FlowSwitch აქტიურობის პიქტოგრამა

პარალელური პროცესების ჩვენება სქემაზე Parallel ქმედების საშუალებით ხორციელდება, რომელიც საშუალებას იძლევა განვსაზღვროთ პარალელურად (ერთდროულად) შესრულებადი Sequence ქმედებათა რაოდენობა (ნახ.6.27).



ნახ. 6.27. Parallel ქმედება

6.3. საგადასახადო დავების განხილვის სისტემის პროგრამული რეალიზაცია Workflow Foundation ტექნოლოგიით

განიხილება საბაჟო სამართალდარღვევის ბიზნეს-პროცესების აქტიურობათა დიაგრამის მოდელირების საკითხები UML-ენის ბაზაზე და მისი შემდგომი პროგრამული რეალიზაციის საშუალებები ახალი ტექნოლოგიების საფუძველზე. შემოთავაზებულია Workflow Foundation ტექნოლოგიის გამოყენება ბიზნესპროცესების და ბიზნესწესების ვიზუალური დაპროგრამებისთვის.

შემოსავლების სამსახურის აუდიტის და საბაჟო დეპარტამენტების ფუნქციებიდან ერთ-ერთი მნიშვნელოვანი საკითხია საგადასახადო სამართალდარღვევების ბიზნეს-პროცესების მართვა, კერძოდ, მათი გამოვლენა და საგადასახადო დავის წარმოება [75]. იმისდა მიხედვით თუ მომჩივანის მიერ საჩივარი დავის განმხილველ რომელ ორგანოში შეიტანება, საგადასახადო სამართალდარღვევის ოქმი და თანდართული მასალები შეიძლება შემოსავლების სამსახურის გარდა ფინანსთა სამინისტროს დავების განხილვის საბჭომ ან სასამართლომ განიხილოს [76].

საგადასახადო სამართალდარღვევების ბიზნეს-პროცესი, რომელიც მოქალაქეთა საჩივრების სადავო საკითხების განხილვა-წარმოებას ეხება შემდეგი ბიზნეს-წესებით იმართება:

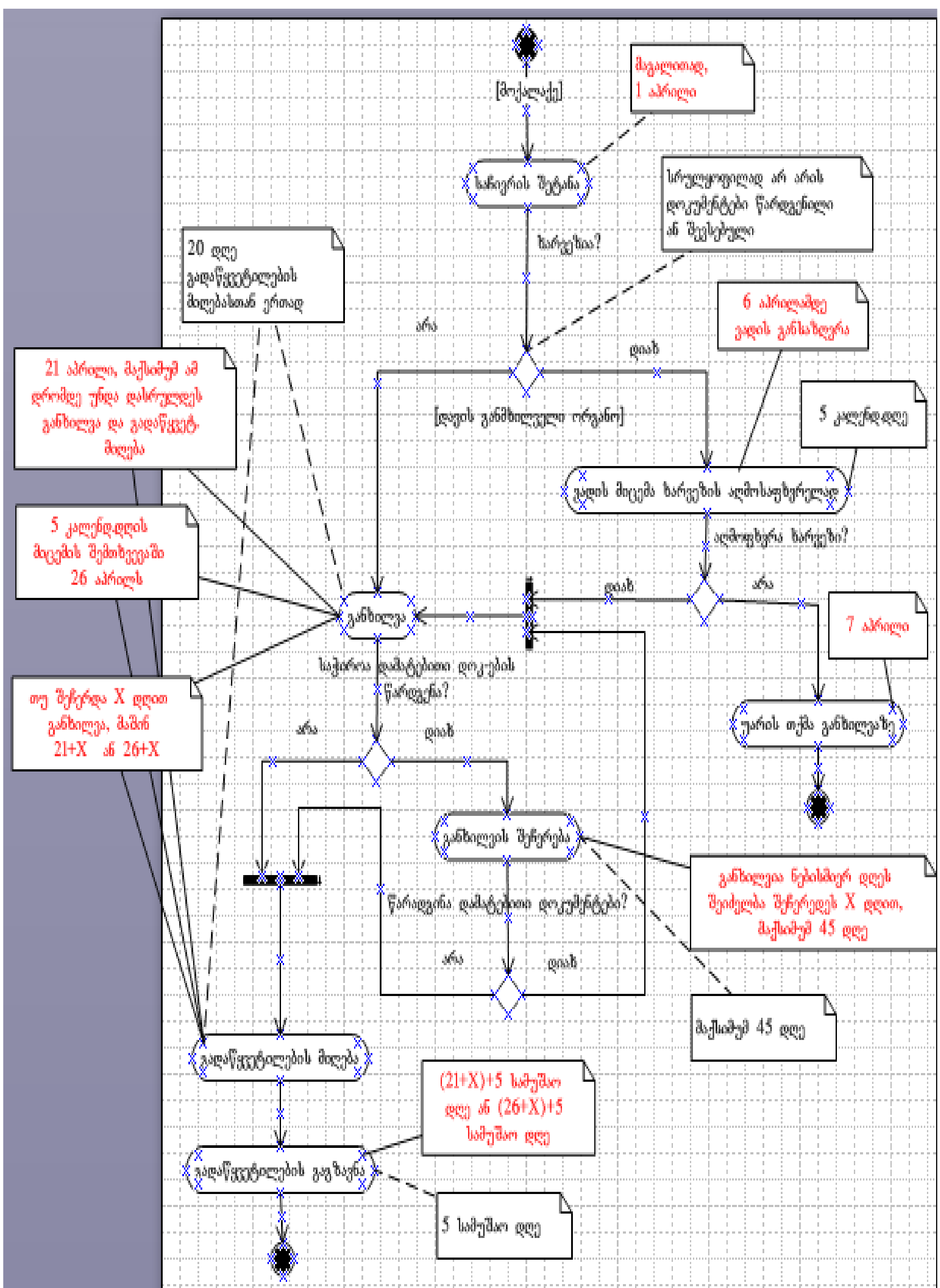
- მოქალაქეს შეაქვს საჩივარი დავის განმხილველ ორგანოში (ათვლის თარიღი ფიქსირდება);
- თუ საჩივარი არ აკმაყოფილებს პროცედურალ მოთხოვნებს, მომჩივანს წერილობით ეცნობება ამის შესახებ და მიეცემა არანაკლებ 5 დღე საჩივარში არსებული ხარვეზის გამოსაწორებლად;
- დავის განმხილვევლ ორგანოს უფლება აქვს მომჩივანის მოტივირებული მოთხოვნის დროს გააგრძელოს ხარვეზის გამოსასწორებლად მიცემული ვადა;
- დავის განმხილველი ორგანო საჩივარს განიხილავს 20 დღის ვადაში;
- დამატებითი ინფორმაციის ან/და დოკუმენტაციის მოპოვების საფუძვლით საცივრის განხილვის შეჩერების საერთო ხანგრძლივობა არ უნდა აღემატებოდეს 45 დღეს;
- დავის განმხილველი ორგანოს გადაწყვეტილების დამოწმებული ასლი, მისი მიღებიდან 5 სამუშაო დღის ვადაში ეგზავნება მხარეებს.

ნაშრომში განიხილება აღნიშნული ამოცანების მოდელირების, დაპროექტების და პროგრამული რეალიზაციის საკითხები უნიფიცირებული მოდელირების ენის (UML) და ახალი Workflow Foundation ტექნოლოგიის გამოყენებით [1,77].

Workflow Foundation ტექნოლოგია .NET Framework 5.0/5.5 -ში არის სრულიად ახალი პარადიგმა სამუშაო პროცესებზე (workflow) ბაზირებული აპლიკაციების ასაგებად. იგი ფუნდამენტურად ახლად გააზრებული ტექნოლოგიაა. აქ ჩვენ განვიხილავთ კონკრეტული ამოცანის, კერძოდ საბაჟო სამართალდარღვევების სამუშაო პროცესის აქტიურობათა დიაგრამიდან პროგრამული კოდის დაპროექტების პროცესს.

6.28 ნახაზზე ნაჩვენებია სამართალდარღვევის სადავო საკითხების ბიზნეს-პროსეცების და ბიზნეს-წესების აქტიურობათა დიაგრამა, აგებული MsVisio ინსტრუმენტის გამოყენებით. სქემაზე გარდა ბიზნეს-პროცესების და ცალკეულ ქმედებათა შესრულების მიმდევრობის აღწერისა, ნაჩვენებია ბიზნეს-წესების კომენტარებიც, რომლებშიც კარგად ჩანს საქმის წარმოების კანონით არსებული ვადების დროითი რეგლამენტი.

ახლა განვიხილოთ აგებული აქტიურობის დიაგრამის ასახვა Workflow Foundation ტექნოლოგიის სამუშაო გარემოში [25]. ესაა ჰიბრიდული (Windows+Web) სისტემა, რომლის კონსტრუირება (დიზაინი) ხდება შესაბამისი Workflow – ინსტრუმენტის ვიზაულური ელემენტებით (XAML ენაზე), ხოლო პროგრამის მუშაობის ლოგიკა, ჩვენს შემთხვევაში C#.NET კოდით [1].



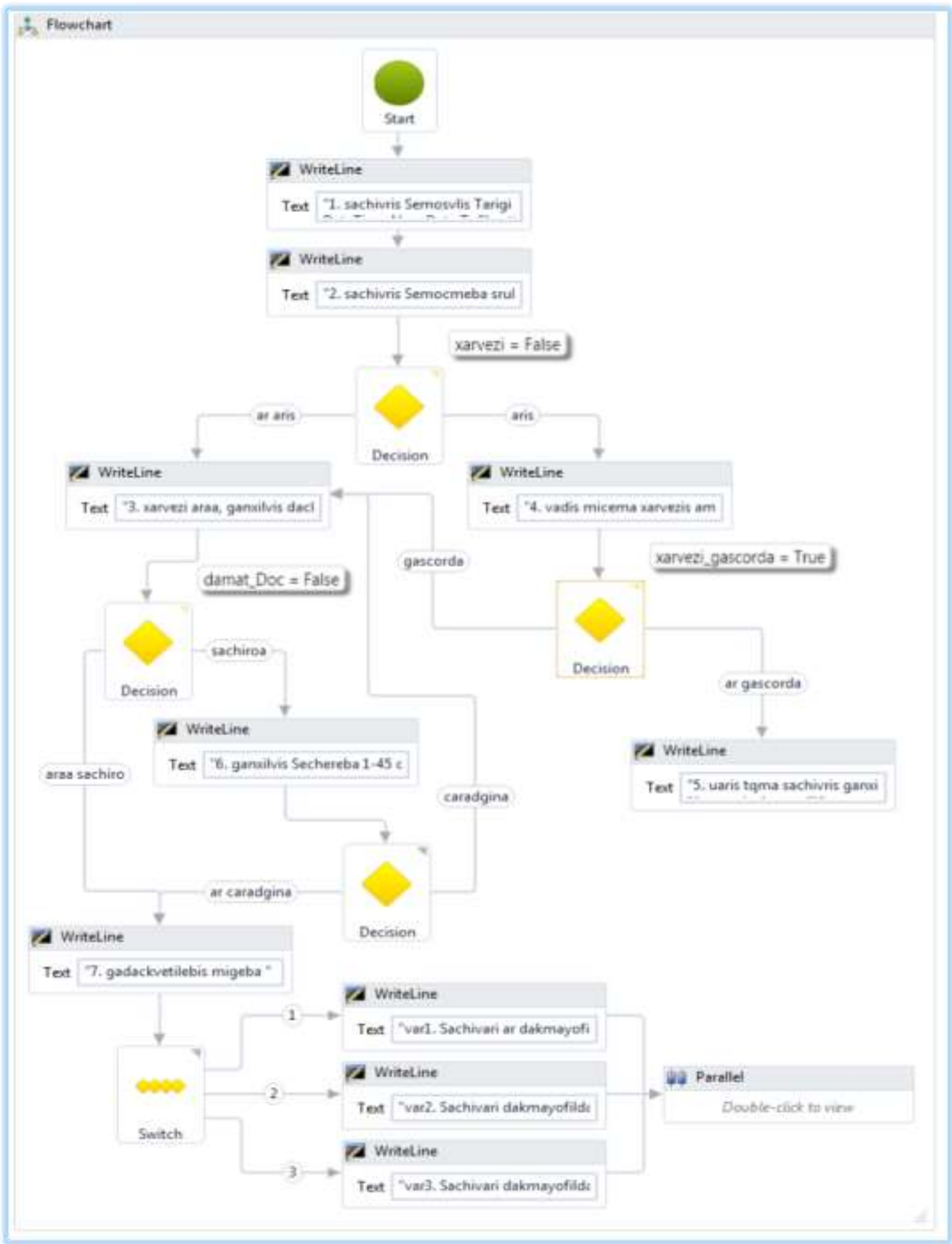
ნახ. 6.29. Activity-დიაგრამა MsVisio -ს გარემოში

6.29 ნახაზზე ნაჩვენებია დასმული ამოცანის გადაწყვეტის ალგორითმის ერთი ვარიანტი Workflow-ინსტრუმენტით. სქემაზე განთავსებულია შემდეგი ქმედებები (აქტიურობები): Flowchart, FlowDecission, If, Switch, Sequence, Assign და სხვ. [25].

ბიზნესპროცესის დიაგრამა (Flowchart Workflow) გამოიყენებს აქტიურობათა დიაგრამას. ქმედებათა ეს დიაგრამა მუშაობს როგორც ბლოკ–სქემა, ქმედებათა სახეები დაკავშირებულია ერთმანეთთან გადაწყვეტილების ხეებით (decision trees). ქმედებათა მიმდევრობითობის გამოყენებით, შვილი–პროცესები სრულდება მიმდევრობით ზემოდან-ქვემოთ (top-down). ამისდა მიუხედავად, შვილი-ქმედებები შეიძლება შესრულდეს ნებისმიერი მიმდევრობით, გადაწყვეტილების მიმღები პირის მიერ. ძირითადი განსხვავება Flowchart ქმედებასა და Sequence ქმედებას შორის ისაა, თუ როგორაა დაკავშირებული შვილი-აქტიურობები. ქმედებათა დამატებისას მიმდევრობითობაში ისინი სრულდება დაღმავალი (top-down) რიგითობით.

აქ შესაძლებელია მიმდევრობის კონტროლი (მართვა), ქმედებათა გადაადგილებით, ოღონდაც ისინი ყოველთვის გასწორებულია ვერტიკალში და განაწილებულია თანაბრად. ისრები ქმედებებს შორის კი აგებულია ავტომატურად. Flowchart აქტიურობით შესაძლებელია ქმედებათა განთავსება პალიტრის ნებისმიერ ადგილას. მნიშვნელოვანია ისიც, რომ აქ ისრები ხელით უნდა გაკეთდეს და შეერთება დასაშვებია უკან, წინა ქმედებასთანაც.

გადაწყვეტილების ნაკადის C ქმედება სქემაზე გამოიყურება „ყვითელი ალმასის“ სახით, როგორც განშტოების სიმბოლო ნორმალურ ბლოკ–სქემაში, Properties-ფანჯარაში შეიტანება მდგომარეობა (Condition), მაგალითად, “ხარვეზი = False”. მაუსის კურსორის მიტანით FlowDecision ქმედებაზე გამოჩნდება ეს წარწერა.

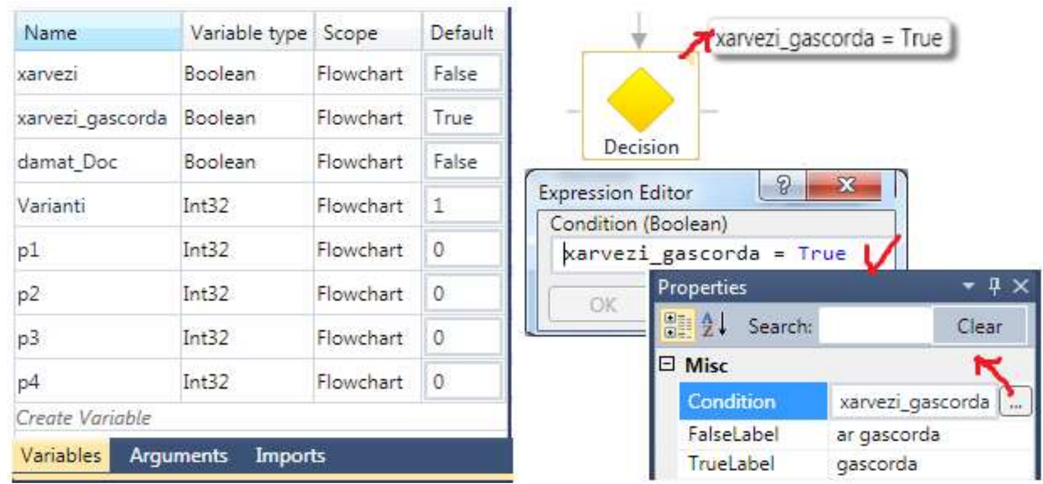


ნახ. 6.29. Workflow -დიაგრამის ფრაგმენტი Visual Studio .NET Framework 5.0 გარემოში

WF 5.0 აქვს რიგი პროცედურული ელემენტების: If, While, Assign, Sequence და სხვა. მაგალითად, ხარვეზების გასწორების პროცესის შედეგი განშტოვდება „გასწორდა“ ან „არ გასწორდა“:

```
if(xarvezi_gascorda = True) t2=t1+5;
    else Console.WriteLine("Sachivari ar ganixileba");
```

6.30 ნახაზზე ნაჩვენებია ცვლადების ცხრილი ჩვენი ალგორითმისთვის, ხოლო 6.31 ნახაზზე პირობის FlowDecision ქმედებაში მდგომარეობის (Condition) მნიშვნელობის განსაზღვრა.



Name	Variable type	Scope	Default
xarvezi	Boolean	Flowchart	False
xarvezi_gascorda	Boolean	Flowchart	True
damat_Doc	Boolean	Flowchart	False
Varianti	Int32	Flowchart	1
p1	Int32	Flowchart	0
p2	Int32	Flowchart	0
p3	Int32	Flowchart	0
p4	Int32	Flowchart	0
Create Variable			

ნახ. 6.30. Workflow -ცვლადების აღწერა

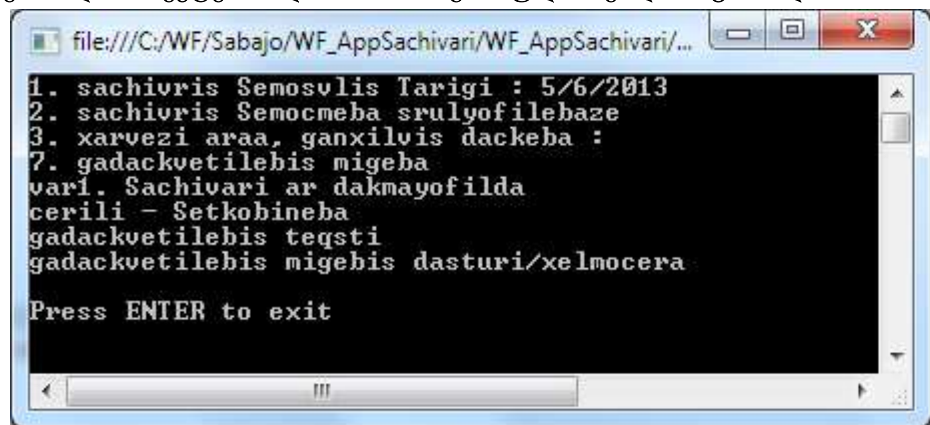
ნახ. 6.31. FlowDecision -ის პირობის განსაზღვრა

Switch ქმედება მუშაობს ისე, როგორც Switch ოპერატორი C# ენაში. ის საშუალებას იძლევა შესრულდეს sequence ქმედებები გამოსახულებათა ანალიზის საფუძველზე. Switch აქტიურობა გამოიყენება გადაწყვეტილების მიღების ვარიანტების განსაზღვრისთვისაც.

Workflow სქემის აგების ამ ეტაპზე კონსოლის აპლიკაციის ამუშავებით მიიღება 6.32 ნახაზზე მოცემული შედეგები, რომელთა შემდგომი დაზუსტება და გაფართოებაა შესაძლებელი.

Workflow Foundation ტექნოლოგიის გამოყენებით ბიზნეს-პროცესების და ბიზნეს-წესების დაპროგრამება ხორციელდება ეფექტურად, შესაბამისი ვიზუალური კომპონენტების საფუძველზე. შედეგად საგრძნობლად მცირდება

სისტემის დაპროექტების და მისი პროგრამული რეალიზაციის დრო.



ნახ. 6.32. პროგრამის მუშაობის შედეგები Workflow-პროცესების მიმდევრობითი აღწერით

6.4. მონაცემთა ბაზის ფიზიკური სტრუქტურის ავტომატიზებული აგება DDL ფაილებით Ms SQL Server - პაკეტისთვის

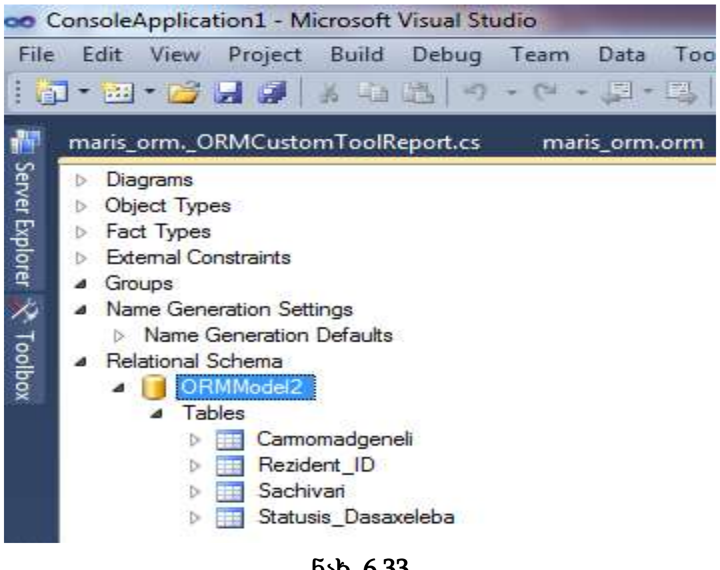
ჩვენი, მეხუთე თავში (5.5 პარაგრაფი) დაპროექტებული მონაცემთა ბაზის DDL-ფაილის ფრაგმენტი შემდეგნაირად გამოიყურება:

```
CREATE TABLE Sachivari
(
        sachivariID INT AUTO_INCREMENT NOT NULL,
        damateb_Doc_Ward_Tarigi DECIMAL(65,65) NOT NULL,
        damateb_Doc_Ward_VadaNr INT NOT NULL,
        documentebiNr INT NOT NULL,
        gadawyvet_Gagz_Tarigi DECIMAL(65,65) NOT NULL,
        ganxilvis_Tarigi DECIMAL(65,65) NOT NULL,
        registraciis_NomeriCode CHAR(63) NOT NULL,
        shetanaisTarigi DECIMAL(65,65) NOT NULL,
        xarvezis_Agmofx_Tarigi DECIMAL(65,65) NOT NULL,
        xarvezis_Agmofx_VadaNr INT NOT NULL,
        CONSTRAINT Sachivari_PK PRIMARY KEY(sachivariID)
);
CREATE TABLE Rezident_ID
(
        `value` INT AUTO_INCREMENT NOT NULL,
```

```
        CONSTRAINT Rezident_ID_PK PRIMARY KEY(`value`)
);
CREATE TABLE Statusis_Dasaxeleba
(
        statusis_DasaxelebaName VARCHAR(16383) NOT NULL,
        statusiID INT AUTO_INCREMENT NOT NULL,
        CONSTRAINT Statusis_Dasaxeleba_PK PRIMARY KEY(statusis_DasaxelebaName),
        CONSTRAINT Statusis_Dasaxeleba_UC UNIQUE(statusiID)
);

CREATE TABLE Carmomadgeneli
(
        carmomadgeneliNr INT NOT NULL,
        sachivariID INT NOT NULL,
        CONSTRAINT Carmomadgeneli_PK PRIMARY KEY(carmomadgeneliNr)
);
    ALTER TABLE Carmomadgeneli ADD CONSTRAINT Carmomadgeneli_FK FOREIGN
KEY (sachivariID) REFERENCES Sachivari (sachivariID) ON DELETE RESTRICT ON UPDATE
RESTRICT;
```

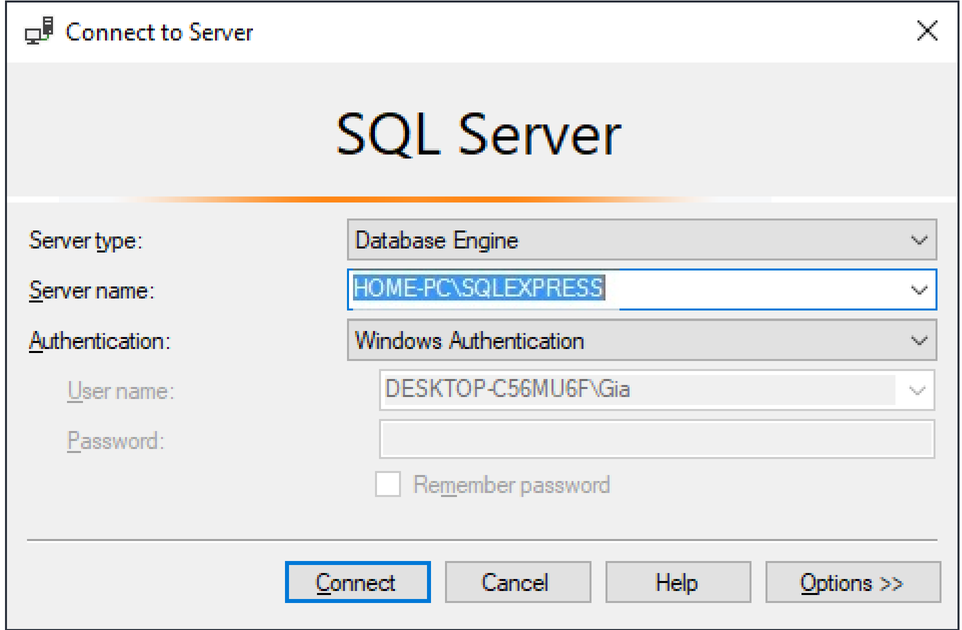
ჩვენი სისტემა Visual Studio.NET გარემოში Ms SQL Server ბაზის შექმნის შემდეგ ასე გამოიყურება (ნახ.6.33).



ნახ. 6.33

6.5. სისტემის ინფორმაციული ბაზა

სადემონსტრაციო სისტემის მონაცემთა ბაზა, რომელიც რეალიზებულია სერვერ-კლიენტის არქიტექტურის პრინციპით Microsoft SQL Server მონაცემთა მართვის სისტემის ბაზაზე, შედგება შემდეგი ძირითადი ცხრილებისგან: sachivari (საჩივარი), momchivani (მომჩივანი), qveyana (ქვეყანა), oqmi (ოქმი), wamomadg (წარმომადგენელი), statusi (სტატუსი), muxli (მუხლი), sfero (სფერო) (ნახ.6.34-6.42). 6.43 კი ნაჩვენებია მბ-ის დიაგრამა.



ნახ. 6.34. SQL server - თან დაკავშირება

მონაცემთა ბაზის ცხრილების სტრუქტურა:

HOME-PC\SQLEXPRE...vari - dbo.muxli | HOME-PC\SQLEXP...

Column Name	Data Type	Allow Nulls
IDmuxli	int	☐
muxlidasax	nvarchar(MAX)	☑
muxlinomi	nvarchar(50)	☑
		☐

ნახ. 6.35. ცხრილი muxli

HOME-PC\SQLEXP... dbo.momchivani | HOME-PC\SQLEXPR.

Column Name	Data Type	Allow Nulls
IDmomchivani	int	☐
IDstatusi	int	☐
IDwarm	int	☐
IDqveyana	int	☐
SK	nvarchar(50)	☑
saxelimomch	varchar(50)	☑
gvarimomch	varchar(50)	☑
PNmomch	nvarchar(50)	☑
Pasnommomch	nvarchar(50)	☑
rezidentobamomch	bit	☑
dasaxmomch	varchar(MAX)	☑
		☐

ნახ. 6.36. ცხრილი momchivani

HOME-PC\SQLEXPR...ivari - dbo.oqmi | HOME-PC\SQLEXPRE

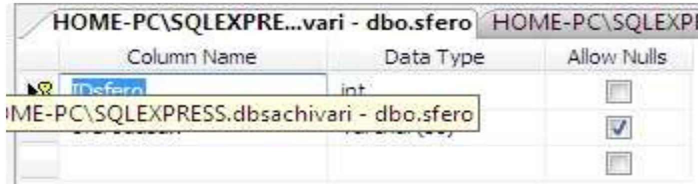
Column Name	Data Type	Allow Nulls
IDoqmi	int	☐
IDmuxli	int	☐
oqminom	nvarchar(50)	☑
		☐

ნახ. 6.37. ცხრილი oqmi

HOME-PC\SQLEXPRE...ri - dbo.qveyana | HOME-PC\SQLEXP

Column Name	Data Type	Allow Nulls
IDqveyana	int	☐
dasaxqveyana	nvarchar(50)	☑
		☐

ნახ. 6.38. ცხრილი qveyana

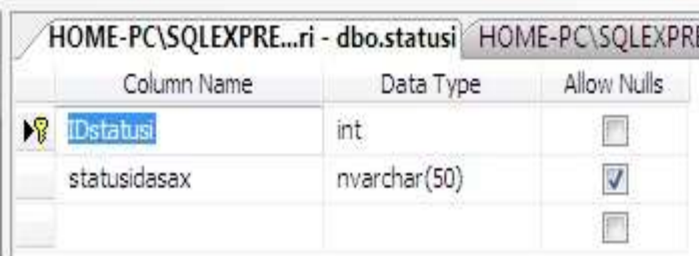


ნახ. 6.39. ცხრილი sfero

HOME-PC\SQLEXPRE... - dbo.sachivari*

Column Name	Data Type	Allow Nulls
IDsachivari	int	☐
IDmomchivani	int	☐
IDoqmi	int	☐
IDsfero	int	☐
shetanisTariRi	date	☑
xarvezisdro	int	☐
xarvezisTariRi	date	☑
dokebisdro	int	☑
dokebisTariRi	date	☑
ganxilvismaqsTariRi	date	☑
gadawyvgagzTariRi	date	☑
		☐

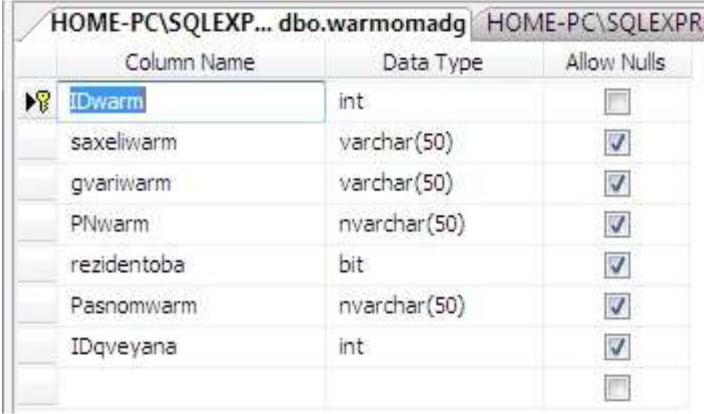
ნახ. 6.40. ცხრილი sachivari



HOME-PC\SQLEXPRE...ri - dbo.statusi | HOME-PC\SQLEXPRE

Column Name	Data Type	Allow Nulls
IDstatusi	int	☐
statusidasax	nvarchar(50)	☑
		☐

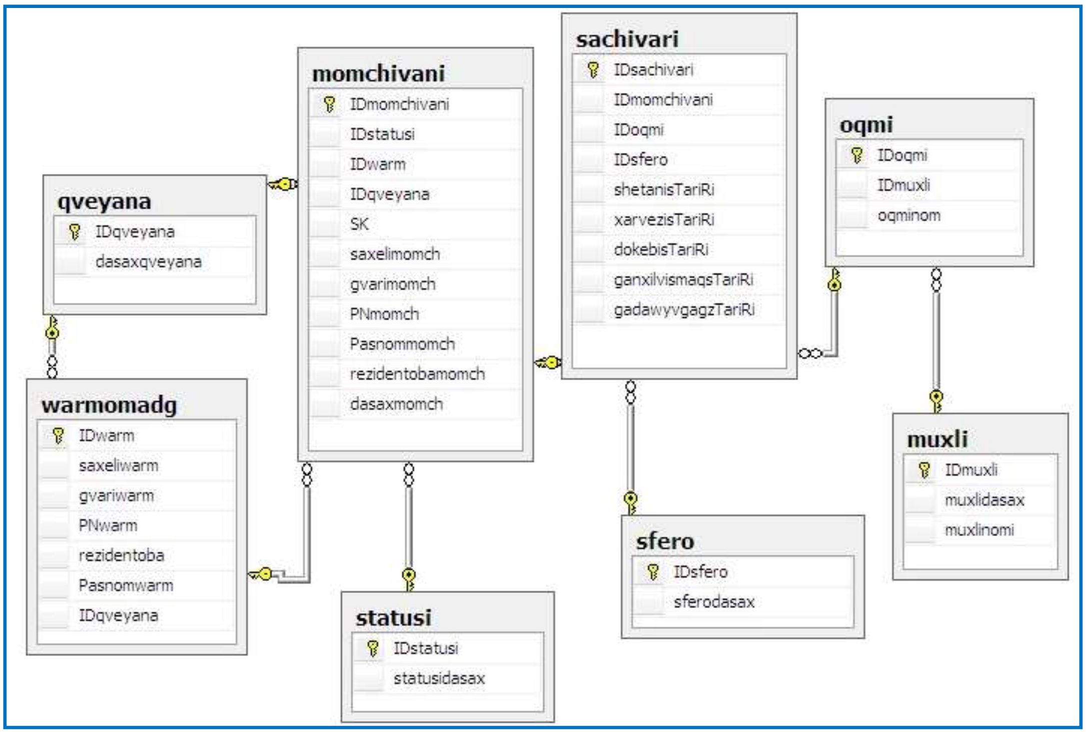
ნახ. 6.41. ცხრილი statusi



HOME-PC\SQLEXP... dbo.warmomadg | HOME-PC\SQLEXPRI

Column Name	Data Type	Allow Nulls
IDwarm	int	☐
saxeliwarm	varchar(50)	☑
gvariwarm	varchar(50)	☑
PNwarm	nvarchar(50)	☑
rezidentoba	bit	☑
Pasnomwarm	nvarchar(50)	☑
IDqveyana	int	☑
		☐

ნახ. 6.42. ცხრილი Tsarmomadg



ნახ. 6.43. მონაცემთა ბაზის სტრუქტურა Relationship

6.6. მომხმარებელთა ინტერფეისების პროგრამული რეალიზაცია

.NET პლატფორმის Windows Application გარემოში მუშაობის აღწერისათვის ვიხილავთ Visual Studio.NET-პლატფორმის visual C# პროგრამირების ენის პაკეტს. იგი ერთ-ერთი ყველაზე მძლავრი ინსტრუმენტია, რომელიც კლასთა ასოციაციის დიაგრამის საფუძველზე ავტომატიზებულად აგებს პროგრამულ კოდს. აქ რეალიზებულია რევერსიული პროგრამირების მეთოდი, ანუ კლასის მოდელიდან მიიღება კოდი და პირიქით, კოდის ცვლილებიდან შესაბამისად კორექტირდება კლასიოს მოდელი (იხ. პარაგრაფი 2.2.3).

თანამედროვე სიანფორმაციო ტექნოლოგიების განვითარება შესაძლებლობას გვაძლევს ვაწარმოოთ ნებისმიერი რთული სტრუქტურული ობიექტების

ავტომატიზაცია, როგორც ლოკალური ქსელის ფარგლებში, ისე გლობალური ქსელისთვის და WAP -დანართისთვის.

ერთ-ერთ ასეთ მძლავრ თანამედროვე ტექნოლოგიას წარმოდგენს Microsoft .NET კონცეფცია, Microsoft პლატფორმის სივრცეში, რომელიც ბაზირებულია სერვის-ორიენტირებულ მიდგომაზე, სრულყოფილს ხდის მონაცემებთან წვდომის ტექნოლოგიას ADO.NET დრაივერის საშუალებით და მოიცავს. .NET Framework სისტემას – ინსტრუმენტალური საშუალებით Visual Studio .NET, კორპორაციულ .NET სერვერებს, უნიფიცირებული სტანდარტის აგების სერვისებს და ა.შ. [9,57]. .NET Framework განსაზღვრავს გარემოს, რომელიც უზრუნველყოფს პლატფორმაზე დამოუკიდებელი პროგრამა-დანართების (Application) შემუშავებასა და შესრულებას. იგი, აგრეთვე, უზრუნველყოფს პროგრამების გადატანადობას. შედეგად, Windows - პროგრამები შეგვიძლია სხვა პლატფორმებზე ვამუშავოთ.

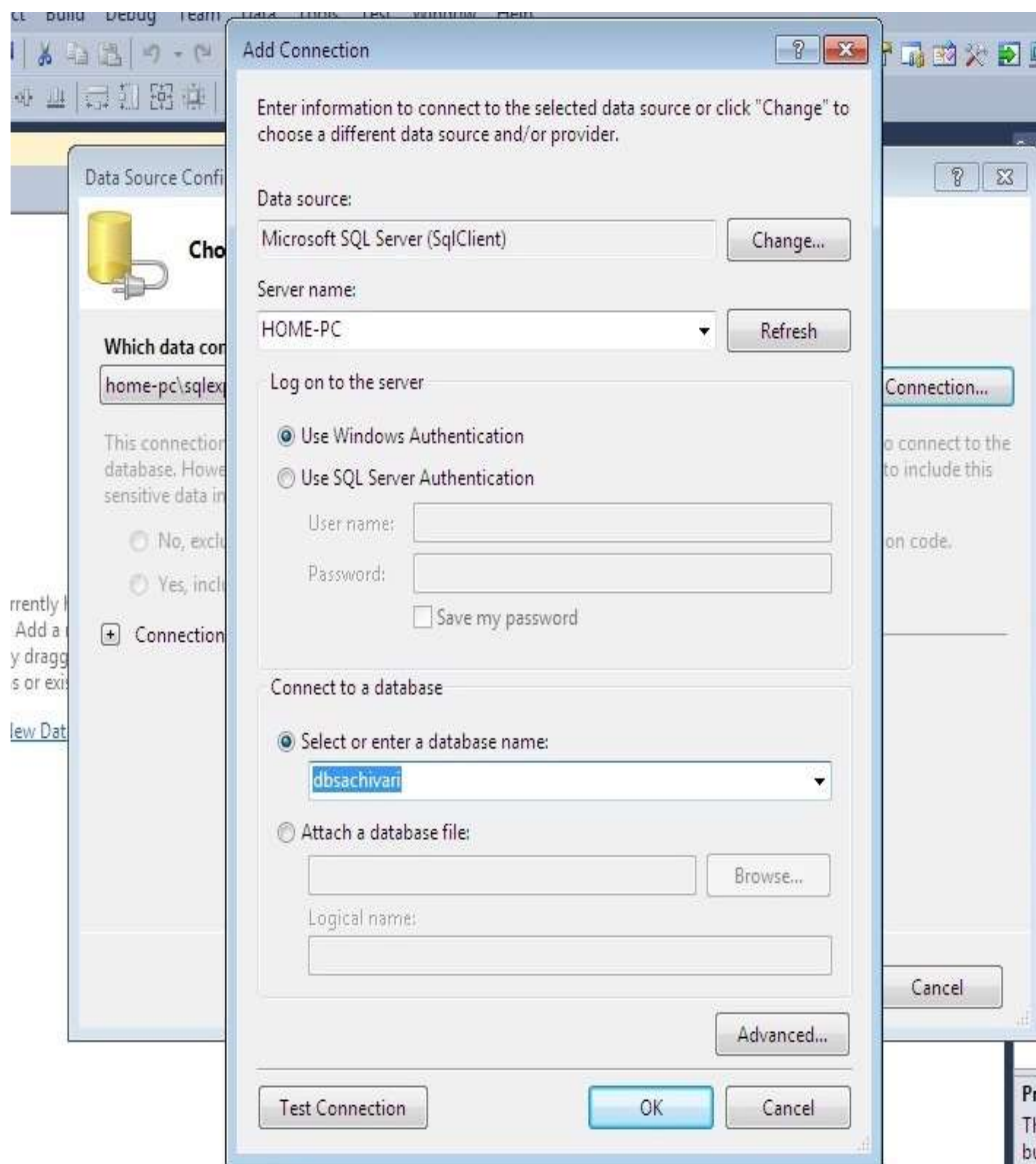
Visual Studio .NET ინსტრუმენტი ADO.NET დრაივერის მონაცემთა კომპონენტები ახდენენ მონაცემებთან წვდომის ფუნქციონალური შესაძლებლობების ინკაფსულირების სხვადასხვა ხერხებით. მაგალითად, SQL ინსტრუქციები, შენახვადი პროცედურები და აშ

სისტემა დამუშავებულია .NET -პლატფორმაზე, C# და SQL Server ობიექტ-ორიენტირებული ინსტრუმენტების გამოყენებით. მონაცემების მართვა, მონაცემთა ბაზის სისტემის გამოყენებით საჭიროებს მონაცემთა წყაროს (DataSource) შექმნას მართვის ელემენტებით - DataSet, BindingSource, TableAdapter (მონაცემთა მიმართვის ADO.NET ტექნოლოგია).

DatagridView კომპონენტის გრაფიკულ ინტერფეისზე გადმოტანისას, ჩნდება მონაცემთა წყაროს არჩევის დიალოგი. მონაცემთა წყაროს შექმნა ანუ მონაცემთა ბაზასთან დაკავშირება წარმოებს ბმულით - Add Project Data Source. შედეგად ჩნდება მონაცემთა წყაროს კონფიგურაციის დიალოგი. ვირჩევთ მონაცემთა ბაზას, ღილაკით “OK” გადავდივართ მონაცემთა ბაზის დაკავშირების/არჩევის (როცა უკვე შექმნილია) დიალოგზე (ნახ.6.44).

კავშირის წარმატებით განხორცილების შედეგად, შესაძლებელია მითითებული მონაცმეთა ბაზის ობიექტის (ცხრილების, შენახვადი პროცედურების და სხვ.) არჩევა (ნახ.6.45).

მოცემულია SQL ბაზებთან წვდომა Visual Studio .NET ინსტრუმენტის საშუალებით.

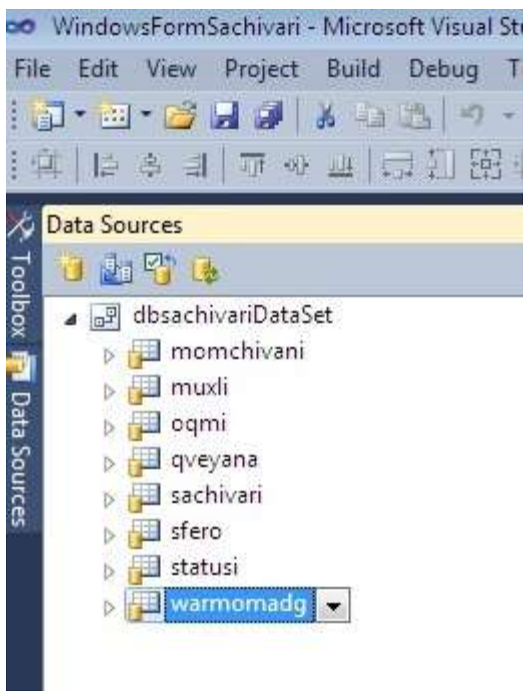


ნახ. 6.44. სერვერისა და შექმნილი მონაცემთა ბაზის არჩევა

ნახ. 6.45. მონაცემთა ბაზის არჩევა

ნახ. 6.46. ბაზასთან კავშირი

ნახ. 6.47. მონაცემთა ბაზის ობიექტის არჩევის დიალოგი



ნახ. 6.48. მონაცემთა ბაზის ცხრილის მიბმა

ორგანიზაციული მართვის სისტემის ავტომატიზაციისთვის, უმეტესად ერთობლივად გამოიყენება პროგრამული და ინფორმაციული უზრუნველყოფა. ამ შემთხვევაში მონაცემთა ბაზების მართვის სისტემა ერთგვარ ბირთვს წარმოადგენს და გამოიყენება როგორც სისტემის ინფორმაციული მართვის სერვერი.

მონაცემთა წყაროს კონფიგურაციის და ბაზასთან წვდომის უზრუნველ-ყოფის შემდგომ შემუშავებულია ინტერფეისები C#.NET პროგრამირების ენის საშუალებით, რომლებიც საგადასახადო დავის სისტემაში შემოსულ საჩივრებთან დაკავშირებული მონაცემების მართვის საშუალებას იძლევა.

დიდი სისტემების დამუშავებისას მონაცემთა ბაზის ცხრილებთან ურთიერთობისთვის რეკომენდებულია SQL-ის შენახვადი პროცედურების გამოყენება და მონაცემთა წყაროსთან კავშირის ერთჯერადი შექმნა და კონფიგურაციის ზოგადი აღწერა.

მონაცემთა ბაზაში ჩაწერისა და ინფორმაციის სამომხმარებლო ეკრანზე გამოსახვისას ხშირია მონაცემთა გაერთიანებისა და დაყოფის (Parsing) ოპერაციის გამოყენება. მაგალითად, ბაზაში სახელის და გვარის ჩაწერა სხვადასხვა ველშია რეკომენდირებული, ხოლო დიალოგზე ხშირ შემთხვევაში იწერება ერთ სტრიქონში.

მონაცემთა სიმბოლოთა გაერთიანებას ეწოდება კონკატენაცია და წარმოებს სიმბოლოთი “+”, ხოლო დაყოფის ოპერაციების შესრულება ხდება ოპერატორით Split, რომელიც აბრუნებს სტრიქონული ტიპის მასივს სინტაქსით:

String[] სახელი=სტრიქონი. Split(‘სიტყვების გამყოფი სიმბოლო’).

მაგალითდ, saxeli ველში იწერება - სახელი, gvari ველში - გვარი, ხოლო saxeligvari-ში საჭიროა დავწეროთ გვარი და სახელი ჰარის გამოტოვებით:

```
saxeligvari.Text = saxeli.Text + " " + gvari.Text;
```

შემდეგ, saxeligvari-ში მიღებული სტრიქონი დავშალოთ და გადავცეთ saxeli-ს:

```
saxeligvari.Text = saxeli.Text + " " + gvari.Text;
string[] words=saxeligvari.Text.Split('');
    foreach(string word in words)
      {
        listBox4.Items.Add(word);
      }
```

თარიღის კომპონენტის გამოსახვისათვის სასურველი კოდირების ფორმატში და თარიღის შემცველი ცვლადების მიღებისათვის სტრიქონულ ფორმატში სასურველ ენაზე, გამოყენებულია Windows ოპერაციული სისიტემის რეგიონალურ პარამეტრებთან წვდომის სისტემური კლასი Globalization და მრავალნაკადურ რეჟიმებთან (პარალელური პროგამირება) მომუშავე სისტემური კლასი Threading.

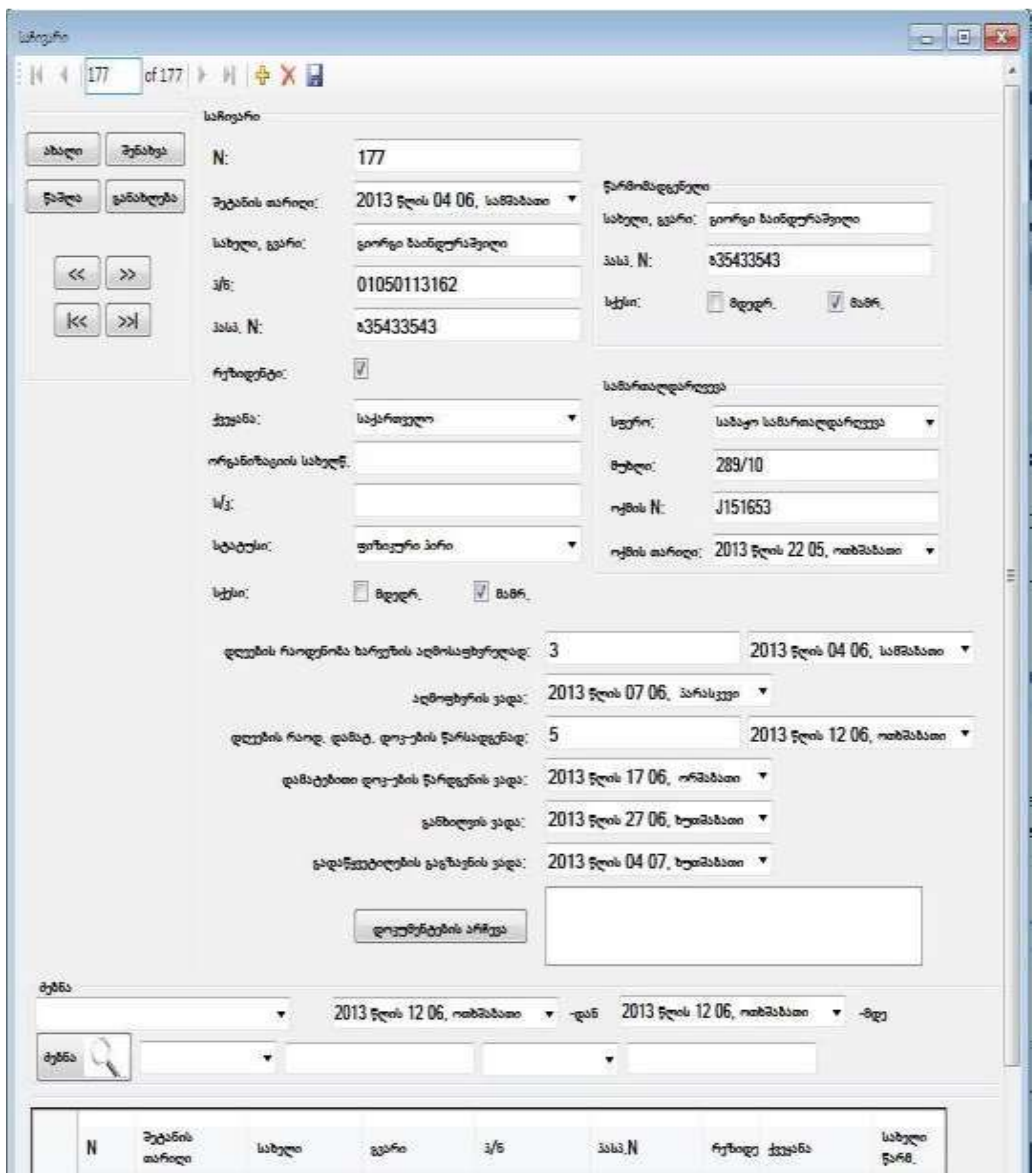
პროგრამული სისტემის დამუშავებისას გამოყენებულია თარიღებს შორის პერიოდის დათვლის და თარიღების შედარების ოპერაციები. ამ თვალსაზრისით, DateTimePicker კლასის მეთოდებში, გათვალისწინებულია თრიღის შემცველი ელემენტების დაშლა თარიღის ცვლადების მიხედვით და თითოეული ცვალდის მნიშვნელობის მიღება/გადაცემა, DateTimePicker კლასის ობიექტზე value ოპერატორის გამოყენებით. ყველა მითითებული ოპერატორი აბრუნებს რიცხვითი int ტიპის მნიშვნელობას.

ინტერფეისს, რომელიც საგადასახადო საჩივრებთან დაკავშირებული მონაცემების მართვას უზრუნველყოფს მოცემულია 6.49 ნახაზზე.

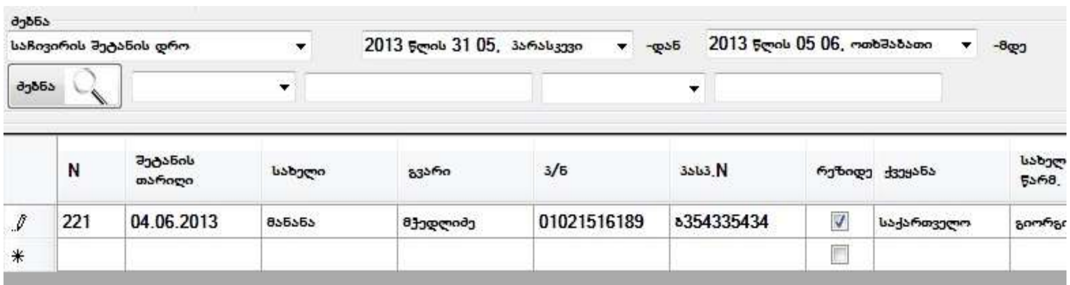
მოცემული სისტემა იძლევა, როგორც მონაცემთა შენახვის, ასევე შენახული მონაცემების გარკვეული კრიტერიუმებით ძებნის და მისი რედაქტირების საშუალებას (ნახ.6.50).

იგი ნაშრომის მეორე თავში აღწერილი და მოდელირებული ბიზნეს-პროცესების მიხედვით, რომელიც საგადასახდო დავის წარმოებას ეხება, მიწოდებული ვადებისა და თარიღების მიხედვით ითვლის და გვიჩვენებს საჩივრის საგადასახდო ორგანოში შეტანისას დოკუმენტებთან დაკავშირებული ხარვეზების აღმოფხვრის და ორგანოს მოთხოვნის შემთხვევაში, დამატებითი დოკუმენტების წარდგენის ვადებს.

ასევე გვიჩვენებს საჩივრის განხილვის და გადაწყვეტილების მომჩივანისთვის გაგზავნის საბოლოო თარიღს.



ნახ. 6.49. მომხმარებლის ინტერფეისის ფორმა



ნახ. 6.50

ნახ. 6.51

ნახ. 6.52

6.7. ინფორმაციის კომბინირებული დამუშავების მეთოდი

კომბინირებული დამუშავების მეთოდით, რომელიც წინამდებარე პარაგრაფში გვაქვს დამუშავებული, საშუალებას იძლევა ორგანიზაციაში, მათ შორის შემოსავლების სამსახურის საგადასახადო დავების დეპარტამენტში რეგისტრირებული ინფორმაცია დაამუშავოს ეფექტურად, მიიღოს სტატისტიკური ანალიზის შედეგები ცხრილებისა და გრაფიკების (დიაგრამების) სახით [65]. კომბინირებული დამუშავების მეთოდი შედგება რამდენიმე ეტაპებისგან, რომელთა დანიშნულება და ფუნქციონალობა მოცემულია ქვემოთ.

- მომხმარებლის მიერ არჩეული კრიტერიუმების გენერირება სპეციალურ ფორმატში;

- კორპორაციაში არსებული ტრანზაქციების დაჯგუფება სახეობებისა და ტიპების მიხედვით;

- ინფორმაციის გენერირება და მისი ასახვა დროებით ცხრილში;

- ინფორმაციის გამოსახვა ცხრილებისა და გრაფიკების (დიაგრამების) საშუალებით.

საჩივრების სტატისტიკური ანალიზის შედეგების მისაღებად შექმნილია სპეციალური ფორმა, რომელიც დაყოფილია სხვადასხვა სექციად. ყველა სექციაში მომხმარებელმა აუცილებლად უნდა აირჩიოს სხვადასხვა კრიტერიუმი. პირველ სექციაში მომხმარებელი ირჩევს მიმდინარე პერიოდს, თუ საიდან სადამდე სურს მას სტატისტიკური ანალიზის ჩატარება. არჩეულ პერიოდს სისტემა აგენერირებს სპეციალურ ფორმატში. ამისათვის შემოვიტანოთ შემდეგი აღვნიშვნები: Date1 - ცვლადი, რომელსაც მიენიჭება პერიოდის საწყისი მნიშვნელობა, Date2 - ცვლადი, რომელსაც მიენიჭება პერიოდის საბოლოო მნიშვნელობა, tve1 და tve2 - ცვლადები, რომლებშიც იწერება არჩეული თვის რიცხობრივი მნიშვნელობები. P - ცვლადი, რომელშიც აისახება მიმდინარე პერიოდი (Date1 ≤ P ≤ Date2).

არჩეული პერიოდის გენერირება ხდება შემდეგნაირად. მაგალითად, მომხმარებელმა აირჩია – "იანვარი, 2024"-დან "ივლისი, 2024"-მდე, მაშინ:

Date1='01' + tve1 + '2024'='04.04.2024'; Date2='07' + tve2 + '2024'='14.08.2024';

მეორე სექციაში მომხმარებელი ირჩევს ტრანზაქციის სახეობას და ტიპს. შემოვიტანოთ შემდეგი აღვნიშვნები: Sj _ ცვლადი, რომელშიც ფიქსირდება არჩეული სახეობა. სადაც j არის სახეობების ინდექსი და j=4..n. სადაც n არის სახეობების რაოდენობა.

სისტემაში შექმნილია სპეციალური კომპონენტები, რომლის საშუალე-ბითაც ხდება ინფორმაციის გამოსახვა Excel-ში გრაფიკების (დიაგრამების) სახით, სადაც მომხმარებლის მიერ დაკონკრეტებული მიმდინარე პერიოდის, ტრანზაქციის სახეობისა და ტიპის მიხედვით მიიღება სტატისტიკური ანალიზის ფორმები.

მეორე ნაწილის დასკვნა

მონოგრაფიის ფარგლებში ჩატარებული საპროექტო–კვლევითი სამუშაოების შედეგების საფუძველზე შესაძლებელია შემდეგი დასკვნების გაკეთება:

– საგადასახადო სფეროს კორპორაციული მენეჯმენტის სისტემებში ბიზნეს-პროცესების სისტემური ანალიზის, გამოვლენილი პრობლემების და მათი გადაწყვეტის მიზნით განხორციელდა საინფორმაციო სისტემების დაპროექტების და მოდელირების თანამედროვე მეთოდების და CASE-ინსტრუმენტული საშუალებების ანალიზი, მათი კლასიფიკაცია ბიზნესპროცესების ასახვის შესაძლებლობათა ევოლუციის თვალსაზრისით;

– UML/2 სრულიად განსხვავებული განზომილებაა უნიფიცირებული მოდელირების სამყაროში. წინა ვერსიებთან შედარებით დოკუმენტაციის სივრცე უფრო გაზრდილია, უფრო კომპლექსურია, დამატებულია ახალი შესაძლებლობები, რაც მომავლში ფართოდ გამოყენების საშუალებას იძლევა. იგი მოიცავს ფორმალურ და კონკრეტულ სემანტიკას. ეს ახალი ფუნქციონალობა შეიძლება გამოყენებულ იქნას მოდელის განვითარებისა და შემდეგ ამ მოდელის ბაზაზე შესაბამისი პროგრამული სისტემის შესაქმნელად;

– საგადასახადო დავების სისტემის საპრობლემო სფეროს ფუნქციონალურ და არაფუნქციონალურ მოთხოვნილებათა განსაზღვრა მისი ბიზნეს-პროცესების აღწერის მიზნით ეფექტურ შედეგებს იძლევა უნიფიცირებული მოდელირების ტექნოლოგიის შესაბამისი მძლავრი, მრავალმხრივი და მრავალპლატფორმიანი სისტემის – Sparx Systems Enterprise Architect ინსტრუმენტის გამოყენებით, რაც შემდგომში წარმატებით ვლინდება პროგრამული სისტემის განვითარების ეტაპზე .NET პლატფორმასთან თავსებადობის გამო;

– CASE-ინსტრუმენტების გამოყენებით კორპორაციული მენეჯმენტის საინფორმაციო სისტემების აგების პროცესში, მნიშვნელოვნად უმჯობესდება პროგრამული უზრუნველყოფის ხარისხი და საგრძნობლად მცირდება დაპროექტების, მისი იმპლემენტაციისა და რეინჟინერინგის პერიოდები. მონაცემთა ბაზების დაპროექტების და აგების პროცესების ავტომატიზაცია ობიექტ-როლური მოდელირებით, მისი შემდგომი რესტრუქტურიზაციის პრობლემების მოქნილად გადაწყვეტის საშუალებას იძლევა;

– კორპორაციული მენეჯმენტის სისტემების მოდელირებისა და ანალიზისთვის, განსაკუთრებთ კი რესურსების მართვის საკითხებში, რიგების თეორიის და მისი ინსტრუმენტული საშუალებების გამოყენება საინჟინრო ამოცანების გადასაწყვეტად, ამაღლებს ორგანიზაციის მომსახურების ეფექტურობას, ამცირებს დროითი და მატერიალური ხარჯებს.

ნაწილი 3
Agile Software Development და Agile Testing

Agile პროგრამული უზრუნველყოფის განვითარების პროექტებში განსაზღვრულ ვადებში მაღალი ხარისხის პროგრამული უზრუნველყოფის მიწოდებისთვის დროის ეფექტური მართვა გადამწყვეტი ფაქტორია. კვლევა ყურადღებას ამახვილებს დროის მართვის ხარვეზზე Agile განვითარების სპრინტის მეთოდში, მართვის საინფორმაციო სისტემის აგების დროს Agile მეთოდოლოგიის გამოყენებით. კვლევის მიზანია გამოავლინოს დროის მენეჯმენტთან დაკავშირებული გამოწვევები სპრინტების წარმართვის დროს და Agile გუნდებს შესთავაზოს გამოსავალი ამ საკითხების შესამსუბუქებლად.

არსებული ლიტერატურისა და ემპირიული კვლევების ანალიზით, ცხადი ხდება, რომ სპრინტზე დაფუძნებული სწრაფი განვითარების მიდგომები ხშირად აწყდებიან დროის მართვის გამოწვევებს, განსაკუთრებით ტესტირების ფაზაში. ტესტირების პროცესში შეფერხებამ შეიძლება გამოიწვიოს პროგრამული ხარისხის გაუარესება და პროექტის დასრულების პოტენციური შეფერხება. ამ საკითხებს შეიძლება ჰქონდეს უარყოფითი გავლენა პროექტის საერთო წარმატებაზე.

ნაშრომის „Agile დეველოპმენტი და ტესტირების მეთოდების შემუშავება და კვლევა მართვის საინფორმაციო სისტემის ასაგებად“ ფარგლებში შექმნილი პროექტი და განხორციელებული კვლევა აჩვენებს, რომ დროის მენეჯმენტის ხარვეზების ერთ-ერთი ხელშემწყობი ფაქტორია Agile დეველოპმენტში ტესტირების პროცესის თანმიმდევრული ბუნება, სადაც UI ტესტირება და მოდულური ტესტირება, როგორც წესი, ტარდება სპრინტის ბოლოს. ეს მიდგომა კი ხშირად იწვევს არასაკმარის დროს საფუძვლიანი ტესტირებისთვის, დაჩქარებული ტესტირება კი აჩენს პოტენციური ხარისხის მოლოდინს.

ამ გამოწვევის დასაძლევად კვლევაში შემოთავაზებული ახალი მიდგომა გულისხმობს მოდულური ტესტების დაწყებას სპრინტის დასაწყისშივე. თავიდანვე სატესტო აქტივობების განხორციელებით, გუნდს შეუძლია დაუყოვნებლივ გამოავლინოს და გამოასწოროს პოტენციური პრობლემები განვითარების პროცესშივე, რაც უზრუნველყოფს პროგრამული უზრუნველ-ყოფის მაღალ ხარისხს და ამცირებს დროის დაკარგვის რისკებს. გარდა ამისა, დაინერგა ჰიბრიდული მეთოდოლოგია, რომელიც აერთიანებს *მოდულურ* ტესტებს *მუტანტის ტესტებთან*. მუტანტის ტესტების ჩართვა ერთეულ

ტესტებთან ერთად მიზნად ისახავს სპრინტის ციკლების დროს შემუშავებული პროგრამული უზრუნველყოფის ხარისხის გაუმჯობესებას. ახალი ჰიბრიდული მიდგომა ამლიერებს ტესტირების ციკლების ეფექტურობას, რაც საშუალებას იძლევა გაუმჯობესდეს პროგრამული უზრუნველყოფის საერთო ხარისხი.

წიგნის ამ ნაწილის მიზანია *განისაზღვროს Agile მეთოდოლოგიის სპრინტ მეთოდის ხარვეზზე ორიენტირებული პროგრამული უზრუნველყოფის შექმნის სქემა.* ნაშრომი კონცენტრირებულია პროცესებზე, რომლებმაც გამოიწვია სპრინტების ჩავარდნა და შესაბამისად ხელი შეუშალა ხარისხიანი პროდუქტის განსაზღვრულ ვადებში შემუშავებას. შემოთავაზებულია ხარვეზების აღმოფხვრის მეთოდები და ასეთი მეთოდების ფარგლებში შესწავლილია საინფორმაციო სისტემების პროექტირების პროცესი.

მართვის საინფორმაციო სისტემებში (სახელმწიფო სტრუქტურები, მაგალითად, სამინისტროები, კერძო სექტორი, ბანკები და სხვ.) პროგრამული უზრუნველყოფის განვითარების სასიცოცხლო ციკლის შესაბამისად, მიმდინარეობს მრავალი სახის პროცესი. ამ პროცესებიდან ყურადღება გამახვილებულია რამდენიმე მნიშვნელოვანზე. ესენია:

- კონკრეტული პროექტის ფარგლებში, გუნდის სამუშაო პროცესის შესწავლა ჰიბრიდული მეთოდოლოგიით მუშაობისას;
- Agile ტესტირება ინტერფეისის ტესტირების მეთოდით;
- Agile ტესტირება მოდულური (unit) ტესტების დახმარებით;
- ჰიბრიდული ტესტირება – Agile მეთოდოლოგიის ადაპტაცია მუტანტ ტესტებთან;
- საინფორმაციო სისტემის (ბონუსების) შექმნის ვადები ჰიბრიდული მეთოდოლოგიის ჭრილში;
- მნიშვნელოვანი გამოთვლების/ტრანზაქციების სისტემის ტესტირების ვადები ჰიბრიდული მეთოდოლოგიის ჭრილში.

კერძოდ, განიხილება:

1. მოდულურ ტესტირებაში კომპანიის მხრიდან დახარჯული რესურსები.
2. მუტანტ ტესტირებაში დახარჯული რესურსები.

კონკრეტულად ეს პროცესი არჩეულ იქნა იმიტომ, რომ თანამედროვე მსოფლიოში ციფრული ტექნოლოგიები უპრეცენდენტოდ სწრაფად ვითარდება.

სხვადასხვა ტიპის ბიზნესი ციფრულდება და შესაბამისად საჭიროებს გარკვეული ტიპის დეველოპმენტს, ასეთ შემთხვევებში კი როგორც

დამკვეთისთვის, ასევე გუნდისათვის უმნიშვნელოვანესია შეთანხმებით განსაზღვრულ ვადებში ხარისხიანი პროგრამული უზრუნველყოფის შექმნა და მიწოდება. ნაშრომში წარმოდგენილი ჰიბრიდული მეთოდოლოგიის სწორად გამოყენებით შესაძლებელია მრავალი პროცესის გამარტივება და დროის ნაკლები რესურსის დახარჯვა.

ნაშრომში წარმოდგენილია „სუფთა არქიტექტურაზე“ შესრულებული პროექტის კვლევა, რომელშიც დამხმარე ტექნოლოგიებად გამოყენებულია CQRS (Command query responsibility segregation) და Mediatr design Pattern. ტესტირების ნაწილში ინოვაციური ჰიბრიდული მეთოდოლოგიის შემუშავება და ამ მეთოდოლოგიის ფარგლებში გამოყენებული დროის და პროგრამული კოდის ხარისხის კვლევა. კერძოდ, რა ვადებშია ახალი ჰიბრიდული მეთოდოლოგიით მართვის საინფორმაციო სისტემების პროექტების შექმნა შესაძლებელი, მეთოდოლოგიის პლიუს-მინუსები, რეკომენდაციები და სხვ.

კვლევის შედეგები ვარაუდობს, რომ შემოთავაზებული ჰიბრიდული მეთოდოლოგიის დანერგვამ კომპანიებში შესაძლოა მნიშვნელოვნად გააუმჯობესოს ტესტირების დროის მართვა სპრინტების ფარგლებში და გააუმჯობესოს მართვის საინფორმაციო სისტემისთვის (MIS) შემუშავებული პროგრამული უზრუნველყოფის ხარისხი. დროის მენეჯმენტის ხარვეზების შერბილებით და ყოვლისმომცველი ტესტირების ტექნიკის ინტეგრირებით, გუნდს შეუძლია მიაღწიოს ბალანსს განვითარებისა და ტესტირების აქტივობებს შორის, შეამციროს ვადებთან დაკავშირებული გამოწვევების ალბათობა, გააუმჯობესოს საერთო პროდუქტიულობა და მიაწოდოს მაღალი ხარისხის პროგრამული უზრუნველყოფა მითითებულ ვადებში დამკვეთს.

შემოთავაზებული მეთოდები სთავაზობს პრაქტიკულ მითითებებს (რეკომენდაციებს) პროექტის მენეჯერებს, სკრამის ოსტატებს და განვითარების გუნდებს, რათა გააძლიერონ სპრინტის დაგეგმვისა და შესრულების პროცესები, რაც საბოლოოდ გამოიწვევს პროექტების უფრო წარმატებულ და დროულ მიწოდებას.

თავი 7

პროგრამული სისტემების აგება Agile Development და Agile Testing მეთოდოლოგიის საფუძველზე

7.1. მართვის საინფორმაციო სისტემები (ზოგადი მიმოხილვა)

საინფორმაციო სისტემებს მნიშვნელოვანი როლი აკისრია თანამედროვე მსოფლიოში, ისინი აძლიერებს ოპერაციულ ეფექტიანობას, მხარს უჭერს გადაწყვეტილების მიღებას, ინოვაციას და აძლიერებს გლობალურ კავშირს. ისინი გახდა ორგანიზაციების განუყოფელი ნაწილი ინდუსტრიაში და ასრულებს გადამწყვეტ როლს მდგრადი ეკონომიკური ზრდის, სერვისების გაუმჯობესებისა და ცხოვრების ხარისხის ამაღლებაში.

მართვის საინფორმაციო სისტემები (Management Information Systems - MIS) საშუალებას აძლევს ორგანიზაციებს გაამარტივოს თავიანთი ოპერაციები და გაზარდოს ეფექტიანობა, ხელი შეუწყოს მონაცემთა მენეჯმენტს და უზრუნველყოს ანალიზისა და გადაწყვეტილების მიღების საშუალებები. ეს იწვევს პროდუქტიულობის გაუმჯობესებას, ხარჯების შემცირებას და რესურსების უკეთ განაწილებას. შესაძლებელია ინფორმაციის მიღება რეალურ დროში, მისი ოპერატიული გააანალიზება და შედეგების პროგნოზირება. ეს კი მარკეტინგისთვის სტრატეგიულ და ტაქტიკურ გადაწყვეტილებათა მყარი საფუძველია.

საინფორმაციო სისტემები ხელს უწყობს თანამშრომლობას და კომუნიკაციას ორგანიზაციებში და გეოგრაფიულად განაწილებულ გუნდებში. უზრუნველყოფს ინფორმაციის გაზიარების, ამოცანების კოორდინაციისა და გუნდური მუშაობის ხელშეწყობის პლატფორმებს. ეს აუმჯობესებს შიგა კომუნიკაციას, აძლიერებს ცოდნის გაზიარებას და ეფექტური თანამშრომ-ლობის საშუალებას იძლევა.

სისტემები ხელს უწყობს საზღვრებს მიღმა კომუნიკაციას, საშუალებას აძლევს განხორციელდეს საერთაშორისო ვაჭრობა და მხარს უჭერს მიწოდების გლობალურ ქსელებს. ამ ურთიერთკავშირმა გახსნა ახალი ბაზრები, გააფართოვა ბიზნეს შესაძლებლობები და ხელი შეუწყო კულტურულ გაცვლას.

გარდა ამისა, საინფორმაციო სისტემები ხელს უწყობს ცოდნის გაზიარებას და თანამშრომლობას მკვლევარებს შორის შესაბამისად განაპირობებს წინსვლას სხვადასხვა სფეროში. საინფორმაციო სისტემებმა შეცვალეს განათლების მიღების გზები ონლაინ სასწავლო რესურსებით, ვირტუალური კლასებით და

ერთობლივ პლატფორმებზე წვდომით. ისინი იძლევა პერსონალიზებულ სასწავლო გამოცდილებას, დისტანციურ განათლებას და უწყვეტი სწავლის შესაძლებლობებს. საინფორმაციო სისტემები ასევე მხარს უჭერს ადმინის-ტრაციულ ამოცანებს საგანმანათლებლო დაწესებულებებში, როგორიცაა სტუ-დენტთა ჩანაწერების მართვა და კურსის დაგეგმვა [78,79].

მათ რევოლუცია მოახდინეს ჯანდაცვისა და საჯარო სერვისების სფეროში. ელექტრონული სამედიცინო ჩანაწერები, ტელემედიცინა და ჯანმრთელობის ინფორმაციის გაცვლა იძლევა სამედიცინო ინფორმაციის უწყვეტი გაზიარების საშუალებას.

საინფორმაციო სისტემები ასევე მხარს უჭერს სამთავრობო სერვისებს, როგორიცაა ელექტრონული მმართველობა, ონლაინ გადასახადების წარდგენა და საზოგადოებრივი უსაფრთხოების სისტემები.

საინფორმაციო სისტემების ეფექტურად ორგანიზება რამდენიმე საკვანძო ნაბიჯს მოიცავს [80]. ესენია:

- განისაზღვროს ორგანიზაციის ან პროექტის საინფორმაციო მოთხოვნები;
- ლოგიკური იერარქიის შემუშავება, მონაცემთა კატეგორიზაცია და სხვა-დასხვა ელემენტს შორის ურთიერთობის დამყარება. მონაცემთა მოდელირება, ერთეულებთან ურთიერთობის დიაგრამები ან ტაქსონომიის შემუშავება;
- ინფორმაციის მართვის სისტემის არჩევა - შესაბამისი ინსტრუმენტები და ტექნოლოგიები ინფორმაციის სამართავად და შესანახად;
- მონაცემთა სტანდარტების და პოლიტიკს ჩამოყალიბება;
- წვდომის კონტროლის დანერგვა: ვის უნდა ჰქონდეს წ;ვდომა სხვადასხვა დონის ინფორმაციაზე, როლების, პასუხისმგებლობისა და უსაფრთხოების მოთხოვნების მიხედვით;
- მომხმარებლის ინტერფეისების დიზაინი;
- მონაცემთა ინტეგრაცია და ურთიერთთანამშრომლობა;
- უნდა იყოს განხორციელებული მონაცემთა სარეზერვო სისტემის რეგულარული პროცედურები მონაცემთა დაკარგვისგან დასაცავად;
- უზრუნველყოფილ იქნას მომხმარებლებისთვის ტრენინგი, თუ როგორ გამოიყენონ საინფორმაციო სისტემა ეფექტურად. შეთავაზებულ იქნეს მუდმივი ტექნიკური მხარდაჭერა და რესურსები წამოჭრილი ნებისმიერი საკითხის ან კითხვის გადასაჭრელად;
- მუდმივ კონტროლს უნდა ექვემდებარებოდეს საინფორმაციო სისტემების გამოყენებადობა და ეფექტურობა.

კვლევის საგანი – *Agile დეველოპმენტი და ტესტირება* გულისხმობს პროგრამული უზრუნველყოფის დეპარტამენტებში პროცესების მართვას ციკლების დახმარებით, რაც ეტაპობრივად მუშა პროდუქტის დამკვეთთან ჩაბარებას განაპირობებს.

7.2. Agile მეთოდოლოგია vs „ჩანჩქერის“ მოდელი

პროგრამული პროდუქტების მენეჯმენტში მნიშვნელოვანი ადგილი უკავია მათ სასიცოცხლო ციკლის საკითხს. მოკლედ შევეხოთ პრობლემას, თუ როგორ იმართებოდა რთული პროგრამული პროექტები Agile-ს არსებობამდე. საბაზო მეთოდებს შორის ერთ-ერთი პოპულარული იყო ე.წ. „ჩანჩქერის“ (waterfall) მოდელი [9]. ჩანჩქერის მოდელის ნაკლოვანებებმა გამოიწვია Agile მეთოდოლოგიების შექმნა.

- **ჩანჩქერის მოდელი:** ესაა პროგრამული უზრუნველყოფის განვითარების მიდგომა, რომელიც ცდილობს დაასრულოს თითოეული ეტაპი შემდეგზე გადასვლამდე, მაგრამ აქვს ჩაშენებული უკუკავშირის მარყუჟები, რომლებიც საშუალებას აძლევს გუნდს დაბრუნდეს წინაზე, იმ შემთხვევაში თუ პრობლემებს აღმოაჩენს. ამრიგად, პროგრესი გამოიხატება ერთი ეტაპიდან მეორეზე გადასვლისას. თითოეული ფაზა მიედინება დასასრულისკენ, რათა დაიწყოს შემდეგი. დავალება შეიძლება სხვა ფაზაში გადავიდეს მხოლოდ მიმდინარე ფაზის დასრულების შემდეგ [81-83]. ჩანჩქერის განვითარების ეტაპებია:

1) *მოთხოვნების ანალიზი* – ამ დროს იკრიბება დამკვეთის ყველა მოთხოვნა. იგი მოიცავს დაინტერესებულ მხარეებთან და საბოლოო მომხმარებლებთან საუბარს;

2) *დიზაინი* – პროგრამული უზრუნველყოფის დიზაინერები იღებენ მოთხოვნის ფაზაში შეგროვებულ ინფორმაციას და შეიმუშავებენ სტრუქტურას, რომელიც მხარს უჭერს საჭიროებებს. ზოგ შემთხვევებში ასევე შეიძლება განხორციელდეს მომხმარებლის ინტერფეისის დიზაინი ამ ნაბიჯის დროს;

3) *პროგრამული დეველოპმენტი* – დეველოპერები იღებენ ყველა მოთხოვნას და დიზაინის დოკუმენტს და წერენ შესაბამის პროგრამულ უზრუნველყოფას;

4) *ტესტირება* - ტესტირების ჯგუფი, ბიზნეს-ანალიტიკოსი და კლიენტი (დამკვეთის წარმომადგენელი) ამოწმებს სისტემას. კლიენტმა უნდა დაადასტუროს, რომ სისტემა მუშაობს და იქცევა ისე, როგორც საჭიროა (ანუ ის

ვალიდურია). ეს ნაბიჯი ხშირად გამოტოვებულია. ამ შემთხვევაში, კლიენტი ამოწმებს სისტემას ტექნიკური მომსახურების ფაზაში;

5) *ტექნიკური მომსახურება* - კლიენტი ადგენს ნებისმიერ დეფექტს, რათა სისტემა საჭიროებებს მოერგოს;

6) ქვემოთ მოცემული დიაგრამა ასახავს ჩანჩქერის განვითარების მოდელის ფაზებს (ნახ.7.1).

➢ ჩანჩქერის მოდელის უპირატესობები და ნაკლოვანებები.

– *უპირატესობები:*

• ხშირად დიზაინის შეცდომა იდენტიფიცირებულია და დაფიქსირებულია პროექტის დასაწყისში. ეს ნიშნავს, რომ მათი გამოსწორების მასშტაბი ამ ეტაპზე გაცილებით დიდია, ვიდრე ეს იქნებოდა, უკვე შემუშავებული პროგრამული უზრუნველყოფის შემთხვევაში;

ნახ. 7.1. „ჩანჩქერის" მოდელი (დაღმავალი მეთოდი იტერაციებით)

• პროგრესის დროის განსაზღვრა მარტივია, რადგან არის კარგად განსაზღვრული ეტაპები, როდესაც ჯგუფი გადადის ერთი ფაზიდან მეორეზე სავარაუდო დროის მიხედვით;

• ზუსტი ხარჯების შეფასება შესაძლებელია – თუნდაც დიდი პროექტებისთვის. დიზაინის ფაზის დასასრულს ჯგუფს ეცოდინება გეგმა და, შესაბამისად, შეუძლია წარმოადგინოს ზუსტი ხარჯთაღრიცხვა მთელი პროექტისთვის.

– *უარყოფითი მხარეები:*

ჩანჩქერის მოდელის შეცვლა რთულია. ვთქვათ, რომ ორი თვე დაჭირდა პროექტის მოთხოვნების შეგროვებას, ერთი თვე დიზაინს და ორი კვირა განხორციელებას. და შემდგომ გაჩნდა ეჭვი ჯგუფში, რომ მოთხოვნები

არასწორია. ამ შემთხვევაში, ისინი უნდა დაუბრუნდნენ საწყისს და ხელახლა გააკეთონ მოთხოვნების ანალიზის ეტაპი. ამას შეიძლება დასჭირდეს მნიშვნელოვანი ადმინისტრაციული ხარჯები და შესაძლოა არა თუ რთული ცალკეულ შემთხვევაში შეუძლებელიცაა. გუნდის წევრებს შეუძლიათ წინააღმდეგობა გაუწიონ ცვლილებას, მაშინაც კი, თუ იციან, რომ მოთხოვნები არ არის შესაფერისი მიზნისთვის.

პროექტები ხშირად ხდება ხანდაზმული და ვადებში გრძელი ჩანჩქერის მოდელით და მიწოდების სავარაუდო თარიღი, როგორც წესი, არ არის დაცული. ეს იმიტომ, რომ დიდი პროექტების დაგეგმვას ბევრი უცნობი რამ აქვს. თუ ჯგუფი ცდილობს დაგეგმოს რესურსები და ხარჯები ცხრათვიანი პროექტისთვის, მაშინ ეს ხარჯები და რესურსების შეფასებები ზუსტი იქნება მხოლოდ იმ შემთხვევაში, თუ რაიმე მოთხოვნები ან გარემოებები არ შეიცვლება პროექტის ხანგრძლივობისას. ეს ხშირად არარეალურია.

ჩანჩქერის მოდელის მთავარი მინუსი არის ის, რომ იგი ვარაუდობს, რომ ჯგუფმა წინასწარ იცის ყველა მოთხოვნა, სანამ კოდის წერას დაიწყებს დეველოპერი. არის შემთხვევები, როცა ყველა დეტალს წინასწარ იცნობს, მაგრამ არა უმეტეს პროგრამული უზრუნველყოფის განვითარების პროექტებისთვის. შეიძლება ჯგუფი ფიქრობდეს, რომ გარკვეული ფუნქცია სასიცოცხლოდ მნიშვნელოვანია კლიენტებისთვის ან რომ მომხმარებლები გაიგებენ მათ მიერ შემუშავებულ ინტერფეისს. თუმცა, სანამ დიზაინი არ შემოწმდება და არ მიიღება გამოხმაურება, ჯგუფი დანამდვილებით ვერ გაიგებს ამ ყველაფერს.

მთავარი მიზეზი იმისა, რომ დღეს „*ჩანჩქერი*" ხშირად არ გამოიყენება პროგრამული უზრუნველყოფისთვის, არის ვარაუდი, რომ ჯგუფს წინასწარ არ ეცოდინება ყველა მოთხოვნა და ისინი შეიცვლება.

ჩანჩქერის მოდელი უგულებელყოფს ამ ხშირ ცვლილებებს პროექტის სასიცოცხლო ციკლის განმავლობაში. აქედან გამომდინარე, ის შეიძლება ეფექტური იყოს, თუ ზუსტად იცის ჯგუფმა, რას მოითხოვს კონკრეტული დამკვეთი.

სამყარო სწრაფად იცვლება, ისევე როგორც მომხმარებლის საჭიროებები. იმ შემთხვევაშიც კი, თუ ჯგუფს თავიდანვე აქვს „სრულყოფილი მოთხოვნები", პროექტის ბოლოს კლიენტისთვის შეიძლება საჭირო გახდეს სხვა პრიორიტეტები/ მოთხოვნილებები. მიუხედავად იმისა, რომ უახლოეს

წარსულში მენეჯერებს მოსწონდათ „ჩანჩქერის“ მოდელი მისი ლოგიკური ნაკადის გამო, პროგრამული უზრუნველყოფის განვითარების ახალი ფორმა, სახელწოდებით „Agile განვითარება“, პოპულარული გახდა. იგი უზრუნველყოფს სისტემის განვითარების მეთოდს განმეორებითი ციკლების მეშვეობით, რომელიც შეიძლება ხელახლა განიხილოს და განისაზღვროს პროექტის ნებისმიერ ეტაპზე.

- *Agile განვითარება და ჩანჩქერის მოდელის შედარება*

ჩანჩქერის მეთოდით გუნდი აგროვებდა ყველა მოთხოვნას, აკეთებდა დიზაინს, ქმნიდან პროგრამულ უზრუნველყოფას, ამოწმებდა და შემდეგ აჩვენებდა კლიენტს. პროექტის ბოლოს კლიენტისთვის ყველაფრის მიწოდება ახასიათებდა ჩანჩქერის მოდელს. ერთი მიწოდების ნაცვლად, Agile მიდგომას ექნება რამდენიმე ციკლი. კლიენტი მიიღებს რაიმე ღირებულს ყოველი ციკლის ბოლოს.

Agile მიდგომის დახმარებით ორგანიზაციაში იქმნება გამჭირვალე სამუშაო გარემო. აღნიშნული მეთოდოლოგია საშუალებას აძლევს გუნდს და მომხმარებლებს იყვნენ ჩართული სამუშაო პროცესის თითოეულ ეტაპზე. შედეგად პროდუქტი/სერვისი იქნება მომხმარებლის საჭიროებებზე მორგებული. Agile მიდგომით შესაძლებელია შეიქმნას და იმართოს მულტიფუნქციური გუნდები. განმეორებითი იტერაციებითა და დაკვირვებით, გუნდს საშუალება ეძლევა დროულად აღმოფხვრას ხარვეზები და შექმნას მაღალი ხარისხის პროდუქტი. Agile მიდგომა ფოკუსირებულია მომხმარებელთა ბიზნეს ღირებულებებზე. მისი საშუალებით შესაძლებელია დამკვეთს მიეწოდოს მაღალი ხარისხის ფუნქციონალი ხშირად და დროულად.

იმისთვის, რომ მომხმარებლის საჭიროებებზე მორგებული პროდუქტი ეფექტურ ვადაში შეიქმნას და მიეწოდოს შემოვიდა ახალი – Agile (მოქნილი, სწრაფი) მეთოდოლოგია. Agile გულისხმობს პროცესების გამარტივებას, ბიუროკრატიის შემცირებას და ეფექტურობის გაზრდას.

თუ „ჩანჩქერით“ მუშაობისას ამ 5 ეტაპის განხორციელებას თვეები ან ხანდახან წელიც კი შეიძლებოდა დაჭირვებოდა, Agile-ს შემთხვევაში თითოეული ეტაპის განხორციელება ხდება რამდენიმე კვირაში. როგორც წესი ძირითადად 2-4 კვირიან ინტერვალში. პროცესი იყოფა ციკლებად და ამ ციკლებს ეწოდებათ *სპრინტები*.

7.3. Agile მეთოდოლოგიის Scrum მეთოდი

Agile მეთოდოლოგიაზე დაყრდნობით შექმნილი ჩარჩო არის Scrum, რომელიც დღეს ყველაზე მეტადაა გავრცელებული სხვადასხვა ტექნოლოგიურ თუ არატექნოლოგიურ კომპანიაში. Scrum - არის Agile მიდგომა ინოვაციური პროდუქტებისა და სერვისების შესაქმნელად. Scrum-ის თეორია გვეუბნება რომ ცოდნა მიიღება გამოცდილებისგან, გადაწყვეტილებები გუნდმა უნდა მიიღოს იმაზე დაყრდნობით თუ რა არის იმ კონკრეტულ მომენტში ცნობილი.

Scrum-ის იდეოლოგიის თანახმად წარმატება მაშინ მიიღწევა, როდესაც მცირე, თვითორგანიზებულ ერთეულებად დაყოფილი გუნდები რთულ, თუმცა, დაძლევად გამოწვევებსა და დავალებებს ეჭიდებიან. გამბედაობა, ღიაობა, ვალდებულება, საკითხზე ფოკუსირება და ურთიერთპატივისცემა ის ღირებულებებია, რომლებსაც Scrum გუნდში უდიდესი მნიშვნელობა აქვს;

სამუშაო პროცესში, გუნდის თითოეული წევრი საკუთარ თავზე იღებს პასუხისმგებლობის ნაწილს და პირადად შეაქვს წვლილი საერთო მიზნის მიღწევაში. ის წარმოადგენს ფრფეიმვორკს, რომელიც გუნდს ერთად მუშაობაში და რთული პროექტების წარმატებით განხორციელებაში ეხმარება.

მისი წესები როლებს, ღონისძიებებსა და არტეფაქტებს აერთიანებს, მთავარ როლს კი Scrum გუნდი ასრულებს. არტეფაქტები სამუშაოს გამჭვირვალობას, ადაპტაციასა და შემოწმებას უწყობს ხელს და ჩართულ მხარეებს საერთო ხედვის ჩამოყალიბებაში ეხმარება. Scrum ღონისძიებები კი – კონკრეტული ხანგრძლივობის მქონე აქტივობებია, რომლებიც დროის ოპტიმიზაციასა და არასაჭირო შეხვედრების რაოდენობის შემცირებას უწყობს ხელს. ფრეიმვორკის ყოველი კომპონენტი გადამწყვეტი მნიშვნელობისაა და გარკვეულ მიზანს ემსახურება. რისკების უკეთ სამართავად, Scrum იტერაციულ (პროცესის განმეორება) და ინკრემენტულ (საფეხურეობრივად ზრდად) მიდგომებს იყენებს [9, 84,85].

7.3.1. Scrum: როლები და ფუნქციები

Scrum შედგება დეველოპერთა თვითმმართვადი გუნდისგან, Scrum ოსტატისგან და პროდუქტის მფლობელისგან (ნახ.7.2).

- **პროდუქტის მფლობელის** – პასუხისმგებლობაა როგორც დეველოპერთა გუნდის მუშაობა, ისე პროდუქტის სარგებლის ზრდა. პროდუქტის ბექლოგის ელემენტების ნათლად განსაზღვრა, გამჭვირვალობის უზრუნველყოფა და სწორი თანმიმდევრობით დალაგება.

ნახ. 7.11. როლები Scrumში

ბექლოგი იმ ამოცანათა ჩამონათვალია, რომელთა შესრულებაც უფრო მსხვილი სტრატეგიული გეგმის განსახორციელებლად არის საჭირო.

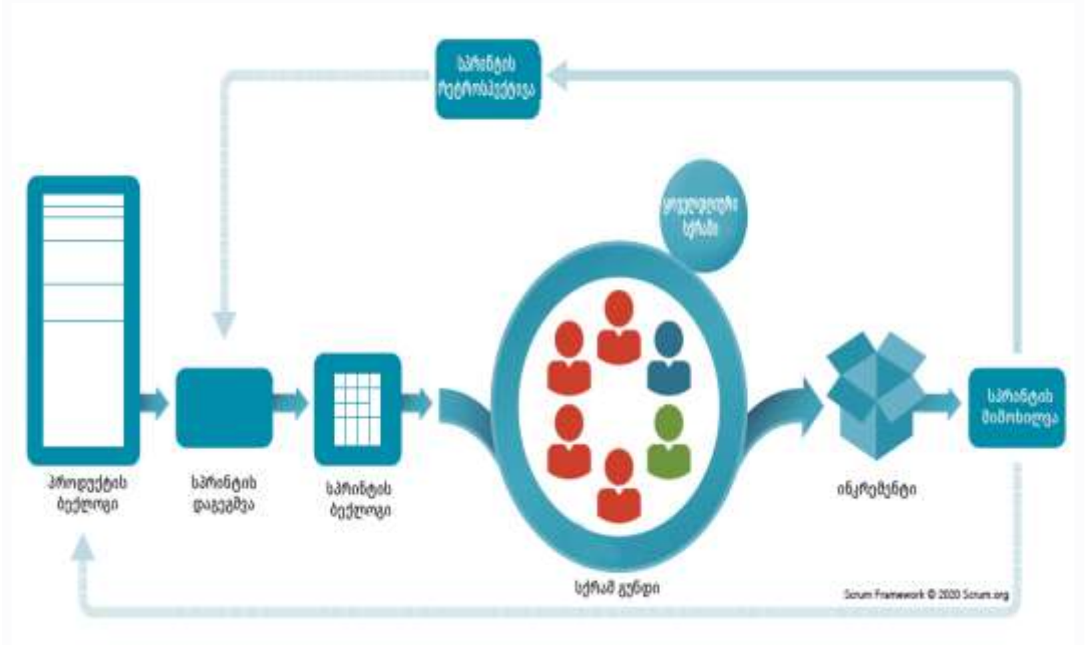
- ***დეველოპერთა გუნდი*** მცირე ზომის, ოპტიმალურად დაკომპლექტებული გუნდია, რომელიც თავადვე მართავს სამუშაო პროცესს და საერთო მიზნად ყოველი სპრინტის ბოლოს მუშა პროდუქტის მიღებას ისახავს. სპრინტი დროის ის მცირე მონაკვეთია, რომლის განმავლობაშიც დეველოპერთა გუნდმა პროდუქტის პოტენციურად გამოყენებადი ვერსია უნდა შექმნან. როგორც წესი, ოპტიმალური შედეგის მისაღებად, დეველოპერთა გუნდში სამზე მეტი და ცხრაზე ნაკლები ადამიანი ერთიანდება.

- ***Scrum ოსტატი*** გუნდის თითოეულ წევრს საკუთარი ამოცანის ნათლად გააზრებაში ეხმარება და სამუშაო პროცესის გამჭვირვალობას უზრუნველყოფს. Scrum ოსტატის ამოცანაა, გუნდმა Scrum-ის პრაქტიკა და ჩარჩოს თეორია კარგად გაიგოს. იგი გუნდის ინტერესების გამტარი ლიდერია, რომელიც პროდუქტის მფლობელს სწორი სამუშაო ტექნიკის შერჩევასა და ბექლოგის ელემენტების დახარისხებაში ეხმარება. კომპანიებისათვის Scrum ოსტატი ძალიან მნიშვნელოვანია, რადგან სწორედ ის ქმნის დეველოპერთა თვითორგანიზებულ გუნდს და ხელს უწყობს მუშა პროდუქტის შექმნას,

პარალელურად კი, ბუნებრივია, Scrum-ის იმპლემენტაციასა და ადაპტაციას უზრუნველყოფს.

იმისათვის, რომ სამუშაო პროცესი გამჭვირვალე იყოს, აუცილებელია, გუნდის თითოეული წევრი საკუთარ შედეგზე იღებდეს პასუხისმგებლობას და საერთო სტანდარტებსა და ხედვას იზიარებდეს. მნიშვნელოვანია, მთელ გუნდს ესმოდეს სამუშაო პროცესისას გამოყენებული ტერმინოლოგია, პროდუქტის მზაობას კი, ყველა ერთნაირად აფასებდეს. ჩარჩოს მიხედვით, მუშაობის განუყოფელი ნაწილი პროდუქტის ხშირი ინსპექციაა, რაც გუნდის წევრებს ცდომილებებისა და დროის ფუჭად კარგვის თავიდან აცილების საშუალებას აძლევს. ბუნებრივია, შემოწმებები არ უნდა იყოს იმდენად ხშირი, რომ ვინმეს მუშაობაში ხელი შეეშალოს, მეტი ეფექტურობისათვის კი, უმჯობესია, პროდუქტი პროფესიონალმა შეამოწმოს. თუ შემოწმება გამოავლენს, რომ სამუშაო პროცესში ცდომილებაა და პროდუქტი ვერ აკმაყოფილებს შეთანხმებულ სტანდარტებს, გუნდმა მყისიერად უნდა დააკორექტიროს მასალა.

Scrumში არსებობს შემოწმება-ადაპტაციის 4 ინსტრუმენტი [9, 86]: 1) სპრინტის დაგეგმვა; 2) ყოველდღიური Scrum; 3) სპრინტის მიმოხილვა; 4) სპრინტის რეტროსპექტივა (ნახ.7.3).



ნახ. 7.3. Scrum-ის ცერემონიები

7.3.2. სპრინტი

სპრინტი დროის მოკლე, შეზღუდული პერიოდია, რომლის დროსაც გუნდმა გარკვეული რაოდენობის სამუშაო უნდა შეასრულოს. სპრინტი Scrum-ის და Agile მეთოდების ბირთვია და თვითორგანიზებად გუნდებს მაქსიმალური პროდუქტიულობის მიღწევაში ეხმარება. როგორც წესი, სპრინტი 1-კვირიდან 1-თვემდე პერიოდს მოიცავს. ყოველი ახალი სპრინტი მხოლოდ მისი წინამორბედის დასრულების შემდეგ იწყება. ყოველი სპრინტისათვის გუნდი კონკრეტულ მიზნებსა და ამოცანებს ისახავს, მოქნილ გეგმას შეიმუშავებს და საბოლოო პროექტს განსაზღვრავს. დიდ პერიოდზე გაწელილი სამუშაო პროცესის განმავლობაში შესაძლოა, დასახული ამოცანა შეიცვალოს ან არასაჭირო გახდეს, მოკლევადიან სპრინტებს კი, მინიმუმამდე დაჰყავს ასეთი რისკები [9,87,88].

ნახ. 7.4. სპრინტი

სპრინტის დაგეგმვისას დროის მონაკვეთში შესასრულებელი სამუშაოები განისაზღვრება. დაგეგმვისას, Scrum-მასტერი გუნდის წევრებს დასახული მიზნების მაქსიმალურ სიცხადეს უზრუნველყოფს. დავალების გააზრების გარდა, იგი გუნდის წევრებს დასახული მიზნის დროის კონკრეტულ მონაკვეთში მოქცევას ასწავლის. სპრინტის დაგეგმვის შემდეგ გუნდის წევრებმა უნდა იცოდნენ რა შედეგს მიიღებენ სპრინტის ბოლოს და კონკრეტულად რა სამუშაო უნდა შეასრულონ პროდუქტის შესაქმნელად.

ყოველდღიური Scrum-სპრინტი დროის 15-წუთიანი მონაკვეთია, რომლის განმავლობაშიც დეველოპერთა გუნდის წევრები ერთმანეთს 8-საათიან სამუშაო გეგმას უზიარებენ. ამ მოკლე შეხვედრებზე განიხილება უკვე შესრულებული სამუშაოები, შესასრულებელი დავალებები, შესაძლო დაბრკოლებები, საერთო მიზანი და გუნდის თითოეული წევრის წვლილი. სპრინტის მიმოხილვა სამუშაოს შესასრულებლად აღებული დროის კონკრეტული მონაკვეთის

დასასრულია. ამ ღონისძიების ორგანიზებას Scrumმასტერი უწევს, გუნდის წევრები კი, მიღწეულ შედეგს და მაქსიმალური ეფექტის მისაღწევად საჭირო ნაბიჯებს განიხილავენ [89,90].

სპრინტის მეთოდოლოგია იძლევა პროგნოზირების საშუალებას რა ვადებშია შესაძლებელი კონკრეტული პროდუქტის შექმნა. მისი დაგეგმვა იწყება სპრინტის განმავლობაში შესასრულებელი სამუშაოს შეგროვებით. შეგროვებული დავალებები პროდუქტის ბექლოგში განთავსდება. შემდეგ გაიმართება შეხვედრა სადაც დეველოპერთა გუნდი ქულებით შეაფასებს თითოეული დავალების სირთულეს და აირჩევს სპრინტის მიზანს (ყველაზე მნიშვნელოვანი ამოცანა). ამასთან გუნდის წევრებს აქვთ შესაძლებლობა ცვლილებები შეიტანონ დავალებებში, შემდეგ დგება „მზადაა“ ეტაპის პირობა. იქამდე სანამ დეველოპერის ყველა კითხვას დავალებასთან დაკავშირებით პასუხი არ გაეცემა სპრინტის სტატუსი „მზადაა“ ეტაპზე არ გადადის, იმის შემდეგ რაც „მზადაა“ ეტაპი დადგება შესაძლებელია უკვე სპრინტის დაწყება.

სპრინტის რეტროსპექტივა სპრინტის მიმოხილვასა და მომდევნო სპრინტის დაგეგმვას შორის არსებული ღონისძიებაა, რომელიც გუნდს საშუალებას აძლევს, მიღებული გამოცდილებიდან გამომდინარე შემდეგი ნაბიჯები უფრო პროდუქტიულად დაგეგმონ. რეტროსპექტივა გუნდის წევრებს იმ საკითხების გამოვლენაში ეხმარება, რომლებსაც გაუმჯობესება სჭირდება.

7.3.3. Scrum ფრეიმვორკის დადებითი და უარყოფითი მხარეები

➢ *დადებითი მხარეები:*

- Scrum ფრეიმვორკი ადაპტირებადი და მოქნილია, შესაფერისია სხვადასხვა გარემოებისა და სიტუაციებისთვის, რომლებსაც თავდაპირველად არ აქვთ მკაფიოდ იდენტიფიცირებადი მოთხოვნები და საჭიროებენ მოქნილ მიდგომას.
- Scrum-ის გუნდები ერთად მუშაობენ, იღებენ და აანალიზებენ იდეებს ყველა წევრისგან, ამასთან კრეატიულობა წახალისებულია. შეიძლება ითქვას, რომ მეთოდოლოგია შემოქმედებით მიდგომებს უწყობს ხელს.
- Scrum-ის მიდგომის მიღება შეიძლება ორგანიზაციისთვის ეფექტური იყოს, რადგან ის ჩვეულებრივ მოითხოვს ნაკლებ დოკუმენტაციას და კონტროლს.

- გუნდში ცალკეული პასუხისმგებლობის აღებას თითოეული წევრის მხრიდან შეუძლია შექმნას პროდუქტიული გარემო, რომელსაც მივყავართ მაღალი ხარისხის საბოლოო შედეგებამდე.
- როდესაც გუნდში ყველა წევრი სრულად არის ადაპტირებული, ეს გარემოება საფუძველია კმაყოფილი თანამშრომლებისგან შედგენილი ძლიერი გუნდის, Scrum უფრო მეტად აჩენს პროექტში ჩართული თანამშრომლების მოტივაციას და კმაყოფილებას.

➢ *მინუსები:*

- მიუხედავად იმისა, რომ Scrum ფრეიმვორკის გამოყენებამ შეიძლება გამოიწვიოს სწრაფი და მაღალი ხარისხის შედეგები, ის მოითხოვს კარგად მომზადებულ და კვალიფიციურ გუნდს მის სწორად განსახორციელებლად. გუნდში ყველამ უნდა გაიგოს ამ მიდგომის უპირატესობები და თავისებურებები, რომ პროექტი წარმატებული იყოს;
- მიდგომის გამოყენება მსხვილი პროექტებისთვის შეიძლება იყოს რთული, რადგან მისი უფრო დიდი მასშტაბის განხორციელება მოითხოვს ვრცელ ტრენინგს და ზუსტ კოორდინაციას. მიუხედავად იმისა, რომ დიდ პროექტებთან ადაპტაციის გზები შემუშავებულია, მათი გაგება და განხორციელება ჩვეულებრივ რთულია;
- მას შეიძლება დასჭირდეს მნიშვნელოვანი გარდაქმნები ორგანიზაციის შიგნით. Scrum ჩარჩოს მიღება ზოგჯერ ნიშნავს, რომ კომპანიამ უნდა გაიაროს გარკვეული ორგანიზაციული ტრანსფორმაციები, რომ ეს გადაწყვეტილება იყოს წარმატებული. პროცესის ზოგიერთმა ნაწილმა შეიძლება მოითხოვოს სხვადასხვა დეპარტამენტების თანამშრომლობა და გუნდურად მუშაობა;
- შეიძლება რთული იყოს პროექტის მართვის კლასიკურ მიდგომასთან ინტეგრირება. მიუხედავად იმისა, რომ ეს ჩვეულებრივ კარგი გამოსავალია პროექტებისთვის, რომლებსაც მუდმივი კორექტირება სჭირდებათ, Scrum მიდგომა შეიძლება არ იყოს შესაფერისი პროექტებისთვის, რომლებიც საჭიროებენ პროგნოზირებადობას და კარგად განსაზღვრულ გეგმას. თუმცა, ამ ტიპის პროექტებთან მიახლოება შესაძლებელია ჰიბრიდული გადაწყვეტის გამოყენებით, რომელიც მოიცავს კლასიკურ, გრძელვადიან დაგეგმვას;
- მიუხედავად იმისა, რომ ეს მიდგომა ზრდის იმის შანსს, რომ თითოეულმა მონაწილემ გამოიყენოს თავისი შესაძლებლობების მაქსიმუმი და

დააკმაყოფილოს მოლოდინები, პროექტის მენეჯერი და დაინტერესებული მხარეები უნდა დარწმუნდნენ, რომ პროექტი დროულად დასრულდება;

- Scrum ფრეიმვორკი, როგორც წესი, საუკეთესოდ მუშაობს მინიმუმ სამი ადამიანისგან შემდგარ გუნდებთან, მაგრამ არაუმეტეს 10-მდე. მიუხედავად იმისა, რომ ამან შეიძლება ხელი შეუწყოს თანამშრომლობას და გუნდურ მუშაობას, ზოგიერთ ორგანიზაციას შეიძლება გაუჭირდეს სამუშაო ძალის გუნდებად გადანაწილება;
- Scrum ფრეიმვორკის მიღება მოიცავს ინტენსიური მუშაობის ხანგრძლივ პერიოდს და თითოეულ წევრს უნდა ჰქონდეს გამოცდილება და უნარები, რათა სწრაფად და წარმატებით შეასრულოს საკუთარი ამოცანები. გუნდში ყველას უნდა შეეძლოს შეასრულოს დაკისრებული ვალდებულება, ჰქონდეს სწორი უკუკავშირი შედეგებსა და მთლიან პროცესზე;

ამგავარად, Scrum, Kanban და ექსტრემალური პროგრამირება (XP) არის სწრაფი ფრეიმვორკები, რომლებიც ჩვეულებრივ გამოიყენება პროდუქტის განვითარებისთვის. ისინი შეესაბამება Agile მანიფესტის პრინციპებს. მიუხედავად იმისა, რომ ყველა მათგანს ახასიათებს მნიშვნელოვანი განსხვავებები, მათ აქვთ საერთო ელემენტები: მოკლე კოდირების ციკლები, გაუმჯობესებული კომუნიკაცია, დროულად დაგეგმვა და პრიორიტეტიზაცია, სწავლის ხელშეწყობა და გაუმჯობესება და ადაპტაცია ბიზნეს გარემოში ცვლილებებზე რეაგირებისთვის [9,82,91].

7.4. Agile ტესტირება

Agile ტესტირება არის ტესტირების პრაქტიკა, რომელიც მიჰყვება Agile პროგრამული უზრუნველყოფის განვითარების წესებსა და პრინციპებს (ნახ.7.5).

ჩანჩქერის მეთოდისგან განსხვავებით, Agile ტესტირება შეიძლება დაიწყოს პროექტის დაწყებისთანავე, დეველოპმენტსა და ტესტი-რებას შორის უწყვეტი ინტეგრაციით.

ტესტირების მეთოდოლოგია არ არის თანმიმდევრული (იმ გაგებით, რომ იგი შესრულებულია მხოლოდ კოდირების ფაზის შემდეგ).

ნახ. 7.5. Agile ტესტირების ეტაპები

- *Agile ტესტის გეგმა და სტრატეგიები*

Agile ტესტის გეგმა მოიცავს ჩატარებული ტესტირების ტიპებს, როგორიცაა ინფრასტრუქტურა, სატესტო გარემო და ტესტის შედეგები.

ჩანჩქერის მოდელისგან განსხვავებით, მოქნილ მოდელში ყოველი გამოშვებისთვის იწერება და განახლდება სატესტო გეგმა. ტიპური ტესტის გეგმის საკითხებია:

- ტესტირების სფერო;
- ახალი ფუნქციები, რომლებიც ტესტირებაშია;
- ტესტირების დონე ან ტიპები მახასიათებლების სირთულის მიხედვით;
- დატვირთვისა და შესრულების ტესტირება;
- ინფრასტრუქტურის განხილვა;
- რისკების გეგმა;
- რესურსების უზრუნველყოფა;
- მიწოდება.

Agile ტესტირება არის ტრადიციული ფუნქციური ტესტირებისა და ტრადიციული მიღების ტესტირების კომბინაცია, როგორც პროდუქტის განვითარების გუნდი, და დაინტერესებული მხარეები ამას ერთად აკეთებენ. მიუხედავად იმისა, რომ დეველოპერის ტესტირება არის ტრადიციული ტესტირებისა და ტრადიციული სერვისის ინტეგრაციის ტესტირების ნაზავი. განვითარების ტესტირება ამოწმებს როგორც აპლიკაციის კოდს, ასევე მონაცემთა ბაზის სქემას.

ამ ეტაპის მიზანია სისტემის წარმატებით დანერგვა წარმოებაში. აქტივობები ამ ფაზაში მოიცავს საბოლოო მომხმარებლების ტრენინგს. ასევე, ის მოიცავს პროდუქტის მარკეტინგის, სისტემის და მომხმარებლის დოკუმენტაციის დასრულებას. სწრაფი მეთოდოლოგიის ტესტირების საბოლოო ეტაპი მოიცავს სისტემის სრულ ტესტირებას და მიღების ტესტირებას. იმისათვის, რომ დაასრულდეს საბოლოო ტესტირების ეტაპი ყოველგვარი დაბრკოლებების გარეშე, უნდა ინახოს პროდუქტი უფრო მკაცრად, სანამ ის მეორე ეტაპზეა. მესამე ეტაპზე ტესტერები იმუშავებენ მისი დეფექტების აღმოსაჩენად.

გამოშვების ეტაპის შემდეგ პროდუქტი გადავა წარმოების ეტაპზე.

შეიძლება ითქვას, რომ Agile ტესტირებას აქვს „კვადრატები", მთელ პროცესი დაყოფილია ოთხ კვადრატად და ეხმარება გუნდს იმის გაგებაში, თუ როგორ ხორციელდება Agile ტესტირება.

ა) Agile Quadrant I - ამ კვადრატში მთავარი აქცენტი არის კოდის ხარისხზე და ის შედგება ტესტის შემთხვევებისგან, რომლებიც ორიენტირებულია ტექნოლოგიაზე და დანერგილია გუნდის მხარდასაჭერად, იგი მოიცავს:

1) მოდულურ ტესტებს;
2) კომპონენტურ ტესტებს.

ბ) Agile Quadrant II – შეიცავს სატესტო შემთხვევებს, რომლებიც ბიზნესზეა ორიენტირებული და დანერგილია გუნდის მხარდასაჭერად. ეს კვადრატი ყურადღებას ამახვილებს მოთხოვნებზე. ამ ფაზაში ჩატარებული ტესტის სახეობებია:

1) შესაძლო სცენარების და სამუშაო პროცესების მაგალითების ტესტირება;
2) პროტოტიპების ტესტირება;
3) წყვილთა ტესტირება.

გ) Agile Quadrant III - უზრუნველყოფს უკუკავშირს პირველ და მეორე კვადრატებს შორის. ტესტის შემთხვევები შეიძლება გამოყენებულ იქნას, როგორც საფუძველი ავტომატიზაციის ტესტირების ჩასატარებლად. ამ კვადრატში ტარდება განმეორებითი მიმოხილვა, რაც ამყარებს ნდობას პროდუქტის მიმართ. ამ კვადრატში ჩატარებული ტესტირების ტიპი არის:

1) გამოყენებადობის ტესტირება;
2) საძიებო ტესტირება;
3) წყვილი ტესტირება მომხმარებლებთან;
4) თანამშრომლობითი ტესტირება;
5) მომხმარებლის მიღების ტესტირება.

დ) Agile Quadrant IV – კონცენტრირდება არაფუნქციურ მოთხოვნებზე, როგორიცაა შესრულება, უსაფრთხოება, სტაბილურობა და ა.შ. ამ კვადრატის დახმარებით, აპლიკაცია მზადდება არაფუნქციური თვისებების და მოსალოდნელი მნიშვნელობის მიწოდებისთვის.

1) არაფუნქციური ტესტები, როგორიცაა სტრესი და შესრულების ტესტირება;
2) უსაფრთხოების ტესტირება ავთენტიფიკაციასთან და ჰაკერებთან მიმართებაში;

3) ინფრასტრუქტურის ტესტირება;

4) მონაცემთა მიგრაციის ტესტირება;

5) მასშტაბურობის ტესტირება;

6) დატვირთვის ტესტირება.

7.5. QA გამოწვევები Agile პროგრამული უზრუნველყოფის განვითარების პროცესში

Agile QA (Quality Assurance - ხარისხის უზრუნველყოფა) პროცესი არის პრაქტიკისა და მეთოდოლოგიების ერთობლიობა, რომელიც მიზნად ისახავს, რომ Agile ფრეიმვორკით შემუშავებული პროგრამული აპლიკაცია აკმაყოფი-ლებდეს სასურველ ხარისხის სტანდარტებს [92,93]. განიხილება შემდეგი მიზეზები:

ა) შეცდომის შანსები დიდია, რადგან დოკუმენტაციას ნაკლები პრიორიტეტი ენიჭება, ეს ფაქტი კი საბოლოოდ სტანდარტულზე მეტ ზეწოლას ახდენს QA გუნდზე;

ბ) სწრაფად დაინერგება ახალი ფუნქციები, რაც ამცირებს ტესტირების გუნდებისთვის ხელმისაწვდომ დროს, რათა დაადგინონ, შეესაბამება თუ არა უახლესი ფუნქციები მოთხოვნებს;

გ) ტესტერებს ხშირად მოეთხოვებათ ნახევრად დეველოპერის როლის შესრულება;

დ) ტესტის შესრულების ციკლები ძალიან შეკუმშულია;

ე) ცოტა დრო ტესტის გეგმის მოსამზადებლად;

ვ) რეგრესიული ტესტირებისთვის მათ ექნებათ მინიმალური დრო;

თ) მოთხოვნების ცვლილებები და განახლებები თანდაყოლილია მოქნილი მეთოდისთვის, რაც ყველაზე დიდი გამოწვევაა და ხდება ხარისხის უზრუნველყოფის მიზნით.

7.5.1. ავტომატიზაციის რისკი Agile პროცესში

ავტომატური ინტერფეისი უზრუნველყოფს მაღალი დონის ნდობას, მაგრამ ისინი ნელია შესრულებაში, მყიფეა შესანარჩუნებლად და ძვირადღი-რებულია შესაქმნელად. ავტომატიზაციამ შეიძლება მნიშვნელოვნად არ გააუმჯობესოს ტესტის პროდუქტიულობა, თუ ტესტერებმა არ იციან როგორ შეამოწმონ.

არასანდო ტესტები, ავტომატური ტესტირების მთავარი პრობლემაა წარუმატებელი ტესტების გამოსწორება და მყიფე ტესტებთან დაკავშირებული საკითხების მოგვარება. ავტომატური ტესტის ინიცირება ხდება ხელით და არა უწყვეტი ინტეგრაციის (CI) მეშვეობით.

პროდუქტის მოსალოდნელი ხარისხის მისაღებად საჭიროა ტესტირების ტიპებისა და დონის ნაზავი. ბევრი კომერციულად ხელმისაწვდომი ავტომატიზაციის ხელსაწყო უზრუნველყოფს მარტივ ფუნქციებს, როგორიცაა ტესტის შემთხვევების აღების და გამეორების ავტომატიზაცია.

ამასთან, ვერსიის კონტროლის სისტემის გარეთ სატესტო შემთხვევების შენახვა არასაჭირო სირთულეს ქმნის. დროის დაზოგვის მიზნით, ხშირად ავტომატიზაციის ტესტის გეგმა ცუდად არის დაგეგმილი ან საერთოდაც დაუგეგმავია, რაც იწვევს ტესტის ჩავარდნას. სატესტო დაყენებისა და დაშლის პროცედურები, როგორც წესი, გამოტოვებულია ტესტის ავტომატიზაციის დროს, ხოლო ხელით ტესტირების ჩატარება, ტესტის დაყენება და პროცედურების დაშლა შეუფერხებლია.

პროდუქტიულობის აღრიცხვა, როგორიცაა დღე-ღამეში შექმნილი ან შესრულებული სატესტო შემთხვევების რაოდენობა, შეიძლება იყოს შეცდომაში შემყვანი და შეიძლება გამოიწვიოს დიდი ინვესტიციის განხორციელება უსარგებლო ტესტების გაშვებაში. Agile ავტომატიზაციის გუნდის წევრები უნდა იყვნენ ეფექტური კონსულტანტები, ხელმისაწვდომები და კოლეგიურები, წინააღმდეგ შემთხვევაში ეს სისტემა სწრაფად ჩავარდება. ავტომატიზაციამ შეიძლება შესთავაზოს და მიაწოდოს სატესტო გადაწყვეტილებები, რომლებიც საჭიროებენ ძალიან დიდ მუდმივ მოვლა-პატრონობას.

მოწოდებულ პროდუქტთან შედარებით ავტომატურ ტესტირებას შეიძლება არ ჰქონდეს გამოცდილება ეფექტური გადაწყვეტილებების მოსაფიქრებლად და მისაწოდებლად. ავტომატური ტესტირება შეიძლება იყოს იმდენად წარმატებული, რომ მათ ამოწურონ გადასაჭრელი მნიშვნელოვანი პრობლემები და, შესაბამისად, გადაერთონ უმნიშვნელო პრობლემებზე [94].

7.5.2. Agile მეთოდოლოგია პროგრამული უზრუნველყოფის ტესტირებაში

ვინაიდან პროგრამული უზრუნველყოფის განვითარების პროცესის სირთულე მუდმივად იზრდება, პროგრამული პროდუქტების ტესტირების მიდგომებიც შესაბამისად უნდა განვითარდეს.

Agile ტესტირების მიდგომა შედარებით ახალია, რომელიც ფოკუსირებულია უფრო *ჭკვიანურ ტესტირებაზე*, ვიდრე დიდი ძალისხმევის დახარჯვაზე და უზრუნველყოფს მაღალი ხარისხის პროდუქტებს. ამ ტესტირების დროს ტესტერებსა და დეველოპერებს სჭირდებათ მჭიდრო თანამშრომლობა. ტესტერებმა უნდა მიაწოდონ მაკორექტირებელი გამოხმაურება დეველოპერების გუნდს პროგრამული უზრუნველყოფის განვითარების ციკლის განმავლობაში. ესაა ტესტირებისა და დეველოპმენტის მიდგომებს შორის მიმდინარე ინტეგრაციის საკითხი.

Agile მეთოდოლოგია პროგრამული უზრუნველყოფის ტესტირებაში გულისხმობს ტესტირებას რაც შეიძლება ადრე პროგრამული უზრუნველყოფის განვითარების სასიცოცხლო ციკლში. ის მოითხოვს მომხმარებლის მაღალ ჩართულობას და ტესტირების კოდს, როგორც კი ის ხელმისაწვდომი გახდება. კოდი უნდა იყოს საკმარისად სტაბილური, რომ ადაპტირდეს სისტემასთან.

ვრცელი ტესტირება შეიძლება გაკეთდეს იმის დასარწმუნებლად, რომ შეცდომები გამოსწორებულია და შემოწმებულია. გუნდებს შორის კომუნიკაცია მნიშვნელოვნად მოქმედებს ტესტირების წარმადობაზე. Agile ტესტირების მეთოდოლოგიაში მოთხოვნები თანდათან ვითარდება მომხმარებლებისა და ტესტირების გუნდებისგან. განვითარება შეესაბამება მომხმარებლის მოთხოვნებს. Agile ტესტირების პროცესი უწყვეტი პროცესია და არა თანმიმდევრული. ტესტირება იწყება პროექტის დასაწყისში და მიმდინარეობს ინტეგრაცია ტესტირებასა და განვითარებას შორის. Agile განვითარებისა და ტესტირების საერთო მიზანია პროდუქტის მაღალი ხარისხის მიღწევა.

- ***Agile ტესტირების პრინციპები:***

არსებობს Agile ტესტირების პროცესის რამდენიმე პრინციპი:

– *ტესტირება უწყვეტია:* Agile გუნდი მუდმივად ამოწმებს, რადგან ეს არის ერთადერთი გზა პროდუქტის უწყვეტი პროგრესის უზრუნველსაყოფად;

– *მთელი გუნდის მიერ შესრულებული ტესტები:* ტრადიციული პროგრამული უზრუნველყოფის განვითარების სასიცოცხლო ციკლში, მხოლოდ

ტესტირების ჯგუფია პასუხისმგებელი ტესტირებაზე, მაგრამ Agile ტესტირე-ბისას დეველოპერები და ბიზნეს ანალიტიკოსები ასევე ამოწმებენ აპლიკაციას;

– *გამოხმაურების პასუხის დროის შემცირება:* ბიზნეს გუნდი ჩართულია სწრაფ ტესტირებაში თითოეულ ციკლში და უწყვეტი გამოხმაურება ამცირებს უკუკავშირის დროს;

– *გამარტივებული და სუფთა კოდი:* ყველა დეფექტი, რომელიც მოქნილი გუნდის მიერ არის დაფიქსირებული ხელს უწყობს კოდის სისუფთავესა და გამარტივებას;

– *ნაკლები დოკუმენტაცია:* სწრაფი გუნდები იყენებენ მრავალჯერადი გამოყენების საკონტროლო სიას, გუნდი ყურადღებას ამახვილებს ტესტზე შემთხვევითი დეტალების ნაცვლად;

– *ტესტის მართვა:* Agile მეთოდებით, ტესტირება ხორციელდება პროგრამული უზრონველყოფის განხორციელების დროს, ხოლო ტრადიციულ პროცესში, ტესტირება ხორციელდება განხორციელების შემდეგ.

➢ ***Agile ტესტირების მეთოდები:***

არსებობს Agile ტესტირების სხვადასხვა მეთოდი, როგორიცაა: ქცევითი განვითარება (BDD); დაშვების ტესტზე ორიენტირებული განვითარება (ATDD) და საძიებო ტესტირება.

- *ქცევითი განვითარება (BDD)* – აუმჯობესებს კომუნიკაციას პროექტის დაინტერესებულ მხარეებს შორის ისე, რომ ყველა წევრმა სწორად გაიგოს თითოეული ფუნქცია განვითარების პროცესის დაწყებამდე. არსებობს უწყვეტ მაგალითზე დაფუძნებული კომუნიკაცია დეველოპერებს, ტესტერებსა და ბიზნეს ანალიტიკოსებს შორის. მაგალითებს ეწოდება სცენარები, რომლებიც იწერება სპეციალურ ფორმატში, სახელწოდებით „Gherkin Given/When/Then“ სინტაქსი. სცენარები შეიცავს ინფორმაციას იმის შესახებ, თუ როგორ უნდა მოიქცეს მოცემული ფუნქცია სხვადასხვა სიტუაციებში სხვადასხვა შეყვანის პარამეტრით. მათ ეწოდება "შესრულებადი სპეციფიკაციები", რადგან ის მოიცავს როგორც სპეციფიკაციას, ასევე ავტომატიზირებულ ტესტებს.
- *დაშვების ტესტზე ორიენტირებული განვითარება (ATDD)* – ყურადღებას ამახვილებს გუნდის წევრების ჩართვაზე სხვადასხვა პერსპექტივით, როგორიცაა მომხმარებელი, დეველოპერი და ტესტერი. მომხმარებელი ორიენტირებულია იმ პრობლემაზე, რომელიც უნდა გადაიჭრას, განვითარება

ორიენტირებულია იმაზე, თუ როგორ მოგვარდება პრობლემა, ხოლო ტესტირება ორიენტირებულია იმაზე, თუ რა შეიძლება არასწორად მოხდეს. *მიღების ტესტები* არის მომხმარებლის თვალსაზრისი და აღწერს, თუ როგორ იმუშავებს სისტემა. ის ასევე ეხმარება იმის შემოწმებას, რომ სისტემა ფუნქციონირებს ისე, როგორც უნდა ფუნქციონირებდეს (ე.ი. ვალიდურია). ზოგიერთ შემთხვევაში მიღების ტესტები ავტომატიზებულია.

- *საძიებო ტესტირება* – ამ ტიპის ტესტირებაში, ტესტის დიზაინი და ტესტის შესრულების ფაზა ერთად მიმდინარეობს. საძიებო ტესტირება იკვლევს სამუშაო პროგრამულ უზრუნველყოფის თანხვედრას დოკუმენტაციაზე. „ინდივიდები და ურთიერთქმედება უფრო მნიშვნელოვანია, ვიდრე პროცესი და ინსტრუმენტები" – მომხმარებელთა თანამშრომლობას უფრო დიდი მნიშვნელობა აქვს, ვიდრე კონტრაქტის მოლაპარაკებას. საძიებო ტესტირება უფრო ადაპტირებულია ცვლილებებთან. ამ ტიპის ტესტერების დროს განსაზღვრავენ აპლიკაციის ფუნქციონირებას აპლიკაციის შესწავლით. ტესტერები ცდილობენ შეისწავლონ აპლიკაცია და შეიმუშავონ და შეასრულონ ტესტის გეგმები მათი დასკვნების მიხედვით.

Agile (სწრაფი) ტესტირების მეთოდოლოგიის უპირატესობები შემდეგია:

- ეს დაზოგავს დროსა და ფულს;
- სწრაფი ტესტირება ამცირებს დოკუმენტაციას;
- ის არის მოქნილი და მალიან ადაპტირებადი ცვლილებების მიმართ;
- ის უზრუნველყოფს საბოლოო მომხმარებლისგან რეგულარული გამოხმაურების მიღების საშუალებას;
- საკითხების უკეთ განსაზღვრა ყოველდღიური შეხვედრებით.[36-38].

7.5.3. ტესტირების გეგმა Agile QA -სთვის

Agile ტესტირებისას, *ტესტის გეგმა* იწერება და ასევე განახლებულია ყველა გამოშვებისთვის [96,97]. ტესტის გეგმაs აქვს: *-Agile ტესტირების გეგმა; -ტესტირების ფარგლები; -შესამოწმებელი ახალი ფუნქციების კონსოლიდაცია; -ტესტირების სახეები/ტესტირების დონეები; -შესრულების და დატვირთვის ტესტირება; -ინფრასტრუქტურის გათვალისწინება; -რისკების გეგმა; -რესურსების დაგეგმვა; -მიწოდება და ეტაპები;*

Agile ტესტირების სასიცოცხლო ციკლი მოიცავს შემდეგ 5 ფაზას {96,98}:

-შემოქმედების შეფასება; -სწრაფი ტესტირების დაგეგმვა; -გათავისუფლების მზადყოფნა; -ყოველდღიური სკრამები; -ტესტის სისწრაფის მიმოხილვა.

- ***Agile Scrum ტესტირების პროცესი***

გავაანალიზოთ, როგორ მოქმედებს ტესტერის როლი Agile მეთოდოლოგიაში. ვნახოთ, როგორ მუშაობს ტესტირება Scrum-ში:

Scrum სთავაზობს გუნდს მარტივ მეთოდებს რთული სამუშაოების შესასრულებლად ამასთან საშუალებას აძლევს Agile გუნდს კონცენტრირება მოახდინოს პროგრამული პროდუქტების შექმნის ყველა ასპექტზე, როგორიცაა ხარისხი, გამოყენებადობა, შესრულება და ა.შ. სირთულის თავიდან ასაცილებლად, აღნიშნული მეთოდოლოგიით შესაძლებელია პროცესის გამჭვირვალობა, შემოწმებადობა და ადაპტაცია პროგრამული უზრუნველყოფის განვითარების პროცესში. საბოლოო კონსტრუქციას აქვს მომხმარებლის მიერ მოთხოვნილი ყველა მახასიათებელი.

- ***ტესტერის როლი Scrum პროცესებში***

სპრინტის დაგეგმვა: გუნდი ამატებს ამოცანებს სპრინტის ბექლოგს, რომელსაც Scrum ოსტატი ხელმძღვანელობს. სპრინტის ბექლოგში თითოეული დავალების შესამოწმებლად საჭირო ძალისხმევა შეფასებულია ტესტერების მიერ. ყოველდღიური Scrum გრძელდება დაახლოებით 15 წუთის განმავლობაში და ხელმძღვანელობს Scrum ოსტატი. წევრები ისაუბრებენ წინა დღით შესრულებულ სამუშაოზე, მომდევნო დღეს დაგეგმილ სამუშაოზე და ყოველდღიური Scrum-ის დროს წარმოქმნილ ნებისმიერ პრობლემაზე. ყოველდღიური 15 წთ_იანი შეხვედრა გამოიყენება გუნდის პროგრესის მონიტორინგისთვის. სპრინტის მიმოხილვა/რეტროსპექტივა ასევე ტარდება Scrum ოსტატის მიერ, გრძელდება დაახლოებით 2-4 საათის განმავლობაში და მოიცავს გუნდის მიღწევებს წინა სპრინტიდან, ისევე როგორც ყველა აღმოჩენილ ხარვეზს [90,95].

- ***ტესტირება Scrumში***

ტესტერმა უნდა შეაფასოს საათების რაოდენობა, რომელიც საჭიროა თითოეული ამოცანის ტესტირებისთვის, ასევე უნდა იცოდეს სპრინტის მიზნების შესახებ. შეიტანოს წვლილი პრიორიტეტიზაციის პროცესში. ტესტერი პასუხისმგებელია ავტომატიზაციის პროგრამების დაწერაზე. უწყვეტი ინტეგრაციის (CI) ტექნოლოგიის გამოყენებით, ის გეგმავს ავტომატიზაციის ტესტირებას. ეს კარგად მუშაობს იმისთვის, რომ დაკმაყოფილდეს ტესტირების ყველა საჭიროება. ზოგიერთ შემთხვევაში, ტესტერი აკეთებს მიღების ტესტს (UAT) სპრინტის დასასრულს, რათა დარწმუნდეს, რომ ტესტირება

დასრულებულია ამ სპრინტისთვის. რეტროსპექტიული სპრინტი აანალიზებს მიმდინარე სპრინტს, როგორც ტესტერი, რათა დაადგინოს, რა მოხდა არასწორად [98,99].

- *ტესტერის როლი ციკლებში და სპრინტის დაგეგმვაში:*
  - გაიგე ამოცანა და მისი შესრულების კრიტერიუმები;
  - მომხმარებლის ამოცანები ზუსტდება შესაბამისი დაინტერესებული მხარეებისგან, სადაც არასაკმარისი ინფორმაციაა;
  - მაღალი დონის ტესტის სტრატეგია უნდა გადაწყდეს მთელი გამოშვებისთვის;
  - ყველა რისკი, რომელიც შეიძლება წარმოიშვას გამოშვების დროს, უნდა აღინიშნოს ან დოკუმენტირებული იყოს;
  - ტესტირების ტიპების რაოდენობა უნდა გადაწყდეს და განიხილოს;
  - გამოთვალეთ დრო თითოეული მომხმარებლის ამოცანის სატესტო შემთხვევის შექმნისა და შესრულებისთვის;
  - დაყავით მომხმარებლის ისტორიები სხვადასხვა სატესტო ამოცანებად;
  - გადაწყვიტეთ თითოეული ამბის ტესტის გაშუქება;
  - განისაზღვროს მომხმარებლის ისტორიების მიღების კრიტერიუმები;
  - გაიგეთ და დაგეგმეთ მომხმარებლის ისტორიების ავტომატიზაცია და მხარი დაუჭირეთ სხვადასხვა დონის ტესტირებას.

- *ტესტერის როლი Scrumში*

როგორც წესი, ტესტირება ხდება ტესტერის/დეველოპერის მიერ მოდულური ტესტის (Unit Test) გამოყენებით [99]. მიუხედავად იმისა, რომ პროდუქტის მფლობელი ხშირად მონაწილეობს თითოეული სპრინტის ტესტირების ფაზაში. პროექტის ბუნებიდან და სირთულიდან გამომდინარე, Scrum-ის ზოგიერთ პროექტს აქვს სპეციალიზებული ტესტის ჯგუფები.

- გაუმჯობესებული „მომხმარებელთა კმაყოფილება“ მაღალი ხარისხის პროგრამული უზრუნველყოფის მიწოდებით, რომელიც უფრო ადრეული და უწყვეტია უმაღლესი პრიორიტეტი;
- ჩართულობა არის ადრეული პროექტის განმავლობაში სპრინტის დაგეგმვის შემდეგ, მაგრამ QA აქტივობები იგივეა;
- დამკვეთებმა, დეველოპერებმა და ტესტერებმა ერთად უნდა იმუშაონ მთელი პროექტის განმავლობაში;

- საჭიროა თითოეული მომხმარებლის და დაინტერესებული მხარის აზრის გაგება და შემდეგ გადაწყვეტილების მიღება;
- ცვალებადი მოთხოვნების მხარდაჭერა. ტესტერი უნდა იყოს ადაპტირებული ნებისმიერ ცვლილებასთან;
- აქტივობების განსაზღვრა, რათა შეფასდეს დრო, განახლდეს ტესტი, როდესაც ცვლილებები გამოჩნდება, დასრულდეს ტესტირება სპრინტის დროს და ა.შ.;
- მდგომარეობის დონის შეფასება და დროის გამოყოფა;
- სატესტო ქეისები უნდა იყოს შემუშავებული მდგომარეობის მიღების კრიტერიუმების მიხედვით და უნდა შეიცვალოს, როდესაც მდგომარეობა იცვლება;
- მაღალი ხარისხის პროგრამული უზრუნველყოფია მიწოდება განმეორე-ბით, რამდენიმე კვირიდან რამდენიმე თვემდე;
- QA (ხარისხის უზრუნველყოფამ) უნდა აკონტროლოს ტესტირების პროგრესი ყოველდღიურად უწყვეტი გამოხმაურებით.

➢ *Agile Scrum ტესტირების გამოწვევები*

- თითოეული მომხმარებლის ამბისთვის ტესტირების ძალისხმევა უნდა შეფასდეს;
- გარემო და რესურსების შეზღუდვა გუნდის შესაძლებლობებს ცვლის;
- მოცულობა და სიჩქარე - დინამიური მოთხოვნები;
- რეგრესიის რისკი იზრდება კოდის ხშირი ცვლილებებით;
- ტესტის დაგეგმვა და ტესტის შესრულება ერთდროულად ხდება თუ კლიენტმა არ იცის მოთხოვნების შესახებ, გუნდის მიმართულება შეიძლება სხვა კუთხით წავიდეს.

➢ *ტესტის მოხსენება (Test Reporting)*

Scrum ტესტის შესახებ მოხსენება დაინტერესებულ მხარეებს აძლევს პროექტში მიმდინარე პროცესებისადმი ხილვადობას და გამჭვირვალობას. გუნდს შეუძლია შეაფასოს თავისი წარმატება და დაგეგმოს მომავალი სტრატეგია პროდუქტის გასაუმჯობესებლად მოხსენებული მეტრიკის გამოყენებით. იგი მოიცავს მონაცემებს სამუშაოს მთლიანი მოცულობის შესახებ, რომელიც უნდა დასრულდეს, თითოეული სპრინტის დროს შესრულებული სამუშაოს რაოდენობა და სხვა ინფორმაცია [100].

თავი 8

ობიექტის კვლევის ამოცანა და ტექნოლოგიები

კვლევის ფარგლებში დაკვირვება განხორციელდა (ერთ-ერთი ორგანიზაციის) პროგრამული უზრუნველყოფის დეპარტამენტის გუნდზე. გუნდი სპრინტ მეთოდოლოგიით მუშაობდა 26 თვე. მათ მუშაობის დასაწყისში გაიარეს მაღალი დონის ტრენინგები და გუნდის მუშაობა პირველი 3 თვის განმავლობაში მიმდინარეობდა პროფესიონალი Scrum-მასტერის ზედა-მხედველობის ქვეშ. ექსპერიმენტი 92 სამუშაო კვირის განმავლობაში მიმდინარეობდა, რაც დაახლოებით 1400 შეხვედრას მოიცავდა.

დაკვირვების ფარგლებში გამოვლინდა სპრინტ მეთოდოლოგიის მნიშვნელოვანი ნაკლოვანება: ტესტერების დროის მენეჯმენტი (2 კვირიანი სპრინტის განმავლობაში, პირველ კვირაში მათ ფაქტობრივად არ აქვთ მუშა პროდუქტი გასატესტად და სპრინტის ბოლოსკენ გროვდება საკმაოდ ბევრი საქმე ტესტირების თვალსაზრისით). ამის შემდეგ გუნდმა შეცვალა სამოქმედო სტრატეგია, მათ დაიწყეს ერთეულის ტესტების წერა. სპრინტის დასაწყისშივე იწყებდა, როგორც დეველოპერი პროგრამული უზრუნველყოფის განვითარებას, ასევე ტესტერი ე.წ. Unit-ტესტების წერას. გარკვეული სამუშაო დღეების შემდეგ პროგრამისტს მზად აქვს კოდი, ხოლო ტესტერს შესაბამისი პროგრამული უზრუნველყოფის გასატესტი ფუნქციები. კვლევა მიმდინარეობდა მასშტაბური პროგრამის *„თანამშრომლების მიღება გადინების და განსაკუთრებული პირობებით ხელფასის დარიცხვის სისტემაზე“.*

სახელფასო სისტემა არის აპლიკაცია, რომელიც შემუშავებულია C# ენის გამოყენებით. პროექტი ეფუძნება თანამშრომლების ჩანაწერების მართვის კონცეფციას, ისევე როგორც მათ სახელფასო სიას. სისტემის ფუნქციების გამოსაყენებლად მომხმარებელმა უნდა გაიაროს აუთენთიფიკაცია. ადმინისტრატორს შეუძლია დაამატოს თანამშრომლების ჩანაწერები, მართოს მომხმარებლები, შექმნას სახელფასო სია, ნახოს ანგარიშები და ა.შ. განსაზღვრული პარამეტრების შეყვანის შემდეგ სისტემა ავტომატურად წარმოქმნის მთლიანი წლის სახელფასო სიას.

მომხმარებელს შეუძლია ნახოს თანამშრომლების ჩანაწერები, მაგრამ არ შეუძლია დაამატოს ან რედაქტირება გაუკეთოს მათ. ასევე შეზღუდული აქვს წვდომა სხვა თანამშრომლების სახელფასო სიებზე.

სახელფასო სისტემის პროექტი ეხმარება მომხმარებელს თანამშრომლების ინფორმაციის მართვაში, ასევე მათი სახელფასო სისტემის მართვაში. მონაცემთა დამუშავებისთვის მონაცემთა ბაზად გამოყენებულია MySQL. ამ პროექტის გასაშვებად საჭიროა დაინსტალირებული იყოს Visual Studio .NET 2022.

8.1. ამოცანის დასმა

1) უნდა შეიქმნას სამართავი პანელი (ინტერფეისი), სადაც ადმინისტრა-ტორს შეეძლება შეიყვანოს მომხმარებლის მონაცემები (სახელი, გვარი პირადობა, სამუშაო გამოცდილება (წელი/თვე). ავტომატურად დაეთვალოს ქულა სამუშაო გამოცდილების მიხედვით და ქულის მიხედვით მიენიჭოს სახელფასო გასაცემის ოდენობა. წინასწარ დაგენერირდეს 1 წლის გასაცემი განრიგი, ამასთან თუ მომხმარებელი დარეგისტრირდა 28 რიცხვის ჩათვლით მას დაეთვალოს მიმდინარე თვის გასაცემელიც, ხოლო სხვა შემთხვევაში გასაცემელი დაეთვალოს შემდეგი თვიდან.

სამუშაოს გამოცდილების მიხედვით ქულა:

- 0-1 წლამდე - 100 ქულა
- 1-3 წლამდე - 200 ქულა
- 3 წლის ზემოთ 300 ქულა

ქულის მიხედვით გასაცემელი:

- 100 ქულა- 100 ლარი + 10% ბონუსი საახალწლო (წლის ბოლოს გასაცემი)
- 200 ქულა - 250 ლარი + 20% ბონუსი საახალწლო (წლის ბოლოს გასაცემი)
- 300 ქულა - 350 ლარი + 30% (მაქსიმუმ 100 ლარი) (წლის ბოლოს გასაცემი)

2) მონაცემები როგორც ემატება, ასევე უნდა

- ნახლდებოდეს (წლოვანების განახლების დროს თავიდან უნდა ითვლებოდეს: ქულა, თანხა და გასაცემი განრიგი);
- იშლებოდეს soft delete-ის სახით (ბაზაში არ უნდა იშლებოდეს, მაგრამ არ უნდა უჩანდეს ოპერატორს).

3) დიდი მონაცემების გამო შესასრულებელია სხვადასხვა ოპტიმალური მიდგომების გამოყენება

პროექტის შესაქმნელად გასათვალისწინებელი მნიშვნელოვანი კრიტერი-უმები:

- პროექტის სტრუქტურა;
- მონაცემთან სამუშაო პატერნები;

- ასინქრონული მიდგომები;
- ოპტიმიზირებული კოდი;
- API აღწერა;
- ბიზნეს ლოგიკის შესრულება.

გარდა ამისა, პროექტის მეორე ნაწილის პირობა ასეთია: - მართვის სისტემის საშუალებით შესაძლებელი უნდა იყოს ადამიანების რეგისტრაცია, მათ მიერ გაყიდული პროდუქციის აღრიცხვა და მათთვის ბონუსების გადათვლა.

სისტემამ უნდა უზრუნველყოს შემდეგი პროცესები:

➢ *თანამშრომლის რეგისტრაცია* (სისტემაში უნდა იყოს თანამშრომლის სარეგისტრაციო ფორმა, საიდანაც მოხდება მათ შესახებ შემდეგი ინფორმაციის შეტანა:

- თანამშრომლის უნიკალური კოდი - სისტემამ უნდა მიანიჭოს ავტო-მატურად;
- სახელი - აუცილებელი, max 50 სიმბოლო;
- გვარი - აუცილებელი, max 50 სიმბოლო;
- დაბადების თარიღი - აუცილებელი;
- სქესი - აუცილებელი;
- ფოტო-სურათი.

პირადობის დამადასტურებელი ინფორმაცია:

- საბუთის ტიპი - შესაძლო მნიშვნელობები: პირადობის მოწმობა;
- პასპორტი - აუცილებელი;
- საბუთის სერია - არააუცილებელი, max 10 სიმბოლო;
- საბუთის ნომერი - არააუცილებელი, max 10 სიმბოლო;
- გაცემის თარიღი - აუცილებელი;
- საბუთის ვადა - აუცილებელი;
- პირადი ნომერი - აუცილებელი, max 50 სიმბოლო;
- გამცემი ორგანო - არააუცილებელი, max 100 სიმბოლო.

საკონტაქტო ინფორმაცია:

- კონტაქტის ტიპი - შესაძლო მნიშვნელობებია: ტელეფონი, მობილური;
- ელ. ფოსტა, ფაქსი - აუცილებელი;
- საკონტაქტო ინფორმაცია - აუცილებელი, max 100 სიმბოლო;
- ინფორმაცია მისამართების შესახებ;

- მისამართის ტიპი - შესაძლო მნიშვნელობები: ფაქტობრივი მისამართი;
- რეგისტრაციის მისამართი;
- მისამართი - აუცილებელი, max 100 სიმბოლო.

თითოეული თანამშრომლისთვის უნდა მოხდეს განსაზღვრა ვისი რეკომენდაციით ხდება სისტემაში თანამშრომლის რეგისტრაცია. ეს შეიძლება იყოს სისტემაში უკვე რეგისტრირებული ნებისმიერი ადამიანი - ამ შემთხვევაში უნდა მოხდეს არჩევა უკვე რეგისტრირებული თანამშრომლების სიიდან, ან შეიძლება ინფორმაცია რეკომენდატორის შესახებ იყოს ცარიელი, რაც იმას ნიშნავს რომ თანამშრომლი სისტემაში რეგისტრირდება რეკომენდატორის გარეშე.

თითოეულ თანამშრომლს თავისი რეკომენდაციით შეუძლია მოიყვანოს მაქსიმუმ სამი ადამიანი, სისტემამ უნდა უზრუნველყოს იმის შეზღუდვა, რომ არ მოხდეს ერთიდაიგივე თანამშრომლის „ქვეშ“ სამზე მეტი ადამიანის რეგისტრაცია. ასევე მოყვანილი თანამშრომლების იერარქიის სიღრმე უნდა იყოს მაქსიმუმ 5 - ანუ თანამშრომლმა შეიძლება მოიყვანოს რეკომენდაციით ადამიანი, რომელიც ასევე თავისი რეკომენდაციით მოიყვანს ადამიანს და ა.შ. მაქსიმუმ 5 დონეზე. შესაბამისად ასეთ ჯგუფში შეიძლება სულ იყოს 1+3+9+27+81=121 ადამიანი. სისტემამ უნდა უზრუნველყოს მოცემული დონეების კონტროლი. შესაძლებელი უნდა იყოს უკვე რეგისტრირებული თანამშრომლების სიის დათვალიერება, ჩანაწერების რედაქტირება და წაშლა.

➢ *პროდუქციის რეგისტრაცია*

სისტემაში შესაძლებელი უნდა იყოს გასაყიდი პროდუქციის რეგისტრაცია, გასაყიდი პროდუქციის შესახებ უნდა მოხდეს შემდეგი ინფორმაციის შეყვანა:

- გასაყიდი პროდუქციის კოდი;
- გასაყიდი პროდუქციის დასახელება;
- გასაყიდი პროდუქციის ერთეულის ფასი.

➢ *თანამშრომლის გაყიდვები:*

სისტემაში შესაძლებელი უნდა იყოს გაყიდვების შესახებ ინფორმაციის შეყვანა. გაყიდვების შესახებ უნდა შევიდეს შემდეგი ტიპის ინფორმაცია:

- გაყიდვის უნიკალური ნომერი - მინიჭება უნდა მოხდეს სისტემის მიერ;
- ავტომატურად;

- დისტრიბუტორი, რომლის მიერაც იქნა შესრულებული პროდუქციის;
- გაყიდვა;
- გაყიდვის თარიღი;
- გაყიდული პროდუქცია (კოდი, დასახელება);
- გაყიდული პროდუქციის ღირებულება;
- გაყიდული პროდუქციის ერთეულის ფასი;
- გაყიდული პროდუქციის საერთო ფასი;

შესაძლებელი უნდა იყოს გაყიდვების ფილტრაცია: თანამშრომლის გაყიდვის თარიღისა და გაყიდული პროდუქციის მიხედვით.

- *თანამშრომლის ბონუსი:*

უნდა შეიქმნას ბონუსების გადათვლის ფორმა, სადაც უნდა მიეთითოს საწყისი თარიღი და საბოლოო თარიღი. გადათვლა ღილაკზე დაჭერის შედეგად სისტემამ უნდა მოახდინოს თანამშრომლების ბონუსების გადათვლა შემდეგი პრინციპით: თანამშრომლის ბონუსი წარმოადგენს წინასწარ განსაზღვრულ პერიოდზე მის მიერ შესრულებული გაყიდვების საერთო მოცულობის (თანხის ჯამური მნიშვნელობის) 10%-ს, პლიუს მისი რეკომენდაციით მოსული თანამშრომლების მიერ შესრულებული გაყიდვების ჯამური მოცულობის 5%, პლიუს მისი მეორე დონის რეკომენდაციით მოყვანილი თანამშრომლების მიერ შესრულებული გაყიდვების ჯამური მოცულობის 1%. ბონუსის ყოველი გადათვლის დროს, დათვლაში შემავალი გაყიდვები ხელმეორედ გადათვლის შემთხვევაში აღარ უნდა მონაწილეობდნენ გადათვლაში. იგივე ფორმიდან მომხმარებელს უნდა ჰქონდეს თანამშრომლის მიერ დაგროვილი ბონუსის შესახებ ინფორმაციის ნახვის საშუალება, ფორმაში შესაძლებელი უნდა იყოს თანამშრომლების ფილტრაცია: სახელით, გვარით, დაგროვილი ბონუსის მინიმალური და მაქსიმალური მნიშვნელობებით.

8.2. პროექტის მოდელი

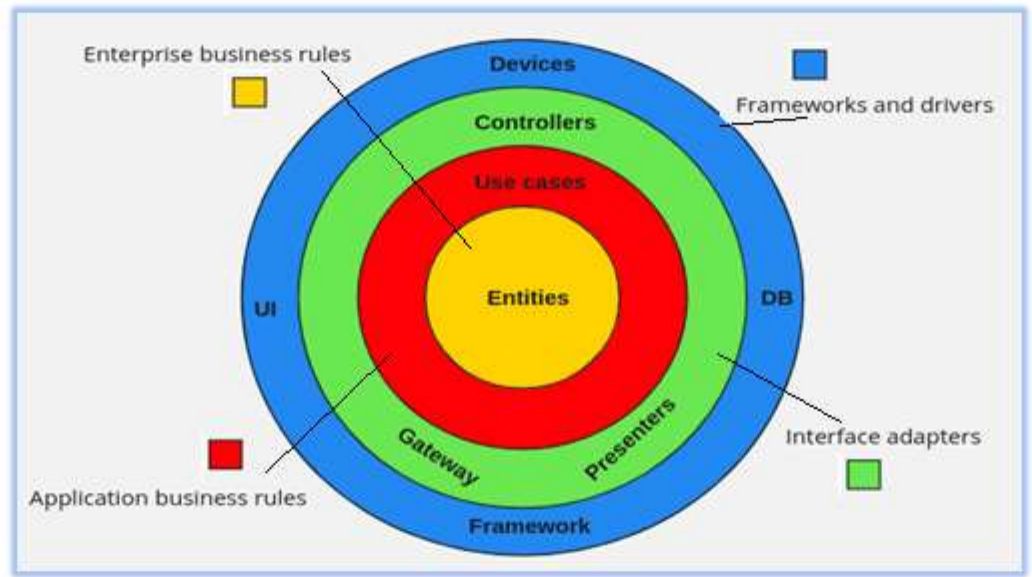
რაც შეეხება პროექტის მოდელს - გადაწყდა „სუფთა არქიტექტურაზე" შესრულება, რადგან ამ არქიტექტურაში ფენებს შორის არის მაქსიმალურად აბსტრაქტული დამოკიდებულება, რაც რაიმე ცვლილების საჭიროების შემთხვევაში კოდის მინიმალურ გადაკეთებას მოითხოვს.

დამხმარე ტექნოლოგიებად გამოყენებულია CQRS და Mediatr design Pattern. ტესტირების ნაწილში ინოვაციური ჰიბრიდული მეთოდოლოგიის

შემუშავება და ამ მეთოდოლოგიის ფარგლებში გამოყენებული დროის და პროგრამული კოდის ხარისხის კვლევა. კერძოდ რა ვადებშია ახალი ჰიბრიდული მეთოდოლოგიით მართვის საინფორმაციო სისტემებში პროექტების მართვა შესაძლებელი, მეთოდოლოგიის უპირატესობები, ნაკლოვანებები, რეკომენდაციები და სხვ.

8.2.1. სუფთა არქიტექტურის დაპროექტების მეთოდი

სუფთა არქიტექტურა (Clean Architecture) არის პროგრამული უზრუნველყოფის პროექტირების კონცეფცია, რომელიც ჰყოფს დიზაინის ელემენტებს რგოლების დონეზე (ნახ.8.1) [101]. სუფთა არქიტექტურის მიზანია დეველოპერებს შესთავაზოს კოდის ორგანიზება ისე, რომ იგი ასახავდეს ბიზნეს ლოგიკას, მაგრამ ინახავდეს მას მიწოდების მექანიზმისგან განცალკევებულად.

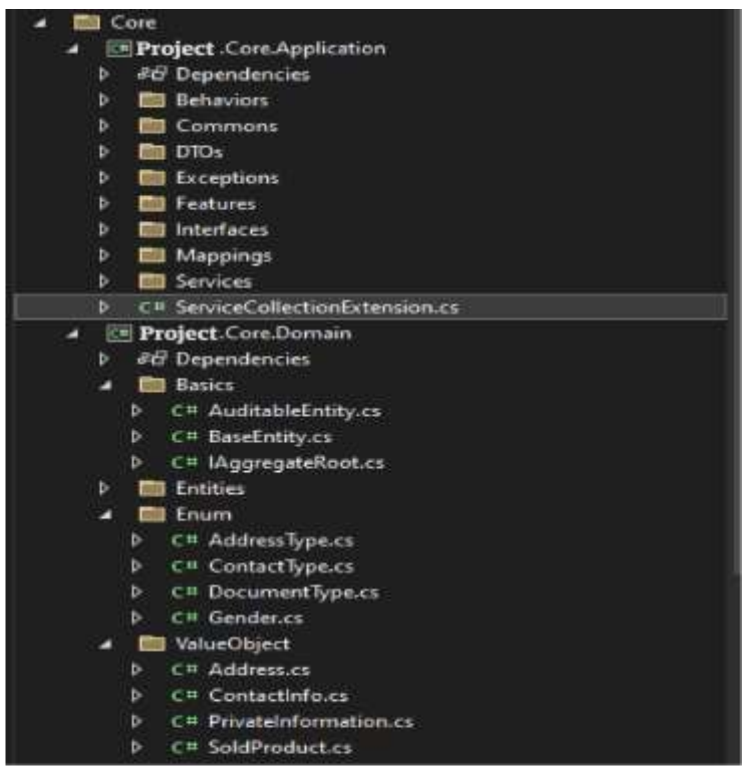


ნახ. 8.1. სუფთა არქიტექტურა

სუფთა არქიტექტურით, დომენის და აპლიკაციის ფენა არის დიზაინის ცენტრში, რომელიც ცნობილია როგორც სისტემის ბირთვი. ბიზნეს ლოგიკა განთავსდება ამ ორ ფენაში, თითოეული ფენა კი შეიცავს სხვადასხვა სახის ბიზნეს ლოგიკას. ბიზნეს ფენები არ უნდა იყოს დამოკიდებული პრეზენტაციისა და ინფრასტრუქტურის ფენებზე. იმის ნაცვლად, რომ ბიზნეს ლოგიკა იყოს დამოკიდებული მონაცემთა წვდომაზე ან სხვა ინფრასტრუქტურულ საკითხებზე, ეს დამოკიდებულება ინვერსიულია: ინფრასტრუქტურა

და განხორციელების დეტალები დამოკიდებულია აპლიკაციის ფენაზე. ეს ფუნქციონირება მიიღწევა აბსტრაქციების, ან ინტერფეისების განსაზღვრით Application-ის ფენაში, რომლებიც შემდეგ განხორციელდება ინფრასტრუქ-ტურის ფენაში განსაზღვრული ტიპების მიხედვით. ამ არქიტექტურის ვიზუალიზაცია შესაძლებელია კონცენტრული წრეების გამოყენებით. პროექ-ტის არქიტექტურად სწორედ „სუფთა არქიტექტურა“ იქნა არჩეული.

Core არ უნდა იყოს დამოკიდებული მონაცემთა ხელმისაწვდომობაზე და სხვა ინფრასტრუქტურულ საკითხებზე, ამიტომ დომენის ფენაში განთავსდა საწყისი ლოგიკა, ხოლო აპლიკაციის ფენაში ბიზნეს ლოგიკა და ტიპები.



ნახ. 8.2. სუფთა არქიტექტურა- Core შრე

- ***დომენის ფენა*** *(Domain Layer)*

დიზაინის მიხედვით, ეს ფენა ძალიან აბსტრაქტული და სტაბილურია. ეს ფენა შეიცავს დომენის ერთეულების მნიშვნელოვან რაოდენობას და არ უნდა იყოს დამოკიდებული გარე ბიბლიოთეკებსა და ჩარჩოებზე.

ჩვენს შემთხვევაში დომეინის ფენაში წარმოდგენილია საწყისი კლასები: *AuditableEntity და BaseEntity* (Basics ფაილი), სადაც განთავსებულია ისეთი მონაცემები, რაც ყველა ობიექტს აერთიანებს: უნიკალური id, ჩაწერის, წაშლის და განახლების დრო და ა.შ. ძირითადი კლასები: *Person, Issuance, Employee, Product, Sale.* მათგან აგრეგატებად მონიშნულია Person და Employee. გარდა ამისა არის *enum და Value Object* ტიპის კლასები. ValueObject ტიპის კლასში გაერთიანებულია ისეთი ობიექტები, რომლებიც თავადაა *entity* კლასის წევრები და არ საჭიროებს ბაზაში ცალკე ცხრილს. შესაბამისად არ საჭიროებს უნიკალურ ID-ს. მაგალითად, valueObject-ში ContactInfo კლასი არის Employee კლასის წევრი და შეიცავს ინფორმაციას კონტაქტის ტიპის (ტელ, მეილი, Fax) და მისამართის შესახებ.

- ***აპლიკაციის ფენა ან გამოყენებითი დონე*** *(Application Layer)*

ახორციელებს პროგრამული აპლიკაციის (დანართის) გამოყენებას დომენზე დაყრდნობით. გამოყენების შემთხვევა შეიძლება ჩაითვალოს, როგორც მომხმარებლის ინტერაქცია ინტერფეისზე (UI). მაგალითად, ჩვენს შემთხვევაში აპლიკაციას სჭირდება გაყიდვების სერვისზე წვდომა, შესაბამისად დაემატა ახალი ინტერფეისი interface ISaleRepository და შეიქმნება იმპლემენტაცია ინფრასტრუქტურაში.

გარდა ამისა აპლიკაციის ფენა შეიცავს:

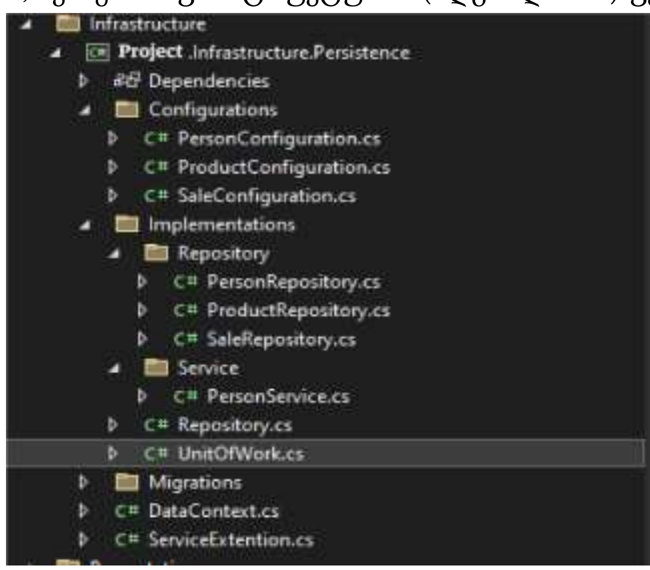
- ინტერფეისებს: ISaleRepository, IEmployeeRepository, IUnitOfWork, IpersonRepository;
- მოდელებს (DTO): GetPaginationDto, GetIssuanceOfPerson, GetPersonDto, GetPersonsQuery, GetIssuanceOfPersonQuery;
- სერვისებს/დამმუშავებლებს: IPersonService;
- გამონაკლისებს: ApplicationBaseException, EntityNotFoundException;
- მაპერებს. მაგალითად: CreateMap<Sale, GetSalesDto>().ReverseMap();
- ვალიდატორებს: ValidationBehaviour.

რაც შეეხება ინფრასტრუქტურის და პრეზენტაციის ფენებს ისინი დამოკიდებულია Core-ზე, მაგრამ არა ერთმანეთზე [102,103].

- ***ინფრასტრუქტურის ფენა*** *(Infrastructure Layer)*

პასუხისმგებელია განახორციელოს კონტრაქტები (ინტერფეისების იმპლე-მენტაცია), რომლებიც განსაზღვრულია აპლიკაციის ფენაში. ინფრასტრუქ-

ტურის ფენა მხარს უჭერს სხვა ფენას მესამე მხარის ბიბლიოთეკასა და სისტემებში აბსტრაქციებისა და ინტეგრაციების განხორციელებას. ყველა იმპლემენტაცია და დამოკიდებულება, რომელიც საჭიროა SQL სერვერის მოხმარებისთვის, იქმნება ინფრასტრუქტურის (მდგრადობის) ფენაზე.



ნახ. 8.4. სუფთა არქიტექტურა - ინფრასტრუქტურის შრე

➢ ***მდგრადობის ფენა*** *(Persistence Layer)*

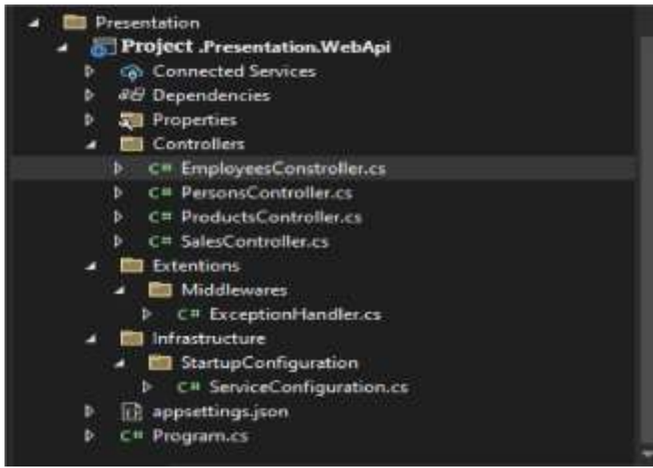
ამუშავებს მონაცემთა ბაზის პრობლემებს და მონაცემთა ხელმისაწვდომობის სხვა ოპერაციებს. ჩვენს შემთხვევაში ინფრასტრუქტურის ფენაში განთავსებულია *ბაზის კონფიგურაციის, მიგრაციის, სერვისების, რეპოზიტორების იმპლემენტაციის ფაილები და გარე ფენებთან საკომუნიკაციო ServiceExtention ფაილი.*

```
public static IServiceCollection AddPersistence(this IServiceCollection services, string databaseConnectionString)
    {  ArgumentNullException.ThrowIfNull(databaseConnectionString);
       services.AddDbContext<DataContext>(options =>
  options.UseSqlServer(databaseConnectionString));
       services.AddScoped<IUnitOfWork, UnitOfWork>();
       services.AddScoped<IPersonRepository, PersonRepository>();
       return services;  }
```

ამ ფაილის დახმარებით Core ფენა უკავშირდება დანარჩენ ფენებს.

➢ ***მომხმარებლის ინტერფეისის ფენა*** *(User Interface (Web/Api) Layer)*

მას ასევე უწოდებენ პრეზენტაციას. პრეზენტაციის ფენა შეიცავს აპლიკაციის UI ელემენტებს (გვერდებს, კომპონენტებს). ის ამუშავებს პრეზენტაციის (UI, API და ა.შ.) პრობლემებს. ეს ფენა პასუხისმგებელია მომხმარებლის გრაფიკული ინტერფეისის (GUI) მომხმარებელთან ან Json მონაცემების სხვა სისტემებთან ურთიერთობისთვის. ეს არის აპლიკაციის შესვლის წერტილი. (C. Martin , Clean Architecture: A Craftsman's Guide to Software Structure and Design, 2017) (https://betterprogramming.pub, 2023)



ნახ. 8.5. სუფთა არქიტექტურა - პრეზენტაციის შრე

ჩვენ შემთხვევაში მომხმარებლის ინტერფეისის ფენა შეიცავს:

- მაკონტროლებლებს: EmployeesController,PersonsController,ProductsController და SalesController;
- შუალედური პროგრამა (Middlewares) - რისი დახმარებითაც შესაძლებელია შეცდომების აღმოჩენა;
- კონფიგურაციებს.

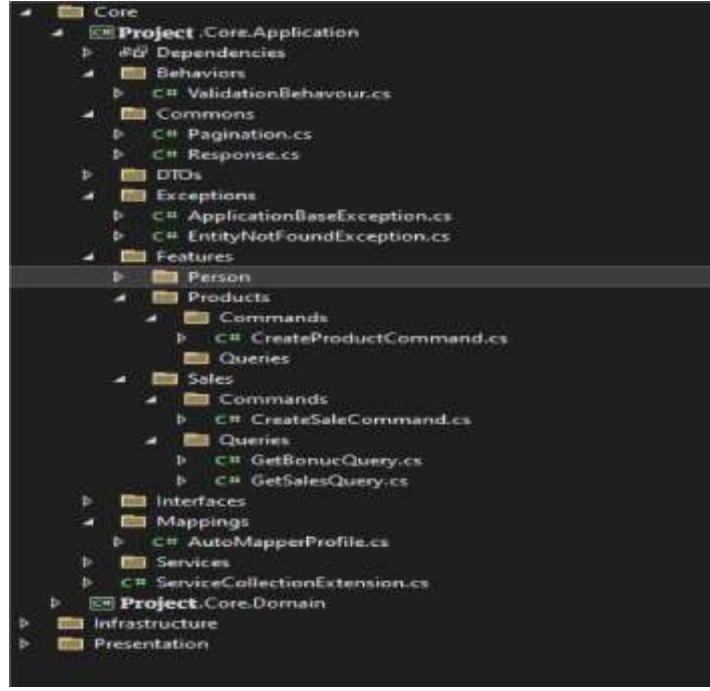
8.2.2. CQRS დიზაინ პატერნი

ბრძანების შეკითხვის პასუხისმგებლობის სეგრეგაცია (Command Query Responsibility Segregation - CQRS) პროგრამირების დიზაინის ნიმუშია, რომელიც განსხვავებულად განიხილავს მონაცემთა მოძიებას და მონაცემთა შეცვლას.

მონაცემებთან მომუშავე მეთოდს მხოლოდ ორი დავალების შესრულება შეუძლია. მეთოდს შეუძლია ან მოიძიოს ინფორმაცია ან შეცვალოს იგი. ობიექტზე ორიენტირებულობის თვალსაზრისით, ეს პარადიგმა ჰყოფს

პასუხისმგებლობებს ორ განსხვავებულ კლასად - ერთი წაკითხვისთვის და მეორე წაშლის, შექმნისა და განახლებისთვის.

ჩვენ შემთხვევაში CQRS pattern გამოყენებულია Core ფენის აპლიკაციის ნაწილში. Commands განთავსებული ფაილები მაგალითად: CreateSaleCommand პასუხიმგებელია მონაცემთა ბაზაში ინფორმაციის ჩაწერაზე, ხოლო Queries განთავსებული კლასები მაგალითად: GetBonusQuery და GetSalesQuery ამუშავებენ და მოაქვთ ინფორმაცია ბაზიდან ბონუსებისა და გაყიდვების შესახებ.



ნახ. 8.6. CQRS Pattern

8.2.3. მედიატორ დიზაინ პატერნი

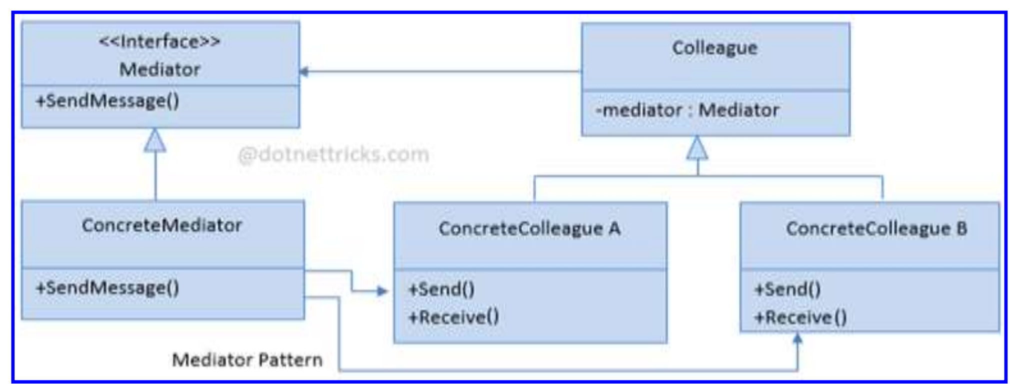
მედიატორის ნიმუშის დახმარებით პროგრამა მართავს დამოკიდებულებებს ობიექტებს შორის, რაც ხელს უშლის მათ ერთმანეთთან უშუალო კომუნიკაციას. კომუნიკაცია ხდება შუამავალი კლასის მეშვეობით. სერვისი უგზავნის თავის მოთხოვნას მედიატორს, რომელიც თავის მხრივ გადასცემს მას შესაბამის მოთხოვნის დამმუშავებელს დასამუშავებლად. შუამავლის არსებობით, შესაძლებელია გაერთიანდეს მოთხოვნები მათი დამმუშავებლებისგან.

გამგზავნს არ სჭირდება არაფერი იცოდეს დამმუშავებლის შესახებ. მედიატორის ნიმუში ეხმარება CQRS-ის განხორციელებას. ბრძანებები და მოთხოვნები ეგზავნება შუამავალს, რომელიც ასახავს მათ შესაბამის დამმუშავებლებს.

გამომდინარე იქიდან, რომ კვლევის მიზანია Agile ტექნოლოგიების ბაზაზე ტესტირების შესწავლა, შესასწავლად აღებული პროექტი კი თავისი მოცულობით უზარმაზარია და მისი ყოველი ნაწილი თანაბრად არ საჭიროებს ტესტირებას (მაგალითად, დარეგისტრირება და სისტემაში შესვლა, თანამშრომლის განაცხადი შვებულებასთან ან უქმე დღესთან დაკავშირებით და ა.შ.) ნაშრომისთვის პროექტიდან აღებულია ტესტირების ჭრილში ყველაზე მნიშვნელოვანი ნაწილი, რომელიც ფინანსურ ტრანზაქციებს ანუ სახელფასო სისტემას ეხება და კვლევაც Agile ტესტირების მიმართულებით გაგრძელდა პროექტის ამ ნაწილის გარშემო [95,104,105].

8.2.4. UML დიაგრამა

Unified Modeling Language (UML) მოდელირების ენა ჩვენ დეტალურად განვიხილეთ წიგნის წინა ნაწილში. იგი პროგრამული უზრუნველყოფის სისტემების და მათი კომპონენტების ვიზუალურად წარმოსაჩენად გამოიყენება. ესაა გრაფიკული გზები სისტემების სტრუქტურების და ურთიერთქმედებების აღსაქმელად. UML-დიაგრამებს ფართო გამოყენება აქვს დიდი პროგრამული აპლიკაციების პროექტირებაში, პროგრამული უზრუნველოფის განვითარების სასიცოცხლო ციკლის დიზაინის ანალიზისა და განხორციელების ეტაპებზე. 8.7 ნახაზზე წარმოდგენილია პროექტისთვის შექმნილი UML დიაგრამა.



ნახ. 8.7. UML-ის კლასების დიაგრამის ფრაგმენტი
ჩვენი სისტემისთვის

თავი 9. პროექტის რეალიზაცია

9.1. თანამშრომლის მოდელი პროექტში - Domain ფენა (დასატესტი პროგრამული კოდი)

Domain ფენა დაყოფილია ორ ნაწილად Basic და Entities. Entities_ში განთავსებულია ძირითადი კლასები:

```
public class Person: AuditableEntity
  {
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string PersonalNumber { get; private init; }
    public int Age { get; set; }
    public int Score { get; set; }
  }
```

სახელფასო მოდელი პროექტში:

```
public class Issuance: AuditableEntity
  {
    public Guid PersonId { get; set; }
    public int Year { get; set; }
    public int Month { get; set; }
    public int Salary { get; set; }
    public int ScoreAtMoment { get; set; }
  }
```

ორივე მოდელი მემკვიდრეა AuditableEntity კლასის, აღნიშნული კლასი განთავსებულია Basic საქაღალდეში

```
public abstract class AuditableEntity : BaseEntity
  {
    public virtual DateTime DateCreated { get; set; }
    public virtual DateTime? DateUpdated { get; set; }
    public virtual DateTime? DateDeleted { get; set; }
  }
```

ხოლო baseEntity კლასი რომელიც ისევ Basic საქაღალდეშია განთავსებული გამოიყურება ასე:

```
public abstract class BaseEntity
  {
```

```
    public virtual Guid Id { get; init; }
  }
```

Core-ის Application ფენაში წარმოდგენილი Dto-ები ვალიდაციისთვის გამოყენებულია FluentValidation ბიბლიოთეკა:

```
public class GetPersonDto
  {
    public string FirstName { get; private set; }
    public string LastName { get; private set; }
    public string PersonalNumber { get; private init; }
    public int Age { get; set; }
    public int Score { get; set; }
  }
public class GetIssuanceOfPersonDto
  {
    public List<GetIssuanceOfPerson> Items { get; set; } =
                                    new List<GetIssuanceOfBeneficiary>();
  }
 public class GetIssuanceOfPerson
  {
    public Guid Id { get; set; }
    public Guid PersonId { get; set; }
    public int Year { get; set; }
    public int Month { get; set; }
    public int Salary { get; set; }
    public int ScoreAtMoment { get; set; }
  }
```

დიდ მონაცემებთან სამუშაოდ გათვალისწინებულია მონაცემების ნაწილ-ნაწილ წამოღება:

```
public class GetPaginationDto<T>
  {  public List<T> Items { get; set; }
    public int PageIndex { get; set; }
    public int PageSize { get; set; }
    public int TotalPages { get; set; }
    public int TotalCount { get; set; }
    public bool HasPreviousPage { get; set; }
   }
```

CQRS პატერნის გათვალისწინებით:

```
public class CreatePersonCommand
{
    public class Request : IRequest<GenericResultModel<bool>>
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string PersonalNumber { get; set; }
        [Required]
        public int Age { get; set; }
    }
    public class Handler : IRequestHandler<Request, GenericResultModel<bool>>
    {
  public async Task<GenericResultModel<bool>> Handle(Request request,
                               CancellationToken cancellationToken)
        {
            GenericResultModel<bool> result = new GenericResultModel<bool>();
Var ExistPerson = await _unit.PersonRepository.IfExistPerson(request.PersonalNumber);
            if(ExistPerson)
            {
                result.Result = false;
                result.ErrCode = 1;
                result.Message = "ასეთი მომხმარებელი უკვე დარეგისტრირებულია!";
                return result;
            }
            var person= _mapper.Map<Person>(request);
           person.Score = await _ personService.CalculateScore(request.Age);
            cancellationToken.ThrowIfCancellationRequested();
            int added = await _unit.PersonRepository.CreateAsync(person);
            var  personId = person.Id;
            await _ personService.CalculateYearlyIssuance(personId);
            if (added == 0)
            {   result.Result = false;
                result.ErrCode = 2;
                result.Message = "ტექნიკური შეცდომა!";
                return result;
            }
```

```
            result.Result = true;
            result.Message = $"მომხმარებელი {request.FirstName} {request.LastName}
                                        წარმატებით დაემატა";
            return result;
        }
    }
    public class Validator : AbstractValidator<Request>
    {
     public Validator()
      {
       RuleFor(x => x.PersonalNumber)
          .NotNull().WithMessage("პირადი ნომერი ცარიელია")
          .Matches("^[0-9]*$").WithMessage("ველი შედგება მხოლოდ ციფრებისგან");
          RuleFor(x => x.FName)
             .NotEmpty().WithMessage("სახელი ცარიელია");
          RuleFor(x => x.LName)
             .NotEmpty().WithMessage("გვარი ცარიელია");
          RuleFor(x => x.Age)
             .Must(y => y > 17).WithMessage("ასაკი უნდა აღემატებოდეს 18 წელს!");
       }
    }
 }
public class DeletePersonCommand
 {
    public record Request(Guid PersonId) : IRequest<GenericResultModel<bool>>;
    public class Handler : IRequestHandler<Request, GenericResultModel<bool>>
    {
       public async Task<GenericResultModel<bool>> Handle(Request request,
                    CancellationToken cancellationToken)
       {
          GenericResultModel<bool> result = new GenericResultModel<bool>();
          var person = await _unit.PersonRepository.ReadAsync(request. PersonId);
          if (person == null)
             throw new EntityNotFoundException("ასეთი პიროვნება არ არსებობს");
          person.DateDeleted = DateTime.Now;
          var person = _mapper.Map<Person>(person);
          await _unit. PersonRepository.UpdateAsync(request. PersonId, personId);
```

```
            var issuances = await _unit.IssuanceRepository.GetAsync(i =>
                                                i.BeneficiaryId == beneficiary.Id);
            if (issuances.Any())
            {
               foreach (var issuance in issuances)
               {
                  issuance.DateDeleted = DateTime.Now;
                  await _unit.IssuanceRepository.Update(issuance);
               }
            }
            result.Result = true;
            result.Message  =  $"მომხმარებელი  {person.FirstName}  {  person.LastName}
                                                წარმატებით გაუქმდა";
            return result;
         }
      }
   }
   public class UpdatePersonCommand
   {
      public class Request : IRequest<GenericResultModel<bool>>
      {
         public Guid PersonId { get; private set; }
         public string PersonalNumber { get; set; }
         public string FirstName { get; set; }
         public string LastName { get; set; }
         public int Age { get; set; }
         public void SetId(Guid beneficiaryId) => this. PersonId = personId;
      }
      public class Handler : IRequestHandler<Request, GenericResultModel<bool>>
      {
       private readonly IUnitOfWork _unit;
       private readonly IMapper _mapper;
       private readonly IPersonService _personService;
       public async Task<GenericResultModel<bool>> Handle(Request request,
                        CancellationToken cancellationToken)   {
        GenericResultModel<bool> result = new GenericResultModel<bool>();
        var updatedPerson = await _unit. PersonRepository.ReadAsync(request. PersonId);
```

```
 if (updatedPerson == null || updatedPerson.DateDeleted != null)
      throw new EntityNotFoundException("ასეთი პიროვნება არ არსებობს");
   var person = _mapper.Map<Person>(request);
   person.Score = await _personService.CalculateScore(request.Age);
   person.DateCreated = updatedPerson.DateCreated;
   cancellationToken.ThrowIfCancellationRequested();
   //ხელახლა გენერირდება, ძველი იშლება
if (updatedPerson.Age != request.Age)
{
  var issuances = await _unit.IssuanceRepository.GetAsync(i =>
          i. PersonId == updatedPerson.Id);
     if (issuances.Any())
     {
       var dateUpdated = DateTime.Now;
       var month = DateTime.Now.Month;
       var day = DateTime.Now.Day;
       foreach (var issuance in issuances)
       {
         if (issuance.DateDeleted == null)
         {
           issuance.DateDeleted = dateUpdated;
           await _unit.IssuanceRepository.Update(issuance);
         }
       }
       //29,30,31 რიცხვები ან თებერვალის 27, 28, 29,
       //დაგენერირდება შემდეგი თვიდან
       if (day > 28 || (month == 2 && day > 26)) {
         for (int i = month + 1; i <= 12; i++)
           await GenerateIssuance(dateUpdated, i);
       }
       else
       {
         //ნებისმიერი სხვა თარიღი, დაგენერირდება მიმდინარე თვიდან
         for (int i = month; i <= 12; i++)
           await GenerateIssuance(dateUpdated,i);
       }
     }
```

```
            }
    public class GetPersonsQuery
    {
       public sealed record  Request : IRequest<GetPaginationDto<GetPersonDto>>
       {
          public int PageIndex { get; set; }
          public int PageSize { get; set; }
          public string FirstName { get; set; }
          public string LastName { get; set; }
          public string PrivateNumber { get; set; }
       }
    public class Handler : IRequestHandler<Request, GetPaginationDto<GetPersonDto>>
     {
      public async Task<GetPaginationDto<GetPersonDto>>
                      Handle(Request request, CancellationToken cancellationToken)
       {
         Var persons = await _unit. PersonRepository.FilterAsync(request.PageIndex,
            request.PageSize, firstName: request.FirstName, lastName: request.LastName);
          return _mapper.Map<GetPaginationDto<GetPersonDto>>( persons);
        }
      }
    }
   public class GetIssuanceOfPersonQuery
    {
     public record Request(Guid beneficiaryId) : IRequest<GenericResultModel
                   <GetIssuanceOfPersonDto>>;
     public class Handler:IRequestHandler<Request,
               GenericResultMode <GetIssuanceOfPersonDto>>
     {
      public async Task<GenericResultModel<GetIssuanceOfPersonDto>> Handle(Request
             request, CancellationToken cancellationToken)
      {
GenericResultModel<GetIssuanceOfPersonDto>result=new
                            GenericResultModel<GetIssuanceOfPersonDto>();
result.Result = new GetIssuanceOfPersonDto();
         var person = await _unit. PersonRepository.ReadAsync(request. personId);
```

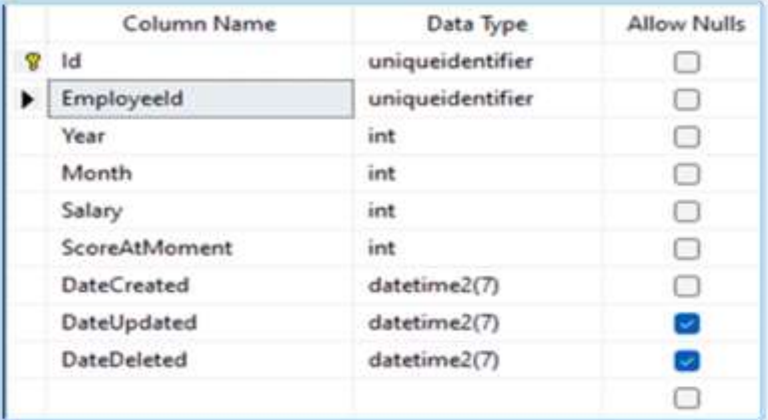
```
if (person == null || person.DateDeleted != null) throw new
                    EntityNotFoundException(" ვერ მოიძებნა");
var issuances = (from issuance in await _unit.IssuanceRepository.GetAsync(i =>
        i. PersonId == person.Id && i.DateDeleted == null)
            select new GetIssuanceOfPerson
            {
                PersonId = issuance. PersonId,
                Id = issuance.Id,
                Year = issuance.Year,
                Month = issuance.Month,
                Salary = issuance.Salary,
                ScoreAtMoment = issuance.ScoreAtMoment
            }).ToList();
    result.Result.Items.AddRange(issuances.ToList());
    return result;
  }
 }
}
```

- **მონაცემთა ბაზა**
  - *თანამშრომლების ცხრილი*

Column Name	Data Type	Allow Nulls
Id	uniqueidentifier	☐
FirstName	nvarchar(20)	☐
LastName	nvarchar(20)	☐
PersonalNumber	nvarchar(11)	☐
Age	int	☐
Score	int	☐
DateCreated	datetime2(7)	☐
DateUpdated	datetime2(7)	☑
DateDeleted	datetime2(7)	☑
		☐

ნახ. 9.1. Employee Table

- *სახელფასო ცხრილი*



	Column Name	Data Type	Allow Nulls
🔑	Id	uniqueidentifier	☐
▶	EmployeeId	uniqueidentifier	☐
	Year	int	☐
	Month	int	☐
	Salary	int	☐
	ScoreAtMoment	int	☐
	DateCreated	datetime2(7)	☐
	DateUpdated	datetime2(7)	☑
	DateDeleted	datetime2(7)	☑
			☐

ნახ. 9.2. Issuence Table

პირობის მეორე ნაწილის რეალიზაცია:

```
public class Employee : AuditableEntity, IAggregateRoot
  {
     public Guid? ParentId { get; set; }
     public string FirstName { get;  set; }
     public string LastName { get;  set; }
     public string Code { get; set; }
     public DateTime BirthDt { get; set; }
     public Gender Sqesi { get; set; }
     public PrivateInformation PrivateInformation { get; set; }
     public PersonAddress PersonAddress { get; set; }
     public ContactInfo ContactInfo { get;  set; }
}
```

სადაც “ValueObject”:

```
public record class PrivateInformation(DocumentType DocumentType,
    string DocumentSeries,
    string DocumentNumber, DateTime IssueDate,
    DateTime ReleaseDate, string PrivateNumber,
    string IssuingAuthority);
public record class PersonAddress(AddressType addressType, string street);
public record class ContactInfo(ContactType contactType, string contactInfo);
```

- გაყიდული პროდუქტის მოდული:

```
public class Product : AuditableEntity
  {
      public int UnitPrice { get; set; }
      public string Name { get; set; }
      public string Code { get; set; }
  }
```

ხოლო გაყიდვების მოდული:

```
public class Sale : AuditableEntity
  {
      public string Code { get; set; }
      public Guid EmployeeId { get; set; }
      public DateTime SaleDate { get; set; }
      public string SoldCode { get; set; }
      public SoldProduct SoldProduct { get; set; }
//კოდი დასახელება და ერთეულის ფასი შევამოწმო არსებობს თუ არა
      public int Price { get; set; }
//საერთო ფასი კონკრეტული პროდუქტის
      public int TotalPrice { get; set; }
//ზოგადად რაც აქვს გაყიდული საერთო ფასი უნდა დავიანგარიშო
  }
```

enum ფაილები:

```
public enum AddressType
  {
      Factual = 0,
      Registration = 1
  }
 public enum ContactType
  {
      Tel = 0,
      Mob = 1,
      Email = 2,
      Fax = 3
  }
public enum DocumentType
  {
```

```
        IDcard = 0,
        Passport = 1
    }
  public enum Gender
    {
        Female = 0,
        Male = 1
    }
```

სადაც “ValueObject”:

```
     public record class SoldProduct(string code, string  name,int unitPrice);
```

- თანამშრომლის შექმნის მოდული:

```
  public record CreateEmployeeCommand
    {
        public sealed record Request : IRequest<Unit>
        {
            public Guid? ParentId { get; set; }
            public string FirstName { get; set; }
            public string LastName { get; set; }
            public DateTime BirthDate { get; set; }
            public Gender Gender { get; set; }
            public  SetPrivateInformationDto PrivateInformation { get; set; }
            public SetContactInfoDto ContactInfo { get;  set; }
            public SetAddressDto Address { get; set; }
        }
    public sealed class Handler : IRequestHandler<Request, Unit>
        {
            public async Task<Unit> Handle(Request request, CancellationToken
                                                                     cancellationToken)
            {
                var employee = _mapper.Map<Employee>(request);
                employee.Code = await _unit.EmployeeRepository.GenerateDigitalCode();
                bool checkHierearchy = await _unit. EmployeeRepository. checkHierarchy
                                                                     (employee.ParentId);
                if(checkHierearchy)
                    await _unit. EmployeeRepository.CreateAsync(employee);
                return Unit.Value;
```

```
    }
}
public class EmployeeValidator : AbstractValidator<Request>
{
    private readonly IUnitOfWork _unit;
    public EmployeeValidator(IUnitOfWork unit)
    {
        _unit = unit;
        RuleFor(x => x.FirstName).Must(y => y.Length <60 )
            .NotEmpty().WithMessage("სახელი ცარიელია");
        RuleFor(x => x.LastName).Must(y => y.Length < 60)
            .NotEmpty().WithMessage("გვარი ცარიელია");
        RuleFor(x => x.BirthDate).NotEmpty();
        RuleFor(x => x.Gender).NotEmpty();
        RuleFor(x => x.PrivateInformation.DocumentType)
            .NotEmpty().WithMessage("საბუთის ტიპი ცარიელია");
        RuleFor(x => x.PrivateInformation.DocumentSeries).Must(y => y.Length < 10)
            .WithMessage("საბუთის სერია უნდა იყოს 10 სიმბოლოზე ნაკლები");
        RuleFor(x => x.PrivateInformation.DocumentNumber).Must(y => y.Length < 10)
         .WithMessage("საბუთის ნომერი უნდა იყოს 10 სიმბოლოზე ნაკლები");
RuleFor(x => x.PrivateInformation.IssueDate)
            .NotEmpty().WithMessage("გაცემის თარიღი უნდა იყოს შევსებული");
RuleFor(x => x.PrivateInformation.ReleaseDate)
           .NotEmpty().WithMessage("საბუთის ვადა უნდა იყოს შევსებული");
RuleFor(x => x.PrivateInformation.PrivateNumber).Must(y => y.Length < 51)
  .WithMessage("პირადი ნომერი შედგება არაუმეტეს 50 სიმბოლოსგან")
            .NotEmpty().WithMessage("პირადი ნომერი არ უნდა იყოს ცარიელი");
RuleFor(x => x.PrivateInformation.IssuingAuthority).Must(y => y.Length < 101)
   .WithMessage("გამცემი ორგანო შედგება არაუმეტეს 100 სიმბოლოსგან");
RuleFor(x => x.ContactInfo.contactType)
           .NotEmpty().WithMessage("კონტაქტის ტიპი ცარიელია");
RuleFor(x => x.ContactInfo.contactInfo).Must(y => y.Length < 101)
    .WithMessage("საკონტაქტო ინფორმაცია არ აღემატება 100 სიმბოლოს")
          .NotEmpty().WithMessage("კონტაქტის ტიპი ცარიელია");
RuleFor(x => x.Address.addressType)
          .NotEmpty().WithMessage("მისამართის ტიპი ცარიელია");
RuleFor(x => x.Address.street).Must(y => y.Length < 101)
```

```
            .WithMessage("მისამართი არ უნდა აღემატებოდეს 100 სიმბოლოს")
            .NotEmpty().WithMessage("მისამართი ცარიელია");
        }
    private async Task<bool> IfExistCode(string Code, CancellationToken cancellationToken)
        {
           return await _unit. EmployeeRepository.CheckAsync(x => x.Code == Code);
        }
      }
}
```

თანამშრომლის შექმნის დროს გამოყენებულია მნიშვნელოვანი ვალიდაციები ფუნქციების სახით:

```
    public async Task<bool> checkHierarchy(Guid? ParentId)
        { var secondHierarchy = _context.Persons.Where(x => x.ParentId ==
                ParentId).ToList();
          bool checkHierarchy =  (secondHierarchy.Count > 3) ? false : true;
          if(checkHierarchy)
           {
             foreach(var secondEmployee in secondHierarchy)
             {
          var thirdHierarchy = _context. Persons.Where(x => x.ParentId ==
                 secondEmployee.Id).ToList();
        if (thirdHierarchy.Count > 3)
              {
                 checkHierarchy = false;
                 return checkHierarchy;
              }
            else {
             foreach (var thirdEmployee in thirdHierarchy)
              {
                var fourthHierarchy = _context. Persons.Where(x => x.ParentId ==
                           thirdEmployee.Id).ToList();
                if (fourthHierarchy.Count > 3)
                  {
                      checkHierarchy = false;
                      return checkHierarchy;
                  }
                 else
```

```
                {
                  foreach (var fourthEmployee in fourthHierarchy)
                  {
                   var fifthHierarchy = _context. Persons.Where(x => x.ParentId ==
                               fourthEmployee.Id).ToList();
                   if (fifthHierarchy.Count > 3)
                     {
                         checkHierarchy = false;
                         return checkHierarchy;
                     }
                    else
                     {
                      foreach (var sixthEmployee in fifthHierarchy)
                       {
                        var sixthHierarchy = _context.v.Where(x => x.ParentId ==
                                  sixthEmployee.Id).ToList();
                        if (sixthHierarchy.Count > 0)
                         {
                             checkHierarchy = false;
                             return checkHierarchy;
                         }
                        else
                        {
                         checkHierarchy = true;
                        }
                       }
                     }
      }}}}}}
       return checkHierarchy;
    }
  //თანამშრომლის id _ის მიხედვით ითვლის ჯამურ გაყიდულ პროდუქციას
    public async Task<int> CountToTalPrice(Guid employeeId,int price)
    {
       var salesList= _context.Sales.Where(x => x. EmployeeId == employeeId).ToList();
       int sum = 0;
       foreach (var item in salesList)
       {
```

```
                sum += item.Price;
            }
            sum += price;
            return sum;
        }
    //ბონუსების დათვლა
      public async Task<double> CountBonuc(Guid employeeId)
        {
            double bonusEmployee = 0;
            double bonusFirstHierarcy = 0;
            double bonusSecondHierarcy = 0;
            double totalBonus = 0;
            var sumEmployee = 0;
            var sumFirstHierarcy = 0;
            var sumSecondHierarcy = 0;
            Sale firstHierarcy;
            Sale secondHierarcy;
            var sales = _context.Sales.ToList();
            foreach (Sale item in sales)
                sumEmployee += item.Price;
            bonusEmployee += sumEmployee * 0.1;
            var salesFirstHierarcy = _context. Employees.Where(x => x.ParentId ==
                        employeeId).ToList();
           foreach (Employee item in salesFirstHierarcy)
            {
              firstHierarcy = (Sale)_context.Sales.Where(x => x. EmployeeId ==
                            item.ParentId);
              sumFirstHierarcy += firstHierarcy.Price;
              var salesSecondHierarcy = _context. Employees.Where(x => x.ParentId ==
                            firstHierarcy. EmployeeId);
            foreach (Employee secondEmployee in salesSecondHierarcy)
                {
                  secondHierarcy = (Sale)_context.Sales.Where(x => x.EmployeeId ==
                          secondEmployee.ParentId);
                  sumSecondHierarcy += secondHierarcy.Price;
                }
            }
```

```
            bonusFirstHierarcy *= sumFirstHierarcy * 0.05;
            bonusSecondHierarcy *= sumSecondHierarcy * 0.01;
            totalBonus = bonusEmployee + bonusFirstHierarcy + bonusSecondHierarcy;
            return totalBonus;
        }
    }
```

როგორც უკვე აღინიშნა გუნდი მუშაობდა „თანამშრომლების მიღება გადინების და განსაკუთრებული პირობებით სახელფასო სისტემის“ (ბონუსები) პროგრამული უზრუნველყოფის განვითარებაზე. გამომდინარე იქიდან რომ საქმე ეხება ფინანსურ ტრანზაქციებს და აქ მნიშვნელოვანია პროგრამული კოდის ხარისხი, გუნდმა გადაწყვიტა გაეძლიერებინა ტესტირების ეტაპი. ე.წ. „მუტანტი ტესტების“ დახმარებით [14,15].

9.2. პროექტის მომზადება ტესტირებისთვის

Agile ტესტირების სასიცოცხლო ციკლი რამდენიმე ეტაპს მოიცავს, ეს დამოკიდებულია პროექტის ტიპზე და გუნდზე:

პირველ ეტაპზე აუცილებლად ხდება ადამიანთა იდენტიფიკაცია ტესტირებისთვის, ტესტირების ხელსაწყოების დაყენება და რესურსების დაგეგმვა:

ა) ბიზნეს პროექტის ჩამოყალიბება;

ბ) პირობების დადგენა და პროექტის მოცულობა;

გ) ძირითადი მოთხოვნები და გამოყენების შემთხვევები;

დ) ერთი ან მეტი სავარაუდო არქიტექტურის ჩამოთვლა;

ე) რისკის იდენტიფიცირება;

ვ) ხარჯთაღრიცხვა და წინასწარი პროექტის მომზადება.

Agile ტესტირების მეთოდოლოგიის მეორე ფაზა არის სამშენებლო პროცესები, ტესტირების უმეტესობა ამ ფაზაში ხდება. ეს ფაზა, შეიძლება ითქვას, რომ არის პროცესების ერთობლიობა გამოსავლის შესაქმნელად. ამ პროცესის დროს Agile გუნდი მიჰყვება პრიორიტეტულ მოთხოვნათა პრაქტიკას: ყოველი განმეორებით პროცესის დროს, ისინი იღებენ შესასრულებელი სამუშაოებიდან დარჩენილ ყველაზე მნიშვნელოვან მოთხოვნებს და ახორციელებენ მათ.

მშენებლობის პროცესი იყოფა ორად, *დამადასტურებელი ტესტირება* და *საკვლევი ტესტირება*. დამადასტურებელი ტესტირება კონცენტრირებულია იმის შემოწმებაზე, რომ სისტემა ასრულებს დაინტერესებული მხარეების განზრახვას, როგორც ეს აღწერილია გუნდისთვის დღემდე და შესრულებულია გუნდის მიერ.

მიუხედავად ამისა, საკვლევ ტესტირებას შეუძლია აღმოაჩინოს პრობლემა,, რომელიც დამადასტურებელმა ჯგუფმა გამოტოვა ან უგულებელყო. საკვლევ ტესტირებაში ტესტერი განსაზღვრავს პოტენციურ პრობლემებს დეფექტების ისტორიების სახით.

ტესტირებისთვის შეირჩა ჯგუფში 4 კადრი, თავდაპირველად მოხდა გაცნობა ბიზნეს ლოგიკასთან, ჩამოყალიბდა ტესტების პროექტი, რაც მოიცავს ერთეულის ტესტების წერას თანამშრომლების მიღება გადინების სისტემასთან ხოლო ხელფასის დარიცხვის ფუნქციონალისთვის ერთეულის ტესტების გარდა ჯგუფმა გადაწყვიტა დამატებითი ტესტირების მექანიზმის შემოღება. ამ შემთხვევაში დამადასტურებელი ტესტების ჭრილში აღებულია მოდულური ტესტები, ხოლო საგამოძიებო ტესტებისთვის მუტანტი ტესტები [106].

9.4. Agile ტესტირება პროექტის ფარგლებში

9.4.1. მოდულური ტესტები

ვირტუალი ორგანიზაციის თანამშრომლების მიღება-გადინების Unit test. .NET Core Framework_ზე უნდა შეიქმნას Unit Test Project(.NET Core) პროექტი, რომელსაც ექნება სერვისის დანიშნულება.

პროექტში შევქნათ კლასი, რომელიც პროგრამის გაშვების დროს მონაცემთა ბაზას შეავსებს სატესტო მონაცემებით [14, 52].

```
1.public class EmployeeDataDBInitializer
2. {
3.    public EmployeeDataDBInitializer   ()
4.     {        }
5.    public void Seed(BlogDBContext context)
6.      {
7.        context.Database.EnsureDeleted();
8.        context.Database.EnsureCreated();
```

```
9.       context.Employee.AddRange(
10.                  new Employee() { Firstname = "Test 1", LastName = "Test Lastname
             1",  CreatedDate = DateTime.Now },
11.           new Employee() { Title = "Test 2", LastName = "Test Lastname
             2",  CreatedDate = DateTime.Now }
12.          );
13.            context.SaveChanges();
14.          }
15.         }
```

პროექტში ვაინსტალირებთ "Fluent Assertions" ბიბლიოთეკას და ვქმნით EmployeeUnitTestController კლასს. პირველ რიგში ვწერთ ბაზასთან მაკავშირებელ ატრიბუტებს:

```
1. public class EmployeeUnitTestController
2. {
3.   private EmployeeRepository repository;
4.   public static DbContextOptions<EmployeeDBContext> dbContextOptions { get; }
5.   public static string connectionString = "Server=ABCD;Database=EmployeeDB;UID=sa
                   ;PWD=xxxxxxxxxx;";
6.   static EmployeeUnitTestController()
7.    {
8.     dbContextOptions = new DbContextOptionsBuilder<EmployeeDBContext>().
              UseSqlServer(connectionString).Options;
9.           }
10.         }
```

constructor-ს ექნება შემდეგი სახე:

```
1. public EmployeeUnitTestController()
2. {
3.   var context = new EmployeeDBContext(dbContextOptions);
4.   EmployeeDataDBInitializer db = new EmployeeDataDBInitializer();
5.   db.Seed(context);
6.   repository = new EmployeeRepository(context);
7. }
```

ამის შემდეგ უკვე შესაძლებელია Unit Test Case-ების დაწერა. ტესტების წერის პროცესში გასათვალისწინებელია სამი მთავარი კრიტერიუმი:

მოწესრიგება (Arrange), განხორციელება (Act) და აპლიკაცია (Assert). პირველი დავწეროთ Get BY ID მეთოდის ტესტები, რაც შეეხება მეთოდს ის id მიხედვით აბრუნებს ობიექტს. ამ შემთხვევაში მოცემული ერთი მეთოდისთვის უნდა დაიწეროს რამდენიმე ტესტი, თითოეული მათგანს განსხვავებული პასუხისმგებლობები აქვს.

Task_GetEmployeeById_Return_OkResult() – კარგი შედეგი,

Task_GetEmployeeById_Return_NotFoundResult() – ვერ მოიძებნა შედეგი,

Task_GetEmployeeById_Return_BadRequestResult() – რექუესთის ხარვეზი,

Task_GetEmployeeById_MatchResult() – შედეგების დადარება.

```
1. [Fact]
2.      public async void Task_GetPostById_Return_OkResult()
3.      {
4.          //Arrange
5.          var controller = new EmployeeController(repository);
6.          var postId = 2;
7.          //Act
8.          var data = await controller.GetEmployee (Id);
9.          //Assert
10.                 Assert.IsType<OkObjectResult>(data);
11.             }
12.             [Fact]
13.         public async void Task_GetEmployeeById_Return_NotFoundResult()
14.             {
15.                 //Arrange
16.                 var controller = new EmployeeController(repository);
17.                 var postId = 3;
18.                 //Act
19.                 var data = await controller.GetEmployee(Id);
20.                 //Assert
21.                 Assert.IsType<NotFoundResult>(data);
22.             }
23.         [Fact]
24.         public async void Task_GetEmployeeById_Return_BadRequestResult()
25.             {
26.                 //Arrange
27.                 var controller = new EmployeeController(repository);
```

```
28.                 int? Id = null;
29.                 //Act
30.                 var data = await controller.GetEmployee (Id);
31.                 //Assert
32.                 Assert.IsType<BadRequestResult>(data);
33.             }
34.         [Fact]
35.         public async void Task_GetEmployeeById_MatchResult()
36.             {
37.                 //Arrange
38.                 var controller = new EmployeeController(repository);
39.                 int? postId = 1;
40.                 //Act
41.                 var data = await controller.GetEmployee(Id);
42.                 //Assert
43.                 Assert.IsType<OkObjectResult>(data);
44.                 var okResult = data.Should().BeOfType<OkObjectResult>().Subject;
45.                 var employee = okResult.Value.Should().BeAssignableTo
                                                  <EmployeeViewModel>().Subject;
46.                 Assert.Equal("Test FirstName 1", employee.FirstName);
47.                 Assert.Equal("Test LastName 1", employee. LastName);
48.             }
```

გაყიდვების შემთხვევაში SalesController ტესტები იქნება ასეთი:

```
using Moq;
using Xunit;
public class SalesControllerTests
{
    private readonly SalesController _controller;
    private readonly Mock<IMediator> _mediatorMock;

    public SalesControllerTests()
    {
        _mediatorMock = new Mock<IMediator>();
        _controller = new SalesController(_mediatorMock.Object);
    }
```

```
    [Fact]
    public async Task Add_Should_Return_StatusCode201()
    {
        // Arrange
        var cancellationToken = new CancellationToken();
        var request = new CreateSaleCommand.Request();

        // Act
        var result = await _controller.Add(request, cancellationToken);

        // Assert
        Assert.IsType<StatusCodeResult>(result);
        var statusCodeResult = (StatusCodeResult)result;
        Assert.Equal(201, statusCodeResult.StatusCode);
    }
    [Fact]
    public async Task GetBonucs_Should_Return_Bonucs_Value()
    {
        // Arrange
        var request = new GetBonucQuery.Request();
        var expectedResult = 10.5;

        _mediatorMock
            .Setup(mediator => mediator.Send(request))
            .ReturnsAsync(expectedResult);

        // Act
        var result = await _controller.GetBonucs(request);

        // Assert
        Assert.Equal(expectedResult, result);
    }
}
```

სხვა შემთხვევაში controller ტესტებს აღარ წარმოვადგენთ და გადავიდეთ უშუალოდ სერვისების გატესტვაზე. CalculateSalary სერვისის ერთეულის (Unit) ტესტები. ამ შემთხვევაში მოცემული ერთი მეთოდისთვის უნდა დაიწეროს რამდენიმე ტესტი, თითოეულს განსხვავებული პასუხისმგებლობები აქვს.

CalculateSalary_MonthIs12_ReturnsCorrectSalary() – კარგი შედეგი,
CalculateSalary_MonthIsNot12_ReturnsCorrectSalary()- არასწორი შედეგი.

```
[Test]
public async Task CalculateSalary_MonthIs12_ReturnsCorrectSalary()
{
    // Arrange
    int score = 100;
    int month = 12;
    // Act
    int salary = await _personService.CalculateSalary(score, month);
    // Assert
    Assert.AreEqual(110, salary);
        // ვივარაუდოთ, რომ ქულა 100 უნდა დააბრუნოს 100 + (100 * 0.1) = 110
}

[Test]
public async Task CalculateSalary_MonthIsNot12_ReturnsCorrectSalary()
{
    // Arrange
    int score = 200;
    int month = 6;
    // Act
    int salary = await _personService.CalculateSalary(score, month);
    // Assert
    Assert.AreEqual(250, salary);
                // ვივარაუდოთ, რომ ქულა 200 უნდა დააბრუნოს 250
}
```

შემდეგ დავწეროთ CalculateScore სერვისის ტესტები. ეს მეთოდი ითვლის თანამშრომლის ქულას:

```
[Test]
public async Task CalculateScore_ReturnsCorrectScore()
{
    // Arrange
    int age = 25;
    // Act
    int score = await _ personService.CalculateScore(age);
```

```
        // Assert
        Assert.AreEqual(100, score); // 18-დან 35 წლამდე ასაკი უნდა დაბრუნდეს 100
    }
```

გამოცდილების მიხედვით წლის ბოლოს დასარიცხი ხელფასის გამომთვლელი სერვისის ტესტები:

CalculateYearlyIssuance_MonthIs12_CreatesCorrectIssuances() - კარგი შედეგი,

CalculateYearlyIssuance_MonthIsNot12_CreatesCorrectIssuances() - არასწორი შედეგი.

```
    [Test]
    public async Task CalculateYearlyIssuance_MonthIs12_CreatesCorrectIssuances()
    {
        // Arrange
        Guid personId = Guid.NewGuid();
        _unitOfWorkMock.Setup(uow => uow.
                    PersonRepository.ReadAsync(employeeId))
          .ReturnsAsync(new Person{ Score = 100 });
        _unitOfWorkMock.Setup(uow =>
                    uow.IssuanceRepository.CreateAsync(It.IsAny<Issuance>()))
          .Returns(Task.CompletedTask);
        // Act
        await _ personService.CalculateYearlyIssuance(personId);
        // Assert
        _unitOfWorkMock.Verify(uow => uow.IssuanceRepository.
                    CreateAsync(It.IsAny<Issuance>()), Times.Exactly(11));
   // Add additional assertions to check the properties of the created issuances if desired
    }
    [Test]
    public async Task CalculateYearlyIssuance_MonthIsNot12
                                          _CreatesCorrectIssuances()
    {
        // Arrange
        Guid personId = Guid.NewGuid();
        _unitOfWorkMock.Setup(uow => uow. PersonRepository.ReadAsync(personId))
          .ReturnsAsync(new Person { Score = 200 });
        _unitOfWorkMock.Setup(uow => uow.IssuanceRepository
                  .CreateAsync(It.IsAny<Issuance>()))
          .Returns(Task.CompletedTask);
```

```
            // Act
            await _ personService.CalculateYearlyIssuance(personId);
            // Assert
            _unitOfWorkMock.Verify(uow =>
uow.IssuanceRepository.CreateAsync(It.IsAny<Issuance>()), Times.Exactly(12));
        }
    }
}
```

პირობის მეორე ნაწილიდან თავდაპირველად განვიხილოთ იერარქიის სერვისი (რადგან დანარჩენი ყველა ფუნქცია მის შედეგებზეა დამოკიდებული), ამიტომ აქცენტი გავაკეთოთ ამ სერვისის ყველა შესაძლო შემთხვევებზე:

CheckHierarchy_ReturnsTrue_WhenHierarchyIsValid() - აბრუნებს true თუ იერარქია სწორია,

CheckHierarchy_ReturnsFalse_WhenHierarchyHasMoreThanThreeLevels()-აბრუნებს false თუ იერარქია 3-ზე მეტი დონისგან შედგება,

CheckHierarchy_ReturnsTrue_WhenNoHierarchyExists() - აბრუნებს true თუ იერარქია არ არსებობს,

CheckHierarchy_ReturnsFalse_WhenThirdHierarchyHasMoreThanThreeItems()-აბრუნებს false როდესაც მესამე იერარქია 3-ზე მეტი ელემენტი აქვს,

CheckHierarchy_ReturnsFalse_WhenFourthHierarchyHasMoreThanThreeItems()-აბრუნებს false, როდესაც მეოთხე იერარქიას 3-ზე მეტი ელემენტი აქვს,

CheckHierarchy_ReturnsFalse_WhenFifthHierarchyHasMoreThanThreeItems()-აბრუნებს false თუ მეხუთე იერარქიას 3-ზე მეტი ელემენტი აქვს.

CheckHierarchy_ReturnsFalse_WhenSixthHierarchyHasItems() - აბრუნებს false თუ მე-6 იერარქიას აქვს ელემენტი.

```
[Test]
public async Task CheckHierarchy_ReturnsTrue_WhenHierarchyIsValid()
{
   // Arrange
  var parentId = Guid.NewGuid();
  var secondHierarchy = new List<Person> { new Person(), new Person(), new Person() };
  var thirdHierarchy = new List<Person> { new Person(), new Person(), new Person() };
  var fourthHierarchy = new List<Person> { new Person(), new Person(), new Person() };
  var fifthHierarchy = new List<Person>();
  _dbContextMock.Setup(mock => mock.Persons)
          .Returns(CreateDbSetMock(secondHierarchy.Concat(thirdHierarchy)
```

```
        .Concat(fourthHierarchy).Concat(fifthHierarchy)));
    // Act
    var result = await _yourClassName.CheckHierarchy(parentId);
    // Assert
    Assert.IsTrue(result);
}
[Test]
public async Task CheckHierarchy_ReturnsFalse_WhenHierarchy
                HasMoreThanThreeLevels()
{
    // Arrange
    var parentId = Guid.NewGuid();
    var secondHierarchy = new List<Person> { new Person(), new Person(), new
            Person(), new Person() };
    _dbContextMock.Setup(mock => mock.Persons)
            .Returns(CreateDbSetMock(secondHierarchy));
    // Act
    var result = await _yourClassName.CheckHierarchy(parentId);
    // Assert
    Assert.IsFalse(result);
}
[Test]
public async Task CheckHierarchy_ReturnsTrue_WhenNoHierarchyExists()
{
    // Arrange
    var parentId = Guid.NewGuid();
    var secondHierarchy = new List<Person>();
    _dbContextMock.Setup(mock => mock.Persons)
        .Returns(CreateDbSetMock(secondHierarchy));
    // Act
    var result = await _yourClassName.CheckHierarchy(parentId);
    // Assert
    Assert.IsTrue(result);
}
[Test]
Public async Task CheckHierarchy_ReturnsFalse_WhenThird
                        HierarchyHasMoreThanThreeItems()
```

```
{
    // Arrange
    var parentId = Guid.NewGuid();
    var secondHierarchy = new List<Person> { new Person(), new Person() };
    var thirdHierarchy = new List<Person> { new Person(), new Person(),
                                            new Person(), new Person() };
    _dbContextMock.Setup(mock => mock.Persons)
       .Returns(CreateDbSetMock(secondHierarchy.Concat(thirdHierarchy)));
    // Act
    var result = await _yourClassName.CheckHierarchy(parentId);
    // Assert
    Assert.IsFalse(result);
}
[Test]
public async Task
CheckHierarchy_ReturnsFalse_WhenFourthHierarchyHasMoreThanThreeItems()
{
    // Arrange
    var parentId = Guid.NewGuid();
    var secondHierarchy = new List<Person> { new Person() };
    var thirdHierarchy = new List<Person> { new Person() };
    var fourthHierarchy = new List<Person> { new Person(), new Person(),
                          new Person(), new Person() };
  _dbContextMock.Setup(mock => mock.Persons)    .Returns(CreateDbSetMock(second
                 Hierarchy.Concat(thirdHierarchy).Concat(fourthHierarchy)));
    // Act
    var result = await _yourClassName.CheckHierarchy(parentId);
    // Assert
    Assert.IsFalse(result);
}
[Test]
public async Task CheckHierarchy_ReturnsFalse_WhenFifth
                                  HierarchyHasMoreThanThreeItems()
{   // Arrange
    var parentId = Guid.NewGuid();
    var secondHierarchy = new List<Person> { new Person() };
    var thirdHierarchy = new List<Person> { new Person() };
```

```
            var fourthHierarchy = new List<Person> { new Person() };
            var fifthHierarchy = new List<Person> { new Person(), new Person(), new Person(),
                    new Person() };
            _dbContextMock.Setup(mock => mock.Persons)
                .Returns(CreateDbSetMock(secondHierarchy.Concat(thirdHierarchy)
                    .Concat(fourthHierarchy).Concat(fifthHierarchy)));
            // Act
            var result = await _yourClassName.CheckHierarchy(parentId);
            // Assert
            Assert.IsFalse(result);
        }
        [Test]
        public async Task CheckHierarchy_ReturnsFalse_WhenSixthHierarchyHasItems()
        {
            // Arrange
            var parentId = Guid.NewGuid();
            var secondHierarchy = new List<Person> { new Person() };
            var thirdHierarchy = new List<Person> { new Person() };
            var fourthHierarchy = new List<Person> { new Person() };
            var fifthHierarchy = new List<Person> { new Person() };
            var sixthHierarchy = new List<SomeModel> { new SomeModel() };
            _dbContextMock.Setup(mock => mock.Persons)
                .Returns(CreateDbSetMock(secondHierarchy.Concat(thirdHierarchy)
                    .Concat(fourthHierarchy).Concat(fifthHierarchy)));
            _dbContextMock.Setup(mock => mock.SomeModels)
                .Returns(CreateDbSetMock(sixthHierarchy));
            // Act
            var result = await _yourClassName.CheckHierarchy(parentId);
            // Assert
            Assert.IsFalse(result);
        }
    }
}
```

- ფასის დათვლის სერვისის მოდულის ტესტები:
  - ფასის დათვლა როდესაც გაყიდული პროდუქცია არსებობს

```
        [Test]
```

```
public async Task CountToTalPrice_ReturnsSumOfSalesAndPrice
        _WhenSalesExist()
{
    // Arrange
    var employeeId = Guid.NewGuid();
    var price = 10;
    var sales = new List<Sale>
    {
        new Sale { EmployeeId = employeeId, Price = 5 },
        new Sale { EmployeeId = employeeId, Price = 8 },
        new Sale { EmployeeId = employeeId, Price = 12 }
    };
    _dbContextMock.Setup(mock => mock.Sales)
        .Returns(CreateDbSetMock(sales));
    // Act
    var result = await _yourClassName.CountToTalPrice(employeeId, price);
    // Assert
    Assert.AreEqual(35, result);
}
```

- ფასის დათვლა როდესაც გაყიდული პროდუქცია არ არსებობს

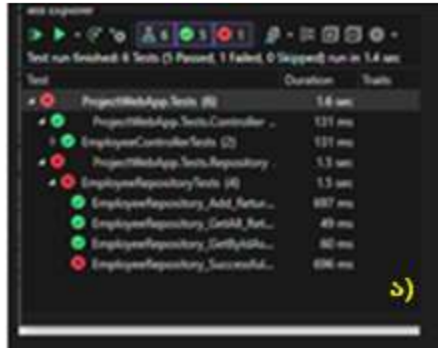
```
[Test]
public async Task CountToTalPrice_ReturnsPrice_WhenNoSalesExist()
{
    // Arrange
    var employeeId = Guid.NewGuid();
    var price = 10;
    _dbContextMock.Setup(mock => mock.Sales)
        .Returns(CreateDbSetMock(new List<Sale>()));
    // Act
    var result = await _yourClassName.CountToTalPrice(employeeId, price);
    // Assert
    Assert.AreEqual(10, result);
}
```

- სწორი ჯამური ფასის დათვლის სერვისის მოდულის ტესტი

```
[Test]
public async Task CountTotalPrice_ReturnsCorrectSum()
```

```
{
   // Arrange
   Guid employeeId = Guid.NewGuid();
   int price = 50;
   var salesList = new List<Sale>
  { new Sale { EmployeeId = employeeId, Price = 10 },
   new Sale { EmployeeId = employeeId, Price = 20 },
   new Sale { EmployeeId = Guid.NewGuid(), Price = 30}
                    //Another sale with different employeeId
   };
   var dbContextMock = new Mock<EmployeeDBContext>();
   var salesDbSetMock = new Mock<DbSet<Sale>>();
   salesDbSetMock.As<IQueryable<Sale>>().Setup(m =>
                    m.Provider).Returns(salesList.AsQueryable().Provider);
   salesDbSetMock.As<IQueryable<Sale>>().Setup(m =>
                    m.Expression).Returns(salesList.AsQueryable().Expression);
   salesDbSetMock.As<IQueryable<Sale>>().Setup(m =>
                    m.ElementType).Returns(salesList.AsQueryable().ElementType);
   salesDbSetMock.As<IQueryable<Sale>>().Setup(m  =>  m.GetEnumerator()).Returns(()
            => salesList.AsQueryable().GetEnumerator());
   dbContextMock.Setup(m => m.Sales).Returns(salesDbSetMock.Object);
   var yourClassName = new YourClassName(dbContextMock.Object);
   // Act
   int result = await yourClassName.CountTotalPrice(employeeId, price);
   // Assert
   int expectedSum = salesList.Where(x => x. EmployeeId == employeeId).Sum(x =>
           x.Price) + price;
   Assert.AreEqual(expectedSum, result);
}
```

9.3-ა,ბ ნახაზებზე მოცემულია მოდულური ტესტების მაგალითები არასწორი და სწორი შედეგებისათვის.

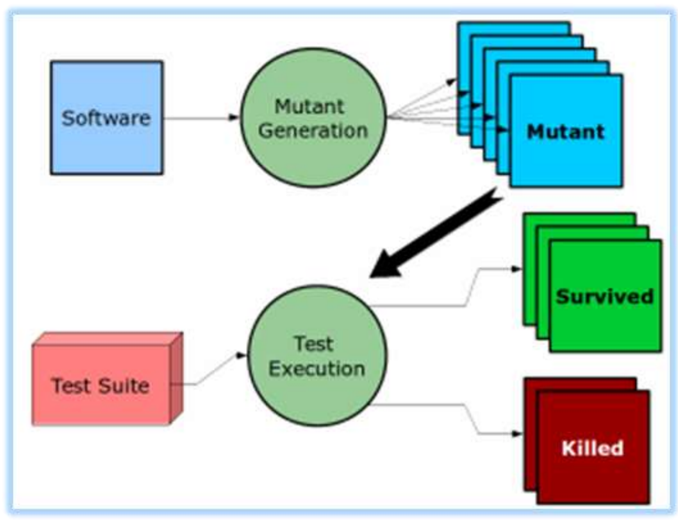


ნახ. 9.3. მოდულური ტესტების შედეგები
ა- ხარვეზით; ბ) სწორი შედეგით

9.4.2 მუტაციური ტესტები

იმ შემთხვევაში თუ გუნდს აქვს ეჭვი მოდულური ტესტების სრულ-ფასოვან სისწორეში, კერძოდ, თუ არსებობს ვარაუდი, რომ შეიძლება ტესტს ჰქონდეს „ბაგი“ ან „ხვრელი“. ტესტების შესამოწმებლად უნდა გამოიყენოს მუტაციის ტესტირება.

მუტაციის ტესტირება არის პროგრამული უზრუნ-ველყოფის ტესტირება, რომ-ლის დროსაც დეველოპერი ახორციელებს ცვლილებებს ანუ „მუტანტებს“ საწყისი კოდის გარკვეული ნაწილის-თვის და ამოწმებს, მის მიერ დაწერილ ტესტებს შეუძლია თუ არა შეცდომების იდენ-ტიფიცირება (ნახ.9.4).



ნახ. 9.4. მუტაციური ტესტები

მუტანტის დანერგვის შემდეგ, თავდაპირველი ერთეულის ტესტით უნდა შემოწმდეს მუტანტის კოდი. თუ ერთ ან მეტმა ტესტმა აჩვენა შეცდომა, ეს ნიშნავს რომ ტესტმა შეძლო „მოკლა მუტანტი“, რაც მიუთითებს იმაზე, რომ ტესტის შემთხვევა ეფექტურია და შეუძლია ამ ხარვეზის იდენტიფიცირება.

თუ ყველა ტესტში, მუტანტმა შეძლო "გადარჩენა". ეს მიუთითებს იმაზე, რომ არ არსებობს მოედულური ტესტი, რომელსაც შეუძლია ამ შეცდომის გამოვლენა და ეს პრობლემაა. ამის გადასაჭრელად, დეველოპერმა უნდა დაამატოს ახალი ტესტები ან განაახლოს არსებული წარუმატებლობის სცენარის დასაფარად.

რაც უფრო მეტი „მუტანტების“ რაოდენობა შეუძლია მოკლას ტესტმა, მით უფრო ძლიერი და ეფექტური იქნება მოდულური ტესტის ნაკრები. ამიტომ, მუტაციის ტესტირება მიზნად ისახავს ხარისხის და გამძლეობის დადგენას და გაზრდას. თუმცა, აუცილებლად გასათვალისწინებელი საკითხი ასეთი ტიპის ტესტირების დროს არის უზრუნველყოფა, რომ შეტანილი ცვლილებები გავლენას არ მოახდენს მთელ კოდზე. ამდენად, მუტანტები რაც შეიძლება პატარა უნდა იყოს. არსებობს მუტანტების რამდენიმე განსხვავებული ტიპი, რომლებიც შესაძლოა დაინერგოს მუტაციის ტესტირებისთვის. მუტანტების ყველაზე გავრცელებული ტიპებია [106-109, 98-99, 104]:

1) *მუტაციური მნიშვნელობები* – ცვლადის მნიშვნელობის შეცვლა პროგრამაში შეცდომების გამოსავლენად. ყველაზე გავრცელებული სტრატეგია ცვლადის მნიშვნელობის გაზრდა ან შემცირებაა;

2) *ლოგიკური ოპერატორების მუტაცია* – ამ ტიპის მუტაციაში შეიძლება „პირობების“ შეცვლა პროგრამული კოდის გადაწყვეტილების შესამოწმებლად ლოგიკური ოპერატორების ინვერსიების გამოყენებით. მაგალითად, თუ პირობა ამოწმებს მნიშვნელობას ტოლობაზე („ == “), შეიძლება იგი შეიცვალოს „ != “ (არ უდრის) მნიშვნელობით;

3) აპლიკაციის შესრულების წესის შეცვლა;

4) მუტაციური აპლიკაციები – კოდის აპლიკაციის წაშლა ან ასლი, რომლებიც სიმულაციას უწევს შეცდომებს კოდის სხვაგან კოპირების ჩასმისას [56-65]. მაგალითად, თუ კოდში გაქვთ გამოხატულება: a + b

მუტაციის ტესტი დროებით შეიცვლება შემდეგით: a – b

9.4. მუტაციური ტესტირების ავტომატიზაცია

მუტაციის ტესტირება ძალზე შრომატევადი და რთულია ხელით შესასრულებლად. პროცესის დასაჩქარებლად მიზანშეწონილია მიმართოთ ავტომატიზაციის ხელსაწყოებს.

მუტაციის ტესტირება შეიძლება ფუნდამენტურად დაიყოს 3 ტიპად - აპლიკაციის მუტაცია, გადაწყვეტილების მუტაცია და მნიშვნელობის მუტაცია.

1) აპლიკაციის მუტაცია – დეველოპერმა ამოჭრა და ჩასვა კოდის ნაწილი;

2) მნიშვნელობის მუტაცია – პირველადი პარამეტრების მნიშვნელობები შეცვლილია;

3) გადაწყვეტილების მუტაცია – საკონტროლო აპლიკაციები შესაცვლელია.

- *მუტაციის ქულა*

მუტაციის ქულა განისაზღვრება, როგორც „მოკლული“ მუტანტების (აღმოჩენილი შეცდომები) პროცენტი მუტანტების საერთო რაოდენობასთან.

მუტაციის ქულა = (მოკლული მუტანტები / მუტანტების საერთო რაოდენობა) * 100

ტესტის შემთხვევები მუტაციის ადეკვატურია, თუ ქულა არის 100%.

- *მუტაციური ტესტირების უპირატესობები*

მუტაციის ტესტირების უპირატესობები შემდეგია:

- ეს არის ძლიერი მიდგომა პროგრამის მაღალი ხარისხით შესასწავლად;
- ამ ტესტირებას შეუძლია პროგრამის ყოვლისმომცველი ტესტირება;
- მუტაციის ტესტირებას მოაქვს შეცდომების გამოვლენის კარგი დონე პროგრამული უზრუნველყოფის შემქმნელისთვის;
- ამ მეთოდს აქვს შესაძლებლობა აღმოაჩინოს პროგრამის ყველა ხარვეზი;
- მომხმარებლები რომლებიც სარგებლობენ ამ ტესტირებით ყველაზე საიმედო და სტაბილური სისტემის მიღების მაღალი ალბათობა აქვთ.

- *მუტაციური ტესტირების უარყოფითი მხარეები*

მეორეს მხრივ, მუტანტის ტესტირების უარყოფითი მხარეებია:

- მუტაციის ტესტირება ძალიან ძვირი და შრომატევადია, რადგან არსებობს მრავალი მუტანტური პროგრამა, რომელიც უნდა შეიქმნას;
- ვინაიდან ეს შრომატევადია, სამართლიანია იმის თქმა, რომ ეს ტესტირება არ შეიძლება გაკეთდეს ავტომატიზაციის ხელსაწყოს გარეშე;

- თითოეულ მუტაციას ექნება იგივე რაოდენობის ტესტის შემთხვევები, რაც ორიგინალ პროგრამას. ამრიგად, მუტანტის პროგრამების დიდი რაოდენობა შეიძლება გახდეს საჭირო.

➢ ***Stryker ბიბლიოთეკა***

კოდისთვის მუტაციების დანერგვა რთული და შრომატევადია, განსაკუთრებით თუ მუტანტების რაოდენობა ძალიან დიდია. ოპტიმიზაციის თვალსაზრისით C# საშუალებას აძლევს დეველოპერებს გამოიყენონ Stryker ბიბლიოთეკა, რათა ავტომატურად შეიტანონ ცვლილებები კოდში. Stryker-ის ბიბლიოთეკა იყენებს ზემოხსენებულ ტექნიკას კოდის მუტაციისთვის და მოდულური ტესტების ეფექტურობის შესამოწმებლად.

Stryker არის ღია კოდის ინსტრუმენტი, რომელიც შესაძლებელია გამოყენებულ იქნას მუტანტების დასანერგად Javascript, Scala და C#-ში დაწერილი კოდისთვის. Stryker.NET არის ვერსია, რომელიც შესაძლებელია დეველოპერმა გამოიყენოს .NET Core და .NET-ში დაწერილი პროექტებისთვის. ის მხარს უჭერს 30-ზე მეტ მუტაციის ტიპს და აწარმოებს შედეგებს წასაკითხად HTML ფორმატში [113].

➢ ***ფუნქციონალის მუტაციური ტესტები***

წარმოდგენილი მუტანტი ტესტები იძლევა სხვადასხვა მოდიფიკაციის მაგალითებს, რომლებიც შეიძლება განხორციელდეს თავდაპირველი მოდულის ტესტში, რათა შეამოწმოს მისი მდგრადობა და უზრუნველყოს მისი ეფექტურობა პოტენციური შეცდომების ან პრობლემების აღმოჩენაში.

თავდაპირველად განვიხილოთ Controller-ის მუტანტი ტესტები:

```
public class EmployeeUnitTestController
{
   private EmployeeRepository repository;
   public static DbContextOptions<EmployeeDBContext> dbContextOptions { get; }
   public static string connectionString =
                    "Server=ABCD;Database=EmployeeDB;UID=sa;PWD=xxxxxxxxxx;";
   static EmployeeUnitTestController()
   {
      dbContextOptions = new DbContextOptionsBuilder<EmployeeDBContext>()
                                             .UseSqlServer(connectionString).Options;
   }
```

```
    public EmployeeUnitTestController()
    {
        // Mutant: წაშალა EmployeeDataDBInitializer ---------
        var context = new EmployeeDBContext(dbContextOptions);
        // EmployeeDataDBInitializer db = new EmployeeDataDBInitializer();
        // db.Seed(context);
        repository = new EmployeeRepository(context);
    }
}
```

ამით ამოწმებს არის თუ არა გათვლილი მოდულურ ტესტებში შემთხვევა, როდესაც მონაცემების შევსება შეუძლებელია.

შემდეგ განვიხილოთ CheckHierarchy_ReturnsTrue_WhenHierarchyIsValid() მოდულური ტესტისთვის დაწერილი მუტანტი ტესტების რამდენიმე ვარიანტი:

- **parentId შეიცვალოს Guid.Empty -ით.**

```
// Arrange
var parentId = Guid.Empty;
var secondHierarchy = new List<Person> { new Person(), new Person(), new Person() };
var thirdHierarchy = new List<Person> { new Person(), new Person(), new Person() };
var fourthHierarchy = new List<Person> { new Person(), new Person(), new Person() };
var fifthHierarchy = new List<Person>();
_dbContextMock.Setup(mock => mock.Persons)
    .Returns(CreateDbSetMock(secondHierarchy.Concat(thirdHierarchy)
        .Concat(fourthHierarchy).Concat(fifthHierarchy)));
// Act
var result = await _yourClassName.CheckHierarchy(parentId);
// Assert
Assert.IsTrue(result);
```

- **შეიცვალოს ერთ-ერთი Person ობიექტი იერარქიის მეორე დონეზე.**

```
// Arrange
var parentId = Guid.NewGuid();
var secondHierarchy = new List<Person> { new Person(), new Person(), new Person() };
secondHierarchy[1] = null; // Modify the second Person object
var thirdHierarchy = new List<Person> { new Person(), new Person(), new Person() };
var fourthHierarchy = new List<Person> { new Person(), new Person(), new Person() };
var fifthHierarchy = new List<Person>();
_dbContextMock.Setup(mock => mock.Persons)
```

```
        .Returns(CreateDbSetMock(secondHierarchy.Concat(thirdHierarchy)
            .Concat(fourthHierarchy).Concat(fifthHierarchy)));
    // Act
    var result = await _yourClassName.CheckHierarchy(parentId);
    // Assert
    Assert.IsTrue(result);
```

გარდა ამ შემთხვევებისა, გამომდინარე პირობის სირთულიდან, არის მრავალი ვარიანტი, ამიტომ განვიხილოთ კიდევ რამდენიმეს:

1-ტესტემა შეცვალოს პირობა და მეორე იერარქიაში დაამატოს 3-ზე მეტი ელემენტი;

2-ტესტემა შეცვალოს პირობა და მესამე იერარქიაში დაამატოს 3-ზე მეტი ელემენტი;

3-ტესტემა შეცვალოს პირობა და მეოთხე იერარქიაში დაამატოს 3-ზე მეტი ელემენტი;

4-ტესტემა შეცვალოს პირობა და მეხუთე იერარქიაში დაამატოს 3-ზე მეტი ელემენტი;

5- ტესტმა გაზარდოს მეექვსე იერარქიის წევრების რაოდენობა.

```
public async Task ReturnsTrue_WhenHierarchyIsValid()
    // Arrange
    var parentId = Guid.NewGuid();
    var secondHierarchy = new List<Person> { new Person() };
    var thirdHierarchy = new List<Person> { new Person() };
    var fourthHierarchy = new List<Person> { new Person() };
    var fifthHierarchy = new List<Person> { new Person(), new Person(), new Person(),
                    new Person() };
    _dbContextMock.Setup(mock => mock.Persons)
        .Returns(CreateDbSetMock(secondHierarchy.Concat(thirdHierarchy)
            .Concat(fourthHierarchy).Concat(fifthHierarchy)));
    // Act
    var result = await _yourClassName.CheckHierarchy(parentId);
    // Assert
    Assert.IsTrue(result);
}
```

- ***ფასის დათვლის სერვისის მუტაციური ტესტები:***

CountTotalPrice_ReturnsSumOfSalesAndPrice_NegativePrices() – ამოწმებს საბოლოო ფასის უარყოფითობის შემთხვევაში არის თუ არა გათვალისწინებული მუტაციური ტესტები;

CountTotalPrice_ReturnsSumOfSalesAndPrice_ZeroPrices() – ამოწმებს საბოლო ფასში 0-ის დაბრუნების შემთხვევაში არსებობს თუ არა ტესტი რომელსაც აქვს გათვალისწინებული ეს შემთხვევა

```
[Test]
public async Task CountTotalPrice_ReturnsSumOfSalesAndPrice_NegativePrices()
{
   // Arrange
   var employeeId = Guid.NewGuid();
   var price = 100;
   var salesList = new List<Sale>
   {
      new Sale { EmployeeId = employeeId, Price = -50 },
      new Sale { EmployeeId = employeeId, Price = -75 },
      new Sale { EmployeeId = employeeId, Price = -100 }
   };
   _dbContextMock.Setup(mock => mock.Sales)
      .Returns(CreateDbSetMock(salesList));
   // Act
   var result = await _yourClassName.CountTotalPrice(employeeId, price);
   // Assert
   Assert.AreEqual(-325, result);
}

[Test]
public async Task CountTotalPrice_ReturnsSumOfSalesAndPrice_ZeroPrices()
{
   // Arrange
   var employeeId = Guid.NewGuid();
   var price = 100;
   var salesList = new List<Sale>
   {
      new Sale { EmployeeId = employeeId, Price = 0 },
      new Sale { EmployeeId = employeeId, Price = 0 },
      new Sale { EmployeeId = employeeId, Price = 0 }
   };
   _dbContextMock.Setup(mock => mock.Sales)
```

```
        .Returns(CreateDbSetMock(salesList));
    // Act
    var result = await _yourClassName.CountTotalPrice(employeeId, price);
    // Assert
    Assert.AreEqual(100, result);
}
```

➢ ქულის დასათვლელი მოდულური ტესტებისთვის დაწერილი მუტანტი ტესტები:

• ცვლის მოსალოდნელი ქულას სხვა მნიშვნელობით.

```
[Test]
public async Task CalculateScore_ReturnsCorrectScore()
{
    // Arrange
    int age = 25;
    // Act
    int score = await _personService.CalculateScore(age);
    // Assert
    Assert.AreEqual(50, score); // მუტაცია: შეცვლა მოსალოდნელი ქულის 50_ით;
}
```

• ცვლის ასაკს სხვა მნიშვნელობით.

```
[Test]
public async Task CalculateScore_ReturnsCorrectScore()
{
    // Arrange
    int age = 30; // მუტაცია: ასაკის შეცვლა 30;
    // Act
    int score = await _personService.CalculateScore(age);
    // Assert
    Assert.AreEqual(100, score);
}
```

მოსალოდნელი შედეგი: ტესტმა უნდა გაიაროს, რადგან ასაკი მოსალოდნელ დიაპაზონშია (18-დან 35 წლამდე) და მოსალოდნელი ქულა ემთხვევა _personService.CalculateScore მეთოდით დაბრუნებულ რეალურ ქულას.

CalculateYearlyIssuance_MonthIs12_CreatesCorrectIssuances()

- შეიცვალა ქულა სხვა მნიშვნელობით.

```
[Test]
public async Task CalculateYearlyIssuance_MonthIs12_CreatesCorrectIssuances()
{
 // Arrange
 Guid beneficiaryId = Guid.NewGuid();
  _unitOfWorkMock.Setup(uow => uow.PersonRepository.ReadAsync(beneficiaryId))
      .ReturnsAsync(new Beneficiary { Score = 50 });
                          // Mutated: Change the beneficiary's score to 50
   _unitOfWorkMock.Setup(uow =>
                        uow.IssuanceRepository.CreateAsync(It.IsAny<Issuance>()))
                               .Returns(Task.CompletedTask);
   // Act
   await _personService.CalculateYearlyIssuance(beneficiaryId);
   // Assert
   _unitOfWorkMock.Verify(uow =>
            uow.IssuanceRepository.CreateAsync(It.IsAny<Issuance>()), Times.Exactly(11));
   // Add additional assertions to check the properties of the created issuances if desired
}
```

- შეიცვალა მოსალოდნელი ხელფასის რაოდენობა სხვა მნიშვნელობით.

```
[Test]
public async Task CalculateYearlyIssuance_MonthIs12_CreatesCorrectIssuances()
{
   // Arrange
   Guid beneficiaryId = Guid.NewGuid();
   _unitOfWorkMock.Setup(uow => uow.PersonRepository.ReadAsync(beneficiaryId))
       .ReturnsAsync(new Beneficiary { Score = 100 });
   _unitOfWorkMock.Setup(uow                                                    =>
uow.IssuanceRepository.CreateAsync(It.IsAny<Issuance>()))
       .Returns(Task.CompletedTask);
   // Act
   await _personService.CalculateYearlyIssuance(beneficiaryId);
   // Assert
```

```
        _unitOfWorkMock.Verify(uow =>
uow.IssuanceRepository.CreateAsync(It.IsAny<Issuance>()), Times.Exactly(10)); // Mutated:
Change the number of expected issuances to 10
        // Add additional assertions to check the properties of the created issuances if desired
    }
```

ტესტი უნდა ჩავარდეს, რადგან მოსალოდნელი ხელფასების რაოდენობა (10) განსხვავდება შეცვლილი რაოდენობის გამო შექმნილი ემისიების რეალური რაოდენობისგან.

CalculateYearlyIssuance_MonthIsNot12_CreatesCorrectIssuances() მოდულური ტეს-ტის გასატესად დაიწერა რამდენიმე ვარიანტი მათ შორის:

- შეიცვალა მომხმარებლის ქულა სხვა მნიშვნელობით.

```
    [Test]
    public async Task CalculateYearlyIssuance_MonthIsNot12_CreatesCorrectIssuances()
    {
     // Arrange
     Guid personId = Guid.NewGuid();
     _unitOfWorkMock.Setup(uow => uow.PersonRepository.ReadAsync(personId))
         .ReturnsAsync(new Beneficiary { Score = 150 });
                              // Mutated: Change the beneficiary's score to 150
        _unitOfWorkMock.Setup(uow =>
                               uow.IssuanceRepository.CreateAsync(It.IsAny<Issuance>()))
                                      .Returns(Task.CompletedTask);
        // Act
        await _personService.CalculateYearlyIssuance(personId);
        // Assert
        _unitOfWorkMock.Verify(uow =>
              uow.IssuanceRepository.CreateAsync(It.IsAny<Issuance>()), Times.Exactly(12));
         }
```

მოსალოდნელი შედეგი: ტესტი უნდა ჩავარდეს, რადგან მოსალოდნელი რაოდენობა (12) განსხვავდება მომხმარებლის შეცვლილი ქულისგან.

- შეიცვალა მოსალოდნელი რაოდენობა სხვა მნიშვნელობით.

```
      [Test]
      public async Task CalculateYearlyIssuance_MonthIsNot12_CreatesCorrectIssuances()
      {
         // Arrange
```

```
    Guid personId = Guid.NewGuid();
    _unitOfWorkMock.Setup(uow => uow.PersonRepository.ReadAsync(personId))
        .ReturnsAsync(new Beneficiary { Score = 200 });
    _unitOfWorkMock.Setup(uow =>
                    uow.IssuanceRepository.CreateAsync(It.IsAny<Issuance>()))
                        .Returns(Task.CompletedTask);
    // Act
    await _personService.CalculateYearlyIssuance(personId);
    // Assert
    _unitOfWorkMock.Verify(uow =>
         uow.IssuanceRepository.CreateAsync(It.IsAny<Issuance>()), Times.Exactly(10));
              // Mutated: Change the number of expected issuances to 10
}
```

მოსალოდნელი შედეგი: ტესტი უნდა ჩავარდეს, რადგან მოსალოდნელი რაოდენობა (10) განსხვავდება შეცვლილი რაოდენობისგან.

9.5. მოდულური და მუტაციის შედეგების შედარება და რეკომენდაციები

მუტანტმა ტესტებმა აღმოაჩინეს ისეთი ვარიანტები, რომელიც არ ჰქონდა მოდულურ ტესტებს გათვალისწინებული. მაგალითად, შემდეგ ვარიანტში წარმოდგენილია მოდულური ტესტი *არასრული შემოწმების* პირობით:

CalculateSalary_MonthIs12_ReturnsCorrectSalary_Wrong() – არის არასწორი მოდულური ტესტი. იგი არასრულად ამოწმებდა CalculateSalary სერვისის ფუნქციონალს.

```
[Test]
public async Task CalculateSalary_MonthIs12_ReturnsCorrectSalary_Wrong()
{
    // Arrange
    int score = 100;
    int month = 12;
    // Act
    int salary = await _personService.CalculateSalary(score, month);
    // Assert
    Assert.AreEqual(120, salary); // Incorrect assertion, expecting an incorrect salary value
```

```
}
```

აქ კი წარმოდგენილია მუტანტი ტესტი რომელმაც ამოიცნო არასრული ფუნქციონალი:

```
 [Test]
public async Task CalculateSalary_MonthIs12_ReturnsCorrectSalary_Mutant()
{
   // Arrange
   int score = 100;
   int month = 12;
   // Act
   int salary = await _personService.CalculateSalary(score, month);
   // Assert
   Assert.AreEqual(150, salary); // განსხვავებული არასწორი ხელფასის მოლოდინი
}
```

- CountTotalPrice - ჯამური ფასის დათვლის ფუნქციაშიც აღმოჩნდა არასწორი ტესტები, წარმოგიდგენთ მუტანტ ტესტს რომელმაც აღმოაჩინა მოდულური ტესტების არასრულფასოვნება:

```
[Test]
public async Task CountTotalPrice_ReturnsSumOfSalesAndPrice_Wrong()
{
   // Arrange
   var employeeId = Guid.NewGuid();
   var price = 100;
   var salesList = new List<Sale>
   {
      new Sale { EmployeeId = employeeId, Price = 50 },
      new Sale { EmployeeId = employeeId, Price = 75 },
      new Sale { EmployeeId = employeeId, Price = 100 }
   };
   _dbContextMock.Setup(mock => mock.Sales).Returns(CreateDbSetMock(salesList));
   // Act
   var result = await _yourClassName.CountTotalPrice(employeeId, price);
   // Assert (Wrong assertion)
   Assert.AreEqual(300, result);
    //არასწორი მტკიცება, არასწორი ჯამური ფასის მოლოდინი
   // მუტანტის ტესტი არასწორი პარამეტრების მიმართვით
```

```
            _dbContextMock.Setup(mock => mock.Sales)
                      .Returns(CreateDbSetMock(new List<Sale>()));
                         // არასწორი დაყენება გაყიდვების ცარიელი სიით
             // არასწორი პარამეტრების მიმართვა გაყიდვების ცარიელი სიის
            // Act
            result = await _yourClassName.CountTotalPrice(employeeId, price);
            // Assert (Wrong assertion)
            Assert.AreEqual(100, result); // Incorrect assertion, expecting a different result
        }
    }
```

მესამე ნაწილის დასკვნა

– წარმოდგენილი ნაშრომის ფარგლებში განხორციელდა გამოყენებითი პროგრამული სისტემების Agile-დეველოპმენტის და Agile-ტესტირების პროცესების დეტალური ანალიზი და კვლევა. შეიქმნა გამოვლენილი ხარვეზების აღმოფხვრაზე ორიენტირებული ახალი ჰიბრიდული მიდგომა, რომელიც დაფუძნდა როგორც მოქნილი მეთოდოლოგიის სპრინტ-პროცესების გუნდის ეფექტიანი მუშაობის ორაგნიზაციული ფორმა. ამასთანავე, სისტემის ფუნქციონალის მაღალი ბიზნეს-მოთხოვნილებებიდან გამომდინარე, მეთოდოლოგიის ფარგლებში წინა პლანზე წამოიწია პროგრამული უზრუნველყოფის ხარისხის სრულყოფის საკითხებმა. ახალი ჰიბრიდული მეთოდოლოგია მოდულური დასატესტი პროგრამების დამუშავებას (გატესტვას) შესაბამისი ფუნქციური ტესტ-პრგრამებისა და სპეციალური მუტაციური ტესტების დახმარებით ახდენს;

– Agile გუნდის მიერ (გუნდი შედგებოდა 6 დეველოპერის, 4 ტესტერის, პროდუქტის მფლობელის და Scrum ოსტატისგან) პროგრამული პროექტი იწერებოდა „სუფთა არქიტექტურის" დიზაინის ნიმუშით, CQRS, mediatr და repository pattern გამოყენებით, რაც კომპონენტებს შორის მაქსიმალურად აბსტრაქტულ დამოკიდებულებას გულისხმობს და მნიშვნელოვანი როლი აკისრია დროის რესურსის დაზოგვაში. 44 სპრინტ-პროცესიდან წარმატებით დასრულდა 40, საშუალოდ, 20 მოდულის ტესტს დაჭირდა 80 მუტაციური ტესტი და გამოვლინდა 5 არასრულად დაწერილი „ხარვეზიანი" მოდული (ერთეულის ტესტი);

– ტესტირების პროცესის ახალმა ჰიბრიდულმა მეთოდოლოგიამ პროექტის

დასაწყისშივე აღმოფხვრა გამოვლენილი ხარვეზები. ამასთან გაზარდა მოქნილობა, გააუმჯობესა პროგრამული უზრუნველყოფის ხარისხი და შეცდომების გამოვლენა;

– დეველოპმენტის და ტესტირების ახალი ჰიბრიდული მეთოდოლოგიისთვის უპირატესობად უნდა ჩაითვალოს ტესტირების მაღალი ხარისხი და საკონტრაქტო ვადებში დამკვეთისთვის მუშა პროდუქტის მიწოდების უზრუნველყოფა; ნაკლოვან მხარედ შეიძლება ჩაითვალოს ტესტერების და დროის რესურსების გაზრდა (მაგ., ერთი მოდულის ტესტს, საშუალოდ 2-დან 10-მდე მუტაციური ტესტი შესაძლოა დაჭირდეს). ამიტომ, სანამ გუნდი გადაწყვეტს მუტაციური ტესტების გამოყენებას, კარგად უნდა განსაზღვროს იმ ფუნქციონალის პასუხიმგებლობის ხარისხი, რომლისთვისაც უნდა დაიწეროს მუტაციური ტესტები და შემდეგ გადაწყვიტოს რამდენად უღირს შესაბამისი დროის და გუნდის წევრების რესურსის დახარჯვა ასეთი ტიპის ტესტირებაში;

– სამომავლოდ მეთოდოლოგია შესაძლოა გამოყენებული იქნას არა მარტო მართვის საინფორმაციო სისტემების ასაგებად, არამედ ნებისმიერი სახის პროგ-რამული უზრუნველყოფისათვის. ინფორმაციული ტექნოლოგიების დეპარტამენტებს ჰიბრიდული მეთოდოლოგიის სწორად გამოყენების შემთხვევაში შეეძლებათ ლოგიკურ ვადებში შეიმუშაონ ხარისხიანი პროგრამული პროდუქტები კონკრეტული პროექტებისთვის. შედეგები აისახება როგორც დამკვეთის ბიზნეს-მოთხოვნილებათა დაკმაყოფილებით, ასევე კომპანიის დროითი და ფინანსური რესურსების დაზოგვით.

გამოყენებული ლიტერატურა:

1. ჩოგოვაძე გ., ფრანგიშვილი ა., სურგულაძე გ., მართვის საინფორმაციო სისტემების დაპროგრამების ჰიბრიდული ტექნოლოგიები და მონაცემთა მენეჯმენტი. ISBN 978-9941-20-790-7. სტუ. „ტექნიკ.უნივერსიტეტი“, თბ., 2017. -1001 გვ. https://gtu.ge/book/monacemta_menejmenti.pdf
2. სურგულაძე გ. კომპიუტერული პროგრამირების მეთოდები და მეთოდოლოგიები (SP, OOP, VP, Agile, UML). ISBN 978-9941-1900-1. სტუ. „IT-კონსალტინგ ცენტრი“. თბ., 2019. -200 გვ. https://gtu.ge/book/Surg_Prog-Method_2019.pdf
3. ჩოგოვაძე გ., სურგულაძე გ., გულიტაშვილი მ., დოლიძე ს. პროგრამული აპლიკაციების ხარისხის მართვა: ტესტირება და ოპტიმიზაცია. ISBN 978-9941-20-629-2. სტუ. „IT-კონსალტინგ ცენტრი“. თბ., 2020. -363 გვ. https://gtu.ge/book /Surgu_SoftwareQuality.pdf
4. სურგულაძე გ., კაკაშვილი გ., მარტიაშვილი გ. მობილური აპლიკაციების დეველოპმენტის საფუძვლები (Java, Android). ISBN978-9941-8-2488-3. სტუ. „IT-კონსალტინგ ცენტრ“. თბ.,2020. 176 გვ. https://gtu.ge/book/SurgMobAppAndr.pdf
5. სურგულაძე გ., წაწიშვილი დ. ვირტუალური რეალობა და თანამედროვე საინფორმაციო ტექნოლოგიები. ISBN 978-9941-8-0626-1. სტუ. „IT-კონსალტინგ ცენტრი“. თბ., 2018. -112 გვ. https://gtu.ge/book/Surgul_VirtualReality.pdf
6. ჩოგოვაძე გ., სურგულაძე გ., თოფურია ნ., ხარიტონაშვილი მ. ინფორმაციული საზოგადოება და ინტერდისციპლინური სწავლება ციფრული ტექნოლოგიების ბაზაზე. ISBN 978-9941-8-3338-0. მონოგრაფია. სტუ. „IT-კონსალტინგის სამეცნიერო ცენტრი”, თბ., 2021. -360 გვ. https://gtu.ge/book /Surgu_InfoSociety-21%20new.pdf
7. სურგულაძე გ., პაპავაძე ს., მაჩალაძე ო. ინფორმაციული საზოგადოება და ინფორმატიკის დიდაქტიკა. ISBN 978-9941-8-5443-9. მონოგრაფია. სტუ. „IT-კონსალტინგ.სამეცნ. ცენტრი”, თბ., 2023. –260 გვ. https://old.gtu.ge/book/file1.pdf
8. სურგულაძე გ., გოგშელიძე დ., დალაქიშვილი გ. საინფორმაციო საზოგადოება: საგამომცემლო მარკეტინგი და მანქანური დასწავლა. ISBN 978-9941-8-5442-2. მონოგრაფია. სტუ. „IT-კონსალტინგ.სამეცნ. ცენტრი”, თბ., 2023. –216 გვ. https://old.gtu.ge/book/file1.pdf
9. სურგულაძე გ., თურქია ე. პროგრამული სისტემების მენეჯმენტის საფუძვლები. ISBN 978-9941-20-651-1. სტუ. „ტექნიკ.უნივ.“, თბ., 2016. -350 გვ. https://gtu.ge/book/gia_sueguladze/GiaSurg1_ ProgSysManag.pdf

10. Naveen Reddy K.P., Undavalli Harichandana, Alekhya T., Rajesh S.M. (2019), A Study of Robotic Process Automation Among Artificial Intelligence. International Journal of Scientific and Research Publications (IJSRP)

11. Surguladze G., Berdzenishvili I. Lean management, Devops, Agile Software-development for Robotic Process Automation. Transactions of Georgian Technical University. Automated Control Systems, No1(33), vol.1, Tbilisi, 2022. pp.51-57. DOI.org/10.36073/1512-3979

12. Lundell B., John B.J. Lings. Comments on ISO 14102: the standard for CASE-tool evaluation. University of Skövde. 2002. Computer Standards & Interfaces 24(5):381-388. DOI:10.1016/S0920-5489(02)00064-8

13. გოგიჩაიშვილი გ., სურგულაძე გ., შონია ო. დაპროგრამების მეთოდები C & C++ ენების ბაზაზე. სტუ, „ტექნიკური უნივერსიტეტი", თბ., 1997. -275 გვ.

14. Booch G., Jacobson I., rambaugh J. Unified Modeling Language for Object-Oriented Development. Rational Software Corporation, Santa Clara, 1996

15. სურგულაძე გ., კიკნაძე მ. დაპროგრამების მეთოდები ინტერნეტისთვის (Java-2 და XML ენების ბაზაზე). თბ., „ტექნიკური უნივერსიტეტი", 2006. http://www.gtu.edu.ge/ katedrebi /kat94/pdf/Java2+XML.pdf

16. გაჩეჩილაძე ლ. დაპროგრამების ენა Python. სტუ. „ტექნიკური უნივერსიტეტი", თბ., 2017. -148 გვ.

17. ბოტჰე კ., სურგულაძე გია, დოლიძე თ., შონია ო., სურგულაძე გიორგი. თანამედროვე პროგრამული პლატფორმები და ენები (WindowsNT, Unix, Linux, C++, Java, XML). სტუ, „ტექნიკური უნივერსიტეტი", თბ., -250 გვ. 2003

18. სურგულაძე გ. ობიექტ-ორიენტირებული დაპროგრამების მეთოდი. სტუ, თბ., -100 გვ. 2007

19. About the Unified Modeling Language Specification Version 2.5.1. https://www.omg.org/spec/UML/2.5.1# document-metadata (20.04.24)

20. Бек К. Шаблоны реализации корпоративных приложений. Экстремальное программирование: Пер. с англ. М.: Вильямс. 2008

21. Visual Programming Language VPL. Internet resource: https://www.techo-pedia.com/definition/22855/visual-programming-language-vpl

22. სურგულაძე გ., ქრისტესიაშვილი ხ., სურგულაძე გიორგი. საწარმოო რესურსების მენეჯმენტის ბიზნეს-პროცესების მოდელირება და კვლევა. ISBN 978-9941-20-557-6. სტუ. „ტექნიკური უნივერსიტეტი", თბ., 2015 -216 გვ.

23. გოგიჩაიშვილი გ., ბოლხი გ., სურგულაძე გ., პეტრიაშვილი ლ. მართვის ავტომატიზებული სისტემების ობიექტ-ორიენტირებული დაპროექტების და

მოდელირების ინსტრუმენტები (MsVisio, WinPepsy, PetNet, CPN). სტუ.. თბ., „ტექნიკური უნივერსიტეტი“. 2013

24. სურგულაძე გ., თურქია ე. პროგრამული ინჟინერიის საფუძვლები. ISBN 978-9941-8-3808-8. საკურსო პროექტის დამხმარე სახელმძღვანელო. სტუ-ს „IT-კონსალტინგ სამეცნიერო ცენტრი". თბ., 2022, - 200 გვ.

25. ვედეკინდი ჰ., სურგულაძე გ., თოფურია ნ. განაწილებული ოფის-სისტემების მონაცემთა ბაზების დაპროექტება და რეალიზაცია UML-ტექნოლოგიით. ISBN 99940-57-17-0. სტუ. „ტექნიკ.უნივერს.“, თბ., 2006 -237 გვ.

26. სურგულაძე გ. თოფურია ნ., ბიტარაშვილი მ. მონაცემთა ბაზის ავტომატიზებული დაპროექტება და აგება ORM/ERM ტექნოლოგიით საგადასახადო დავების სისტემისთვის. სტუ-ს შრ.კრებ.. „მას"- N2(15), თბილისი, 2013. - გვ.40-45

27. OMG. Unified Modeling Language. Version 2.5.1 (OMG UML). 2017. -796 p. https://www.omg.org/spec/UML/2.5.1/pdf

28. About Object Management Group. OMG Unified Modeling LanguageTM (OMG UML), Superstructure. Version 2.4.1 (22.05.24)

29. ჩოგოვაძე გ., გოგიჩაიშვილი გ., სურგულაძე გ., შეროზია თ., შონია ო. მართვის ავტომატიზებული სისტემების დაპროექტება და აგება (თეორიული და პრაქტიკული ინფორმატიკა). ISBN 99928-882-7-X. სტუ, „ტექნიკური უნივერსი-ტეტი“, 2001. – 740 გვ.

30. ბოტჰე კლაუს, სურგულაძე გ., კაშიბაძე მ. მემკვიდრეობითობა მართვის ინფორმაციული სისტემების დაპროგრამებაში: მონაცემთა ბაზებიდან UML-ტექნოლოგიამდე. საერთაშ. კონფ., სტუ-ს შრ.კრებ. N4(437). თბ., 2001. გვ. 55-62

31. თურქია ე. ბიზნეს-პროექტების მართვის ტექნოლოგიური პროცესების ავტომატიზაცია. სტუ. „ტექნიკური უნივერსიტეტი". თბ., 2010

32. სურგულაძე გ., ბიტარაშვილი მ., გელაძე ბ., ნარეშელაშვილი გ., შურღაია ი. რევერსული პროგრამირების CASE მეთოდი საინფორმაციო სისტემების ოო-დაპროექტების პროცესის სრულყოფისათვის. სტუ-ს შრ.კრებ.. „მას"- N1(37), თბილისი, 2024. - გვ.115-121

33. სურგულაძე გ., პეტრიაშვილი ლ. აპლიკაციების დაპროგრამება და მონაცემთა მენეჯმენტი. ISBN 978-9941-8- 3810-1, დამხმარე სახელმძღვანელო. სტუ-ს „IT-კონსალტინგ სამეცნიერო ცენტრი". თბ., 2022, - 135 გვ.

34. Model–view–controller. Internet resource: https://en.wikipedia.org/wiki/-Model%E2%80%93view%E2%80%93controller (10.06.24)

35. სურგულაძე გ., თოფურია ნ., ბერულავა ა. პროგრამული პროდუქტების დეველოპმენტი. ISBN 978-9941-8- 3810-1, დამხმარე სახელმძღვანელო. სტუ-ს „IT-კონსალტინგ სამეცნიერო ცენტრი". თბ., 2022, - 250 გვ.

36. Halpin T. ORM-2 Graphical Notation. Neumont University, 2005. http://www.orm.net/ pdf/ORM2_TechReport1.pdf

37. Model Driven UML Tool. Internet resource: https://www.sparxsystems.eu/ (12.06.24)

38. What is Agile Software Development? http://www.inflectra.com/Methodologies/AgileDevelopment .aspx#Scrum

39. Борута Я. Методология Agile. Матерь драконов или всех гибких методологий. w-Blog. 2018. https://worksection. com/blog/agile.html

40. code-and-fix model: in Development Life Cycle Models. Internet resource: http://zone.ni.com/reference/en-XX/help/371¬361R-01/lvdevconcepts/-lifecycle_models/

41. Principles behind the Agile Manifesto. Internet resource: http://agilemanifes¬to.org/¬princip-les.html.

42. Beck K. et al. Manifesto for Agile Software Development. 2001. Internet resource: https://agilemanifesto.org/

43. Ambler S.W. The Object Primer: Agile Model-Driven Development With Uml 2.0. Cambridge University Press 2004-05-28. 2001. 572 p. https://www.researchgate.net/ publica¬tion/235616285_The_object_primer_agile_modeling-driven_deve¬lopment_ with_UML_20

44. Амблер С. Гибкие технологии: экстремальное программи¬ро¬вание и унифицированный процесс разработки. Библиотека программиста. Спб.: Питер.

45. Rumpe B. Agile Modellierung mit UML. Berlin, „Springer“. 2-te Auflage. 2012

46. Resources Scrum. Internet resource: 2019. https://www. scrum.org/resources

47. Lean Software Development - in What is Agile Kanban Meth¬o¬do¬logy ? https://www.inflectra.com/methodologies/kan¬ban.aspx

48. Six Sigma. Internet resource: https://en.wikipedia.org/ wiki /Six_Sigma

49. Berchez J.P., Kapp U. Kriterien für eine Entscheidung für Scrum oder Kanban. IBM, Heise online. 2010. https://www.heise.de/¬develo¬per/artikel/Kriterien-fuer-eine-Entsch¬ei¬dung-fuer-Scrum-oder-Kanban-1071172.html?seite=all

50. Anderson, David J. (April 2010). Kanban: Successful Evolutionary Change for Your Technology Business. Blue Hole Press. ISBN 978-0-9845214-0-1.

51. Kanban (development). Internet resource: https://de.wiki-pedia.org/wiki/Kanban_(Softwareentwicklung)

52. Deming E.W. ციკლი (PDSA Plan-Do-Study-Act) „დაგეგმე, გააკეთე, შეისწავლე, იმოქმედე". Internet resource: https://de.wikipedia.org/wiki/William_Edwards_Deming.

53. PDCA (plan–do–check–act). Internet resource: https://en.wikipedia.org/wiki/PDCA

54. Пименов М. Разбираемся в Scrum и Kanban. Блог Нетологии. 2017. Internet resource: https://netology.ru/blog/ scrum-kanban

55. Зыкова С. Agile, scrum, kanban: в чем разница и для чего использовать? Блог Нетологии. 2018. Internet resource: https://rb.ru/story/agile-scrum-kanban/

56. Schindler M. Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P. Aachener Informatik-Berichte, Germany. Software Engineering Band 11. Hrsg.: Prof. Dr. rer. nat. Bernhard Rumpe. 2012. -363 p.

57. სურგულაძე გ., პეტრიაშვილი ლ. ვიზუალური დაპროგრამება C# ენის ბაზაზე ინფორმაციულ სისტემებისათვის (Visual Studio.NET 2019 პლატფორმაზე). სტუ, „IT კონსალტინგ ცენტრი“. თბ., 200 გვ.

58. სურგულაძე გ., ურუშაძე ბ. საინფორმაციო სისტემების მენეჯმენტის საერთაშორისო გამოცდილება (BSI, ITIL, COBIT). ISBN 978-9941-20-458-6. სტუ. „ტექნიკ.უნივერს.“, თბ., 2014 -345 გვ. https://gtu.ge/book/gia_sueguladze/sainfo_sistemebi_BSI_ITIL_COBIT.pdf

59. სურგულაძე გ., გულიტაშვილი მ., კაკულია ი., ჩერქეზიშვილი გ., ჯავახიშვილი ი. პროგრამული სისტემების სასიცოცხლო ციკლის პროცესის მოდელირება უნივერსალური და ექსტრემალური პროგრამირების პრინციპების კომპრომისული გადაწყვეტით. სტუ-ს შრ.კრ. „მას", N1(8), თბ., 2010. გვ. 63-70

60. სურგულაძე გ., დოლიძე ს. მომხმარებლის ინტერფეისის დაპროგრამება (AngularJS, ReactJS). ISBN 978-9941-8-0625-4. სტუ. „IT-კონსალტინგ ცენტრი“. თბ., 2019. -106 გვ. https://gtu.ge/book/SurguDoliReact.pdf

61. სურგულაძე გ., გულიტაშვილი მ., კივილაძე ნ. Web-აპლიკაციების ტესტირება, ვალიდაცია და ვერიფიკაცია. ISBN 978-9941-0-7682-4. სტუ. „IT-კონსალტინგ ცენტრი“. თბ., 2015. -205 გვ. https://gtu.ge/book/gia_sueguladze/4 GiaSurg WebSoftwareTesting.pdf

62. Mala, D.J. Integrating the Internet of Things Into Software Engineering Practices. Advances in Systems Analysis, Software Engineering, and High Performance Computing. IGI Global. p. 16. 2019. ISBN 978-1-5225-7791-1

63. DevOps. Internet resource: https://en.wikipedia.org/wiki /DevOps

64. Source Code Control System. Internet resource: https://de. wikipedia.org/wiki/Source_Code_Control_System

65. Fowler M. Continuous Integration. Internet resource: 2006. https://martinfowler.com/articles/continuousIntegration.html

66. The Relationship between Risk and Continuous Testing. 2014. https://www.stickyminds.com/interview/relationship-between-risk-and-continuous-testing-interview-wayne-ariola

67. Rahm E. Data Warehouses. Einführung. S.2, 2015. Vorle¬sungsskript, Universität Leipzig, Germany

68. Application Release Automation. Internet resource: https://en.wikipedia.org/ wiki/Application-release_automation

69. Infrastructure as code. Internet resource: https://en.wikipedia.org/wiki/ Infrastructure_as_code

70. Kendall's notation. Internet resource: http://en.wikipedia.org/wiki/Kendall

71. Bolch G., Greiner S., De Meer H., Trivedi K. Queueing Networks and Markov Chains, Modeling and Performance Evaluation With Computer Science Application. John Wiley & Sons, 1998

72. ბოლხი გ., სურგულაძე გ., პეტრიაშვილი ლ., ჩიხრაძე ზ. მულტი-პროცესორული სისტემების რესურსების მართვის პროგრამული უზრუნველყოფის დამუშვება Borland_C++Bulider ინსტრუმენტით. სტუ-ს შრ., 4(437), თბილისი, 2001

73. სურგულაძე გ., ბულია ი., ურუშაძე ბ. საინფორმაციო სისტემების ინტეგრაციის პროცესების მენეჯმენტი სერვის-ორიენტირებული არქიტექტუ-რით და UML/AGILE ბაზირებული მეთოდებით. სტუ შრ.კრ.: „მას“-N2(13). თბილისი, 2012, გვ. 7–22.

74. სურგულაძე გ., ბულია ი. კორპორაციულ Web-აპლიკაციათა ინტეგ-რაცია და დაპროექტება. ISBN 978-9941-20-165-3. სტუ. „ტექნიკ.უნივერს.“, თბ., 2012 -324 გვ.

75. საქართველოს საგადასახადო კოდექსი. თბ., 17 სექტ., 2010

76. ბიტარაშვილი მ., სურგულაძე გ. საგადასახადო სამართალდარღვევის საქმის წარმოების სისტემის ბიზნეს-პროცესების მოდელირება UML2 ტექნო-

ლოგიით. სტუ-ს შრ.კრ. „მართვის ავტომატიზებული სისტემები“. N2(13), თბილისი, 2012, გვ. 217–222.

77. სურგულაძე გ., ბიტარაშვილი მ. ბიზნეს-პროცესების UML-მოდელირება და პროგრამული რეალიზაცია Workflow Foundation ტექნოლოგიით საგადასახდო დავების სისტემის მაგალითზე. სტუ-ს შრ.კრ. „მართვის ავტომატიზებული სისტემები“. N1(14), თბილისი, 2013, გვ. 229–233

78. https://curiousdesire.com. Internet resource: https://curiousdesire.com: https://curiousdesire.com/reasons-why-information-system-is-important/ (3.01.23)

79. https://www.wharftt.com. Internet resource: https://www.wharftt .com¬/¬the-importance-of-information-systems-in-organizations/ (3.01.23)

80. Henry C. Lucas, Jr., E. Burton Swanson, Robert W. Zmud. Implementation, Innovation, and Related Themes Over The Years In Information Systems. Journal of Associaton for INformation Systems. 2007, Vol. 8, Issue 4, pp. 206-210

81. Torgeir Dingsøyr et , a. An Empirical Investigation of the Scrum Method: A Controlled Case Study. IEEE Transactions on Software Engineering. 2012.

82. Torgeir Dingsøyr et, a. A Comparative Study of Agile Methods: Assessing the Applicability of Scrum and XP. Information and Software Technology. 2012.

83. Pressman, R. Software Engineering: A Practitioner's Approach. 2014.

84. Sutherland, J. Scrum: The Art of Doing Twice the Work in Half the Time. 2014

85. Chris Sims, Hillary Louise Johnson. Scrum: A Breathtakingly Brief and Agile Introduction. 2012

86. https://learn.microsoft.com. Internet resource: https://learn.micro¬soft.com/ en-us/azure/architecture/patterns/cqrs (3.0 1,23).

87. https://asana.com. Internet resource: https://asana.com/reso¬urces/extreme-programming-xp (3.01.23)

88. https://asana.com. Internet resource: https://asana.com/resources/agile-methodology (3.01.23)

89. https://www.atlassian.com. Internet resource: https://www.atlassian.com/ agile/scrum/sprint-planning (3.01.23)

90. https://www.atlassian.com. Internet resource: https://www.atlassian.com /software/jira?&aceid=&adposition=&adgroup=143540977612&campaign=17623645605 &creative=607582093117&device=c&keyword=agile%20scrum%20software%20develo pment&matchtype=e&network=g&placement=&ds_kids=p71908590908&ds_e=GOOGL E&ds_ei (3.01.23)

91. https://www.nimblework.com. Internet resource: https://www.nimblework.com/agile/agile-methodology/ (15.01.23)

92. Päivärinta T., Lassenius C., Bold C. Testing Challenges in Agile Software Development. IEEE Software. 2010.

93. Gregory J., Crispin L. More Agile Testing: Learning Journeys for the Whole Team. 2014

94. Bluemke I., Kamsties E., Heinz-Dietrich W. Agile Testing: Challenges and Strategies. Information and Software Technology. 2013

95. Marick B. The Agile Testing Quadrants. Internet resource: https://agile-testing.blogspot.com/ (10.12.23)

96. Hannah Son. Agile Testing Methodology: Life Cycle, Techniques, & Strategy. 2023. Internet resource: https://www.testrail.com/blog/agile-testing-methodology/

97. Hannah Son. Agile QA Process: Principles, Steps, and Best Practices. 2024. https://www.testrail.com/blog/agile-qa-best-practices/

98. Hamilton T. What is Agile Testing? Process & Life Cycle. 2024. Internet resource: https://www.guru99.com/agile-testing-a-beginner-s-guide.html

99. Hamilton T. What is Unit Testing? 2024. Internet resource: https://www.guru99.com/unit-testing-guide.html

100. Ultimate guide Agile testing modern software teams. 2024. Internet res.: https://sjinnovation.com/ultimate-guide-agile-testing-modern-software-teams-part-1

101. Martin R. Clean Architecture: A Craftsman's Guide to Software Structure and Design. 2017

102. Pavlutin, D. Clean Architecture and Design: Understanding The Basics. 2020

103. The Clean Code Blog. Internet resource: https://blog.cleancoder.com, 2023

104. Jia Hui Liang et,a. The Art of Mutation Testing. IEEE Transactions on Software Engineering. 2019

105. Lwin Khin Shar et,a. Improving Test Suite Effectiveness through Integration of Mutation and Property-Based Testing. IEEE Transact. on Software Engineering. 2020

106. Papadakis M., Kintis M., Zhang J., Jia Y., Traon Y., Harman M. Chapter Six - Mutation Testing Advances: An Analysis and Survey. Advances in Computers Vol.112, 2019, pp. 275-378

107. Felipe Padilha et,a. A Systematic Review of Mutation Testing: 35 Years Later. Information and Software Technology. 2019

108. Gordon F., Andrea A. Advances in Computers, Volume 116: Mutation Testing for the New Century. 2020

109. Gordon F., Wotawa F. Mutation Testing for the New Century. 2018

110. ხატიაშვილი ხ. Agile-ტესტირება მუტაციური ტესტებით მენეჯმენტის საინფორმაციო სისტემის აპლიკაციაში. მართვის ავტომატიზებული სისტემები, 2022, გვ. 63-69. N 2(34), . DOI.org/10.36073/1512-3979

111. ხატიაშვილი ხ. ორგანიზაციული მართვის საინფორმაციო სისტემების Agile ტესტირება. საქართველოს ტექნიკური უნივერსიტეტის 100 და იმს ფაკულტეტის 65 წლისთ. საერთაშ. სამეც.-პრაქტ.კონფ. „ინოვაციები და თანამედროვე გამოწვევები 2022", გვ. 298-302. თბ., სტუ.

112. ხატიაშვილი, ხ. პროგრამული აპლიკაციების Agile დეველოპმენტი და ტესტირება. სტუ-ის შრ.კრ., მართვის ავტომატიზებული სისტემები, N1(33), 2022. გვ.144-147, DOI.org/10.36073/1512-3979

113. ხატიაშვილი, ხ., სურგულაძე, გ. სუფთა არქიტექტურაზე აგებული ორგანიზაციული მართვის საინფორმაციო სისტემის Agile ტესტირება. მართვის ავტომატიზებული სისტემები, N2(34), 2022. გვ. 55-63, DOI.org/10.36073/1512-3979

Gia Surguladze, Marine Bitarashvili, Khatia Khatiashvili

Software Development and Testing Technologies
(CASE, UML, Agile)

ISBN 978-9941-8-6334-9

"IT Consulting Center" of GTU
Tbilisi, Georgia - 2019

გადაეცა წარმოებას 25..07.2024 წ. ოფსეტური ქაღალდის
ზომა 60X84 1/17. პირობითი ნაბეჭდი თაბახი 8. ტირაჟი 50 ეგზ.

სტუ-ს „IT კონსალტინგის სამეცნიერო ცენტრი"
(თბილისი, მ.კოსტავას 77)

ISBN 978-9941-8-6334-9