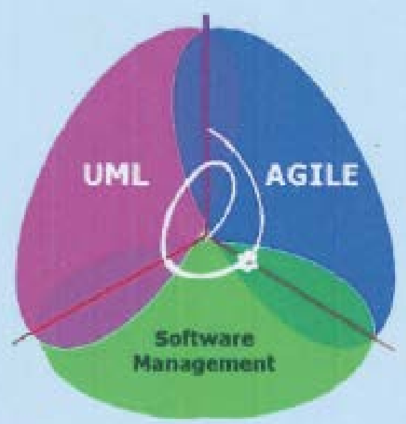
გია სურგულაძე, ეკატერინე თურქია

# პროგრამული სისტემების მენეჯმენტის საფუძვლები




„ტექნიკური უნივერსიტეტი"

საქართველოს ტექნიკური უნივერსიტეტი

გია სურგულაძე, ეკატერინე თურქია

# პროგრამული სისტემების მენეჯმენტის საფუძვლები

დამტკიცებულია სახელმძღვანელოდ
საქართველოს ტექნიკური უნივერსიტეტის
სარედაქციო-საგამომცემლო საბჭოს მიერ.
30.03.2016, ოქმი N1

თბილისი
2016

უაკ 004.5

განხილულია მართვის საინფორმაციო სისტემების პროგრამული უზრუნველყოფის ობიექტორიენტირებული ანალიზის, დაპროექტების, დეველოპინგის, ტესტირების, დანერგვისა და რეინჟინერინგის საკითხები. განსაკუთრებით გამახვილებულია ყურადღება გამოყენებითი სისტემების პროგრამული უზრუნველყოფის შექმნის უნიფიცირებულ (UML) და მოქნილ (Agile) მეთოდოლოგიებზე, სასიცოცხლო ციკლის გუნდური მენეჯმენტის ამოცანებზე.

სახელმძღანელოში შემოთავაზებულია პროგრამული სისტემების მენეჯმენტის როგორც თეორიული საფუძვლები, ისე ლაბორატორიული პრაქტიკუმის ამოცანები. გამოყენებულია MsVisual Studio.NET Framework 4.5 ინტეგრირებული სამუშაო გარემოში.

წიგნი განკუთვნილია პროგრამული ინჟინერიისა და მართვის საინფორმაციო სისტემების სპეციალობის ბაკალავრების, მაგისტრანტ-დოქტორანტებისა და პროგრამული უზრუნველყოფის მენეჯმენტის საკითხებით დაინტერესებული მკითხველებისთვის.

რეცენზენტები:
- პროფ, გიორგი გოგიჩაიშვილი (საქ. მეცნიერებათა ეროვნული აკადემიის წევრ-კორესპოდენტი)
- პროფ. რომან სამხარაძე

© საგამომცემლო სახლი „ტექნიკური უნივერსიტეტი", 2016

ISBN 978-9941-20-651-1

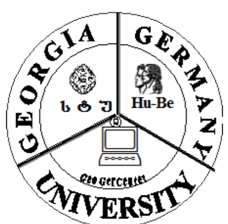
ყველა უფლება დაცულია, ამ წიგნის არც ერთი ნაწილი (იქნება ეს ტექსტი, ფოტო, ილუსტრაცია თუ სხვა) გამოყენება არანაირი ფორმითა და საშუალებით (იქნება ეს ელექტრონული თუ მექანიკური), არ შეიძლება გამომცემლის წერილობითი ნებართვის გარეშე. საავტორო უფლებების დარღვევა ისჯება კანონით.

Georgian Technical University

**Gia Surguladze, Ekaterine Turkia**

# SOFTWARE MANAGEMENT BASICS

**Supported by DAAD (Germany)**




The present book discusses Software design, modeling, development, testing and implementation problems of Management Information Systems (MIS) based on object-oriented methodology. Modern methods and tools of applications construction are presented. Results of practical experiments are realized in the environment of Ms Visual Studio.NET Framework 4.5.

© Publication House "Technical University", Tbilisi, 2016

ISBN 978-9941-20-651-1

**ნაშრომი ეძღვნება**

# საქართველოს ტექნიკური უნივერსიტეტის

# *„მართვის ავტომატიზებული სისტემების“*

# კათედრის დაარსების 45 წლის იუბილეს

# (20 მაისი: 1971-2016 წწ.)

**გია სურგულაძე** - სტუ-ს პროფესორი, ტექნიკის მეცნიერებათა დოქტორი, გაეროსთან არსებული „ინფორმატიზაციის საერთაშორისო აკადემიის (IIA)” ნამდვილი წევრი, სტუ-ს „IT-კონსალტინგის სამეცნიერო ცენტრის“ ხელმძღვანელი, გერმანიის DAAD-ის გრანტის მრავალგზის მფლობელი, ბერლინის ჰუმბოლდტის, ნიურნბერგ-ერლანგენის და სხვა უნივერსიტეტებში მიწვეული პროფესორი 1991-2014 წწ. 300-ზე მეტი სამეცნიერო ნაშრომის, მათ შორის მართვის საინფორმაციო სისტემების ინჟინერინგის სფეროში 65 წიგნის და 50 ელ-სახელმძღვანელოს ავტორი.

**ეკატერინე თურქია** - სტუ-ს პროფესორი, ტექნიკის მეცნიერებათა კანდიდატი, გერმანიის DAAD-ის გრანტის მრავალგზის მფლობელი, ნიურნბერგ-ერლანგენის, ბაიროითის, ზაარბრუკენის და სხვა უნივერ-სიტეტების მიწვეული პროფესორი 2002-2012 წწ. 50-ზე მეტი წიგნისა და სამეცნიერო ნაშრომის ავტორი. საქართველოს პრეზიდენტის გრანტის მფლობელი ახალგაზრდა მეცნიერთათვის, რუსთაველის სამეცნიერო ფონდის კონკურსის გამარჯვებული საზღვარგარეთ სამეცნიერო კვლევების შესასრულებლად. მონაწილეობდა საქართველოს სასამართლო სისტემის რეფორმის მსოფლიო ბანკის პროექტში პროგრამისტად. ფლობს „მაიკროსოფთის“ ვინდოუს- და ვებსისტემების დაპროგრამების უახლოეს ტექნოლოგიებს.

## სარჩევი

- შესავალი: ........................................................................................ 8
**1 თავი. თეორიული ნაწილი: პროგრამული სისტემების მენეჯმენტის ძირითადი მოდელები და მეთოდები** ............. 9
1.1. პროგრამული სისტემების მენეჯმენტის არსი, მიზნები და ამოცანები, განვითარების პერსპექტივები ............................... 9
1.2. პროგრამული ინჟინერიის ტექნოლოგიების რეტროსპექტული ანალიზი ....................................................... 16
1.3. პროგრამული სისტემების დამუშავების ბიზნესპროცესის გუნდი: როლები და ფუნქციები ............................................... 24
1.4. პროგრამული პროექტის სისტემა და მისი ელემენტები......... 30
1.5. პროექტების მენეჯმენტი ............................................................ 33
1.6. პროგრამული უზრუნველყოფის სასიცოცხლო ციკლის მოდელები ..................................................................... 37
1.6.1. პროგრამული უზრუნველყოფის სასიცოცხლო ციკლის საბაზო მოდელი ........................................................... 37
1.6.2. კლასიკური იტერაციული მოდელი ...................................... 42
1.6.3. კასკადური მოდელი ................................................................. 45
1.6.4. განტერის მოდელი: ფაზები და ფუნქციები ....................... 49
1.6.5. სპირალური მოდელი ............................................................... 53
1.6.6. MSF მოდელი .................................................................................. 54
1.7. ობიექტორიენტირებული დაპროგრამების პრინციპები......... 55
1.8. პროგრამული სისტემების დაპროექტების UML ტექნოლოგია და მისი ინსტრუმენტები .................................... 70
1.9. UML დიაგრამები ........................................................................... 75
1.10. პროგრამული სისტემების აგება მოქნილი (Agile) ტექნოლოგიებით ........................................................................ 81
1.10.1. ექსტრემალური დაპროგრამების პრინციპები.................... 81

1.10.2. მოქნილი (სწრაფი) მოდელირება (Agile Modeling) ………. 85
1.10.3. მოქნილი მოდელირების ფასეულობანი ............................ 85
1.10.4. Scrum - მოქნილი მეთოდის მაგალითი .............................. 92
1.11. ITIL მეთოდოლოგია ................................................................ 97
1.11.1. ITILv3-ის ლექსიკონის განსაზღვრებანი ............................. 100
1.11.2. IT-სერვისის სასიცოცხლო ციკლი ....................................... 106
1.11.3. სერვისების ცოდნის ბაზის მართვის სისტემა .................... 112
1.11.4. სტრატეგიის აგება - სერვისების სასიცოცხლო ციკლის ეტაპი ......................................................................... 114
1.12. COBIT საერთაშორისო სტანდარტული მეთოდოლოგია...... 116
1.13. ADO.NET ტექნოლოგია: მოქნილი კავშირი მომხმარებლის ინტერფეისსა და მონაცემთა ბაზებს შორის ............................ 124
1.13.1. გამოყოფილ მონაცემებთან მიმართვა.......................... 126
1.13.2. ADO.NET-ის მონაცემთა არქიტექტურა ............................. 127
1.13.3. მონაცემთა ბაზასთან მიერთება ............................................ 132
1.14. IT-მენეჯმენტის მხარდამჭერი და ტრანზაქციის დამუშავების სისტემები .............................................................. 134
1.15. საინფორმაციო ტექნოლოგიების სერვისების იმპლემენტაცია და ხარისხის მართვა ................................ 139
**2 თავი. ლაბორატორიული პრაქტიკუმის ნაწილი:** ........................ 148
2.1. აპლიკაციის აგება დაპროგრამების რამდენიმე ენის საფუძველზე .dll ფაილების შექმნით (ლაბ.N1) ........................ 148
2.2. გამონაკლის შემთხვევათა აღმოჩენისა და გამორიცხვის ვიზუალური საშუალებები (ლაბ.N2 )......................................... 165
2.3. კლასების და ობიექტების დაპროგრამება მართვის საინფორმაციო სისტემების ამოცანებში (ლაბ.N3) .................... 190
2.4. ცხრილების წარმოდგენის მართვის ელემენტი DataGridWiew (ლაბ.N4)................................................................ 205

2.5. DataGridViewComboBoxColumn და DataGridViewTextbox
Column კლასების გამოყენება ცხრილებთან სამუშაოდ (ლაბ.N5).. 215
2.6. C# აპლიკაციის მუშაობა Ms Access ბაზასთან (ლაბ.N6)…..… 220
2.7. C# აპლიკაციის მუშაობა MySQL ბაზასთან (ლაბ.N7) .........…. 237
2.8. C# აპლიკაციის მუშაობა Ms SQL Server ბაზასთან .................. 251
2.8.1. მონაცემთა განახლების Input, Update, Delete მეთოდების
დაპროგრამება ListView კლასის გამოყენებით (ლაბ.N8)........ 251
2.8.2. SQL Server ბაზის ცხრილების პროგრამული განახლების
რეალიზაცია ListView კლასის საფუძველზე (ლაბ.N9)........... 258
2.8.3. SQL Server ბაზის განახლება ADO.NET დრაივერისა და
DataGridView კლასის გამოყენებით (ლაბ.N10) ...................... 262
2.9. რეფაქტორინგი: კოდის დამუშავება და მისი
რეორგანიზაცია (ლაბ.N11) ……..……….........................……. 277
2.10. Visual Studio.NET -ის რევერსიული ინჟინერიის ინსტრუ-
მენტული საშუალებები: „Model-Code-Model“ (ლაბ.N12) ...... 283
2.11. პროგრამული აპლიკაციების ტესტირება (ლაბ.N13) ……... 294
2.12. პროგრამული კოდების ხარისხის შეფასება (ლაბ.N14)......... 304
2.13. პროექტის ბიუჯეტის შედგენისთვის სამომხმარებლო
აპლიკაციის მომზადება (ლაბ.N15) ............................................ 309
2.14. პროგრამული სისტემების შექმნის დავალებათა
კალენდარული გეგმის ფორმირება (ლაბ.N16).......................... 325
2.15. პროგრამული აპლიკაციის დისტრიბუციული ფაილის
(საინსტალაციო პაკეტის) შექმნა (ლაბ.17) ............................... 339
- ლიტერატურა ......................................................................... 346

# შესავალი

მართვის საინფორმაციო სისტემების პროგრამული უზრუნველყოფის ობიექტორიენტირებული ანალიზის, დაპროექ-ტების, დეველოპინგის, ტესტირების, დანერგვისა და რეინჟინერინ-გის საკითხების ინტენსიური კვლევა და სწავლება განსაკუთრებით მნიშვნელოვანი და აქტუალურია თანამედროვე კომპიუტერული ინდუსტრიისა და ინფორმაციული ტექნოლოგიების უახლესი მიღწევების ფონზე. სახელმძღვანელოში შემოთავაზებულია გამოყენებითი პროგრამული სისტემების დაპროექტების, რეალიზაციის, ტესტირებისა და რეინჟინერინგის საკითხები.

პირველ თავში გამახვილებულია ყურადღება გამოყენებითი სისტემების პროგრამული უზრუნველყოფის შექმნის უნიფიცირე-ბულ (UML) და მოქნილ (Agile) მეთოდოლოგიებზე, სასიცოცხლო ციკლის გუნდური მენეჯმენტის ამოცანებზე. მეორე თავში გადმოცემულია „პროგრამული სისტემების მენეჯმენტის საფუძვლების" საგნის 17 ლაბორატორიული ამოცანა, მათ შორის ორიგინალური საკითხებიცაა, კერძოდ, .NET პლატფორმაზე მრავალენიანი პროგრამული პროექტის აგება dll-ფაილებით, მომხმარებელთა ინტერფეისებისა და ბაზების გამოყენება, პროგრამების ტესტირება, მათი ხარისხის შეფასება, დისტრიბუ-ციული (საინსტალიაციო) პაკეტების მომზადება და ა.შ. გამოყენებულია MsVisual Studio.NET Framework 4.5 ინტეგრირე-ბული პაკეტი.

სახელმძღვანელო ორიენტირებულია პროგრამული ინჟინერიისა და მართვის საინფორმაციო სისტემების სპეციალობათა ბაკალავრებზე, მაგისტრანტ-დოქტორანტებსა და პროგრამული უზრუნველყოფის მენეჯმენტის საკითხებით დაინტერესებულ მკითხველზე.

# 1 თავი. თეორიული ნაწილი:

# პროგრამული სისტემების მენეჯმენტის ძირითადი მოდელები და მეთოდები

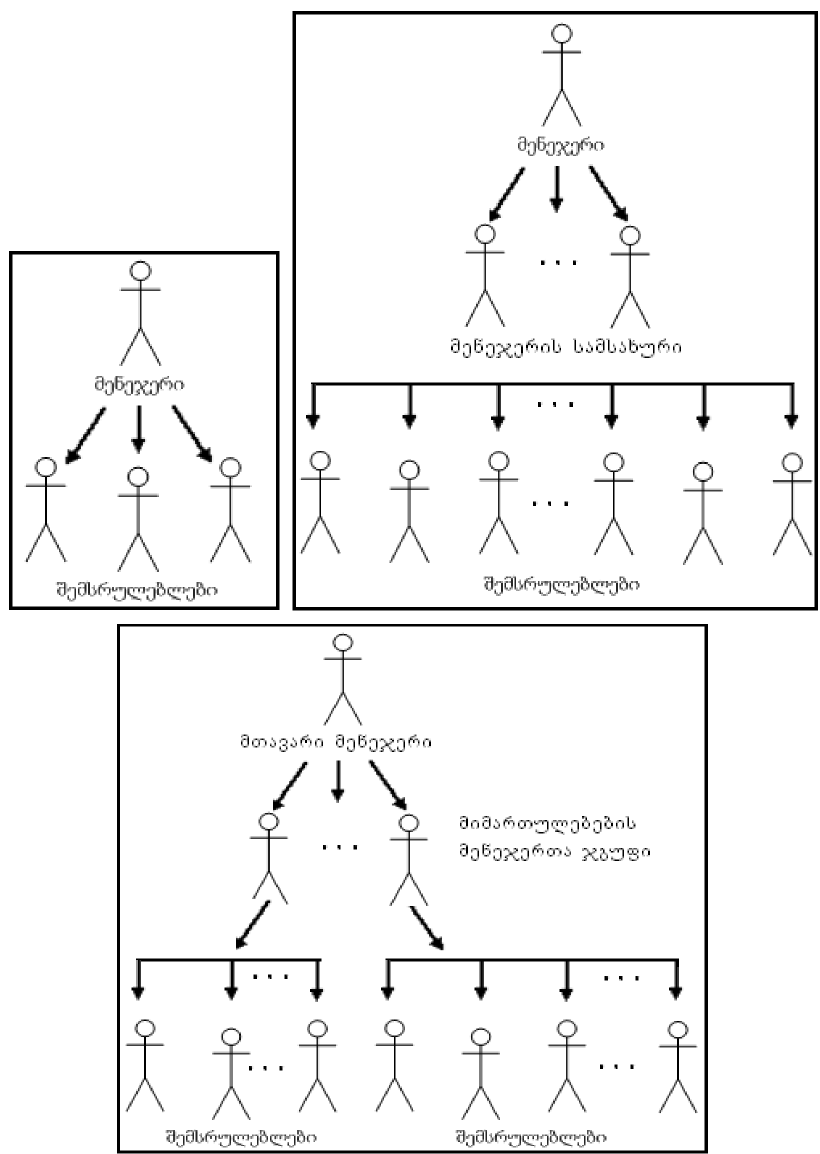
## 1.1. პროგრამული პროექტების მენეჯმენტის არსი, მიზნები და ამოცანები

პროგრამული სისტემების (პროექტების) დამუშავება ხშირ შემთხვევაში განიხილება როგორც სპეციალისტთა კოლექტიური ნაშრომი, რომელიც მიმართულია მომხმარებელთა მოთხოვნილე-ბების დასაკმაყოფილებლად მათი საქმიანი პროცესების (ბიზნეს-პროცესების) ავტომატიზაციის მიზნით [1-6].

როგორც ნებისმიერი სხვა კოლექტიური შრომა, ესეც მოითხოვს გარკვეულ ორგანიზაციას, კერძოდ, მენეჯმენტს (მართვა სოციალურ და ორგანიზაციულ სისტემებში). ეს ხანგრძლივი პროცესია, რომელიც ერთმანეთთან აკავშირებს პროგრამული პაკეტების მწარმოებელ სპეციალისტებს საწარმოო და სხვა დამოკიდებულებებში [7,8].

პროგრამული სისტემების დამუშავების აუცილებელი ელემენტია, ერთი მხრივ, მომხმარებელთა (მოთხოვნილებების) შესწავლა და, მეორე მხრივ, მათთან უკუკავშირის უზრუნველყოფა, რომლითაც წარიმართება პროგრამების შექმნის საწარმოო პროცესები. ასეთი ამოცანები{ს გადაწყვეტა ევალება ხელმძღვანელს ანუ პროექტის მენეჯერს [3,4,7].

პროგრამული სისტემის ზომებიდან და სირთულიდან გამომდინარე, პროექტს შეიძლება ჰყავდეს ერთპიროვნული მენეჯერი (მცირე პროექტი) ან მენეჯერული სამსახური (დიდი პროექტი), ან მთავარი მენეჯერი და მიმართულებათა მენეჯერები (განსაკუთრებით დიდი პროექტი) ნახ.1.




ნახ.1. პროექტის მენეჯმენტის ორგანიზაციის სქემები

პროგრამული სისტემების შექმნის პროექტების მენეჯმენტი, როგორც ნახაზიდან ჩანს, შრომის განაწილების ორგანიზაციის თვალსაზრისით, მეტად მრავალფეროვანია და დამოკიდებულია როგორც კონკრეტული პროექტის მიზნებზე, მის სირთულესა და შესრულების ვადებზე, ისე შრომით კოლექტივში დავალებათა დელეგირების (შესასრულებელი ფუნქციების გადანაწილება) განსხვავებული მოდელების გამოყენებაზე.

მაგალითად, პროგრამისტთა მცირე ჯგუფებში (2-:10 კაცი), გამოიყენება პროგრამული სისტემების „სწრაფი დამუშავების“ (agile development) პრინციპები, რომელიც თანამედროვე პროგრამირების თეორიაში ცნობილია „ექსტრემალური დაპროგრამების“ მეთოდის სახელით (იხ. Google: ავტორი Kent Beck). ამ შემთხვევაში მთავარი მენეჯერის მოვალეობები გადანაწილდება პროგრამისტთა გუნდის წევრებზე, რომლებიც თვითონ განსაზღვრავენ დავალებათა დაგეგმვის, შესრულებისა და კონტროლის საკითხებს. ექსტრემალურ დაპროგრამების მეთოდს ჩვენ შემდგომში დავუბრუნდებით.

პროგრამული სისტემების მენეჯმენტის ორგანიზაციის ყველა ვარიანტის ჩამოთვლა შეუძლებელია. ისინი ზოგჯერ აღმოცენდება სტიქიურად, ზოგჯერ კი იგეგმება პროგრამების წარმოების გარკვეული მეთოდოლოგიის გამოყენებით. ხშირად ასეთი სამუშაოები იგეგმება და წარიმართება არსებული კოლექტივის ტრადიციებიდან გამომდინარე.

პროგრამული პროექტების მენეჯერისთვის ყოველთვის საყურადღებოა ერთმანეთთან მჭიდროდ დაკავშირებული ორი ასპექტი:

- **პროექტის მართვა,** როგორც პროგრამული პროდუქტის წარმოების პროცესი გარკვეული მიზნის მისაღწევად;
- პროექტში მონაწილე ადამიანების **ხელმძღვანელობა**.

პროგრამული სისტემების დამუშავების მენეჯმენტის უმთავრესი და მუდმივი ამოცანაა პროექტის თანამიმდევრული რეალიზაცია-განვითარება დასმული მიზნისა და შედეგების მისაღწევად. გარდა იმ ხერხებისა და მეთოდებისა, რომლებითაც შესაძლებელია ასეთი ამოცანის გადაწყვეტა, ასევე საჭიროა პროექტის რესურსების ეფექტური განაწილებისა და გამოყენების კონტროლის საკითხის გადაწყვეტა. რესურსებში ტრადიციულად მოიაზრება დრო, ფინანსები, ტექნიკური საშუალებები და კადრების საწარმოო პოტენციალი.

პროექტის მენეჯერმა ყოველთვის უნდა გაითვალისწინოს ისეთი ორგანიზაციული და საწარმოო კონტექსტები, როგორიცაა პროექტის დამკვეთთა და შემსრულებელთა ინტერესების შეთანხმება, პროექტის შესასრულებლად საქმიანობის სფეროთა მრავალფეროვნება, აგერთვე სხვა კრიტერიუმების ერთობლიობა.

განასხვავებენ მომხმარებელთა და სისტემურ მოთხოვნი-ლებებს:

- **მომხმარებელთა მოთხოვნილებები,** ესაა ბუნებრივ ენაზე აღწერილი ფუნქციები (ბიზნესპროცესები) და დიაგრამები, რომლებიც სრულდება სისტემის მიერ, აგრეთვე მასზე დადებული შეზღუდვები (ბიზნესწესები).
- **სისტემური მოთხოვნილებები** არის სისტემური ფუნქციების და შეზღუდვების დეტალური აღწერა, რომელსაც ზოგჯერ ფუნქციურ სპეციფიკაციასაც უწოდებენ. იგი კონტრაქტის გაფორმების საფუძველია სისტემის დამკვეთსა და პროგრამული სისტემის შემსრულებელს შორის.

ზოგჯერ ასევე განიხილავენ **საპროექტო სისტემურ სპეციფიკაციასაც**, რომელიც პროგრამული სისტემის სტრუქტურის განზოგადებული აღწერაა. იგი საფუძველია სისტემის უფრო დეტალური დაპროექტებისა და მისი შემდგომი რეალიზაციისა.

ამგვარად, პროგრამული პროექტი შეიძლება განვიხილოთ როგორც სისტემის რეალიზაციის პროცესი, რომელიც აკმაყოფილებს სრულიად განსაზღვრულ შეზღუდვებს. ყველა შეზღუდვა არ ფორმირდება ერთბაშად, ხშირად ზოგი მათგანი აღმოცენდება პროექტის შესრულების პროცესში, ან თვით პროგრამული სისტემის გამოყენების ეტაპზეც კი. ამიტომაც მენეჯერმა უნდა გაითვალისწინოს ის გარემოება, რომ მას პროექტზე მუშაობა მოუხდება დიდი განუსაზღვრელობის პირობებში საბოლოო შედეგთან მიმართებით.

რა ამოცანების გადაწყვეტა უხდება მენეჯერს პროექტზე მუშაობისას, მისი მიზანმიმართულად და ეფექტიანად წარმართვის მიზნით ? აქ განვიხილავთ ორ საკითხს:

- ვინ მონაწილეობს პროექტის დამუშავებაში;
- რა ეტაპებს გადის პროექტი მისი განხორციელებისას.

პირველი მათგანი გამოკვეთს მენეჯერის ფუნქციებს სისტემის დამმუშავებლების გუნდში. აქ ორ ასპექტს ექცევა ყურადღება: მართვისას და ხელმძღვანელობისას.

მართვის თვალსაზრისით, პროექტის მონაწილეები - აბსტრაქტული მოქმედი აგენტებია, რომლებიც არსულებენ მათზე დაკისრებულ ფუნქციებს. იყენებენ გარკვეულ რესურსებს და იღებენ განსაზღვრულ შედეგებს. ასეთი ფუნქციური განხილვა შესაძლებელს ხდის პროექტის მონაწილეთა ურთიერთშენაც-ვლების შესაძლებლობას და მას მივყავართ „როლის“ ცნებამდე.

ხელმძღვანელობის თვალსაზრისით, მთავარი ამოცანაა საქმიანი, შეთანხმებული და მეგობრული გუნდის ჩამოყალიბება, რომელსაც შეუძლია პროექტის შესრულება. გუნდის ძირითადი მოთხოვნა მისი წევრების კომპეტენტურობა და კვალიფიკაციაა, მაგრამ ეს არასაკმარისია. გარდა ამისა, აუცილებელია გუნდური მუშაობის (ურთიერთდამოკიდებულება), ხელმძღვანელობასთან, დამკვეთებთან ურთიერთობის წესების ჩამოყალიბება.

ასეთი ურთირთობების შექმნა და მისი მართვა პროექტის მენეჯერის ამოცანაა, როგორც გუნდის ხელმძღვანელისა. იგი ამყარებს გარე და შიგა კონტაქტებს (იგულისხმება ურთიერთობები პროგრამული სისტემის დამკვეთებთან და საკუთარი ორგანიზაციის ხელმძღვანელობასთან).

პროგრამული სისტემის შესრულებისა და განვითარების ეტაპების საკითხები პროექტის მენეჯერის ამოცანაა, როგორც ხელმძღვანელისა. ის ანაწილებს ფუნქციებს გუნდის წევრებს შორის დროში. ადგენს გადასაწყვეტ ამოცანათა სირთულეს და ლოგიკურ თანამიმდევრობას, მათი გადაწყვეტის კოლექტიურ ხასიათს. ახორციელებს სამუშაოების შესრულების მენეჯერულ კონტროლს.

მთელი ეს პროცესი კავშირშია **პროგრამული უზრუნველ-ყოფის სასიცოცხლო ციკლი**ს ცნებასთან.

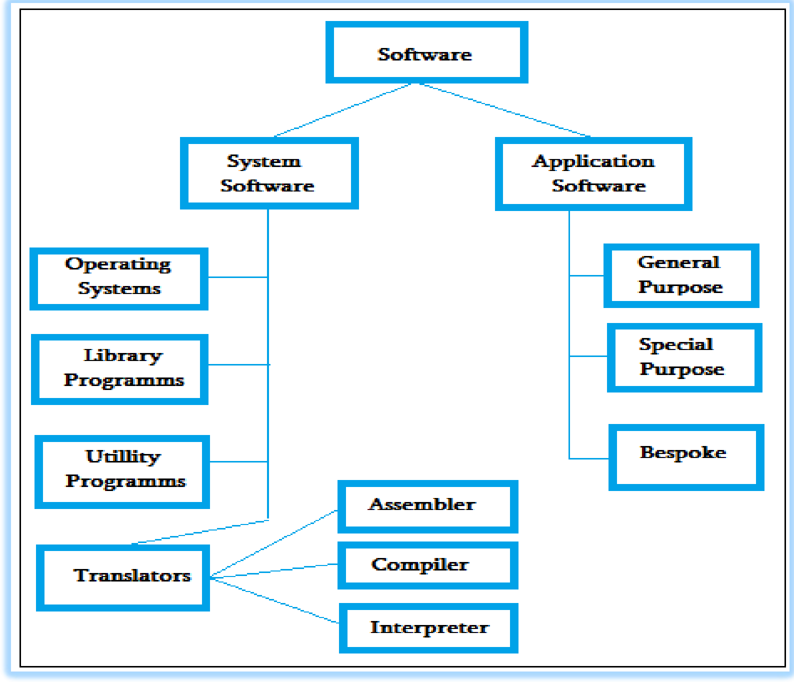
- **პროგრამული უზრუნველყოფის სასიცოცხლო ციკლი** არის პროგრამული უზრუნველყოფის კონსტრუირების პროცესი. პროგრამულ პროექტზე მოთხოვნილებანი განსაზღვრავს ამ პროცესის მიზანს და რეგლამენტს.
- არსებობს **მთავარი, საწარმოო და დამხმარე რესურსები.**
- **მთავარი რესურსებია: კადრები** (პუ-ს დამმუშავებლები, შემქმნელები), აგრეთვე პროგრამული პროექტის შესრულების **დრო** (ვადები) და **ფინანსები**, რომლებიც გამოიყოფა საპროექტო დავალების შესასრულებლად.
- **საწარმოო რესურსებია:** ტექნიკური საშუალებები, აგრეთვე პროგრამები, რომლებიც გამოიყენება ინსტრუმენტული საშუალე-ბების, პროტოტიპების ან ჩასასმელი კომპონენტების სახით.
- **დამხმარე რესურსებია:** გათბობა, ელექტროენერგია, სამუშაო ოთახები, ტექნიკური და ორგანიზაციული საშუალებები, რომლების უშუალოდ პროექტის წარმოებასთან არაა კავშირში, მაგრამ ხელს უწყობს მის წარმატებით შესრულებას.

- **ადამიანური** რესურსი კარგი პროექტისათვის სტაბილურია, არ იცვლება პროექტის შესრულების პროცესში. საქმის ცუდად ორგანიზებისას ადგილი აქვს კადრების დენადობას, რაც პროგრამული პროექტების საბოლოო ხარისხზე უარყოფით გავლენას ახდენს.
- **დრო** - შეუქცევადი რესურსია, ამიტომ მისი სწორად დაგეგმვა მენეჯერის მთავარი საზრუნავია.
- **ფინანსები** - ძირითადი სახარჯო რესურსია, რომელიც ყოველთვის შეზღუდულია. კერძოდ, იგი გამოიყენება არა მხოლოდ ხელფასის სახით, არამედ ტექნიკური და ინსტრუმენტული საშუალებების შესაძენად.
- პროგრამული პროექტების მენეჯმენტისათვის საჭიროა **რესურსების ხარჯვის ოპტიმალური ბალანსის შედგენისა და მისი შესრულების** ამოცანების გადაწყვეტა. ეს ამოცანები ეხება ყველა სახის პროექტის მენეჯერს. მასზე მოქმედებს პროექტის სპეციფიკა და მისი შესრულების პირობები.
- არსებობს მენეჯმენტის მხარდამჭერი მიდგომების, მეთოდების, მეთოდიკებისა და ტექნოლოგიების სიმრავლე, მაგრამ მათი გამოყენება არაა მარტივი და გარკვეულ ცოდნას და მომზადებას მოითხოვს. აგრეთვე პროგრამული პროექტები ხშირად ინდივიდუალურია და დიდი მნიშვნელობა ექცევა მენეჯერისა და შემსრულებლების პრაქტიკულ გამოცდილებას.
- ამგვარად, მენეჯერის ამოცანების განხილვისას პროგრამული პროდუქტების დამუშავების პროცესში გამოიკვეთება სამი ურთიერთდაკავშირებული მიმართულება:

1. ფუნქციები, რომლებიც აუცილებელია პროექტის წარმატებული განვითარებისთვის. აქ აუცილებელია გაირკვეს მოცემული პროექტისათვის თანამშრომელთა რომელი როლებია საჭირო;
2. პროექტის დაგეგმვა და შესრულების კონტროლი შესაქმნელი პროგრამული სისტემის სასიცოცხლო ციკლის შესაბამისად;
3. პროექტის გუნდის ფორმირება (კადრები).

## 1.2. პროგრამული ინჟინერიის ტექნოლოგიების რეტროსპექტული ანალიზი

პროგრამული უზრუნველყოფის (Software) დამუშავება (Software Engineering, Development) განსაკუთრებულად რთული სისტემების კლასს მიეკუთვნება. მრავალფეროვანია დღეისათვის არსებული პროგრამული სისტემების სიმრავლე და მათი სახეები. თუ მათ კლასიფიკაციას შვეხებით, ძირითადად შეიძლება ორი დიდი მიმართულება გამოვყოთ: სისტემური პროგრამები (System Software) და გამოყენებითი პროგრამები (Applied Software), რომლებიც თავის მხრივ სპეციალური სახის პროგრამულ ქვესიმრავლეებად იყოფა (ნახ.2) [10].




ნახ.2. პროგრამული უზრუნველყოფის კლასიფიკაცია

ჩვენი წიგნის კვლევის ობიექტი, როგორც ზემოთ აღვნიშნეთ, გამოყენებითი პროგრამული პაკეტების მენეჯმენტის საკითხებია. კერძოდ კი, კორპორაციული ორგანიზაციების ბიზნესპროცესების ავტომატიზაციის მიზნით შესაბამისი ინფორმაციული, მოდელური და პროგრამული უზრუნველყოფის შემუშავება ახალი ტექნოლოგიების ინტეგრაციის საფუძველზე, ინფორმაციული უსაფრთხოების დაცვით [11].

პროგრამული სისტემების მენეჯმენტის თანამედროვე მეთოდოლოგიები, პროგრამული პარადიგმების განვითარების ფონზე, მნიშვნელოვნად განსხვავდება ერთმანეთისაგან, რაც, ერთგვარად, მრავალ ობიექტურ და სუბიექტურ ფაქტორზეა დამოკიდებული. ამ მიმართულებათა მიმდევარი მეცნიერები და პრაქტიკოსი პროგრამისტები 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 (ზ. კენტი, დ. მარტინი და სხვ.) [12,13]. თუმცა ისეთებიც მოინახება, რომლებიც კომპრომისული მიდგომით ჰიბრიდულ ვარიანტსაც გვთავაზობენ (ს. ამბლერი, ბ. რუმპე და სხვ.) [14,15].

საქართველოში ამ მიმართულებას ავითარებენ სტუ-ს პროფესორები გ.გოგიჩაიშვილი, გ. სურგულაძე, ე. თურქია, თ. სუხიაშვილი [1,2,5,9, 15-18].

ახლა მოკლედ განვიხილოთ გამოყენებითი პროგრამული სისტემების დაპროგრამების ტექნოლოგიების განვითარების ისტორია, რაც ლიტერატურულ წყაროებში ვრცლადაა წარმოდგენილი [19-21].

თვდაპირველად საჭიროა გაირკვეს, თუ რა პრობლემების გადაჭრის მიზნით წარმოიშვა მოქნილი მეთოდოლოგიები. მოკლედ რომ ვთქვათ, პრობლემის არსი ისაა, რომ დაწყებული

პირველი პროგრამული პროექტებიდან დღემდე, პროგრამული უზრუნველყოფის დამუშავება იყო და რჩება ნაკლებ პროგნოზირებად და ხშირად წარუმატებელ საქმედ [19].

პროგრამული უზრუნველყოფის შექმნის პროექტების დიდი რაოდენობა კვლავაც მთავრდება ბიუჯეტისა და ვადების გადამეტებით, ხოლო შედეგად შექმნილი პროგრამები ხშირად ბოლომდე ვერ პასუხობს მომხმარებელთა მოთხოვნებს, ან მოაქვთ მცირე რეალური სარგებლობა ბიზნესისათვის.

აღნიშნული პრობლემები არის პროგრამული უზრუნველ-ყოფის კრიზისის ძირითადი გამოვლინება. მიუხედავად მნიშვნელოვანი ინტელექტუალური ძალისხმევისა კრიზისის დასაძლევად, დღემდე ვერ მოინახა რაიმე უნივერსალური გადაწყვეტა (როგორც ხშირად ამბობენ „ვერცხლის ტყვია“ - მეტაფორა ფრედ ბუკსის 1986 „No Silver Bullet“: IT-ში ახლი ეფექტური ტექნოლოგიური გადაწყვეტა („ვამპირების წინააღმდეგ“), მაგალითად, პროექტების მართვის სტრუქტურული მიდგომა) [20,21].

პროექტების სტრუქტურული მართვის არსი მდგომარეობს 10-ეტაპიანი სამუშაოს შესრულებაში:

1. მიზნის ვიზუალური წარმოდგენა; ყურადღების გამახვილება პრიზისათვის;

2. ამოცანათა სიის შემუშავება, რომლებიც უნდა შესრულდეს;

3. უნდა არსებობდეს მხოლოდ ერთი ლიდერი;

4. ადამიანთა მიმაგრება ამოცანებზე;

5. მოსალოდნელი შედეგების მართვა, რეზერვების გათვლა მოსალოდნელი შეცდომებისთვის, სათადარიგო პოზიციების გამომუშავება (რისკების მართვა);

6. ხელმძღვანელობის შესაბამისი სტილის გამოყენება;

7. ცოდნა იმისა, რაც ხდება;

8. შემსრულებელთა ინფორმირება იმაზე, რაც ხდება;

9. გამეორდეს 1-8 ეტაპები მე-10 ეტაპამდე;

10. პრიზი (ჯილდო).

ამ ეტაპების შესრულება გარანტიაა პროექტის წარმატების დასასრულებლად. გამოიყენება ტერმინები: „ჩანჩქერის“ მეთოდი, იტერაცია. დაგეგმვა. რისკები.

**მოქნილი მეთოდები** - ესაა პროგრამული ინდუსტრიის თანამედროვე პასუხია თუ როგორ უნდა შესრულდეს პროექტები, რათა ისინი წარმატებით დასრულდეს მაღალი ალბათობით და მოუტანოს კმაყოფილება ყველა დაინტერესებულ მხარეს, პირველ რიგში, დამკვეთს და პროექტის შემსრულებელ გუნდს [19,22].

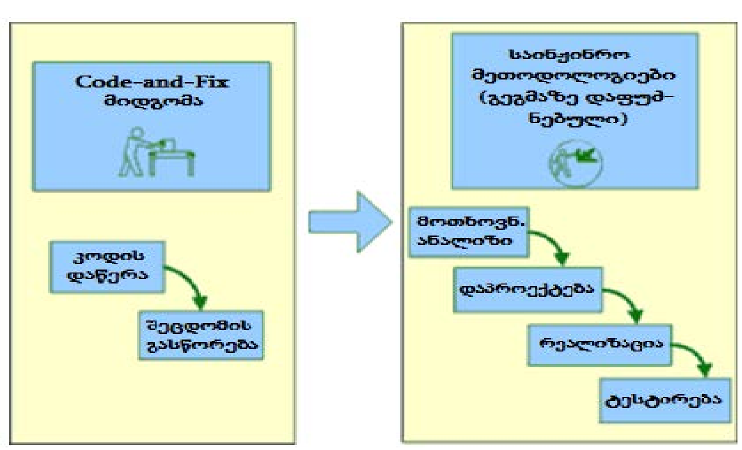
მოკლედ  გადავავლოთ თვალი პროგრამული უზრუნველ-ყოფის განვითარების ისტორიას. დაპრგრამების ენების განვი-თარება და დიდი პროგრამული სისტემების შექმნა დაიწყო 1950-იანი წლებიდან. დაპროგრამების თავდაპირველი მიდგომები იყო ნაკლებად ფორმალიზებული და წარმოადგენდა პროცესს სახელწოდებით Code-and-Fix (კოდირება და გასწორება).

ამ მიდგომის დროს პროგრამული უზრუნველყოფის დამუშავება იწყებოდა უშუალოდ კოდირებით (ყოველგვარი წინასწარი დაგეგმვის, მოთხოვნილებათა ანალიზისა და დაპროექ-ტების გარეშე). ამის შემდეგ კოდში ნაპოვნი პრობლემები (დეფექტები, მოთხოვნილებებთან შეუთავსებლობა და ა.შ.) სწორდებოდა ხელით უშუალოდ კოდში ცვლილებების შეტანით.

დროთა განმავლობაში გასაგები გახდა, რომ დიდი სტაბილური პროგრამული სისტემების ასაგებად საჭირო იყო უფრო გააზრებული და ფორმალური მიდგომები. პრობლემების გადასაწყვეტად ყურადღება მიექცა იმ დროისთვის უკვე კარგად განვითარებულ ისეთ ადამიანური შემოქმედების სფეროს, რომელიც კავშირშია რთულ საწარმოო პროცესებთან. მაგალითად, როგორიცაა სისტემოტექნიკა (System Engineering), დაპროექტება და სხვა საინჟინრო დისციპლინები.

ასე გაჩნდა ახალი მიმართულება - პროგრამული ინჟინერია (Software Engineering), ხოლო ფაქტობრივი სტანდარტის სახით წლების განმავლობაში დამკვიდრებულ იქნა ე.წ. პროგრამული უზრუნველყოფის დამუშავების საინჟინრო მეთოდოლოგიები. ამ მეთოდოლოგიებს ასევე ხშირად უწოდებენ გეგმაზე დაფუძნებულს (Plan-driven), რადგანაც მის საფუძველში ძევს მოსაზრება, რომ პროგრამული უზრუნველყოფის დამუშავების პროცესი არის დეტერმინირებული ინჟინრული პროცესი, რომელიც შეიძლება დაიგეგმოს თავიდან ბოლომდე და შემდეგ შესრულდეს გეგმის მიხედვით ფორმალური ინჟინრული მიდგომების გამოყენებით.

პროგრამის სასიცოცხლო ციკლის აგების ძირითად ვარიანტად გამოიყენებოდა „ჩანჩქერის“ მოდელი, რომელიც ითვალისწინებს მოთხოვნილებათა ანალიზის, დაპროექტების, კოდირების, ტესტირების და სხვა ფაზების ერთჯერადად შესრულებას (ნახ.3).




ნახ.3. საინჟინრო მეთოდოლოგიებზე
გადასვლა

გარდა აქ განხილული ორი მიდგომისა, აგრეთვე არსებობდა სასიცოცხლო ციკლის სხვა ვარიანტებიც, როგორც გარკვეული მოდიფიკაციები ჩანჩქერული და იტერაციული მოდელებისა. ჩვენ აქ ვიხილავთ მხოლოდ ძირითადს, რომლებიც პრაქტიკაში მასობრივად გამოყენება.

- **საინჟინრო მეთოდოლოგიების პრობლემები:**

გადასვლა საინჟინრო მეთოდლოგიებზე და ჩანჩქერულ მოდელზე ნამდვილად იყო წინგადადგმული ნაბიჯი. მან შემოიტანა განსაზღვრული მოწესრიგება და ორგანიზებულობა დამუშავების პროცესში, ასევე აღმოფხვრა Code-and-Fix მიდგომის მრავალი პრობლემა [19].

მაგრამ საინჟინრო მეთოდოლოგიებმა ვერ შეძლეს პროგრამული პროექტების ყველა პრობლემის გადაწყვეტა, უპირველეს ყოვლისა, მასში ჩადებული შინაგანი კონფლიქტის გამო. საქმე ისაა, რომ ისინი თავდაპირველად იქმნებოდა სხვა საინჟინრო დისციპლინების ნიმუშის მიხედვით და სრულად არ ითვალისწინებდა თავისებურებებს, რომლებიც პროგრამულ უზრუნველყოფას ახასიათებს, როგორც უნიკალური სახის პროდუქციას (მაგალითად, Java-პროგრამული პროდუქტის შექმნა და მაცივრის წარმოება).

პროგრამული უზრუნველყოფის უნიკალურობის კონ-ფლიქტს საინჟინრო მეთოდოლოგიების მიდგომებთან აქვს მინიმუმ ორი ძირითადი გამოვლიენება:

1. **ადამიანური ფაქტორის როლის არასაკმარისი შეფასება.**

საინჟინრო მეთოდოლოგიები განიხილავდა პროგრამული უზრუნველყოფის დამუშავების პროცესს როგორც აბსტრაქტული შემსრულებლების მიერ ჩატარებულ ბიჯების თანამიმდევრობას, ანუ ძირითადი ყურადღება მახვილდებოდა მხოლოდ იმაზე თუ რა უნდა გაკეთებულიყო და არა იმაზე, ვის უნდა გაეკეთებინა.

სინამდვილეში, პროგრამული პროექტების უმეტეს პრობლემას აქვს სოციალური და არა ტექნოლოგიური ხასიათი. სწორედ პროექტის მონაწილეები და მათ შორის ურთიერთობები არის პროექტის წარმატების განმსაზღვრელი ძირითადი ფაქტორი. როგორც ამ სფეროში გამოკვლევები გვიჩვენებს, კომპანიის საუკეთესო დეველოპერების მწარმოებლურობა 10-ჯერ მაღალია, ვიდრე დაბალი დონისა, და 2,5-ჯერ მეტი, ვიდრე საშუალოსი. ამგვარად, მწარმოებლურობაზე მოქმედი ძირითადი ფაქტორებია არა დამუშავების საშუალებები და მეთოდოლოგიები, არამედ სამუშაო ადგილის ხარისხი, ფსიქოლოგიური გარემო გუნდში, ეფექტური კომუნიკაციები და სხვა სოციალური ფაქტორები;

2. **ჩანჩქერული სასიცოცხლო ციკლის შეუსაბამობა პროგრამული უზრუნველყოფის ბუნებასთან.**

აქ არ განიხილება ჩანჩქერული მოდელის დეტალები (ისინი მრავლადაა ლიტერატურულ წყაროებში). საჭიროა აღინიშნოს მხოლოდ პრინციპული მომენტი, რომ ყოველი პროგრამული პროდუქტი პრაქტიკულად არის უნიკალური და ახალი. მისი დამუშავება მიმდინარეობს მაღალი განუსაზღვრელობის პირობებში, და მცდელობა იმისა, რომ წინასწარ იქნას გათვალისწინებული ყველა ფაქტორი პროექტის დასაწყისში, (დეტალური მოთხოვნების შედგენა, სისტემის დიზაინის სრული დამუშავება და სხვ.) წინასწარაა განწირული წარუმატებლობისა-თვის. ჩანჩქერული მოდელის ნაკლოვანებები არა მხოლოდ ართულებს პროგრამული სისტემის დამუშავებას, არამედ იგი ავიწროვებს ადამიანურ ურთიერთობებს. მაგალითად, პროექტის დასაწყისში შედგენილი და დამტკიცებული დეტალური მოთხოვნები ხშირად ხდება დამკვეთისა და მიმწოდებლის უთანხმოების მიზეზი რეალიზაციის სტადიაზე, როცა საჭიროა აუცილებელი ცვლილებების განხორციელება. რა თქმა უნდა, არსებობს პროექტებიც, რომლისთვისაც ჩანჩქერული სასიცოცხლო

ციკლი მისაღებია, მაგრამ უმრალესი პროექტისათვის ზემოაღწერილი პრობლემები აქტუალურია და მათ არგათვალისწინებას მივყავართ პროექტების ჩავარდნამდე.

დასკვნის სახით შეიძლება ითქვას, რომ საინჟინრო მეთოდოლოგიებმა ვერ შემლო პროგრამული უზრუნველყოფის კრიზისის გადაწყვეტა. პროექტების წარუმატებლობის მრავალმა შემთხვევამ, რომლებიც საინჟინრო მეთოდების საფუძველზე იქმნებოდა, დღის წესრიგში დააყენა ალტერნატიული მიდგომების ძიების საკითხი, რაც შემდგომ განხორციელდა და ხორციელდება კვლავ.

- **მსუბუქი მეთოდოლოგიები:**

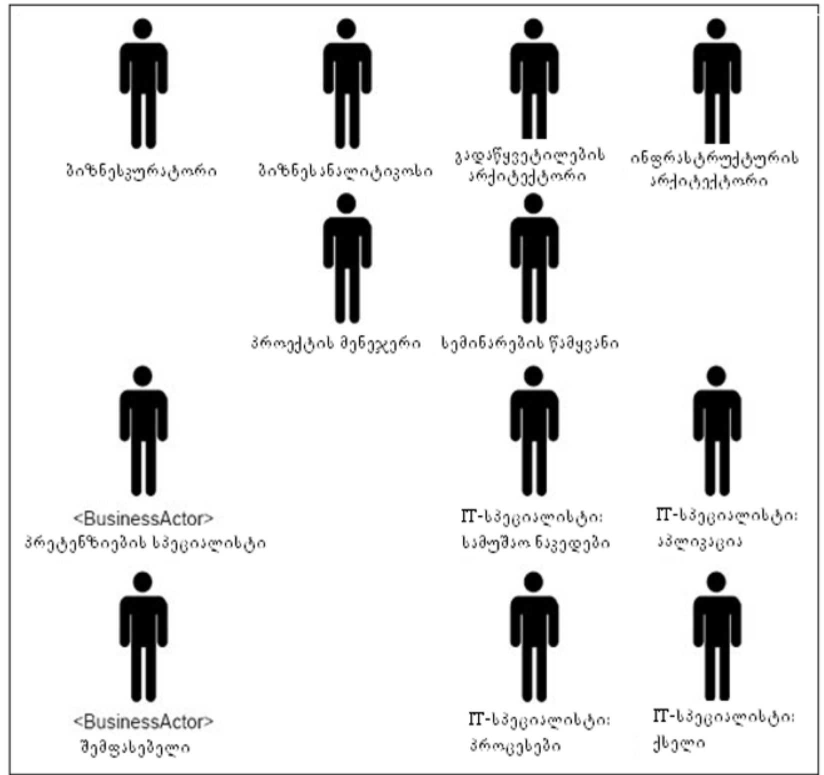
პროგრამული უზრუნველყოფის დამუშავების საინჟინრო მეთოდოლოგიების პრობლემების შეცნობამ გამოიწვია ახალი ალტერნატიული მიდგომების შექმნა ამ მიზნით. განსაკუთრებით სწრაფად განვითარდა ეს პროცესი 90-იანი წლების მეორე ნახევარში, როცა დიდი პოპულარობა მოიპოვა მსუბუქმა (ან მსუბუქი წონის) მეთოდებმა, როგორებიცაა Scrum, DSDM, Crystal და სხვა [19,23]. მიუხედავად ზოგიერთი განსხვავებისა, პრაქტიკული გამოყენების თვალსაზრისით, ეს მიდგომები მსგავსია იმით, რომ ალტერნატივის სახით მძიმე და ზედმეტად ფორმალიზებული საინჟინრო მეთოდოლოგიებისაგან განსხვავებით, ისინი იძლევა ადაპტურ, იტერაციულ და ადამიანზე ორიენტირებულ მიდგომებს.

მსუბუქი მიდგომების ავტორები, მ. ფოულერი, კ. ბეკი, ა. კოუბერნი და სხვ. [13,24]. ერთობლივი შეხვედრებისა და კონსულტაციების საფუძველზე (მაგალითად, 2001წ. სნოუბორდში, იუტას შტატი) მივიდნენ იმ გადაწყვეტილებამდე, რომ ამ მიმართულებისათვის დაერქმიათ „Agile” (მოქნილი, სწრაფი). შეიქმნა დოკუმენტი „პროგრამული სისტემების მოქნილი დამუშავების მანიფესტი“ (4 პუნქტით), მას მოგვიანებით დაემატა პრინციპების სია (12 პუნქტით), ის დეტალურად ხსნიდა მანიფესტს [24].

## 1.3. პროგრამული სისტემების დამუშავების ბიზნესპროცესის გუნდი: როლები და ფუნქციები

როგორც აღინიშნა, პროგრამული სისტემების დაპროექტების და რეალიზაციის პროცესებში მნიშვნელოვანი ადგილი უჭირავს ადამიანური რესურსების ეფექტურ გამოყენებას. ყოველი ახალი კონკრეტული პროექტის დასაწყისში ყალიბდება შემსრულებელთა გუნდი, რომლის ხელმძღვანელობა პროგრამული პროექტის მენეჯერს ევალება. მე-4 ნახაზზე მოცემულია პროგრამული უზრუნველყოფის შექმნის გუნდის ტიპური როლები და მათი ფუნქციები [5]:

- **ბიზნესკურატორი:** ორგანიზაციის ხელმძღვანელი, რომელიც პასუხისმგებელია პროექტის მონიტორინგზე და პროექტის შემსრულებლებზე საქმიან ხელშეწყობაზე, კონსულტაციების მიცემაზე. აქვს დირექტორის მიერ მინიჭებული უფლებამოსილება.
- **ბიზნესანალიტიკოსი:** ესაა კონკრეტული სფეროს სპეციალისტი, რომელიც კარგად ერკვევა შესაბამისი კომპანიის საქმიანობაში. გაცნობიერებული აქვს ბიზნესის კონკრეტული ფუნქციები, რაზეც იგი პასუხისმგებელია. აწარმოებს დაგეგმვისა და ცვლილებების ოპერაციებს ბიზნესის მოთხოვნების შესა-ბამისად. ამავდროულად ბიზნესანალიტიკოსი უნდა იყოს სპეციალისტი მოდელირების საშუალებებში, რომელიც გამოიყე-ნება ბიზნესპროცესების აღწერის, ანალიზისა და დოკუმენტი-რებისათვის (Ms Visio, EA, Rational_Rose).




ნახ.4. ბიზნესპროცესში მონაწილე როლები
(აბსტრაქტული დაზღვევის კომპანიის მაგალითი)

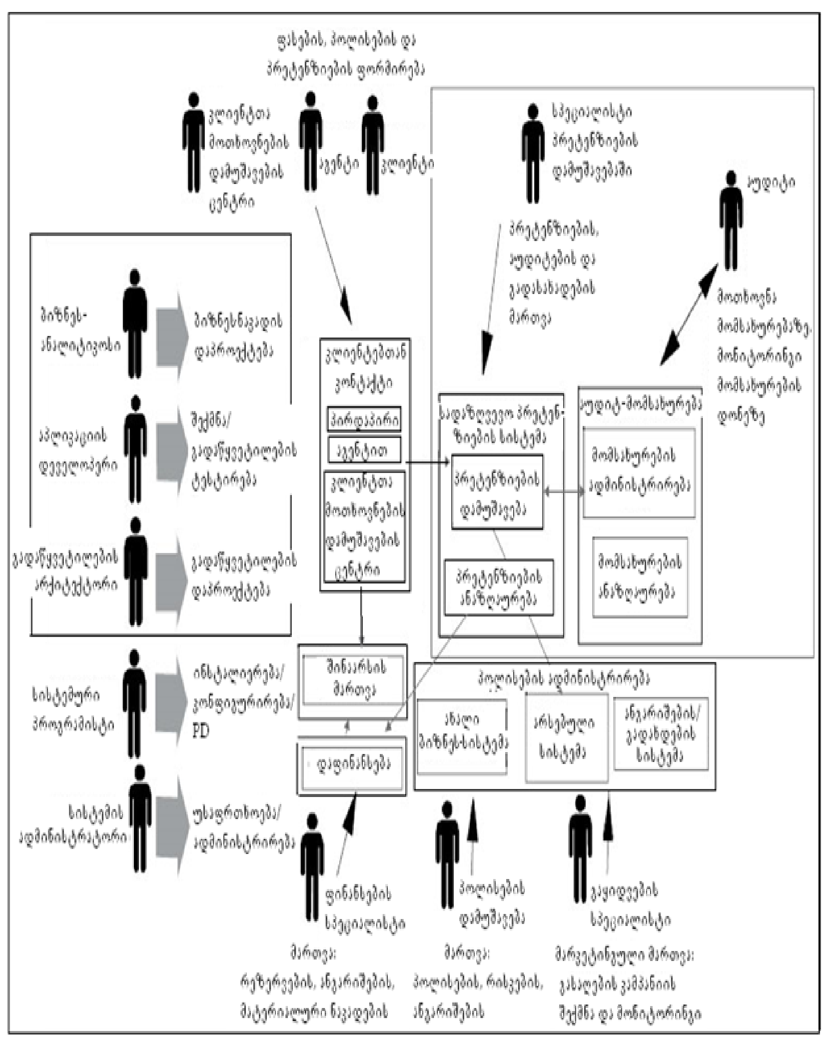
- **გადაწყვეტილების არქიტექტორი:** არის სპეციალისტი, რომელსაც ესმის ბიზნესმიზანი, რისი მიღწევისთვისაც იქმნება მართვის ავტომატიზებული სისტემა, აქვს სპეციალური ცოდნა ბიზნესის კონკრეტულ ასპექტებზე, კერძოდ, პრეტენზიების დამუშავების პროცესებზე. გადაწყვეტილების არქიტექტორი

კარგად ერკვევა არსებულ IT-ინფრასტრუქტურაში, იგი პასუხს აგებს ახალი ბიზნესმიზნებისა და ტექნიკური გადაწყვეტილებების შესაბამისობის დამყარებაში, რომელიც თავსებადია არსებულ IT-სტრუქტურასთან. მას აქვს ცოდნა SOA (სერვის ორიენტირებული აპლიკაციების) და Web-სამსახურების შესახებ.

- **პრეტენზიების დამუშავების სპეციალისტი:** ჩვენს შემთხვევაში ესაა სადაზღვევო კომპანიის წარმომადგენელი, რომელიც მართავს საპრეტენზიო გადასახადების საქმეს, ადმინისტრირებას უწევს პოლისებს და ადგენს გარე რეპორტებს, რომლებიც აუცილებელია პრეტენზიათა დასაკმაყოფილებლად. გარე შემფასებლებთან (აგენტებთან) ურთიერთობებისათვის დავალებების მისაწოდებლად იგი იყენებს ტელეფონს, ფაქსს ან ელ-ფოსტას.
- **გარე შემფასებელი (აუდიტები):** ესაა მომსახურების მიმწოდებელი სპეციალისტები, რომლებიც არაა სადაზღვევო კომპანიის საშტატო თანამშრომლები. ისინი მუშაობენ სააუდიტო ფირმებში და, როგორც დამოუკიდებელი ექსპერტები, შეაფასებენ სადაზღვევო ობიექტებს (მაგალითად, დაზიანებულ ავტომანქანებს). აუდიტი, მიიღებს რა დავალებას პრეტენზიების დამუშავების სპეციალისტისაგან, გადის შესაფასებელ ობიექტთან და ადგენს დაზიანების მოცულობას. შედგების რეპორტს იგი გადააგზავნის დანიშნულებისამებრ.
- **პროექტის მენეჯერი:** თანამშრომლობს ორგანიზაციის მომხმარებლებთან, ბიზნესანალიტიკოსებთან, აუდიტებთან, თვალყურს ადევნებს ბიუჯეტის ფარგლებში მოთხოვნილებათა შესრულებას.

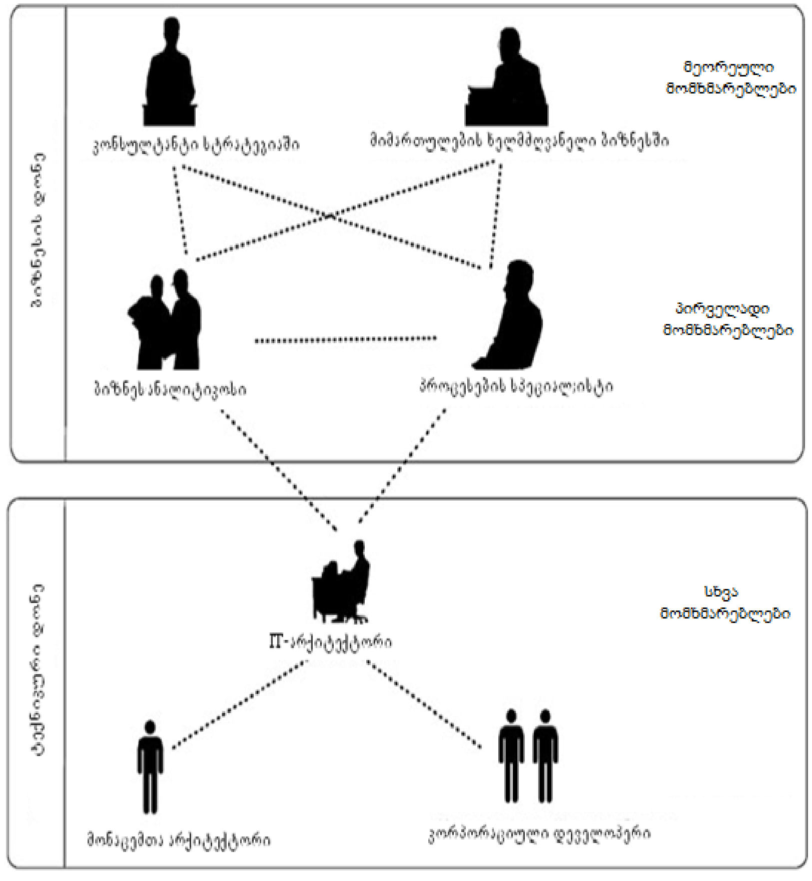
- **სემინარების წამყვანი:** ეს შეიძლება იყოს პროექტის მენეჯერი, ბიზნესანალიტიკოსი ან პროცესების სპეციალისტი, რომელსაც აქვს პრეზენტაციების ჩატარების გამოცდილება. იგი იყენებს Ms Visio ან Rational Rose ან Enterprise Architect, PowerPoint და ა.შ.;
- **IT-სპეციალისტი (აპლიკაციის დეველოპერი):** გამოყენებითი სფეროს დანართის ანუ აპლიკაციის დამმუშავებელი ღრმად ერკვევა ბიზნესკომპონენტებში, რომლებიც შედის შემოთავა-ზებულ გადაწყვეტაში. მან უნდა დააპროგრამოს ეს კომპონენტები ახალი დანართებისათვის, მაგალითად, C++, Java, C# ან სხვა ინსტრუმენტების საშუალებით, აგრეთვე უნდა შეეძლოს არსებულ სისტემაში მონაცემების ფორმატებთან მუშაობა;
- **ინფრასტრუქტურის არქიტექტორი:** მისი დანიშნულებაა IT-ინფრასტრუქტურის სტანდარტების განსაზღვრა და ბიზნესის სტრატეგიული განვითარებისთვის IT-მიზნების ჩამოყალიბება. იგი კარგად ერკვევა თანამედროვე IT -ტექნოლოგიებში, მათ საიმადოობასა და მოთხოვნების დამუსავების გამტარუნარიანობის საკითხებში ორგანიზაციის შესაბამისად;
- **მონაცემთა არქიტექტორი:** ეხება კორპორაციის მონაცემთა მოდელირება, მონაცემთა ბაზის და მისი ცხრილების განსაზღვრა. მას კარგად უნდა ესმოდეს ბიზნესის საქმე და იყოს პროფესიონალი მონაცემთა რელაციურ ბაზებში. იცის UML-ის ინსტრუმენტი;
- **უსაფრთხოების არქიტექტორი:** განსაზღვრავს მონაცემთა დაცვისა და უსაფრთხოების საკითხებს. პასუხს აგებს ინფორმაციის კონფიდენციალობაზე.

მე-5 ნახაზზე ნაჩვენებია დაზღვევის კორპორაციის საილუს-ტრაციო მაგალითი აქ არსებული როლებით [5].




ნახ.5. კორპორატიული სისტემა: აბსტრაქტული დაზღვევის კომპანიის მაგალითზე

მე-6 ნახაზზე მოცემულია კორპორაციის ბიზნესპროცესების მოდელირების ინსტრუმენტების მომხმარებელთა როლები და მათი ურთიერთკავშირების სქემა.




ნახ.6. ბიზნესპროცესების მოდელირების ინსტრუმენტების მომხმარებლები

## 1.4. პროგრამული პროექტის სისტემა და მისი ელემენტები

პროცესი: სუბიექტის მიერ მიზანმიმართული საქმიანობა განსაზღვრული შედეგის მისაღებად, გარკვეული მასალების, რესურსებისა და ტექნოლოგიების (მეთოდები და ინსტრუმენ-ტები) გამოყენებით.

ნახ.7. პროგრამული პროექტის შესრულების

**მიზანი:**
მომხმარებლის პრობლემების გადაწყვეტა მისი მოღვაწეობის ავტომატიზაციის გზით

**მასალები და რესურსები:**
მოთხოვნები პროგრამულ სისტემასთან, მოწყობილობები, დამხმარე რესურსები

**სუბიექტი:**
პროექტის შემსრულებელი პროგრამისტთა გუნდი, სამუშაოს ინიციატორები

**შედეგი:**
პროგრამული სისტემა, დოკუმენტაცია, ტესტები, მუშაობის მეთოდიკა, სასწავლო მასალები

**საშუალებები და ინსტრუმენტები:**
დაპროგრამების სისტემები, ბიბლიოთეკები, CASE-საშუალებები

**მეთოდები:**
განვითარების სტრატეგიები, მეთოდოლოგიები, რეგლამენტები და შეთანხმებები

პროცესის მოდელი

ელემენტები:

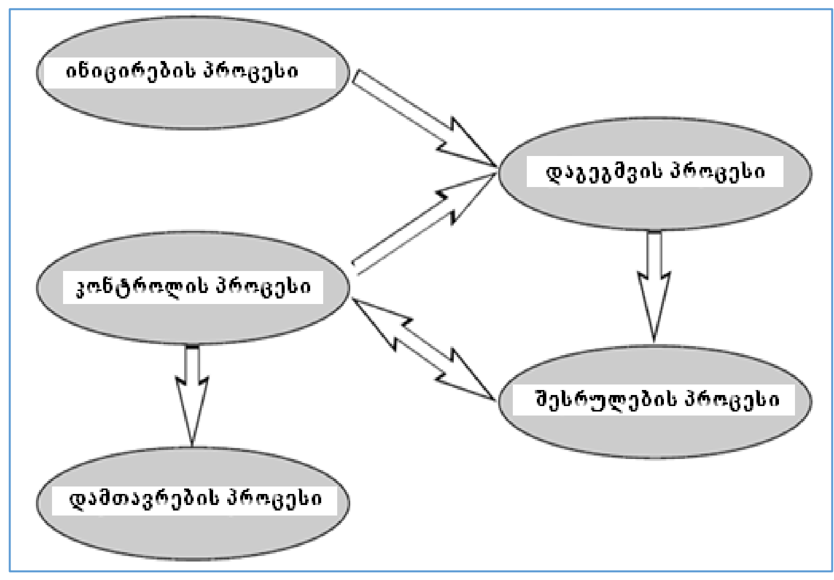
- **მიზანი** – ის, რისთვისაც მუშავდება პროგრამული სისტემა;

- **სუბიექტი** – ის, ვინც ქმნის პროგრამულ სისტემას (პიროვნება, გუნდი);
- **მასალები და რესურსები** – ის, საიდანაც იწარმოება პროგრამული სისტემა;
- **საშუალებები და ინსტრუმენტები** – ის, რითაც იწარმოება პროგრამული სისტემა;
- **მეთოდები** – ის, რის საფუძველზეც იქმნება სისტემა;
- **შედეგი** – ის, რაც უნდა იქნას მიღებული პროექტის შესრულებისას.

პროექტების მენეჯმენტის საერთაშორისო ინსტიტუტის (*PMBOK*-Project Management Body of Knowledge) განსაზღვრებით: „პროცესი არის ქმედებათა სერია შედეგის მისაღებად“.
იგი გამოყოფს პროცესთა ჯგუფებს:

- **ინიცირების პროცესები**: იწყებს პროექტს, განსაზღვრავს მისთვის მიზანს, საწყის რესურსებს, მასალებს და სხვა ელემენტებს. ესაა წინასაპროექტო ქმედების დასაწყისი;
- **დაგეგმვის პროცესები:** განსაზღვრავს, რა ხერხებით (მეთოდებით) არის სავარაუდო საპროექტო ქმედებების ორგანიზება, რა ვადებში და რა რესურსებით განხორციელდება პროექტი, რა ქვეპროცესებისა და ეტაპებისგან შედგება იგი და როგორ იქნება შემოწმებული მათი შედეგები;
- **შესრულების პროცესები:** დაგეგმილი პროცესების ეტაპობრივი შესრულება კონკრეტულ პირობებში;
- **კონტროლის პროცესები:** პროცესების მსვლელობის და მათი შედეგების შემოწმება;
- **დამთავრების პროცესები:** ქმედებები, რომლებიც სრულდება პროექტის დამთავრებისას;

პროცესთაშორის კავშირები ასახულია მე-8 ნახაზზე:

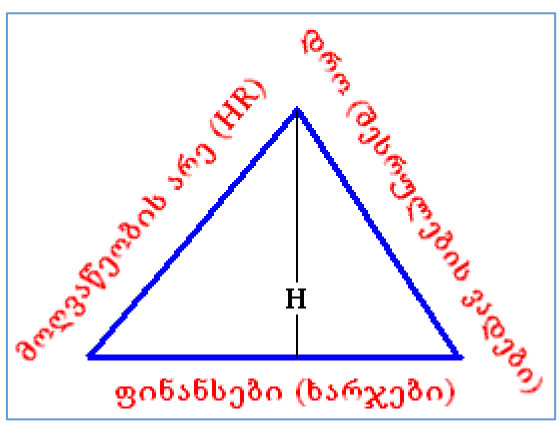



ნახ.8

ეს სქემა აბსტრაქტულ დონეზე ასახავს პროექტების მენეჯმენტის არსს, როგორც ქმედებათა ერთობლიობისა, რომელიც პროცესთა ჯგუფების მიზანმიმართული შესრულების ორგანიზაციის მხარდამჭერია.

## 1.5. პროექტების მენეჯმენტი

ამგვარად, მენეჯერის საქმიანობა პროექტის შესრულების მიზნის მიღწევაა, რომელიც განიხილება პროექტის ისეთი პარამეტრების შეთანხმებაში, როგორიცაა: **მოღვაწეობის არეალი, ვადები და ფინანსები.** ამ პარამეტრების ურთიერთკავშირის ანალიზური და გრაფიკული ინტერპრეტაციის სხვადასხვა ხერხი არსებობს. ერთ-ერთი მარტივი სქემა მე-9 ნახაზზეა მოცემული, ე.წ. „პროექტების მენეჯმენტის სამკუთხედი"-ის სახით.




**ნახ.9. პროექტების მენეჯმენტის სამკუთხედი**

პროგრამული სისტემის მენეჯერის ამოცანა შეიძლება ასე ავხსნათ: საჭიროა **მოღვაწეობის არეალის** (კადრების მწარმოებლურობა გარკვეული სამუშაოს შესასრულებლად), **დროის** (სამუშაოს შესრულების ვადების) და **ფინანსების** (რესურსების ხარჯვის) ისეთი შეთანხმება, რომელიც დააკმაყოფილებს **ხარისხის** მოთხოვნებს. ასეთი ბალანსის დამყარება, შესასრულებელი პროცესების სირთულისა და არაერთგვაროვნობის გამო, საკმაოდ ძნელია. ამიტომ მენეჯერი მიზნობრივად ირჩევს ერთ ან ორ

პარამეტრს, დანარჩენს კი მათზე დამოკიდებულად. ამგვარად, სამკუთხედი იღებს ასეთ ინტერპრეტაციას: „კარგად-სწრაფად-იაფად: აირჩიეთ მათ შორის ორი“.

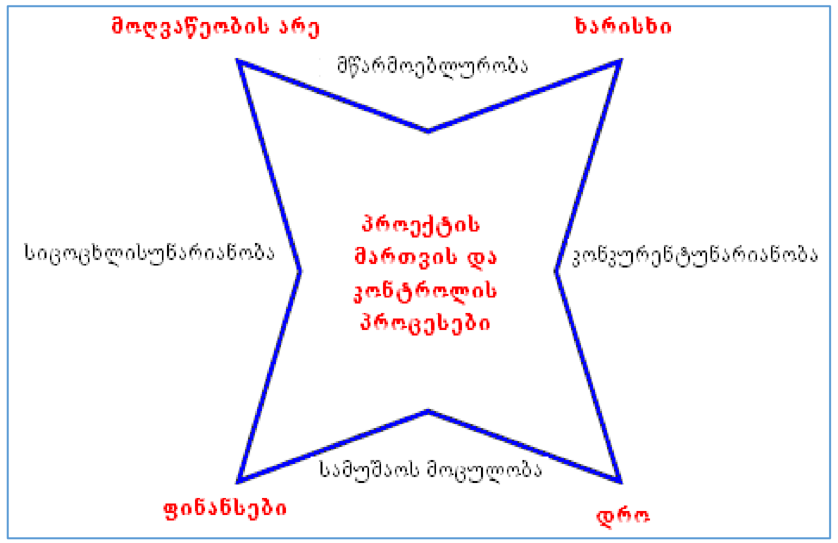
მე-9 ნახაზზე სამკუთხედის გვერდების სიგრძეების (ცვლადების) მნიშვნელობათა ვარირებით მიიღება სხვადასხვა შედეგები. ამ პარამეტრთა ურთიერთდამოკიდებულების ასახვის ინვარიანტულ მაჩვენებლად შემოაქვთ სამკუთხედის ფართობი

**(S = a * H/2),**

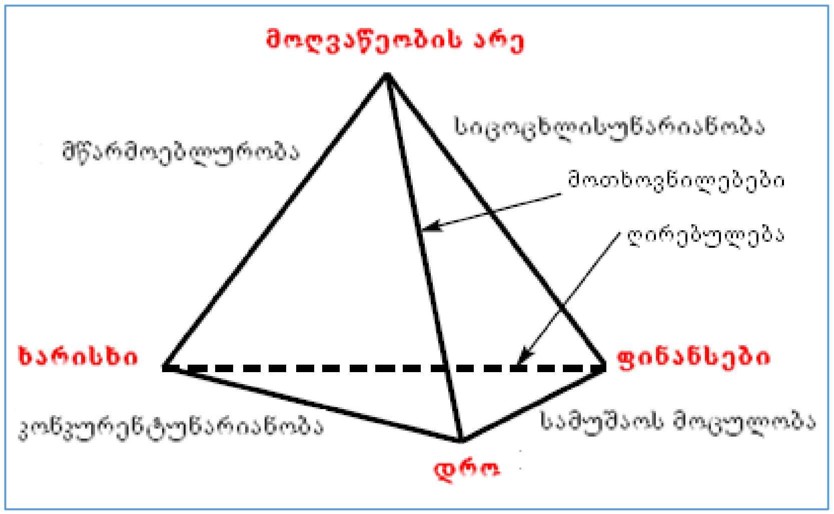
რომელიც კონკრეტული პროექტისა და კონკრეტული შემსრულე-ბელი გუნდისთვის უცვლელია (კონსტანტაა). აქედან გამომდინარე, თუ სამკუთხედში საწყისად შერჩეულ იქნება ორი გვერდი, მაშინ მესამის გამოთვლა ყოველთვის შესაძლებელია მისი ფართობის და H-სიმაღლის გამოყენებით.

პროექტების მართვის ინსტიტუტის (*PMBOK*) მეცნიერის, რ. ვიდემანის მიერ შემოტანილ იქნა მე-4 პარამეტრი - **ხარისხი** (ნახ.10). მიღებულია ოთხქიმიანი ვარსკვლავის სქემა, რომელშიც, მიუხედავად შინაარსობრივი აღწერის უკეთ შესაძლებლობისა, დაიკარგა სამკუთხედის სიმარტივის პრინციპი და მნიშვნელოვანი ინვარიანტული პარამეტრი - ფართობი.

აღნიშნული პრობლემის გადაჭრა განხორციელდა სამ-განზომილებიან სივრცეში გადასვლით, სადაც შესაძლებელი გახდა ინვარიანტული ცვლადის გამოყენება, ამჯერად ტეტრაედრის მოცულობის სახით (ნახ.11).

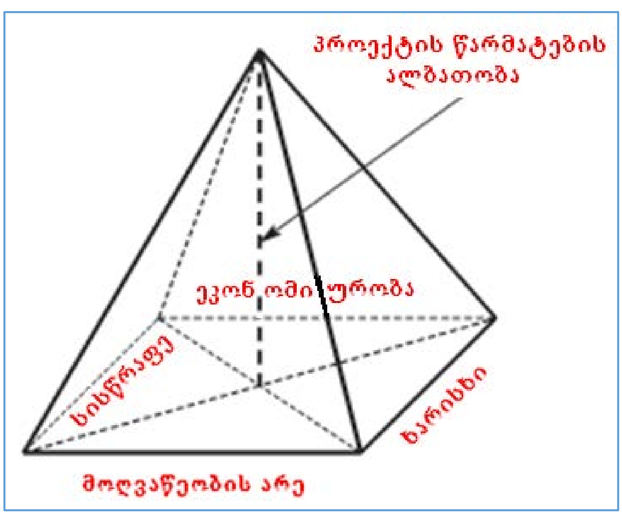



ნახ.1.10




ნახ.1.11

მესამე განზომილების შემოტანის საფუძველზე ჯ. მარასკომ მოახერხა „საპროექტო პირამიდის“ აგება (ნახ.12), რომელშიც კიდევ ერთი ახალი პარამეტრია დამატებული, როგორიცაა პროექტის წარმატების ალბათობა (Probability of Success).




ნახ.12

პირამიდის ფუძე ოთხკუთხედია, რომლის გვერდებია: სისწრაფე (Speed), ეკონომიურობა (Frugality), ხარისხი (Quality) და მოღვაწეობის არე (Scope). პირამიდის სიმაღლეს შეესაბამება პროექტის წარმატების ალბათობა. ეს ხუთი პარამეტრი განიხილება როგორც ცვლადები, რომელთა მნიშვნელობები დაკავშირებულია პროექტის განვითარების პირობების ინვარიანტთან მისი შემსრულებელი გუნდის ძალებით. ინვარიანტად მიიღება პირამიდის მოცულობა.

აღნიშნული მოდელით უფროო რეალურადაა შესაძლებელი პროექტის მენეჯერის საქმიანობაზე დაკვირვება პროექტის შესრულების და კორექტირების პროცესებში. ვინაიდან პროექტის

წარმატების ალბათობა დროში ცვალებადია, ხოლო პირამიდის მოცულობა მოცემული პროექტისთვის მუდმივია, მაშინ პირამიდის სიმაღლის ცვლილება აუცილებლად გამოიწვევს შედეგად პირამიდის ფუძის ცვლადების შეცვლას. ასევე მართებულია უკუგამონათქვამი, რომ პროექტის წარმატების ალბათობა დამოკიდებულია პირამიდის ფუძის ფართობზე.

ამგვარად, პროექტების მართვის ე.წ. „მენეჯმენტის სამკუთხედი“, „მენეჯმენტის პირამიდა“ და ა.შ. წარმოადგენს საილუსტრაციო მოდელებს, რომელთა საშუალებითაც შესაძლე-ბელია პროგრამული სისტემების პროექტირების პროცესების უფრო ღრმად გამოკვლევა.

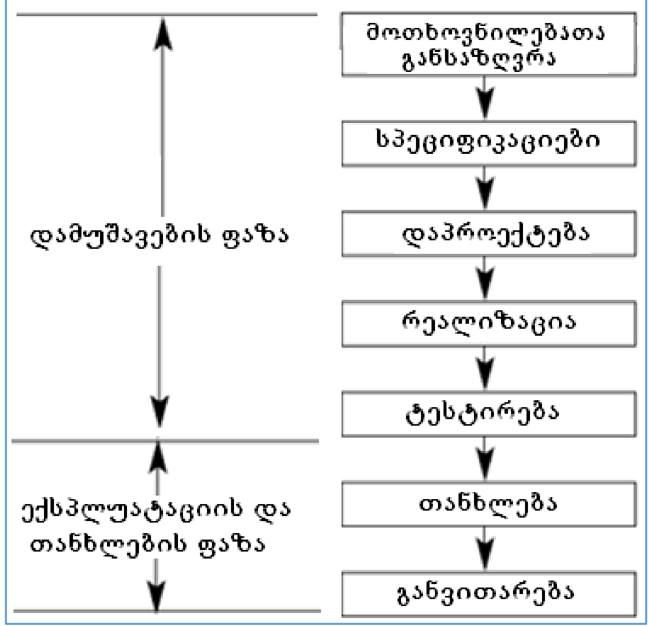
მენეჯერის მეთოდური მუშაობის ზოგადი სქემა მდგომარეობს შემდეგში: თავიდან განისაზღვრება საპროექტო საქმიანობა, შემდეგ ირკვევა, თუ რა კორექტირებებია შესასრუ-ლებელი და რა საშუალებები არსებობს ამისათვის. მენეჯერის საქმიანობის შედეგები ძალზე მრავალფეროვანია, მაგრამ ისინი თავსდება სათანადო ჩარჩოებში ბალანსის დამყარების თვალსა-ზრისით პროექტის მიზნის მისაღწევად.

## 1.6. პროგრამული უზრუნველყოფის სასიცოცხლო ციკლის მოდელები

### 1.6.1. პროგრამული უზრუნველყოფის სასიცოცხლო ციკლის საბაზო მოდელი

- **პროგრამული უზრუნველყოფის სასიცოცხლო ციკლი** ესაა პროგრამული უზრუნველყოფის კონსტრუირების პროცესი. პროგრამულ პროექტზე მოთხოვნილებანი განსაზღვრავს ამ პროცესის მიზანს და რეგლამენტს.

მე-13 ნახაზზე ნაჩვენებია პროგრამული სისტემის სასიცოც-ხლო ციკლის ტრადიციულად მიღებული მოდელი. იგი 2 ფაზისა და 7 ეტაპისაგან შედგება.




ნახ.13. სასიცოცხლო ციკლის საბაზო მოდელი

თითოეული ეტაპი პროექტის მენეჯერისა და შემსრულე-ბელთა სათანადო როლით ხასიათდება. თუ ჩვენ განვიხილავთ IBM-კომპანიის ცენტრის ობიექტ-ორიენტირებულ ტექნოლოგიას, მაშინ შეიძლება გამოვყოთ საკმაოდ სრული სია ტიპური როლებისა, რომლებიც პროგრამული პროექტების შესრულებასა და განვითარებას ემსახურება.

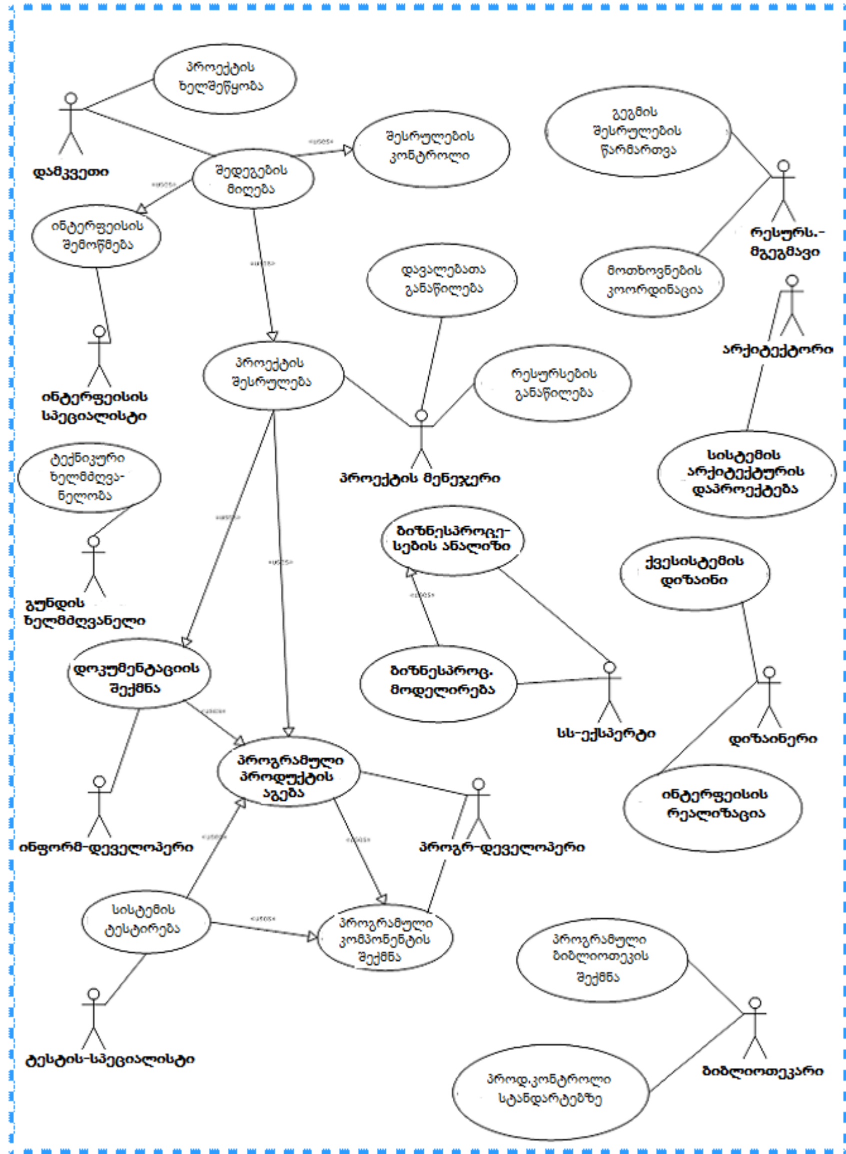
ესაა პროექტში მონაწილეთა როლები პროგრამული უზრუნველყოფის ფირმებიდან და დამკვეთი კორპორაციებიდან, რომლებიც გავლენას ახდენენ როგორც პროექტის ამოცანების ჩამოყალიბებაზე, ასევე შესაბამისი რესურსების გამოყოფასა და საერთოდ პროექტის განხორციელების სამუშაოთა განვითარებაზე.

როლების დახასიათება მოიცავს მათი შესასრულებელი ორგანიზაციული და საწარმოო ფუნქციების ერთობლიობას.

- დამკვეთი (Customer) – პროექტის რეალურად არსებული ინიციატორი, რომელსაც ემორჩილება საპროექტო გუნდი, ან ორგანიზაციის სხვა წარმომადგენელი, რომელსაც დავალებული აქვს პროექტის ხელშეწყობა, შესრულების კონტროლი და შედეგების მიღება.
- რესურსების მგეგმავი (Planner) – წარმოადგენს და აკოორდინირებს პროექტისადმი მოთხოვნებს ორგანიზაციაში, სადაც მუშავდება პროექტი. აგრეთვე ორგანიზაციის თვალსაზ-რისით ავითარებს და წარმართავს პროექტის შესრულების გეგმას,.
- პროექტის მენეჯერი (Project Manager) – პასუხისმგებელია მთლიანად პროექტის შესრულებაზე, დავალებებისა და რესურსების განაწილებაზე, სამუშაოთა შედეგების მიღებაზე გრაფიკის მიხედვით, შედეგების შესაბამისობაზე მოთხოვნებთან. იგი აქტიურად თანამშრომლობს დამკვეთთან და რესურსების დამგეგმავთან.
- გუნდის ხელმძღვანელი (Team Leader) – ახორციელებს გუნდის ტექნიკურ ხელმძღვანელობას პროექტის შესრულების პროცესში. დიდ პროექტებში შეიძლება რამდენიმე ხელმძღვანელის არსებობა ქვეგუნდების მიხედვით, რომლებიც ცალკეულ ამოცანებზე იქნება პასუხისმგებელი.
- არქიტექტორი (Architect) – პასუხისმგებელია სისტემის არქიტექტურის დაპროექტებაზე, ათანხმებს პროექტთან დაკავშირებულ სამუშაოთა განვითარებას.
- ქვესისტემის დამპროექტებელი (Designer) – პასუხს აგებს ქვესისტემის ან კლასთა კატეგორიების დაპროექტებაზე, განსაზღვრავს რეალიზაციის და ინტერფეისების საკითხებს სხვა ქვესისტემებთან.

- საგნობრივი სფეროს ექსპერტი (Domain Expert) – პასუხისმგებელია საკვლევი სფეროს დანართის ბიზნესპროცესების შესწავლასა და მოდელირებაზე, ახორციელებს პროექტის მხარდაჭერას ამ სფეროს ამოცანების გადასაწყვეტად.
- დამმუშავებელი (Developer) – ახორციელებს საპროექტო კომპონენტების რეალიზაციას, ფლობს და ქმნის სპეციფიკურ კლასებსა და მეთოდებს, აგებს კოდებს და ატესტირებს მათ, ქმნის პროგრამულ პროდუქტს.
- ინფორმაციული მხარდაჭერის დამმუშავებელი (Information Developer) – ქმნის პროგრამული პაკეტის თანმხლებ დოკუმენ-ტაციას. მას თან ურთავს სისტემის საინსტალიაციო მასალებს, მომხმარებლის ინსტრუქციებს.
- სამომხმარებლო ინტერფეისის სპეციალისტი (Human Factors Engineer) – პასუხისმგებელია სისტემის ადვილად და მოხერხე-ბულად გამოყენებაზე. მუშაობს დამკვეთთან, რათა დარწმუნდეს, რომ მომხმარებლის ინტერფეისი აკმაყოფილებს მოთხოვნებს.
- ტესტირების სპეციალისტი (Tester) – ამოწმებს პროდუქტის ფუნქციონალობას, ხარისხსა და ეფექტურობას. აგებს და იყენებს საკონტროლო ტესტებს პროექტის ყოველი ფაზისათვის.
- ბიბლიოთეკარი (Librarian) – პასუხს აგებს პროექტის საერთო ბიბლიოთეკის შექმნასა და მოვლაზე, რომელიც მოიცავს ყველა სამუშაო პროექტის პროდუქტს, აგრეთვე ამ პროდუქტების შესაბამისობას სტანდარტებთან.

UML მოდელირების ენის დახმარებით აგებული გვაქვს შესაბამისი საპრობლემო სფეროს გამოყენებითი შემთხვევის Use Case დიაგრამა (ნახ.14):




ნახ.16. UseCase-დიაგრამა

## 1.6.2. კლასიკური იტერაციული მოდელი

როგორც აღვნიშნეთ, **პროგრამული უზრუნველყოფის სასიცოცხლო ციკლი** - პროგრამული უზრუნველყოფის კონსტრუირების პროცესია, რომლის მიზანს და რეგლამენტს განსაზღვრავს პროგრამული სისტემის ფუნქციური და არაფუნქციური მოთხოვნილებანი.

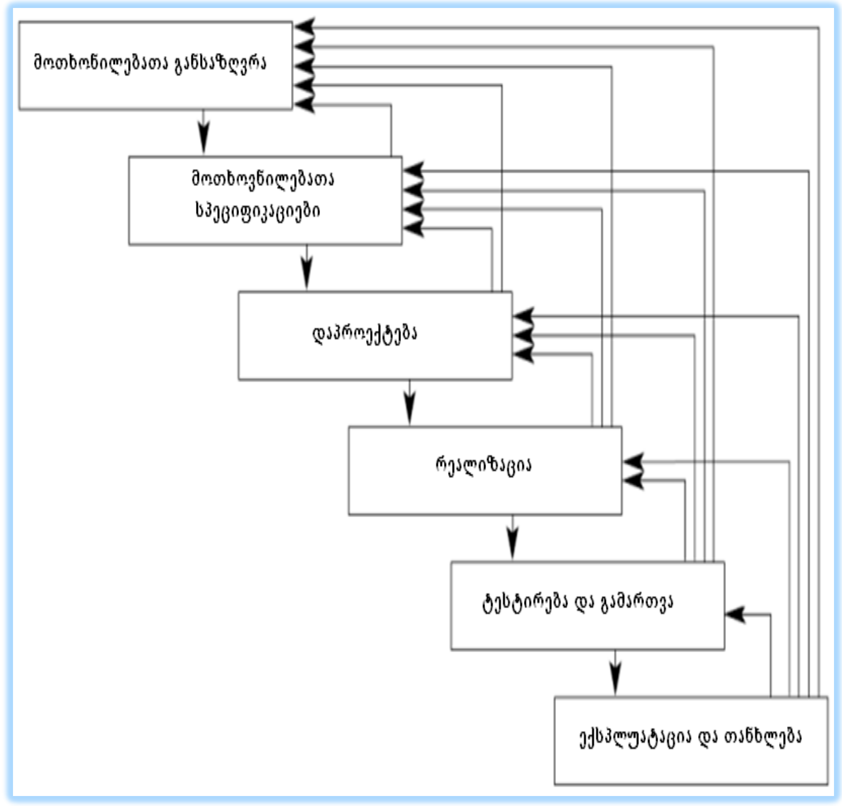
ჩვენ განვიხილეთ პროგრამული სისტემის სასიცოცხლო ციკლის ტრადიციულად მიღებული 7-ეტაპიანი მოდელი (ნახ.13). დაისმის კითხვა - გამოდგება კი ასეთი მოდელი ზოგადად ყველანაირი პროგრამული სისტემის აგების პროცესის მენეჯმენტისათვის: დაგეგმვის, აღრიცხვისა და კონტროლის ორგანიზებისათვის ?

აღნიშნული მოდელის შეზღუდულობის გამო (მაგალითად, აქ თითოეულ ეტაპზე არ ჩანს ცხადად პროექტის შემსრულებელთა-როლების ფუნქციები) იგი არ შეიძლება ჩაითვალოს სრულყოფი-ლად, თუმცა როგორც საწყისი (პირველადი) მოდელი მისაღებია. მისი გამოყენება უპრობლემოდ შეიძლება მარტივი პროექტების შესრულებისას, სადაც არაა საჭირო იტერაციული პროცესები.

იტერაცია არის პროგრამული სისტემის შესრულების პროცესში წინა ბიჯებზე (ეტაპებზე) დაბრუნება, გარკვეული დასაკორექტირებელი პროცედურების ჩასატარებლად.

რთული პროექტები აუცილებლად მოითხოვს მრავალ-ჯერადი იტერაციული პროცესების ჩატარებას პროგრამული სისტემის სასიცოცხლო ციკლის ყველა ეტაპზე, როგორც წინა ეტაპების შეცდომების გასწორების, განუსაზღვრელობის გათვა-ლისწინების მიზნით, ისე სისეტემის ექსპლუატაციის პირობების მოთხოვნილებების ცვლილებების გამო.

კლასიკური იტერაციული მოდელის სასცოცხლო ციკლის ზოგადი სქემა მოცემულია მე-15 ნახაზზე.




**ნახ.17.კლასიკური იტერაციული მოდელი**

ისრებით (ზემოთ) მითითებულია იტერაციული პროცესები წინა ეტაპებზე, რაც გამოწვეულია სათანადო შეცდომებისა და გაურკვევლობების გასწორების მიზნით. ტრადიციული მეთოდები ცხდილობს ასეთი პროცესების მინიმიზაციას ანუ მკაცრად მოითხოვს წინასწარ იყოს ყველაფერი ზუსტად განსაზღვრული და გამორიცხავს უკუიტერაციებს, რაც მათ ნაკლად შეიძლება ჩაითვალოს.

პროგრამული პროექტების იტერაციული განვითარების მეთოდები აქტუალურია, რადგან ისინი უარს ამბობს ეტაპების სრულფასოვნებაზე და არ გამორიცხავს მათ იტერაციულ განვითარებას, ფუნქციობის და ინტერფეისული შესაძლებლობების გაფართოების თვალსაზრისით.

ამგვარად, პროგრამული სისტემის სასიცოცხლო ციკლის კლასიკური მოდელი მისაღებია, ოღონდ ერთი იტერაციის ფარგლებში და მნიშვნელოვანი შესწორებით: რომ ყველაფერი, რაც სასარგებლო იყო ადრე, შეინახება.

ძველი კოდის შენახვის მხარდაჭერის იდეა, მისი ეფექტიანობის დაკარგვის გარეშე, დაკავშირებულია მთლიანად ობიექტ-ორიენტირებულ დაპროგრამებასა და პროექტირებასთან, CASE-ინსტრუმენტების გამოყენებასთან (ამ საკითხს მომავალში დავუბრუნდებით).

იტერაციული გაფართოების მიზანია სისტემის მოქნილობის ამაღლება, მისი ადაპტაციის შესაძლებლობის უზრუნველყოფა პროგრამული სისტემისა და პროექტის პირობების ცვლილებების დროს. იტერაციული ბიჯების შემოტანა არის სწორედ პროექტის ადაპტურობის ამაღლების საშუალება, რადგანაც ასეთი მიდგომის გამოყენებისას პროექტი უფრო მომარჯვებულია ცვლილებე-ბისადმი.

ასეთი კონცეფცია ეწინააღმდეგება პროგრამული სისტემების აგების ტრადიციულ მეთოდოლოგიებს, თუ პროექტების დამუშავების პროცესში და სისტემის არქიტექტურაში არ იქნება გათვალისწინებული ადაპტაციის მექანიზმები.

ობიექტ-ორიენტირებულ მიდგომაში ეს მექანიზმები უფრო განვითარებულია, ამიტომ ის შეიძლება მიღებულ იქნას პროგრამული პროექტების განვითარების მთლიანი იდეოლოგიის საფუძვლად.

პროგრამული სისტემების ახალი მეთოდოლოგიები მთლია-ნობაში კი რჩება ობიექტორიენტირებული, მაგრამ ცდილობენ ამ მიდგომის მკაცრი მოთხოვნებიდან გათავისუფლებას. მაგალითად, ექსტრემალური პროგრამირების მეთოდი დასაშვებად მიიჩნევს სისტემის პირველადი დეკომპოზიციის შედეგების (სისტემის დიზაინის) ცვლილებას, პროექტის შესრულების პროცესში მომხმარებელთა მოთხოვნილებების დაზუსტების შედეგად.

ასეთი თვალსაზრისი მოითხოვს პროგრამული პროექტების მენეჯმენტის ახალი, სპეციალური ვარიანტის შექმნას.

### 1.6.3. კასკადური მოდელი

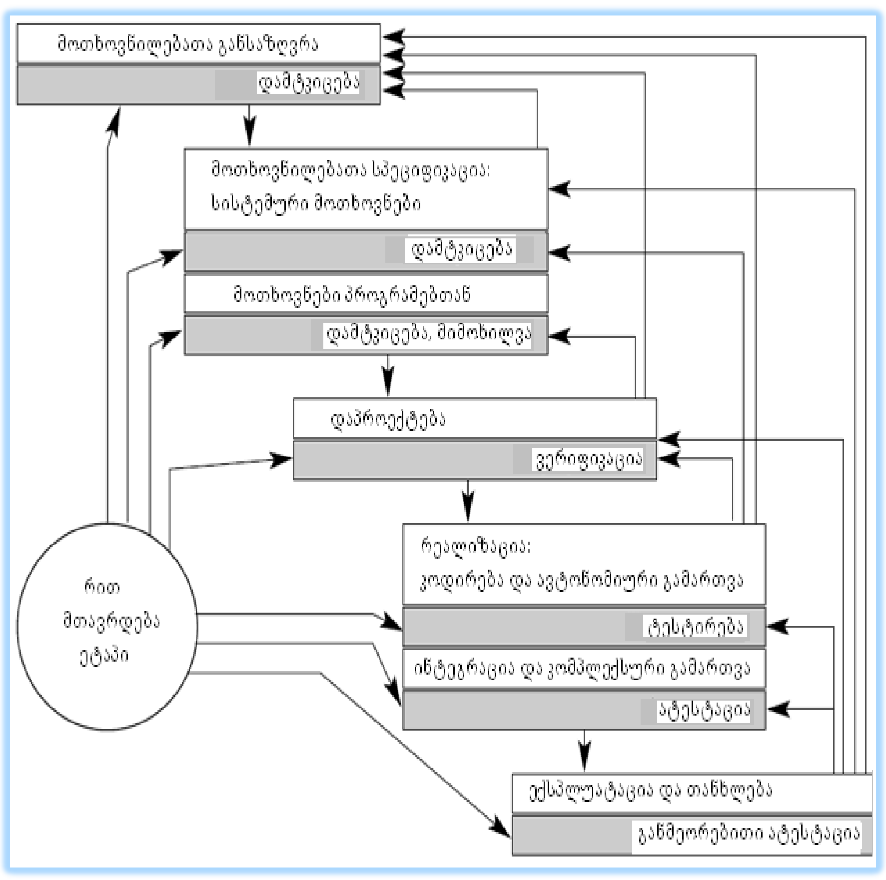
კლასიკური იტერაციული მოდელის შედარებით მკაცრი ნაირსახეობაა **კასკადური მოდელი,** რომელიც საილუსტრაციო მაგალითია იმის, თუ როგორ შეიძლება უკანდაბრუნების (იტერაციების) მინიმიზაცია.

კასკადური მოდელისათვის დამახასიათებელია:

- ყოველი ეტაპის დასრულება (როგორც კლასიკურ მოდელში) მიღებული შედეგების შემოწმებით, რათა აღმოფხვრილ იქნას პროგრამის დამუშავების რაც შეიძლება მეტი პრობლემა;
- გავლილი ეტაპების ციკლური გამეორება (როგორც კლასიკურ მოდელში).

კასკადური მოდელის მოტივაცია დაკავშირებულია პროგრამული უზრუნველყოფის ხარისხის მართვასთან. ამასთან დაკავშირებით ზუსტდება ეტაპების არსები, ზოგი მათგანი

სტრუქტურირდება (მოთხოვნილებათა სპეციფიკაცია და რეალიზაცია). მე-16 ნახაზზე მოცემულია კასკადური მოდელის სქემა, რომელიც აგებულია როგორც კლასიკური იტერაციული მოდელის მოდიფიკაცია.




ნახ.16. კასკადური მოდელი

თითოეულ ბლოკში (ეტაპში) მითითებულია ქმედება, რომლითაც თავდება ეტაპი. აქ ტესტირება არაა გამოყოფილი ცალკე ეტაპად, იგი ითვლება ზღურბლად, რომელიც გადალახულ უნდა იქნას, რათა დასრულდეს ეტაპი, ისევე როგორც სხვა მსგავსი ქმედებები (მაგალითად, მიმოხილვები - დოკუმენტებია, რომლებშიც აღიწერება სისტემური მოთხოვნები და ისინი უნდა შეთანხმდეს და დამტკიცდეს დამკვეთის მიერ).

დაპროექტების შედეგი ვერიფიცირდება (მოწმდება), უზრუნველყოფს თუ არა სისტემის მიღებული სტრუქტურა და სარეალიზაციო მექანიზმები სპეციფიკური ფუნქციების შესრულებას. რეალიზაცია კონტროლდება კომპონენტების ტესტირების გზით, ხოლო კომპონენტების ინტეგრაციის შემდეგ სისტემაში ხორციელდება ატესტაცია კომპლექსური გამართვისთვის. ანუ ხდება სისტემის რეალიზებული ფუნქციების შემოწმება-ფიქსაცია, რეალიზაციის შეზღუდვების აღწერა და ა.შ.

კასკადურ მოდელში ვერიფიკაცია და ატესტაცია მიწერილია სხვადასხვა ეტაპთან:

- ვერიფიკაცია უპასუხებს კითხვას: - სწორად არის თუ არა აგებული პროგრამული სისტემა (ანუ იგი ამოწმებს სპეციფიკა-ციასთან შესაბამისობას და მას ატარებენ დამპროექტებლები და პროგრამისტები);
- ატესტაცია კი - თუ მუშაობს სწორად პროგრამული სისტემა (ის განიხილება სისტემის ექსპლუატაციის დროს და ვარგისიანობაზე დასკვნას ამზადებენ დამკვეთი სპეციალისტები-მომხმარებლები ).

ექსპლუატაციის და თანხლების პროცესში დგინდება, თუ რამდენად კარგად შეესაბამება სისტემა მომხმარებელთა მოთხოვნებს ანუ ხდება განმეორებითი ატესტაცია.

ყოველი შემოწმების შემდეგ შესაძლებელია დამპროექტებ-ლების დაბრუნება ნებისმიერ გავლილ ეტაპზე, რაც უკუ-ისრებითაა ნაჩვენები. უკანდაბრუნების ბიჯების მინიმიზაციის მიზნით კასკადურ მოდელში დამპროექტებლები იყენებენ გამკაცრებულ შემოწმებას (ეს მკაცრი კასკადური მოდელების სახელწოდებითაა ლიტერატურაში ცნობილი. აქ ჩვენ მას დეტალურად არ განვიხილავთ). აღვნიშნოთ მისი სასიცოცხლო ციკლის დამახასიათებელი მომენტები:

- სამუშაოების, დავალებების და პასუხისმგებლობათა ზუსტი განაწილება ეტაპების დამმუშავებელთა და მათ შემმოწმებელთა შორის, რომლებიც მომდევნო ეტაპზე გადასვლის თანხმობას იძლევა;
- მცირე ციკლების არსებობა მეზობელ ეტაპებს შორის, რომელთა შედეგად მიიღწევა კომპრომისული დავალება.

პროგრამული სისტემების სასიცოცხლო ციკლების განხილული მეთოდების გამოყენება არაა უნივერსალური და დამოკიდებულია კონკრეტულ ობიექტსა და ბევრ სხვა ფაქტორზეც.

### 1.6.4. განტერის მოდელი: „ფაზები და ფუნქციები“

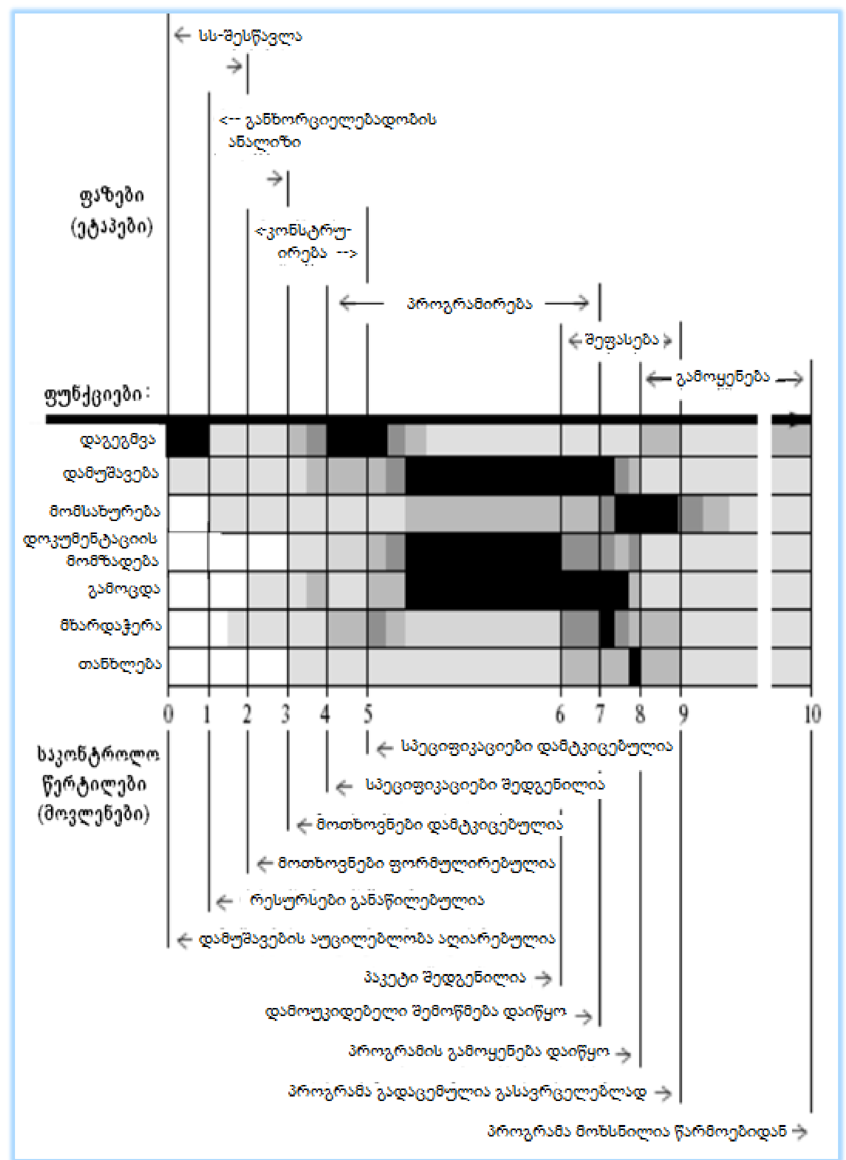
პროგრამული პროდუქტების სასიცოცხლო ციკლის მოდელებში აისახება საწარმოო ფუნქციები, რომლებსაც ასრულებენ დამპროექტებლები. ეს ფუნქციები უნდა იყოს კავშირში პროექტების მართვის საკონტროლო ელემენტებთან ანუ სასიცოცხლო ციკლის ეტაპებთან. ისინი სრულდება პროექტის განვითარების მთელი პერიოდის მანძილზე სხვადასხვა ინტენსივობით. განტერის მოდელი „ფაზები და ფუნქციები“ საფუძველია განვითარებული სასიცოცხლო ციკლის ასაგებად, სადაც ასახულია ორგანიზაციული და ტექნიკური საწარმოო ფუნქციები. ამასთანავე განტერის მოდელი ითვალისწინებს იტერაციებსაც.

მოდელი უნდა იყოს პროექტის დამმუშავებლებს შორის ურთიერთდამოკიდებულების ორგანიზაციის საფუძველი. ამგვარად, მის ერთ-ერთ მიზანი მენეჯერის ფუნქციების მხარდაჭერაა. ეს კი აუცილებლად მოითხოვს პროექტზე საკონტროლო წერტილების დალაგებას, რომელიც ასახავს პროექტის ორგანიზაციულ-დროით ჩარჩოებს, ორგანიზაციულ-ტექნიკურ საწარმოო ფუნქციებს, რომლებიც სრულდება პროექტის განვითარების დროს.

განტერის მოდელს აქვს ორი განზომილება:

- ფაზური, რომელიც ასახავს შესრულების ეტაპებს და თანმხლებ მოვლენებს;
- ფუნქციური, სადაც ჩანს, თუ რომელი საწარმოო ფუნქციები სრულდება პროექტის განვითარების პროცესში და როგორია თითოეულ ეტაპზე მათი ინტენსივობა.

მე-17 ნახაზზე მოცემულია განტერის მოდელის სქემა. აბსცისთა ღერძი პროექტის განვითარებას ასახავს, იგი დროის ღერძია. მასზე დასმულია საკონტროლო წერტილები და მოვლენათა დასახელებები.




ნახ.17. განტერის მოდელი „ფაზები-ფუნქციები“

მოდელში სასიცოცხლო ციკლი მოიცავს შემდეგ ფაზებს (ეტაპებს):

- გამოკვლევის ეტაპი;
- განხორციელებადობის ანალიზი;
- კონსტრუირება;
- დაპროგრამება;
- შეფასება;
- გამოყენება.

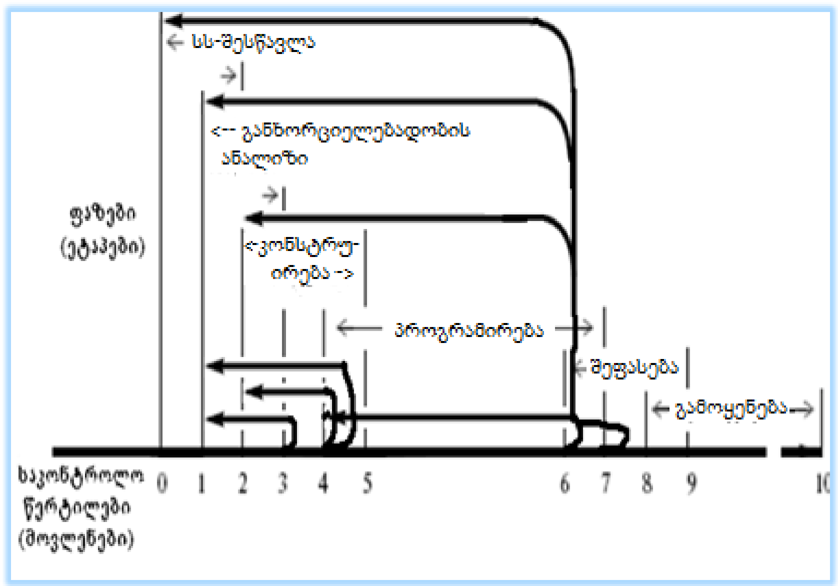
სასიცოცხლო ციკლის მოდელის საწარმოო-ორგანიზაციული ფუნქციები შემდეგია:

- დაგეგმვა;
- დამუშავება;
- მომსახურება;
- დოკუმენტაციის მომზადება;
- გამოცდა (შეფასება);
- მხარდაჭერა;
- თანხლება.

ფუნქციების გამოყენების ინტენსივობა ფაზების მიხედვით მოცემულია მუქი ფერით.

განტერის მოდელი არ ასახავს იტერაციულობას, დაბრუნებას წინა ეტაპებზე, რაც ერთგვარად არასახარბიელოდ ითვლება, თუმცა ტრადიციულ და კასკადურ მოდელებში ეტაპების გარკვეული გადაფარვა ხორციელდება.

იმისათვის, რომ იტერაციულობა იყოს გათვალისწინებული, საჭიროა მცირე მოდიფიკაცია, რაც მე-18 ნახაზზეა ნაჩვები. აქ სასიცოცხლო ციკლის საკონტროლო წერტილებში (მაგალითად, 3,4,6) ნაჩვენებია იტერაციული ციკლების შესაძლებლობები.




ნახ.18. იტერაციულობა განტერის მოდელში
„ფაზები-ფუნქციები“

განიხილება პროექტების შესრულების ორი შემთხვევა:

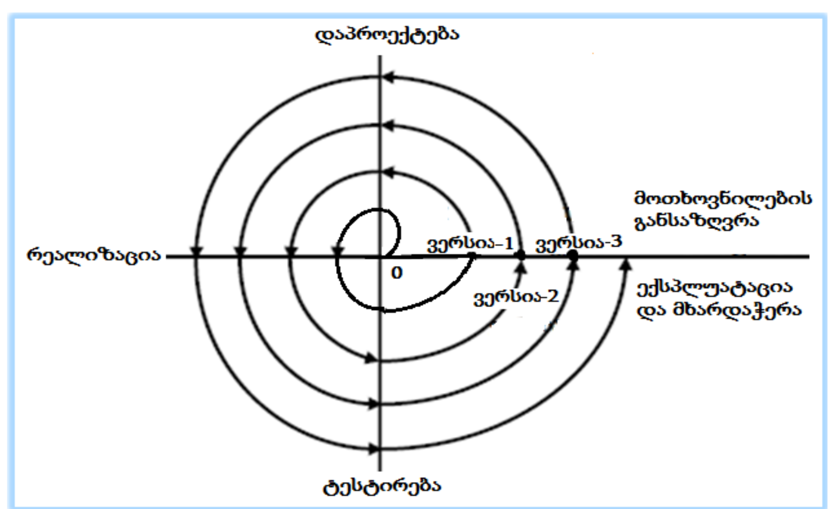
- რომელიმე საკონტროლო წერტილში მოხდება ძირითადი პროცესის შეჩერება, დაბრუნება რომელიმე წინა ეტაპზე სათანადო დროის განმავლობაში. შემდეგ ისევ გადმოსვლა ძირითად პროცესზე და შედეგების შერწყმა ცვლილებების გათვალისწინებით;
- ძირითადი პროცესი არ წყდება იტერაციული ციკლის შესრულების მინით. იგი გრძელდება ჩვეულებრივად, ხოლო იტერაცია განიხილება როგორც დამატებით პროცესი, რომელიც უნდა მიუერთდეს სასიცოცხლო ციკლს. ასეთი დამატებითი პროცესი ორგანიზებულად უნდა წარიმართოს, ანუ შესაძლებელია მისი წინასწარი დაგეგმვა.

იტერაციულობა გარდაუვალია რთული პროგრამული სისტემების დასაპროექტებლად. ამიტომაც მიზანშეწონილია ასეთი პროცესების დაგეგმვა. ისინი უნდა იქნას აღქმული არა როგორც „შეცდომების“ გასწორების იტერაციული პროცედურები, არამედ როგორც სისტემის გაფართოების აუცილებელი იტერაციები.

ეს საკითხი განსაკუთრებით საყურადღებოა ობიექტ-ორიენტირებული დაპროგრამების პრინციპების გამოყენებისას.

### 1.6.5. სპირალური მოდელი

პროგრამული სისტემების დამუშავების სასიცოცხლო ციკლის სპირალური მოდელის ძირითადი არსი მდგომარეობს მისი ეტაპების (ანალიზი, დაპროექტება, რეალიზაცია, ტესტირება და დანერგვა-ექსპლუატაცია) ციკლურ ევოლუციურ განვითარებაში (ნახ.19).




ნახ.19. სპირალური მოდელი

პროგრამული სისტემის ვერსიები (1,2,3,..) იქმნება პროტოტიპების სახით, იგი ვითარდება სპირალურად და ყოველი ახალი ვერსია სულ უფრო ახლოა დამკვეთი-მომხმარებლების მოთხოვნილებებთან.

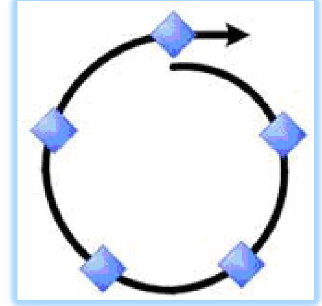
ტესტირების და საექსპლუატაციო დანერგვის პროცესში მონაწილეობენ სისტემური ანალიტიკოსები და დამკვეთები, რაც საშუალებას იძლევა ოპერატიულად განისაზღვროს საჭირო ცვლილებები შემდეგი ვერსიისათვის.

### 1.6.6. MSF მოდელი

პროგრამული სისტემების აგების Microsoft Solution Framework (MSF) მეთოდოლოგია (ნახ.20) აერთიანებს კასკადური და სპირალური მოდელების მოწესრიგებადობისა და მოქნილობის თვისებებს [6].

საბაზო პრინციპები შემდეგია:

1. პროექტის ერთიანი ხედვა. თავიდან დამკვეთსაც და საპროექტო გუნდსაც ექნებათ თავიანთი საკუთარი ხედვა პროექტის მიზნებსა და ამოცა-ნებზე. აგრეთვე თუ რა უნდა იქნას მიღწეული პროექტზე მუშაობისას;



ნახ.20

პროექტის წარმატება შეუძლებელია დამკვეთისა და საპროექტო გუნდის ერთიანი ხედვის გარეშე, რადგანაც გაურკვევ-ლობის გამო მიზანი მიუღწეველი რჩება.

MSF მეთოდოლოგია ამისათვის ითვალისწინებს ცალკე ეტაპის (ფაზის) გამოყოფას - „კონცეფციის შემუშავება“.

2. მოქნილობის გამოვლენა - მზადყოფნა ცვლილებებისადმი. კასკადური მოდელისაგან განსხვავებით MSF მიდგომა ეფუძნება პროექტის პირობების უწყვეტი ცვლილების პრინციპს.

3. კონცენტრირება ბიზნესპრიორიტეტებზე. დასამუშავებელმა პროდუქტმა უნდა მოიტანოს განსაზღვრული სარგებელი ან უკუგება საბოლოო მომხმარებლებისათვის. ვინაიდან პროგრამულ პროდუქტს შეუძლია თავისი სარგებლობის მოტანა მხოლოდ დანერგვის შემდეგ ორგანიზაციაში, ამიტომ MSF მიდგომა სასიცოცხლო ციკლში ითვალისწინებს დანერგვის ფაზას.

4. თავისუფალი ურთიერთობის წახალისება. MSF-ის პროცესების მოდელი გვთავაზობს ინფორმაციის ღია და თავისუფალ გაცვლას როგორც პროექტის შემსრულებლებს შორის, ისე პროექტით დაინტერესებულ პირებს (stakeholders) შორის. ამან ხელი უნდა შეუწყოს მათ შორის გაუგებრობის, გაურკვევლობის და უსაფუძვლო ხარჯების შემცირებას. ამიტომაც MSF ითვალისწინებს მუშაობის პროცესის სისტემატურ ანალიზს დადგენილი დროის პუნქტებში, რომლებშიც მონაწილეობენ დამკვეთებიც და შემსრულებლებიც.

## 1.7. ობიექტორიენტირებული დაპროგრამების პრინციპები

პროგრამული პროექტების შერულების პრინციპული მომენტები, რომლითაც ობიექტორიენტირებული მიდგომა განსხვავდება ტრადიციული მიმდევრობითი მეთოდოლოგიებისაგან, შემდეგია:

**1. განვითარების იტერაციულობა:**

დაწყებული ანალიზის ფაზიდან, დამთავრებული რეალიზაციით, ობიექტორიენტირებული პროექტირების პროცესი, მიმდევრობითი განვითარების მიდგომისაგან განსხვავებით, აიგება როგორც იტერაციათა სერია, რომელსაც შესაძლოა წინ უძღოდეს

საგნობრივი სფეროსა და საპროექტო ამოცანათა მიმდევრობითი შესწავლის პერიოდი (მოთხოვნილებათა განსაზღვრისა და საწყისი დაგეგმვის ეტაპები).

**2. ფუნქციურობის ცვლილებები:**

პროექტის შესრულების პროცესში მისი მოთხოვნები პრაქტიკულად ყოველთვის გადაისინჯება. ამასთან მოითხოვება, რომ შეცვლილ იყოს ადრე მიღებული მოთხოვნები. ამგვარად, ფუნქციურობა ისე უნდა იყოს რეალიზებული, რომ მისი გადასინჯვა მოხდეს მინიმალური დანახარჯით.

**3. პროექტის ცნებათა სისტემის ფორმირება:**

პროექტის ცნებათა სისტემა ვითარდება მოთხოვნილებათა გადასინჯვის პროცესში და ფუნქციობის ცვლილებისას, აგრეთვე ცვლილებათა გავრცელებისას. სისტემის მთლიანობის უზრუნველსაყოფად საჭიროა შესაბამისი ღონისძიებები, სპეციალური ინსტრუმენტების გამოყენება.

ეს ამოცანა წყდება პროექტის ლექსიკონის (გლოსარიის) დახმარებით. ესაა სპეციალური ცოდნის ბაზა ცნებების, მათი კავშირებისა და ცვლილებათა ისტორიისა პროექტის იტერაციული განვითარების პროცესში. თუ ლექსიკონი ცხადად არაა გათვალისწინებული, მაშინ პროექტის ცნებათა ლექსიკონი სტიქიურად იქმნება, რაც უარყოფითად აისახება პროექტის განვითარებაზე.

კერძოდ, იზრდება კონცეფციის ცვლილების რისკი პროგრამული კოდის ცვლილებების შედეგად, რასაც მივყავართ კონცეპტუალურად შეუთავსებელი პროგრამული პროდუქტის აგებამდე.

**4. ფუნქციურობის გაფართოება სცენარების შესაბამისად**:

გამოყენებით შემთხვევათა (Use Case) დიაგრამის აგება იმის დასადგენად, თუ როგორ კავშირშია მომხმარებელი სისტემასთან,

არის პირველი დონე იმ მოდელებისა, რომელთაც ობიექტორიენ-ტირებული დაპროექტება გვთავაზობს.

ეს მოდელები აფიქსირებს სიტუაციებს, რომლებიც მომხამარებლისა და სისტემის ურთიერთქმედებისას აღმოცენდება. სისტემის შინაარსობრივი ქმედებები მომხმარებლებთან აღიწერება სცენარების სახით. გამოყენებითი შემთხვევები და სცენარები საფუძველია სისტემის მოთხოვნილებათა იმ სახით წარმო-სადგენად, რომელსაც გამოიყენებს დამპროექტებელი სისტემის არქიტექტურის ასაგებად. ფუნქციების გაფართოება ნიშნავს მისი სცენარების შემდგომ განვითარებას დაპროექტების ეტაპზე. სრული ფუნქციობა შედგება ყველა სცენარის ფუნქციობისგან. ამასთანავე, იტერაციული გაფართოება მოითხოვს, რომ იტერაციის ყოველ ბიჯზე პროგრამულ სისტემას ჰქონდეს სრულიად მზა ფუნქციონალობა, ხოლო მომდევნო ბიჯებზე ემატებოდეს მას სხვა, ახალი ფუნქციონალობა.

**5. არაფერი არ კეთდება ერთჯერადად**

პროგრამირების კლასიკური მიმდევრობითი მიდგომა ითვალისწინებს ანალიზის, შემდეგ კონსტრუირების და, ბოლოს პროგრამირების ეტაპებს. განტერის მოდელში ეს ეტაპები იკვეთება, მაგრამ პრინციპულად სიტუაციას ვერ ცვლის.

ობიექტ-ორიენტირებულ პროექტებში ანალიზი არასდროს თავდება სისტემის განვითარების მთელი პერიოდის მანძილზე, ხოლო კონსტრუირების პროცესი თანმხლებია პროგრამული პაკეტის მთელი სასიცოცხლო ციკლისა.

**6. ოპერირება გამრავლების ფაზებზე მსგავსია**

როგორც პროექტირების დასაწყისში, ისე მომდევნო იტერაციებზე ანალიზი წინ უსწრებს კონსტრუირებას, რომელსაც მოსდევს დაპროგრამება, ტესტირება და პროგრამული უზრუნველყოფის დამუშავებისა და გამოყენების ტრადიციული სასიცოცხლო ციკლის სხვა სახის სამუშაოები.

ობიექტორიენტირებული დაპროექტების დროს იტერაციული გაფართოების პროცესში ჩვეულებისამებრ სრულდება ტრადიციული ეტაპები:

- მოთხოვნილებათა განსაზღვრა, ან იტერაციათა დაგეგმვა, - ფიქსირდება, თუ რა უნდა იყოს შესრულებული მოცემულ იტერაციაზე სფეროს აღწერის სახით, რომლისთვისაც იგეგმება ფუნქციობის დამუშავება, და რა არის ამისთვის საჭირო. ამ ეტაპზე შეირჩევა ის სცენარები, რომლებიც უნდა იყოს რეალიზებული მოცემულ იტერაციაზე.
- ანალიზი - ხდება დაგეგმილ მოთხოვნილებათა შესრულების პირობების გამოკვლევა, მოწმდება შერჩეული სცენარების სისრულე მოთხოვნილი ფუნქციობის რეალიზაციის თვალსაზრისით.
- მოდელირება მომხმარებლის ინტერფეისის - რადგანაც იტერაცია უნდა უზრუნველყოფდეს ფუნქციურად დასრულებულ რეალიზაციას, საჭიროა განისაზღვროს ურთიერთმოქმედების წესები, რომლებიც აუცილებელია წარმოდგენილი ფუნქციების გააქტიურებისათვის. ინტერფეისის მოდელი ასახავს მომხმარებლის წარმოდგენას მოცემული იტერაციის ობიექტების ყოფაქცევის შესახებ.
- კონსტრუირება - პროექტის დეკომპოზიციაა, განხორციელებული ობიექტორიენტირებულ სტილში. იგი მოიცავს კლასთა სისტემის იერარქიის აგებას ან გაფართოებას, მოვლენათა აღწერას და მათზე რეაქციის განსაზღვრას. კონსტრუირების დროს განისაზღვრება ობიექტები, რომლებიც რეალიზდება ან ფართოვდება მოცემულ იტერაციაზე, ასევე ფუნქციათა ერთობლიობა (ობიექტთა მეთოდები), რომელიც უზრუნველყოფს მოცემული იტერაციის ამოცანის გადაწყვეტას.
- რეალიზაცია (დაპროგრამება) - პროგრამული განხორციელება გადაწყვეტილებისა, რომელიც მიღებულ იქნა

მოცემულ იტერაცაზე. რეალიზაციის აუცილებელ კომპონენტად აქ მოიაზრება შემადგენელი მოდულების თავსებადობის ავტონომიური შემოწმება მათ სპეციფიკაციებთან (კერძოდ, უზრუნველყოფილ უნდა იქნეს ობიექტების საჭირო ყოფაქცევა).

- ტესტირება - კომპლექსური შემოწმება შედეგებისა, რომლებიც მიიღება მოცემულ იტერაციაზე. ზოგჯერ, როგორც ტრადიციულ სქემებში, ტესტირების ეტაპს აერთიანებენ სასიცოცხლო ციკლის შემდეგ ეტაპთან.
- იტერაციის შედეგების შეფასება ხდება მიღებული შედეგების ანალიზი მთლიანი პროექტის ჭრილში. კერძოდ, გაირკვევა, თუ პროექტის რომელი ამოცანის გადაწყვეტაა შესაძლებელი მოცემული იტერაციის შედეგით, რომელ ადრე დასმულ კითხვებზე იქნას პასუხი გაცემული, რა სახის ახალი კითხვები გაჩნდა ახალ პირობებში.

- **განტერის „ფაზა-ფუნქციები"-მოდელის მოდიფიკაცია**

ობიექტორიენტირებული დაპროექტების დროს სასიცოცხლო ციკლის ფაზური განზომილება თითქმის არ იცვლება, მხოლოდ ემატება ერთი ეტაპი - მომხმარებლის ინტერფეისის მოდელირება. ეს საკითხი ძველ მოდელებში განიხილებოდა ანალიზისა და კონსტრუირების ეტაპებზე. საჭიროა აღინიშნოს, რომ ეს მეტად მნიშვნელოვანი დამატებაა ობიექტორიენტირებული მოდგომით სასიცოცხლო ციკლის მოდელირებისათვის. შემდეგი ორი პრინციპი დაკავშირებულია ამასთან.

**7. რელიზებულ მოთხოვნილებათა განაწილება იტერაციების მიხედვით:**

სცენარების ერთობლიობა, რეალიზებული წინა და მომდევნო იტერაციებზე, ყოველთვის ქმნის დასრულებულ, მაგრამ არასრული ვერსიის სისტემას, რომელიც გადაეცემა მომხმარებლებს. სხვადასხვა მიზეზის გამო, მათ შორის

ორაზროვნების გამორიცხვის მიზნით, საჭიროა სარეალიზაციოდ მიღებული დასაგეგმი საშუალებების წარმოდგენა მოდელების სახით, რომლებშიც შეთანხმებულია მომხმარებელთა (დამკვეთის) შეხედულებანი სისტემაზე, დამპროექტებელთა თვალსაზრისით. ეს მოდელები ჩნდება ანალიზის ეტაპზე, ამიტომაც მათ უწოდებენ - ანალიზის დონის მოდელებს.

**8. სისტემის შესაძლებლობათა გაფართოების განსაკუთრებული სტილი და მისი განვითარება:**

ობიექტორიენტირებული მიდგომის დროს სისტემა წარმოიდგინება როგორც სხვადასხვა დამოკიდებულებით დაკავშირებულ კლასთა ერთობლიობა, რაც პროექტის დეკომპოზიციის საფუძველია. ყოველი ახალი იტერაცია აფართოებს ამ ერთობლიობას ახალი კლასების დამატებით, რომლებიც განსაზღვრულ დამოკიდებულებებში შედის არსებულ კლასებთან. ასეთი გაფართოების კორექტული შესრულება, არსებულისა და პერსპექტივის დეტალების აბსტრაჰირების გათვალისწინების გარეშე, შეუძლებელია. ანუ, აუცილებელია **კონსტრუირების დონის მოდელების** აგება, რომლებიც ასახავს საპროექტო სისტემის სარეალიზაციო სახეს.

გარდა ზემოაღნიშნული ანალიზისა და კონსტრუირების დონეთა მოდელირებისა, არსებობს მესამე ასპექტი მოდელირებისა, რომელიც დაკავშირებულია პროგრამული სისტემის ყოველი ვერსიის წარდგენასთან მომხმარებელთან. თუ განტერის მოდელის სტილს გამოვიყენებთ სასიცოცხლო ციკლის აღსაწერად, მაშინ სწორი იქნება არა მოდელირების ეტაპის გამოყოფა (როგორც ტრადიციულადაა მიღებული), არამედ ორგანიზაციულ-ტექნიკური (საწარმოო) ფუნქციის მოდელირებისა, რომელიც გამჭოლად განიხილავს პროექტის დამუშავების მთელ პროცესს.

მე-17 და 21-ე ნახაზებზე, შესაბამისად, მოცემულია განტერის სასიცოცხლო ციკლის მოდელი (17) და მისი მოდიფიკაცია ობიექტ-ორიენტირებული მიდგომისათვის (21).

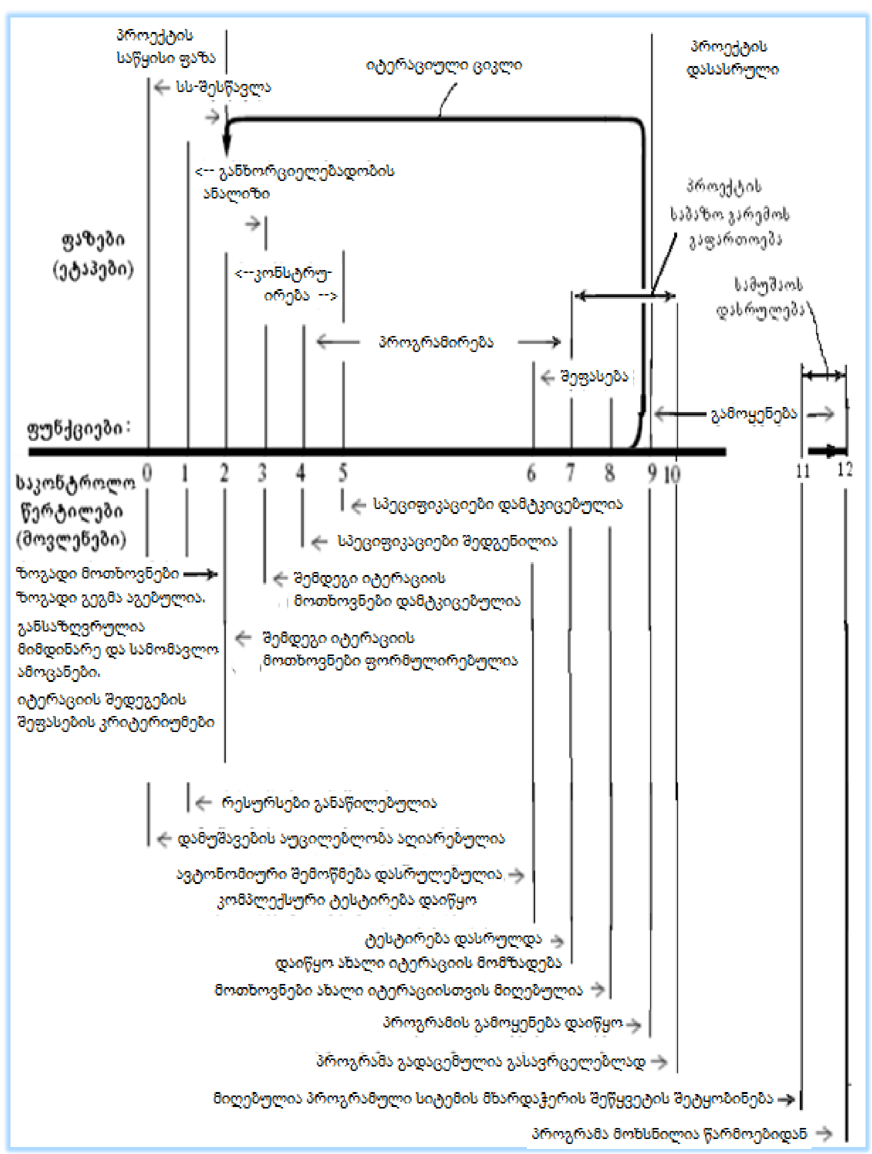
ოო-მიდგომის დროს იტერაცია გულისხმობს არა წინა ბიჯების შეცდომების გასწორებას, არამედ დაგეგმილ აქტს - შესაძლებლობების გაფართოებას.

აქვე შეფასების ეტაპზე გამოყოფილია სპეციალური ჩადგმული ეტაპი - პროექტის საბაზო გარემოს გაფართოება, ანუ იგულისხმება პროგრამული უზრუნველყოფის ხელმეორე გამოყენების დაგეგმვა და რეალიზაცია (პროტოტიპების გამოყენება). სასურველია შეფასების ეტაპზე ხდებოდეს ასეთი სასარგებლო პროექტების შენახვა (საცავებში).

განტერის მოდელი არ ასახავს იტერაციულობას, დაბრუნებას წინა ეტაპებზე, რაც ერთგვარად არასახარბიელოდ ითვლება, თუმცა ტრადიციულ და კასკადურ მოდელებში ეტაპების სათანადო გადაფარვა ხორციელდება.

იმისათვის, რომ იტერაციულობა იყოს გათვალისწინებული, საჭიროა მცირე მოდიფიკაცია, რაც მე-17 ნახაზზეა ნაჩვები. აქ სასიცოცხლო ციკლის საკონტროლო წერტილებში (მაგალითად, 3,4,6) ნაჩვენებია იტერაციული ციკლების შესაძლებლობები.

- რომელიმე საკონტროლო წერტილში მოხდება ძირითადი პროცესის შეჩერება, დაბრუნება რომელიმე წინა ეტაპზე გარკვეული დროის განმავლობაში. შემდეგ ისევ გადმოსვლა ძირითად პროცესზე და შედეგების შერწყმა ცვლილებების გათვალისწინებით;
- ძირითადი პროცესი არ წყდება იტერაციული ციკლის შესრულების მინით. იგი გრძელდება ჩვეულებრივად, ხოლო იტერაცია განიხილება როგორც დამატებით პროცესი, რომელიც უნდა მიუერთდეს სასიცოცხლო ციკლს. ასეთი დამატებითი პროცესი ორგანიზებულად უნდა წარიმართოს ანუ შესაძლებელია მისი წინასწარი დაგეგმვა.




ნახ.21. განტერის მოდელი იტერაციით ობიექტორიენტირებული პროგრამული პროექტისთვის

განიხილება პროექტების შესრულების ორი შემთხვევა:

იტერაციულობა გარდაუვალია რთული პროგრამული სისტემების დასაპროექტებლად. ამიტომაც მიზანშეწონილია ასეთი პროცესების დაგეგმვა. ისინი უნდა იქნას აღქმული არა როგორც „შეცდომების“ გასწორების იტერაციული პროცედურები, არამედ როგორც სისტემის გაფართოების აუცილებელი იტერაციები.

ეს საკითხი განსაკუთრებით საყურადღებოა ობიექტ-ორიენტირებული დაპროგრამების პრინციპების გამოყენებისას.

ახლა განვიხილოთ მე-21 ნახაზზე მოცემული 12 საკონ-ტროლო წერტილის (სწ) არსი.

**0. დამუშავების აუცილებლობა აღიარებულია**: ეს საკონტროლო წერტილი (სწ-0) მიუთითებს პროექტის სამუშაოების დაწყებაზე, ოღონდ თვით პროექტი ჯერ ოფიციალურად არაა დამტკიცებული. იგი ხდება ოფიციალური მას მერე, როცა დამტკიცდება მისი განხორციელებადობა, განისაზღვრება პროექტის მთლიანი ამოცანები, მათ შორის პირველ რიგში გადასაწყვეტ ამოცანათა ერთობლიობა (სწ-2). ამ ეტაპზე მენეჯერი ახორციელებს წინასაპროექტო საქმიანობას (შესაძლოა გუნდთან ერთად). მას გამოაქვს გადაწყვეტილება პროექტის განხორცი-ელების მიზანშეწონილობის შესახებ. ამ ეტაპზე მნიშვნელოვანი საკითხია საპროექტო გუნდის ფორმირების სტრატეგია და ტაქტიკა.

**1. რესურსები განაწილებულია**: საპროექტო სამუშაოების დასაწყებად აუცილებელია იმის ცოდნა, თუ რა რესურსები გვექნება. მენეჯერი იღებს საჭირო ინფორმაციას დამკვეთი ორგანიზაციის მგეგმავისა და ინვესტორისაგან. ამის საფუძველზე იგი განსაზღვრავს პროექტის განვითარების კონცეფციას, რომელიც მოტივირებულია დამკვეთის მოთხოვნილებებით.

**2. მოხოვნილებანი შემდეგი იტერაციისთვის განსაზღვრულია, ზოგადი მოთხოვნები და ზოგადი გეგმა შედგენილია, უახლესი ამოცანა, შედეგების შეფასების კრიტერიუმები და პერსპექტიული ამოცანები განსაზღვრულია**: ეს სწ-2 პროექტის განვითარების სტაციონარული ციკლის დასაწყისია, რომელზეც იგეგმება მიმდინარე იტერაციის მსვლელობა არსებული ინფორმაციის საფუძველზე. პირველ და მომდევნო იტერაციებზე აქ უნდა განისაზღვროს:

- **ზოგადი მოთხოვნილებანი** - რა მოითხოვება პროექტიდან მთლიანად ამ მომენტში;
- **საერთო გეგმა** - როგორ მიიღწევა დასმული მიზნები;
- **უახლესი ამოცანა -** მიმდინარე იტერაციის ამოცანა (ცხადად ჩამოყალიბებული).

იტერაციის შედეგების შეფასება ხდება **კრიტერიუმებით**. მათი შედგენილობა დამოკიდებულია როგორც პროექტის სპეციფიკაზე, ისე მისი შესრულების მდგომარეობაზე. ტიპური კრიტერიუმებია:

- **აქტუალობა მომხმარებლისათვის:** ბიზნესის თალსაზრისით მნიშვნელოვანია, რომ პროგრამულ სისტემაში აისახოს ის ფუნქციობა, რომელიც შეძლებს ავტომატიზებული საქმიანობის „ვიწრო ადგილების“ აღმოფხვრას;
- **შემოთავაზებული საშუალებების სისრულე და ფუნქციური ჩაკეტილობა:** იგულისხმება მომხმარებელთა საქმიანობის რომელიმე სახის სრული ავტომატიზაცია, რომელიც არ მოითხოვს დამატებით რაიმე სხვა საშუალებებს;
- **სისტემური მნიშვნელობა:** პირველ რიგში მიზანშეწონილია იმ ფუნქციობის რეალიზაცია, რომელიც სასარგებლოა პროექტის შემდგომი განვითარებისათვის ანუ იმ კომპონენტებისა და საშუალებების დამუშავება, რომლებიც გამოსადეგი იქნება პროექტის მომდევნო იტერაციებზე;

- სადემონსტრაციო მნიშვნელობა: ამ კრიტერიუმის თვალსაზრისით პირველ რიგში გამოიყოფა ის საშუალებები, რომელთაც შეუძლია სისტემის შრომისუნარიანობის და მისი საუკეთესო თვისებების დემონსტრირება, რაც სასარგებლოა სისტემის ხარისხის საჩვენებლად. ამ კრიტერიუმს დიდი მნიშვნელობა აქვს პროექტის საწყისი ეტაპის პერიოდში, როცა მიღებული შედეგების საფუძველზე დამკვეთიც და შემსრულებლებიც დარწმუნდება პროექტის შესრულების სტრატეგიის და მეთოდების სისწორეში. ეს კრიტერიუმი შეიძლება იყოს წინააღმდეგი სხვა კრიტერიუმებთან - მაგ., აქტუალობის, ფუნქციონალობის, სისტემის მნიშვნელობის;

- **რეალიზაციის სისწრაფე**: გამოიყოფა ის სამუშაოები, რომლებიც შეიძლება შესრულდეს ყველაზე მოკლე ვადებში. თუ იტერაციის დრო ფიქსირებულია, მაშინ გამოიყოფა სამუშაოთა ის მოცულობა, რომელსაც გუნდი განახორციელებს ამ დროში. დრო, ამ თვალსაზრისით განიხილება არა როგორც კრიტერიუმი, არამედ როგორც შეზღუდვა.

- **სისრული კრიტერიუმი.** განასხვავებენ სისრულის სამ ასპექტს, როლებიც ასახავს პროგრამული სისტემის ხარისხის სამ შემადგენელს:

- ფუნქციური სისრულე - შეთავაზებული საშუალებების შესაბამისობა მომხმარებელთა ფუნქციების ერთობლიობასთან, რომლებიც შეიძლება შესრულდეს ავტომატიზებულ სისტემაში;
- რეალიზაციის სისრულე - შესაძლებლობა ფუნქციური სისრულის უზრუნველსაყოფად, სისტემის საბაზო საშუალებების კომბინაციის გზით, სარეალიზაციო მექანიზმებითა და სამუშაო გარემოს საშუალებებით;
- საინტერფეისო სისრულე - სისტემის ყოფაქცევის მართვისათვის ისეთი ენის შერჩევა, რომელიც მომხმარებელს ხელს შეუწყობს სისტემის ადეკვატურად აღქმაში.

შეიძლება აღინიშნოს, რომ რეალიზაციისა და საინტერფეისო სისრულეები წარმოებულია ფუნქციური სისრულიდან.

ზოგჯერ მოცემული იტერაციისთვის განიხილავენ იმ მინიმალური რეალიზაციის შესაძლებლობას, რაც უზრუნველყოფს სისრულეს და ფუნქციურ ჩაკეტილობას. ეს განსაკუთრებით ხაზგასმულია **ექსტრემალური დაპროგრამების** მიდგომის დროს.

პირველი იტერაციის ამოცანის გადაწყვეტისას კრიტერიუ-მები მოწესრიგდება შემდეგი მიმდევრობით: სადემონსტრაციო მნიშვნელობა, სამომხმარებლო აქტუალობა, რეალიზაციის სისწრაფე, სისტემური მნიშვნელობა და ფუნქციური სისრულე. **მენეჯერის** ხელოვნება მდგომარეობს იმაში, რომ არსებული შეზღუდვების პირობებში დაიცვას უპირატეს კრიტერიუმთა ბალანსი და უზრუნველყოს მეთოდების, მეტრიკების, სტრატეგიებისა და პროგნოზების შემოწმებისა და კორექტირების შესაძლებლობა.

პირველი რიგის ამოცანისათვის უნდა განისაზღვროს:

- რეალიზაციის გეგმები - დაყოფა ეტაპებად, შესრულების ვადები და რესურსული მოთხოვნები.
- შედეგების შეფასების კრიტერიუმები - მეთოდიკები, რომლებითაც დადგინდება გადაწყვეტილი ამოცანის მოთხოვნილებების შეფასება ვადებისა და ხარისხის მიხედვით;
- პერსპექტიული ამოცანები - მათი დაფიქსირება აუცილებელია მკაცრი ანგარიშგების პირობებში, ის ყოველთვის მოწმდება მე-2 საკონტროლო წერტილში. პერსპექტიული ამოცანების მოთხოვნილებათა შემუშავების ხარისხი შემდგომ იტერაციებზე შეიძლება იყოს განსხვავებული.

ბოლო პუნქტი, მაგალითად არ სრულდება ექსტრემალური დაპროგრამების დროს, თუმცა ამ დროსაც სასარგებლოა

პერსპექტიული განვითარების ჰიპოთეზის შემუშავება, სამომავლო სამუშაოებისათვის.

**3. შემდგომი იტერაციების მოთხოვნები დამტკიცებულია:** პროექტის შესახებ ყველა ცნობები, რომლებიც მე-2 საკონტროლო წერტილშია წარმოდგენილი, მე-3 საკონტროლო წერტილში მისვლის მომენტში უნდა იყოს შეთანხმებული დასამტკიცებლად. დიდი და რთული პროექტებისათვის ეს საკითხი ფორმალურად უნდა აისახოს ანუ გაფორმდეს საანგარიშო რეპორტის სახით, რომელიც დამტკიცდება. მარტივ შემთხვევებში ეს არაა საჭირო. მას პროექტის მენეჯერი აკონტროლებს.

**4. სარეალიზაციო სცენარების სპეციფიკაციები შედგენილია:** კონსტრუირების ეტაპის დასაწყისი უკავშირდება პროექტის (იტერაციის) ამოცანის დეკომპოზიციას და სისტემის არქიტექტურის აგებას. როდესაც განსაზღვრული იქნება არქიტექტურა (თუნდაც ზოგადად), შესაძლებელი იქნება ქვესისტემების დამმუშავებლების სამუშაოების დაწყება. გუნდის ხელმძღვანელებს ექნებათ საპასუხისმგებლო სფეროები მოცემული იტერაციისათვის. პროექტის მენეჯერის მოვალეობა ამ წერტილში არის გუნდების მიერ მომზადებული სარეალიზაციო სცენარების დასამტკიცებლად გაფორმება.

**5. სპეციფიკაციები დამტკიცებულია:** საკონტროლო წერტილი აღნიშნავს საკონსტრუქტორო ეტაპის დასასრულს. შემდეგი იტერაციის არქიტექტურა დამტკიცებული და დაფიქსირებულია დავალებების სახით ქვესისტემების დამმუშავებლების და გუნდის ხელმძღვანელებისათვის, რომელთაც ევალებათ პროექტის კლასების შექმნა ან მოდელირება.

**6. პროგრამული სისტემის ავტონომიური შემოწმება დასრულდა და კომპლექსური ტესტირება დაიწყო:** დაპროგრამების ეტაპის დასრულებისას საჭიროა სისტემის მუშაობისუნარიანობის კომპლექსური შემოწმება. ესაა შეფასების ეტაპის დაწყება, ვინაიდან

ამ მომენტიდან არის შესაძლებელი პროექტის (ან იტერაციის) შესახებ მოსაზრებების სისწორის პრაქტიკულად შემოწმება. სწრაფი დამუშავების (ექსტრემალური პროგრამირება) პროექტებში არ თვლიან აუცილებლად კონსტრუირების ეტაპის ცალკე გამოყოფას (სწ-ები 3,4,5 შერწყმულია). ასევე ამ მეთოდის გამოყენებისას არ იყოფა შედეგების შემოწმება ავტონომიურად და კომპლექსურად, რადგანაც სისტემის ყოველი ახალი შესაძლებლობა თავიდანვე ინტეგრირდება უკვე არსებულ შესაძლებლობებთან. აქ უნდა გადაწყდეს, როგორ ჩაშენდეს ახალი შესაძლებლობა არსებულ არქიტექტურულ კარკასში.

**7. ტესტირება დასრულდა, დაიწყო მზადება ახალი იტერაციისთვის:** ეს საკონტროლო წერტილი ასახავს პროგრამული სამუშაოების დასასრულს მოცემულ იტერაციაზე. პროგრამული სამუშაოები გრძელდება, რომლებიც დაკავშირებულია პროექტის საბაზო გარემოს გაფართოებასთან. იგი განიხილება როგორც შედარებით დამოუკიდებელი ეტაპი, რომელიც ჩადგმულია შეფასების ეტაპში, და რომელიც თავდება მასთან ერთად (სწ-10). ეს სამუშაოები ორი სახისაა: ა) მხოლოდ ამ პროექტისათვის მრავალჯერადი გამოყენების ზოგადი კომპონენტების გამოყოფა (სწ-9-ში); ბ) ზოგადი (არა მხოლოდ ამ პროექტისათვის) მრავალჯერადი გამოყენების კომპონენტების გამოყოფა (ამ სამუშაოების დაწყება მიზანშეწონილია მაშინ, როცა იტერაციის პროგრამული პროდუქტი განიხილება როგორც მზა აპლიკაცია (სწ-9) და დასრულდეს სისტემის გასავრცელებლად გადაცემის მომენტში (სწ.10)). პროექტის იტერაციული განვითარების უზრუნველყოფა ხდება ცნობების შეკრებით (ცოდნის მიღებით) ახალი იტერაციისათვის ამ საკონტროლო წერტილში.

**8. ახალი იტერაციისთვის მოთხოვნილებები მიღებულია:** ახალი იტერაციის ცნობების დამუშავების დროს განისაზღვრება მისაღები მოთხოვნილებანი, რომლებიც ფართოვდება სისტემის

გამოყენების პირველ პერიოდში დამმუშავებლების მიერ (მას **ალფა-ტესტირების** პერიოდს უწოდებენ). მისი დასრულების შემდეგ ხდება სასიცოცხლო ციკლის გახლეჩა სისტემის ექსპლუატაციაში გადაცემისა და ახალი იტერაციული სამუშაოების ორგანიზების პროცესების პარალელურად შესრულების მიზნით. გახლეჩა იწვევს სისტემის ორი ვერსიის არსებობას: ერთი გამოყენების პროცესშია, ხოლო მეორე ჩაისახება ახალი მოთხოვნილებების სახით. ესაა დრო, როდესაც უნდა შემოწმდეს პროექტის აპრიორული ჰიპოტეზები, ჩატარდეს პროექტის მაჩვენებელთა და ნორმატივების კორექტირება,

**9. დაიწყო სისტემის გამოყენება:** ეს საკონტროლო წერტილი ასახავს ე.წ. **ბეტა-ტესტირების** დაწყებას ანუ სისტემის გადაცემას ექსპლუატაციაში, მისი გარე შეფასების მიზნით. შესაბამისი სამუშაოების ჩატარების შემდეგ (ნაპოვნი დეფექტების გასწორება) შესაძლებელია იტერაციის შედეგების გავრცელება (სწ-10). სწ-9 მიუთითებს **პროექტის (იტერაციის) ფაზის დასასრულს**. ეს ფაზა მსგავსია ექსპლუატაციის და თანხლების ტრადიციული ფაზებისა, მაგრამ არის განსხვავებაც. ვინაიდან ოო-პროექტს საქმე აქვს სისტემის ვერსიების იერარქიასთან, რომლებშიც ასახულია შესაძლებლობათა გაფართოება. ეს ფაზა იკვეთება შეფასების ეტაპთან. პროექტის (იერაციის) ფაზის დასასრული მოიცავს:

- სისტემის პაკეტირება ან მიწოდება მომხმარებელზე (სწ-9);
- პროგრამული პროდუქტის თანხლება;
- სამუშაოს დასრულების ეტაპი (სწ.: 11,12)

**10. პროგრამული სისტემა ან მისი ვერსია გადაცემულია გასავრცელებლად:** ახალ ვერსიას უჩნდება ახალი მომხმარებლები, რომელთაც ესაჭიროებათ მომსახურება.

**11. შეტყობინება სისტემის (ვერსიის) მხარდაჭერის შეწყვეტის შესახებ:** აუცილებელია მომხმარებელთა ინფორმირება ასეთი

გადაწყვეტილების მიღებისას და დასაბუთება. მომხმარებელმა უნდა შეძლოს ახალ რესურსებზე გადაწყობა ან დაასაბუთოს ძველის თანხლების აუცილებლობა.

**12. პროგრამული სისტემა მოხსნილია წარმოებიდან:** სისტემაზე (ან მის ვერსიაზე) თანხლება შეწყვეტილია. ზოგიერთ მომხმარებელს დასჭირდება თანხლების გაგრძელება სათანადო ვადით. კლიენტები რომ არ დაკარგონ, პროგრამისტები ეთანხმებიან დროებით თანხლებას, ან სისტემის განახლებას. დაფინანსება შესაბამისად ხორციელდება კლიენტის მიერ.

## 1.8. პროგრამული სისტემების დაპროექტების UML ტექნოლოგია და მისი ინსტრუმენტები

მოდელირების უნიფიცირებული ენა – UML (Unified Modeling Language) შექმნილია, როგორც უნივერსალური მოდელირების ენა ობიექტორიენტირებული პროგრამირების სფეროში და არის სტანდარტული ვიზუალური მოდელირების ენა, რომელიც იძლევა საშუალებას სისტემა აღიწეროს გრაფიკულად და ტექსტურად.

UML პრაქიკულად წარმოადგენს ბუჩის მეთოდის (Booch Method), ობიექტის მოდელირების ტექნიკის (Object-modeling technique – OMT) და ობიექტორიენტირებული პროგრამული უზრუნველყოფის ინჟინერიის (Object-oriented software engineering - OOSE) სინთეზსს და გვევლინება როგორც ერთი, საერთო და ფართო გამოყენების მოდელირების ენა [12].

UML არის მსოფლიოში ყველაზე ფართოდ გამოყენებადი უნიფიცირებული მოდელირების ენა. იგი შექმნილია საერთაშორისო ასოციაციის, ობიექტების მართვის ჯგუფის – OMG (Object Management Group) მიერ, რომელიც ქმნის ღია სტანდარტებს

ობიექტორიენტირებული აპლიკაციებისათვის ანუ UML ეს არის გრაფიკული ენა, რომელიც გამოიყენება ობიექტორიენტირებული მოდელირების აღწერისათვის პროგრამული უზრუნველყოფის სფეროში და რაც მნიშვნელოვანია, იგი არის „ღია სტანდარტი“, რომელიც ხემისაწვდომია ყველასათვის [25]

UML აერთიანებს მონაცემთა მოდელირების (entity relationship diagrams) ბიზნესმოდელირების (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 [26-28].

არის რამდენიმე მიზეზი, რატომ უნდა გამოვიყენოთ UML ენა მოდელირებისთვის:

- უნიფიცირებული ტერმინოლოგიის და მნიშვნელობათა სტანდარტიზაციის საშუალებით მნიშვნელოვნად გამარტივებული არის ურთიერთობა მოდელირებადი სისტემის სხვადასხვა მხარეს შორის. ეს აადვილებს მოდელების გაცვლას სხვადასხვა დეპარტამენტსა და კომპანიას შორის, განსაკუთრებით პროექტების გადაგზავნას პროექტზე მომუშავე ჯგუფსა და საპროექტო ჯგუფებს შორის;

- მოდელირებაზე მოთხოვნის გაზრდასთან ერთად ვითარდება UML - ენაც. ვინაიდან UML არის მძლავრი მოდელირების 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 დიდ მიღწევებზე. იგი მოდელირების არსებული ენების გამოყენებით რეალური პრობლემების გადასაჭრელად შემუშავდა, რაც უზრუნველყოფს მის მოხერხებულობას და პრაქტიკაში გამართულ ფუნქციონირებას. UML დიდ მხარდაჭერას ფლობს.

**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, რომელიც ჯერჯერობით ფართოდ არ არის ცნობილი [27].

OMG-მ დღესდღეობით UML2.0 ვერსიის სტანდარტიზაცია დაასრულა. ახალი UML2.0 სპეციფიკაცია UML-ისთვის შეიცავს ისეთ სრულყოფილებებს, სიახლეებს, რომელიც რესტრუქტურიზაციას უკეთებს და ხვეწს ენას, რათა ის უფრო ადვილი გამოსაყენებელი, შესასრულებელი და ასაგები გახადოს. ყველაზე აშკარა ცვლილებები UML1 ვერსიისგან განსხვავებით, რაც UML2.0 –ს აქვს არის ახალი დიაგრამები: კომპოზიციური სტრუქტურის დიაგრამა (Composite structure diagram), დროითი დიაგრამა (Timing diagram), ურთიერთქმედების მიმოხილვის დიაგრამა (Interactive overview diagram) [27].

რაც შეეხება მთლიანად UML2-ს, UML1-ისგან განსხვავებით აქ შემუშვებულია ახალი დიაგრამები: 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 მოღვაწეობის დიაგრამის (activity diagrams) და მიმდევრობის დიაგრამის (sequence diagrams) გაფართოება.

UML2.0-ის ოთხი უმთავრესი დოკუმენტაციაა:

**სუპერსტრუქტურა.** სუპერსტრუქტურა იყოფა 6 ძირითად დიაგრამად (3 ქცევის და 4 ურთიერთქმედების დიაგრამა) და მათში შემავალი ელემენტებისაგან.

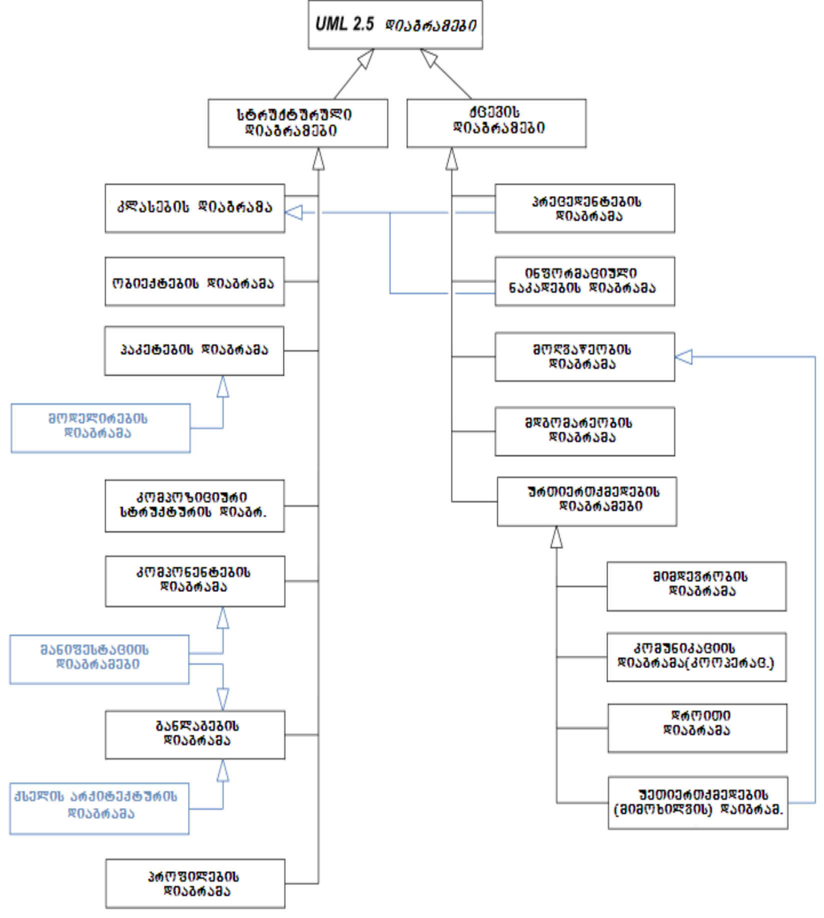
**ინფრასტრუქტურა.** UML2.0 – ინფრასტრუქტურა განსაზღვრავს საბაზისო კლასებს, რომელიც ქმნის საფუძველს არამარტო UML2.0 – ის სუპერსტრუქტურისათვის, არამედ MOF 2.0 - სთვისაც.

**დიაგრამების ურთიერთქმედება (ურთიერთჩანაცვლება).** UML2.0-ის ურთიერთქმედება აფართოებს UML-ის მეტამოდელს დამატებითი პაკეტით, გრაფორიენტირებული 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 - არის ენა, რომელიც მოქმედებებისა და შესრულებადი კოდების დაწერისათვის კი არა, არამედ შეზღუდვების და მოთხოვნების განსაზღვრისთვის არის განკუთვნილი.

**UML2.0-ის 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; ურთიერთ-ქმედების დიაგრამის განახლება; მოღვაწეობის დიაგრამის განახლება; UML - პროფილი [27,28].

ერთ-ერთი ბოლო ვერსიაა UML 2.5. მას აქვს 15 ტიპის დიაგრამა, რომელიც იყოფა 2 კატეგორიად. აქედან 7 დიაგრამა გვაძლევს სტრუქტურულ ინფორმაციას, ხოლო დანარჩენი 8 ქცევის საერთო ტიპს. მათ შორის 4 სახის დიაგრამა ასახავს ურთიერთქმედების სხვადასხვა ასპექტს. ამ დიაგრამების სტრუქტურა შეიძლება მოცემულია 22-ე ნახაზზე.




ნახ.22. UML2.5-ის დიაგრამების სტრუქტურა

UML არ აწესებს ელემენტებს რომელიმე ერთი კონკრეტული ტიპის დიაგრამისათვის. ზოგადად, ყველა UML ელემენტი შეიძლება შეგვხვდეს დიაგრამების თითქმის ყველა ტიპში. ეს მოქნილობა ნაწილობრივ შეზღუდულია UML2.0-ში. UML პროფილებმა შეიძლება განსაზღვროს დამატებითი დიაგრამის ტიპები ან გააფართოვოს უკვე არსებული დიაგრამები დამატებითი აღნიშვნებით [27].

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.

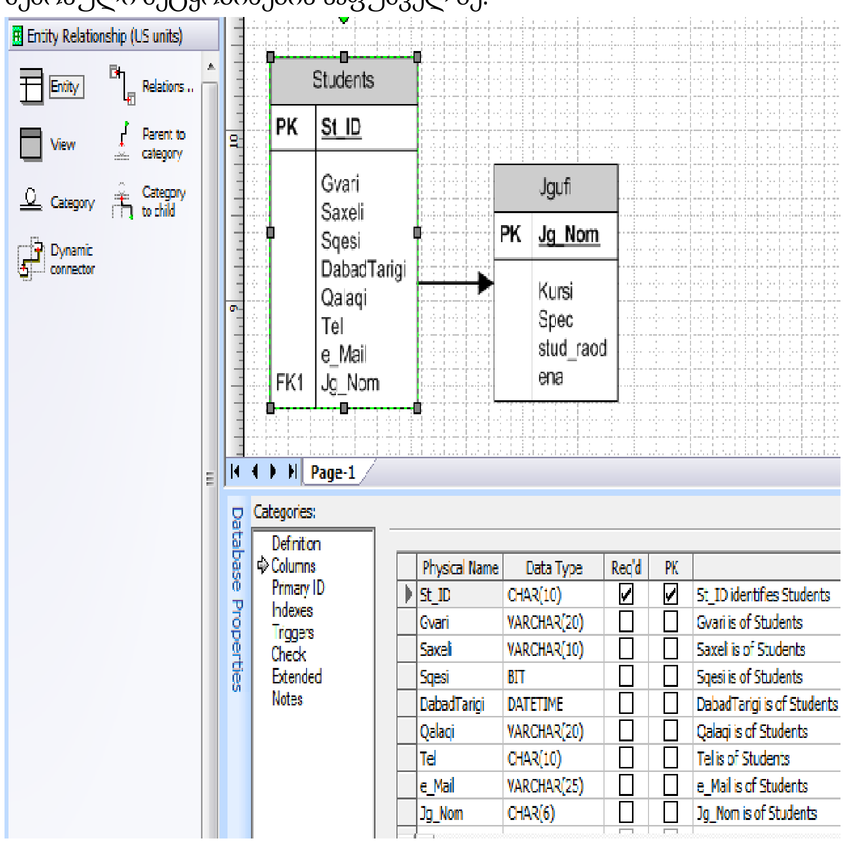
### 1.9. UML დიაგრამები

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 [17,18,27,28]. ქვემოთ მოყვანილი მაგალითები აგებულია Ms Visio და Enterprixe Architect ინსტრუმენტებით.

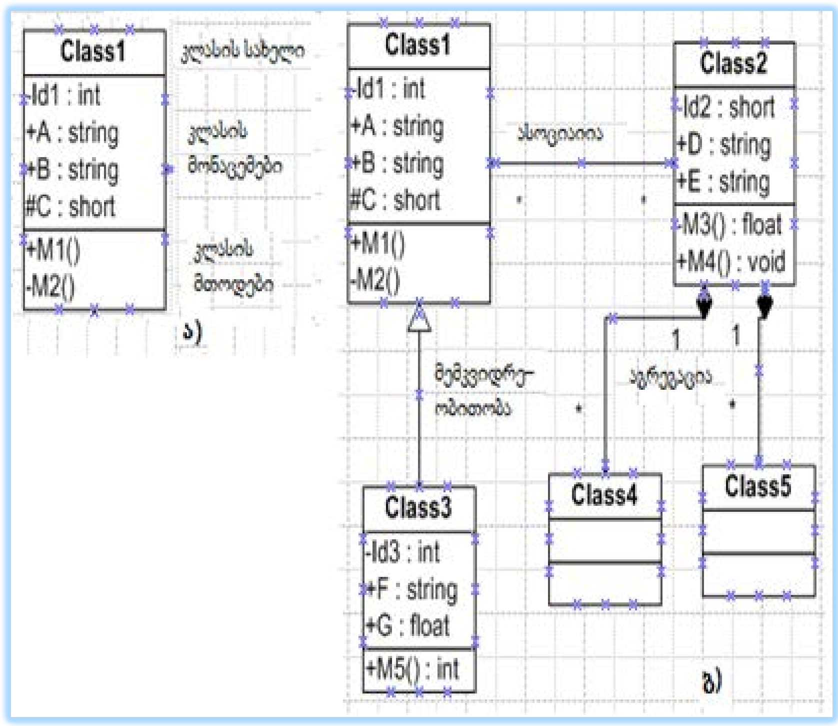
- ER დიაგრამის ვიზუალური აგება: Database Model Diagram არჩევით ეკრანზე გამოიტანება მონაცემთა ER-მოდელის ასაგები რედაქტორი (Entity-Relationship Model). 23-ე ნახაზზე ნაჩვენებია ინტერფეისი კონკრეტული „სტუდენტთა-ჯგუფის“ მონაცემთა მოდელისათვის.

განსაკუთრებით მნიშვნელოვანია კლასთა-ასოციაციის დიაგრამის აგება და შემდეგ პროგრამული კოდის გენერაცია (ნახ.24, 25). სხვა დიაგრამებს აქ აღარ შევეხებით, რადგან ისინი ლიტერატურულ წყაროებში მრავლადაა.

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



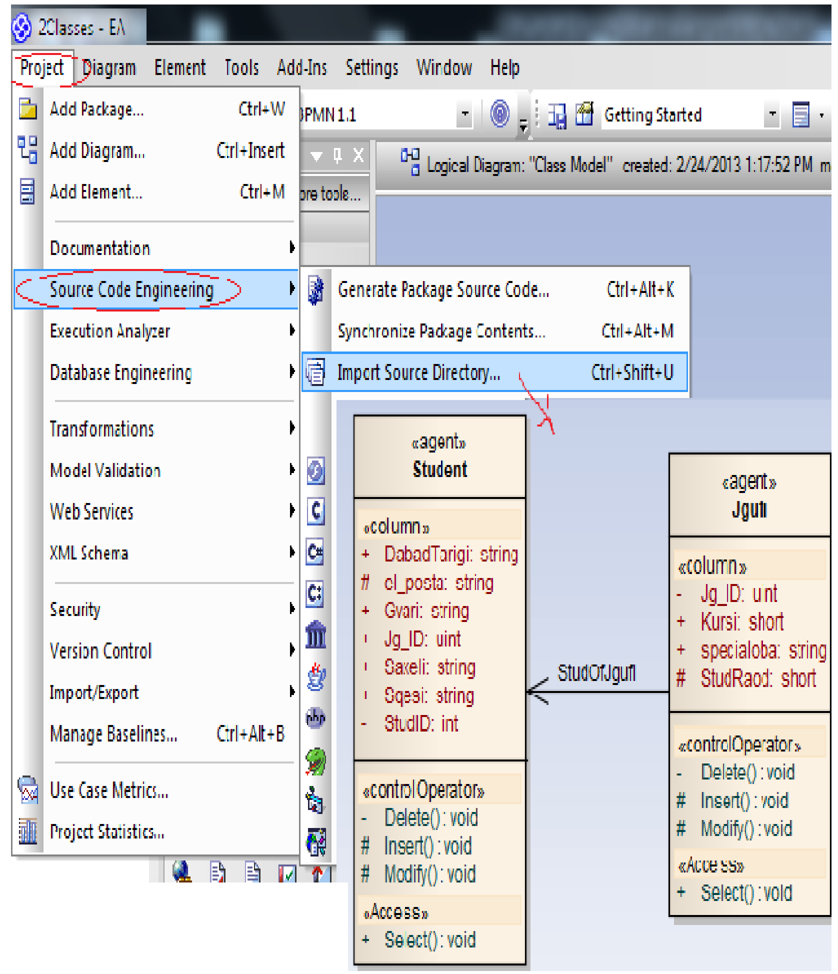

ნახ.23. Database Model Diagram –ის აგება




ნახ.24. კლასი (ა) და კლასთა დიაგრამა (ბ)

➢ მემკვიდრეობითი (Generalization) ასახავს „გენეტიკურ", კლასებს შორის განზოგადებულ კავშირებს. ასეთ დროს ერთი კლასი („შვილი") მთლიანად იღებს მეორე კლასის („მშობელი") ყველა ატრიბუტს, მეთოდსა და კავშირს;

➢ აგრეგირებული (Aggregation) ნიშნავს კავშირს „მთელი"-„ნაწილი". მაგალითად, „ავტომობილი" – „ძარა, ძრავი, საბურავები და ა.შ.";




ნახ.25. Student და Jgupi კლასების მომზადება
„Code Engineering“ პროცესისთვის

- 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 [18,28]. ანუ კლასების დიაგრამიდან შესაძლებელია პროგრამული კოდის გენერაცია და პირიქითაც, კოდიდან აიგება ავტომატურად გრაფიკული დიაგრამა. 25-ე ნახაზზე კოდის გენერირების პროცედურა Enterprise Architect ინსტრუმენტის სამუშაო გარემოში.

26-ე ნახაზზე ილუსტრირებულია გენერირებული კოდის ფრაგმენტი.

2Classes - EA    Student.cs

Student

- 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.26. C#-კოდის ლისტინგი Student კლასისათვის

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.

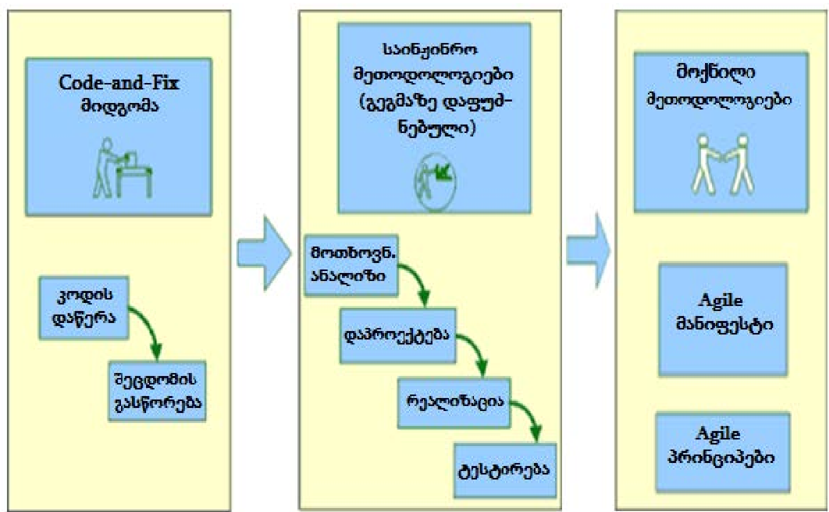
მცირე და სწრაფად შესასრულებელი პროექტებისათვის, დამკვეთის დაკმაყოფილების თვალსაზრისითაც, უფრო მისაღებია ე.წ. მოქნილი (Agile), ან ექსტრემალური დაპროგრამების მეთოდების გამოყენება, რაზეც შემდგომში გვექნება საუბარი.

## 1.10. პროგრამული სისტემების აგება მოქნილი (Agile) ტექნოლოგიებით

### 1.10.1. ექსტრემალური დაპროგრამების პრინციპები და ინსტრუმენტული საშუალებანი

2001 წლის თებერვლიდან ოფიციალურად იქნა ჩამოყალიბებული პროგრამული სისტემების დამუშავების ახალი, Agile - მეთოდოლოგია [13]. 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.27) [19, 24].

ფორმალიზაციის ხარისხის ნიხედვით 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.




ნახ.27. Agile მეთოდოლოგიებზე გადასვლა

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

- **მოქნილი დამუშავების მანიფესტი (Agile Manifesto) და პრინციპები**

მანიფესტი ოთხი პუნქტისაგან შედგება, რომელთაგანაც თითოეული ალტერნატივაა [24]. 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 კონტრაქტით შეთანხმებული პირობები;
- მზადყოფნა ცვლილებებისადმი უფრო მნიშვნელოვანია, ვიდრე პირველსაწყისი 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 ბოლოო 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 უზრუნველყოფილია მუშაობის საჭირო პირობებით, მხარდაჭე-რითა 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.

### 1.10.2. 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 !

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

AM ითვალისწინებს აგრეთვე CASE-საშუალებების ზომი-ერად გამოყენებასაც, თუკი ამით ეფექტურობა მაღლდება. ინსტრუმენტული საშუალებებიდან შეირჩევა უმარტივესი, რომელიც დასმული ამოცანის გადაწყვეტის საშუალებას იძლევა.

### 1.10.3. მოქნილი მოდელირების ფასეულობანი

AM-ის ფასეულობებია [14]: ურთიერთობა (კომუნიკაცია), სიმარტივე, უკუკავშირი, გამბედაობა და თავმდაბლობა.

აქედან პირველი ოთხი „ექსტრემალური დაპროგრამების“ მამას, კენტ ბეკსაც ჰქონდა მოცემული (2000 წ. XP) [13]. ს. ამბლერმა მეხუთე დაამატა. განვიხილოთ თითოეული მათგანი მოქნილი მოდელირების თვალსაზრისით.

- **კომუნიკაცია (ურთიერთობა)**

ტერმინოლოგიური ლექიკონის 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 მეორეხარისხოვანი 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 დამოკიდებულებისათვის.

- **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 შეძლების-დაგვარად 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, კომპონენტები, 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, 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 ინფრა-სტრუქტურის 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 უკუკავშირს მოდელებისა:

- **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 მოითხოვა 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 უცილებელი 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 ამოდელი-რებენ მოთხოვნილებებს ან ქმნიან არქიტექტურებს. მომხმარებ-ლები 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.

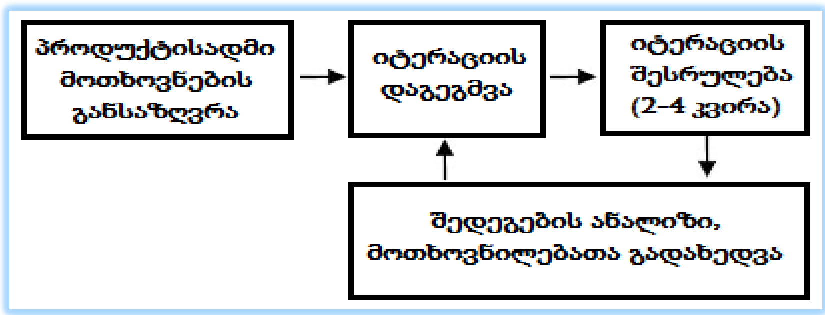
ამგვარად, პროგრამული სისტემის მენეჯერი, კონკრეტული პროექტის ამოცანებისა და მოთხოვნების შესაბამისად, უნდა განსაზღვრავდეს როგორც პროგრამირების მეთოდის, ეტაპთა ფაზებისა და იტერაციათა მოთხოვნების შერჩევა-ფორმირებას, ასევე მუშა გუნდის შემადგენლობას.

### 1.10.4. Scrum - მოქნილი მეთოდის მაგალითი -ეკა

პროგრამული ინდუსტრიის სფეროში Agile მეთოდოლოგიის მნიშვნელოვანი წარმომადგენელია Scrum მეთოდი [19,29]. იგი პირველად იაპონელებმა მოიხსენიეს, როგორც ახალი მიდგომა ახალი სერვისებისა და პროდუქტების დასამუშავებლად (არა მხოლოდ პროგრამული პროდუქტებისთვის) [30]. მეთოდის ძირითადი არსი მდგომარეობდა მცირე ზომის უნივრსალური გუნდის შეკრულ, თანამიმდევრულ მუშაობაში, რომელიც ამუშავებს პროექტის ყველა ფაზას. სიმბოლურ ანალოგიას აკეთებდნენ „რაგბის" თამაშთან, როდესაც ერთიანი გუნდი მოძრაობს წინ და უკან ბურთის გადაცემის შესაბამისად.

Scrum - ნიშნავს შეჭიდებას (схватка).

**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.28).




**ნახ.28. 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 მინიმალური ფუნქციური შესაძლე-ბლობებით.

ამის შემდეგ ხდება შედეგების განხილვა და პროდუქტი-სადმი 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.

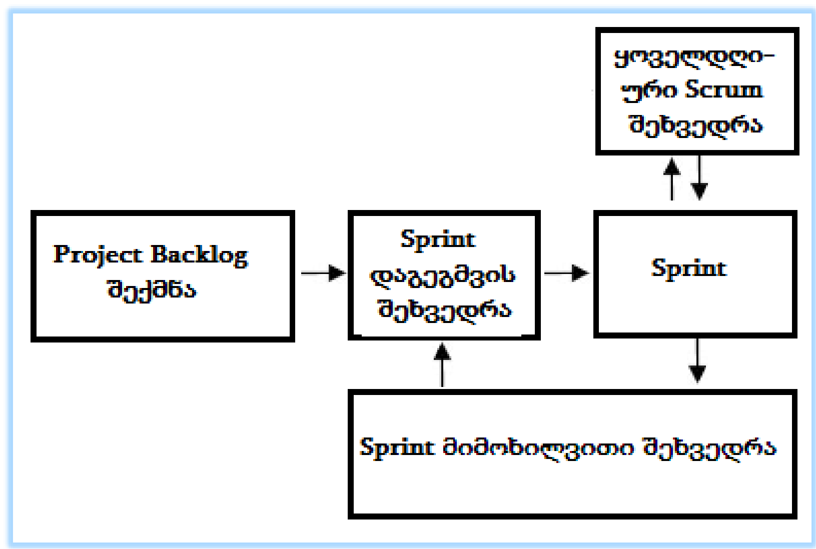
**როლები.** Scrum მეთოდში არის სამი როლი:

- 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 ადმინისტრაცული ამოცანების გადასაწყვეტად. კერძოდ, მისი ამოცანაა გუნდის დაცვა იტერაციის დროს ყოველგვარი გარე ზემოქმედებიდან.
- Scrum-გუნდი (Scrum Team) - ჯგუფია, რომელიც შედგება 5-9 დამოუკიდებელი, ინიციატივიანი პროგრამისტების. 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, კარგად აისახება მოვალეობათა დისციპლინა.

პრაქტიკები. Scrum-ში განსაზღვრულია შემდეგი პრაქტიკები:

- Sprint Planning Meeting. შეხვედრა Sprint-ის დასაგეგმად (ესაა მოკლე დისტანცია, ანუ იტერაცია). თავიდან პროდუქტის მფლობელი, Scrum-ოსტატი, გუნდი, ასევე დამკვეთის წარმომადგე-ნელი 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 აქვსა და რა უნდა გააკეთოს მომდევნო შეხვედრამდე. ამ თათბირზე დასწრება შეუძლია ნებისმიერ დაინტერესებულ პირს, მაგრამ გადაწყვეტილების მიღების უფლება აქვთ მხოლოდ Scrum-გუნდის წევრებს. ეს წესი მართებულია, რადგან ისინი იღებს ვალდებულებას იტერაციის მიზნის მიღსაღწევად. გარე პირის ჩარევა ასეთ დროს ხსნის მათგან შედეგისაგან პასუხისმგებლობას.
- Sprint Review Meeting. Sprint მიმოხილვის შეხვედრა. იმართება ყოველი სპრინტის დამთავრების შემდეგ (ნახ.29). ჯერ 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 დასკვნებსა და იღებს მნიშვნელოვან გადაწყვეტილებებს შემდგომი მუშაობისათვის. Scrum-გუნდი ასევე ეძებს გზებს მომავალი სამუშაოს ეფექტურობის ასამაღლებლად. შემდეგ ციკლი მეორდება.




**ნახ.29. Scrum-მეთოდი Sprint-ბიჯებით**

## 1.11. ITIL მეთოდოლოგია

ITIL - Information Technology Infrastructure Library არის ინფორმაციული ტექნოლოგიების ინფრასტრუქტურის ბიბლიოთეკა [10,11]. იგი დღეისათვის ძალზე აქტუალური და საყოველთაოდ ცნობილი ცოდნის ბაზაა სერვისების მართვის სფეროში მთელი მსოფლიოს მასშტაბით [22,25]. ის ასახავს IT–სფეროს მსოფლიოს წამყვანი პრაქტიკოსების ფუნდამენტურ საფუძვლებს. ევროპაში არსებობს ITIL სერტიფიცირების ორი ცენტრი: EXIN – ჰოლანდიის საგამოცდო ინსტიტუტი და ISEB (Information Systems Examination Board)-ბრიტანეთის კომპიუტერული საზოგადოების განყოფილება [31,32].

ITIL განიხილავს სერვისების მართვას ურთიერმოქმედების კონტექსტში: „სერვისების მიმწოდებელი– სერვისების დამკვეთი“.

**დამკვეთი (Customer)** - ესაა საქონლის ან მომსახურების მყიდველი. IT–სერვისების მიმწოდებლისთვის დამკვეთი არის ადამიანი (ან ჯგუფი), რომელიც აფორმებს შეთანხმებას მიმწოდებელთან IT–მომსახურების მისაღებად და პასუხს აგებს მიღებული მომსახურების ასანაზღაურებლად.

**მიმწოდებელი (Service provider)** - ესაა ორგანიზაცია, რომელიც აწვდის სერვისს ერთ ან რამდენიმე შიგა ან გარე დამკვეთს.

**მომხმარებელი** – ესაა IT –სერვისების გამომყენებელი თანამშრომელი დამკვეთ ორგანიზაციაში.

**IT-მომსახურება (სერვისი)** – დამკვეთებისთვის ფასეულობის მიწოდების ხერხი, რომელთა საშუალებითაც ისინი იღებს გამოსასვლელზე საჭირო შედეგს მათთვის სპეციფიური დანახარჯებისა და რისკების გარეშე.

შეიძლება განვიხილოთ სხაგვარი განსაზღვრებაც. IT–მომსახურება, როგორც – ერთი ან მეტი ტექნიკური ან პროფესიო-

ნალური შესაძლებლობა, რომელიც ხელს უწყობს ბიზნეს – პროცესს. ტერმინები „სერვისი“ და „მომსახურება“ ეკვივალენტურია. მათ აქვს შემდეგი მახასიათებლები:

- აკმაყოფილებს დამკვეთის ერთ ან მეტ მოთხოვნას;
- მხარს უჭერს დამკვეთის ბიზნეს–მიზნებს;
- დამკვეთისაგან აღიქმება როგორც ერთი მთლიანი პროდუქტი, რომელიც მზადაა გამოსაყენებლად.

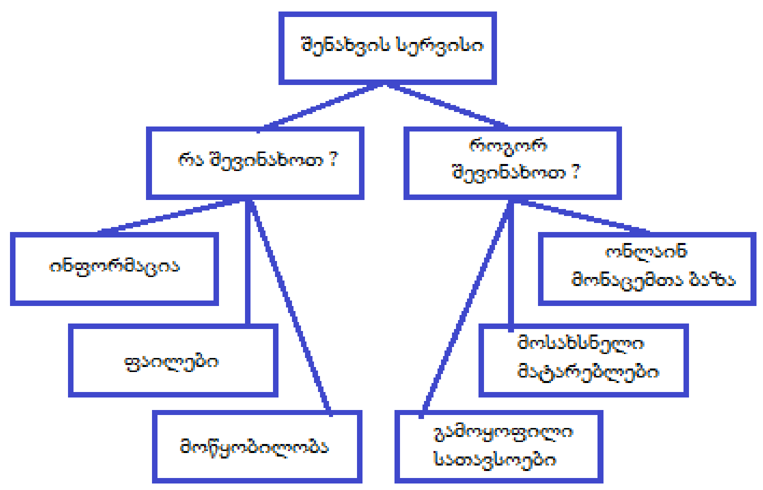
განვიხილოთ დეტალურად ძირითადი ცნებები სერვისის განსაზღვრებაში.

**შედეგები გამოსასვლელზე (outcomes)** – ის რასაც იღებს დამკვეთი საბოლოოდ. ცხადია, რომ იგი განსხვავდება დამკვეთის საწყისი მოთხოვნებისაგან, სათანადო შემზღუდველი ფაქტორების არსებობის გამო. სერვისის დანიშნულებაა ამ ფაქტორების შემცირებისა და მწარმოებლურობის ამაღლების გზით გამოსასვლელი შედეგების გაუმჯობესება. სერვისების გამოყენების შედეგია გამოსასვლელზე სასურველი შედეგების მიღების ალბათობის გაზრდა.

მომსახურების მოდელები, რომელთაც ITIL გვთავაზობს, გვეხმარება IT–სფეროს სირთულეების, ხარჯების, მოქნილობისა და მრავალსახეობის მართვაში. ყოველ მოდელს აქვს გამოყენების ვარიანტების სიმრავლე კონკრეტული შემთხვევისაგან დამოკიდე-ბულებაში, რაც მისი გამოყენების იდეას ხდის უნივერსალურს, მოქნილსა და ეფექტურს.

IT–სერვისის მოდელი შეიძლება განვიხილოთ ინფორმაციის შენახვის სისტემის მაგალითზე. სისტემა დანიშნულია ინფორმაციის შენახვის, მოწესრიგებისა და დაცვის განსახორ-ციელებლად რაიმე სამუშაოს ან მოქმედების კონტექსტში. თუ მიმწოდებელი აძლევს დამკვეთს არა მხოლოდ დამხსომებელ მოწყობილობას, არამედ აგრეთვე ინფორმაციის შენახვის სერვისსაც, მაშინ უნდა გაეცეს პასუხი კითხვებს „რა შევინახოთ“ და

„როგორ შევინახოთ“ (ნახ.30). ამასთანავე პრინციპულად მნიშვნელოვანია მოვალეობებისა და პასუხისმგებლობების განაწილება მიმწოდებელსა და დამკვეთს შორის.




**ნახ.30. ინფორმაციის შენახვის სისტემის სქემა**

დამკვეთებს სურთ სასურველი შედეგების მიღება, მაგრამ სხვადასხვა მიზეზთა გამო, არ სურთ თანმხლები პასუხისმგებლობა, ხარჯები და რისკები. მაგალითად, ორგანიზაციას უნდა დაცული ინფორმაციის შენახვის სისტემის შექმნა რამდენიმე ტერაბაიტით ონლაინ–ვაჭრობის მხარდასაჭერად.

ასეთი სისტემის შესაქმნელად „ნულიდან“ ამ ორგანიზციამ უნდა განვლოს გრძელი გზა, დაწყებული იმის გაგებით, თუ როგორ გააკეთოს, დამთავრებული ძვირადღირებული ტექნიკის შესყიდვით და კვალიფიციური პერსონალის დაქირავებით. ეს კი მეტად ძვირადღირებული სიამოვნება და დროის დიდ დანახარჯია.

შედარებით მარტივია ამ შემთხვევაში მიმწოდებლის სერვისების გამოყენება, რომელიც უკვე ფლობს ინფორმაციის შენახვის დიდ სისტემასა და აქვს შესაბამისი გამოცდილება და შესაძლებლობები. ეს იქნება ინფორმაციის დაცული შენახვის სერვისის შეთავაზება.

**სერვისის ფასი (value)** – იგი იზომება ორი ცნების კონტექსტში:

- სერვისის სარგებლობა (Service Utility) – არის ის, რასაც იღებს დამკვეთი სერვისის გამოყენებით;
- სერვისის ხარისხის გარანტია (Service Warranty) – არის ის, თუ როგორ აძლევს მიმწოდებელი დამკვეთს სერვისს – წვდომის, მწარმოებლურობისა და უსაფრთხოების ტერმინებში.

### 1.11.1. ITILv3 –ის ლექსიკონის განსაზღვრებანი

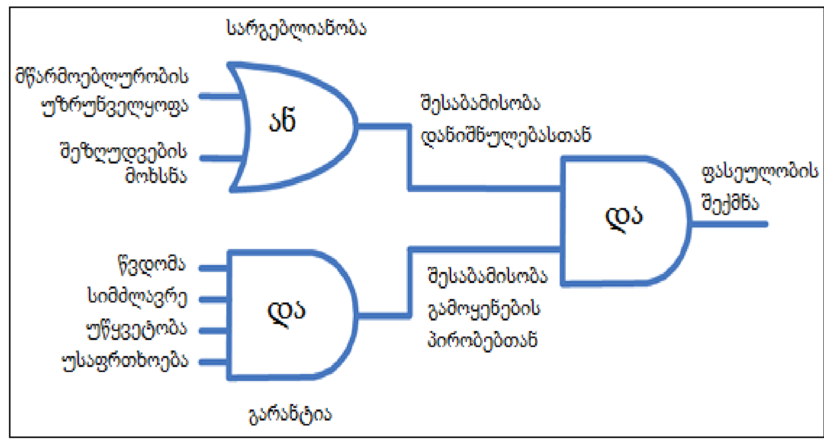
**სარგებლიანობა** – ფუნქციონალობა, რომელსაც იძლევა პროდუქტი ან სერვისი განსაზღვრულ მოთხოვნილებათა უზრუნველსაყოფად. ხშირად განისაზღვრება როგორც „რას აკეთებს პროდუქტი/სერვისი“.

**სერვისის სარგებლიანობა** – IT-მომსახურების ფუნქციობა დამკვეთის თვალსაზრისით.

**გარანტია** – დაპირება ან გარანტია იმისა, რომ პროდუქტი ან სერვისი დააკმაყოფილებს შეთანხმებულ მოთხოვნებს.

**მომსახურების ხარისხის გარანტია** – დარწმუნება იმაში, რომ IT–სერვისი შესაბამისი იქნება შეთანხმებულ მოთხოვნებთან. შესაძლებელია ფორმალური შეთანხმების არსებობა, ხელშეკრულება ან როგორც მარკეტინგული შეტყობინება.

ამგვარად, სარგებლიანობაა ის, რასაც დამკვეთი იღებს, ხარისხის გარანტიაა ის, თუ როგორ იღებს (ნახ.31).




**ნახ.31. სერვისის ფასეულობის ფორმირების სქემა**

დამკვეთს, შეიძენს რა სერვისს, უნდა შედეგის მიღება მისი გამოყენებით ანუ ფასეულობის ამოღება.

სარგებლიანობა მიიღწევა ერთ–ერთი ხერხით:

1. დამკვეთის მიერ მოთხოვნილი მწარმოებლურობის უზრუნველყოფით;
2. არსებული შეზღუდვების მოცილებით ან შემცირებით.

**მწარმოებლურობა (Performance)** - შეფასებაა იმის, რაც იქნა მიღწეული ან შემუშავებული სისტემის, ადამიანის, გუნდის, პროცესის ან IT– სერვისის მიერ [22].

მწარმოებლურობის ქვეშ იგულისხმება დამკვეთის შესაძლებლობა გააკეთოს მეტი ნაკლებ დროში, ნაკლები დანახარჯებით, ანუ ნაკლები რესურსების გამოყენებით. სხვა სიტყვებით, ესაა სათანადო ოპტიმიზაცია, რომელიც უზრუნველყოფს დამკვეთს გადაწყვიტოს ამოცანა ნაკლები დროისა და ფულის გამოყენებით.

**შეზღუდვა** - ესაა აკრძალვა ან შეუძლებლობა რაღაც ქმედებათა შესასრულებლად.

გარანტია შედგება ოთხი ძირითადი ასპექტისაგან:

- წვდომა;
- სიმძლავრე;
- უსაფრთხოება;
- უწყვეტობა.

სერვისის ხარისხის გარანტიის შეფასება უფრო მარტივია, ვიდრე მისი სარგებლიანობისა ბიზნესისათვის. როცა ადამიანი აჭერს ღილაკს, ის ელოდება, რომ აინთება სინათლე. სამწუხაროდ, IT–სერვისების დროს არც ასე მარტივადაა საქმე. IT–სერვისის გამოყენების შედეგი დამოკიდებულია არა მხოლოდ სერვისის თვისებებზე, არამედ ამ სერვისის მართვაზეც. სწორედ აქ ჩნდება ტერმინი service management.

**IT–სერვისების მართვა** – ესაა სპეციალიზებული ორგანიზაციული შესაძლებლობების ერთობლიობა, დამკვეთისთვის ფასეულობის მისაწოდებლად სერვისის ფორმაში [22]. „სპეციალიზებული შესაძლებლობების“ ქვეშ იგულისხმება პროცესები, მეთოდები, ფუნქციები და როლები, რომელთა გამოყენება შეუძლია მიმწოდებელს, დამკვეთისთვის სერვისის მიწოდების მიზნით. ასევე გამოიყენება აღნიშვნა **ITSM** (IT Service Management), რომელიც ექვივალენტურია „სერვისების მართვის“.

**ხარისხი** - ობიექტის მახასიათებელთა ერთობლიობაა, რომელიც მიეკუთვნება მის შესაძლებლობას, რათა დააკმაყოფილოს დადგენილი და შემოთავაზებული მოთხოვნები.

ორგანიზაციას შეუძლია ძალზე ძვირი IT–სერვისის ყიდვა, მაგრამ თუ მიმწოდებელს არ შეუძლია ხარისხიანი და საპასუხისმგებლო მართვის უზრუნველყოფა – მაშინ ეს შესყიდვა იქნება უაზრო. დამკვეთის დაკმაყოფილება დიდად არის

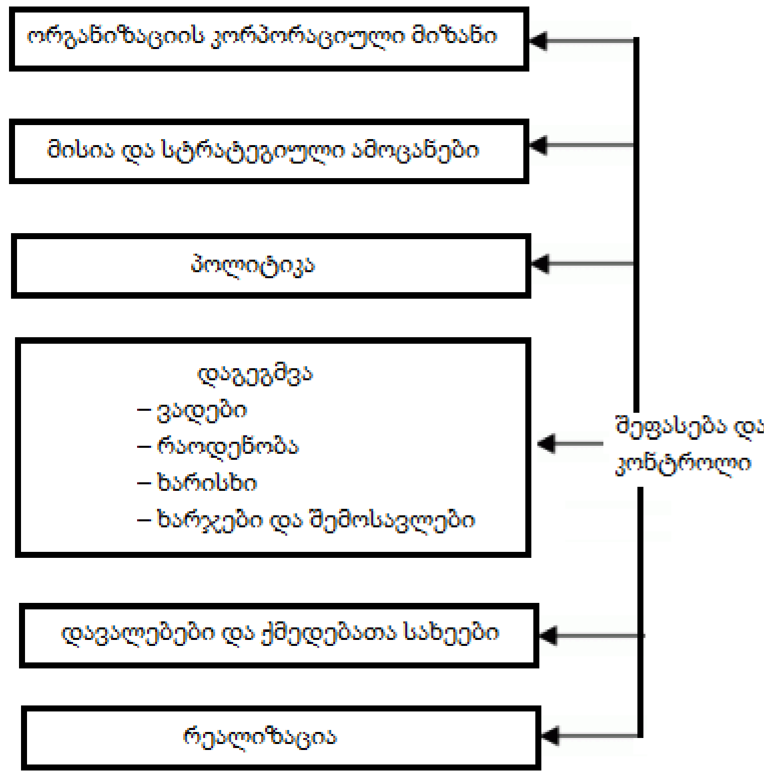
დამოკიდებული იმაზე, თუ რამდენად სწორად იქნა შეთანხმებული სერვისის პარამეტრები წინასწარ სერვისის მიმწოდებელთან.

ამგვარად, სერვის – მენეჯმენტის ძირითადი მიზანი ITIL კონტექსტში არის დამკვეთებისადმი საიმედო, სტაბილური IT–სერვისების მიწოდება, რომლებიც სრულად დააკმაყოფილებს მათ მოთხოვნებს მოცემულ სფეროში. ერთ–ერთი საკვანძო ტერმინი ITIL–ში არის „ორგანიზაცია“. IT–სერვისის დამკვეთი და IT-სევრისის მიმწოდებელი განიხილება როგორც ორგანიზაციები.

**ორგანიზაცია** - ესაა ადამიანთა თანამშრომლობის განსაზღვრული ფორმა. დაისმის კითხვა: რაში მდგომარეობს მიზანი ორგანიზაციად გაერთიანებისა ასეთი კორპორატიული მიზნი (vision) შეიძლება იყოს, მაგალითად, ფულის გამომუშავების სურვილი, პერსონალური კომპიუტერების გაყიდვით ან სერვისის შეთავაზება ინტერნეტში ჩასართავად. იმისათვის, რომ ორგანიზაცია იყოს მიმზიდველი დამკვეთების, ინვესტორებისა და კომპანიის თანამშრომლებისათვის, საჭიროა ინფორმაციის მიწოდება, თუ რა უპირატესობა ექნებათ მათ ამ ორგანიზაციასთან თანამშრომლობისას.

კორპორატიული მიზნის გასაცნობად კომპანიას შეუძლია მისი წარმოდგენა თეზისის სახით თავის მისიაზე (mission) (ნახ.32).

**მისია** - ესაა ამოცანების მოკლე და ცხადად აღწერა, რომლებიც დგას ორგანიზაციის წინაშე და ის იდეალები, რომლებისაც მას (ორგანიზაციას) სწამს.




ნახ.32. ორგანიზაციის კორპორაციული მიზნის ფორმირება

**სტრატეგიული ამოცანები (objectives)** – ესაა სრული აღწერა იმისა, რასაც უნდა მიაღწიოს ორგანიზაციამ გრძელვადიან პერსპექტივით. კარგად ფორმულირებული სტრატეგიული ამოცანები უნდა ფლობდეს ხუთ ძირითად თვისებას (შეესაბამებოდეს SMART პრინციპს):

- იყოს კონკრეტული (Specific);
- ექვემდებარებოდეს შეფასებას (Measurable);

- იყოს სიტუაციისადმი შესაფერისი და შესაბამისი (Appropriate);
- იყოს რეალისტური (Realistic);
- ჰქონდეს მკაფიო დროითი საზღვრები (Time-bound).

**ორგანიზაციის პოლიტიკა (policy)** – ესაა გადაწყვეტილებებისა და ზომების ერთობლიობა, მიღებული ორგანიზაციის მიერ სტრატეგიული ამოცანების დასასმელად და მათ გადასაჭრელად.

ორგანიზაცია, თავისი პოლიტიკის შემუშავების დროს, განსაზღვრავს პრიორიტეტებს, რომლებიც მის წინაშეა სტრატეგიული ამოცანებისა და მათი გადაწყვეტის მიზნით. პრიორიტეტები შეიძლება შეიცვალოს დროის შესაბამისად. ზუსტად ჩამოყალიბებული კომპანიის პოლიტიკა (წესები) ხელს უწყობს ორგანიზაციის სტრუქტურის მოქნილობას, რადგან კომპანიის ყველა დონეზე შესაძლებელია სიტუაციის ცვლილებებზე სწრაფი რეაგირება [22].

პოლიტიკის რეალიზაცია კონკრეტული სახის ქმედებებისათვის მოითხოვს სტრატეგიის შემუშავებას. სტრატეგია მუშავდება განსაზღვრული პერიოდებისთვისა და შედგება რამდენიმე ეტაპისაგან. მნიშვნელოვანია აქ კონტროლის შესაძლებლობა სამუშაოთა შესრულებისას.

არსებობს სხვადასხვა მეთოდი. მაგალითად, ბიზნესში ცნობილია **ბალანსირებულ შეფასებათა რუკა (Balanced Score Card - BSC)**. ამ მეთოდის შესაბამისად, ორგანიზაციის სტრატეგიული მიზნების ან პროცესების მიზნების საფუძველზე განისაზღვრება წარმატების კრიტიკული ფაქტორები (Critical Success Factor - CSF).

**წარმატების კრიტიკული ფაქტორები (Critical Success Factor - CSF) –** ესაა ფაქტორები, რომლებიც აუცილებლად უნდა განხორციელდეს პროექტის, პროცესის, გეგმის ან სერვისის

წარმატებისათვის. ასეთი ფაქტორები ფორმულირდება კომპანიის ინტერესების რამდენიმე უმნიშვნელოვანესი სფეროსთვის, რომელთაც უწოდებენ ორგანიზაციის პერსპექტივებს (პროექციებს): დამკვეთები / ბაზარი, ბიზნეს–პროცესები, პერსონალი / ინოვაციები და ფინანსები. რამდენად წარმატებით რეალიზდება CSFs, გამოიყენებენ KPI–ს.

**მწარმოებლურობის გასაღებური მაჩვენებელი (Key Performance Indicator ან KPI) –** ესაა მეტრიკა, რომელიც გამოიყენება პროცესების, სერვისის ან ქმედებების სამართავად [22]. შესაძლებელია ეფექტურობის მრავალი მაჩვენებლის შეფასება, მაგრამ განსაკუთრებით მნიშვნელოვანია მხოლოდ KPI.

მაგალითად, ფაქტორი „სერვისის დაცვა ცვლილებების რეალიზაციისას“ შეიძლება გაიზომოს ისეთი KPI–ით, როგორიცაა „არაწარმატებული ცვლილებების რაოდენობის შემცირება %–ში“, „პროცენტული შემცირება, ცვლილებათა რაოდენობის, რომელთაც მივყავართ ინციდენტების აღმოცენებამდე“ და ა.შ.

სხვადასხვა გარემოებათა ზემოქმედებისა და ეფექტურობის შეფასების შედეგებისაგან ზემოქმედებით საკონტროლო წერტილებში, სტრატეგიული ამოცანები, მისიები და კორპორა-ციული მიზნები შეიძლება მნიშვნელოვნად შეიცვალოს.

ამასთანავე IT-განყოფილების ან სერვისის მიმწოდებლების სტრატეგიული ამოცანები აგრეთვე უნდა შეიცვალოს ბიზნესის მიზნების მოთხოვნების შესაბამისად.

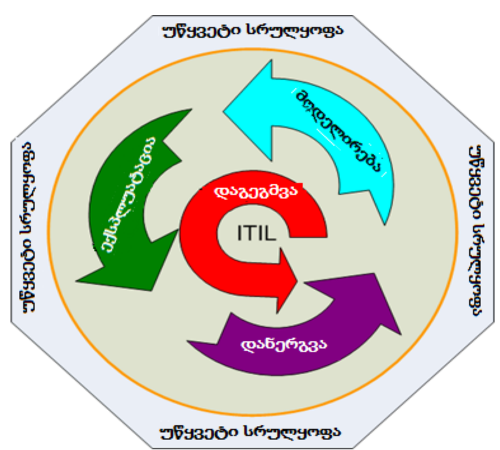
## 1.11.2. IT-სერვისის სასიცოცხლო ციკლი

ITILv3 საფუძველს შეადგენს შემდეგი ექვსი პუბლიკაცია [22]:

1. შესავალი ITIL – ში
2. სერვისის დაგეგმვა (Service Strategy)

3. სერვისის პროექტირება (Service Design)
4. სერვისის დანერგვა (Service Transition)
5. სერვისის ექსპლუატაცია (Service Operation)
6. სერვისის უწყვეტი სრულყოფა (Continual Service Improvement).

ხუთი წიგნი შეესაბამება სერვისების სასიცოცხლო ციკლის ეტაპებს (შესავლის გარდა): ბიზნესის მოთხოვნების პირველადი ანალიზიდან დაწყებული, სტრატეგიის აგებისა და პროექტირების ეტაპებზე, და დამთავრებული სერვისების სრულყოფით ექსპლუატაციის პროცესით. სერვისის სასიცოცხლო ციკლი მოცემულია 33 ნახაზზე.




ნახ.33. სერვისის სასიცოცხლო ციკლი

**სერვისის დაგეგმვა (ან სტრატეგიის აგება)** – ესაა სერვისის სასიცოცხლო ციკლის საფუძველი. მისი შესაბამისი პუბლიკაცია აღნიშნავს სერვის–მენეჯმენტის ცნების ფუნდამენტურობას სერვისის სასიცოცხლო ციკლის კონტექსტში. წიგნში განიხილება შემდეგი საკითხები: IT–სერვისის ბაზრის განვითარება, სერვისების მიმწოდებელთა მახასიათებლები და ტიპები, სერვისის ძირითადი

ხარისხები და რეალიზაციის სტრატეგია სასიცოცხლო პროცესის ციკლში. საკვანძო თემებია ასევე ფინანსური მართვა, მოთხოვნების მართვა, ორგანიზაციული განვითარება და სტრატეგიული რისკები.

მიმწოდებელმა უნდა გამოიყენოს სერვისის დაგეგმვის ეტაპი მიზნების დასასმელად, მომხმარებელთა და გასაღების ბაზრის მოლოდინის (სურვილების) გასარკვევად. სტრატეგიის აგების დანიშნულება, უპირველეს ყოვლისა, არის ის, რომ სერვისების მიმწოდებელმა შეაფასოს საკუთარი შესაძლებლობები და გადაწყვიტოს, შეძლებს თუ არა იგი, განახორციელოს სერვისული პორტფელის მოთხოვნები ყველა ხარჯისა და რისკის გათვალისწინებით.

**სერვისების პორტფელი (ან პორტფოლიო)** - ესაა სერვისების სრული ერთობლიობა, რომელიც წარმოდგინდება სერვისების მიმწოდებლის მიერ. პორტფელი გამოიყენება ყველა სერვისის მართვისათვის სასიცოცხლო ციკლის მთელ მანძილზე. იგი შეიცავს სამ კატეგორიას:

1) სერვისები დამუშავებაშია (Service Pipeline) – სერვისები , რომლებიც დამუშავების სტადიაშია;
2) სერვისების კატალოგი - უკვე გამოყენებადი ან შეთავაზებული სერვისების;
3) სერვისები, რომლებიც ამოღებულია ექსპლუატაციიდან (retired Services).

**სერვისის დაპროექტება.** ყოველი IT-სერვისისათვის ყველაზე მნიშვნელოვანია ბიზნესს წარუდგინოს სათანადო სარგებელი ან ფასეულობა. ამიტომ მიმწოდებელმა უნდა გაითვალისწინოს ბიზნესის მიზნები.

პუბლიკაცია „სერვისების დაპროექტება“ არის სახელმძღვანელო სერვისების მოდელირებისა და სრულყოფისათვის, ასევე რეკომენდაციებისათვის მათ სამართავად პრაქტიკაში. ამ ეტაპზე

აღიწერება ძირითადი პრინციპები და მოდელირების მეთოდები სტრატეგიული მიზნების გარდაქმნისათვის განსაზღვრული ხარისხის კონკრეტული სერვისების ერთობლიობაში. იგი მოიცავს ასევე ახალი სერვისების შექმნის, არსებულის ცვლილებისა და სრულყოფის საკითხებს სასიცოცხლო ციკლის ფარგლებში, რაც აუცილებელია მის ფასეულობათა ასამაღლებლად მომხმარებელთა თვალსაზრისით.

ასევე წიგნის საკვანძო თემებია სერვისების კატალოგი, სარგებლიანობა, მწარმოებლურობა და სერვისის უწყვეტობა, სერვისების მართვის დონე, რომლებიც შემდგომ განიხილება.

**სერვისის დანერგვა. Transition** - გადაადგილება, გადასვლა ან ერთი მდგომარეობის შეცვლა მეორით (პოზიციის, პერიოდის, სტადიის, თემისა და სხვ.). მისი შესაბამისი პუბლიკაცია ITIL ბიბლიოთეკაში არის სახელმძღვანელო იმაზე, თუ ეფექტურად როგორ რეალიზდეს მოთხოვნები, რომლებიც ფორმულირებულ იქნა პროექტირებისა და სტრატეგიის აგების სტადიებზე, ექსპლუატაციის ეტაპზე რისკების, მტყუნებებისა და გაუმართაობების კონტროლით. განიხილება რისკების მართვის საკითხებიც.

**სერვისის ექსპლუატაცია** ახორციელებს სერვისის ბიზნეს–მნიშვნელობის „მიტანის“ ეტაპს მიმწოდებლიდან დამკვეთამდე. აქ მნიშვნელოვანია სერვისის მიწოდების ეფექტურობა და მისი ხარისხიანი თანხლება. წიგნი აღწერს თუ როგორ შეიძლება განხორციელდეს სერვისის სტაბილური ექსპლუატაცია, ცვლილების განხორციელების შესაძლებლობასთან ერთად დიზაინში, მასშტაბში, საზღვრებში და ა.შ. ორგანიზაციებს მიეცემათ ინსტრუქციები, მეთოდები და ინსტრუმენტები კონტროლის ორი მეთოდის სარეალიზაციოდ – პრევენციული (პროფილაქტიკური) და პროაქტიური.

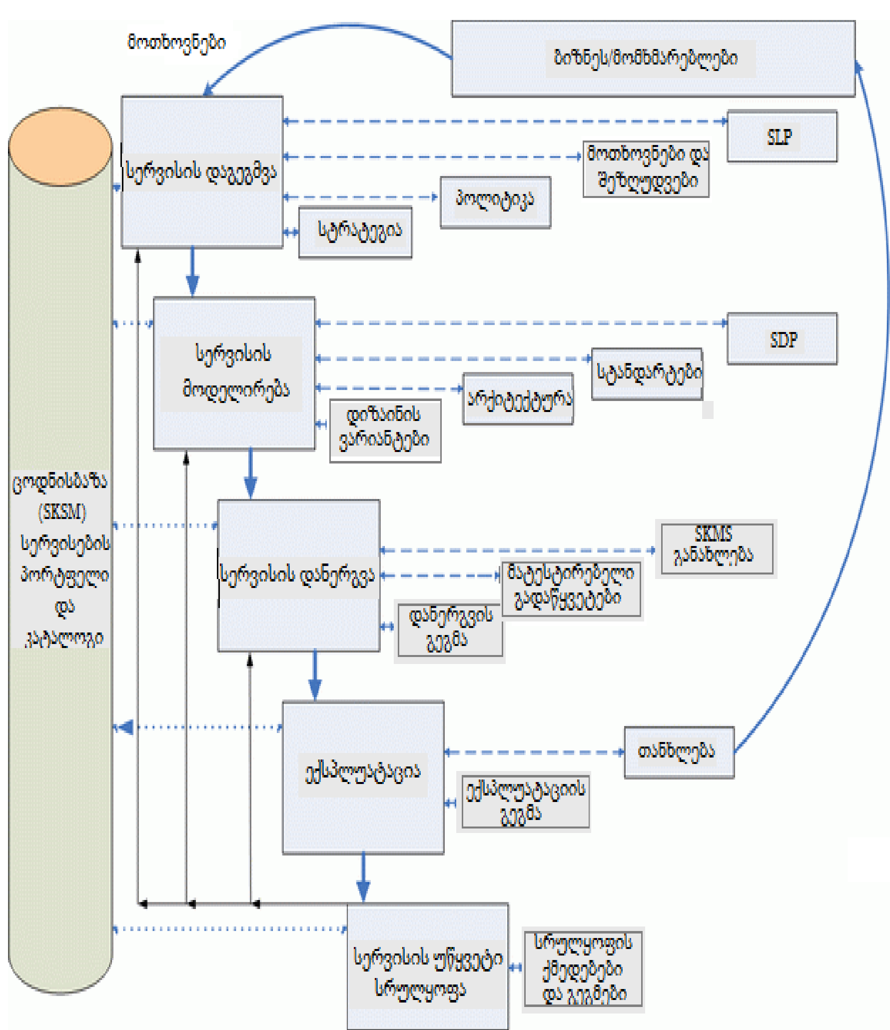
წიგნში მოცემული ინფორმაცია სასარგებლო იქნება გადაწყვეტილების მისაღებად სერვისის წვდომის მართვის საკითხებში, სერვისზე მოთხოვნილების კონტროლისთვის, დატვირთვის ოპტიმიზაციისა და მიმდინარე პრობლემების გადასაწყვეტად.

აღწერილი ყველა ხერხი ითვალისწინებს ახალი მოდელებისა და არქიტექტურის შესაძლებლობებს, როგორიცაა განაწილებული სერვისები, გაანგარიშებები სქემით „კომუნალური სერვისი" (utility computing), ვებ-სერვისი და ელ-კომერცია.

სიტყვა utility computing აღწერს ახალ შემოტანილ ბიზნეს–მოდელს, როცა სერვისების მიმწოდებელი იღებს ფულს სერვისის გამოყენების ფაქტზე, მაგალითად, მისი გამოყენების დროის მიხედვით. ტრადიციულ ბიზნეს–მოდელში კი მომხმარებელი იხდის სისტემის (სერვისის) ფლობისათვის.

ასეთი სერვისების პროვაიდერს შეუძლია თავისი რესურსების გამოყენების ოპტიმიზაცია, მომხმარებელთა განსხვავებული საჭიროების გათვალისწინებით.

**სერვისის უწყვეტი სრულყოფა** მდგომარეობს სერვისის ფასეულობის ამაღლების მეთოდებისა და საშუალებების აღწერაში სასიცოცხლო ციკლის სხვადასხვა ეტაპზე სრულყოფის რეალიზაციის გზით. ეს ეტაპი აერთიანებს თავის თავში ხარისხის, ცვლილებებისა და მწარმოებლურობის სრულყოფის მართვის პრინციპებს, პრაქტიკებსა და მეთოდებს. წიგნიდან ორგანიზაციებმა შეიძლება მიიღოს რეკომენდაციები იმის შესახებ. თუ ეტაპობრივად როგორ განახორციელონ მსხვილმასშტაბური სრულყოფები სერვისების ხარისხში, ექსპლუატაციის ეფექტურობასა და სერვისების მიწოდების უწყვეტობაში. სახელმძღვანელო დანიშნულია სრულყოფის შედეგების უკუკავშირის უზრუნველსაყოფად დაგეგმვის, მოდელირებისა და გარდაქმნების ეტაპებთან.




ნახ.34. სერვისის სასიცოცხლო ციკლის ეტაპების ძირითადი კავშირები, შესასვლელები და გამოსასვლელები

34-ე ნახაზი გვიჩვენებს, თუ როგორაა დამოკიდებული სერვისის სასიცოცხლო ციკლის ეტაპები ბიზნესის მოთხოვნილებათა ცვლილებებზე.

მოთხოვნილებები იქმნება სერვისის დაგეგმვის ეტაპზე **სერვისების დონეების პაკეტი**ს ჩარჩოში **(Service Level Package ან SLP).** ესაა სარგებლიანობის განსაზღვრული დონე და გარანტიები ცალკეული სერვისების პაკეტისათვის. ყოველი SLP მუშავდება ცალკეული პროფილის ბიზნეს-ქმედების მოთხოვნილებათა სარეალიზაციოდ. ეს პროცესი გადადის სერვისის დაპროექტებაში, სადაც გადაწყვეტილებები, მიღებული პირველ ეტაპზე, გროვდება ერთად და რეალიზდება **სერვისის საპროექტო დოკუმენტაციის** სახით **(Service Design Package ან SDP)**. ესაა დოკუმენტები, რომლებიც განსაზღვრავს სერვისის ყველა ასპექტსა და მისდამი მოთხოვნებს სასიცოცხლო ციკლის ყოველ ეტაპზე [22].

ფაქტობრივად ესაა საპროექტო დოკუმენტაცია, რომელიც მუშავდება ახალი სერვისისთვის, მნიშვნელოვანი ცვლილებების შესატანად ან სერვისის ექსპლუატაციიდან მოხსნის დროს.

SDP გადადის დანერგვის ეტაპზე, რომელზეც ხდება სერვისის ტესტირება, გადის შეფასებასა და ვალიდაციას. შედეგად განახლდება სერვისების ცოდნის მართვის სისტემა და სერვისი გადადის ექსპლუატაციის სტადიაზე.

## 1.11.3. სერვისების ცოდნის ბაზის მართვის სისტემა

**სერვისების ცოდნის ბაზის მართვის სისტემა (Service Knowledge Management System ან SKMS)** - ესაა ინსტრუმენტებისა და მონაცემთა ბაზების ერთობლიობა, რომლებიც გამოიყენება ცოდნისა და სერვისების შესახებ ინფორმაციის სისტემატიზაციისათვის. იგი ინახავს, მართვას, განაახლებსა და წარმოადგენს მთელ ინფორმაციას, რომელიც საჭიროა მიმწოდებლისათვის სერვისის მართვისათვის სასიცოცხლო

ციკლის ყველა ეტაპზე. ბუნებრივია, რომ სასიცოცხლო ციკლის მთელ მანძილზე სერვისი უნდა გაუმჯობესდეს, ამის აუცილებლობისა და შესაბამისი შესაძლებლობების დროს.

IT–ინფრასტრუქტურის მართვის წარმოდგენა პროცესების კომპლექსის სახით საშუალებას იძლევა უნიფიცირებულ იქნას სერვისების მიმწოდებლებისა და დამკვეთების მრავალი ასპექტი. ყოველი პროცესისთვის განისაზღრება როლები, მიზნები, ამოცანები, მეთოდები და საშუალებები, აგრეთვე შემავალ–გამომავალი ინფორმაცია.

ამგვარად, ITILv3–ის ძირითადი დანიშნულებაა IT–სერვისების ხარისხიანი წარმოდგენა და მხარდაჭერა ბიზნესის მოთხოვნების შესაბამისად. პრინციპული განსხვავება მე–3 ვერსიისა მე–2 ვერსიისაგან ისაა, რომ შეიქმნა სერვისების მართვის პრინციპების აღწერის მიდგომა.

სერვისების მართვის პროცესების დაჯგუფებასთან ერთად სასიცოცხლო ციკლის ცალკეული პერიოდების მიხედვით, ITILv3 გვთავაზობს IT–სერვისზე ვისაუბროთ ბიზნესისთვის დამატებითი ფასეულობის შეთავაზების კონტექსტში.

ITILv2–ში IT–სამსახური სერვისს სთავაზობდა ბიზნესს თავისი არსებული ინფრასტრუქტურის ბაზაზე, ცდილობდა რა ბიზნესისთვის გასაგებ ტერმინებში ჩამოეყალიბებინა მომსახურების (სერვისს) ხარისხის მახასიათებლები.

ITILv3 გვთავაზობს სრულიად ახალ მიდგომას. IT–სამსახური აანალიზებს ბიზნესის მიზნებსა და ამოცანებს და, აქედან გამომდინარე, გვთავაზობს სერვისებს, რომლებიც ნამდვილად სჭირდება ბიზნესს ახლა.

ITIL–ის გამოყენება არაა სავალდებულო, როგორც ეს მაგალითად, უსაფრთხოების ან აუდიტის სტანდარტებითაა გათვალისწინებული, მაგრამ ITIL ბოლო ათი წლის მანძილზე

მთელ მსოფლიოში გახდა დე–ფაქტო სტანდარტი IT–სერვისების მართვის სფეროში.

### 1.11.4. სტრატეგიის აგება – სერვისების სასიცოცხლო ციკლის ეტაპი

სერვისს თანამედროვე მსხვილი მიმწოდებლები მსგავსი მახასიათებლებით და შესაძლებლობებით ხასიათდება. მათ შორის მთავარი განმასხვავებელი თავისებურება სტრატეგიაა, რომელსაც კონკრეტული მიმწოდებელი იყენებს სერვისისათვის.

სტრატეგიის აგების დროს სერვისის მიმწოდებელი ორიენტირებულ უნდა იყოს, უპირველეს ყოვლისა, თავისი პოტენციალური დამკვეთის მიზნებზე. ამიტომ ცხადად უნდა ესმოდეს, თუ რა როლი უნდა შეასრულოს მიწოდებულმა IT-სერვისმა დამკვეთის ბიზნესში.

IT-სფეროს სწრაფი განვითარება უკვე დღეს მოითხოვს მიმწოდებლებისაგან არა მხოლოდ დამკვეთების მოთხოვნებზე ოპერატიულ რეაგირებას, არამედ იმის ცოდნასაც, თუ მომავალში რა დასჭირდება დამკვეთს. ამიტომაც სტრატეგიის აგება არის ფუნდამენტური ეტაპი სერვისის სასიცოცხლო ციკლში. მიმწოდებელს უნდა ესმოდეს, რომ დამკვეთი მისაგან ყიდულობს არა კონკრეტულ პროდუქტს, არამედ საშუალებებს თავიანთი ბიზნეს–მოთხოვნების დასაკმაყოფილებლად.

სტრატეგიის ასაგებად მიმწოდებელმა უნდა გაითვალის-წინოს ფაქტორების სიმრავლე, რომელთაგან ძირითადია:

1. ყველაფერი, რაც IT-სერვისების ირგვლივაა, რთულია: ეს ეხება არა მხოლოდ კონკრეტულ სერვისების ინდივიდუალურ თავისებურებებს, არამედ იმ სირთულეს, რომლებიც აღმოცენდება IT-სფეროში ცვალებადი და ურთიერთდამოუკიდებული ფაქტორების სიმრავლის შედეგად. საჭიროა განვასხვავოთ მოკლევადიანი და გრძელვადიანი დაგეგმვა, რადგან ბაზრის,

მომხმარებელთა და თვით IT-სფეროს ქცევები განსხვავებულია განსახილველ პერიოდთან მიმართებაში. პირველი რიგის ამოცანად განიხილება მეთოდების შემუშავება, რომლებიც დაეხმარება ორგანიზაციებს გადაწყვეტილების მისაღებად და შემდგომი ქმედებების სტრატეგიის განსაზღვრაში;

2. დამკვეთების მოთხოვნები ყოველთვის არაა ცხადი, გასაგები და კორექტულიც კი. მრავალი მათგანი იკარგება საპროექტო დოკუმენტაციიდან სერვისის რეალიზაციაზე გადასვლის პროცესში. სტრატეგიული აზროვნების ყველაზე მნიშვნელოვანი ასპექტია იმის გაცნობიერება, თუ რა უნდა იქნას მიღებულის შედეგად. ის, რასაც დამკვეთი იღებს თავისი სერვისის ტექნიკური მოთხოვნების სანაცვლოდ, არის საფუძველი მისი დაგეგმვის. დამკვეთთა მოთხოვნებისა და მიზნების გაგება გვთავაზობს არა მხოლოდ ცოდნას, თუ როდისა და რატომ წარმოიშვა კონკრეტული მოთხოვნები, არამედ ცხადად გაცნობიერებასაც, თუ ვინაა IT–სერვისის საბოლოო მომხმარებელი;

3. კონტექსტისაგან დამოუკიდებლად, რომელშიც მუშაობს მიმწოდებელი, სტრატეგიის აგების დროს მან უნდა გაითვალისწინოს კონკურენციის არსებობა. სახელმწიფო და კერძო IT–ორგანიზაციები მონაწილეობენ კონკურენციაში. სერვისების მიმწოდებლისთვის აუცილებელია ცოდნა თუ რა მდგომარეობა უჭირავს მას ამ ბაზარზე, და მისი სერვისები რითი განსხვავდება კონკურენტების ანალოგური სერვისებისაგან.

სერვისების დაგეგმვა, როგორც სასიცოცხლო ციკლის ეტაპი, საშუალებას აძლევს მიმწოდებელს გაერკვეს შემდეგ საკითხებში:

1. რომელი სერვისების შეთავაზება ღირს?
2. ვის უნდა შევთავაზოთ სერვისები?
3. რა სარგებელს (შედეგს) მიიღებენ მომხმარებლები სერვისი გამოყენებით?

4. რა სარგებელს (შედეგს) მიიღებენ ინვესტორები სერვისი გამოყენებით?

5. როგორ განვითარდეს შეგა და გარე გასაღების ბაზრები?

6. როგორ განისაზღვროს სერვისის ხარისხი?

7. როგორ იღებენ გადაწყვეტილებას დამკვეთები სერვისების მიმწოდებლების ამორჩევისას კონკურენციის პირობებში?

8. როგორ გაკონტროლდეს სერვისის ფასეულობის შექმნა ფინანსური მართვის ტერმინებში?

9. როგორ განაწილდეს არსებული რესურსები დასახული მიზნების უფრო ეფექტურად მისაღწევად?

მომხმარებლები აფასებენ IT–სერვისის გამოყენების შედეგებს ყველაზე ხშირად ეკონომიკური ტერმინებით. IT–ორგანიზაციისთვის აუცილებელია ფიქრი როგორც ინვესტიციებზე სერვისების განვითარების მიზნით, ასევე ბიზნესზე მათი დანერგვის გზით.

სერვისისთვის მნიშვნელოვანია საბაზრო ადეკვატური ფასი, მწარმოებლურობა, სტაბილურობა (დამკვეთი ითვალისწინებს ამათ). სერვის–მენეჯმენტის წარმატება დამოკიდებულია, უპირველეს ყოვლისა, დამკვეთისა და მწარმოებლის ურთიერგაგებაზე. ამიტომაც, წარმატების მიღწევის მიზნით (სერვისების აგებისათვის), არის შემოთავაზებული ITIL პუბლიკაციები [11,22].

## 1.12. COBIT საერთაშორისო სტანდარტული მეთოდოლოგია

ინფორმაციული ტექნოლოგიების მენეჯმენტის საერთაშორისო სტანდარტებისა და მეთოდოლოგიების ერთ-ერთი აქტუალური წარმომადგენელია COBIT (Control Objectives for Information and Related Technology - საკონტროლო ობიექტები საინფორმაციო და მასთან დაკავშირებული

ტექნილოგიებისათვის), რომელიც შეიქმნა ISACA (Information Systems Audit and Control Association - საინფორმაციო სისტემების აუდიტისა და კონტროლის ასოციაცია) ორგანიზაციის მიერ ამერიკის შეერთებულ შტატებში 1969 წელს, საფინანსო აუდიტებისთვის ინფორმაციული ტექნოლოგიების კონტროლის მიზნით. ამჟამად ამ ორგანიზაციას აქვს მსოფლიოში ერთ-ერთი ლიდერის როლი ინფორმაციული ტექნოლოგიების აუდიტის სტანდარტების შემუშავების სფეროში [33,34].

COBIT არის ღია დოკუმენტების ერთობლიობა, 40-მდე საერთაშორისო სტანდარტი და სახელმძღვანელო IT-მართვის, აუდიტისა და ინფორმაციული უსაფრთხოების სფეროებში. ესაა ავტორიტეტული, თანამედროვე, საერთაშორისო აღიარებული მეთოდოლოგიის კვლევა, დამუშავება, პუბლიკაცია კორპორაციული მენეჯმენტისათვის IT-სფეროში. მისი დანიშნულებაა ორგანიზაციებში ამ სტანდარტების დანერგვა და ყოველდღიური გამოყენება IT- მენეჯერებისა და აუდიტორების მიერ [35].

COBIT-ის ძირითადი მიზანია ინფორმაციული ტექნოლოგიების მენეჯმენტი. ამავდროულად, ინფორმაციული ტექნოლოგიების მენეჯმენტი, თავის მხრივ, არის კორპორაციული მენეჯმენტის განუყოფელი ნაწილი. კორპორაციული მენეჯმენტი - მმართველობითი გადაწყვეტილებისა და მეთოდების კომპლექსია, რომელიც გამოიყენება უმაღლესი ხელმძღვანელობის მიერ შემდეგი მიზნებისათვის:

- სტრატეგიული მიმართულების განსაზღვრისათვის;
- მიზნების მიღწევის უზრუნველსაყოფად;
- რისკების ადეკვატურად სამართავად;
- კორპორაციული რესურსების ეფექტურად გამოსაყენებლად.

კორპორაციული მენეჯმენტი და IT-მენეჯმენტი მოითხოვს მიზნებს შორის ბალანსს, რაც დაკავშირებულია ზემდგომი

ხელმძღვანელობის მიერ დადგენილი მოთხოვნილებების შესაბამისობის აუცილებლობასა და ეფექტიანობის ამაღლებასთან.

COBIT-ში გამოიყენება ტერმინი „დაინტერესებული მხარეები“ (Stakeholders), რომლებსაც მიეკუთვნება:

- დირექტორთა საბჭო და უმაღლესი ხელმძღვანელობა: IT-ის განვითარების მიმართულების განსაზღვრა, შედეგების შეფასება და ნაკლოვანებათა აღმოფხვრის მოთხოვნების დადგენა;
- განყოფილებების ხელმძღვანელები: ბიზნეს-მოთხოვნების განსაზღვრა IT-ის მიმართ, სარგებლიანობის მიღწევის უზრუნველყოფა IT-დან და რისკების მართვა;
- IT-სამსახურის ხელმძღვანელობა: IT- სერვისებით უზრუნველყოფა და მათი სრულყოფა ბიზნესის მოთხოვნილებათა შესაბამისად;
- შიგა აუდიტი / შიგა კონტროლის სამსახური / IT-აუდიტი: დამოუკიდებელი შეფასების უზრუნველყოფა, რომ IT იძლევა საჭირო სერვისებს;
- რისკების მართვა და შესაბამისობის დაცვა: ნორმატიულ დოკუმენტებთან შესაბამისობის შეფასება რისკების გათვალის-წინებით.

### ➢ COBIT-ის მიზნები და პრინციპები:

COBIT-ის საკვანძო ცნებაა სერვისი ან მომსახურება (service). მაგალითად, ინტერნეტში წვდომის ან დაცულ მონაცემთა საცავთან მიმართვის უზრუნველყოფა მიეკუთვნება მომსახურების სახეებს. ჩვენ სერვისის განსაზღვრა შემოვიტანეთ ITIL მეთოდოლოგიის განხილვისას, რომელიც ასევე სერვისების მენეჯმენტს ეხება. სერვისი არის სათანადო ფასეულობის მიწოდების ხერხი დამკვეთზე, რომელიც მას ხელს უწყობს სასურველი შედეგების მისაღებად თავისი სისტემის გამოსასვლელზე, ყოველგვარი სპეციფიური ხარჯებისა და რისკების გარეშე. სერვისების მიწოდება რთული და

არატრივიალური ამოცანაა, რომელიც, პირველ რიგში, მოითხოვს შიგა კონტროლის სისტემას.

COBIT-ის ძირითადი პრინციპებია:

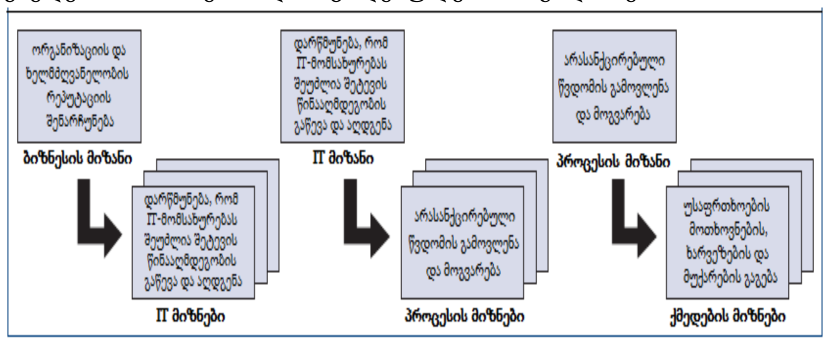
- IT მიზნები უნდა შეესაბამებოდეს ბიზნესის მიზნებს;
- პროცესული მიდგომის გამოყენება;
- IT კონტროლის სისტემა უნდა იყოს შერჩევითი, ანუ განსაზღვროს IT ძირითადი რესურსები და იმუშაოს მასთან;
- კონტროლის მიზნები უნდა იყოს მკაფიოდ განსაზღვრული.

სერვისების მართვის თანამედროვე მიდგომა ყურადღებას ამახვილებს ბიზნესისა და IT-ს ურთიერთქმედებაზე.

მიზნები განსაზღვრულია დაღმავალად (top-down ზემოდან-ქვევით) ისე, რომ ბიზნეს-მიზანმა უნდა განსაზღვროს IT-მიზნები თავის მხარდასაჭერად. IT-მიზანი მიიღწევა ერთი პროცესის ან რამდენიმე პროცესის ურთიერთმოქმედებით. ამგვარად, IT-ის მიზანია განსაზღვროს განსხვავებული პროცესების მიზნები.

თავის მხრივ, თითოეული პროცესის მიზანი მოითხოვს აქტიურობათა (ქმედებთა) გარკვეულ რაოდენობას, ასევე მათი მიზნების დადგენას.

35-ე ნახაზზე მოცემულია ბიზნესის, IT-ის, პროცესებისა და ქმედებათა მიზნების დამოკიდებულების მაგალითები.




**ნახ.35**

განვიხილოთ დეტალურად COBIT-ის ძირითადი პრინციპები:

1. **ბიზნესისა და IT-ის მიზნები უნდა იყოს ურთიერთდაკავშირებული, მაგრამ განმსაზღვრელი ამ ურთიერთობაში არის ბიზნესის მიზნები.** კომპანიის მოგება პირდაპირ დამოკიდებულებაშია IT-ის ეფექტურ გამოყენებასთან. ამიტომ ხელმძღვანელობამ მეტი ყურადღება უნდა გაამახვილოს მის სარგებლიანობაზე, ინვესტირებაზე, შედეგების მონიტორინგსა და შეფასებაზე;

2. **პროცესული მიდგომის გამოყენება**. პროცესი არის საქმიანობათა სახეების სტრუქტურირებული ერთობლიობა, რომელიც დაპროექტებულია განსაზღვრული მიზნის მისაღწევად ანუ პროცესი, ზოგადად, პროცედურების ერთობლიობაა, რომელზეც გავლენას ახდენს ორგანიზაციის პოლიტიკა და სხვა წყაროების პროცესები. ბიზნესი განაპირობებს პროცესის წარმოქმნის მიზეზს, მის პასუხისმგებელ მფლობელს, თანამდებობრივ მოვალეობებს, რომლებიც დაკავშირებულია პროცესის შევსების, შესრულებისა და ეფექტიანობის გაზომვის საშუალებებთან.

## COBIT-ის პროცესები:

პროცესებს აქვს შემდეგი მახასიათებლები:

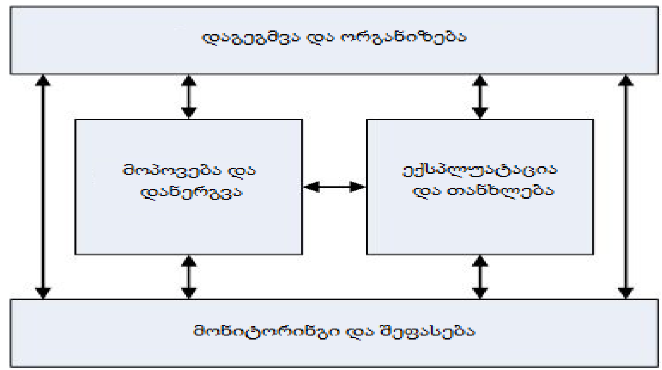
- **პროცესები გაზომვადია** ანუ ისინი შეიძლება შეფასდეს რომელიმე შესაბამისი მეთოდით. მაგალითად, მენეჯერები იყენებენ პროცესების ღირებულებასა და ხარისხს, მომხმარებლები კი - პროცესების ხანგრძლივობასა და პროდუქტიულობას;
- **პროცესები ემსახურება კონკრეტული შედეგების მიღწევას.** პროცესის არსებობის მიზეზი არის კონკრეტული შედეგის მიღება,

რომელიც შეიძლება იდენტიფიცირდეს (გამოვლინდეს) და რაოდენობრივად შეფასდეს (დათვლილ იქნეს);

- **პროცესებს ჰყავს მომხმარებლები.** ყოველი პროცესი თავის შედეგებს აწვდის მომხმარებლებს ან სხვა პროცესებს, ორგანიზაციის შიგნით ან გარეთ;
- **პროცესები შედგება ქმედებებისაგან.** ქმედება (Activity) არის საქმიანობის ძირითადი სახეები პროცესის ფარგლებში.

COBIT განიხილავს 34 IT-პროცესს, რომლებიც გაერთიანებულია 4 დომენში (Domain - საკონტროლო მიზნების დაჯგუფება ლოგიკურ ეტაპებში IT-ინვესტიციის სასიცოცხლო ციკლის შიგნით). 36-ე ნახაზზე მოცემულია დომენების ურთიერთკავშირის სქემა.

- **დაგეგმვა და ორგანიზება (PO - Plan and Organise):** განსაზღვრავს მიმართულებებს გადაწყვეტილებათა დანერგვის (AI) და სერვისების მიწოდების (DS - Delivery Services) თვალსაზრისით;
- **მოპოვება და დანერგვა (AI - Acquire and Implement)**: უზრუნველყოფს გადაწყვეტილებათა დანერგვასა და სერვისებს მათ საფუძველზე;




**ნახ.36**

- **ექსპლუატაცია და თანხლება (DS - Deliver and Support)**. უზრუნველყოფს გადაწყვეტილებათა შესრულებასა და საბოლოო მომხმარებლებისათვის მათი გამოყენების მხარდაჭერას;
- **მონიტორინგი და შეფასება (ME - Monitor and Evaluate).** ახორციელებს პროცესების მონიტორინგსა (კონტროლს) და შეფასებას.

პროცესების ასეთი სტრუქტურა იძლევა სფეროების სისტემატიზაციისა და ინფორმაციის ორგანიზების უზრუნველყოფის საშუალებას, რომლებიც აუცილებელია მათი ბიზნეს-მიზნების მისაღწევად.

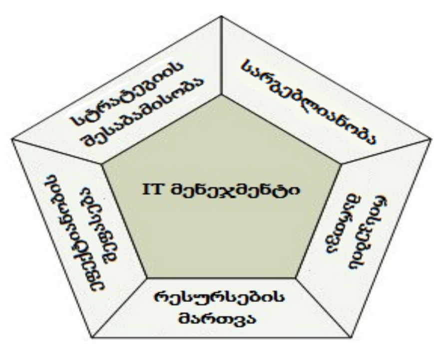
3. **რესურსების რანჟირების პრინციპი.** არაა აუცილებელი ყველა რესურსის თვალყური, მხოლოდ იმათზე უნდა გამახვილდეს ყურადღება, რომლებიც გავლენას ახდენს ბიზნეს-პროცესებს და მათ შედეგებზე.

4. **მიზნების განსაზღვრა** - ერთ-ერთი ყველაზე რთული და მნიშვნელოვანი ამოცანაა. გლობალური გაგებით ხელმძღვანელობა და მენეჯერები ორ მიზანს ითვალისწინებს - დასმული ბიზნეს-მიზნების მიღწევასა და არასასურველი მოვლენების თავიდან აცილება (ან მათი შედეგების გამოსწორება).

მაგალითად, სარეზერვო კოპირების ამოცანას არ მოაქვს პირდაპირი მოგება ბიზნესისათვის, მაგრამ სისტემის მწყობრიდან გამოსვლის შემთხვევაში მისი საშუალებით შესაძლებელია ინფორმაციის სწრაფი აღდგენა, რაც მეტად მნიშვნელოვანია ბიზნესის ნორმალური ფუნქციონირებისათვის.

ასევე მნიშვნელოვანია საკითხები ხელმძღვანელობისთვის, მაგალითად, სადაა საჭირო პროცესების სრულყოფა, რამდენი ინვესტიციაა საჭირო და როგორ გაიზომოს შედეგი. COBIT იძლევა მეთოდლოგიას IT-ხელმძღვანელობისათვის დასმულ საკითხებზე.

COBIT გამოყოფს IT-მენეჯმენტის შემდეგ საკვანძო სფეროებს (ნახ.37):




ნახ.37. IT-მენეჯმენტის საკვანძო სფეროები

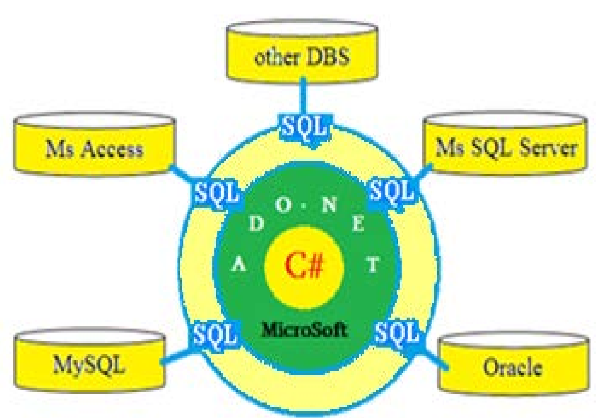
- **სტრატეგიის შესაბამისობა**. უზრუნველყოფს ერთმანეთთან ბიზნესისა და IT-ის თავსებადობას;
- **სარგებლიანობა** პასუხს აგებს: იმის რეალიზაციაზე, რასაც შეუძლია ბიზნესისათვის ფასეულობის მოტანა; კონტროლზე, რათა IT-იმ უზრუნველყოს სტრატეგიით განსაზღვრული უპირატესობები; ხარჯების ოპტიმიზაციასა და ჭეშმარიტი ღირებულების დადასტურებაზე;
- **რესურსების მართვა** პასუხისმგებელია კრიტიკული IT-რესურსების მენეჯმენტზე, ინვესტიციების ოპტიმიზაციაზე და აპლიკაციების, ინფორმაციის, ინფრასტრუქტურისა და პერსონალის სათანადო ხელმძღვანელობაზე;
- **რისკების მართვა** მოითხოვს ზემდგომი ხელმძღვანელობის ინფორმირებას რისკების სფეროში; კორპორაციული მიდგომის ნათლად წარმოდგენას მათთან მიმართებით; გამჭვირვალობის მოთხოვნების შესაბამისობას არსებულ რისკებთან დამოკიდებულებით; ორგანიზაციის პრაქტიკაში რისკების მართვის ფუნქციების დანერგვას;

**ეფექტიანობის შეფასება** პასუხს აგებს სტრატეგიის, გეგმების, რესურსების გამოყენებისა და პროცესების ეფექტიანობის რეალიზაციის კონტროლზე;

## 1.13. ADO.NET ტექნოლოგია: მოქნილი კავშირი მომხმარებლის ინტერფეისსა და მონაცემთა ბაზებს შორისა

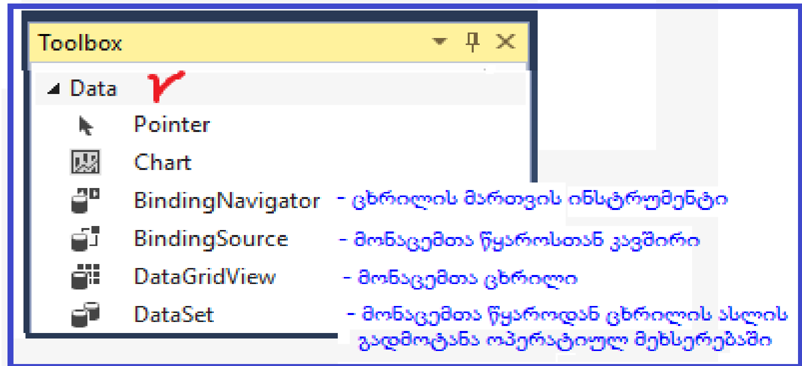
მომხმარებელთა პროგრამულ აპლიკაციებს (დანართებს) აუცილებლად ესაჭიროება ურთიერთქმედება მონაცემთა ცენტრალიზებულ ბაზებთან, მონაცემთა საცავებთან XML–ფორმატით (Extensible Markup Language), ან მონაცემთა ლოკალურ ბაზებთან, როდესაც ისინი მუშაობენ კლიენტის მანქანებზე.

Microsoft Visual Studio .NET Framework 4.0/4.5 პლატფორმა, რომელსაც ჩვენ ვიხილავთ C#-ენის საფუძველზე, მონაცემთა ბაზებთან სამუშაოდ იყენებს ADO.NET ტექნოლოგიასა და SQL ენას (ნახ.38). პირველი დამაკავშირებელი დრაივერია C#-ენასა და ბაზებს შორის, მეორე კი - მომხმარებლის საკონტაქტო ენაა ბაზებთან, ე.წ. სტრუქტურირებულ მოთხოვნათა ენა [1,14].




**ნახ.38. C# <-> ADO.NET <-> SQL <–> DBS**

C# ენა .NET გარემოში მონაცემთა ბაზებთან სამუშაოდ გვთავაზობს შემდეგ კომპონენტებს (ნახ.39), რომელთა საფუძველია ADO.NET.




ნახ.39

ADO.NET - მაიკროსოფთის ტექნოლოგიაა (ActiveX Data Object), რომელიც გვთავაზობს მონაცემებთან მიმართვისათვის გამოსაყენებლად მარტივ, მაგრამ მეტად მძლავრ საშუალებებს. იგი უზრუნველყოფს სისტემის რესურსების მაქსიმალურად სრულ ურთიერთქმედებას. წინამდებარე პარაგრაფში გავეცნობით:

- ADO.NET დრაივერის მონაცემებთან მიმართვის ძირითად კომპონენტებს;
- თითოეული კომპონენტის როლის ფუნქციობას;
- ADO.NET-ის მონაცემებთან მიმართვის ორგანიზაციის სცენარის აღწერას.
- C# პროგრამული აპლიკაციის კავშირს მონაცემთა რელაციური ბაზების მართვის სისტემებთან: Ms Access, MySQL, Ms SQL Server.

სხვადასხვა დანართი მონაცემებთან მიმართვის ორგანიზაციისათვის აყენებს სხვადასხვა მოთხოვნას. მნიშვნელობა არა აქვს იმას, თუ რას აკეთებს დანართი: ასახავს ცხრილების შინაარსს, თუ გადაამუშავებსა და განაახლებს მონაცემებს ცენტრალურ SQL-სერვერზე. რეალიზაციის სხვადასხვა სცენარით ADO.NET აძლევს მომხმარებელს მონაცემებთან მიმართვის მარტივ და ეფექტურ საშუალებებს.

### 1.13.1. გამოყოფილ მონაცემებთან მიმართვა

მონაცემებთან მიმართვის ტრადიციული ტექნოლოგიები, ჩვეულებრივად, მუდმივი მიერთების გზით ახორციელებდა მონაცემების წვდომას წყაროსთან.

ასეთი მოდელის გამოყენებისას პროგრამული აპლიკაცია გახსნის მონაცემთა ბაზასთან მიერთებასა და არ დახურავს მას მუშაობის დამთავრებამდე. დანართის სირთულის ზრდასთან ერთად იზრდება მონაცემთა ბაზის კლიენტების რაოდენობაც, რაც არაეფექტურს ხდის ბაზასთან მუდმივი მიერთების ტექნოლოგიას, კერძოდ:

- სისტემური რესურსები გამოიყენება არაეფექტურად. რაც უფრო მეტი მუდმივად მიერთებული (გახსნილი) ფაილია, მით უფრო დაბალია სისტემის მწარმოებლურობა;
- დანართები, რომლებიც იყენებს მონაცემებთან მიმართვას მუდმივი მიერთების საშუალებით, ცუდად მასშტაბირებადია. ასეთი დანართები კარგად ემსახურება ორ მიერთებულ კლიენტს, 10-თან უკვე უჭირს მუშაობა და 100-თან საერთოდ ვერ ფუნქციონირებს.

ADO.NET სისტემაში ეს პრობლემები წყდება მონაცემებთან მიმართვის ისეთი მოდელის გამოყენებით, როგორიცაა **გამოყოფილი** მონაცემები. ასეთი მოდელის შემთხვევაში მონაცემთა წყაროსთან მიერთება გახსნილია მხოლოდ სათანადო პროცედურის შესასრულებლად.

მაგალითად, თუ აპლიკაციას დასჭირდა მონაცემები ბაზიდან, იგი მიუერთდება მას ამ მონაცემების გადმოტვირთვამდე, შემდეგ კი მიერთება დაიხურება.

ასევე როდესაც ხორციელდება მონაცემთა განახლება ბაზაში, მიერთება წყაროსთან განხორციელდება UPDATE-ბრძანების შესრულების დამთავრებამდე, შემდეგ იგი დაიხურება. ამგვარად, მონაცემებთან მიერთების დროის

(გახსნა-დახურვის პერიოდი) შემცირებით, ADO.NET უზრუნველყოფს სისტემური რესურსების ეკონომიურ გამოყენებასა და მონაცემთა წვდომის ინფრასტრუქტურის მასშტაბირებას, მწარმოებლურობის შემცირების გარეშე.

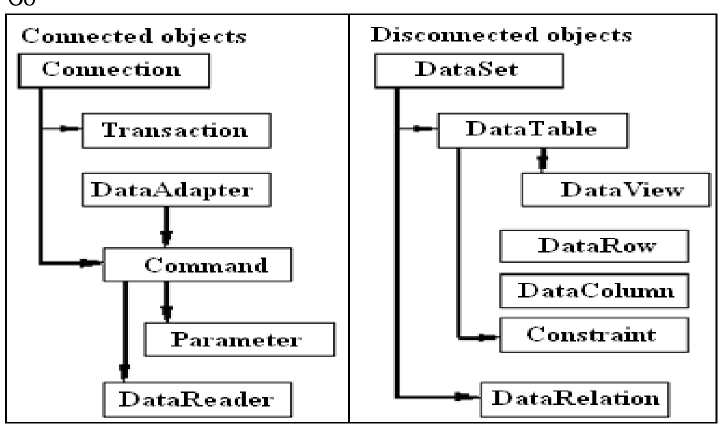
### 1.13.2. ADO.NET -ის მონაცემთა არქიტექტურა

სისტემის ობიექტური მოდელი ორი ნაწილისაგან შედგება: მარცხენა - მიერთებადი ობიექტები (Connected Objects) და მარჯვენა - განცალკევებადი ობიექტები (Disconnected Objects). მე-40 ნახაზზე ნაჩვენებია ADO.NET ობიექტური მოდელის შემადგენელი კლასები, რომელთა დანიშნულებასაც ამ პარაგრაფში მოკლედ შევეხებით.

მონაცემებთან მიმართვა ADO.NET-ში ხორციელდება ორი კომპონენტით:

- მონაცემთა ერთობლიობით (DataSet ობიექტით), რომელშიც მონაცემები ინახება ლოკალურ კომპიუტერში;

- მონაცემთა მიმწოდებლით (DataProvider პროვაიდერით), რომელიც ასრულებს შუამავლის ფუნქციას პროგრამასა და მონაცემთა ბაზას შორის.




ნახ.40

**ობიექტი DataSet:** ესაა მონაცემთა წარმოდგენა კომპიუტერის მეხსიერებაში მონაცემთა წყაროსაგან იზოლირებულად. ეს ობიექტი შეიძლება განვიხილოთ, როგორც მონაცემთა ბაზის ფრაგმენტის ლოკალური ასლი (კოპიო).

DataSet-ში მონაცემთა ჩატვირთვა შესაძლებელია ნებისმიერი დასაშვები წყაროდან, მაგალითად, SQL Server, Ms Access ბაზებიდან ან XML–ფაილიდან. დასაშვებია მეხსიერებაში ამ მონაცემებით მანიპულირება, აგრეთვე მთავარი წყაროსაგან დამოუკიდებლად მათი განახლება.

ობიექტი DataSet შედგება DataTable ობიექტთა ერთობლიობისაგან (ის შეიძლება ცარიელიც იყოს ანუ არ შეიცავდეს არც ერთ DataTable-ს).

ყოველი DataTable ობიექტი კომპიუტერის მეხსიერებაში ასახავს ერთ ცხრილს. მისი სტრუქტურა შეიცავს ორ ერთობლიობას: DataColumns, რომელშიც თავსდება ცხრილის სვეტები, და ცხრილის შეზღუდვათა ერთობლიობა. ეს ორი ერთობლიობა ქმნის ცხრილის სქემას.

DataTable ობიექტი შეიცავს აგრეთვე DataRows ერთობლიობaს, რომელშიც ინaხება DataSet-ობიექტის მონაცემები.

გარდა ამისა, DataSet ობიექტი შეიცავს DataRelations ერთობლიობას, რომელიც უზრუნველყოფს კავშირების შექმნას სხვადასხვა ცხრილის სტრიქონებს შორის. DataRelations შეიცავს DataRelation ობიექტთა ერთობლიობას, რომლებიც განსაზღვრავს ცხრილთაშორის კავშირებს (მაგალითად, 1:M კავშირის სარეალიზაციოდ).

და ბოლოს, DataSet ობიექტი შეიცავს ExtendedProperties ერთობლიობას, რომელშიც შეინახება დამატებითი მონაცემები.

**მონაცემთა პროვაიდერი:** ესაა ურთიერთდაკავშირებულ კომპონენტთა ერთობლიობა, რომელიცმონაცემთა ბაზასთან უზრუნველყოფს ეფექტურ მაღალმწარმოებლურ კავშირს.

.NET Framework-s აქვს ორი პროვაიდერი: SQL Server .NET Data Provider, რომელიც შექმნილია SQL Server 7.0 ან უფრო მაღალ ვერსიებთან სამუშაოდ, და OleDb .NET Data Provider - სხვა ტიპის მონაცემთა ბაზებთან დასაკავშირებლად.

მონაცემთა ნებისმიერი პროვაიდერი შედგება მსგავსი უნივერსალური კლასების კომპონენტებისაგან:

- Connection, რომელიც უზრუნველყოფს მონაცემთა ბაზასთან მიერთებას;

- Command, რომელიც გამოიყენება მონაცემთა წყაროს სამართავად. იგი გამოიყენებს ბრძანებებს, რომლებიც არ აბრუნებს მონაცემებს, მაგალითად, INSERT, UPDATE da DELETE, ან ბრძანებებს, რომლებიც აბრუნებს DataReader ობიექტს (მაგალითად, SELECT);

- DataReader გამოიყენება მხოლოდ ჩანაწერთა ერთობლიობის წასაკითხად მიერთებული მონაცემთა წყაროდან;

- DataAdapter შეავსებს გამოყოფილ DataSet ან DataTable ობიექტსა და განაახლებს მათ შედგენილობას.

მონაცემებთან მიმართვა ხორციელდება შემდეგნაირად: ობიექტი Connection აყენებს დანართის (აპლიკაციის) მონაცემთა ბაზასთან მიერთებას, რომელიც პირდაპირ მისაწვდომია Command და DataAdapter ობიექტებისთვის. Command ობიექტი უზრუნველყოფს ბრძანებათა შესრულებას უშუალოდ მონაცემთა ბაზაში. თუ შესასრულებელი ბრძანება აბრუნებს რამდენიმე მნიშვნელობას, მაშინ Command ხსნის მათთან მიმართვას DataReader ობიექტის საშუალებით. მიღებული შედეგები შესაძლებელია დამუშავდეს უშუალოდ დანართის კოდით, ან DataSet ობიექტით, რომელიც შეივსება DataAdapter ობიექტის დახმარებით. მონაცემთა ბაზის განახლებისათვის ასევე გამოიყენება Command და DataAdapter ობიექტები.

**ობიექტი Connection** გვთავაზობს მიერთებას მონაცემთა ბაზასთან. Visual Studio .NET–ს აქვს Connection-ის ორი კლასი:

- SqlConnection (MsSQL_Server-თან შესაერთებლად) და

- OleDbConnection (სხვა ტიპის მონაცემთა ბაზებთან დასაკავშირებლად).

მონაცემთა ბაზასთან კავშირის არხის გასახსნელი აუცილებელი მონაცემები ინახება Connection ობიექტის ConnectionString თვისებაში. ეს ობიექტი ინახავს აგრეთვე რიგ მეთოდს, რომლებიც საჭიროა ტრანზაქციების გამოყენებით მონაცემთა დასამუშავებლად.

**ობიექტს Command** აქვს ორი კლასი: SqlCommand და OleDbCommand. იგი უზრუნველყოფს ბრძანებათა გამოყენებას მონაცემთა ბაზაზე, რომელთანაც დამყარებულია კავშირი (მიერთება). აქ შეიძლება გამოყენებულ იქნას შენახვადი პროცედურები (Stored Procedures), SQL-ენის ბრძანებები, აგრეთვე ოპერატორები მთლიანი ცხრილების მისაღებად. Command ობიექტს აქვს სამი მეთოდი :

- Execute Non Query: იყენებს ბრძანებებს, რომლებიც არ აბრუნებს მონაცემებს, მაგალითად, INSERT, UPDATE და DELETE;

- Execute Scalar: იყენებს მოთხოვნებს მონაცემთა ბაზისადმი, რომლებიც აბრუნებს მხოლოდ ერთ მნიშვნელობას;

- Execute Reader: აბრუნებს საშედეგო ერთობლიობას DataReader ობიექტის საშუალებით.

**ობიექტი** DataReader გვთავაზობს ნაკადს მონაცემთა ბაზის ჩანაწერების ერთობლიობით, ოღონდ მხოლოდ ერთი მიმართულებით წასაკითხად. მონაცემთა პროვაიდერის სხვა კომპონენტებისაგან განსხვავებით DataReader-ის ეგზემპლარების შექმნა პირდაპირ არაა დასაშვები. მისი მიღება შეიძლება Command ობიექტის ExecuteReader მეთოდით:

- SqlCommand.ExecuteReader მეთოდი აბრუნებს SqlDataReader ობიექტს;

- OleDbCommand.ExecuteReader მეთოდი კი - OleDbDataReader ობიექტს.

თუ DataReader ობიექტის შემცველი მონაცემების ჩაწერა დისკზე არაა საჭირო, მაშინ ეს სტრიქონები შეიძლება პირდაპირ გადაეგზავნოს დანართს. ვინაიდან დროის ნებისმიერ მომენტში მეხსიერებაში იმყოფება მხოლოდ ერთი სტრიქონი, DataReader ობიექტის გამოყენება თითქმის არ ამცირებს სისტემის მწარმოებლურობას, ოღონდ მოითხოვს მონოპოლურ მიმართვას გახსნილ Connection-ობიექტზე DataReader ობიექტის სასიცოცხლო დროის განმავლობაში.

**ობიექტი DataAdapter** არის ADO.NET–ის ძირითადი კლასი, რომელიც უზრუნველყოფს გამოყოფილ მონაცემებთან მიმართვას. არსებითად, იგი ასრულებს შუამავლის ფუნქციებს მონაცემთა ბაზისა და DataSet-ობიექტის ურთიერთქმედებისათვის.

Fill მეთოდის გამოძახებისას DataAdapter ობიექტი შეავსებს მონაცემებით DataTable-ს ან DataSet-ს მონაცემთა ბაზიდან. მონაცემების დამუშავების შემდეგ, რომლებიც ჩატვირთულია მეხსიერებაში, შესაძლებელია მოდიფიცირებული ჩანაწერების მოთავსება მონაცემთა ბაზაში, DataAdapter ობიექტის Update მეთოდის გამოძახებით. DataAdapter-ს აქვს ოთხი თვისება, რომლებიც წარმოადგენს მონაცემთა ბაზის ბრძანებებს:

- SelectCommand შეიცავს ტექსტს ან ბრძანების ობიექტს, რომელიც ახორციელებს მონაცემთა ბაზიდან ამორჩევას (მაგალითად, მეთოდი Fill);

- InsertCommand შეიცავს ტექსტს ან ბრძანების ობიექტს, რომელიც ახორციელებს სტრიქონის ჩასმას ცხრილში;

- DeleteCommand შეიცავს ტექსტს ან ბრძანების ობიექტს, რომელიც ახორციელებს სტრიქონის წაშლას ცხრილიდან;

- UpdateCommand შეიცავს ტექსტს ან ბრძანების ობიექტს, რომელიც ახორციელებს მნიშვნელობათა განახლებას მონაცემთა ბაზაში;

Update მეთოდის გამოძახებისას ყველა შეცვლილი მონაცემი კოპირდება DataSet ობიექტიდან მონაცემთა ბაზაში, შესაბამისი ბრძანებების InsertCommand, DeleteCommand ან UpdateCommand გამოყენებით.

### 1.13.3. მონაცემთა ბაზასთან მიერთება

Visual Studio .NET სისტემას აქვს სტანდარტული ოსტატი პროგრამებისა და დიზაინერების სიმრავლე, რომელთა საშუალებითაც ადვილად და ეფექტურად ხორციელდება მონაცემებთან წვდომის არქიტექტურა დანართების დამუშავების პროცესში. ამასთანავე ADO.NET ობიექტური მოდელის ყველა შესაძლებლობა მისაწვდომია პროგრამულად, რაც უზრუნველ-ყოფს არასტანდარტული ფუნქციების რეალიზაციის ან დანართების აგების შესაძლებლობას, რომლებიც მომხმარებელთა მოთხოვნილებებზეა ორიენტირებული.

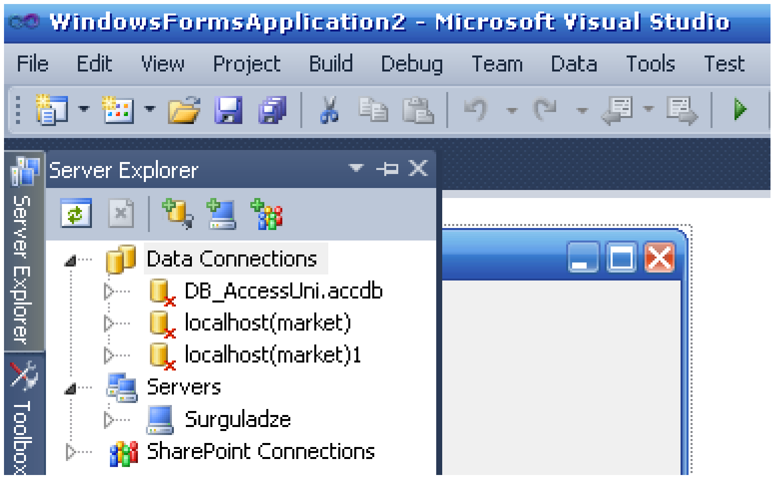
აქ ჩვენ გავეცნობით, თუ როგორ დავუკავშირდეთ მონაცემთა ბაზას ADO.NET–ის გამოყენებით, როგორ ამოვიღოთ საჭირო მონაცემები და გადავცეთ ისინი პროგრამულ აპლიკაციას. ეს საკითხები შეიძლება შესრულდეს Visual Studio .NET-ის გრაფიკული ინსტრუმენტებითაც და პროგრამულადაც.

C# პროგრამულ აპლიკაციაში არსებობს მონაცემთა ბაზასთან მიერთების რამდენიმე ხერხი. ყველაზე მარტივია ამის განხორციელება Visual Studio .NET-ის გრაფიკული ინსტრუმენტით. მონაცემთა წყაროსთან (DataSource) მიერთებისა და მისი მართვისათვის გამოიყენება ფანჯარა Server Explorer.

ძირითადი ამოცანა, რომელსაც ჩვენ აქ განვიხილავთ, არის ADO.NET პროგრამული პაკეტის გამოყენებით მომხმარებელთა

სამუშაო ინტერფეისის დამუშავების სადემონსტრაციო მაგალითის აგება. ამასთანავე, მონაცემთა ბაზების სახით უნდა გამოვიყენოთ Ms_Access, MySQL და Ms_SQL_Serever პაკეტებით აგებული ცხრილები.

დავუშვათ, რომ ზემოაღნიშნული მონაცემთა ბაზების მართვის სისტემები დაინსტალირებულია ჩვენს კომპიუტერზე. თვით Visual Studio .NET -ის დაინსტალირებისას ავტომატურად ყენდება Ms SQL Server Expression, რომელზეც ასევე შესაძლებელია ექსპერიმენტის ჩატარება. .NET სამუშაო გარემოს ჩატვირთვის შემდეგ საჭიროა Server Explorer-ის გახსნა და ბაზებთან კავშირის შემოწმება (მაგალითად, ნახ.41).




ნახ.41

წიგნის მერე თავში დეტალურადაა განხილული მომხმარებლის ინტერფეისის (პროგრამული სისტემის) კავშირის რეალიზაცია მონაცემთა ბაზებთან.

## 1.14. IT-მენეჯმენტის მხარდამჭერი და ტრანზაქციების დამუშავების სისტემები

ტრანზაქციის დამუშავების სისტემები (Transaction procesing system TPS) ძირითადი საწარმოო სისტემაა (ე.წ. ბირთვი სისტემა - core system), სადაც ფიქსირდება და მუშავდება ორგანიზაციის ყოველდღიური საქმიანობა. მიკუთვნება საოპერიაციო დონის სისტემების ანუ პროცესების მართვის სისტემების რგოლს (Process Control Systems). შემავალი ინფორმაცია (ტრანზაქცია) - მოვლენა, რომელიც გავლენას ახდენს ორგანიზაციის საქმიანობაზე, მუშავდება და გარდაიქმნება გამომავალ ინფორმაციად, რომელსაც იყენებს სხვადასხვა მომხმარებელი. შემავალი ინფორმაციის დამუშავებაში იგულისხმება მონაცემების ვალიდაცია, სორტირება, გაერთიანება, განახლება, კალკულაცია და ა.შ.

ტრანზაქციის დამუშავების სისტემების მაგალითებია - სახელფასო, დაჯავშნის, გადახდის, გადარიცხვების, შეკვეთების დამუშავების სისტემები და სხვ.

მენეჯმენტის მხარდამჭერი სისტემები (Management Support Systems) განკუთვნილია ორგანიზაციული მართვის საშუალო და ზედა დონისათვის, მართვის ამოცანებისა და გადაწყვეტილების მიღების პროცესების ხელშესაწობად. ასეთი სისტემებია მართვის საინფორმაციო სისტემები (Management Informations System - MIS), გადაწყვეტილების მიღების მხარდამჭერი სისტემები (Decision Support System - DSS), ექსპერტული სისტემები (Expert System -ES) , ადმინისტრაციული მხარდამჭერი სისტემები (Executive Support Systems - ESS) [3,7-,10,36,37].

მართვის საინფორმაციო სისტემები განკუთვნილია საშუალო დონის მენეჯმენტისათვის ტაქტიკური გადაწყვეტილების მისაღებად. იგი, გამოიყენება ორგანიზაციის მწარმოებლურობის შეფასებისათვის (მაგალითად, მიმდინარე და წინა შედეგების

შედარება), რათა უზრუნველყოს ორგანიზაციის უწყვეტი მუშაობა მოკლე და საშუალოვადიანი პერსპექტივის ფარგლებში. არ განეკუთვნება სტრატეგიული და გრძელვადიანი პროგნოზირების ამოცანების გადაწყვეტის საშუალებას. მუშაობს სტრუქტურულ მონაცემებთან (მონაცემთა ბაზასთან), ბაზირებულია შიგა ინფორმაციულ ნაკადზე, არ შეიცავს ძლიერ ანალიზურ ინსტრუმენტებს, იყენებს წარსულ და მიმდინარე მონაცემებს, შესაბამისად ეყრდნობა ტრანზაქციის დამუშავების სისტემებისაგან მიღებულ ინფორმაციას. მართვის საინფორმაციო სისტემების მაგალითებია: ბიუჯეტის, საკადრო, გაყიდვების, რეპორტინგის მართვის სისტემები.

გადაწყვეტილების მიღების მხარდამჭერი სისტემები განკუთვნილია მაღალი დონის მენეჯმენტისთვის. ძირითადად, ინტერაქტიული სისტემებია, რომელიც ახდენს სხვადასხვა წყაროებიდან მონაცემების ინტეგრირებას, აქვს ძლიერი ანალიტიკური პროგრამული და მათემატიკური მოდელირების მექანიზმები, ახდენს მათ გამოსახვას მოხერხებული ფორმით. ასეთი ტიპის სისტემები ეფუძნება ცოდნასა და განკუთვნილია პროგნოზირების ამოცანების გადასაწყვეტად. აწყობილია მიზეზ-შედეგობრივი და სიმულაციური კომპონენტებით, რაც მენეჯერებს საშუალებას აძლევს, დაინახონ მათი გადაწყვეტილებატა პოტენციური შედეგები. ასეთი ტიპის სისტემები ანალიზის, პროგნოზირებისა და შედეგების ინტერპრეტაციის საშუალებას იძლევა.

გადაწყვეტილების მხარდამჭერი სისტემების ჭრილში განიხილება ექსპერტული და ხელოვნური ინტელექტის სისტემებიც. მათი საფუძველია ცოდნის ბაზა და წესების ერთობლიობა. ასეთია დიაგნოსტიკური, კლიენტის გადახდისუნარიანობის კვლევის და სხვ. სისტემები. გადაწყვეტილების მხარდამჭერი სისტემა ეფუძნება მართვის საინფორმაციო და ტრანზაქციის დამუშავების სისტემებისაგან მიღებულ ინფორმაციას, ახდენს მის მანიპულირებასა და ქმნის ახალ ინფორმაციას.

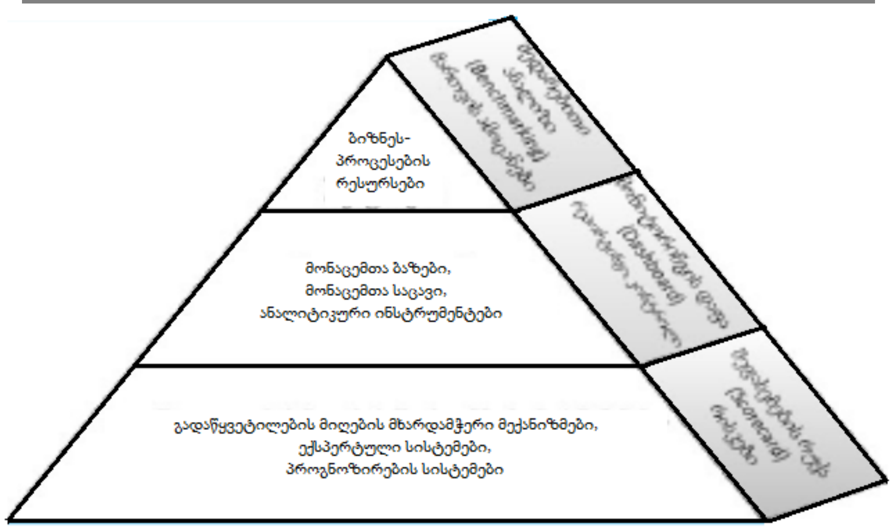
გადაწყვეტილების მხარდამჭერი სისტემების მაგალითებია - ლოგისტიკის, ფინანსური დაგეგმვის, კადრების შერჩევის, რესურსების მართვის სისტემები.

ადმინისტრაციული მხარდამჭერი სისტემები არის სტრატეგიული დონის საინფორმაციო სისტემები. განკუთვნილია აღმასრულებელი და მაღალი დონის მენეჯმენტისათვის პროაქტიული გადაწყვეტილებების მისაღებად. იყენებს როგორც შიგა, ისე გარე ინფორმაციას. ახდენს იმ გარემოს ანალიზს, რომელშიც ოპერირებს ორგანიზაცია, სამომავლო გეგმებისა და მიზნების შესამუშავებლად. როგორც წესი, ასეთი სისტემები გამოსაყენებლად უნდა იყოს ადვილი, აგენერირებდეს ანალიტიკურ რეპორტებს, უნდა შეიცავდეს გადაწყვეტილების მიღების ინსტრუმენტებს, კალენდალური დაგეგმვის კომპონენტს, ელექტრონულ ფოსტას, სიტუაციების მონიტორინგის ელემენტებს, როგორიცაა მაგალითად, ე.წ. მართვის დაფა (Dashboard), გრაფიკული ინსტრუმენტები და ა.შ.

ადმინისტრაციული მხარდამჭერი სისტემების მაგალითია, დღესდღეობით საკმაოდ პოპულარული ბიზნესის ანალიტიკური მართვის ტექნოლოგია (Business Inteligence – BI) [38].

BI ტექნოლოგია იმ საშუალებათა კომპლექსია, რომელიც გვაწვდის საჭირო და სასარგებლო ინფორმაციას კორპორაციაში მიმდინარე ყველა მოვლენის შესახებ. საშუალებათა კომპლექსი გულისხმობს: საინფორმაციო ბაზებს, სპეციალიზებულ ბიზნეს-დანართებს, ელექტრონული ბიზნესის სისტემებსა და განაპირობებს მონაცემების ბიზნეს-ანალიტიკოსისათვის მოსახერხებელი სახით მიწოდებას (ნახ.42).

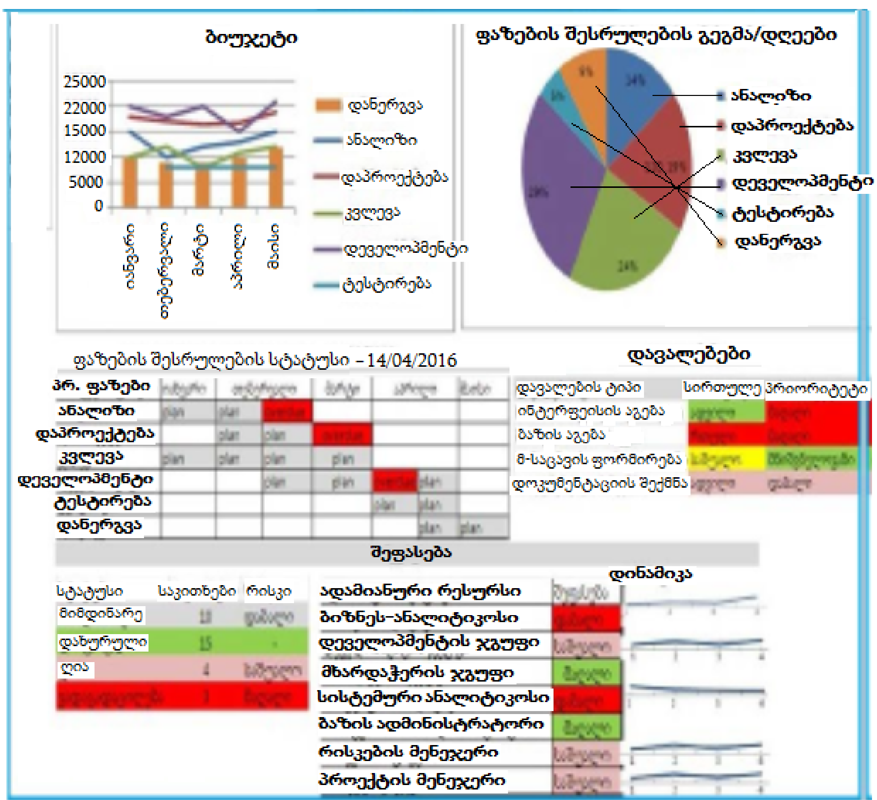
Business Intelligence პროდუქტები შეიცავს: BI-ინსტრუმენ-ტებსა და BI-დანართებს. BI- ინსტრუმენტები იქმნება მოთხოვნათა და ანგარიშთა გენერეტორების საშუალებით.




ნახ.42

განვითარებული BI-ინსტრუმენტები: პირველი - ოპერატიული ანალიტიკური დამუშავების ინსტრუმენტი (Online analytical processing, OLAP); კორპორატიული BI-კომპლექტი (Enterprise BI suites, EBIS); BI- პლატფორმები. მთავარი ნაწილი BI-ინსტრუმენტებისა იყოფა BI- კორპორაციული BI-კომპლექტებად და BI-პლატფორმებად. მოთხოვნებისა და ანგარიშების გენერაციის საშუალებები ხშირ შემთხვევებში იცვლება კორპორაციული BI-კომპლექტებით.

მრავალგანზომილებიანი OLAP-მექანიზმები ან სერვერები, აგრეთვე რელაციური OLAP-მექანიზმები წარმოადგენს BI ინსტრუმენტებსა და BI პლატფორმის ინფრასტრუქტურას. BI ინსტრუმენტების უმრავლესობა გამოიყენება საბოლოო მომხმარებლების მიერ მონაცემთა წვდომისათვის, ანალიზისა და ანგარიშების გენერაციისათვის, რომელიც ყველაზე ხშირად განთავსებადია მონაცემთა საცავებში, მონაცემთა ვიტრინებში ან მონაცემთა ოპერატიულ საწყობებში (ნახ.43) [39].




ნახ.43

მონაცემთა საცავების (Data Warehousing) კონცეფცია, მეთოდები და საშუალებები განსაზღვრავს მიდგომებსა და უზრუნველყოფს ინტეგრაციას, გაფილტვრას, ინფორმაციის რეტროსპექტულ შენახვას, რომელიც გამიზნულია ანალიზისათვისა და პასუხობს შეკითხვას „როგორ მოვამზადოთ ინფორმაცია ანალიზისათვის?". ბიზნეს-ინტელექტის ტექნოლოგია განსაზღვრავს მეთოდებს, წვდომისა და ინფორმაციის ოპერატიული ანალიზის საშუალებებს მოცემული გარემოს ტერმინებში (საზღვრებში).

## 1.15. საინფორმაციო ტექნოლოგიების სერვისების იმპლემენტაცია და ხარისხის მართვა

ინფორმაციული ტექნოლოგიების სერვისების მხარდაჭერის მენეჯმენტი (ITSM - Information Technology Service Management) ეხება IT- სერვისების იმპლემენტაციისა და ხარისხის მართვას [10]. იგი მოიცავს შემდეგ თემებს:

- ინციდენტების მართვა (Incident management);
- პრობლემების მართვა (Problem management);
- კონფიგურაციის მართვა (Configuration management);
- ცვლილებების მართვა (Change management);
- ვერსიების მართვა (Release management);
- მომსახურების დონის მართვა (Service level management);
- IT სერვისებისთვის ფინანსების მართვა (Financial management for IT services);
- მწარმოებლურობის მართვა (Capacity management);
- IT სერვისების უწყვეტობის მართვა (IT service continuity management);
- წვდომის მართვა (Availability management).

ქვემოთ შევეხებით ზოგიერთ მათგანს უფრო დეტალურად.

➢ **ინციდენტების მართვის პროცესი.**

ინციდენტები შეიძლება კლასიფიცირებული იყოს სამ ძირითად კატეგორიად: პროგრამული უზრუნველყოფა (აპლიკაციები), მოწყობილობა და მომსახურების მოთხოვნა.

ITIL ყოფს ინციდენტების მართვას ექვს ძირითად კომპონენტად:

- ინციდენტების გამოვლენა და ჩაწერა;
- კლასიფიკაცია და პირველადი მხარდაჭერა;
- გამოკვლევა და დიაგნოზის დასმა;
- რეზოლუცია და აღდგენა;

- ინციდენტების დახურვა;
- საკუთრება, მონიტორინგი, და კომუნიკაცია;

ინციდენტების მართვის ძირითადი საქმიანობაა:

- მიიღოს ინციდენტზე საკუთრების უფლება და იმოქმედოს, როგროც პირველადი კონფლიქტის ესკალაცია;
- უზრუნველყოს სწრაფი აღდგენა ბიზნესის;
- დაარწმუნოს, რომ ძირითადი ფოკუსი ინციდენტების გაფართოების არაა გაკეთებული სხვა ღონისძიებების მიერ;
- მზარდი ინციდენტები: ფუნქციური (ამ პრობლემის გადაწყვეტა მოითხოვს უმაღლეს ტექნოლოგიურ რესურსებს) და იერარქიული (მენეჯერი მეტი უფლებამოსილებით იღებს გადაწყვეტილებას, რომელიც სცილდება ამ დონეზე მინიჭებულ კომპეტენციას);
- გაუგზავნოს შეტყობინება ინციდენტებზე მომხმარებელს (დოკუმენტები, რომელიც შეიცავს დეტალურ ინფორმაციას);
- შექმნასა და წარუძღვეს საკონფერენციო ზარებს ან კომუნიკაციის ხიდებს ყველა ჩართულ მხარეებს შორის;
- ჩაატაროს აღრიცხვა და ჩანაწერები.

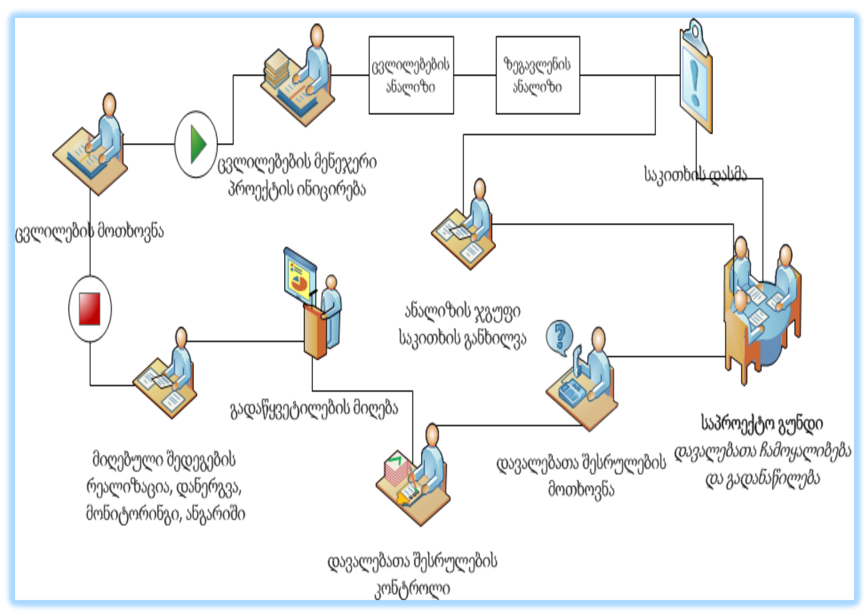
➢ **ცვლილებების მართვა:**

კომპანიების და მასთან დაკავშირებული ბიზნესპროცესების წარმატებული მუშაობის ბირთვია საინფორმაციო სისტემები, ინფორმაციული ინფრასტრუქტურა და პროგრამული პროდუქ-ტები. როგორც კომპანიის ბიზნესსტრატეგია, ისე საინფორმაციო სისტემების მთელი ინფრასტრუქტურა მიდრეკილია მუდმივი ცვლილებებისაკენ. ცვლილება, როგორც წესი, დამოკიდებულია ორ ძირითად მოვლენაზე:

1. განვითარება კომპანიის მოთხოვნების შესაბამისად და
2. მოულოდნელი ხარვეზების აღმოჩენა იმპლემენტაციის შემდგომ.

საინფორმაციო სისტემების ცვლილებების მართვის პროცესის სასიცოცხლო ციკლი, რომელსაც მართავს ცვლილებების მენეჯერი, მოიცავს შემდეგ ბიჯებს: ცვლილების რეგისტრაცია, ზეგავლენის შეფასება, საკადრო, დროითი და მატერიალური ხარჯების ანალიზი, შედეგის მომგებიანობა, რისკის ანალიზი, ბიზნეს-საფუძველი, ბიზნეს-ანალიზი, ტექნიკური ანალიზი, მენეჯმენტის სხვადასხვა ერთეულის დასტურის მიღება ცვლილებების დანერგვაზე, ცვლილებების მომზადებისა და რეალიზაციის კოორდინაცია და მართვა, ცვლილებების რეალიზაციის მონიტორინგი და ანგარიში [39].

44-ე ნახაზზე ასახულია ცვლილებების მართვის პროცესის სასიცოცხლო ციკლის ფრაგმენტი „ცვლილების მოთხოვნიდან - გადაწყვეტილების მიღებამდე“.




**ნახ.44. ცვლილებების მართვის პროცესის სასიცოცხლო ციკლის ფრაგმენტი**

თითოეული ეტაპის შესრულება მოიცავს შესაბამისი ეტაპის დოკუმენტირების მხარეს, ცვლილებების მენეჯმენტის საერთაშორისო სტანდარტების (ITSM, ITIL, ISO 20000) რეგულაციებისა და მოთხოვნების მიხედვით. დოკუმენტაციის მხარე იყოფა ორ ნაწილად:

1. დოკუმენტები, რომელიც დაკავშირებულია ცვლილების პროცესის აღწერასთან. ასეთი ტიპის დოკუმენტებია: ბიზნეს-ანალიზი, ტექნიკური ანალიზი, ცვლილების რეგისტრაცია, ზეგავლენის შეფასება და სხვა;

2. მენეჯმენტის სხვადასხვა დონის ერთეულის მიერ დასადასტურებელი დოკუმენტები, რომლის დადასტურების გარეშე ვერ ინიცირდება შესაბამისი პროცესი, ეტაპი ან მოვლენა. ასეთი ტიპის დოკუმენტებია თანხმობა ცვლილებაზე, რისკების ანალიზი, ინციდენტ-ანალიზი, ტექნიკური, რესურსული და ფინანსური ანალიზისა და ა.შ.

დღეისათვის საინფორმაციო ტექნოლოგიების ინდუსტრიაში ცნობილია IT ცვლილების მართვის ავტომატიზაციის პროგრამული პროდუქტები, რაც მოქნილი ინსტრუმენტებია ცვლილების მართვის სასიცოცხლო ციკლის პროცესების შესრულებისათვის (მაგალითად, Jira, WFMS, Six Sigma, Enterprise Architect და სხვ.). ამ ინსტრუმენტების გამოყენებით, შესაძლებელია, როგორც ცვლილების პროცესის სრული კოორდინაცია (შეტყობინებათა რეჟიმის ჩათვლით) და მონიტორინგი, ისე ბიზნეს-პროცესების აღწერა, ზეგავლენის ანალიზი და შესაბამისი დოკუმენტაციის წარმოება.

IT ცვლილებების მართვის პროცესზე პასუხისმგებელია ცვლილებების მენეჯერი, რომელიც ცვლილებების მოთხოვნის საფუძველზე ახდენს ცვლილებების დანერგვისთვის შესაბამის ტექნიკურ, შინაარსობრივ, ფუნქციურ, ტექნოლოგიურ, რესურსულ, ინფრასტრუქტურულ და ცვლილების სპეციფიკასთან

დაკავშირებულ მოვლენათა ანალიზს. კვლევის ეტაპზე ცვლილებების პროცესისთვის დამახასიათებელია მრავასტრუქტურული ანალიზი, რომლის კოორდინაციასაც ახდენს ცვლილებების მენეჯერი, განსაზღვრავს რა პროცესში მონაწილე პირებს, ახდენს პროცესის დეკომპოზიციას დავალებების სახით, გეგმავს დავალებათა გადანაწილებას შესაბამის შემსრულებლებზე, აყალიბებსა და აკონტოლებს დავალებათა შესრულების ვადებს. გამომდინარე მრავასტრუქტურული და ფუნქციური ანალიზიდან, დეკომპოზიციური დავალებები დასაშვებია  ატარებდეს, როგორც მიმდევრობით, ისე პარალელურ ხასიათს. შესაბამისად, დავალებათა შესრულების კონტროლი დროული და სრული შესრულების თვალსაზრისით, შესრულების შეფერხებისას ჩანაცვლების ან ადეკვატური გადაწყვეტილების დროული მიღება, მნიშვნელოვანი ფაქტორია პროცესის სწორად და ეფექტურად შესრულების მიმართებით. მით უფრო, რომ ცვლილებების დანერგვა, როგორც წესი, შემოსაზღვრულია დროით.

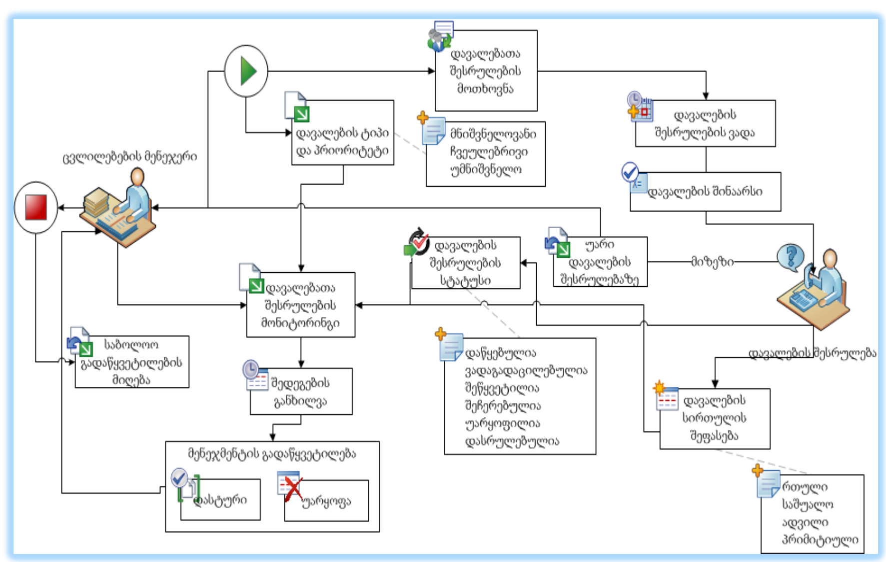
აღწერილი პროცესი მიეკუთვნება საქმიანი ნაკადების მართვის პროცესის, ბიზნეს-პროცესების მართვისა და პროექტების მართვის ტექნოლოგიის სტანდარტებს. ამ ტექნოლოგიების მიხედვით, ცვლილებების მართვის სამუშაო პროცესის კოორდინაცია აღიწერება შემდეგი სახით: ცვლილებების მენეჯერი აანალიზებს ცვლილების ძირითად, დამხმარე და იმ დაკავშირებულ პროცესებს, რაზეც ცვლილება იქონიებს ზეგავლენას. შესაბამისად, ცვლილებების პროცესი დეკომპოზირდება ცალკეულ დავალებებად, რომელთაც ენიჭებათ პრიორიტეტი.

სტანდარტული პრიორიტეტის ტიპებია: მაღალი, დაბალი, საშუალო, მნიშვნელოვანი, ჩვეულებრივი, სტანდარტული, უმნიშვნელო. პრიორიტეტის ტიპის შესაბამისად, შესაძლებელია დაიგეგმოს დავალების შესრულების ვადა. დავალება შესაბამის

მოთხოვნასთან და აღწერილობასთან ერთად გადაეცემა შესაბამის სტრუქტურულ ერთეულს. მისი წარმომადგენელი ახდენს დავალების სირთულის შეფასებას, მიღებას ან აპელირებას.

დავალების სირთულის შეფასების მიხედვით შესაძლებელია განისაზღვროს დავალების შემსრულების კვალიფიკაცია (პოზიცია), იმ შემთხვევაში, თუ დავალება მარტივია, დასაშვებია დაკავდეს დაბალი პოზიციის რესურსი. ყოველ დავალებას აქვს შესრულების სტატუსი: მიმდინარე, აქტიური, დაწყებული, დასრულებულია, შესრულებულია, უარყოფილია და ა. შ.

პროცესის კოორდინაცია ამ მიმართულებით, მხარს უჭერს მოქნილ მონიტორინგს, კრიტიკული და გაუთვალისწინებელი ჩიხური სიტუაციების მინიმიზაციას, გადაწყვეტილებისა და შეფერხებებზე დროული რეაგირების საკითხებს. 45-ე ნახაზზე ასახულია IT ცვლილებების მართვის პროცესის აღწერის გრაფიკული წარმოდგენა.




ნახ.45. IT ცვლილებების მართვის პროცესის აღწერის ფრაგმენტი

ყოველი შესრულებული დავალება და შემდგომში ეტაპი აუცილებელია იქნეს დოკუმენტირებული. რიგი დავალება, უსაფრთხოების მიზნით ვერ შესრულდება შესაბამისი ექსპერტის (სპეციალისტის, ჯგუფის ხელმძღვანელის, მენეჯერის) დასტურის გარეშე. არსებობს, ცვლილებების მართვის დოკუმენტირების შაბლონური სტანდარტები. დოკუმენტაციის სტანდარტული ნუსხაა: 1. ცვლილების მოთხოვნის დოკუმენტი (RFC - Requests for Change); 2. ფინანსური ანალიზი და დასტური; 3. ტექნიკური ანალიზი და დასტური; 4. ბიზნეს-ანალიზი (ფუნქციური ანალიზი) და დასტური; 5. ზეგავლენის ანალიზი; 6. ცვლილებების ანალიზი და დასტური (RFC approval); 7. ცვლილებების შესრულების გეგმა (FSC - Forwerd Schedule of Changes); 8. საკონსულტაციო საბჭოს ცვლილებების შეფასება (CAB - Change Advisory Board); 9. ტესტირების გეგმა და ჩატარებული ტესტირების დოკუმენტაცია; 10. ცვლილებების ხარისხის შეფასება და რისკების ჯგუფის დასტური (Post implementation review); 11. ცვლილებების დანერგვისა და ტესტირების გეგმა (Building and testing); 12. დანერგილი ცვლილების მონიტორინგის ანალიზი და პროექტის დახურვა (RFC closure); 13. რეპორტინგი მენეჯმენტისათვის (Management reporting); 14. ბეკაპირებისა და უკუპროცესის აღდგენის გეგმა.

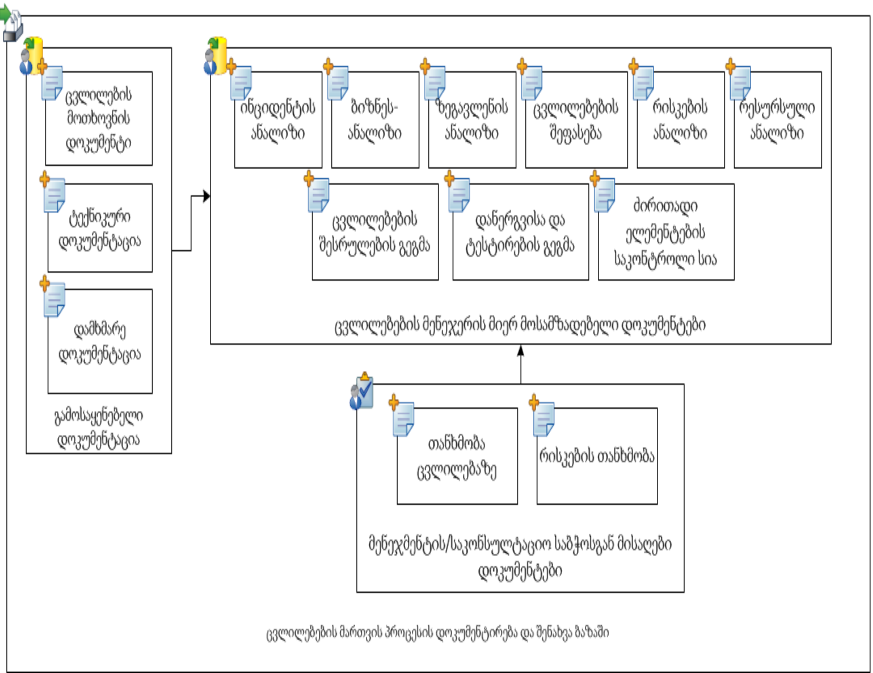
1-ელ ცხილში წარმოდგენილია ცვლილების განხორციელების დოკუმენტის შაბლონი.

**ცვლილების მოთხოვნის დოკუმენტი** **ცხრ.1**

| RFC ძირითადი ინფორმაცია | |
|---|---|
| ცვლილების ინიციატორი | RFC ნომერი |
| ცვლილების მენეჯერი | შესრულების ვადა |
| ბიზნეს გარემოზე გავლენა | IT სერვისებზე გავლენა |

| | |
|---|---|
| | |
| **RFC პრიორიტეტები** | |
| ☐ გადაუდებელი ☐ მაღალი ☐ საშუალო ☐ დაბალი | |
| **RFC კატეგორია** | |
| ☐ უმნიშვნელოვანესი ☐ მნიშვნელოვანი ☐ უმნიშვნელო ☐ სტანდარტული | |
| **RFC დეტალები** | |
| **ცვლილების ბიზნეს-საფუძველი**<br>ძირითადი მიზეზი, რის გამოც უნდა გატარდეს ცვლილება | |
| **ინციდენტის და/ან პრობლემის არსებობა** (თუ არსებობს) | **რა სერვისებზე იქონიებს გავლენას ცვლილება** |
| **ზეგავლენის ანალიზი**<br>კონფიგურაცია, DB, App, Service | |
| **ინფრასტრუქტურული პოლიტიკა, პროცედურები ან სტანდარტები ცვლილებისთვის**<br>ცვლილების გატარებასთან დაკავშირებული პროცედურების გეგმა | |
| **ცვლილებისთვის რესურების მოთხოვნა**<br>შემსრულებლები | **ცვლილების ღირებულება**<br>ბიუჯეტი |
| **დასტური** | |
| **გვარი, სახელი, თანამდებობა** | **თარიღი** |
| **უარყოფა** | |
| **გვარი, სახელი, თანამდებობა** | **თარიღი** |
| **უარყოფის მიზეზები** | |

IT ცვლილებების მართვის სრული სასიცოცხლო ციკლის განხორციელების თანმდევი და აუცილებელი პროცესია **დოკუმენტბრუნვა**. ცვლილებების განხორციელების ყოველი ეტაპი მოითხოვს მკაცრად და სტანდარტიზებულად წარმოებულ დოკუმენტირებას. 46-ე ნახაზზე მოცემულია IT ცვლილებების წარმოების დოკუმენტბრუნვის პროცესის სქემა.




ნახ. 46. IT ცვლილებების წარმოების დოკუმენტბრუნვის პროცესი

# 2 თავი: ლაბორატორიული პრაქტიკუმის ნაწილი

## 2.1. პროგრამული აპლიკაციის აგება დაპროგრამების რამდენიმე ენის საფუძველზე (.dll ფაილების შექმნა)

### ლაბორატორიული სამუშაო N 1

**მიზანი:** ობიექტორიენტირებული დაპროგრამების ენების: C++, Visual Basic და C# გამოყენებით ერთი პროგრამული პროექტის აგების ტექნოლოგიის შესწავლა .NET პლატფორმაზე. ამ ენებზე .dll ფაილების შექმნისა და მათი ერთად მუშაობის პროცესის გაცნობა.

### 1. თეორიული ნაწილი

ილუსტრირებულია .NET ტექნოლოგიის ერთ-ერთი ძირითადი პრინციპი, რომ პროგრამული აპლიკაციის დამუშავება შესაძლებელია დეველოპერების გუნდში სხვადასხვა პროგრამული ენის მცოდნე სპეციალისტების მიერ, კერძოდ, C++, Visual Basic და C# ენების საფუძველზე. პროექტის აგებისას გათვალისწინებულ უნდა იქნას საერთო მოთხოვნები ღია ცვლადებზე, მეთოდებსა და თვისებებზე.

სამუშაო გეგმა:

- შეიქმნას C++ -ის საბაზო კლასი;
- შეიქმნას Visual Basic.NET კლასი, რომელიც იქნება C++ -კლასის მემკვიდრე;
- შეიქმნას C# კლასი, რომელიც კონსოლის აპლიკაციაში შექმნის Visual Basic.NET კლასის ეგზემპლარსა და გამოიძახებს მის მეთოდს.

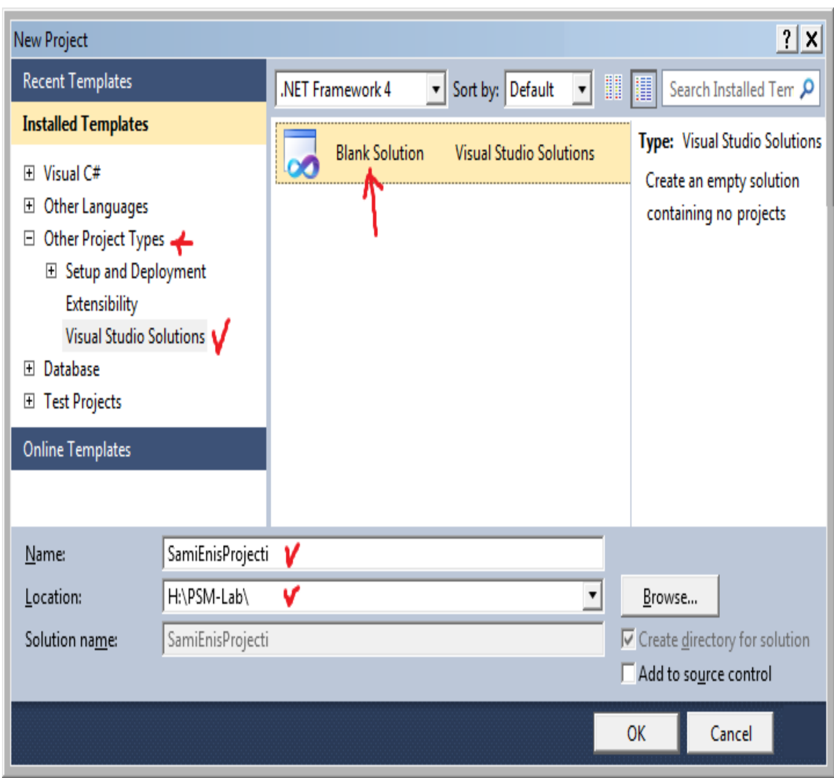
## 2. პრაქტიკული ნაწილი

განვახორციელოთ პროექტის პროგრამული რეალიზაცია Visual Studio .NET Framework 4.0 სამუშაო გარემოში (შესაძლებელია სხვა ვერსიის გამოყენებაც, მაგალითად, 4.5).

- შევარჩიოთ ახალი პროექტის განთავსების ადგილი
- მენიუდან გამოვიძახოთ ახალი ცარიელი პროექტის

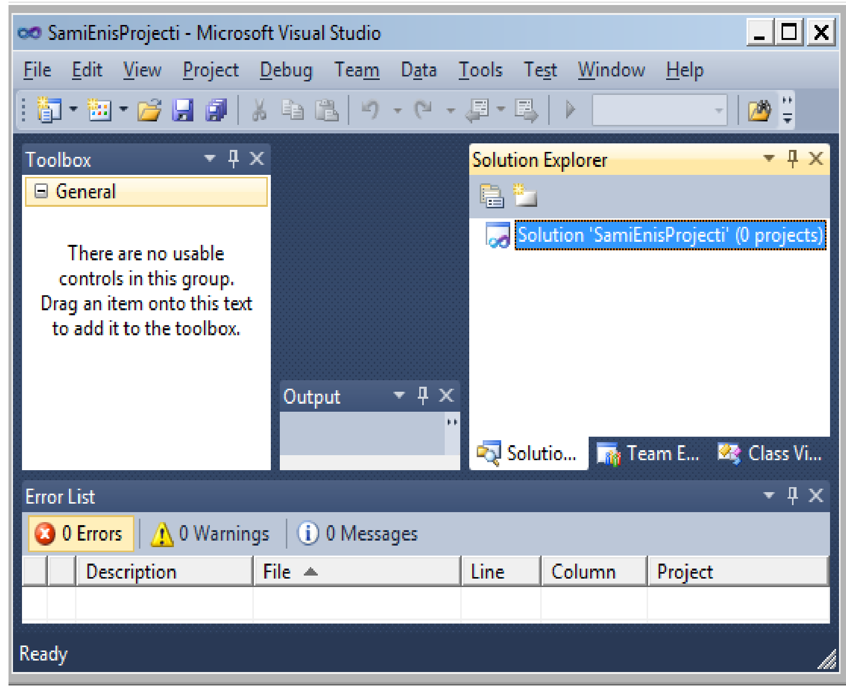
შექმნის პროგრამა:

(New->Project და Visual Studio Solution-> Blank Solution)




ნახ.1.1.

პროექტის სახელი (Name: SamiEnisProjecti) და მისი შენახვის ადგილი (Location) მივუთითოთ ჩვენი სურვილით. OK-ის შემდეგ Blank Solution შაბლონის დახმარებით შეიქმნება აპლიკაცია, რომელიც *ნეიტრალური* იქნება დაპროგრამების ენებისადმი. შედეგად მივიღებთ 1.2 ნახაზზე ნაჩვენებ ფანჯარას.



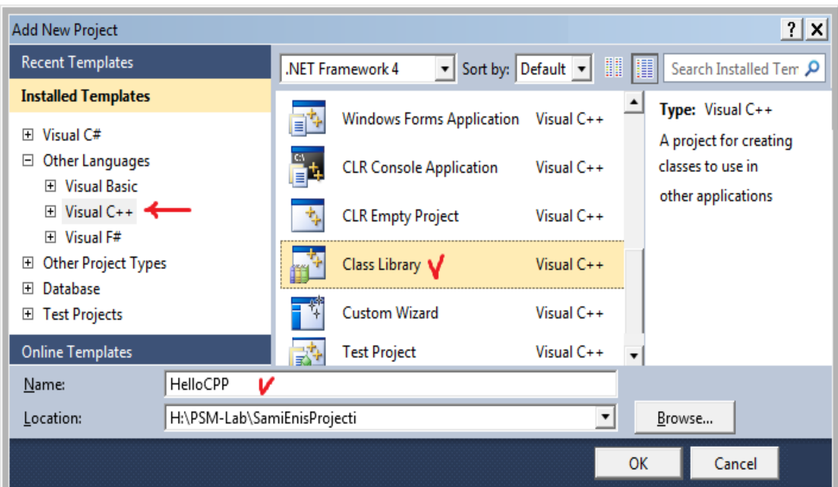

ნახ.1.2

- **კლასის შექმნა ახალ პროექტში C++.NET ენაზე**

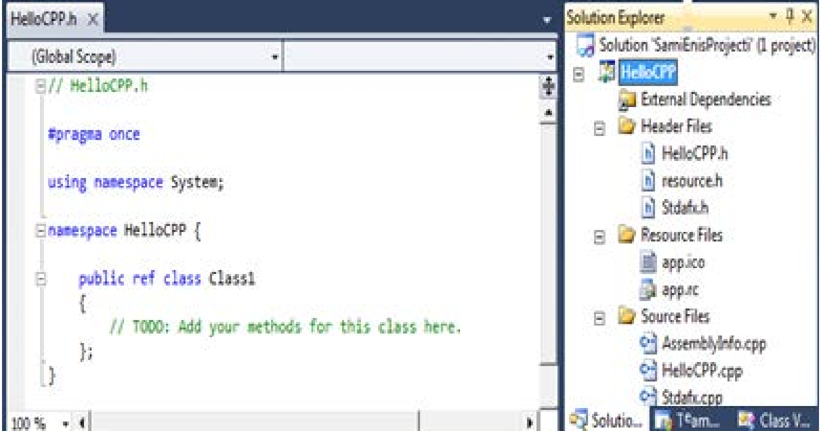
ცარიელ პროექტს (გადაწყვეტა - Solution 'SamiEnisProjecti') დავამატოთ ახალი (ქვე)პროექტი მასზე მარჯვენა ღილაკის დაწკაპუნებით და შემდეგ Add -> New Project არჩევით. 1.3 ნახაზზე ნაჩვენებია, თუ როგორ მიეთითება VisualC++ ენა, Class Library და ქვეპროექტის სახელი Name: HelloCPP, შემდეგ OK.

Solution Explorer -ში გამოჩნდება შედეგი (ნახ.1.4). გავხსნათ Hello.CPP.h ფაილი რედაქტირებისათვის. ახლადშექმნილი კლასის Class1 მაგივრად ჩავწეროთ სახელი HelloCPP.

HelloCPP.h კოდი მოცემულია 1-ელ ლისტინგში.




ნახ.1.3




ნახ.1.4

```
// ---- ლისტინგი_1 -- შეცვლილი HelloCPP.h ----
#pragma once
using namespace System;
namespace HelloCPP
{
  public ref class HelloCPP
   {
     // --- აქ ჩაემატება კლასის მეთოდი ----
     public:
        virtual void Hello()
         {
          Console::WriteLine("Mogesalmebit C++ enis
garemodan da !\n
                        viZaxeb Visual Basics !\n\n“);
         }
   };
}
```

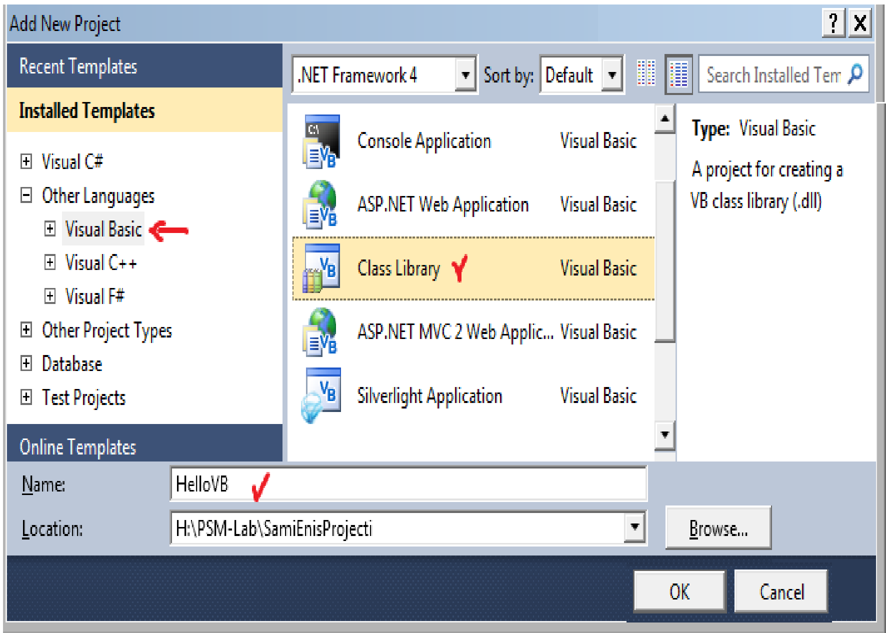
HelloCPP::Hello() მეთოდი გამოიყენებს .NET Framework კლასების ბიბლიოთეკიდან System::Console:: WriteLine() ფუნქციას, რათა კონსოლზე გამოიტანოს მისალმება C++ კოდიდან.

პროექტი გავუშვათ სინტაქსური შეცდომების გასას-წორებლად, თუ ასეთი იქნება. იგი ჯერ არ უნდა ავამუშავოთ შესასრულებლად.

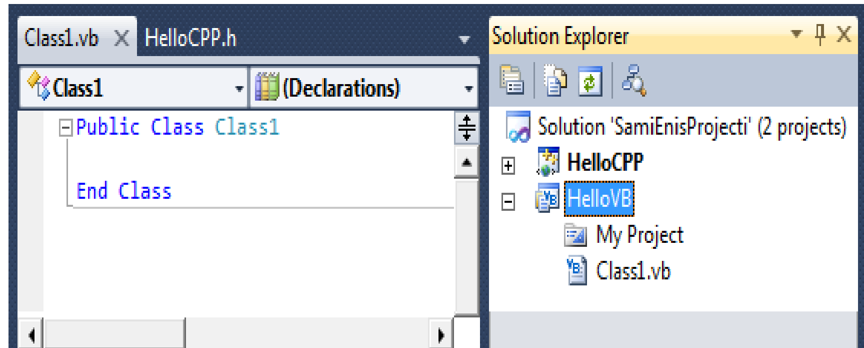
- **კლასის შექმნა ახალ პროექტში Visual Basic .NET ენაზე**

დავამატოთ Solution-ში ახალი პროექტი და მივუთითოთ, რომ იგი შეიქმნას Visual Basic .NET -ში სახელით Name: HelloVB (ნახ.1.5).

მივიღებთ ასეთ შედეგს (ნახ.1.6). ახალი პროექტი HelloVB დაემატება Solution Explorer პანელზე თავისი შემადგენელი ფაილებით.




ნახ.1.5




ნახ.1.6

ახლა უკვე გვაქვს HelloCPP და HelloVB ორი პროექტი.

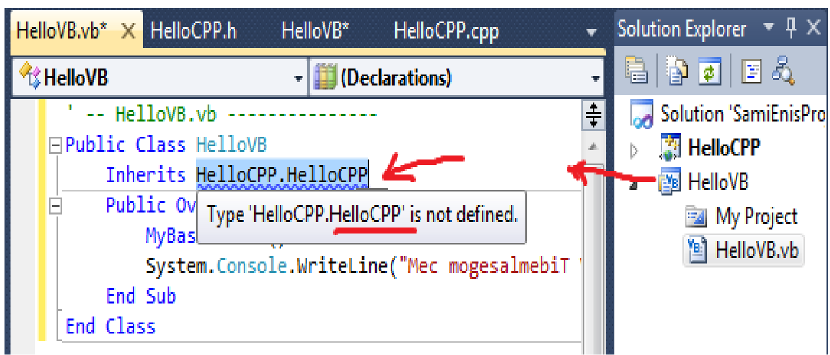
მეორეში ავირჩიოთ Class1.vb ფაილი და შევუცვალოთ სახელი შინაარსის გათვალისწინებით, მაგალითად, HelloVB.vb, შემდეგ გავხსნათ იგი და ჩავწეროთ ჩვენთვის საჭირო ტექსტი (ლისტინგი_2).

```
' --ლისტინგი_2 ---  HelloVB.vb ------------
Public Class HelloVB
   Inherits HelloCPP.HelloCPP
   Public Overrides Sub Hello()
      MyBase.Hello()
      System.Console.WriteLine("Mec mogesalmebiT
                                                 Visual Basic.NET-idan !!")
    End Sub
End Class
```

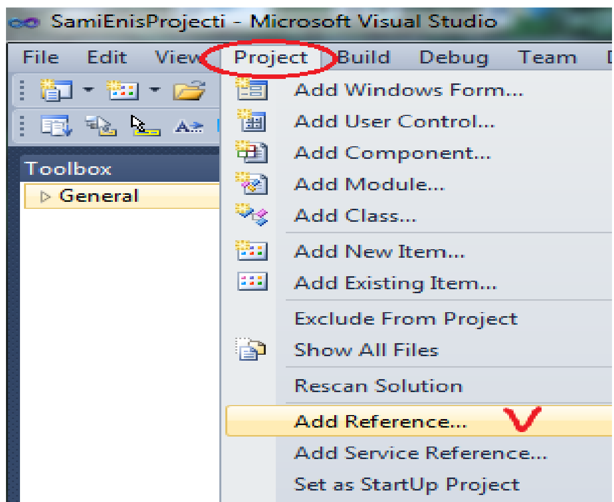
ეს კოდი განსაზღვრავს HelloVB კლასს, რომელიც მემკვიდრეობით იქნა მიღებული C++ ის მმართველი კლასიდან HelloCPP. ამგვარად, HelloVB კლასი ჩაანაცვლებს (Overrides) მშობელი HelloCPP კლასის ვირტუალურ Hello() მეთოდს, ოღონდ ჯერ გამოიძახებს Hello() მეთოდის ვერსიას მშობელი კლასიდან HelloCPP, შემდეკ კი გამოყავს ეკრანზე თავისი მისალმება.

საყურაღებოა, რომ კოდის რეადქტორში საბაზო კლასის სახელი (HelloCPP.HelloCPP) და ჩასანაცვლებელი მეთოდის სახელი Hello() ტალღისებური ხაზით. ეს ნიშნავს, რომ კოდის რედაქტორი ამ სახელებს ვერ ხედავსა და წინასწარ იძლევა სინტაქსურ შეცდომას. თუ კურსორს დავაყენებთ ამ ფრაგმეტზე, იგი მოგვცემს შეცდომის მნიშვნელობას (ნახ.1.7).

აუცილებელია Visual Basic-ის კომპილატორს მიეთითოს თუ სად იმყოფება ეს ნაკრები (assembly), რომელიც განსაზღვრავს მოცემულ ტიპს. ამისათვის მთავარი მენიუს Project-> Add Reference->Projects-დან (ნახ.1.8) ავირჩევთ HelloCPP-სა და OK.

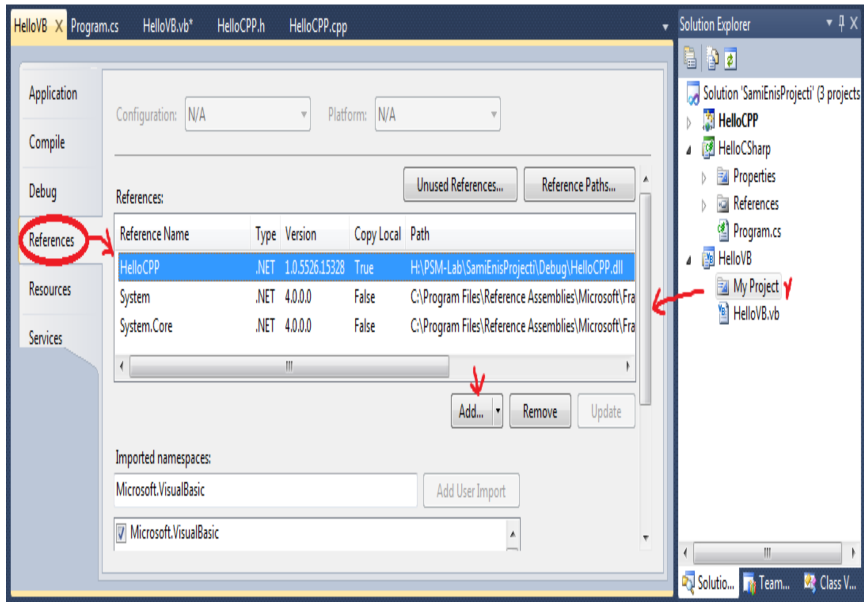



ნახ.1.7



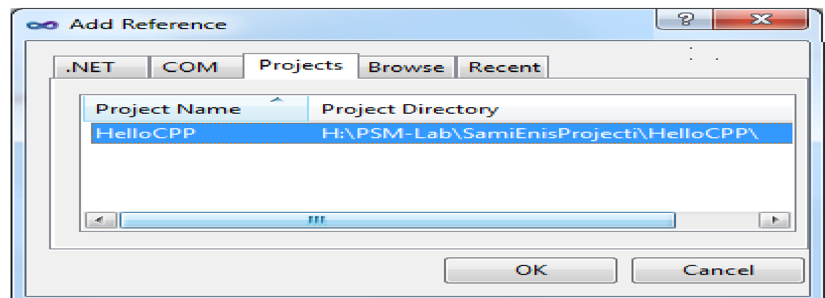

ნახ.1.8.

Solution-ში HelloVB-ს MyProject-ის ამოქმედებით გაიხსნება 1.9 ნახაზის მსგავსი ფანჯარა, რომელშიც References-ზე გადართვით გაჩნდება შიგთავსი. ვირჩევთ HelloCPP და Add-ით ვადასტურებთ.




ნახ.1.9

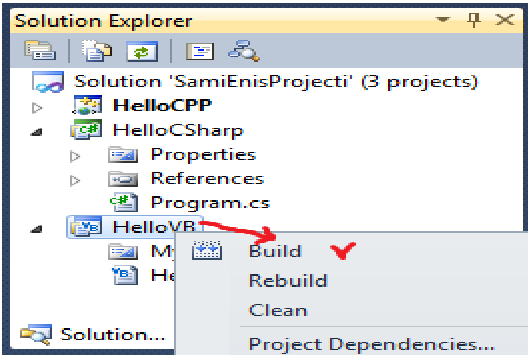
ამის შემდეგ გამოჩნდება Add Reference (ნახ.1.10) და OK ღილაკით ვასრულებთ CPP-კლასთან დაკავშირების პროცესს.




ნახ.1.10

ამის შემდეგ საჭიროა HelloVB პროექტზე მარჯვენა ღილაკით გამოვიტანოთ 1.11 კონტექსტური მენიუ და ავირჩიოთ Build (ეს

პროექტი ტრანსლატორით ამუშავდება და სინტაქსური გამართვის შედეგს მოგვცემს).

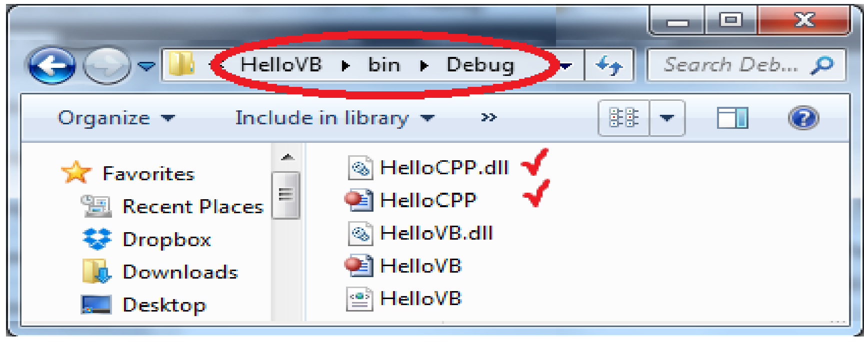



ნახ.1.11

როგორც წესი, თუ შეცდომები არაა HelloVB პროექტში, მაშინ მის შესაბამის ფოლდერის bin->Debug -ში უნდა გამოჩნდეს დაკავშირებული Helli.CPP.dll კლასის ფაილი (ნახ.1.12).

ამგვარად, Hello.CPP პროექტი მიუერთდება გადაწყვეტას (Solution-ში) და გამოჩნდება მომავალში HelloVB-ში, როგორც ზემოთ იქნა ნაჩვენები. ამასთანავე, 1.7 ნახაზზე ნაჩვენები „ქვეშგახაზული“ შეცდომების აღნიშვნები HelloVB.vb პროგრამის ტექსტიდან გაქრა ! პროგრამის საბოლოო ტექსტი მოცემულია ლისტინგში:

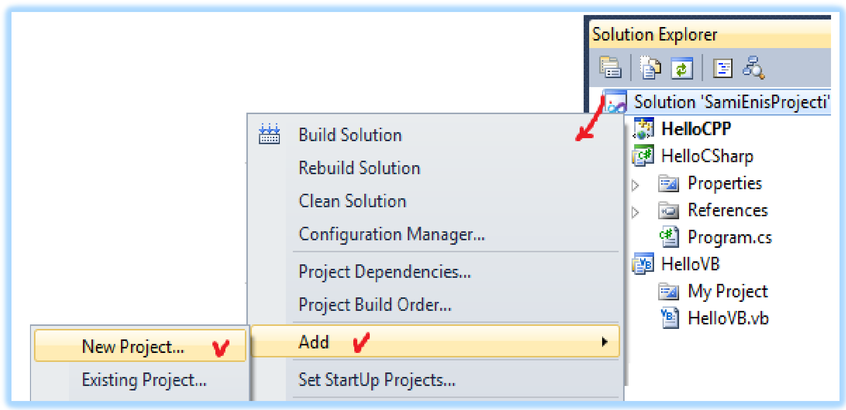
```
' -- HelloVB.vb ---------------
Public Class HelloVB
    Inherits HelloCPP.HelloCPP
    Public Overrides Sub Hello()
        MyBase.Hello()
        System.Console.WriteLine("Mec mogesalmebiT Visual
Basic.NET -idan !!!")
    End Sub
End Class
```



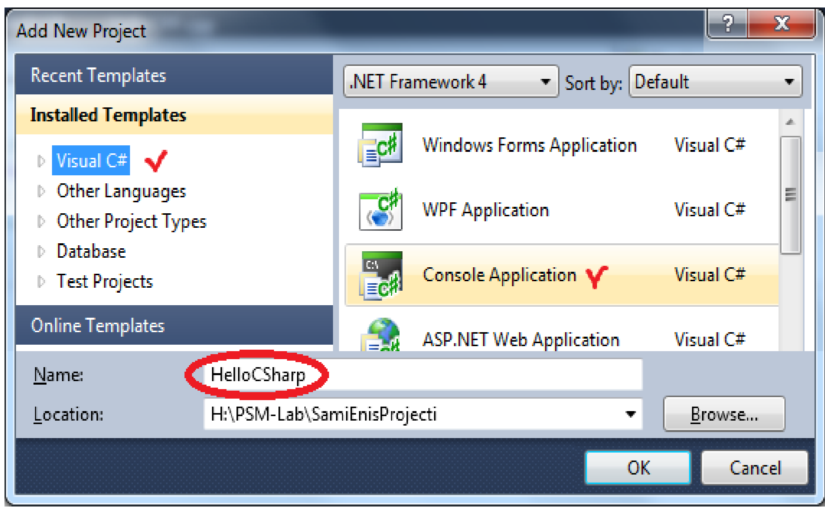

ნახ.1.12

- **სასტარტო პროექტის დამატება Solution-ში C#.NET ენაზე**

დასკვნით ფაზაში ჩვენი გადაწყვეტისათვის (Solution-ში) SamiEnisProjecti უნდა შევქმნათ მესამე, მთავარი სასტარტო პროექტი, რომელიც აგებული იქნება C#.NET ენაზე Console Application შაბლონის სახით. დავარქვათ მას HelloCSharp. ამგვარად, ვამატებთ პროექტს (ნახ.1.13-ა,ბ) `HelloCSharp` სახელით.

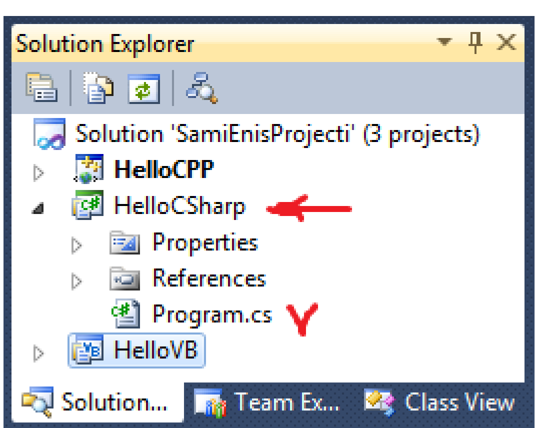



ნახ.1.13-ა




ნახ.1.13-ბ

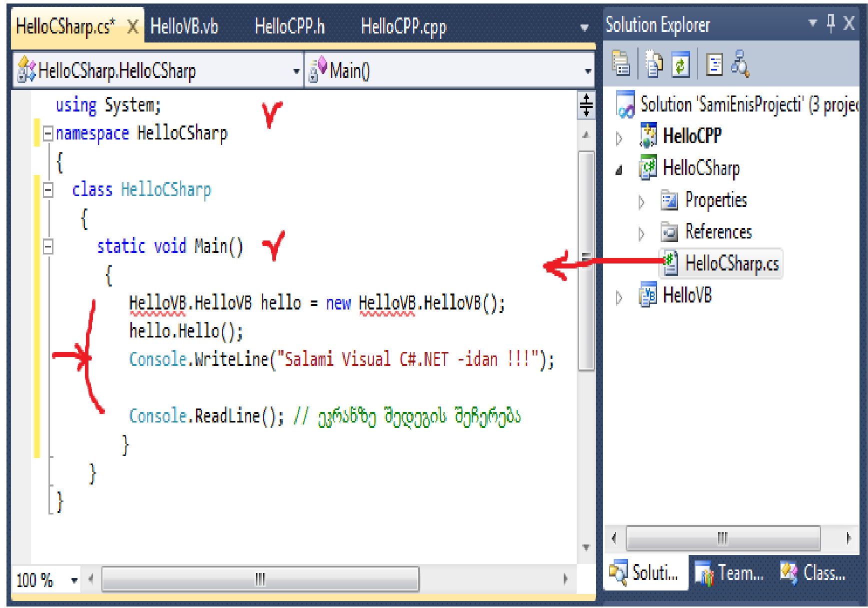
OK ღილაკის ამოქმედების შემდეგ მივიღებთ Solution-ში ახალ, HelloCSharp პროექტს თავისი შემადგენელი კომპონენტებით (ნახ.1.14).




ნახ.1.14

გადავარქვათ სახელი Program.cs ფაილს ჩვენი პროექტის შინაარსის შესაბამისად: HelloCSharp.cs.

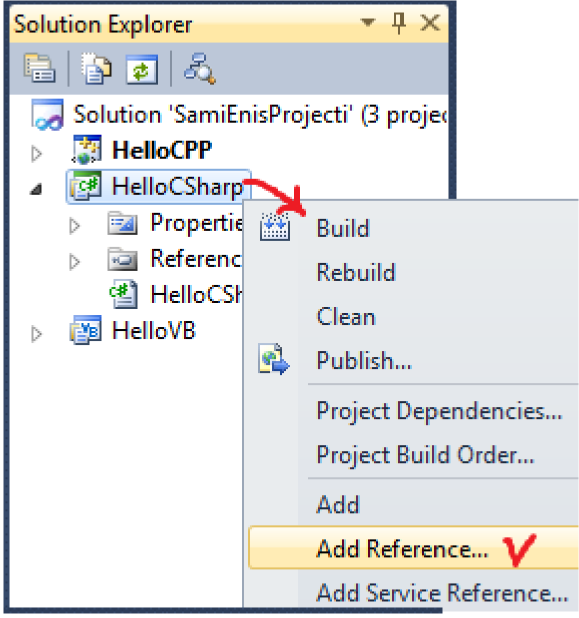
გამოვიტანოთ ეს ფაილი რედაქტორების ფანჯარაში და ჩავამატოთ Main() -ში შესაბამისი სტრიქონები. ასევე შეიძლება Main() -ის არგუმენტების წაშლა, აქ არ გვჭირდება. რედაქტორების შემდეგ ტექსტი მოცემულია 1.15. ნახაზზე.




**ნახ.1.15**

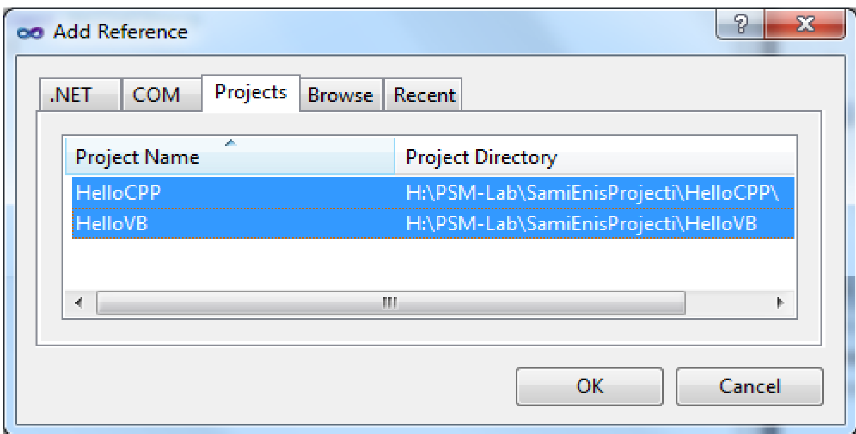
პროგრამაში Main() სტატიკური მეთოდი არის აპლიკაციაში შესვლის წერტილი. ეს მეთოდი ქმნის HelloVB კლასის ახალ ობიექტს hello და მისთვის იძახებს Hello() მეთოდს, რომელშიც ინახება ორი შეტყობინება. ერთი CPP.NET-იდან, მეორე VB.BET-დან, რომლებიც ადრე მოვამზადეთ. მესამე შეტყობინებას თვით C#.NET გამოიტანს, დამშვიდობებასთან ერთად.

ახლა საჭიროა HelloCSharp პროექტში ჩავამატოთ მიმთითებლები (კავშირები) HelloCPP და HelloVB პროექტებზე. მაშინ 1.15 ნახაზზე ნაჩვენები შეცდომები (ქვეშხაზგასმული HelloVB) გასწორდება. ამისათვის ვირჩევთ Add Reference...-ს (ნახ.1.16).

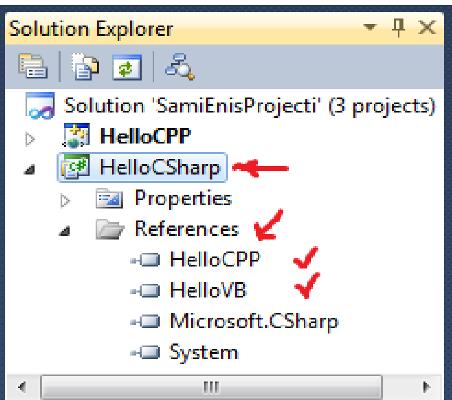



ნახ.1.16

Add Reference ფანჯარაში გადავრთოთ Project-ზე, მოვნიშნოთ ორივე, HelloCPP და HelloVB პროექტები და ღილაკი OK (ნახ.1.17). შედეგად გაქრება შეცდომები, ანუ 1.15 ნახაზზე „ხაზები", რაც მიუთითებს იმაზე, რომ კავშირი პროექტებს შორის კარგად შესრულდა. ეს შედეგი აისახება Solution-ფანჯარაში HelloCSharp პროექტის References –ში (ნახ.1.18).

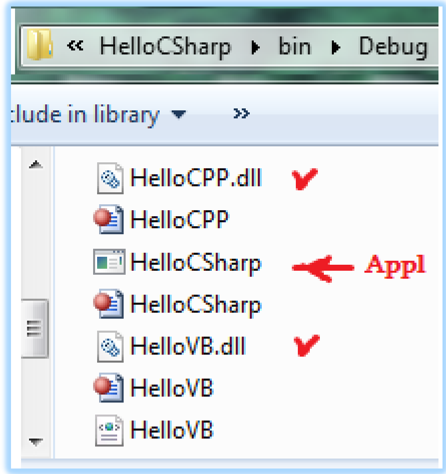



ნახ.1.17



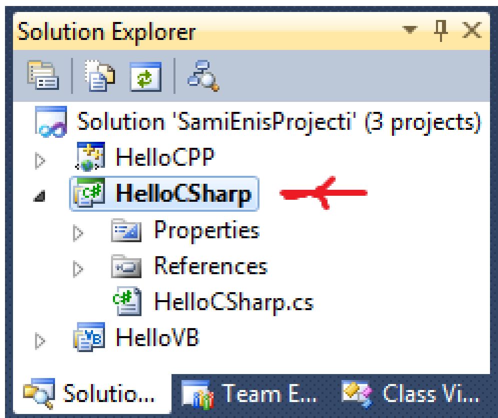

ნახ.1.18

ავამუშავოთ HelloCSharp პროექტი მარჯვენა ღილაკით და build-ით, რათა სინტაქსურად დამუშავდეს იგი. შეცდომების არარსებობის შემთხვევაში მოხდება HelloCPP.dll და HelloVB.dll ფაილების განთავსება HelloCSharp პროექტის bin->Debug ფოლდერში (ნახ.1.19). აქვეა HelloCSharp.exe აპლიკაციის ფაილიც.

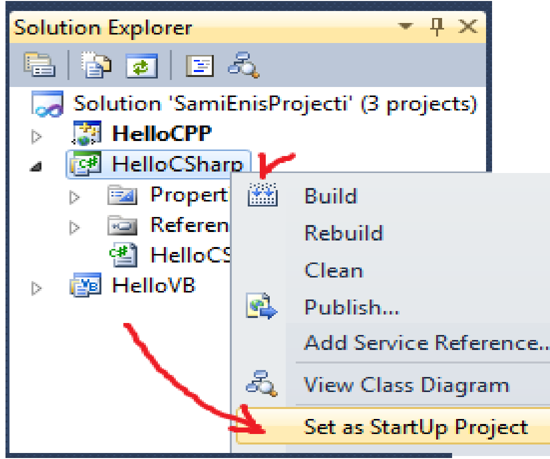



ნახ.1.19

Solution-ში HelloCSharp პროექტი გადავაკეთოთ სასტარტო ფაილად (ნახ.1.20-ა,ბ).

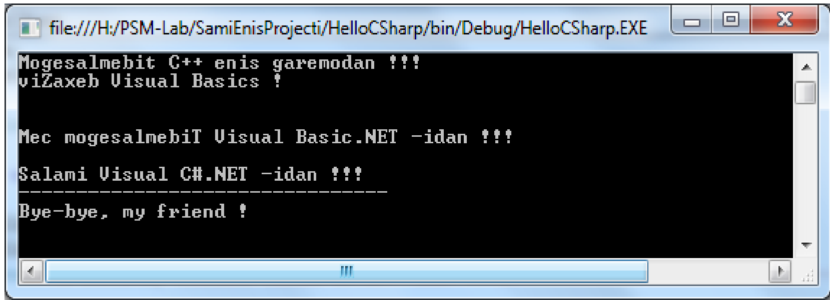



ნახ.1.20-ა




ნახ.1.20-ბ

ბოლოს, ავამუშავოთ აპლიკაცია და მივიღებთ შედეგს (ნახ.1.21).




ნახ.1.21

**რეზიუმე:**

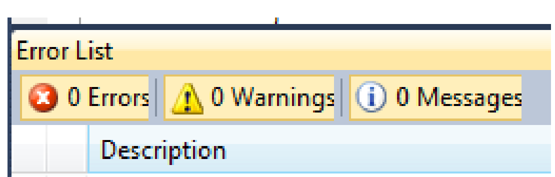
აპლიკაციის სხვადასხვა პროექტი დამუშავდა დაპროგრამების სხვადასხვა ენებზე, რომლებიც გამოიყენება .NET გარემოში. მრავალჯერადი გამოყენების ფაილები შემუშავებულ იქნა კლასების დახმარებით და რეალიზებულია დინამიკური .dll - ფაილების სახით.

## 2.2. გამონაკლის შემთხვევათა აღმოჩენისა და გამორიცხვის ვიზუალური საშუალებები

### ლაბორატორიული სამუშაო N2

როგორც ცნობილია, პროგრამის გამართვისა და ტესტირების (შესრულების) პროცესში ადგილი აქვს პროგრამული შეცდომების გამოვლენას. ეს შცდომები სამი სახისაა: სინტაქსური, პროცედურული და ლოგიკური.

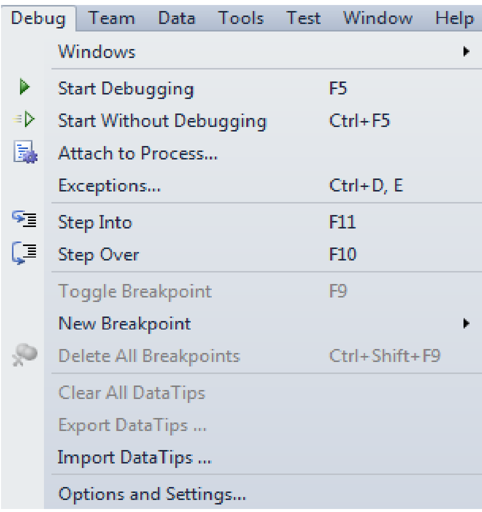
სინტაქსური შეცდომების აღმოჩენა ხდება C#-ენის კომპილატორის საშუალებით და გამოიტანება პროგრამული ტექსტების რედაქტორის ქვედა ნაწილში, Error List ფანჯარაში (ნახ.2.1). მათი პოვნა და შესწორება შედარებით ადვილია.




ნახ.2.1

პროცედურული შეცდომები ვლინდება პროგრამის შესრულების პროცესში, როცა პროგრამაში აღარაა სინტაქსური შეცდომები და ხდება მისი ამუშავება: Start Debugging ან F5 (ნახ.2.2), შესაძლებელია ისეთი შეცდომის გამოვლენა, რომელიც წყვეტს პროგრამის შესრულების პროცესს (ანუ სრულდება ავარიულად).

დებაგერს გამოაქვს ამ დროს სათანადო შეტყობინება (ნახ.2.3), რომელიც პროგრამისტის მხრიდან მოითხოვს ანალიზსა და შეცდომის გამორიცხვას (Exception Handling).




ნახ.2.2

```
if (i == 1)
 {
    trackBar2.Value = SachiroSartuli;
    Thread.Sleep(500);
    label17.Text = trackBar2.Value.ToS
 }
else
   if (i == 2)
   {
     trackBar3.Value = SachiroSartuli;
     Thread.Sleep(500);
      label18.Text = trackBar3.Value.
   }
   else
```

ArgumentOutOfRangeException was unhandled

Value of '51' is not valid for 'Value'. 'Value' should be between 'Minimum' and 'Maximum'.
Parameter name: Value

Troubleshooting tips:

Make sure the arguments to this method have valid values.
If you are working with a collection, make sure the index is less than the size of the collectio
When using the overloaded two-argument FindString or FindExactString methods with a Cter.
Get general help for this exception.

Search for more Help Online...

Actions:

View Detail...
Copy exception detail to the clipboard

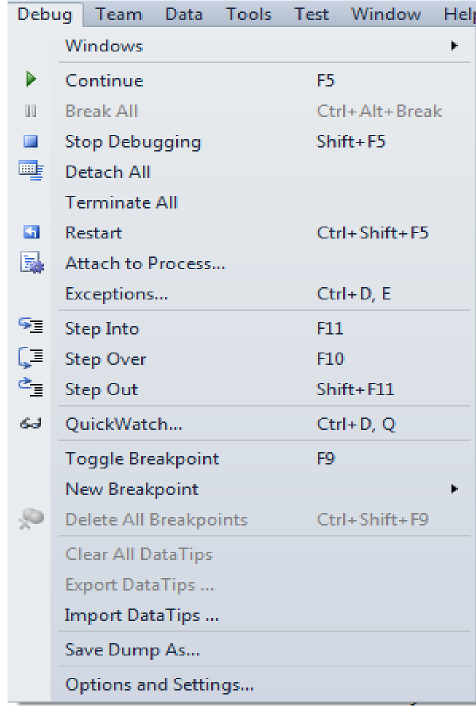
0 Messages

ნახ.2.3

ლოგიკური შეცდომების აღმოჩენა შედარებით რთულია. პროგრამა ამ დროს მუშაობსა და სრულდება ნორმალურად (არაავარიულად), მაგრამ შედეგები „საეჭვოა“.

ტესტირების პროცესში, რომელიც აუცილებლად მოსდევს პროგრამის გამართვას, საჭიროა ასეთი „კვლევის“ ჩატარება, რათა გამოვლენილ იქნას მოსალოდნელი, ფარული ლოგიკური შეცდომები.

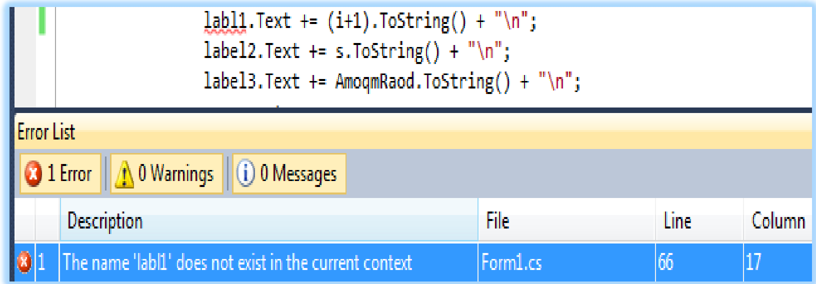
C# ენის რედაქტორს აქვს კარგი ინსტრუმენტული საშუალებები (Debugger) ამ ტიპის შეცდომების მოსაძებნად და აღმოსაფხვრელად (ნახ.2.4).




ნახ.2.4

## 2.1) სინტაქსური შეცდომების აღმოფხვრის საშუალებანი

თუ პროგრამის არაკორექტულ კოდში შეცდომითაა ჩაწერილი ენის ოპერატორი (მაგალითად, “Whail” ნაცვლად while - ისა) ან კონსტრუქცია (მაგალითად, “case: “, რომელსაც არ უძღვის წინ switch() {…}) და ა.შ. როგორც ზემოთ აღვნიშნეთ, ამ დროს კომპილატორი მიუთითებს არსებულ შეცდომასა და მის ადგილმდებარეობას (ნახ.2.5).




ნახ.2.5

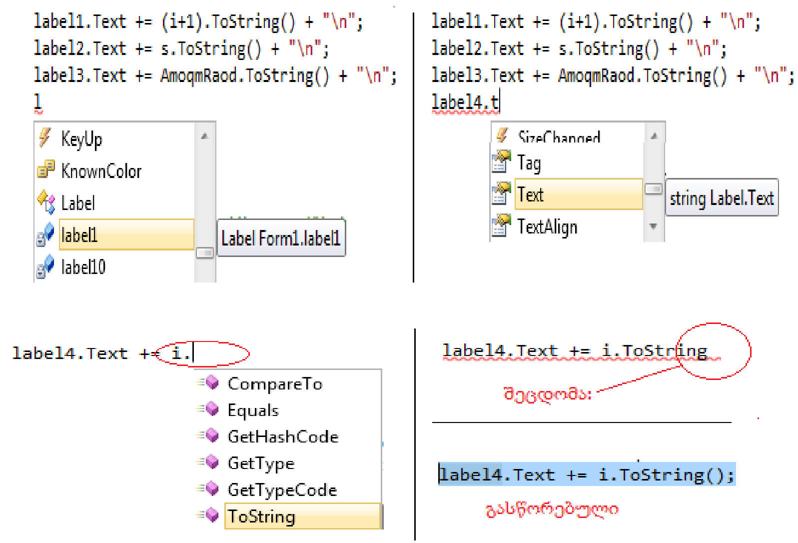
სინტაქსური შეცდომები თუ არ გასწორდა, მაშინ ვერ მოხერხდება პროგრამის ობიექტური (.obj) და შესრულებადი (.exe) კოდების ფორმირება.

სინტაქსური შეცდომების თავიდან ასაცილებლად C# ენის რედაქტორს აქვს სხვადასხვა ვიზუალური დამხმარე საშუალება. მაგალითად, პროგრამაში ოპერატორების ტექსტის შეტანისას, ან ობიექტის მეთოდისა და მოვლენის არჩევისას (წერტილის „.“ დასმისას) ხდება ვიზუალური ბლოკის (Intellisense - ავტოდამატება) შემოთავაზება (ნახ.2.6), საიდანაც ამოირჩევა საჭირო სიტყვა და Enter-კლავიშით სწრაფად ჩაისმება მითითებულ ადგილას.

ეს გამორიცხავს როგორც ოპერატორის (ობიექტის თვისების, მეთოდისა და ა.შ.) არასწორ სინტაქსურ ჩაწერას, ისე

არარელევანტური სიტყვის მითითებას (სიტყვა, რომელიც აქ „უადგილოა“).

შესაძლებლია აგრეთვე კოდში „გახსნილ-დასახურ“ ფრჩხილების რაოდენობის კონტროლი, რაც ძალზე ხშირი შეცდომების წყაროა. მთლიანად, შეიძლება ითქვას, რომ ენის ასეთი ვიზუალური კონტროლისა და დამხმარე საშუალებები პროგრამისტის მუშაობას ეფექტურს ხდის.



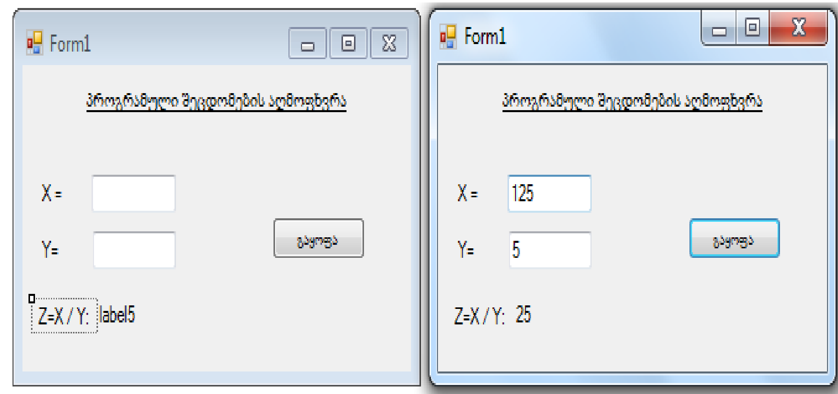

ნახ.2.6

## 2.2) განსაკუთრებული შემთხვევები: შეცდომები პროგრამის შესრულებისას და მათი აღმოფხვრა

თუ პროგრამის ტექსტი სინტაქსური შეცდომებისაგან თავისუფალია, ის შეიძლება ამუშავდეს შესარულებლად. ამ დროს შესაძლებელია ისეთი შეცდომების გამოვლენა, რომლებიც პროგრამას ავარიულად დაასრულებს ან საერთოდ არ დაასრულებს („გაჭედავს“).

მაგალითად, უსასრულო ციკლი ან სხვ. განსაკუთრებული შემთხვევა. განვიხილოთ ასეთი მაგალითები:

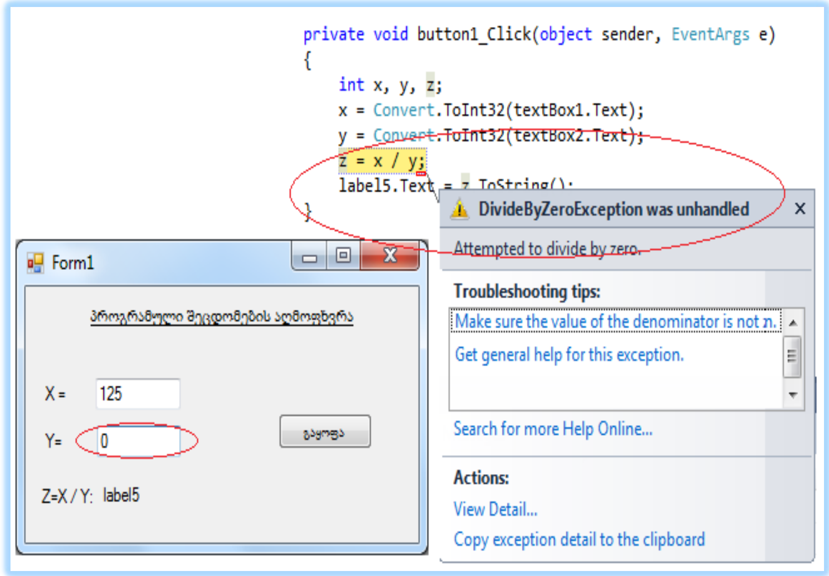
**ამოცანა_2.1:** ავაგოთ პროგრამის კოდი, რომელიც შეასრულებს მთელი რიცხვების შეტანასა და გაყოფის ოპერაციას. 2.7 ნახაზზე ნაჩვენებია ფორმა ორი ტექსტბოქსით (შესატანად), label5 შედეგის გამოსატანად და ღილაკი „გაყოფა“ პროგრამული კოდით (ნახ.2.8).



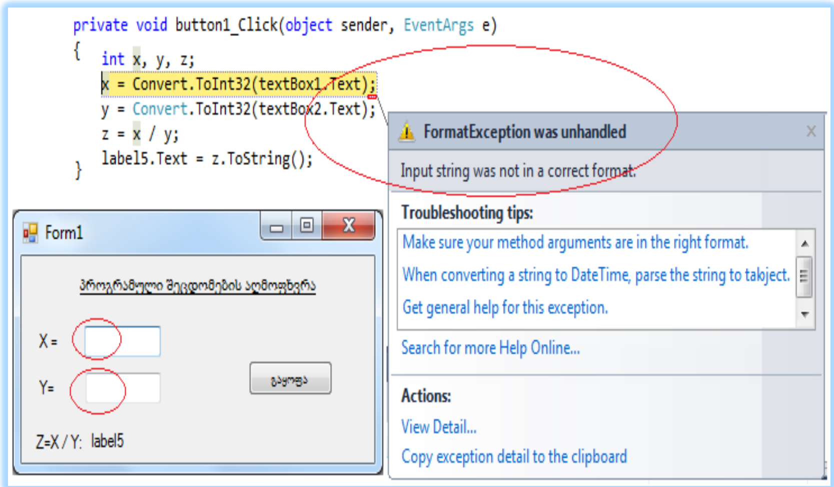

ნახ.2.7

როგორც ვხედავთ, პროგრამა მუშაობს თითქოს ნორმალურად, ასრულებს გაყოფას. ტესტირების პროცესში, რომელიც გულისხმობს კოდის ფუნქციის გამოკვლევას საწყისი მონაცემების სხვადსხვა მნიშვნელობისათვის, ვიღებთ „ნულზე გაყოფის“ შეცდომას (ნახ.2.8).

პროგრამაში საჭიროა ამ სიტუაციის გათვალისწინება (მომხმარებელმა ყოველთვის შეიძლება შეიტანოს შემთხვევით ან „არცოდნის“ გამო 0 !). ანუ თუ იქნება შეტანილი „0“, მაშინ პროგრამამ „გვერდი აუაროს“ (გამორიცხოს, აღმოფხვრას) ასეთი ტიპის შეცდომა და თან შეატყობინოს მომხმარებელს, რომ შეიტანოს კორექტული რიცხვი (0-საგან განსხვავებული) (ნახ.2.9).

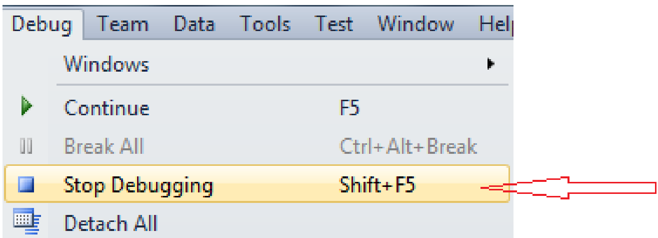



ნახ.2.8




ნახ.2.9

რედაქტირების რეჟიმში გადასასვლელად საჭიროა მენიუდან „დებაგერის შეჩერება“ (ნახ.2.10).

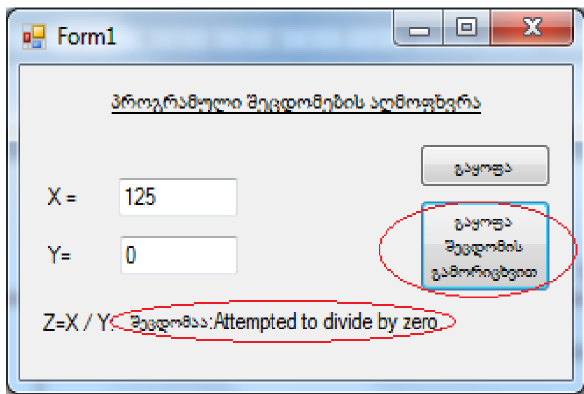



**ნახ.2.10**

ახლა შესაძლებელია ზემოაღწერილი ტიპის შეცდომებისათვის გამორიცხვის პროცედურის კოდის ფორმირება. მაგალითად, 2_1 ლისტინგზე მოცემულია ასეთი კოდი.

```
 //ლისტინგი_2.1 --- Exception -----
private void button2_Click(object sender, EventArgs e)
{
  int x, y, z;
  try
   {
     x = Convert.ToInt32(textBox1.Text);
     y = Convert.ToInt32(textBox2.Text);
     z = x / y;
     label5.Text = z.ToString();
   }
  catch (Exception nul_Div)   // ობიექტი nul_Div
   {
    label5.Text = "შეცდომაა:" + nul_Div.Message;
   }
}
```

2.11 ნახაზზე ნაჩვენებია ამ კოდის მუშაობის შედეგი, რომელიც მოთავსებულია ღილაკზე „გაყოფა შეცდომის გამორიცხვით“.



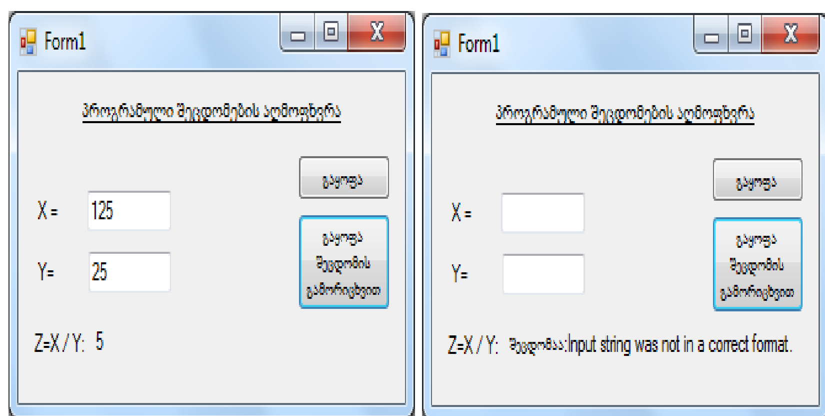

**ნახ.2.11**

**საყურადღებო:** გამოყენებულია კონსტრუქცია try { }… catch{ }. საკვანძო სიტყვა try ინიცერებას უკეთებს გამორიცხვის მექანიზმს. ამ წერტილიდან პროგრამა იწყებს {..} ბლოკის შესრულებას. ამ ბლოკის ოპერატორების შესრულებისას თუ გაჩნდა გამორიცხვის (Exception) შემთხვევა, მაშინ მას „დაიჭერს“ catch და შეცვლის გამორიცხვის სიტუაციას თავის {...} ბლოკით.

ჩვენ შემთხვევაში catch ბლოკში მოთავსებულია Exception კლასის ობიექტი (`nul_Div`), რომელიც შეიცავს ინფორმაციას აღმოცენებული შეცდომის შესახებ. Message თვისებით ხდება შეტყობინების გამოტანა ეკრანზე. პროგრამა ბოლომდე სრულდება არაავარიულად.

2.12 ნახაზზე მოცემულია try…catch - გამორიცხვის მექანიზმით შესრულებული პროგრამული კოდის შედეგები: როცა რიცხვები შეტანილია ნორმალურად გაყოფის ოპერაციისათვის (ამ დროს არ ხდება „შეცდომის“ დაფიქსირება try-ში), და მეორე, როცა

არასწორადაა შეტანილი საწყისი მონაცემები (აქ იმუშავებს შეცდომების გამორიცხვის ბლოკი).




**ნახ.2.12**

განსაკუთრებულ შემთხვევათა დამუშავების try…catch მექანიზმი შეიძლება გაფართოვდეს ბიბლიოთეკაში არსებული Exception-კლასის საფუძველზე. აქ იგულისხმება სპეციფიური შეცდომების აღმოჩენის შესაძლებლობა, მაგალითად, ტიპების გარდაქმნისას, ნულზე გაყოფისასა და ა.შ. თუ შეცდომის სახე წინასწარ არაა განსაზღვრული, მაშინ გამოიყენება ზოგადი Exception კლასის ობიექტი.

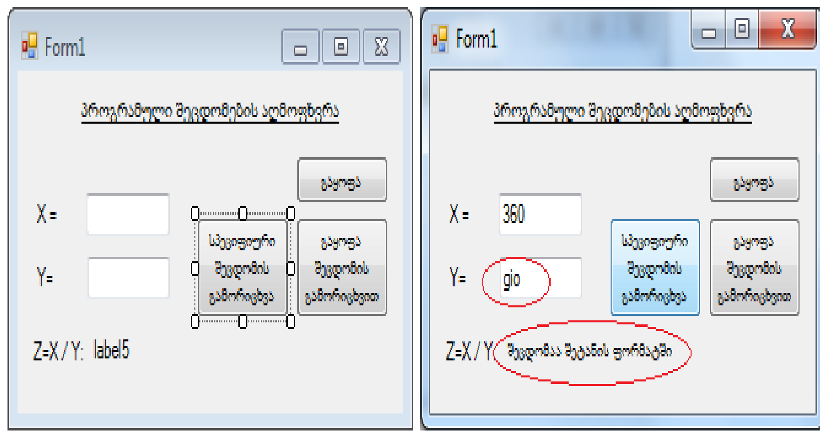
2.2_ლისტინგში მოცემულია ღილაკის „შეცდომის გამორიცხვა“ შესაბამისი კოდის ფრაგმენტი.

```
// ლისტინგი_9.2 ---- - სპეციფიური და ზოგადი შეცდომები ---------
private void button3_Click(object sender, EventArgs e)
 {
  int x, y, z;
  try
   {
     x = Convert.ToInt32(textBox1.Text);
```

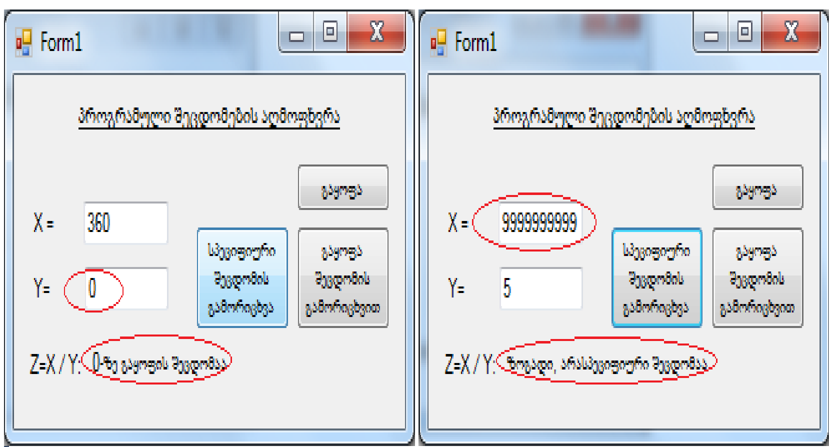
```
    y = Convert.ToInt32(textBox2.Text);
    z = x / y;
    label5.Text = z.ToString();
  }

catch(FormatException nul_Div) //ტიპის გარდაქმნის შეცდომა
  {
    label5.Text = "შეცდომაა შეტანის ფორმატში\n" +
                                                nul_Div.Message;
  }

catch(DivideByZeroException nul_Div) // 0-ზე გაყოფის შეცდ.
  {
    label5.Text = "0-ზე გაყოფის შეცდომაა\n" +
                                              nul_Div.Message; ;
  }

catch (Exception nul_Div)  // ზოგადი შეცდომა
  {
   label5.Text = "ზოგადი, არასპეციფიური შეცდომაა\n"+
                                                nul_Div.Message;
  }
}
```

ლისტინგში catch{...} ბლოკების მიმდევრობას აქვს მნიშვნელობა (თუ სრულდება პირველი, წყდება პროცესი. თუ არა, გადადის შემდეგზე).

შედეგები ასახულია 2.13-ა,ბ ნახაზებზე.




ნახ.2.13-ა



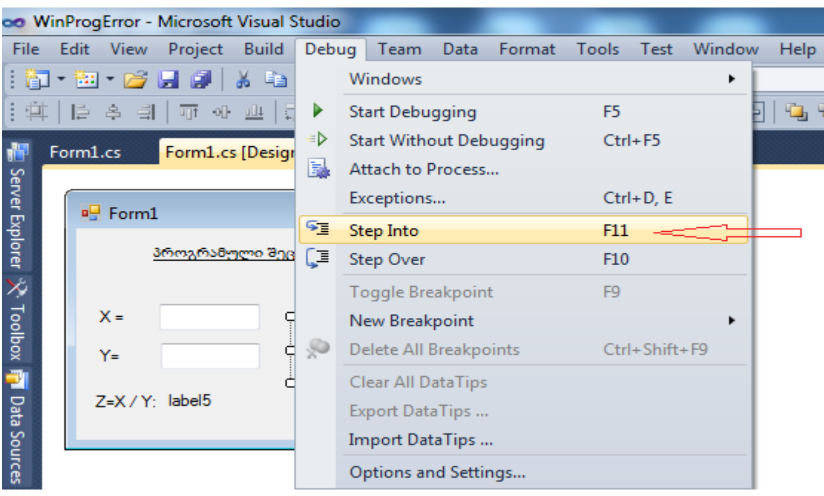

ნახ.2.13-ბ

## 2.3) ლოგიკური შეცდომები და პროგრამის გამართვა (Debugging) ბიჯური შესრულების რეჟიმში

როგორც აღვნიშნეთ, ლოგიკური შეცდომები მაშინ აღმოჩნდება, როდესაც სინტაქსური და შესრულების პროცესის შეცდომები აღარაა, მაგრამ სასურველ (დაგეგმილ სავარაუდო) შედეგს პროგრამა არ გვაძლევს.

ეს ნიშნავს, რომ პროგრამის აგების ლოგიკა არასწორია!

ასეთი შეცდომების აღმოჩენა საკმაოდ რთულია და მოითხოვს ტესტირების პროცესისა და პროგრამის შესრულების მიმდევრობის შედეგების ანალიზს. ვიზუალური C# ენა ფლობს პროგრამის გამართვის (Debugging) კარგ დამხმარე საშუალებებს. განვიხილოთ ისინი ჩვენი WinProgError პროექტის მაგალითზე (ნახ.2.14). გავხსნათ პროექტი, მოვამზადოთ Form1 ფორმა და მენიუს Debug-ში ავირჩიოთ Step Into (ან F11 ღილაკი კლავიატურის ზედა რიგში).




ნახ.2.14

ჩაირთვება პროგრამის გამართვის ბიჯური (Step Into) რეჟიმი. ეკრანზე დიზაინის ფორმა შეიცვლება (იხ. Solution Explorer) Program.cs დამწყები პროგრამის ტექსტით, რომელშიც მოთავსებულია Main() მთავარი ფუნქცია (ნახ.2.15). მარცხნივ ჩანს ყვითელი ისარი, რომელიც F11-ით ბიჯურად გადაადგილდება იმ სტრიქონზე, რომელიც სრულდება მოცემულ მომენტში.

```
Form1.Designer.cs   Program.cs*   Form1.cs   Form1.cs [Design]
WinProgError.Program

// Program.cs - Main() - დამწყები პროგრამის ტექსტი ----
using System;
using System.Windows.Forms;

namespace WinProgError
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

ნახ.2.15

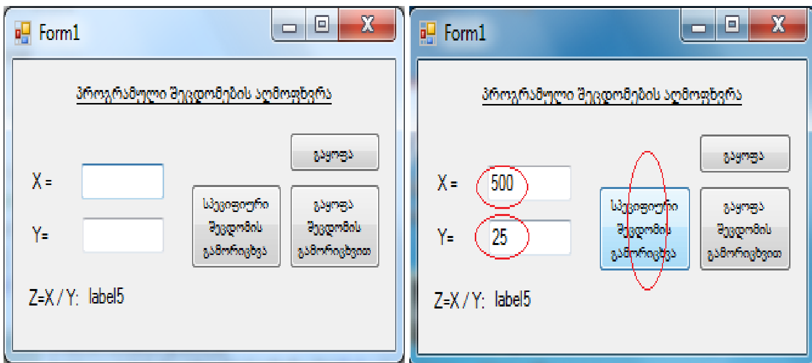
Application.Run (new Form1()); სტრიქონის ამუშავებით ისარი გადადის (იხ. Solution Explorer) Form1.Designer.cs პროგრამაში (ნახ.2.16). შემდეგ InitializeComponent()-ში გაივლის ფორმაზე დალაგებულ ყველა ელემენტს.

```
Form1.Designer.cs X   Program.cs   Form1.cs   Form1.cs [Design]
WinProgError.Form1                                components
namespace WinProgError
{
    partial class Form1
    {
        /// <summary> ...
        private System.ComponentModel.IContainer components = null;

        /// <summary> ...
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary> ...
        private void InitializeComponent()
        {
            this.label1 = new System.Windows.Forms.Label();
            this.textBox1 = new System.Windows.Forms.TextBox();
            this.textBox2 = new System.Windows.Forms.TextBox();
            this.label2 = new System.Windows.Forms.Label();
            this.label3 = new System.Windows.Forms.Label();
            this.label4 = new System.Windows.Forms.Label();
```

ნახ.9.16

Form1.Designer.cs პროგრამის ბიჯურად გავლის შემდეგ Main()-იდან ამუშავდება Run და ეკრანზე გამოვა Form1 (ნახ.2.17), სადაც უნდა შევიტანოთ X და Y მნიშვნელობები და ავამოქმედოთ ღილაკი „სპეციფიური შეცდომის გამორიცხვა“ (ნახ.2.18).

ბიჯის ისარი გადადის Form1.cs პროგრამის ტექსტზე (ნახ.2.18), სადაც button3_Click მოვლენის შესაბამის try {…} ბლოკში შედის.

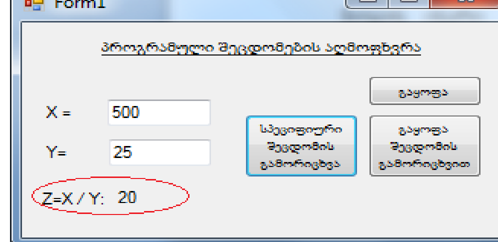



ნახ.2.17

```
Form1.Designer.cs    Program.cs    Form1.cs ×    Form1.cs [Design]
WinProgError.Form1

        private void button3_Click(object sender, EventArgs e)
        {
            int x, y, z;
            try
            {
                x = Convert.ToInt32(textBox1.Text);
                y = Convert.ToInt32(textBox2.Text);
                z = x / y;
                label5.Text = z.ToString();
            }
```

ნახ.2.18

ვინაიდან X და Y-ის შეტანილი მნიშვნელო-ბები დასაშვებია, შედეგში აისახება label5.Text=z.ToString() მნიშვნელობა, ანუ 20 (ნახ.9.19).




ნახ.2.19

თუ ახლა განვიხილავთ Y=0 შემთხვევას, და ავამოქმედებთ იგივე ბუტონს, მაშინ try ბლოკში მომზადდება განსაკუთრებული შემთხვევა, როცა გამყოფი 0-ია.

```
private void button3_Click(object sender, EventArgs e)
{
    int x, y, z;
    try
    {
        x = Convert.ToInt32(textBox1.Text);
        y = Convert.ToInt32(textBox2.Text);
        z = x / y;
        label5.Text = z.ToString();
    }
```

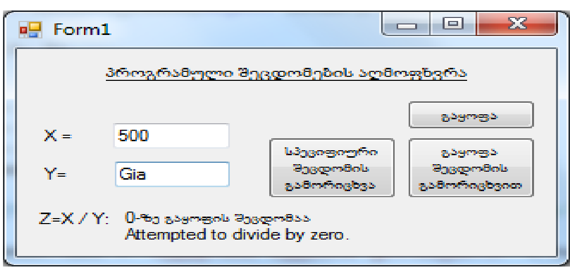
ნახ.2.20

მოქმედება გადაეცემა catch ბლოკს:

```
catch (DivideByZeroException nul_Div)   // 0-ზე გაყოფის შეცდომა
{
    label5.Text = "0-ზე გაყოფის შეცდომაა\n" + nul_Div.Message;
}
```

ნახ.2.21

დავუშვათ, რომ X ან Y არარიცხვითი სიმბოლო ან სტრიქონია, მაგალითად, “G, Gia,…” (ნახ.2.22).



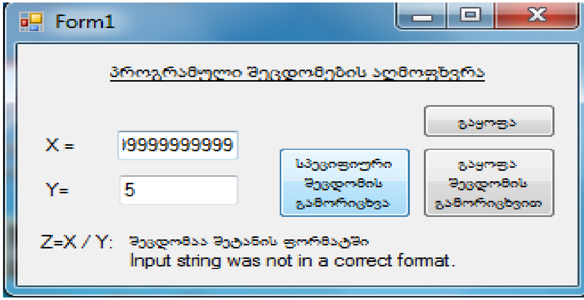

ნახ.2.22

F11-ით გადაადგილების შემდეგ პროგრამის ტექსტის აქტიური ფრაგმენტი იქნება შემდეგი (ნახ.2.23).

```
private void button3_Click(object sender, EventArgs e)
{
    int x, y, z;
    try
    {
        x = Convert.ToInt32(textBox1.Text);
        y = Convert.ToInt32(textBox2.Text);
        z = x / y;
        label5.Text = z.ToString();
    }
    catch (FormatException nul_Div)  // ტიპის გარდაქმნის შეცდომა
    {
        label5.Text = "შეცდომაა შეტანის ფორმატში\n" + nul_Div.Message;
    }
```

ნახ.2.23

მესამე, ზოგადი ანუ არასპეციფიური შემთხვევაა, ამ დროს სისიტემა თვითონ აღმოაჩენს, თუ რა სახის შეცდომასთან გვაქვს საქმე. მაგალითად, თუ X ან Y - ში შევიტანთ „დიდ რიცხვს“ (ნახ.2.24), მაშინ კოდის ფრაგმენტი ასე გამოიყურება (ნახ.2.25).




ნახ.2.24

```
catch (Exception nul_Div)  // ზოგადი შეცდომა
{
    label5.Text = "ზოგადი, არასპეციფიური შეცდომაა\n" + nul_Div.Message;
}
```

ნახ.9.25

პროგრამასთან მუშაობის დასამთავრებლად დავხუროთ Form1. ამ დროს მართვა (ისარი) გადაეცემა Form1.Designers.cs (ნახ.2.26) და ბოლოს პროგრამას Form1.cs, რომელშიც არის Main(), და რომლითაც დაიწყო თავიდან ამ პროგრამის მუშაობა (ნახ.2.27).

```
Form1.Designer.cs  Program.cs  Form1.cs  Form1.cs [Design]
WinProgError.Form1
namespace WinProgError
{
    partial class Form1
    {
        /// <summary> ...
        private System.ComponentModel.IContainer components = null;

        /// <summary> ...
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }
```

ნახ.2.26

```
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
```

ნახ.2.27

ამით დასრულდება დებაგერის მუშაობის პროცესი.

პროგრამის ანალიზის პროცესში შესაძლებელია ცვლადების მნიშვნელობათა ვიზუალური შემოწმება. ამისათვის მაუსის კურსორი უნდა მივიტანოთ ცვლადთან. 2.28 ნახაზზე ნაჩვენებია პროგრამაში X, Y და Z -ის მნიშვნელობები.

```
try
{
    x = Convert.ToInt32(textBox1.Text);
     x | 46     .ToInt32(textBox2.Text);
    y = Convert.ToInt32(textBox2.Text);
     y | 2
    z = x / y;
     z | 23     = z.ToString();
}
```

ნახ.2.28

დიდი პროგრამების ანალიზის დროს ბიჯურ რეჟიმში მუშაობა არაეფექტურია, სჭირდება ხანგრძლივი დრო. უფრო მოსახერხებელია ვიზუალური კონტროლის ორგანიზების მეორე ხერხი, რომელიც წყვეტის წერტილების კონცეფციითაა ცნობილი.

წყვეტის წერტილი არის პროგრამის კოდის ის ადგილი (სტრიქონი), სადაც წყდება პროგრამის შესრულება. ამ სტრიქონში მოთავსებული გამოსახულება (ოპერატორი ან მეთოდი) არ შესრულდება და მართვა მომხმარებელს გადაეცემა.

წყვეტის წერტილის შესაქმნელად კოდის საჭირო სტრიქონის გასწვრივ მარცხენა ველში მოვათავსოთ კურსორი და დავაჭიროთ თაგვის მარცხენა კლავიშს (ან F9 კლავიშს). გამოჩნდება შინდისფერი წრე. პროგრამის კოდში შეგვიძლია შევქმნათ წყვეტის რამდენიმე წერტილი (ნახ.2.29). მენიუდან Debug→Windows→ Autos არჩევით რედაქტორის ქვედა ნაწილში გამოიტანება ცვლადების მონიტორინგის ფანჯარა, რომელშიც ერთდროულად ჩანს რამდენიმე ცვლადი მათი აქტუალური მნიშვნელობებით (ნახ.2.30). მენიუდან Debug→Windows→Locals -ის არჩევით მონიტორინგის ფანჯარაში გამოიტანება მხოლოდ ლოკალური ცვლადები და მათი მნიშვნელობები.

```
try
{
    x = Convert.ToInt32(textBox1.Text);
    y = Convert.ToInt32(textBox2.Text);
    z = x / y;
    label5.Text = z.ToString();
}
catch (FormatException nul_Div)  // ტიპის გარდაქმნის შეცდომა
{
    label5.Text = "შეცდომაა შეტანის ფორმატში\n" + nul_Div.Message;
}
catch (DivideByZeroException nul_Div)  // 0-ზე გაყოფის შეცდომა
{
    label5.Text = "0-ზე გაყოფის შეცდომაა\n" + nul_Div.Message;
}
catch (Exception nul_Div)  // ზოგადი შეცდომა
{
    label5.Text = "ზოგადი, არასპეციფიური შეცდომაა\n" + nul_Div.Message;
}
```

ნახ.2.29

```
private void button3_Click(object sender, EventArgs e)
{
    int x, y, z;
    try
    {
        x = Convert.ToInt32(textBox1.Text);
        y = Convert.ToInt32(textBox2.Text);
        z = x / y;
        label5.Text = z.ToString();
    }
    catch (FormatException nul_Div)  // ტიპის გარდაქმნის შეცდომა
    {
        label5.Text = "შეცდომაა შეტანის ფორმატში\n" + nul_Div.Message;
    }
    catch (DivideByZeroException nul_Div)  // 0-ზე გაყოფის შეცდომა
    {
        label5.Text = "0-ზე გაყოფის შეცდომაა\n" + nul_Div.Message;
    }
    catch (Exception nul_Div)  // ზოგადი შეცდომა
    {
        label5.Text = "ზოგადი, არასპეციფიური შეცდომაა\n" + nul_Div.Message;
    }
}
```

utos

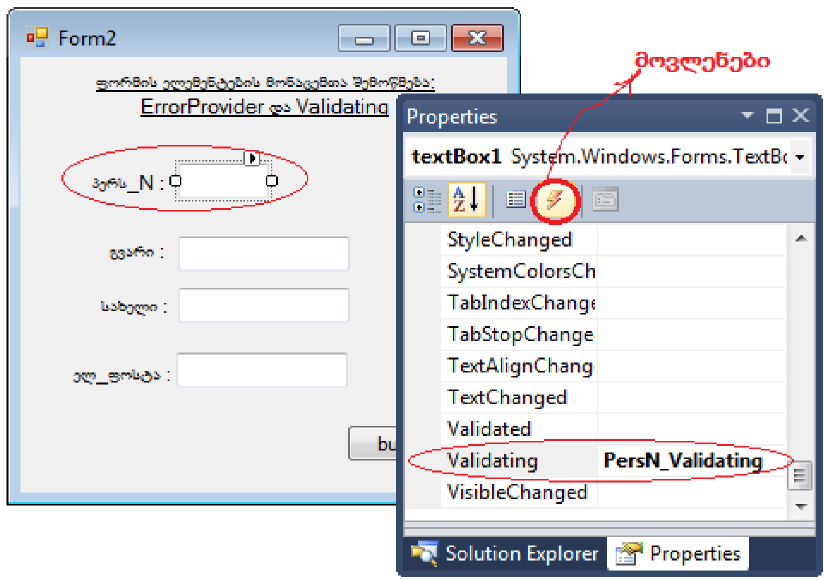
| Name | Value | Type |
|---|---|---|
| label5 | {System.Windows.Forms.Label, Text: 0-ზე გაყოფის შეცდო | System.Windows.Forms.Label |
| label5.Text | "0-ზე გაყოფის შეცდომაა\nAttempted to divide by zer | string |
| this | {WinProgError.Form1, Text: Form1} | WinProgError.Form1 |
| x | 400 | int |
| y | 40 | int |
| z | 10 | int |

ნახ.2.30

## 2.4) შესატან მონაცემთა კონტროლი

ვინდოუს-ფორმის ელემენტების შევსების პროცესში, როცა მომხმარებელს უხდება მათი ხელით შეტანა, შესაძლებელია შეცდომების არსებობა. მაგალითად, ამას ხშირად აქვს ადგილი ტქსტბოქსების შევსებისას.

იმისათვის, რომ შესაძლებელი იყოს შესატან მონაცემთა კონტროლი, საჭიროა ErrorProvider ვიზუალური კომპონენტის გადმოტანა ინსტრუმენტების პანელიდან და საკონტროლო ელემენტების Properties-ში CausesValidation თვისებაში "True" მნიშვნელობის არსებობა (ნახ.2.31).



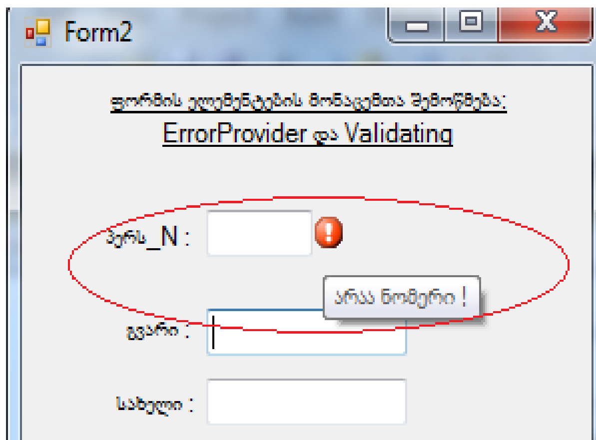

ნახ.2.31

ტექსტბოქსის შევსების შემდეგ, როცა მომხმარებელი გადადის სხვა ელემენტზე, ხდება ამ ტექსტბოქსში შეტანილი მნიშვნელობის შემოწმება. თუ რა ლოგიკით შემოწმდება ტექსტბოქსში შეტანილი მონაცემი, დამოკიდებულია ჩვენ მიერ განსაზღვრულ მეთოდზე, რომელიც მიებმება მოვლენის დამმუშავებელს (Event Handling).

ამგვარად ახლა საჭიროა მოვლენის დამმუშავებელის შექმნა. მაგალითად, ვდგებით „პერს_N" ტექსტბოქსზე და Properties-ში Validating თვისებას ვაძლევთ სახელს: PersN_Validating. დამმუშავებლის მეთოდის კოდი მოცემულია 2_3 ლისტინგში.

```
// ლისტინგი_9.3 ---  მოვლენების დამმუშავებელი ------
private void PersN_Validating(object sender, CancelEventArgs e)
{
  if (textBox1.Text.Length == 0)
   {
    errorProvider1.SetError(textBox1, "არაა ნომერი !");
   }
  else
     errorProvider1.SetError(textBox1, "");
}
```

შედეგი მოცემულია 2.32 ნახაზზე. ველში „პერს_N" არ ჩაწერეს მონაცემი, ისე გადავიდნენ სხვა ტექსტბოქსზე. ამ დროს მოხდა მოვლენის შემოწმება და შეცდომის აღმოჩენა, რომ ველში არაა შეტანილი მონაცემი (`textBox1.Text.Length` არის `0`). ჩაირთვება `errorProvider1.SetError` და „პერს_N"-ის გვერდით გამოიტანს წითელი ფერის ძახილის ნიშანს (გაფრთხილება). მაუსის კურსორის მიტანისას ამ ველზე ჩნდება ქართული წარწერა „არაა ნომერი" (`SetError`-ის მეორე პარამეტრი).

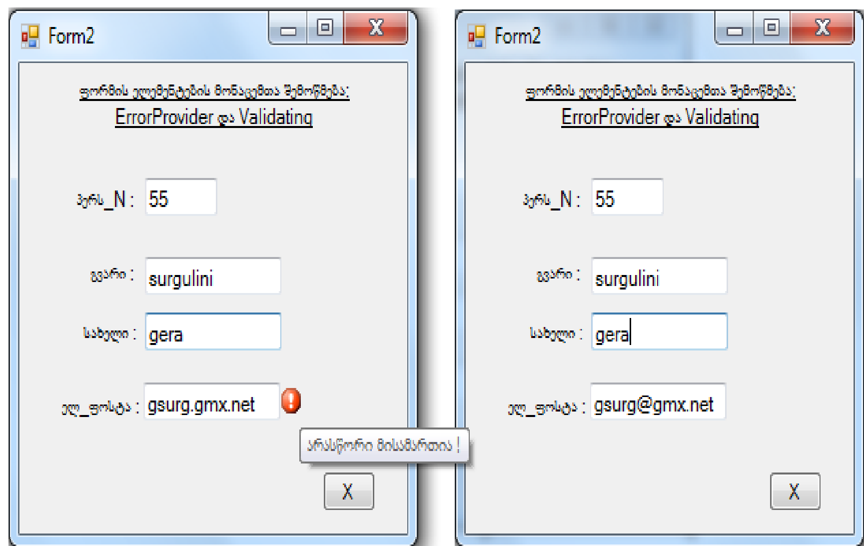



ნახ.2.32

მეოთხე ტქსტბოქსში უნდა ჩაიწეროს ელ_ფოსტის მისამართი. იგი სტრიქონული მონაცემია, რომელიც აუცილებლად უნდა შეიცავდეს '@' და ' . ' სიმბოლოებს. თუ რომელიმე აკლია, მაშინ შეცდომაა და უნდა ამუშავდეს მოვლენის დამმუშავებელი ამ ველისთვის (2.4_ლისტინგი).

```
// ლისტინგი_2.4 --- eMail_Validating მეთოდი --------
 private void eMail_Validating(object sender, CancelEventArgs e)
{
  string email = textBox4.Text;
     // კონტროლის ლოგიკა
   if(email.IndexOf('@')==-1 || email.IndexOf('.')==-1)
   {
    errorProvider1.SetError(textBox4,"არასწორი  მისამართია !");
   }
  else
    errorProvider1.SetError(textBox4, "");
}
```

შედეგი ნაჩვენებია 2.33 ნახაზზე.




ნახ.2.33

## 2.3. კლასებისა და ობიექტების დაპროგრამება მართვის ამოცანებში

### ლაბორატორიული სამუშაო N3

**მიზანი:** ობიექტორიენტირებული დაპროგრამების ინკაფსულაციის თვისების დეტალიზებული შესწავლა. კლასის განსაზღვრა როგორც მონაცემებისა და მეთოდების ერთობლიობა და მათი პროგრამული რეალიზაციის განხორციელება. როგორ შევქმნათ კლასები, მათი ობიექტები, მეთოდები და კლასთაშორის კავშირები კონსოლისა და ვინდოუს_ფორმების პროექტებში ?

განვიხილოთ აღნიშნული საკითხები პრაქტიკული მართვის ამოცანის საფუძველზე, მათი შემდგომი განზოგადების მიზნით.

**ამოცანა_3.1:** ავაგოთ ობიექტორიენტირებული პროგრამის კოდი (კლასებისა და მეთოდების გამოყენებით) მაღლივ შენობაში ლიფტის გადაადგილების მოდელირებისათვის.

მაგალითად, შენობა N სართულიანია. მას აქვს ლიფტი. პერსონა, რომელიც შედის ლიფტში (ხდება მგზავრი), ირჩევს მისთვის საჭირო სართულსა და მიემგზავრება (ზევით ან ქვევით). საჭიროა ამ პროცესის მოდელირება და დაპროგრამება ისე, რომ დაფიქსირდეს ლიფტის საწყისი მდგომარეობა, ამუშავების შემდეგ მისი საბოლოო მდგომარეობა, გავლილი სართულების რაოდენობა (ერთი ამუშავებისას). საბოლოოდ გამოიცეს რეპორტი, თუ სულ რამდენი სართული გაიარა ლიფტმა ერთი სეანსის (სათანადო პერიოდის) განმავლობაში.

### 3.1) კლასთა მოდელი და მათი აგების პირობები:

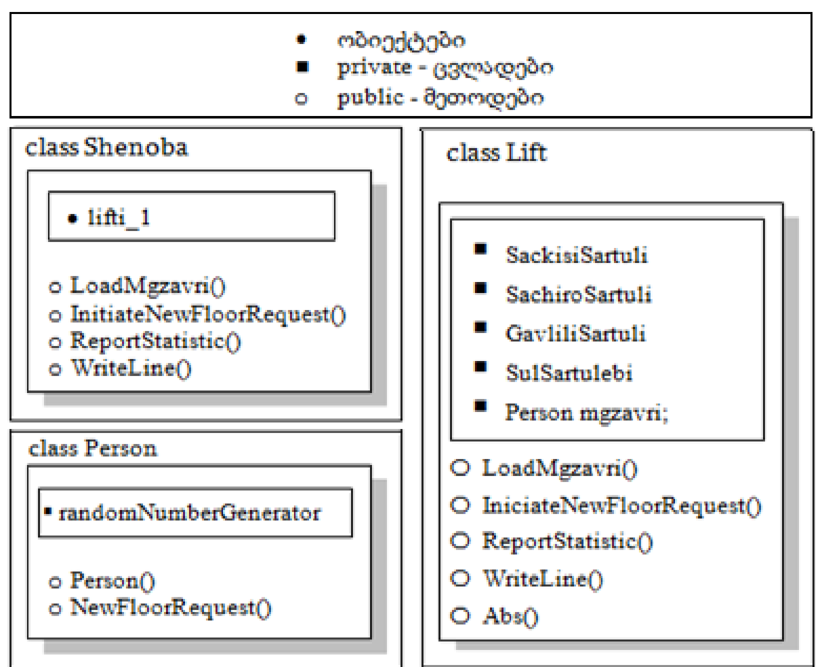
ინკაფსულაციის სქემა მოცემულია 3.1 ნახაზზე.

- პროგრამაში სამი კლასია: Shenoba, Lift და Person;

- Shenoba კლასს აქვს Lift კლასის ერთი ობიექტი, სახელით lifti_1 ;

- Person კლასის ერთი ობიექტი იმყოფება mgzavri - ცვლადის შიგნით, რომელიც იყენებს lifti_1;

- Lift კლასის ობიექტს შეუძლია გადაადგილება ნებისმიერ სართულზე, ინტერვალში [1,N=60].




ნახ.3.1

- პროგრამაში სართულის არჩევა ხდება მგზავრის მიერ („გამოიყენება „შემთხვევით რიცხვთა გენერატორი" Random-ფუნქციით);

- lifti_1 ლიფტი მოძრაობს საჭირო სართულისაკენ, რაც აისახება კონსოლზე;

- მგზავრ(ებ)ის ლიფტით მოძრაობის სეანსი შედგება რამდენიმე ეტაპისაგან, რომლბზეც შეირჩევა ახალი მიზნობრივი სართულები;

- სეანსის ბოლოს გამოიცემა ანგარიშის ტექსტი (რეპორტი), თუ ჯამში რამდენი სართული გაიარა ლიფტმა;

- შუალედური და საბოლოო შედეგები გამოიტანება ეკრანზე.

## 3.2) კლასთა კავშირების აღწერა UML ტექნოლოგიით:

მომხმარებლის სამი კლასი: Shenoba, Lift, Person და ერთი სისტემური კლასი System.Random ამოდელირებს ლიფტის მუშაობის პროცესს:

- Shenoba-კლასი შეიცავს Lift-ობიექტსა და იძახებს მის მეთოდებს;

- Lift-ობიექტი იყენებს Person-ობიექტს, რათა მართოს ლიფტის მოძრაობა;

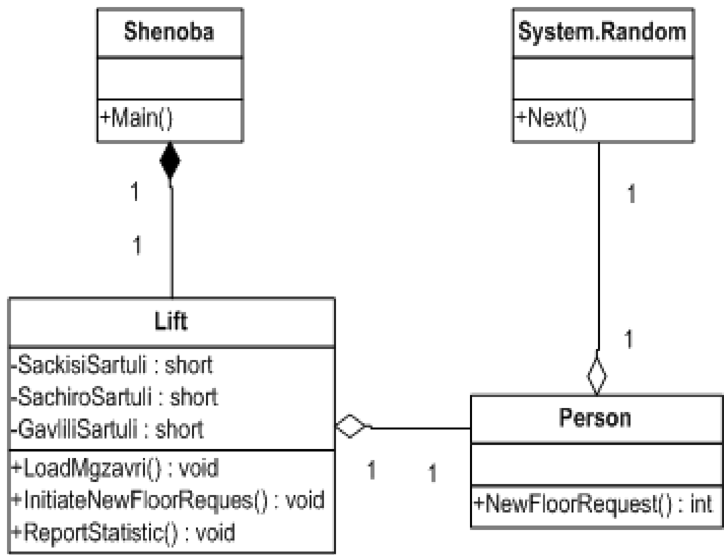
- Person-ობიექტი კი იყენებს System.Random-ობიექტს, რათა აირჩიოს საჭირო სართული შემდგომი გადაადგილებისათვის.

ობიექტ - ორიენტირებული პროგრამული სისტემების დაპროექტებისას კლასებმა მსგავსი ფუნქციები უნდა შეასრულოს. თითოეულს უნდა ჰქონდეს თავისი უნიკალური ფუნქციონალობა, და ყველა ერთად უნდა უზრუნველყოფდეს მთლიანი პროგრამის მუშაობას.

ნებისმიერი მაღლივი შენობა, როგორც „მთელი“ შედგება სხვადასხვა ნაწილისაგან: იატაკი, კედლები, ფანჯრები, ლიფტები და ა.შ. ამგვარად, შენობასა და ლიფტს შორის არსებობს დამოკიდებულება „მთელი-ნაწილი“ (აგრეგატული კავშირი UML ენაზე). ამიტომაცაა, რომ Lift-კლასის ობიექტის ცვლადი გამოცხადებულია Shenoba-კლასის შიგნით (ნახ.3.1, lifti_1). პროგრამულად იგი რეალიზებულია სტატიკური ცვლადის სახით (47-ე სტრიქონი, ლისტინგი_3.1): private static Lift lifti_1;

იგი ინახავს Lift კლასის ობიექტსა და შეუძლია მისი მეთოდების გამოძახება. გარდა ამისა, Shenoba-კლასს შეიძლება ჰქონდეს აგრეთვე სხვა ცვლადებიც, რომლებიც აგრეგატულად დაქვემდებარებული კლასების ობიექტების ცვლადები იქნება.

3.2 ნახაზზე მოცემულია ჩვენი მაგალითის კლასებს შორის კავშირების UML დიაგრამა, აგებული Ms Visio პაკეტის გარემოში.




**ნახ.3.2**

Shenoba და Lift კლასებს შორის აგრეგაციას უწოდებენ კომპოზიციასა და იგი შავი რომბიკით გამოისახება. „1“-ები ნიშნავს, რომ 1 შენობაში არის 1 ლიფტი (ჩვენს შემთხვევაში). Lift და Person, აგრეთვე Person და System.Random კლასებს შორის აგრეგატული დამოკიდებულებაა, მაგრამ არა-კომპოზიციური. ის პატარა რომბითაა ნაჩვენები. განსხვავება ისაა, რომ შენობას ლიფტი ყოველთვის აქვს. ლიფტში კი პერსონა შეიძლება იყოს ან არ იყოს, ლიფტი ისეც მუშაობს.

## 3.3) პროგრამული რეალიზაცია:

ცხრილში მოცემულია განხილული კლასების პროგრამული რეალიზაციის ლისტინგი.

■ private - ცვლადები
○ public - მეთოდები

class Lift

■ SackisiSartuli
■ SachiroSartuli
■ GavliliSartuli
■ SulSartulebi
■ Person mgzavri;

○ LoadMgzavri()
○ IniciateNewFloor Request()
○ ReportStatistic()
○ WriteLine()
○ Abs()

```
1   using System;
2   namespace ConsoleApp_Lift
3   {
4    class Lift
5    {
6      private int SackisiSartuli = 1;
7      private int SachiroSartuli = 0;
8      private int GavliliSartuli = 0;
9      private int SulSartulebi = 0;
10     public  int i = 1;
11     private Person mgzavri;
12   public void LoadMgzavri()
13     {
14        mgzavri = new Person();
15     }
16   public void InitiateNewFloorRequest()
17   {
18     SachiroSartuli = mgzavri.NewFloorRequest();
19     GavliliSartuli = Math.Abs(SackisiSartuli -
                                      SachiroSartuli);
20     Console.WriteLine("{0,2}.| Sackisi: {1,2}|
          Sachiro: {2,2}| Gavlili: {3,2} ",  i.ToString(),
          SackisiSartuli.ToString(),
          SachiroSartuli.ToString(),
          GavliliSartuli.ToString());
21   // Math.Abs(SackisiSartuli - SachiroSartuli);
22     SulSartulebi += GavliliSartuli;
23      SackisiSartuli = SachiroSartuli;
24      i++;
25    }
26   public void ReportStatistic()
27   {   Console.WriteLine("\n===>>> Sul gavlili
28                       sartulebi: " + SulSartulebi);
29     }
```

| | | |
|---|---|---|
| class Person<br>▪ randomNumber Generator;<br>o Person()<br>o NewFloorRequest() | 30<br>31<br>32<br>33<br><br>34<br>35<br>36<br><br>37<br>38<br>39<br>40<br>41<br><br>42<br>43<br>44 | }<br><br>class Person<br>{<br>private System.Random<br>randomNumberGenerator;<br>public Person() // კონსტრუქტორი<br>{<br>randomNumberGenerator = new<br>System.Random();<br>}<br><br>public int NewFloorRequest()<br>{<br>// აბრუნებს არჩეულ შემთხვევით<br>// რიცხვს 1-60 დიაპაზონში<br>return randomNumberGenerator.Next(1,60);<br>}<br>} |
| • ობიექტი<br>class Shenoba<br>• lifti_1<br>o LoadMgzavri()<br>o InitiateNewFloor Request()<br>o ReportStatistic()<br>o WriteLine() | 45<br>46<br>47<br>48<br><br>49<br>50<br>51<br>52<br>53<br>54<br>55<br>56<br>57<br>58<br>59<br>60 | class Shenoba // კლასი შენობა<br>{<br>private static Lift lifti_1;<br>private static void Main()<br>{<br>lifti_1 = new Lift();<br>lifti_1.LoadMgzavri();<br>Console.WriteLine(" samgzavro sartulebi \n<br>============================\n");<br>lifti_1.InitiateNewFloorRequest();<br>lifti_1.InitiateNewFloorRequest();<br>lifti_1.InitiateNewFloorRequest();<br>lifti_1.InitiateNewFloorRequest();<br>lifti_1.InitiateNewFloorRequest();<br>lifti_1.ReportStatistic();<br>}<br>}} |

ლისტინგი_3.1: კონსოლის რეჟიმი

## 3.4) პროგრამის ლისტინგის ანალიზი

| N | დანიშნულება |
|---|---|
| 4 | Lift - კლასის განსაზღვრების დასაწყისი |
| 6 | SackisiSartuli - ცვლადის გამოცხადება **int**-ტიპით, private-წვდომის სპეციფიკატორით და საწყისი მნიშვნელობით=1 |
| 11 | Mgzavri - ცვლადის გამოცხადება, რომელიც შეიცავს Person- კლასის ობიექტს. Person- კლასი ასრულებს მგზავრის როლს Lift-კლასთან მიმართებით |
| 12 | LoadMgzavri() - მეთოდის განსაზღვრების დასაწყისი. ის არის Lift-კლასის ინტერფეისის ნაწილი და გამოცხადებულია როგორც public |
| 14 | Person-კლასის ახალი (**new**) ობიექტის შექმნა. ეს ობიექტი მიენიჭება ცვლადს - mgzavri |
| 16 | InitiateNewFloorRequest() - მეთოდის განსაზღვრების დასაწყისი. ის არის Lift-კლასის ინტერფეისის ნაწილი და გამოცხადებულია როგორც public |
| 18 | mgzavri-ობიექტისთვის NewFloorRequest() მეთოდის გამოძახება. ამ მეთოდით დაბრუნებული მნიშვნელობა მიენიჭება ცვლადს SachiroSartuli |
| 19 | ერთი მგზავრობის შემდეგ იანგარიშება გავლილი სართულების რაოდენობა |
| 20 | ეკრანზე გამოიტანება: საწყისი_სართული, საჭირო_სართული, გავლილი_სართულების_რაოდენობა |
| 21 | იანგარიშება სულ გავლილი სართულების რაოდენობა ლიფტის მუშაობის მთელი სეანსის დროს |
| 22 | ლიფტის საწყისი სართულის მნიშვნელობას ენიჭება მისი ბოლო გაჩერების სართულის მნიშვნელობა |
| 23 | ლიფტის ამუშავების მომდევნო ბიჯის ინკრემენტი |
| 26 | ReportStatistic() მეთოდის გამოძახებით ეკრანზე |

| | |
|---|---|
| | გამოიტანება სტატისტიკა SulSartulebi ცვლადით |
| 31 | Person კლასის განსაზღვრების დასაწყისი |
| 33 | randomNumberGenerator ცვლადის გამოცხადება System.Random კლასის ობიექტის შესანახად |
| 34 | სპეციალური მეთოდის (კონსტრუქტორის !) განსაზღვრების დასაწყისი, რომელიც გამოიძახება ავტომატურად Person კლასის ობიექტის შექმნის დროს |
| 36 | System.Random-კლასის ახალი ობიექტის შექმნა და მისი მინიჭება randomNumberGenerator - ცვლადზე |
| 39 | int ტიპის NewFloorRequest() მეთოდის განსაზღვრა. ის Person-კლასის ინტერფეისის ნაწილია |
| 42 | პერსონა (ვირტუალური მგზავრი) ლიფტში ირჩევს საჭირო სართულს (შემთხვევით რიცხვთა გენერატორი ასრულებს ამ ფუნქციას) დიაპაზონში [1-60] |
| 45 | Shenoba კლასის აღწერა |
| 47 | Shenoba კლასში გამოცხადებულია Lifti ტიპის ცვლადი Lifti_1. Shenoba კლასი კომპოზიციურ კავშირშია Lift კლასთან (ნახ.3.2) |
| 48 | Main( ) მეთოდის აღწერის დასაწყისი |
| 49 | Lifti კლასის ახალი ობიექტის შექმნა და მისი მინიჭება lifti_1 ცვლადზე |
| 50 | lifti_1 ობიექტისთვის LoadMgzavri( ) მეთოდის გამოძახება |
| 51-56 | lifti_1 ობიექტისთვის .IniciateNewFloorRequest( ) მეთოდის გამოძახება 5-ჯერ |
| 57 | lifti_1 ობიექტისთვის . ReportStatistic() მეთოდის გამოძახება (შედეგების გამოსაცემად ეკრანზე) |

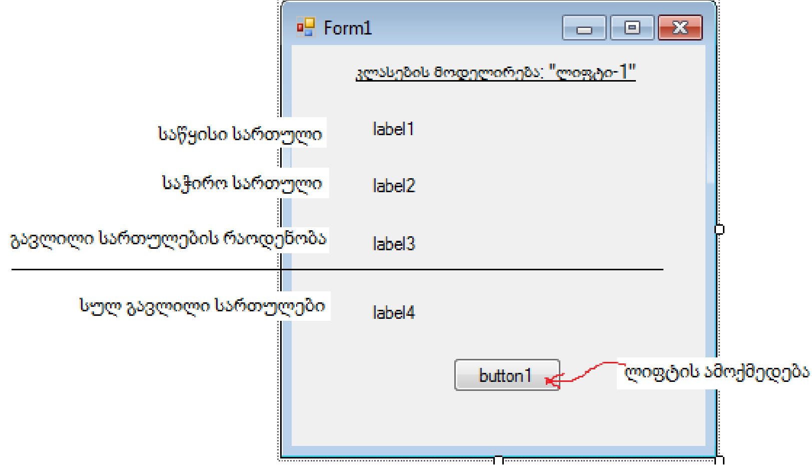
## 3.5) პროგრამის მუშაობის შედეგები:

3.3 ნახაზზე ნაჩვენებია განხილული პროგრამის შესრულების შედეგები კონსოლის რეჟიმში.

```
file:///C:/C#2010/ConsoleLift/ConsLift1/Con...
      samgzavro sartulebi
========================================

 1.| Sackisi:  1| Sachiro: 32| Gavlili: 31
 2.| Sackisi: 32| Sachiro: 58| Gavlili: 26
 3.| Sackisi: 58| Sachiro: 11| Gavlili: 47
 4.| Sackisi: 11| Sachiro: 37| Gavlili: 26
 5.| Sackisi: 37| Sachiro: 23| Gavlili: 14

===>>>         Sul gavlili sartulebi: 144
```

ნახ.3.3

**ამოცანა_3.2:** განხილული ამოცანისათვის ავაგოთ პროგრამული კოდი კლასების საფუძველზე ვინდოუსის ფორმის რეჟიმში. 3.4 ნახაზზე ნაჩვენებია სამუშაო ფანჯარა, რომელიც Form კლასის Form1 ობიექტია. მასზე განთავსებულია ოთხი label (1,2,3,4) და ერთი button1, რომლითაც მოდელირდება ლიფტის ამოქმედება (ლიფტის გამოძახება, როცა პერსონა გარეთაა ან სართულის არჩევა ლიფტის ღილაკით, როცა პერსონა ლიფტშია, ანუ მგზავრია).

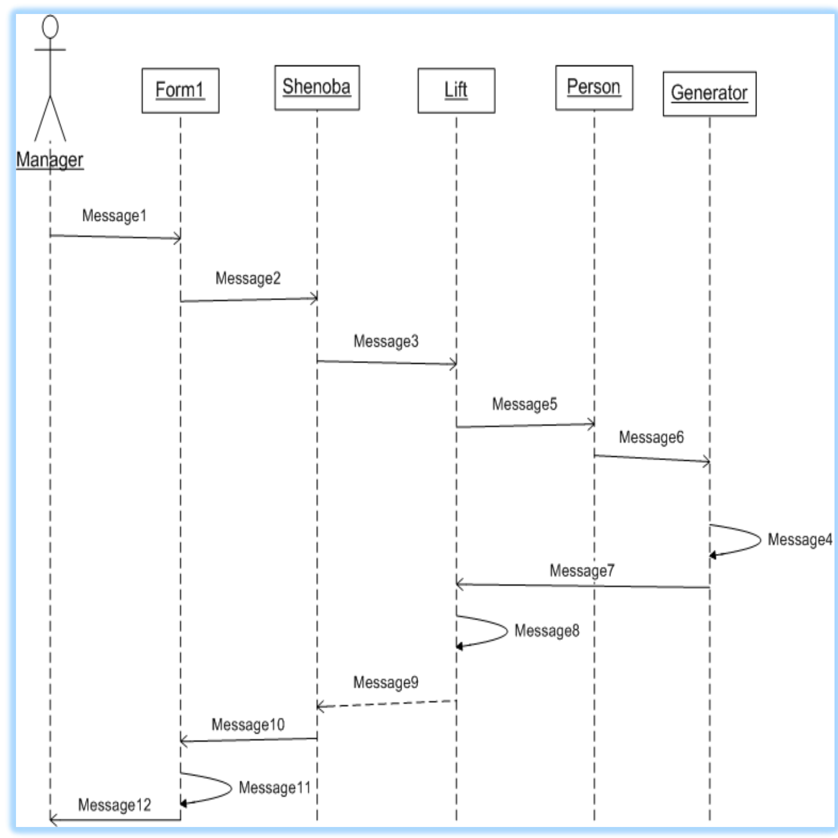



ნახ.3.4

```
// ლისტინგი_3.2--Form1  ელემენტებით და  button1-ის კოდი ---
using System;
using System.Drawing;
using System.Windows.Forms;
namespace WinFormLift
{
   public partial class Form1 : Form
   {
      Shenoba shenoba_1;    // კლასი Shenoba  უნდა შეიქმნას
      public Form1() { InitializeComponent();   }
      private void button1_Click(object sender, EventArgs e)
      {   // shenoba_1  ობიექტის შექმნა
         shenoba_1 = new Shenoba(label1, label2, label3, label4);
      }
   }
}
```

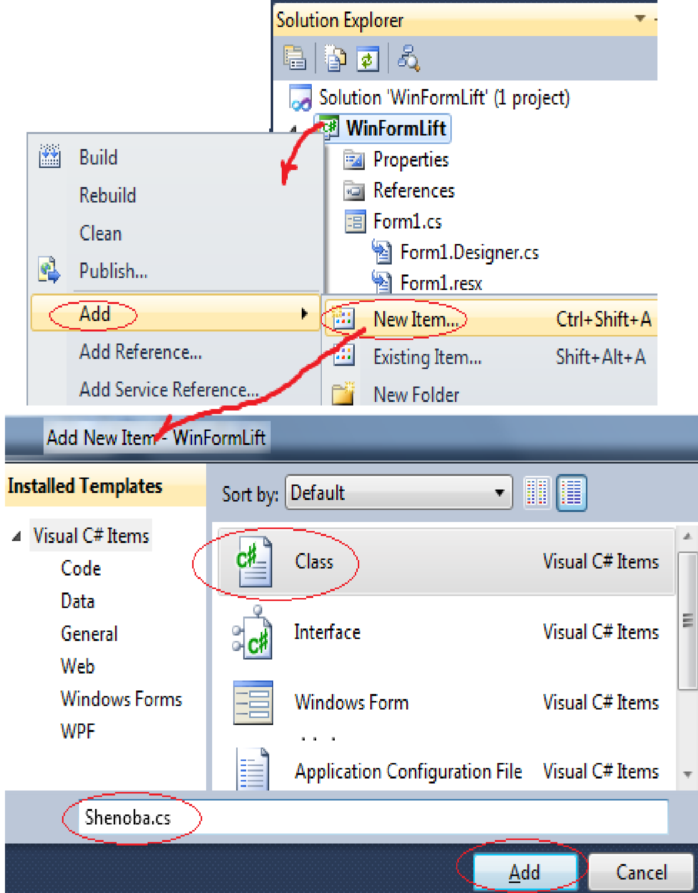
გარდა Form კლასისა (Form1 ობიექტით), რომელიც ასრულებს პროგრამის მომხმარებლის ინტერფეისის ფუნქციას, ლიფტის მუშაობის პროცესის ობიექტ - ორიენტირებული მოდელის ასაგებად საჭიროა კლასები: Shenoba, Lift და Person.

ამ კლასების თვისებები და ფუნქციობა ჩვენ ზემოთ აღვწერეთ. ახლა საჭიროა ავაგოთ სცენარი „ლიფტის მუშაობა“, რომლისთვისაც გამოვიყენებთ UML-ის მიმდევრობითობის (Sequence) დიაგრამას.




ნახ.3.5

ახლა შევქმნათ დანარჩენი კლასები და აღვწეროთ მათი ფუნქციობები. Solution Explorer-ში დავამატოთ (Add new Items) ახალი კლასები, როგორც ეს 3.6 ნახაზზეა ნაჩვენები.




ნახ.3.6

Shenoba კლასის კოდი მოცემულია 3.3_ ლისტინგში.

```
// ლისტინგი_3.3 --- კლასი Shenoba --------
using System;
using System.Windows.Forms;
```

```
namespace WinFormLift
{
   public class Shenoba
  {
    static Lift lift_1 = new Lift();

  public Shenoba(Label label1, Label label2, Label
                                                      label3, Label label4)
 {
  lift_1.LoadMgzavri();
    lift_1.InitiateNewFloorRequest(label1,label2, label3);
  lift_1.ReportStatistic(label4);
 }
 }
}
```

Lift კლასის პროგრამული კოდი მოცემულია 3.4_ლისტინგში.

```
// ლისტინგი_3.4 --- კლასი Lift  --------
using System;
using System.Windows.Forms;
namespace WinFormLift
{
  public class Lift
   {
     private int SackisiSartuli = 1;
     private int SachiroSartuli = 0;
     private int GavlilSartulebi = 0;
     private int SulSartulebi = 0;
     public int i;
     private Person mgzavri;
```

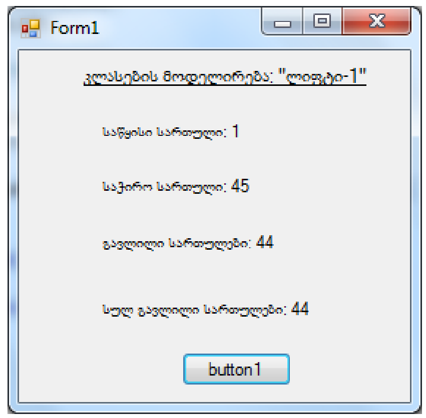
```
  public void LoadMgzavri()
   {
     mgzavri = new Person();
   }
 public void InitiateNewFloorRequest(Label label1,
                                    Label label2, Label label3)
  {
   SachiroSartuli = mgzavri.NewFloorRequest();
   GavlilSartulebi = Math.Abs(SackisiSartuli - SachiroSartuli);
   label1.Text = "საწყისი სართული: " + SackisiSartuli.ToString();
   label2.Text = "საჭირო სართული: " + SachiroSartuli.ToString();
   label3.Text = "გავლილი სართულები: " + GavlilSartulebi.ToString();
   SulSartulebi += GavlilSartulebi;
   SackisiSartuli = SachiroSartuli;
   i++;
  }
 public void ReportStatistic(Label label4)
 {
  label4.Text = "სულ გავლილი სართულები: " + SulSartulebi;
} } }
```

Person კლასის პროგრამა მოცემულია 3.5_ლისტინგში.

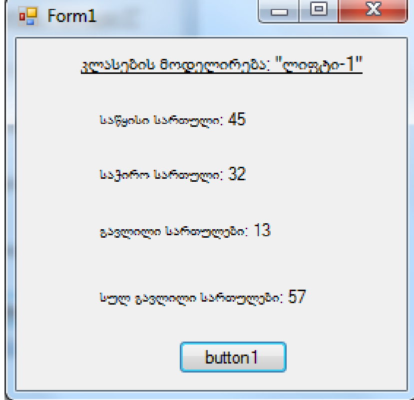
```
// ლისტინგი_3.5 --- კლასი Person  --------
using System;
using System.Windows.Forms;
namespace WinFormLift
{  public class Person
   {  private System.Random randomNumberGenerator;
      public Person()    // კონსტრუქტორი
      {
        randomNumberGenerator = new System.Random();  }
```

```
public int NewFloorRequest()
  { return randomNumberGenerator.Next(1, 60);
}}}
```

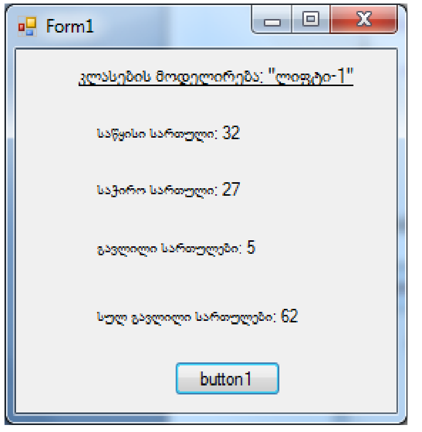
პროგრამის ამუშავების შემდეგ მიიღება 3.7 ნახაზზე ნაჩვენები შედეგები.



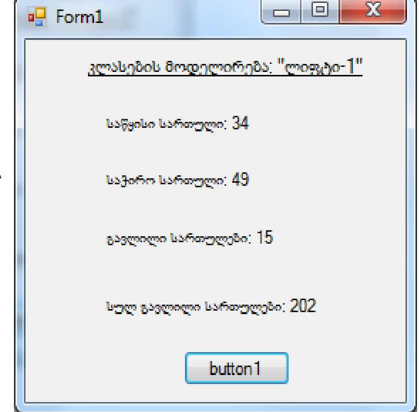

ნახ.3.7-ა. 1-ელი ბიჯი




ნახ.3.7-ბ. მე-2 ბიჯი




ნახ.3.7-გ. მე-3 ბიჯი




ნახ.3.7-დ. მე-10 ბიჯი,

და ა.შ.

## 2.4. ცხრილების წარმოდგენის მართვის ელემენტი DataGridView

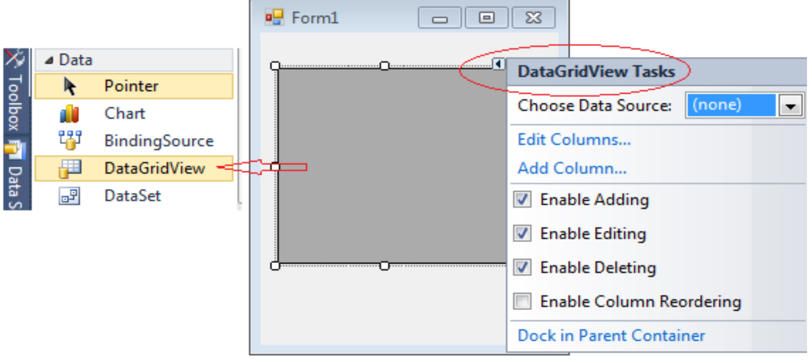
### ლაბორატორიული სამუშაო N4

მიზანი: DataGridView მართვის ელემენტის შესწავლა ცხრილებთან (ბრტყელ ფაილებთან) სამუშაოდ.

### 1. თეორიული ნაწილი

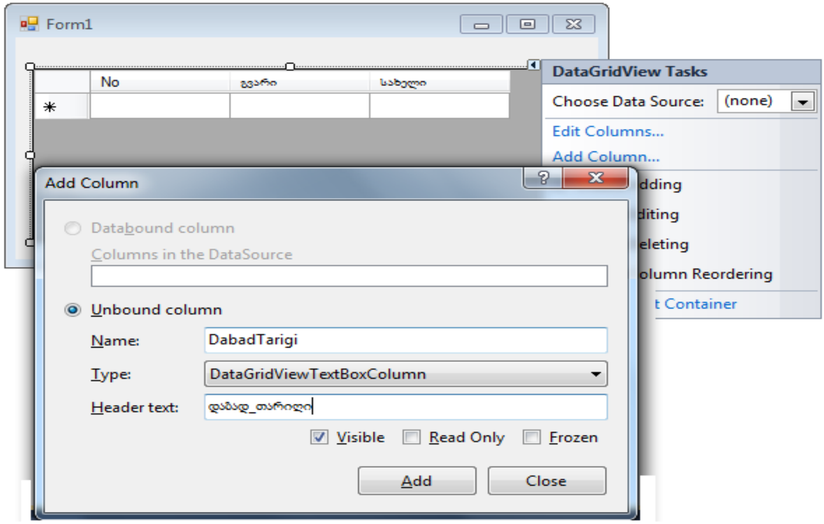
ცხრილები (Tables), რომლებიც სტრიქონებისა და სვეტებისაგან შედგება, საუკეთესო საშუალებაა მონაცემთა ორგანზომილებიანი ველების, მასივების წარმოსადგენად. C# ენაში ასეთი ობიექტების ასახვის მიზნით გამოიყენება მართვის ელემენტი, ტიპით DataGridView. ამ ელემენტს, პრაგმატული თვალსაზრისით, დიდი გამოყენება აქვს მონაცემთა ბაზების სისტემებში (ADO.NET), რასაც ჩვენ მომდევნო თავში დავუბრუნდებით.

ახლა განვიხილოთ ცხრილებთან მუშაობის ვიზუალური საშუალებანი. 4.1 ნახაზზე მოცემულია ToolBox-დან Form1-ფორმაზე გადმოტანილი DataGridView ელემენტი და საწყისი სიტუაცია.



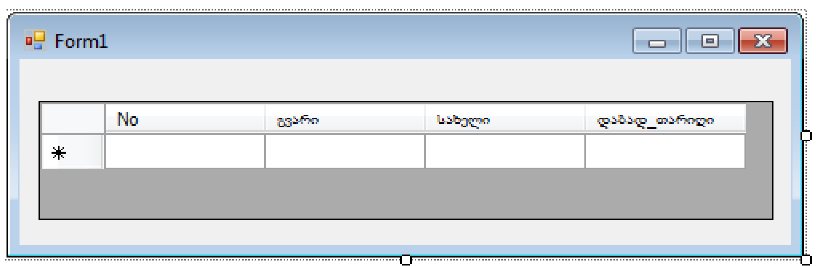

ნახ.4.1

დიალოგურ რეჟიმში ცხრილის ველების (სვეტების) შესატანად ვირჩევთ Add Columns და გადავდივართ 4.2 ნახაზზე მოცემულ ფანჯარაში. შევიტანოთ მიმდევრობით „სტუდენტები“-ს ატრიბუტები, მაგალითად, No, First_Name (გვარი), Last_Name (სახელი),Birth_data (დაბად_თარიღი) და ა.შ.



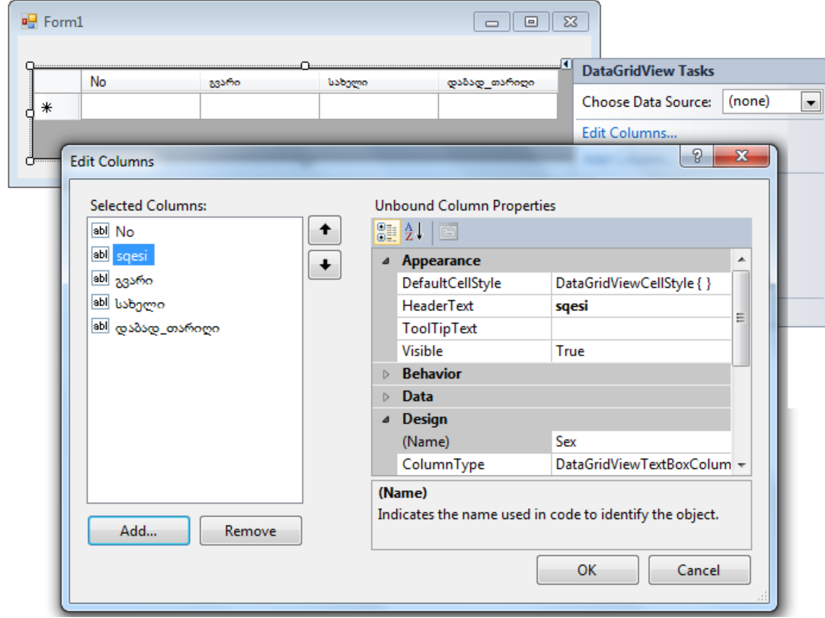

ნახ.4.2

პროგრამის ამუშავების შემდეგ მივიღებთ 4.3 ნახაზზე მოცემულ ცხრილს.

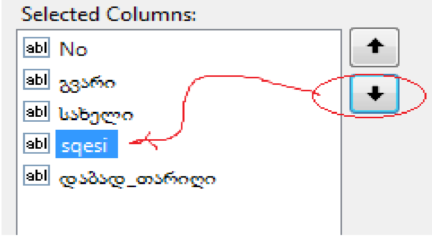



ნახ.4.3

შეტანილი ველების კორექტირებისათვის ვიყენებთ Edit Columns პუნქტს. ჩავამატოთ ველები Sex (სქესი) ან შევასწოროთ უკვე ჩაწერილი დასახელებები. მაგალითად, 4.3 ნახაზზე, ედიტორის ფანჯარაში გვინდა ველი sqesi შევცვალოთ ქართული შრიფტით და ამასთანავე იგი გადავიტანოთ ველი „სახელი“-ს შემდეგ. 4.4 და 4.5 ნახაზებზე ნაჩვენებია ეს შემთხვევები.

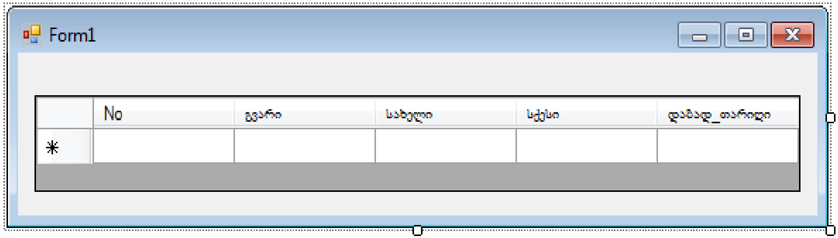



ნახ.4.4




ნახ.4.5

კოდის მუშაობის ახალი შედეგი მოცემულია 4.6 ნახაზზე.




ნახ.4.6

აქვე შეიძლება მონაცემთა სტრიქონების (Rows) შეტანა ველების ქვეშ. მაგალითი ნაჩვენებია 4.7 ნახაზზე.

Form1

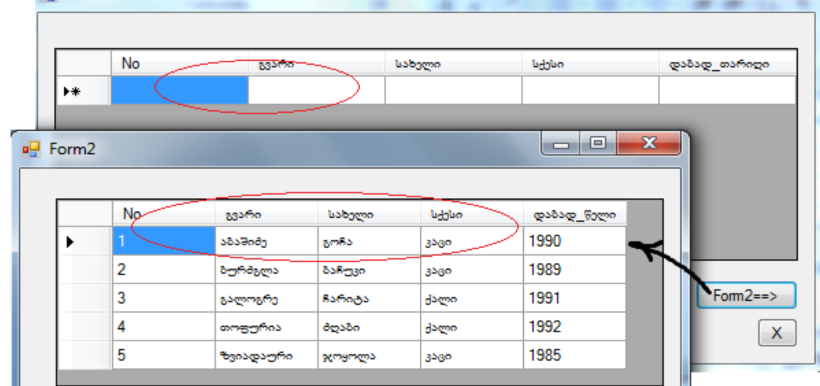
| No | გვარი | სახელი | სქესი | დაბად_თარიღი |
|---|---|---|---|---|
| 93501 | არაბული | ავთანდილი | კაცი | 1990 |
| 93502 | ბურდული | ნატალია | ქალი | 1991 |
| 93515 | დოლიძე | ნათია | ქალი | 1991 |
| 93601 | ავალიანი | უშგულა | კაცი | 1990 |
| 93521 | ჯავახიშვილი | ვანო | კაცი | 1992 |

ნახ.4.7

პროგრამის დამთავრებისა და ხელახალი ამუშავების შემდეგ შეტანილი მონაცემები იკარგება, ანუ არაა შენახული მეხსიერებაში. თუ გვინდა, რომ პროექტის გაშვებისას მონაცემები ჩაიტვირთოს პროგრამულად, მაშინ ან უნდა გამოვიყენოთ მონაცემთა ბაზასათან კავშირი (იხ. მომდევნო თავი), ან კოდის Form2_Load მეთოდში უნდა ჩავწეროთ შემდეგი სტრიქონები (ლისტინგი 11_1).

```
   // ლისტინგი_4.1 --- DataGridView -------
private void Form2_Load(object sender, EventArgs e)
{
  int i;
  // ველების (სვეტების) შევსება ------
  dataGridView1.Columns.Add("No", "No");
  dataGridView1.Columns.Add("FirstName", "გვარი");
  dataGridView1.Columns.Add("LastName", "სახელი");
  dataGridView1.Columns.Add("Sex", "სქესი");
  dataGridView1.Columns.Add("Dab_Celi", "დაბად_წელი");
 // ველების სიგანის დაყენება -----
 for (i = 0; i < dataGridView1.Columns.Count; i++)
    dataGridView1.Columns[i].Width = 75;
  // სტრიქონების შევსება ----------
 dataGridView1.Rows.Add("1","აბაშიძე","გოჩა","კაცი","1990");
 dataGridView1.Rows.Add("2","ბურძგლა","ბაჩუკი","კაცი","1989");
 dataGridView1.Rows.Add("3","გალოგრე","ჩარიტა","ქალი","1991");
 dataGridView1.Rows.Add("4","თოფურია","ძღაბი","ქალი","1992");
 ataGridView1.Rows.Add("5","ზვიადაური","ჯოყოლა","კაცი","1985");
}
```

პროგრამის ამუშავებით მიიღება ასეთი შედეგი (ნახ.4.8).




ნახ.4.8

როგორც აღვნიშნეთ, კოდში ხისტადაა შეტანილი კონკრეტული მონაცემები სტუდენტების შესახებ. ნებისმიერი დამატება ან ცვლილება მოითხოვს პროგრამის გადაკეთებას, რაც არაა რეკომენდებული. ამის თავიდან აცილება შესაძლებელია მონაცემთა ბაზის გამოყენებით. შედეგიდან ჩანს, რომ Form1 ცარიელია, ხოლო Form2 შევსებულია საწყისი მონაცემებით.

ახლა განვიხილოთ ჩვენი კოდის მაგალითით DataGridView ცხრილში შეტანილი მონაცემების საფუძველზე მომხმარებლის მოთხოვნების პროგრამული დამუშავების შესაძლებლობანი.

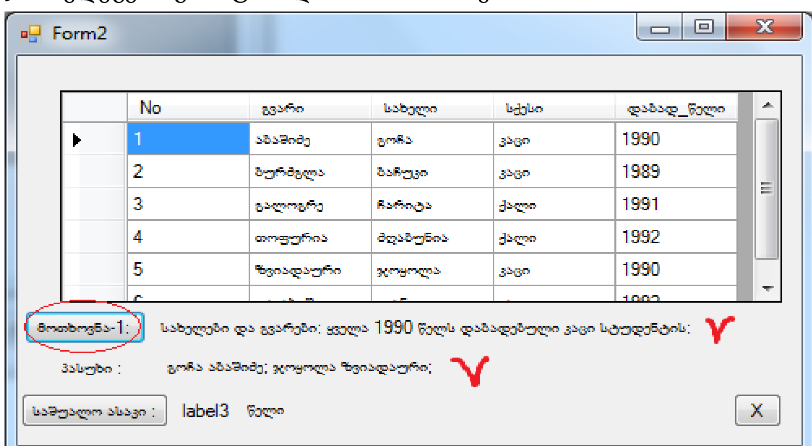
**ამოცანა_4.1:** ვიპოვოთ სახელები და გვარები 1990 წელს დაბადებული ყველა მამაკაცი სტუდენტის:

მოთხოვნის ფორმალური მხარე მდგომარეობს „გვარი“ ველის მნიშვნელობების გამობეჭდვაში, ველი „სქესი“=“კაცი“ მნიშვნელობისათვის.

საჭიროა Form2-ზე დავდოთ ბუტონი და მივაბათ მას 4.2_ლისტინგის პროგრამული კოდი.

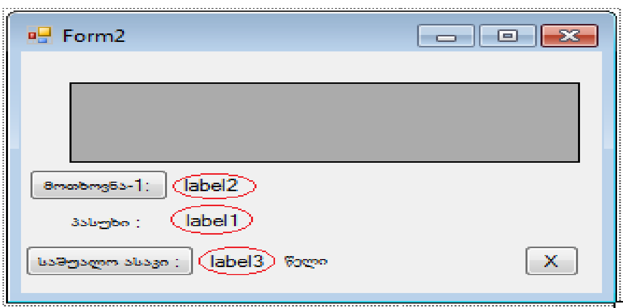
```
//---ლისტინგი_4.2--DataGridView-ში სტრიქონების ამორჩევა
//-- პირობით ----
private void button1_Click(object sender, EventArgs e)
  {
    label1.Text = " ";
    label2.Text = "სახელები და გვარები: 1990 წელს
                  დაბადებული ყველა  კაცი  სტუდენტის:";
 for (int i = 0; i < dataGridView1.Rows.Count; i++)
    {
   if (dataGridView1.Rows[i].Cells[3].Value == "კაცი" &&
      dataGridView1.Rows[i].Cells[4].Value == "1990")
     {
```

```
    label1.Text += dataGridView1.Rows[i].Cells[2].Value
    + " " +dataGridView1.Rows[i].Cells[1].Value + "; ";
   }
  }
} // შედეგები გამოტანილია 4.9 ნახაზზე.
```




ნახ.4.9

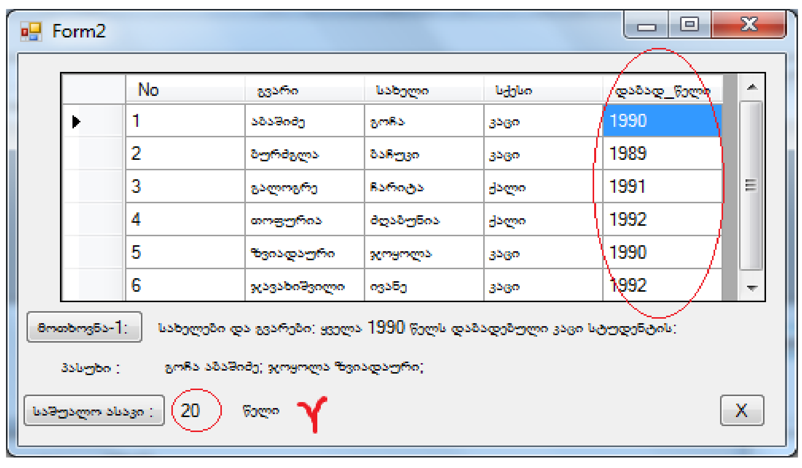
**ამოცანა_4.2:** შევადგინოთ კოდი ღილაკისათვის „საშუალო ასაკი“, რომელიც გამოიტანს label3-ში ყველა სტუდენტის საშუალო ასაკის მნიშვნელობას (ნახ. 4.10). 11_3 ლისტინგზე მოცემულია ეს კოდი.



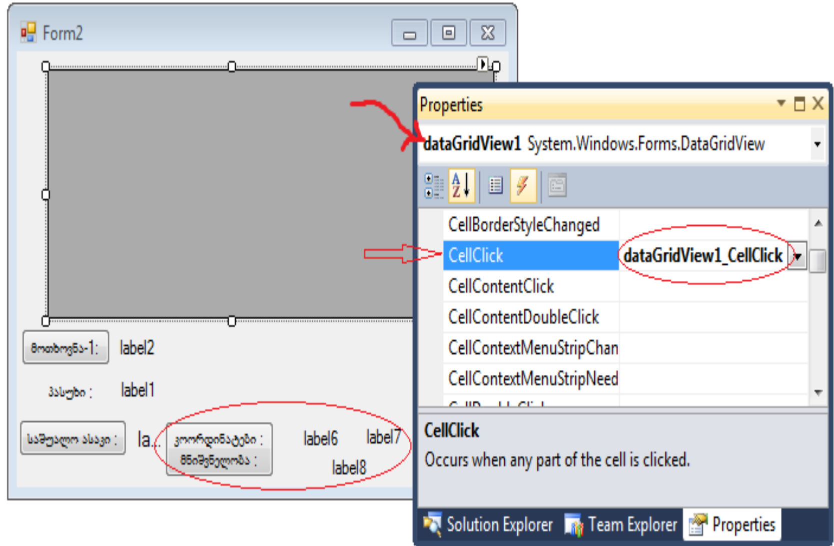

ნახ.4.10

```
 // ლისტინგი_4.3 --- DataGridView საშუალო ასაკის ანგარიში ----
-
private void button2_Click(object sender, EventArgs e)
{
  int Birth_Year, Averag_Age, Sum=0;
  DateTime now = DateTime.Now; // მიმდინარე თარიღი
  int age,a,b;
  for (int i = 0; i < dataGridView1.Rows.Count; i++)
   {
    Birth_Year=Convert.ToInt32(dataGridView1.Rows[i].Cells[4].Value);
     a = now.Year;  b = Birth_Year;   age = a - b;
     Sum += age;
   }
  Averag_Age = Sum / dataGridView1.Rows.Count;
  label3.Text = Averag_Age.ToString();
 }
```

შედეგები ასახულია 4.11 ნახაზზე.



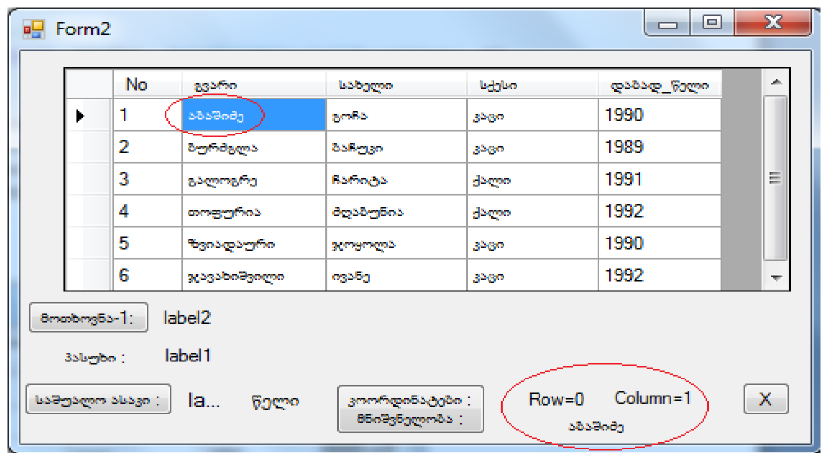

ნახ.4.11



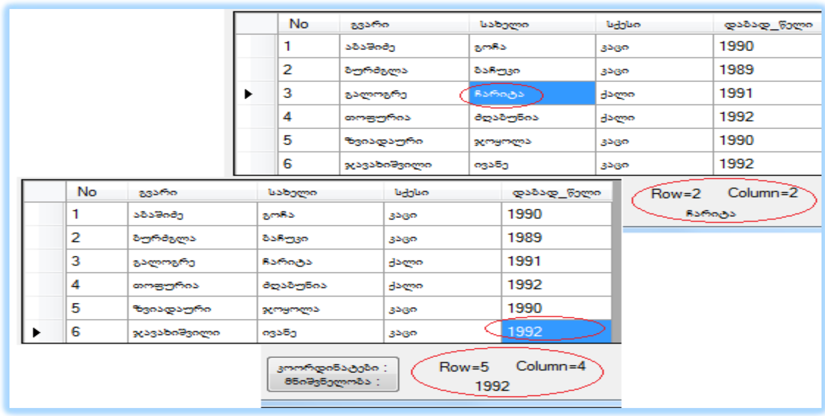

ნახ.4.12

```
// ლისტინგი_4.4 ---- ცხრილის კოორდინატები -----

private void dataGridView1_CellClick(object sender,
                                    DataGridViewCellEventArgs e)
 {
  label8.Text = "";
  label6.Text = "Row="+e.RowIndex.ToString();
  label7.Text = "Column="+e.ColumnIndex.ToString();
  if (e.RowIndex>=0 && e.ColumnIndex >=0)
    label8.Text += dataGridView1.Rows[e.RowIndex].
                                     Cells[e.ColumnIndex].Value;
 }
```

შდეგები ასახული 4.13-ა,ბ ნახაზზე.




ნახ.4.13-ა




ნახ.4.13-ბ

**დავალება:**

ააგეთ ცხრილი „წიგნები“, რომელსაც ექნება შემდეგი ველები: შიფრი, დასახელება, ავტორი, გამოცემის_წელი, გამომცემლობა, ქალაქი, ჟანრი, ენა, გვერდების_რაოდ., ფასი. შეიტანეთ 10 სტრიქონი და განახორციელეთ მასში წიგნის ძებნა ავტორით, წიგნების ძებნა ჟანრით, დაალაგეთ წიგნები გამოცემის წლის კლებადობით.

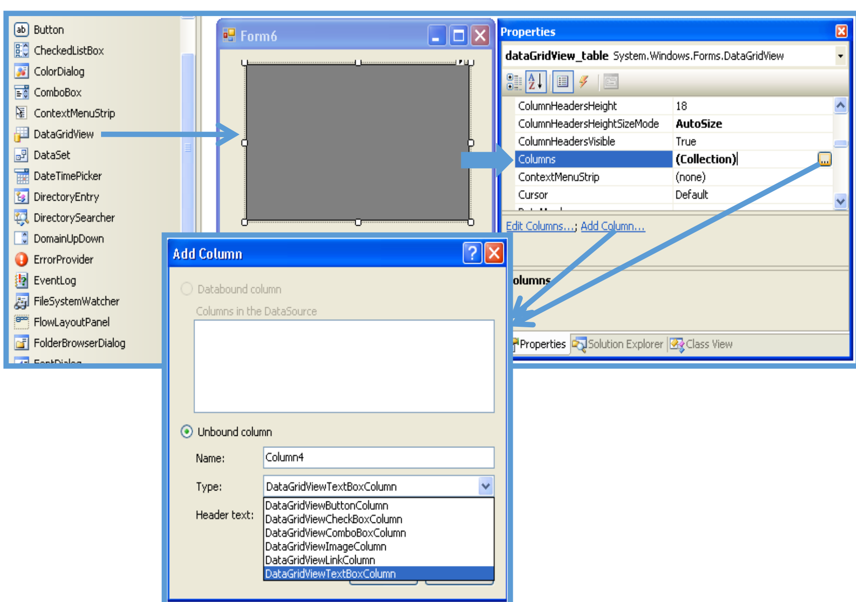
## 5. DataGridViewComboBoxColumn და DataGridViewTextboxColumn კლასების გამოყენება ცხრილებთან სამუშაოდ

### ლაბორატორიული სამუშაო N5

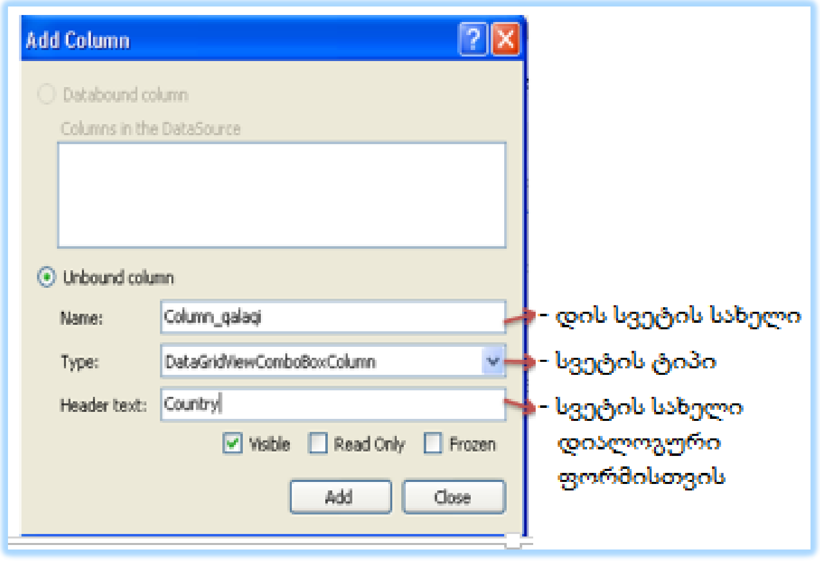
მიზანი: მონაცემთა ბაზის ცხრილებთან მუშაობის დროს DataGridViewComboBoxColumn და DataGridViewTextboxColumn კლასების გამოყენების შესწავლა.

### 1. თეორიული ნაწილი

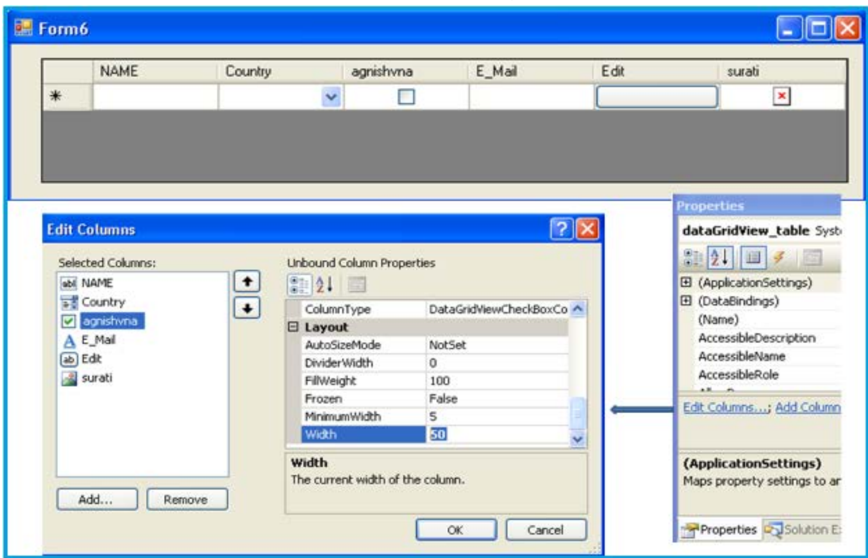
განიხილება შემდეგი საკითხები: DataGridView კომპონენტის სვეტების შექმნა სხვადასხვა ტიპის კომპონენტებით, DataGridView კომპონენტში DataGridViewComboBoxColumn და DataGridViewTextboxColumn სვეტების გამოყენება, DataGridView-ს სხვადასხვა ტიპის ველების შევსება DataTable ვირტუალური ცხრილიდან



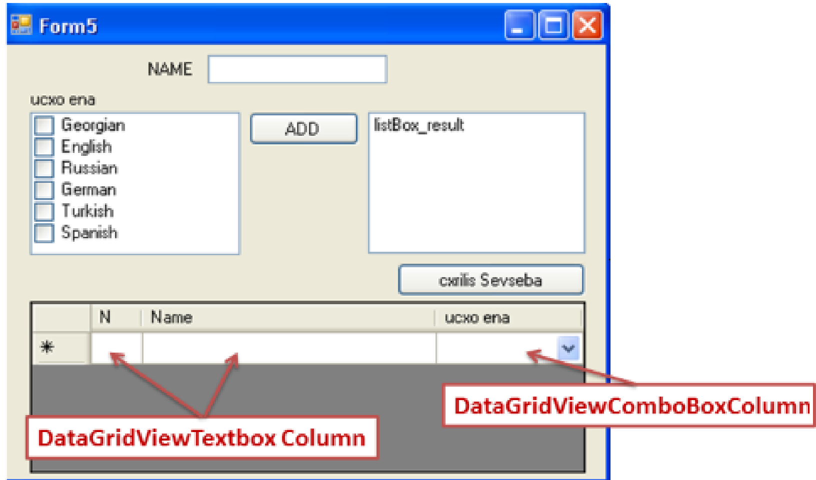

ნახ.5.1. DataGridView -ს სვეტების შექმნა სხვადასხვა ტიპის კომპონენტით



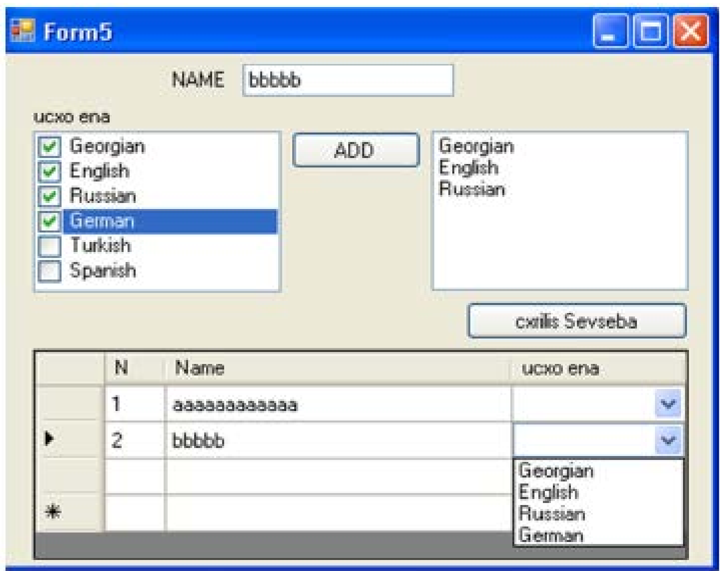

ნახ.5.2. DataGridView -ში სხვადასხვა ტიპის სვეტების დამატება თვისებათა ფანჯრიდან




ნახ.5.3. სვეტების დამატების შედეგი




ნახ.5.4. CheckedLisBox და TextBox კომპონენტებიდან DataGridView-ის ველების შევსება




ნახ.5.5. შედეგი

```
public partial class Form5 : Form
  {
      DataTable table = new DataTable();
      public Form5()
      {
        InitializeComponent();
        basa();
      }
```

მეთოდის გამოძახება

```
void basa()
    {
        table = new DataTable();
       DataColumn col_N = new DataColumn("", typeof(int));
       DataColumn col_name = new DataColumn("", typeof(string));
       DataColumn col_combo = new DataColumn("", typeof(List<String>));

       table.Columns.AddRange(new DataColumn[] { col_N, col_name, col_combo });
    }
```

ვირტუალური ცხრილის შექმნა

სტრიქონული ტიპის მასივის სვეტი

ნახ.5.6. DataTable ვირტუალური ცხრილის შექმნა DataGridView-ს ველების შესავსებად

```
private void button_table_Click(object sender, EventArgs e)
      {
         String name = textBox_name.Text;
         table = this.table;
         List<String> cheked_list = new List<string>();
         int lists_count = checkedListBox_list_tipy.Items.Count;
         for (int i = 0; i < lists_count; ++i)
         {
            if (checkedListBox_list_tipy.GetItemCheckState(i) == CheckState.Checked)
            {
               String result = (String)checkedListBox_list_tipy.Items[i];
               cheked_list.Add(result);
            }
         }
         int tableN = table.Rows.Count;
         table.Rows.Add(new object[] { tableN + 1, name, cheked_list });
         fill_Table(table);
      }
```

სტრიქონული ტიპის მასივის შექმნა

სტრიქონული ტიპის მასივში ელემენტების დამატება

dataGridView1 ცხრილში მონაცემების დამატების მეთოდის გამოძახება

ნახ.5.7. ვირტუალური ცხრილის შევსება CheckedListBox და TextBox კომპონენტებიდან

```
public void fill_Table(DataTable  table)
 {
    int tableN = table.Rows.Count;
    if (tableN != 0)
    {
       for (int i = 0; i < tableN; i++)
       {
          dataGridView1.Rows.Add();
          dataGridView1.Rows[i].Cells[0].Value  = (int)table.Rows[i][0];
          dataGridView1.Rows[i].Cells[1].Value  = (String)table.Rows[i][1];
          List<String> list_rezult = (List<String>)table.Rows[i][2];
          DataGridViewComboBoxCell cells = (DataGridViewComboBoxCell)dataGridView1.Rows[i].Cells[2];
          cells.Items.Clear();
          for (int j = 0; j < list_rezult.Count;  j++)
          {
             cells.Items.Add(list_rezult[j]);
          }
       }
    }
 }
```

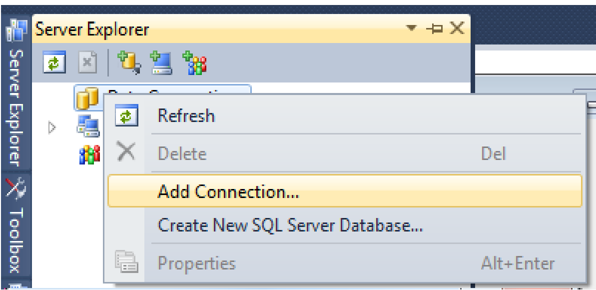
ნახ.5.8. fill_Table(table) მეთოდი dataGridView-ს ველების შესავსებად DataTable ვირტუალური ცხრილიდან

## 2.6. C# აპლიკაციის მუშაობა Ms Access ბაზასთან

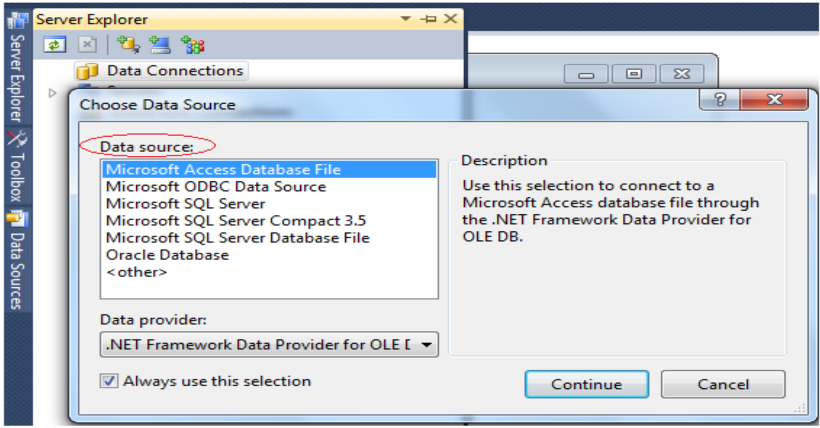
### ლაბორატორიული სამუშაო N6

მიზანი: მონაცემთა ბაზების მართვის სისტემებისა და მომხმარებელთა ინტერფეისების ერთობლივი გამოყენების პროექტების აგების პროცესების მენეჯმენტის შესწავლა Ms Access-პაკეტით.

სისტემის მენიუდან View | Server Explorer-ით გამოვიტანოთ ფანჯარა (ნახ.6.1) და ავირჩიოთ Add Connection.

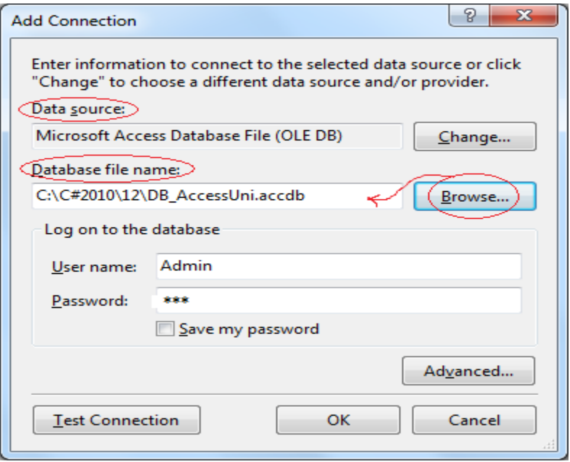



ნახ.6.1

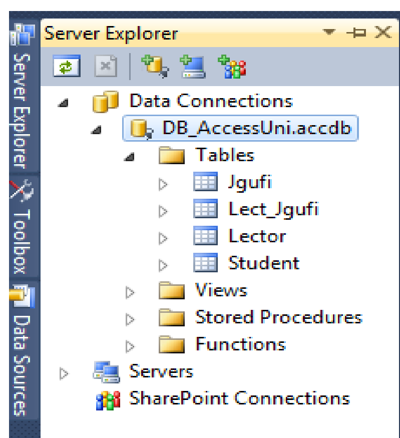



ნახ.6.2

6.2 ნახაზზე Data Source ველში ავირჩიოთ სტრიქონი Microsoft Access Database File და Continue. მივიღებთ 6.3 ფანჯარას, რომელშიც Browse ღილაკით გამოვიძახებთ კატალოგის მართვის დიალოგის ფანჯარასა და მივუთითებთ ჩვენ მიერ წინასწარ მომზადებულ Ms Access-ის ბაზის ფაილს. მაგალითად, როგორც ეს Database file name ველშია ჩაწერილი.



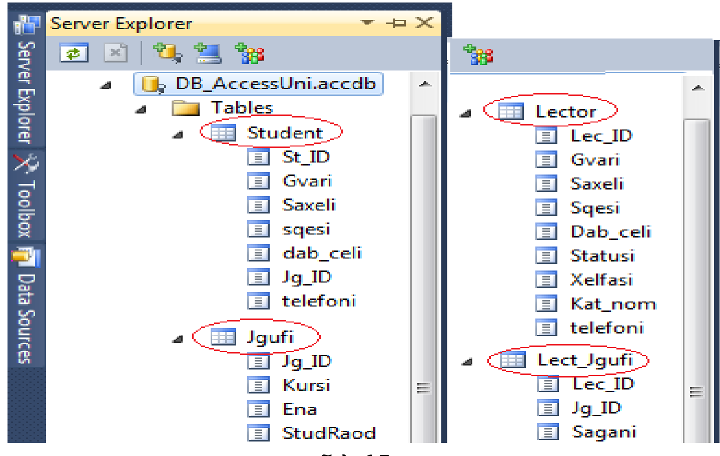

ნახ.6.3




აქვე, საჭიროებისამებრ, მიეთითება User name და Password. ამის შემდეგ Server Explorer-ში გამოჩნდება 6.4 ნახაზზე მოცემული სურათი.

ნახ.6.4

როგორც ვხედავთ, Data Connection-ში უნივერსიტეტის მონაცემთა ბაზის ფაილი - **DB_AccessUni.accdb** გამოჩნდა, რომელიც შედგება ოთხი ცხრილისაგან: Jgufi, Lect_Jgufi, Lector და Student. ცხრილები შედგება ველებისაგან, რომელთაგან ერთ-ერთი გასაღე-ბურია (ინდექსი): Lec_ID, St_ID, Jg_ID და ერთიც შედგენილი გასაღებია ორი ატრიბუტით: Lec_ID+Jg_ID (ნახ.6.5).

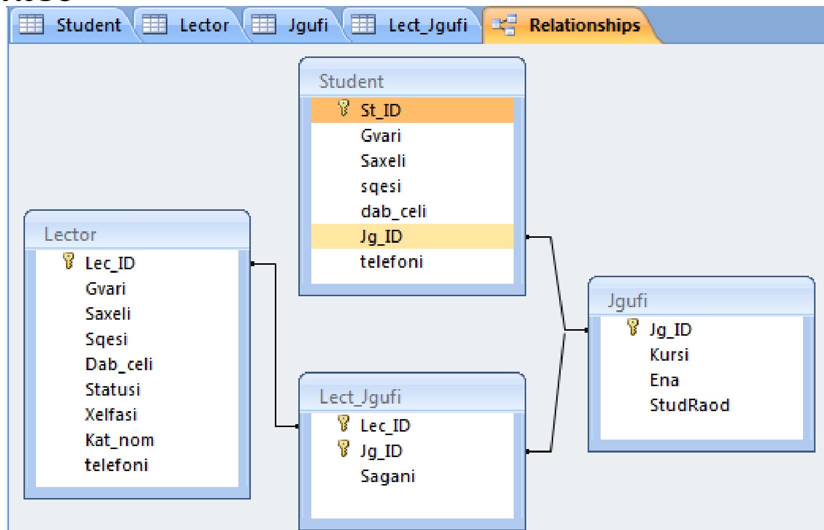



ნახ.6.5

რელაციური კავშირები მათ საფუძველზეა აგებული (ნახ.6.6).

ცხრილში **სტუდენტი** პირველადი გასაღებური ატრიბუტია St_ID ინდექსი, ხოლო მისი მეორადი გასაღებია Jg_ID, რომლითაც იგი უკავშირდება ცხრილს **ჯგუფი,** პირველადი ინდექსით Jg_ID. ესაა კავშირი 1:N, რომელიც ასახავს ბიზნეს-წესს (არსებულ კანონზომიერებას), რომ ერთი სტუდენტი შეიძლება იყოს მხოლოდ ერთ ჯგუფში და ერთ ჯგუფში შეიძლება იყოს რამდენიმე (N) სტუდენტი. ასევე, **ლექტორი** ასწავლის რამდენიმე (N) ჯგუფს, მაგრამ ჯგუფსაც ჰყავს რამდენიმე (M) ლექტორი. ესაა M:N კავშირი. მისი რეალიზაცია არაა შესაძლებელი **ლექტორი**-ს და **ჯგუფი**-ს პირდაპირი კავშირით (განმეორებადი ველების პრობლემა !). ამისათვის შემოტანილია დამატებითი ცხრილი (რელაცია)

**ლექტორი_ჯგუფი.** მასში შედგენილი ინდექსი იქმნება Lec_ID+Jg_ID, რომლებიც უკავშირდება ცალკ-ცალკე **ლექტორსა და ჯგუფს** (ნახ.6.7).

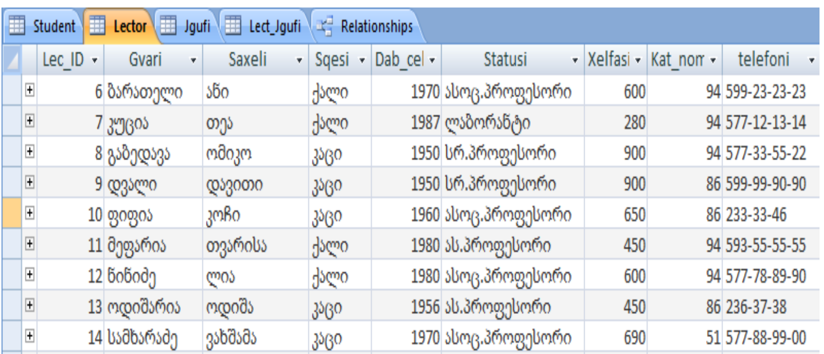



ნახ.6.6

Student | Lector | Jgufi | Lect_Jgufi | Relationships

| St_ID | Gvari | Saxeli | sqesi | dab_cel | Jg_ID | telefoni |
|---|---|---|---|---|---|---|
| 8 | აკოფოვი | რობერტინო | კაცი | 1989 | 108936 | 297-11-11-11 |
| 1 | ალავიძე | ალეკო | კაცი | 1990 | 108935 | 222-22-20 |
| 2 | ბურდული | ნინო | ქალი | 1991 | 108935 | 137-33-33 |
| 3 | ბურძგლა | დიტო | კაცი | 1989 | 108935 | 599-10-20-20 |
| 4 | გაბედავა | ვახტანგ | კაცი | 1992 | 108935 | 333-67-89 |
| 5 | გაბელია | ცუცა | ქალი | 1992 | 108935 | 222-45-67 |
| 13 | გალოგრე | ხვიჩა | კაცი | 1989 | 108937 | 270-44-44 |
| 6 | დანელია | მიმოზა | ქალი | 1990 | 108935 | 577-44-44-45 |
| 9 | დოლიძე | რიჩარდი | კაცი | 1990 | 108936 | 597-34-56-78 |
| 14 | დუნდუა | გოჩა | კაცი | 1991 | 108937 | 599-22-33-44 |
| 15 | ვასაძე | სოკრატე | კაცი | 1985 | 108937 | 577-33-67-55 |
| 10 | ზარანდია | მუშნი | კაცი | 1980 | 108936 | 597-12-23-34 |
| 11 | თოფურია | ძღაბი | ქალი | 1991 | 108936 | 577-10-10-10 |
| 12 | კეკელია | კეკელა | ქალი | 1992 | 108936 | 579-30-30-30 |
| 7 | ხვიტია | ძუკუ | კაცი | 1992 | 108935 | 593-45-67-89 |
| 16 | ჯალაღონია | მაცი | კაცი | 1992 | 108937 | 577-99-00-00 |

ნახ.6.7-ა



Student | Lector | Jgufi | Lect_Jgufi | Relationships

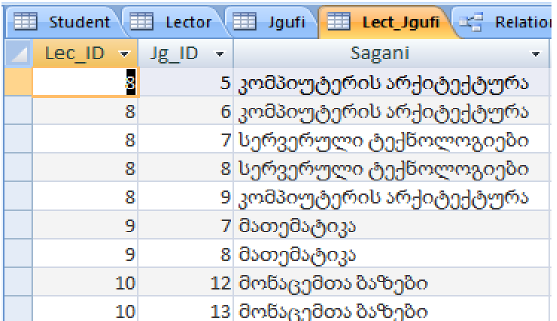
| Lec_ID | Gvari | Saxeli | Sqesi | Dab_cel | Statusi | Xelfasi | Kat_nom | telefoni |
|---|---|---|---|---|---|---|---|---|
| 6 | ბარათელი | ანი | ქალი | 1970 | ასოც.პროფესორი | 600 | 94 | 599-23-23-23 |
| 7 | კუცია | თეა | ქალი | 1987 | ლაბორანტი | 280 | 94 | 577-12-13-14 |
| 8 | გაბედავა | ომიკო | კაცი | 1950 | სრ.პროფესორი | 900 | 94 | 577-33-55-22 |
| 9 | დვალი | დავითი | კაცი | 1950 | სრ.პროფესორი | 900 | 86 | 599-99-90-90 |
| 10 | ფიფია | კოჩი | კაცი | 1960 | ასოც.პროფესორი | 650 | 86 | 233-33-46 |
| 11 | მეფარია | თვარისა | ქალი | 1980 | ას.პროფესორი | 450 | 94 | 593-55-55-55 |
| 12 | ნინიძე | ლია | ქალი | 1980 | ასოც.პროფესორი | 600 | 94 | 577-78-89-90 |
| 13 | ოდიშარია | ოდიშა | კაცი | 1956 | ას.პროფესორი | 450 | 86 | 236-37-38 |
| 14 | სამხარაძე | ვახშამა | კაცი | 1970 | ასოც.პროფესორი | 690 | 51 | 577-88-99-00 |

ნახ.6.7-ბ

Student | Lector | Jgufi | Lect_Jgufi | Relatio

| Jg_ID | Kursi | Ena | StudRaod |
|---|---|---|---|
| 108050 | 2 | ქართული | 29 |
| 108051 | 2 | ქართული | 30 |
| 108059 | 2 | რუსული | 12 |
| 108835 | 4 | ქართული | 29 |
| 108836 | 4 | ქართული | 25 |
| 108935 | 3 | ქართული | 28 |
| 108936 | 3 | ქართული | 30 |
| 108937 | 3 | ქართული | 17 |
| 108940 | 3 | ინგლისური | 20 |

ნახ.6.7-გ

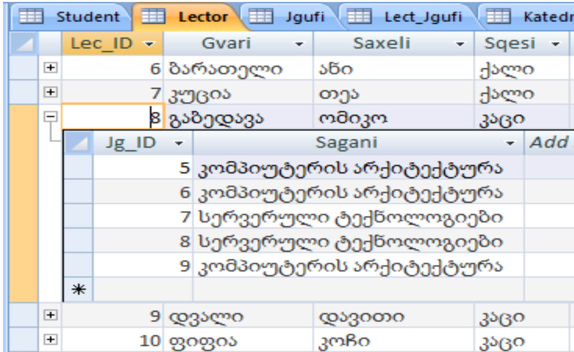


Student | Lector | Jgufi | Lect_Jgufi | Relatio

| Lec_ID | Jg_ID | Sagani |
|---|---|---|
| 8 | 5 | კომპიუტერის არქიტექტურა |
| 8 | 6 | კომპიუტერის არქიტექტურა |
| 8 | 7 | სერვერული ტექნოლოგიები |
| 8 | 8 | სერვერული ტექნოლოგიები |
| 8 | 9 | კომპიუტერის არქიტექტურა |
| 9 | 7 | მათემატიკა |
| 9 | 8 | მათემატიკა |
| 10 | 12 | მონაცემთა ბაზები |
| 10 | 13 | მონაცემთა ბაზები |

ნახ.6.7-დ

ჩანაწერის წინ „+” სიმბოლო ხსნის კავშირს მეორე, იერარქიულად დაქვემდებარებულ ცხრილთან (ნახ.6.8).




ნახ.6.8

ამგვარად, მონაცემთა ბაზა DB_AccessUni.acce მზადაა. ახლა განვიხილოთ C# პროგრამიდან მონაცემთა ბაზასთან წვდომის საკითხი. ეს პროცესი შედგება ოთხი ბიჯისაგან:

- მონაცემთა ბაზასთან მიერთება (რაც ზემოთ განვიხილეთ Server Explorer->Data Connection-ში );
- SQL-ბრძანების გადაცემა მონაცემთა ბაზაზე;
- SQL-ბრძანების შეფასება (და შესრულება);
- მონაცემთა ბაზასთან კავშირის დახურვა.

SQL-ბრძანება წარმოადგენს სტრუქტურირებული მოთხოვნების ენაზე დაწერილ სკრიპტს, რომელიც გასაგებია მონაცემთა ბაზების მართვის სისტემისათვისა და ასრულებს მას. ძირითადად, არსებობს ორი ტიპის მოთხოვნა:

- select : მონაცემთა ამორჩევის SQL-ბრძანება;
- insert, delete, update : მონაცემთა ბაზაში ცვლილებების განსახორციელებელი SQL-ბრძანებები.

C# პროგრამულ პროექტს მონაცემთა ბაზასთან სამუშაოდ სჭირდება სახელსივრცე OleDb, რომელიც using.System.Data.OleDB ბრძანებითაა რეალიზებული.

SQL-ბრძანებები Ms_Access ბაზაში გადაიგზავნება OleDb სახელსივრცის OleDBCommand კლასის ობიექტით. ამ კლასის ორი მნიშვნელოვანი თვისებაა: Connection (დავალება ბაზასთან დასაკავშირებლად, საითაც გაიგზავნება SQL-მოთხოვნა) და CommandText (თვით SQL ბრძანების ტექსტი).

მოთხოვნის ტიპებისაგან დამოკიდებულებით (არჩევითი, ცვლილებების), OleDBCommand კლასს აქვს შემდეგი მეთოდები:

ExecuteReader() - აქვს select მოთხოვნის გაგზავნისა და შედეგების მიღების ფუნქცია;

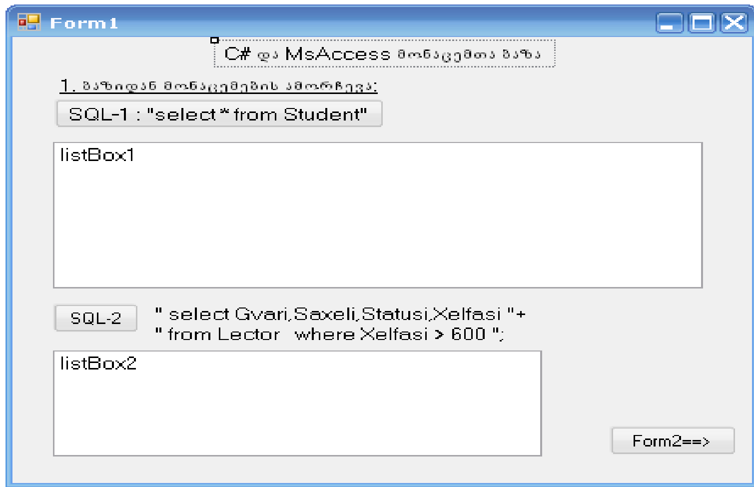
ExecuteNonQuery() - ემსახურება ცვლილების (insert, delete, update) აქციის ჩატარებასა და რიცხვის მიღებას, რომელიც გვიჩვენებს, მონაცემთა ბაზის რამდენ ჩანაწერს შეეხო ეს პროცედურა.

მონაცემთა ბაზიდან select-მოთხოვნით ამორჩეული ჩანაწერები (ველებით და მნიშვნელობებით) ინახება OleDb სახელსივრცის OleDBReader კლასის ობიექტში. განვიხილოთ კონკრეტული მაგალითი.

**ამოცანა_6.1:** DB_AccessUni.acce უნივერსიტეტის მონაცემთა ბაზაში: 1 - ვნახოთ ყველა სტუდენტის ყველა ატრიბუტის მნიშვნელობა (ანუ მთლიანი ინფორმაცია); 2 - ვიპოვოთ იმ ლექტორთა გვარი, სახელი, სტატუსი და ხელფასი, რომელთა ხელფასი მეტია 600 ლარზე.

6.9-ა ნახაზზე 1-ელ მოთხოვნას შეესაბამება SQL-1, ხოლო მეორეს კი - SQL-2 "select" კონსტრუქცია. ისინი ორ ღილაკზეა მიმაგრებული. საილუსტრაციო მაგალითში შედეგები გამოიტანება listBox1 და listBox2 ველებში.

C#-პროგრამის კოდი ამ ორი ღილაკისათვის მოცემულია 6.1_ლისტინგში.




ნახ.6.9-ა

```
// ლისტინგი_6.1 – C# + Ms Access -------
private void button1_Click(object sender, EventArgs e)
{
  OleDbConnection con = new OleDbConnection();
  OleDbCommand cmd = new OleDbCommand();
  OleDbDataReader reader;

con.ConnectionString =
      "Provider=Microsoft.ACE.OLEDB.7.0;" +
      "Data Source=C:\\C#2010\\WinADO\\DB_AccessUni.accdb";

cmd.Connection = con;
cmd.CommandText = "select * from Student";
try
 {
   con.Open();
   reader = cmd.ExecuteReader();
   listBox1.Items.Clear();
     while (reader.Read())
```

```
      {
       listBox1.Items.Add(reader["St_ID"] + " : " +
                    reader["Gvari"] + " : " +
                    reader["Saxeli"] + " : " +
                    reader["sqesi"] + " : " +
                    reader["dab_celi"] + " : " +
                    reader["Jg_ID"] + " : " +
                    reader["telefoni"]);
      }
      reader.Close();
      con.Close();
     }
    catch (Exception ex)
     {
      MessageBox.Show(ex.Message);
     }
  }
// end button1_click ----------------

private void button2_Click(object sender, EventArgs e)
  {
    OleDbConnection con = new OleDbConnection();
    OleDbCommand cmd = new OleDbCommand();
    OleDbDataReader reader;

 con.ConnectionString =
       "Provider=Microsoft.ACE.OLEDB.7.0;" +
       "Data Source=C:\\C#2010\\WinADO\\DB_AccessUni.accdb";
 cmd.Connection = con;
 cmd.CommandText =
     "select  Gvari,Saxeli,Statusi,Xelfasi " +
     "from Lector " +
     "where Xelfasi > 600 ";
 try
```

```
{
  con.Open();
  reader = cmd.ExecuteReader();
  listBox2.Items.Clear();
  while (reader.Read())
   {
     listBox2.Items.Add( reader["Gvari"] + " : " +
                    reader["Saxeli"] + " : " +
                    reader["Statusi"] + " : " +
                    reader["Xelfasi"]);
     }
    reader.Close();
    con.Close();
   }
  catch (Exception ex)
    {
      MessageBox.Show(ex.Message);
    }
  }
// end button2_click ----------------
```

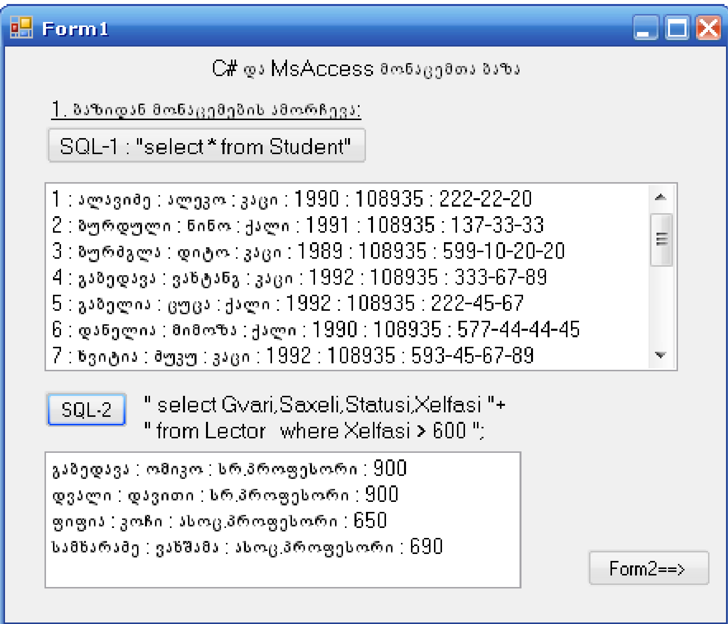
შენიშვნა: პროგრამის თავში using-სტრიქონებში უნდა ჩაემატოს:

using System.Data.OleDb;

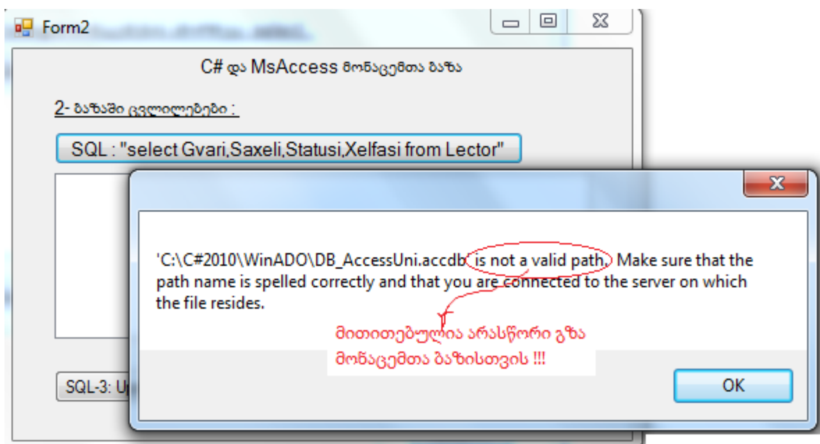
6.9-ბ ნახაზზე ნაჩვენებია აღნიშნული მოთხოვნების შესრულების შედეგები MsAccess ბაზის Student და Lector ცხრილებიდან. პირველში „ * ”- ნიშნავს „ყველა“ - ველს, ხოლო მეორეში აიღება მხოლოდ „გვარი, სახელი, სტატუსი და ხელფასი“.

პროგრამაში **try…catch** ბლოკით ხდება მონაცემთა ბაზასთან წვდომის პროცესში შესაძლო შეცდომების აღმოჩენა, რაც აადვილებს პროგრამისტის მუშაობას. მაგალითად, ინფორმაციის მიღება, რომ მონაცემთა ბაზის ფაილი არაა მითითებულ patch-

კატალოგში (ნახ.6.10), ან რომ SQL-მოთხოვნის სინტაქსში შეცდომაა და ა.შ.




ნახ.6.9-ბ




ნახ.6.10

Open() მეთოდით ხდება ბაზასთან პროგრამის კავშირის გახსნა. შემდეგ, ExecuteReader() მეთოდით მოთხოვნა გადაეგზავნება ბაზას. შედეგები ბრუნდება OleDbReader კლასით, რაც ხორციელდება reader მიმთითებლით (მაჩვენებლით).

ვინაიდან წინასწარ არაა ცნობილი თუ რამდენი სტრიქონი იქნება შედეგში, გამოიყენება ListBox, რომელიც წინასწარ სუფთავდება.

Read() მეთოდი გვაწვდის ბაზიდან ერთ ჩანაწერს (სტრიქონს) და ამავდროულად სპეც - მაჩვენებლით მიუთითებს მომდევნო ჩანაწერზე. თუ ჩანაწერი ბოლოა, მაშინ მაჩვენებლის მნიშვნელობა ხდება false. ეს მართვა ხორციელდება while ციკლით try-ბლოკში.

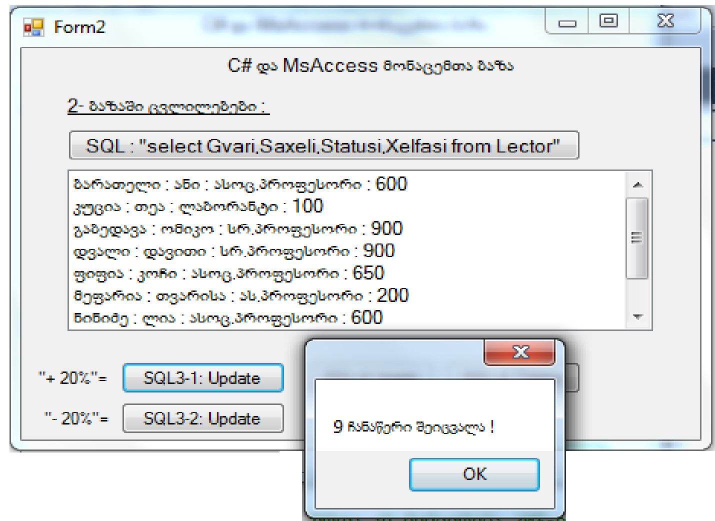
ჩანაწერის შიგნით ველების მნიშვნელობები შეესაბამება მათ ნომრებს ან დასახელებებს. შესაძლებელია ასევე ყველა ველის გამოტანა („ * “-ით), რომლებიც სპეც - გამყოფითაა (მაგ., „ : “) დაცილებული. ბოლოს, Reader ობიექტი და კავშირი უნდა დაიხუროს Close() მეთოდით.

**ამოცანა_6.2:** მონაცემთა ბაზაში „უნივერსიტეტი“ DB_AccessUni.acce საჭიროა ცვლილებების განხორციელება insert(), delete() და update() მეთოდების გამოყენებით. Form2-ზე მოვათავსოთ შესაბამისი ელემენტები და განვახორციელოთ ტრანზაქციები:

ა) ყველა ლექტორის ხელფასი გაიზარდოს 20%-ით. (ამასთანავე შესაძლებელი უნდა იყოს საწყისი მონაცემების აღდგენა, ანუ შემცირდეს ხელფასები 16.67 %-ით);

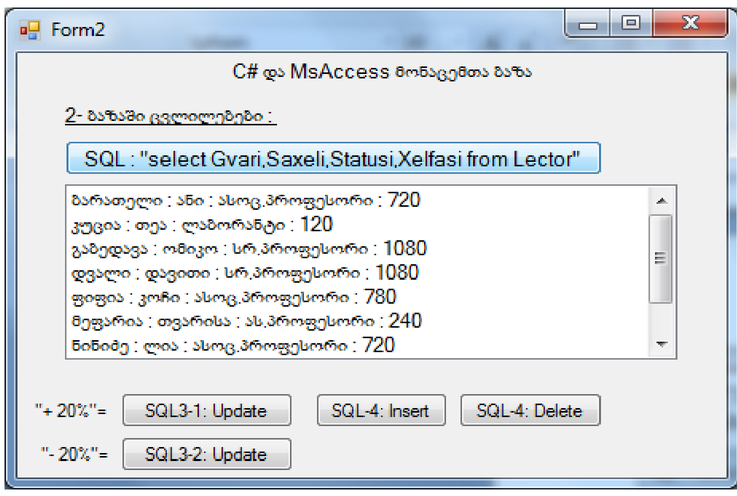
ბ) 108935 ჯგუფში დაემატოს ახალი სტუდენტი გვარით „ახალაძე“, მეტსახელით „ნოუთბუკა“ და ა.შ.;

გ) ამოიშალოს ბაზიდან 108836 ჯგუფი, რომელმაც დაასრულა 4-წლიანი სწავლების კურსი.




**ნახ.6.11**

ლექტორთა ხელფასების შეცვლილი შედეგები მოცემულია 6.12 ნახაზზე.




**ნახ.6.12**

SQL3-2 ღილაკით შემცირდება ხელფასის რაოდენობა 20%-ით, ანუ აღდგება პირველადი მონაცემები ბაზაში.

შესაბამისი update - კოდი მოცემულია 6.2_ლისტინგში.

```
// ლისტინგი_6.2 --- Ms Acees  “update” ------
using System;
using System.Data.OleDb;
using System.Windows.Forms;
namespace WinADO
{
 public partial class Form2 : Form
 {
  public Form2()  {  InitializeComponent();  }
  private void button1_Click(object sender, EventArgs e)
   {
    OleDbConnection con = new OleDbConnection();
    OleDbCommand cmd = new OleDbCommand();
    int Raod;
```

```
   // ხელფასის ზრდა ან შემცირება 20%-ით ----
   con.ConnectionString =
         "Provider=Microsoft.ACE.OLEDB.7.0;" +
         "Data Source=C:\\C#2013\\12\\DB_AccessUni.accdb";
   cmd.Connection = con;
 if(ReferenceEquals(sender,button1))
    // op = "*"  or " /" -----
     cmd.CommandText = "update Lector set Xelfasi=Xelfasi * 1.2";
 else
     cmd.CommandText = "update Lector set Xelfasi=Xelfasi / 1.2";
 try
  {
    con.Open();
    Raod = cmd.ExecuteNonQuery();
     MessageBox.Show(Raod + " ჩანაწერი შეიცვალა !");
    con.Close();
  }
  catch (Exception ex)
   {
     MessageBox.Show(ex.Message);
   }
}
private void button4_Click(object sender, EventArgs e)
 { OleDbConnection con = new OleDbConnection();
   OleDbCommand cmd = new OleDbCommand();
   OleDbDataReader reader;
   con.ConnectionString =
             "Provider=Microsoft.ACE.OLEDB.7.0;" +
             "Data Source=C:\\C#2013\\12\\DB_AccessUni.accdb";
  cmd.Connection = con;
    cmd.CommandText = "select Gvari,Saxeli,Statusi,
                                          Xelfasi from Lector";
  try
   {
```

```
    con.Open();
    reader = cmd.ExecuteReader();
    listBox1.Items.Clear();
    while (reader.Read())
     {
      listBox1.Items.Add(reader["Gvari"] + " : " +
                    reader["Saxeli"] + " : " +
                    reader["Statusi"] + " : " +
                    reader["Xelfasi"]);
     }
    reader.Close();
     con.Close();
    }
  catch (Exception ex)
    {
     MessageBox.Show(ex.Message);
    }
```

კოდში update ტიპის ცვლილებისათვის DB_MsAccessUni.accedb ფაილში გამოყენებული კონსტრუქცია:

**"update Lector set Xelfasi=Xelfasi * 1.2"**

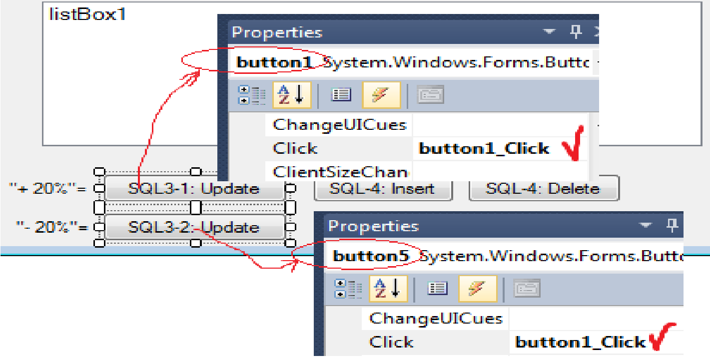
რაც ნიშნავს ცხრილის აქტუალიზაციას (update… set), სახელით Lector და ცხრილის ველის (ატრიბუტის) ცვლილების ალგორითმს (Xelfasi=Xelfasi * 1.2), რომელიც უნდა შესრულდეს ჩანაწერებზე;

try…catch ბლოკში იხსნება ბაზა (Open-ით) და გამოიძახება მეთოდი ExecuteNonQuery(), რომელიც დააბრუნებს ჩანაწერების რაოდენობას (Raod), რომელთაც შეეხო ცვლილება. ეს ინფორმაცია გამოიტანება MessageBox-ით.

მონაცემთა ბაზის ცხრილში Lector ველისათვის Xelfasi უნდა მოხდეს მნიშვნელობათა გაზრდა 20%-ით (button1: SQL3-1) ან შემცირება (button5: SQL3-2), ნახ.6.13.

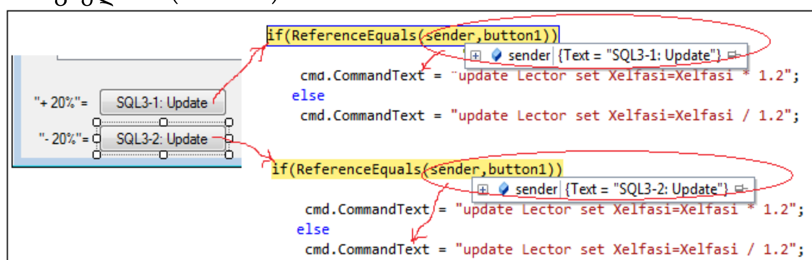
ამის განსახორციელებლად კოდში გამოყენებულია მეთოდი ReferenceEquals(sender object). ამ მეთოდით განისაზღვრება - აქვს თუ არა ორ ობიექტს ერთი მისამართი, ანუ ინახება თუ არა ისინი მეხსიერების ერთი და იმავე უჯრედში. თუ „კი“, მაშინ „true“ და

სრულდება გამრავლება (ხელფასის მომატება), ხოლო თუ სხვადასხვა ობიექტია, მაშინ გვექნება “false” და შესრულდება გაყოფა (ხელფასის შემცირება).




ნახ.6.13

კოდის ReferencialEquals(sender, button1) მეთოდში, იმის მიხედვით, თუ რომელი ბუტონი (SQL3-1 თუ SQL3-2) იქნება არჩეული, sender-ს მიენიჭება button1 ან button5 მიმთითებლის მნიშვნელობა (ნახ.6.14).




ნახ.6.14

ამგვარად, როდესაც sender და button1 ერთსა და იმავე მისამართზეა, მაშინ ხდება ხელფასის მომატება. დასასრულ, პროგრამის button4_Click ღილაკით listBox1-ში გამოიტანება MsAccess ბაზის Lector ცხრილის განახლებული მონაცემები (მომატებული ან დაკლებული ხელფასით).

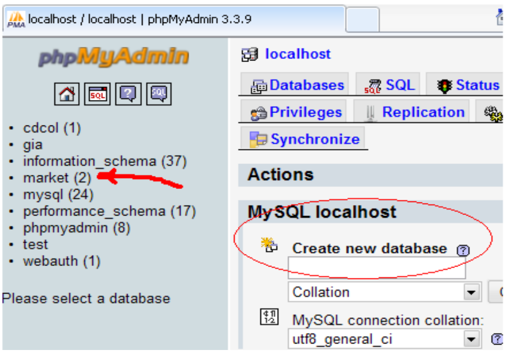
## 2.7. C# აპლიკაციის მუშაობა MySQL ბაზასთან

### ლაბორატორიული სამუშაო N7

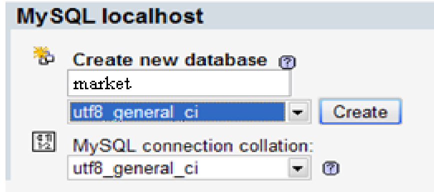
**მიზანი:** C# ენაზე აგებული აპლიკაციის მუშაობის შესწავლა MySQL მონაცემთა ბაზების მართვის სისტემასთან.

MySQL განაწილებული რელაციური ბაზა ერთ-ერთი აქტუალური და ფართოდ გამოყენებადი კლიენტ - სერვერული პაკეტია დღეისათვის, განსაკუთრებით php ვებ-ტექნოლოგიებში. ვგულისხმობთ, რომ MySQL სისტემა დაინსტალირებულია კომპიუტერზე (თუ არა, მაშინ ის ჩამოწერილ უნდა იქნეს http://www.mysql.com საიტიდან და დაინსტალირდეს).

როგორც წესი MySQL სისტემის გამოძახება ხდება რომელიმე ბრაუზერში, მაგალითად, Internet Explorer-ში url-მისამართში უნდა მიეთითოს: http://localhost/phpmyadmin/ სტრიქონი. 7.1-ა ნახაზზე გამოტანილია შედეგი. აქ ავირჩევთ ჩვენთვის საჭირო ბაზას, მაგალითად, market, რომელსაც აქვს 2 ცხრილი. თუ საჭიროა ახალი ბაზის შექმნა, მაშინ გამოვიყენებთ ველს: create new database. აქ ჩავწერთ ახალი ბაზის სახელს, Collation ველ-ში ვირჩევთ სტრიქონს utf8_general_ci, რაც უნიკოდის გამოყენების საშუალებას იძლევა (ნახ 7.1-ბ).



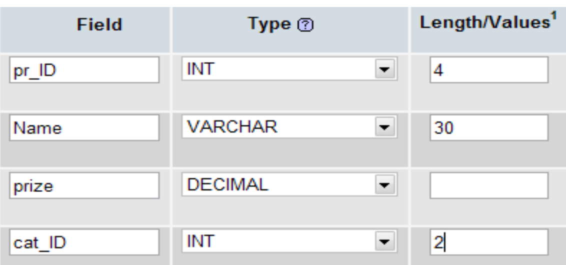

ნახ.7.1-ა




ნახ.7.1-ბ

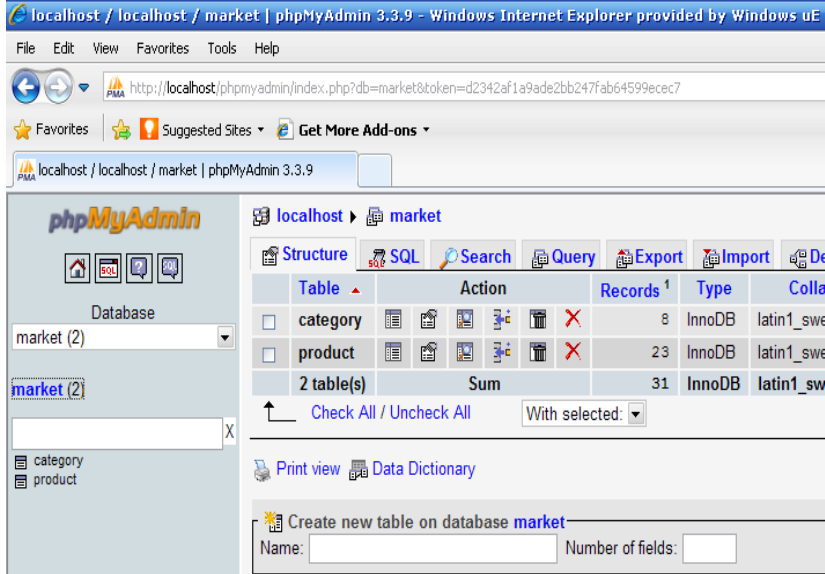
შემდეგ უნდა შევქმნათ

ბაზის ცხრილები, რისთვისაც ველში “Create new table on database”: market - ჩავწერთ ცხრილის სახელს, მაგალითად, product ან category და დავიწყებთ ცხრილის ველების (სვეტების) შექმნას (მაგალითად, ნახ.7.2-ა).



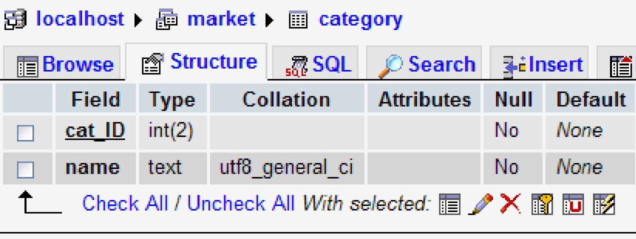

ნახ.7.2-ა

7.2-ბ ნახაზზე ნაჩვენებია ცხრილები: category და product.



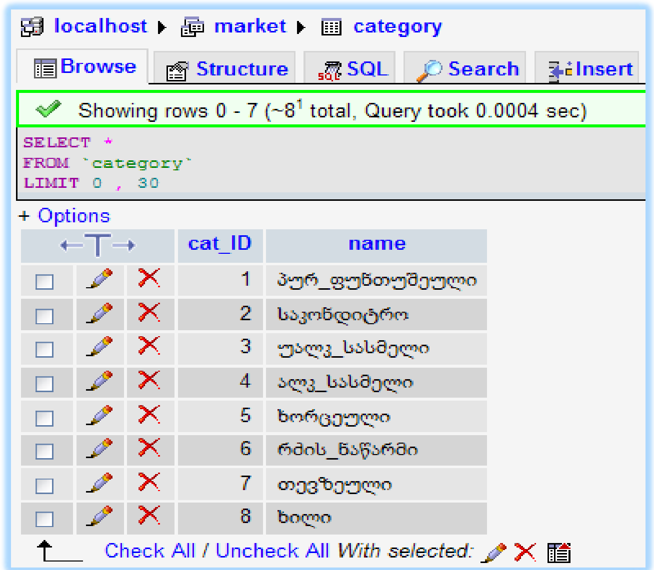

ნახ.7.2-ბ

ავირჩიოთ category ცხრილი და მენიუში Structure. 7.3-ა ნახაზზე ნაჩვენებია შედეგი.



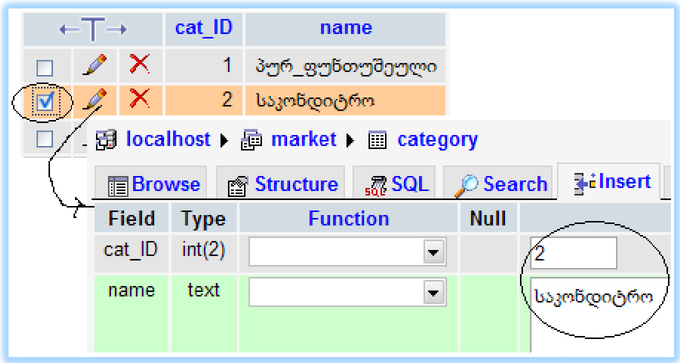

ნახ.7.3-ა

cat_ID გასაღებური ველი (პირველადი ინდექსი) აქ გახაზულია. ახლა უნდა შევავსოთ სტრიქონები კონკრეტული მნიშვნელობებით. ამისათვის ავამოქმედებთ Browse გადამრთველსა და გამოჩნდება 7.3-ბ ნახაზზე მოცემული ფანჯარა.




ნახ.7.3-ბ

„ჩეკბოქსის" ჩართვით და ფანქრის არჩევით შევალთ სტრიქონის რედაქტირების რეჟიმში და შევასრულებთ ჩვენთვის საჭირო ფუნქციას (ნახ.7.4).




ნახ.7.4

ახლა გადავიდეთ product ცხრილზე. მისი ველების განსაზღვრით და სტრიქონების შეტანით შესაძლოა ქართული უნიკოდების მაგივრად '????? ????' სტრიქონის მიღება. ასეთი ველის სტრუქტურაში Collation სვეტში უნდა ჩავსვათ utf8_general_ci (ნახ.7.5-ა,ბ).

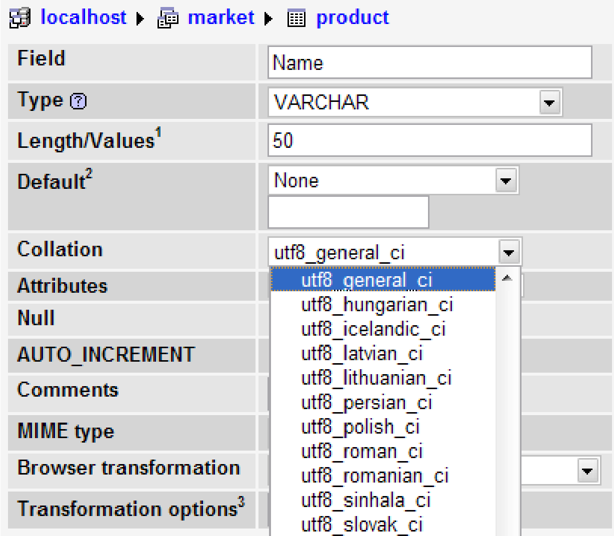
localhost ▸ market ▸ product

Browse | Structure | SQL | Search | Insert | Export | Import

| | Field | Type | Collation | Attributes | Null | Default | Extra | |
|---|---|---|---|---|---|---|---|---|
| ☐ | pr_ID | int(3) | | | No | None | | |
| ☐ | Name | varchar(50) | utf8_general_ci | | No | None | | |
| ☐ | prize | decimal(6,0) | | | No | None | | |
| ☐ | cat_ID | int(3) | | | No | None | | |

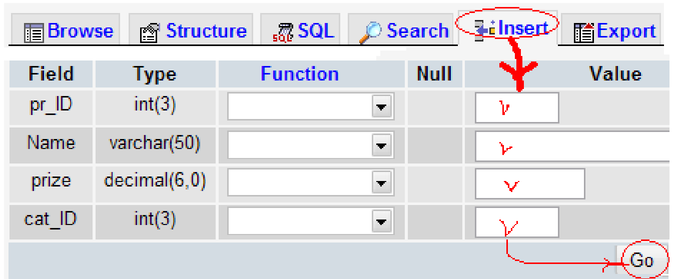
Check All / Uncheck All *With selected:*

ნახ.7.5-ა




ნახ.7.5-ბ

შევავსოთ პროდუქციის ცხრილი მნიშვნელობებით Browse და Insert გადამრთველების გამოყენებით (ნახ.7.6-ა).




ნახ.7.6-ა

შევსებული ცხრილის ფრაგმენტი მოცემულია 7.6-ბ ნახაზზე.

| | | | pr_ID | Name | prize | cat_ID |
|---|---|---|---|---|---|---|
| ☐ | ✎ | ✕ | 1 | არაჟანი | 4 | 6 |
| ☐ | ✎ | ✕ | 2 | პური | 1 | 1 |
| ☐ | ✎ | ✕ | 3 | ძეხვი | 11 | 5 |
| ☐ | ✎ | ✕ | 4 | ხაჭო | 3 | 6 |
| ☐ | ✎ | ✕ | 5 | ფუნთუშა ქიშმიშით | 1 | 1 |
| ☐ | ✎ | ✕ | 6 | კოკა_კოლა | 2 | 3 |
| ☐ | ✎ | ✕ | 7 | ფანტა | 2 | 3 |
| ☐ | ✎ | ✕ | 8 | ღვინო ქინძმარაული | 18 | 4 |
| ☐ | ✎ | ✕ | 9 | ღვინო ხვანჭკარა | 23 | 4 |
| ☐ | ✎ | ✕ | 10 | ტირამისუ | 7 | 2 |
| ☐ | ✎ | ✕ | 11 | საქონლის ხორცი | 18 | 5 |
| ☐ | ✎ | ✕ | 12 | კონიაკი "ვარციხე" | 20 | 4 |
| ☐ | ✎ | ✕ | 13 | არაყი "გომი" | 13 | 4 |
| ☐ | ✎ | ✕ | 14 | სათალი | 25 | 7 |
| ☐ | ✎ | ✕ | 15 | ვაშლი "გოლდენი" | 3 | 8 |
| ☐ | ✎ | ✕ | 16 | კონსერვი "შპროტი" | 5 | 7 |
| ☐ | ✎ | ✕ | 17 | ფორთოხალი | 3 | 8 |
| ☐ | ✎ | ✕ | 18 | მინერალური "ბორჯომი" | 2 | 3 |
| ☐ | ✎ | ✕ | 19 | მინერალური "ნაბეღლავი" | 1 | 3 |
| ☐ | ✎ | ✕ | 20 | ორცხობილა "ნუშის" | 4 | 2 |
| ☐ | ✎ | ✕ | 21 | ნაყინი "ვანილის" | 6 | 2 |
| ☐ | ✎ | ✕ | 22 | მინერალური "ლიკანი" | 2 | 3 |
| ☐ | ✎ | ✕ | 23 | ღორის სამწვადე | 23 | 5 |

Check All / Uncheck All With selected:

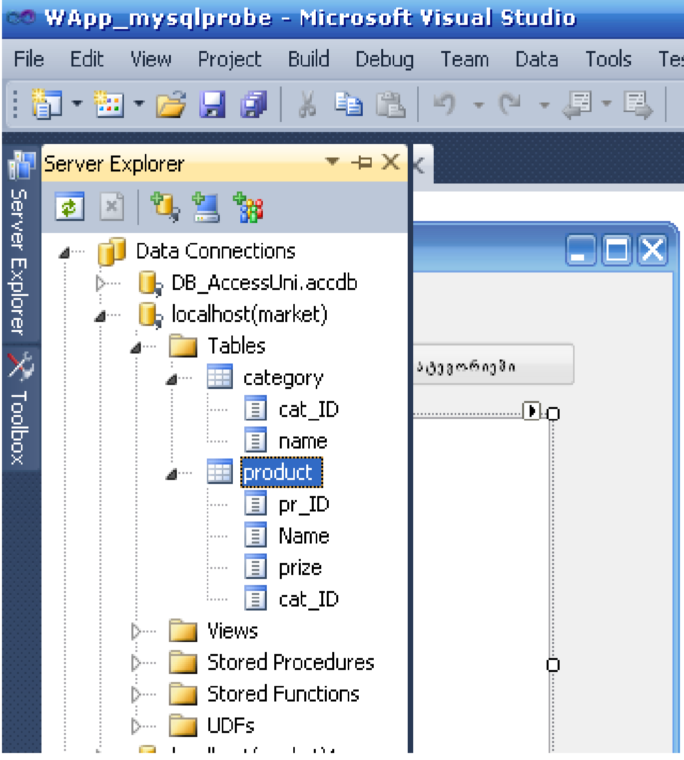
ნახ.7.6-ბ

ამგვარად, MySQL ბაზის ორი ცხრილი category და product მზადაა გამოსაყენებლად. დავხუროთ ეს ბაზა.

ახლა ავამუშავოთ Ms Visual Studio 2013 პაკეტი და C# ენის WindowsForm რეჟიმში Form1-ზე გამოვიტანოთ MySQL ბაზის მონაცემები, სხვადასხვა ჭრილში, მოთხოვნების შესაბამისად.

7.7 ნახაზზე ნაჩვენებია Server Explorer-ის ფანჯარა, სადაც localhost(market) მიერთებულ ბაზაში ჩანს Tables: category და

product თავიანთი ველებით (ატრიბუტებით). მთავარი მომენტია C# კოდიდან MySQL-ის market ბაზის მიერთება სამუშაოდ.




ნახ.7.7

პროგრამის კოდში უნდა მოთავსდეს MySQL ბაზის დასაკავშირებელი რამდენიმე სტრიქონი, რომელიც ქვემოთაა მოცემული.

```
MySql.Data.MySqlClient.MySqlConnection msqlConnection = null;

msqlConnection = new  MySql.Data.MySqlClient. MySqlConnection
     ("server=localhost;"+
      "User Id=user@localhost; database=market;
                                  Persist Security Info=True");
```

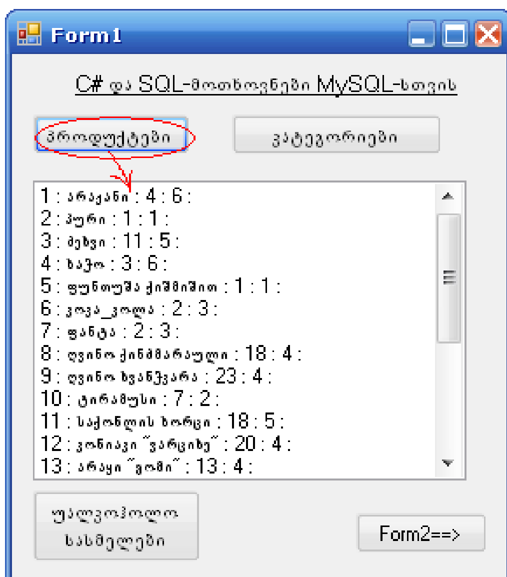
აქ განსაზღვრულია კლიენტის ბაზის მისაერთებლად აუცილებელი მონაცემები, როგორიცაა localhost-სერვერი, მომხმარებლის User-იდენტიფიკატორი და ბაზის სახელი market.

პროგრამის მთლიან კოდს შემდეგ განვიხილავთ. ახლა Form1-ფორმაზე მოთავსებული რამდენიმე ღილაკი განვიხილოთ.

ღილაკით "**პროდუქტები**" ListBox1-ში გამოიტანება ყველა პროდუქტის სია, იდენტიფიკატორით, დასახელებით, ფასით და კატეგორიის ნომრით. მისი SQL-მოთხოვნის "ამორჩევის" კოდს ექნება ასეთი სახე:

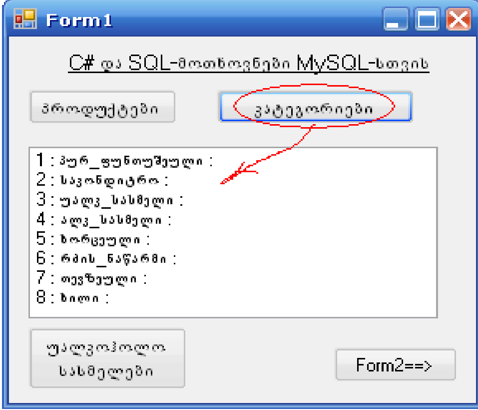
```
 private void button1_Click(object sender, EventArgs e)
  {
     Amorcheva("select * from product order by pr_ID, Name",
                                 "pr_ID", "Name", "prize", "cat_ID");
  }
```

შედეგები ასახულია 7.8 ნახაზზე მოცემულ ფანჯარაში.




ნახ.7.8

ღილაკით "**კატეგორიები**" ListBox1-ში გამოიტანება ბაზაში არსებული პროდუქციის ყველა კატეგორიის დასახელება. როგორც 7.9 ნახაზიდან ჩანს, სახეზეა 8 კატეგორია.




ნახ.7.9

ღილაკით „**უალკოჰოლო სასმელები**" ListBox1-ში გამოიტანება იმ პროდუქტების სია, რომელთა ველში „კატეგორიის იდენტიფიკატორი" შეესაბამება უალკოჰოლო სასმელებს ანუ ჩვენ შემთხვევაში `cat_ID=3`. შედეგი ასახულია 7.10 ნახაზზე.



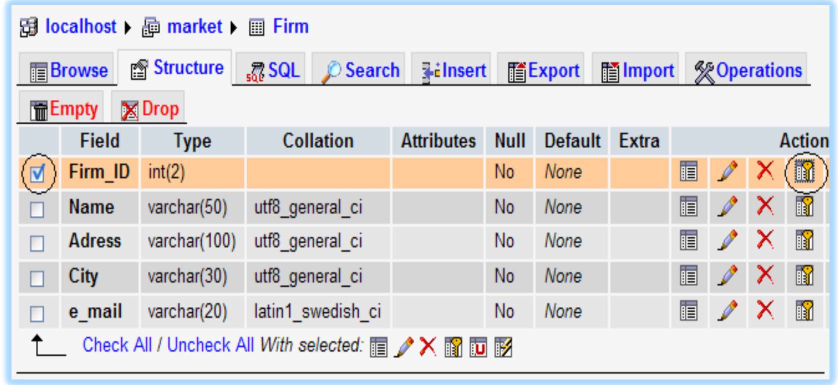
ნახ.7.10

პროგრამის კოდი მოცემულია 7.1 ლისტინგში.

```
// ლისტინგი_7.1 ---------------
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WApp_mysqlprobe
{
 public partial class Form1 : Form
 {
   public Form1() { InitializeComponent(); }
   private void Amorcheva(string sqlbefehl,
                                  params string[] velebi)
    {
     int i;
     string striqoni;
MySql.Data.MySqlClient.MySqlConnection msqlConnection = null;
     msqlConnection = new
MySql.Data.MySqlClient.MySqlConnection("server=localhost;"+
       "User Id=user@localhost;database=market;Persist
                                       Security Info=True");
MySql.Data.MySqlClient.MySqlCommand msqlCommand = new
                        MySql.Data.MySqlClient.MySqlCommand();
    msqlCommand.Connection = msqlConnection;
    msqlCommand.CommandText = sqlbefehl;
 try
    {
      msqlConnection.Open();
     MySql.Data.MySqlClient.MySqlDataReader msqlReader =
                                msqlCommand.ExecuteReader();
     listBox1.Items.Clear();
     while (msqlReader.Read())
      {
       striqoni = "";
      for (i = 0; i < velebi.Length; i++)
         striqoni += msqlReader[velebi[i]] + " : ";

      listBox1.Items.Add(striqoni);
```

```
      }
    }
  catch (Exception ex)
    { MessageBox.Show(ex.Message);
     }
  finally
    {  msqlConnection.Close();
     }
  }
 private void button1_Click(object sender, EventArgs e)
  {
    Amorcheva("select * from product order by pr_ID, Name",
                         "pr_ID", "Name", "prize", "cat_ID");
  }
 private void button2_Click(object sender, EventArgs e)
  {
    Amorcheva("select * from category order by cat_ID",
                                             "cat_ID", "Name");
  }
private void button3_Click(object sender, EventArgs e)
  {
    Amorcheva("select * from product where cat_ID=3 order by
           pr_ID", "pr_ID", "Name", "prize", "cat_ID");
} }}
```

**ამოცანა_7.7:** განხილულ ბაზას დავამატოთ ახალი ცხრილი პროდუქციის მწარმოებელი ფირმებისათვის, სახელით Firm, რომელსაც ექნება ხუთი ველი: იდენტიფიკატორი, დასახელება, მისამართი, ქალაქი და ელ-ფოსტა (ნახ.7.11-ა).


localhost ▸ market ▸ Firm

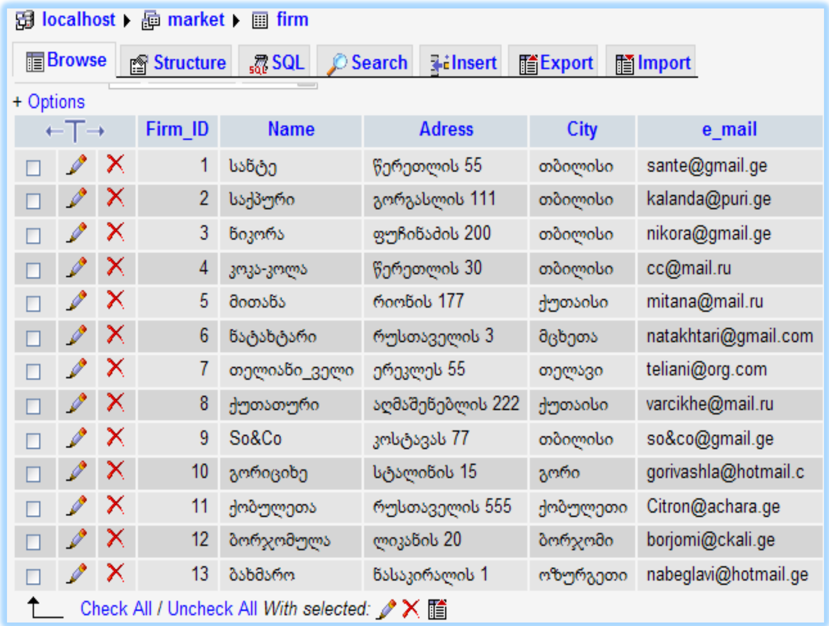
Browse | Structure | SQL | Search | Insert | Export | Import | Operations | Empty | Drop

| | Field | Type | Collation | Attributes | Null | Default | Extra | Action |
|---|---|---|---|---|---|---|---|---|
| ☑ | Firm_ID | int(2) | | | No | None | | |
| ☐ | Name | varchar(50) | utf8_general_ci | | No | None | | |
| ☐ | Adress | varchar(100) | utf8_general_ci | | No | None | | |
| ☐ | City | varchar(30) | utf8_general_ci | | No | None | | |
| ☐ | e_mail | varchar(20) | latin1_swedish_ci | | No | None | | |

Check All / Uncheck All *With selected:*

ნახ.7.11-ა

Browse გადამრთველით გამოვიტანოთ ფირმების ცხრილი და შევავსოთ იგი (ნახ.7.11-ბ).


localhost ▸ market ▸ firm

Browse | Structure | SQL | Search | Insert | Export | Import

+ Options

| Firm_ID | Name | Adress | City | e_mail |
|---|---|---|---|---|
| 1 | სანტე | წერეთლის 55 | თბილისი | sante@gmail.ge |
| 2 | საქპური | გორგასლის 111 | თბილისი | kalanda@puri.ge |
| 3 | ნიკორა | ფუჩინაძის 200 | თბილისი | nikora@gmail.ge |
| 4 | კოკა-კოლა | წერეთლის 30 | თბილისი | cc@mail.ru |
| 5 | მითანა | რიონის 177 | ქუთაისი | mitana@mail.ru |
| 6 | ნატახტარი | რუსთაველის 3 | მცხეთა | natakhtari@gmail.com |
| 7 | თელიანი_ველი | ერეკლეს 55 | თელავი | teliani@org.com |
| 8 | ქუთათური | აღმაშენებლის 222 | ქუთაისი | varcikhe@mail.ru |
| 9 | So&Co | კოსტავას 77 | თბილისი | so&co@gmail.ge |
| 10 | გორიციხე | სტალინის 15 | გორი | gorivashla@hotmail.c |
| 11 | ქობულეთა | რუსთაველის 555 | ქობულეთი | Citron@achara.ge |
| 12 | ბორჯომულა | ლიკანის 20 | ბორჯომი | borjomi@ckali.ge |
| 13 | ბახმარო | ნასაკირალის 1 | ოზურგეთი | nabeglavi@hotmail.ge |

Check All / Uncheck All *With selected:*

ნახ.7.11-ბ

product ცხრილში საჭიროა დავამატოთ ველი Firm_ID, რომლითაც ის დაუკავშირდება ამ პროდუქციის მწარმოებელი ფირმის სტრიქონს. შევიტანოთ product ცხრილის Firm_ID ველის სვეტში შესაბამისი ფირმების იდენტიფიკატორები (ნახ. 7.12).

| ←T→ | pr_ID | Name | prize | cat_ID | Firm_ID |
|---|---|---|---|---|---|
| | 1 | არაჟანი | 4 | 6 | 1 |
| | 2 | პური | 1 | 1 | 2 |
| | 3 | ძეხვი | 11 | 5 | 3 |
| | 4 | ხაჭო | 3 | 6 | 1 |
| | 5 | ფუნთუშა ქიშმიშით | 1 | 1 | 3 |
| | 6 | კოკა_კოლა | 2 | 3 | 4 |
| | 7 | ფანტა | 2 | 3 | 6 |
| | 8 | ღვინო ქინძმარაული | 18 | 4 | 7 |
| | 9 | ღვინო ხვანჭკარა | 23 | 4 | 7 |
| | 10 | ტირამუსი | 7 | 2 | 5 |
| | 11 | საქონლის ხორცი | 18 | 5 | 5 |
| ☑ | 12 | კონიაკი "ვარციხე" | 20 | 4 | 8 |
| | 13 | არაყი "გომი" | 13 | 4 | 8 |
| | 14 | სათალი | 25 | 7 | 9 |
| | 15 | ვაშლი "გოლდენი" | 3 | 8 | 10 |
| | 16 | კონსერვი "შპროტი" | 5 | 7 | 9 |
| | 17 | ფორთოხალი | 3 | 8 | 11 |
| | 18 | მინერალური "ბორჯომი" | 2 | 3 | 12 |
| | 19 | მინერალური "ნაბეღლავი" | 1 | 3 | 13 |
| | 20 | ორცხობილა "ნუშის" | 4 | 2 | 5 |
| | 21 | ნაყინი "ვანილის" | 6 | 2 | 1 |
| | 22 | მინერალური "ლიკანი" | 2 | 3 | 12 |
| | 23 | ღორის სამწვადე | 23 | 5 | 5 |

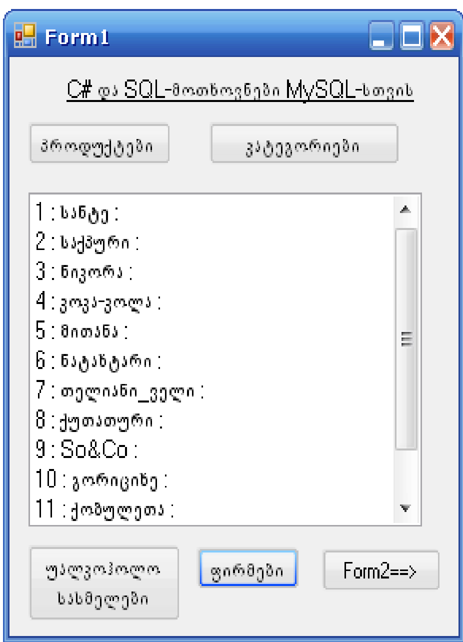
Check All / Uncheck All With selected:

ნახ.7.12

ღილაკით „**ფირმები**" ListBox1-ში გამოვიტანოთ პროდუქციის მწარმოებელი ფირმების სია. C#კოდის ფრაგმენტს ექნება შემდეგი სახე:

```
private void button5_Click(object sender, EventArgs e)
   {
     Amorcheva("select * from Firm order by Firm_ID",
                                 "Firm_ID", "Name");
   }
```

შედეგები მოცემულია 7.13 ნახაზზე.




ნახ.7.13

## 2.8. C# აპლიკაციის მუშაობა Ms SQL Server ბაზასთან

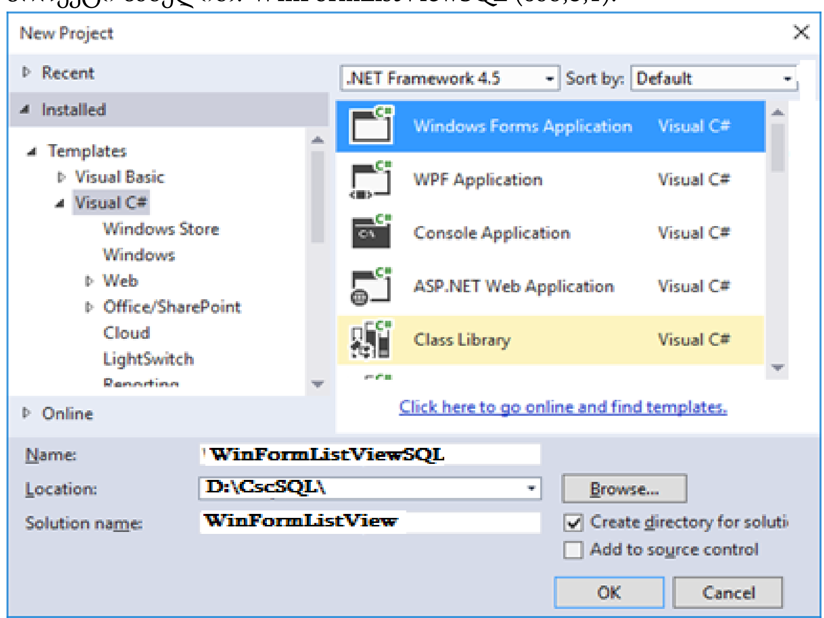
### 2.8.1. მონაცემთა განახლების Input, Update, Delete მეთოდების დაპროგრამება ListView კლასის გამოყენებით

### ლაბორატორიული სამუშაო N 8

**მიზანი:** ვიზუალური დაპროგრამების C# ენის ListView კლასის საფუძველზე მონაცემთა მენეჯმენტის განხილვა, SQL Server ბაზის განახლების პროცედურების შესწავლა Insert, Update და Delete მეთოდებით.

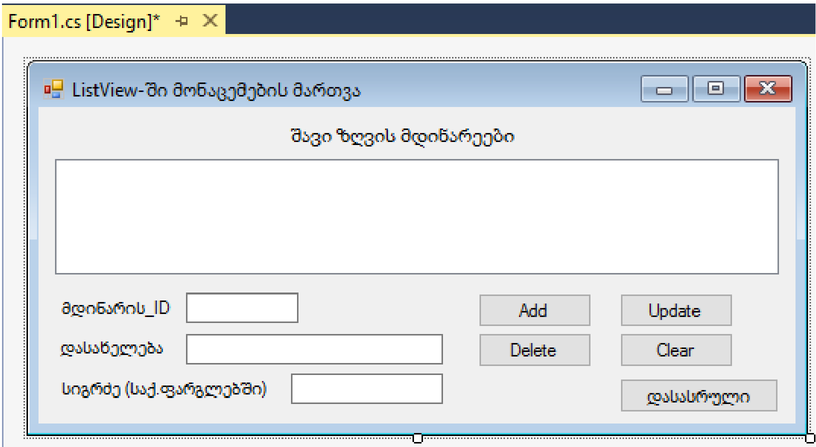
პროგრამული პროექტის აგების მაგალითისათვის განვიხილოთ ამოცანა საქართველოს აკვატორიაში შავი ზღვის მდინარეების მონაცემთა ბაზის შექმნისა და მისი ცხრილების შევსებისა და განახლების ინტერფეისული პროგრამის დამუშავება.

Visual Studio.NET 2013/15 გარემოში შევქმნათ ახალი პროექტი სახელით: WinFormListViewSQL (ნახ,8,1).




ნახ.8.1

ჩვენ მიერ დაპროექტებული ინტერფეისის ფორმა მოცემულია 8.2 ნახაზზე.




ნახ.8.2

მონაცემთა ბაზის ჩანაწერებიგამოიტანება ListView სვეტებში (ნახ.8.3).

ListView-ში მონაცემების მართვა

შავი ზღვის მდინარეები

| მდინარის_ID | დასახელება | სიგრძე (საქართველოს ფარგლებში-km) |
|---|---|---|
| 1 | ჭოროხი | 26 |
| 2 | კინტრიში | 25 |
| 3 | ნატანები | 60 |
| 4 | სუფსა | 108 |
| 5 | რიონი | 327 |
| 6 | ხუბუსწყალი | 15 |
| 7 | ინგური | 213 |
| 8 | მტკვარი | 400 |

მდინარის_ID  Add  Update

დასახელება  Delete  Clear

სიგრძე (საქ.ფარგლებში)  დასასრული

ნახ.8.3

მე-6 სტრიქონში შეცდომაა და საჭიროა ცვლილების განხორციელება Update მეთოდით. ვირჩევთ სტრიქონს (ნახ.8.4).

ListView-ში მონაცემების მართვა

შავი ზღვის მდინარეები

| მდინარის_ID | დასახელება | სიგრძე (საქართველოს ფარგლებში-km) |
|---|---|---|
| 1 | ჭოროხი | 26 |
| 2 | კინტრიში | 25 |
| 3 | ნატანები | 60 |
| 4 | სუფსა | 108 |
| 5 | რიონი | 327 |
| 6 | ხუბუსწყალი | 15 |
| 7 | ინგური | 213 |
| 8 | მტკვარი | 400 |

მდინარის_ID: 6
დასახელება: ხუბუსწყალი
სიგრძე (საქ.ფარგლებში): 15

Add | Update | Delete | Clear | დასასრული

ნახ.8.4

„ხუბუსწყალი“ შევცვალოთ „ხობისწყალი“-თ (ნახ.8.5).

შავი ზღვის მდინარეები

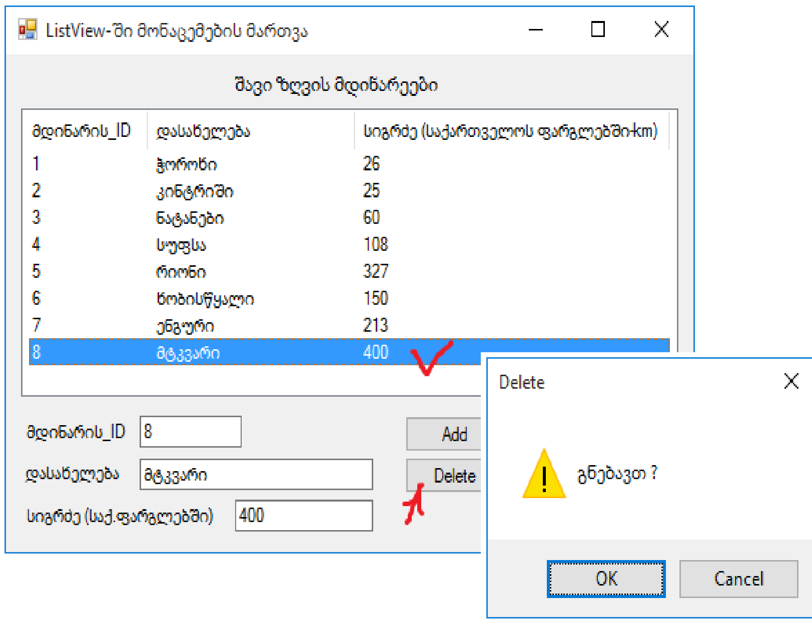
| მდინარის_ID | დასახელება | სიგრძე (საქართველოს ფარგლებში-km) |
|---|---|---|
| 1 | ჭოროხი | 26 |
| 2 | კინტრიში | 25 |
| 3 | ნატანები | 60 |
| 4 | სუფსა | 108 |
| 5 | რიონი | 327 |
| 6 | ხობისწყალი | 150 |
| 7 | ინგური | 213 |
| 8 | მტკვარი | 400 |

მდინარის_ID

Add | Update

ნახ.8.5

„მტკვარი“ არაა შავი ზღვის მდინარე, ამიტომ ის უნდა წაიშალოს Delete მეთოდით (ნახ.8.6).




ნახ.8.6

8.1_ლისტინგში მოცემულია ListView-ში მონაცემების დამატების, ცვლილებისა და წაშლის მეთოდების კოდები.

// -- ლისტინგი_8.1 ---- ListView-ის Add, Update Delete მეთოდები --

```
using System;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
```

```
namespace WinFormListViewSeqD
{
  public partial class Form1 : Form
   {
     public Form1()
      {
        InitializeComponent();
        // ListView-ის თვისებები -----
        listView1.View = View.Details;
        listView1.FullRowSelect = true;
        // ListView-ის ველების სიგანე-----
        listView1.Columns.Add("მდინარის_ID", 50);
        listView1.Columns.Add("დასახელება", 150);
        listView1.Columns.Add("სიგრძე (საქართველოს
                  ფარგლებში-კმ)",50);
     }
 // Add
private void add(String id, String name, String length)
 {
   //row---
   String[] row = {id, name, length};
   ListViewItem item = new ListViewItem(row);
   listView1.Items.Add(item);
 }
 // update-----
private void update()
 {
   listView1.SelectedItems[0].SubItems[0].Text =
         idTxt.Text;
   listView1.SelectedItems[0].SubItems[1].Text =
         nameTxt.Text;
   listView1.SelectedItems[0].SubItems[2].Text =
         lengthTxt.Text;
   // clear txt
    idTxt.Text = "";
    nameTxt.Text = "";
    lengthTxt.Text = "";
  }
```

```
  // delete -----
  private void delete()
   {
    if (MessageBox.Show("გნებავთ წაშლა?", "DELETE",
                  MessageBoxButtons.OKCancel,
           MessageBoxIcon.Warning) == DialogResult.OK)
      {
listView1.Items.RemoveAt(listView1.SelectedIndices[0]);
       // clear txt
        idTxt.Text = "";
        nameTxt.Text = "";
        lengthTxt.Text = "";
      }
   }
private void button1_Click(object sender, EventArgs e)
  {
    add(idTxt.Text, nameTxt.Text, lengthTxt.Text);
    // clear txt
      idTxt.Text = "";
      nameTxt.Text = "";
      lengthTxt.Text = "";
  }
private void button2_Click(object sender, EventArgs e)
  {
      update();
  }

private void button3_Click(object sender, EventArgs e)
  {
      delete();
  }

private void button4_Click(object sender, EventArgs e)
  {
    listView1.Items.Clear();
    // clear txt
    idTxt.Text = "";
    nameTxt.Text = "";
```

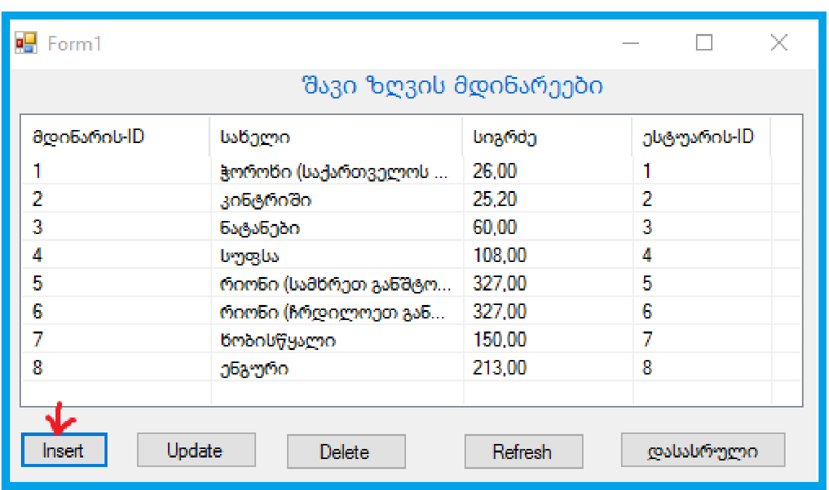
```
    lengthTxt.Text = "";
   }
 private void listView1_MouseClick(object sender,
           MouseEventArgs e)
   {
idTxt.Text=listView1.SelectedItems[0].SubItems[0].Text;
nameTxt.Text=listView1.SelectedItems[0].SubItems[1]
       .Text;
lengthTxt.Text=listView1.SelectedItems[0].SubItems[2]
       .Text;
   }
private void button5_Click(object sender, EventArgs e)
   {
       Close();
   }
  }
}
```

## 2.8.2. SQL Server ბაზის ცხრილების პროგრამული განახლების რეალიზაცია ListView კლასის საფუძველზე

### ლაბორატორიული სამუშაო N 9

**მიზანი:** ListView-ში მონაცემების გამოტანა SQL Server-დან და მასში ცვლილებების განხორციელება.

9.1 ნახაზზე მოცემულია ListView საწყისი ფანჯარა ბაზიდან ამოღებული სტრიქონებით:




ნახ.9.1

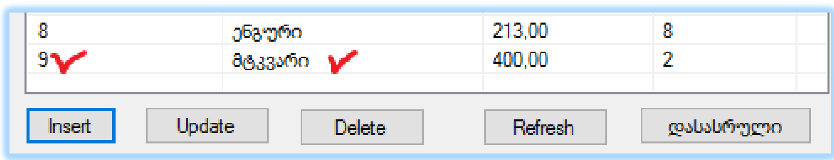
Insert ღილაკის არჩევით გამოიტანება 9.2 ფანჯარა და ვამატებთ ახალ სტრიქონს :

Form2
შავი ზღვის მდინარეები
ID 9
სახელი მტკვარი
სიგრძე (კმ) 400
ესტუარის-ID 2
(საქართველოს საზღვრებში)
ჩამატება ბაზაში

ნახ.9.2

მდინარე

მტკვარი ჩაემატა SQL Server ბაზაში და განახლდა ListView სია თავიდან ახალი ვერსიით (ნახ.9.3).




ნახ.9.3

- Insert მეთოდის შესაბამისი კოდი მოცემულია 9.1 ლისტინგში

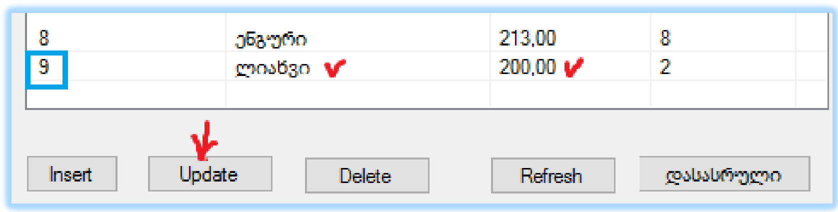
```
public partial class Form1 : Form
{  // SQL Server ბაზასთან მიერთება----
  SqlConnection con = new SqlConnection(@"Data Source=gtu-
205A-08;Initial Catalog=SeaEco;Integrated Security=True");
...
//. . . ლისტინგი_9.1 ---- Insert() ------
private void button1_Click(object sender, EventArgs e)
      {
          Form2 frm = new Form2(nID+1, null);
          //დამატება მოვლენის დასაჭერად Form2-დან---
          frm.UpdateListView += new
               EventHandler(frm_UpdateListView);
          frm.Show();
      }
  void frm_UpdateListView(object sender, EventArgs e)
        {
         //როგორც კი მონაცემები დაემატება Form2-ში,
         // მოვლენა განაახლებს ListView1-ს----
           ListViewLoad();
        }
```

- Delete მეთოდის შესაბამისი კოდი მოცემულია 9.2 ლისტინგში

```
   //.. ლისტინგი_9.1 ---- Delete
   private void button4_Click(object sender, EventArgs e)
        {
            SqlDataAdapter dass = new SqlDataAdapter();
```

```
            con.Open();
            DataTable dt = new DataTable();
            string deleete_r =
listView1.SelectedItems[0].SubItems[0].Text;
          try
            {
               dass.DeleteCommand = new
            SqlCommand(@"DELETE FROM River WHERE riverID =
                      @riverID", con);

        dass.DeleteCommand.Parameters.Add("@riverID",
            SqlDbType.Int).Value = int.Parse(deleete_r);
        dass.DeleteCommand.ExecuteNonQuery();
        con.Close();
            }
          catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
            ListViewLoad();
        }
```

9.4 ნახაზზე მოცემულია Update მეთოდის შედეგი, ანუ დავდექით მე-9 ჩანაწერზე და ავამოქმედეთ Update ღილაკი. ამ მოვლენამ აამუშავა Update() მეთოდი. შევიტანეთ მდინარე „ლიახვი 200 კმ“. კოდი ნაჩვენებია 9.1 ლისტინგში.




ნახ.9.4

```
// ლისტინგი_9.3 - -- Update() ъуЕщвш ---
private void button3_Click(object sender, EventArgs e)
{
            Form2 frm = new Form2(nID, listView1);
```

```
            frm.UpdateListView += new
EventHandler(frm_UpdateListView);
            frm.Show();
        }
```

შეცვლილი ბაზის ხელახალი ჩატვირთვა ListView ფანჯარაში ხორციელდება შემდეგი კოდით (ლისტინგი 9.4).

```
// -- ლისტინგი_9.4 ---- ---
void ListViewLoad()
 {
   SqlDataAdapter da = new SqlDataAdapter("SELECT riverID,
     riverName, river_Length, estuarID FROM River", con);
   DataTable dt = new DataTable();
   // მონაცემების გადატანა ბაზიდან ცხრილში ---
    da.Fill(dt);
   // ListView-ს შევსება ცხრილიდან---
    listView1.Items.Clear();
    listView1.View = View.Details;
    listView1.GridLines = true;

    foreach (DataRow dr in dt.Rows)
    {
      ListViewItem lvi = new
            ListViewItem(dr["riverID"].ToString());
     lvi.SubItems.Add(dr["riverName"].ToString());
     lvi.SubItems.Add(dr["river_Length"].ToString());
     lvi.SubItems.Add(dr["estuarID"].ToString());
     listView1.Items.Add(lvi);
     // გლობალური ცვლადი (int), გამოცხადებულია დასაწისში---
     nID =int.Parse( dr["riverID"].ToString());
    }
}
```

### 2.8.3. SQL Server ბაზის განახლება ADO.NET დრაივერისა და DataGridView კლასის გამოყენებით

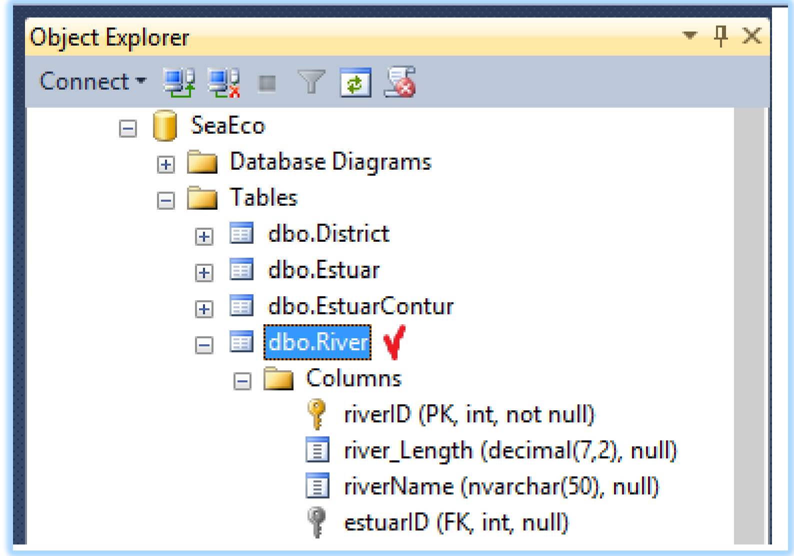
**ლაბორატორიული სამუშაო N 10**

**მიზანი:** ვიზუალური დაპროგრამების C# ენის, Ms SQL Server-მონაცემთა ბაზების პაკეტის და ADO.NET-დრაივერის ერთობლივი გამოყენებით პროგრამული აპლიკაციების აგების შესწავლა.

თეორიული ნაწილი ვრცლად იქნა განხილული 1-ელი თავის მე-13 პარაგრაფში. აქ წარმოდგენილი იყო ADO.NET-ის ობიექტები, რომელთა საშუალებითაც ხორციელდება კავშირი C# ენაზე დაწერილ ინტერფეისსა და მონაცემთა ბაზას შორის.

**ამოცანა 10.1.** მოცემულია Ms SQL Server მონაცემთა ბაზა (მაგალითად, შავი ზღვის ეკოლოგიური სისტემა), რომლის ერთ-ერთი ცხრილი (Table) არის River.dbo (მდინარეები). საჭიროა ავაგოთ C# პროექტი (მომხმარებლის ინტერფეისი), რომელიც მონაცემთა ბაზიდან ამოიღებს მდინარეების მონაცემებს DataGridView ცხრილში, შეძლებს Insert, Update და Delete ოპერაციების განხორციელებას. გამოყენებულ უნდა იქნას ADO.NET დრაივერის საშუალებები.

10.1 ნახაზზე ნაჩვენებია საწყისი მონაცემთა ბაზა, რომელიც Ms SQL Server 2012 ვერსიაშია რეალიზებული. (ა) შეესაბამება ბაზის ცხრილების იერარქიასა და აქვე ჩანს River ცხრილის სტრუქტურაც, მონაცემთა შესაბამისი ტიპებით. (ბ) -ზე მოცემულია ჩანაწერები, რომელთა შეტანა მოხდა წინასწარ (თუმცა ჩვენი პროგრამისთვის არაა აუცილებელი მისი არსებობა. შესაძლებელია იგი შეივსოს ინტერფეისიდან).



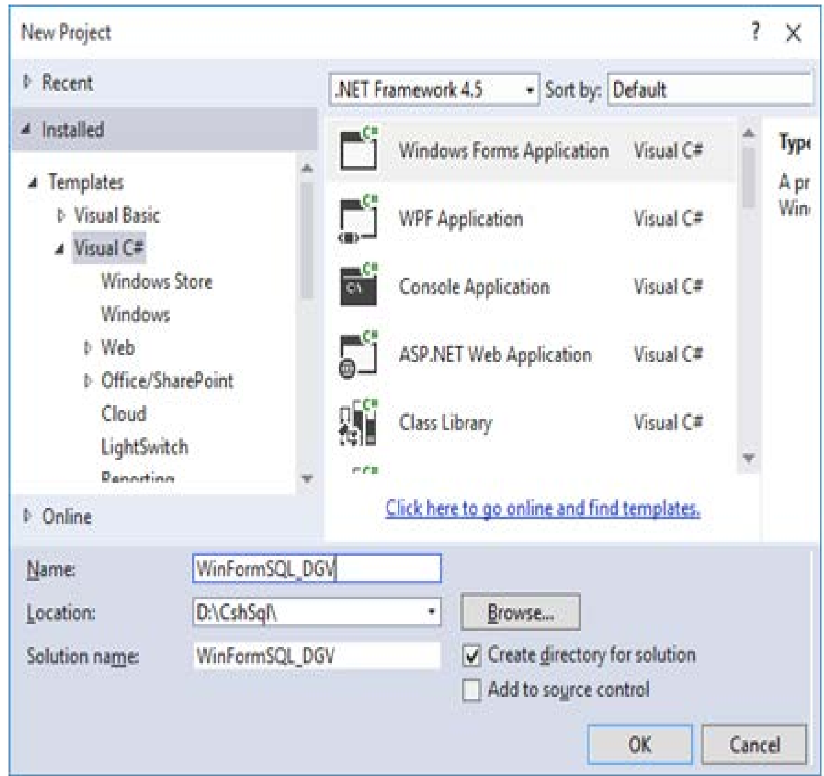

ნახ.10.1-ა. მონაცემთა ბაზა Ms SQL Server-ში

GTU-205A-08.SeaEco - dbo.River | GTU-205A-08.SeaEco - dbo.River

| | riverID | river_Length | riverName | estuarID |
|---|---|---|---|---|
| ▶ | 1 | 26,00 | ჭოროხი (საქართველოს საზღვრებში) | 1 |
| | 2 | 25,20 | კინტრიში | 2 |
| | 3 | 60,00 | ნატანები | 3 |
| | 4 | 108,00 | სუფსა | 4 |
| | 5 | 327,00 | რიონი (სამხრეთ განშტოება) | 5 |
| | 6 | 327,00 | რიონი (ჩრდილოეთ განშტოება) | 6 |
| | 7 | 150,00 | ხობისწყალი | 7 |
| | 8 | 213,00 | ენგური | 8 |
| * | *NULL* | *NULL* | *NULL* | *NULL* |

ნახ.10.1-ბ. River (მდინარეების) საწყისი ცხრილი
Ms SQL Server-ში

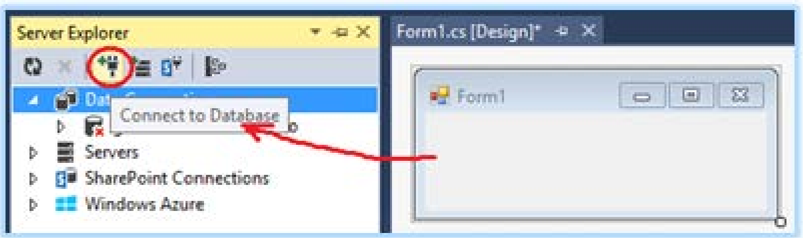
დავიწყოთ ახალი პროექტის აგება Ms Visual Studio.NET 2013 სამუშაო გარემოში. 10.2 ნახაზზე ნაჩვენებია პროექტის შექმნის პროცედურა.




ნახ.10.2. WinFormSQL_DGV პროექტის შექმნა

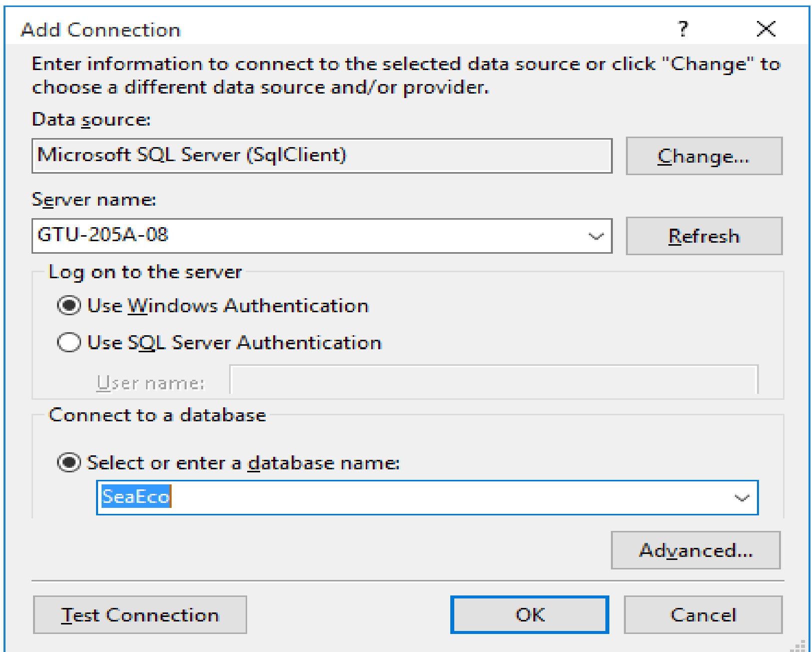
ვირჩევთ პროექტის სახელს (Name), მისი შენახვის ადგილს (Browse-ს დახმარებით) და Solutionname-ს. შემდეგ OK.

საჭიროა პროექტისთვის განვახორციელოთ მონაცემა ბაზის მიერთება (Connect to Database), რაც 10.3 ნახაზზეა ნაჩვენები.



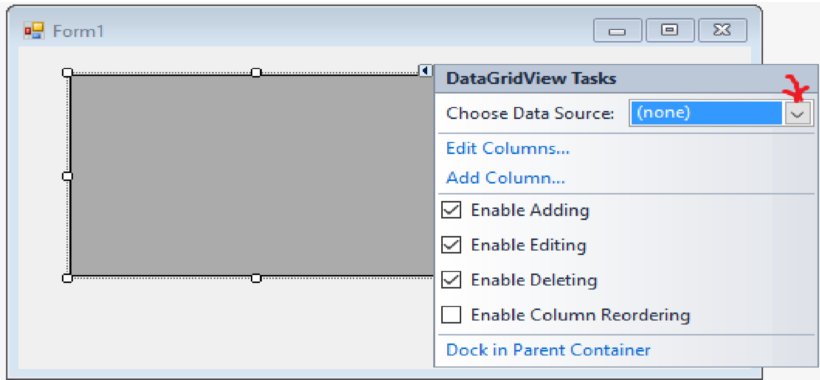

ნახ.10.3. Connect Database ამოქმედება

გამოიტანება ახალი ფანჯარა (ნახ.10.4), რომელშიც უნდა შეირჩეს შესაბამისი წყარო (Data Source), სერვერი (Server name) და მონაცემთა ბაზა (Database name).



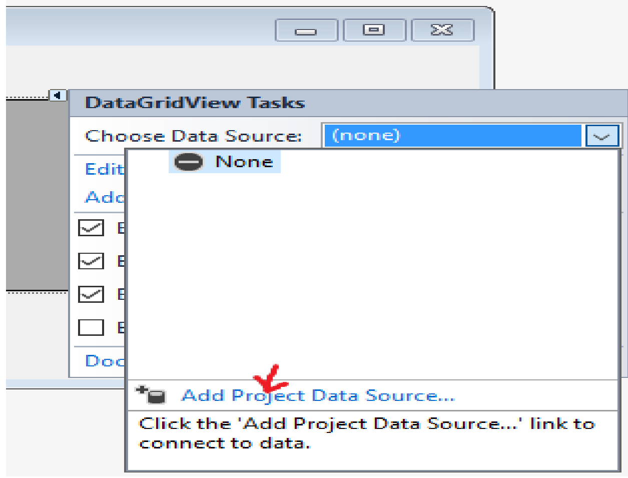

ნახ.10.4. მონაცემთა წყაროს, სერვერისა და ბაზის არჩევა

ინსტრუმენტების პანელიდან ფორმაზე გადავიტანოთ DataGridView ელემენტი და ზედა მარჯვენა კუთხე პატარა ისრით ავამოქმედოთ. მივიღებთ 10.5 ნახაზს.



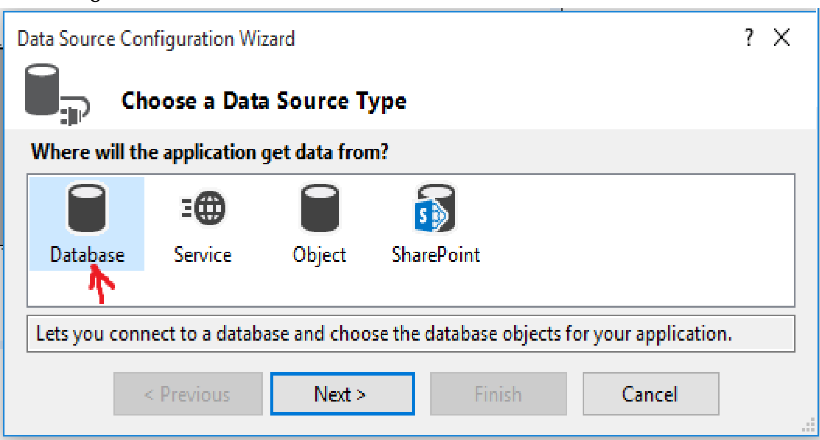

ნახ.10.5. პარამეტრების განსაზღვრა

ნახაზზე ჩანს, რომ ჩამატების, რედაქტირებისა და წაშლის ოპერაციები ნებადართულია (ჩეკბოქსები მონიშნულია). ავირჩიოთ Choose Data Source კომბობოქსის ღილაკი, მივიღებთ 10.6 ნახაზს.



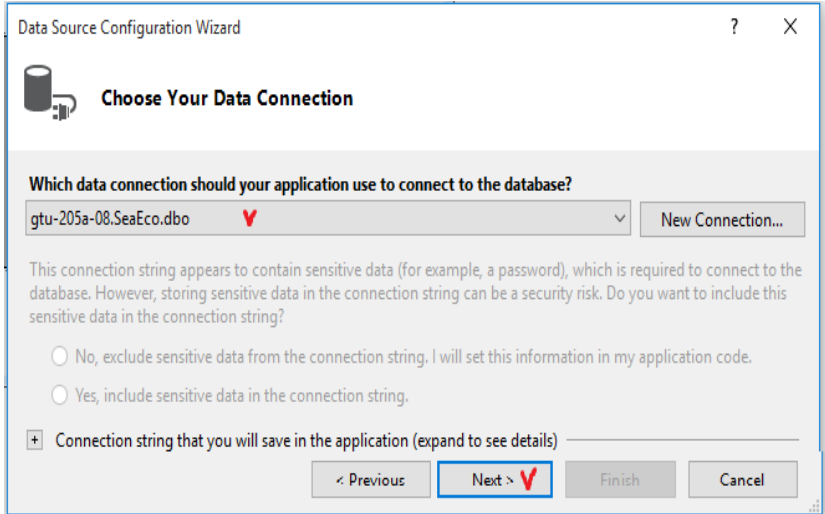

ნახ.10.6. მონაცემთა წყაროს პროექტის დამატება

ავამოქმედოთ Add Project Data Source და გადავიდეთ 10.7 ნახაზზე.



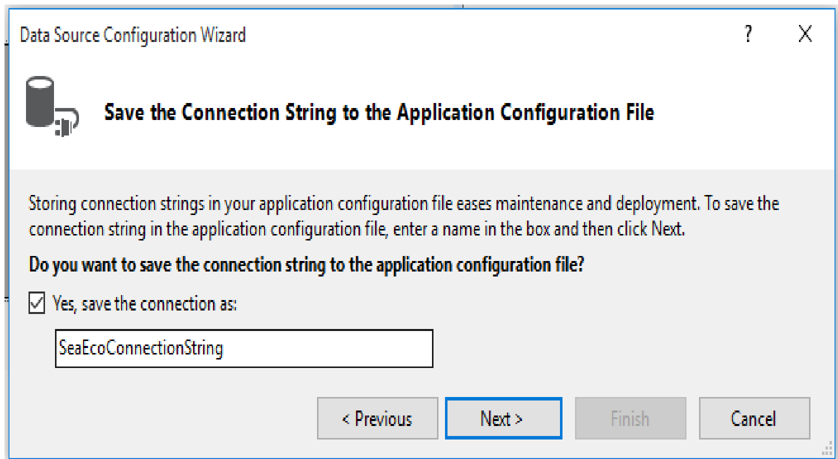

ნახ.10.7. მონაცემთა წყაროს ტიპის არჩევა

ვირჩევთ Database-სა და Next (ნახ.10.8).




ნახ.10.8. მონაცემთა Connection (მიერთების) შერჩევა

Connection პარამეტრი ყველა კომპიუტერს ექნება თავისი. მისი განსაზღვრა შესაძლებელია Server Explorer-იდან (ჩვენ შემთხვევაში იგი არის: GTU-205a-08.SeaEco.dbo). ბოლოს Next და გადავალთ 10.9 ნახაზზე.



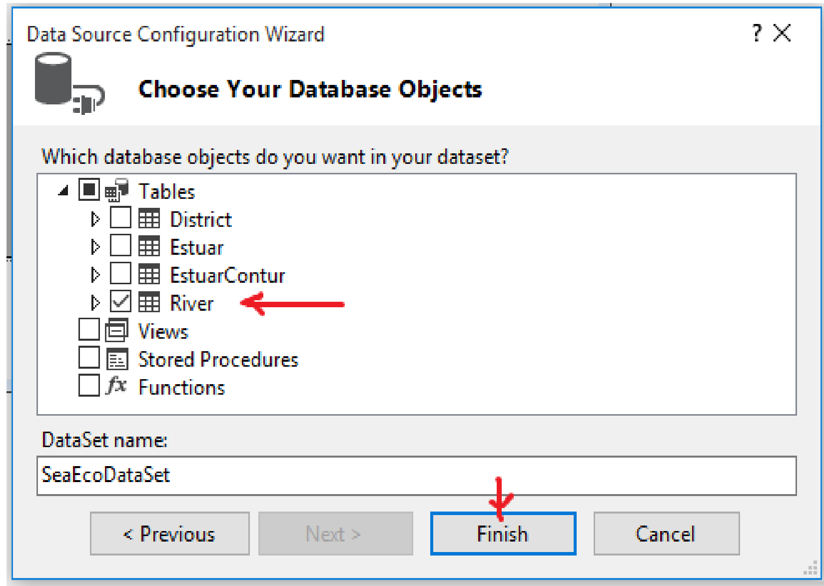

**ნახ.10.9. Connection String-ის შენახვა აპლიკაციის კონფიგურაციის ფაილში**

შემდეგ გამოიძახება მონაცემთა ბაზის ობიექტების არჩევის ფანჯარა (ნახ.10.10).

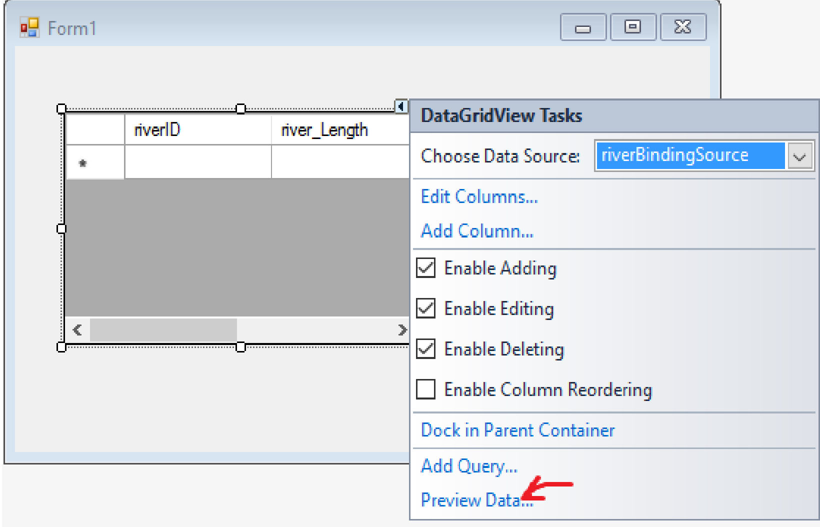
10.11 ნახაზზე ნაჩვენებია მონაცემთა წყაროს (Data Source) განსაზღვრის შედეგის მნიშვნელობა, ჩვენ შემთხვევაში იგი არის:

**riverBindingSource.**

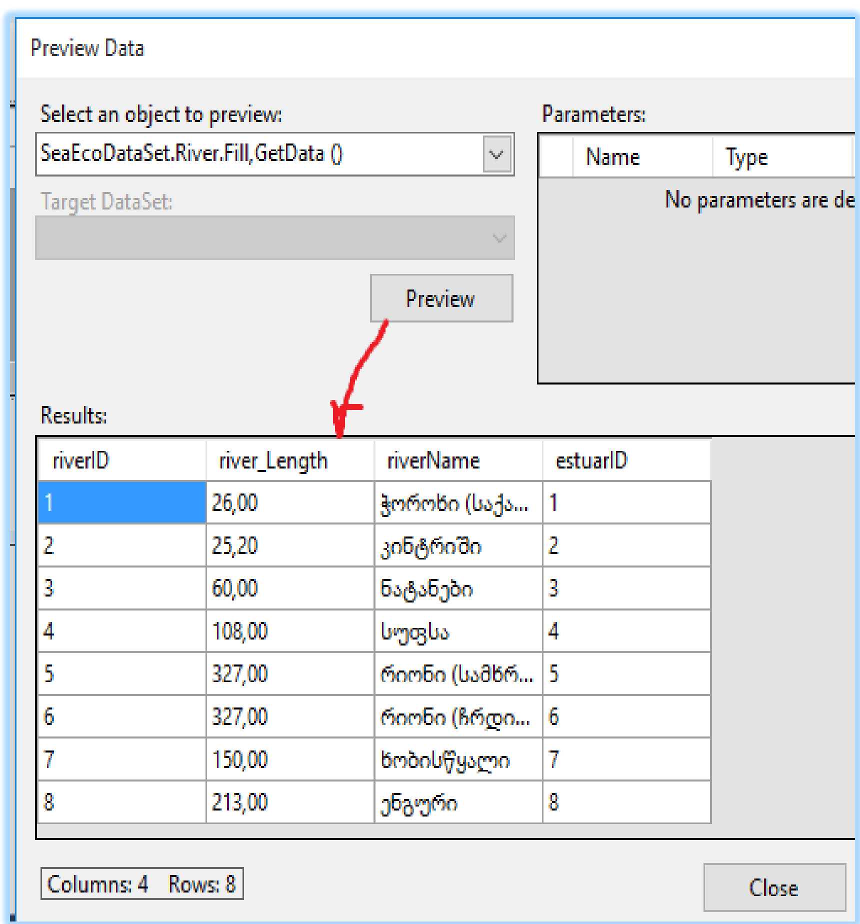
აქვე შეიძლება გამოვიყენოთ Preview Data ლინკი და დავათვალიეროთ წინასწარ მიერთებული ბაზის ცხრილის ჩანაწერები (ნახ.10.12).




ნახ.10.10. ობიექტების მონიშვნა (მაგალითად, River)

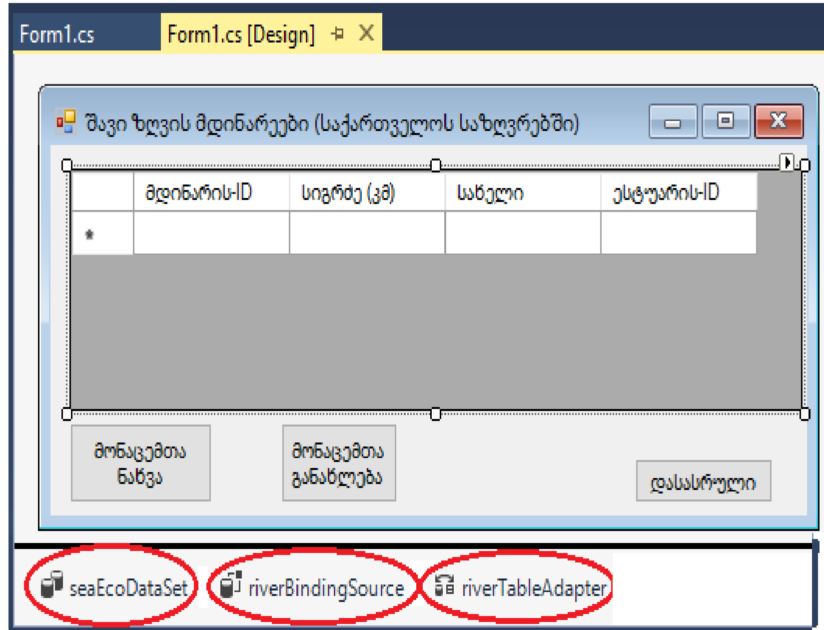



ნახ.10.11. Data Source შედეგი: "riverBindingSource"




ნახ.10.12. **Preview Data ცხრილი**

ბოლოს, Form1-ს Properties-ში შევუცვალოთ სახელი „შავი ზღვის მდინარეები (საქართველოს საზღვრებში)“, ინსტრუმენტების პანელიდან გადმოვიტანოთ სამი ღილაკი (Button1,2.3), დავარქვათ სახელები. მიიღება 10.13 ნახაზი.




ნახ.10.13. პროექტის ძირითადი ინტერფეისი

ახლა გადავიდეთ ღილაკების ფუნქციების დაპროგრამებაზე, ანუ უნდა განხორციელდეს SQL Server -შესაბამისი ბაზის ცხრილის ჩანაწერების დათვალიერება, ჩამატება, შეცვლა და წაშლა.

თავდაპირველად ჩავამატოთ სახელსივრცის სტრიქონი:

```
using System.Data.SqlClient;
```

შემდეეგ გამოვაცხადოთ გლობალური ცვლადები:

```
SqlDataAdapter sda;
SqlCommandBuilder scb;
DataTable dt;
```

„მონაცემთა ნახვის“ ღილაკისათვის (`button1` ) გვექნება შემდეგი კოდი:

```
private void button1_Click(object sender, EventArgs e)
 {
   SqlConnection con = new SqlConnection("Data Source=
                 GTU-205A-08;Initial Catalog=SeaEco;
                 Integrated Security=True");
   sda = new SqlDataAdapter(@"SELECT riverID,
                     river_Length, riverName,
                     estuarID from River", con);
    dt = new DataTable();
   sda.Fill(dt);
   dataGridView1.DataSource = dt;
 }
```

პროგრამის ამუშავებით, თუ მასში არაა შეცდომები, მიიღება 10.14 ნახაზი.

შავი ზღვის მდინარეები (საქართველოს საზღვრებში)

| | მდინარის-ID | სიგრძე (კმ) | სახელი | ესტუარის-ID |
|---|---|---|---|---|
| ▶ | 1 | 26,00 | ჭოროხი (საქარ... | 1 |
| | 2 | 25,20 | კინტრიში | 2 |
| | 3 | 60,00 | ნატანები | 3 |
| | 4 | 108,00 | სუფსა | 4 |
| | 5 | 327,00 | რიონი (სამხრე... | 5 |
| | 6 | 327,00 | რიონი (ჩრდი... | 6 |
| | 7 | 150,00 | ხობისწყალი | 7 |
| | 8 | 213,00 | ენგური | 8 |
| * | | | | |

მონაცემთა ნახვა | მონაცემთა განახლება | დასასრული

ნახ.10.14. „მონაცემთა ნახვის“ (DataShow) ღილაკის ამოქმედებით მიღებული შედეგი

„მონაცემთა განახლების“ ღილაკის კოდი (Insert, Update, Delete) ნაჩვენებია ქვემოთ:

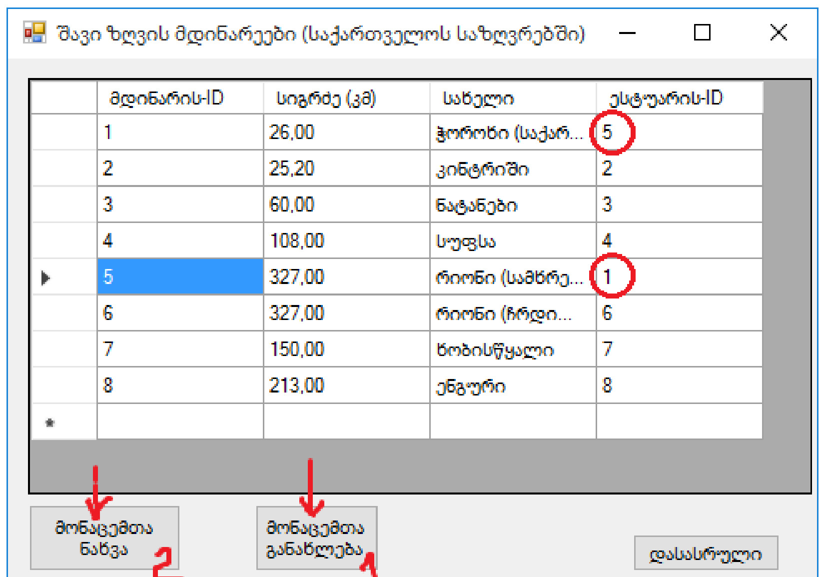
```
private void button2_Click(object sender, EventArgs e)
  {
    scb = new SqlCommandBuilder(sda);
    sda.Update(dt);
  }
```

მთლიანი პროგრამის კოდი მოცემულია 10.1 ლისტინგში.

```
// --- ლისტინგი_10.1 --- Insert, Update, Delete for DataGridView_SQL----
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Data.SqlClient;

namespace WinFormSQL_DGV
{
  public partial class Form1 : Form
   {
      SqlDataAdapter sda;
      SqlCommandBuilder scb;
      DataTable dt;
      public Form1()
       {
          InitializeComponent();
       }

    private void Form1_Load(object sender, EventArgs e)
     {

     }
      // მონაცემთა ნახვა
    private void button1_Click(object sender, EventArgs e)
```

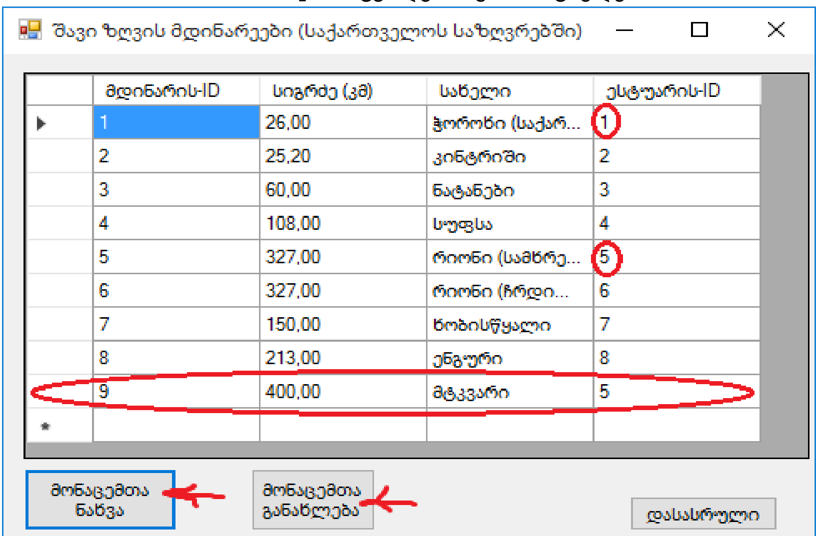
```
    {
      SqlConnection con = new SqlConnection("Data
           Source=GTU-205A-08;Initial Catalog=SeaEco;
                Integrated Security=True");
      sda = new SqlDataAdapter(@"SELECT riverID,
                   river_Length, riverName, estuarID
                   from River", con);
      dt = new DataTable();
      sda.Fill(dt);
      dataGridView1.DataSource = dt;
    }

   // განახლება
   private void button2_Click(object sender, EventArgs e)
      {
         scb = new SqlCommandBuilder(sda);
         sda.Update(dt);
      }

    private void button3_Click(object sender, EventArgs e)
        {
            Close();
        }
    }
}
```

10.15 ნახაზზე მოცემულია არსებულ ცხრილში ორი მნიშვნელობის (ესტუარისID) შეცვლა (ახალი რიცხვები ნაჩვენებია წრეში), შემდეგ 1-ელი და მე-2 ღილაკების ამოქმედებით ბაზაში ჩაიწერება ეს შეცვლილი მნიშვნელობები (შეიძლება დავხუროთ პროგრამა და თავიდან ავამუშავოთ).




| მდინარის-ID | სიგრძე (კმ) | სახელი | ესტუარის-ID |
|---|---|---|---|
| 1 | 26,00 | ჭოროხი (საქარ... | 5 |
| 2 | 25,20 | კინტრიში | 2 |
| 3 | 60,00 | ნატანები | 3 |
| 4 | 108,00 | სუფსა | 4 |
| 5 | 327,00 | რიონი (სამხრე... | 1 |
| 6 | 327,00 | რიონი (ჩრდი... | 6 |
| 7 | 150,00 | ხობისწყალი | 7 |
| 8 | 213,00 | ენგური | 8 |

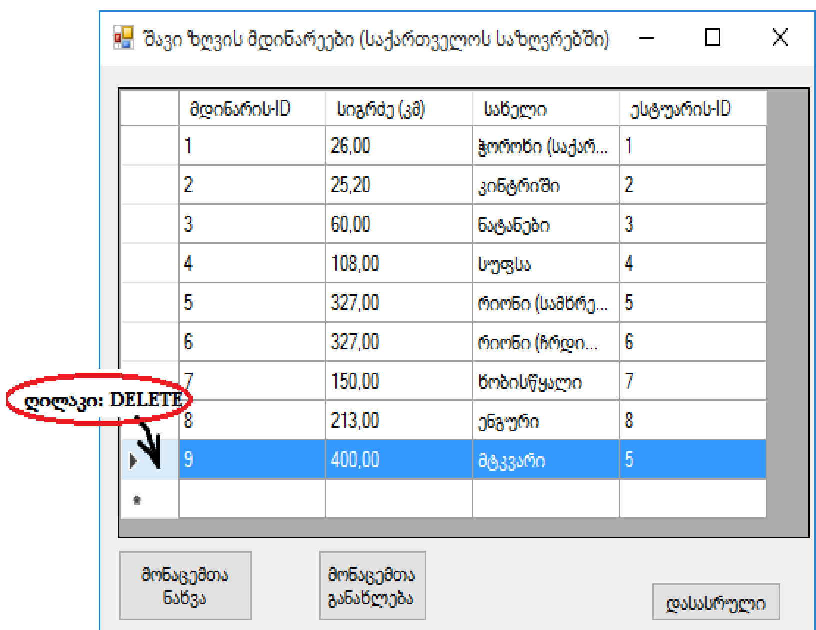
ნახ.10.15. Update ცვილების განხორციელება




| მდინარის-ID | სიგრძე (კმ) | სახელი | ესტუარის-ID |
|---|---|---|---|
| 1 | 26,00 | ჭოროხი (საქარ... | 1 |
| 2 | 25,20 | კინტრიში | 2 |
| 3 | 60,00 | ნატანები | 3 |
| 4 | 108,00 | სუფსა | 4 |
| 5 | 327,00 | რიონი (სამხრე... | 5 |
| 6 | 327,00 | რიონი (ჩრდი... | 6 |
| 7 | 150,00 | ხობისწყალი | 7 |
| 8 | 213,00 | ენგური | 8 |
| 9 | 400,00 | მტკვარი | 5 |

ნახ.10.16. Insert ოპერაციის განხორციელება

ბოლოს ნაჩვენებია სტრიქონის წაშლის (Delete) ოპერაცია. იგი ხორციელდება მაგალითად, **„მდინარის-ID“** შესაბამისი სტრიქონის მონიშვნით და შემდეგ კომპიუტერის კლავიატურის Delete- ღილაკის ამოქმედებით (ნახ.10.17).




| მდინარის-ID | სიგრძე (კმ) | სახელი | ესტუარის-ID |
|---|---|---|---|
| 1 | 26,00 | ჭოროხი (საქარ... | 1 |
| 2 | 25,20 | კინტრიში | 2 |
| 3 | 60,00 | ნატანები | 3 |
| 4 | 108,00 | სუფსა | 4 |
| 5 | 327,00 | რიონი (სამხრე... | 5 |
| 6 | 327,00 | რიონი (ჩრდი... | 6 |
| 7 | 150,00 | ხობისწყალი | 7 |
| 8 | 213,00 | ენგური | 8 |
| 9 | 400,00 | მტკვარი | 5 |

**ნახ.8.17. Delete ოპერაციის განხორციელება**

ამით დავასრულეთ ეს ლაბორატორიული სამუშაო.

**დავალება:** SQL Server მონაცემთა ბაზის ცხრილის ჩანაწერების დასამუშავებლად გამოიყენეთ BindingNavigator ინსტრუმენტი, გამოიკვლიეთ მისი ფუნქციები.

ააგეთ მომხმარებლის ინტერფეისი რომელიმე საპრობლემო სფეროსთვის (ამოცანა შეათანხმეთ მასწავლებელთან).

## 2.9. რეფაქტორინგი: კოდის დამუშავება და მისი რეორგანიზაცია

### ლაბორატორიული სამუშაო N11

მიზანი: C# პროგრამული კოდის დამუშავებისა და მოდიფიკაციის პროცედურების შესწავლა, „რეფაქტორინგის" არსის გაცნობა და მისი გამოყენება არსებული კოდის ვერსიის განახლების მიზნით.

პროგრამული კოდის დამუშავება ხდება არჩეული პროექტის ტიპისა და კოდის შაბლონის მიხედვით, რომელსაც გვთავაზობს Visual Studio.NET 2013 სამუშაო გარემო. ამავე დროს ხორციელდება საწყისი კოდის სინტაქსის სისწორის შემოწმება, კოდის გენერირების ავტომატური დასრულება, აგრეთვე კოდში ნავიგაციის პროცესის გამოყენება (მაგალითად, კონტექსტური მენიუდან Go to definition -ით).

ყოველივე ეს მნიშვნელოვნად ზრდის პროგრამისტ-დეველოპერების მუშაობის მწარმოებლურობას!

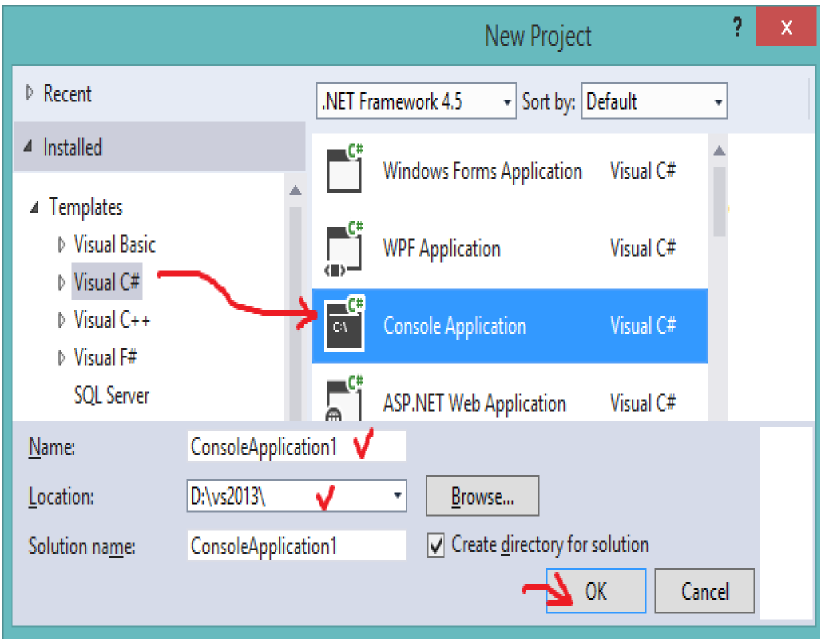
„რეფაქტორინგი" არის არსებული პროგრამული კოდის სისტემატური მოდიფიკაცია და სრყლყოფა, მისი ფუნქციონირების სემანტიკის ძირფესვიანი ცვლილების გარეშე. კოდში ცვლილებები ხორციელდება ავტომატიზებული გარდაქმნებით, რომლებსაც .NET სამუშაო გარემო გვთავაზობს.

ამის ტიპური მაგალითია - ***მეთოდის სახელის შეცვლა.*** ამოცანა მდგომარეობს რომელიმე კონკრეტული მეთოდისათვის *დასახელებისა* და მისი *განსაზღვრების* ცვლილების განხორციელებაში, ამასთანავე *მისი გამოყენების ყველა ადგილზე* !

თუ პროექტი საკმაოდ დიდია, მაშინ **ხელით** ასეთი ცვლილებების ჩატარების ამოცანის გადაწყვეტა საკმაოდ შრომატევადი და მოუხერხებელია. არაა გამორიცხული შემთხვევა,

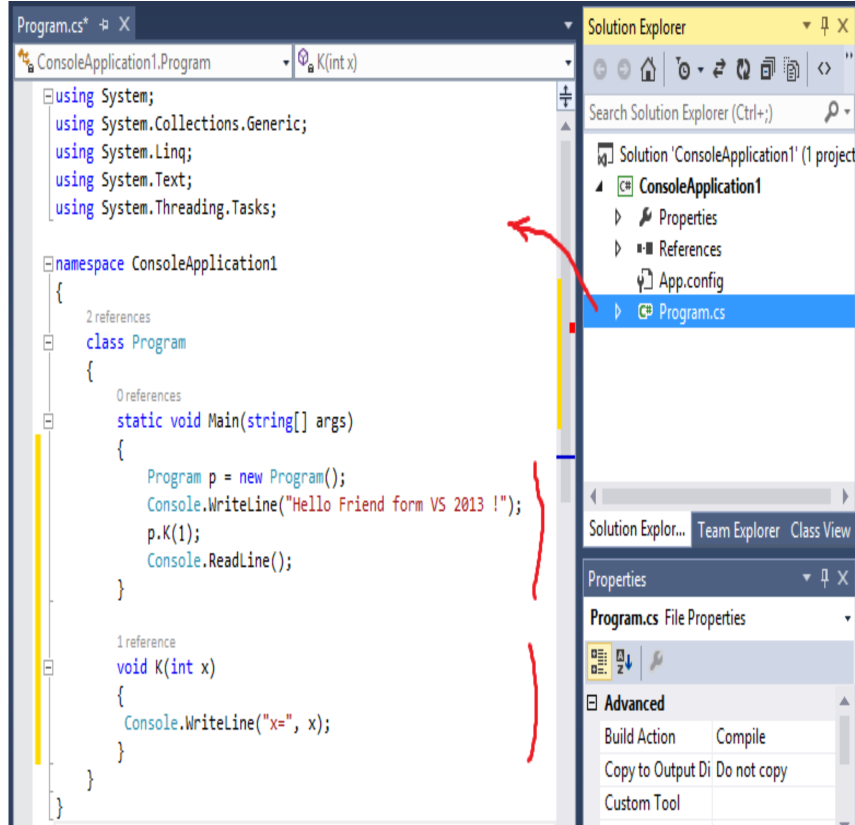
რომ რომელიმე მოდულში დაგვავიწყდეს ამ ცვლილების განხორციელება, რაც შემდგომში, პროგრამის ამუშავებისას, აუცილებლად გამოჩნდება შეცდომის სახით და მოგვიწევს პროექტის ხელახალი შესწორება.

განვიხილოთ რეფაქტორინგის ამოცანა მარტივი კონსოლური აპლიკაციისათვის (ნახ.11.1).




**ნახ.11.1. კონსოლური აპლიკაციის შექმნა**

მიღებული პროგრამის საწყის ტექსტს ჩავამატეთ რამდენიმე სტრიქონი, რომელსაც აქვს 11.2 ნახაზზე ნაჩვენები სახე.



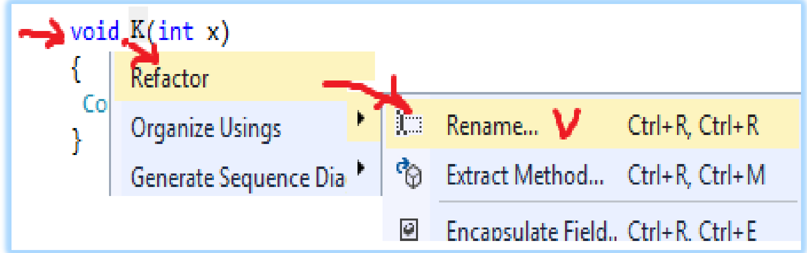

ნახ.11.2

პროგრამაში განსაზღვრულია Program კლასი, სტატიკური მეთოდი Main და K- ეგზემპლარის მეთოდი **x** არგუმენტით.

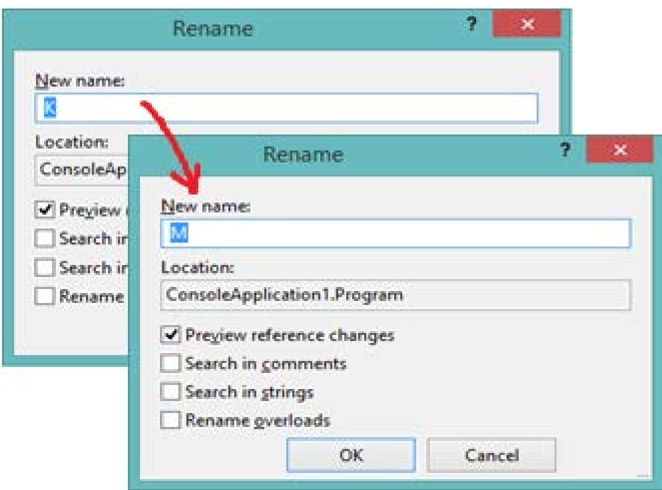
დავუშვათ, რომ საჭიროა შეიცვლაოს K მეთოდის სახელი M-ით. ცვლილება უნდა განხორციელდეს არა მხოლოდ ამ მეთოდის განსაზღვრებაში, არამედ მისი გამოყენების ადგილებზეც!

კოდის რედაქტირების გარემოში მაუსის კურსორს ვაყენებთ K მეთოდის პირველ სტრიქონზე, მოვნიშნავთ შესაცვლელ

სიტყვასა და კონტექსტური მენიუდან ვირჩევთ პუნქტს Refactor / Rename (ნახ.11.3).

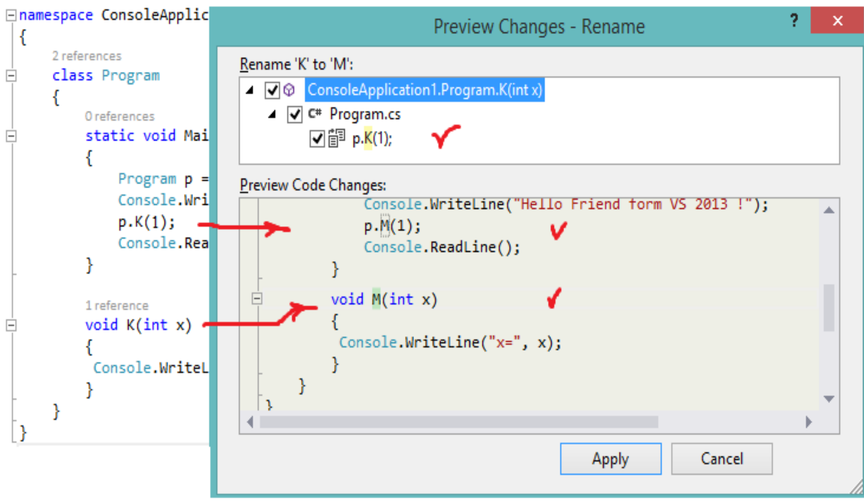



ნახ.11.3. K მეთოდისთვის რეფაქტორინგის ქმედების არჩევა




ნახ.11.4. K-იცვლება M-ით

მიიღება 11.5 სურათი.




ნახ.11.5. დაგეგმილი ცვლილებების წინასწარ ნახვა (Preview Changes)

თუ ყველაფერი წესრიგშია, მაშინ ვირჩევთ Apply-სა და ვიღებთ რეფაქტორინგის შედეგს (ნახ.11.6).

```
Program.cs
ConsoleApplication1.Program
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    2 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Program p = new Program();
            Console.WriteLine("Hello Friend form VS 2013 !");
            p.M(1);
            Console.ReadLine();
        }
        1 reference
        void M(int x)
        {
         Console.WriteLine("x=", x);
        }
    }
}
```

ნახ.11.6. რეფაქტორინგის შედეგი

VS 2013 სამუშაო გარემოს აქვს რეფაქტორინგის შემდეგი შესაძლებლობები:

- Rename - სახელის შეცვლა;
- Extract Method - მეთოდის ამოღება: კოდის მონიშნული ფრაგმენტის მოდიფიკაცია მეთოდში მითითებული სახელით;
- Encapsulate Field - ველის ინკაფსულირება, მისი პრივატად გაკეთებით, ოღონდ Public თვისების დამატებით მისი წვდომის მიზნით;
- Extract Interface - ინტერფეისის ამღება: კლასის ტექსტის მონიშვნა მისთვის ავტომატურად შესაბამისი ინტერფეისის ფორმირება (თუ ეს შესაძლებელია);
- Remove parameters - მეთოდის პარამეტრების ნაწილის წაშლა;
- Reorder parameters - მეთოდის პარამეტრების რიგითობის შეცვლა.

**საშინაო დავალება -დამოუკიდებელი სამუშაოსათვის:**

ააგეთ მარტივი კონსოლის აპლიკაციის პროექტი Visual Studio.NET 2013 სამუშაო გარემოში (მაგალითად: უნივერსიტეტი, სუპერმარკეტი, აფთიაქი, პროდუქციის_წარმოება_მიწოდება ან სხვა). ჩაატარეთ რეფაქტორინგის პროცედურები და გამოიკვლიეთ მისი ზემოჩამოთვლილი სახეების შესაძლებლობები.

## 2.10. Visual Studio.NET -ის რევერსიული ინჟინერიის ინსტრუმენტული საშუალებები:: „Model-Code-Model…“

### ლაბორატორიული სამუშაო N12

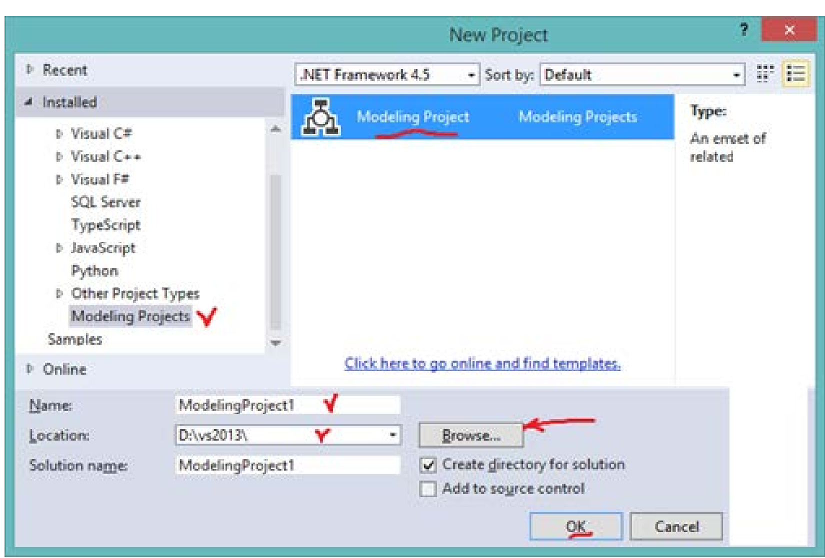
<u>მიზანი:</u> C# პროგრამული კოდის ავტომატური გენერაციის პროცედურის შესწავლა UML-ის კლასთა დიაგრამის საფუძველზე.

განვიხილოთ მარტივი UML-მოდელის გენერაცია ორი ურთიერთდაკავშირებული კლასის საფუძველზე, შემდეგ ამ მოდელის ვიზუალიზაცია და კოდის გენერაცია C# ენაზე.

UML-მოდელის ასაგებად გამოიყენება სპეციალური სახის პროექტი **Modeling Project**.

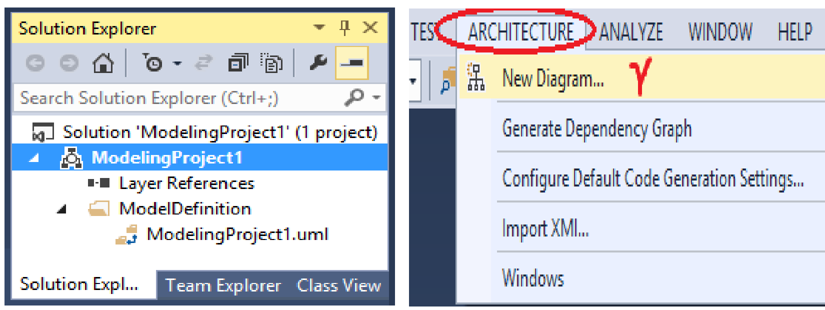
შევქმნათ პროექტი მოდელირების მიზნით. VS 2013 გარემოს მთავარ მენიუში ავირჩიოთ (ნახ.12.1):

**File / NewProject და პროექტის ტიპი: Modeling Project**




ნახ.12.1. პროექტის შექმნა UML-მოდელის ასაგებად

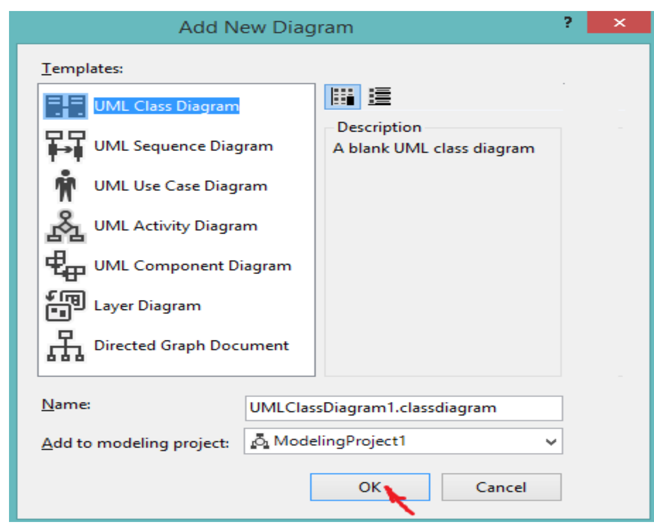
მიიღება 12.2 ნახაზზე ნაჩვენები Solution Explorer.




ნახ.12.2

ნახ.12.3

პროექტში ახალი UML-დიაგრამის დასამატებლად ავირჩიოთ მენიუს პუნქტი Architecture, რომელიც სპეციალურად გამოიყენება პროგრამის არქიტექტურის წარმოსადგენად UML-მოდელის სახით და მის შიგნით პუნქტი Add new Diagram (ნახ.12.3). მიიღება ფანჯარა (ნახ.12.4).

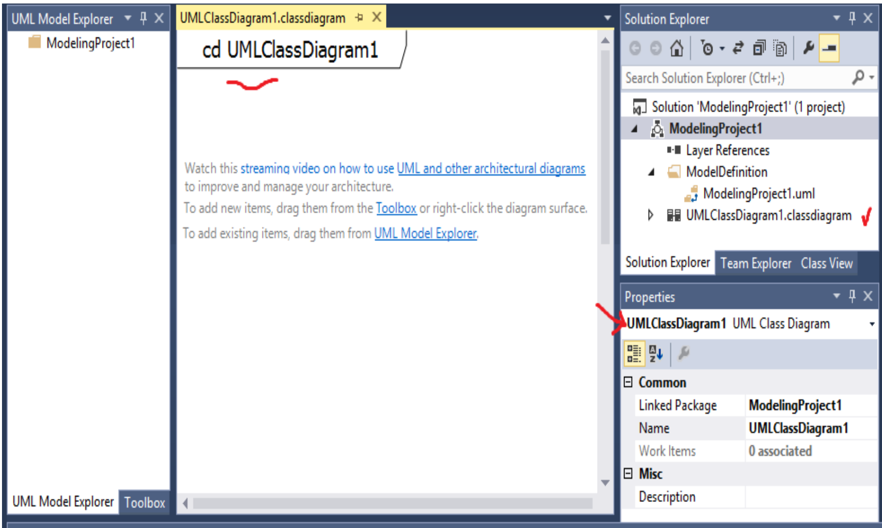



ნახ.12.4. ახალი UML-დიაგრამის არჩევა

ავტომატურად აირჩევა კლასების დიაგრამა (**UML Class Diagram**), როგორც ხშირად გამოყენებადი. მაგრამ UML-ენის ახალი ვერსიის შესაბამისად, აქ შესაძლებელია სხვა დიაგრამების შექმნაც, როგორიცაა:

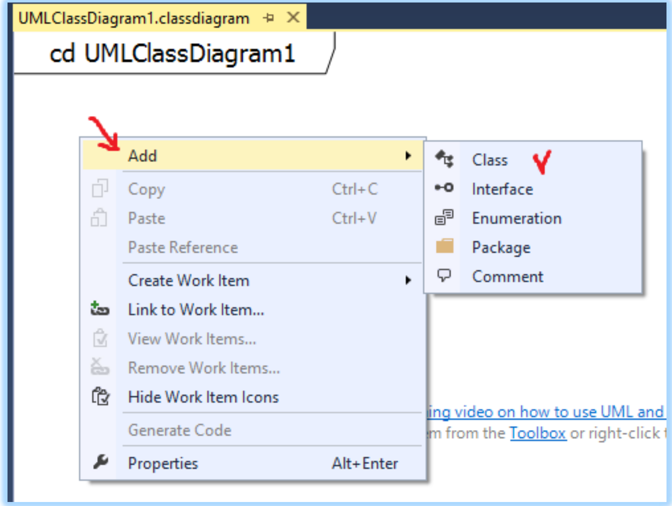
- UML Sequence Diagram - მიმდევრობითობის დიაგრამა;
- UML Use Case Diagram - გამოყენებით შემთხვევათა დიაგრამა;
- UML Activity Diagram - აქტიურობის დიაგრამა;
- UML Component Diagram - კომპონენტების დიაგრამა;
- Layer Diagram - დონეების დიაგრამა;
- Directed Graph Document - დიაგრამა, რომელიც ასახავს დოკუმენტს ორიენტირებული გრაფის სახით.

შევქმნათ ახალი კლასთა დიაგრამა, თავიდან ცარიელი (ნახ.12.5).



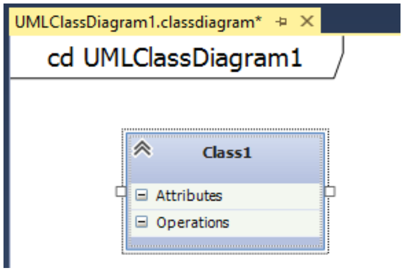

**ნახ.12.5. კლასების (ცარიელი) დიაგრამის შექმნა**

კლასის დასამატებლად დიაგრამაზე ვირჩევთ მენიუს პუნქტს Add / class (ნახ.12.6).



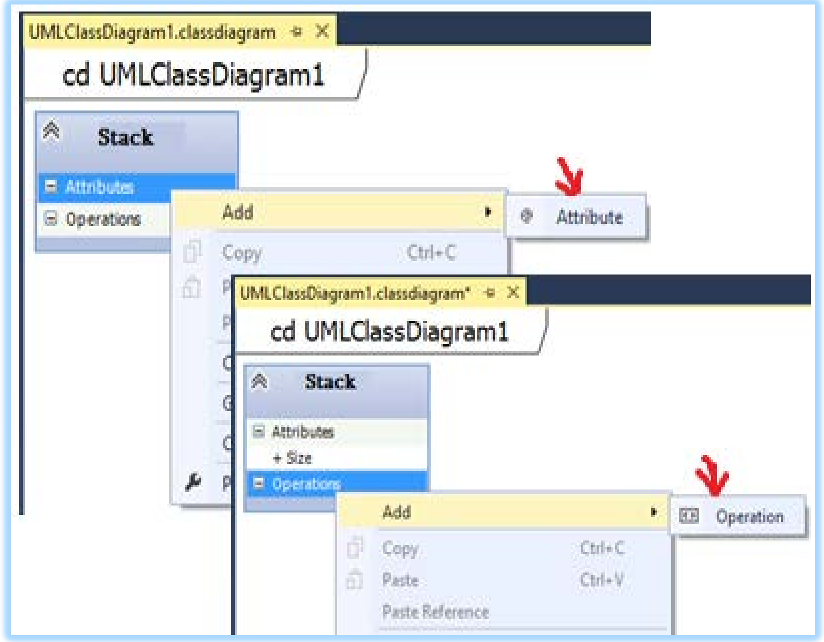

ნახ.12.6. დიაგრამაზე ახალი კლასის დამატება

ახალი კლასი ემატება ავტომატურად სახელით Class1. იგი შედგება ატრიბუტებისა და ოპერაციებისაგან (ნახ.12.7).

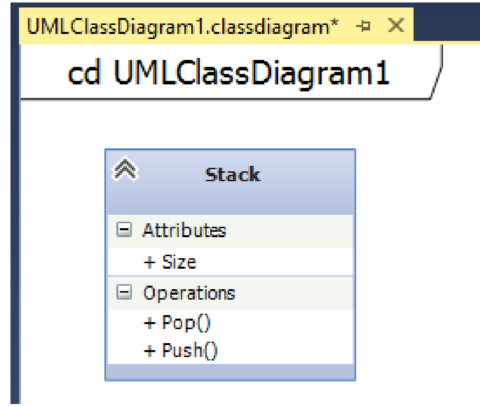



ნახ.12.7. კლასი ატრიბუტებითა და ოპერაციებით

შევცვალოთ მოდელში კლასის სახელი Stack-ით, ატრიბუტის სახით დავამატოთ Size, ხოლო ოპერაციის სახით Push და Pop (ნახ.12.8). დასამატებლად გამოიყენება კონტექსტური მენიუ (მაუსის მარჯვენა ღილაკით) და პუნქტი Add. შედეგი მოცემულია 12.9 ნახაზზე.

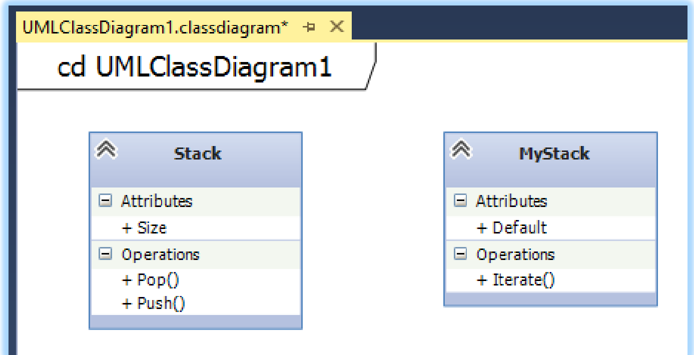



ნახ.12.8. ატრიბუტისა და ოპერაციის დამატება კლასში




ნახ.12.9. Stack კლასის შედეგი

ახლა დავამატოთ მეორე კლასი - MyStack, ატრიბუტით Default და ოპერაციით Iterate (ნახ.12.10).




**ნახ.12.10. MyStack კლასის დამატება Default ატრიბუტითა და Iterate ოპერაციით**

ახლა დიაგრამაზე უნდა ავსახოთ ინფორმაცია იმის შესახებ, რომ MyStack კლასი არის Stack კლასის მემკვიდრე. ვდგებით MyStack კლასზე და კონტექსტურ მენიუში ვირჩევთ Add-ს.

აქ უნდა ამოვირჩიოთ კავშირის ელემენტი და დავამატოთ დიაგრამაზე:

Attribute - ახალი ატრიბუტი,

Operation - ახალი ოპერაცია,

Comment - კომენტარი;

Association - ასოციაცია,

Aggregation - აგრეგაცია,

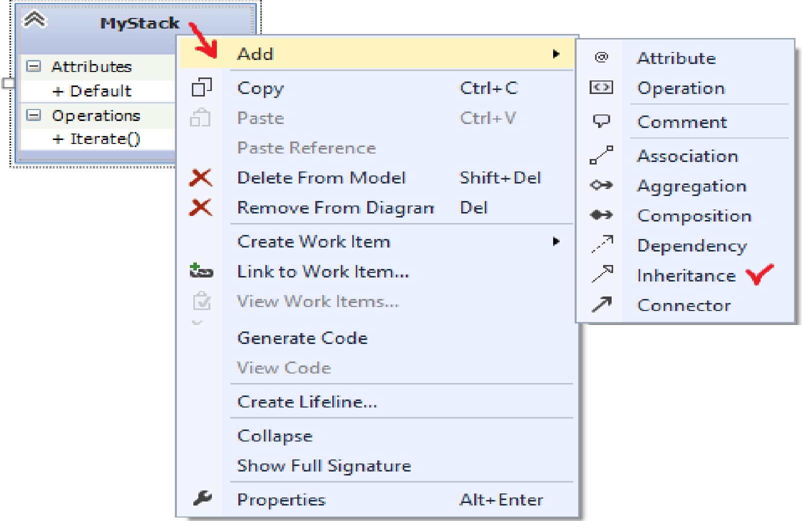
Composition - კომპოზიცია,

Dependency - დამოკიდებულება,

Inheritance - მემკვიდრეობითობა,

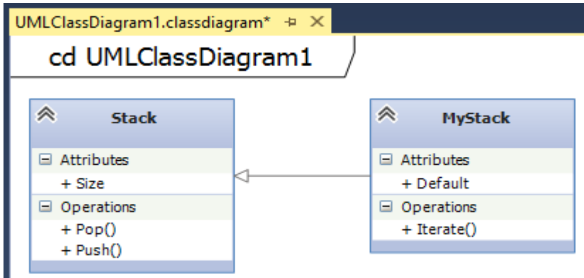
Connector- კონექტორი.

ჩვენ გვაინტერესებს Inheritance (ნახ.12.11).



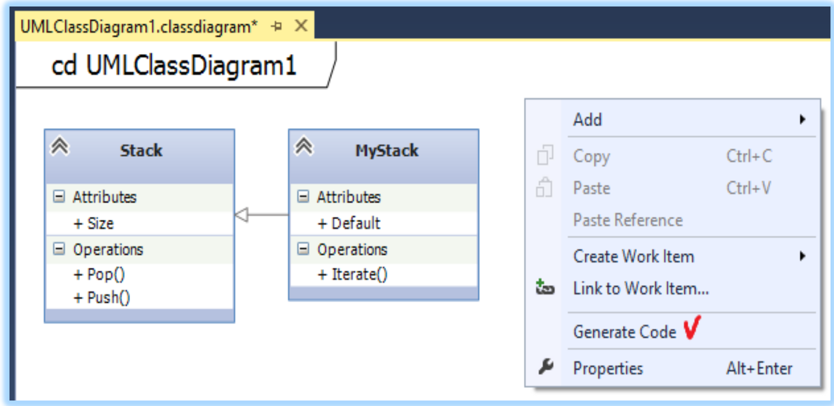

ნახ.12.11. ორ კლასს შორის Inheritance-კავშირის არჩევა

Add / Inheritance - არჩევით დიაგრამაზე დაემატა მემკვიდრეობითობის კავშირი ისრით მიმართული MyStack-იდან Stack-ისკენ (ნახ.12.12).



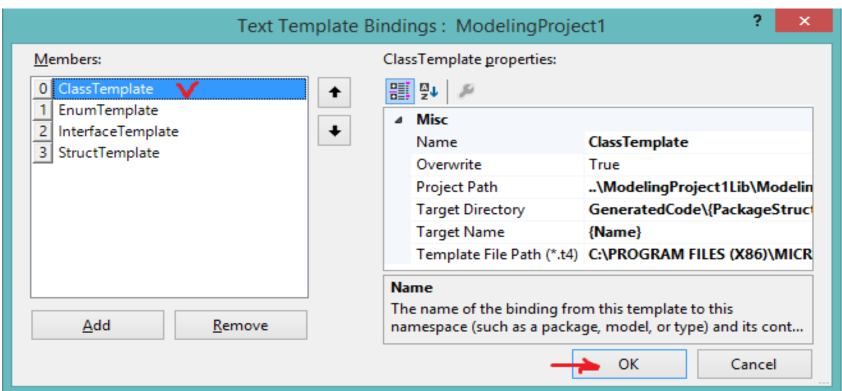

ნახ.12.12. კლასთა კავშირი Inheritance

ახლა შეიძლება კოდის გენერირება, რომელსაც შემდომში გამოვიყენებთ. კონტექსტურ მენიუში ვირჩევთ პუნქტს - Generate code (ნახ.12.13).



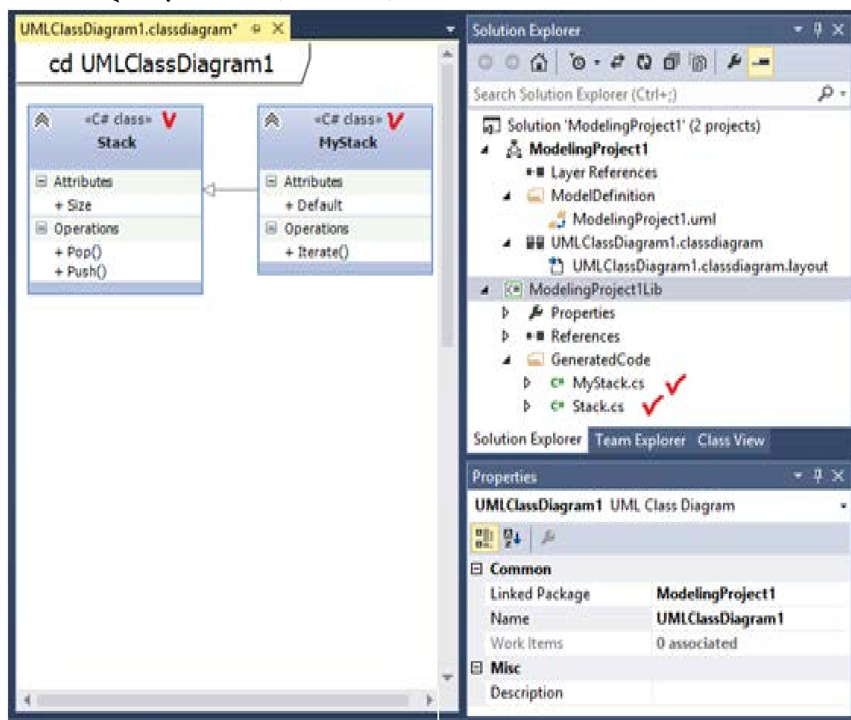

ნახ.12.13. კოდის გენერაცია UML-დიაგრამაზე

კოდის გენერატორი გვთხოვს, დავაზუსტოთ, რომელი შაბლონით (Template) მოხდება გენერაცია. ვირჩევთ შაბლონს კლასისათვის (ნახ.12.14).




ნახ.12.14. ClassTemplate - შაბლონის არჩევა კოდის გენერაციისათვის

კოდის გენერაციის შემდეგ Solution Explorer-ში გამოჩნდება ორი ახალი სტრიქონი, C#-ის ფაილებისათვის: Stack.cs და MyStack.cs (ნახ.12.15).




**ნახ.12.15. Stack.cs და MyStack.cs კოდების გენერაცია UML-დიაგრამიდან**

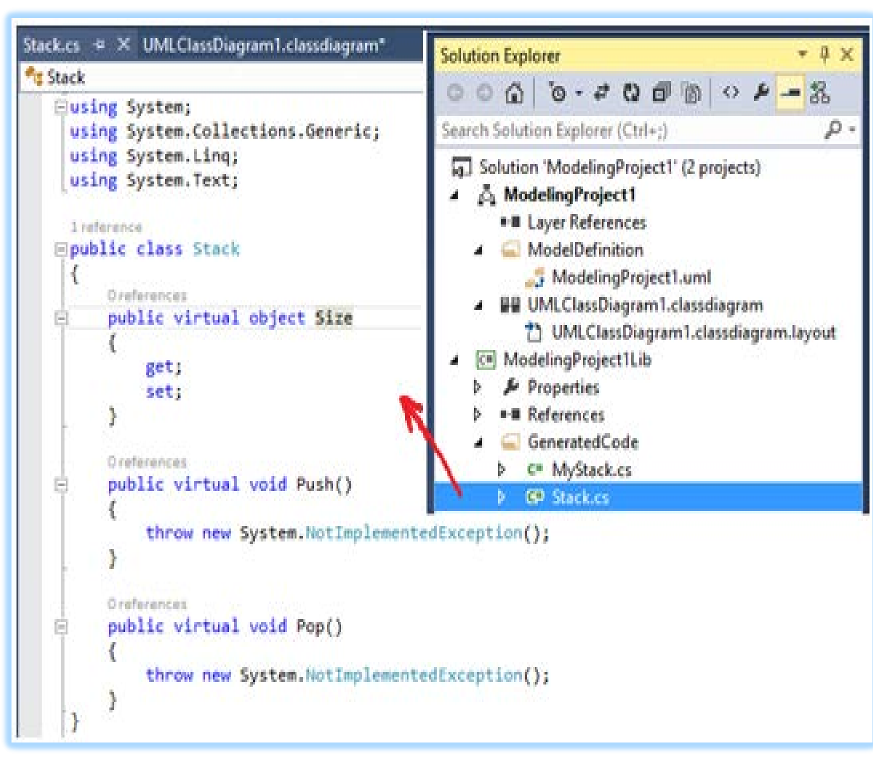
გავხსნათ ახალი Stack.cs და MyStack.cs ფაილები. ტექსტები მოცემულია 12.16 და 12.17 ნახაზებზე.

ატრიბუტი Size რეალიზებულია კლასისი თვისების (propety) სახით get და set მეთოდებით. ხოლო მეთოდების სახშობები წარმოდგენილია ვირტუალური მეთოდების სახით და რეალიზებულია როგორც გამონაკლისის გენერაცია. მაგალითად,

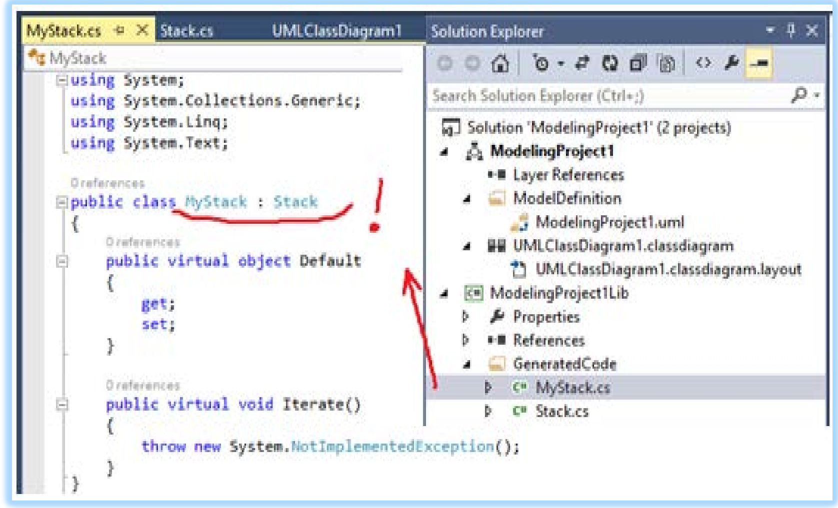
```
public virtual void Push()
  {
    throw new System.NotImplementedException();
  }
```

12.17 ნახაზზე წარმოდგენილი კოდიდან ჩანს კლასების მემკვიდრეობითობა:

Public class MyStack : Stack




ნახ.12.16. გენერირებული **Stack.cs** ფაილის კოდი




ნახ.12.17. გენერირებული MyStack.cs ფაილის კოდი

ამის შემდეგ გენერირებული ფაილები შეიძლება გამოყენებულ იქნას დამუშავების მომდევნო ეტაპებზე. აგებული მოდელი თამაშობს მნიშვნელოვან როლს: ესაა მოდელირებისა და პროექტირების წინასწარი ეტაპების შედეგების ასახვა პროექტში.

პროექტის შეცვლის აუცილებლობისათვის შეიძლება ცვლილებები ჩატარდეს UML-დიაგრამებში, საიდანაც კოდები ავტომატურად იქნება გენერირებული. შესაძლებელია პირიქითაც (**reverse engineering**-ის გამოყენება), რეალიზებული კლასებიდან მოხდეს UML-დიაგრამების გენერირება.

**<u>საშინაო დავალება:</u>** შევქმნათ ახალი პროექტი Visual Studio.NET სამუშაო გარემოში Modeling Project-ის გამოყენებით, რომელშიც აგებულ იქნება კლასთა დიაგრამა სამი კლასით: „მშობელი“-Person, „შვილები“ Student და Lector. შეიტანეთ ატრიბუტები და ოპერაციები თითოეულისათვის, გამოიყენეთ მემკვიდრეობითობის კავშირი „მშობელ-შვილების“ კლასებს შორის. ბოლოს განახორციელეთ კლასების კოდის ავტომატური გენერირება C# ენაზე.

## 2.11. პროგრამული აპლიკაციების ტესტირება

## ლაბორატორიული სამუშაო N13

მიზანი: პროგრამული კოდების ტესტირების პროცესის შესწავლა Visual Studio.NET ინტეგრირებულ გარემოში.

### 1. თეორიული ნაწილი

Unit testing და Coded UI არის Microsoft-ის ტესტირების ინსტრუმენტები, რომლებიც სრულდება Visual Studio.NET-გარემოში.

Unit testing - ანუ მოდულური ტესტირება დაპროგრამების პროცესია, რომლის საშუალებითაც მოწმდება საწყისი კოდის ცალკეული მოდულების კორექტულობა. ასეთი ტესტირების იდეა მდგომარეობს იმაში, რომ ყოველი არატრივიალური ფუნქციის ან მეთოდისათვის დაიწეროს ტესტი. ეს უზრუნველყოფს კოდის სწრაფად შემოწმებას, ხომ არ მიიყვანა კოდის ბოლო ცვლილებამ პროგრამა რეგრესიამდე ანუ შეცდომების გაჩენამდე პროგრამის უკვე ტესტირებულ ნაწილებში [39].

Coded UI ტესტი კი ავტომატურად იწერს, შესრულებაზე უშვებსა და ამოწმებს ტესტ - ქეისებს.

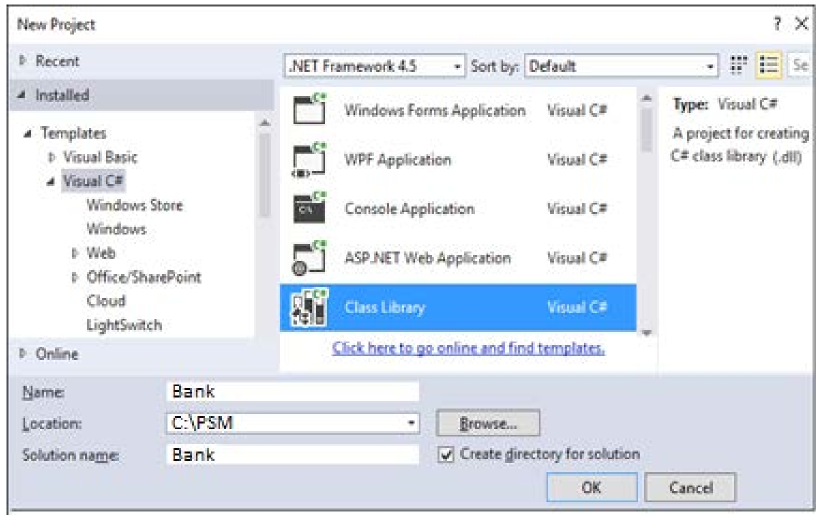
ასეთი ტესტების წერა შესაძლებელია C# ან Visual Basic-ზე Visual Studio გარემოში. Unit testing ტესტირების ტექნოლოგია განვიხილოთ ვირტუალური ობიექტის, მაგალითად, ფინანსური ობიექტის, ბანკის მაგალითზე.

- დასატესტი პროგრამის პროექტის შექმნა: Visual Studio–ში საჭიროა ავირჩიოთ: **File -> New -> Project.**

შედეგად გამოჩნდება დიალოგური ფანჯარა (ნახ.13.1). სადაც ავირჩევთ:

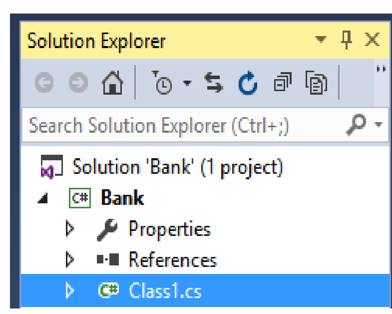
**Visual C# => ClassLibrary**

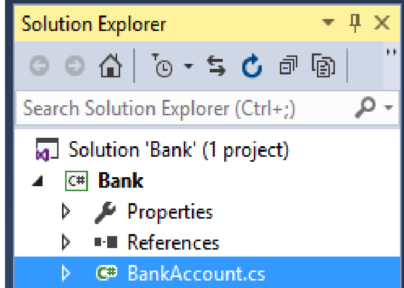
პროექტი სახელით Bank.




ნახ.13.1. დასატესტი კლასის პროექტის შექმნა

მიიღება 13.2 ნახაზზე ნაჩვენები Solution E[plorer ფანჯარა. აქ Class1.cs სახელი შევცვალოთ BankAccount.cs -ით.




ნახ.13.2. Class1 - > BankAccount

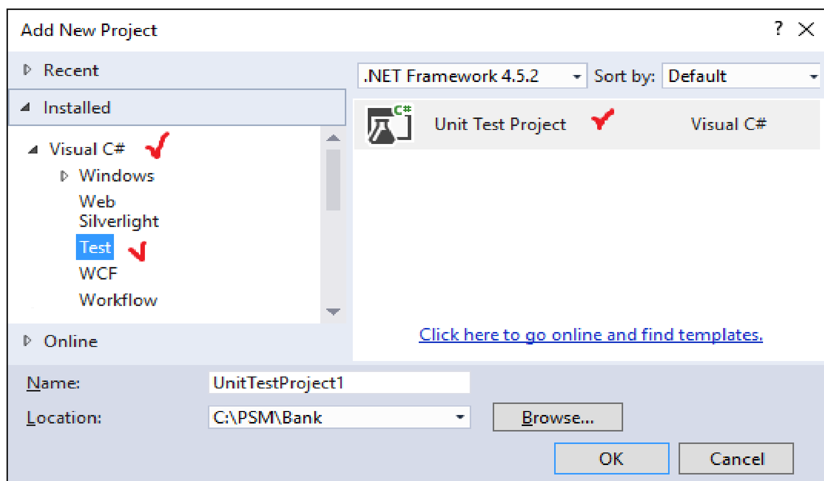
შემდეგ BankAccount.cs-ის ტექსტი რედაქტორის არეში შევცვალოთ ჩვენი დასატესტი პროგრამის კოდით.

ეს საწყისი ტექსტი, მაგალითად, მოცემულია 13.1 ლისტინგში.

//-- ლისტინგი_13.1 --- BankAccount.cs --------
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BankAccountNS
{
 public class BankAccount
  {
    private string m_customerName;
    private double m_balance;
    private bool m_frozen = false;
   private BankAccount()
    {
    }

   public BankAccount(string customerName, double balance)
    {
      m_customerName = customerName;
      m_balance = balance;
    }
   public string CustomerName
    {
      get { return m_customerName; }
    }
   public double Balance
    {
      get { return m_balance; }
    }
   public void Debit(double amount)
    {
      if (m_frozen)
       {
         throw new Exception("Account frozen");
       }
```

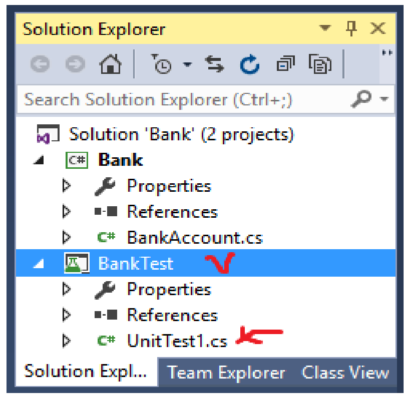
```
     if (amount > m_balance)
      {
        throw new ArgumentOutOfRangeException("amount");
       }
      if (amount < 0)
       {
        throw new ArgumentOutOfRangeException("amount");
       }
       m_balance += amount; // განზრახ არასწორი კოდი
       // m_balance -= amount;   // გასწორებული
    }
   public void Credit(double amount)
    {
      if (m_frozen)
        {
        throw new Exception("Account frozen");
        }
      if (amount < 0)
       {
        throw new ArgumentOutOfRangeException("amount");
       }
       m_balance += amount;
    }
   private void FreezeAccount()
    {
       m_frozen = true;
    }
   private void UnfreezeAccount()
    {
       m_frozen = false;
    }
   public static void Main()
    {
     BankAccount ba = new BankAccount("Mr.Bryan Walton",
                                      11.99);
     ba.Credit(5.77); ba.Debit(11.22);
     Console.WriteLine("Current balance is ${0}",
                         ba.Balance);
    }
}}
```

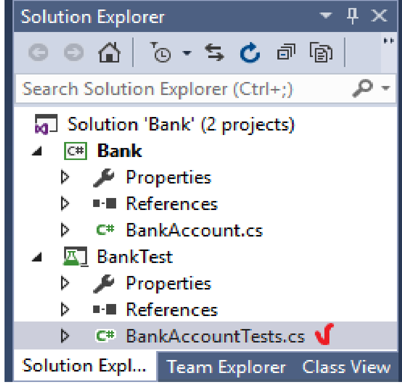
- ტესტ-ფაილის პროექტის აგება (Unit Test Project):




ნახ.13.3. Unit Test პროექტის შექმნა

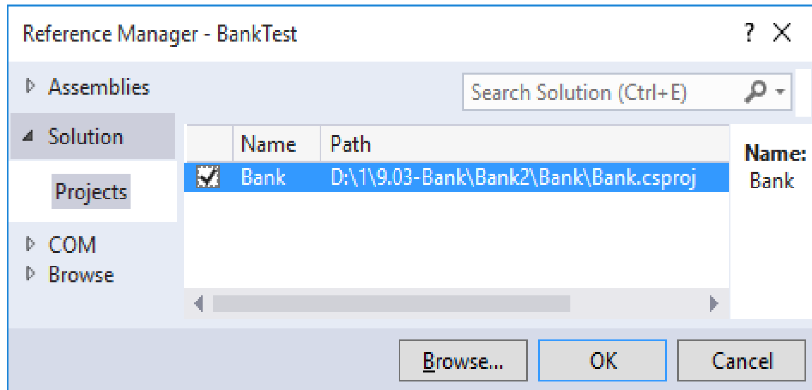
მივიღებთ 14.4. ნახაზზე ნაჩვენებ სურათს BankTest პროექტით. მარჯვენა სურათზე შეცვლილია UnitTest კლასის სახელი BankAccountTests-ით.



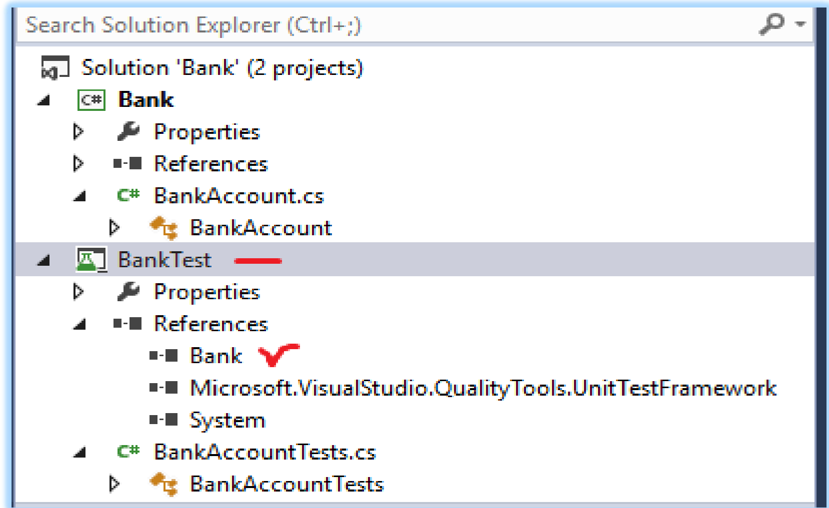

ნახ.13.4

**BankTests** პროექტში დავამატოთ reference **Bank** solution-იდან. ამისათვის **BankTests**-ზე მაუსის მარჯვენა ღილაკით ავირჩიოთ **Add Reference** და მივიღებთ 13.5 ნახაზზე ნაჩვენებ ფანჯარას. აქ Solution სტრიქონში ვირჩევთ Projects და Bank-ის ჩეკბოქსს მოვნიშნავთ.




ნახ.13.5

მივიღებთ 14.6 ნახაზზე ნაჩვენებ შედეგს.




ნახ.13.6

ჩავამატოთ BankAccountTests პროგრამაში სახელსივრცე Bank-ის პროექტიდან: `using BankAccountNS;`

ამგვარად, BankAccountTests.cs ფაილს ექნება 13.2 ლისტინგზე ნაჩვენები სახე.

//-- ლისტინგი_13.2 ----- BankAccountTests.cs -----
```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using BankAccountNS;

namespace UnitTestProject1
{
    [TestClass]
    public class BankAccountTests
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```
ახლა შევქმნათ პირველი ტესტ - მეთოდი. ამ პროცედურაში, დაიწერება უნიტ - ტესტის მეთოდები BankAccount class-ის Debit მეთოდის ქცევის ვერიფიკაციისათვის. ეს მეთოდები ზემოთაა ჩამოთვლილი.

დასატესტი მეთოდების ანალიზის გზით გაირკვა, რომ საჭიროა მინიმიუმ სამი ქცევის შემოწმება:

1. მეთოდი ქმნის ArgumentOutOfRangeException - გამონაკლისს, თუ კრედიტის ჯამი გადააჭარბებს ბალანსს;
2. იგი ქმნის ArgumentOutOfRangeException - გამონაკლისს მაშინაც, როცა კრედიტის ზომა უარყოფითია;
3. თუ 1 და 2 პუნქტები წარმატებით დასრულდა, მაშინ მეთოდი ითვლის ჯამს ბალანსის ანგარიშიდან.

პირველ ტესტში შევამოწმოთ, რომ კრედიტის დასაშვები მნიშვნელობისთვის (როცა დადებითი მნიშვნელობისაა და ბალანსის ანგარიშზე ნაკლებია) ანგარიშიდან მოიხსნება საჭირო თანხა.

1. დავამატოთ BankAccountTests კლასს შემდეგი მეთოდი:

```
// unit test code -----
[TestMethod]
public void Debit_WithValidAmount_UpdatesBalance()
{
  // arrange
   double beginningBalance = 11.99;
   double debitAmount = 4.55;
   double expected = 7.44;
   BankAccount account = new BankAccount("Mr. Dito",
                 beginningBalance);
   // act
   account.Debit(debitAmount);
   // assert
   double actual = account.Balance;
   Assert.AreEqual(expected, actual, 0.001, "Account not debited
              correctly");
}
```

მეთოდი საკმაოდ მარტივია. ჩვენ ვქმნით ახალ BankAccount ობიექტს საწყისი ბალანსით და შემდეგ ვაკლებთ სწორ ოდენობას. ჩვენ ვიყენებთ Microsoft-ის unit-ტესტის ფრეიმვორკს მართვადი კოდის AreEqual მეთოდისათვის, რათა მოხდეს საბოლოო ბალანსის ვერიფიკაცია - არის ის, რასაც ჩვენ ველით.

ტესტ - მეთოდის მოთხოვნები ასეთია:

1. მეთოდი მონიშნული უნდა იყოს [TestMethod] ატრიბუტით;
2. მეთოდმა უნდა დააბრუნოს void;
3. მეთოდს არ შეიძლება ჰქონდეს პარამეტრები.

ტესტის აგება და ამუშავება:

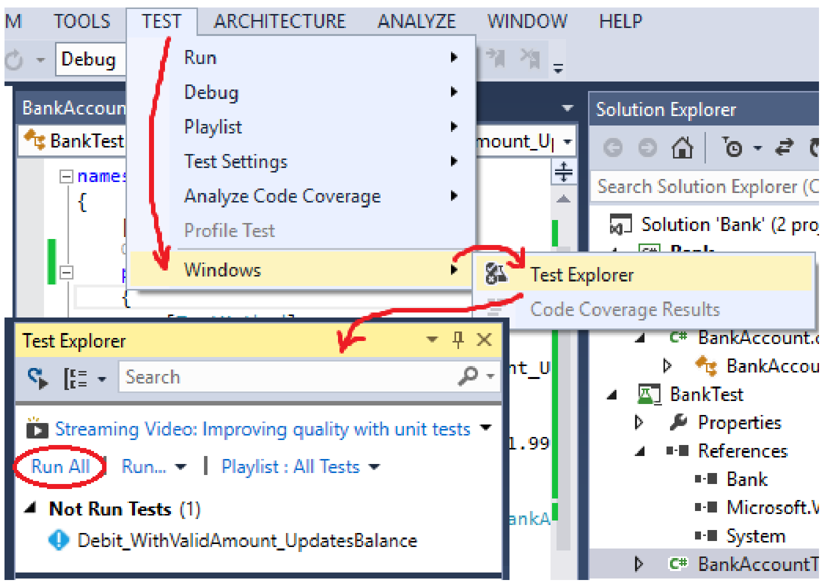
მთლიანი ტესტის კოდი მოცემულია 13.3 ლისტინგში.

**//-- ლისტინგი_13.3 ----- BankAccountyTests.cs ----**

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using BankAccountNS;

namespace UnitTestProject1
{
  [TestClass]
  public class BankAccountTests
  {
     [TestMethod]
     public void Debit_WithValidAmount_UpdatesBalance()
     {
       // arrange
       double beginningBalance = 11.99;
       double debitAmount = 4.55;
       double expected = 7.44;
       BankAccount account = new BankAccount("Mr. Dito",
                    beginningBalance);
       // act
       account.Debit(debitAmount);

       // assert
       double actual = account.Balance;
       Assert.AreEqual(expected, actual, 0.001, "Account
                not debited correctly");
     }
  }
}
```

- BUILD მენიუდან ვირჩევთ Build Solution;

- TEST მენიუდან ვირჩევთ Windows  და Test Explorer პუნქტებს. იხსნება Test Explorer ფანჯარა (ნახ.13.7).




ნახ.13.7.

აქ ვირჩევთ Run All - სა და ვიღებთ შედეგს (ნახ.14.8).

Test Explorer

Search

Streaming Video: Improving quality with unit tests

Run All | Run... | Playlist : All Tests

Failed Tests (1)

Debit_WithValidAmount_UpdatesBalance 74 ms

Summary

Last Test Run Failed (Total Run Time 0:00:01)

1 Test Failed

```
namespace UnitTestProject1
{
    [TestClass]
    0 references
    public class BankAccountTests
    {
        [TestMethod]
        | 0 references
        public void Debit_WithValidAmount_UpdatesBalance()
        {
            // arrange
            double beginningBalance = 11.99;
            double debitAmount = 4.55;
            double expected = 7.44;
            BankAccount account = new BankAccount("Mr. Dito",
            . . .
```

ნახ.13.8

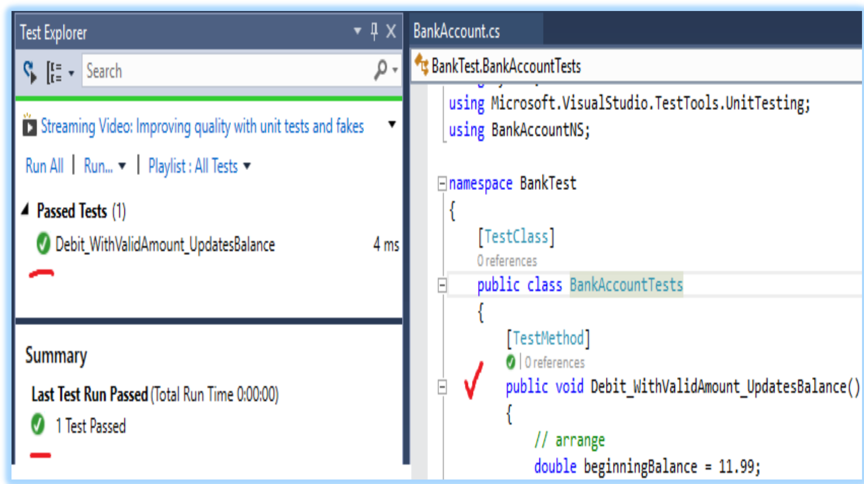
ნახაზზე მითითებული „x“-სიმბოლოები წითელ წრეშია მოთავსებული, ე.ი. ტესტირებამ აღმოაჩინა შეცდომები და კოდი წარმატებით ვერ შესრულდა.

თუ მეთოდი წარმატებით ჩაივლიდა, მაშინ მივიღებდით მწვანე ფერის x-სიმბოლოებს.

შემდეგი ეტაპი კოდის გასწორება და ხელახალი ტესტირებაა. დასატესტ პროგრამაში შევცვალოთ სტრიქონში „ + „ ნიშანი „ – „ -ით.

```
// m_balance += amount; // intentionally incorrect code
m_balance -= amount; // intentionally correct code
```

ტესტის თავიდან ამუშავებით ვიღებთ წარმატებულ შედეგს ანუ მიიღება მწვანე ფერის სიმბოლოები (ნახ.13.9).




ნახ.13.9. სწორი შედეგი

## 2.12. პროგრამული კოდების ხარისხის შეფასება

### ლაბორატორიული სამუშაო N14

**მიზანი:** პროგრამული პროექტების შემუშავების პროცესში პროგრამული მოდულების ხარისხის შეფასების მეთოდების შესწავლა.

პროგრამული უზრუნველყოფის ხარისხი მოცემული დავალების ფარგლებში დადგენილი მოთხოვნილებების სრულყოფილად შესაბამისობაა რეალიზებულ პროგრამულ პროდუქტთან. პროგრამული უზრუნველყოფის ხარისხის საერთაშორისო სტანდარტებია - ISO/IEC 25000:2014, IEEE Std 610.12-1990 [40,41].

პროგრამული უზრუნველყოფის ხარისხის მახასიათებლები არაფუნქციონალური მოთხოვნების ნაწილია, რომელიც ძირითადად მოიცავს შემდეგ კრიტერიუმებს:

**გასაგები** - პროგრამული უზრუნველყოფის დანიშნულება გასაგები უნდა იყოს როგორც სისტემის მუშაობიდან, ისე მისი დოკუმენტაციიდან;

**სრული** - პროგრამული უზრუნველყოფის ყველა აუცილებელი კომპონენტი უნდა იყოს სრულად წარმოდგენილი და რეალიზებული;

**ლაკონიური** - ზედმეტი და დუბლირებული ინფორმაცია უნდა იყოს შეზღუდული. კოდის განმეორებადი ნაწილებისათვის დაცული უნდა იყოს პოლიმორფიზმის პრინციპი;

**პორტირების შესაძლებლობა** - პროგრამული უზრუნველყოფა ადვილად უნდა ადაპტირდეს ახალ გარემოში (მაგალითად, არქიტექტურის, პლატფორმის, ვერსიისა და ა.შ. შეცვლისას).

**შეთანხმებული** -პროგრამული უზრუნველყოფის ნებისმიერ კომპონენტში (პროგრამული კოდი, ტექნიკური დავალება,

ტექნიკური პროექტი, დოკუმენტაცია და ა.შ.) გამოყენებულ უნდა იქნას ერთიანი, შეთანხმებული ტერმინოლოგია და აღნიშვნები.

არსებობს კოდის შემოწმებისა და ანალიზის შემდეგი მიდგომები:

1. **Maintainability Index** (მხარდაჭერის ინდექსი) - კოდის ხარისხის კომპლექსური მაჩვენებელია. იგი განისაზღვრება ფორმულით [40]:

**MI=MAX(0,(171 - 5.2 * ln(HV) - 0.23 * CC - 16.2 * ln(LoC)) * 100 / 171),**

სადაც,

- HV – Halstead Volume, გამოთვლითი სირთულე. მეტრიკის სიდიდე პირდაპირპროპორციულად იზრდება გამოყენებული ოპერატორების სიმრავლის შესაბამისად;
- CC – Cyclomatic Complexity. კოდის სტრუქტურული სირთულე ანუ კოდში სხვადასხვა განშტოებების რაოდენობა. რაც უფრო მაღალია მაჩვენებელი, მითი უფრო მეტი ტესტირებაა გასაწერი;
- LoC – კოდის სტრიქონების რაოდენობა.

ამ მეტრიკის ფარგლებში კოდის სირთულის მაჩვენებელი ვარირებს 0-დან 100-მდე. რაც უფრო მაღალია მნიშვნელობა, მით უფრო მოქნილია კოდის მხარდაჭერა.

Visual Studio პაკეტში თვალსაჩინოებისათვის შემუშავებულია ფერებად რანჟირებული დაყოფა - მწვანე ფერი შეესაბამება 20-100 დიაპაზონის მნიშვნელობას, ყვითელი ფერი შეესაბამება 10-20 დიაპაზონის მნიშვნელობას, ხოლო 10-ზე ნაკლების დიაპაზონის მნიშვნელობის ფერია წითელი.

2. **Depth of Inheritance** – მემკვიდრეობითობის სიღრმე. თითოეული კლასისათვის აჩვენებს მემკვიდრეობის ჯაჭვის იერარქიას. მაგალითად, პირველი კლასის მემკვიდრეა მეორე კლასი, ხოლო მესამე კლასი მეორე კლასის მემკვიდრეა. შესაბამისად, პირველი კლასის მაჩვენებელია 1, მეორესი 2, მესამესი 3.

3. **Class Coupling** – კლასების ურთიერთდამოკიდებულებისა და დაკავშირების მაჩვენებელი.

კარგი პრაქტიკაა კლასებს შორის დამოკიდებულება არ იყოს „ჩახლართული“ და არ იყოს გამოყენებული დიდი რაოდენობის ქვეკავშირი (გამოთვლაში მონაწილეობს - პარამეტრული კლასები, ლოკალური ცვლადები, მეთოდით დაბრუნებული ტიპები, საბაზო კლასები, ატრიბუტები და სხვ.)

4. **Lines of Code** – კოდის სტრიქონების რაოდენობა (არ გაითვალისწინება კოდში ცარიელი სტრიქონები და კომენტარები).

Visual Studio პაკეტში პროგრამული კოდის მეტრიკის შედეგების მიღება პროექტისთვის ხდება შემდეგი ბრძანებით:

ძირითად მენიუში ღილაკით Analyze - Calculate Code Metrics - for Solution. ასევე, შესაძლებელია ნაწილში Solution Explorer, solution- Calculate Code Metrics გამოძახება თაგუნას მარჯვენა ღილაკით.

აღნიშნული ბრძანება აჩვენებს როგორც ზოგად, ისე კლასების დონეზე ჩაშლილ შედეგს (Code Metrics Results) პარამეტრებით - Maintainability Index, Cyclomatic Complexity, Depth of Inheritance, Class Coupling, Lines of Code (ნახ.14.1).

კოდში კომპილიატორის შეცდომების ან გაფრთხილების ფანჯარა Error List იხსნება ძირითადი მენიუს ღილაკით View-Error List, ასევე ძირითად მენიუში ღილაკით Analyze – Run code Analysis and suppress active issues.

კომპილატორის გაფრთხილება (Compiler warnings) - პროგრამულ კოდში საეჭვო ადგილების არსებობაა, რომელიც პროგრამული ენის თვალსაზრისით არ არის შეცდომა და არ იწვევს პროგრამული კოდის კომპილირების პროცესის შეწყვეტას, თუმცა არის პროგრამული შეცდომა.

Code Metrics Results

Filter: None Min: Max:

| Hierarchy | Maintainability Index | Cyclomatic Comple... | Depth of Inheritance | Class Coupling | Lines of Code |
|---|---|---|---|---|---|
| PojectBudjet (Debug) | 61 | 50 | 7 | 51 | 439 |
| { } PojectBudjet | 61 | 50 | 7 | 51 | 439 |
| Program | 81 | 1 | 1 | 3 | 3 |
| gamotvlebi | 69 | 7 | 1 | 7 | 16 |
| Form_SubXarji | 50 | 9 | 7 | 25 | 118 |
| button1_Click(object, | 92 | 1 | | 3 | 1 |
| Dispose(bool) : void | 80 | 3 | | 3 | 3 |
| Form_SubXarji(Form_ | 62 | 1 | | 5 | 10 |
| dataGridView_SubXarj | 67 | 3 | | 5 | 6 |
| InitializeComponent() | 32 | 1 | | 19 | 98 |
| Form_Xarji | 58 | 15 | 7 | 30 | 103 |
| Form3 | 45 | 18 | 7 | 42 | 199 |

ნახ.14.1

კომპილატორის გაფრთხილება შესაძლებელია მნიშვნელოვანი იყოს საინფორმაციო სისტემების რისკების მართვის პროცესისათვის, რაც იმის მანიშნებელია, რომ კოდს შესაძლოა ჰქონდეს მრავალი სისუსტე, ღია ადგილები ან გადატვირთული გამოუყენებელი ელემენტები. ასეთი ტიპის შევდომებმა შესაძლოა გამოიწვიოს კოდის შესრულების შენელებაც.

ნაწილში (Warnings) ველი suppress (გაბათილებადია) მიანიშნებს, კოდის რომელი ელემენტია გამოცხადებული, თუმცა გამოუყენებელი.

## 2.13. პროექტის ბიუჯეტის შედგენისათვის სამომხმარებლო აპლიკაციის მომზადება

### ლაბორატორიული სამუშაო N15

**მიზანი:** პროექტის ბიუჯეტისა და ხარჯის ელემენტების გაცნობა. ცხრილის ჩანაწერებსა და ტესტურ ველებში მონაცემების მიმოცვლა. ორ დიალოგურ ფორმას შორის (კლასებს შორის) მონაცემების ურთიერთგაცვლა. პროექტის ბიუჯეტის ბაზაზე ხარჯებისა და ქვეხარჯების აღრიცხვა.

პროექტების მართვაში ერთ-ერთი მთავარი დატვირთვა აქვს პროექტის ბიუჯეტის მართვის საკითხს. პროექტების მართვის სტანდარტის მიხედვით, პროექტების მართვის მოდელია ე.წ. სამმაგი შეზღუდვა ან რკინის სამკუთხედი, რომლის პარამეტრებია მიზანი, ბიუჯეტი/ღირებულება და გეგმა.

პროექტების მართვისას ფასთან მიმართებაში განიხილება ორი ვარიანტი: 1. პროექტის ბიუჯეტი ცნობილია და უნდა გადანაწილდეს თანხა რესურსებისა და დასახარჯი დროის მიხედვით (აღმავალი მიდგომა); 2. პროექტის ღირებულება უნდა შედგეს რესურსებზე თანხის გადანაწილებით გონივრულ ფარგლებში (დაღმავალი მიდგომა).

მიღებულია პროექტის ბიუჯეტის შედგენისას შემდეგი ხარჯების გათვალისწინება:

- პირდაპირი ხარჯები: მაგალითად, ადამიანური რესურსები, საკონსულტაციო მომსახურება, პროგრამული უზრუნველყოფის შესყიდვა/ლიცენზიები, მივლინება, აპარატურა/მომსახურების ხარჯი
- ირიბი ხარჯი: მაგალითად, ზედნადები ხარჯები სატელე-ფონო/საკომუნიკაციო ხარჯი, დაზღვევა (მაგალითად, მივლინების დროს), საკანცელარიო ხარჯი, სხვ.

განვიხილოთ ბიუჯეტის ფორმირების ამოცანა. ამოცანის გამარტივებისთვის გავაერთიანოთ პირდაპირი და ირიბი ხარჯები - ხარჯების კატეგორიად.

ვარიანტი 1. აღმავალი მიდგომა - ცნობილია პროექტის ბიუჯეტი

**სცენარი:**

1. პროექტის ბიუჯეტის ფორმირების დიალოგური ფორმაა (Form_Xarji), სადაც:

2. ველში პროექტის ბიუჯეტი (Textbox_budjet) მიეთითება ბიუჯეტისათვის განკუთვნილი თანხა

3. ხარჯების ცხრილში (DataGrid_Xarji) მიეთითება - ხარჯების კატეგორია (მაგ., ადამიანური რესურსები) და ხარჯი

4. ცხრილი - DataGrid_Xarji შედგება 3 სვეტისაგან (Column-col_kategoria, col_xarji, col_button). აქედან, col_button სვეტის ტიპია DataGridViewButtonColumn (ღილაკის ტიპი, რომლის საშუალებითაც შესაძლებელია ოპერაციების გამოძახება/შეს-რულება);

5. ხარჯების კატეგორია შესაძლებელია შეიცავდეს ქვე-ხარჯებს (მაგალითად, ადამიანური რესურსებია ბიზნეს-ანალიტიკოსი, პროგრამისტი და სხვ. განსხვავებული სახელფასო თანხით). col_button ღილაკი გამოიყენება ხარჯების ქვეკატეგორიის თანხების ჯამის შესავსებად და გამოსათვლელად შემდეგი პრინციპით:

6. ხარჯების ქვეკატეგორიის ფორმირების დიალოგური ფორმაა (Form_SubXarji):

6.1. თუ ხარჯების კატეგორია განშტოვდება ქვე - ხარჯებად, მაშინ იხსნება დიალოგური ფორმა - Form_SubXarji, იმავე მართვის ობიექტებით, სადაც ცხრილში (DataGrid_SubXarji) მოხდება შესაბამის ხარჯის ქვე - კატეგორიის შევსება.

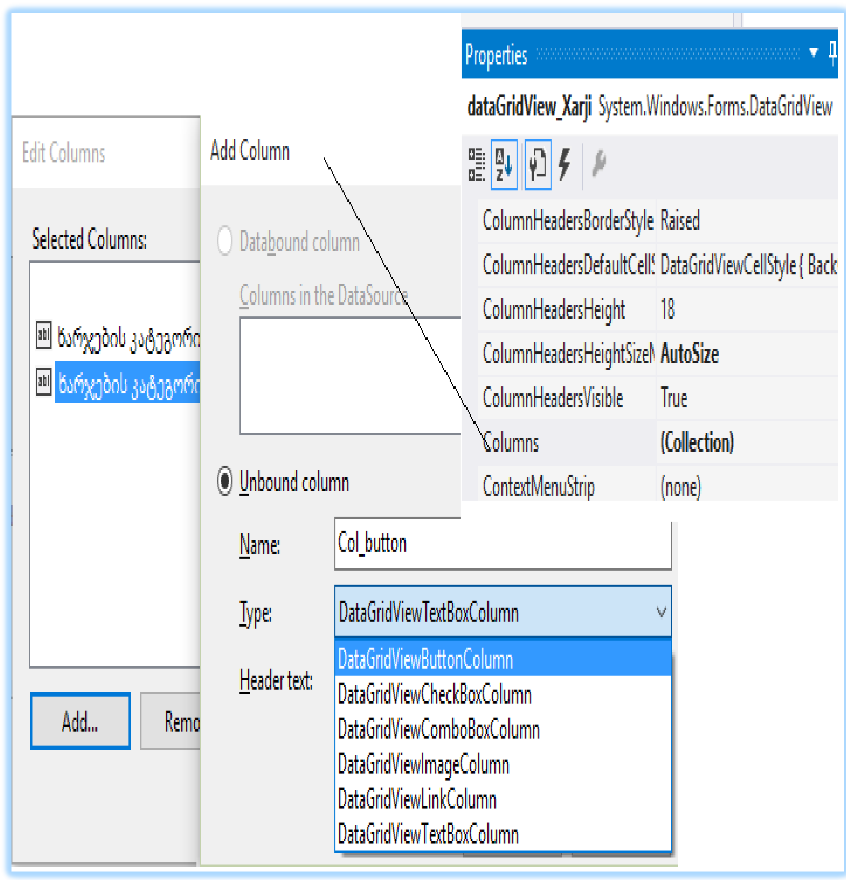
6.2. ცხრილი - DataGrid_SubXarji შედგება 2 ველისაგან ხარჯების ქვეკატეგორია (Col_SubKategoria) და თანხა (Col_ SubTanxa)

6.3. დიალოგურ ფორმაში Form_SubXarji, ცხრილის - DataGrid_SubXarji, ველის Col_SubTanxa - ჯამი იკრიბება და გადმოეცემა დიალოგურ ფორმას Form_Xarji. ქვეკატეგორიის ხარჯების ჯამი თავსდება DataGrid_Xarji-ის შესაბამის ველში, საიდანაც გამოძახებულ იქნა დამატებითი დიალოგური ფორმა.

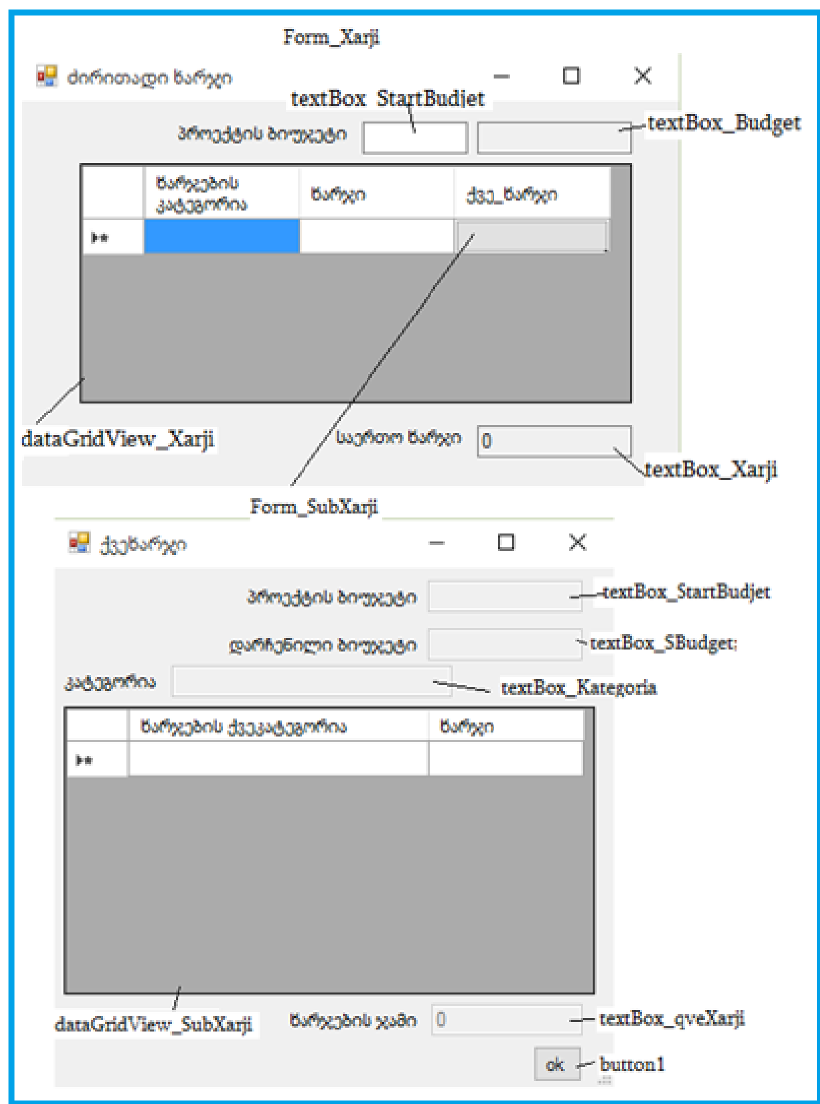
7. ძირითად ფორმაზე - Form_Xarji, ველში Textbox_Xarji, ჩაიწერება საერთო ხარჯების ჯამი დინამიკურად ცხრილის DataGrid_Xarji ველიდან Col_Tanxa, რაც ასევე დინამიკურად დააკლდება ველს Textbox_budjet.

DataGridView ცხრილში DataGridViewButtonColumn სვეტის შექმნის მაგალითი წარმოდგენილია 15.1 სურათზე.

დიალოგური ფორმების სტრუქტურა, პროგრამულ კოდში გამოყენებული მართვის ობიექტების დასახელებებით ნაჩვენებია 15.2 სურათზე.




სურ.15.1. DataGridView ცხრილში DataGridViewButtonColumn სვეტის შექმნის მაგალითი




სურ.15.2. დიალოგური ფორმების სტრუქტურა, პროგრამულ კოდში გამოყენებული მართვის ობიექტების დასახელებებით

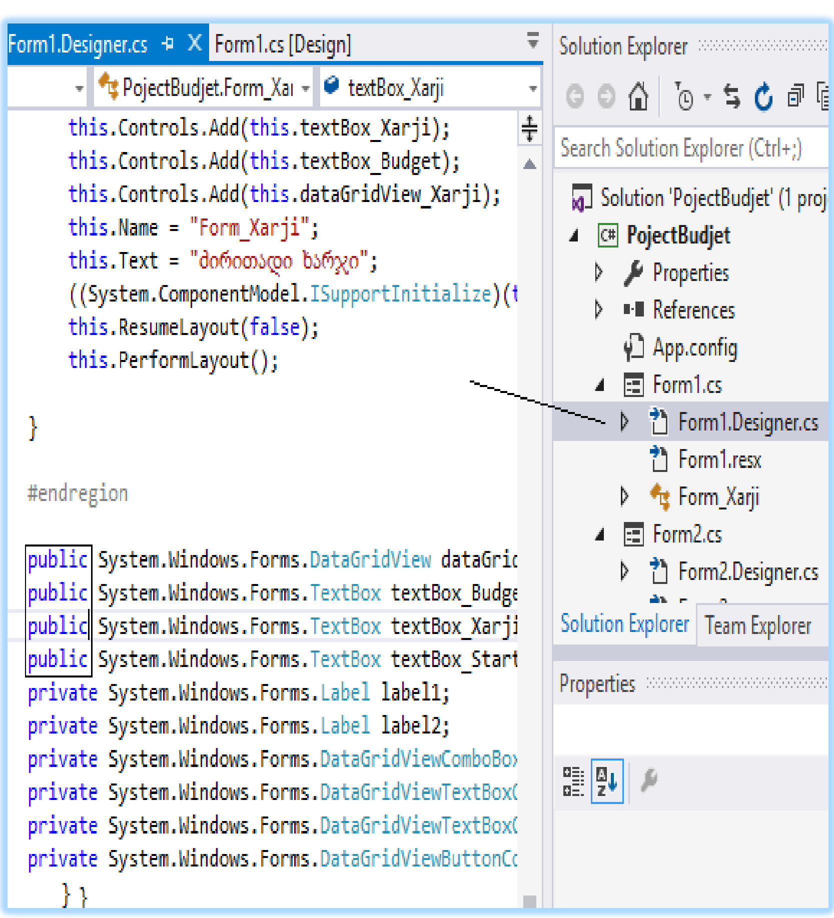
ამოცანის სარეალიზაციოდ ვხსნით 3 კლასს.

1. ძირითადი დიალოგური ფორმა - Form_Xarji
2. დამატებითი დიალოგური ფორმა - Form_SubXarji
3. დამხმარე კლასი gamotvlebi.cs, სადაც ვწერთ ერთ საერთო მეთოდს ორივე დიალოგურ ფორმაში გამოსაყენებლად.

ობიექტ - ორიენტირებული დაპროგრამების პრინციპებიდან გამომდინარე შესაძლებელია ერთი კლასის ობიექტებზე მანიპულაცია მეორე კლასიდან. მოცემულ მაგალითში გამოიყენება ერთი დიალოგიდან, მეორე დიალოგის გამოძახება, პირველი დიალოგის მონაცემის გადაცემა მეორე დიალოგში, მეორე დიალოგში მონაცემის დამუშავება და უკან დაბრუნება პირველ დიალოგში.

ამისათვის ხდება შემდეგი ოპერაციების ჩატარება:

1. იმ ობიექტს, რომელიც უნდა მანიპულირდეს მეორე კლასიდან, პირველ კლასში მიენიჭება ღია (Public) სტატუსი. დიალოგური ფორმის კლასის შემთხვევაში ობიექტზე Public სტატუსის მინიჭება ხდება -designers.rs ნაწილიდან, რომელიც ნებისმიერი დიალოგური ფორმის კომპლექტში შედის, ფორმის მართვის ელემენტების აღწერისათვის (სურ.15.3);
2. მეორე კლასს, რომლის გამოძახებაც ხდება პირველი კლასიდან პარამეტრად გადაეცემა პირველი კლასის ფორმის ობიექტი (შესაძლებელია ჩაიწეროს - this). მეორე კლასი მოითხოვს ამისათვის პარამეტრს პირველი კლასის პარამეტრის მითითებას (სურ.15.4).
3. აღწერილი ოპერაციების წარმატებით განხორციელებისას, მეორე კლასი ხედავს პირველი კლასის ყველა ღია ობიექტს.




სურ. 15.3. დიალოგური ფორმის ობიექტზე Public სტატუსის მინიჭება

```
private void dataGridView_Xarji_CellClick(object sender, DataGridViewCellEventArgs e)
{  if (e.ColumnIndex == 3) {
 String kategoria = dataGridView_Xarji.Rows[e.RowIndex].Cells[1].Value.ToString();
 Form_SubXarji subxarji = new Form_SubXarji(this, kategoria, e.RowIndex);
 subxarji.ShowDialog();
}     }
```

პირველი კლასი

```
public partial class Form_SubXarji : Form
{
    gamotvlebi gamotvla = new gamotvlebi();
    public int row_indexs;
    public string  darchenili;
    private Form_Xarji ziritadia;

     public Form_SubXarji(Form_Xarji ziritadi,  string kategoria, int row_index)
    {
        InitializeComponent();
        textBox_Kategoria.Text = kategoria;
        this.row_indexs = row_index;
        ziritadia = ziritadi;
        this.darchenili = ziritadia.textBox_Budget.Text;
        textBox_StartBudjet.Text = ziritadia.textBox_StartBudjet.Text;
        textBox_SBudget.Text = ziritadia.textBox_Budget.Text;
    }
```

მეორე კლასი

სურ.15.4. კლასის ობიექტის პარამეტრად გადაცემა სხვა კლასში და მისი გამოყენება

კოდის ლისტინგი:

1. ძირითადი დიალოგური ფორმა - Form_Xarji

```
namespace PojectBudjet
{
   public partial class Form_Xarji : Form
   {
        public string tanxa_sub = ""; // Form_SubXarji - ფორმიდან
ქვეხარჯის მონაცემის მიღებისათვის
       public String kategoria = "";// Form_SubXarji - ფორმაზე მონაცემის
გადაცემისთვის
       public int row_indexs = 0;// Form_SubXarji - ფორმაზე ცხრილის
იმ ჩანაწერის ნომრის გადაცემისთვის, სადაც უნდა მოთავსდეს
დაბრუნებული მონაცემი

       public Form_Xarji()
       {
          InitializeComponent();
       }

       private void sumBudjet() //გამოთვლის მეთოდი.
     //ამ მეთოდით ხდება სასტარტო ბიუჯეტისა და ხარჯის
დინამიკური გამოკლება
       {
          try// გამორიცხვის ოპერაცია
          {
             if (textBox_StartBudjet.Text != "")
             {
                Double startBudjet =
Double.Parse(textBox_StartBudjet.Text);
//ტექსტური ველის რიცხობრივ ველად გადაკეთება
ანგარიშისათვის
```

```
                Double xarji = Double.Parse(textBox_Xarji.Text);
//ტექსტური ველის რიცხობრივ ველად გადაკეთება
ანგარიშისათვის
                textBox_Budget.Text = (startBudjet - xarji).ToString();
            }
        }
        catch (FormatException) // გამორიცხვის ოპერაცია, რაც
გამოსათვლელ ველებში რიცხვისა და სიმბოლოს მითითებას
აკონტროლებს
        {
            MessageBox.Show("only nummber");
        }
    }

    private void dataGridView_Xarji_CellValueChanged(object sender,
DataGridViewCellEventArgs e)
    //თავსდება ავტომატურად ამ ფორმის დიზაინში ცხილის
მოვლენებში CellValueChange 2-ჯერ მაუსის დაწკაპუნებით
    {
        gamotvlebi gamotvla = new gamotvlebi();// დამხმარე კლასის
ობიექტის შექმნა

                if (e.ColumnIndex == 2) // მე-2 ველი არის ხარჯის ველი
ცხრილში
                {
            textBox_Xarji.Text =
gamotvla.CellSum(dataGridView_Xarji,2).ToString(); //დამხმარე
კლასის მეთოდს გადაეცემა 2 პარამეტრი - ცხრილის ობიექტი და
სვეტის ნომერი
            sumBudjet();//გამოთვლის მეთოდის გამოძახება
```

```
        }
    }

    public void cxrilis_shevseba(string tanxa_sub, int rowid, ComboBox
fill_subkategory)
    {
//ეს მეთოდი ავსებს სხვა ქვეხარჯების ფორმიდან მონაცემების
ჩამატებას ცხრილის ხარჯების ველში
        dataGridView_Xarji.Rows[rowid].Cells[2].Value = tanxa_sub;
    }

    private void dataGridView_Xarji_CellClick(object sender,
DataGridViewCellEventArgs e)// ცხრილში ღილაკის ველზე დაჭერის
ოპერაცია
    {
        // თავსდება ავტომატურად ამ ფორმის დიზაინში ცხილის
მოვლენებში CellClick 2-ჯერ მაუსის დაწკაპუნებით

        if (e.ColumnIndex == 3) // მე-3 ველი არის ღილაკის ველი
ცხრილში
        {
            try// გამორიცხვის ოპერაცია
            {
 kategoria =
dataGridView_Xarji.Rows[e.RowIndex].Cells[1].Value.ToString();
                row_indexs = e.RowIndex;
 Form_SubXarji subxarji = new Form_SubXarji(this);
// ქვეკატეგორიის დიალოგური ფორმის ობიექტის შექმნა
                subxarji.ShowDialog();// ქვეკატეგორიის დიალოგური
ფორმის გამოძახება
```

```
            }
            catch// გამორიცხვის ოპერაცია, აკონტროლებს ცხრილში
ხარჯვის კატეგორიის ველის შევსებას
            { MessageBox.Show("შეავსეთ ცხრილში ხარჯვის
კატეგორიის ველი"); }
          }
        }

        private void textBox_StartBudjet_TextChanged(object sender,
EventArgs e)
        {
            sumBudjet();//გამოთვლის მეთოდის გამოძახება
        }
    }
}
```

2. დამატებითი დიალოგური ფორმა - Form_SubXarji

```
namespace PojectBudjet
{
    public partial class Form_SubXarji : Form
    {
        gamotvlebi gamotvla = new gamotvlebi();
        public int row_indexs; // პირველი კლასიდან გადმოცემული
ცვლადისთვის
        public string darchenili;// პირველი კლასიდან გადმოცემული
ცვლადისთვის
        private Form_Xarji ziritadia;// პირველი კლასის ობიექტის შექმნა

        public Form_SubXarji(Form_Xarji ziritadi)
        {
```

```
InitializeComponent();
ziritadia = ziritadi;// პირველი კლასის ობიექტზე
გადმოცემული კლასის ობიექტის მინიჭება

// პირველი კლასიდან გადმოცემული მონაცემის მინიჭება მეორე
კლასის ობიექტზე----------
    textBox_Kategoria.Text = ziritadia.kategoria;
    this.row_indexs = ziritadia.row_indexs;
    this.darchenili = ziritadia.textBox_Budget.Text;
    textBox_StartBudjet.Text = ziritadia.textBox_StartBudjet.Text;
    textBox_SBudget.Text = ziritadia.textBox_Budget.Text;
//--------პირველი კლასიდან გადმოცემული მონაცემის მინიჭება
მეორე კლასის ობიექტზე
  }

  private void dataGridView_SubXarji_CellValueChanged(object
sender, DataGridViewCellEventArgs e)
  {
        if (e.ColumnIndex == 1) // 1 ველი არის ხარჯის ველი
ცხრილში
    {
  textBox_qveXarji.Text = gamotvla.CellSum(dataGridView_SubXarji,
1).ToString();
      if (textBox_SBudget.Text != "")
      {
   Double darchBudjet = Double.Parse(textBox_SBudget.Text);
   Double subxarji = Double.Parse(textBox_qveXarji.Text);
   textBox_SBudget.Text = (Double.Parse(darchenili) -
subxarji).ToString();
      }
```

```
        }
      }

          private void button1_Click(object sender, EventArgs e)
      {

          ziritadia.cxrilis_shevseba(textBox_qveXarji.Text, row_indexs);
// პირველი კლასის მეთოდის გამოძახება. გადმოცემული ცხრილის ჩანაწერის ნომრის მიხედვით ქვეხარჯის გადაცემა პირველ კლასზე
      }
    }
}
```

3. დამხმარე კლასი gamotvlebi.cs

```
using System.Windows.Forms; //ვამატებთ დიალოგური ფორმის მართვის ელემენტების გამოსაჩენად
namespace PojectBudjet
{
  public class gamotvlebi
   {
      public double CellSum(DataGridView dataGridView_, int cel_n)
      {
          double sum = 0;
          try
          {
              for (int i = 0; i < dataGridView_.Rows.Count; ++i)
              {
                  double dxarji = 0;
```

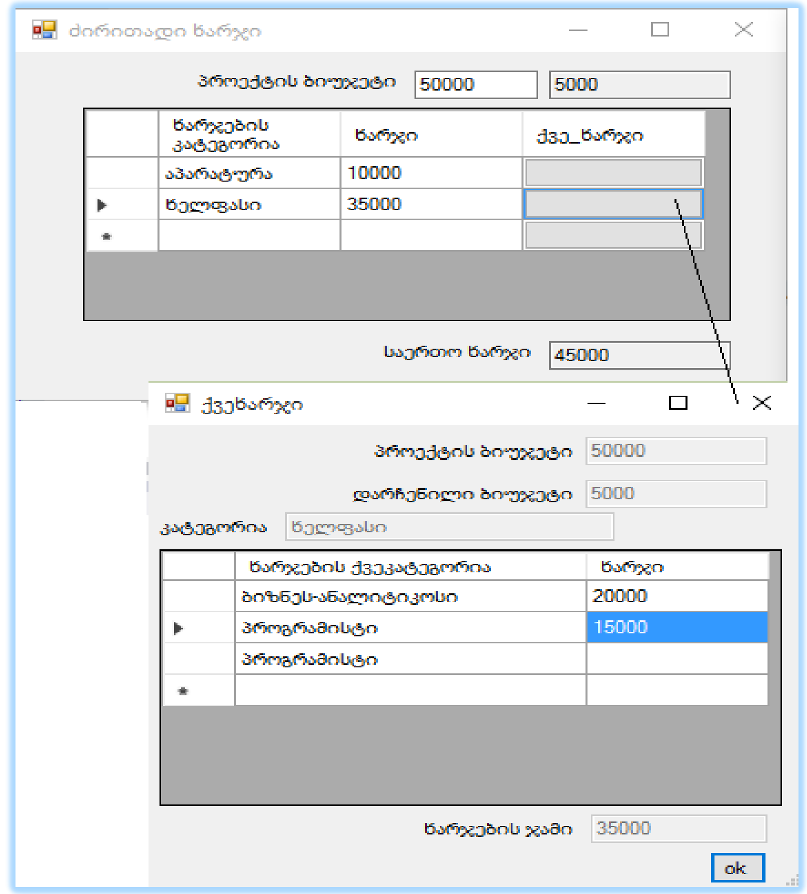
```
dxarji =
Double.Parse(dataGridView_.Rows[i].Cells[cel_n].Value.ToString());
                sum += dxarji;
             }

          }
          catch
          {
          }
          return sum;
       }
```

შედეგი მოცემულია 13.5 ნახაზზე.

**დავალება:**

ააგეთ მოცემული მაგალითის მიხედვით პროექტის ბიუჯეტის ფორმირების პროგრამული კოდი , ბიუჯეტის ფინანსური მართვის მე-2 ვარიანტისთვის -

პროექტის ღირებულება არაა ცნობილი. პროექტის ღირებულება შეადგინეთ რესურსებზე ხარჯის დათვლით (დაღმავალი მიდგომა).




სურ.13.5. პროგრამის მუშაობის შედეგი

## 2.14. პროგრამული სისტემების შექმნის დავალებათა კალენდარული გეგმის ფორმირება

### ლაბორატორიული სამუშაო N 16

**მიზანი:** dataGridView ცხრილისა და dateTimePicker თარიღის მართვის ელემენტებთან მუშაობა; DateTime და dataGridView კლასის სხვადასხვა მეთოდის გამოყენების შესწავლა.

**ამოცანა:** პროექტების მართვის სისტემისათვის დავალებათა კალენდარული გეგმის ფორმირება Gantt Chart მოდელის ბაზაზე:

1. კალენდარული გეგმისათვის ცხრილის სვეტების ფორმირება
2. პროექტის დავალებათა გადანაწილება ცხრილში

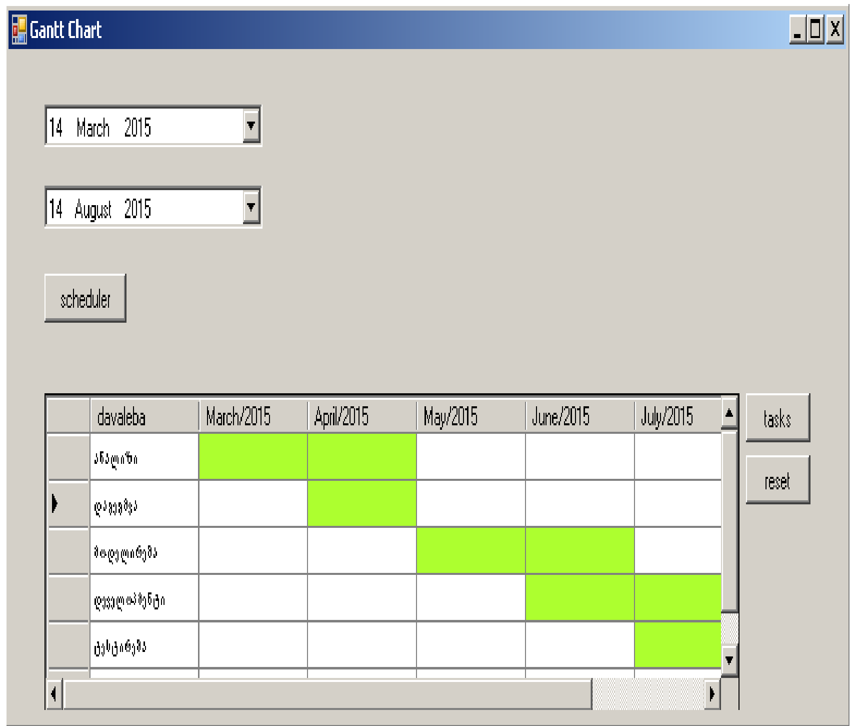
ორ dateTimePicker (საწისი და საბოლოო თარიღი) - ობიექტიდან არჩეული თარიღების მიხედვით dataGridView ცხრილზე თარიღების გადანაწილებული ასახვა სვეტების სახით:

მაგალითად, დავთვალოთ ორ არჩეულ თარიღს შორის არსებული თვეები, რომელთა რაოდენობის მიხედვით dataGridView ცხრილზე დავამატებთ FOR ციკლით სვეტებს დინამიკურად. ასევე, დინამიკურად მივანიჭოთ სვეტებს თვეების დასახელება მიმდევრობით და გადავანაწილოთ დავალებები პერიოდების მიხედვით.

დავუშვათ მოცემულია 6 თვიანი პროექტი 5 საფეხურიანი ფაზით, რომელიც სრულდება მარტიდან-აგვისტოს ჩათვლით: ანალიზი, დაგეგმვა, მოდელირება, დეველოპმენტი და ტესტირება. ფაზების დამუშავების პერიოდის გეგმაა: ანალიზი -2 თვე, დაგეგმვა შემდგომი ეტაპი 1 თვე, ანალიზის დამუშავების შუა ეტაპიდან, მოდელირება 2 თვე დაგეგმვის დასრულებიდან, დეველოპმენტისათვის საჭიროა 3 თვე და ამასთან ერთად დავალებაში გარკვევისა და მისი დახვეწისათვის საჭიროა 1 თვე

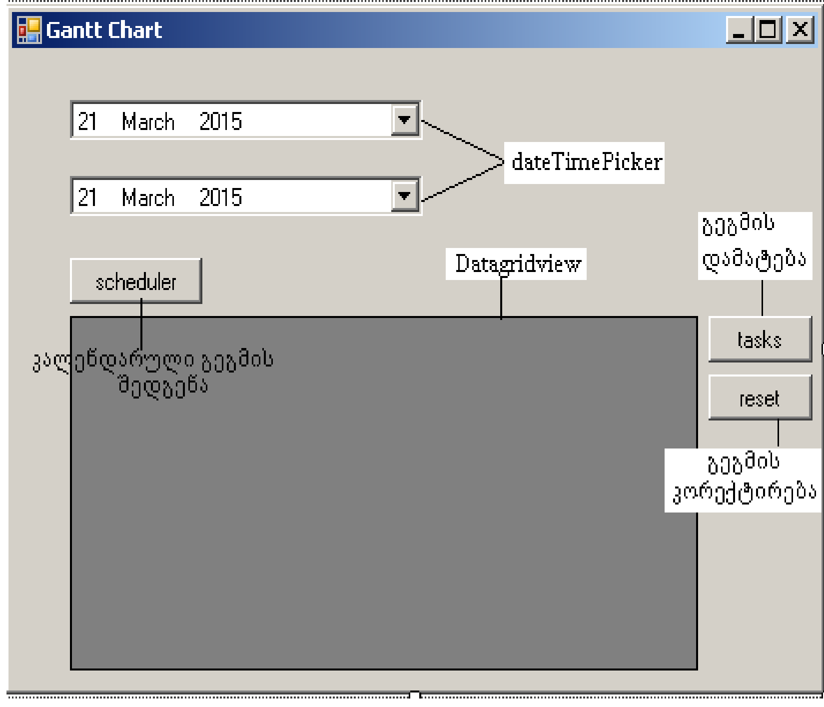
ანალიტიკოსებისა და პროგრამისტების ერთობლივი მუშაობისათვის.

დავალების შესრულების შედეგი ნაჩვენებია 16.1 ნახაზზე.




**ნახ.16. 1**

დიალოგურ ფორმაზე მმართველი ობიექტები შემდეგნაირია (ნახ.16. 2):




ნახ.16. 2

განვიხილოთ ობიექტებზე მუშაობა:

**dataGridView ცხრილის მართვის ელემენტზე DataGridViewTextBoxColumn სვეტის** პროგრამულად დამატება:

```
DataGridViewTextBoxColumn column_name= new
DataGridViewTextBoxColumn();
 //ცხრილში TextBox სვეტის ობიექტის შექმნა

DataGridViewCellStyle dataGridViewCellStyle_p  =new
DataGridViewCellStyle();
```

```
//ცხრილის უჯრედის სტილის ობიექტის შექმნა

column_column_name.Width = 100; // ცხრილის უჯრედის სიგანის
                                 // განსაზღვრა
column_name.HeaderText = „tarigi“;  //ცხრილის სვეტის სახელის
                                     // დარქმევა
column_name.Name = „tarigi“;  //ცხრილის სვეტის კოდური სახელის
                              // დარქმევა
dataGridView1.Columns.Add(column_name.); //ცხრილის სვეტის
                                        //დამატება ცხრილზე
```

- **DateTime კლასით თარიღების განსაზღვრა:**

```
DateTime ad_days = dateTimePicker1.Value.AddMonths(1);
   //dateTimePicker1 ობიექტიდან DateTime კლასის ობიექტზე
   // „თვეების“ რაოდენობის გადაცემა
String twe_S =
Thread.CurrentThread.CurrentCulture.DateTimeFormat.MonthNames[a
d_days.Month - 1];
      // თვეების სახელების გაფორმება      -  საჭიროა using
      // System.Threading; კლასის გამოყენება

 DateTime startDate = (DateTime) dateTimePicker1.Value;
 DateTime endDate = (DateTime)enddata.Value;
    // DateTime კლასზე dateTimePicker ობიექტის დაყვანა

 TimeSpan ts = endDate.Subtract(startDate);
    // TimeSpan- დროის ინტერვალი, Subtract- ორ თარიღს შორის
    // სხვაობის გამოთვლა
int days = ts.Days;
   // TimeSpan ობიექტით დღეების განსაზღვრის მეთოდი
```

```
    int Week = days / 7;
// ორ დროის ინტერვალს შორის კვირის რაოდენობის დათვლა
```

მაგალითები:

1. **ორ თარიღს შორის თვეების რაოდენობის განსაზღვრის მეთოდი**

```
  public int monthDifference(DateTimePicker startDate, DateTimePicker
endDate)
      {
         int months_raod = 12 * (startDate.Value.Year -
endDate.Value.Year) +
         startDate.Value.Month - endDate.Value.Month;

         return Math.Abs(months_raod); // აბრუნებს აბსოლუტურ
მნიშვნელობას
      }
```

2. **ორ თარიღს შორის წლის რაოდენობის განსაზღვრის მეთოდი**

```
      public static int yearDifference(DateTimePicker startDate,
DateTimePicker endDate)
      {
         int months_raod = 12 * (startDate.Value.Year -
endDate.Value.Year) +
         startDate.Value.Month - endDate.Value.Month;
         int years = months_raod / 12;
         return Math.Abs(years); // აბრუნებს აბსოლუტურ
მნიშვნელობას
      }
```

3. **საწყისი და საბოლოო თარიღების მიხედვით თვეების რაოდენობის განსაზღვრა და ცხრილზე თარიღების გადანაწილებული ასახვა სვეტების სახით:**

```
DataGridViewCellStyle dataGridViewCellStyle_p;
DataGridViewTextBoxColumn column_period_all;
private void button_scheduler_Click(object sender, EventArgs e)
      {
         dataGridView1.Columns.Clear();
         int month_caunt = statics.monthDifference(dateTimePicker1,
                        dateTimePicker2);
         for (int i = 0; i < month_caunt + 1; i++)
          {
            dataGridViewCellStyle_p = new DataGridViewCellStyle();
            column_period_all = new DataGridViewTextBoxColumn();
            column_period_all.Width = 100;
            dataGridViewCellStyle_p.Format = "N2";
             column_period_all.ReadOnly = true;
            DateTime ad_days = dateTimePicker1.Value.AddMonths(i);
            String[] days_data_list =token_data(ad_days.ToString());
            String days_data = days_data_list[0];
            int year = ad_days.Year;
      String twe_S =
      Thread.CurrentThread.CurrentCulture.DateTimeFormat.Month
           Names[ad_days.Month - 1];
              String tarigi = twe_S + "/" + year;
              column_period_all.HeaderText = tarigi;
              column_period_all.Name = tarigi;
              dataGridView1.Columns.Add(column_period_all);
            }
         }
```

```
public String[] token_data(String text)
      {
         char[] seps = { ' ' };
         String[] values = text.Split(seps);

         return values;
      }
  public int monthDifference(DateTimePicker startDate, 
DateTimePicker endDate)
      {
         int monthsApart = 12 * (startDate.Value.Year - 
endDate.Value.Year) +
         startDate.Value.Month - endDate.Value.Month;
         return Math.Abs(monthsApart);
      }
      public static int yearDifference(DateTimePicker startDate, 
DateTimePicker endDate)
      {
         int monthsApart = 12 * (startDate.Value.Year - 
endDate.Value.Year) +
         startDate.Value.Month - endDate.Value.Month;
         int years = monthsApart / 12;
         return Math.Abs(years);
      }
```

**კოდის ლისტინგი:**

**1. დამხმარე კლასი მეთოდებთან სამუშაოდ**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Forms;
```

```
namespace WindowsFormsApplication_IT_Project
{
// სიმბოლოების დაშლის მეთოდი
 class Class_statistic
  {
    public String[] token_data(String text)
      {
         char[] seps = { ' ' };
         String[] values = text.Split(seps);
         return values;
      }
// სიმბოლოების დაშლის მეთოდი
public int monthDifference(DateTimePicker
           startDate, DateTimePicker endDate)
 {
   int monthsApart = 12 * (startDate.Value.Year -
       endDate.Value.Year) +
       startDate.Value.Month - endDate.Value.Month;

   return Math.Abs(monthsApart);
 }

public static int yearDifference(DateTimePicker
        startDate, DateTimePicker endDate)
 {
   int monthsApart = 12 * (startDate.Value.Year -
      endDate.Value.Year) +
      startDate.Value.Month - endDate.Value.Month;
   int years = monthsApart / 12;
  return Math.Abs(years);
 }
public String rezult_data_string(DateTimePicker
      startdata, DateTimePicker enddata)
 {
```

```
String rezult_data = "";
int year = enddata.Value.Year -
       startdata.Value.Year;
int month = monthDifference(startdata, enddata);
int years = yearDifference(startdata, enddata);

DateTime startDate = (DateTime)startdata.Value;
DateTime endDate = (DateTime)enddata.Value;
TimeSpan ts = endDate.Subtract(startDate);
String aaas = ts.ToString();
String ddd = ts.Days.ToString();
int days = ts.Days;
int Week = days / 7;
String monat = "";
int monat_all = Math.Abs(years * 12 - month);
if (year != 0)
 {
   if (month > 11)
     {
       rezult_data = "";
       rezult_data = years + "წელი";
     }
   if (startdata.Value.Month !=
                 enddata.Value.Month)
     {
       rezult_data = "";
       monat = (enddata.Value.Month -
              startdata.Value.Month).ToString();
       rezult_data = years + " წელი, " +
                monat_all + " თვე";
     }
   if (startdata.Value.Day !=
                 enddata.Value.Day)
     {
```

```
            rezult_data = "";
            String Day = (enddata.Value.Day -
                   startdata.Value.Day).ToString();
            rezult_data = years + " წელი, " +
                monat_all + " თვე, " + Day + " დღე";
          }
        }
      if (year == 0 & month != 0)
       {
         rezult_data = "";
         monat = (enddata.Value.Month -
              startdata.Value.Month).ToString();
         rezult_data = monat_all + " თვე";
         if (startdata.Value.Day !=
                       enddata.Value.Day)
           {
            rezult_data = "";
            String Day = (enddata.Value.Day -
                 startdata.Value.Day).ToString();
            rezult_data = monat_all + " თვე, " + Day
                      + " დღე";
           }
       }
      if (year == 0 & month == 0 & days != 0)
       {
         rezult_data = "";
         String Day = (enddata.Value.Day -
              startdata.Value.Day).ToString();
         rezult_data = Day + " დღე";
       }
       return rezult_data;
    }
  }
}
```

## 2. დიალოგური ფორმის კლასი

```
// ---- ლისტინგი -------------------
sing System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Threading;

namespace WindowsFormsApplication_IT_Project
{
 public partial class Form1 : Form
 {
  public Form1()
   {
    InitializeComponent();
   }
  DataGridViewCellStyle dataGridViewCellStyle_p;
  DataGridViewTextBoxColumn column_period_all;
  DataGridViewTextBoxColumn tasks;
  Class_statistic statics = new Class_statistic();

//დამხმარე კლასის ობიექტის შექმნა
//   ========კალენდარული გეგმის შედგენა=============
private void button_scheduler_Click(object sender,
       EventArgs e)
 {
  DateTime startDate =
          (DateTime)dateTimePicker1.Value;
  DateTime endDate =
          (DateTime)dateTimePicker2.Value;
```

```
TimeSpan ts = endDate.Subtract(startDate);
String aaas = ts.ToString();
String ddd = ts.Days.ToString();
int days = ts.Days;
int Week = days / 7;

string monat = (dateTimePicker2.Value.Month -
        dateTimePicker1.Value.Month).ToString();
dataGridView1.Columns.Clear();
tasks = new DataGridViewTextBoxColumn();
tasks.Width = 100;
tasks.HeaderText = "davaleba";
tasks.Name = "davaleba";
dataGridView1.Columns.Add(tasks);
int month_caunt =
     statics.monthDifference(dateTimePicker1,
            dateTimePicker2);
for (int i = 0; i < month_caunt + 1; i++)
 {
  dataGridViewCellStyle_p = new
              DataGridViewCellStyle();
  column_period_all = new
              DataGridViewTextBoxColumn();
  column_period_all.Width = 100;

  dataGridViewCellStyle_p.Format = "N2";

  column_period_all.ReadOnly = true;
  DateTime ad_days =
        dateTimePicker1.Value.AddMonths(i);
  String[] days_data_list =
      statics.token_data(ad_days.ToString());
  String days_data = days_data_list[0];
  int year = ad_days.Year;
```

```
    String twe_S =
         Thread.CurrentThread.CurrentCulture.
       DateTimeFormat.MonthNames[ad_days.Month - 1];
    String tarigi = twe_S + "/" + year;
    column_period_all.HeaderText = tarigi;
    column_period_all.Name = tarigi;

    dataGridView1.Columns.Add(column_period_all);
   }
  }
// ========კალენდარული გეგმის შედგენა=============
// ============გეგმის დამატება===================
private void button_tasks_Click(object sender,
            EventArgs e)
 {
   foreach (DataGridViewCell cell in
              dataGridView1.SelectedCells)
      {
       dataGridView1.Rows[cell.RowIndex].
         Cells[cell.ColumnIndex].Style.BackColor =
              System.Drawing.Color.GreenYellow;
      }
 }

//  ============გეგმის დამატება===================
//  ============გეგმის კორექტირება===================

private void button_reset_Click(object sender,
          EventArgs e)
{
 foreach (DataGridViewCell cell in
                dataGridView1.SelectedCells)
  {
```

```
    dataGridView1.Rows[cell.RowIndex].
     Cells[cell.ColumnIndex].Style.BackColor =
          System.Drawing.Color.White;
     }
  }
// =============გეგმის კორექტირება====================
```

**დავალება:**

ორ dateTimePicker (საწისი და საბოლოო თარიღი) - ობიექტიდან არჩეული თარიღების მიხედვით dataGridView ცხრილზე გადანაწილებულად ასახეთ სვეტები წლებით 3 - წლიანი პროექტისათვის; კვირებით - 4 - კვირიანი პროექტისთვის; დღეებით 20 - დღიანი პროექტისათვის.

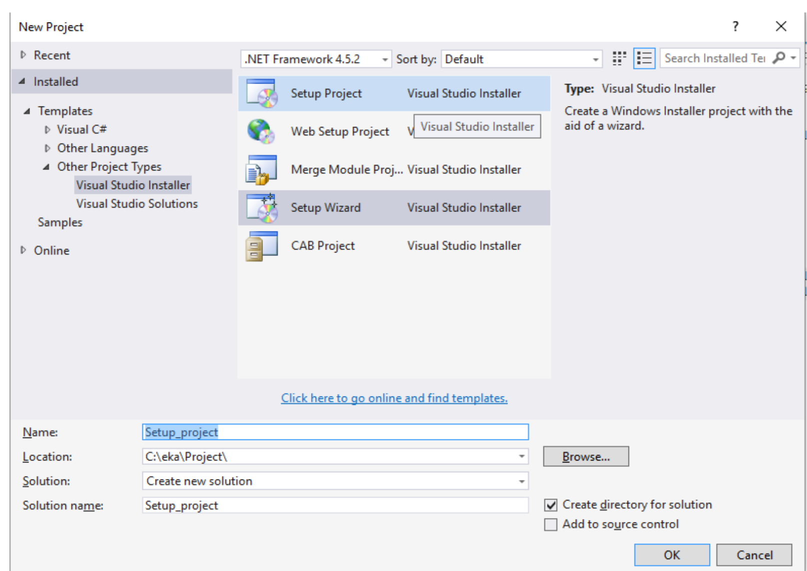
## 2.15. პროგრამული აპლიკაციის დისტრიბუციული ფაილის (საინსტალაციო პაკეტის) შექმნა

### ლაბორატორიული სამუშაო N 17

**მიზანი:** Setup და Deployment პროექტში რეალიზებული პროგრამული პაკეტის ინტეგრირება. რეალიზებული პროგრამული პაკეტის გამშვები (exe) ფაილის შექმნის შესწავლა.

**ამოცანა:** პროგრამული პაკეტის გამშვები ფაილის მომზადება

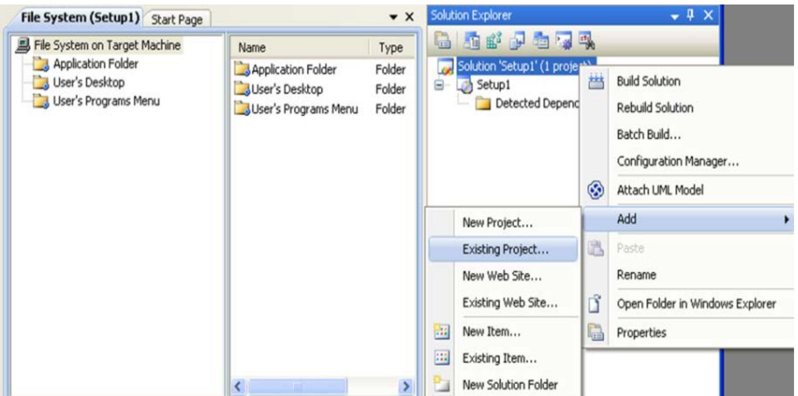
Visual Studio სისტემაში ახალი პროექტის შექმნის მენიუში Other Project Type-Visual Studio Installer პროგრამული პაკეტის გამშვები (exe) ფაილის შექმნისათვის (ნახ.17.1) არის ორი არჩევანი Setup Project და Setup Wizard. მიუხედავად ამისა, მათი ფუნქციონირება მცირედით განსხვავდება.



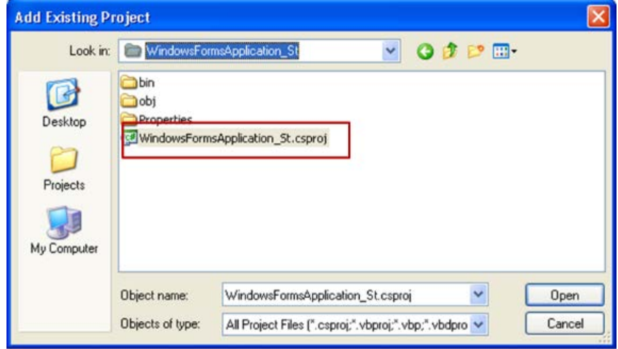

ნახ.17. 1

Setup Wizard საშუალებას იძლევა ბიჯური რეჟიმით შეიქმნას Web ან Windows ფორმის საინსტალაციო გარემო და ინტეგრირდეს სისტემისათვის თანამდევი ფაილებით. თუმცა გამშვები ფაილის მომზადების ბირთვის ავტომატიზებულ მექანიზმებს Setup Wizard საშუალება არ შეიცავს.

Setup Project ან Setup Wizard საშუალებებით პროექტის შექმნის შემდეგ, პროექტში უნდა ინტეგრირდეს რეალიზებული პროგრამული პაკეტი ფუნქციით Add – Existing Project (მაუსის მარჯვენა ღილაკი Solution Explorer ფანჯარაში. მიეთითება ფაილი გაფართოებით - .csproj (ნახ. 17.2, 17.3).

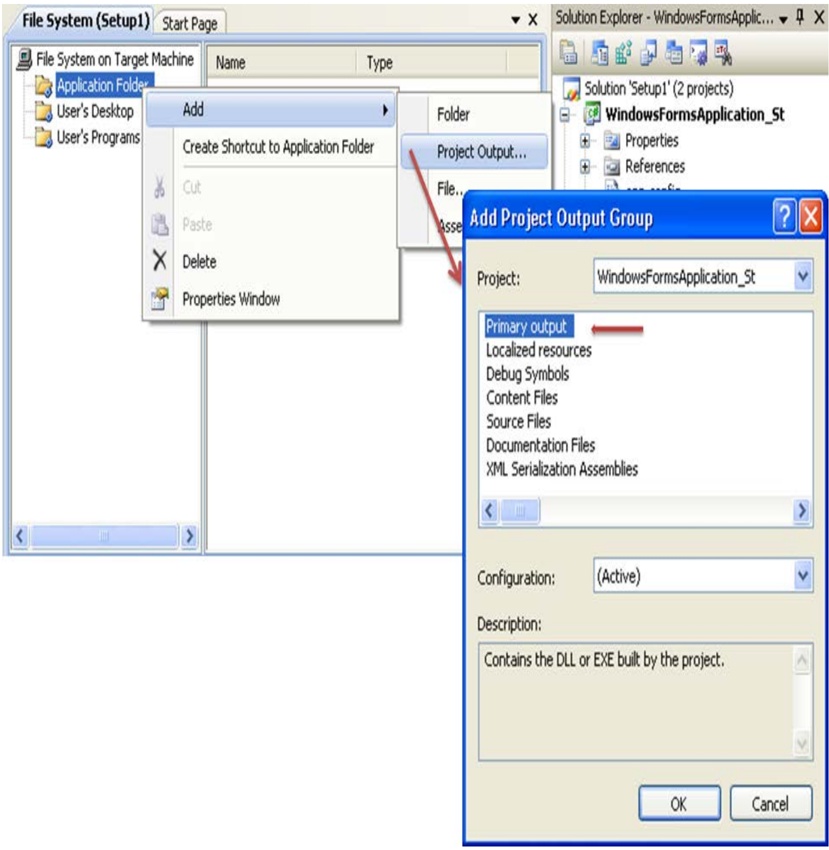



ნახ.17.2

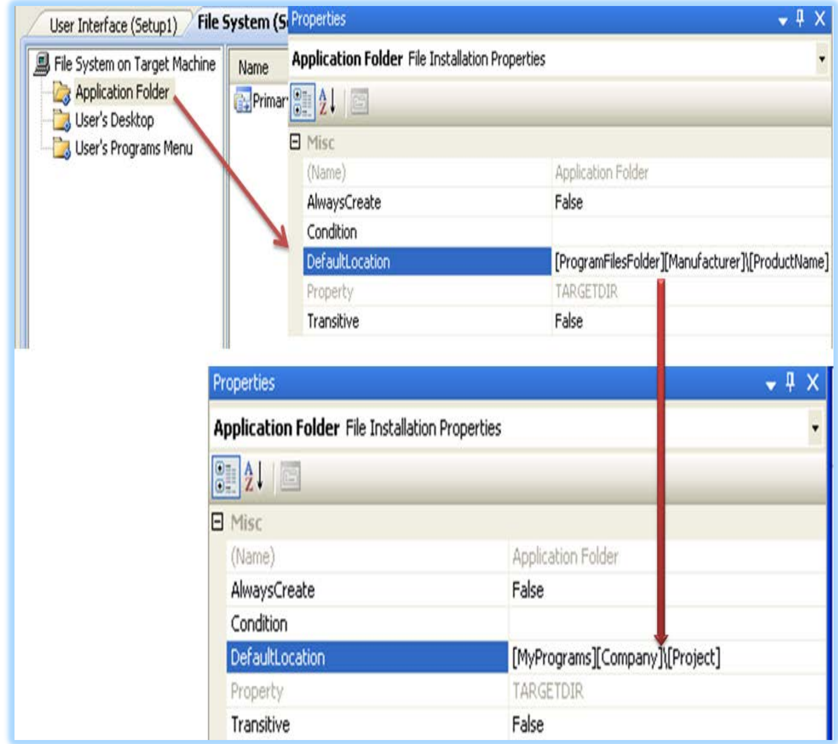



ნახ.17.3

პროექტის ნაწილში File System საქაღალდე Application Folder განკუთვნილია საინსტალაციო პაკეტისადთვის. ამ საქაღალდეზე ფუნქციით Add (მაუსის მარჯვენა ღილაკი) იხსნება პროექტის შედეგების დასაპროექტებელი ფანჯარა, სადაც რეკომენდებულია ბაზური შედეგის - Primer output მითითება (ნახ.17.4). Application Folder საქაღალდეს თვისებების ფანჯარაში შესაძლებელია დასაინსტალირებელი პაკეტის სახელის მითითებაც (ნახ.17.5)



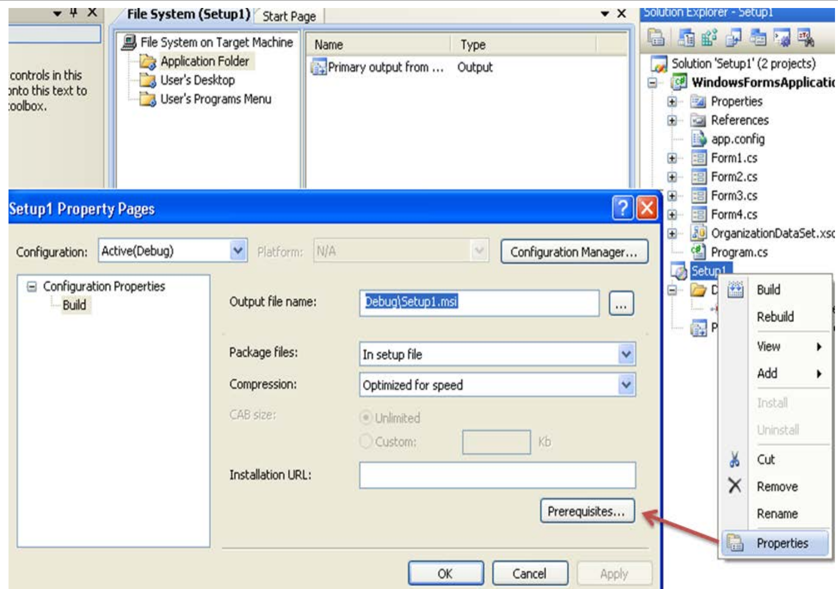

ნახ.17.4



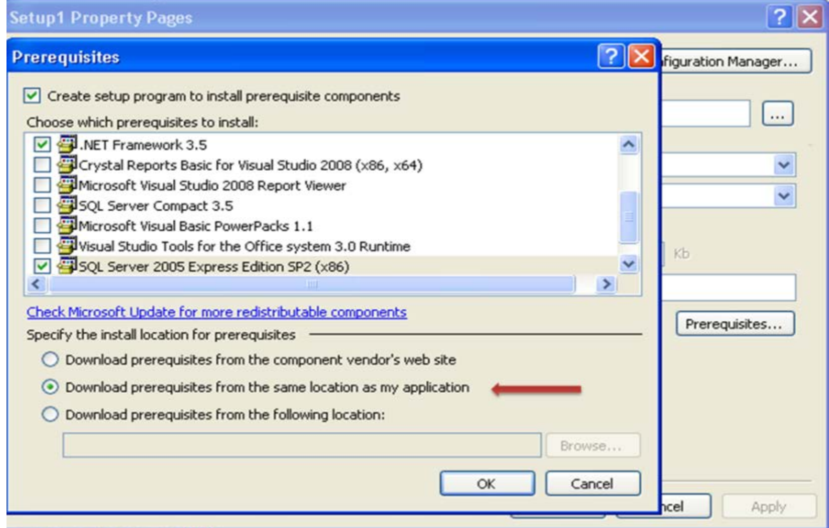

ნახ.17.5

სისტემის ინსტალირების დროს, ხშირად საჭიროა დამხმარე პროგრამების, კომპონენტების ან ბიბლიოთეკების დაყენებაც (მაგ., Windows Installer, .NET Framework, SQL Server Express და სხვ.).

პროექტის Solution Expoler-ის დიალოგურ ფანჯარაში რეალიზებული პროგრამული პაკეტის დამატების შემდეგ ჩნდება ორი პროექტი 1. მიმდინარე ანუ Setup პროექტი და 2. მიერთებული ანუ რაც უნდა დაინსტალირდეს. Setup პროექტზე თაგუნას მარჯვენა ღილაკის მეშვეობით ფუნქციაზე Property ღილაკით ხდება თვისებების დიალოგური ფორმის გამოძახება, სადაც ღილაკით Prerequisites იხსნება წინაპირობების მისათითებელი დიალოგური ფანჯარა (ნახ.17.6 და 17.7).



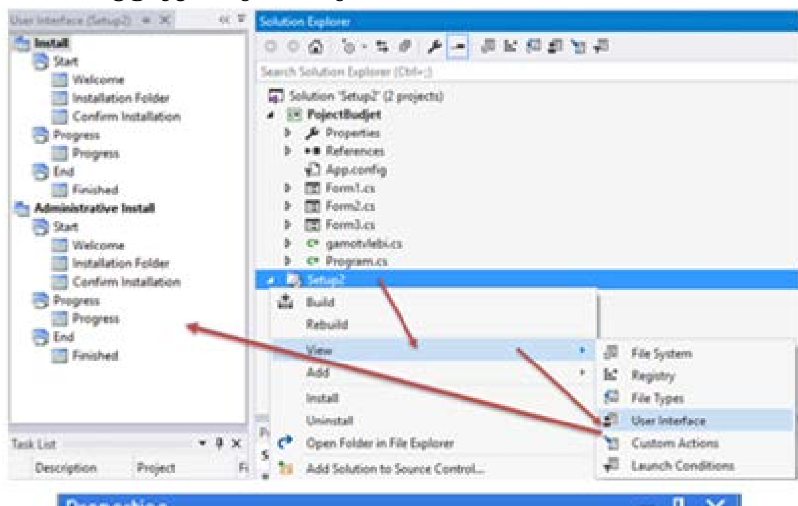

ნახ.17.6

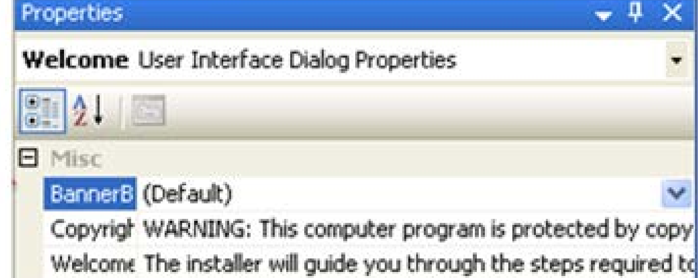



ნახ.17.7

პროექტის ნაწილში File System მოთავსებულია ასევე ორი დამატებითი საქაღალდე User’s Desktop (რა განთავსდეს მომხმარებლის სამუშაო მაგიდის საქაღლდეში) და User’s Programs Menu (რა განთავსდეს პროგრამული ჩამონათვალის საქაღალდეში).

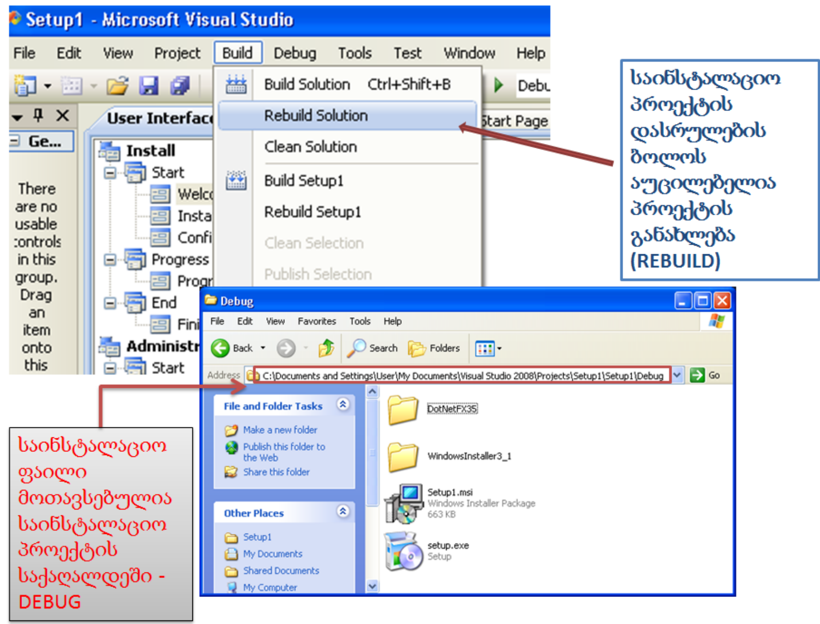
შესაძლებელია ინსტალაციის პროგრამის დიზაინის ცვლილება (ძირითადად, სურათებით/იკონებით გაფორმება) Setup პროექტზე თაგუნას მარჯვენა ღილაკის მეშვეობით View-User Interface ფუნქციის გამოძახება (ნახ.17.8).




ნახ.17.8

საბოლოოდ, აუცილებელია Setup პროექტის განახლება ფუნქციით Rebuild (ნახ.17.9). Setup.exe გამშვები ფაილი მოთავსებულია Setup პროექტის საქაღალდის ქვესაქაღალდეში - Debug.




ნახ.17.9

## ლიტერატურა:

1. სურგულაძე გ. თურქია ე. (2003). ბიზნესპროცესების მართვის ავტომატიზებული სისტემების დაპროექტება. მონოგრ., ISBN 99940-14-81-1. თბ. „ტექნიკური უნივერსიტეტი“.

2. თურქია ე. (2010). ბიზნეს–პროექტების მართვის ტექნოლოგიური პროცესების ავტომატიზაცია. თბ., „ტექნიკური უნივერსიტეტი“.

3. ფრანგიშვილი ა., სურგულაძე გ., ვაჭარაძე ი. (2009). ბიზნეს-პროგრამების ექსპერტულ შეფასებებში გადაწყვეტილებათა მიღების მხარდამჭერი მეთოდები და მოდელები. სტუ. მონოგრ., ISBN 978-9941-14-450-9. თბ., „ტექნიკური უნივერსიტეტი“.

4. Скопин И. (2004). Основы менеджмента программных проектов. Новосибирский Гос.Унив., INTUIT. http://www.intuit.ru/-studies/courses/38/38/info

5. Бадави Х., Изуно А., Левики П., Свитинбенк П., Хи Дж., Шварцер Х., Юсуф Л. Создание бизнес-процесса с помощью инструментов Rational и WebSphere. http://www.intuit.ru-/studies/courses/1152/258/info

6. Марчуков А.В., Савельев А.О. (2009). Работа в Ms VisualStudio. INTUIT. www.intuit.ru/studies/courses/499/355/lecture/8449.

7. სურგულაძე გ., ფხაკაძე ც., კეკენაძე ა. (2016). ორგანიზაციული მართვის ბიზნესპროცესების მოდელირება და დაპროექტება. მონოგრ., ISBN 978-9941-0-8259-7. სტუ, თბ., „IT კონსალტინგის ცენტრი“.

8. სურგულაძე გ., ქრისტესიაშვილი ხ., სურგულაძე გ. (2015). საწარმოო რესურსების მენეჯმენტის ბიზნესპროცესების მოდელირება და კვლევა. მონოგრ., ISBN 978-9941-20-557-6. თბ., „ტექნიკური უნივერსიტეტი“.

9. სურგულაძე გ., ბულია ი. (2012). კორპორაციულ Web-აპლიკაციათა ინტეგრაცია და დაპროექტება. მონოგრ., ISBN 978-9941-20-165-3.სტუ. თბ., „ტექნიკური უნივერსიტეტი".

10. სურგულაძე გ., ურუშაძე ბ. (2014). საინფორმაციო სისტემების მენეჯმენტის საერთაშორისო გამოცდილება (BSI, ITIL, COBIT). დამხმ.სახელმძღვ., ISBN 978-9941-20-458-6. თბ., „ტექნიკური უნივერსიტეტი".

11. ITIL moving towards Enterprise Architecture. http://blogs.msdn.com/b/mikewalker/archive/2007/07/06/itil-moving-towards-enterprise-architecture.aspx?Redirected=true.

12. Booch G., Jacobson I., rambaugh J. (1996). Unified Modeling Language for Object-Oriented Development. Rational Software Corporation, Santa Clara,

13. Бек К. (2008). Шаблоны реализации корпоративных приложений. Экстремальное программирование: Пер. с англ. М.: Вильямс.

14. Амблер С. (2005). Гибкие технологии: экстремальное программирование и унифицированный процесс разработки. Библиотека программиста. Спб.: Питер.

15. Rumpe B. (2012). Agile Modellierung mit UML. Berlin, „Springer". 2-te Auflage.

16. სურგულაძე გ., გულიტაშვილი მ., კაკულია ი., ჩერქეზიშვილი გ., ჯავახიშვილი ი. (2010). პროგრამული სისტემების სასიცოცხლო ციკლის პროცესის მოდელირება უნივერსალური და ექსტრემალური პროგრამირების პრინციპების კომპრომისული გადაწყვეტით. სტუ-ს შრ.კრ. „მართვის ავტომატიზებული სისტემები", N 1(8). გვ.63-70.

17. გოგიჩაიშვილი გ., თურქია ე. (2009). პროგრამული უზრუნველყოფის რეალიზაცია Rational Rose ინსტრუმენტის ბაზაზე. სტუ, თბ.

18. გოგიჩაიშვილი გ., სუხიაშვილი თ. (2012). სისტემების ობიექტ - ორიენტირებული ანალიზი და დაპროექტება. სტუ, თბ.,

19. რაზუმოვსკი კ. შესავალი პროგრამული უზრუნველყოფის მოქნილ დამუშავებაში. http://www.kv.by/index2008334201.htm.

20. Фергус О'Коннел. (2003). Как успешно руководить проектами. Серебряная пуля. М.: КУДИЦ-ОБРАЗ,

21. Закис А. Структурное руководство проектом. Серебряная пуля ? http://citforum.ru/SE/project /silver_bullet/.

22. ITILv3. Глоссарий терминов и определений, ITIL® V3 Glossary Russian Translation. v0.92, 30 Apr 2009.

23.070 Кознов Д.В. (2009). Введение в программную инженерию. http://www. intuit.ru/department/se/inprogeng.

24. Principles behind the Agile Manifesto. http://agilemanifes-to.org/princip-les.html.

25. UML: Basics Principles and Background. http://source-making.com/uml

26. სურგულაძე გ., გულიტაშვილი მ., ჩერქეზიშვილი. Web აპლიკაციების დამუშავების პროცესის მოდელირება UML/2 ტექნოლოგიით. Intern. Science Conf.“Automated Control Systems & new IT”, 20-22 Mai. GTU, Tbilisi, 2011. გვ. 180-184

27. The Unified Modeling Language. http://www.uml-diagrams.org/ უკანასკნ. გადამოწმ. 10.05.14

28. გოგიჩაიშვილი გ., ბოლხი გ., სურგულაძე გ., პეტრია-შვილი ლ. (2013). მართვის ავტომატიზებული სისტემების ობიექტ - ორიენტირებული დაპროექტებისა და მოდელირების ინსტრუმენ-ტები (MsVisio, WinPepsy, PetNet, CPN). სახელმძღვ., ISBN 99940-56-77-8. სტუ.. თბ., „ტექნიკური უნივერსიტეტი“.

29. Scrum.org. https://www.scrum.org/Resources

30. Кознов Д.В. (2009.). Введение в программную инженерию. http://www. intuit.ru/department/se/inprogeng.

31. Information Systems Examinations Board (ISEB). ttp://www.bcs.org/.

32. Foundation Certificate in IT Service Management. https:// www.exin.com/NL/en/exams/?exam=itil-v3-foundation.

32. COBIT Overview ISACA. http://www.isaca.org/knowledge-center/cobit/Pages/Overview.aspx. გადამოწ.15.01.14

33. COBIT 5 for Information Security. http://www.isaca.org/COBIT /Documents/COBIT-5-for-Information-Security-Introduction.pdf.

34. Скрипник Д. Управление ИТ на основе COBIT 4.1. М., 2012. www.intuit.ru/studies/courses/3704/946/info.

35. მეიერ-ვეგენერი კ., სურგულაძე გ., ბასილაძე გ. (2014). საინფორმაციო სისტემების აგება მულტიმედიური მონაცემთა ბაზებით. ISBN 978-9941-20-468-5. სტუ, თბილისი, „ტექნიკური უნივერსიტეტი“.

36. სურგულაძე გ., თურქია ე., ქაჩლიშვილი თ., ფხაკაძე ც. საფინანსო კორპორაციის ბიზნეს-პროცესების მენეჯმენტი ITIL მეთოდოლოგიის საფუძველზე (რეპორტების ავტომატიზაცია). სტუ-ს შრ.კრ. „მას“. 2, 18, თბ., 2014, გვ.51-56.

37. გიუტაშვილი მ. (2013). ბიზნესის ანალიზისა და ინტელექტუალური მართვის ტექნოლოგია (BI). სახელმძღვ., ISBN 978-9941-20-399-2. სტუ, თბილისი, „ტექნიკური უნივერსიტეტი“.

38. სურგულაძე გ., პეტრიაშვილი ლ. (2007). მონაცემთა საცავის აგების ტექნოლოგია ინტერნეტული ბიზნესის სისტემებისთვის. მონოგრ., ISBN 99940-36-7-7. სტუ.. თბ., „ტექნიკური უნივერსიტეტი“.

39. სურგულაძე გ., გულიტაშვილი მ., კივილაძე ნ. Web-სისტემების ტესტირება, ვალიდაცია და ვერიფიკაცია. (2015). მონოგრ. ISBN 9789941-0-7682-4. სტუ. „IT-კონსალტინგის ცენტრი“. თბ.,

40. Carnegie Mellon Software Engineering Institute. http://www.sei.cmu.edu/

41.Software quality assurance. https://en.wikipedia.org/wiki/-Software_quality_assurance

42. Sommerville I. (2010). Softwear engineering, 9th ed., ISBN 978-0137035151. Addison-Wesley.

სტუ-ს „ტექნიკური უნივერსიტეტი“

(თბილისი, მ.კოსტავას 77)