დოქტორანტის ბიბლიოთეკა

გია სურგულაძე

გამოყენებითი პროგრამული ინჟინერია

(პრაქტიკულის მეთოდური მითითებანი)

„სტუ-ს IT კონსალტინგის სამეცნიერო ცენტრი"

საქართველოს ტექნიკური უნივერსიტეტი

გია სურგულაძე

გამოყენებითი პროგრამული ინჟინერია

(პრაქტიკულის მეთოდური მითითებანი)

დამტკიცებულია:
სტუ-ს „IT კონსალტინგის სამეცნიერო ცენტრის" სარედაქციო კოლეგიის მიერ ოქმი N7, 15.10.2020

თბილისი
2020

უაკ 004.5

წარმოდგენილა საქართველოს ტექნიკური უნივერსიტეტის ინფორმატიკისა და მართვის სისტემების ფაკულტეტის სადოქტორო პროგრამის „ინფორმატიკა“ არჩევითი აკადემიური დისციპლინის „გამოყენებითი პროგრამული ინჟინერია“ სილაბუსის საფუძველზე თეორიული და პრაქტიკული საკითხების შესწავლის მეთოდური მითითებები. განსაკუთრებით გამახვილებულია ყურადღება პროგრამული ინჟინერიის, როგორც ინფორმატიკის მეცნიერული მიმართულების დისციპლინაზე, რომლის საფუძველზე შესაძლებელია სხვადასხვა დარგის ობიექტის მხარდამჭერი საინფორაციო სისტემების დიზაინისა და დეველოპმენტის განხორციელება თანამედროვე პროგრამული პლატფორმების, ენებისა და ტექნოლოგიების გამოყენებით. წიგნში შემოთავაზებულია პროგრამული პროდუქტების აგების სასიცოცხლო ციკლის ეტაპების დეტალური ანალიზის და პროექტირების საკითხები უნიფიცირებული (UML) და მოქნილი (Agile) მეთოდოლოგიებით, დაპროგრამების ჰიბრიდული და CASE-ტექნოლოგიებით, პროგრამული სისტემების ტესტირებით, ხარისხის შეფასებით და მართვით. განკუთვნილია სტუ-ის „ინფორმატიკის“ სადოქტორო პროგრამის სტუდენტებისათვის,

რეცენზენტი:

პროფ. ე. თურქია (საქართველოს ეროვნული ბანკის განყოფილების გამგე, ტ.მ.კ.)

პროფ. მ. ჩხაიძე (სტუ, ხელოვნური ინტელექტის დეპ.-ის უფროსი)

რედკოლეგია:

ა. ფრანგიშვილი (თავმჯდომარე), მ. ახობაძე, გ. გოგიჩაიშვილი, ზ. ბოსიკაშვილი, ე. თურქია, რ. კაკუბავა, ნ. ლომინაძე, ჰ. მელაძე, თ. ობგაძე, გ. სურგულაძე (რეადაქტორი), გ. ჩაჩანიძე, ა. ცინცაძე, ზ. წვერაიძე

© სტუ-ს „IT-კონსალტინგის სამეცნიერო ცენტრი“, 2020

ISBN 978-9941-8-2871-3

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

სასწავლო კურსის მიზანი

„გამოყენებითი პროგრამული ინჟინერიის“ (Applied SE) კურსი შეასწავლის „ინფორმატიკის“ საგანმანათლებლო პროგრამის დოქტორანტებს გამოყენებითი კომპიუტერული სისტემების დიზაინისა და დეველოპმენტის თანამედროვე პროგრამულ პლატფორმებს, ენებს და ტექნოლოგიებს. კერძოდ, დაპროგრამების პარადიგმებს, მეთოდოლოგიებსა და მეთოდებს, მოდელებს, ალგორითმებს და ენებს. აგრეთვე პროგრამული სისტემების რეალიზაციის ინსტრუმენტულ საშუალებებს. კურსი დოქტორანტებს უფართოებს და უღრმავებს ცოდნას სამეცნიერო კვლევით საქმიანობაში პროგრამული სისტემების დაპროექტებისა და აგების უნიფიცირებული და მოქნილი ტექნოლოგიების საფუძველზე, პროგრამების სასიცოცხლო ციკლისა და საჭირო რესურსების ეფექტიანი მართვის, პრობლემურ-ორიენტირებული დაპროგრამების მეთოდებისა და ხერხების ათვისებით.

საგნის შესწავლის შედეგად მიღებული ცოდნა და შეძენილი უნარები

საგნის შესწავლის შედეგად სტუდენტი ღებულობს ცოდნას და იძენს შემდეგ უნარებს:

1. განსაზღვრავს სხვადასხვა დარგის საკვლევი-საპროექტო ობიექტის მართვის პროცესის მხარდამ-ჭერი გამოყენებითი პროგრამული აპლიკაციის განვითარების (დეველოპმენტის) პრობლემებს და მათი გადაწყვეტის ამოცანებს და გზებს;
2. განსაზღვრავს დასაპროექტებელი გამოყენებითი პროგრამული აპლიკაციის ბიზნეს-მოთხოვნილებებს (სპეციფიკაციებს), მათი რეალიზაციის ინფრასტრუქტურას (IT-რესურესებს) და სამუშაო გუნდის შემადგენლობას (ადამიანური რესურსები) – პროექტის შესასრულებლად განსაზღვრული ფინანსებისა და ვადების გათვალისწინებით;
1. მიუსადაგებს გამოყენებით სფეროს მხარდამჭერი პროგრამული სისტემის დასაპროექტებლად და ასაგებად შესაბამის ეფექტიან პროგრამულ პლატფორმას, მეთოდოლოგიის, მეთოდების და ფრეიმვორკების შერჩევისას;
4. აკეთებს დასაბუთებულ დასკვნებს საკვლევი ბიზნეს-ობიექტის მხარდამჭერი პროგრამული სისტემის ინფრასტრუქტურის კომპონენტების ეფექტიანობის შესახებ: პროგრამული პლატფორმები, ენები, მონაცემთა ბაზები (საცავები) და სხვ.;
5. აანალიზებს გამოყენებითი პროგრამული აპლიკაციის მომხმარებელთა ინტერფეისების პროგრამებს, ახორციელებს მონაცემთა ბაზებთან (საცავებთან) კავშირს ინფორმაციის ეფექტური დამუშავების და უსაფრთხოების უზრუნველყოფის მიზნით;
6. აფასებს გამოყენებითი პროგრამული სისტემის ხარისხს და მართვას, რისკების ანალიზს, აპლიკაციის შეფასების მიზნით, ტესტირების, ოპტიმიზაციისა და საიმედოობის მეთოდების გამოყენების საფუძველზე;
7. ინტელექტუალური ამოცანების გადაწყვეტისას ახდენს საჭირო სამეცნიერო ლიტერატურის მოძიებას, რეპორტების შედგენას და ზეპირი პრეზენტაციების ჩატარებას;
8. ახდენს საერთაშორისო სამეცნიერო საზოგადოებასთან თემატურ პოლემიკაში ჩართვას, დასახული მიზნების მისაღწევად თანამედროვე ინფორმაციული ტექნოლოგიების/რესურსების ეფექტიანად გამოყენებას;
9. აფორმირებს საკუთარი თვითგანათლების მიმართულებებს – პროფესიული ცოდნის, გამოცდილების გაღრმავებისა და სრულყოფის მიზნით.

საათების განაწილება (სტუდენტის დატვირთვა)

„გამოყენებითი პროგრამული ინჟინერიის“ კურსი 7 კრედიტიანია (1 კრ = 25 სთ). სწავლის ფორმებია: ლექცია (30 სთ/სემესტრში), პრაქტიკული (30 სთ), კონსულტაცია (5 სთ) და დამოუკიდებელი მუშაობა (112 სთ). აგრეთვე 1-შუასემესტრული (1 სთ) და დასკვნითი გამოცდა (2 სთ).

პრაქტიკული მეცადინეობების თემების დასახელება და შინაარსი

„გამოყენებითი პროგრამული ინჟინერიის“ კურსის პრაქტიკული მეცადინეობების მასალა მოიცავს შემდეგ თემებს და საკითხებს:

1. მართვის საინფორმაციო სისტემების პროგრამული ინჟინერია: მართვა და მენეჯმენტი. საინფორმაციო სისტემები. კონკრეტული კვლევის ობიექტის საინფორმაციო სისტემის მოდელირების, დაპროექტების და დაპროგრამების პრობლემების განხილვა. გამოყენებითი პროგრამული სისტემების ინფრასტრუქტურის მაგალითის განხილვა.

პრაქტიკული დავალება: კონკრეტული საპრობლემო სფეროს სისტემური ანალიზი, პრობლემები, მიზნების, ფაქტორებისა და კრიტერიუმების ფორმირება. შესაბამისი პროგრამული სისტემის კომპონენტების კრეატიული დაგეგმარება;

2. გამოყენებითი პროგრამული სისტემის სასიცოცხლო ციკლი: კონკრეტული საპრობლემო სფეროს ობიექტ-ორიენტირებული მოდელირების და დაპროექტების მეთოდოლოგია. UML-ტექნოლოგიის შესაბამისი ინსტრუმენტული საშუალებები.

პრაქტიკული დავალება: გამოყენებითი პროგრამული სისტემების სასიცოცხლო ციკლის აგება და ანალიზი განტერის „ფაზა-ფუნქციების“, სპირალური მოდელის ან სხვ. საფუძველზე;

1. პრაქტიკული დავალება: კონკრეტული საპრობლემო სფეროსთვის დაპროგრამების მეთოდის, სტილის და ენის არჩევა (ფუნქციონალური, ლოგიკური, დეკლარაციული (სკრიპტული), სტრუქტურული, ობიექტ-ორიენტირებული, კომპონენტური, პროცეს-ორიენტირებული, სერვის-ორიენტირებული, ასპექტ-ორიენტირებული, სუბიექტ–ორიენტირებული, პარალელური, რეკურსიული პარადიგმების საფუძველზე);

4. კონკრეტული საპრობლემო სფეროს დაპროექტებისა და მოდელირების ტექნოლოგიის შერჩევა: სისტემების ავტომატიზებული დაპროექტებისა და დამუშავების მეთოდები და ინსტრუმენტული საშუალებები. CASE, MDA, RAD და DDD - მაგალითების განხილვა;

5. საპრობლემო სფეროს პროგრამული სისტემის დაპროექტება UML-ის ბაზაზე: კონკრეტული დომაინის საინფორმაციო სისტემის ბიზნეს–პროცესების ფუნქციონალური მოთხოვნილებების განსაზღვრა და შესაბამისი დიაგრამების დაპროექტება. პროგრამული სისტემების აგების სტანდარტები და ხარისხის მართვის ელემენტები;

6. საპრობლემო სფეროს პროგრამული სისტემის დაპროექტება Agile-ს ბაზაზე: კონკტრეტული დომაინის პროგრამული უზრუნველყოფის აგება მოქნილი (Agile) ტექნოლოგიით (ექსტრემალური დაპროგრამების საფუძველზე). Scrum - ტექნოლოგიის გაცნობა;

7. საწარმოო რესურსების მართვის და ინფორმაციის ინტეგრაციის თანამედროვე ტექნოლოგიები: ორგანიზაციული მართვის სისტემების კომპლექსური ავტომატიზაციის მაგალითების განხილვა და მათი ანალიზი. კონკრეტული საპრობლემო სფეროს ERP და CRM სისტემები და ბიზნეს-პროცესების დაპროგრამება ავტომატიზაციის მიზნით;

8. საპრობლემო სფეროს პროგრამული სისტემის შეფასება: კონკრეტული კვლევის სფეროს პროგრამული სისტემის დაპროექტება, პროგრამული რეალიზაცია, ტესტირება და ხარისხის შეფასება ობიექტ-ორიენტირებული პროგრამული სისტემების მეტრიკების საფუძველზე;

9. დაპროგრამების ტექნოლოგიები და გამოყენებითი პროგრამული სისტემები: კონკრეტული მართვის საინფორმაციო სისტემის აპლიკაციის დაპროგრამების ჰიბრიდული და მობილური ტექნოლოგიები. Windows- და Web-აპლიკაცების დაპროგრამების ინსტრუმენტული საშუალებების განხილვა და ექსპერიმენტული მაგალითების პროგრამირება;

10. პროგრამული სისტემების ხარისხის მართვა: კონკრეტული მაგალითების განხილვა პროგრამების ხარისხის შესაფასებლად. შეფასების მეტრიკები. პროგრამების ტესტირების, ვერიფიკაციის და ვალიდაციის მეთოდების მაგალითები. შესაბამისი პროგრამული საშუალებებით ექსპერიმენტუ;ლი მაგალითების განხილვა;

11. დაპროგრამების ავტომატიზაცია: კონკრეტული კვლევის ობიექტისთვის გრაფო–ანალიზური და იმიტაციური მოდელების აგება და ანალიზი. რიგების (მასობრივი მომსახურეობის) თეორიის (WinPepsy) და პეტრის ფერადი ქსელების (CPN) იმიტაციური მოდელირების სპეციალიზირებული ინსტრუმენტების განხილვა, მათი სპეციალური პროგრამირების ენების განხილვა;

12. გამოყენებითი პროგრამული სისტემები და ოპერაციათა კვლევა: კონკრეტული კვლევის ობიექტის ბიზნეს-პროცესების მართვის ამოცანების დაპროექტება და პროგრამული რეალიზაცია, შესაბამისი საიფორმაციო-საძიებო სისტემების განხილვა;

11. განაწილებული გამოყენებითი სისტემების დაპროგრამება: კონკრეტული კვლევის ობიექტისთვის კლიენტ-სერვერული და სერვის-ორიენტირებული არქიტექტურების რეალიზაციის კონცეფციის განხილვა. ექსპერიმენტული ამოცანების გადაწყვეტა დაპროგრამების ახალი ტექნოლოგიების - WPF, WCF, ASP.NET MVC პლატფორმებზე. ინფორმაციის დაცვის მაგალითების განხილვა;

14. უახლესი ვებ-ტქნოლოგიები და ვებ-აპლიკაციები: საილუსტრაციო მაგალითების განხილვა ვებ-სისტემების პროგრამირების ახალი პლატფორმებისა და ენების მაგალითზე. Javascript ენა და მისი შესაძლებლობის გაფართოებული ბიბლიოთეკები - jQuery, Angular, ReactJS. მაგალითების განხილვა;

15. მონაცემთა მენეჯმენტის თანამედროვე ტექნოლოგიები: მონაცემთა საცავებში ინფორმაციის დამუშავების თანამედროვე ტექნოლოგიების მაგალითების განხილვა (OLAP, DataMining), Hadoop ტექნოლოგიის თვალსაზრიით NoSQL მონაცემთა ბაზების აგების და პრაქტიკული გამოყენების მაგალითები MongoDB პაკეტით.

პრაქტიკული სამუშაოს შესრულების ნიმუშები

გამოყენებითი პროგრამული პროდუქტების *სასიცოცხლო ციკლის* მოდელებში აისახება საწარმოო ფუნქციები, რომლებსაც ასრულებენ დამპროექტებლები. ეს ფუნქციები კავშირშია პროექტების მართვის 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 იგი ითვალისწინებს იტერაციებსაც.

მოდელი არის პროექტის დამმუშავებლებს შორის ურთიერთდამოკიდებულების ორგანიზაციის საფუძველი. ამგვარად, მისი ერთ-ერთი მიზანი მენეჯერის ფუნქციების მხარდაჭერაა. ეს კი აუცილებლად მოითხოვს პროექტზე საკონტროლო წერტილების დალაგებას, რომელიც ასახავს პროექტის ორგანი-ზაციულ-დროით ჩარჩოებს, ორგანიზაციულ-ტექნიკურ საწარმოო ფუნქციებს, ისინი 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 მათი ინტენსივობა თითოეულ ეტაპზე.

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

ფუნქციების გამოყენების ინტენსივობა ფაზების მიხედვით მოცემულია მუქი ფერით.

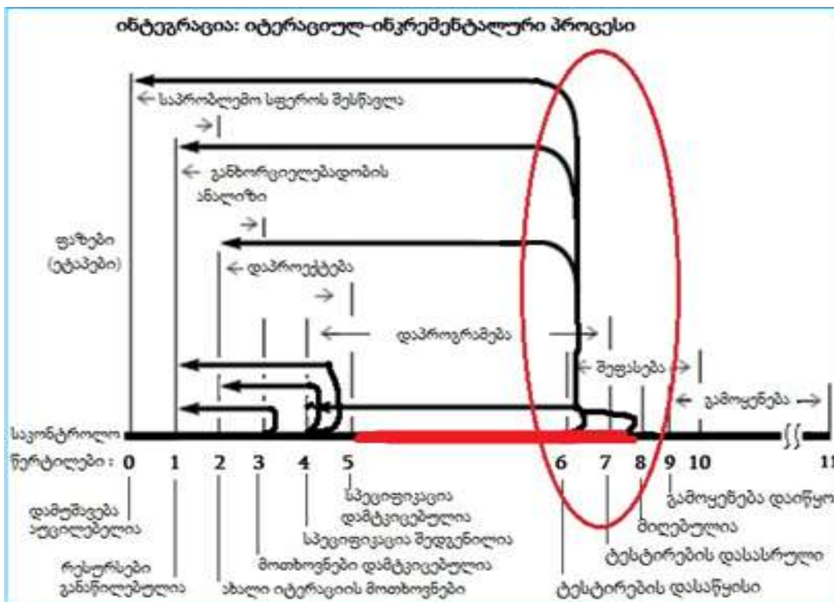
1.2 ნახაზზე ნაჩვებია სასიცოცხლო ციკლის საკონტროლო წერტილებში (მაგალითად, 3,4,6) იტერაციული ციკლების შესაძლებლობები.

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.

ქვემოთ ჩვენ განვიხილავთ გამოყენებითი პროგრამული სისატემების აგების ორ მეთოდოლოგიას: UML - დიდი პროექტებისათვის და Agile – სწრაფად გადასაწყვეტი პროექტებისათვის.

ფაზები (ეტაპები)
სს-შესწავლა
განხორციელებადობის ანალიზი
კონსტრუ-ირება
პროგრამირება
შეფასება
გამოყენება
ფუნქციები:
დაგეგმვა
დამუშავება
მომსახურება
დოკუმენტაციის მომზადება
გამოცდა
მხარდაჭერა
თანხლება
0 1 2 3 4 5 6 7 8 9 10
საკონტროლო წერტილები (მოვლენები)
სპეციფიკაციები დამტკიცებულია
სპეციფიკაციები შედგენილია
მოთხოვნები დამტკიცებულია
მოთხოვნები ფორმულირებულია
რესურსები განაწილებულია
დამუშავების აუცილებლობა აღიარებულია
პაკეტი შედგენილია
დამოუკიდებელი შემოწმება დაიწყო
პროგრამის გამოყენება დაიწყო
პროგრამა გადაცემულია გასავრცელებლად
პროგრამა მოხსნილია წარმოებიდან

ნახ.1.1. განტერის მოდელი „ფაზები-ფუნქციები“



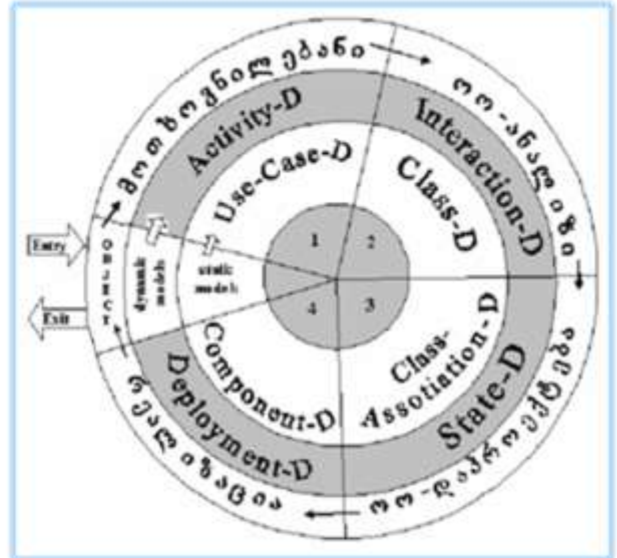
ნახ.1.2. იტერაციულობა განტერის მოდელში

1. UML მეთოდოლოგია და მისი ინსტრუმენტული საშუალებები (პრაქტიკული 5)

UML (Unified Modeling Language) არის განაწილებული მართვის საინფორმაციო სისტემების დაპროექტების მეთოდოლოგიური საფუძველი (ამერიკული ტექნოლოგია).

UML-ის ოთხეტაპიანი კლასიკური მოდელი შედგება 4 წყვილი დიაგრამით. მათგან 4 სტატიკური და 4 - დინამიკური მოდელია (ნახ.1.3).

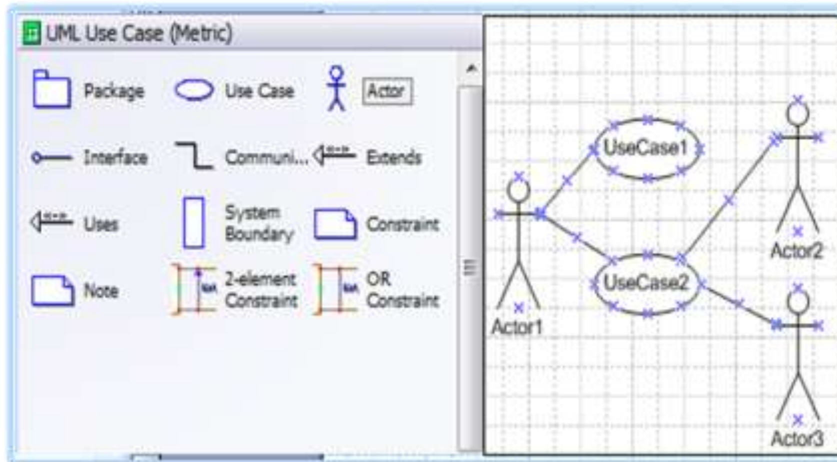
შეიძლება ითქვას, რომ უნიფიცირებული მოდელირების ენის ეტაპები და ამოცანები ზუსტად შეესაბამება ჩვენ მიერ ზემოთ განხილულ გამოყენებითი პროგრამული პროდუქტების სასიცოცხლო ციკლის ეტაპებს (ნახ.1.1).



ნახ.1.3

➢ Use Case დიაგრამა

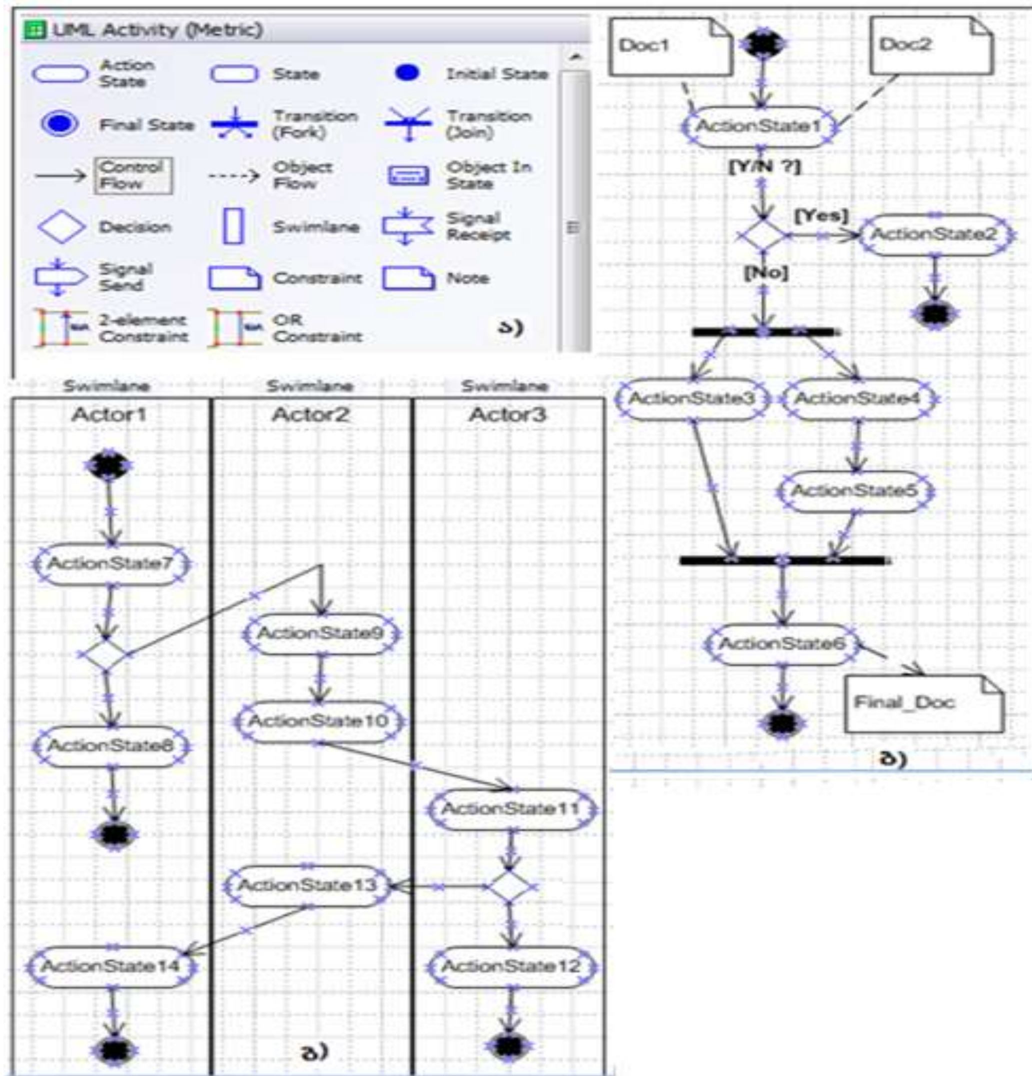
პირველი დიაგრამა, რომელიც UML ტექნოლოგიით უნდა აიგოს, არის გამოყენებით შემთხვევათა (პრეცედენტების) UseCase დიაგრამა. იგი როლების (Actors) და ფუნქციების (Actions) ურთიერთდაკავშირებული სქემაა (ნახ.1.4). აქ UseCase1 ეკუთვნის მხოლოდ პირველ როლს, ხოლო UseCase2-ის შესასრულებლად ორივე როლი მონაწილეობს.



ნახ.1.4. UseCase დიაგრამის აგების ინტერფეისი

➢ Activity დიაგრამა

ბიზნესპროცესებისა და ბიზნესწესების გაცნობის, ანალიზის და სტრუქტურული ფორმალიზაციის საფუძველზე აიგება აქტიურობის (ქმედებათა) დიაგრამა. იგი კონკრეტული როლის (როლების) კონკრეტული ფუნქციაა, რომელიც შედგება იერარქიულად სივრცესა და დროში დალაგებული მიმდევრობით ან პარალელურად შესასრულებელი სუბქმედებებისგან. აქვს ერთი დასაწყისი და რამდენიმე შესაძლო დასასრული, ბიზნესწესებით განსაზღვრული განშტოების ან შეერთების პროცედურები, საწყისი, შუალედური ან საშედეგო დოკუმენტაცია და ა.შ. საილუსტრაციო მაგალითი მოცემულია 1.5 ნახაზზე.



ნახ.1.5. Activity დიაგრამის:

ა) ინსტრუმენტების პანელი; ბ) ქმედებათა სქემა (მარტივი);
გ) ქმედებათა სქემა (მართვის სფეროების ან როლების ბილიკებით)

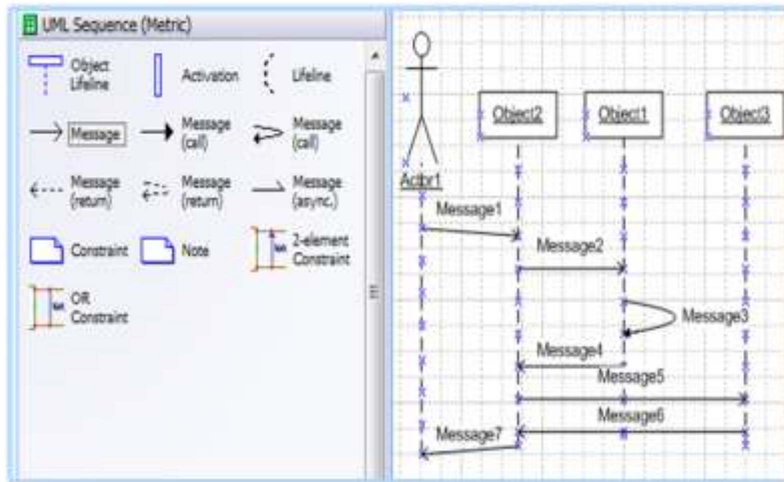
ქმედებათა დიაგრამა მიეკუთვნება პროცესების აღწერის დინამიკურ მოდელს, იგი ასახავს საკვლევი ობიექტის ქცევას. ასეთი დინამიკური მოდელების პროცესების გამოსაკვლევად გამოიყენება პეტრის ქსელები (გრაფო-ანალიზური მათემატიკური აპარატი იმიტაციური მოდელების ასაგებად).

საბოლოოდ, ამ ორი სახის (UseCase, Activity) დიაგრამათა ერთობლიობის ანალიზის საფუძველზე კეთდება დასკვნები საავტომატიზაციო ობიექტის მართვის სისტემის ფუნქციური და არაფუნქციური მოთხოვნილებების განსაზღვრის შესახებ.

ამავდროულად დგება მომავალი პროგრამული სისტემის შექმნის *ტექნიკური დავალება*, შესაბამის ეკონომიკურ გაანგარიშებებთან ერთად, რომელიც დამკვეთ ორგანიზაციის ხელმძღვანელობასთან კონსულტაციების შემდეგ, ორმხრივად მტკიცდება. ამის შემდეგ იწყება ობიექტორიენტირებული ანალიზის ეტაპი, რომელზეც აიგება სისტემის მომხმარებელთა ინტერაქტიული სქემები: მიმდევრობითი და თანამოქმედების დიაგრამათა სახით.

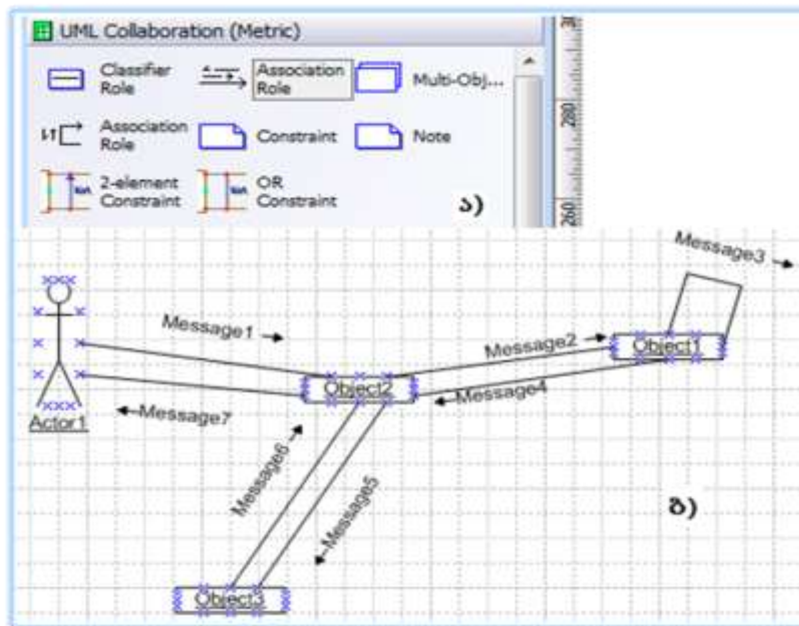
➢ Sequence და Collaboration დიაგრამები

მიმდევრობითობის დიაგრამა აღწერს საპრობლემო სფეროს კონკრეტული ამოცანის შესრულების სცენარს. აქ ხდება როლის სისტემასთან ურთიერთქმედების ბიზნესპროცესის ქმედებათა და მათი მაინიცირებელ, სინქრონულ ან ასინქრონულ შეტყობინებათა დროში მიმდევრობით განლაგება (ნახ.1.6).



ნახ.1.6. Sequence დიაგრამისაგების ინტერფეისი

1.7 ნახაზზე ნაჩვენებია თანამოქმედების დიაგრამის აგების ინსტრუმენტების პანელი (ა) და თვით დიაგრამა (ბ), რომელიც შესაბამისი მიმდევრობითობის დიაგრამის ტრანსფორმაციით იქნა მიღებული.



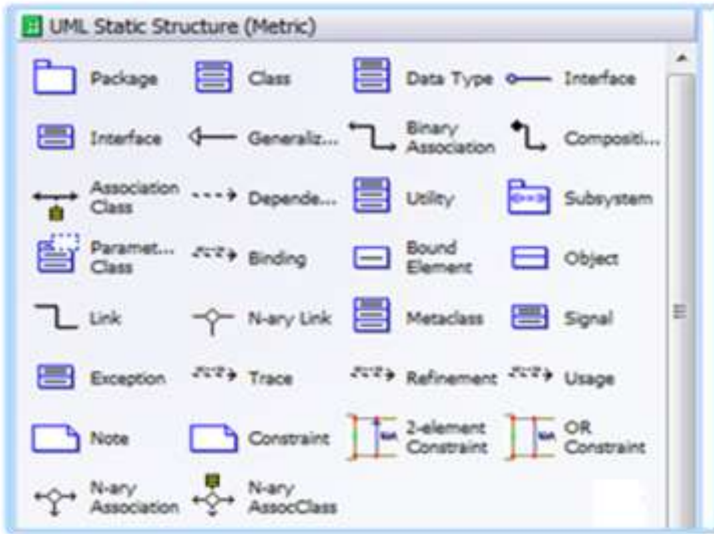
ნახ.1.7. Collaboration დიაგრამის აგების ინტერფეისი (ა)
და სქემის მაგალითი (ბ)

აქ შეტყობინებათა და მონაცემთა გაცვლის მიმდევრობა არაა დროში დალაგებული, სამაგიეროდ ჩანს კლასის ობიექტებს შორის კავშირებისა და ინფორმაციული ნაკადების გაცვლის სემანტიკა.

CASE ტექნოლოგიის მრავალ პროდუქტში (მაგალითად, ფირმა Rational Rose) თანამიმდევ-რობითობის დიაგრამის აგება ხდება ავტომატიზებულ რეჟიმში. ანუ ჯერ აიგება მიმდევრობითობის დიაგრამა და შემდეგ ინსტრუმენტის რომელიმე სწრაფი ღილაკით (მაგალითად, F5) სისტემა თვითონ ააგებს თანამოქმედების დიაგრამას].

➢ კლასების (Class) დიაგრამა

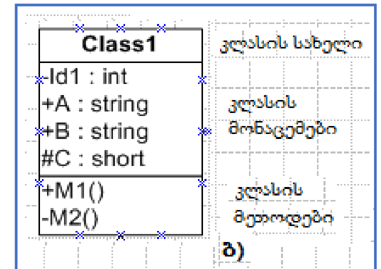
კლასი არის ერთგვაროვან ობიექტთა ერთობლიობის სტრუქტურა. მაგალითად, ყველა მოქალაქე (ზოგადად Person), სტუდენტები, ლექტორები, ავტომანქანები, ცხოველები და ა.შ.



ნახ.1.8

ტერმინი „კლასიფიკაცია" სწორედ განსახილველი სფეროს ობიექტების სისტემატურ მოწესრიგებას ემსახურება (*გენერალიზაცია* – იერარქიაში განზოგადება ზევით და *სპეციფიკაცია* – იერარქიაში დეტალიზაცია ქვევით). MsVisio-ში კლასებისა და კლასთაშორის კავშირების მოდელირებისათვის გამოიყენება Static Structure ინსტრუმენტების პანელი (ნახ.1.8).

ობიექტორიენტირებული მოდელირების და პროგრამირების გაგებით, კლასი არის „დასახელების", „კლასის მონაცემებისა" და „კლასის მეთოდების" ინკაფსულაცია. მართვის სფეროს შესაბამისი კლასი, ზოგადად ასე უნდა გამოიყურებოდეს (ნახ.1.9). კლასის ატრიბუტებს შეესაბამება მონაცემთა გარკვეული ტიპი (int, float, string ან სხვ.) და „ხილვადობის" ანუ მონაცემთა წვდომის მოდიფიკატორები („-" private, „+" public, „#" protection). ასევეა კლასის მეთოდებისთვისაც.

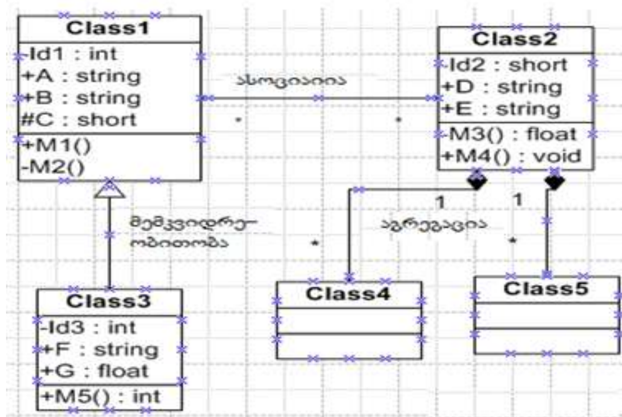


ნახ.1.9. ინკაფსულაცია

კლასის მეთოდები (ფუნქციები) ის პროგრამული მოდულებია, რომლებიც ამუშავებს კლასის მონაცემებს. მათი ინიციალიზაცია ხდება გარედან შემოსული შეტყობინებით.

➢ Class-Assotiation დიაგრამა

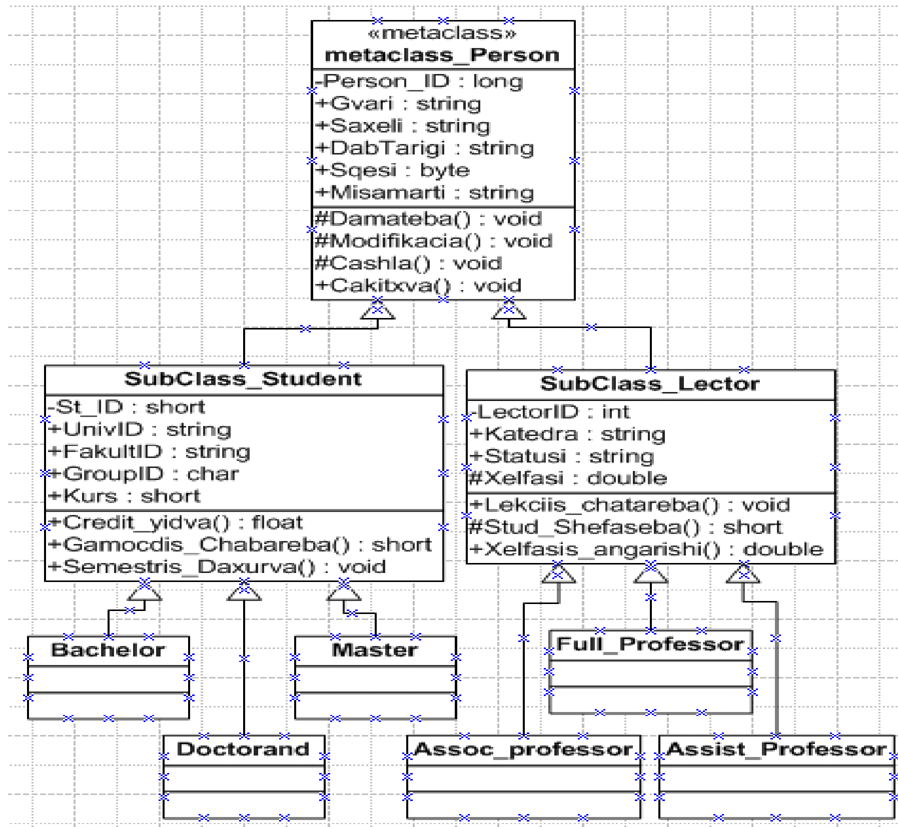
კლასთაშორისი კავშირები შეიძლება იყოს: მემკვიდრეობითი, აგრეგატული, რელაციური და ასოციაციური (ნახ.1.10):



ნახ.1.10. კლასთაშორისი კავშირების დიაგრამა

- მემკვიდრეობითი (Generalization) ასახავს „გენეტიკურ“, განზოგადოებულ კავშირებს კლასებს შორის. ასეთ დროს ერთი კლასი („შვილი“) მთლიანად იღებს მეორე კლასის („მშობელი“) ყველა ატრიბუტს, მეთოდს და კავშირს;
- აგრეგირებული (Aggregation) ნიშნავს კავშირს „მთელი“–„ნაწილი“. მაგალითად, „ავტომობილი“ – „ძარა, ძრავი, საბურავები და სხვ.“;
- ასოციაციური (Assotiation) ნიშნავს სემენტიკურ კავშირს კლასებს შორის. ის შეიძლება გამოისახოს ერთ- ან ორმიმართუ-ლებიანი ხაზით. ისარი გვიჩვენებს შეტყობინების გადაცემის მიმართულებას. ასოციაციური კავშირის რეალიზება ხდება ერთ კლასში დამატებით მეორე კლასის ატრიბურტის ჩასმით.
- რელაციური (Dependency) ნიშნავს ერთი კლასის დამოკიდებულებას მეორეზე. იგი ერთმიმართულებიანი წყვეტილი ისრით გამოიხატება. მასში დამატებითი დამაკავშირებელი ატრიბუტები არ გამოიყენება.

1.11 ნახაზზე ილუსტრირებულია კლასთა ასოციაციის დიაგრამა მემკვიდრეობითი კავშირების საფუძველზე. ისარი მიმართულია „შვილიდან“ „დედისკენ“, რაც მათ ცალსახა დამოკიდებულებაზე მეტყველებს. „შვილს“ ჰყავს ერთი „დედა“, ხოლო „დედას“ შეიძლება ჰყავდეს რამდენიმე „შვილი“, ამიტომაც ეს არაა ცალსახა.



ნახ.1.11. Class-Assotiation დიაგრამა მემკვიდრეობითი კავშირებით

მშობელი კლასი ლიტარატურაში ზოგჯერ „მეტაკლასად“ (MetaClass) მოიხსენიება, რომელიც შედგება ქვეკლასებისაგან (SubClasses). შეიძლება იერარქიაში ქვეკლასი იყოს მის ქვევით მდგარი კლასისათვის მეტაკლასი. მაგალითად, SubClass_Student არის ქვეკლასი MetaClass_Person კლასისათვის და, ამავდროულად იგი არის მეტაკლასი სამი ქვეკლასისთვის: Bachelor, Master და Doctorand. იგივე შეიძლება ითქვას კლასებისათვის:

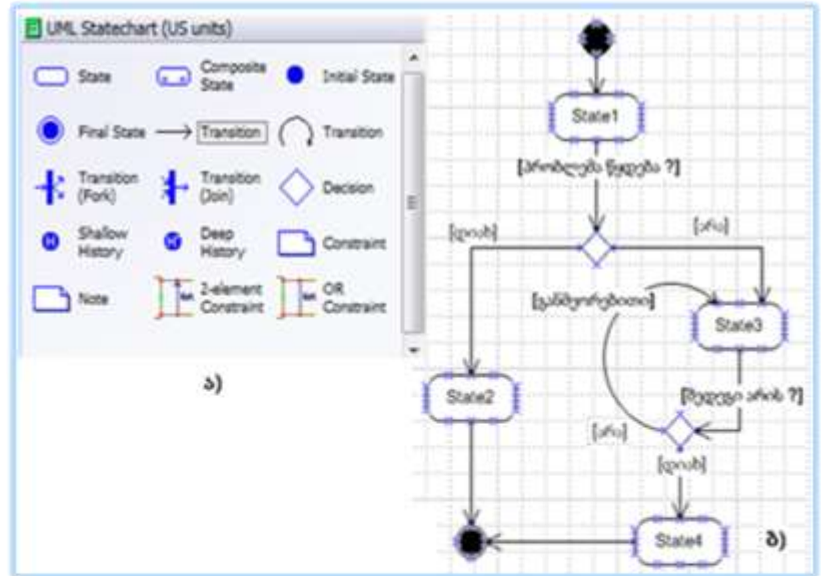
Metaclass_Person <- SubClass_Lector <- {Full_Professor, Assoc_Professor, Assist_Professor},

სადაც როლები ასეა განაწილებული: „მშობელი“-Person, „შვილი“-Lector და „შვილიშვილები“ Full_Professor, Assoc_Professor, Assist_Professor.

მდგომარეობათა (Statechart) დიაგრამა

ყოფაქცევის დიაგრამებიდან არსებობს კიდევ ერთი *კლასების მდგომარეობათა* დიაგრამა - Statechart-D. იგი აღწერს ქმედებებს, ობიექტთა მდგომარეობებს, მდგომარეობათა გადასვლებს და მოვლენებს. 1.12-ა,ბ ნახაზებზე ნაჩვენებია ინსტრუმენტული პანელისა და მდგომარეობათა დიაგრამის ფრაგმენტი ზოგადი მაგალითისთვის.

მისი გამოყენება ყველა კლასისათვის არაა საჭირო. აუცილებელია მხოლოდ მაშინ, როდესაც კლასი შეიძლება იმყოფებოდეს რამდენიმე მდგომარეობაში და თითოეულ მათგანში მისი ქცევა უნდა იყოს სხვადასხვანაირი.



ნახ.1.12. Statechart-ის პანელი (ა) და ზოგადი დიაგრამა (ბ)

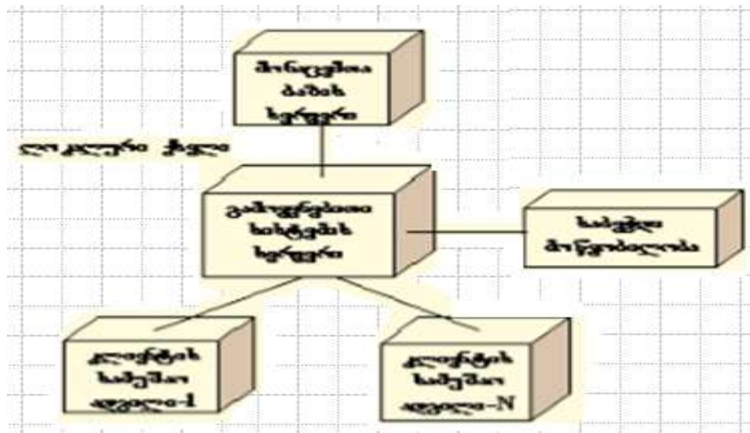
- ### კომპონენტების (Components) დიაგრამა

პროგრამული რეალიზაციის (development-ის) ეტაპზე აიგება კომპონენტების დიაგრამა (Component-D), რომელშიც იგულისხმება პროგრამული კოდების (მაგალითად: .cs, .cpp, .h, .dll, .exe და ა.შ.) დამუშავება (ნახ.1.13).

ნახ.1.11. კომპონენტების დიაგრამა

- ### განთავსების (Deployment) დიაგრამა

განაწილებული სისტემის რეალიზაციის ბოლო ეტაპზე აიგება განთავსების დიაგრამა (Deployment-D), რომელიც აღწერს კომპონენტების განაწილებას „კლიენტ-სერვერის" ქსელში (ნახ.1.14).

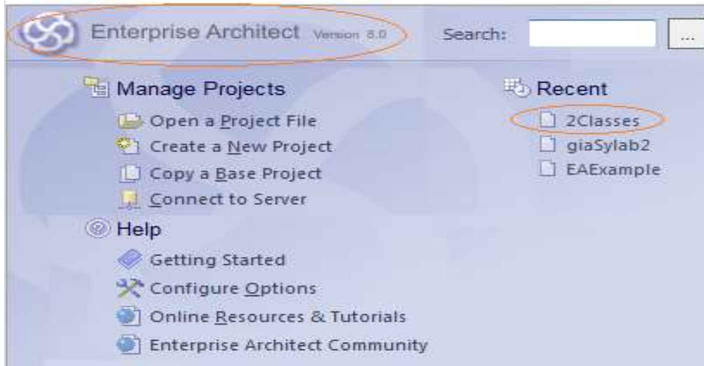


ნახ.1.14. განთავსების დიაგრამა

➢ კლასების დიაგრამიდან პროგრამული კოდის გენერაცია

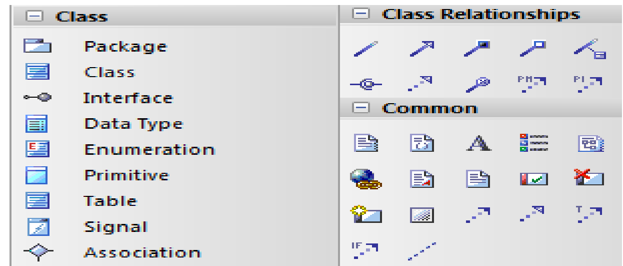
თანამედროვე CASE-ტექნოლოგიები, რომლებიც სისტემების დაპროგრამების ავტომატიზაციაზეა ორიენტირებული, მაგალითად, Rational Rose, Visual Paradigm, Enterprise Architect და მრავალი სხვა, ახორციელებს რევერსული დაპროგრამების კონცეფციას. ანუ კლასების დიაგრამიდან შესაძლებელია პროგრამული კოდის გენერაცია და პირიქითაც,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

Ms Visual Studio .NET Framework-ისთვის შექმნილია ინსტრუმენტები და მათი ინტეგრაციით .NET გარემოში, შესაძლებელია დიაგრამებიდან კოდის გენერაცია. ასეთი პაკეტები ყოველთვის ფასიანი და ძვირადღირებულია. აქ განვიხილავთ SparX ფირმის Enterprise Architect პროდუქტის ამ კონკრეტულ ფუნქციას, კლასების დიაგრამიდან კოდის გენერაციის ამოცანას. 1.15 ნახაზზე ნაჩვენებია პაკეტის ამუშავების საწყისი გვერდი.



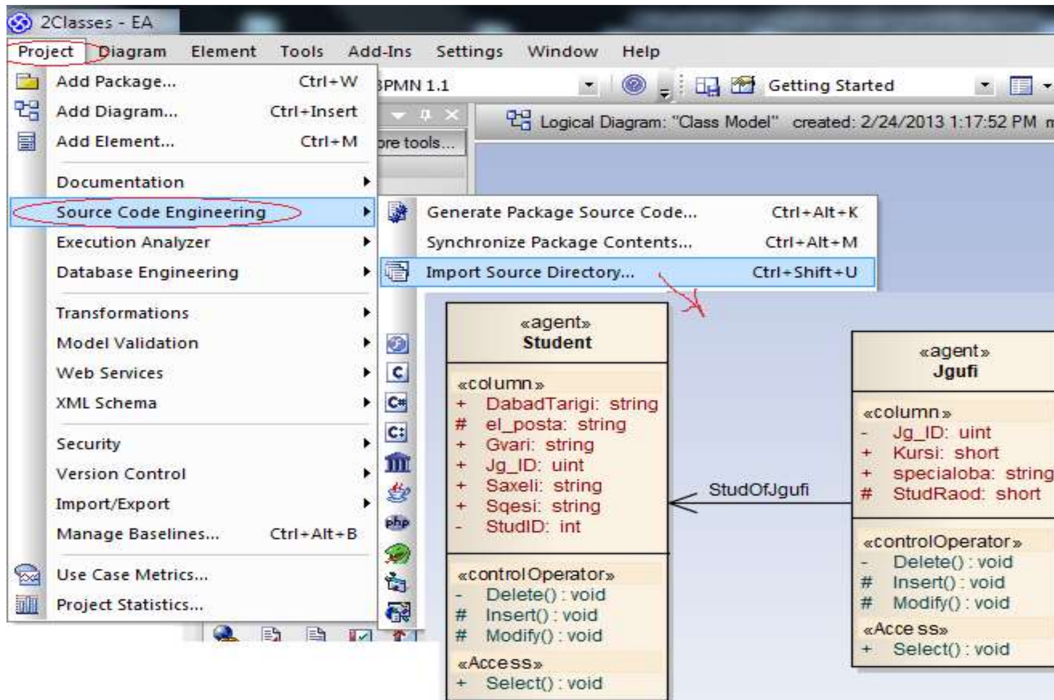
ნახ.1.15. Enterprose Architect საწყისი გვერდი

მაგალითისთვის ვიხილავთ ამ გარემოში ორი კლასის (2Clases მოდელი) აგების და მათი პროგრამულ კოდში გადაყვანის ამოცანას. 1.16 ნახაზზე მოცემულია Enterprise Architect პაკეტის კლასთა დიაგრამის აგების ინსტრუმენტების პანელი.



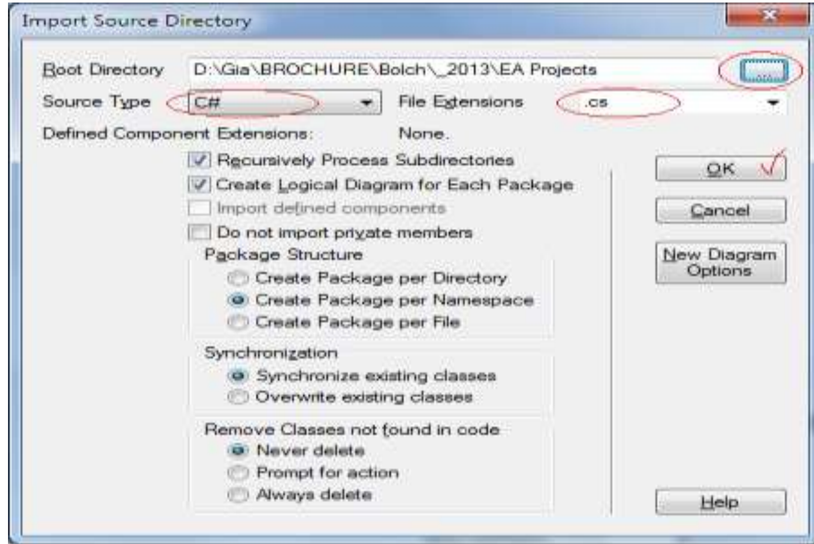
ნახ.1.16

ვირჩევთ Report->Source Code Engineering->Import Source Directory-ის (ნახ.1.17).



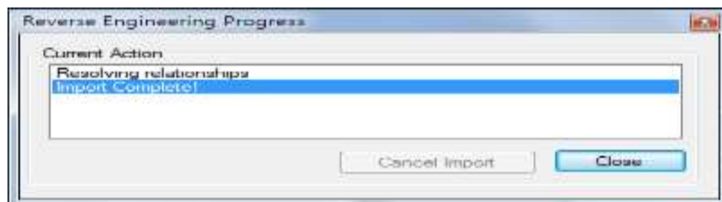
ნახ.1.17. Student და Jgufi კლასების მომზადება „Code Engineering“ პროცესისათვის Enterprise Architect გარემოში

არჩევის შემდეგ გამოვა 10.18 ნახაზზე ნაჩვენები ფანჯარა, რომელშიც უნდა განისაზღვროს ზოგიერთი მნიშვნელოვანი პარამეტრი, მაგალითად, ენა (C#), მომავალი კოდის შესანახი ადგილი (დირექტორია) და ა.შ.



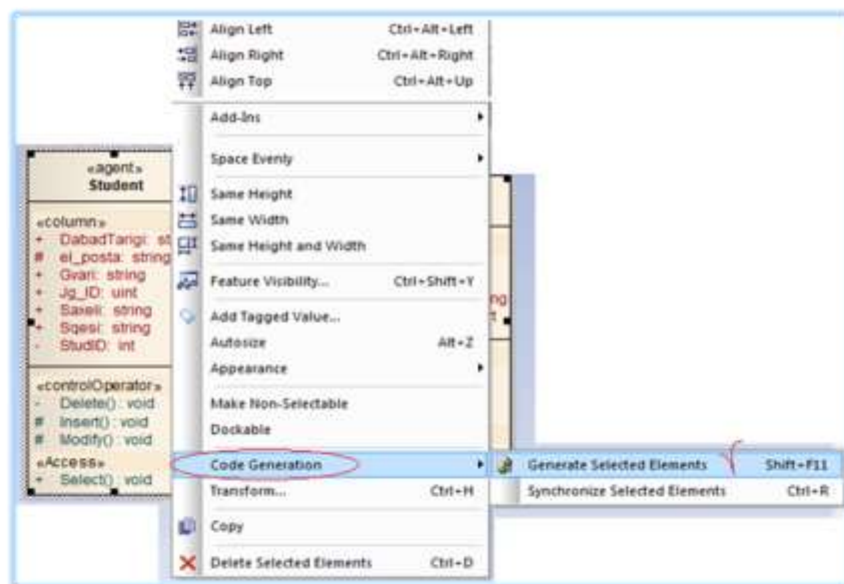
ნახ.1.18. განისაზღვროს C#-კოდის დირექტორია

ბოლოს „Ok“ და მივიღებთ 1.19 ნახაზზე მოცემულ შედეგს.

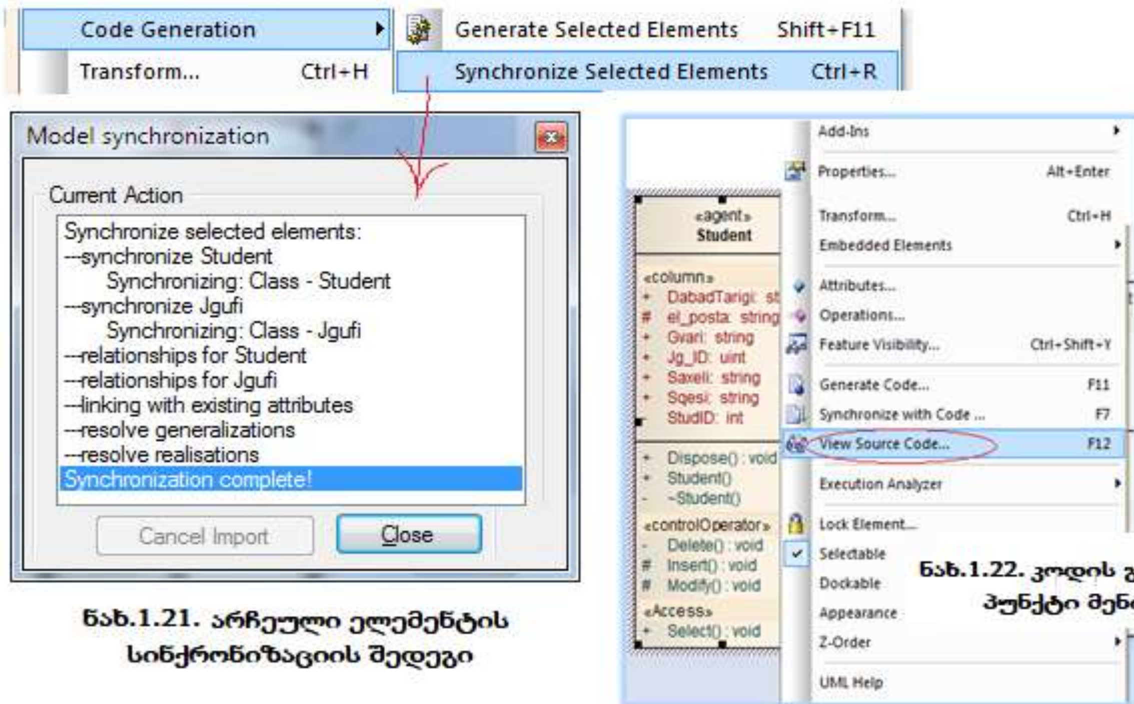


ნახ.1.19. იმპორტი დასრულებულია

ახლა უნდა ჩატარდეს უშუალოდ კოდის გენერაცია წინასწარ მომზადებული (Student-Jgufi) კლასების დიაგრამიდან. ვააქტიურებთ კლასებს და მაუსის ღილაკით გამოტანილ კონტექსტური მენიუდან ვირჩევთ „Code Generation“-ს (ნახ.1.20-21).



ნახ.1.20. კოდის გენერაციის დაწყება



ნახ.1.21. არჩეული ელემენტის სინქრონიზაციის შედეგი

ნახ.1.22. კოდის გამოტანის პუნქტი მენიუში

ბოლო ფაზაზე საჭიროა ეკრანზე გამოვიტანოთ კლასების ბაზაზე გენერირებული კოდის ლისტინგები (ნახ.1.22).

1.23 ნახაზზე ნაჩვენებია Enterprise Architect გარემოში Student კლასის დიაგრამიდან ავტომატრურად გენერირებული C#-კოდის საწყისი ტექსტი. ფანჯრის მარცხენა ნაწილში მოთავსებულია Student კლასის კაფსულა, თავისი მონაცემებით და მეთოდებით, მათ შორის კონსტრუქტორით და დესტრუქტორით. ფანჯრის მარჯვენა ნაწილში სისტემას გამოაქვს პროგრამის ტექსტი, რომელიც შედგება კომენტარული ნაწილის (სტრიქონები 1-7) და კლასის აღწერის ნაწილისაგან (8-31).

```
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.1.21. C#-კოდის ლისტინგი Student კლასისათვის

მომდევნო ლისტინგში მოცემულია Jgufi კლასის საწყისი ტექსტი. აქაც, C# კოდის ტექსტი შედგება კომენტარული ნაწილისაგან, რომელშიც ასახულია პროგრამის ზოგადი მახასიათებლები, სახელი, ინსტრუმენტი, შექმნის თარიღი, ავტორი. პროგრამის ტექსტი კლასიკური ფორმატით აღიწერება კლასის მონაცემები ხილვადობის private, public და protection ატრიბუტებით. შემდეგ მოსდევს კონსტრუქტორის public Jgufi(){ } და დესტრუქტორის ~Jgufi(){ } სტრიქონები. Dispose() მეთოდი გამოიყენება პროგრამის შესრულების დამთავრების შემდეგ ოპერაციული სისტემის მიერ გამოყოფილი რესურსების გასათავისუფლებლად.

```
///////////////// ლისტინგი //////////////////////
//  Jgufi.cs
//  Implementation of the Class Jgufi
//  Generated by Enterprise Architect
//  Created on:    14-Feb-2020 1:15:15 PM
//  Original author: user
public class Jgufi {
          private uint Jg_ID;
          public short Kursi;
          public string specialoba;
          protected short StudRaod;
          public Student m_Student;
          public Jgufi() {   }     // კონსტრუქტორი
          ~Jgufi() { }    // დესტრუქტორი
          public virtual void Dispose() {  }
          private void Delete(){
          // . . . code-1          }
          protected void Insert(){
          // . . . code-2          }
          protected void Modify(){
         // . . . code-3          }
          public void Select(){
         // . . . code-4
          }
}//end Jgufi
```

2. Agile მეთოდოლოგია, მისი მეთოდები და ინსტრუმენტული საშუალებები (პრაქტიკული 6)

➢ პროგრამების მოქნილი დეველოპმენტის მანიფესტი და პრინციპები

მანიფესტი ოთხი პუნქტისაგან შედგება, რომელთაგანაც თითოეული ალტერნატივაა. 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 ბოლო ეტაპზეც კი (ეს ზრდის საშედეგო პროდუქტის კონკურენტუნარიანობას);

3. სამუშაო პროგრამული უზრუნველყოფის ხშირი მიწოდება დამკვეთზე (ყოველთვიური, კვირეული ან უფრო ხშირი);

4. დამკვეთის ხშირი, ყოველდღიური კონტაქტი მიმწოდებელთან პროექტის შესრულების მთელ მანძილზე;

5. პროექტზე მუშაობენ მოტივირებული პირები, რომლებიც უზრუნველყოფილია მუშაობის საჭირო პირობებით, მხარდაჭერითა და ნდობით;

6. ინფორმაციის გადაცემის რეკომენდებული მეთოდი – პირადი საუბრები (პირისპირ);

7. მომუშავე პროგრამული უზრუნველყოფა – პროგრესის საუკეთესო საზომია. პროექტების მიზანია პროგრამული სისტემის შექმნა და არა გეგმებისა და დოკუმენტაციის. აპლიკაციის მუშაობისუნარიანობის შეფასებით შეიძლება პროექტის პროგრესის ობიექტური გაზომვა;

8. სპონსორებს, მიმწოდებლებსა და მომხმარებლებს უნდა ჰქონდეთ მხარდაჭერის შესაძლებლობა მუდმივი ტემპის შესანარჩუნებლად გაურკვეველი ვადით;

9. მუდმივი ყურადღება ტექნიკური ოსტატობის (უნარების) სრულყოფას და მოსახერხებელ დიზაინს;

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

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

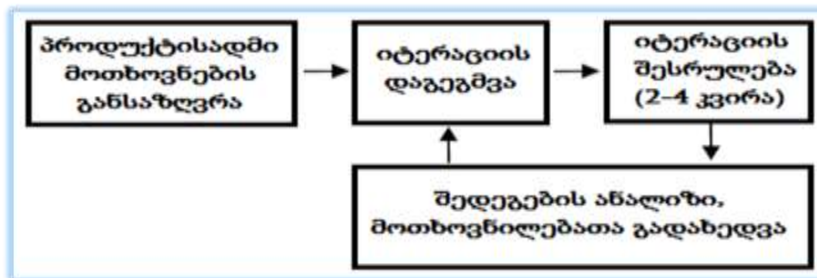
12. მუდმივი ადაპტაცია ცვალებად გარემოებებისადმი. იტერაციული სასიცოცხლო ციკლი ბაზირებულია მართვაზე უკუკავშირით, რომლის მნიშვნელოვანი ელემენტია შედეგების ანალიზი, უკუკავშირის განხორციელება და პროცესის სრულყოფა.

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

- Scrum - მოქნილი მეთოდის ფრეიმვორკი

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

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.2.1).



ნახ.2.1. 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 შესაძლებლობებით.

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

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

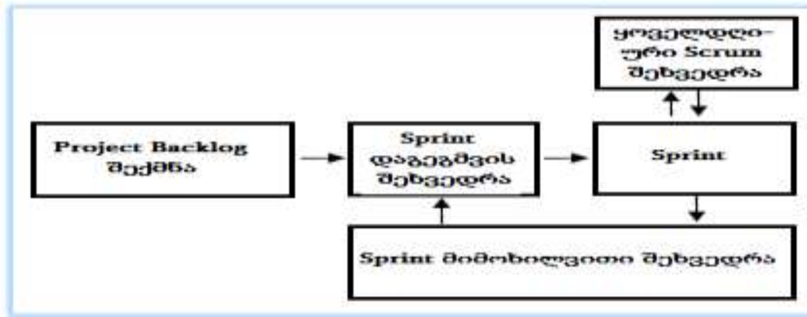
როლები. Scrum მეთოდში არის სამი როლი:

- *პროდუქტის მფლობელი (Product Owner)* – ესაა პროექტის მენეჯერი, რომელიც წარმოადგენს დამკვეთის ინტერესებს. მისი მოვალეობაა პროდუქტის საწყისი მოთხოვნების (Product Backlog) განსაზღვრა, მათი დროულად კორექტირება, პრიორიტეტების, ჩაბარების ვადების დადგენა და სხვ. იგი არ მონაწილეობს უშუალოდ იტერაციის შესრულებაში;
- *Scrum-ოსტატი (Scrum Master)* – უზრუნველყოფს გუნდის მაქსიმალურ მწარმოებლურობასა და პროდუქტიულობას, როგორც Scrum-პროცესის შესასრულებლად, ისე სამეურნეო და ადმინისტრაცული ამოცანების გადასაწყვეტად. კერძოდ, მისი ამოცანაა გუნდის დაცვა იტერაციის დროს ყოველგვარი გარე ზემოქმედებიდან;
- *Scrum-გუნდი (Scrum Team)* – ჯგუფია, რომელიც შედგება 5-9 დამოუკიდებელი, ინიციატივიანი პროგრამისტ-დეველოპერებისგან. გუნდის *პირველი ამოცანაა* იტერაციისათვის რეალურად მიღწევადი და პროექტისთვის პრიორიტეტული დავალებების განსაზღვრა (Project Backlog-ის საფუძველზე და პროდუქტის მფლობელისა და Scrum-ოსტატის აქტიური მონაწილეობით). *მეორე ამოცანაა* ამ დავალებების უეჭველი შესრულება დადგენილ ვადებში და მოთხოვნილი ხარისხით. მნიშვნელოვანია, რომ გუნდი თვითონ მონაწილეობს დავალებათა დასმის პროცესში და თვითონ წყვეტს მათ. აქ შეთავსებულია თავისუფლება და პაუხისმგებლობა, დონეზეა მოვალეობათა დისციპლინა.

პრაქტიკები. Scrum-ში განსაზღვრულია პრაქტიკები:

- *Sprint Planning Meeting.* შეხვედრა Sprint-ის დასაგეგმად (ესაა მოკლე დისტანცია, ანუ იტერაცია). თავიდან პროდუქტის მფლობელი, Scrum-ოსტატი, გუნდი, ასევე დამკვეთის წარმომადგენელი და სხვა დაინტერესებული პირები განსაზღვრავენ, თუ რომელი მოთხოვნებია Project Backlog-დან უფრო პრიორიტეტული და რომელი უნდა განხორციელდეს მოცემული სპრინტის ფარგლებში. ფორმირდება Sprint-Backlog. შემდეგ Scrum-ოსტატი და Scrum-გუნდი განსაზღვრავენ, თუ როგორ უნდა იქნას მიღწეული დასმული მიზნები Sprint-Backlog-დან. მისი ყოველი ელემენტისათვის დადგინდება ამოცანათა სია და შეფასდება მათი შრომატევადობა;
- *Daily Scrum Meeting* – ყოველდღიური, 15-წუთიანი Scrum თათბირი, რომლის მიზანია იმის გარკვევა, თუ რა მოხდა წინა თათბირის შემდეგ, კორექტირდეს სამუშაო გეგმა დღევანდელი დღის შესაბამისად და განისაზღვროს არსებული პრობლემების გადაწყვეტის გზები. Scrum-გუნდის ყოველი წევრი პასუხობს სამ კითხვას: რა გააკეთა წინა თათბირის შემდეგ, რა პრობლემები აქვს და რა უნდა გააკეთოს მომდევნო შეხვედრამდე. ამ თათბირზე დასწრება შეუძლია ნებისმიერ დაინტერესებულ პირს, მაგრამ გადაწყვეტილების მიღების უფლება აქვთ მხოლოდ Scrum-გუნდის წევრებს. ეს წესი მართებულია, რადგან ისინი იღებენ ვალდებულებას იტერაციის მიზნის მიღსაღწევად. გარე პირის ჩარევა ასეთ დროს ხსნის მათგან შედეგზე პასუხისმგებლობას;
- *Sprint Review Meeting.* Sprint მიმოხილვის შეხვედრა. იმართება ყოველი სპრინტის დამთავრების შემდეგ (ნახ.2.2).

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



ნახ.2.2. Scrum-მეთოდი Sprint-ბიჯებით

შეხვედრის მეორე ნაწილი ეხება წინა სპრინტის ანალიზს, რომელსაც წარმართავს Scrum-ოსტატი. Scrum-გუნდი აანალიზებს ბოლო სპრინტის დროს ერთობლივი მუშაობის დადებით და უარყოფით მომენტებს, გამოიტანს დასკვნებს და იღებს მნიშვნელოვან გადაწყვეტილებებს შემდგომი მუშაობისათვის. Scrum-გუნდი ასევე ეძებს გზებს მომავალი სამუშაოს ეფექტურობის ასამაღლებლად. შემდეგ ციკლი მეორდება.

- **Kanban – ეკონომიური მოქნილი მეთოდი**

Kanban – არის პროგრამული უზრუნველყოფის დამუშავების ეკონომიური მეთოდი (Lean method of software development).

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

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

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

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

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

- **Kanban-ის პრინციპები:**

- *სპ1. დაიწყეთ იმით, რასაც ამჟამად აკეთებთ*: რომელი აქტუალური სამუშაოც სრულდება, ჯერ ის უნდა დასრულდეს და მხოლოდ ამის შემდეგ დაიწყოს ახალი სამუშაო;
- *სპ2. დათანხმდით, რომ ევოლუციური ცვლილებები მოსალოდნელია*: შემდგომ განვითარებას აქვს განსაკუთრებული მნიშვნელობა, ოღონდაც აქ სრულყოფა მიღწეულ უნდა იქნას ძირითადად მცირე / ევოლუციური ბიჯების ხარჯზე;

- *სპ3. პატივი ეცით პირველარსებულ პროცესებს / როლებს /ვალდებულებებს*: Kanban ადვილად რეალიზებადია, ყველა როლი, პროცესი და ა.შ., რჩება;
- *სპ4. წაახალისეთ ლიდერობა ორგანიზაციის ყველა დონეზე*: სრულყოფა შესაძლებელია მხოლოდ იმ შემთხვევაში, თუ ამოქმედებულია ორგანიზაციის ყველა დონე. განსაკუთრებით მნიშვნელოვანია ის, რომ სამუშაოს შემსრულებლები უშუალოდ უკეთებდნენ დემონსტრირებას „ლიდერობის აქტებს“ და ახმოვანებდნენ გაუმჯობესების კონკრეტულ წინადადებებს.

❖ **Kanban-ის ძირითადი პრაქტიკები:**

- *ძპ1. სამუშაოს მსვლელობის (ნაკადის) ვიზუალიზაცია:* ღირებულებათა ჯაჭვი პროცესის სხვადასხვა ეტაპებისათვის (მაგალითად, მოთხოვნილების განსაზღვრა, პროგრამირება, დოკუმენტირება, ტესტირება, დანერგვა) კარგადაა ვიზუალიზირებული ყველა მონაწილისათვის. ეს ხორციელდება Kanban-დაფის (Kanban-Board) დახმარებით, რომელზეც სხვადასხვა კვანძები (Stations) აისახება სვეტების სახით (ნახ.2.3).

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

ნახ.2.3. Kanban-ის დაფა (იყენებს Software Development Life Cycle-ს)

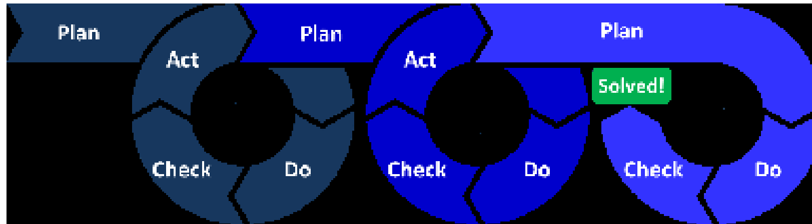
ინდივიდუალური მოთხოვნილებები (ამოცანები, ფუნქციები, მომხმარებელთა ისტორიები, მინიმალური საბაზრო მახასიათებლები და ა.შ.) ჩაიწერება სააღრიცხვო ბარათებში („ბილეთები“, მიმაგრებული დაფის უჯრაზე, Tiket - TK).

ისინი დროის და შესრულებული ბიჯის შესაბამისად გადაადგილდება დაფაზე მარცხნიდან მარჯვნივ;

- *ძპ2. დაწყებული სამუშაოს მოცულობის შეზღუდვა*: ბილეთების რაოდენობა (Work in Progress - WiP), რომლებიც შეიძლება დამუშავებულ იქნას ერთდროულად ერთ კვანძში, შეზღუდულია. მაგალითად, თუ კვანძი „პროგრამირება“ ამუშავებს ორ ბილეთს და ამ კვანძის ლიმიტი 2-ია, მაშინ მას არ შეუძლია მე-3 ბილეთის მიღება, თუნდაც მოთხოვნის განსაზღვრება ამის უფლებას აძლევდეს. ეს ქმნის ამოთრევის-სისტემას (Pull-System), სადაც ყოველ კვანძს გადააქვს თავისი სამუშაო წინა კვანძში, და არ გადასცემს დასრულებულ სამუშაოს მომდევნო კვანძს;
- *ძპ3. ნაკადის მართვა (Manage flow)*: Kanban-პროცესის მონაწილეები ზომავენ ტიპურ მაჩვენებლებს, როგორიცაა რიგის სიგრძე, ციკლის დრო, გამტარუნარიანობა, რათა დაადგინონ თუ რამდენად კარგადაა ორგანიზებული სამუშაო პროცესი, სად შეიძლება მისი სრულყოფა და რა შეიძლება დაპირდეს პარტნიორებს, ვისთვისაც მუშაობენ. ეს აადვილებს დაგეგმვას და ამაღლებს საიმედოობას;
- *ძპ4. პროცესისთვის წესები უნდა გაკეთდეს ცხადად:* იმისათვის, რომ პროცესში მონაწილეებმა იცოდნენ თუ რა მოსაზრებებით და კანონებით მუშაობენ, აუცილებელია რაც შეიძლება მეტი ცხადი წესების გაკეთება. მათ შორის:
  - განმარტებულ იქნას ტერმინი „დასრულებულია“, ასევეა განსაზღვრება „მზადაა Scrum-ში“;

  - Kanban-დაფის თითოეული სვეტის მნიშვნელობა;
  - პასუხები შეკითხვებზე: ვინ მოძრაობს, როდის მოძრაობს, როგორ შეირჩეს შემდეგი ბილეთი არსებულებიდან და ა.შ.;
- *ძპ5. უკუკავშირის ციკლების რეალიზაცია*: ფიქსირებულ თარიღებში გუნდები ერთმანეთს უკავშირდება. მაგალითად:
  - რეტროსპექტივები: თანამშრომლობის მიმოხილვა;
  - მომდევნო შეხვედრა: მომავალი დავალებების შეთანხმება / ბლოკირებების მოხსნა / ნაკადის კოორდინაცია;
  - სამუშაო ოპერაციების რეცენზირება: კომპანიის Kanban გუნდები ხვდება და გამოცდილებას უზიარებს ერთმანეთს;
- *ძპ6. მოდელების გამოყენება პროცესის ერთობლივად გაუმჯობესების შესაძლებლობების დასადგენად:* მოდელები ამარტივებს პროცესს. პოპულარული მოდელია, მაგალითად, ღირებულება, ნაკადი, ნარჩენები „ეკონომიური IT“-დან (Lean IT).

სხვა მოდელები ბაზირებულია *ედუარდ დემინგის* იდეებზე (ციკლი PDCA [Plan-Do-Check-Act] „დაგეგმე, გააკეთე, შეამოწმე, იმოქმედე" (ნახ.2.4) – არის სრულყოფის პროცესი, რომელიც ეფუძნება ცოდნის თეორიის გაგებას და გამოიყენება ხარისხის მენეჯმენტში) ან ვიწრო ადგილების თეორიაზე, სისტემურ აზროვნებაზე ან კომპლექსურობის (სირთულის) თეორიაზე.



ნახ.2.4. PDCA ციკლის იტერაციული განმეორება პრობლემის მოგვარებამდე

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

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

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

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

3. პროგრამული სისტემის შეფასება და ხარისხის მართვა (პრაქტიკული 8, 10)

➢ პროგრამების შეფასების მეტრიკების კლასიფიკაცია და კრიტრიუმები

მეტრიკები ემსახურება განვითარებადი პროგრამის (developing program) სხვადასხვა ასპექტს, გამოყენებული პროცესის მოდელს (used process model) და მოთხოვნების შესრულების შეფასებას (requirements execution assessment). მეტრიკების გამოყენება ვრცელდება პროგრამის განვითარების ფაზების შეფასებიდან: ფაზის შედეგების შეფასებით – გამოყენებული ტექნოლოგიების შეფასებამდე. მეტრიკების მიზანი პროგრამული უზრუნველყოფის შემუშავებისას არის შეცდომების პროგნოზირება და ხარჯების შეფასება, რის მიხედვითაც განასხვავება ხდება წინა, პარალელურ- და რეტროსპექტულ გამოყენებას შორის.

❖ შეზღუდვა

ძირითადად, მეტრიკები, რომლებიც მართვადია, არის ერთგანზომილებიანი. ეს ამარტივებს მათ გამოყენებას. როგორც წესი, იგი მიიღწევა თითოეული მეტრიკის ერთი ხედვით (view) შემცირებით. ეს ნიშნავს, რომ სხვა ხედვები ამავდროულად არ განიხილება იმავე ხარისხით.

1. *მენეჯმენტის ხედვა:*
   - პროგრამული უზრუნველყოფის განვითარების ხარჯები (შეთავაზება, ხარჯების მინიმიზაცია);
   - პროდუქტიულობის გაზრდა (ბიზნეს-პროცესები ან ბიზნეს-მეთოდები, გამოცდილების მრუდი);
   - რისკები (ბაზრის პოზიცია, დრო ბაზარზე – ახალი პროდუქტის 1-ელი გამოშვებიდან მის ფართოდ დამკვიდრებამდე);
   - სერტიფიკაცია (მარკეტინგი).
2. *დეველოპერის ხედვა:*
   - კითხვადობა – Readability (მომსახურება, განმეორებითი გამოყენება);
   - ეფექტურობა და შედეგიანობა;
   - *ნდობა* (trust) – *შეცდომის კოეფიციენტი* (ErrorQuotient) ხარისხის მენეჯმენტში (ან შეცდომის სიხშირე) არის დეფექტური ელემენტების ფარდობითი წილი მთლიანთან მიმართებაში, ანუ ის ფარდობითი სიხშირე, რომელთან დაკავშირებითაც ხდება შეცდომა პროდუქტის, სერვისის, წარმოების პროცესის ან მუშაობის ხარისხში]; *MTBF* (Mean time between failures – საშუალო დრო მტყუნებებს შორის (სთ), მუშაობის დრო მტყუნებამდე (საიმედოობის მაჩვენებელი));, *ტესტები*);
3. *დამკვეთის ხედვა:*
   - ხარჯთაღრიცხვა (ბიუჯეტის დაცვა, მიწოდების თარიღების დაცვა);
   - ხარისხი (საიმედოობა, სისწორე);
   - ინვესტიციის დაბრუნება (თანხლება, გაფართოება).

❖ **კლასიფიკაცია**

შეფასების სხვადასხვა ასპექტისთვის არსებობს დიზაინის (პროექტირების) მეტრიკა, ეკონომიკური მეტრიკა, საკომუნიკაციო მეტრიკა და ა.შ. მეტრიკები შეიძლება განისაზღვროს სხვადასხვა კლასისათვის, რომელიც მიუთითებს გაზომვის ან შეფასების ობიექტზე:

1. *პროცესის მეტრიკა:*
   - რესურსების ხარჯი (თანამშრომლები, დრო, ხარჯები);
   - შეცდომა;
   - საკომუნიკაციო ხარჯები.
2. *პროდუქტის მეტრიკა:*
   - ზომა (კოდის სტრიქონები, განმეორებითი გამოყენება, პროცედურები ...);

- გამოთვილითი სირთულე – არის ინფორმატიკაში ალგორითმების თეორიის ცნება, რომელიც გულისხმობს სამუშაოს მოცულობის დამოკიდებულების ფუნქციას საწყის (შესატან) მონაცემთა რაოდენობასთან [26];
- კითხვადობა (სტილი);
- დიზაინის ხარისხი (მოდულარობა, ერთიანობა, შეკავშირება ...);
- პროდუქტის ხარისხი (ტესტის შედეგები, ტესტური გადაფარვა [27]...).

3. *ხარჯების მეტრიკა:*
   - *ხარჯების სტაბილურობა;*
   - *ხარჯების განაწილება;*
   - *პროდუქტიულობა;*
   - *ხარჯები - დროული მიწოდება.*
4. *პროექტის ხანგრძლივობის მეტრიკა:*
   - განვითარების (დეველოპმენტის) დრო;
   - განვითარების საშუალო დრო;
   - ეტაპების შესრულების ანალიზი (Milestone Trend Analyse [28]) - პროექტების შესრულების გრაფიკი ეტაპების მიხედვით (პროექტების მენეჯმენტში);
   - პუნქტუალობა (დროული შესრულება).
5. *კომპლექსურობის (სირთულის) მეტრიკა* :
   - პროგრამული უზრუნველყოფის ზომა;
   - დასრულების ხარისხი.
6. *აპლიკაციის მეტრიკა:*
   - ტრეინინგის ხარჯი;
   - მომხმარებლის კმაყოფილება

❖ ხარისხის კრიტერიუმები

მხოლოდ პროგრამული უზრუნველყოფის წარმოების ფაზიდან მიღებული მეტრიკა არ არის ხარისხის კრიტერიუმი. როგორც წესი, ხარისხის მახასიათებლები ფასდება მომხმარებლის მოთხოვნების შესრულებისა და მათი გამოყენების შესაბამისად. შედეგების გადაცემა და გაზომილი მნიშვნელობების ასახვა მნიშვნელოვანია მომხმარებლისათვის:

- *ობიექტურობა*: არ აქვს სუბიექტური გავლენა მზომავი პირისგან;
- *საიმედოობა:* განმეორებისას იგივე შედეგების მიღება;
- *სტანდარტიზაცია:* გაზომვის შედეგების სკალა და შესაბამისობის (შედარების) სკალა;
- *შესაბამისობა:* ზომა (measure - გაზომვა) შეიძლება დადგეს სხვა ზომებთან მიმართებაში;
- *ეკონომიურობა:* მინიმალური ხარჯები;
- *სარგებლობა:* პრაქტიკული მოთხოვნილებების შესრულების გაზომვა;
- *ვალიდურობა:* გაზომვის სიდიდეებიდან სხვა პარამეტრებამდე (სირთულე).

➢ პროგრამული სისტემების რაოდენობრივი შეფასების მეტრიკები

მეტრიკები – ინსტრუმენტებია, რომლებიც მიზნად ისახავს გადაწყვეტილების მიღების პროცესის გამარტივებას, მწარმოებლურობის და პასუხისმგებლობის დონეთა ამაღლებას, შესაბამისად, დასმული პრობლემის და მისი გადაწყვეტის ამოცანებთან დაკავშირებული მონაცემების შეგროვების, დამუშავების და ასახვის მეთოდების საფუძველზე.

შედარებით გამოყენებადი და ცნობილი მეტრიკები შემდეგია:

- *კოდის სტრიქონების რაოდენობა* (Lines Of Code – LOC). გამოიყენება პროგრამის საწყისი ტექსტის მოცულობის დასადგენად მისი სტრიქონების რაოდენობის განსაზღვრის საფუძველზე. ეს მაჩვენებელი

გამოიყენება პროგრამის შემუშავების დანახარჯების პროგნოზირების მიზნით დაპროგრამების კონკრეტულ ენაზე, ან შრომის ნაყოფიერების შესაფასებლად პროგრამის შედგენის შემდეგ;

- *ფუნქციონალური წერტილების ანალიზი* (Function Point Analysis – FPA) – არის მეთოდი, რომელიც რაოდენობრივად განსაზღვრავს პროგრამაში მოცემულ ფუნქციებს იმ ტერმინებით, რომლებიც მნიშვნელოვანია პროგრამის მომხმარებლებისათვის. FP ითვალისწინებს სპეციფიკაციის მოთხოვნების შესაბამისად დამუშავებული (განვითარებული) ფუნქციების რაოდენობას;
- *განვითარების მოდელის ღირებულება* (COnstructive COst MOdel – COCOMO) არის პროგრამული უზრუნველყოფის შემუშავების ხარჯთაღრიცხვის ალგორითმი, რომელიც შეიმუშავა ამერიკელმა მეცნიერმა ბ. ბოემ (B. Boehm - სამხრეთ კალიფორნიის უნივერსიტეტი, პროგრამული ინჟინერის პროფე-სორი). მოდელი იყენებს მარტივი რეგრესიის ფორმულას, რიგი პროექტებიდან შეგროვილი მონაცემების მიხედვით განსაზღვრული პარამეტრებით.
- *ციკლომატიურობის სირთულე* (Cyclomatic complexity) არის პროგრამის სირთულის გასაზომი მეტრიკა, მისი სტრუქტურული (ან ტოპოლოგიური) საზომი. პროგრამის საწყის კოდში (ბლოკსქემის ანალოგიურად) იზომება წრფივი, დამოუკიდებელი გზების რაოდენობა დასაწყისიდან დასასრულამდე. იგი შეიმუშავა თომას ჯ. მაკკეიამ 1976 წელს [32]; თუ პროგრამაში არაა ციკლი (მაგალითად, for..., while...), ან პირობითი განშტოება (if...else...), მაშინ ციკლომატიური სირთულე 1-ის ტოლია. თუ არის ერთი if...else... ბლოკი, მაშინ სირთულე 2-ია (ორი გზა: 1 – true და 2-false -სკენ) და ა.შ. ციკლომატური სირთულე ასევე შეიძლება გამოთვლილ იქნას პროგრამის ფარგლებში ინდივიდუალური ფუნქციების, მოდულების, მეთოდებისა თუ კლასებისთვის. სტრუქტურული პროგრამის მათემატიკური ციკლომატური სირთულე განისაზღვრება ორიენტირებული გრაფის საშუალებით, რომლის წვეროები პროგრამის ბლოკებია, შეერთებული წიბოებით, თუ მართვა შეიძლება გადავიდეს ერთი ბლოკიდან მეორეზე. ამ შემთხვევაში სირთულის განსაზღვრა ხდება ფორმულით:

$$M = E - N + 2P,$$

სადაც:

M - ციკლომატური სირთულეა,
E - წიბოების რაოდენობა გრაფაში,
N - წვეროების რაოდენობა გრაფაში,
P - კავშირის კომპონენტების რაოდენობა.

პროგრამის ტესტირების პროცესში ციკლომატური სირთულე შეიძლება გამოყენებულ იქნას მეორე ამოცანისათვისაც, კერძოდ, რამდენი ტესტი იქნება საჭირო დასატესტი კოდის სრულად დასაფარად.

- *ჰოლსტედის სირთულის ზომები* (Halstead Complexity Measures) – ესაა პროგრამული უზრუნველყოფის რეალიზაციის (დანერგვის) სირთულის შეფასება, წინასწარ, პროექტირების (დიზაინის) ეტაპზე. პროგრამული სირთულის გაზომვის ერთ-ერთი სტატიკური, ანალიტიკური მეთოდია, რომელიც შემოიტანა მ. ჰოლსტედმა (M. Halstead,) 1977 წელს. მისი კონცეფცია მდგომარეობდა პროგრამული უზრუნველყოფის დეველოპმენტის ემპირიული მეცნიერების შექმნის შესახებ. მან დაადგინა, რომ პროგრამული უზრუნველყოფის მეტრიკები უნდა ასახავდეს რეალიზაციას (დანერგვას) ან ალგორითმების გამოსახვას სხვადასხვა ენაზე, მაგრამ დამოუკიდებელი უნდა იყოს მათი შესრულებიდან კონკრეტულ პლატფორმაზე. ეს მეტრიკები, ამიტომაც, გამოითვლება კოდიდან სტატიკურად. ჰოლსტედის მიზანი იყო პროგრამული უზრუნველყოფის გაზომვადი თვისებების და მათ შორის ურთიერთობების დადგენა.

ჰოლსტედის მეტრიკები იყენებს ვარაუდს, რომ შესრულებადი პროგრამის ნაწილები შედგება ოპერატორებისა და ოპერანდებისაგან. მაგალითად, ცვლადები და მუდმივები განიხილება, როგორც ოპერანდები; საკვანძო სიტყვები, ლოგიკური და შედარებითი ოპერატორები და ა.შ., როგორც ოპერატორები.

შემდეგ, თითოეული პროგრამისთვის ფორმირდება ასეთი ძირითადი ზომები:

– $\eta 1$ = განსხვავებული ოპერატორების რაოდენობა;

– η_2 = განსხვავებული ოპერანდების რაოდენობა;
– N1 = ოპერატორთა საერთო რაოდენობა;
– N2 = ოპერანდების საერთო რაოდენობა

ამ რიცხვებიდან შეიძლება გამოითვალოს რამდენიმე მაჩვენებელი:

– პროგრამის ლექსიკა: $\eta = \eta_1 + \eta_2$
– პროგრამის სიგრძე: $N = N_1 + N_2$
– გაანგარიშებული პროგრამის სიგრძე: $\hat{N} = \eta_1 * \log_2 \eta_1 + \eta_2 * \log_2 \eta_2$
– მოცულობა: $V = N * \log_2 \eta$
– სირთულე: $D = \frac{\eta_1}{2} * \frac{N_2}{\eta_2}$
– დანახარჯი: $E = D \times V$

სირთულის ზომა უკავშირდება პროგრამის დაწერის სირთულეს ან მის გაგებას, მაგალითად, კოდის მიმოხილვის დროს. დანახარჯების ზომა გადაიყვანება კოდის ფაქტობრივი დაწერის დროში შემდეგი დამოკიდებულებით;

$$T = \frac{E}{18}$$

ჰოლსტედმა ექსპერიმენტულად დაადგინა, რომ ხარჯების რაოდენობის გაყოფით 18-ზე იძლევა დროის მიახლოებით მნიშვნელობას წამებში. ამ ლიტერატურულ წყაროში განიხილება კოდის სირთულის განსაზღვრის საზომი საშუალებები: C, C ++, Java და C # ენებისათვის.

გაითვლება ასევე მიღებული შეცდომების რაოდენობა (B), რომელიც შეესაბამება პროგრამის მთლიან სირთულეს. აქ ჰოლსტედი იძლევა შემდეგ გამოსახულებას წარმოებული შეცდომების რაოდენობისთვის:

$$B = \frac{E^{\frac{2}{3}}}{3000}$$

ან ბოლო პერიოდის შემთხვევისთვის:

$$B = \frac{V}{3000}$$

- *მართვის ნაკადზე ორიენტირებული მეტრიკები* (Control Flow Oriented Metrics). მართვის ნაკადი (control flow) პროგრამირებაში არის გამოთვლების (შესრულების) მიმდევრობის მოწესრიგების ხერხი (მაგალითად, განშტოება, ციკლი და სხვ.). იგი სტრუქტურული პროგრამირების ერთ-ერთი ძირითადი საკითხია. ნაკადების მართვაზე ორიენტირებული პროგრამების ტესტირების მეტრიკები (ან გადაფარვის მეტრიკები - Coverage metrics) შეისწავლის: ოპერატორების გადაფარვას (statement coverage), შტოების გადაფარვას (branch coverage), გზების გადაფარვას (path coverage) და პირობების გადაფარვას (condition coverage). ასეთი ტესტირების მეთოდები ეფუძნება პროგრამის ნაკადების მართვის გრაფებს, მაგალითად, ტესტირების თეთრი-ყუთის სახით, ვინაიდან პროგრამის სტრუქტურა აქ ცნობილი უნდა იყოს;
- არსებული მეტრიკების კომბინაციით ვითარდება ახალი მეტრიკები, რომელთა ნაწილი აისახება პროგრამულ ინჟინერიაში. ამის მაგალითია C.R.A.P. (Change, Risk, Analysis, Predictions) (შეცვლა. რისკი. ანალიზი. პროგნოზი) მეტრიკა, რომ შენარჩუნდეს არსებული კოდის ტანი. Unit-ტესტირებისას, მაგალითად, C.R.A.P.-ინდექსი ასახავს თუ რამდენად არის unit-ტესტით დაფარული კოდი. რაც მეტადაა დაფარული, მით მცირეა C.R.A.P.-ინდექსის მნიშვნელობა. მისი თვითნებურად შემცირებამ შეიძლება გამოიწვიოს სისტემის სირთულის გაზრდა, რაც არაა სასურველი;
- *ინფორმაციის უსაფრთხოების მეტრიკა* (Information Security Metrics) – გამოიყენება სისტემებსა და ინფრასტრუქტურებში ინფორმაციის უსაფრთხოების დონის შესაფასებლად, შესაძლებელი უნდა იყოს უსაფრთხოების გაზომვა. უსაფრთხოების ინდიკატორები ობიექტური რაოდენობრივი საზომია, რომლის საფუძველზე შესაძლებელია გადაწყვეტილების მიღება უსაფრთხოების შესახებ, როგორც მონაცემთა შეგროვების ეტაპზე, ასევე ექსპლუატაციის დროს.

ლიტერატურულ და ინტერნეტულ წყაროებში მრავლადაა გადმოცემული უსაფრთხოების მეტრიკების მნიშვნელობა, განსაკუთრებით ორგანიზაციული მართვის (კორპორაციების) სფეროში. ამ

ნაშრომში მოცემულია უსაფრთხოების მეტრიკების მიმოხილვა და მისი განსაზღვრება, მოთხოვნილებები, ატრიბუტები, უპირატესობები, ზომები, ტიპები, პრობლემები/ასპექტები, აგრეთვე კლასიფიკაციის საკითხები. განხილულია უსაფრთხოების მეტრიკების ურთიერთობა რისკის მენეჯმენტთან და ა.შ.

უსაფრთხოების მეტრიკები, რომელთა საზომია მწარმოებლურობა, ძირითადად, კლასიფიცირდება ორ ჯგუფში:

1) უსაფრთხოების მეტრიკები, დაკავშირებული ეფექტურობასთან (შეაფასოს, თუ რამდენად არის მიზნები მიღწეული;

2) უსაფრთხოების მეტრიკები, დაკავშირებული შედეგიანობასთან (რაც ასახავს პროპორციულობას მიღწეულ მიზნებსა და მიღებულ შედეგებს შორის).

უსაფრთხოების მეტრიკის გამოყენება ადასტურებს, რომ ორგანიზაცია იყენებს პროაქტიურ (proactive) უსაფრთხო მეთოდებს. უსაფრთხოების ეს მეტრიკები ინფორმაციას უწევს ორგანიზაციაში დანერგილი პროცესების, პროცედურების და კონტროლის საშუალებების ეფექტიანობას.

➢ პროგრამული აპლიკაციების ტესტირება

განხილულია პროგრამული კოდების ტესტირების პროცესი Visual Studio.NET ინტეგრირებულ 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.

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.

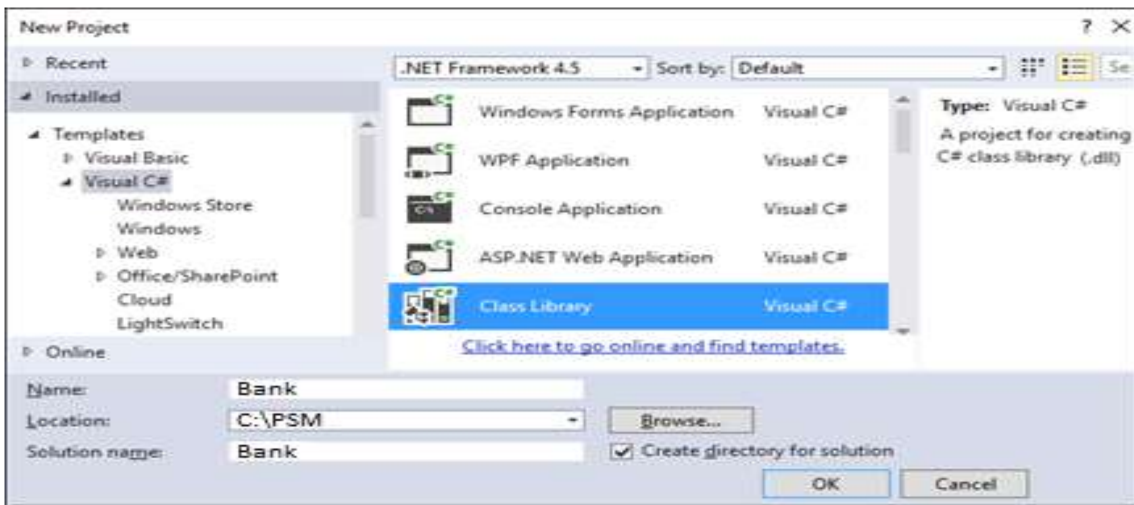
❖ დასატესტი პროგრამის პროექტის შექმნა

Visual Studio.NET-ში საჭიროა ავირჩიოთ: **File -> New -> Project.**

შედეგად გამოჩნდება დიალოგური ფანჯარა (ნახ.3.1). სადაც ავირჩევთ:

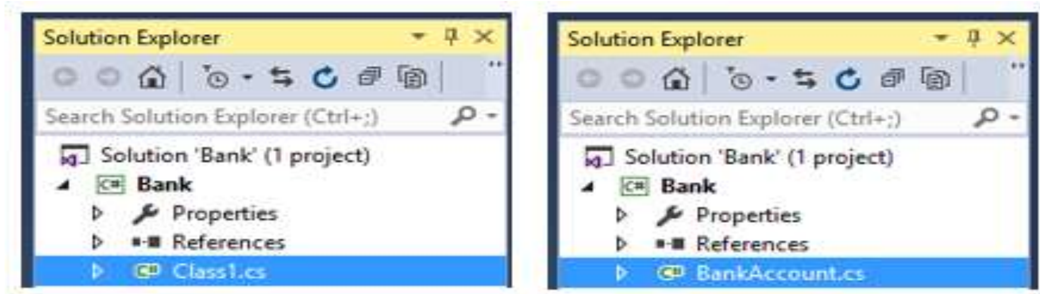
Visual C# => ClassLibrary

პროექტი სახელით Bank.



ნახ.3.1. დასატესტი კლასის პროექტის შექმნა

მიიღება 3.2 ნახაზზე ნაჩვენები Solution Explorer ფანჯარა. აქ Class1.cs სახელი შევცვალოთ BankAccount.cs -ით.



ნახ.3.2. Class1 - > BankAccount

შემდეგ BankAccount.cs-ის ტექსტი რედაქტორის არეში შევცვალოთ ჩვენი დასატესტი პროგრამის კოდით.

ეს საწყისი ტექსტი, მაგალითად, მოცემულია 3.1 ლისტინგში.

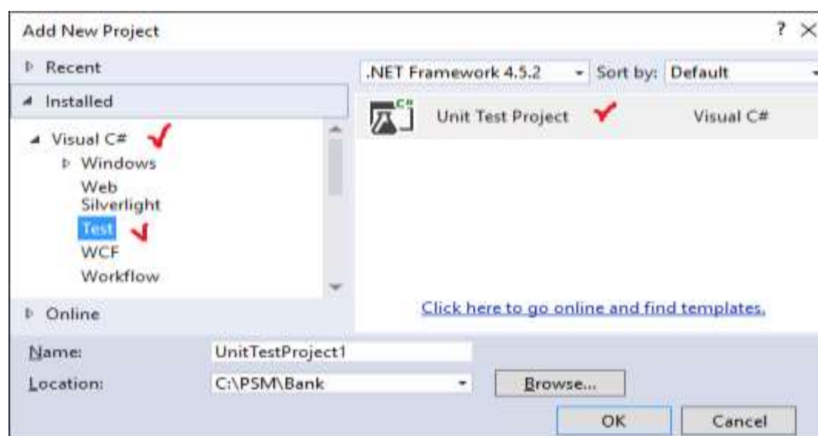
```
//-- ლისტინგი_3.1 --- BankAccount.cs --------
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

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);
  }
} }
```

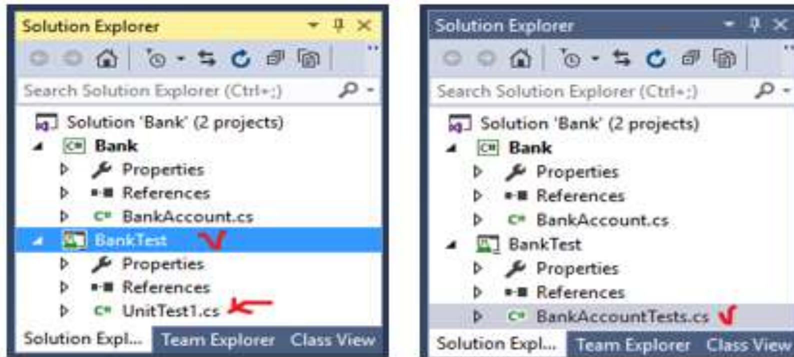
❖ Unit ტესტ-ფაილის პროექტის აგება

დავამატოთ ახალი პროექტი. Visual Studio-ში საჭიროა ავირჩიოთ: **File -> New -> Project.**



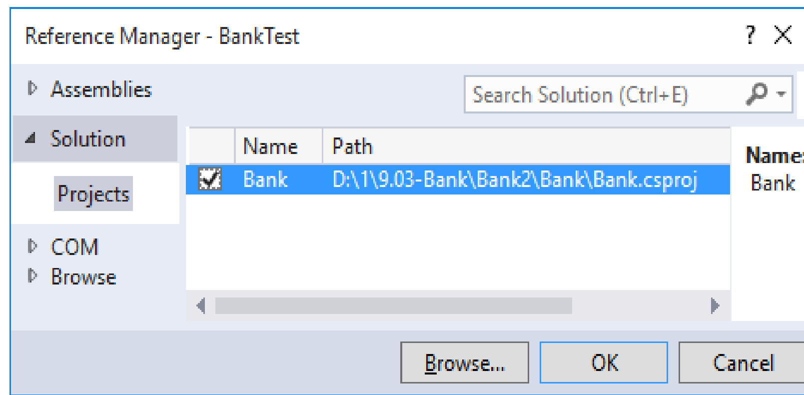
ნახ.3.3. Unit Test პროექტის შექმნა

მივიღებთ 3.4 ნახაზზე ნაჩვენებ სურათს BankTest პროექტით. მარჯვენა სურათზე შეცვლილია UnitTest კლასის სახელი BankAccountTests-ით.



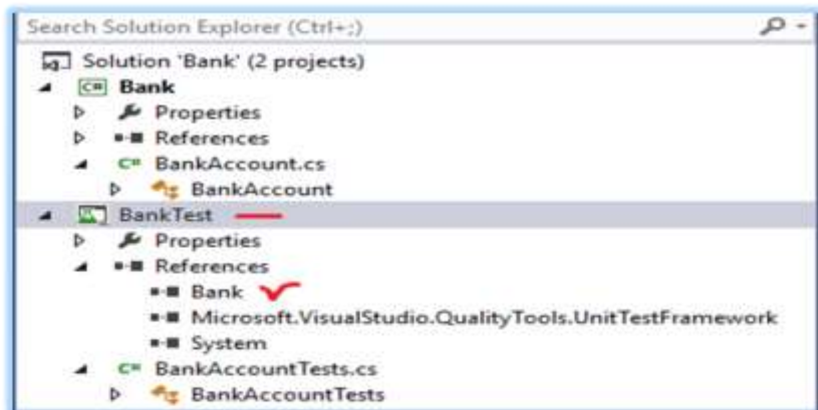
ნახ.3.4

BankTests პროექტში დავამატოთ reference **Bank** solution-იდან. ამისათვის **BankTests**-ზე მაუსის მარჯვენა ღილაკით ავირჩიოთ **Add Reference** და მივიღებთ 3.5 ნახაზზე ნაჩვენებ ფანჯარას. აქ Solution სტრიქონში ვირჩევთ Projects და Bank-ის ჩეკბოქსს მოვნიშნავთ.



ნახ.3.5

მივიღებთ 3.6 ნახაზზე ნაჩვენებ შედეგს.



ნახ.3.6

ჩავამატოთ BankAccountTests პროგრამაში სახელსივრცე Bank-ის პროექტიდან: using Bank;

ამგვარად, BankAccountTests.cs ფაილს ექნება 3.2 ლისტინგზე ნაჩვენები სახე.

```
//-- ლისტინგი_3.2 -----  BankAccountTests.cs -----
using System;
```

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Bank;

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 - (რასაც ჩვენ ველოდებით).

ტესტ-მეთოდის მოთხოვნები ასეთია:

1) მეთოდი მონიშნული უნდა იყოს [TestMethod] ატრიბუტით;
2) მეთოდმა უნდა დააბრუნოს void;
3) მეთოდს არ შეიძლება ჰქონდეს პარამეტრები.

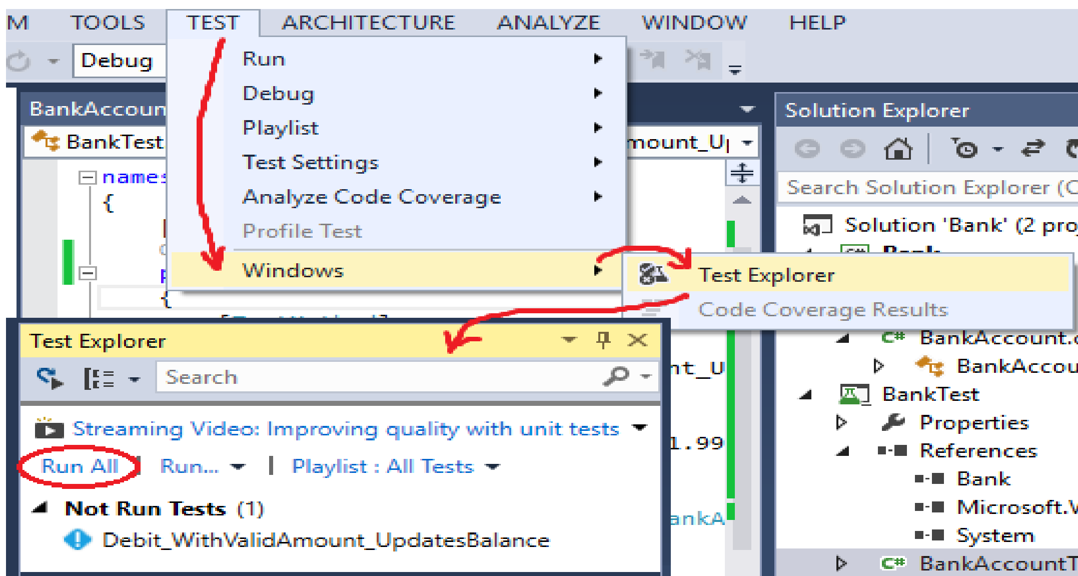
❖ ტესტის კოდის ამუშავება და შედეგის ფორმირება

მთლიანი ტესტის კოდი მოცემულია 3.3 ლისტინგში.

```
//-- ლისტინგი_3.3  ----- BankAccountyTests.cs ----
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Bank;
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.3.7).

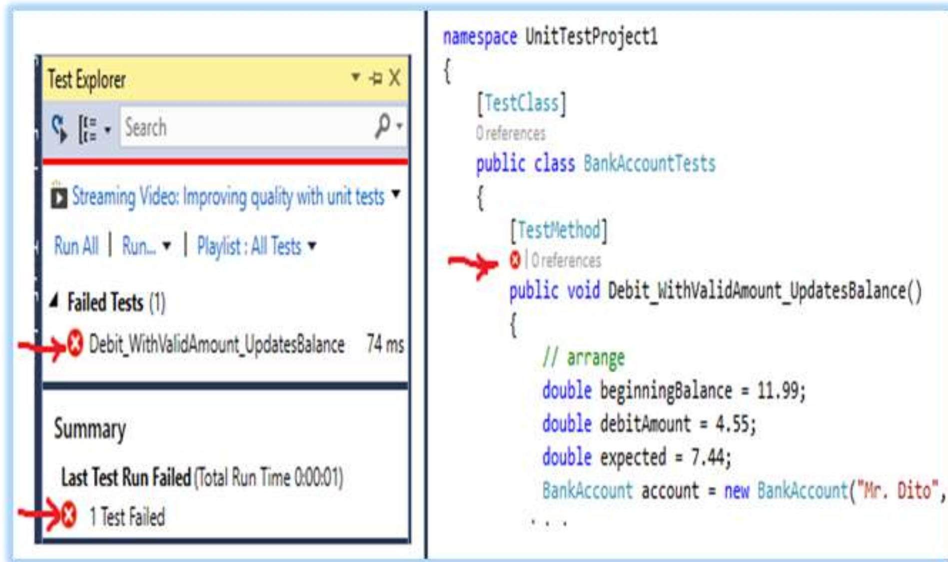


ნახ.3.7

აქ ვირჩევთ Run All - სა და ვიღებთ შედეგს (ნახ.3.8).

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.



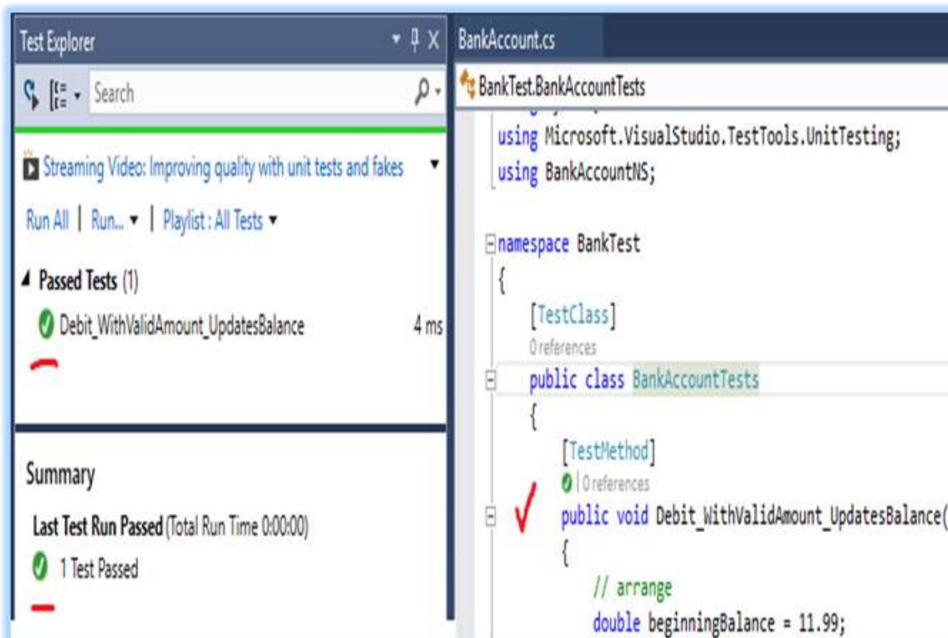
ნახ.3.8

შემდეგი ეტაპი კოდის გასწორება და ხელახალი ტესტირებაა. დასატესტ პროგრამაში შევცვალოთ სტრიქონში „ + „ ნიშანი „ – „ -ით (ნახ.3.9).

```
// m_balance += amount; // intentionally incorrect code
m_balance -= amount; // intentionally correct code
```

ნახ.3.9

ტესტის თავიდან ამუშავებით ვიღებთ წარმატებულ შედეგს ანუ მიიღება მწვანე ფერის სიმბოლოები (ნახ.3.10).



ნახ.3.10. სწორი შედეგი

➢ პროგრამული კოდების ხარისხის შეფასება

განხილულია პროგრამული პროექტების შემუშავების პროცესში აპლიკაციის მოდულების ხარისხის შეფასების მეთოდები.

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.

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:

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 მაუსის 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.3.11).

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.3.11

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.

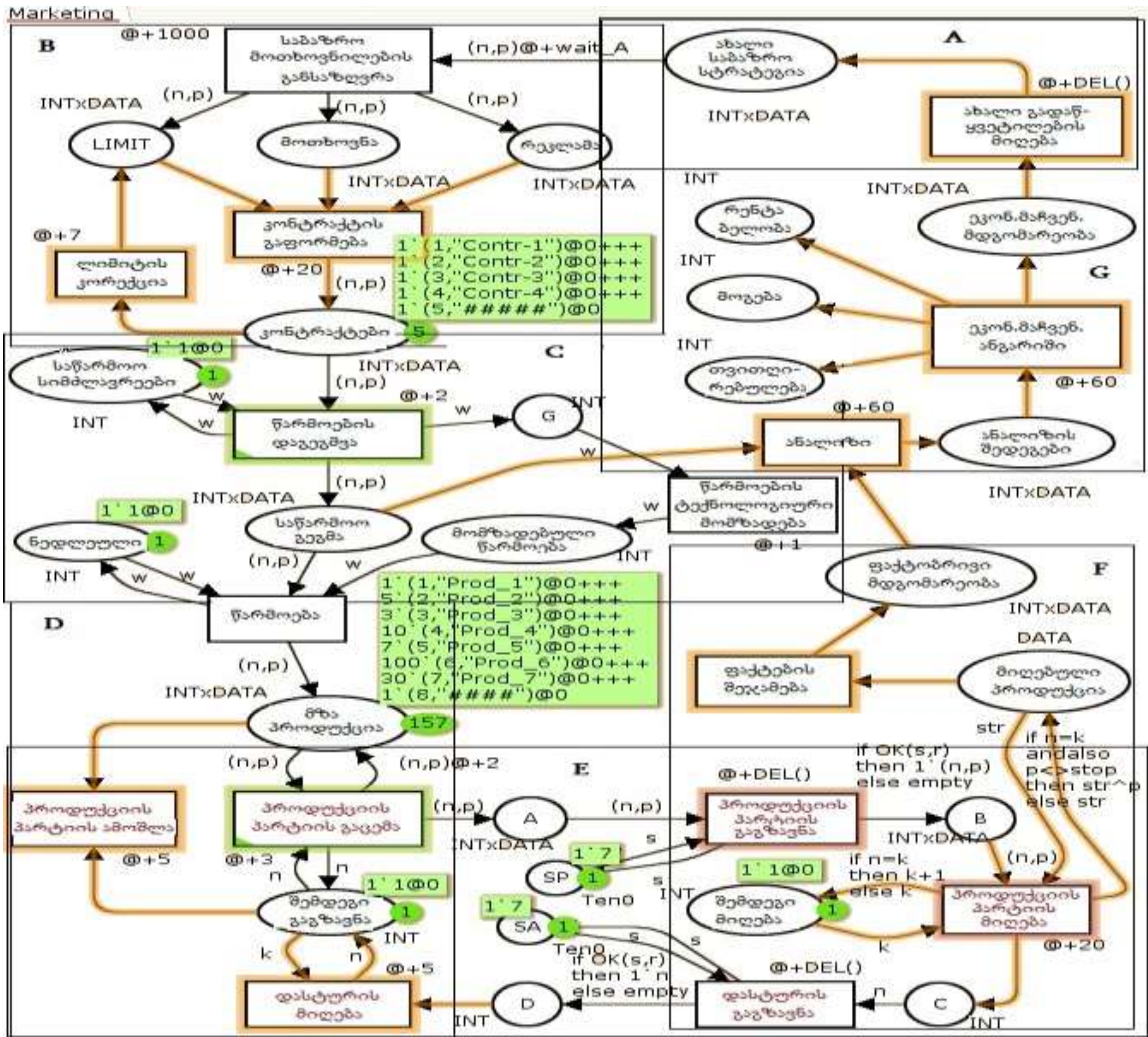
ნაწილში (Warnings) ველი suppress (გაბათილებადია) მიანიშნებს, კოდის რომელი ელემენტია გამოცხადებული, თუმცა გამოუყენებელი.

4. კვლევის ობიექტის იმიტაციური მოდელის აგება პეტრის ფერადი ქსელებით და მისი პროგრამული რეალიზაცია კლიენტ-სერვერული არქიტექტურით (პრაქტიკული 11, 13)

➢ მარკეტინგული პროცესების მოდელირება პეტრის ფერადი ქსელებით

განვიხილოთ მარკეტინგული პროცესების მოდელირების საილუსტრაციო მაგალითი პეტრის ფერადი ქსელების (CPN-Coloured Petri Net) ინსტრუმენტის გამოყენებით.

პროდუქციის საწარმოო ფირმის მარკეტინგული პროცესების მოდელირებისათვის გვაქვს შემდეგი ძირითადი იერარქიული მოდულები (ნახ.4.1):



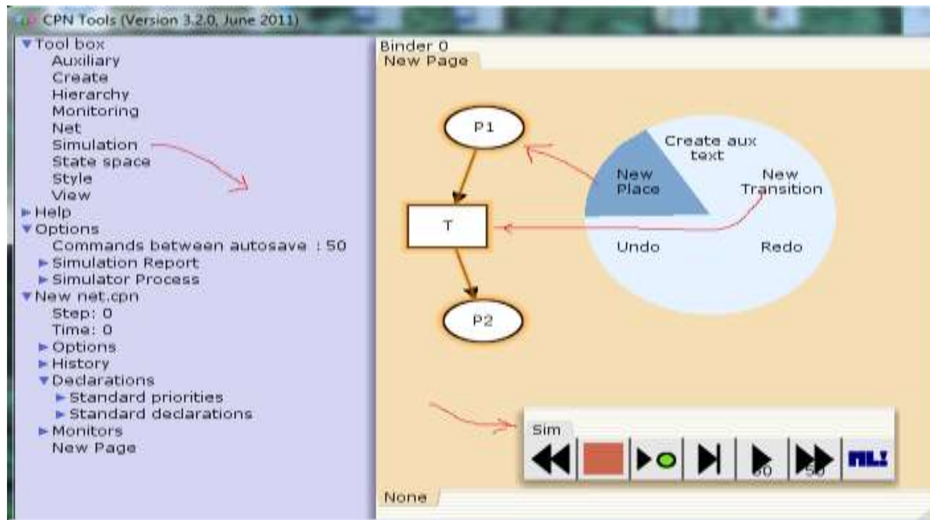
ნახ.4.1. მარკეტინგული პროცესების პეტრის ქსელის მოდელი (CPN)

ახალი საბაზრო სტრატეგიის ფორმირება (A), საბაზრო მოთხოვნების განსაზღვრა (B); პროდუქციის წარმოების დაგეგმვა (C); წარმოების ტექნიკური მომზადება და პროდუქციის წარმოება (D); პროდუქციის გაცემა (სასაწყობო მეურნეობა) და პროდუქციის გადაგზავნა (ტრანსპორტირება) (E), პროდუქციის მიღება და დამკვეთის ინფორმირება (F); ფაქტობრივი მდგომარეობის აღრიცხვა, საწარმოო და სარეალიზაციო გეგმების შესრულების ანალიზი, ეკონომიკური მაჩვენებლების ანგარიში და ანალიზი (G); გადაწყვეტილების მიღება ახალი საბაზრო სტრატეგიისათვის (A) და ა.შ. ციკლურად.

ჩვენი მიზანია ზემოაღწერილი იერეარქიული მოდულებიდან გამოვყოთ, მაგალითად, E ბლოკი, რომელიც აღწერს პროდუქციის მიწოდების პროცესს კლიენტებზე და მოვახდინოთ „მიმწოდებელი–დამკვეთი” პროცესის მოდელირება შეტყობინებების გაცვლის იმიტაციით, კლიენტ–სერვერ არქიტექტურის პრინციპების საფუძველზე (შემდეგ კი დავაპროგრამოთ ეს ბიზნესპროცესი).

ბიჯი_1) მოკლედ განვიხილოთ CPN ინსტრუმენტის სამუშაო გარემო, მისი დახმარებით მოდელის აგების და შემდეგ ბიზნესპროცესის იმიტაციური რეჟიმის ამოქმედება.

4.2 ნახაზზე ნაჩვენებია CPN-ის მომხმარებლის საწყისი ინტერფეისი (ვუშვებთ, რომ CPN პაკეტი დაინსტალირებულია კომპიუტერში. იგი უფასო ვერსიაა და ფართოდ გამოიყენება აშშ, ევროპის, ჩინეთისა და სხვა ქვეყნების უნივერსიტეტებში).



ნახ.4.2. CPN სამუშაო გარემო

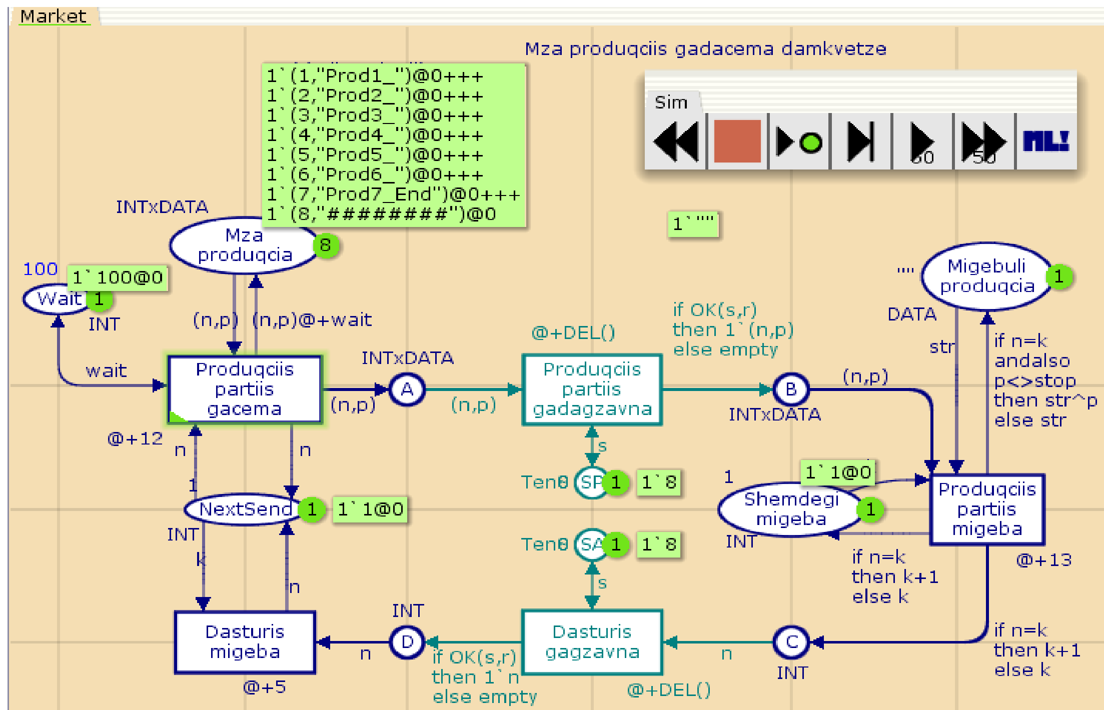
პეტრის ქსელის პოზიციების (Places), გადასასვლელების (Transitions) და რკალების (Arcs) აგება (მაუსის მარჯვენა ღილაკით), შემდეგ მარკერების დამატება და იმიტაციური პროცესის (Simulation) ამუშავება მარტივად ხორციელდება.

ნახაზის მარცხენა ნაწილში ჩანს პეტრის ქსელის კონკრეტული პარამეტრების მნიშვნელობები. მაგალითად, საწყის მდგომარეობაში პოზიცია „მზა პროდუქცია“ შეიცავს INTxDATA ტიპის ფერად მარკერთა 8-ელემენტიან სიმრავლეს (საინიციალიზაციო მარკირება) (ნახ.4.3).

```
▼Marketing-4.cpn
   Step: 0
   Time: 0
  ►History
  ▼Declarations
    ▼colset INT =int timed;
    ▼colset DATA = string;
    ▼colset INTxDATA = product INT*DATA timed;
    ▼var n, k, w, d, wait: INT;
    ▼var p, str: DATA;
    ▼val wait_A=8760;
    ▼val stop = "#####";
    ▼colset Ten0 = int with 0..12;
    ▼colset Ten1 =int with 1..12;
    ▼var s: Ten0;
    ▼var r: Ten1;
    ▼fun OK(s:Ten0, r:Ten1) = (r<=s);
    ▼colset NetDelay = int with 25..75;
    ▼fun DEL() = NetDelay.ran();
```

ნახ.4.3. CPN-ის პროგრამული ენის ფრაგმენტი

4.4 ნახაზზე ნაჩვენებია აგებული E ბლოკის (მარტივი შემთხვევის) მაგალითი, რომლისათვისაც ჩავატარეთ კონკრეტული ექსპერიმენტები.



ნახ.4.4. „პროდუქციის მიწოდების“ ბიზნესპროცესის პეტრის ქსელის ფრაგმენტი (საწყისი პოზიცია)

{1`(1,”Prod1“), 1`(2, "Prod2“), 1`(3, "Prod3“), 1`(4, "Prod4“), . . . , 1`(8, „##### “) }. აქ ბოლო, მე-8 ელემენტი შეესაბამება დასასრულის იდენტიფიკაციას -Stop.

„1“-იანი ყოველი ელემენტის დასაწყისში (მას კოეფიციენტი ეწოდება), რომელიც მიუთითებს, რომ პოზიციაშია არაუმეტეს 1 ცალი მოცემული ფერის მონაცემი (ანუ არსებობს მხოლოდ ერთი პროდუქტი ნომრით „Prod1“, რომლის ფერი - რიგითი ნომერია 1). ამ შემთხვევაში გვაქვს მონაცემთა ელემენტების *სიმრავლე*.

4.1 ნახაზზე ნაჩვენები E ბლოკის რთული შემთხვევისათვის გვექნებოდა 157 ელემენტი (1+5+3+10+7+100+30+1), რომლებიც 7 სხვადასხვა (მარკერების ფერის) დამზადებული პროდუქტის რაოდენობას, ანუ *მულტისიმრავლეს* ასახავს.

4.4 ნახაზზე პროცესების შესრულების დრო (დაყოვნება) აისახება გადასავლელთან სიმბოლოს და დროის ერთეულის (მაგალითად, @+12, @+wait) მითითებით, სადაც wait წინასწარ განსაზღვრული კონსტანტაა.

ამავე ნახაზზე ასახულია არადეტერმინირებული ლოგიკური გამოსახულება (პირობის ბლოკი) ფერადი პეტრის ქსელის რკალებზე, რომელიც გადასასვლელთა გაშვების სხვადასხვა პირობებს და შედეგებს ასახავს, ანუ ლოგიკური პირობის ჭეშმარიტებისას გადასასვლელს განსხვავებული მნიშვნელობა მიეწოდება (ან გადასასვლელიდან განსხვავებული მნიშვნელობა გამოვა), მცდარობისას – განსხვავებული.

მაგალითად, გადასასვლელს „პროდუქციის პარტიის გადაგზავნა“ გამოსასვლელ რკალზე აქვს ლოგიკური პირობა - თუ გამოგზავნილი პროდუქციის ნომერი (n) ემთხვევა კლიენტის კონტრაქტით მისაღებ პროდუქციის ნომერს (k), მაშინ გვაქვს “true”, წინააღმდეგ შემთხვევაში ”false“, რაც იმას ნიშნავს, რომ საჭირო პროდუქცია არაა მოსული. თუ ყველაფერი წესრიგშია, მაშინ მიმღები უგზავნის მწარმოებელს შეტყობინებას გადასასვლელით „დასტურის გამოგზავნა“.

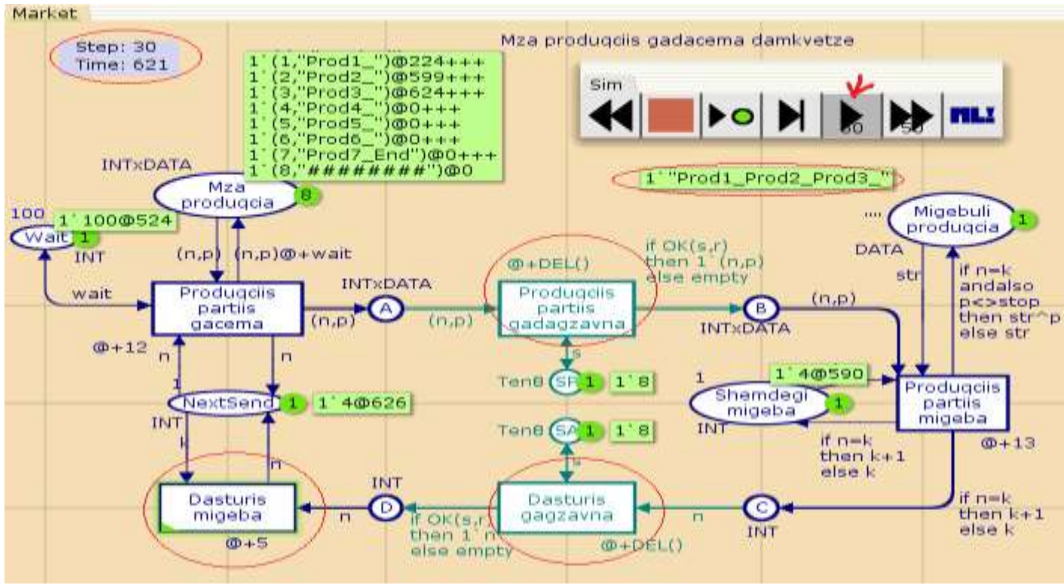
პროდუქციის და შეტყობინების გადაცემათა ქსელში ***შემთხვევითი პროცესის*** არსებობა განპირობებულია დაყოვნების ცვლადი დროის გამო, რაც აისახება colset NetDelay=int with 25..75, fun DEL( ) =NetDelay.ran( ) random-ფუნქციით. ლოგიკური პირობის მნიშვნელობა სხვადასხვა შემთხვევებში

სხვადასხვანაირად განისაზღვრება. ინტერაქტიულ სიმულატორებში ჭეშმარიტება-მცდარობას თავად მომხმარებელი განსაზღვრავს, ავტომატური სიმულაციისას – შემთხვევით რიცხვთა გენერატორი.

ბიჯი_2) 4.5 ნახაზზე სიმულატორის ღილაკებით იმართება იმიტაციური პროცესი და მიიღება შუალედური და საბოლოო შედეგები.

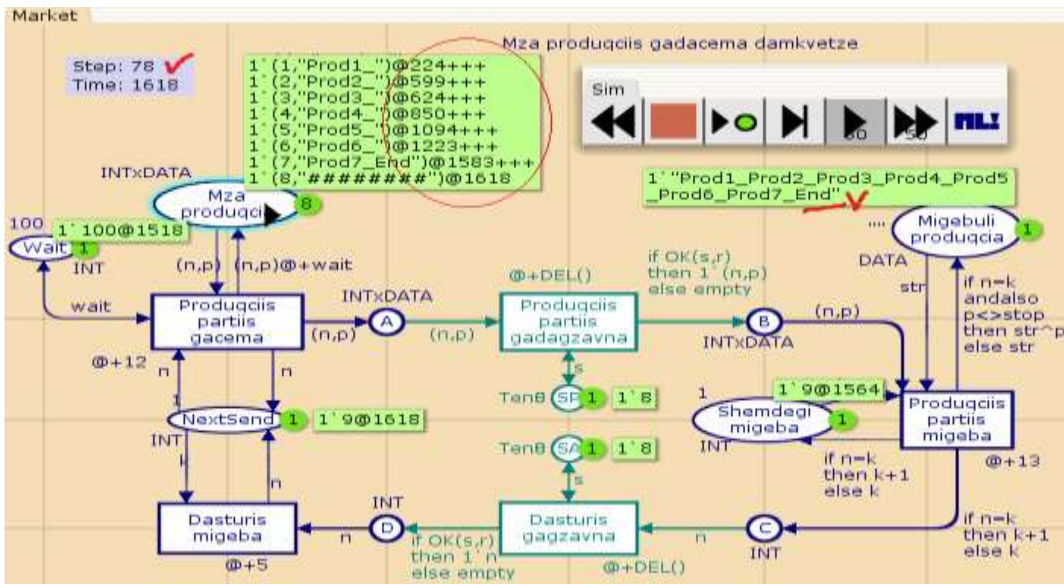
ნახ.4.5

მაგალითად, 4.6 ნახაზზე ნაჩვენებია პროცესის დინამიკის ფრაგმენტი 30-ე ბიჯზე. აქ მიმდევრობით ხდება „ფერადი მარკერების“ (სხვადასხვა სახის პროდუქციის) გაგზავნა დამკვეთებზე. წითელი წრეხაზებით აღნიშნული გვაქვს პარალელურად შესრულებადი პროცესები (გადასასვლელების გახსნა). სქემაზე ჩანს ასევე დროის მომენტები და დამკვეთთან უკვე მისული პროდუქციის დასახელებები.



ნახ.4.6. იმიტაციური პროცესი (შუალედური ბიჯი = 30)

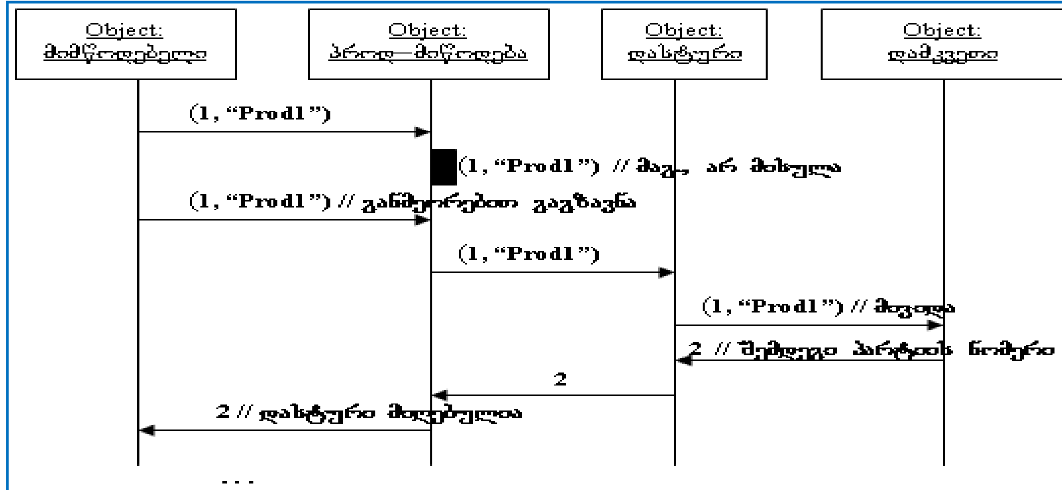
4.7 ნახაზზე კი ნაჩვენებია დასრულებული ბიზნესპროცესი, ანუ ყველა შეკვეთილი პროდუქტი გადაცემულია დამკვეთზე, რაც ვიზუალურად ფიქსირდება “End”-ით.



ნახ.4.7. პროცესი დასრულდა (ბიჯი = 78)

➢ პროცესების შესრულება უნიფიცირებული მოდელირების ენის UML

აღნიშნული პროცესების შესრულება უნიფიცირებული მოდელირების ენის UML-ტექნოლოგიაში მოგვაგონებს შეტყობინებათა (Messages) მართვას ინტერაქტიურობის დინამიკურ მოდელში, რომელსაც მიმდევრობითობის დიაგრამით (Sequence-D) ვიცნობთ. 4.8 ნახაზზე მოცემული გვაქვს ასეთი დიაგრამის ფრაგმენტი:



ნახ.4.8. იმიტაციური პროცესის ეკვივალენტური მიმდევრობითობის დიაგრა

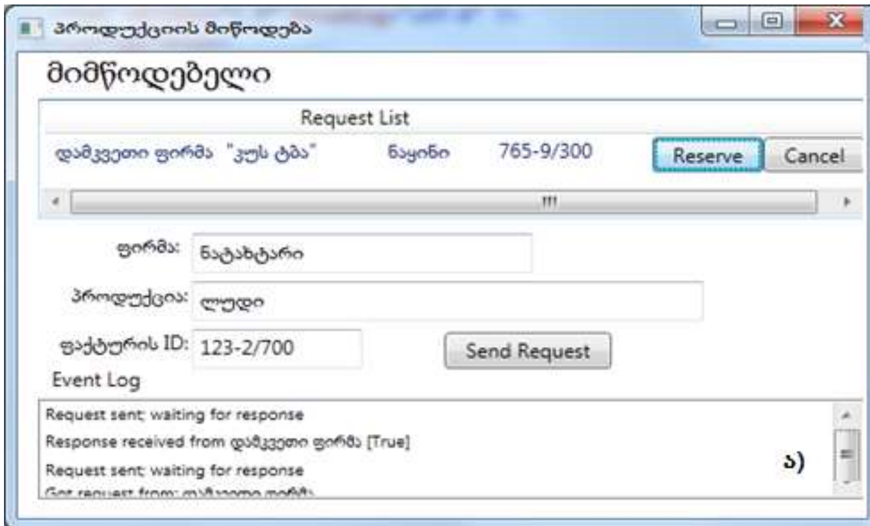
➢ მარკეტინგული პროცესების პროგრამული რეალიზაცია კლიენტ-სერვერული არქიტექტურით

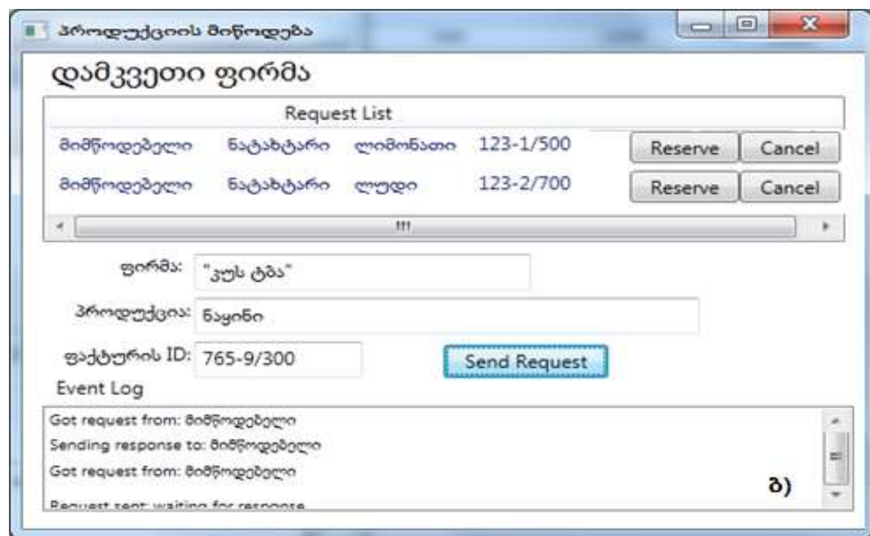
როგორ დავაპროგრამოთ მარკეტინგული მენეჯმენტის ის ბიზნესპროცესები, რომელთა მოდელი ავაგეთ პეტრის ქსელის CPN ინსტრუმენტით. კლიენტ-სერვერული ბიზნესპროცესების შესრულებისას ერთ-ერთი მნიშვნელოვანი საკითხია კომუნიკაცია *„მიმწოდებელ-დამკვეთ"* აპლიკაციებს შორის, ან კლიენტებსა და სერვერებს შორის. ასეთი კავშირების რეალიზაცია შესაძლებელია ბიზნესპროცესებსა და ჰოსტდანართებს შორის. ქვემოთ განვიხილავთ ჩვენი მაგალითის შესაბამისი პროგრამების ფრაგმენტებს.

აპლიკაციის მაგალითის სახით ვიხილავთ პროექტის აგებას პროდუქციის მიმწოდებელ ფირმასა და დამკვეთ ორგანიზაციას შორის. კერძოდ, მიმწოდებელი (Firm) აგზავნის პროდუქციის პარტიას (Product), შესაბამისი თანმხლები დოკუმენტით, ფაქტურით (Invoice) დამკვეთთან. როდესაც დამკვეთი ფირმა მიიღებს პროდუქციას, იგი საპასუხო შეტყობინებას უბრუნებს მიმწოდებელს, რითაც ეს „ტრანზაქცია" დასრულებულად ითვლება. მიმწოდებლის იმავე აპლიკაციას შეუძლია მოთხოვნის გაგზავნა სხვა დამკვეთთან და ასევე საპასუხო შეტყობინების მიღება მათგან. თუ საპასუხო შეტყობინება არ დაბრუნდა მიმწოდებელთან, ეს ნიშნავს, რომ შეკვეთა არაა შესრულებული და შესაძლებელია პროდუქციის იგივე პარტია კვლავ გაიგზავნოს დამკვეთთან (საჯარიმო სანქციების პრევენციის მიზნით).

მთავარი ქმედებები, რომლებიც კომუნიკაციისათვის გამოიყენება არის Send და Receive ქმედებები (და მათი ვარიაციები: SendReply და ReceiveReply). ეს ქმედებები გამოიყენებს Windows Communication Foundation (WCF) ტექნოლოგიას შეტყობინებათა გადასაცემად და სამეთვალყურეოდ. ჩვენ ავაგებთ მარტივ WPF აპლიკაციას (Windows Presentation Foundation), რომელიც გამოიყენებს კომუნიკაციას, მაგალითად ორ სხვადასხვა აპლიკაციის (მიმწოდებელი და დამკვეთი) ბიზნესპროცესებს შორის.

4.9-ა,ბ ნახაზებზე მოცემულია საილუსტრაციო ფრაგმენტები „მიმწოდებელ-დამკვეთ" აპლიკაცებს შორის, თუ როგორი შეიძლება იყოს მათი ინტერფეისები.





ნახ.4.9-ა,ბ. „მიმწოდებელი-დამკვეთი“ აპლიკაციის ინტერფეისები

მაგალითად, ფირმა „ნატახტარი” აგზავნის შეკვეთილი „ლიმონათის პარტიას”, ფაქტურით „123-1/500“, ამოქმედდა „Send Request” ღილაკი და დამკვეთი ფირმის ინტერფეისზე „Request List”-ში გამოჩნდა სტრიქონი ამის შესახებ. ეს ნიშნავს პროდუქციის ადგილზე მიტანას (ჩვენ მაგალითში ეს მყისიერად მოხდა, თუმცა შესაძლებელია გარკვეული დაყოვნების დროის გამოყენებაც, შემთხვევით რიცხვთა გენერატორის დახმარებით, რადგან პროცესი სტოქასტურია). გარკვეული დროის შემდეგ, მიმწოდებელი აგზავნის მეორე შეკვეთას, „ლუდის პარტიას”, ფაქტურით „123-2/700” და ა.შ.

დამკვეთი ფირმა, პროდუქციის პარტიის მიღების შემთხვევაში, იყენებს „Reserve” ღილაკს, რაც უზრუნველყოფს მიმწოდებლის ინფორმირებას პროდუქციის ამ პარტიის მიღების შესახებ. ეს შეტყობინება მიმწოდებლის ინტერფეისზე აისახება მოვლენათა რეგისტრაციის, „Event Log” ლისტბოქსში. 4.1 ლისტიგნში მოცემულია „მიმწოდებლის” ინტერფეისის პროგრამული აპლიკაციის კონფიგურაციის ფაილი (App.config). აქ ყურადსაღებია პორტის ნომრები (Address, Request Address), რომლებიც „დამკვეთი ფირმისთვის” იგივეა, ოღონდ შებრუნებული.

<-- ლისტინგი–4.1: --- App.config ---

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="Branch Name" value="მიმწოდებელი"/>
```

```
    <add key="ID" value="{43E6DADD-4751-4056-8BB7-7459B5C361AB}"/>
    <add key="Address" value="8730"/>
    <add key="Request Address" value="8000"/>
  </appSettings>
</configuration>
```

ორივე პროგრამული აპლიკაცია, გაიშვება „Run as administrator" რეჟიმში. ერთ კომპიუტერზე (ექსპერიმენტისათვის) ერთდროულად ჩანს ორი ფანჯარა (ნახ.4.9-ა,ბ). ინფორმაციის მომზადება და გადაცემა, ასევე შეტყობინების გაგზავნა შესაძლებელია ორივე მიმართულებით. 4.2 და 4.3 ლისტინგებში ნაჩვენებია.

```
// -- ლისტინგი_4.2--- CreateRequest.cs   ---
using System;
using System.Activities;
using System.Configuration;
namespace ProductDelivery  // პროდუქციის მიწოდება
{
   public sealed class CreateRequest : CodeActivity
    {
       public InArgument<string> Product { get; set; }
       public InArgument<string> Firm { get; set; }
       public InArgument<string> InvoiceID { get; set; }
       public OutArgument<ReservationRequest> Request {get; set;}
       public OutArgument<string> RequestAddress {get; set;}

  protected override void Execute(CodeActivityContext context)
   {
     // config ფაილია გახსნა და Request Address-ის მიწოდება
    Configuration config = ConfigurationManager
       .OpenExeConfiguration(ConfigurationUserLevel.None);
     AppSettingsSection app =
             AppSettingsSection)config.GetSection("appSettings");
    // ReservationRequest კლასის შექმნა და მისი შევსება არგუმენტებით
      ReservationRequest r = new ReservationRequest
        (
          Product.Get(context),
          Firm.Get(context),
          InvoiceID.Get(context),
          new Branch
           {
             BranchName = app.Settings["Branch Name"].Value,
             BranchID = new Guid(app.Settings["ID"].Value),
             Address = app.Settings["Address"].Value
           },
          context.WorkflowInstanceId
         );

       // მოთხოვნის შენახვა OutArgument-ში
```

```
        Request.Set(context, r);

      // მისამართის შენახვა OutArgument-ში
       RequestAddress.Set(context, app.Settings["Request Address"].Value);
    }
  }
}
```

// -- ლისტინგი_4.3 --- CreateResponse. cs ---

```
using System;
using System.Activities;
using System.Configuration;
namespace ProductDelivery
 {
  public sealed class CreateResponse : CodeActivity
   {
    public InArgument<ReservationRequest> Request {get; set;}
    public InArgument<bool> Reserved { get; set; }
    public OutArgument<ReservationResponse> Response {get; set;
   }

protected override void Execute(CodeActivityContext context)
 {
   // config ფაილის გახსნა ---
   Configuration config = ConfigurationManager
             .OpenExeConfiguration(ConfigurationUserLevel.None);
   AppSettingsSection app =
             (AppSettingsSection)config.GetSection("appSettings");

   // ReservationResponse  კლასის შექმნა და მისი შევსება ----
      ReservationResponse r = new ReservationResponse
      (
         Request.Get(context),
         Reserved.Get(context),
         new Branch
          {
            BranchName = app.Settings["Branch Name"].Value,
            BranchID = new Guid(app.Settings["ID"].Value),
            Address = app.Settings["Address"].Value
          }
         );
    // პასუხის  შენახვა OutArgument- ში
       Response.Set(context, r);
   }
  }
}
```

4.4 ლისტინგში მოცემულია კლიენტის სერვისის კლასის კოდის ფრაგმენტი.

```
// -- ლისტინგი_4.4 --- ClientService.cs ---
using System;
using System.ServiceModel;
namespace ProductDelivery
{
  public class ClientService : IProductDelivery
   {
     public void RequestProduct(DeliveryRequest request)
       {
          ApplicationInterface.RequestProduct(request);
       }

   public void RespondToRequest(DeliveryResponse response)
       {
          ApplicationInterface.RespondToRequest(response);
       }
    }
  }
```

დასკვნა

გამოკვლეულია პროდუქციის მწარმოებელი ორგანიზაციის (ფირმის) მარკეტინგის ფუნქციები, ამ სფეროში კომპიუტერული ტექნიკისა და ახალი ინფორმაციული ტექნოლოგიების დანერგვის აქტუალურობა. დასაბუთდა იმიტაციური მოდელირების გამოყენების ეფექტურობა. აუცილებელი გახდა კომპიუტერული ტექნოლოგიების გამოყენებით მარკეტინგის მართვის პროცესების მოდელირება და შესაბამისი პროგრამული პაკეტების მოძიება და შემუშავება;

იმიტაციური მოდელირების დახმარებით პეტრის ქსელების ინსტრუმენტით შესაძლებელია რთული, დინამიკური პროცესების ანალიზი, სტატისტიკური მონაცემების დამუშავება, მთლიანად ქსელის ან ცალკეული ქვექსელების უსაფრთხოების კვლევა;

ინფორმაციული სისტემების დაპროექტებისა და მათი პროგრამული რეალიზების თანამედროვე ტექნოლოგიების გამოყენება ობიექტორიენტირებული, პროცესორიენ-ტირებული და სერვისორიენტირებული მიდგომების საფუძველზე მნიშვნელოვნად აუმჯობესებს განაწილებული მართვის საინფორმაციო სისტემების დაპროექტებისა და აგების პროცესს;

დაპროექტებული მართვის საინფორმაციო სისტემის ეფექტური პროგრამული რეალიზაცია მომხმარებელთა ინტერფეისების ჩათვლით, უნდა განხორციელდეს პროგრამირების ჰიბრიდული ახალი ტექნოლოგიებით .NET პლატფორმაზე, კერძოდ, WPF და WCF პაკეტებით.

რეკომენდებული ლიტერატურა:

1. ჩოგოვაძე გ., ფრანგიშვილი ა., სურგულაძე გ. მართვის საინფორმაციო სისტემების დაპროგრამების ჰიბრიდული ტექნოლოგიები და მონაცემთა მენეჯმენტი. მონოგრ., ISBN 978-9941-20-790-7. სტუ, „ტექნიკ.უნივ.“, თბ., 2017. -1001 გვ., ბიბლ.ინდ. 004.42/7

2. ჩოგოვაძე გ., სურგულაძე გ., გულიტაშვილი მ., დოლიძე ს. პროგრამული აპლიკაციების ხარისხის მართვა: ტესტირება და ოპტიმიზაცია. მონოგრაფია. ISBN 978-9941-8-0629-2. სტუ. „ITკონსალტინგ ცენტრი". თბ., 2020. -365 გვ. CD-6016. https://gtu.ge/book/Surgu_SoftwareQuality.pdf

3. სურგულაძე გ., გულუა დ., კახელი ბ. პროგრამული აპლიკაციების აგება ვირტუალიზაციის პირობებში. მონოგრ. ISBN 978-9941-8-0627-8. სტუ. „ITკონსალტინგ ცენტრი“. თბილისი. 2019, -159 გვ. ბიბლ.ინდე. 004.92/8

4. სურგულაძე გ., ურუშაძე ბ. საინფორმაციო სისტემების მენეჯმენტის საერთაშორისო გამოცდილება (BSI, ITIL, COBIT). ISBN 978-9941-20-458-6. სტუ, „ტექნიკ.უნივ.“. თბ., 2014. -320 გვ. ბიბლ.ინდ. 004/23

5. Sommerville I. Software engineering (Ch.24: Software Quality Management). 10th edition. ISBN 978-0-13-394303-0, published by Pearson Education. 2016. USA. 811 p. Internet resource: http://dinus .ac.id/repository/docs/ajar/Sommerville-Software-Engineering-10ed.pdf (15.05.2020)

6. Fowler A. NoSQL For Dummies®. Published by: John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030-5774, www.wiley.com Copyright © 2015, New Jersey. -450 p. http://index-of.es/ Miscellanous/LIVRES/Wiley.NoSQL.Mar.2015.ISBN.1118905741.pdf (25.04.2020)

7. Tom White, O'Reilly Media; 4 edition (April 11, 2015), Hadoop: The Definitive Guide 4th Edition, https://www.amazon.com/gp/ product/1491901632/ (15.01.2020)

8. სურგულაძე გ. ქრისტესიაშვილი ხ., სურგულაძე გ. საწარმოო რესურსების მენეჯმენტის ბიზნეს-პროცესების მოდელირება და კვლევა. მონოგრ., ISBN 978-9941-20-557-6. სტუ, „ტექნიკ. უნივ.“. თბ., 2015. -216 გვ. ბიბლ.ინდ. 004.5/2

9. სურგულაძე გ., ბულია ი. კორპორაციულ Web-აპლიკაციათა ინტეგრაცია და დაპროექტება. მონოგრ., ISBN 978-9941-20-165-1. სტუ, „ტექნიკ.უნივ.“. თბ., 2012. -324 გვ. ბიბლ.ინდ. 681.327/18

10. სურგულაძე გ., კივილაძე გ. (2017). შესავალი NoSQL მონაცემთა ბაზებში. ISBN 978-9941-0-9642-6. სტუ. „ტექნიკ.უნივ.“. თბ., – 152 გვ.: ბიბლ.ინდ. 004.65(02) /2

11. პეტრიაშვილი ლ., სურგულაძე გ. მონაცემთა მენეჯმენტის თანამედროვე ტექნოლოგიები (Oracle, MySQL, MongoDB, Hadoop). ISBN 978-9941-27-176-2. სტუ. „ITC-ცენტრი“, თბ., 2017. 202 გვ. ბიბლ.ინდ. 004.42(02)/8

გადაეცა წარმოებას 12..11.2020. ხელმოწერილია დასაბეჭდად 20.11.2020. ოფსეტუ-
რი ქაღალდის ზომა 60X84 1/16. პირობითი ნაბეჭდი თაბახი 2,5. ტირაჟი 50 ეგზ.

სტუ-ს „IT კონსალტინგის სამეცნიერო ცენტრი“, თბილისი, მ.კოსტავას 77

ISBN 978-9941-8-2871-3