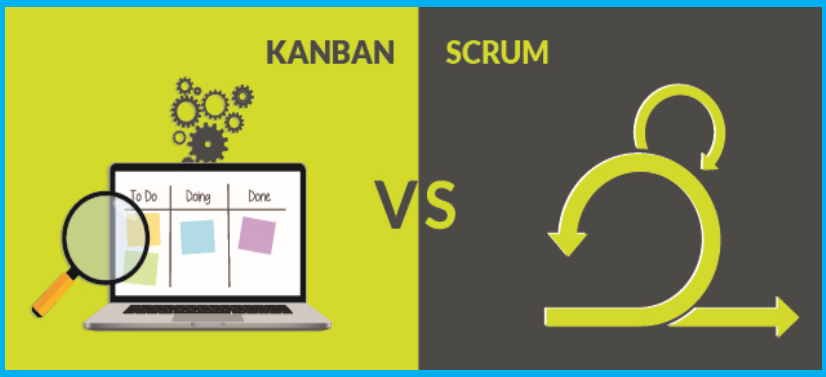
გია სურგულაძე

კომპიუტერული პროგრამირების მეთოდები და მეთოდოლოგიები

(SP, OOP, VP, Agile, UML)



„სტუ-ს IT კონსალტინგის ცენტრი”

გია სურგულაძე

კომპიუტერული პროგრამირების მეთოდები და მეთოდოლოგიები

(SP, OOP, VP, Agile, UML)

დამტკიცებულია:
სტუ-ს „IT კონსალტინგის
სამეცნიერო ცენტრის “სარე-
დაქციო კოლეგიის მიერ

თბილისი
2019

უაკ 004.5

განხილულია კომპიუტერული პროგრამირების მეთოდებისა და მეთოდოლოგიების საბაზო საკითხები და ძირითადი პარადიგმები. წარმოდგენილია პროგრამირების პროცესი, რომელიც შედგება მეთოდის, სტილის, მოდელის, ალგორითმისა და ენის ძირითადი კომპონენტების ერთობლიობისგან და მოიცავს, თავის მხრივ, სინტაქსურ, სემანტიკურ და პრაგმატულ მოსაზრებებს. განხილულია სტრუქტურული, ობიექტ-ორიენტირებული, ვიზუალური პროგრამირების მეთოდები და UML/Agile მეთოდოლოგიები, ექსტრემალური პროგრამირების, Scrum და Kanban/ Lean მაგალითებზე. წარმოდგენილია C, C++, Java, C# ენებზე რეალიზებული საილუსტრაციო კოდები პროგრამირების მეთოდების ინტერპრეტაციის მიზნით. დამხმარე სახელმძღვანელო გამიზნულია პროგრამული ინჟინერიის სპეციალობის ინფორმატიკის დარგის სტუდენტებისა და ამ საკითხებით დაინტერესებული მკითხველისათვის.

რეცენზენტები:

- ასოც. პროფ. რ. სამხარაძე (სტუ)
- ასოც. პროფ. დ. გულუა (საქ. თავდაცვის აკადემია)

რედკოლეგია:

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

ISBN 978-9941-8-1900-1

შინაარსი

შესავალი

პროგრამული ინჟინერიის მეცნიერული მიმართულება და მისი საუკეთესო პრაქტიკები მნიშვნელოვნად განვითარდა და კვლავაც განიცდის უწყვეტ სრულყოფას. 2020 წლისათვის მკვეთრად გაიზარდა უახლესი კომპიუტერული და მობილური ტექნიკისა და ინფორმაციული ტექნოლოგიების გამოყენება სოციალური, ეკონომიკური, სამრეწველო, საინჟინრო და სხვ. სფეროებში, მათი შესაბამისი ახალი თაობის პროგრამული აპლიკაციების შექმნისა და დანერგვის საფუძველზე.

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

ამერიკული და ევროპული საუნივერსიტეტო საგანმანათლებლო პროგრამები კომპიუტერული მეცნიერებების (ინფორმატიკის, კომპიუტინგის) მიმართულებით მნიშვნელოვნად შეიცვალა ბოლო წლებში ახლი კონცეფციების, მეთოდოლოგიებისა და ტექნოლოგიების მიმართულებით. განსაკუთრებით მნიშვნელოვანია აქ გამოყენებითი პროგრამული ინჟინერიის (Applied Software Engineering) სფერო, რომლის ძირითადი მიზანი მომხმარებლებზე ორიენტირებული კომპიუტერული სისტემების დიზაინი და დეველოპმენტია, უახლესი ჰიბრიდული და მობილური პროგრამირების გამოყენების საფუძველზე [1].

წიგნის პირველ თავში მოკლედაა გადმოცემული კომპიუტერული პროგრამირების კონცეფცია და ძირითადი ცნებები, მეთოდებისა და მეთოდოლოგიების, მათი სტილებისა და ენების კლასიკური, ტრადიციული მიღწევები და პერსპექტივები [2].

პროგრამირების მეთოდოლოგია და პროგრამირების მეთოდები... რა განსხვავებაა მათ შორის ?

ზოგადად, *მეთოდი* – კვლევის ინსტრუმენტი, კომპონენტია. მაგალითად, საკვლევი ობიექტის რომელიღაც მაჩვენებლის მნიშვნელობის განსაზღვრის რაოდენობრივი ან ხარისხობრივი მეთოდი (ან მეთოდები). *მეთოდოლოგია* კი არის მეთოდების და პრინციპების სისტემური, თეორიული ანალიზი კონკრეტული საკვლევი სფეროსათვის. მეთოდოლოგია არ გვაძლევს პრობლემის გადაწყვეტის შედეგს, ამიტომაც იგი არ ემთხვევა მეთოდს. მეთოდოლოგია გვაძლევს თეორიულ საფუძველს იმისათვის, რომ სწორად განვსაზღვროთ თუ რომელი მეთოდი ან მეთოდთა ჯგუფი გამოვიყენოთ კონკრეტულ შემთხვევებში [3].

პროგრამულ ინჟინერიაში *პროგრამული აპლიკაციის დამუშავების მეთოდოლოგია* არის დეველოპმენტის სამუშაოს დაყოფის პროცესი ცალკეულ ფაზებად – დიზაინის, პროდუქტებისა და პროექტების მართვის სრულყოფის მიზნით. იგი ცნობილია აგრეთვე როგორც პროგრამების დეველოპმენტის სასიცოცხლო ციკლი (Software Development Life Cycle - SDLC) [4].

პროგრამული დეველპომენტის პროცესების მეთოდოლოგიებია, მაგალითად, ჩანჩქერის (Waterfall), იტერაციულ-ინკრემენტალური (Iterative and Incremental), სპირალური

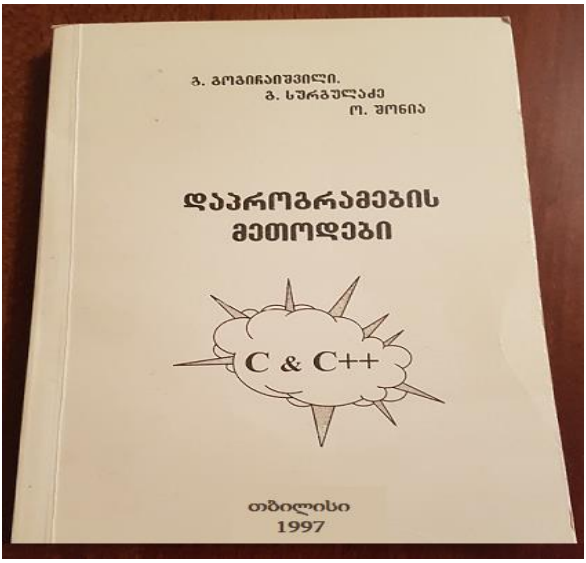
(Spiral), აპლკაციების სწრაფი დამუშავების (Rapid) და სხვ. [5]. 2000 წლიდან მნიშვნელოვანი ყურადღება მიექცა და განვითარდა უნიფიცირებული მოდელირების ენის (UML) და მოქნილი (Agile) პროგრამირების მეთოდოლოგიები. განსაკუთრებით გამახვილებულია ყურადღება ექსტრემალური პროგრამირების, Scrum და Kaban მეთოდებზე. ეს საკითხები წიგნის მეოთხე თავშია მოცემული.

მეხუთე თავი ეხება პროგრამირების სხვადასხვა სტილის ენებზე კოდების აგებისა და მათი ფუნქციონირების ანალიზს, კერძოდ, განხილულია C, C++, Java და C# ენების სტრუქტურული, ობიექტ-ორიენტირებული და ვიზუალური საილუსტრაციო პრაქტიკული მაგალითები.

წინამდებარე ნაშრომის ძირითადი მიზანია კომპიუტერული პროგრამირების მეთოდებისა და მეთოდოლოგიების საფუძვლების გაცნობა მათი პრაქტიკული გამოყენების მიზნით.

I თავი
კომპიუტერული პროგრამირების კონცეფცია და ძირითადი ცნებები

1997 წელს, საქართველოს ტექნიკური უნივერსიტეტის დაარსების 75-ე წლისთავს მიეძღვნა „მართვის ავტომატიზებული სისტემების“ კათედრაზე შექმნილი პირველი ქართული სახელმძღვანელოს „დაპროგრამების მეთოდები C & C++ ენების ბაზაზე“ (ნახ.1.1) გამოცემა [2].



ნახ.1.1. სტუ-ს პირველი წიგნი C და C++

მასში თავმოყრილი იყო კომპიუტერული სისტემების აგების სტრუქტურული და ობიექტ-ორიენტირებული მოდელირების, ალგორითმიზაციისა და პროგრამირების

მეთოდებისა და ინსტრუმენტული საშუალებების უახლესი (იმ დროისთვის) მასალა C და C++ ენების ბაზაზე, თეორიული და პრაქტიკული ინფორმატიკის სფეროს, მოდელირებისა და პროგრამირების მსოფლიო დონის ისეთი მეცნიერებისა და კლასიკოსების ნამოღვაწარი, როგორებიცაა გ. ბუჩი, ი. ჯაკობსონი, დ. მარტინი, ბ. სტრაუსტრუპი, ბ. კერნიგანი, დ. რიტჩი, მ. უეიტი, ჯ. ჰიუზი, ე. დეიკსტრა და სხვ. [1].

აგრეთვე ასახულია სტუ-ს და ნიურნბერგ-ერლანგენის (გერმანია) უნივერსიტეტების ერთობლივი პროექტების შედეგები მონაცემთა განაწილებული რელაციური ბაზების (პროფ. ჰ. ვედეკინდი – გერმანელი მეცნიერი, კათედრის გამგე) და ხელოვნური ინტელექტის სისტემების დაპროგრამების (პროფ. ჰ. შტოიანი – უნგრელი მეცნიერი, კათედრის გამგე) მიმართულებით C & C++ ენების ბაზაზე (1991-1996).

წიგნის ექსპერიმენტული ნაწილის (პროგრამების კომპლექსის) შექმნაში განსაკუთრებული წვლილი შეიტანეს „მართვის ავტომატიზებული სისტემების“ კათედრის სტუდენტებმა: ე. თურქიამ, დ. გულუამ, ლ. პეტრიაშვილმა, ა. ჩახუნაშვილმა (1994-1997 წწ). დღეს ისინი მონაცემთა ბაზების, პროგრამირებისა და კომპიუტერული ქსელების სფეროს წამყვანი სპეციალისტები და სტუ-ს პროფესორებია.

მნიშვნელოვანი წვლილი პროგრამირების C, C++, Java და C# ენების განვითარებასა და გამოყენებაში სტუ-ს პროფესორების: თ. ბახტაძის, ზ. ბოსიკაშვილის, თ. ზარქუას, დ. კაპანაძის, რ. სამხარაძის და სხვათა განსაკუთრებული დამსახურებაა, როგორც სამეცნიერო-საპროექტო საქმიანობის, ასევე საუნივერსიტეტო განათლების სფეროში.

ახალი, წინამდებარე წიგნი (2019) თავისი წინამორბედის (1997) გადამუშავებისა და გაფართოების საფუძველზე დაიწერა. კომპიუტერული რესურსების, ინფორმაციული ტექნოლოგიებისა და ინტერნეტული სისტემების არნახული სწრაფი ტემპებით განვითარებამ ძირეულად შეცვალა თანამედროვე პროგრამირების მეთოდები და მეთოდოლოგიები, რომლებიც კვლავაც განიცდის უწყვეტ განვითარებას. მათი გაცნობა და ათვისება ჩვენი სტუდენტობის ერთ-ერთი ძირითადი მიზანი და აუცილებელი საქმეა.

1.1. დაპროგრამების მეთოდი, სტილი, მოდელი, ალგორითმი და ენა

„მეთოდი“ ბერძნული სიტყვაა (methodos) და ნიშნავს გზას, კვლევის გზას, თეორიას. ესაა მიზნის მიღწევის ხერხი კონკრეტული ამოცანის გადაწყვეტისას, გარკვეული წესების სიმრავლის სისტემატური გამოყენების საფუძველზე.

„პროგრამა“ ასევე ბერძნულია (program, routine) და გამოხატავს ოპერაციების შესრულების მიმდევრობის აღწერას წინასწარ ცნობილი ალგორითმის საფუძველზე.

დაპროგრამება არის პროგრამის შექმნის პროცესი. ის მოიცავს პროგრამის დაპროექტების, კოდირებისა და ტესტირების ეტაპებს. ამგვარად, დაპროგრამების მეთოდი მართვის პროცესია, რომლის დროსაც იქმნება პროგრამული პროდუქტი (პროგრამული პაკეტი).

პროგრამის ხარისხი და მუშაობის ეფექტურობა დამოკიდებულია როგორც დაპროგრამების მეთოდზე (ობიექტური ფაქტორი), ისე დეველოპერის ცოდნაზე,

გამოცდილებასა და ტემპერამენტზე (სუბიექტური ფაქტორი). პროგრამა შეიძლება ასრულებდეს თავის მოვალეობას, ხსნიდეს ამა თუ იმ ამოცანას, მაგრამ იგი იყოს ცუდად ან კარგად შედგენილი. „ცუდად“ ან „კარგად“ ნიშნავს იმას, თუ რამდენად გასაგებადაა პროგრამა დაწერილი (შინაარსობრივად), როგორია მისი გამოსახვის სტრუქტურა ანუ ფორმა.

დაპროგრამების პროცესის განხილვისას გამოყოფენ ოთხ ძირითად ცნებას: დაპროგრამების მეთოდს, დაპროგრამების სტილს, დამუშავების ალგორითმს (მოდელს) და დაპროგრამების ენას [2,15].

დაპროგრამების მეთოდი ასახავს პროგრამის შესრულების ხერხს, ტექნოლოგიას. მაგალითად, სტრუქტურული დაპროგრამება ნიშნავს პროგრამული პაკეტების შექმნას დაღმავალი დაპროექტებისა და პროგრამების მოდულური პრინციპების გამოყენებით.

მაგალითად, არსებობს სახლის აშენების ტრადიციული მეთოდი, დაწყებული საძირკვლიდან, პირველი, მეორე და ა.შ. სართულები და ბოლოს, სახურავი. არსებობს მეორე მეთოდიც, საძირკველი, შემდეგ სახურავი, ზედა სართული, ქვედა სართული და ბოლოს პირველი სართული.

რა თქმა უნდა, შენობის აგების ამ ორ შესრულებას განსხვავებული ტექნოლოგია სჭირდება. საბოლოო პროდუქტი შენობაა, რომელსაც არ ემჩნევა რა მეთოდით აშენდა, ე.ი. მეთოდი არ ემჩნევა პროდუქტს. ასევეა პროგრამაში. მისი წაკითხვით ძნელი დასადგენი იქნება, თუ რომელი მეთოდია გამოყენებული (მაგალითად, დაღმავალი თუ აღმავალი დაპროექტების).

ის, რაც შეიძლება შევამჩნიოთ პროგრამას ან შენობას, არის ფორმა, ანუ გამოსახვის, აღწერის *სტილი*. „სტილი“ ბერძნული სიტყვაა (stylus) და ჩხირს ნიშნავს, რომლითაც ძველად სანთლის დაფებზე წერდნენ (ჩხაპნიდნენ).

მშენებლობის სტილი: მაგალითად, სვეტიცხოველი (მე-10 საუკუნე) მართლმადიდებლური ეკლესიების სტილითაა აგებული, პარიზის ღვთისმშობლის ტაძარი (მე-12 საუკუნე) – გოთური სტილია (ბაზილიკა), რითაც კათოლიკური ეკლესიები გამოირჩევა. პროტესტანტული ეკლესია (მე-16 საუკუნე) სტილით არ ჰგავს თავის წინამორბედებს, არა აქვს გუმბათები, წმინდა ხატები და ქანდაკებები.

სტილის საშუალებით ხშირად განსაზღვრავენ მხატვრული ნაწარმოების ან სპორტული კლუბის ხელწერას. მაგალითად, პიკასოს სტილი (კუბიზმი, ნეოკლასიკური მიმდევრობა, მე-20 საუკუნე), ფეხბურთში – ბრაზილიური სტილი და ა.შ. ამგვარად, სტილია ის, რისი შემჩნევაცაა შესაძლებელი, სხვა ობიექტებთან შედარების გზით შესაძლებელია მისი გამორჩევა.

პროგრამირების სტილი – ამბობენ, რომ პროგრამების დეველოპერს აქვს ცუდი სტილი, თუ ის ხშირად გამოიყენებს GOTO ოპერატორს, ან მონაცემთა აღწერისათვის იყენებს ისეთ სიმბოლოებს და სახელებს, რომელთაც არა აქვს კავშირი შინაარსთან, დანიშნულებასთან [6].

პროგრამა შეიძლება აგებული იქნეს ოპერატორების გამოყენებით, მაშინ იგი *პროცედურული* სტილისაა (მაგალითად, C, Pascal და სხვ.). თუ პროგრამაში იყენებენ მხოლოდ ფუნქციებს, მაშინ საქმე გვაქვს *ფუნქციურ* სტილთან (მაგალითად, Lisp). შედეგების მისაღებად

პროგრამა შეიძლება აგებული იქნეს ლოგიკური პროგრამების (წესების) სიმრავლით და ამ წესების დამუშავებისათვის ლოგიკური გამოყვანის ალგორითმებით. მაშინ ვლაპარაკობთ დაპროგრამების *ლოგიკურ* სტილზე (მაგალითად, Prolog). თუ დაპროგრამების დროს გამოიყენება ობიექტების ასახვის აბსტრაქტული ხერხი, რომლის საფუძველზეც ისინი შეიძლება გაერთიანებულ იქნას ერთ კლასში და აქვთ შესაძლებლობა გადასცეს თავიანთი თვისებები ახლად შექმნილ ობიექტებს, მაშინ საქმე გვაქვს ობიექტებზე ორიენტირებულ ანუ დაპროგრამების *ობიექტ-ორიენტირებულ* სტილთან (მაგალითად, C++, C#, Java, Python და სხვ.).

ერთ პროგრამაში შესაძლებელია გამოყენებული იქნეს რამდენიმე სტილი, მაშინ ასეთ ენებს ჰიბრიდულს უწოდებენ. ენა მით უფრო მძლავრია, რაც მეტი სტილია მასში რეალიზებული. აღნიშნულ საკითხებს მომდევნო პარაგრაფებში უფრო დეტალურად შევეხებით. დავსვათ კითხვა: რომელი სტილია უკეთესი, რომელი სტილი შევარჩიოთ ამოცანის გადასაწყვეტად? სწორ პასუხს თვით კონკრეტული ამოცანის ანალიზი მოგვცემს. საჭიროა დადგინდეს ამოცანის მიზანი, შინაარსი, ძირითადი ელემენტების ნაირსახეობა და ბოლოს, რაც მთავარია, – ამ ამოცანის გადამუშავების მოდელი (ალგორითმი).

ამგვარად, თუ დაპროგრამების სტილი პროგრამის ფორმას გვაძლევს, გადამუშავების მოდელი ამ ფორმას შინაარსით ავსებს. სტილის მსგავსად, გადამუშავების სხვადასხვა მოდელი არსებობს. განვიხილოთ ზოგიერთი მათგანი.

– **გადამუშავების ტრადიციული მოდელი** ფონ-ნეიმანის (ამერიკელი მათემატიკოსი, 1903-1957) მანქანის სახელითაა ცნობილი. ამ მანქანის პრინციპით მუშაობს დღემდე არსებული თითქმის ყველა კომპიუტერი. ესაა მიმდევრობითი, ბიჯური შესრულება ყოველი ოპერატორისა, რომელიც ცვლის მანქანის მდგომარეობას. იგი ფლობს მეხსიერებას და მონაცემთა დამუშავების ისეთ ოპერაციებს, როგორიცაა შედარება, გადაცემა, შეერთება, გამოყვანა (წარმოება), წაშლა და ა.შ.

– **გადამუშავების ფუნქციური მოდელი,** იგულისხმება მათემატიკური ფუნქციის ცნება, რომელიც გვაძლევს განსაზღვრის არის ცალსახა ასახვას მნიშვნელობათა არეზე. განსაზღვრისა და მნიშვნელობათა არეები არსებობს აბსტრაქტულად (სიმრავლის სახით), იმისგან დამოუკიდებლად, თუ რა მიმდევრობით აირჩევა წყვილები, რომლებიც ფუნქციურ დამოკიდებულებებს ასრულებს. მანქანას მიეწოდება ფუნქციის სახელი და არგუმენტები, რომლის საფუძველზე ის იპოვის, გადაამუშავებს და გამოსცემს მონაცემთა შესაბამის მნიშვნელობებს.

– **გადამუშავების რელაციური მოდელი,** იგულისხმება მათემატიკური დამოკიდებულებების, მიმართების ცნება. რელაციები განისაზღვრება როგორც ქვესიმრავლეები მონაცემთა სხვადასხვა დომენების დეკარტული ნამრავლიდან. ესაა ცხრილი სტრიქონებით (კორტეჟებით), რომელთა სვეტები შეესაბამება ატრიბუტთა (ველების) სახელებს, ხოლო ცხრილის სტრიქონისა და სვეტის გადაკვეთაზე მოთავსებულია მონაცემის კონკრეტული მნიშვნელობა შესაბამისი დომენიდან (ერთგვაროვან მონაცემთა სიმრავლე).

მანქანას მიეწოდება რელაციის სახელი და არგუმენტები, მაგალითად, მოსაძებნ ატრიბუტთა სახელები, რომლებიც წინასწარ განსაზღვრულ პრედიკატს, ლოგიკურ პირობას უნდა აკმაყოფილებდეს. იგი ამუშავებს ამ პრედიკატს და ეძებს რელაციაში შესაბამის მონაცემთა სტრიქონებს.

– მტკიცებათა მოდელი. იგი ეფუძნება დედუქციას და იყენებს ლოგიკურ ფორმულებს, რომლებიც ქმნის სიმრავლეს აქსიომათა სახით. ამათგან მტკიცდება, გამოიყვანება ახალი ფორმულები თეორემების სახით. მტკიცებათა თეორიის ფუძემდებელია გერმანელი მათემატიკოსი დავით გილბერტი (1862-1943), რომელმაც პირველმა შემოიტანა ეს ცნება და გამოიყენა მათემატიკური თეორიის, კერძოდ, აქსიომათა სისტემის არაწინააღმდეგობრიობის დასამტკიცებლად. მტკიცებათა მოდელის გამოსაყენებლად მანქანას უნდა შეეძლოს პირველი რიგის ლოგიკური პრედიკატების დამუშავების მექანიზმის რეალიზება.

– პრობლემათა გადამწყვეტი. იგი ამუშავებს პრობლემების სპეციფიკაციებს. პრობლემა აღიწერება განსაზღვრული ფორმით, ხოლო პრობლემათა გადამწყვეტი გამოიყვანს მისი ამოხსნის გზას. ხელოვნური ინტელექტის სისტემებში დღემდე გამოყენებული ასეთი მექანიზმები ორიენტირებულია იმ მარტივი პრობლემების აღსაწერად, რომლებიც მოცემულია საწყისი სიტუაციების მიზნობრივი პრედიკატებისა და დასაშვებ ოპერატორთა ერთობლიობის ბაზაზე.

როგორც ზემოაღწერილ, ისე სხვა სახის დამუშავების მოდელების ანალიზი გვიჩვენებს, რომ უმრავლესი მათგანი მათემატიკური მოდელებია, მათ შორის გარკვეული წილი ლოგიკურ მოდელებზე მოდის. ამ საკითხების შესწავლა და

შემდგომი განვითარება დღესაც აქტუალურ მეცნიერულ მიმართულებად ითვლება, განსაკუთრებით კი მათემატიკური მოდელების სემანტიკური პრობლემების გადაწყვეტა.

ჩვენ გავეცანით დაპროგრამების მეთოდის, სტილისა და მოდელის ცნებებს. დავსვით ამოცანა (ზოგადად), განვსაზღვრეთ მისი გადაწყვეტის გზა. შევარჩიეთ დაპროგრამების მეთოდი და სტილი, მოვახდინეთ ამოცანის მოდელისა და მისი შესრულების ალგორითმის ფორმალიზება. დაგვრჩა ამ ალგორითმის კომპიუტერში გადატანა და შედეგების მიღება. ეს საკითხები წყდება პროგრამირების ენების გამოყენებით.

პროგრამირების ენები ფორმალური სისტემებია, რომელთა საშუალებით აღიწერება ამოცანის გადაწყვეტის ალგორითმები ისე, რომ შესაძლებელი ხდება მათი შემდგომი მანქანური გადამუშავება. ამგვარად, დაპროგრამების ენა ის ინსტრუმენტული საშუალებაა, რომლითაც ალგორითმი მანქანურ კოდებში ჩაიწერება.

ყოველ ენას და, მათ შორის, მანქანურსაც, აქვს სიმბოლოების ალფაბეტი, რომლის ბაზაზე აიწყობა ენის კონსტრუქციული ელემენტები: *ოპერატორები* (დაჯავშნულ სიტყვათა ერთობლიობა), *ფუნქციები* (სტანდარტული და არასტანდარტული), *გამოსახულებები და ოპერაციები* (არითმეტიკული და ლოგიკური), *რელაციები, წესები* და ა.შ. ყოველ ენაში რეალიზებულია ამ კონსტრუქციული ელემენტების მანიპულირების გრამატიკული წესები, მონაცემთა აბსტრაქტული სტრუქტურების აღწერის საშუალებანი, ოპერაციების, ფუნქციებისა და ქვეპროგრამების შესრულების

მიმდევრობის მართვის საშუალებანი (განშტოებები, ციკლები, გადამრთველები, რეკურსიები) და ა.შ.

დაპროგრამების ყოველ ენას აქვს თავისი *სტილი*. შეიძლება ერთ ენაში პროგრამირების რამდენიმე სტილი იყოს რეალიზებული, ამით იზრდება ენის სიმძლავრე და მოქნილობა. ენა ხშირად ორიენტირებულია გარკვეული კლასის ამოცანების გადაწყვეტაზე, მათ აბსტრაქციაზე.

მაგალითად, Prolog ენა ლოგიკური დაპროგრამების ამოცანებს წყვეტს, Lisp – ფუნქციურ, C და Pascal სტრუქტურული დაპროგრამების კარგ ინსტრუმენტებად ითვლება, C++, C# და Python ჰიბრიდული (სხვადასხვა სტილით) ენებია. პროცედურული და დესკრიფციული ტიპის მონაცემთა მანიპულირების ალგებრული ენა ISBL (Information System Base Language), შეკითხვების ენა ეკრანული რედაქტორით QBE (Query By Example), SQL ან Sequel ენები, რომლებიც ალგებრულ და აღრიცხვის ენებს შორის მდგომი ენებია და ა.შ.

ბოლო ათწლეულში განსაკუთრებით განვითარება და ფართო გამოყენება ჰპოვა ობიექტორიენტირებული პროგრამირების ენებმა (C++, java, C#, Python, Visual Basic, X#), ფუნქციური (F#, Haskel, Clos) და ლოგიკური დაპროგრამების (Prolog++) კონცეფციებით.

ობიექტორიენტირებული ენები ორ ჯგუფად იყოფა:

1. დაპროგრამების ენების გაფართოებით მიღებული ჰიბრიდული ობიექტორიენტირებული ენები, მათ მიეკუთვნება Simula-67, C++, Java, C#, Prolog++ და სხვ.;

2. ახლად შექმნილი, ე.წ. სუფთა ობიექტორიენტირებული ენები. მაგალითად, Smaltalk-80, Eiffel და ა.შ.

ორივე ჯგუფს თავისი დადებითი და უარყოფითი მომენტი აქვს გამოყენების თვალსაზრისით. მაგალითად, სუფთა ობიექტორიენტირებული ენები კომპაქტური და ადვილად ასათვისებელია, მაგრამ აკლია დაპროგრამების უნივერსალური ენების ბევრი თვისება (საშუალებები), გამოყენების არე ასევე შეზღუდულია. თავის მხრივ, ჰიბრიდული ენები მოცულობით დიდია, მაგრამ მათში შერწყმულია როგორც ძირითადი ენის (მაგალითად C) დადებითი მხარეები, ისე ახალი ობიექტ-ორიენტირებული ენების თვისებები. ეს კი განაპირობებს იმას, რომ C++/C# ენებს შეუძლია ფართო კლასის ამოცანების გადაჭრა. იგი ყოველგვარი გადაპროგრამების გარეშე თითქმის მთლიანად ამუშავებს იმ არსებულ სისტემებსაც, რომლებიც C-ზეა დაწერილი მათ შექმნამდე.

1.2. პროგრამირების პარადიგმები

დაპროგრამების ენების ევოლუციური განვითარების საფუძველს დაპროგრამების პარადიგმებით ხსნიან [16]. ჩვენ წინა პარაგრაფში განვიხილეთ დაპროგრამების მეთოდები, სტილი, მოდელი და ენა. ეს საკითხები იკვეთება შინაარსობრივად პარადიგმის ცნებასთან.

არსებობს „დაპროგრამების პარადიგმების“ ცნების განსაზღვრების სხვადასხვა ვერსია. მაგალითად:

„პარადიგმა არის დაპროგრამების სტილი პროგრა-მისტ-დეველოპერის განზრახვის აღსაწერად“ (დ. ბობოროვი) [17];

„პარადიგმა არის მოდელი ან მიდგომა პრობლემის გადასაწყვეტად“ (ბ. შრაივერი) [18];

„პარადიგმები დაპროგრამების ენების კლასიფიკაციის წესებია გარკვეული პირობების შესაბამისად, რომელთა შემოწმება შესაძლებელია“ (პ. ვეგნერი) [19];

„ესაა კონცეპტუალიზაციის ხერხი იმისა, თუ რას ნიშნავს „გაანგარიშების წარმოება“ და როგორ უნდა იქნას კომპიუტერზე გადასაწყვეტი ამოცანა სტრუქტურირებული და ორგანიზებული“ (ტ. ბადი) [20] და ა.შ.

ამგვარად, თუ განვაზოგადებთ პარადიგმის სხვადასხვა განსზაღვრებას, შეიძლება ვთქვათ, რომ *პროგრამირების პარადიგმა, როგორც დასმული პრობლემის და მისი გადაწყვეტის საწყისი კონცეპტუალური სქემა, არის გრამატიკული ინსტრუმენტი ფაქტების, მოვლენებისა და პროცესების აღსაწერად.*

დაპროგრამების ძირითადი პარადიგმები შემდეგია:

- ***იმპერატიული დაპროგრამება*** (imperative programming): განსაზღვრავს გამოთვლებს მიმდევრობითი ბრძანებების (ინსტრუქციების) სახით, რომლითაც იცვლება პროგრამის მდგომარეობა. ბრანებით მიღებული მონაცემები ინახება მეხსიერებაში და ხელმისაწვდომია. ასეთი ენებია: Assembler, Algol, Fortran, C, C++ და სხვ.;
- ***პროცედურული დაპროგრამება*** (procedural programming): განსაზღვრავს ბიჯებს, რომლებიც პროგრამამ უნდა შეასრულოს რათა მიაღწიოს სასურველ მდგომარეობას. მიმდევრობით შესასრულებელი ოპერატორები შესაძლებელია მოთავსდეს ქვეპროგრამებში. ეს პარადიგმა ტრადიციული კომპიუტერის არქიტექტურას შეესაბამება, რომელიც

ფონ ნეიმანმა შემოგვთავაზა. ასეთი ენებია: Basic, Fortran, PL/1, Pascal, C და სხვ.;

- ***სტრუქტურული დაპროგრამება*** (structured programming): პროგრამა წარმოდგენილია იერარქიული სტრუქტურის ბლოკების (მოდულების) სახით. იგი მუშავდება დაღმავალი ტექნოლოგიით. არ გამოიყენება GOTO ოპერატორი [6,21]. მმართველი სტრუქტურებია: მიმდევრობითობა, განშტოება და ციკლი. სტრუქტურული დაპროგრამების მიზანია პროგრამისტების შრომის ნაყოფიერების ამაღლება დიდი და რთული პროგრამული სისტემების შექმნის, გამართვის და მოდიფიკაციის დროს. ასეთი ენებია: C, Pascal, C++ (არ ვიყენებთ GOTO ოპერატორს) და სხვ. ობიექტ-ორიენტირებული ენები [22,23];
- ***დეკლარაციული დაპროგრამება*** (declarative programming): განსაზღვრავს გამოთვლების ლოგიკას მისი მართვის ნაკადების განსაზღვრის გარეშე. ანუ, მოიცემა დასმული ამოცანის გადაწყვეტის სპეციფიკაცია (მოთხოვნებისა და პარამეტრების ერთობლიობა) თუ რა უნდა იყოს შედეგი. აქ არ ჩანს თუ როგორ უნდა იქნეს ეს შედეგი მოღებული (იმპერატიულ დაპროგრამებაში ეს ჩანს). ასეთი ენის მაგალითებია SQL და HTML და სხვ.
- ***ფუნქციონალური დაპროგრამება*** (functional programming): განიხილავს გამოთვლებს როგორც მათემატიკური ფუნქციების შესრულების შედეგებს საწყისი მონაცემების ან სხვა ფუნქციების საფუძველზე. იგი არ ითვალისწინებს პროგრამის მდგომარეობის შენახვას (როგორც ეს იმპერატიულში იყო). ასეთი ენებია: Lisp, APL, Haskel, C++, F#.NET და სხვ. [22, 24].

- *ობიექტორიენტირებული დაპროგრამება* (object-oriented programming): მეთოდოლოგიაა, რომელიც პროგრამას განიხილავს როგორც ობიექტების ერთობლიობას, სადაც თითოეული მათგანი გარკვეული კლასის ეგზემპლარია. კლასი ინკაფსულირებულია, აქვს სახელი, თვისებები (კლასის წევრები), მეთოდები (კლასის ფუნქციები), რომელთა ინიცირება ხდება გარედან შემოსული შეტყობინების საფუძველზე [2,13]. კლასები ქმნის მემკვიდრეობით იერარქიას და აქვს პოლიმორფიზმის თვისება. ობიექტორიენტირებული ენა არაა ალგორითმული ენა ! ასეთი ენებია: C++, Java, C#, Python, Smalltalk, Ruby, Eiffel, PHP და სხვ. ჩვენ წიგნში ოო-დაპროგრამების ენებს შედარებით დეტალურად განვიხილავთ;
- *მოვლენებზე ორიენტირებული დაპროგრამება* (event-driven programming): ახორციელებს პროგრამის შესრულებას მოვლენების საფუძველზე [7]. ეს შეიძლება იყოს მომხმარებლის მოქმედება (კლავიატურა, მაუსი), სხვა პროგრამებიდან ან ოპერაციული სისტემიდან მიღებული შეტყობინება და ა.შ. პროგრამის ამოცანაა, გააანალიზოს მოვლენა და აირჩიოს შესაბამისი დამმუშავებელი (event handler). ასეთი ენებია: Javascript, ActionScript, Visual Basic, Elm და სხვ. საყურადღებოა აგრეთვე Node.js პლატფორმა, რომელიც Javascript გარემოში ქმნის მარტივ, მასშტაბირებად ქსელურ აპლიკაციებს [8,9]. იგი იყენებს მოვლენებით მართვად I/O არაბლოკირებულ მოდელს, რომელიც ხდის მას მსუბუქს და ეფექტურს. იდეალურია განაწილებული სისტემის რეალურ დროში მომუშავე აპლიკაციების ასაგებად, რომლებიც მონაცემთა ინტენსიური გამოყენებისთვისა.

- *ლოგიკური დაპროგრამება* (Logic programming): ეყრდნობა თეორემების ავტომატურ მტკიცებას და დისკრეტული მათემატიკის იმ განყოფილებას, რომელიც შეისწავლის ინფორმაციის ლოგიკური გამოყვანის პრინციპებს მოცემული ფაქტებისა და გამოყვანის წესების საფუძველზე [10]. მისი მაგალითებია პირველი ენა Planner და მისგან წარმოებული Popler, Conniver და Qlisp, ასევე Prolog ენა და მისი შვილები Mercury, Visual Prolog და Oz [11,12];
- *ავტომატებზე ბაზირებული დაპროგრამება* (Automata-based programming): არის დაპროგრამების პარადიგმა, რომლის გამოყენებისას პროგრამა ან მისი ფრაგმენტი მოიაზრება როგორც რაიმე ფორმალური ავტომატის მოდელი. დასმული ამოცანისგან დამოკიდებულებით, ავტომატიზებული დაპროგრამებისას შეიძლება გამოყენებულ იქნას სასრული ავტომატები, პეტრის ქსელები ან სხვა გრაფული მოდელები. შეიძლება აქ ვახსენოთ UML ენის Statechart Diagram, რომელიც სწორედ მდგომარეობათა დიაგრამის სახელითაა ცნობილი და ა.შ. [13,14].

პროგრამირების პარადიგმთა ეს ჩამონათვალი შეიძლება გაგრძელდეს და უფრო დაკონკრეტდეს, თითოეულის გამოყენება შესაბამის კლასთა ამოცანების გადაწყვეტას ემსახურება.

უნდა აღინიშნოს, რომ დაპროგრამების ზოგიერთი თანამედროვე ენა *მულტიპარადიგმული* თვისების მატარებელია, ანუ მასში გამოყენებულია ერთდროულად დაპროგრამების რამდენიმე სტილი. მაგალითად, C++, C#,

Python – ობიექტორიენტირებული, სტრუქტურული, ფუნქციური, იმპერატიული და სხვა პარადიგმთა მატარებელია.

შეიძლება სტრუქტურული დაპროგრამების C ენაზეც დაიწეროს ობოექტორიენტირებული პროგრამები, მაგრამ ეს საგრძნობლად გაართულებს პროგრამულ კოდს.

1.3. კომპილატორები და ინტერპრეტატორები

არსებობს დაპროგრამების ენების ფართო სპექტრი. ისინი განსხვავდება ერთმანეთისაგან სინტაქსური, სემანტიკური და პრაგმატული ასპექტებით. პირველში იგულისხმება ენის სინტაქსი და კონსტრუქციები (წესები), მეორეში ინფორმაციის შინაარსის ასახვის დონე, მესამეში კი – მისი გამოყენების დანიშნულების სფერო.

კომპიუტერული ენები პირობითად შეიძლება მოწესრიგებულ იქნეს „ქვემოდან-ზემოთ“: მანქანური (ასემბლერული) ენები, დაპროგრამების უნივერსალური ენები, მონაცემთა ბაზების მართვის სისტემების ენები, მაღალი დონის (სალაპარაკო, ბუნებრივ ენასთან ახლოს მდგომი) ენები.

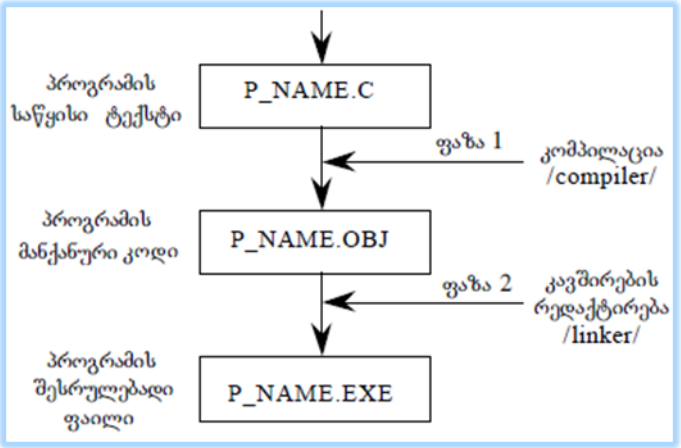
გრამატიკული და სტრუქტურული კონსტრუქციებით ენები თავსდება სისტემურ პროგრამულ პაკეტებში, რომელთაც სამუშაო ინსტრუმენტული საშუალება ეწოდება. მათ საფუძველზე იქმნება სამუშაო გარემო ანუ მომხმარებლის ინტერფეისი. სისტემური პროგრამული პაკეტი კომპილატორი ან ინტერპრეტატორია. დაბალი დონისა და უნივერსალური ენები მეტწილად პირველს

მიეკუთვნება, მონაცემთა ბაზებისა და მაღალი დონის ენები კი – მეორეს. არსებობს გამონაკლისებიც.

განვიხილოთ კომპილატორსა და ინტერპრეტატორს შორის ძირითადი განსხვავება. ზოგადად, პროგრამის მომზადება და მისი მანქანაზე შესრულება სამ ძირითად ეტაპს გაივლის (ნახ.1.2):

- საწყისი ტექსტის,
- ობიექტური (მანქანური) კოდის და
- შესრულებადი ფაილის.

პროგრამის საწყის ტექსტს, რომელიც ამ ენის ოპერატორებისა და დასაშვები კონსტრუქციებისაგან შედგება, წერს ადამიანი და მიაწვდის კომპიუტერში ამ ენის კომპილატორს ან ინტერპრეტატორს. საწყისი ტექსტი უმეტესწილად ინგლისური ენის ტერმინებით და ამოცანაში გამოყენებული ფორმალური კონსტრუქციებითაა აგებული, რაც კომპიუტერისთვის გაუგებარია. ამიტომაც მას სჭირდება „თარჯიმანი“. ესაა სწორედ კომპილაციის ფაზა: პროგრამის საწყისი ტექსტის სინტაქსური შემოწმება (შეცდომების ანალიზით) და შესაბამისი მანქანური კოდის ფორმირება (*.obj - გაფართოებით). კოდებში გამოხატული ტექსტი მანქანისათვის მისაღებია და იგი აანალიზებს, თუ რა უნდა პროგრამას მისგან, ანუ რა სახის რესურსებს თხოულობს იგი მანქანური მეხსიერებისა და პროგრამული სტანდარტული ბიბლიოთეკიდან. მე-2 ფაზაზე, კავშირების რედაქტორი ჩვენი პროგრამის მანქანურ კოდს მიუერთებს მისთვის საჭირო სხვა ქვეპროგრამებს და ფუნქციებს.



ნახ.1.2. პროგრამის კომპილაციის ეტაპები
(„ C ”-ენის მაგალითზე)

ამის შემდეგ იქმნება შესრულებადი ფაილი (*.exe-გაფართოებით). თუ კომპილატორთან გვაქვს საქმე, მაშინ ეს exe-ფაილი თავსდება მოცემულ დირექტორიაში და მისი გამოყენება შეიძლება მრავალჯერადად (წინა ეტაპების გაუვლელად). პროგრამის შესრულების პროცესი სწრაფია.

ინტერპრეტატორის შემთხვევაში ორივე ფაზა სრულდება, მაგრამ არ იქმნება exe-ფაილი. პროგრამა ერთხელ შესრულდება. თუ ხელმეორედ გვინდა ავამუშაოთ იგი, მაშინ ეს პროგრამა ისევ უნდა გავხსნათ (მისი საწყისი ტექსტი) და ჩავატაროთ კომპილაცია და ლინკირება. ეს კი, რა თქმა უნდა, ანელებს პროგრამის შესრულებისა და შედეგების მიღების პროცესს.

კომპილირებული exe-ფაილი მობილურია, მისი შესრულება შეიძლება სხვა თავსებად მანქანებზეც

კომპილატორის გარეშეც. ინტერპრეტატორზე შესრულებული პროგრამა კი მოითხოვს თავისი სისტემური ინტერპრეტატორის არსებობასაც, რაც არაა ეფექტური.

დაპროგრამების უნივერსალური ენა „C" („C++") მიეკუთვნება სწორედ კომპილატორთა ოჯახს. მისი გამოყენების ეფექტურობის დამადასტურებელია ის ფაქტი, რომ თვით ქსელური ოპერაციული სისტემა UNIX, ოპერაციული სისტემა Windows, Microsoft-ის პაკეტები და სხვა დაწერილია ამ ენაზე. დაპროგრამების ენები Java და Prrolog კი მიეკუთვნება ინტერპრეტატორებს. მათი „დედა˜-ენებია - შესაბამისად C++ და Pascal.

II თავი
სტრუქტურული პროგრამირების მეთოდი

სტრუქტურული მიდგომა პროგრამირების ერთ-ერთი მეთოდოლოგიური საფუძველია, რომელიც 70-იანი წლების დასაწყისიდან იღებს სათავეს და მისი მეცნიერული კვლევა და განვითარება უკავშირდება ჰოლანდიელი მეცნიერის ედსგერ დეიკსტრას (Edsger Dijkstra) სახელს [6].

სტრუქტურული მეთოდის მიზანია „დაპროგრამების პროცესის“ მართვა კომპაქტური, ადამიანისათვის გასაგები და საიმედოდ მომუშავე პროგრამული მოდელების ასაგებად. დაპროგრამების პროცესის სწორად ორგანიზება საგრძნობლად ზრდის პროგრამისტ-დეველოპერის შრომის ნაყოფიერებას და ამცირებს პროგრამული პროდუქტის შექმნის და დანერგვის დროს.

სტრუქტურული პროგრამირების იდეოლოგია ერთ-ერთი მთავარი კომპონენტია დაპროგრამების ობიექტ-ორიენტირებული ენებისათვისაც [2].

დაპროგრამების პროცესი რამდენიმე ეტაპისაგან შედგება:

1. *ამოცანის დასმა.* ეს ძალზე საპასუხისმგებლო ეტაპია, ვინაიდან აქ უნდა ჩამოყალიბდეს ამოცანა, რომელშიც ზუსტად იქნება ასახული ძირითადი მიზანი, საწყისი პირობები, მონაცემები და საბოლოო შედეგები. ამასთანავე, უნდა განისაზღვროს ამოცანის გადაწყვეტის მოდელი, მისი შესრულების ალგორითმი და ვადები;

2. *დაპროექტება.* ამ ეტაპზე ხორციელდება დასმული ამოცანის გადაწყვეტის სირთულის შეფასება. მის საფუძველზე დაპროექტდება მომავალი პროგრამის ცალკეული მოდულები (აქ ამოცანის დეკომპოზიცია ანუ მისი დაშლა, გამარტივება იგულისხმება) და მათი ურთიერთკავშირების სქემები. ამ ეტაპის ბოლო პუნქტია მოდულების გამჭოლი კონტროლი. მისი არსი მდგომარეობს შეცდომების აღმოჩენაში დაპროექტების ეტაპის ადრეულ სტადიაზე (ვიდრე არ დაწყებულა დაპროგრამება);

3. *კოდირება.* ესაა დაპროგრამება რომელიმე ენაზე, ანუ მოდულებისა და მთლიანი სისტემის აღწერა მანქანისათვის გასაგებ ენაზე, კოდებით;

4. *ტესტირება.* ამ ეტაპზე მოწმდება პროგრამის უნარი საკონტროლო მონაცემებზე. საკონტროლოა მონაცემები, რომელთა გადამუშავების შედეგები წინასწარ ცნობილია. პროგრამამ, რომელიც კოდირებულ იქნა მანქანაზე, უნდა მოგვცეს იგივე შედეგები. თუ შედეგები არადამაკმაყოფილებელია, მაშინ მიზეზები უნდა ვეძებოთ დაპროგრამების პროცესის წინა ეტაპებზე.

სტრუქტურული პროგრამირების მეთოდის არსი სწორედ ისაა, რომ საგრძნობლად შეამციროს ზემოაღწერილი ეტაპების ციკლური გამოყენების რაოდენობა და პროგრა-მული პროდუქტი ჩაბარდეს წინასწარ შეთანხმებულ, განსაზღვრულ ვადებში.

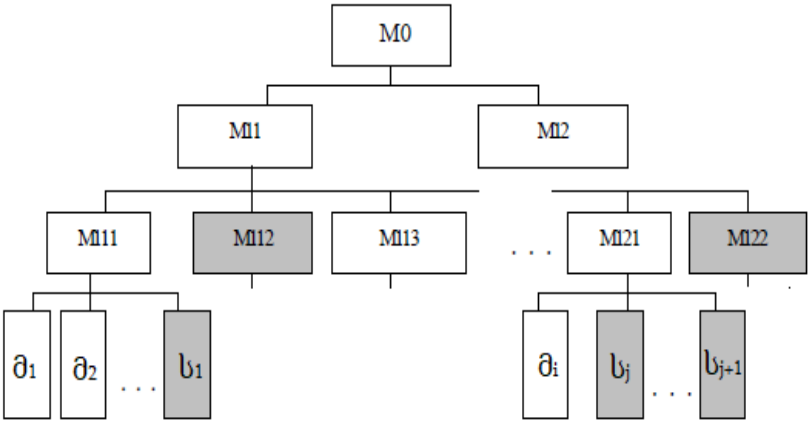
2.1. სტრუქტურული პროგრამირების ძირითადი ელემენტები

სტრუქტურული დაპროგრამება სამი ძირითადი ნაწილისაგან შედგება:

- დაღმავალი ტექნოლოგია;
- სტრუქტურული დაპროგრამება და
- გამჭოლი სტრუქტურული კონტროლი.

ლიტერატურაში ხშირად დაღმავალ სტრუქტურულ პროგრამირებასაც მოიხსენიებენ. ამ საკითხს შემდეგ პარაგრაფში დავუბრუნდებით, ახლა კი გამოვკვეთოთ სტრუქტურული პროგრამირების ძირითადი ელემენტები.

პროგრამული პროდუქტი (პაკეტი) იერარქიულად სტრუქტურირებული მოდულების ერთობლიობა (ნახ.2.1).



ნახ. 2.1. პროგრამული პაკეტის იერარქიულად სტრუქტურირების ზოგადი სქემა, მi - ქვედა დონის მოდული, სj - სახშობი

იერარქიულად ორგანიზების პრინციპი რთული ამოცანის (პროგრამის) დეკომპოზიციის საშუალებაა შედარებით მცირე ზომის მოდულებად. მაგალითად, M0 მთავარი პროგრამაა, M11 და M12 ქვეპროგრამების იერარქიით მე-2 დონეზე და ა.შ. $მ_1$, $მ_2$, ... პროგრამული მოდულებია, რომლებიც იერარქიის ყველაზე ქვედა დონეზე იქმნება.

ს – სახშობი „ფორმალური“ პროგრამული მოდულია, რომელსაც კონკრეტული შინაარსი ჯერ კიდევ მიღებული არა აქვს.

მi და **სj** მოდულებით ხდება ზედა დონის პროგრამული (აგრეგირებული) მოდულების აგება.

მოდულებად დაყოფის საკითხი დამპროექტებლის ფუნქციაა და ეფექტურობაც მის პრაქტიკულ გამოცდილებაზეა დამოკიდებული.

პროგრამირების დროს მნიშვნელოვან ამოცანად ითვლება კოდის შესრულების პროცესის ეფექტური განხორციელება, რომლის დროსაც დაცულია დაპროგრამების კარგი სტილი. კლასიკურ ლიტერატურაში ამ დროს უპირობო გადასასვლელის, კერძოდ, ოპერატორ GOTO-ს „არ-გამოყენებას“ მოიხსენიებენ [6]. არასასურველად ითვლება ამ ოპერატორის გამოყენება, თუმცა, შეიძლება არასტრუქტურირებული პროგრამაც დაიწეროს GOTO-ს გარეშე ისე, როგორც სტრუქტურირებული GOTO-ებით.

ამგვარად, უპირობო გადასასვლელის – GOTO-ს გამოყენება ითვლება მართვის სქემების უსისტემო კონსტრუირების საშუალებად, რომლის დროსაც პროგრამა კარგავს აღქმადობას და მართვადობას.

მის ნაცვლად გამოყენებულ უნდა იქნას ისეთი მოქნილი და მძლავრი საშუალებები, როგორიცაა პირობითი გადასვლის (განშტოების) ბლოკი (if else …), ციკლების ორგანიზების ბლოკები (while, case, do case ...) და ა.შ.

სტრუქტურული პროგრამირების ერთ-ერთი ძირითადი ელემენტია პროგრამის ცვლადებისა და კონსტანტების შერჩევა.

პროგრამის აღქმადობა მეტია, როცა მასში შინაარსობრივი სახელები გამოიყენება. ყოველ პროგრამას, ქვეპროგრამას, ფუნქციას, ცვლადებს, კონსტანტებს, მასივებს და სხვა აგრეგატულ მონაცემებს სასურველია ჰქონდეს *სემანტიკური სახელები,* რომლებშიც ასახული იქნება მათი დანიშნულება, ფუნქცია.

დაპროგრამების კარგ სტილად შეიძლება ჩავთვალოთ პროგრამაში *შეწეული აგზაცების, ცარიელი სტრიქონებისა* და *კომენტარების* გამოყენება. ყოველივე ეს ხელს უწყობს პროგრამის კითხვის პროცესში პროგრამისტ-დეველოპერის მხედველობითი და გონებრივი რესურსების ეფექტურ გამოყენებას.

2.2. დაპროგრამების დაღმავალი ტექნოლოგია

დავუბრუნდეთ 2.1 ნახაზზე წარმოდგენილი პროგრამული პაკეტის ზოგად სქემას. როგორც აღვნიშნეთ, იგი მოდულების იერარქიულად მოწესრიგებული ერთობლიობაა. დაპროგრამების ტექნოლოგია არის მეთოდი და კომპიუტერზე რეალიზაციის ინსტრუმენტული საშუალება (დაპროგრამების ენა).

დაღმავალი ტექნოლოგია ნიშნავს, რომ პროგრამული მოდულები უნდა დამუშავდეს ზემოდან ქვემოთ. ამ დროს ქვედა დონეებზე გამოიყენება სახშობები. სახშობიც პროგრამული მოდულია, ოღონდ იგი დროებით ცარიელია, შეიძლება მასში მხოლოდ შესვლა და გამოსვლა. დროთა განმავლობაში სახშობები შეიცვლება რეალური პროგრამული მოდულებით.

არსებობს დაპროგრამების ტრადიციული ანუ *აღმავალი ტექნოლოგია*, რომლის დროსაც სისტემის დაპროექტება ხდება ზემოდან ქვემოთ, ხოლო პროგრამული მოდულების დაპროგრამება *ქვემოდან ზემოთ*, მაგალითად, 2.1 ნახაზისთვის დამუშავდება ყველაზე ქვედა დონის პროგრამათა მოდულები, შემდეგ ზემოთა და ა.შ.

პრობლემა მდგომარეობს შემდეგში. დავუშვათ, რომ ქვედა დონის ყველა პროგრამული მოდული ფუნქციონირებს. ზედა დონის მოდულის დაპროგრამების დროს გამოიყენება ქვედა დონის მოდულები. ხშირად აღმოჩნდება, რომ ეს მოდულები ვერ მუშაობს ერთმანეთთან. ე.ი. ცალ-ცალკე ყველა მუშაობს, ერთად კი არა. ეს იწვევს პროგრამის დაპროექტებისა და კოდირების ეტაპების გამეორებას, რაც, რა თქმა უნდა, შრომატევადი და არასასურველი პროცესია. რაც უფრო რთული და დიდია სისტემა, და რაც მეტი პროგრამისტი მონაწილეობს მისი მოდულების კოდირებაში, მით მეტია არაფუნქციონირებადი პროგრამული პაკეტის მიღების შანსი.

დაპროგრამების დაღმავალი ტექნოლოგია კი ყოველივე ამას ითვალისწინებს, ვინაიდან იგი პროგრამულ მოდულებს აპროექტებს და აკოდირებს ზემოდან ქვემოთ.

თავიდან ხდება სისტემის ფუნქციური სპეციფიკატორებისა და მისი შემადგენელი პროგრამული მოდულების თავსებადობის პირობების განსაზღვრა. იერარქიის ზედა დონიდან ქვედაზე ჩასვლა ისე არ ხდება, რომ არ მუშაობდეს ზედა დონის მოდული. სწორედ მისი მუშაობის უზრუნველსაყოფად (თუ მას სჭირდება კავშირი ქვედა დონის მოდულებთან) ქმნიან დროებით მოდულებს ანუ „სახშობებს“. ასეთ მეთოდს „ჩანჩქერს“ ადარებენ, სადაც ქვემოთ ჩამოყოლა გაცილებით ადვილია, ვიდრე ზემოთ ასვლა.

მოდულებისა და სახშობების დაპროგრამების რიგითობა განისაზღვრება დაგეგმვის ეტაპზე. აქ განიხილავენ იერარქიულ, ოპერაციულ და კომბინირებულ მიდგომას.

იერარქიული მიდგომით დაგეგმვა ითვალისწინებს მოდულების დაპროგრამებას და ტესტირებას იერარქიული განლაგების მიხედვით. ჯერ იქმნება ერთი დონის ყველა მოდული, შემდეგ ხდება გადასვლა ქვედა დონეზე. შესაძლებელია აგრეთვე იერარქიის ჯერ ერთი შტოს მოდულების შექმნა მთლიანად, შემდეგ გადასვლა მეორეზე და ა.შ. იერარქიული დაგეგმვის გამოყენებისას საყურადღებოა მონაცემთა თავსებადობის უზრუნველყოფა, რომლის გარეშეც გარკვეული პრობლემები იქმნება მოდულებს შორის მონაცემთა გაცვლის დროს.

ოპერაციული მიდგომა გულისხმობს მოდულების დამუშავებას მათი შესრულების მიმდევრობის მიხედვით მზა პროგრამის ამუშავების დროს. ეს კი დამოკიდებულია საწყისი მონაცემების შემადგენლობაზე და განისაზღვრება

მგეგმავის მიერ. ოპერაციული მიდგომის დადებითი მხარეა მონაცემთა თავსებადობის პრობლემების იოლი გადაწყვეტა (ვიდრე იერარქიულის შემთხვევაში), ვინაიდან მოდულები შესრულების მიმდევრობით იქმნება და მონაცემთა გადაცემაც ამ მიმდევრობას ემორჩილება. ნაკლად შეიძლება ჩაითვალოს მოდულების შესრულების მკაცრი რიგითობის არსებობა, როცა რომელიმე მოდული ვერ დამუშავდება მანამ, სანამ მის წინ მდგომები არ შეიქმნება. ამან შეიძლება გაზარდოს მთლიანი სისტემის შექმნის ვადები.

კომბინირებული მიდგომა აერთიანებს იერარქიული და ოპერაციული მიდგომების შესაძლებლობებს და უკეთეს შედეგსაც იძლევა. ამოცანა არატრივიალური და მრავალვარიანტულია, ამიტომაც მისი ზუსტი გადაწყვეტა დამოკიდებულია მგეგმავის ცოდნასა და პრაქტიკულ გამოცდილებაზე.

პროგრამული პაკეტის რეალიზაციის ეტაპი რამდენიმე ბიჯისაგან შედგება. პირველ რიგში, მზადდება მონაცემები ტესტირებისათვის, რომელთა გამოყენებითაც უნდა შეფასდეს დაპროგრამებული მოდულებისა და მთლიანი სისტემის შრომის უნარი. შემდგომ ბიჯზე იქმნება პროგრამის ძირითადი ბირთვი (მთავარი მოდული), ბოლოს კი – მოდულები და სახშობები, რომლებიც უზრუნველყოფს ტესტური მონაცემების შემოწმებას. დაღმავალი სტრუქტურული დაპროგრამების მეთოდის გამოყენებისას შესაძლებელია პროგრამული პაკეტის დეტალური დაპროექტების, კოდირების, ტესტირებისა და დოკუმენტირების პროცესების პარალელური შესრულება, რაც საბოლოო ჯამში უზრუნველყოფს სისტემის დაპროგრამების ვადების დაცვას.

2.3. მოდულურობის პრინციპი

რთული სისტემების დასაპროექტებლად გამოიყენება *სტრუქტურული დაღმავალი დაპროგრამების* ტექნოლოგია. სისტემის სირთულეს ამცირებენ მისი დეკომპოზიციის გზით. ამ დროს მიიღება რამდენიმე ქვესისტემა, რომელთა შემდგომი ანალიზი და შესასრულებელი ფუნქციების მიხედვით დაყოფა კიდევ უფრო ამარტივებს და ცხადყოფს პროგრამული პაკეტის აგების პროცესს.

ერთი ან რამდენიმე ფუნქცია უნდა იქნეს რეალიზებული პროგრამული მოდულის სახით (ზოგჯერ მას ქვეპროგრამას უწოდებენ). მოდულს აქვს სახელი, ერთი შესასვლელი და ერთი გამოსასვლელი. მას მიეწოდება საწყისი მონაცემები, რომელთაც გადაამუშავებს ფუნქციური ალგორითმის მიხედვით და იძლევა შედეგებს.

მოდული პროგრამული პაკეტის ძირითადი კომპონენტია, რომლის დამუშავება და ტესტირება შესაძლებელია დამოუკიდებლად. თუ პროგრამული პაკეტი დიდია (შედგება მრავალი მოდულისაგან), მაშინ იზრდება მოდულთაშორისი ურთიერთკავშირების ორგანიზაციის სირთულე, რაც გათვალისწინებულ უნდა იქნეს სისტემის დაპროექტების პროცესის მიმდინარეობისას.

2.1 ნახაზზე წარმოდგენილი იყო პროგრამული პაკეტის მოდულებად დაყოფის ზოგადი სქემა. ყველაზე ზედა დონეზე მდგომი მოდული მთავარი მოდულია, ანუ ის პროგრამა, რომელიც იწყებს მუშაობას კომპიუტერში კონკრეტული ამოცანის გაშვებისას. მთავარი (ან ზედა

დონის) მოდული უკავშირდება ქვედა დონის მოდულებს, გადასცემს მათ მონაცემებს და ღებულობს შედეგებს მათგან.

მოდულურობის პრონციპის გამოყენება ხასიათდება შემდეგი უპირატესობით:

– დიდი პროგრამული პაკეტი შეიძლება დაიწეროს ერთდროულად რამდენიმე პროგრამისტის მიერ, რაც საგრძნობლად შეამცირებს შესრულების ვადებს;

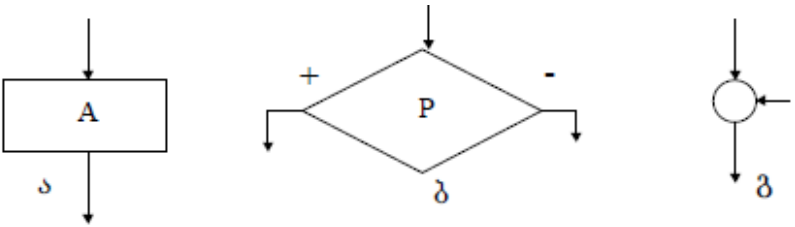
– შესაძლებელია პროგრამული (სტანდარტული) ბიბლიოთეკების შექმნა;

– სისტემების დაპროექტებისა და პროგრამული რეალიზაციის მართვის პროცესისათვის არსებობს მეტი ბუნებრივი საკონტროლო წერტილები, რაც აადვილებს მის ტესტირებას;

– პროგრამული პაკეტის განახლებისათვის მცირე დრო და რესურსებია საჭირო, ვინაიდან მოდიფიკაცია შეეხება არა მთლიან სისტემას, არამედ მის ცალკეულ მოდულებს. მოდიფიკაციაში ვგულისხმობთ ახალი მოდულის (ამოცანის) დამატებას, ძველის შეცვლას ან ამოშლას. უკანასკნელ შემთხვევაში განსაკუთრებული ყურადღება უნდა მიექცეს მოდულთშორისი კავშირების შენარჩუნებას.

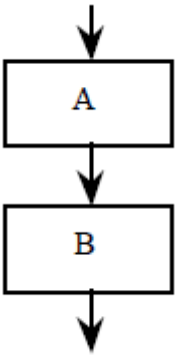
2.4. მმართველი სტრუქტურები

პროგრამულ მოდულში ნებისმიერი ლოგიკური ამოცანის რეალიზება შეიძლება ისეთი სტრუქტურებით, როგორიცაა მიმდევრობა, განშტოება და ციკლი (განმეორება). არასტრუქტურული პროგრამის სტრუქტურულში გადასაყვანად, ელემენტების სწორად ორგანიზებაა საჭირო. 2.2 ნახაზზე ნაჩვენებია მათი გრაფიკული გამოსახვის საშუალებანი.



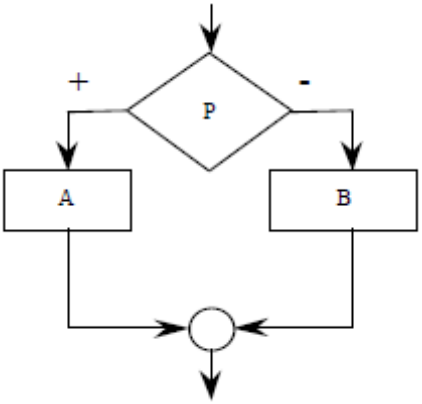
ნახ.2.2. დამუშავების ბლოკი (ა), შემოწმების ბლოკი (ბ), შერწყმის ბლოკი (გ)

A შეიძლება იყოს პროგრამის ოპერატორი, სხვა პროგრამის გამოძახების ფუნქცია ან ქვეპროგრამა; P შესამოწმებელი პრედიკატი, ანუ ლოგიკური პირობაა. „+“ ნიშნავს True (ჭეშმარიტს), ხოლო „-“ ნიშნავს False (მცდარს). შემოწმების ბლოკს ყოველთვის ერთი შესავალი და ორი გამოსავალი აქვს. შერწყმის კვანძი ის წერტილია, სადაც თავს იყრის ორი ან მეტი შესავალი და ერთი გამოსავალი. აქ არავითარი გარდაქმნის პროცესი არ სრულდება. ნახაზზე ყველგან მონაწილეობს შემაერთებელი ისრები, რომლებიც ბლოკ-სქემაში პროცესების მართვის მიმართულებას აღწერს (ნახ.2.3).

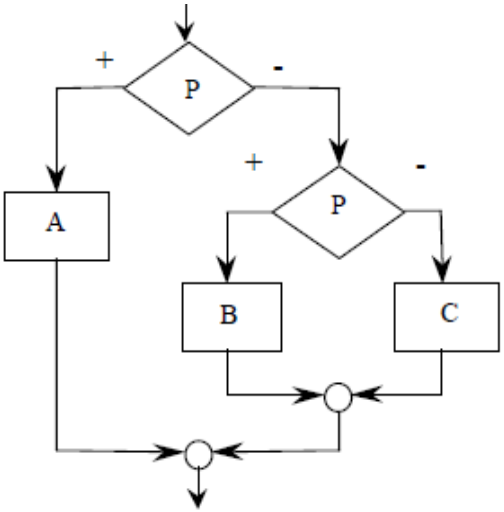


ნახ.2.3. მიმდევრობა

ამ საბაზო ელემენტების საფუძველზე შეიძლება წარმოვადგინოთ *სტრუქტურული დაპროგრამების* ძირითადი მმართველი სტრუქტურების სქემები (ნახ. 2.4-2.8).

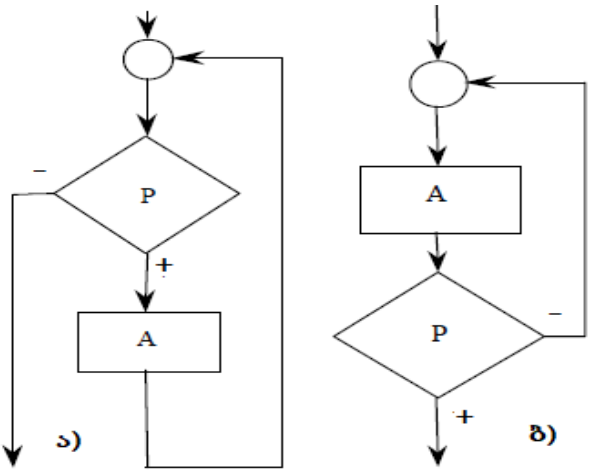


ნახ.2.4. განშტოება
(თუ ... , მაშინ ...)

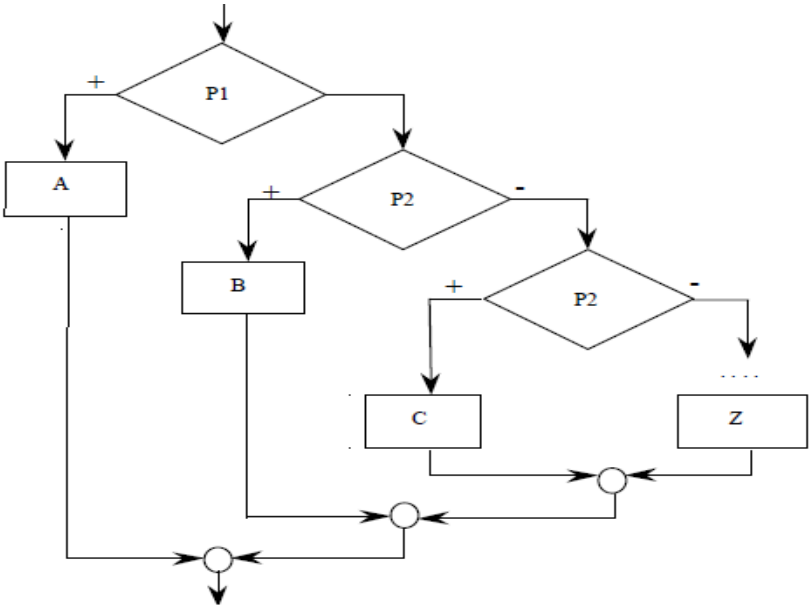


ნახ.2.5. ჩალაგებული
სტრუქტურა განშტოებით

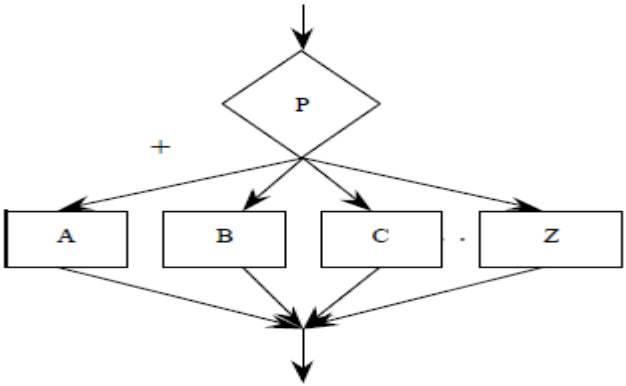
2.6-ბ ნახაზზე პირველად სრულდება A პროცედურა, შემდეგ მოწმდება პრედიკატი. ამ ორ სტრუქტურას შორის ის განსხვავებაა, რომ ა-ში შეიძლება A საერთოდ არ შესრულდეს, ხოლო ბ-ში ერთხელ მაინც შესრულდება აუცილებლად.



ნახ.2.6. ციკლები A პროცედურის შესრულებით P პრედიკატის წინასწარი (ა) და შემდგომი (ბ) შემოწმებით



ნახ.2.7. ამორჩევა განშტოებით (თუ..., მაშინ.....)

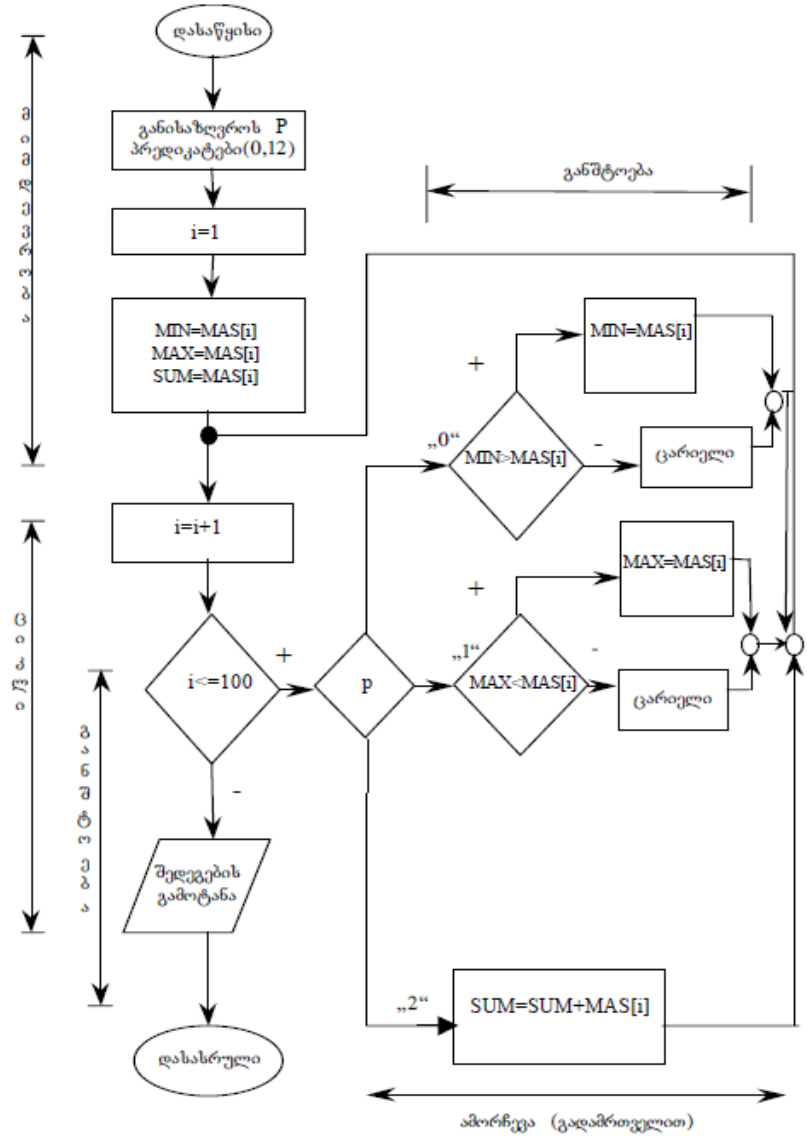


ნახ.2.8. ამორჩევა გადამრთველით

2.7 და 2.8 ნახაზებზე წარმოდგენილია ამორჩევის სტრუქტურული სქემები. ორივე ერთსა და იმავე შედეგს იძლევა. პირველში რეალიზებულია („თუ ... , მაშინ...“) კონსტრუქციის ჩალაგებული სტრუქტურები, მეორეში თვისებრივად ახალი ელემენტებია შემოტანილი გადამრთველის (switch) სახით. პროგრამულ მოდულებში, რომლებშიც ბევრია ჩალაგებული სტრუქტურები (მაგალითად, მენიუს ტიპის), უმჯობესია გადამრთველის გამოყენება. პროგრამა გასაგები, კომპაქტური და ეფექტურიცაა.

განვიხილოთ კონკრეტული მაგალითი იმ ბლოკ-სქემის აგებით, რომელიც მოიცავს ზემოაღნიშნულ სტრუქტურულ ელემენტებს.

დავუშვათ, გვაქვს 100 ნამდვილი რიცხვის მასივი mas[100]. პროგრამამ უნდა მოძებნოს ამ რიცხვებიდან min-მინიმუმი (თუ პრედიკატი p=0), max-მაქსიმუმი (თუ p=1), ან ყველა რიცხვის ჯამი sum (თუ p=2). 2.9 ნახაზზე მოცემულია ეს სქემა.



ნახ.2.9. სტრუქტურული ბლოკ-სქემა

2.5. ფსევდოკოდები

ფსევდოკოდები არის პროგრამის ლოგიკის ასახვის საშუალება, რომელიც შეიძლება გამოყენებულ იქნეს ბლოკ-სქემის ნაცვლად. იგი ბუნებრივსა და მანქანურს შორის მდგომი ფორმალური ენაა, არ მოითხოვს დაპროგრამების კონკრეტული ენის სინტაქსის ასახვას. მასში გამოიყენება მმართველი სტრუქტურები და სტრიქონების ჩაწერის საფეხურებრივი აბზაცები. ფსევდოკოდი იქმნება დაპროგრამებამდე. ფსევდოკოდის მაგალითი 2.9 ნახაზზე მოცემული ბლოკ-სქემისათვის ნაჩვენებია 2.10 ნახაზზე.

```
პროგრამის დასაწყისი
    100-ელემენტიან მასივში საწყისი მნიშვნელობების შეტანა, ქვეამოცანის
რეტომის პრედიკატის მნიშვნელობის განსაზღვრა: P-კოდი, ინდექსი i=1
    min, max და sum ცვლადებზე მასივის პირველი ელემენტის
მნიშვნელობის მინიჭება
    ციკლი-სანამ i<=100
      ამორჩევა P-კოდი
            შემთხვევა 0
                  თუ min > მასივის ელემენტზე
                        მაშინ min=მასივის ელემენტი
                     არადა
                        ცარიელი ოპერატორი
                  თუ–დასასრული
            შემთხვევა 1
                  თუ max < მასივის ელემენტზე
                        მაშინ max=მასივის ელემენტი
                     არადა
                        ცარიელი ოპერატორი
                  თუ–დასასრული
             შემთხვევა 2
                  sum = sum + მასივის ელემენტი
      ამორჩევის–დასასრული
      ინდექსის გაზრდა i=i+1
ციკლის-დასასრული
შედეგების გამობეჭდვა
პროგრამის დასასრული
```

ნახ.2.10. ფსევდოკოდის მაგალითი

2.6. ბიჯური დეტალიზაცია და პროგრამული კოდის სეგმენტაცია

დაღმავალი სტრუქტურული პროგრამირების ტექნოლოგია ითვალისწინებს პროგრამული პაკეტის დაპროექტების ეტაპზე მის დაყოფას ფუნქციურ მოდულებად და მანქანაზე დაპროგრამებას ზემოდან-ქვემოთ. შემადგენელი ფუნქციების სირთულისა და მათი შესრულების შინაგანი ლოგიკის მიხედვით პროგრამული მოდულები შეიძლება კვლავაც დაყოფილ იქნას უფრო დეტალურ ნაწილებად.

ბიჯური დეტალიზაცია იტერაციული პროცესია, რომელიც პროგრამული მოდულის შემდგომ დეკომპოზიციას ეთანადება. იტერაციის ყოველ მომდევნო ბიჯზე მოდულის ფუნქციების უფრო დეტალური სიღრმისეული საკითხები განიხილება, რომლებიც პირველ ბიჯზე არ განიხილებოდა. ბიჯების რაოდენობა დამოკიდებულია კონკრეტულ ამოცანაზე, რომელიც პროგრამულ მოდულში აისახება.

ბიჯური დეტალიზაცია აღიწერება ფსევდოკოდის საშუალებით. პირველ ბიჯზე იგი ზოგადი ხასიათისაა, შემდგომ და ბოლო ბიჯებზე კი თანდათან უახლოვდება დაპროგრამების ენის წინადადებებს (ოპერატორებს), რაც იტერაციის პროცესის დამთავრების მაუწყებელია.

თუ პროგრამული მოდულები დიდია, ისინი შეიძლება დაიყოს სეგმენტებად. *სეგმენტი* არის მოდულის ლოგიკური ან ფიზიკური ნაწილი. ლოგიკურად იგი მოდულის ფუნქციის ქვეფუნქციაა. ფიზიკურად სეგმენტი შემოისაზღვრება პროგრამის საწყისი ტექსტის სტრიქონების იმ რაოდენობით, რომლებიც ერთ საბეჭდ გვერდზე თავსდება (დაახლოებით 50-60 სტრიქონი).

III თავი
ობიექტ-ორიენტირებული პროგრამირების მეთოდი

3.1. ობიექტ-ორიენტირებული დაპროგრამების არსი

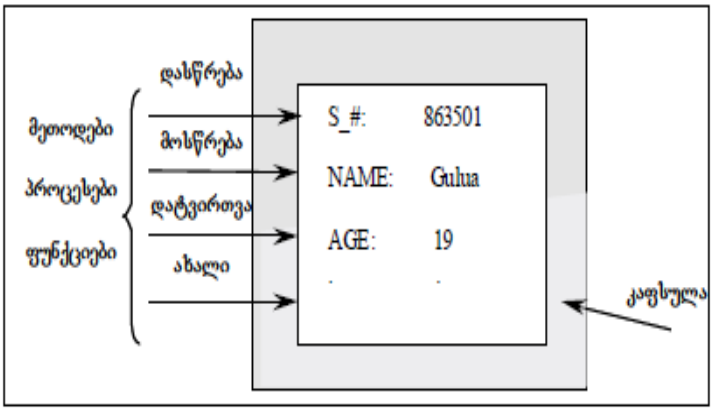
ობიექტ-ორიენტირებული (ანუ ობიექტზე ორიენტირებული) დაპროგრამება თანამედროვე ინფორმაციული ტექნოლოგიის ერთერთი აქტუალური და მძლავრი მეთოდოლოგიური საშუალებაა. მისი მიზანია დიდი, რთული პროგრამული სისტემების კონსტრუირება (Software Engineering). იგი თვისებრივად ახალი კონცეფციების მატარებელი დაპროგრამების ტექნოლოგიაა. სისტემების ობიექტორიენტირებული ანალიზისა და ობიექტორიენტირებული დაპროექტების მეთოდებითა და რეალიზაციის მოქნილი ინსტრუმენტული საშუალებებით, მთლიანად მოიცავს სტრუქტურული დაპროგრამების იდეოლოგიასაც.

ობიექტ-ორიენტირებული დაპროგრამების აქტუალურ მნიშვნელობაზე მეტყველებს ის ფაქტიც, რომ ახალი კომპიუტერული სისტემების შეფასების ერთ-ერთ პირველ კრიტერიუმად ობიექტორიენტირებული მიდგომის რეალიზებადობას მიიჩნევენ.

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

ორიენტირებული პროგრამებისა, ამიტომ დიდი მნიშვნელობა აქვს მის სწორად გაგებას. 3.1 და 3.2 ნახაზებზე განხილულია ბიოლოგიური და ინფორმაციული უჯრედის მოდელები.

ნახ.3.1. ბიოლოგიური უჯრედი



ნახ.3.2. ინფორმაციული უჯრედი

ბიოლოგიური უჯრედი კაფსულირებულია მემბრანის (გარსის) საშუალებით, რომლითაც ის გამოიყოფა სხვა უჯრედებისა და გარემოსაგან. მას უნარი აქვს გარემოსგან მიიღოს ქიმიური სახის ინფორმაცია და გადასცეს უჯრედს შიგნით. შუაგულში მოთავსებულია ბირთვი, რომელიც უჯრედის ძირითადი ინფორმაციის მატარებელია. იგი შედგება ქრომოსომებისგან, რომლებიც გარკვეული გენეტიკის მქონეა. უჯრედის დაყოფის (გამრავლების) დროს ხდება მემკვიდრული თვისებების გადაცემა. ბირთვის გარშემო დაჯგუფებულია სხვადასხვა ფუნქციის ელემენტები. მაგალითად, ენდოპლაზმური ბადე – ცილების წარმოქმნის ფუნქცია, მიტოქონდრები – ენერგიის გარდაქმნის ფუნქცია, ციტოპლაზმა (თხევადი მოძრავი გარემო) – ტრანსპორტირების ფუნქცია და ა.შ.

როგორია „ინფორმაციული უჯრედის" აგებულება და რა ანალოგიაა ბიოლოგიურთან ?

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

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

და ა.შ.

კაფსულაში შეღწევა და მონაცემების დამუშავება გარედან ყველა ფუნქციას არ შეუძლია, არსებობს წინასწარ განსაზღვრული მეთოდები (პროცესები ან ფუნქციები), რომელთაც ზოგადად სერვისული პროგრამები შეიძლება ვუწოდოთ. მათ აქვს უნარი შეაღწიოს კაფსულაში და გადაამუშაოს მონაცემები. ამგვარად, ინფორმაციული

ობიექტი კაფსულირებული მონაცემებისა და მათი დასამუშავებელი მეთოდებისაგან შედგება.

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

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

კლასის ცნება ამერიკელმა მეცნიერმა გრადი ბუჩმა (1990-იანი წლები) დაუკავშირა ობიექტთა სიმრავლის ისეთ სტრუქტურას, რომლის იერარქია გარკვეული მემკვიდრეობით და ქცევით განისაზღვრება. ამრიგად, ობიექტი კლასის კონკრეტულ გამოვლინებად, ეგზემპლარად შეიძლება განვიხილოთ [13].

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

კლასისა და ობიექტის მოდელირების და დაპროგრამების საკითხები ობიექტ-ორიენტირებული ენების (მაგალითად, C++, Java, C#, Python და სხვ.) შესწავლისას განხილება მრავალ ნაშრომში [25-28].

3.2. ობიექტები და კლასები. მონაცემთა აბსტრაქტული ტიპები

ობიექტები და კლასები ობიექტორიენტირებული დაპროგრამების ძირითადი კომპონენტებია. ობიექტის ცნებას, როგორც ინფორმაციული უჯრედისა, წინა პარაგრაფში გავეცანით. ახლა დავაკონკრეტოთ მისი განსაზღვრება და ავხსნათ ამ ცნების მიმართება კლასის ცნებასთან.

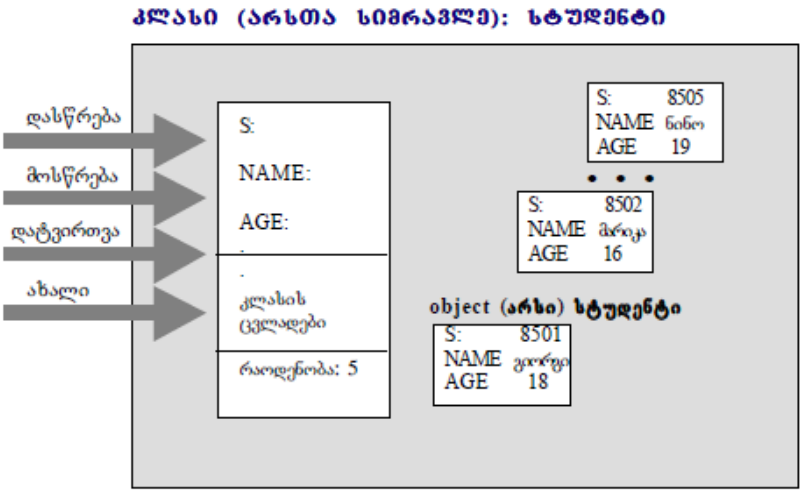
ობიექტი, როგორც აღვნიშნეთ, რეალური ან წარმოსახვითი სამყაროს ნაწილი, ინდივიდუალური ეგზემპლარია. ის არსია (Entity) და აქვს უნიკალური იდენტიფიკატორი, რომლითაც სხვა არსებისაგან განსხვავდება. ობიექტს აქვს ინდივიდუალური *თვისებები*, რომლებიც გამოიხატება მონაცემებით (ატრიბუტები, ცვლადები) და *ქცევით* (მეთოდები,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

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

ობიექტის ცნება დაპროგრამების ენაში პირველად გამოყენებულ იქნა Simiula-ში, ხოლო მისი ზემოაღნიშნული

კლასიკური განმარტება მოგვცა გრადი ბუჩმა [13]. მან ასევე ჩამოაყალიბა კლასის განმარტება.

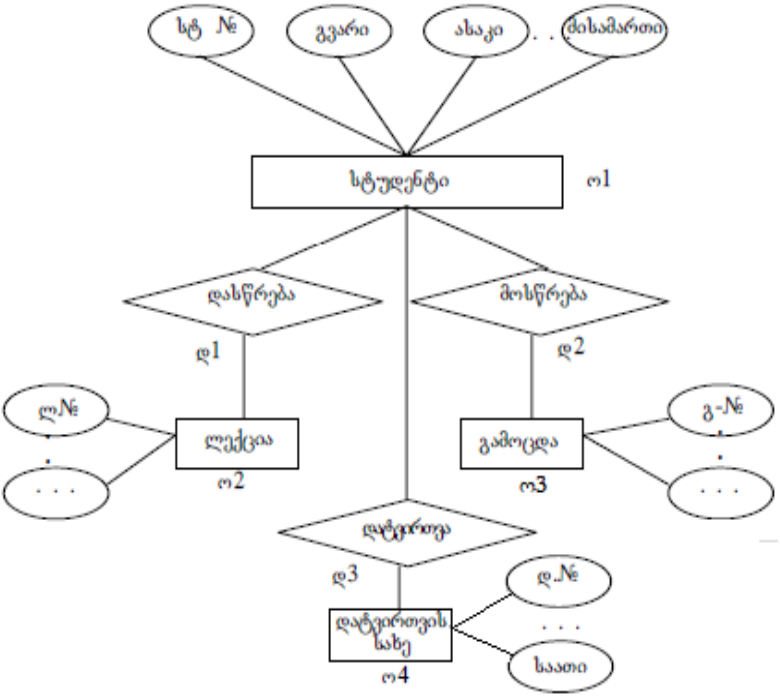
კლასი არის ერთი ტიპის ობიექტების სიმრავლე, რომელთაც აქვს მსგავსი სტრუქტურა და ქცევა. 3.3 ნახაზზე ნაჩვენებია ერთი კლასის მაგალითი.



ნახ.3.3. კლასი და ობიექტები

ვინც იცნობს მონაცემთა რელაციური ბაზების თეორიას და არსთა-დამოკიდებულებათა (Entity-Relationhip) მოდელის აგების საკითხებს, ადვილად შეამჩნევს გარკვეულ ანალოგიას საპრობლემო სფეროს კონცეპტუალური მოდელის, ლოგიკური სტრუქტურისა და მისი ფიზიკური ორგანიზაციის რეალიზაციასთან. მსგავსება შემდეგი თვალსაზრისით შეიძლება იქნეს განხილული:

კლასი ***სტუდენტი*** ეთანადება 3.4 ნახაზზე წარმოდგენილი კონცეპტუალური სქემის ფრაგმენტს.



ნახ.3.4. საპრობლემო გარემოს კონცეპტუალური (ER) მოდელი
ო-ობიექტი, დ - დამოკიდებულება

ამ კონცეპტუალური სქემის გადატანით მონაცემთა ბაზების ლოგიკურ სტრუქტურაში მივიღებთ სქემას, რომელიც ახლოა კლასის მოდელთან. ლოგიკური სტრუქტურა ატრიბუტებისგან შემდგარი სქემაა, რომელიც ობიექტების სტატიკურ კომპონენტებად მოვიხსენიეთ ზემოთ. კლასის

ობიექტები კი ეთანადება ამ ლოგიკური სტრუქტურის ქვეშ მდგარ ფიზიკურ ჩანაწერებს (ჩანაწერის უნიკალური ნომრითა და ველების მნიშვნელობებით).

მონაცემთა რელაციური ბაზების თეორიაში გამოიყენება მონაცემებისა და პროგრამების ერთმანეთისაგან იზოლირების პრინციპი, რათა განხორციელდეს მათ შორის სრული დამოუკიდებლობა.

მონაცემთა აბსტრაქტული სტრუქტურების გამოყენებით ეს საკითხი დადებითად იქნა გადაჭრილი.

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

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

```
class Magistrant {
    private:
        char Name [20];
        int Age;
        char Specification [30];
```

```
public:
        Input-Name (NAME);
        Input-Age (AGE);
        Input-Spec (SPEC); };
void main(void) {
         Magistrant M1,M2,M3.....; // da misi obieqtebic:
. . . }
```

ამგვარად, Magistrant ისეთივე ტიპია, როგორც int, char და ა.შ. ყოველი ობიექტი M1,M2,M3..... არის კონკრეტული მაგისტრანტი, რომელიც სტრუქტურას იღებს class Mgistrant(....)-იდან private და public ოპერატორები განსაზღვრავს მონაცემებს (Name, Age, Specifikation) და ფუნქციებზე (Input-Name, Input-Age, Input-Spec) მიმართვის შესაძლებლობას. პირველი ლოკალურია და მალავს ამ მონაცემებს სხვა ობიექტებისათვის. მათთან მიმართვა შეუძლია მხოლოდ მოცემული ობიექტის ფუნქციებს და ზოგჯერ მათ „მეგობრებსაც“- friend). Public იძლევა ნებართვას, რათა მის შემდეგ მდგომი ფუნქციები გამოყენებულ იქნას ობიექტის გარედან. განხილული მაგალითის განზოგადებით შეიძლება დავასკვნათ, რომ *კლასების საფუძველი მონაცემთა აბსტრაქტული ტიპებია.*

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

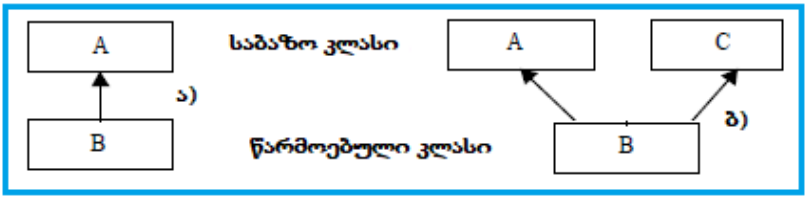
3.3. კლასების იერარქია, მემკვიდრეობითობა

ობიექტორიენტირებული პროგრამირების სტილის საბაზო პრინციპებია *ინკაფსულაცია, მემკვიდრეობითობა და პოლიმორფიზმი.* პირველი მათგანი წინა პარაგრაფში განვიხილეთ და კლასის ცნებამდე მივედით. კლასებს შორისაც არსებობს გარკვეული დამოკიდებულებანი. რას ნიშნავს ეს ?

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

ახალი კლასები უნდა განისაზღვროს არსებულის ბაზაზე, მოხდეს მათი გაფართოება და მოდიფიკაცია. ყოველივე ეს მნიშვნელოვნად ამცირებს ახალი სისტემების დაპროექტებისა და რეალიზაციის ვადებს. სწორედ ამაშია ობიექტორიენტირებული პროგრამირების მეთოდის გამოყენების ეფექტურობის საიდუმლოება.

ორი კლასიდან ერთი ბაზისური, ხოლო მეორე წარმოებულია. 3.5 ნახაზზე ნაჩვენებია მარტივ-მემკვიდრეობითი (single inheritance) და მრავალ-მემკვიდრეობითი (multiple inheritance) იერარქიული კავშირები.



ნახ.3.5. მემკვიდრეობითობა

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

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

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

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

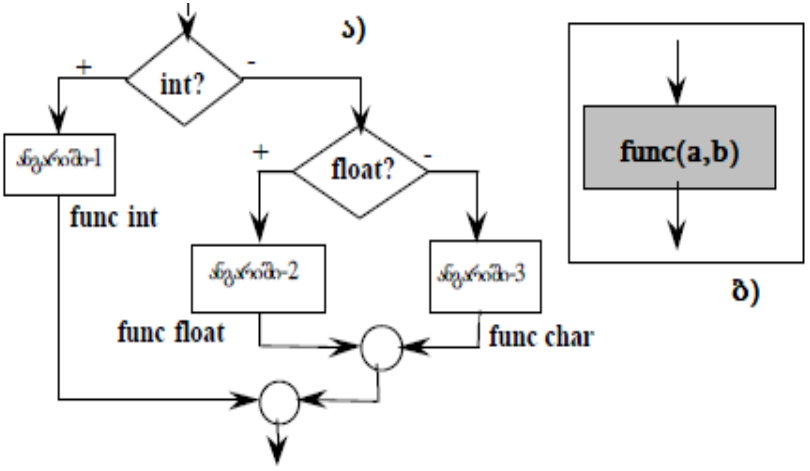
3.4. პოლიმორფიზმი

ობიექტორიენტირებულ პროგრამირებაში ინკაფსულაციისა და მემკვიდრეობითობის გვერდით მნიშვნელოვანი ადგილი უკავია *პოლიმორფიზმის* ცნებას. ის ბერძნული სიტყვაა polymorphism და „მრავალფორმიანობას“ ნიშნავს.

პოლიმორფიზმი ობიექტის თვისებაა. იგი უზრუნველყოფს ერთსა და იმავე ფუნქციების გამოყენებას სხვადასხვა ამოცანათა გადასაწყვეტად. ეს ამოცანები შეიძლება

მოითხოვდეს ფუნქციების და მათი არგუმენტების სხვადასხვა ტიპებს, რასაც პოლიმორფიზმი ადვილად წყვეტს ე.წ. *ვირტუალური ფუნქციების* რეალიზაციით (virtual function) ან *ფუნქციათა გადატვირთვით* (overloaded function).

მონომორფულ სისტემებში ყოველი ფუნქცია და მისი არგუმენტები მხოლოდ ერთი ტიპითაა შეზღუდული. მაგალითად, ჩვეულებისამებრ, C ენაში საჭიროა დაიწეროს ორი სხვადასხვა ფუნქცია int - func(int, int) და float – func(float, float), თუკი გვინდა ორ მთელ რიცხვზე ან ორ ნამდვილ რიცხვზე არითმეტიკული ოპერაციების ჩატარება (ნახ. 3.6-ა). პოლიმორფიზმის იდეა C++ ენაში საშუალებას გვაძლევს დავწეროთ მხოლოდ ერთი func(a,b) ფუნქცია, შემდეგ კი, არგუმენტების ტიპების ანალიზის საფუძველზე, ენის კომპილატორი თვითონ აირჩევს, თუ რომელი ოპერაციები შეასრულოს (ნახ.3.6-ბ).



ნახ.3.6. პოლიმორფიზმი (ბ) C++

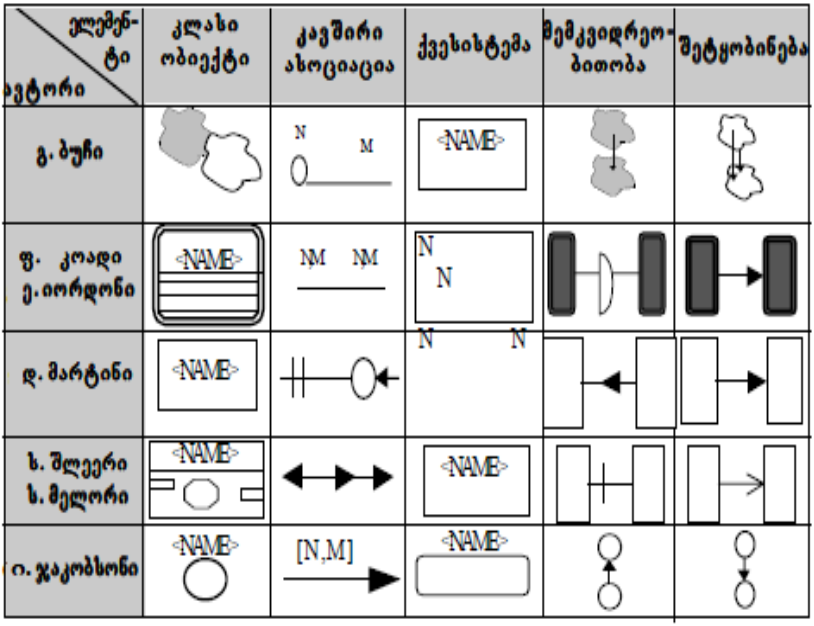
3.5. ობიექტ-ორიენტირებული დიაგრამების აგება სისტემების დაპროექტების ეტაპზე

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

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

3.7 ნახაზზე წარმოდგენილია ობიექტ-ორიენტირებული ანალიზისა და დაპროექტების ცნობილი კლასიკოსების ბუჩის, კოად-იორდონის, ჯაკობსონის, მარტინისა და შლეერ-მელორის გრაფიკული აღწერის საშუალებანი [23]. რა თქმა უნდა, არსებობს სხვა სისტემებიც, მაგრამ ამ სფეროში ძირითადად ზემოაღნიშნული ინსტრუმენტული საშუალებები დომინირებს.

ამ საკითხების დეტალური და უფრო ღრმა გადმოცემა აქ შეუძლებელია, მისი მოცულობის გამო. იგი ცალკე შესასწავლი საგანია [28-30].



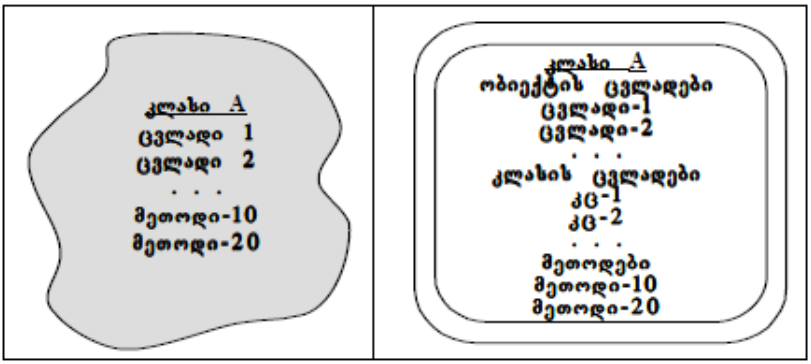
ნახ.3.7. ოო-დიაგრამების აგების აღნიშვნათა სისტემა

ზოგიერთი კომენტარი ნახაზთან დაკავშირებით.

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

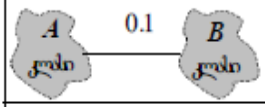
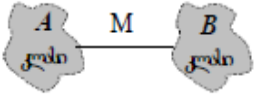
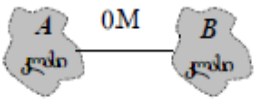
3.8 ნახაზზე გადმოცემულია კლასის წარმოდგენა ოო - დიაგრამისთვის ბუჩისა და კოად-იორდონის აღნიშვნათა სისტემების მაგალითზე.

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



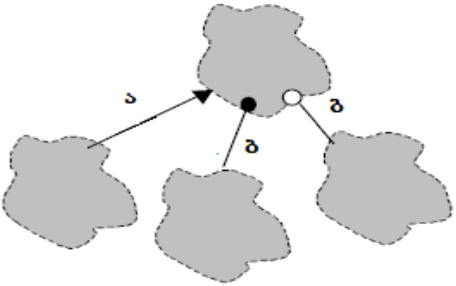
ნახ.3.8. კლასის წარმოდგენა ოო-დიაგრამით გ. ბუჩისა (ა) და კოად-იორდონის (ბ) მიხედვით

– *კავშირის ასოციაციები* კლასებს ან ობიექტებს შორის მრავალი სახისაა. ამიტომაც 3.9 ნახაზზე მოცემული გვაქვს მათი ძირითადი ტიპები. ესაა ტიპური მათემატიკური ასახვის ფუნქციები, დამოკიდებულებანი კლასებსა და მის ელემენტებს შორის.

	$A \rightarrow B$	A კლასის ყოველ ობიექტს შეესაბამება B კლასის მხოლოდ 1 ობიექტი
	$A \backslash B$	A კლასის ყოველ ობიექტს შეესაბამება B კლასის არც ერთი ან ერთი ობიექტი
	$A \rightarrow\rightarrow B$	A კლასის ყოველ ობიექტს შეესაბამება B კლასის ერთი ან რამდენიმე ობიექტი
	$A \backslash\backslash B$	A კლასის ყოველ ობიექტს შეესაბამება B კლასის 0, 1 ან რამდენიმე ობიექტი

ნახ.3.9. ძირიად კავშირთა ასოციაციები

შეიძლება განვიხილოთ აგრეთვე ისეთი დამოკიდებულებები, რომლებიც სტრუქტურულ მოწესრიგებას ემსახურება, მაგალითად, *კლასიფიკაცია* და *აგრეგაცია*. პირველი მათგანი აერთიანებს ობიექტებს გარკვეული კრიტერიუმებით, რომლებიც მსგავს, მაგრამ არაიდენტურ თვისებებს ეყრდნობა. მეორე კი მთელისა და შემადგენელი ნაწილის მიმართების ტიპური ასახვაა. 3.10 ნახაზზე ნაჩვენებია სქემატურად კლასებს შორის მემკვიდრეობითი, აგრეგირებული და რელაციური კავშირების გამოსახვა.



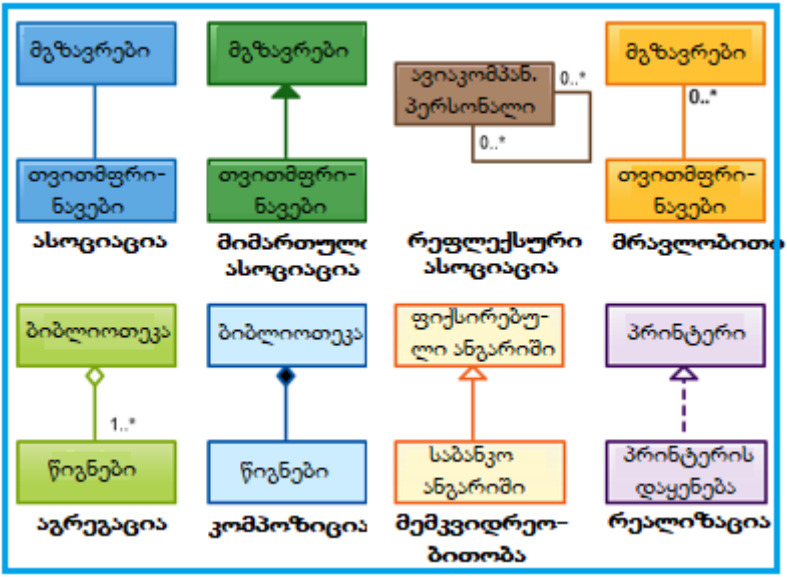
ნახ.3.10. კლასთაშორის ასოციაციები: მემკვიდრეობითი (ა), აგრეგირებული (ბ) და რელაციური (გ) /გ. ბუჩი, 1996/

შენიშვნა: UML მეთოდოლოგია, მისი ინსტრუმენტული საშუალებები საგრძნობლად განვითარდა. 2018 წლისთვის UML/2-ის ბოლო ვერსია არის 2.5.1 [31]. განვითარებაში იგულისხმება ამ მეთოდოლოგიის შემადგენელი კომპონენტებისსა და კლასთა ასოციაციური კავშირების გაფართოება და სრულყოფა საკვლევი ობიექტის ასახვის სემანტიკური სიმძლავრის ამაღლების მიზნით.

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

ბოლო პერიოდში მეტი პრიორიტეტი შეიძინა UML-ის კონკურენტმა მეთოდოლოგიამ, რომელიც Agile სახელწოდებითაა ცნობილი (მაგალითად, ექსტრემალური პროგრამირება (XP), Scrum, Kaban და სხვ.) [32].

UML/2-ის კლასთა ასოციაციის კავშირების საილუსტრაციო მაგალითები მოყვანილია 3.11 ნახაზზე, მისი დეტალური ანალიზი კი მრავლად არის წარმოდგენილი ინტერნეტ-წყაროებში [33].



ნახ.3.11. კლასთა ასოციაციური კავშირები (UML/2.5.1,2018)

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

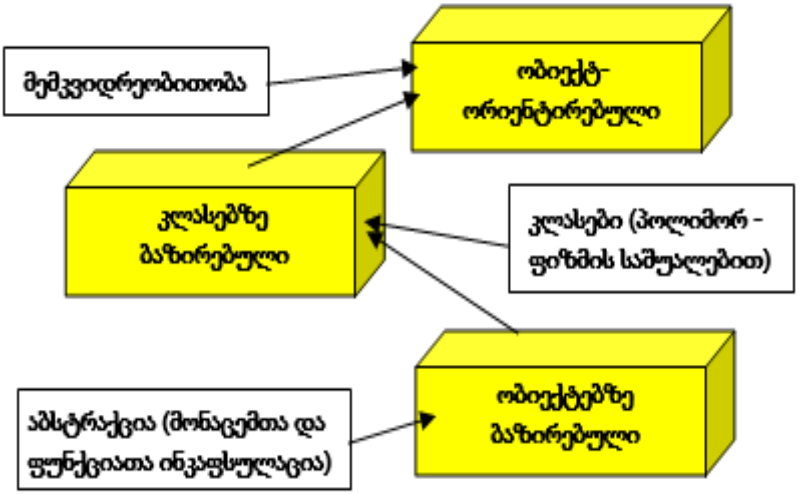
1-ელი სუბიექტური ჯგუფი შეიძლება იყოს ***სასწავლო აუდიტორიები, ლაბორატორიები, კომპიუტერული კლასები*** და სხვ.;

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

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

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

მემკვიდრეობითობა ობიექტორიენტირებული მიდგომის ერთ-ერთი ძირითადი და აუცილებელი კომპონენტია. 3.12 ნახაზზე მოცემულია ოო-მიდგომის ერთ-ერთი ძირითადი პრინციპის მოდელი, რომლის „მწვერვალზეც“ სწორედ მემკვიდრეობითობის კონცეფციის რეალიზებით (ენაში) ვაღწევთ ასვლას.

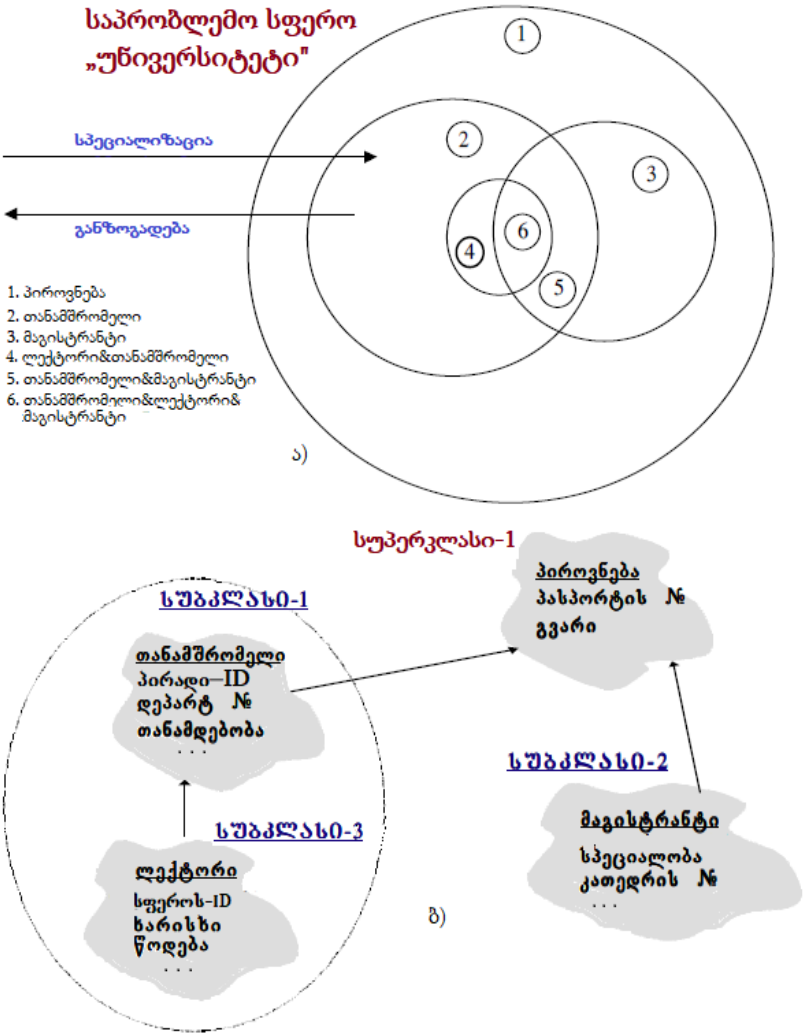


ნახ.3.12. ოო-მიდგომის პრინციპის სქემა

განვიხილოთ საპრობლემო გარემო უნივერსიტეტი (ნახ.3.13). მემკვიდრეობითი სტრუქტურები გამოიკვეთება სპეციალიზაციითა და განზოგადებით, რომლებიც ურთი-ერთსაპირისპირო პროცედურებია.

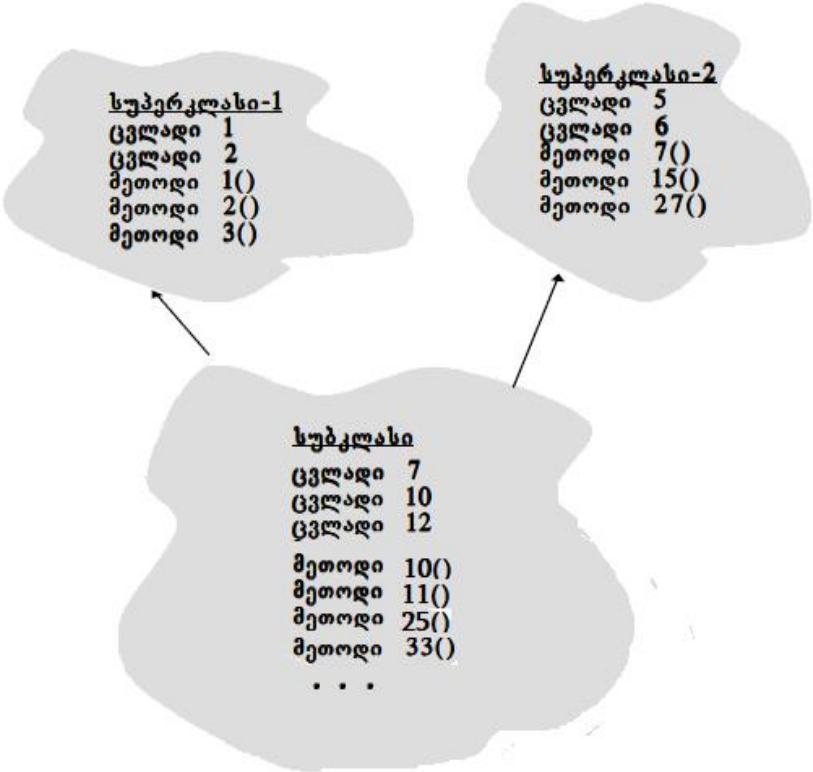
მაგალითად, უნივერსიტეტში არსებული ადამიანები განსაზღვრული დროის მომენტში შეიძლება დახარისხდეს სტატუსის მიხედვით: *პიროვნება, თანამშრომელი, მაგისტრანტი, ლექტორი* და ა.შ.

შეიძლება ადგილი ჰქონდეს მათ გადაკვეთასაც.



ნახ.3.13. მემკვიდრეობითობის სტრუქტურა: სპეციალიზაცია და განზოგადება (ა), მათიწარმოდგენა ოო-დიაგრამით (ბ)

ხშირად საჭიროა გამოისახოს *მრავალჯერადი მემკვიდრეობითობის კავშირი*. ამ დროს „შვილი“ (სუბკლასი) იღებს მემკვიდრეობის „გენებს“ (ცვლადებს და მეთოდებს) თავისი „მშობლებიდან“ (სუპერკლასებიდან). ამავე დროს მათ თვითონაც ექნება განსხვავებული, ახალი თვისებებიც (სხვა ცვლადები და მეთოდები), ზოგადი მაგალითი მოცემულია 3.14 ნახაზზე.



ნახ.3.14. მრავალჯერადი მემკვიდრეობითობის მაგალითი ორი სუპერკლასით

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

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

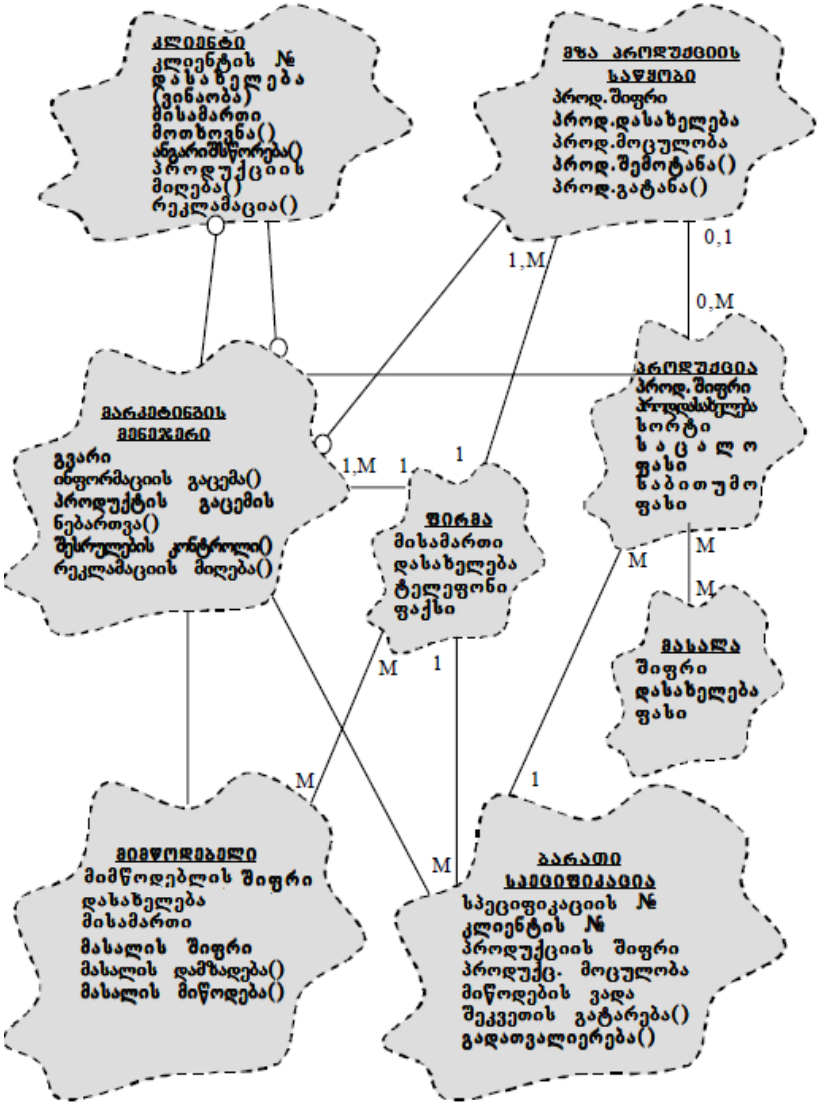
ამ საბაზო ოო-დიაგრამის საფუძველზე შესაძლებელია მოთხოვნების შესრულება. 3.16 ნახაზზე მოცემულია ორი შეტყობინების მაგალითი.

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

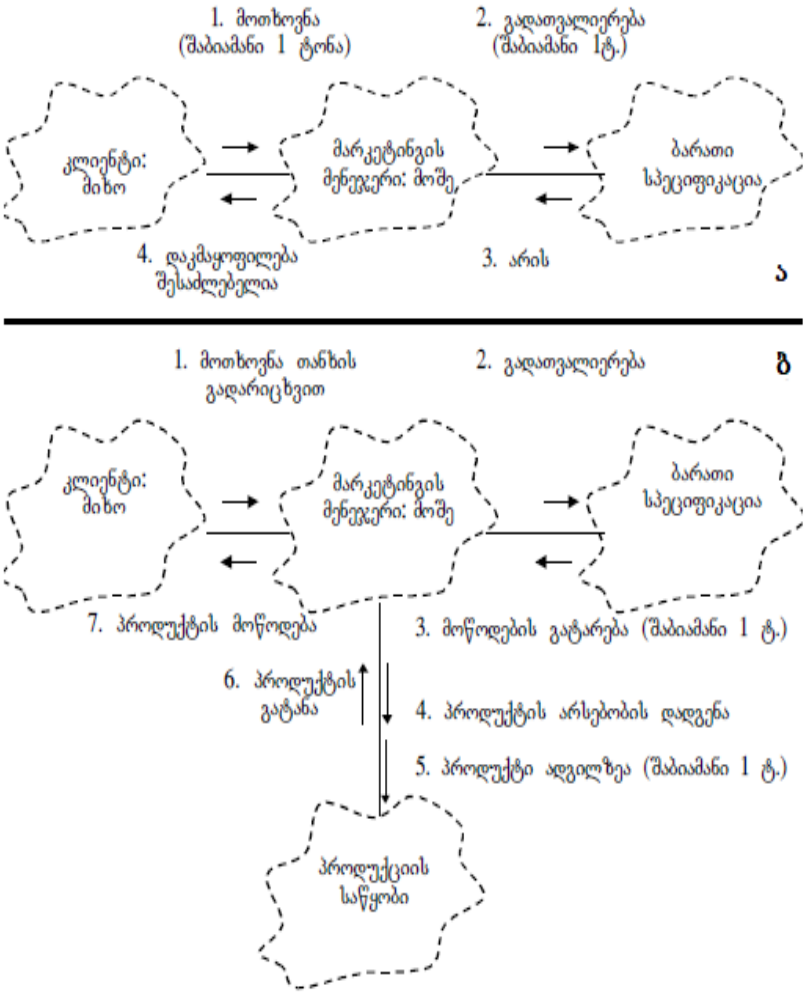
მეორე – კლიენტმა უნდა გაიტანოს ფირმიდან მისი კუთვნილი პროდუქცია (ბ).

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

- პირველი – ობიექტთა მდგომარეობის მოდელირება (პროცესები შეტყობინების დამუშავებამდე) და
- მეორე, ობიექტთა ქცევის მოდელირება (პროცესები შეტყობინების დამუშავებით და შედეგებით).



ნახ.3.15. ოო-დიაგრამა საპრობლემო სფეროსათვის „მარკეტინგი“



ნახ.3.16. შეტყობინების დამუშავების სცენარი
ოო-დიაგრამით

IV თავი

პროგრამირების ვიზუალური მეთოდები და მეთოდოლოგიები

ვიზუალური პროგრამირების ენა (VPL) იყენებს გრაფიკულ ელემენტებს და ფიგურებს პროგრამების ასაგებად. VPL-ის ბაზაზე იქმნება 2D და 3D (განზომილებიანი) პროგრამული აპლიკაციები, რისთვისაც მის რესურსებში წარმოდგენილია გრაფიკული ელემენტები, ტექსტები, სიმბოლოები და სურათები (ნახატები) [34,35]. ვიზუალური ენის იდეალური მაგალითია UML მეთოდოლოგია თავისი დიაგრამებით და CASE ინსტრუმენტული საშუალებები, რა თქმა უნდა, პროგრამული კოდის ავტომატიზებული გენერაციის შესასძლებლობებით [1,14].

რა განსხვავებაა დაპროგრამების მეთოდოლოგიასა და მეთოდებს შორის ?

ზოგადად (ფილოსოფიურად), *მეთოდი* – კვლევის ინსტრუმენტი, კომპონენტია. მაგალითად, საკვლევი ობიექტის რომელიღაც მაჩვენებლის მნიშვნელობის განსაზღვრის რაოდენობრივი ან ხარისხობრივი მეთოდი (ან მეთოდები). *მეთოდოლოგია* კი არის მეთოდების და პრინციპების სისტემური, თეორიული ანალიზი კონკრეტული საკვლევი სფეროსათვის. მეთოდოლოგია არ გვაძლევს პრობლემის გადაწყვეტის შედეგს, ამიტომაც იგი არ ემთხვევა მეთოდს. მეთოდოლოგია გვაძლევს თეორიულ საფუძველს იმისათვის, რომ სწორად განვსაზღვროთ თუ რომელი მეთოდი ან მეთოდთა ჯგუფი და საუკეთესო პრაქტიკები შეიძლება იქნას გამოყენებული კონკრეტულ შემთხვევებში [3],

პროგრამულ ინჟინერიაში *პროგრამული აპლიკაციის დეველოპმენტის მეთოდოლოგია* არის დეველოპმენტის სამუშაოს დაყოფის პროცესი ცალკეულ ფაზებად, დიზაინის, პროდუქტებისა და პროექტების მართვის სრულყოფის მიზნით. იგი ცნობილია აგრეთვე როგორც პროგრამების დეველოპმენტის სასიცოცხლო ციკლი (Software Development Life Cycle – SDLC) [36,37].

დეველპომენტის პროცესების მეთოდოლოგიებია, მაგალითად, ჩანჩქერის (waterfall), იტერაციულ-ინკრემენტალური (iterative and incremental), სპირალური (spiral), აპლკაციების სწრაფი დამუშავების (rapid) და სხვ. [5].

პროგრამული სისტემების დეველოპმენტის *ვიზუალური მეთოდოლოგიების* კარგი მაგალითებია:

- უნიფიცირებული მოდელირების ენა (Unified Modeling Language – UML), რომელიც იყენებს ობიექტ-ორიენტირებული დიზაინის, ობიექტების მოდელირების ტექნიკისა და ობიექტ-ორიენტირებული პროგრამული უზრუნველყოფის ინჟინერიის მეთოდებს და ინსტრუმენტებს. ამ სამი მიდგომის სიმბიოზის სიმძლავრე ძალზე ეფექტურ და თანმიმდევრულ მეთოდოლოგიას ქმნის [3,14];
- მოქნილი (Agile) მეთოდოლოგიები, როგორიცაა Scrum და Kanban [5, 38-40]. მაგალითად, Kanban ძალზე თვალსაჩინო მეთოდია, რომელიც ფართოდ გამოიყენება Agile-პროექტების მართვაში. იგი ხატავს დოკუმენტბრუნვის პროცესის სურათს მისი სუსტი ადგილების ადრეულ ეტაპებზე გამოვლენის მიზნით, რათა უზრუნველყოფილ იქნას უფრო ხარისხიანი პროდუქტი ან მომსახურება.

ჩვენ წინამდებარე თავში გადმოცემული გვაქვს სწორედ UML და Agile მეთოდოლოგიების ძირითადი კონცეფციები, მათი შედგენილობა, სტრუქტურა, ძირითად კომპონენტთა ტიპები და საილუსტრაციო მაგალითები ვიზუალური დაპროგრამების ჭრილში.

4.1. უნიფიცირებული მოდელირების ენა (UML)

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

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

UML არის მსოფლიოში ყველაზე ფართოდ გამოყენებადი უნიფიცირებული მოდელირების ენა. იგი შექმნილია საერთაშორისო ასოციაციის, ობიექტების მართვის ჯგუფის – OMG (Object Management Group) მიერ [30], 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 [31].

UML აერთიანებს მონაცემთა მოდელირების (entity relationship diagrams), ბიზნესმოდელირების (work flows), ობიექტების და კომპონენტების მოდელირების მეთოდებს. ის გამოიყენება დიდი პროექტების პროგრამული უზრუნველყოფის და მისი ტექნიკური რეალიზაციის მთელი სასიცოცხლო ციკლის განმავლობაში.

UML-ის მიზანია იყოს სტანდარტული მოდელირების ენა, რომლის საშუალებითაც შესაძლებელია განაწილებული სისტემების მოდელების შექმნა [28,29].

არის რამდენიმე მიზეზი, რატომ უნდა გამოვიყენოთ UML ენა მოდელირებისთვის:

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

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

UML-ის განვითარება. UML პირველად 1997 წელს გამოჩნდა. მას შემდეგ UML1-ის სხვადასხვა ვერსიები გამოდიოდა 2005 წლის ჩათვლით. გაფართოვდა და დაიხვეწა აქტიურობისა და მიმდევრობითობის დიაგრამები. კლასები გაფართოებულია შიგა სტრუქტურებით და პორტებით ე.წ. კომპოზიციური სტრუქტურებით. დაემატა ინფორმაციული ნაკადები და სხვ.

მას შემდეგ UML2-ის სხვადასხვა ვერსია გამოვიდა: UML2.1-UML2.5, FTF_Beta1 სახელწოდებით, რომელიც ჯერჯერობით ფართოდ არაა ცნობილი [31].

OMG-მ დღესდღეობით UML2.0 ვერსიის სტანდარტიზაცია დაასრულა. ახალი UML2.0 სპეციფიკაცია UML-ისთვის შეიცავს ისეთ სრულყოფილ სიახლეებს, რომელიც რესტრუქტურიზაციას უკეთებს და ხვეწს ენას, რათა ის უფრო ადვილი გამოსაყენებელი, შესასრულებელი და გასაგები გახადოს.

ყველაზე მნიშვნელოვანი ცვლილებები UML1 ვერსიისგან განსხვავებით, რაც UML2.0–ს აქვს, არის ახალი დიაგრამები: კომპოზიციური სტრუქტურის (Composite

structure diagram), დროითი (Timing diagram), ურთიერთქმედების მიმოხილვის (Interactive overview diagram) [31].

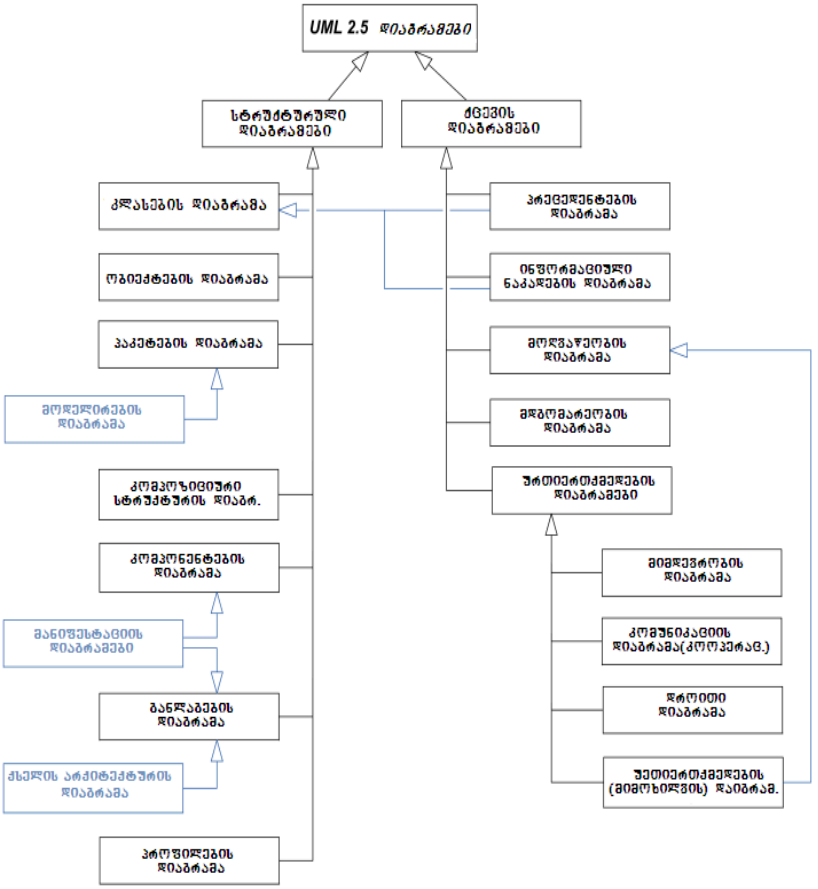
რაც შეეხება მთლიანად UML2-ს, UML1-ისგან განსხვავებით, აქ შემუშვებულია ახალი დიაგრამები: ობიექტების დიაგრამა (object diagrams), პაკეტების დიაგრამა (package diagrams), სტრუქტურის შემადგენლობის დიაგრამა (composite structure diagrams), ურთიერთქმედების მიმოხილვის დიაგრამა (interaction overview diagrams),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

UML2.5.1-ის დადებითი მხარეებია: ახალი სტრუქტურა; არქიტექტურული მოდელირების კონსტრუქციები; პორტები, კავშირები და ნაწილები; ახალი UML2-დიაგრამები; სტრუქტურის შემადგენლობის დიაგრამა;

ქცევის დიაგრამების UML2.5-განახლება. მათ შორის მდგომარეობათა დიაგრამის; ურთიერთქმედების დიაგრამის; აქტიურობათა დიაგრამის და ა.შ. [30].

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 მოცემულია 4.1 ნახაზზე.



ნახ.4.1. UML2.5-ის დიაგრამების სტრუქტურა

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.

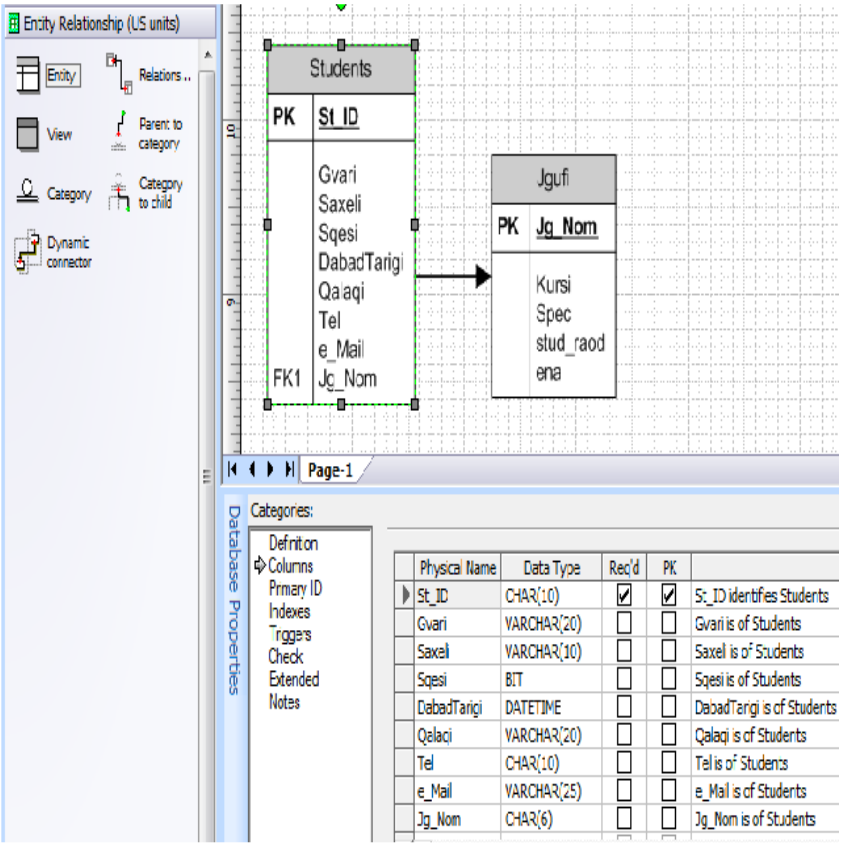
4.1.1. UML-ის კლასიკური დიაგრამები

4.2 ნახაზზე ნაჩვენებია UML/1 ტექნოლოგიის კლასიკური მოდელი 4 წყვილი (ეტაპის) დიაგრამით. მათგან 4 სტატიკური მოდელია და 4 - დინამიკური [14, 41].

ნახ.4.2. UML/1 მეთოდოლოგიის კლასიკური მოდელი

ახლა განვიხილოთ UML ენის ინსტრუმენტული საშუალებები, რომლებიც ფართოდ გამოიყენება დიდი პროგრამული პროექტების განხორციელების მიზნით. შეიძლება რამდენიმე ძირითადი ჩამოვთვალოთ, ვინაიდან CASE ტექნოლოგიების და, განსაკუთრებით, UML-ენის სპექტრი საკმაოდ დიდია. ქვემოთ მოყვანილი მაგალითები აგებულია Ms Visio და Enterprixe Architect ინსტრუმენტებით [42].

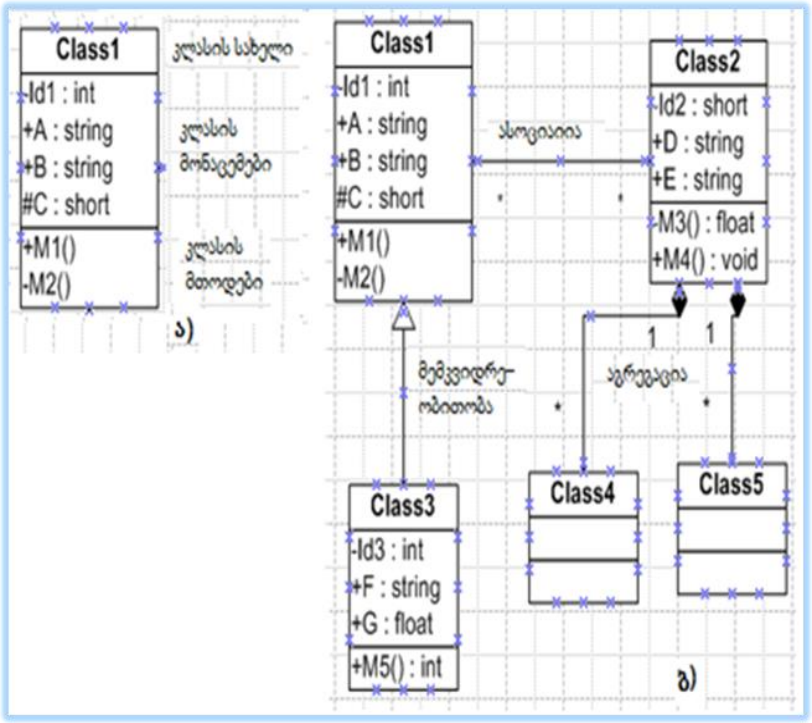
➢ *ER დიაგრამის* ვიზუალური აგება: Database Model Diagram არჩევით ეკრანზე გამოიტანება მონაცემთა ER-მოდელის ასაგები რედაქტორი (Entity-Relationship Model). 4.3 ნახაზზე ნაჩვენებია ინტერფეისი კონკრეტულ „სტუდენტთა-ჯგუფის“ მონაცემთა მოდელისათვის.



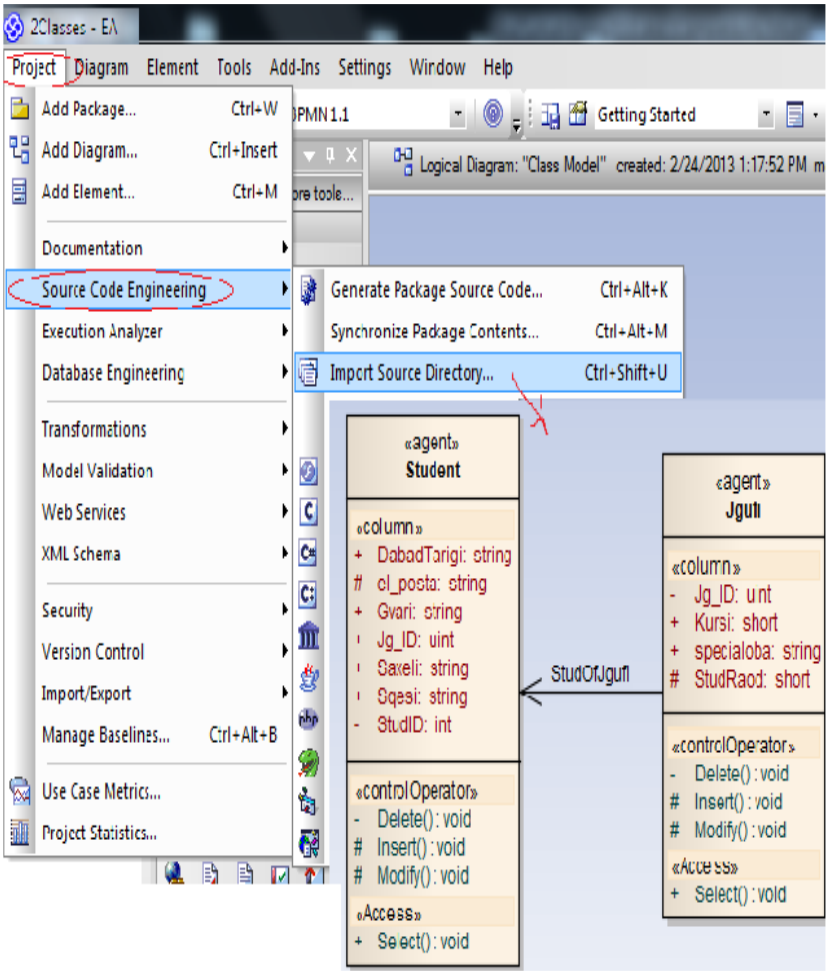
ნახ.4.3. Database Model Diagram –ის აგება

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

განსაკუთრებით მნიშვნელოვანია კლასთა-ასოციაციის დიაგრამის აგება და შემდეგ პროგრამული კოდის გენერაცია (ნახ.4.4, 4.5). სხვა დიაგრამებს აქ აღარ შევეხებით, რადგან ისინი ლიტერატურულ წყაროებში მრავლადაა მოცემული.



ნახ.4.4. კლასი (ა) და კლასთა დიაგრამა (ბ)



ნახ.4.5. Student და Jgupi კლასების მომზადება
„Code Engineering“ პროცესისთვის

4.1.2. კლასთაშორის კავშირების სისტემა

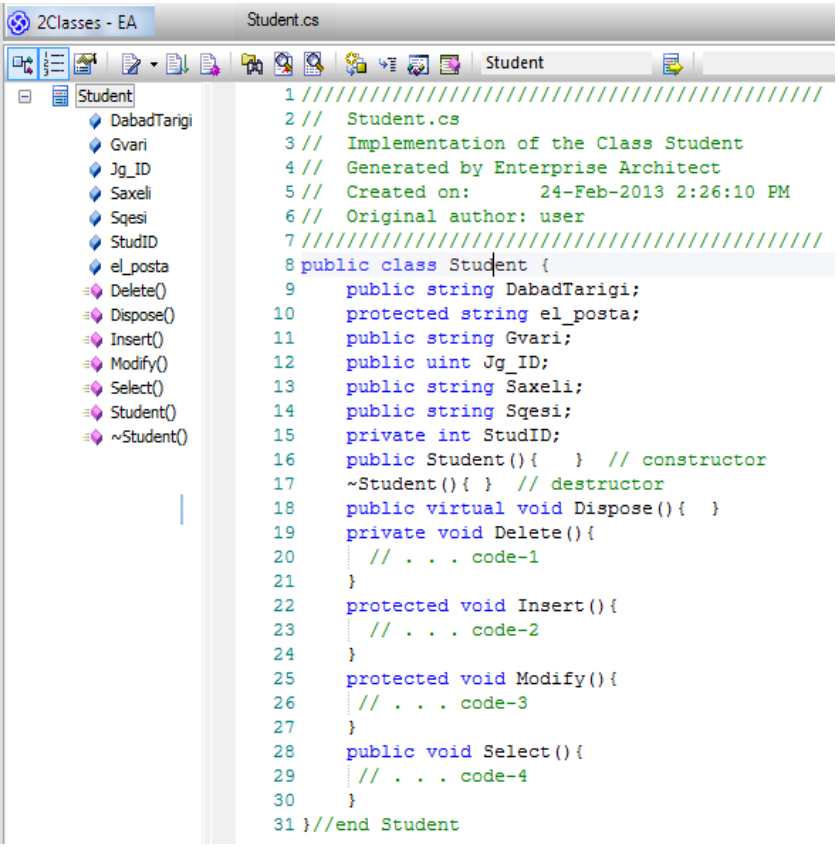
4.4 ნახაზზე ასახული კლასთაშორისი კავშირები შეიძლება ასე დავახასიათოთ:

- მემკვიდრეობითი (Generalization) ასახავს „გენეტი-კურ“, 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.;
- ასოციაციური (Assotiation) ნიშნავს სემენტიკურ კავშირს კლასებს შორის. ის შეიძლება გამოისახოს ერთ- ან ორმიმართულებიანი (იგივეა, რაც უისრო) ხაზით. ისარი გვიჩვენებს შეტყობინების გადაცემის მიმართულებას. ასოციაციური კავშირის რეალიზება ხდება ერთ კლასში დამატებით მეორე კლასის ატრიბურტის ჩასმით. ეს ჰგავს პირველადი (Primary) და მეორეული გასაღებური ატრიბუტების შეერთებას;
- რელაციური (Dependency) ნიშნავს ერთი კლასის დამოკიდებულებას მეორეზე. იგი ერთმიმართულებიანი წყვეტილი ისრით გამოიხატება. მასში დამატებითი დამაკავშირებელი ატრიბუტები არ გამოიყენება;

- *კლასთა დიაგრამიდან კოდის გენერაცია:* თანამედროვე CASE-ტექნოლოგიები, რომლებიც სისტემების დაპროგრამების ავტომატიზაციაზეა ორიენტირებული, მაგალითად, Rational Rose, Visual Paradigm, Enterprise Architect და მრავალი სხვა, ახორციელებს რევერსული დაპროგრამების კონცეფციას [1,42]. ანუ კლასების დიაგრამიდან შესაძლებელია პროგრამული კოდის გენერაცია და პირიქითაც, კოდიდან აიგება ავტომატურად გრაფიკული დიაგრამა.

4.5 ნახაზზე მოცემულია კოდის გენერირების პროცედურა Enterprise Architect ინსტრუმენტის სამუშაო გარემოში, ხოლო 4.6 ნახაზზე კი ილუსტრირებულია გენერირებული კოდის ფრაგმენტი.

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



ნახ.4.6. C#-კოდის ლისტინგი Student კლასისათვის

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

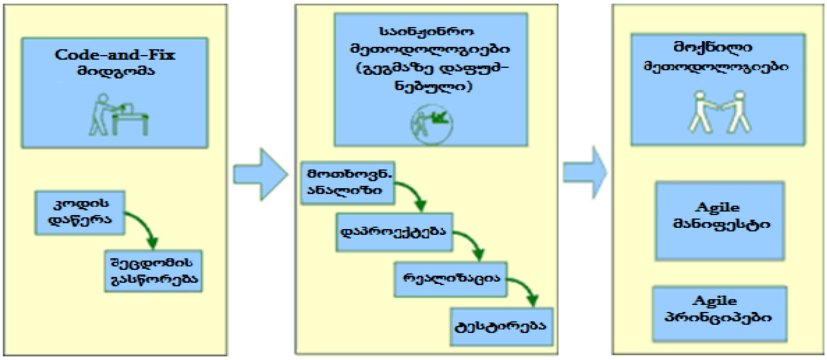
4.2. დაპროგრამების Agile მეთოდოლოგია და აპლიკაციების დეველოპმენტის მეთოდები

დაპროგრამების მოქნილი მეთოდოლოგია (Agile Software Methodology) განიხილავს განსხვავებულ Agile-მეთოდებს, როგორიცაა მაგალითად, Extreme Programming (XP), Scrum, Kanban/Lean, Agile Unified Process (AUP), Dynamic Systems Development Method (DSDM), Disciplined Agile Delivery (DAD), Feature-Driven Development (FDD), Test-Driven Development (TDD), Rapid Application Development (RAD) და Scaled Agile Framework (SAFe) [43, 44].

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

4.2.1. ექსტრემალური დაპროგრამების პრინციპები და ინსტრუმენტული საშუალებანი

2001 წ.ლის თებერვლიდან ოფიციალურად იქნა ჩამოყალიბებული პროგრამული სისტემების დამუშავების ახალი, Agile (მოქნილი) – მეთოდოლოგია [5]. 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.4.7) [1, 37-39].



ნახ.4.7. Agile მეთოდოლოგიებზე გადასვლა

ფორმალიზაციის ხარისხის მიხედვით მოქნილი მეთოდოლოგიები იკავებს შუალედურ ადგილს წინა ორ განხილულ მიდგომას შორის ანუ ისინი არც ისე მკაცრად ფორმალიზებულია, როგორც საინჟინრო მიდგომები და არც ქაოსური, უსისტემო მიდგომაა, როგორც code-and-fix [37].

Agile მეთოდოლოგიაში ჩადებული იდეები არც თუ ახალია. იტერაციული დამუშავება და ადამიანური ფაქტორის განსაკუთრებული როლი და სხვა იდეები, 2001 წლის თებერვლამდეც იყო ცნობილი, მაგრამ ამ დროიდან პირველად მოხდა პროგრესულ ფასეულობათა და პრინციპების ერთად შეკრება და რეკომენდაციების სახით შემოთავაზება, როგორც პრაქტიკულად შემოწმებული და საკმარისი ალტერნატივა პროგრამული სისტემების დამუშავების ტრადიციული მიდგომებისათვის [39].

4.2.2. პროგრამების მოქნილი დეველოპმენტის მანიფესტი და პრინციპები

მანიფესტი ოთხი პუნქტისაგან შედგება, რომელთა-განაც თითოეული ალტერნატივაა [38,39]. 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 მნიშვნელო-ვანია, *ვიდრე* პროცესები და ინსტრუმენტები;
- სამუშაო პროდუქტი უფრო მნიშვნელოვანია, *ვიდრე* ვრცელი-ამომწურავი დოკუმენტაცია;
- დამკვეთთან თანამშრომლობა უფრო მნიშვნელოვანია, *ვიდრე* კონტრაქტით შეთანხმებული პირობები;
- მზადყოფნა ცვლილებებისადმი უფრო მნიშვნელო-ვანია, *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.

4.2.3. მოქნილი (სწრაფი) მოდელირება (Agile Modeling)

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

მოქნილი მეთოდებით პროგრამული სისტემების დამუშავების სპეციალისტები ქმნიან ერთ გუნდს დამკვეთთან ერთად, რომლის წარმომადგენლებიც უშუალოდ და აქტიურად მონაწილეობენ სისტემის ანალიზის, დაპროექ-

ტებისა 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* ! [39].

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

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

4.2.4. მოქნილი მოდელირების ფასეულობანი

AM-ის ფასეულობებია [45,46]: ურთიერთობა (კომუნიკაცია), სიმარტივე, უკუკავშირი, გამბედაობა და თავმდაბლობა. აქედან პირველი ოთხი „ექსტრემალური დაპროგრამების“ მამას, კენტ ბეკსაც ჰქონდა მოცემული (2000 წ. XP) [39]. ს. ამბლერმა მეხუთე დაამატა. განვიხილოთ თითოეული მათგანი მოქნილი მოდელირების ჭრილში.

- **კომუნიკაცია ურთიერთობა**

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

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

პროექტის პრობლემები ჩნდება იქ, სადაც ურთიერთობა შეწყვეტილია. მაგალითად, როდესაც დეველოპერი არ ეუბნება კოლეგებს, რომ მისი კოდი არ მუშაობს და საჭიროა დახმარება. ან დამკვეთი ვერ ამახვილებს ყურადღებას პროექტის მნიშვნელოვან მახასიათებლებზე და დეველოპერები 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 ასაგებად ერთ-ერთი მარტივი და შესაძლებელი ხერხია (პროტოტიპების თვალსაზრისით), მაგრამ დასმული ამოცანისთვის ის უნდა იყოს შესაბამისად მისაღები, არ უნდა იყოს უფრო რთული, ვიდრე ამას ამოცანა მოითხოვს;
- *სისტემის არქიტექტურის გართულება მომავალი შესაძლო გაფართოებების მხარდასაჭერად.* ხშირად ტრადიციული პროგრამული ტექნოლოგიების მიმდევრები, რომელთაც არ სურთ მომავალში ცვლილებების განხორციელება, ცდილობენ წინასწარ გაითვალისწინონ და დააპროექტონ სისტემის ჭარბი, გაფართოებული ვარიანტი (თუმცა, შეიძლება ასეთი მოთხოვნილება მომავალში ნაკლებალბათური იყოს, ან საერთოდ არ იყოს საჭირო).

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

- *რთული ინფრასტრუქტურის შემუშავება.* ეს ფართოდ გავრცელებული შეცდომაა, რომელსაც გუნდი უშვებს. კერძოდ, გუნდის საწყისი ძალისხმევა მიმართულია პროექტის ინფრასტრუქტურის შექმნაზე (მაგალითად, 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, რომ მიიღოს მიმწოდებლიდან კონკრეტული პროდუქტი, რომელიც მას დაეხმარება სამუშაოს შესრულებაში (და არა ცარიელი მონაცემთა ბაზების სისტემა ან შეცდომების დამუშავების ქვესისტემა).

ვინაიდან ვერ ხერხდება საჭირო ფუნქციობის პროგრამების სწრაფი დამუშავება, ეს წარმოშობს სათანადო რისკს პროექტის შესასრულებლად. საუკეთესო მიდგომაა, შემცირდეს ინფრასტრუქტურის დამუშავების მასშტაბები პროექტის შესრულებისას მანამ, სანამ ის არ გახდება

აუცილებელი. მაგალითად, შეცდომების გასწორების ქვესისტემა შეიძლება დამუშავდეს მოგვიანებით, როცა ის აუცილებელი გახდება.

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

- **უკუკავშირი**

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

- *მოდელი მუშავდება გუნდში.* აქ გამომახილი 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 სპეციალისექსპერტთან, რომელიც შემდეგ მოგვცემს გამოძახილს;
- *მოდელის რეალიზაცია.* ესაა ყველაზე საიმედო ხერხი გამოძახილის მისაღებად ანუ, მოდელი მუშავდება პროგრამის სახით და მიეწოდება სამუშაოდ დამკვეთმომხმარებელს;
- *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 რეალი-

ზაციის დროს პროგრამის საშუალებით შედეგები მიიღება სწრაფად, რამდენიმე წუთში, საათში ან დღეში. მიღება-ჩაბარების ტესტირება მოითხოვს რამდენიმე კვირას ან თვეს.

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

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

- **გამბედაობა**

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

უპირველეს ყოვლისა, გამბედაობაა საჭირო, რათა მიღებულ იქნას გადაწყვეტილება მოქნილი მიდგომის გამოყენების შესახებ. შემდეგ კი მისი გამოყენების გაგრძელების შესახებ, თუ საქმე ვერ წავიდა წარმატებით (რაც ხშირად მოსალოდნელია). ორგანიზაციაში მოიძებნება

სხვა შეხედულების ადამიანები, რომელთა წინააღმდეგობა დასაძლევია. ეს პოლიტიკაა.

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

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

- **თავმდაბლობა**

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

თავიანთი სფეროს განვითარების ტენდენციები, ხოლო თანამშრომლებს კარგად ესმით, რისი გაკეთების უფლება აქვთ და რისი არა საკუთარ პროდუქციასთან.

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

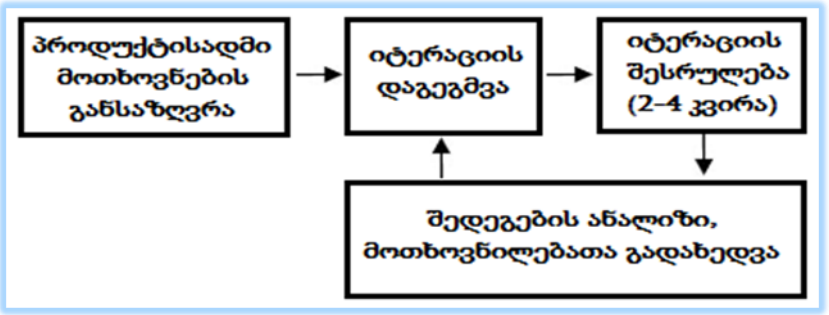
ამგვარად, პროგრამული სისტემის მენეჯერი, კონკრეტული პროექტის ამოცანებისა და მოთხოვნების შესაბამისად, უნდა განსაზღვრავდეს როგორც პროგრამირების მეთოდის, ეტაპთა ფაზებისა და იტერაციათა მოთხოვნების შერჩევა-ფორმირებას, ასევე მუშა გუნდის შემადგენლობას.

4.2.5. Scrum - მოქნილი მეთოდის ფრეიმვორკი

პროგრამული ინდუსტრიის სფეროში Agile მეთოდოლოგიის მნიშვნელოვანი წარმომადგენელია Scrum მეთოდი [47,48]. 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.4.8).



ნახ.4.8. 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 ადმინისტრაციული 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:

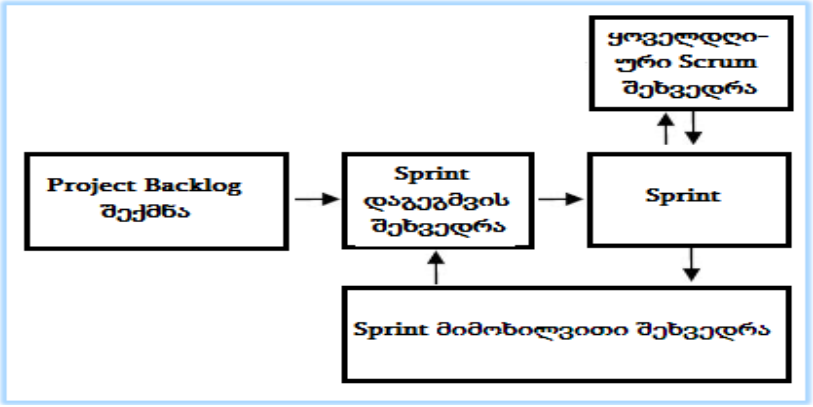
- *Sprint Planning Meeting.* შეხვედრა Sprint-ის დასაგეგმად (ესაა მოკლე დისტანცია, ანუ იტერაცია). თავიდან პროდუქტის მფლობელი, Scrum-ოსტატი, გუნდი, ასევე დამკვეთის წარმომადგენელი და სხვა დაინტერესებული პირები განსაზღვრავენ, თუ რომელი მოთხოვნებია Project Backlog-დან უფრო პრიორიტეტული და რომელი უნდა განხორციელდეს 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.4.9).

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



ნახ.4.9. Scrum-მეთოდი Sprint-ბიჯებით

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

4.2.6. Kanban – ეკონომიური მოქნილი მეთოდი

Kanban – არის პროგრამული უზრუნველყოფის დამუშავების ეკონომიური მეთოდი (Lean method of software development) [40].

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 ღირებუ-

ლების შენარჩუნებაზე. ეკონომიურ (Lean) მიდგომებს ხშირად უწოდებენ Six-Sigma ან Just-In-Time (JIT) [40,49]. აღნიშნული კონცეფცია შემუშავდა Motorola კორპორაციაში 1986 წ. და გამოყენებულ იქნა პირველად General Electric-ში.

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

კანბანსაც და სკრამსაც აქვს საკუთარი ფესვები ეკონომიურ წარმოებაში (Lean production), რომლის მიზანიც სამი სახის ნარჩენების მოცილებაა (Lean-პრინციპები იაპონური ავტომრეწველობის სფეროდან) [50]:

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

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

Kanban-ის გამოყენების ფუძემდებლად ინფორმაციული ტექნოლოგიების სფეროში ითვლება *დევიდ ანდერსონი*, რომელმაც 2007 წელს პირველმა წარმოადგინა ამ მეთოდის ზოგადი კონცეფცია [51]. მან ჩამოაყალიბა *4 საბაზო პრინციპი* და *6 ძირითადი პრაქტიკა*, რომლებსაც თავიანთ საქმიანობაში ინტეგრირებულად იყენებს კომპანიები Kanban-ის საფუძველზე.

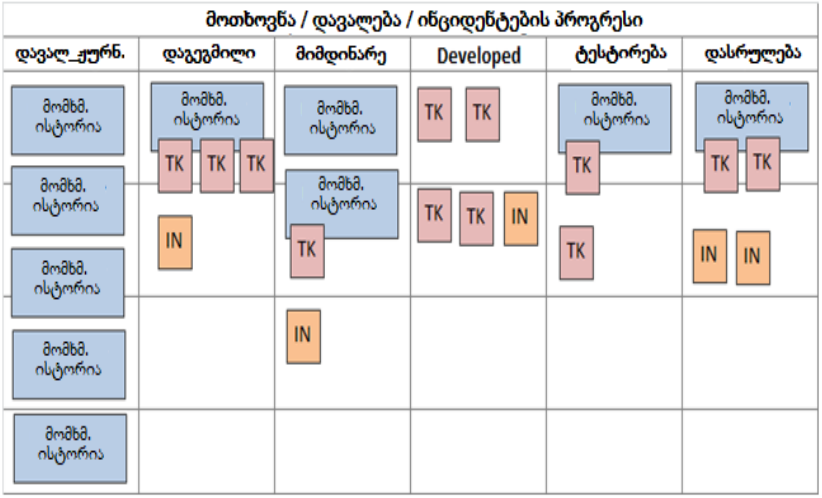
- **Kanban-ის პრინციპები:**

- *პ1. დაიწყეთ იმით, რასაც ამჯერად აკეთებთ*: რომელი აქტუალური სამუშაოც სრულდება, ჯერ ის უნდა დასრულდეს და მხოლოდ ამის შემდეგ დაიწყოს ახალი სამუშაო;
- *პ2. დათანხმდით, რომ ევოლუციური ცვლილებები მოსალოდნელია*: შემდგომ განვითარებას აქვს განსაკუთრებული მნიშვნელობა, ოღონდაც აქ სრულყოფა მიღწეულ უნდა იქნას ძირითადად მცირე / ევოლუციური ბიჯების ხარჯზე;
- *პ3. პატივი ეცით პირველარსებულ პროცესებს / როლებს / ვალდებულებებს*: Kanban ადვილად რეალიზებადია, ყველა როლი, პროცესი და ა.შ., რჩება;
- *პ4. 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 [52].

- **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.4.10).

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



ნახ.4.10. Kanban-ის დაფა (იყენებს Software Development Life Cycle-ს)

ისინი დროის და შესრულებული ბიჯის შესაბამისად გადაადგილდება დაფაზე მარცხნიდან მარჯვნივ;

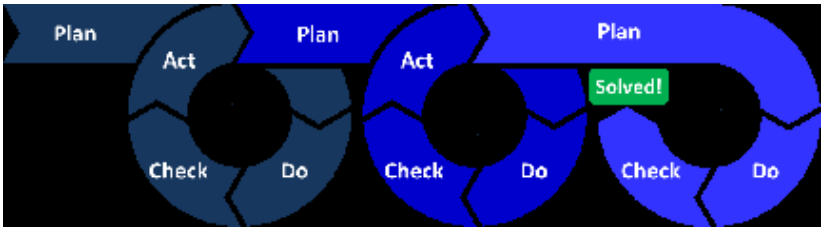
- *ძპ2. დაწყებული სამუშაოს მოცულობის შეზღუდვა*: ბილეთების რაოდენობა (Work in Progress - WiP), რომლებიც შეიძლება დამუშავებულ იქნას ერთდროულად ერთ კვანძში, შეზღუდულია. მაგალითად, თუ კვანძი „პროგრამირება“ ამუშავებს ორ ბილეთს და ამ კვანძის ლიმიტი 2-ია, მაშინ მას არ შეუძლია მე-3 ბილეთის მიღება, თუნდაც მოთხოვნის 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.4.11) – არის სრულყოფის პროცესი, რომელიც ეფუძნება ცოდნის თეორიის გაგებას და გამოიყენება ხარისხის მენეჯმენტში) ან ვიწრო ადგილების თეორიაზე, სისტემურ აზროვნებაზე ან კომპლექსურობის (სირთულის) თეორიაზე [53,54].

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



ნახ.4.11. PDCA ციკლის იტერაციული განმეორება პრობლემის მოგვარებამდე

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

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

მაგალითად, შეზღუდვები შეიძლება შეიცვალოს ცალკეული კვანძებისათვის, შეიძლება შემოტანილ იქნას ბუფერები (განსაკუთრებით ვიწრო ადგილების გაჩენამდე, რაც გამოწვეულია დროებით რესურსების წვდომის გამო), კვანძებზე მომუშავეთა რაოდენობა შეიძლება შეიცვალოს, აღმოიფხვრას ტექნიკური პრობლემები და ა.შ. სრულყოფის ასეთი უწყვეტი პროცესი არის Kanban-ის განუყოფელი ნაწილი [51,55].

4.2.7. Scrum და Kanban მეთოდების შედარება

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

მეორე როლი, სკრამ-ოსტატი ეხმარება პირველს ბიზნეს-პროცესების ორგანიზებაში. კერძოდ, ატარებს საერთო შეკრებებს, აგვარებს საყოფაცხოვრებო პრობლემებს, ხელს უწყობს გუნდის მოტივაციას და აკვირდება სკრამ-მიდგომის შესრულების დაცვას. Scrum-მიდგომა ყოფს სამუშაო პროცესს თანაბარ სპრინტებად - ესაა პერიოდი ერთი კვირიდან თვემდე, რაც დამოკიდებულია პროექტსა და გუნდზე. პროცესები იტერაციულია (ნახ.4.12).

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

ნახ.4.12

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

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

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

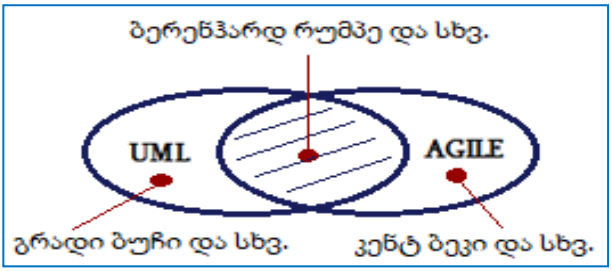
Agile მიდგომები პროცესების ვიზუალიზაციისათვის იყენებს ფიზიკურ და ელექტრონულ დაფებს. ისინი უზრუნველყოფს სამუშაო პროცესები გახადოს გამჭვირვალე და გასაგები ყველა სპეციალისტისთვის, რც მნიშვნელოვანია, რადგან გუნდს არ ჰყავს ერთი ფორმალური ხელმძღვანელი.

4.3. UML და Agile მეთოდოლოგიების გამოყენების კომპრომისული გადაწყვეტა

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

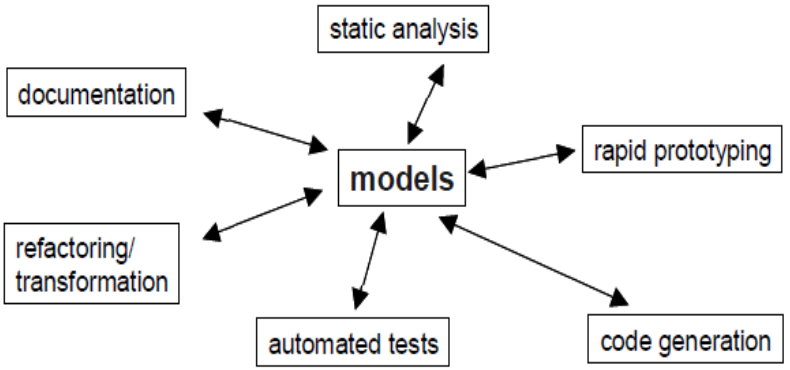
მეცნიერები და პრაქტიკოსი დეველოპერები ცდილობენ დაასაბუთონ თავიანთ მოსაზრებათა ჭეშმარიტება და იდეალურად წარმოადგინონ ესა თუ ის კონცეფცია. მაგალითად, უნიფიცირებული მოდელირების ენა (UML - გ. ბუჩი, ი. ჯაკობსონი, ჯ. რამბო და სხვ.) თუ მოქნილი (Agile) დაპროგრამება, ექსტრემალური პროგრამირების მაგალითზე (კ. ბეკი, დ. მარტინი და სხვ.) [13,32].

ლიტერატურულ წყაროებში მოიძებნება ისეთ მეცნიერთა ნაშრომებიც პროგრამული ინჟინერიის სფეროში, რომლებიც კომპრომისული მიდგომით ჰიბრიდულ ვარიანტსაც გვთავაზობენ (მაგალითად, კანადელი ს. ამბლერი, გერმანელი ბ. რუმპე და სხვ.) [46,47,57] (ნახ.4.13).



ნახ.4.13. UML და Agile მეთოდოლოგიების ერთობლივი გამოყენების კონცეფცია

მიუნხენის ტექნიკური უნივერსიტეტის პროფესორმა ბ. რუმპემ თავის ფუნდამენტურ კვლევებში ნათლად ჩამოაყალიბა UML მოდელების (დიაგრამების) გამოყენების პოტენციალური შესაძლებლობები პროგრამული აპლიკაციების შექმნის სასიცოცხლო ციკლის სხვადასხვა ეტაპებისა და ამოცანებისათვის (ნახ.4.14) [47].



ნახ.4.14. UML მოდელების გამოყენების პოტენციალი

განსაკუთრებული ყურადღება ამ ნაშრომში გამახვილებული იყო Agile მეთოდოლოგიის ერთ-ერთი ყველაზე პოპულარულ, ექსტრემალური პროგრამირების მეთოდის საფუძველზე პროგრამული სისტემის კოდირების ეფექტურობის ამაღლების მიზნით – სწორედ უნიფიცირებული მოდელების გამოყენებაზე.

ექსტრემალური პროგრამირება (XP) ხასიათდება შემდეგი მახასიათებლებით:

- *იგი ორიენტირებულია ძირითად მიზანზე* – ეფექტურ პროგრამულ კოდზე. დოკუმენტაცია იგნორირებულია,

მაგრამ გამოყენებულია კოდირების სტანდარტები მისი აღწერის მიზნით;

- *ავტომატიზებული ტესტები გამოიყენება ყველა დონეზე.* პრაქტიკული გამოცდილება უჩვენებს, რომ თუ კოდი სწორადაა აგებული, მაშინ წუნის (შეცდომების) პროცენტი საგრძნობლად დაბალია. ამ პროცესის ავტომატიზაცია კი შესაძლებელს ხდის ტესტირების ექსპერიმენტი ჩატარდეს მრავალჯერადად;
- *გამოიყენება ძალზე მცირე იტერაციები უწყვეტი ინტეგრაციით, ხოლო სისტემა მაქსიმალურად მარტივია;*
- *რეფაქტორინგი გამოიყენება კოდის სტრუქტურის გასაუმჯობესებლად,* ხოლო ტესტები უზრუნველყოფს დეფექტების გამოვლენის დაბალ სიხშირეს რეფაქტორინგის შედეგად.

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

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

არქიტექტურული ნაკლოვანებები აღმოიფხვრება *რეფაქტორინგის მეთოდების* გამოყენებით [5,58]. ესაა ტრანსფორმაციული მეთოდები, რომლებითაც შესაძლებელია სისტემის მცირე გარდაქმნების ჩატარება მისი სტრუქტურის სრულყოფის მიზნით.

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

ამ თვალსაზრისით, მოდელებით მართვადი არქიტექტურების გამოყენება, შეიძლება მივიჩნიოთ ძალზე პერსპექტიულ მიმართულებად. მაგალითად, ASP.NET MVC ტექნოლოგა და სხვ. [5, 59].

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

- მოდელი,
- წარმოდგენა და
- კონტროლერი.

მოდელი არის მონაცემები და რეაგირებს კონტროლერის ბრძანებებზე, იცვლის თავის მდგომარეობას;

წარმოდგენა უზრუნველყოფს მოდელის მონაცემთა ასახვას მომხმარებლისთვის, რომელსაც შეუძლია მოდელის შეცვლა;

კონტროლერი ასრულებს (ინტერპრეტაციას უკეთებს) მომხმარებლის ქმედებებს, აცნობებს მოდელს ცვლილებების შესახებ.

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

ჩვენი წიგნი სცილდება ამ თემატიკის უფრო დეტალურ განხილვას. იმედია, რომ დაინტერესებული მკითხველი საკმაოდ იპოვნის ამ მიმართულებით ინტერნეტულ ინფორმაციას და გაიღრმავებს ცოდნას ამ აქტუალურ სფეროში.

V თავი
კომპიუტერული პროგრამირების ენები
(საილუსტრაციო ფრაგმენტები ანალიზისათვის)

კომპიუტერული პროგრამირება არის ალგორითმის აგების პროცესი და მისი კოდირება პროგრამირების რომელიმე ენის ნოტაციაზე, რათა მისი შესრულება შესაძლებელი იყოს კომპიუტერის მიერ. ალგორითმები აღწერს *პრობლემის გადაწყვეტას* შესაბამის მონაცემთა ტერმინებში და *აუცილებელი ბიჯების ერთობლიობას* მიზნობრივი შედეგების მისაღებად.

განიხილავენ პრობლემის მოგვარების პროცესის ოთხ ძირითად ეტაპს:

1. პრობლემის განსაზღვრა, გადასაწყვეტი ამოცანის ჩამოყალიბება;
2. ალტერნატიულ გადაწყვეტილებათა შემუშავება;
3. ალტერნატივათა შეფასება და შერჩევა მიზნობრივი სტანდარტების შესაბამისად;
4. გადაწყვეტილების დანერგვა (განხორციელება) და შედეგების კონტროლი.

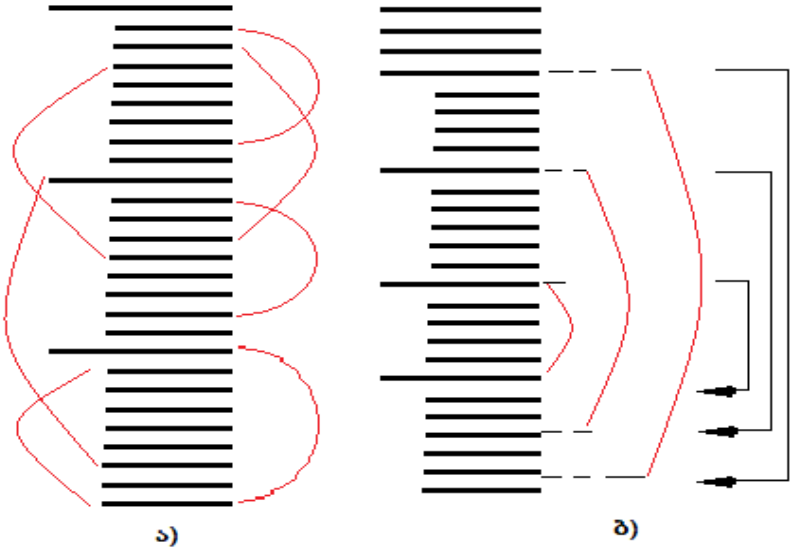
როგორც წიგნის პირველ თვაში აღინიშნა, *სტრუქტურული პროგრამირება* ისეთი პარადიგმაა, რომელიც კოდს ყოფს მოდულებად (ან ფუნქციებად) და მისი წაკითხვა, აღქმა დეველოპერისთვის ადვილია.

არასტრუქტურული პროგრამირება კი ისეთი პარადიგმაა, რომელიც კოდს განიხილავს როგორც ერთიან ბლოკს. მისი წაკითხვა და გაგება შედარებით რთულია.

არასტრუქტურული მიდგომით შექმნილი პროგრამები ლიტერატურაში მოიხსენიება „სპაგეტი“ კოდის სახელით. მათი ძირითადი ნიშანია პროგრამის ტექსტში GOTO-ოპერატორების გამოყენება (ნახ.5.1-ა) [60].

სპაგეტი-კოდების არსებობა შეიძლება გამოწვეული იყოს რამდენიმე მიზეზით. კერძოდ, პროექტის მოთხოვნების ცვლილებით, პროგრამირების სტილის წესების არარსებობით, დეველოპერის გამოცდილების სტაჟის ნაკლებობით დ ა.შ.

5.1-ბ ნახაზზე ნაჩვენებია სტრუქტურული პროგრამის ზოგადი სქემა (GOTO-ს გარეშე), აქ გამოყენებულია ჩადგმული ციკლების კონცეფცია.



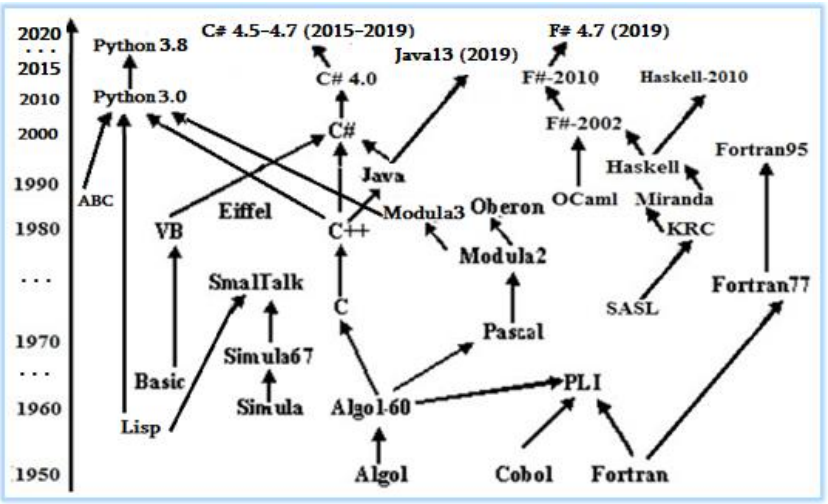
ნახ.5.1. პროგრამების ზოგადი სქემები:
არასტრუქტურული (ა) GOTO-თი და სტრუქტურული (ბ)
ჩადგმული ციკლებით

წიგნის 1-4 თავებში ჩვენ განვიხილეთ კომპიუტერული პროგრამირების მეთოდებისა და მეთოდოლოგიების საბაზო საკითხები და ძირითადი პარადიგმები. გავეცანით პროგრამირების პროცესს, რომელიც მოიცავს მეთოდის, სტილის, მოდელის, ალგორითმისა და ენის საკითხებს.

ბოლო თავში განვიხილავთ სწორედ *პროგრამირების ენის* კონცეფციას, რომელიც, თავის მხრივ მოიცავს სინტაქსურ, სემანტიკურ და პრაგმატულ მოსაზრებებს.

ჩვენი მიზანი აქ კონკრეტული ენის შესწავლა კი არა, განსხვავებული პარადიგმების მქონე ენებს შორის მსგავსება-განსხვავების ილუსტარციაა.

პროგრამირების *თანამედროვე ენების* განვითარება დაახლოებით 1950 წლიდან იწყება და უწყვეტად ვითარდება (ნახ.5.2).



ნახ.5.2. პროგრამირების ენების განვითარების ისტორიული ფრაგმენტი

შესაბამისი ლიტარატურული და ინტერნეტული წყაროების ანალიზი გვიჩვენებს პროგრამირების ენების განვითარების მდგომარეობას, მათ დინამიკას და პერსპექტივებს [61,62]. 2019 წლისთვის განსაკუთრებული პოპულარობა მოიპოვა Python და JavaScript ენებმა (ან ენის ოჯახებმა), „ავტორიტეტი“ შეინარჩუნეს C, C++, C# და Java ენებმა, როგორც ფუნდამენტურმა მოჰიკანებმა. პრაქტიკული გამოყენების თვალსაზრისით ასევე წინ წამოიწია შედარებით ახალმა ენებმა, როგორიცაა R, Ruby, Swift, Go და სხვ. [61].

ენის შერჩევის დროს აუცილებელია განისაზღვროს თუ რომელ სფეროში და რა მიზნით ვაპირებთ მუშაობას.

მაგალითად, თუ ვებ-აპლიკაციებთან გვსურს მუშაობა, მაშინ დღეისთვის JavaScript შეიძლება იდეალურად ჩაითვალოს (მის საფუძველზე განვითარებულ სხვა ენებთან ერთად, როგორებიცაა Angular, ReactJS და სხვა Frontend, Backend ენები) [9]. თუ გვჭირდება სისტემურ დონეზე ან დესკტოპ-აპლიკაციებთან მუშაობა, მაშინ გამოგვადგება C/C++/C# ენების ოჯახი. მობილურ აპლიკაციებთან სამუშაოდ (Android, iOS) გამოდგება Java და Swift ენები. განსაკუთრებით აქტუალურია ისინი ინტერნეტული საგნების მიმართულებით (Internet of things – IOT). და ბოლოს, მულტიპარადიგმული Python ენა შეუდარებელია თავისი სიმარტივითა და სისწრაფით ქსელურ აპლიკაციებსა და მონაცემთა მეცნიერების (Data Science) სისტემებში გამოსაყენებლად (BigData, Machine Learning და ა.შ.).

ახლა განვიხილოთ ზოგიერთი პოპულარული ენის დამახასიათებელი სინტაქსური კონსტრუქციები.

5.1. სტრუქტურული პროგრამირების C ენა

ამერიკელი მეცნიერ-პროგრამისტების კერნიგანისა და რიჩტის მიერ 1972 წელს შექმნილი ახალი C-ენა გახდა UNIX ქსელური ოპერაციული სისტემის „დედა“ და შემდგომ ობიექტ-ორიენტირებული და ჰიბრიდული ენების C++, Java, C# და სხვ. – „ბებია“ [62-64]. C-ენის ოჯახმა მნიშვნელოვანი წვლილი შეიტანა 50 წლის მანძილზე შექმნილი პროგრამული აპლიკაციებით თითქმის ყველა სფეროს ყველა დარგში.

C ენა სტრუქტურული დაპროგრამების კონცეფციის რეალიზაციის ძლიერი წარმომადგენელია. მართალია, იგი ფლობს თავის არსენალში უპირობო გადასასვლელის - goto ოპერატორს, მაგრამ მისი „ჭკვიანური“ გამოყენება უკვე თვით პროგრამისტ-დეველოპერის საქმეა. ბატონი დეიკსტრა კატეგორიულად მოითხოვს მისი ხმარებიდან ამოღებას და სამაგიეროდ, პროგრამული რესურსებიდან განშტოებების, ციკლების და გადამრთველების გამოყენებას გვთავაზობს.

განვიხილოთ მის მოსაზრებაზე აგებული ჩვენი პირველი C-კოდები, რომელთა სტრუქტურირებული ალგორითმებიც ამ ელემენტებითაა აგებული [2].

შენიშვნის სახით აღვნიშნავთ, რომ ამჯერად არ აქვს მნიშვნელობა თუ პროგრამირების რომელ პლატფორმაზე (კომპილატორზე) შევასრულებთ კოდის აგებას და ამუშავებას (იქნება ეს ბორლანდის, მაიკროსოფთის Visual Studio.NET თუ სხვ. - კონსოლის რეჟიმში).

5.1-5.10 ლისტინგებში ასახულია ჩვენი კონკრეტული ამოცანების გადაწყვეტის სტრუქტურული ალგორითმების პროგრამული კოდები [65].

- **პირობის შემოწმების ოპერაცია (if..else..):**

```
/* ლისტინგი 5,1 ---------- */
#include <stdio.h>
main()
{
  int a, b, y;
  printf("Input a = "); scanf("%ld", &sec);
  printf("Input b = "); scanf("%ld", &sec);
  if (a > b)
       y=a*a + b*b;
     else
       y=500;
   /* ან იგივე:
       y = (a > b)? a*a +b*b : 500; */
  printf(„%d a= %d, %d, %d \n", a, b, y);
  getch();
}
```

- **ციკლური ოპერაცია (for, while, do…while):**

```
/* ლისტინგი 5,2 ---------- */
#include <stdio.h>
#define Num 5
main()
{
   int i=1, j, parol;
   /* — while —— */
   while(i <= Num)
     {
       printf("ისწავლეთ  C და  C++ ! \n");
       i++;
     }
  /* — for — */
```

```
    for(i=1; i <= Num; i++) /* ციკლი ზრდადი ინდექსით*/
        printf("ვისწავლი  ! \n");
    for(j=Num; j > 0; j—)   /* ციკლი  კლებადი ინდექსით*/
        printf("%d seconds ! \n", j);
        printf("Bu-u-u-x ! \n");
   /* — do while —*/
    do {
          printf("შეიტანე paroli: ");
         scanf("%d",&parol); /* სრულდება 1-ელ  მაინც */
        }
       while(parol != 94) /* თუ არ უდრის 94 */
                    ;                    /*ცარიელი ოპერატორი  !!! */
   printf("Ok ! \n");
   getch();
}
```

- **დამოკიდებულებები და ლოგიკური ოპერაციები:**

ამოცანა: „კლავიატურიდან შეტანილ ტექსტში პროგრამამ დათვალოს სიმბოლოების, სტრიქონებისა და სიტყვების რაოდენობა“

```
/* ლისტინგი 5,3 ---------------- */
#include <stdio.h>
#define YES 1 /* ვართ სიტყვაში */
#define  NO  0 /* არ ვართ სიტყვაში  */
main()
{
  int a;
  int nL=0; /* სტრიქონები */
  int nW=0; /* სიტყვები */
  int nC=0;  /* სიმბოლოები */
  int word =NO; /* მდგომარეობა:  "არ ვართ სიტყვაში“  */
/* getchar() - ტექსტის შეტანა მეხსიერების ბუფერში  */
```

```
  while((a=getchar()) != EOF) /* არაა  End Of File */
    {
      nC++;
      if(a == '\n') /* თუ ახალი სტრიქონია,  მაშინ  */
          nL++ ;
      if(a != ' ' && a != '\n' && word == NO) /* &&-ლოგიკური 'და' */
        {
          word =YES; /*  მდგომარეობა: შესვლა ახალ სიტყვაში  */
          nW++;
        }
      if((a == ' ' || a == '\n') && word == YES) /* ‖ლოგიკური  'ან'' */
        word=NO; /* მდგომარეობა:  "გამოსვლა სიტყვიდან"  */
    }
  printf(„Simb=%d, Striq=%d, Sitkv=%d \n", nC, nL, nW);
  getch();
}
```

- **რეკურსიული ოპერაცია:**

```
/* ლისტინგი 5.4 ---recursion --- */
#include <stdio.h>
main()
{
  int N;
  double fac;
  clrscr();
  printf("ფაქტორიალისთვის N=");
  scanf("%d", &N);
  if(N > 0)
     fac=Factorial(N);  /* ფუნქცია-ქვეპროგრამის გამოძახება  */

  printf("%d ! = %.0f \n", N, fac);
}
```

```
/* — ფაქტორიალის შიგა ფუნქცია-ქვეპროგრამა — */
double Factorial(int n)
{
  return(n == 1)? 1 : n*Factorial(n-1); /*თავის თავის გამოძახება!*/
}
```

- **გადამრთველი ოპერატორი (switch):**

ამოცანა: „ეკრანზე გამოიტანის იმ ასოზე დაწყებული სახელები, რომელ ასოსაც ვირჩევთ კლავიატურაზე“.

```
/* ლისტინგი 5,5 ---------- */
#include <stdio.h>
main()
{
 int x; clrscr();
 printf("შემოიტანე სიმბოლო: a, n, N, d, p \n");
 while((x=getch()) != 27)
 switch(x)
 {
  case 'a':
        printf("Ann, Akaki, Arsena \n");
        break;
  case 'n':
  case 'N':
        printf("Nana, Nino, Nona, Niko \n");
        break;
  case 'd':
  case 'p':
        printf("Dato, Dito, Pako, Pluto \n");
        break;
  default :
       printf("კიდევ სცადე ! \n");
```

```
        break;
  }
}
```

ამოცანა: „ეკრანის ცენტრში გამოვიტანოთ სიტყვა “INTERNET” და მმართველი კლავიშებით (up, down, left, right) ვამოძრაოთ იგი ეკრანზე ვერტიკალურად და ჰორიზონტალურად“.

```
/* ლისტინგი 5,6 ---------------  */
#include<io.h>
#include <conio.h>
#define word “INTERNET”
main()
{
  int x; int i=40, j-12; /* ეკრანის ცენტრის კოორდინატები */
  clrscr();
  gotoxy(i,j): /* კურსორი დგება ცენტრში */
  printf(“%s”, word);
  while((x=getch()) != 27)
   {
    if(x==0)
      x=getch();

   switch(x)
    {
      case 72:                                    /* Up */
               j—; clrscr();
              gotoxy(i,j); printf(“%s”, word);
              break;
      case 75:                                    /* Left */
               i—; clrscr();
               gotoxy(i,j); printf(“%s”, word);
```

```
                break;
        case 77:                                    /* Right */
                i++; clrscr();
                gotoxy(i,j); printf("%s", word);
                break;
        case 80:                                    /* Down */
                 j++; clrscr();
                 gotoxy(i,j); printf("%s", word);
                 break;
          default :
                 printf("დააჭირე მმართველ ღილაკს  ! \n");
                 break;
       } /* end switch() */
    } /* end while() */
} /* end main() */
```

- **ფანჯრების აგება ინტერფეისისთვის (window):**

ამოცანა_1: „ეკრანზე ოთხი ფანჯრის აგება ფერებით“.

```
/* ლისტინგი 5,7 ----კლასიკური --------- */
#include <stdio.h>
#include <conio.h>
#define S1 " Lector "
#define S2 " Student "
#define S3 " Department "
#define S4 " University "
main()
{
  textbackground(0); /* 0 - black */
  textcolor(14); /* 14 - yellow */
```

```
  clrscr();
  gotoxy(15,1); cprintf("%s \n", S1);
  gotoxy(55,1); cprintf("%s \n", S2);
  gotoxy(15,19); cprintf("%s \n", S3);
  gotoxy(55,19); cprintf("%s \n", S4);

  textbackground(1); /* 1 - blue */
  textcolor(15); /* 15 - white */
  window(2,2, 39,18);
  clrscr();
    cprintf(" Shonia ");

  textbackground(2); /* 2 - green */
  textcolor(15);
  window(42,2, 79,18);
  clrscr();
    cprintf(" Ramishvili \n");

  textbackground(3); /* 3 - cyan */
  textcolor(15);
  window(2,20, 39,25);
  clrscr();
    cprintf(" Computer Science \n");

  textbackground(4); /* 4 - red */
  textcolor(15);
  window(42,20, 79,25);
  clrscr();
      cprintf(" Informatics \n");
getch();
}
```

ამოცანა_2: „ეკრანზე ოთხი ფანჯრის აგება ფერებით კასკადური სტილით“.

```
/* ლისტინგი 5,8 ----კასკადური---------- */
#include <stdio.h>
#include <conio.h>
main()
{
  textbackground(0); /* 0 - black */
  textcolor(14); /* 14 - yellow */
  clrscr();
        textbackground(1); /* 5 - magenta */
        textcolor(15); /* 15 - white */
        window(2,2, 22,8);
        clrscr();
        cprintf(“ Shonia “);

            textbackground(2); /* 2 - green */
            textcolor(15);
           window(12,6, 32,12);
           clrscr();
           cprintf(“ Ramishvili \n”);

               textbackground(3); /* 4 - red */
               textcolor(15);
               window(22,10, 42,16);
               clrscr();
               cprintf(“ Computer Science \n”);

                   textbackground(4); /* 1 - blue */
                   textcolor(15);
                   window(32,14, 52,20);
```

```
                clrscr();
                cprintf(“ Informatics \n”);

                   textbackground(5); /* 3 - cyan */
                  textcolor(15);
                 window(42,18, 62,24);
                  clrscr();
                  cprintf(“ OK ! my love \n”);
        getch();
}
```

ამოცანა_3: „ფანჯრის აგება ჭადრაკის დაფის სტილში ჩადგმული ციკლების გამოყენებით“.

```
/* ლისტინგი 5,9 ----ჭადრაკის დაფა ---------- */
#include <conio.h>
#define W textbackground(15)
#define B textbackground(1)
main()
{
  int i, j, k;
  textbackground(0);
  window(1,1,80,25); clrscr();
  k=1;
  for(i=20; i<=55; i+=5)
   {
     k++;
     for(j=5; j<=19; j+=2)
       {
          if(k%2==0)
            {
              W;
```

```
            window(i,j, i+4, j+1);
            clrscr();
          }
        else
          {
            B;
            window(i,j, i+4, j+1);
            clrscr();
          }
      k++;
    }
  }
  getch();
}
```

- **ეკრანზე შედეგების გამოტანის ორგანიზება:**

ამოცანა: „ეკრანზე გამოვიტანოთ ყველა ლუწი რიცხვი [1-:-N] დიაპაზონში, დალაგებული M სვეტებში“.

```
/* ლისტინგი 5,10 ---[M,N] ცხრილი ლუწი რიცხვებით -- */
#include <stdio.h>
main()
 {
   int a=0, M, N, i;
   printf("N= "); scanf("%d", &N);
   printf("M= "); scanf("%d", &M);
   for(i=1; i<=N; i++)
     {
       if(i % 2 == 0)
         {
           printf("%5d", i);
```

```
            a++;
            if(a % M == 0)
                printf("\n");
           }
      }
   getch();
}
```

- **მასივები (მატრიცების გამრავლება):**

ამოცანა: „ორგანზომილებიანი მატრიცების გამრავლება ჩადგმული ციკლებით“.

```
/* ლისტინგი 5,11 ---მატრიცების გამრავლება --- */
#include <stdio.h>
#define M 3
#define N 4
#define K 2
main()
{
  int a[3][4]={{1,1,1,1},
               {2,2,2,2},
               {3,3,3,3}};
  int b[4][2]={{2,3},
               {2,3},
               {2,3},
               {2,3}};
   int c[3][2], i, j, k;
   for(i=0; i<M; i++)
      {
        for(j=0; j<K; j++)
          {
```

```
            c[i][j]=0;
            for(k=0; k<N; k++)
               c[i][j] += a[i][k]*b[k][j];
          }
       }
      clrscr();
      printf("Results :\n————————\n");
      for(i=0; i<M; i++)
       {
          for(j=0; j<K; j++)
            {
               printf("c[%d][%d]=%d", i,j,c[i][j]);
               printf("\n");
             }
       }
   getch();
}
```

- **ფაილებთან მუშაობა :**

ამოცანა: „ტექსტური ფაილის ელემენტების რეგისტრების ავტომატური ცვლილება".

```
/* ლისტინგი 5.12 --- file.txt  --- Up <==> Low --- */
#include <stdio.h>
#include <ctype.h>
#define UPPER 1
#define LOWER 0
main()
{
   int par;
  /* საწყისი და საშედეგო ფაილთა სახელები */
    char file1[14], file2[14];
    par = choose();
```

```
    getch();

/* ფაილთა სახელების მიღება */
getfiles(file1, file2);
getch();
/* გარდაქმნის პროცედურა */
conv(file1, file2, par);
getch();
}
choose()
{
int ch;
printf("შეიტანეთ U, თუ მთავრული ტექსტი გნებავთ \n");
printf("ან შეიტანეთ L, თუ სტრიქონული \n");
while ((ch=getchar()) != 'U' && ch != 'u' && ch != 'L' && ch !='l');
printf("შეიტანეთ U ან  L ! \n");
while (getchar() != '\n')
          ;              /* ჩამოაცილებს ახალი სტრიქონის
                            სიმბოლოს '\n' ბოლოში */
if (ch == 'U' || ch == 'u')
  {
   printf("Ok! დიდი ასოების რეგისტრი \n");
   return(UPPER);
  }
 else
  {
   printf("Ok! პატარა ასოების რეგისტრი \n");
   return(LOWER);
  }
}

 getfiles(char *name1, char *name2)
 {
  printf("რომელი ფაილის გარდაქმნა გნებავთ ? \n");
  gets(name1);
```

```
   printf("ესაა \" %s\". \n", name1);
   printf("ახალი საშედეგო ფაილის სახელი არის ?\n");

   while(strcmp(gets(name2), name1) == NULL)
        printf("შეარჩიეთ სხვა სახელი \n");

   printf("თქვენი გამომავალი ფაილია \"%s\".\n", name2);
  }

conv(char *name1, char *name2, int par)
  {
   FILE *f1, *f2;
   int ch;
   if ( (f1 = fopen(name1, "r")) == NULL)
     printf("ვერ ვხსნი %s-s. \n", name1);
   else
     {
       puts("პროცესის დაწყება !");
       f2 = fopen(name2, "w");
       while((ch = getc(f1)) != EOF)
         {
           if (par == UPPER)
               ch = islower(ch) ? toupper(ch) : ch;
           else
               ch = isupper(ch) ? tolower(ch) : ch;

          putc(ch, f2);
         }
   fclose(f2);
   fclose(f1);
   puts("the End !");
  }
}
```

5.2. ობიექტ-ორიენტირებული პროგრამირების C++ ენა

C++ ენას აქვს სტრუქტურები (struct) და კლასები (class). სტრუქტურა იგივე კლასია, ოღონდ „ყველასთვის". მის ატრიბუტებზე მიმართვა შეუძლია სტრუქტურის გარეთ მდებარე ყველა ფუნქციას, ვინაიდან გამოუცხადებლად მის ატრიბუტებს აქვს public ოფცია.

კლასი სტრუქტურაა, რომელსაც ქმნის მომხმარებელი და მის ატრიბუტებზე მიმართვა ყველას არ შეუძლია, ვინაიდან გამოუცხადებლად მის ატრიბუტებს აქვს private ოფცია, ე.ი. ისინი „დამალულია". კლასის კომპონენტები (ატრიბუტები და ფუნქციები) შეიძლება ცხადი სახით აღიწეროს: public, private da protected სპეციფიკატორებით. ბოლო ნიშნავს შეზღუდულ მიმართვას „მხოლოდ მეგობრებისათვის, მემკვიდრეობითი კავშირის მქონეებსთვის".

კლასის ატრიბუტები (data_members) განსაზღვრავს მისი ობიექტების მდგომარეობას, ხოლო წევრი-ფუნქციები (member_functions) ეთანადება კლასის მეთოდებს და განსაზღვრავს მის ქცევას. C++ ენის ფუნქციები შეიძლება დავყოთ სამ ჯგუფად:

1. საბაზო კლასის (სუპერკლასის) წევრი-ფუნქციები და ამ კლასის მეგობარი-ფუნქციები (friend_functions);

2. წარმოებული კლასის (სუბკლასის) წევრი-ფუნქციები და მისი მეგობარი-ფუნქციები. მათ შეუძლია დამატებით გამოიყენოს სუპერკლასის protected ატრიბუტით განსაზღვრული კომპონენტები;

3. ყველა დანარჩენი ფუნქცია, რომელთაც უფლება აქვს გამოიყენოს მხოლოდ public ატრიბუტით განსაზღვრული კომპონენტები.

ფუნქციების განსაზღვრა ხდება " :: " - მიკუთვნების ოპერატორით. ოთხი წერტილის მარცხნივ თავსდება კლასის სახელი, მარჯვნივ - წევრი-ფუნქცია. კავშირის ცალსახობა აქ აუცილებელია, ვინაიდან სხვადსხვა კლასს შეიძლება ჰქონდეს ერთნაირი სახელის წევრიფუნქციები.

ახლა განვიხილოთ C++ პროგრამის ტექსტები ობიექტ-ორიენტირებული კონცეფციის რეალიზაციის თვალსაზრისით, ანუ კლასები და ობიექტები, ინკაფსულაცია, მემკვიდრეობითობა და პოლიმორფიზმი და ა.შ. [66].

- **სტრუქტურების გადაცემა (struct):**

ამოცანა: „საჭიროა პიროვნების (Person სახელით) დოკუმენტის ნორმალიზებული ფორმის სტრუქტურით წარმოდგენა, იგი შედგება შემდეგი ატრიბუტებისგან: **გვარი, სტატუსი, ასაკი, კათედრის_#, შემოსავალი, ელ-ფოსტის-მისამართი.** სტრუქტურის გადაცემა ხდება მაჩვენებლის **(Person&)** გამოყენებით.

```
// ლისტინგი 5.13 ---struct transfer ---
#include <iostream.h>
#include <conio.h>
struct Person {
        char Name[20];
        char Status[15];
        int Age;
```

```
            int Deprtm_N;
            float Money;
            char E_mail[30]; };
Person& Person_info(Person& P) // ფუნქციაა !!!
 {
   cout<<"gvari : " <<P.Name<<endl<<
          "statusi : " <<P.Status<<endl<<
         "asaki : " <<P.Age<<endl<<
         "kat-# : " <<P.Deprtm_N<<endl<<
         "xelpasi : " <<P.Money<<endl<<
         "e-mail : " <<P.E_mail<<endl<<endl;
return P;
}

int main() // აქედან მიეწოდება კონკრეტული მნიშვნელობები
{
  Person P1={“Dolidze”,”Prof”,45,94,1500.55,
                ”tdol@gtu.edu.ge”};
  Person P2={“Gulua”,”Dr.Engineer”,25,94,900,
                ”dgulu@posta.ge”};
  Person P3={“Topuria”,”Doz”,30,94,850.5,
                ”ntopur@yahoo.com”};
  clrscr();
  cout<<“Results:”<<endl<<“======================”<<endl;
  Person_info(P1);
  Person_info(P2);
  Person_info(P3);
}
```

- **კლასები და ობიექტები :**

```
// ლისტინგი 5.14 ---class and objects ---
#include <conio.h>
class University { // კლასის სახელი
    private: // კლასის ლოკალური ატრიბუტები
        char Name[40];
        char Yubilee[];
    public: // კლასის გლობალური ფუნქციების გამოცხადება
        void Set_Uni(char*);
        void Pc_Uni(void);
        char* return_Uni(void); };
// ------ კლასის ფუნქციების აღწერა -------
void University :: Set_Uni(char *s) { strcpy(Name, s); }
void University :: Pc_Uni(void) { cout<<Name<<endl; }
char* University :: return_Uni(void){ Name;}

//---- მთავარი ფუნქცია ---------
void main(void)
{
  University Uni_name; // Uni_name ობიექტის გამოცხადება
  Uni_name.Set_Uni("Georgian Technical University");
  Uni_name.Pc_Uni();
  clrscr();
  cout<<Uni_name.return_Uni()<<endl;
  cout<<« 100 Year !»;
}
```

- **კლასის მეგობრები:**

```
// ლისტინგი 5.15 ---class friend function ---
#include <iostream.h>
#include <conio.h>
class Student; // პირველი კლასის გამოცხადება
class Magistr{ // მეორე კლასის გამოცხადება და აღწერა
   private:
        int Age;
        friend void F_F(Magistr&, Student&);
   public:
        Magistr(int P) : Age(P){};
   };

class Student{ // პირველი კლასის აღწერა
  private:
         int Age;
         friend void F_F(Magistr&, Student&);
  public:
         Student(int P) : Age(P){};
  };

// -----  მეგობარი კლასის ფუნქციის აღწერა ——
void F_F(Magistr& M1, Student& M2)
{ // ---  მონაცემთა კონტროლი --------
cout<<“Pers1=”<<M1.Age<<“ Pers2=”<<M2.Age<<endl;
if(M1.Age == M2.Age)
      cout<<“is equi \n”;
  else
      cout<<“is’t equi \n”;
}
```

```
// ——— მთავარი ფუნქცია ——————
void main(void)
{
   clrscr();
   Magistr Person1=20, Person3=25;
   Student Person2=20;
   F_F(Person1, Person2);
   Person1=Person3;
   F_F(Person1, Person2);
}
```

- **საბაზო და წარმოებული კლასები:**

```
// ლისტინგი 5.16 --- Based and Derived class ----
#include <iostream.h>
#include <conio.h>
class Person { // based class ----------
      char *Name;
public:
      void inform(char *n){ Name=n; }
      void disp() {cout<<endl<<Name;}
      char* ret_name(){return Name; }
   };

class Doctor : public Person { // derived class --------
          int status;
public:
      void inform(char *n, int s)
        {
           Person :: inform(n);
           status=s;
```

```
    }
   void disp()
    {
       Person :: disp;
       cout<<'\t'<<status<<endl;
    }
  char* ret_status() {return status;}
};

void main(void)
 {
    Person p. *pp;
    Doctor d, *pd;
    p.inform("Sharasha Bejo");
    d.inform("Petro Lily", 2);
    pp=&p;
      cout<<pp->ret_name(); // Sharasha Bejo
    pp->disp(); // Sharasha Bejo
    pp=pd=&d;
    cout<<endl<<pd->ret_status()<<endl; // 2
    cout<<pd->ret_name(); // Petro Lily
    pp->disp(); // Petro Lily
    pd->disp(); // Petro Lily 2
}
```

მთავარ პროგრამაში შემოტანილია საბაზო და წარმოებულ კლასთა ობიექტები. აქედან მათი დახმარებით კონკრეტული მნიშვნელობები გადაეცამა კლასთა ფუნქციებს შესასრულებლად. Person მშობელი კლასი მხოლოდ გვარს ბეჭდავს, Doctor შვილი კლასი - კი სტატუსს, გვარს და გვარს და სტატუსს ერთად.

- **კლასთა მემკვიდრეობითობის იერარქია:**

```
// ლისტინგი 5.17 --- class inheritance hierarchy ---
#include <iostream.h>
#include <conio.h>
class Babu { // based class ----------
      int b1;
  public:
         void sum(int a){ b1=a; }
         void disp() {cout<<b1<<endl;}
};

class Mama : public Babu { // derived and based class -------
       int b2;
public:
         void sum(int a){ Babu :: sum(5*a); b2=a;}
         void disp(){ cout<<b2<<endl;}
};

class Shvili : public Mama { // derived class ----------
       int b3;
public:
         void sum(int a){ Mama :: sum(10*a); b3=a;}
         void disp(){ cout<<b3<<endl;}
};

void main(void)
{
      Babu b, *sb;
      Mama m;
      Shvili s;
```

```
    b.sum(20);
    m.sum(30);
    s.sum(50);
    cout<<"Results:\n ————————————\n";
    b.disp(); // 20
    m.disp(); // 30
    s.disp(); // 50
    sb=&m;
    sb->disp(); // 150
    sb=&s;
    sb->disp(); // 2500
}
```

- **პოლიმორფიზმი და ვირტუალური ფუნქციები:**

```
// ლისტინგი 5.18 --- polymorphism and virtual function ----
#include <iostream.h>
#include <conio.h>
const char *const f_msg=" \nგამოძახებული ფუნქცია";
class Base {
        int d;
   public:
          void f(){ cout<<f_msg<<" Base :: f()\n"; }
          virtual void f(int i) {cout<<f_msg<<" Base :: f(int)\n";}
          void f(int i, int j) {cout<<f_msg<<" Base :: f(int,int)\n";}
};

class Deriv1 : public Base
 {
     char d1;
  public:
```

```
        void f(){ cout<<f_msg<<" deriv1 :: f()\n"; }
        virtual void f(int i) {cout<<f_msg<<"Deriv1 :: f()\n";}
};

class Deriv2 : public Base
{
  public:
        void f(){ cout<<f_msg<<" deriv2 :: f()\n"; }
        virtual void f(int i) {cout<<f_msg<<" Deriv2 :: f(int)\n";}
};

class Deriv12 : public Deriv1 {int d;};
//————————————————————————————-
typedef Base *pBase;
typedef Deriv1* pDeriv1;
int main()
{
  Base b_obj;
  Deriv1 d1_obj;
  Deriv2 d2_obj;
  Deriv12 d12_obj;
  cout<<" წევრი-ფუნქცია გამოიძახება პირდაპირ: \n";
  b_obj.f();
  b_obj.f(1);
  d2_obj.f(1);
  d2_obj.Base::f(1);
  d2_obj.Base::f(1,1);
  pBase b_ptr[4]; // მასივი Base-ს სამი მაჩვენებლისთვის
  b_ptr[0]=&b_obj;
  b_ptr[1]=&d1_obj; // Deriv1-ის არაცხადად გარდაქმნა Base-ში
   b_ptr[2]=&d2_obj; // Deriv2-ის არაცხადად გარდაქმნა Base-ში
```

```
// Deriv12-ის არაცხადად გარდაქმნა Base-ში
b_ptr[3]=&d12_obj;
// --------
char *types[4]={“Base”,”Deriv1",”Deriv2",”Deriv12"};
char *msg=”\nმითითებული ობიექტის ტიპი: “;
cout<<“ წევრი-ფუნქციები გამოიძახება მაჩვენებლიტ კლასზე:\n”;
getch();
cout<<“ არავირტუალური ფუნქცია f() ———\n”;
int c;
cout<<“Base-ს მაჩვენებლის ტიპი \n”;
//———————-
for(c=0; c<4; c++)
  {
   cout<<msg<<types[c];
   b_ptr[c]->f();
  }
cout<<“Deriv1*-ის მაჩვენებლის ტიპი \n”;
for(c=0; c<4; c++)
  {
    cout<<msg<<types[c];
    (pDeriv1(b_ptr[c]))->f();
   }
cout<<“ ვირტუალური ფუნქცია f(int)———\n”;
cout<<“მაჩვენებლის ტიპი Base* \n”;
for(c=0; c<4; c++)
  {
    cout<<msg<<types[c];
    b_ptr[c]->f(1);
   }
cout<<“Deriv1*-ის მაჩვენებლის ტიპი\n”;
for(c=0; c<4; c++)
```

```
  {
   cout<<msg<<types[c];
   (pDeriv1(b_ptr[c]))->f(1);
  }
// ვირტ. ფუნქციის გამოძახება, რომელიც არ ეთანადება
(pDeriv1(b_ptr[2]))-> Deriv1:: f(1); // ობიექტის ტიპს
 return 0;
}
```

- **აბსტრაქტული კლასები:**

აბსტრაქტულია კლასი შეიცავს აბსტრაქტულ ვირტუალურ ფუნქციას. ასეთ კლასს არ შეუძლია ობიექტის შექმნა. აბსტრაქტულია ვირტუალური ფუნქცია, რომელიც 0-ის ტოლია. მაგალითად, კლასი *გრაფიკული ფიგურა*:

```
// ლისტინგი 5.19 --- polymorphism and virtual function ---
#include <iostream.h>
#include <conio.h>
class Graph_Figur // ვირტუალური კლასი
  {
     . . .
     public:
     virtual void draw()=0;   // დახაზვა
     virtual void move()=0;  // გადატანა
     virtual void rotate()=0;  // მობრუნება
     virtual void resize()=0;  // ზომის შეცვლა
     virtual void copy()=0;   // დუბლირება
  };
```

აბსტრაქტულია კლასები გამოიყენება როგორც საბაზო კლასები სხვა წარმოებული კლასების მისაღებად. ასეთი

წარმოებული კლასები იქნება კონკრეტული, ობიექტების შექმნის უნარით და ისინი გამოიყენებს აბსტრაქტულ ვირტუალურ ფუნქციებს. მაგალითად:

```
class Circle : public Graph_Figur // წარმოებული კლასი
  {
      int x, y, radius;
   public:
      void draw(void);
      void move(void);
      void rotate(void);
      void resize(void);
      void copy(void);
  };
```

ასევე შეიძლება დამუშავდეს სხვა კონკრეტული ობიექტები: ელიფსი, მართკუთხედი, კუბი და სხვ.

- **მრავალჯერადი მემკვიდრეობითობა და ვირტუალური საბაზო კლასები:**

თუ წარმოებული კლასი აგებულია ორი ან მეტი საბაზო კლასის საფუძველზე, მაშინ გვაქვს მრავალჯერადი მემკვიდრეობითი კავშირები. პროგრამული რეალიზაციის ფრაგმენტი ორი საბაზო კლასის მაგალითზე მოცემულია 5.20 ლისტინგში.

```
// ლისტინგი 5.20 --- multiple inheritances ---
#include <iostream.h>
#include <conio.h>
class Base_Class1 {. . .}
class Base_Class2 {. . .}
```

```
. . .
class New_Class :
public Base_Class1{};
public Base_Class2{};
```

მრავალჯერადი მემკვიდრეობითი კავშირების მქონე წარმოებული კლასი, გარდა საკუთარი ატრიბუტებისა და ფუნქციებისა, გამოიყენებს მისი ყველა საბაზო კლასის შესაბამის კომპონენტებს. წარმოებულ კლასში შესაძლებელია ერთიდაიმავე საბაზო კლასის რამდენჯერმე დაფიქსირება. მაგალითად:

```
/* ლისტინგი 5.21 --- multiple inheritances2 --- */
#include <iostream.h>
#include <conio.h>
class Base_ClassT {. . .}
. . .
class Base_Class1 : public Base_ClassT{. . .}
class Base_Class2 : public Base_ClassT{. . .}
. . .
class New_Class :
public Base_Class1{};
public Base_Class2{};
```

ერთი და იგივე საბაზო კლასი შეიძლება გამოცხადდეს როგორც ერთი ვირტუალური საბაზო კლასი.

```
/* ლისტინგი 5.22 --- multiple inheritances3 --- */
#include <iostream.h>
#include <conio.h>
class Base_ClassT {. . .}
. . .
class Base_Class1 : virtual public Base_ClassT{. . .}
```

```
class Base_Class2 : virtual public Base_ClassT{. . .}
class New_Class :
      public Base_Class1{};
      public Base_Class2{};
```

ვირტუალური საბაზო კლასის გამოყენება საშუალებას იძლევა მისთვის აგებულ იქნეს ამ კლასის მხოლოდ ერთი ობიექტი. წინააღმდეგ შემთხვევაში, თუ ერთი სახელის ორი ან მეტი არავირტუალური საბაზო კლასი გვაქვს, მაშინ მისთვის აგებულ უნდა იქნეს ერთზე მეტი ობიექტი ამ ერთი კლასისათვის.

5.3. ობიექტ-ორიენტირებული პროგრამირების Java ენა

➢ მონაცემთა *შეტანა-გამოტანის* ოპერაციები:

Java-ენაში გამოიყენება მონაცემთა გამოტანის შემდეგი მარტივი კონსტრუქცია:

System.out.println( );

სადაც ფრჩხილებში თავსდება ეკრანზე გამოსატანი არგუმენტები და სტრიქონები. „ + “ ოპერატორით შესაძლებელია არგუმენტების კონკატენაცია (გადაბმა). განვიხილოთ კონკრეტული პროგრამა [22,67].

```
// ლისტინგი 5.23 --- simple date-output ---
public class dateOut {
    public static void main(String[] args) {
        System.out.println("55-33=" + (55-33) ); }
}
```

შედეგში მიიღება: 55 - 33 = 22.

მონაცემთა შესატანად java-ენაში გამოიყენება System.in მარტივი კონსტრუქცია, მაგრამ იგი დამატებით მოითხოვს დაზუსტებას. მონაცემების შეტანისას გამოიყენება ნაკადებზე ბაზირებული შეტანა-გამოტანის ოპერაციები. ის რომ გარდაიქმნას მარტივი ტიპის მონაცემებად, საჭიროა BufferedReader და InputStreamReader კლასის გამოყენება. განვიხილოთ პროგრამის ტექსტი.

```
/* ლისტინგი 5.24 --- simple date.in  --- */
import java.io.*;
public class dateIn
 {
   public static void main(String[] args)
   throws IOException // throws - განსაკუთრებული (გამოსარიცხი)
    { int a, b, c;          // შემთხვევის გენერირების ოპერატორი
                            // new - მეხსიერების გამოყოფა
      BufferedReader din=new BufferedReader(
                                new InputStreamReader(System.in) );
      System.out.println("a = ");
      a=Integer.parseInt(din.readLine());
      System.out.println("b = ");
      b=Integer.parseInt(din.readLine());
      c = a + b;
      System.out.println("a+b = " + c);
    }
}
```

შედეგს შეიძლება ჰქონდეს ასეთი სახე:

```
a = 150
b = 400
a + b = 550
```

შედეგი, რომელიც მიიღება din.readLine-ით, არის String (სტრიქონული). ვინაიდან ჩვენ აქ რიცხვებს ვიყენებთ, ამიტომ საჭიროა მათი გარდაქმნა. განვიხილოთ პროგრამა, რომელშიც გამოყენებულია მონაცემთა სხვადასხვა მარტივი ტიპები და ახალი კონსტრუქციები.

```
// ლისტინგი 5.25 --- simple date-in2 ---
class Keyboard {
                // Primitive Keyboard input of integers, reals,
                // strings and characters.
    static boolean iseof = false;
    static char c;
    static int i;
    static double d;
    static String s;

    static BufferedReader input = new BufferedReader (
      new InputStreamReader(System.in), 1); // ბუფერში 1 მონაცემი

    public static int readInt () {
       if (iseof) return 0;
       System.out.flush();
       try {
         s = input.readLine();
       }
       catch (IOException e) {
          System.exit(-1);
       }
    if (s==null) {
      iseof=true;
      return 0;
    }
   i = new Integer(s.trim()).intValue();
```

```
    return i;
   }

    public static char readChar () {
if (iseof) return (char)0;
 System.out.flush();
try {
     i = input.read();
 }
 catch (IOException e) {
   System.exit(-1);
 }
if (i == -1) {
   iseof=true;
   return (char)0;
  }
  return (char)i;
}

  public static double readDouble () {
   if (iseof) return 0.0;
   System.out.flush();
   try {
       s = input.readLine();
     }
       catch (IOException e) { System.exit(-1);
     }
  if (s==null) {
   iseof=true;
   return 0.0;
  }
     d = new Double(s.trim()).doubleValue();
```

```
    return d;
  }
public static String readString () {
  if (iseof) return null;
  System.out.flush();
  try {
     s=input.readLine();
   }
    catch (IOException e) {
     System.exit(-1);
   }
  if (s==null) {
    iseof=true;
    return null;
  }
 return s;
}

public static boolean eof () {
   return iseof;
 }
}
```

- **მასივები:**

Java-ენის ინსტრუმენტს აქვს ჩადგმული კლასი შტრინგ, რომელიც მასივების დროს ხშირად გამოიყენება. შესაძლებელია აგრეთვე ახალი კლასების შექმნაც. მასივების გამოსაყენებლად Java-ენაში სამი პროცედურაა შესასრულებელი: *მასივის გამოცხადება* (სახელით, ტიპით და ელემენტებით), *მისთვის მეხსიერების გამოყოფა*

(ელემენტთა რაოდენობის მითითებით) და *ინიციალიზება* (ელემენტებზე საწყისი მნიშვნელობების მინიჭება). განვიხილოთ პროგრამები.

```
// ლისტინგი 5.26 --- Array-1 ---
public class ArrayElem {
    public static void main(String[] args) {
      int Mas[] = new int[5];
      boolean bulbul[] = {true, false};
      Mas[0]=1; Mas[1]=8; Mas[2]=0; Mas[3]=2; Mas[4]=15;
      System.out.println("Mas has "+Mas.length+" Elements");
      System.out.println("bulbul has "+bulbul.length+" Elements");
      System.out.println(Mas[0]); System.out.println(Mas[1]);
      System.out.println(Mas[2]); System.out.println(Mas[3]);
      System.out.println(Mas[4]);
      System.out.println(bulbul[0]); System.out.println(bulbul[1]);
    }
}
```

შედეგს ექნება ასეთი სახე:

```
Mas has 5 Elements
bulbul has 2 Elements
1 8 0 2 15
true false
```

მეორე Java-პროგრამა მასივზე:

```
/* ლისტინგი 5.27 --- Array_Year --- */
class Monate {
    public final static String[]   // კონსტანტებია
    MONTH_NAME = {"",
          „Januar“, „Februar“, „Maerz“, „April“,
          „Mai“, „Juni“, „Juli“, „August“,
          „September“, „Oktober“, „November“, „Dezember“};
```

```
    public final static int[]
    DAYS_OF_MONTH = {0,
                              31, 28, 31, 30,
                              31, 30, 31, 31,
                              30, 31, 30, 31
                            };
    public final static int
        JANUAR=1, FEBRUAR=2, MAERZ=3, APRIL=4, MAI=5,
        JUNI=6, JULI=7, AUGUST=8, SEPTEMBER=9,
        OKTOBER=10, NOVEMBER=11, DEZEMBER=12;
    static int M = MAI;

    public static void main (String[] args) {
        System.out.println(MONTH_NAME[M]+ "თვეს "
            + " აქვს " + DAYS_OF_MONTH[M] + " დღე !");
        }
}
```

შედეგი: Mai თვეს აქვს 31 დღე !

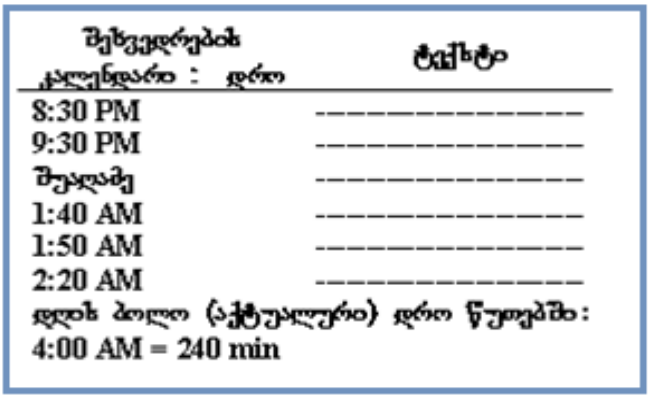
- **სტრუქტურული ელემენტები (განშტოება, ციკლი):**

```
// ლისტინგი 5.28 --- if…else ---
class ZeitPlan {
  private static int hour, minute; // აქტუალური დრო !
  private static void addMinutes (int m)
    {  // აქტუალურ დროს ზრდის m წუთით
     int totalMinutes = (60*hour + minute + m) % (24*60);
     if (totalMinutes < 0)
        totalMinutes = totalMinutes + 24*60;

     hour = totalMinutes/60;
```

```
      minute = totalMinutes%60;
    }
private static int timeInMinutes ()
   {
    // ითვლის წუთებს 0:00 საათიდან აქტუალური დროისთვის
     int totalMinutes = (60*hour + minute) % (24*60);
     if (totalMinutes < 0)
        totalMinutes = totalMinutes + 24*60;

     return totalMinutes;
   }
private static void printTime ()
  {
    // ბეჭდავს აქტუალური დროს AM, PM -ით ----------
     if ((hour == 0) && (minute == 0))
           System.out.print("შუაღამე");
       else
           if ((hour == 12) && (minute == 0))
                System.out.print("შუადღე");
              else {
                  if (hour == 0) System.out.print(12);
                 else
                   if (hour > 12) System.out.print(hour-12);
                     else
                       System.out.print(hour);

     if (minute < 10)
           System.out.print(":0" + minute);
        else
           System.out.print(":" + minute);
     if (hour < 12)
           System.out.print("AM");
```

```
        else
            System.out.print("PM");
     }
   }
private static void printTimeInMinutes ()
    { // ბეჭდავს აქტუალურ დროს წუთებში ----------
      printTime ();
      System.out.println(„ = „ + timeInMinutes() +„. Minute des
                                                  Tages „);
     }
private static void includeNewEntry (int intervalInMinutes)
    { // ბეჭდავს სტრიქონს: დროს და , _____ ;
      // ზრდის დროს ინტერვალებისა შესაბამისად
      printTime();
      System.out.println(" _____");
      addMinutes(intervalInMinutes);
    }
//-----------------------------------------------

public static void main (String[] args)
 {
   hour = 8; minute = 30;
   // ბეჭდავს დროით მონაცემებს ცხრილის სახით
    System.out.println(„შეხვედრების კალენდარი: დრო +
ტექსტი");
     System.out.println(„————————————————————");
    printTime();
     System.out.println(" _____");
    addMinutes(30);
    printTime();
     System.out.println(" _____");
```

```
    addMinutes(30);
    printTime();
    System.out.println(“ _____”);
    addMinutes(150);
    printTime();
    System.out.println(“ _____”);
    addMinutes(100);
    printTime();
    System.out.println(“ _____”);
    addMinutes(10);
    printTime();
    System.out.println(“ _____”);
    addMinutes(30);
    includeNewEntry(100);
    System.out.println(“ “);
System.out.println(“დღის ბოლო (აქტუალური) დრო წუთებში“);
    printTimeInMinutes ();
  }
} // შედეგი (ნახ..5.3).
```



ნახ.5.3

- **ციკლის ოპერატორები:**

ამოცანა: „აიგოს პროგრამა მასივში ელემენტების სწრაფი მოწესრიგებისათვის“.

```
// ლისტინგი 5.29 --- for, while, do…while ---
public class Quicksort {
    public static void quicksort (int[] a, int links, int rechts) {
    int help; int i = links; int j = rechts;
    int x = a[(links+rechts) / 2];  // შედარების ელემენტი
do {
    while (a[i] < x)  i++;
    while (a[j] > x)  j—;
    if ( i <= j ) {
          help = a[i];
          a[i] = a[j];
          a[j] = help;
          i++; j—;
      }
} while (i <= j);
// მარცხენა ნაწილში ელემენტები ნაკლებია მარჯვენაზე.
// მარცხენა-მარჯვენა ნაწილების მოწესრიგება
if (links < j) quicksort(a, links, j);
if (i < rechts ) quicksort(a, i, rechts );
} // --- end class ----------------------------------

public static void main (String argv[])
{
  int[] a; int n;
 System.out.print(" ველის სიგრძე  : ");
     n = Keyboard.readInt(); a = new int[n];
 System.out.println(n + " ველის მნიშვნელობა : ");
```

```
  for (int i = 0; i < n; i++)
    {
      System.out.print(" შემდეგი რიცხვი :");
          a[i] = Keyboard.readInt();
    }

   quicksort(a, 0, n-1);  // საშედეგო ველის დალაგება----
   System.out.println("მოწესრიგებული ველი : "); // შედეგი
   for (int i=0; i<a.length; i++)
        System.out.print(“ “ + a[i]);
    System.out.println();
  }
}
```

- **კლასები, ობიექტები და მეთოდები:**

Java ენაში არ გამოიყენება struct ელემენტი, რომელიც C&C++ ენის ერთ-ერთი მთავარი კონსტრუქციაა. მის ფუნქციას აქ ასრულებს ტერმინი class. როგორც ცნობილია, კლასი მონაცემების (ატრიბუტების) და მათი დამუშავების მეთოდების (ფუნქციების) ინკაფსულირებითაა მიღებული.

განვიხილოთ პროგრამის მაგალითი Java ენაზე კლასებისა და ობიექტების წარმოსადგენად.

```
// ლისტინგი 5.30 --- class, object, methods ---
class Schedule { private static void includeNewEntry
                    (Time t, String s, int intervalInMinutes) {
// ბეჭდავს სტრიქონს: დრო, ღონისძიების დასახელება s;
// უმატებს დროს ღონისძიების ხანგრძლიობას
        t.printTime();
        System.out.println(“ “+ s);
```

```
        t.addMinutes(intervalInMinutes);
      }
public static void main (String[] args) {
    Time t1 = new Time(8,30); Time t2 = new Time(); Time t3, t4;
    // ბეჭდავს შეხვედრების გეგმას
     System.out.println("პირველი გეგმა");
     includeNewEntry(t1,“PI1",90);
     includeNewEntry(t1,"შესვენება",15);
     includeNewEntry(t1,”TI1",90);
     includeNewEntry(t1,"შესვენება",15);
       t3 = t1; // მიმართვა იგივე ობიექტზე
       t4 = t1.copy(); // მთელი ობიექტის დუბლირება
  includeNewEntry(t1,“Ma1",100);
  includeNewEntry(t1,"შესვენება",15);
  includeNewEntry(t1,”Ma2",20);
  System.out.println("დღის ბოლო(აქტუალური)  დრო წთ-ში:");
  t1.printTimeInMinutes ();
  System.out.println(" ");
System.out.println("მეორე გეგმა:");
  includeNewEntry(t2,"მეორადი საგანი",100);
  includeNewEntry(t2,“სემინარი“,90);
  System.out.println("დღის ბოლო(აქტუალური)  დრო წთ-ში");
  t2.printTimeInMinutes ();
  System.out.println(" ");
System.out.println("მესამე გეგმა:"); // t1, t3 -> ერთი ობიექტია
  includeNewEntry(t3,"სპორტი",100); // t3: აქტუალური  t1-დრო
  includeNewEntry(t3,"დასვენება",200);
  System.out.println("დღის ბოლო(აქტუალური)  დრო წთ-ში:");
  t3.printTimeInMinutes ();
  System.out.println(" ");
if (t4.before(t1)) {
```

```
    System.out.println("მეოთხე გეგმა:"); // t1 <> t4
includeNewEntry(t4,"მეორე დამატ.საგანი",100);
                                        //t4: ძველი  t1-დრო
    includeNewEntry(t4,"თავისუფალი“,200);
    System.out.println("დღის ბოლო(აქტუალური)  დრო წთ-ში ");
    t4.printTimeInMinutes ();
   }
  }
}
```

ამ პროგრამაში გამოყენებულია კლასი Time, რომლითაც განისაზღვრება t1, t2, t3 და t4 ობიექტები.

➢ ინტერფეისების დაპროგრამება:

Java ენაში მრავალჯერადი მემკვიდრეობითი კავშირების რეალიზაციისთვის გამოიყენება კლასის განსაკუთრებული ფორმა, რომელსაც ინტერფეისს (Interfaces) უწოდებენ. იგი შედგება მხოლოდ აბსტრაქტული მეთოდებისა და კონსტანტებისაგან და შეიძლება არ ჰქონდეს კონსტრუქტორი. ინტერფეისი არის უფრო მოქნილი სტრუქტურა, ვიდრე აბსტრაქტული კლასი, ვინაიდან მისი საშუალებით შესალებელია კლასი გამდიდრდეს ახალი ფუნქციონალური შესაძლებლობებით.

კლასთა მემკვიდრეობითობის განხილვისას მოიხსენიებენ ტერმინს „წარმოებული" (საბაზო A-კლასიდან წარმოებული B-ქვეკლასი). ინტერფეისის შემთხვევაში ამ ცნების ანალოგიურია „რეალიზაცია, შესრულება˜ (Implementation). ინტერფეისის რეალიზაციის საშუალებით კლასი ვალდებულია მასში გამოცხადებული ყველა მეთოდი

აღწეროს. ერთი მეთოდიც რომ აკლდეს, მივიღებთ შეცდომას. განვიხილოთ მაგალითი:

```
// ლისტინგი 5.31 --- Interfaces-1 ---
public class Person
implements ClassName
 {
    public String name;
    public int Atr1;
    public int Atr2; // მაგ., დაბადების წელი
    public int Atr3; // ასაკი
    private double Atr4; // თვიური ხელფასი
    private double Atr5; // ხელზე ასაღები თანხა

    public int method1()
       {
         return 2003-Atr2;
       }
    public double method2()
      {
        return Atr4 - Atr4*KP;  // KP დასაბეგრი-%  (მაგ.,  20%)
      }
 }
```

პროგრამაში საკვანძო სიტყვა implements გამოიყენება იმისათვის, რომ კომპილატორს გადაეცეს შეტყობინება კლასში რეალიზებული ინტერფეისის შესახებ. კომპილატორი სინტაკსურად ამოწმებს ინტერფეისის ტექსტს და გამოსცემს ინფორმაციას არსებული შეცდომების შესახებ (თუ ასეთი არსებობს). მაგალითად, თუ ინტერფეისში გამოცხადებული რამდენიმე მეთოდიდან რომელიმე მათგანი არაა აღწერილი.

ერთი ინტერფეისიდან შეიძლება სხვა ინტერფეისების წარმოება, როგორც ეს კლასისთვის იყო დამახასიათებელი. მათ გადაეცემა მემკვიდრეობით პირველი ინტერფეისის ყველა თვისება. მას შემდეგ, რაც ინტერფეისი რეალიზებულ იქნება პროგრამაში, შესაძლებელია მისი გამოყენება როგორც *ახალი ტიპისა* (კლასის მაგვარად).

განსხვავება კლასსა და ინტერფეისს შორის ამ შემთხვევაში ისაა, რომ კლასი (როგორც ტიპი) გადასცემს ობიექტს თავის *ყველა* თვისებას, ინტერფეისი კი - *მხოლოდ ნაწილს* (რაც ხშირად უფრო მოსახერხებელი და მოქნილია).

ინტერფეისებს შეუძლიათ static და final ატრიბუტებით კონსტანტების გამოყენება. შეიძლება ინტერფეისი მხოლოდ კონსტანტებიდან შედგებოდეს:

```
// ინტერფეისის ტანში მეთოდის გამოცხადება
public void aDarizxTanxa();

// კონსტანტის გამოცხადება
public static final long kMinute=11520;
```

თუ ინტერფეისი გამოცხადებულია როგორც public, მაშინ ნებისმიერ კლასს შეუძლია მასში გამოცხდებული კონსტანტების წაკითხვა. განვიხილოთ პროგრამის ტექსტი ინტერფეისით.

```
// ლისტინგი 5.32 --- Interfaces-2 ---
interface TimeI {
    public void addMinutes (int m);
    public void subtractMinutes (int m);
    public void printTime ();
```

```
    public void printTimeInMinutes () ;
}
class Time implements TimeI
 {
    private int hour, minute; // აქტუალური დრო
    public Time (int h, int m)
      {
       hour = h;
       minute = m;
      }
    public Time ()
     {
      hour = 0;
      minute = 0;
     }
    public void addMinutes (int m)
     {
       //  აქტუალური დრეოს ზრდის m წუთით
      int totalMinutes = (60*hour + minute + m) % (24*60);
      if (totalMinutes < 0)
            totalMinutes = totalMinutes + 24*60;

      hour = totalMinutes/60;
      minute = totalMinutes%60;
    }
public void subtractMinutes (int m)
   {
    addMinutes(-m);
   }
public void printTime () {
// ბეჭდავს აქტუალურ დროს AM, PM -ით -----
```

```
if ((hour == 0) && (minute == 0))
    System.out.print("შუაღამე");
else
    if ((hour == 12) && (minute == 0))
        System.out.print("შუადღე");
      else {
        if (hour == 0)
            System.out.print(12);
         else
           if (hour > 12)
               System.out.print(hour-12);
            else
               System.out.print(hour);

          if (minute < 10)
               System.out.print(":0" + minute);
            else
               System.out.print(":" + minute);

          if (hour < 12)
               System.out.print("AM");
            else
               System.out.print("PM");
        }
  }

 public void printTimeInMinutes ()
  { //  ბეჭდავს აქტუალურ დროს წუთებში
    printTime ();
    System.out.println(„ = „ + timeInMinutes() + " წუთი ");
  }
```

```
private int timeInMinutes ()
  { // private: დამხმარე ფუნქცია
    // ითვლის წუთებს 0:00 საათიდან აქტუალური დროისთვის
    int totalMinutes = (60*hour + minute) % (24*60);
    if (totalMinutes < 0)
        totalMinutes = totalMinutes + 24*60;

    return totalMinutes;
  }
}
class ScheduleInt {
    private static void includeNewEntry
              (TimeI t, String s, int intervalInMinutes) {
    // ბეჭდავს სტრიქონს: დროს, s-ღონისძიების დასახელებას;
    // ზრდის ტ-დროს ღონისძიების ინტერვალის შესაბამისად
    t.printTime();
    System.out.println(“ “+ s);
    t.addMinutes(intervalInMinutes);
  }
public static void main (String[] args) {
    TimeI t1 = new Time(8,30);
    // ბეჭდავს შეხვედრათა გეგმებს:
    System.out.println("პირველი გეგმა:");
    includeNewEntry(t1,“PI1",90);
    includeNewEntry(t1,"შესვენება",15);
    System.out.println("ბოლო (აქტუალური) დრო წუთებში: ");
    t1.printTimeInMinutes ();
    System.out.println(" ");
  }
}
```

- **მემკვიდრეობითობის დაპროგრამება:**

```
// ლისტინგი 5.33 --- Inheritance ---
class Time
 {
  protected int hour, minute;
  public Time (int h, int m) { hour = h; minute = m; }
  public void addMinutes (int m)
   {
    int totalMinutes = (60*hour + minute + m) % (24*60);
    if (totalMinutes < 0)
        totalMinutes = totalMinutes + 24*60;

    hour = totalMinutes/60;
    minute = totalMinutes%60;
  }
public void printTime ()
 {
   if ((hour == 0) && (minute == 0))
      System.out.print("შუაღამე");
    else if ((hour == 12) && (minute == 0))
           System.out.print("შუადღე");
         else { if (hour == 0)
                  System.out.print(12);
              else if (hour > 12)
                     System.out.print(hour-12);
                   else System.out.print(hour);
                if (minute < 10)
                   System.out.print(":0" + minute);
                 else System.out.print(":" + minute);
                if (hour < 12)
                   System.out.print("AM");
```

```
                else System.out.print("PM");
              }
            }
      }
class PreciseTime extends Time
  {
    private int second;
    public PreciseTime (int h, int m, int s)
      {
        super(h, m);
        second = s;
      }
    public void addSeconds (int s)
      {
        int advMinutes = s / 60;
        second += s % 60;
        if (second < 0)
          {
            advMinutes—;
            second += 60;
          }
        else
            if (second >= 60)
              {
                advMinutes++;
                second -= 60; }
                addMinutes(advMinutes);
              }
   public void printTime ()
     {
       if ((hour == 0) && (minute == 0))
          System.out.print("შუაღამე");
```

```
        else
          if ((hour == 12) && (minute == 0))
              System.out.print("შუადღე");
            else
              { if (hour == 0)
                    System.out.print(12);
                  else
                    if (hour > 12)
                       System.out.print(hour-12);
                     else
                       System.out.print(hour);

                if (minute < 10)
                    System.out.print(":0" + minute);
                  else
                    System.out.print(":" + minute);

                if (second < 10)
                     System.out.print(":0" + second);
                   else
                     System.out.print(":" + second);
                if (hour < 12)
                     System.out.print("AM");
                   else
                      System.out.print("PM");
                }
            }
        }
class Time2 { public static void main(String[] args)
  {
    PreciseTime lunchtime = new PreciseTime(12, 1, 0);
```

```
  lunchtime.addMinutes(1);
  lunchtime.printTime();
    System.out.println();
  lunchtime.addSeconds(-61);
  lunchtime.printTime();
    System.out.println();
  lunchtime.addSeconds(1);
  lunchtime.printTime();
    System.out.println();
  }
}
```

5.4. ობიექტ-ორიენტირებული პროგრამირების C# ენა

ამ პარაგრაფში გადმოცემულია C# ენის სტრუქტურული, ობიექტ-ორიენტირებული და ვიზუალური პროგრამული კოდების ფრაგმენტები Console Application და Windows Forms Application რეჟიმებში Visual Studio.NET პლატფორმაზე [59, 68].

- **სტრუქტურული ელემენტები (ციკლი, გადამრთველი):**

ამოცანა_1: დავწეროთ C# ინტერაქტიული კოდი მარტივი კალკულატორის მაგალითისათვის. დიალოგში შეიტანება ორი მთელი რიცხვი და ერთი არითმეტიკული ოპერაცია (+, -, * ან /). პროგრამას გამოაქვს გამოთვლის შედეგი. პროცესის გაგრძელება (ციკლური გამეორება) „Yes” ან დასასრული „No” (პროგრამიდან გამოსვლა).

Solution Explorer-დან ვამატებთ ახალ პროექტს ConsoleApp3 სახელით და მასში ვათავსებთ ამოცანის

გადაწყვეტის შესაბამის კოდს. აქ გამოყენებულია ციკლის (while (…)) და გადამრთველის [switch(op), სადაც op - არითმეტიკული ოპერაციის კოდია] ოპერატორები. პროგრამის ტექსტი მოცემულია ლისტინგში:

// ლისტინგი 5.34 --- while…switch() ---

```
using System;
namespace ConsoleApp3
{
 class Program
  {
   static void Main(string[] args)
   {
     int x, y, s;
     char op, yn;
     Console.Write("Calculation - y, End - n: ");
     yn = Convert.ToChar(Console.ReadLine());
     while (yn == 'y' || yn == 'Y')
      {
        Console.Write("Input \n");
        Console.Write("First number: ");
          x = Convert.ToInt32(Console.ReadLine());
        Console.Write("Second number: ");
          y = Convert.ToInt32(Console.ReadLine());
        Console.Write("Operacia: ");
         op = Convert.ToChar(Console.ReadLine());
        switch (op)
         {
          case '+':
                    s = x + y;
                  Console.Write("Shedegi: ");
                  Console.WriteLine("Sum = " + s);
                  break;
           case '-':
                    s = x - y;
                    Console.Write("Shedegi: ");
                    Console.WriteLine("Dif = " + s);
                    break;
```

```
            case '*':
                     s = x * y;
                     Console.Write("Shedegi: ");
                     Console.WriteLine("Prod = " + s);
                     break;
            case '/':
                     s = x / y;
                     Console.Write("Shedegi: ");
                     Console.WriteLine("Div = " + s);
                     break;
            case '%':
                     s = x % y;
                     Console.Write("Shedegi: ");
                     Console.WriteLine("Rest = " + s);
                     break;
            default:
                Console.Write("operation not Correct !");
                break;
          }
         Console.Write("\nCalculation - y, End - n: ");
         yn = Convert.ToChar(Console.ReadLine());
       }
       Console.Write("End the process !");
    }
   }
}
```

ამოცანა_2: ავაგოთ პროგრამული კოდი, რომელიც:

ა) /მაგ., button1/ გაითვლის ნებისმიერი მითითებული მთელი რიცხვისათვის **ფიბონაჩის** მწკრივში მის მომდევნო „ფიბონაჩის რიცხვის" მნიშვნელობას (მაგალითად, რიცხვი=50, რომელია მის მარჯვნივ მდგარი უახლოესი ფიბონაჩის რიცხვი ?);

ბ) /მაგ., button2/ გაითვლის ფიბონაჩის რიცხვის მნიშვნელობას მე-n ელემენტისათვის ფიბონაჩის რიცხვთა

მწკრივში. (მაგალითად, რას უდრის მწკრივში რიგით მე-15 ფიბონაჩის რიცხვი ?);

გ) /მაგ., button3/ გაითვლის ფიბონაჩის მწკრივის რიცხვთა ჯამს მითითებული მე-n ელემენტის ჩათვლით (მაგალითად, რას უდრის ფიბონაჩის მწკრივის [1-20] რიცხვების ჯამი ?).

როგორც ცნობილია, *ფიბონაჩის რიცხვთა მწკრივი* შემდეგი სახისაა:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

ანუ, რიცხვთა მწკრივის აგების წესი ასეთია: *მომდევნო რიცხვი არის წინა ორი რიცხვის ჯამი.*

```
// ა) ლისტინგი 5.35 --- while…Fibonachi_1 ---
private void button1_Click(object sender, EventArgs e)
 {  // ა-ღილაკი
  int Previous, Next;   // ფიბონაჩის მწკრივის „წინა“ და
                                  // „მომდევნო“ რიცხვები
  Previous = -1;
  Next = 1;
  string ns = textBox2.Text;  // საწყისი მონაცემის შეტანა
  int n = Convert.ToInt32(ns);  // ტიპის გარდაქმნა
 int fib = 1;
 while (fib <= n)  // ციკლი !!!
 {
   fib = Previous + Next;
   Previous = Next;
   Next = fib;
 label12.Text = " " + fib.ToString();  // შედეგი
  }
 }


// ბ) ლისტინგი 5.36 --- while…Fibonachi_2 ---
private void button2_Click(object sender, EventArgs e)
```

```
{
 int Fibon;
 string ns = textBox3.Text;
 int n = Convert.ToInt32(ns);
 Fibon=Fibonacci(n-1);
 label12.Text = Fibon.ToString();
}

int Fibonacci(int n)   // რეკურსიული მეთოდი
 {
     return n > 1 ? Fibonacci(n-1) + Fibonacci(n-2) : n;
 }

// გ) ლისტინგი 5.37 --- while…Fibonachi_3 ---
private void button3_Click(object sender, EventArgs e)
 {
  string ns = textBox3.Text;
  int n = Convert.ToInt32(ns);
  int zinaFib = 0, momdevnoFib = 1;
  int SumFib = 0;
  int i = 1;
  while (i <=n)
     {
      int temp = momdevnoFib;
       momdevnoFib += zinaFib;
       SumFib += zinaFib;
       zinaFib = temp;
       i++;
      }
     label12.Text = SumFib.ToString();
  }
```

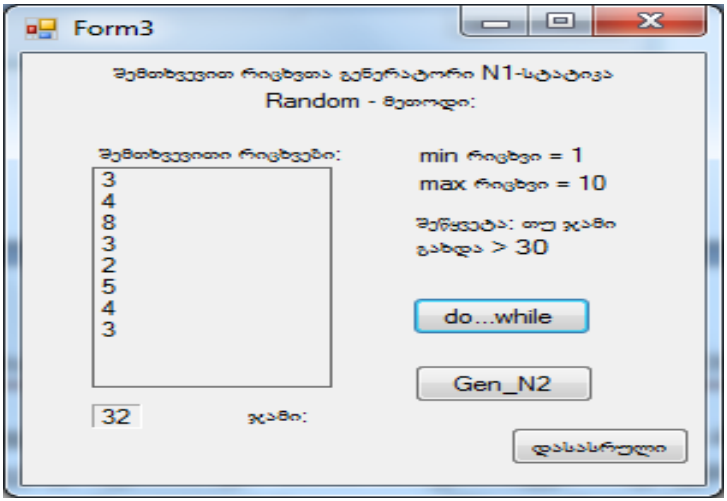
ამოცანა_3: „ავაგოთ C# კოდი, რომელიც შემთხვევით რიცხვთა გენერატორის საშუალებით (Random-კლასის

ობიექტით) ციკლურად ამოიღებს რიცხვებს მითითებულ დიაპაზონში (Next(min,max-1)) და მოათავსებს label3-ში. უნდა განისაზღვროს ამ რიცხვთა ჯამი label4-ში".

label3-ის თვისებების შეცვლით AutoEllipsis=True და AutoSize+False შესაძლებელია უჯრის გაფართოება და მასზე რამდენიმე სტრიქონის ან ტექსტის გამოტანა. label4 გავხადეთ 3-განზომილებიანი: BorderStyle=Fix3D.

```
// ლისტინგი 5.38 --- do…while…Random() ---
using System;
using System.Drawing;
using System.Windows.Forms;
namespace WinFormCycle
{
 public partial class Form3 : Form
 {
  Random r = new Random(); //Random-კლასი -შრ-გენერატორი
  public Form3() { InitializeComponent(); }
 private void button1_Click(object sender, EventArgs e)
    { // გენერატორის ამოქმედება
      int summe = 0, z;
      label3.Text = "";
      do
       {
        z=r.Next(1, 10); // Next მეთოდია Random-კლასის
        summe += z;
        label3.Text += z.ToString() + "\n";
       }
        while (summe < 30)
                ;
      label4.Text = summe.ToString();
   }
 private void button2_Click(object sender, EventArgs e)
```

```
    { Close(); }
  }
} // შედეგი მოცემულია 5.4 ნახაზზე.
```



ნახ.5.4. ციკლი შრგ-ით

ნახაზზე ნაჩვენებია შედეგები. “do…while” ღილაკის ყოველ ახალ ამოქმედებაზე მუშაობას იწყებს შემთხვევით რიცხვთა გენერატორი და label3-ში გამოაქვს რიცხვები 1-10 დიაპაზონიდან, როგორც ეს Next(1,10) არის მითითებული. while(summe<30) ოპერატორი ამოწმებს ამ რიცხვების ჯამი ხომ არაა მეტი 30-ზე. თუ არაა მეტი, მაშინ სრულდება „ ; “ - ცარიელი ოპერატორი, რომელიც აბრუნებს პროცესს do { } - ციკლში. თუ ჯამი მეტია 30-ზე, მაშინ ციკლი მთავრდება და label4-ში იწერება semme-შედეგი.

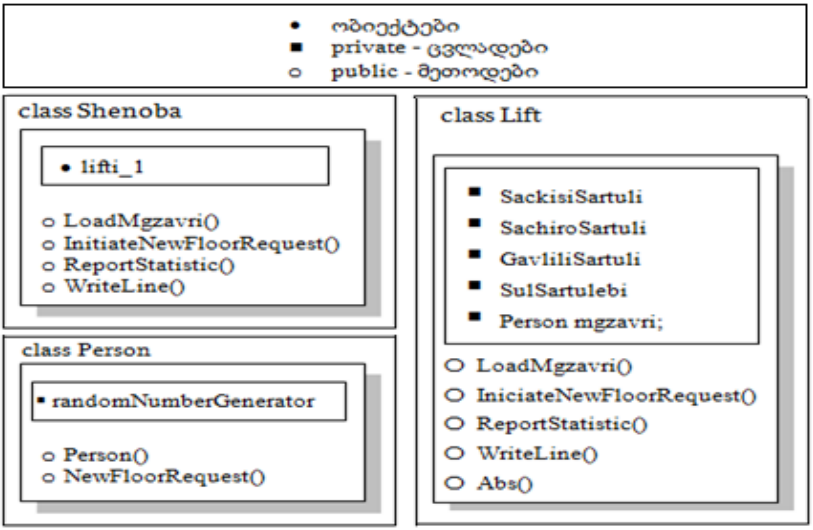
- **ობიექტ-ორიენტირებული მეთოდი (კლასების შექმნა):**

ამოცანა_1: „ავაგოთ ობიექტ-ორიენტირებული პროგრამის კოდი (კლასების და მეთოდების გამოყენებით) მაღლივ შენობაში ლიფტის გადაადგილების მოდელირებისათვის“ [68].

მაგალითად, შენობა N სართულიანია. მას აქვს ლიფტი. პერსონა, რომელიც შედის ლიფტში (ხდება მგზავრი), ირჩევს მისთვის საჭირო სართულს და მიემგზავრება (ზევით ან ქვევით). საჭიროა ამ პროცესის მოდელირება და პროგრამის აგება 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 (გარკვეული პერიოდის) განმავლობაში.

ინკაფსულაციის სქემა მოცემულია 5.5 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.5.5

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

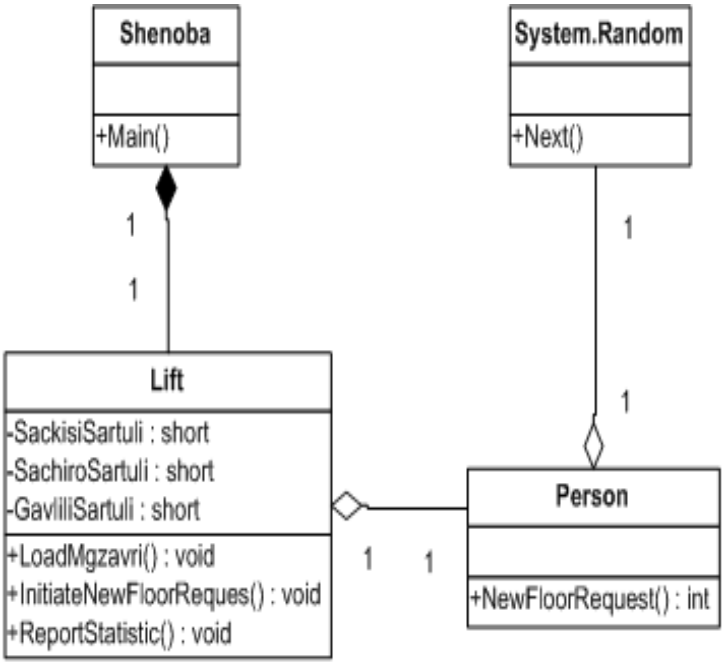
- `lifti_1` ლიფტი მოძრაობს საჭირო სართულისკენ, რაც აისახება კონსოლზე;

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

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

- შუალედური და საბოლოო შედეგები გამოიტანება ეკრანზე.

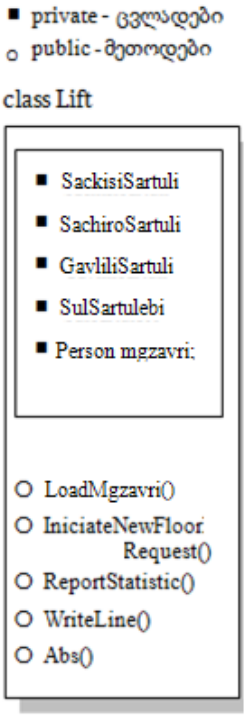
5.6 ნახაზზე მოცემულია ჩვენი მაგალითის კლასებს შორის კავშირების UML დიაგრამა, აგებული Ms Visio პაკეტის გარემოში.



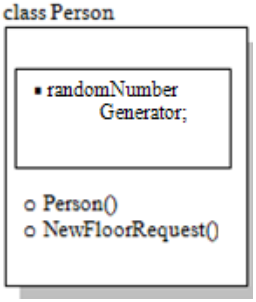
ნახ.5.6. ამოცანის გადაწყვეტის UML-ის კლასთა დიაგრამა

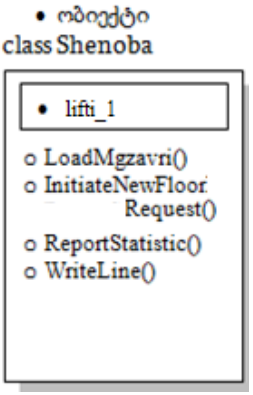
Shenoba და Lift კლასებს შორის აგრეგაციას უწოდებენ კომპოზიციას და იგი შავი რომბიკით გამოისახება. „1“-ები ნიშნავს, რომ 1 შენობაში არის 1 ლიფტი (ჩვენს შემთხვევაში). Lift და Person, აგრეთვე Person და System.Random კლასებს შორის აგრეგატული დამოკიდებულებაა, მაგრამ არა-კომპოზიციური. ის თეთრი რომბიკითაა ნაჩვენები.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

პროგრამული რეალიზაია ნაჩვენებია ლისტინგში:




```
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());

          // Math.Abs(SackisiSartuli - SachiroSartuli);
21      SulSartulebi += GavliliSartuli;
22       SackisiSartuli = SachiroSartuli;
23       i++;
24    }
25
26    public void ReportStatistic()
27    {
28      Console.WriteLine("\n===>>> Sul gavlili
                          sartulebi: " + SulSartulebi);
29        }
30     }
31
32   class Person
33    {
        private System.Random
```






```
34                      randomNumberGenerator;
35    public Person()   // კონსტრუქტორი
36    {
        randomNumberGenerator = new
37                              System.Random();
38      }
39
40    public int NewFloorRequest()
41    {
        // აბრუნებს არჩეულ შემთხვევით
42      //  რიცხვს 1-60 დიაპაზონში
43      return randomNumberGenerator.Next(1,60);
44      }
  }
45
46  class Shenoba   // კლასი შენობა
47  {
48    private static Lift lifti_1;
      private static void Main()
49    {
50      lifti_1 = new Lift();
51      lifti_1.LoadMgzavri();
        Console.WriteLine("      samgzavro sartulebi \n
52      ============================\n");
53      lifti_1.IniciateNewFloorRequest();
54      lifti_1.IniciateNewFloorRequest();
55      lifti_1.IniciateNewFloorRequest();
56      lifti_1.IniciateNewFloorRequest();
57      lifti_1.IniciateNewFloorRequest();
58      lifti_1.ReportStatistic();
59    }
    }
60  }
```

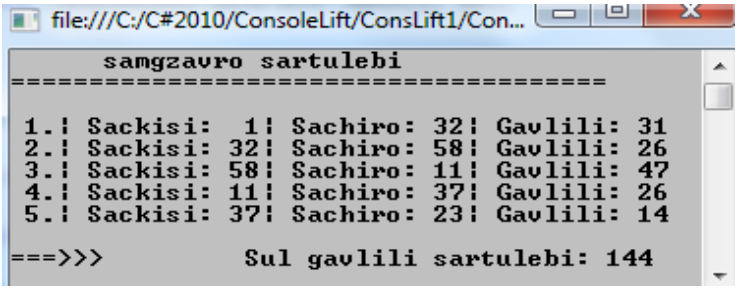
ლისტინგი 5.39: კლასები (მარცხნივ) და კოდები (მარჯვნივ)

5.1 ცხრილში მოცემულია კოდის ზოგიერთი სტრიქონის დანმიშნულება და მათი 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 მიმართებაში
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

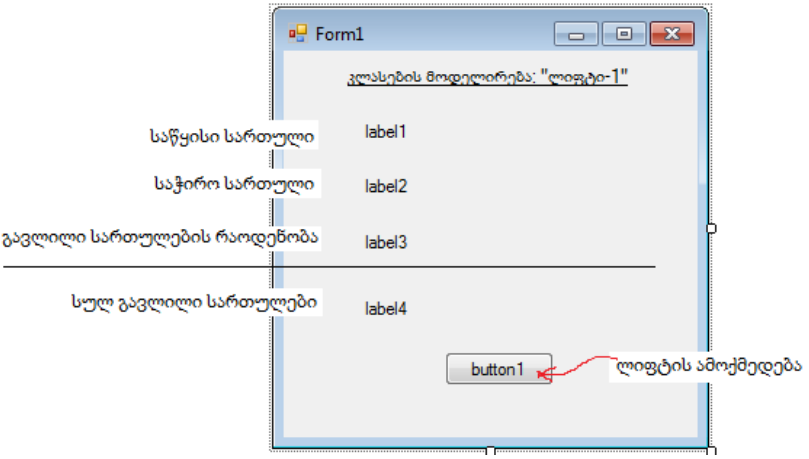
	დასაწყისი, რომელიც გამოიძახება ავტომატურად Person კლასის ობიექტის შექმნის დროს
36	System.Random-კლასის ახალი ობიექტის შექმნა და მისი მინიჭება randomNumberGenerator - ცვლადზე
39	int ტიპის NewFloorRequest() მეთოდის განსაზღვრა. ის Person-კლასის ინტერფეისის ნაწილია
42	პერსონა (ვირტუალური მგზავრი) ლიფტში 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.8.2)
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)

შედეგი ნაჩვენებია 5.7 ნახაზზე.



ნახ.5.7. შედეგი კონსოლის რეჟიმში

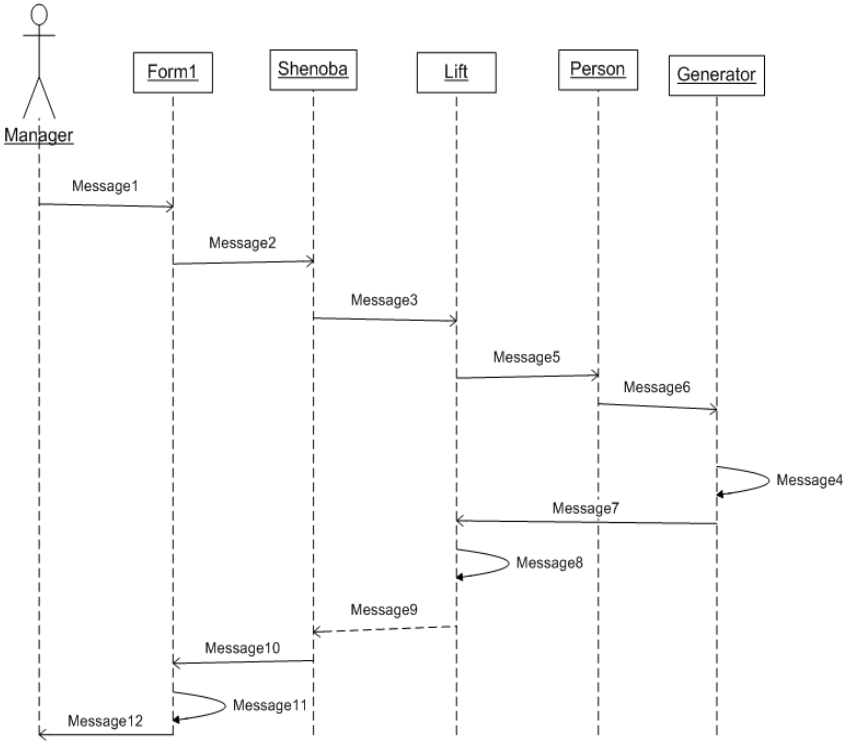
ამოცანა_2: „ავაგოთ ვიზუალური, ობიექტ-ორიენტირებული პროგრამის კოდი ლიფტის მართვის ამოცანისათვის Windows Forms Application რეჟიმში (ნახ.5.8).



ნახ.5.8. საწყისი ფორმის ელემენტების დიზაინი

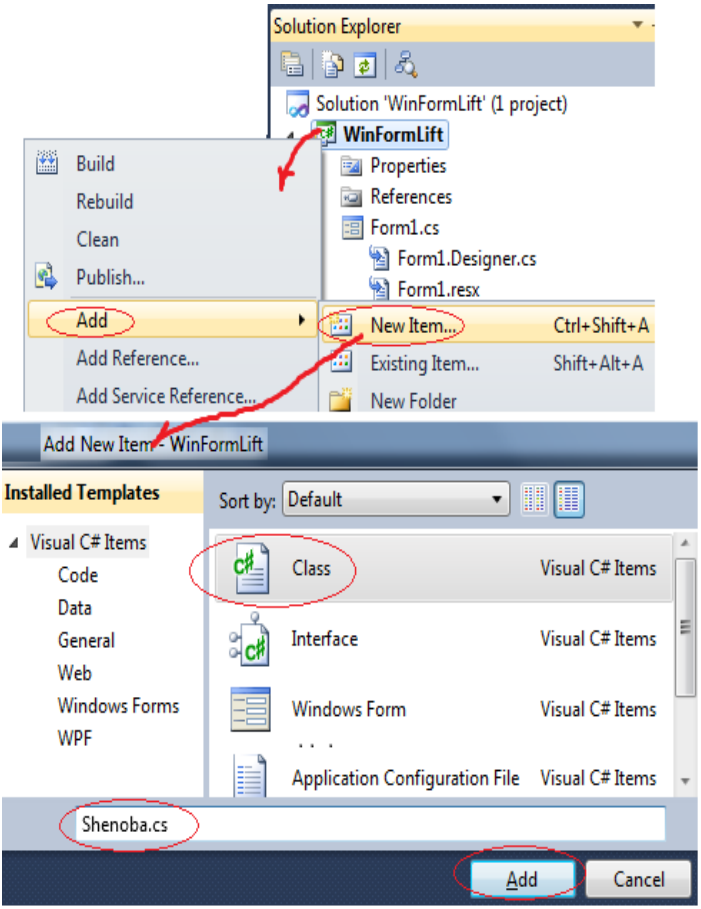
```
// ლისტინგი 5.40 --- Form1  ელემენტებით და  button1-ის კოდი  ---
using System;
using System.Drawing;
using System.Windows.Forms;
namespace WinFormLift
{
   public partial class Form1 : Form
   {
      Shenoba shenoba_1;    // კლასი Shenoba  უნდა შეიქმნას
      public Form1() { InitializeComponent();   }
      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 ფუნქციონალობა ჩვენ ზემოთ აღვწერეთ. ახლა საჭიროა ავაგოთ სცენარი „ლიფტის მუშაობა“, რომლისთვისაც გამოვიყენებთ UML-ის მიმდევრობითობის (Sequence) დიაგრამას (ნახ.5.9).



ნახ.5.9. UML-ის Sequence დიაგრამა (სცენარი)

ახლა შევქმნათ დანარჩენი კლასები და აღვწეროთ მათი ფუნქციონალობები. Solution Explorer-ში დავამატოთ (Add new Items) ახალი კლასები, როგორც ეს 5.10 ნახაზზეა ნაჩვენები.



ნახ.5.10. ახალი კლასის - Shenoba.cs შექმნა

Shenoba კლასის კოდი მოცემულია ლისტინგში.

```
// ლისტინგი_5.41 --- 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 5.42 ლისტინგში.

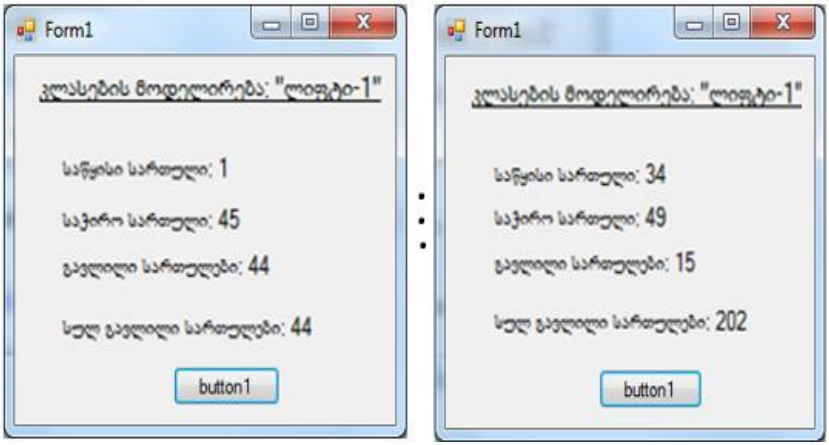
```
// ლისტინგი_5.42 --- კლასი Lift  --------
using System;
using System.Windows.Forms;

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 5.43 ლისტინგში.

```
// ლისტინგი_5.43 --- 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 5.11 ნახაზზე ნაჩვენები შედეგები.



ნახ.5.11. 1-ელი და მე-10 ბიჯის შედეგები

- **პროგრამის ტესტირება :**

მოდულური ტესტირება (Unit testing) პროგრამირების პროცესია, რომლის საშუალებითაც მოწმდება საწყისი კოდის ცალკეული მოდულების კორექტულობა. ასეთი ტესტირების იდეა მდგომარეობს იმაში, რომ ყოველი არატრივიალური ფუნქციის ან მეთოდისათვის დაიწეროს ტესტი [5,69]. ჩვენ დაგვჭირდება *დასატესტი პროგრამა* და *ტესტ-პროგრამა.*

ამოცანა_1: „ავაგოთ C# ტესტ-კოდი Visual Studio.NET გარემოში. Unit testing ტესტირების ტექნოლოგია განვიხილოთ ვირტუალური ობიექტის, მაგალითად, ფინანსური ბანკის მაგალითზე" [5].

- *დასატესტი პროგრამის კოდი* (BankAccount.cs):

```
//-- ლისტინგი_5.43 --- BankAccount.cs --------
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BankAccountNS   // სახელსივრცე -----
{
 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 კოდი* (BankAccountyTests.cs)

```
//-- ლისტინგი_5.44  ----- BankAccountyTests.cs ----
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using BankAccountNS;

namespace UnitTestProject1  // სახელსივრცე ----
{
  [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");
    }
  }
}
```

ტესტირებამ აღმოაჩინა შეცდომები და კოდი წარმატებით ვერ შესრულდა (იხ. დასატესტი კოდი).

თუ მეთოდი წარმატებით ჩაივლიდა, მაშინ მივიღებდით მწვანე ფერის x-სიმბოლოებს.

შემდეგი ეტაპი კოდის გასწორება და ხელახალი ტესტირებაა. დასატესტ პროგრამაში შევცვალოთ სტრიქონში „ + „ ნიშანი „ – „ -ით (იხ. იქვე, დასატესტი კოდი).

შენიშვნა: ტესტირების პროექტის სრული ვერსია მოცემულია მე-[5] ნაშრომში.

ლიტერატურა:

1. ჩოგოვაძე გ., ფრანგიშვილი ა., სურგულაძე გ. მართვის საინფორმაციო სისტემების დაპროგრამების ჰიბრიდული ტექნოლოგიები და მონაცემთა მენეჯმენტი. მონოგრ., სტუ, „ტექნიკური უნივერსიტეტი“, თბ., 2017. -1001 გვ.

2. გოგიჩაიშვილი გ., სურგულაძე გ., შონია ო. დაპროგრამების მეთოდები (C, C++). სახელმძღვ., სტუ, 1997. -275 გვ.

3. Deborah G. Methods and methodology. 2011. https://deborahgabriel.com/2011/05/13/methods-and-methodology/

4. Software development process. Internet resource: https://en.wikipedia.org/wiki/Software_development_process

5. სურგულაძე გ., თურქია ე. პროგრამული სისტემების მენეჯმენტის საფუძვლები. სტუ, თბ., 2016. 350 გვ. http://gtu.ge/book/gia_sueguladze/GiaSurg1_%20ProgSysManag.pdf

6. Dijkstra E. GOTO Statement Considered Harmful. Communications of the ACM, Vol.11, No.3, March 1968, pp.147-148. Assoc.for Computing Machinery, https://files.ifi.uzh.ch/rerg/arvo/courses/-kvse/uebungen/Dijkstra_Goto.pdf

7. Event-driven programming. Internet resource: https://en.wikipedia.org/wiki/Event-driven_programming5

8. Polo-François Poli. Node.js & Event-driven programming. 2013. Internet resource: http://www.baloo.io/blog /2013/11/30/node-event-driven-programming/

9. სურგულაძე გ., დოლიძე ს. მომხმარებლის ინტერფეისის დაპროგრამება (AngularJS, ReactJS). სტუ, „IT კონსალტინგ ცენტრი“. თბ., 106 გვ.

10. Prolog – Logic programming. Internet resource: https://en.wikipedia.org/wiki/Prolog

11. Microsoft Planner. Internet resource: https://en.wikipedia .org/wiki/Microsoft_Planner

12. Visual prolog. Internet resource: https://en.wikipedia .org/wiki/Visual_Prolog

13. Booch G., Jacobson I., rambaugh J. Unified Modeling Language for Object-Oriented Development. Rational Software Corporation, Santa Clara, 1996

14. გოგიჩაიშვილი გ., ბოლხი გ., სურგულაძე გ., პეტრია-შვილი ლ. მართვის ავტომატიზებული სისტემების ობიექტ-ორიენტირებული დაპროექტების და მოდელირების ინსტრუმენ-ტები (MsVisio, WinPepsy, PetNet, CPN). სტუ.. თბ., „ტექნიკური უნივერსიტეტი“. 2013

15. Stoyan H. Programmiermethoden der Kunstlichen Intelligenz. Springer-Verlag, Berlin, B1,2, 1988.

16. Floyd R.W. Paradigms of Programming. Communications of the ACM. #8, vol.22, August ‘79. Stanford University. USA. 1979

17. Bobrow D.G. If Prolog is the answer, what is the question. 5-th Generation of Computer Systems, pp. 138-145, Tokyo, Japan, November ‘84. Inst. for New Generation Computer Technology (ICOT). North-Holland. 1984

18. Shriver B.D. Software paradigms. IEEE Software, 3(1):2, January ‘86. 1986

19. Wegner P. Concepts and paradigms of object-oriented programming. {OOPS messenger}, 1(1): 7-87, August ‘90. 1990

20. Budd T.A. Multy-Paradigm Programming in LEDA. Addison-Wesley, Reading, Massachusets. 1995

21. სურგულაძე გ. სტრუქტურული დაპროგრამების მეთო-დი. სტუ, თბ., -54 გვ. 2005.

22. ბოტჰე კ., სურგულაძე გია, დოლიძე თ., შონია ო., სურგულაძე გიორგი. თანამედროვე პროგრამული პლატფორმები და ენები (WindowsNT, Unix, Linux, C++, Java, XML). სტუ, „ტექნიკური უნივერსიტეტი”, თბ., -250 გვ. 2003.

23. სურგულაძე გ. ობიექტ-ორიენტირებული დაპროგრამების მეთოდი. სტუ, თბ., -100 გვ. 2007

24. ფრანგიშვილი ა., წვერაიძე ზ,, ნამიჩეიშვილი ო. დაპროგრამება HASKELL ენაზე (მულტიმედიური პრეზენტაცია). https://gtu.ge/View/index.html#http://gtu.ge/book/preze-ntacia_daprogrameba_HASKELL_enaze.pdf

25. სურგულაძე გ., კიკნაძე მ. დაპროგრამების მეთოდები ინტერნეტისთვის (Java-2 და XML ენების ბაზაზე). თბ., „ტექნიკური უნივერსიტეტი“, 2006. http://www.gtu.edu.ge/ katedrebi /kat94/pdf/-Java2+XML.pdf

26. გაჩეჩილაძე ლ., ნონიკაშვილი ლ. მრავალნაკადური დაპროგრამება Java ენაზე. სტუ. თბ., 2016. 164 გვ.

27. გაჩეჩილაძე ლ. დაპროგრამების ენა Python. სტუ. „ტექნიკური უნივერსიტეტი“, თბ., 2017. -148 გვ.

28. სუხიაშვიული თ. სურგულაძე გ. ორგანიზაციულ-ადმინისტრაციული მართვის განაწილებული სისტემების არქი-ტექტურა. სამეცნ. ჟურნ.„ინტელექტი“, N2 (18) , 2006, გვ. 43-47.

29. სუხიაშვილი თ., განაწილებული სისტემების მოდელი-რება პროცესების თვალთახედვით. სტუ-ს შრ., N 4(437), 2006. გვ. 206-207

30. OMG. Unified Modeling Language. Version 2.5.1 (OMG UML). 2017. -796 p. https://www.omg.org/spec/UML/2.5.1/pdf

31. About the Unified Modeling Language Specification Version 2.5.1. https://www.omg.org/spec/UML/2.5.1# document-metadata

32. Бек К. Шаблоны реализации корпоративных приложений. Экстремальное программирование: Пер. с англ. М.: Вильямс. 2008

33. UML Class Diagram Relationships Explained with Examples. Int.res: https://creately.com/blog/diagrams/class-diagram-relationships/

34. ვიზუალური პროგრამირების ენა. VPL-programming language. https://en.wikipedia.org/wiki/Visual_programming_language

35. Visual Programming Language VPL. Internet resourcr: https://www.techopedia.com/definition/22855/visual-programming-language-vpl

36. Software Development Life Cycle. Internet resource; https://en.wikipedia.org/wiki/Software_development_process

37. code-and-fix model: in Development Life Cycle Models. Internet resource; http://zone.ni.com/reference/en-XX/help/371361R-01/lvdevconcepts/-lifec¬ycle_models/

38. Principles behind the Agile Manifesto. Internet resource; http://agilemanifes¬to.org/¬princip-les.html.

39. Beck K. et al. Manifesto for Agile Software Development. 2001. Internet resource; https://agilemanifesto.org/

40. Lean Software Development - in What is Agile Kanban Methodology ? https://www.inflectra.com/methodologies/kanban.aspx

41. ჩოგოვაძე გ., გოგიჩაიშვილი გ., სურგულაძე გ., შეროზია თ., შონია ო. მართვის ავტომატიზებული სისტემების დაპროექტება და აგება (თეორიულ-პრაქტიკული ინფორმატიკა). თბ., სტუ, 2001.

42. თურქია ე. ბიზნეს–პროექტების მართვის ტექნოლოგიური პროცესების ავტომატიზაცია. სტუ. „ტექნიკური უნივერსიტეტი". თბ., 2010

43. What is Agile Software Development? http://www.inflectra.com/Methodologies/AgileDevelopment .aspx#Scrum

44. Борута Я. Методология Agile. Матерь драконов или всех гибких методологий. w-Blog. 2017. https://worksection. com/blog/-agile.html

45. Ambler S.W. The Object Primer: Agile Model-Driven Development With Uml 2.0. Cambridge University Press 2004-05-27. 2001. 572 p. https://www.researchgate.net/publication/2356 16285_The-_object_primer_agile_modeling-driven_development_ with_UML_20

46. Амблер С. Гибкие технологии: экстремальное программирование и унифицированный процесс разработки. Библиотека программиста. Спб.: Питер.

47. Rumpe B. Agile Modellierung mit UML. Berlin, „Springer“. 2-te Auflage. 2012

48. Resources Scrum. Internet resource: 2019. https://www.scrum.org/resources

49. Six Sigma. Internet resource: https://en.wikipedia.org/ wiki /Six_Sigma

50. Berchez J.P., Kapp U. Kriterien für eine Entscheidung für Scrum oder Kanban. IBM, Heise online. 2010. https://www.heise.de/-developer/artikel/Kriterien-fuer-eine-Entscheidung-fuer-Scrum-oder-Kanban-1071172.html?seite=all

51. Anderson, David J. (April 2010). Kanban: Successful Evolutionary Change for Your Technology Business. Blue Hole Press. ISBN 978-0-9845214-0-1.

52. Kanban (development). Internet resource: https://de.wikipedia.org/wiki/Kanban_(Softwareentwicklung)

53. Deming E.W. ციკლი (PDSA Plan-Do-Study-Act) „დაგეგმე, გააკეთე, შეისწავლე, იმოქმედე". Internet resource: https://de.-wikipedia.org/wiki/Wil-liam_Edwards_Deming.

54. PDCA (plan–do–check–act). Internet resource: https://en.-wikipedia.org/wiki/PDCA

55. Пименов М. Разбираемся в Scrum и Kanban. Блог Нетологии. 2016. Internet resource: https://netology.ru/blog/ scrum-kanban

56. Зыкова С. Agile, scrum, kanban: в чем разница и для чего использовать? Блог Нетологии. 2017. Internet resource: https://rb.ru/story/agile-scrum-kanban/

57. სურგულაძე გ., გულიტაშვილი მ., კაკულია ი., ჩერქეზიშვილი გ., ჯავახიშვილი ი. პროგრამული სისტემების სასიცოცხლო ციკლის პროცესის მოდელირება უნივერსალური და ექსტრემალური პროგრამირების პრინციპების კომპრომისული გადაწყვეტით. სტუ-ს შრ.კრ. „მას", N1(8), თბ., 2010. გვ. 63-70

58. Schindler M. Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P. Aachener Informatik-Berichte, Germany. Software Engineering Band 11. Hrsg.: Prof. Dr. rer. nat. Bernhard Rumpe. 2012. -363 p.

59. სურგულაძე გ., პეტრიაშვილი ლ. ვიზუალური დაპროგრამება C# ენის ბაზაზე ინფორმაციულ სისტემებისათვის (Visual Studio.NET 2019 პლატფორმაზე). სტუ, „IT კონსალტინგ ცენტრი". თბ., 200 გვ.

60. Spaghetti_code. Internet resource: https://en.wikipedia.org/wiki/Spaghetti_code

61. Kumar A. What Stats & Surveys are Saying about Top Programming Languages in 2020. Internet resource: 2019. https://codinginfinite.com/top-programming-languages-2020-stats-surveys/

62. C Programming Tutorial. Internet resource: https://www.learnbix.com/cprogramming/learn-c

63. Керниган Б., Ритчи Д. Фьюэр А. Язык программирования С. Задачи по языку С. -М., Финансы и статистика, 1985. -279 с.

64. Керниган Б., Ритчи Д. Язык программирования С. -М., Вильямс, 2015. -304 с.

65. დაპროგრამების საფუძვლები (C-ენის ბაზაზე). სტუ, „ტექნიკური უნივერსიტეტი", თბ., 2005. ნაწ.I-II. ელ-სახელმღვანელო http://www.gtu.edu.ge/katedrebi/kat94/pdf/C-1.pdf და http://www.gtu.edu.ge/katedrebi/kat94/pdf/C-2.pdf

66. სურგულაძე გ. ობიექტ-ორიენტირებული დაპროგრამების ენა C++. სტუ. „ტექნიკური უნივერსიტეტი". თბ., 2005. ელ-სახ.http://www.gtu.edu.ge/katedrebi/kat94/pdf/C++.pdf

67. სურგულაძე გ., კიკნაძე მ. დაპროგრამების მეთოდები ინტერნეტისთვის (Java-2 და XML ენების ბაზაზე). სტუ. თბ., 2006. ინტერნეტ რესურსი: http://www.gtu.edu.ge/katedrebi/kat94/pdf/Java2-+XML.pdf

68. სურგულაძე გ. ვიზუალური დაპროგრამება C#_2010 ენის ბაზაზე. სახელმძღვანელო. სტუ. „ტექნიკური უნივერსიტეტი“, თბ., 2011. -445 გვ.

69. სურგულაძე გ., გულიტაშვილი მ., კივილაძე ნ. Web-სისტემების ტესტირება, ვალიდაცია და ვერიფიკაცია. მონოგრ. სტუ. „IT-კონსალტინგის ცენტრი“. თბ., 2015.

Gia Surguladze

Computer Programming Methods and Methodologies ***(SP, OOP, VP, Agile, UML)***

ISBN 978-9941-8-1900-1

"IT Consulting Center" of GTU

Tbilisi, Georgia - 2019

გადაეცა წარმოებას 5..12.2019 წ. ხელმოწერილია დასაბეჭდად 16.12.2019 წ. ოფსეტური ქაღალდის ზომა 60X84 1/16. პირობითი ნაბეჭდი თაბახი 7. ტირაჟი 100 ეგზ.

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

ISBN 978-9941-8-1900-1