კომპიუტერული პროგრამირების სამყარო

ნაწილი I

ფ ი ლ ო ს ო ფ ი ა

გიორგი ბაწაშვილი

მეორე რედაქტირებული გამოცემა

WWW.G3B.GE

2006

ეს არის წიგნის

„კომპიუტერული პროგრამირების სამყარო“

პირველი ნაწილის რედაქტირებული ვარიანტი.

პირველი ნაწილი გამოვიდა 2006 წელს ელექტრონული სახით.

წიგნის სრული ვერსია - 1-ელი, მე-2 და მე-3 ნაწილები, არის მხოლოდ ბეჭდვითი სახით

დაწვრილებით ამ წიგნის შესახებ იხილეთ საიტზე **http://www.g3b.ge**

I ნაწილის შინაარსი

შ ე ს ა ვ ა ლ ი

წინასიტყვაობა. კომპიუტერი, თავისი შესაძლებლობებითა და სარგებლიანობით ფართოდ შეიჭრა ადამიანის ცხოვრებაში, მისი მოღვაწეობისა და საქმიანობის ყველა სფეროში, მიუხედავად იმისა, რომ ის თავიდან შეიქმნა მხოლოდ სამეცნიერო და სამხედრო მიზნებისათვის, მხოლოდ, როგორც გამომთვლელი მოწყობილობა.

ალბათ ძალიან შორს არ არის ის დრო, როდესაც ყველა საზოგადოებრივი მომსახურების სისტემა გადავა სრულ ციფრულ მართვებზე, რასაც რა თქმა უნდა ექნება თავისი უარყოფითი მხარეებიც, მაგრამ დადებითი უფრო მეტი. უარყოფითში უმთავრესად ვგულისხმობ გარდაუვალ ხარვეზებს, რომელიც ყოველთვის თან ახლავს პროგრამულ სისტემებს.

კომპიუტერმა ისევე, როგორც საკომუნიკაციო სისტემებმა შეცვალა ადამიანთა ცხოვრების წესი, მათი აზროვნებისა და ურთიერთობის სტილი, ეს კი ჯაჭვურად აისახება ყველაფერ იმაში, რასაც კი ადამიანი ეხება.

სურ. 1.1

ის რაც რამოდენიმე ათეული წლის წინ იყო ფანტასტიკის საგანი დღეს რეალობაა და ამაში გასაკვირი არაფერია, რადგან ჩვენ ადამიანები ვხედავთ თუ როგორ და რა ტემპებით ვითარდება მაღალი ტექნოლოგიები და ეს ყველაფერი ხდება ჩვენს თვალწინ და წარმოადგენენ ჩვენთვის სრულიად გამჭვირვალე პროცესებს.

კომპიუტერმა დიდი ხანია გააბათილა ის სტერეოტიპი, რომლის მიხედვითაც იგი თავისი სახელიდან გამომდინარე (Computer - გამომთვლელი) წარმოადგენდა მხოლოდ და მხოლოდ გამომთვლელ მანქანას და იარაღს სახვადასხვა სამეცნიერო თუ საყოფაცხოვრებო დარგებში არსებული პრობლემების გადაწყვეტისათვის და თუ როგორ მოახერხა მან ეს და რამ მისცა მას ამის საშუალება, წიგნის პირველ ნაწილში სწორედ ამ საკითხით მინდა დავაინტერესო მკითხველი, რომელიც შეიძლება იყოს, როგორც პროფესიონალი ამ დარგში ასევე კომპიუტერთან ნაკლებად დაახლოებული პირი.

მიუხედავად ამ ყველაფრისა დღეისათვის საზოგადოების დიდი ნაწილისათვის მაინც უცნობია, რომ კომპიუტერის ფუნქციონალური სისტემები და მასთან ურთიერთობის მქონე ადამიანები იყოფიან რამოდენიმე ჯგუფად. ასეთი ჯგუფები თავის მიმართულებებით და ქვეჯგუფები დროთა განმავლობაში მკვეთრად დაშორდებიან ერთმანეთს, რისი ტენდენციაც დღეს უკვე აშკარაა.

დღეისათვის სახელდების მიზნით შეიძლება გამოვკვეთოთ სამი ჯგუფი:

1. პროგრამისტები (Programmers)
2. ადმინისტრატორები (Administrators)
3. მომხმარებლები (Users)

რაც უფრო მეტია ჯგუფის რიგითი ნომერი მით ნაკლებად არის ის დაახლოებული კომპიუტერის ფუნქციონალურ-კოდურ ნაწილთან. დაჯგუფება დიდადაა განზოგადებული და იგი გამომდინარეობს ამ საკითხისადმი ზოგადი მიდგომიდან.

ამ ჯგუფებს შორის მკვეთრი ზღვარის გავლება ჯერ კიდევ ნაადრევია, თუმცა როგორც ზემოთ აღვნიშნე ტენდენცია უკვე არსებობს და ამაზე მიუთითებს ისტორიაც.

CSIRAC Mark1 1949-1964

ერთ-ერთი პირველი კომპიუტერი

სურ. 1.2

პირველად ყველაფერს ამას ითავსებდა ერთი ადამიანი. პროგრამისტი იყო პროგრამისტიც და მომხმარებელიც, კომპიუტერის მოხმარება იცოდა მხოლოდ იმან, ვინც აპროგრამებდა მას.

შემდეგ გამოიკვეთა მომხმარებელი პროგრამისტისაგან (50-იანი წლები) და ფუნქციები გადანაწილდა, პროგრამისტები მუშაობდნენ მომხმარებლებისათვის.

ბოლო ათწლეულებში კი თვით პროგრამირების სამყაროც დაიყო, მეცნიერებად, ხელოვნებად(გრაფიკა, მუსიკა), გამოყენებითი ტენოლოგიურ მიმართულებებად, დაცვა-ჰაკინგი(Hack), ქსელები და სხვა და ასეთი დანაწევრება დღემდე გრძელდება.

მოკლედ აღვწერ ამ სამ ჯგუფს თავისი დანიშნულებისამებრ.

პროგრამისტი. პიროვნება რომელიც ქნის კომპიუტერის Software(პროგრამული ნაწილი) ნაწილს. ზოგადად მას შეუძლია ცარიელ კომპიუტერს, როგორც მხოლოდ ელექტრულ-ციფრულ მოწყობილობას მისცეს ფუნქცია – აქციოს იგი მაღალი ტექნოლოგიის ნაწილად, მასში კოდების შეყვანის საფუძველზე. პროგრამისტი ქმნის პროგრამებს, და პროგრამისტივე ქმნის იმ პროგრამულ გარემოს, რაშიც უნდა შეიქმნას ეს პროგრამები. პროგრამისტი ქმნის ე.წ. ოპერაციულ სისტემებს(Operating System - OS) და აპლიკაციებს(Application - სამომხმარებლო პროგრამა, დანართი ოპერაციული სისტემისათვის) ამ სისტემებისათვის.

სწორედ ამ ყველაფრის განსახორციელებლად პროგრამისტს უწევს დიდად თუ მცირედ შეხება ისეთ ფუნდამენტალურ მეცნიერებებთან, როგორიცაა მათემატიკა და ფიზიკა. ასევე მასვე უწევს თავისი პრაქტიკის განმავლობაში შეიმუშავოს საკუთარი მეთოდები და კონკრეტული ტექნიკური მიდგომები, რათა მიაღწიოს დასახულ მიზნებს. და ბოლოს პროგრამისტი თავის ცხოვრების ერთ ნაწილში იქმნის თავისებური ფორმის მიკრო სამყაროს, რომელშიც რეალურად არსებობს ყველა ის ზემოთ ჩამოთვლილი მოვლენა.

დღეისათვის ასე გაზრდილმა მოთხოვნებმა პროგრამულ უზრუნველყოფაზე გამოიწვია პროგრამისტის სტატუსის დანაწევრება და შედეგად მივიღეთ ცნობილი ტერმინები: პროგრამისტი-დეველოპერი, პროგრამისტი-ინჟინერი და პროგრამისტი-არქიტექტორი.

შევეცდები მოკლედ აღვწერო თითოეული მათგანი, გამომდინარე ცნობილი ტერმინების ზოგადი განმარტებებიდან, ამის უფლებას ნამდვლად ვიტოვებ, რადგან პროგრამირების სამყაროს ბევრი ფორმალური მხარე ჯერ კიდევ ჩამოუყალიბებელია:

პროგრამისტი-დეველოპერი: პროგრამისტი რომელიც ანვითარებს მოცემულ პროგრამულ გარემოს, IDE(Integrated Development Environment – განვითარების ინტეგრირებული გარემო) სისტემის საფუძველზე, ქმნის რა სამომხმარებლო, სამეცნიერო ან გასართობი ტიპის აპლიკაციებს. დეველოპერი – ფაქტიურად არის პროგრამისტის საქმიანობის მახასიათებელი, ზოგადი ტერმინი.

პროგრამისტი-ინჟინერი: პროგრამისტი, რომელიც აპროექტებს და ქმნის ე.წ. Framework(გარსი, კარკასი, ღერძი)-ებს, ანუ წინასწარ გამზადებული პროგრამული მოდულების ნაკრებს ანუ მზა ბიბლიოთეკებს, რაც უადვილებს სხვა პროგრამისტებს შესაბამისი ტიპის პროექტებზე მუშაობას (მაგალითად: ალექსი სტეფანოვი – STL(Standart Temlate Library - სტანდარტული შაბლონების ბიბლიოთეკა)-ის ავტორი).

პროგრამისტი-არქიტექტორი: პროგრამისტი, რომელიც ქმნის ახალ პროგრამირების ენებს, ახალ კონცებციებს პროგრამირების სამყაროში და საერთოდ საფუძველს უყრის საბაზისო იდეებს პროგრამულ უზრუნველყოფის წარმოების სფეროში (მაგალითად ანდერს ჰეილსბერგი(Anders Hejlsberg) წამყვანი პროგრამისტი-არქიტექტორი კომპანია მაიკროსოფტში, Turbo Pascal, Delphi და C# - ის ავტორი).

როგორც წესი პროგრამისტს თავის პრაქტიკის განმავლობაში მეტნაკლებად უხდება ყველა ამ ეტაპების გავლა, თუმცა რა თქმა უნდა ის ძირითად დროს ახმარს მის საკუთარ მიმართულებას. მაგალითად, თუ პროგრამისტი-დეველოპერი ქმნის პროექტს რომელიც განკუთვნილია საბანკო გათვლებისათვის, მას ასევე უწევს გარკვეული რაოდენობის მცირე მოცულობის Framework-ების შექმნა, რომლებიც რა თქმა უნდა ორიენტირებულია ამ პროექტზე და აადვილებს მასზე მუშაობას.

ადმინისტრატორი. ეს არის პიროვნება, რომელსაც ძირითადად შეხება აქვს უკვე არსებულ პროგრამულ სისტემებთან, იცის მათი ავ-კარგი, მათი სუსტი და ძლიერი მხარეები, შეუძლია რამოდენიმე ალტერნატივიდან საუკეთესოს ამორჩევა, შეუძლია არსებული სისტემების თვისებათა კარგად ცოდნის ბაზაზე მათი გამართვა, გადაკეთება გარკვეულ დონეზე, ამ სისტემების კომბინირებით ახლის აწყობა. ადმინისტრატორისავე პრეროგატივაა სისტემების ურთიერთ შეთავსება, რათა ისინი მუშაობდნენ გამართულად მოცემულ გარემოში, მოცემულ ამოცანებისათვის.

ადმინისტრატორების პრეროგატივაა ასევე სისტემის ხარვეზთა დაჭერა და მათი სასარგებლოდ გამოყენება, აქედან გამომდინარე პროგრამული სისტემების შეფასება.

რაც დრო გავა მით უფრო გაიზრდება ადმინისტრატორის ფუნქციები, რადგან გაიზრდება პროგრამულ პროდუქტთა რაოდენობა და მათი დანიშნულება. ეს უკვე არის დამოუკიდებელი სფერო. ამ თვალსაზრისით პროგრამისტი შეგვიძლია შევადაროთ წიგნის ავტორს, ხოლო ადმინისტრატორი კი ბიბლიოთეკის მფლობელს, ანალიტიკოსს, რომელმაც გაცილებით მეტი იცის წიგნების შესახებ ვიდრე ცალკეულმა ავტორმა.

მომხმარებელი. პიროვნება რომლისთვისაც მზადდება ესა თუ ის პროგრამული პროდუქტი, პიროვნება რომელიც განსაზღვრავს ამა თუ იმ პროგრამული სისტემის სახეს და რომელიც განსაზღვრავს ასევე ამ სისტემების პროგრესსაც.

თავისთავად ხდება ის რომ კომპიუტერთან ურთიერთობის ასეთი დაჯგუფების არსებობის არცოდნა იწვევს სხვადასხვა სახის წარმოდგენების ჩამოყალიბებას ადამიანებში. ბოლოს იქმნება ამ წარმოდგენათა მახინჯი ფორმები და შესაბამისად მცდარი დამოკიდებულება პროგრამირების, როგორც ერთდროულად ხელოვნების, ხელობისა და მეცნიერების დარგის მიმართ, ეს კი თავის მხრივ ჯაჭვურად აისახება სოციალურ და ეკონომიურ სფეროებზე, შეიძლება შეუმჩნევლად, მაგრამ მნიშვნელოვნად.

წიგნის შესახებ

კომპიუტერული პროგრამირება პროფესიის ის იშვიათი შემთხვევაა როცა საქმიანობის სფერო გვევლინება ერთდროულად, როგორც ხელობის(პროფესიის) ასევე ხელოვნებისა და მეცნიერებისათვის დამახასიათებელი ნიშნებით.

ხელოვნება უფრო სუბიექტური ხასიათისაა, ხელობა და პროფესია კი სუბიეტურიც და ობიექტურიც.

შევეცადე რომ ჩემი წიგნის პირველ ნაწილში სწორედ ამ მომენტებზე გამემახვილებინა ყურადღება, რაც საშუალებას მომცემდა, რომ გამევლო პარალელები ჩვენს ყოველდღიურ

ცხოვრებასთან და ადამიანის მოღვაწეობის სხვადასხვა სფეროებთან, როგორც ამას აკეთებს ყველა თავისი საქმის პროფესიონალი.

ამ წიგნში არ იქნება განხილული კონკრეტული პროგრამირების ენები და იგი არ არის განკუთვნილი საერთოდ პროგრამირების შესასწავლად, რადგან ამისათვის უამრავი წყარო და საშუალება არსებობს. არამედ წიგნის მიზანია უფრო მნიშვნელოვანი რამ - პროგრამირების, როგორც ფენომენის გარშემო სამყაროს დათვალიერება და ანალიზი, სოგიერთი საკვანძო საკითხების განხილვა და შესაბამისი პრობლემებიდან გამოსავლის მოძებნის მეთოდების შემოთავაზება.

კარგად მოგეხსენებათ რომ ყველა საგანს, ყველა მოვლენას ამ ქვეყნად აქვს თავისი ფილოსოფია და ფილოსოფია კი ყოველთვის არ არის აბსტრაქტული და ხშირ შემთხვევაში ის უძლიერესი იარაღია ადამიანისათვის. ამ შემთხვევაშიც წიგნის პირველი ნაწილის წამყვან სტილს წარმოადგენს თეორიული მიდგომა.

რაც შეეხება კონკრეტულ პროგრამირებას, მისი შესწავლის ერთად ერთი გზა ეს არის კონკრეტული ამოცანების კეთება, კონკრეტულ პროექტებზე მუშაობა, კონკრეტული მიზნების დასახვა და თან რაც შეიძლება ხშირად და მრავალმხრივად. იმიტომ, რომ ეს არის პრაქტიკა და პროგრამისტის პროფესიონალიზმი მდგომარეობს მის პრაქტიკულ გამოცდილებაში.

“ერთად-ერთი წესი პროგრამირების ენის შესწავლისა არის კონკრეტული პროგრამების წერა”

ბრაინ კერნიგანი(Brian Kernigan)

როდესაც ამ წიგნის შედგენის სტილზე ვფიქრობდი არჩევანი მქონდა ორ ვარიანტს შორის, ერთი, რაც შეიძლება ბევრი მაგალითები და ამოცანები, მათი განხილვა და ანალიზი, რაც რა თქმა უნდა უზრუნველყოფდა წიგნის პრაქტიკულობას და მოცულობას, მეორე კი ლაკონურობა და აქსიომატურ-პრინციპული მიდგომა.

ბოლოს უპირატესობა მივანიჭე მეორე ვარიანტს, როდესაც ძირითად პრინციპად არჩეულია საკითხის ლაკონური და აქსიომატური სტილი, რადგან ვთვლი რომ წიგნი არ უნდა იყოს ერთჯერადი საკითხავი.

საქმე იმაშია რომ არსებობს პრაქტიკაში შემთხვევები, როდესაც მაგალითად სტუდენტი ხსნის ას ამოცანას და ვეღარ ხსნის ასმეერთეს. რატომ? იმიტომ რომ პირველ ას ამოცანას აქვს ამოხსნის საერთო მეთოდი, ხოლო ასმეერთე უკვე განსხვავებულ პრინციპზეა აგებული. ამიტომ აქცენტი უნდა მივმართოთ მეთოდების გაგებაზე, ხოლო მათი გათავისება დროისა და პრაქტიკის საქმეა, რადგან თვით ეს პრინციპები მოდის პრაქტიკიდან, დროიდან და პროგრამისტების გამოცდილებიდან. რაც ადრე მიიღებს ამ ინფორმაციას პროგრამისტი, მით უფრო ადრე მოხდება მისი გათავისება.

დასწავლა, გაგება და გათავისება ცოდნის მიღების ეტაპებია და ერთმანეთისგან განსხვავდება, ამათგან პირველი ორს ადამიანის უნარი და ნებისყოფა განსაზღვრავს, ხოლო მესამე აბსოლიტურად დამოკიდებულია დროზე. გათავისება ნიშნავს იმას, რომ ადამიანი უკვე აზროვნებს მიღებული ცოდნიდან გამომდინარე - მიღებული ცოდნა არის მისი შემადგენელი ნაწილი.

წიგნი შესგება სამი ნაწილისაგან.

პირველში განიხილება პროგრამირების, როგორც ხელოვნების, პროფესიის და მეცნიერების თვისობრივი მახასიათებლები.

მეორე ნაწილში მოყვანილია კონკრეტული წესები და მიდგომები რეალურ პროექტებთან მუშაობისათვის, ძირითადი მიზანია პროგრამისტის ინფორმირება იმ აუცილებელი საბაზისო ცოდნით, რის გარეშეც წარმოუდგენელია დღევანდელი პროგრამული მოთხოვნების დაკმაყოფილება, ყურადღება გამახვილებულია პროექტებში ხარვეზების(Bugs) არსებობასა და მათი აღმოფხვრის ხერხებზე.

ხოლო მესამე ნაწილში კი წარმოდგენილია მნიშვნელოვანი დანართები, მათ შორის არის ჩემს მიერ ადრე გამოქვეყნებული სტატიების რედაქტირებული ვარიანტები, რომლებიც თავისი შინაარსით ახლოს დგანან ამ წიგნის შინაარსთან. ასევე განხილულია ერთი პატარა პროექტი Visual C# ენის ბაზაზე.

წიგნს რა თქმა უნდა არა აქვს სისრულის პრეტენზია. მისი მიზანია ის, რომ დააინტერესოს მკითხველი და სხვა მხრიდან შეახედოს პროგრამირებას და საერთოდ ციფრულ სამყაროს.

ამ წიგნის წაკითხვით:

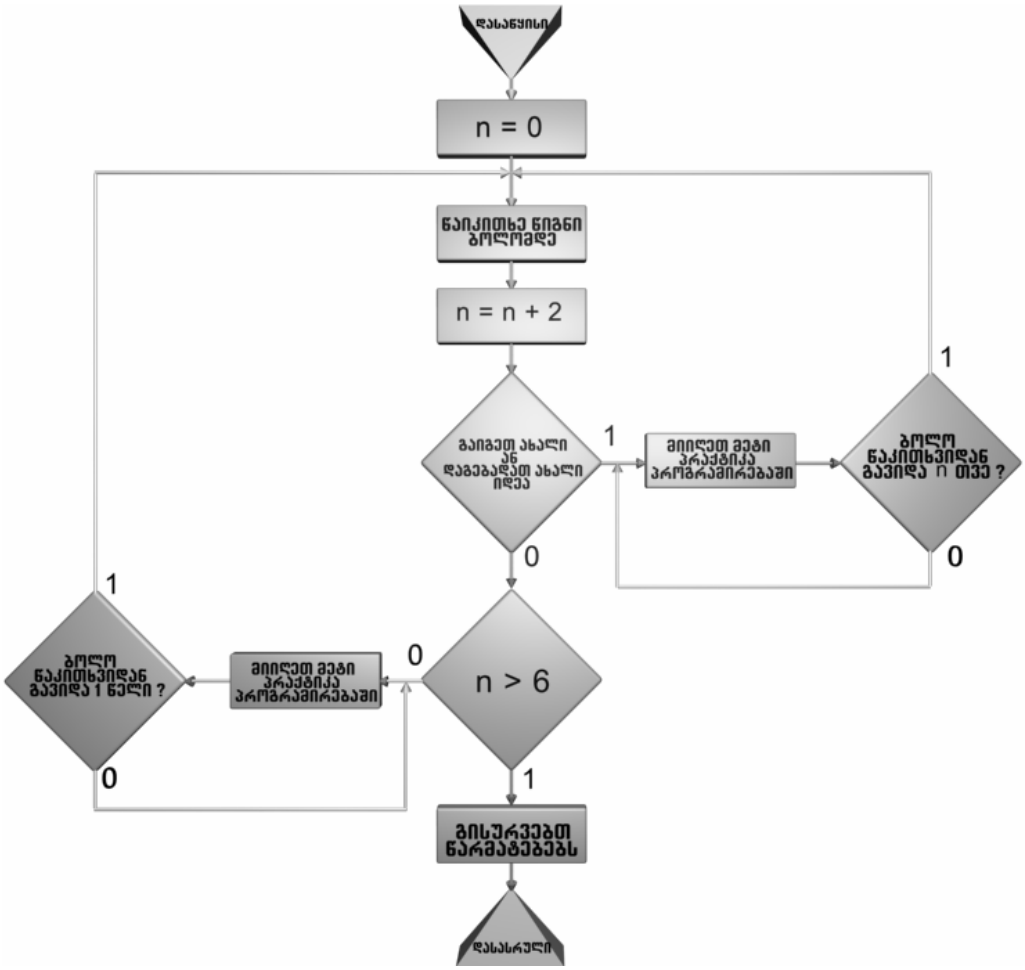
a. გაიგებთ ახალს
b. ისწავლით ახალს
g. ითქმება თქვენი სათქმელი

ამ წიგნის წაკითხვით:

1. კომპიუტერულ სამყაროსთან დაახლოებული ადამიანებისთვის შესრულდება ა პუნქტი.
2. დამწყები პროგრამისტებისათვის შესრულდება ბ პუნქტი.
3. პროფესიონალთათვის გ პუნქტი.

როგორ წავიკითხოთ წიგნი?

იმდენად, რამდენადაც ეს არის მეთოდური ხასიათის წიგნი, მისი წაკითხვის სტილი რამდენადმე განსხვავდება კონკრეტული ტექნიკური საკითხებისადმი მიძღვნილი ლიტერატურის(მაგ. კონკრეტული პროგრამირების ენა) წაკითხვის სტილისაგან. ამიტომ პროგრამისტებს გთავაზობთ წიგნის წაკითხვის შემდეგ ალგორითმს:



აქ მნიშვნელოვანია ძირითადი პირობა: "გაიგეთ ახალი ან დაგებადათ ახალი იდეა", იგულისმება ის რომ თუ წიგნის წაკითხვისას მასში აღმოაჩინეთ თუნდაც ერთი მაინც თქვენთვის მნიშვნელოვანი წინადადება ან წაკითხული წინადადების საფუძველზე დაგებადათ ახალი აზრი, ესე იგი ეს წიგნი წარმოადგენს გარკვეულ ფასეულობას თქვენთვის. ამ ალგორითმში ასევე განიხილება უკიდურესი შემთხვევებიც, ანუ ის რომ წიგნი მკითხველისთვის შეიძლება არ წარმოადგენდეს არანაირ სიახლეს ორი მიზეზით: ერთი, მან აბსოლიტურად ყველაფერი იცის ის რაც წერია წიგნში და აქვს გათავისებული ცოდნის სახით ან მეორე, მისთვის ჯერ სრულიად გაუგებარია მოყვანილი თემების შინაარსი. ამიტომ ალგორითმი ყოველი შემთხვევისათვის ითვალისწინებს წიგნის მინიმუმ ოთხჯერ წაკითხვას სამი წლის განმავლობაში.

რა თქმა უნდა ეს ალგორითმი უპირველეს ყოვლისა წარმოდგენილია იმისათვის, რომ უფრო ცხადი გახდეს წიგნის დანიშნულება.

ხ ე ლ ო ვ ნ ე ბ ა

```
void main()
{
  char  s;
  cout << “Hello World. . . ”<< “ ვარ თუ არა ხელოვნების ნაწილი? (Y/N)”;
  cin >> s;
  cout << “პრინციპში მე სულაც არ მაინტერესებს რას ფიქრობ ამაზე . . .” << endl;
}
```

ანალიზი. მე ვუყურებ და-ვინჩის «ჯოკონდა»-ს და ვცდილობ დავინახო მასში რაღაც განსაკუთრებული, ამოუცნობი ან უბრალოდ საინტერესო. ის არის ხელოვნების ნიმუში, რომელიც განკუთვნილია ჩემთვის – ერთ-ერთი რიგითი ადამიანისათვის.

და-ვინჩის “ჯოკონდა”

სურ. 1.3

რა აზრიც არ უნდა იყოს ჩადებული მასში, მას უყურებს რა ვიღაც, მოსწონს იგი, არ მოსწონს, აფასებს, არ აინტერესებს ან უბრალოდ თავისი ინტელექტუალური პოტენციალიდან და მიმართულებიდან გამომდინარე უკეთებს შესაბამის ანალიზს და მას, როგორც ამოცანას უძებნის რაიმე ახსნას. ადამიანის სუბიექტური დამოკიდებულება გარესამყაროს მიმართ კი ყოველთვის ასოცირდება ფარდობითობის ფენომენთან და საერთოდ ფარდობითი აზროვნების საწყისებთან: “*. . . ქარი ქრის და ხის ფოთლებს არხევს. გვერდით გაიარეს დედაშვილმა, დედამ შეხედა და თქვა: ქარი ქრის და ფოთლები ირხევიან, პატარამ შეხედა და თქვა: ქარი ქრის, ფოთლებს სცივათ და კანკალებენ . . .*”

მე აქ გამოვკვეთე ორი პიროვნება: ნახატის ავტორი და ნახატის დამთვალიერებელი. თქვენის აზრით ჩემს ადგილას რა ტერმინოლოგიით მოიხსენიებდა თანამედროვე ბიზნესმენი ამ ორ ადამიანს?

მოდით წარმოვიდგინოთ, რომ ვართ თანამედროვე ბიზნესმენები და დავარქვათ ყველას თავისი სახელი. კერძოდ, ნახატის ავტორი არის პროდუქციის მწარმოებელი, ნახატი პროდუქციაა, ხოლო ჩვენ კი ვართ პროდუქციის მომხმარებლები.

ადრე ნაკლებად თუ შეუსაბამებდნენ და-ვინჩისა და ჯოკონდას ამ ტერმინოლოგიას, თუმცა დღეისათვის სხვა სიტუაციაა.

ასე რომ საქმე გვაქვს ორ განსხვავებულ პიროვნებასთან, ერთი, რომელიც ქმნის ხელოვნების ნიმუშს და მეორე რომელიც იყენებს, მოიხმარს მას.

ახლა, ლოგიკურია რომ ეს ორი ადამიანი ერთმანეთს არავითარ შემთხვევაში არ უნდა შევადაროთ, მაშინ როცა შედარების კრიტერიუმები გამომდინარეობს ამ კონკრეტული ხელოვნების სფეროდან.

ერთი არის შემოქმედი, ხელოვანი ამ დარგში, იმიტომ რომ მისმა შედევრმა დაიმსახურა მოხმარება ანუ ამ შემთხვევაში შეფასება, ხოლო მეორე კი არის მომხმარებელი, რომელიც უყურებს ნახატს და აფასებს შეძლებისდაგვარად.

იმედია ზემოთქმულის აზრი გასაგებია. მართალია, დიდი არც არაფერი ფილოსოფია სჭირდება ამის გაგებას, მაგრამ ზოგჯერ ელემეტარული ჭეშმარიტებანი გვავიწყდება ხოლმე და საჭიროებიდან გამომდინარე მათი გამოკვეთა და გამეორება სულაც არ არის ველოსიპედის გამოგონება.

პარალელი. ახლა კი პარალელი, რომელიც უნდა გაივლოს ჩვენს საქმესა და ზემოთ ნათქვამს შორის.

პროგრამული პროდუქტი არის ხელოვნების ნიმუში. ის შეიძლება იყოს ცუდიც, საშუალოც კარგიც და შედევრიც. მას ჰყავს ავტორი, შემოქმედი და ხელოვანი, და მას ჰყავს მომხმარებელი. შესაბამისად დაუშვებელია ერთმანეთს შევადაროთ პროგრამისტი და კომპიუტერული პროგრამების მომხმარებელი, ისევე როგორც ამაზე ვილაპარაკეთ ფერწერის შემთხვევაში.

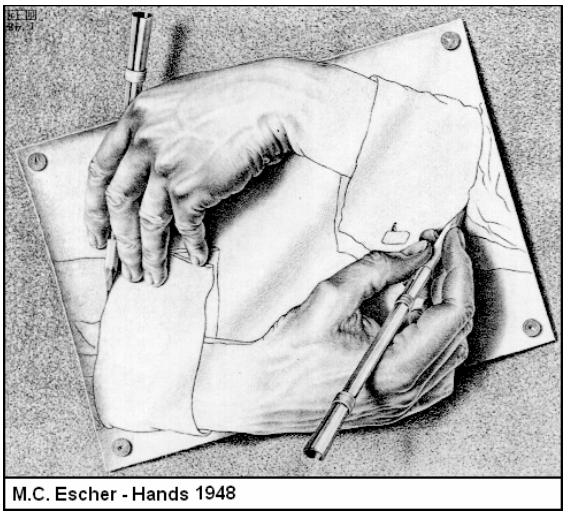
ის რასაც ვუყურებთ დღეს კომპიუტერებში ასე ერთი ხელის მოსმით და თავისით არ შექმნილა და მის ამ სახემდე მოყვანას ადამიანების მთელი ინტელექტუალური რესურსები და შემოქმედებითი უნარი დასჭირდა.

კომპიუტერი ერთის მხრივ არის ქიმიურ ელემენტთა ერთობლიობა, ხოლო მეორეს მხრივ კი ელექტრული დენის გარდაქმნილი სახე, ასე რომ ის თავისით არც არაფერს მოიგონებს და მასში არც არაფერი დევს უპირობოდ თუ ეს არ გააკეთა ადამიანმა.

რატომ არის პროგრამა ხელოვნების ნიმუში?

ჯერ ერთი საქმე გვაქვს ფერებთან, რომელიც აისახება კომპიუტერის ეკრანზე და თანაც ეს ფერები იმდენია, რომ რამოდენიმე უახლოეს ფერს შორის განსხვავებას ძნელად თუ არჩევს ადამიანის თვალი. ლაპარაკია მილიონებზე. ეს რიცხვი ჩვეულებრივ გამოთვლადია და იგი წარმოადგენს 16777216-ს, ფერთა აგების ერთ-ერთი სქემის მიხედვით – RGB(**R**ed**G**reen**B**lue): RGB(0,0,0)-დან, RGB(255,255,255)-მდე ანუ 256 * 256 * 256 = 16777216.

დამეთანხმებით რომ იქ სადაც არჩევანი უზარმაზარია, უკვე მუშაობს არა მარტო ადამიანური ლოგიკა, არამედ ის უნივერსალური ადამიანური ფენომენები, რასაც ჰქვია ფანტაზია, წარმოსახვა, შემოქმედებითობა და ბოლოს ადამიანის ნიჭი ხელოვნებაში.



ეშერის ‘‘ხელები’’
სურ.1.4

ისევე, როგორც ნახატში ჩანს ავტორის შინაგანი სამყარო, მისი ფსიქოლოგიური მახასიათებლები, ინტელექტი და მენტალიტეტი, ზუსტად ასევე კომპიუტერულ პროგრამასთან ურთიერთობისას საქმე გვაქვს ამ პროგრამის ავტორის(თუ ის ერთია) ან ავტორების(თუ ბევრნი არიან) შინაგან სამყაროსთან, მაგალითად ისეთ თვისებებთან, როგორიცაა, ნებისყოფა, გემოვნება, ხელოვნების სიყვარული და მისი ცოდნის დონე, პუნქტუალობა, მოწესრიგებულობა, მომხმარებლის და საკუთარი თავის პატივისცემა, მიუხედავად იმისა, რომ პროგრამირებაში ამ მიმართულებით არსებობს გარკვეული დოგმები, რომლებიც მრავალწლიანი გამოცდილების შედეგად ჩამოყალიბდა პროგრამისტებში, როგორიცაა მაგალითად პროგრამული ინტერფეისის სწორად აწყობის წესები, მაინც პროგრამა მთლიანადაა დამოკიდებული მის ავტორზე და იძლევა დასაშვებ ფარგლებში ვარირების საშუალებას.

არაერთხელ დავრწმუნებულვარ, რომ ამ საკითხში ასევე სრულიად მისაღებია ორიგინალობაც, ეს ხომ ხელოვნებაა და თანაც ამ დარგი სრულუფლებიანი წევრი. გარდა ამისა ისევე, როგორც ნებისმიერი პროფესია გავლენას ახდენს ადამიანის ხასიათზე და პიროვნების ჩამოყალიბებაზე პროგრამირებაც შესაბამისად იძლევა უკუ ეფექტს და ადამიანს უყალიბებს პუნქტუალურობის, დისციპლინის, სილამაზის დანახვისა და სხვა უამრავ ღირსეულ თვისებებს, რაც თავის მხრივ ვლინდება მის ყოველდღიურ ცხოვრებაში, მაგრამ ხაზს ვუსვამ იმას, რომ ეს აშკარად ჩანს მხოლოდ იმ შემთხვევაში თუ პროგრამისტი თვითონ უდგება პროგრამირებას როგორც ხელოვნებას.

ასე რომ ეს ყველაფერი რეალობაა და მისი გათვალისწინება და ჩვენსავე სასარგებლოდ გამოყენება ნამდვილად ღირს, ლაპარაკი აღარ არის იმაზე, რომ ადამიანი ყოველთვის იღებს სიამოვნებას საყვარელი საქმისაგან და ამით კი მისი, როგორც შემოქმედის წარმადობა მნიშვნელოვნად იზრდება.

საწყისები. მეორე რაც ხელოვნებასთან აახლოვებს პროგრამირებას არის მისი ტექნიკური კოდური ნაწილი, რომელიც თავის მხრივ ასევე მრავალფეროვანია.

როგორც კი კაცობრიობას რაღაც ახალი მოევლინება ხოლმე, განსაკუთრებით მეცნიერებაში, მაშინვე იწყება ამ სიახლის გარშემო უამრავი ტალანტებისა და ფენომენების

თავმოყრა და როგორც წესი ისინი დიდ კვალს ტოვებენ შესაბამისი დარგის ისტორიაში და ცვლიან სამყაროს თუ მათი ეს მოქმედება ხდება საჭირო დროსა და სივრცეში:

“პეპელას ფრთების ერთმა დაქნევამ შეიძლება დედამიწის მეორე მხარეს გამოიწვიოს ცუნამი».

“პეპელას ეფექტი”. ედვარდ ლორენცი(Edward Lorenz) 1960

ამ შემთხვევაში არც პროგრამირების სფეროა გამონაკლისი. ისეთი სახის პროგრამირება, როგორიც დღეს არის თავის სათავეებს იღებს 40-იანი წლებიდან (პროგრამირების მოკლე ისტორია მოყვანილია დანართში), 50-იანი წლების დასაწყისისათვის უკვე დაიწყო ადამიანის მიერ პროგრამირების, როგორც დამოუკიდებელ, სრულფასოვან, საქმიან სფეროდ მიღება და უამრავი სამეცნიერო სამუშაო მიეძღვნა მას, რაც რა თქმა უნდა დღემდე გრძელდება.

მათ შორის გამორჩეულია სტანფორდის უნივერსიტეტის(ა.შ.შ. კალიფორნია) პროფესორის დონალდ ერვინ კნუტის(Dr. Donald Ervin Knuth) შვიდ ტომეული სახელად “კომპიუტერული პროგრამირების ხელოვნება”. ამ შვიდი ტომიდან ყველაზე გავრცელებულია პირველი სამი, სადაც ლაპარაკია ძირითად ალგორითმებზე, მათ ანალიზსა და სხვა საჭირბოროტო ალგორითმულ საკითხებზე. ეს არ არის ზოგადად ალგორითმების თეორია, რომელიც დღემდე დაუმთავრებლად ითვლება თავისი სირთულის გამო, არამედ არის უფრო პრაქტიკული ცოდნის წყარო პროგრამისტებისათვის.

ეს წიგნები დაიწერა 60-იანი წლების დასაწყისიდან იმ დროისა და მდგომარეობის შესაბამისად, მაგრამ მათ მნიშვნელობა დღესაც არ დაუკარგავთ, მიუხედავად იმისა, რომ კომპიუტერული სამყარო ამ ხნის განმავლობაში ასე სწრაფად და მნიშვნელოვნად შეიცვალა.

მათში ლაპარაკია ხელოვნებაზე, რომელიც პირდაპირ კავშირშია მათემატიკურ აზროვნებასთან და თავისთავში აერთიანებს კიდევ ალგორითმულ აზროვნებასაც. აღსანიშნავია, რომ თავისი მენტალური მხარით ეს წიგნები შედარებულია 25 საუკუნის წინანდელ სუნ ძის წიგნთან “ბრძოლის წარმოების ხელოვნება” (სუნ ძი - ძვ.წ.ა. VI-V საუკუნეების ჩინელი მხედართმთავარი და სამხედრო თეორეტიკოსია. მისი წიგნი «ბრძოლის წარმოების ხელოვნება» გასული საუკუნეების ყველა დიდი მხედართმთავრის სახელმძღვანელო ტრაქტატი იყო. დღეს კი იგი ყველა დიდი ბიზნესმენის ბესტსელერად იქცა).

სუნ ძი - ძვ.წ.ა. VI-V საუკუნეების ჩინელი მხედართმთავარი და სამხედრო თეორეტიკოსი
სურ. 1.5

ამ წიგნებში აქცენტი უფრო გამახვილებულია პროგრამისტის მათემატიკურ ბაზისის სრულყოფაზე, ბაზისისა, როგორც ერთ-ერთი მძლავრი იარაღისა პროგრამისტისათვის, თუმცა დონალდ კნუტი თავის პირველ ტომში ხაზს უსვამს იმ მომენტს, რომ ალგორითმების არსის გასაგებად არა არის აუცილებელი უმაღლესი მათემატიკის ცოდნა, ანუ თავისთავად ალგორითმული აზროვნება დამოუკიდებელი ფენომენია. რაც შეეხება მათემატიკას, მის მთავარ როლს პროგრამირებაში ჩვენ ცოტა მოგვიანებით განვიხილავთ.

გასაგებია რომ მაშინ პროგრამების ზომები გაცილებით მცირე იყო დღევანდელთან შედარებით და ასეთი სახის ლიტერატურა უშუალოდ წარმოადგენდა კომპიუტერული პროგრამების დამუშავების ერთადერთ მძლავრ იარაღს. სერიოზული პროგრამის შედგენამდე ხდებოდა ალგორითმების აგება და მათი მათემატიკური ანალიზი, მათივე ოპტიმიზაციის მიზნით(მეხსიერების დაზოგვის ან სისწრაფის გაზრდისათვის), რაც რა თქმა უნდა თავის მხრივ მოითხოვდა მათემატიკის კარგ ცოდნას.

დღეს სიტუაცია შეიცვალა, გამზადდა უამრავი ბიბლიოთეკები, შეიქმნა უამრავი მზა ალგორითმები(STL Standart Template Library - სტანდარტული შაბლონების ბიბლიოთეკა, ენა Fortran-ის რიცხვითი მეთოდების ბიბლიოთეკები) რიცხვითი მეთოდებისა და სხვა საბაზისო პროგრამული სისტემებისათვის, აქცენტი მიმართულია უკვე პროგრამისტისათვის პროექტზე მუშაობის მაქსიმალურად გამარტივებაზე, მაგრამ ეს ყველაფერი იმას არ ნიშნავს რომ ის მათემატიკა და ალგორითმული ანალიზის საკითხები რაც მოყვანილია ამ და კიდევ ბევრ სხვა წიგნში აღარ წარმოადგენს საჭიროებას დღევანდელი პროგრამისტისათვის, პირიქით, სანამ იარსებებს ასეთი ტიპის კომპიუტერები ანუ მთვლელ-ტრიგერული (დღევანდელი კომპიუტერების საბაზისო ორობითი(ციფრული) ოპერაციები ხორციელდება მთვლელებზე და ინფორმაცია ინახება რეგისტრებში, რომლებიც თავის მხრივ შესდგებიან ტრიგერებისაგან - ლოგიკური 0-ის ან ლოგიკური 1-ის შემნახველი მოწყობილობისაგან) და ელემენტარული გათვლების საჭიროება, მანამ ეს წიგნები იქნება პოპულარული და ერთ-ერთი საყვარელი ლიტერატურა პროგრამისტისათვის.

დონალდ კნუტი ახდენს პროგრამირების ხელოვნების ინტერპრეტირებას თავისი დროის და მდგომარეობის შესაბამისად და მიუხედავად იმისა რომ ხელოვნებამ

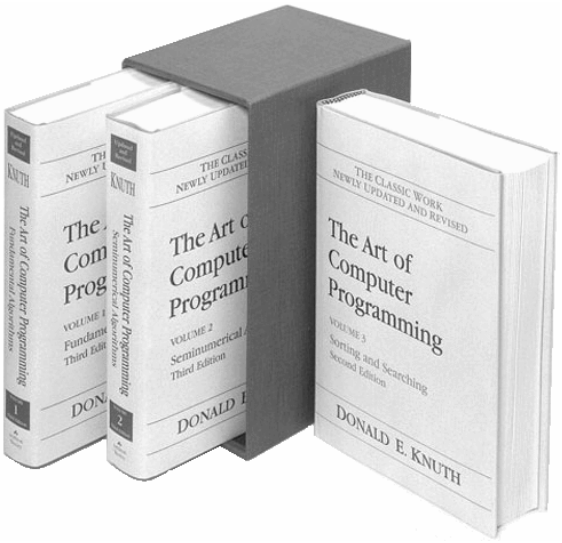
პროგრამირებაში ახლა უკვე სხვა სიბრტყეებზეც გადაინაცვლა (თუ გავითვალისწინებთ გრაფიკის გიგანტებსაც Descreet, Adobe და სხვა) მისი შეფასებები დარჩა მაინც უცვლელი და ფასეული.

დონალდ ერვინ კნუტი
Donald Ervin Knuth
高德纳,
სურ. 1.6

ბოლოს და ბოლოს ნებისმიერი თანამედროვე სერიოზული პროექტი მოითხოვს რაღაც ტიპის მათემატიკურ გათვლებს და ისეთი კოდების შედგენას რომელთა შესაბამის ალგორითმების ანალიზი და ოპტიმიზირება სრულიად მიზანშეწონილი და საჭიროც კია.

“თუ ფიქრობ რომ ხარ კარგი პროგრამისტი, მაშინ წაიკითხე “კომპიუტრული პროგრამირების ხელოვნება” (The Art of Computer Programming). . .”
ბილ გეიტსი - მაიკროსოფტის პრეზიდენტი (Bill Gatse)

“ბოლო ოცი წლის განმავლობაში მსოფლიო შეიცვალა”
ბილ გეიტსი. 1995.



დონალდ კნუტის სამტომეული
სურ 1.7

- მაგრამ შედევრები დარჩა უცვლელი.

წესრიგი. ახლა მოდით გავაანალიზოთ რა სიტუაცია გვაქვს დღეისათვის. დღეს ძირითადად ფიგურირებს ლოკალური და ვებ-საინტერნეტო(Web-Applications) ტიპის აპლიკაციები, ასევე აქტუალურია აპლიკაციები მინი კომპიუტერისა და მობილური სისტემებისათვის. დიზაინის თვალსაზრისით განსაკუთრებით მრავალფეროვანია ვებ-

აპლიკაციები და ეს მრავალფეროვნება უკვე მიდის იქამდე, რომ საქმეში ჩაერთვნენ ფერწერული ნიჭით დაჯილდოვებული ადამიანები.

მაგალითისათვის ავიღოთ პროგრამის გარე ინტერფეისის აწყობის მეთოდები.

როგორც უკვე ვახსენე დროთა განმავლობაში პროგრამისტს უყალიბდება თავისი დამოკიდებულება კომპიუტერული ხელოვნების მიმართ და ადგენს საკუთარ კანონზომიერებს ამ მიმართულებით. ამას საფუძვლად უდევს პროგრამირების ესთეტიური ხასიათიც – "თანამშრომლობა" მომხმარებელთან, რასაც უკავშირდება ასევე ტერმინის User Friendly Interface (მომხმარებლისთვის მოხერხებული ინტერფეისი) არსებობა.

დღეისათვის ყველაზე გავრცელებული, ან უკიდურეს შემთხვევაში ბევრი სხვა წესის წყაროდ შეიძლება ჩაითვალოს შემდგი ორი წესი.

"გარე ინტერფეისი უნდა იყოს ინტუიციური"

ანუ

"მომხმარებელი არასოდეს კითხულობს ინსტრუქციას"

და

"სამომხმარებლო ინტერფეისი არ უნდა ჰგავდეს პროგრამულ კოდს"

პირველი ნიშნავს, რომ პროგრამის სამომხმარებლო ინტერფეისი უნდა იყოს ინტუიციური მომხმარებლისათვის, ანუ ის უნდა იყოს აგებული ისე, რომ ასე ვთქვათ წინასწარ ხვდებოდეს მომხმარებლის სურვილს, ან უფრო სწორად მიყვებოდეს მის აზრებს პროგრამისადმი წაყენებულ მოთხოვნებთან დაკავშირებით, ანუ არც ერთი ნაწილი პროგრამული ინტერფეისისა არ უნდა იყოს ორაზროვანი, ინფორმაციულად გადატვირთული, დიდი არჩევანის მქონე და ა.შ.

ამ საკითხში განსაკუთრებულ და რევოლუციურ მიგნებას წარმოადგენენ მენიუები, მაგრამ მენიუს გამოყენებასაც გააჩნია თავისი სწორი მეთოდები და მისი დაუდევარი მოხმარება ხშირად იძლევა ხოლმე უკუ ეფექტს.

მენიუები ამ შემთხვევაში არის მხოლოდ იარაღი, თორემ ინტერფეისის სწორად აგება მათ გარეშეცაა შესაძლებელი ისე, რომ ის არ სცდებოდეს ზომიერების ფარგლებს (მაგალითად ფანჯრებისა და ქვეფანჯრების სტრუქტურებით).

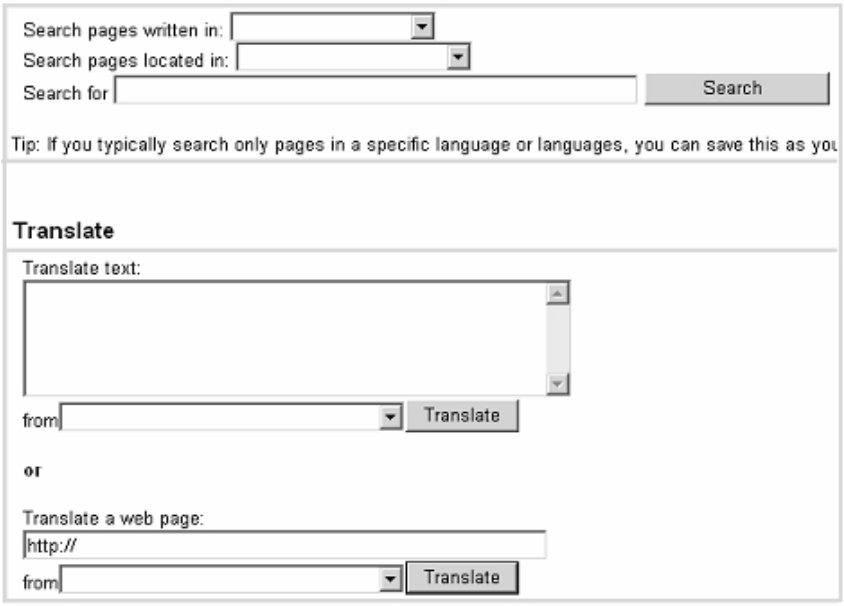
რაც შეეხება ინსტრუქციას, ყველა სერიოზულ პროდუქტს თან ახლავს ინსტრუქციაც, გასაგებია რომ პროგრამის გამოყენებამდე მას კითხულობს ვიღაც მაინც თუ ლაპარაკია განსაკუთრებით პუნქტუალურ მომხმარებელზე, მაგრამ იმდენად რამდენადაც ასეთები ცოტაა, რეალურად პროგრამისტს უწევს მაქსიმალუად დაიცვას პირველი წესი და ეს არ არის ადვილი გასაკეთებელი, რადგან ამ შემთხვევაში თავისუფალი არჩევანი ძალიან დიდია - *"ადამიანი დაწყევლილია თავისუფლებით".*

ახლა რა იგულისმება მეორე წესში.

ხშირ შემთხვევაში პროგრამა იწერება ისეთი მომხმარებლისათვის რომელიც ან საერთოდ არ იცნობს კომპიუტერს ან კიდევ იცნობს მას და მიჩვეულია ტერმინოლოგიასა და ინტერფეისის სტანდარტულ სტრუქტურებს.

ის რომ პროგრამისტი თავის პროდუქციაზე მუშაობისას აქცენტს უნდა მიმართავდეს სტანდარტების დაცვისაკენ, გასაგებია, მაგრამ გარდა ამისა ინტერფეისს უნდა ჰქონდეს მარტივი ენა და მისი კომპონენტების განლაგებიდან, თუ საერთო სტრუქტურიდან თუ წარწერების შინაარსიდან უნდა გამოირიცხოს ყოველგვარი ლოგიკური და დედუქციურ პრინციპებზე დამყარებული მექანიზმები თუ რა თქმა უნდა ეს პროდუქცია არ არის განკუთვნილი სამეცნიერო მიმართულების პერსონალისათვის, მაგალითად: Mathcad, Archicad და სხვა.

ახლა მოვიყვან ერთ მაგალითს, რომელიც აღებულია რეალური პროგრამიდან და რომელიც ნათლად აჩვენებს თუ როგორი არ უნდა იყოს ინტერფეისი პირველ რიგში.



ცუდი სამომხმარებლო ინტერფეისის მაგალითი

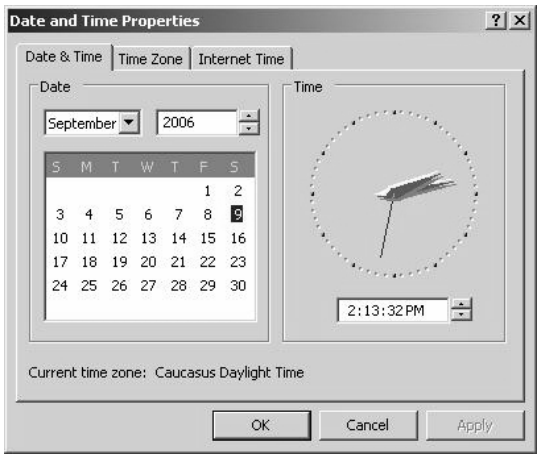
სურ. 1.8

აქ ზოგიერთი წარწერა შეგნებულადაა ამოშლილი.

რა შეიძლება ითქვას? ბევრი რამ. უპირველეს ყოვლისა კი ის, რომ ამ ინტერფეისის სტრუქტურა ვერანაირ კრიტიკას სიმეტრიულობის შესახებ ვერ უძლებს;

როდესაც ლაპარაკია ოპერაციული სისტემა Windows-ის სტანდარტულ სამომხმარებლო კომპონენტებზე (ღილაკები, სარედაქტირო ბოქსები, სიები და სხვა) პროგრამისტი ყოველთვის უნდა იცავდეს სიმეტრიის პრინციპს. ადვილად ჩანს რომ ჩვენს შემთხვევაში არანაირ სიმეტრიაზე და განლაგების კულტურაზე ლაპარაკი არ არის.

ეს კი ყველასათვის ცნობილი ფანჯარაა, განკუთვნილი დროისა და თარიღის დაყენებისათვის Windows-ში.



კარგი სამომხმარებლო ინტერფეისის მაგალითი
სურ. 1.9

სავსებით დასაშვებია, რომ საათის განყოფილება იყოს შედარებით დიდი ან პატარა, მაგრამ პროფესიონალურობიდან გამომდინარე დაცულია ელემენტარული სიმეტრია.

ახლა დავუშვათ რომ ამ ინტერფეისის ავტორმა გამოიყენა ორიგინალობის პრინციპი, რაც თავის მხრივ გადახრაა ყოველგვარი სიმეტრიიდან და თავისებურ ხიბლს აძლევს პროგრამის გარე სახეს, ასეთი ვარიანტები გამოიყენება განსაკუთრებით ვებ-აპლიკაციებზე მუშაობისას (Web Design), მაგრამ ამ შემთხვევაში როგორც წესი იყენებენ არასტანდარტულ კომპონენტებს - ე.წ. ამოვარდნები ფერთა ან სიმეტრიის სისტემიდან, ესეც გასაგებია რომ ჩვენს მაგალითში არავითარ ორიგინალობას და ამოვარდნას სტანდარტული წყობიდან არა აქვს ადგილი.

და კიდევ ერთი მაშველი, რაც შეიძლება ამართლებდეს მის ავტორს, ეს არის სიმეტრიების დარღვევა შინაარსისა და პრიორიტეტის გამოკვეთის მიზნით.

მაგრამ ერთი კონკრეტული ინტერფეისული ჯგუფისათვის ასეთი დარღვევა შეიძლება იყოს მხოლოდ რამოდენიმე, ყოველ შემთხვევაში ჩვენი მაგალითი არ წარმოადგენს იმ სირთულის ინფორმაციულად გადატვირთულ ინტერფეისს რომ იგი იმსახურებდეს ასეთ უდიერ მიდგომას, როდესაც ყოველი მეორე კომპონენტი განთავსებულია ყოველგვარი გემოვნების და ხელოვნების პრინციპების დაცვის გარეშე.

მეორეც კიდევ ის რომ სამწუხაროდ ასეთი შემთხვევები არ არის მხოლოდ რამოდენიმე და ხშირად გვხვდება სახელგანთქმული კომპანიების პროდუქტებშიც კი.

სხვა სკითხია ასევე პროგრამის ფუნქციონალური მხარე.

პროგრამის ყოველი პუნქტი უნდა იყოს მიზანმიმართული. რაც შეიძლება ცალსახა, მარტივი და გასაგები და რა თქმა უნდა ხარვეზებისგან(Bugs) თავისუფალი. თუ როგორ უნდა

უზრუნველყოთ ეს ყველაფერი, ამის შესახებ დაწვრილებით განვიხილავთ წიგნის მეორე ნაწილში.

დასკვნა. ნახატის მიხედვით შეგვიძლია ვიმსჯელოთ ნახატის ავტორზე. ასე რომ ჩვენი პროგრამა ჩვენივე სარკეა და ის გვამხელს ისეთ მომენტებშიც რომლებიც შეიძლება გვეგონოს ყველაზე დაფარულნი. პროგრამირების საშუალებით ჩვენ სწრაფად ვხედავთ ჩვენი შრომისა და შემოქმედებითი პოტენციალის შედეგს, სწრაფად შევიგრძნობთ ჩვენი ნაწარმოების მომხმარებელთან ურთიერთობის სასიამოვნო პროცესს, მაგრამ უნდა გვახსოვდეს რომ შედევრები უგულო დამოკიდებულებით არ იქმნება. ამაში კიდევ უფრო დარწმუნდებით თუ გადახედავთ რამოდენიმე სტატიას ინტერნეტში ამ საკითხებთან დაკავშირებით, სადაც ლაპარაკია იმაზე, რომ დღევანდელ პროგრამულ პროექტირებაში არ არის საკმარისი მხოლოდ ტექნიკური რესურსების დახვეწა, არამედ ძალიან დიდ როლს თამაშობს ასევე პროგრამისტის პიროვნული თვისებები და ფაქტიურად ამ ფაქტორის გათვალისწინების გარეშე გიგანტური პროექტები ვერც იქმნება; დიდი მნიშვნელობა აქვს იმას თუ როგორ უდგება პროგრამისტი თავის საქმეს, აქედან გამომდინარე პროექტი ან კარგი გამოდის ან ცუდი ან საერთოდ ვარდება. ცუდად დაპროექტებული და განხორციელებული სისტემის წარამარა გადაკეთება ყოველთვის ხარჯებთანაა დაკავშირებული.

კომპიუტერული პროგრამირება ხელოვნებაა არა მარტო მომხმარებლისათვის არამედ თვით პროგრამისტის თვალთახედვიდანაც;

ხელოვნების კანონები მოქმედებს, როგორც პროგრამის გარე ანუ დანახვად ნაწილში, ასევე მომხმარებლისათვის უხილავ შიდა ნაწილშიც, რასაც ჰქვია პროგრამის კოდი. აქ უკვე პროგრამას როგორც ხელოვნების ნიმუშს აფასებს არა უბრალო მომხმარებელი, არამედ თვითონ პროგრამის ავტორი ან სხვა, მისი შემოქმედებით დაინტერესებული პროგრამისტი.

მინდა გამიგოთ სწორად; ეს ყველაფერი არ ნიშნავს იმას, რომ მაინცდამაინც საჭიროა ურიცხვი ფერები და სუპერთანამედროვე სპეცეფექტები, უბრალოდ პროგრამა არ უნდა იყოს უგემოვნო, არც დიზაინში და არც ფუნქციონალობაში.

მოკლედ პროგრამირება ვითარდება და მისი ხელოვნებაც დროთა განმავლობაში ალბათ სხვადასხვა ინტერპრეტაციით მოგვევლინება. უნივერსალური პრინციპები კი არასოდეს იკარგება და მათი დამახსოვრება და გათავისება, რაც გულისხმობს პრაქტიკაში გატარებას ფრიად სასარგებლო რამეს წარმოადგენს ყოველი ადამიანისათვის, განსაკუთრებით კი პროგრამისტისათვის, რადგან პროგრამირება გაცოცხლებული მათემატიკაა, მათემატიკა კი ზუსტი მეცნიერებაა და მილიონი ჭეშმარიტებიდან თუნდაც ერთ მცდარობას, უკიდურეს შემთხვევაში კი კენტი რაოდენობის შეცდომას ადამიანს არასოდეს პატიობს.

“. . . მე ყოველთვის ვიქნები დარწმუნებული იმაში, რომ რაღაც მაგიური ხდება, როდესაც პროგრამაზე ვმუშაობ. მე შემიძლია ვაპროგრამო მთელი დღის განმავლობაში და

ამით ვიყო სრულიად კმაყოფილი და ბედნიერი. მე ვიცი რომ ამ სამყაროში კიდევ არიან ისეთები, რომლებიც ფიქრობენ და განიცდიან ჩემნაირად. მიუხედავად იმისა, რომ არსებობს უთვალავი მიზეზი რომელთა მიხედვითაც პროგრამირება შეიძლება განხილული იქნეს როგორც ხელოვნება, მე მაინც არც ერთ მათგანს არ ჩავთვლი ჩემთვის არსებითად. მე შემიძლია ვთქვა ასეც და ისეც, თუმცა ეს ყველაფერი მაინც დამოკიდებულია პიროვნულ შეხედულებაზე. ჩვენი შეხედულება კი დამოკიდებულია ჩვენს გამოცდილებაზე, შედეგად თუ ჩვენ არ გვაქვს გამოცდილება კომპიუტერულ პროგრამირებაში, მაშინ ნაკლებად თუ შევძლებთ დავინახოთ მისი სილამაზე. ეს ძალიან ჰგავს გალილეოს მცდელობას გაეგებინებინა შუასაუკუნეების გაუნათლებელი ხალხისთვის რომ დედამიწა ნადვილად ბრუნავს მზის გარშემო. მას მაშინ არავინ უჯერებდა. . .”

Justin Erenkrants(ჯასტინ ერენკრანტსი) 1998

პ რ ო ფ ე ს ი ა

“ თუკი გსურს კარგად შეასრულო შენი მოვალეობა,
გააკეთე სულ ცოტა იმაზე მეტი ვიდრე საჭიროება მოითხოვს”
“წებისმიერი შემეცნება საბოლოოდ - ეს არის საკუთარი თავის შეცნობა”
“მე არ შემიძლია გასწავლო შენ, მე შემიძლია მხოლოდ
დაგეხმარო გამოიკვლიო საკუთარი თავი. . . მეტი არაფერი”
ბრუს ლი(Bruce Lee)

პროგრამირების აზროვნება. დღეს პროგრამირებისათვის ეს სტატუსი უფრო აქტუალურია ვიდრე დანარჩენი ორი – ხელოვნება და მეცნიერება. ეს იმიტომ რომ ის არის იარაღი, და ლოგიკა რომელიც აღწერს ყველაფერს რაც კი ლოგიკას ემორჩილება.

რა არის ხელობის, როგორც ფენომენის მახასიათებელი თვისებები?

1. პროფესიონალიზმი – საკუთარი საქმის უკეთ ცოდნა
2. კონკურენტუნარიანობა
3. ბრძოლისუნარიანობა
4. გამოცდილება

ყველა ეს პუნქტი სრულად ახასიათებს პროგრამირებას, როგორც ხელობას და პროგრამისტს უწევს მათი, როგორც გზამკვლევი პრინციპების დახვეწა და პატივისცემა.

ახლა შევუდგები რამოდენიმე მათგანის გაანალიზებას, რომლებიც ნამდვილად ღირს, რომ იქნას გათვალისწინებული პროგრამირების სფეროში მოღვაწე ადამიანების მიერ, თუმცა ის თემა რაზეც აქ მექნება საუბარი სამწუხაროდ თუ საბედნიეროდ ამოუწურავია, რადგან ამას განაპირობებს თვით პროგრამირების განვითარების და პროგრამული სისტემების შესაძლებლობათა სფეროს განუწყვეტელი გაფართოების პროცესები.

შეიძლება პროგრამირებაში ნაკლებად ჩახედულ მკითხველს გაუჩნდეს გაურკვევლობის შეგრძნება: რა არის ამ საქმეში ასეთი განსაკუთებული, რომ ამდენ ყურადღებას იმსახურებს? ასეა თუ ისე ბოლოს და ბოლოს რამენაირად მაინც დაწერ პროგრამას და დანარჩენს გააკეთებს კომპიუტერი.

ჩიტი ბრდღვნად ნამდვილად ღირს.

ასე ვთქვათ, არც ისე მარტივად არის ყველაფერი, როგორც ეს ჩანს ერთი შეხედვით, საქმე იმაშია, რომ ადამიანი, ამ შემთხვევაში პროგრამისტი, პროექტზე მუშაობის პროცესში, აქვს რა არჩევანის დიდი არეალი, ამავდროს შეზღუდულია გარკვეული კრიტერიუმებით, როგორიცაა კომპიუტერის მუშაობის სისწრაფე და მისი ინფორმაციული ტევადობა, გარდა

ამისა თავს იჩენენ ე.წ. პოტენციური ხარვეზები, პოტენციური იმიტომ, რომ ისინი არ ჩანან პროგრამის შექმნის საწყის ეტაპზე და ქმნიან უკვე დიდ პრობლემებს მაშინ როცა პროექტი უახლოვდება საბოლოო სახეს. ამას ემატება კიდევ იმ კომპრომისთა ზღვა, რომლებიც დაკავშირებული არიან სამუშაოს გამარტივებასა და სამომხმარებლო მოთხოვნების ზედმიწევნით დაკმაყოფილებასთან. ეს ყველაფერი კი იწვევს იმას, რომ საჭირო ხდება გარკვეული წესების, ანუ კოდირებისას ყოფაქცევის კრიტერიუმების შემოღება და მათი პრაქტიკაში გატარება, რისი უგულებელყოფაც აუცილებლად განაპირობებს არასასურველ შედეგებს.

როდესაც პროგრამისტი პირველ ნაბიჯებს დგამს თავის საქმეში, მას რაღაც მომენტში უწევს ჩვეული აზროვნების გადაყვანა სულ სხვა კალაპოტში.

სიცხადისათვის, განვიხილოთ ერთი ცნობილი ამოცანა:

ამოცანა. ჩამოწერეთ პუნქტების სახით ჩაიდანის საშუალებით წყლის სამჯერ ადუღების თანმიმდევრობა.

დაახლოებით ათიდან ცხრა შემთხვევაში არაპროგრამისტი დაწერს შემდეგს:

1. ჩაასხი წყალი ჩაიდანში.
2. დადგი გაზქურაზე.
3. აადუღე.
4. ჩაასხი წყალი ჩაიდანში.
5. დადგი გაზქურაზე.
6. აადუღე.
7. ჩაასხი წყალი ჩაიდანში.
8. დადგი გაზქურაზე.
9. აადუღე.

დაახლოებით ათიდან ცხრა შემთხვევაში კი პროგრამისტი დაწერს შემდეგს:

1. ჩაასხი წყალი ჩაიდანში.
2. დადგი გაზქურაზე.
3. აადუღე.
4. აიღე ერთი ასანთის ღერი.
5. სამი ასანთის ღერი გაქვს? თუ კი მაშინ დაამთავრე, თუ არა, მაშინ გადადი პირველ პუნქტზე.

როგორ ფიქრობთ რომ მოეთხოვათ წყლის 20-ჯერ ადუღება რომელი ალგორითმი იქნებოდა უფრო პრაქტიკული წაკითხვის თვალსაზრისით?

ეს მაგალითი არის ერთი წვეთი პროგრამირებაში არსებული განსხვავებული მიდგომებისა და აზროვნებათა ოკეანიდან, შესაბამისად ჩნდება თავისებური ხასიათის ამოცანებიც, თუმცა მათ ბევრი საერთო შეიძლება გამოვუნახოთ ჩვენი ცხოვრების სხვა სფეროებში არსებულ ამოცანებთან, რაც მოგვცემს იმის საშუალებას რომ ერთ სფეროში მიღებული გამოცდილება გამოვიყენოთ მეორეში არსებული ბარიერების გადასალახად.

“**ჩასაფრებული დრაკონები**”. ცნობილია რომ აღმოსავლურ ორთაბრძოლების ადამიანის მიერ შესწავლისა და გათავისების პროცესი არასოდეს არ მთავრდება ანუ გრძელდება მთელი სიცოცხლე.

რატომ? ეს ცოტა დიდი თემაა, მაგრამ შევეცდები ძალინ მოკლედ ჩამოვაყალიბო.

თვით იგივე კარატის ფილოსოფია არის ფართო და მრავლის მომცველი და ითვალისწინებს ადამიანის სრულყოფას, როგორც ფიზიკური ასევე გონებრივი-ფსიქოლოგური მიმართულებით, ხოლო ბრძოლის მამოძრავებელ ძალას კი წარმოადგენს ქვეცნობიერი მიზანი იყო სხვაზე სრულყოფილი, იბრძოლო ბოლომდე და დაუსრულებლად განივითარო საკუთარი თავი, რატომ დაუსრულებლად? იმიტომ, რომ ის სხვაც ვითარდება შენთან ერთად. აქვე მინდა ვთქვა, რომ ამ წინსვლის კიდევ ერთი მამოძრავებელი ძალაა თვმდაბლობა, ხოლო მტერი კი – სიამაყე.

როდესაც ორი ოსტატი ერთმანეთს ებრძვის და ეს ბრძოლა მიმართულია გამარჯვებისაკენ, მათგან როგორც წესი ერთ-ერთი აუცილებლად დამარცხდება, გაიმარჯვებს ის ვინც თავის დროზე ყურადღებას აქცევდა ისეთ წვრილმან მომენტებს როგორიც იყო მაგალითად სწორი სუნთქვა, ფსიქიკური წონასწორობის შენარჩუნება და სხვა, ლაპარაკი არ არის ტექნიკურ საკითხებზე, ეს ყველაფერი კი ამ შემთხვევაში აუცილებელია მებრძოლისათვის, რათა მან შეინარჩუნოს ენერგია და საკუთარ თავზე კონტროლი დიდი ხნის განმავლობაში. ანუ ის პატარა თვისებები, რომელთა არ ქონა არ ჩანს მცირე დროით ინტერვალში, დროთა განმავლობაში თავს იჩენენ დიდი პრობლემების სახით.

ორთაბრძოლა

სურ. 1.10

ამ მახასიათებელთა უგულებელყოფის შედეგად მებრძოლი გარკვეული ხნის შემდეგ კარგავს ენერგეტიკულ რესურსებს და მარცხდება.

ამ მაგალითიდან მე გამოვკვეთ იმ მნიშვნელოვან მომენტს, რომ არსებობენ ისეთი მოვლენები, რომლებიც თავს იჩენენ მხოლოდ სისტემის მოცულობის გაზრდის შემთხვევაში და ამის შესახებ აუცილებლად უნდა ვიცოდეთ. თუ თქვენ შეხება გაქვთ აღმოსავლური ორთაბრძოლების რომელიმე სახეობასთან, ასეთი მომენტების დანახვა ძალიან ადვილია.

ისინი გვეჩვენებიან უმნიშვნელოდ, მანამ, სანამ სისტემა პატარაა, თუმცა მათ დასანახად საჭიროა დრო და რისკი.

ჩვენს შემთხვევაში პროგრამისტს არავინ ებრძვის, მაგრამ თუ ზემოთ ნახსენები მოვლენები უგულებელყო მისი პროექტი ან არ გამოვა ან უკეთეს შემთხვევაში ექნება უამრავი ნაკლი, როგორც სამომხმარებლო ასევე შიდა ფუნქციონალობის თვალსაზრისით. და მისი როგორც მებრძოლის დამარცხებაც იქნება სწორედ ეს - ის ებრძოდა ალგორითმულ ლოგიკას და ლოგიკამ აჯობა მას რადგან მან ვერ მოუძებნა სწორი მიდგომის წერტილები.

არსებობს წესები პროგრამულ კოდშიც რომლებიც წარმოადგენენ ფრიად საჭირო ნივთებს და ზოგჯერ მათი გამოყენება გარდაუვალიც კია.

წარმოიდგინეთ რომ მხატვარმა არ დაიცვას ფერების შეხამებისა და შერჩევის ელემენტარული წესები მიუხედავად იმისა რომ მას არავინ ავალდებულებს ამის კეთებას, დადგება მომენტი როცა ის მიხვდება, რომ რატომღაც ნახატს ვერ ამთავრებს, ის არ გამოდის ისეთი, როგორიც უნდა იყოს და როგორიც მას სურს რომ იყოს.

ეს საკითხი განსაკუთრებით აქტუალურია პროგრამირებაში დიდ პროექტებთან მუშაობისას, მაშინ, როცა საქმე ეხება შეზღუდულ დროს და მომხმარებლის ფენომენს.

აქ ხაზი მინდა გავუსვა ტერმინს “პროექტი”, იმათთვის ვისაც მიზეზთა და მიზეზთა გამო ძირითადად საქმე აქვთ მცირე ზომის პროგრამებთან, შეიძლება ბევრი რამ იყოს ახალი, დიდი პროგრამების შემთხვევაში კი სულ სხვა ვითარებაა, რადგან ერთი კონკრეტული

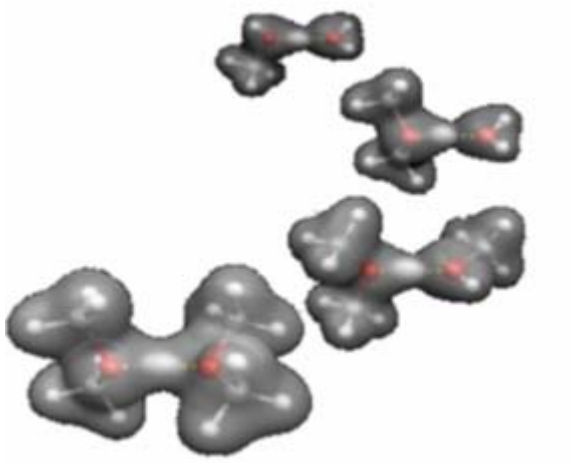
პროექტი გულისხმობს უამრავ პარალელურ, დამხმარე, სატესტირო და მოსამზადებელ სამუშაოებს.

აქ და შემდგომაც ლაპარაკი მაქვს დიდ პროგრამებზე, როდესაც პროგრამირების, როგორც ხელოვნების, ხელობის და სამეცნიერო პლატფორმის ხასიათი უკვე აშკარად ვლინდება.

თუმცა არც მცირე, თუნდაც რამდენიმე ხაზიან პროგრამებშია ეს ყველაფერი დაფარული, უბრალოდ იქ ისინი არ ჩანან ანუ პროგრამული კოდი მათ შეიცავს პოტენციურად.

ამასთან დაკავშირებით მოვიყვან მაგალითებს ფიზიკიდან:

მიკრო ნაწილაკები(ელექტრონები, ნეიტრონები და სხვა) ავლენენ სულ სხვა თვისებებს ცალკე ყოფნისას და სულ სხვა თვისებებს მათი დიდი რაოდენობით გაერთიანებისას, რითაც მიიღება ჩვენი სამყარო და რაშიც მოქმედებს უკვე მექანიკის კანონები. დამოუკიდებლად ყოფნისას ცნობილია, რომ თითოეული მათგანი იქცევა ერთდროულად, როგორც ტალღა და როგორც ნაწილაკი, ხოლო მათი მილიარდობით და წარმოუდგენელი რაოდენობით გაერთიანების შედეგად კი მიიღება უკვე ჩვენთვის ხილული საგნები, რომლებიც როგორც ვთქვი ავლენენ უკვე კოლექტიურ თვისებებს რასაც ფიზიკოსები მექანიკის კანონებს უწოდებენ.



ატომური სტრუქტურები
სურ. 1.11

გრავიტაციული ველი, რომელიც მართავს უზარმაზარ კოსმოსურ სხეულებს, თუმცა ნაწილაკების ტოლ მანძილებზე იგი ფაქტიურად უმნიშვნელოა, ანუ ყველა ნაწილაკი პოტენციურად შეიცავს გრავიტაციის თვისებას, მაგრამ მას აზრი ეძლევა მხოლოდ უზარმაზარი რაოდენობით გაერთიანების შემთხვევაში.

ან კიდევ ასეთი მაგალითი. ცალკეული ადამიანი ავლენს სულ სხვა თვისებებს და მილიონობით ადამიანის გაერთიანებისას უკვე მიიღება ბრბო, რომელიც იმართება ბრბოს კანონებით. შეიძლება რაღაც კონკრეტული თვისება ის რაც ახასიათებს ბრბოს მცირე სიდიდით იდოს ცალკეულ ადამიანში ისე რომ მასთან უშუალო ურთიერთობისას ის არც ჩანდეს, მაგრამ როგორც კი შეიკრიბება ათასობით პიროვნება უკვე საქმე გვაქვს სხვა ხასიათის ყოფაქცევასთან.

ასევეა პროგრამირებაშიც. პროგრამულ კოდის ზრდისას ის ავლენს მასში პოტენციურად დამალულ თვისებებს რაც ძირითადად გამოიხატება შემდგომ პროგრამაში

ხარვეზების წარმოშობაში და რასაც მივყავართ არასრულფასოვან ან დაუმთავრებელ პროგრამასთან, რომლის გადაკეთება უკვე ცნობილია რომ წარმოუდგენელია და სჭიროა ყველაფრის თავიდან დაწყება; მისი შემდგომი გადაკეთება კი გამოიწვევს ე.წ. პროგრამის სიკვდილს რაც გამოიხატება იმაში რომ ცვლილებები კი არ აუმჯობესებენ მას არამედ პირიქით, პროგრამა ხდება სარისკო მოხმარების თვალსაზრისით და იმ დავალებასაც ვერ ასრულებს ბოლომდე რა მიზნითაც ის თავიდან შეიქმნა.

მინდა იცოდეთ, რომ არსებობს ძირითადი ღეძი, ეს ღერძი არის საწყისი ამოცანის არსი, რომლის მიხედვითაც იგება პროგრამა, გადაკეთებისას არ უნდა მოხდეს ამ ღერძის მკვეთრი გადახრა თავისი საწყისი მდგომარეობიდან, წინააღმდეგ შემთხვევაში ის აუცილებლად გადატყდება, რაც ნიშნავს იმას რომ პროექტი გამოვიდა თავის დრეკადობის არეალიდან და იგი საჭიროებს ძირფესვიან განახლებას.

მაგალითად თუ შემოვიღებთ ასეთ მცნებას - დრეკადობის კოეფიციენტი, რომელიც ახასიათებს სისტემის გადაკეთების უნარიანობას, მაშინ შეიძლება ითქვას რომ დრეკადობის კოეფიციენტი ტვინისა მიახლოებით არის 100%, ხოლო მაგალითად ელექტრული საათის კი თითქმის 0%, რადგან იგი მუშაობს მხოლოდ ერთი ფიქსირებული ალგორითმით. სხვათაშორის, საინტერესოა ის მომენტი, რომ "მოძრავ" - აქტიურ ტვინს, რომელიც გამუდმებით განიცდის "გადაკეთებას", ყველაზე ხშირად სჭირდება ხოლმე სწორედ საათი.

პრაქტიკის შედეგად პროგრამისტი ხედავს ასეთ "ჩასაფრებულ დრაკონებს" და ადგენს მათთან ბრძოლის აქსიომებსა და თეორემებს.

ასეთი თეორემები როგორც ადრე ვახსენე უამრავია და ბევრი მათგანი საერთო შინაარს ატარებს კიდეც. როგორც წესი ლაპარაკია პროგრამირების სწორ სტილზე, ოსტატობაზე პროგრამირებაში ან სხვა ამდაგვარ საკითხებზე.

რთული სიმარტივე. პროგრამისტს პროფესიული პრაქტიკის დასაწყისიდან იმ დრომდე სანამ ის ამ საქმეში იმყოფება გამუდმებით უწევს აზროვნების მიმართულებათა მკვეთრი ცვლა, რაც დამეთანხმება ბევრი მათგანი, რომ აუცილებელი მომენტია პროექტის წინსვლისათვის.

კერძოდ, ცნობილია ამოცანა სახელად "ოთხი წერტილის ამოცანა». იგი მდგომარეობს შემდეგში.

ამოცანა: კალამის აუშვებლად ოთხ წერტილზე გაავლეთ სამი სწორი ხაზი ისე რომ ეს ოთხი წერტილი შეერთდეს და კალამი საწყის მდგომარეობას დაუბრუნდეს.

ამ ამოცანის ერთ-ერთი ამონახსენი განგებაა გადატანილი ერთ-ერთ დანართში, რათა შეეცადოთ მის ამოხსნას დამოუკიდებლად, რა თქმა უნდა თუ იგი უკვე ცნობილი არ არის თქვენთვის, ეს უფრო გასაგებს გახდის იმას, რაზეც ქვემოთ მექნება საუბარი.

ცნობილია რომ თავის დროზე ამ ამოცანის დასმისას ექსპერიმენტში მონაწილე ათი ადამიანიდან ცხრა ვერ ხსნიდა და მათგან ზოგმა კი ის საერთოდ ამოუხსნელად ჩათვალა.

აღმოჩნდა რომ ეს ამოცანა ერთ-ერთ საუკეთესო ტესტს წარმოადგენა ადამიანის გონების ზოგიერთი ნაკლოვანებების გამოსავლენად.

კიდევ ერთი საინტერესო მაგალითი.

ალბათ იცით რას წარმოადგენენ მარტივი რიცხვი. გაგახსენებთ, რომ ეს არის ისეთი რიცხვი რომელიც იყოფა მხოლოდ ერთზე და თავისთავზე:

1, 2, 3, 5, 7, 11, 13, 17, 19, 23. . .

არსებობს ამოცანები პროგრამირებაში, რომლებიც მოითხოვენ ასეთი რიცხვების მიღებას გარკვეულ რიცხვამდე ან გარკვეული რაოდენობის. შედეგად საჭირო გახდა სხვადასხვა ტიპის ალგორითმისა და ალგორითმული მეთოდების შემუშავება, რომელთა ძირითადი მიზანია გამოთვლათა სისწრაფე.

ახლა კი მთავარი.

როდესაც პროგრამისტი ამ ამოცანის ალგორითმიზაციას იწყებს პირველ რიგში ის ისევე აზროვნებს, როგორც მას კარნახობს აზროვნების ინსტიქტები, ანუ ალგორითმიზაციის ერთ-ერთი მიდგომა მდგომარეობს იმაში, რომ პროგრამით ამოცანა გადაწყვიტო ისე, როგორც ამას გადაწყვეტდი ტვინით. ესე იგი მარტივი რიცხვები უნდა ვიპოვოთ გამომდინარე მათივე განმარტებიდან.

ასევე მოვიქცევით ჩვენც.

ამოცანა. ჩამოწერეთ ყველა მარტივი რიცხვი 100-მდე.

ჩვენც ავდგეთ და ავაგოთ ალგორითმი, რომელიც განიხილავს ყველა რიცხვს 1-დან 100-მდე და შეამოწმებს თითოეულ მათგანს იყოფა თუ არა მის ქვემდგომ რიცხვებზე. შემდეგ მივხვდებით რომ ეს მეთოდი ზედმეტად უხეშია და შევეცდებით გაყოფათა რაოდენობის შემცირებას ზედმეტ რიცხვებზე გაყოფის უგულებელყოფის ხარჯზე, რათა ალგორითმი უფრო დაჩქარდეს, შემდეგ კიდევ მოვიფიქრებთ რაიმე მათემატიკურ კანონზომიერებას, რათა მივიღოთ რაც შეიძლება ნაკლებ დანაკარგების მქონე ალგორითმი და ასე მივალთ ალბათ რაღაც ზღვრამდე, რომლის იქითაც უკვე ალგორითმი აღარ მარტივდება – გაყოფების რაოდენობა არ მცირდება. ზოგადად, ცნობილია, რომ გაყოფის ოპერაცია ყველაზე ნელია კომპიუტერის არითმეტიკულ ოპერაციებს შორის, ამიტომ რაც ნაკლები გაყოფა შესრულდება პროგრამაში მით უკეთესი.

ერთ-ერთი ასეთი ალგორითმი მოყვანილი აქვს დონალდ კნუტს თავის წიგნში, აქ გაყოფები მაქსიმალურადაა შემცირებული.

ეს ყველაფერი კარგი, მაგრამ შეამჩნევდით ალბათ, რომ ამ დამქანცველ გაყოფის პროცესს ვერაფრით ვერ გავცდით, თუმცა მთელი პრობლემაც იმაშია, რომ როცა აზროვნება დავიწყეთ ისე რომ, მაინცდამაინც უნდა მოხდეს გაყოფის პროცესი, შემდეგ ძნელია თავში მოგვივიდეს აზრად სრულიად განსხვავებული მიდგომა ამ საკითხისადმი.

თურმე არსებობს მარტივი რიცხვების დათვლის ისეთი ალგორითმიც, რომელიც სრულიად არ საჭიროებს არანაირ გაყოფას და თანაც განსახორციელებლადაც გაცილებით მარტივია და სწრაფი:

ვიღებთ ყველა რიცხვს 100-მდე შემდეგ ვიწყებთ ოთხიდან და ვშლით ყველა რიცხვს ორის ბიჯით 100-მდე, შემდეგ ვიწყებთ 3-იდან და ვშლით ყველა რიცხვს სამის ბიჯით 100-მდე, შემდეგ კიდევ ხუთიდან და ვშლით ყველა რიცხვს ხუთის ბიჯით 100-მდე და ასე შემდეგ ყველა კენტი რიცხვისთვის 100-ის ფესვამდე, რადგან შემდგომებს უკვე აზრი აღარ აქვს, შედეგად ბოლოს დაგვრჩება მხოლოდ მარტივი რიცხვები:

თავიდან გვაქვს:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 . . .

წავშალოთ ყველა ორის ჯერადი:

1, 2, 3, , 5, , 7, , 9, , 11, , 13, , 15, , 17, , 19, . . .

წავშალოთ ყველა სამის ჯერადი:

1, 2, 3, , 5, , 7, , , , 11, , 13, , , , 17, , 19, . . .

და. ა.შ.

ეს ალგორითმი პროგრამულად განხორციელებადია.

რა ხეირი აქედან? ხეირი ძალიან დიდია. გაყოფები არა თუ შემცირდა, არამედ საერთოდ აღარც გვაქვს და ალგორითმი გახდა უფრო მარტივი და მოქნილი.

“ერთობ ძნელია აიძულო შენი თავი, რომ მოსწყვიტო გონება აზროვნების მოცემულ ფორმას”

ვიქტორ პეკელისი – “შენი შესაძლებლობანი ადამიანო”

ალბათ ცნობილი შეგრძნებაა როცა უნებურად წამოგცდებათ: რა მარტივი ყოფილა ჭეშმარიტება და რა ძნელია მისი მიგნება.

ასეთი «მარტივი» ჭეშმარიტებები პროგრამირებაში უხვადაა, ისინი მარტივია თავისი არსით, მაგრამ მათი მიგნება საჭიროებს დროს და გამოცდილებას.

ცოდნა. პროგრამირების სწავლა არ ნიშნავს კონკრეტული ფუნქციების სწავლას ან იმის სწავლას თუ ტექნიკურად როგორ კეთდება ესა თუ ის პროგრამული ობიექტი, არამედ მნიშვნელოვანია პროგრამირების არსის და მეთოდიკის სწავლა, ეს იგივეა როგორც შეადარო

ერთმანეთთან ბავშვისათვის თევზის მიცემა და თევზის ჭერის სწავლება. პრინციპი მარტივია, შეიძლება პროგრამისტმა არ იცოდეს კონკრეტული ფუნქციის სახელი, მაგრამ იცოდეს როგორ გააკეთოს ის.

რაც შეეხება ტექნიკურ მხარეს ის დღეისათვის ნამდვილად არ წარმოადგენს რაიმე მიუღწეველს, თუ გავითვალისწინებთ ინტერნეტის ფენომენს.

მოგეხსენებათ, რომ უკვე დიდი ხანია რაც პროგრამირების ენების შესწავლა გასცდა იმ ზღვარს, როდესაც საკმარისი იყო ერთი სამაგიდო წიგნის გადაკითხვა და ესა თუ ის ენა უკვე გამოწვლილვით იცოდი, თუ არ ჩავთვლით დამხმარე საბიბლიოთეკო ფუნქციებს.

როგორც უკვე აღვნიშნე დროთა განმავლობაში კომპიუტერული სამყარო დაიყო უამრავ ნაწილებად, თავიდან გამოეყო მომხმარებელი პროგრამისტს, შემდეგ მომხმარებლებიც დაიყო კატეგორიებად, ისევე, როგორც პროგრამისტები; გრაფიკა, მონაცემთა ბაზები, სისტემა, ქსელები, სამეცნიერო და სხვა, ასევე დაიყო თვით პროგრამირების IDE(Integrated Development Environment) გარემოები, ერთადერთი მიზეზის გამო - გაიზარდა მათი მიმართულებათა რაოდენობა და მათი მოცულობა. შედეგად მალე თითქმის შეუძლებელი გახდება ერთდროულად მრავალ მიმართულებაში ეფექტური მუშაობა.

თუმცა ეს ყველაფერი იმას არ ნიშნავს, რომ სიახლეები უნდა მივიღოთ დაუფიქრებლად და ჩავიკარგოთ უკეთესის მიღების იმედით განპირობებული ლოდინის დაუსრულებელ მორევში. უბრალოდ საჭიროა არსებული ცოდნის შესაძლებლობიდან მაქსიმალურის მიღება და ჩვენი შესაძლებლობათა რეალიზაციის ევოლუცია თვითონ მიგვიყვანს საჭირო სიახლეებამდე, მხოლოდ ამ შემთხვევაშია დაცული პროფესიონალიზმი. ამის ცხადი მაგალითია ობიექტურად-ორიენტირებული პროგრამირების ისტორია, სანამ აშკარა არ გახდა ის, რომ საქმე გვქონდა პრინციპულად ადამიანურ ბარიერთან და არა ტექნიკურ, მანამ არ იქნა შემუშავებული ობიექტურად-ორიენტირებული პროგრამირების პარადიგმები, არსებულმა სტრუქტურულმა მიდგომებმა ამოწურეს საკუთარი თავი. ამის შემდეგაც თითქმის ათი წელი დასჭირდა იმას, რომ ეს რევოლუციური სიახლე ეფექტურად გატარებულიყო პრაქტიკაში (Microsoft Windows 1.1).

ისეთი მკვეთრი წინსვლა, როგორიც მოიტანა ობიექტურად-ორიენტირებული პროგრამირების კონცებციებმა, ჯერჯერობით არც ერთ სიახლეს არ შეედრება ამ სფეროში, თუმცა ალბათ მასაც აქვს თავისი შესაძლებლობათა ზღვარი.

უნდა გვახსოვდეს, რომ ერთია სიახლე შესწავლა, ხოლო მეორეა მისი გათავისება, რომელსაც სჭირდება აუცილებლად დრო, ცნობილი ფაქტია პროგრამირებაში ის, რომ სიახლეები მიღებული უნდა იქნას განსაკუთრებული სიფრთხილით, როცა საქმე გვაქვს დიდ პროექტებთან.

გამოცდილება. მახსოვს, რომ ხანში შესული მათემატიკის ლექტორი ლექციის მსვლელობისას მაგალითის ყოველ დეტალს განიხილავდა დაწვრილებით და ყოველგვარი წვრილმანი ელემენტარული ოპერაციის გამოტოვების გარეშე მიუხედავად იმისა, რომ ეს მას გაკეთებული ჰქონდა უკვე მილიონჯერ. დაახლოებითი წარმოდგენა რომ შეგიქნათ, მოვიყვან ერთ ანალოგს. ლაპარაკია მაგალითებზე, რომელთაგან თითოეულის გამოყვანა უხეშად რომ ვთქვათ საჭიროებს არანაკლებ ათ რვეულის ფურცელს:

გამოყვანები იწერებოდა ასეთი სტილით:

$$\cdots \frac{5(a_1+a_2)}{7(b_1+b_2)} = \frac{5a_1+5a_2}{7(b_1+b_2)} = \frac{5a_1+5a_2}{7b_1+7b_2} \cdots$$

ასეთი მიდგომისას შეცდომის დაშვება ძნელია, მისი მოძებნა კი ადვილი.

“რაც ცხადია ალგორითმი, მით უფრო რთულია მასში შეცდომის დაშვება”

პროგრამირების ერთ-ერთი წესი

მაშინ როცა, როგორც წესი ჩვენ სტუდენტები ამას დავწერდით შუალედური ეტაპების გამოტოვებით.

$$\cdots \frac{5(a_1+a_2)}{7(b_1+b_2)} = \frac{5a_1+5a_2}{7b_1+7b_2} \cdots$$

მოგეხსენებათ, რომ უმაღლეს მათემატიკაში უამრავი მსგავსი მომენტია.

მათემატიკაში თუ კიდევ შეიძლება რამენაირად გამართლება, პროგრამირებაში ასეთი ზერელე მიდგომა დაუშვებელია, არამედ საჭიროა პუნქტუალურობა ყველგან და ყოველ დეტალში.

პირველად ვერ ვხვდებოდი ასეთი ჩვევის აზრს, მაგრამ იმდენად რამდენადაც პროგრამირებაც მათემატიკის ნაწილია, დროთა განმავლობაში დავინახე, რომ მრავალწლიანი გამოცდილება პროგრამირებაში არ ნიშნავს იმას, რომ უკვე 1000-ჯერ შედგენილი ესა თუ ის ალგორითმი, რომელიც არ თავსდება გონების მიერ ერთბაშად აღქმის დიაპაზონში 1001-ედ შედგენისას იქნება უშეცდომოდ, იმუშავებს ხარვეზების გარეშე.

ასეთ ალგორითმებს თითქმის ერთნაირი დრო სჭირდებათ, როგორც მეათედ შედგენისას ასევე მეასედ. რატომ? იმიტომ რომ ეს არის ზუსტი მეცნიერება და იგი ყოველთვის მოითხოვს პუნქტუალურობას, ერთი ეტაპიდან მეორეზე გადასვლას თითოეული

დეტალის გამოტოვების გარეშე. ადამიანის აზროვნება კი არ არის განკუთვნილი ფიქსირებული, რობოტული ოპერაციებისათვის, ამისათვის ადამიანში არსებობენ ინსტიქტები, რომლებიც მუშაობენ მხოლოდ შეზღუდული სიდიდის ალგოლითმებზე, მაგალითად ხელის ცხელ საგანზე მოხვედრისას იგი ავტომატურად სცილდება მას ტვინში ყოველგვარი გააზრების გარეშე, ან კიდევ როდესაც იწყებთ წერას თქვენ არ ფიქრობთ იმაზე თუ როგორ უნდა მოხაზოთ ესა თუ ის ასო, არამედ ამას აკეთებენ ინსტიქტები, სანამ არ შეხვდებათ უცნობის სიტუაცია; იგივე სუნთქვა, რომელიც მუშაობს ინსტიქტების წყალობით, თქვენ სრულიადაც არ ფიქრობთ იმაზე თუ როგორ ჩაისუნთქოთ და როგორ ამოისუნთქოთ.

მაგრამ პროგრამირებაში, ერთი და იგივე ალგორითმი, როგორც წესი კეთდება სხვადასვა კონტექსტში სხვადასხვა მიზნისათვის, შესაბამისად ყოველთვის საჭიროა იმ შეგრძნების ქონა, რომ იქ არაფერი არ მეორდება და ყურადღება უნდა იყოს მუდმივ მზადყოფნაში რათა არ მოხდეს შეცდომის დაშვება.

პროგრამირებაში ინსტიქტების გამომუშავება საჭიროა სხვა კონტექსტში და ამაზე დაწვრილებით ვისაუბრებთ წიგნის მეორე ნაწილში.

ასე რომ მრავალწლიანი პრაქტიკა არ ნიშნავს იმას რომ, პროგრამები უნდა წეროთ სწრაფად და დაუფიქრებლად.

ხარვეზების სიმცირე არ არის მნიშვნელოვნად დამოკიდებული პროგრამისტის გამოცდილებაზე, სინამდვილეში პროგრამისტს გამოცდილება ეხმარება მხოლოდ იმაში, რომ პროექტირების დროს დარწმუნებული იყოს ადრინდელი გამოცდილების შედეგად დადგენილი ამა თუ იმ მეთოდის სისწორესა, და წარმატებულობაში.

ხშირად პროფესიონალიზმზე ფიქრს მივყავარ, ხოლმე კურიოზულ სიტუაციამდე, ახლა, რომ მითხრას ვინმემ დამიწერე კვადრატული განტოლების ამოხსნის პროგრამაო, შეიძლება ამას მოვანდომო რამოდენიმე საათი, გამომდინარე იმ პრინციპებიდან რომ ჯერ უნდა გავიაზრო ამოცანის პირობის სრული სქემა, მოვიფიქრო “პროექტის” სტრუქტურა, შემდეგ გავამზადო საჭირო პროგრამული რესურსები, შემდეგ შევიმუშავო სამომხმარებლო ინტერფეისი და კოდის აგების მიმართულება და ა.შ. და ბოლოს მივხვდე რომ ეს ყველაფერი სრულიადაც არ არის საჭირო და დავალების შესრულებას სჭირდება სულ რამოდენიმე წუთი.

ეს რაც შეეხება ხუმრობას, თუმცა თუ თქვენთვის ნაცნობია ასეთი შემთხვევები გაძლევთ გარანტიას რომ დგახართ პროფესიონალ პროგრამისტად ჩამოყალიბების დაუსრულებელ გზაზე.

დასკვნა. თუკი გადავხედავთ პროგრამირების განვითარების ისტორიას, არ არის გამორიცხული, რომ რამოდენიმე ათეული წლის შემდეგ პროგრამისტის ერთ-ერთი წესი არც მეტი და არც ნაკლები ჟღერდეს ასე:

პატივისცემით მოეპყარით CYBER7 სისტემას და გახსოვდეთ რომ მისი ინტელექტუალური ალგორითმები პროექტისადმი თქვენს ზერელე დამოკიდებულებას ვერასოდეს შეეგუებიან !

და ეს სრულიადაც არ იყოს სახუმარო.

მ ე ც ნ ი ე რ ე ბ ა

“თავდაპირველად იყო სიტყვა
და სიტყვა იყო ღმერთთან,
და სიტყვა იყო ღმერთი. . .”
(*მათე. თავი 1.*)

საწყისი. თავდაპიველად არის არსი, ყველაფრის საწყისი, მიზეზთა მიზეზი, ხოლო შემდეგ რეალიზება, განხორციელება, საწყისი სამყაროს სხვაგვარი გამოვლინება, პროექცირება, მსგავსება და ხატება.

ეს კანონთა კანონი მოქმედებს ყველგან სადაც კი ლაპარაკია საგანთა შექმნაზე და მათ არსზე.

გავიმეორებ ადრე დასმულ კითხვას:

რატომ გახდა კომპიუტერი ასე ახლობელი საგანი ადამიანთათვის?

იმიტომ რომ ის შეიცავს ისეთ რაღაცას რაც აქვს ყველა მატერიალურ საგანს, და რითაც ფუნქციონირებს ეს საგანი, ეს არის ლოგიკა, მათემატიკური ლოგიკა - ენა მატერიალური სამყაროსი, ხოლო კომპიუტერში კი მათემატიკა ცოცხლდება და საგნობრივად ადაპტირებულ სახეს იღებს.

“. . . ღმერთმა კარგად იცოდა მათემატიკა როცა სამყაროს ქმნიდა . . .”
შუა საუკუნეების ფილოსოფიიდან

შეიძლება ადამიანს რომელიც ნაკლებადაა დაახლოეული კომპიუტერულ პროგრამირებასთან გაუჩნდეს კითხვა: რატომ ვისწავლო კომპიუტერი მთელი თავისი შესაძლებლობებითა და განშტოებებით თუ კი ის მხოლოდ ადამიანის გამოგონილია და ასეთი კი ამ ქვეყნად ბევრი რამ შეიძლება შეიქმნას.

შესაძლებელია ეს კითხვა ასე ცხადად არა, მაგრამ ქვეცნობიერში მაინც არსებობდეს.

კი მაგრამ მაშინ ისიც ვიკითხოთ რატომ არ გავრცელდა სამობითი მანქანები, თუ კი არსებობენ ორობითები, ისინი ხომ უფრო მეტს გააკეთებდნენ ვიდრე დღევანდელი კომპიუტერები? რატომ არ მოიპოვეს მათ ისეთი პოპულარობა, როგორც ეს ორობითმა სისტემებმა? (დღევანდელი კომპიუტერები მიეკუთვნებიან ე.წ. ორობით მანქანებს, ეს ტერმინი მოდის კომპიუტერის პროგრამული ერთეულის ორ ძირითად მდგომარეობიდან “ლოგიკური 0” და “ლოგიკური 1”).

მიზეზებს თუ მივყვებით პასუხიც მარტივად გაიცემა: იმიტომ რომ ამას განსაზღვრავს ბუნება და მისი კანონები, ეს მოდის ისევ ბუნებიდან და ჩვენ პროგრამისტებს საქმე გვაქვს არა მხოლოდ ხელოვნების ნიმუშის შესწავლა გამოყენებასთან, არამედ ბუნების ერთ-ერთ საინტერესო მოვლენასთან, ორობით აზროვნებასთან, ლოგიკასთან, რომელზეც შეიძლება პროექცირდეს სამყაროში მიმდინარე ნებისმიერი პროცესი.

კომპიუტერი ყველაფერთან ერთად ექსპერიმენტალური დანადგარიცაა. იგი აადვილებს ნააზრევის ექპერიმენტად განხორციელების პროცესს, რაც პატარპატარა აღმოჩენების გაკეთებასთან ასოცირედება. ეს კი თავისებური აზარტია.

როგორ განიმარტება მეცნიერება? ცნობილია მისი ასეთი განმარტება: მეცნიერება არის ის დარგი, რომელსაც:

1. აქვს კვლევის საგანი
2. აქვს კვლევის მეთოდები

ფიზიკა: კვლევის საგანი – სამყარო, კვლევის მეთოდები – მათემატიკური და ემპირიული.

ქიმია: კვლევის საგანი ატომები, მოლეკულები, კვლევის მეთოდები – მათემატიკური და ემპირიული.

მათემატიკა: კვლევის საგანი – რიცხვითი სიმრავლეები, კვლევის მეთოდები – ოპერაციები რიცხვით სიმრავლეებზე.

და ა. შ.

პროგრამირება: კვლევის საგანი – ყველა საგანი რაც კი შეიძლება წარმოიდგინოს პროგრამისტმა, კვლევის მეთოდები – ყველა მეთოდი რაც კი შეიძლება წარმოიდგინოს პროგრამისტმა.

აი შემდეგი კითხვაც: კი მაგრამ ბოლოს და ბოლოს კომპიუტერი ჩაკეტილი სისტემაა და შესაბამისად ამოწურვადი, მაშ სადღაა მისი მრავალფეროვნება?

მართალია, კომპიუტერი ჩაკეტილი სისტემაა, ბოლოს და ბოლოს ლაპარაკია მისი მეხსიერების სასრულ სიდიდეებზე, რიცხვების სასრულ წარმოდგენაზე, მონიტორის სასრულ ზომებზე, ფერების სასრულ რაოდენობაზე და შესაბამისად ამ ყველაფრის სასრულ კომბინაციაზე, ხშირ შემთხვევაში ამან შეიძლება გამოიწვიოს ე.წ. მობეზრების სინდრომის წარმოშობა კომპიუტერის მომხმარებელში. ჩნდება შეგრძნება რომ კომპიუტერში ყველაფერი უაზროდ მეორდება და მასში ახალს და საინტერესოს ვერაფერს იპოვის კაცი. ეს ბუნებრივიცაა.

მე როგორც პროგრამისტს არასოდეს გამჩენია ეს შეგრძნება და არც გამიჩნდება, რადგან მიუხედავად ამ სასრულობისა, კომპიუტერის ზემოთ ჩამოთვლილი მდგომარეობათა

კომბინაცია იმდენად დიდია რომ შეუძლებებელია მათი ყველა გამოვლინება ოდესმე ამოწუროს ადამიანმა. მითუმეტეს, რომ, როგორც უკვე განვიხილეთ საქმე გვაქვს ბუნების კანონებთან და ხელოვნებასთან და პროგრამისტი ამ ხელოვნებაში ავლენს საკუთარ თავს, ხოლო თვით ადამიანი კი ამოუწურავია.

ბევრი ცნობილი ავტორის წიგნში შეიძლება შეგხვდეთ საკითხი იმის შესახებ რომ კომპიუტერული ტექნოლოგიები თავისი ყველა განხრით, ეს იქნება ავტომატიზაცია, სიახლის შემოტანა, კომფორტის გაზრდა თუ სისტემის სხვაგვარი სრულყოფა გაცილებით სწრაფად ვითარდება ვიდრე ნებისმიერი სხვა ტექნოლოგია: ავტომობილები, მშენებლობა, მექანიზაცია და სხვა. ამის მიზეზი მარტივია, Software Development-ი გაცილებით უფრო მოქნილი სისტემაა ვიდრე სხვა ნებისმიერი ტექნოლოგია, მაგრამ გაცილებით უფრო რთულად მოსახელთებელია ორგანიზების თვალსაზრისით ვიდრე დანარჩენი, გარდა ამისა თვითონ ის ფაქტი, რომ პროგრამულ ტექნოლოგიებს ადარებენ სხვა ტექნოლოგიებს, მიუთითებს იმაზე რომ პროგრამული ობიექტები ადამიანთა წარმოსახვაში აღიქმება როგორც რეალური “ხელით შეხებადი” ობიექტები და არა რაღაც არ არსებული აბსტრაქტული სუბსტანცია, რომელზედაც ლაპარაკი არ ღირს გარდა იმისა რასაც ვხედავთ ეკრანზე.

ის, რისგანაც შედგება ნებისმიერი პროგრამა ჩვეულებრივი მატერიალური და თქვენ წარმოიდგინე გარკვეულ პირობებში გრძნობადი ნივთია – ლაპარაკი მაქვს ელექტრულ ველზე.

კომპიუტერი ერთდროულად არის კვლევის იარაღიც და კვლევის ობიექტიც.

ყველასათვის კარგად არის უკვე ცნობლი, რომ მისი საშუალებით ტარდება სერიოზული სამეცნიერო ექსპერიმენტები და ეს ექსპერიმენტი არაა აუცილებელი ჩატარდეს მაინცდამაინც ბიოქიმიაში ან გენურ ინჟინერიაში; კომპიუტერის დაბადებიდან აწუხებს ადამიანს ხელოვნური ინტელექტისა და რობოტოტექნიკის საკითხები და დღეს უკვე დიდი მიღწევებია ამასთან დაკავშირებით: არსებობს ექსპერიმენტები სადაც მიმდინარეობს რომოდენიმე რობოტის თანაარსებობის შედეგად მათ შორის ურთიერთობების თვითჩამოყალიბების პროცესები(პრიორიტეტების განსაზღვრა, თანამშრომლობა და სხვა). ეს არ არის ფანტასტიკა ან მისტიკა, არამედ ეს არის უბრალოდ მეცნიერება, ადამიანისათვის ჯერ კიდევ თითქმის მთლიანად გააზრებად სამყაროში.

ასე რომ, ერთის მხრივ კომპიუტერის მხოლოდ, როგორც პოტენციურად შემოქმედებით არსებად მიღება და მეორეს მხრივ კომპიუტერის მხოლოდ, როგორც გასართობი დანადგარად მიღება არის ილუზიის ნაყოფი და ამ ყველაფერში ბოლომდე გარკვევა კიდევ დროს და რევოლუციურ ცვლილებებს მოითხოვს კომპიუტერულ სამყაროში და ამაში არც არაფერია უჩვეულო, რადგან ილუზიები ყოველთვის ტანჯავენ ადამიანებს.

ინი და იანი. აღმოსავლური ფილოსოფიის გზამკვლევი, და საფუძველთა საფუძველი, ასევე მატერიალური სამყაროს ერთ-ერთი ძირითადი პრინციპთაგანი, ინისა და იანის პრინციპი რამოდენიმე ათეული საუკუნეა რაც ყალიბდებოდა და იაზრებოდა ადამიანთა მიერ სხვადასხვა კონტექსტში.

თვით ჩინელებისა და იაპონელთა ყოველგვარი საქმიანობა, ბრძოლის ხელოვნებიდან დაწყებული და კვების რიტუალებით დამთავრებული ემყარება ინისა და იანის ანუ ურთიერთ საწინააღმდეგოთა თანარსებობისა და ურთიერთ შევსების უნივერსალურ პრინციპს.

ხილული სამყაროს ფუნქციონალობის სქემა -
ინი და იანი
სურ. 1.12

ამ სიმბოლოს განმარტება არ არის ერთი რაღაც კონკრეტული სახის, არამედ იმისდა მიხედვით თუ რა კონტექსტში გამოიყენება, ის(განმარტება) იღებს შესაბამის ფორმას.

თუმცა ცნობილია მისი ზოგადი ფორმულირებაც:

"*წინააღმდეგობები ერთმანეთს კი არ გამორიცხავენ, არამედ ავსებენ*"

შეგახსენებთ რომ შავი რგოლი თეთრში ნიშნავს იმას, რომ როცა თეთრი აღწევს მაქსიმუმს, იმ დროს იბადება შავი და პირიქით თეთრი რგოლი შავში ნიშნავს, რომ როცა შავი აღწევს პიკს მაშინ იბადება თეთრი ანუ ერთის გაქრობა მეორის დაბადების იდენტურია. ამით ამ სიმბოლოში ჩადებულია სისტემის დინამიურობა.

ზოგჯერ ეს სიმბოლო გაგებულია შეცდომით რაც გამოიხატება შემდეგში, რომ მაგალითად ჩვენი ყოველდღიური ცხოვრება მაინცდამაინც ან უკიდურესად კარგია ან უკიდურესად ცუდი, ან მაგალითად ჩვენს გარშემო არიან ადამიანები მხოლოდ ან უკიდურესად ცუდნი ან უკიდურესად კარგნი.

რა თქმა უნდა ეს არასწორია და მეტსაც გეტყვით ასეთი უკიდურესობებს რომ ჰქონდეს ადგილი ეს სამყარო დიდი ხანია არ იარსებედა.

საქმე იმაშია რომ ეს ორი ფერი გამოსახავს ორ ურთიერთ საწინააღმდეგო, მაგრამ სიდიდით ტოლ გამწევ ძალას, სიტყვა "გამწევს" მინდა გავუსვა ხაზი, ანუ სინამდვილეში კონკრეტული სისტემა არსებობს ამ ორი ძალის თანაარსებობის შედეგად და თუ ერთ-ერთი მათგანი გაქრა შედეგად სისტემაც ქრება ანუ მეორეც იკარგება.

წარმოიდგინეთ ვერტიკალურად დაყენებული ჯოხი, რომელსაც ურთიერთ საწინააღმდეგო მიმართულებით ეწევა ორი თანაბარი ძალის მქონე ადამიანი, ერთერთმა თუ გაუშვა ხელი, ან თუ მეორემ მეტისმეტად გადააჭარბა პირველს(რაც იდენტური პროცესებია), მაშინ ჯოხი გადავარდება მეორე უკიდურესობაში და რა თქმა უნდა წაიქცევა ანუ სისტემა სახელად "ვერტიკალური ჯოხი" აღარ იარსებებს, არამედ იარსებებს სრულიად სხვა სახის სისტემა – ჰორიზონტალური ჯოხი.

ორი ურთიერთ საწინააღმდეგო გამწევი ძალა

სურ. 1.13

ასე რომ სისტემის არსებობა ნიშნავს იმას, რომ მასში ეს ორი გამწევი ძალა განაწილებულია თანაბრად, ანუ ფერების მიხედვით თუ ვიმსჯელებთ სისტემა ყოველთვის იმყოფება ნაცრისფერი არის მახლობლად. მაშ, თუ დაირღვა წონასწორობა, სისტემა იწყებს ნგრევას, მაგალითად, თუ ადამიანი მიისწრაფის უკიდურეს თავაშვებული ცხოვრებისაკენ ან თუ იგი მიმართულია უკიდურესი კარჩაკეტილობისაკენ, ეს მას კარგს არაფერს მოუტანს.

ცალკეული ინი და იანი შესდგება ქვე ინი და იანისგან, ცალკეული ინი და იანი არსებობს მისი საკუთარი ქვე ინი და იანის არსებობის წყალობით და ასე ისინი ქმნიან ერთმანეთში ჩალაგებულ სტრუქტურებს, რაც როგორც დასაწყისში ვთქვი მატერიალური სამყაროში მიმდინარე პროცესების სხვადასხვა კონტექსტში სხვადასხვანაირად ვლინდება, მაგრამ პრინციპი ერთია.

რეალურად პრობლემაა, ხოლმე ამ პრინციპის დანახვა კონკრეტულ სისტემაში, მაგრამ მისი გადაწყვეტა გამოცდილებასა და პრაქტიკის მიღებას ემორჩილება.

ეს საკითხები დაწვრილებითაა განხილული არაერთ ლიტერატურაში სადაც კი ლაპარაკია აღმოსავლურ ფილოსოფიაზე, მისი გაგების საუკეთესო ვარიანტი კი ჩემის აზრით აღმოსავლურ საბრძოლო ხელოვნებასთან დაახლოვება და რაც მთავარია მისი გააზრებაა.

პარალელი. ახლა მივყვეთ ლოგიკას და დავუკავშიროთ ეს ყველაფერი ჩვენს პროგრამირებას.

მე ადამიანს მინდა შევქმნა მანქანა რომელიც გათვლებს შეასრულებს ჩემზე გაცილებით უფრო სწრაფად.

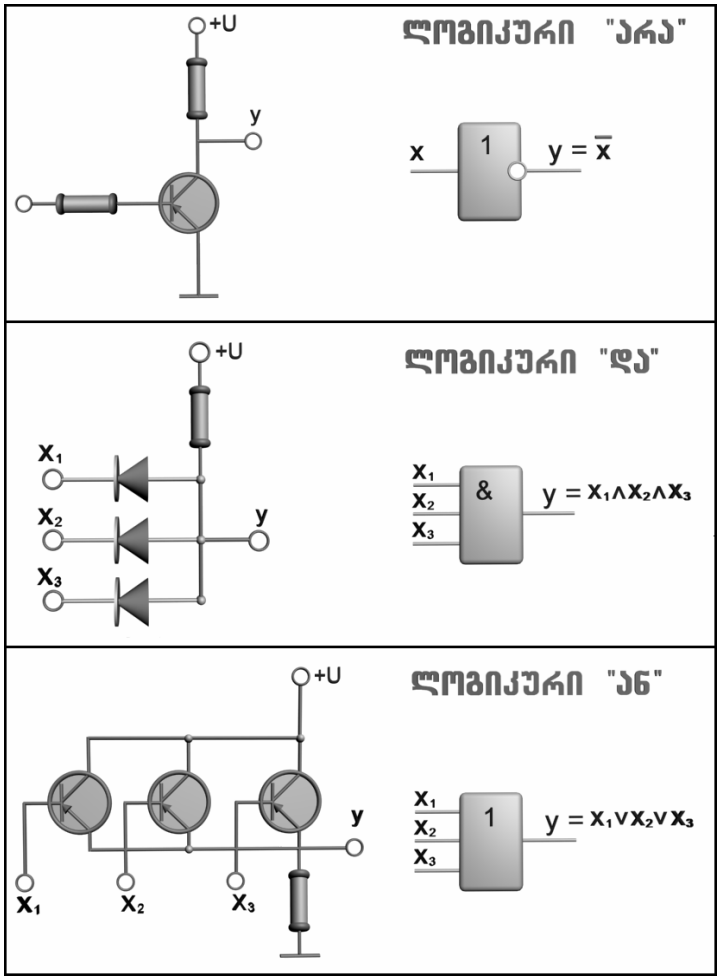
როგორ მოვიფიქრო ასეთი მანქანა?

როგორ და უნდა მივყვე ლოგიკას:

1. - რა მოძრაობს ყველაზე სწრაფად ამ სამყაროში?
   - ელექტრული ველი
2. – ვაკონტროლებ თუ არა მას?
   - დიახ, ვაკონტროლებ, დენის სახით.
3. – მაშ თუ მატერიალური სამყარო ამოდის ინისა და იანის პრინციპიდან, მეც შევქმნი დენისაგან ინსა და იანს და შესაბამისად შეიქმნება ახალი ქვე სამყარო. შესაძლებელია თუ არა ეს?

- უკვე ვიცით, რომ შესაძლებელია !

ახლა მე მჭირდება ისეთი სისტემა რომელიც დენისგან შექმნის ინსა და იანს. და ეს სიტემა ჩვენს შემთხვევაში შეიძლება იყოს მაგალითად ასეთი:



სამი ძირითადი ლოგიკური ელემენტი

სურ. 1.14

ამ სამ ძირითად ლოგიკურ ელემენტზეა დამყარებული მთელი ციფრული სისტემა ანუ კომპიუტერის მყარი ლოგიკური ნაწილი, მარცხნივ გამოსახულია სისტემის ელექტრონულ-ტექნიკური აღწერა, მარჯვნივ კი შესაბამის ლოგიკური აღნიშვნები. კომპიუტერის ყველა დანარჩენი ელემენტი აიგება მათი რთული კომბინაციების საშუალებით. სხვათაშორის ისინი წარმოადგენენ ტვინის ნეირონების ერთ-ერთ ძირითად ფუნქციის – აღგზნებულ და მოშვებულ მდგომარეობების სიმულატორებს. რა თქმა უნდა თანამედროვე პროცესორები ამ ზომის დეტალებზე უკვე კარგა ხანია აღარაა დაფუძვნებული, არამედ სისტემა დაყვანილია მოლეკულების დონეზე, მაგრამ ის მოლეკულური სტრუქტურებიც ანხორციელებენ ზუსტად იგივე ლოგიკას რასაც ეს სქემები.

შევნიშნოთ რომ აქ X1, X2, და X3 შეიძლება იყოს მხოლოდ ორიდან ერთერთ მდგომარეობაში “ლოგიკური 0” ან “ლოგიკური 1”. ეს საკითხები დაწვრილებითაა განხილული ციფრული ელექტრონიკისა და ორობითი ალგებრისათვის (ბულის ალგებრისათვის) განკუთვნილ ლიტერატურებში.

ახლა ავიღო ძაბვის ერთი მნიშვნელობა, მაგალითად 2 ვოლტი და გამოვყოთ მისგან ორი მიმართულება:

1.5 და 2.5 ვოლტი

პირველს დავარქვათ

“ლოგიკური 0” (ინი), - არაფერი არ არსებობს

მეორეს დავარქვათ

“ლოგიკური 1” (იანი) – ყველაფერი არსებობს

და ამ ორი ურთიერთსაწინააღმდეგო მდგომარეობის მიხედვით შევქმნა მატერიალური სამყაროს ერთ-ერთი პატარა ქვესამყარო პროგრამირების სამყაროს სახით.

...01000111 00110011 01000010 00101110 01000111 01000101...

თუ ერთ-ერთი მათგანი მაინც გაქრა, მთელი პროგრამირების სამყარო გაქრება მეყსეულად, ანუ ეს ორი პროცესი იდენტურია და ერთი და იგივე რამეს ნიშნავს.

ზოგადად მნიშვნელობა აქვს იმას თუ რა დონის ძალაა ცალკეული ინი და ცალკეული იანი, ან უფრო სწორად «შკალის» რა ნაწილში იმყოფება თითოეული მათგანი. ეს მეტად მნიშვნელოვანი შენიშვნაა განსაკუთრებით იმათთვის ვინც თვლის რომ ურთირთ საწინააღმდეგოთა არსებობა არ არის აუცილებელი მოცემული სისტემის არსებობისათვის.

ასე რომ ზემოთ მოყვანილი ინტერპრეტაციები სულაც არ არის აბსტრაქტული და საკმაოდ სერიოზული საფუძვლები გააჩნია, რაც მიუთითებს იმაზე რომ კომპიუტერული პროგრამირება, როგორც დამოუკიდებელი სფერო, სრულად წარმოადგენს არა მარტო უბრალოდ ხელობას ან იარაღს რაღაცის გადასწყვეტად, არამედ დამოუკიდებელ, ლამაზ, მიკრო სამყაროს. 1 და 0 რეალურად არსებული ობიექტებია და მათი და მათგან წარმოებული სისტემების ფილოსოფიურად განხილვა სავსებით მართებულია.

რაც შეეხება ინსა და იანს მინდა შევნიშნო, რომ მან არ უნდა დაგვაბნიოს და არ უნდა გავიგოთ ის რაღაც დამახინჯებული სახით, ის ისეთივე ფიზიკის კანონია, როგორც მაგალითად ნიუტონის ცნობილი მეორე კანონი, ოღონდ უფრო ზოგადი და ძნელად შესამჩნევი, რომელიც ჩანს სხვადასხვა კონტექსტი სხვადასხვანაირად, ის არსებობს, ყველგანაა, ბევრ რამეს ხსნის, მიმზიდველია, მაგრამ არ წარმოადგენს იმ აბსოლუტს, რომელზეც ადამიანს შეუძლია ბოლომდე უსაფრთხოდ გავიდეს;

დასკვნა. პრინციპში სულ ეს იყო რაც მინდოდა მეთქვა ზოგადად პროგრამირების პატარა სამყაროს შესახებ, მასში არსებულ შესანიშნავ მოვლენებზე და პრობლემურ საკითხებზე.

იმედია ცოტათი მაინც დაინტერესდით აქ მოყვანილი მაგალითებითა და მოსაზრებებით, გაიგეთ რაიმე ახალი ან სხვა თვალით შეხედეთ თქვენთვის ნაცნობ სიტუაციებს ან მოუძებნეთ

რაიმე ახსნა მათ, გამომდინარე აქ წაკითხულიდან, თუ ეს ასეა, მაშინ შემიძლია ჩავთვალო რომ წიგნის ამ ნაწილმა თავისი მოვალეობა პირნათლად შეასრულა.

შემდეგ ნაწილში კი განხილული იქნება დიდ პროექტებთან მუშაობის საკვანძო საკითხები და მათში არსებული ხარვეზების აღმოფხვრის სხვადასხვა ხერხები.