გია სურგულაძე

კორპორაციული მენეჯმენტის სისტემების Windows დეველოპმენტი: Workflow ტექნოლოგია

საქართველოს ტექნიკური უნივერსიტეტი

გია სურგულაძე

კორპორაციული მენეჯმენტის სისტემების Windows დეველოპმენტი: Workflow ტექნოლოგია

(ლაბორატორიული პრაქტიკუმი, ნაწ.2)

დამტკიცებულია:
სტუ-ს სარედაქციო-
საგამომცემლო
საბჭოს მიერ

თბილისი
2015

უაკ 004.5

განიხილება კორპორაციული მენეჯმენტის პროცესების ავტომატიზაციის საფუძვლები ახალი ინფორმაციული ტქნოლოგიების ბაზაზე. კერძოდ შემოთავაზებულია MsVisual Studio.NET Framework 4.0/5 ინტეგრირებულ გარემოში ჰიბრიდული კომპიუტერული სისტემების (Windows- და Web-აპლიკაციების) ვიზუალური დაპროგრამების ინსტრუმენტული საშუალება Workflow Foundation (WF) ტექნოლოგიის ბაზაზე. იგი ეფუძნება XAML (სისტემის დიზაინის ნაწილი) და C# (სიტემის ლოგიკური ნაწილი) ენების კომპლექსურ გამოყენებას. წარმოდგენილია ლაბორატორიული პრაქტიკუმის ამოცანები და მეთოდური ინსტრუქციები ასეთი სისტემების კომპონენტების დასაპროგრამებლად.

დამხმარე სახელმძღვანელო ლაბორატორიული პრაქტიკუმის სახით განკუთვნილია ინფორმატიკისა და მართვის საინფორმაციო სისტემების სპეციალობის ბაკალავრიატის მაღალი კურსის სტუდენტების და მაგისტრანტებისთვის.

რეცენზენტები:

- პროფ. ე. თურქია
- პროფ. გ. ღვინეფაძე

პროფ. გ. სურგულაძის რედაქციით

ISBN 978-9941-0-7103-4 (ყველა ნაწილის)
ISBN 978-9941-0-7520-9 (მეორე ნაწილის)

შინაარსი

შესავალი

აპლიკაციების (დანართების) ორი ნაირსახეობაა ცნობილი: ვინდოუს სისტემები, რომელთაც ასევე სამაგიდო აპლიკაციებს უწოდებენ და ვებ-აპლიკაციები, რომელთა გამოყენებაც ინტერნეტ ბრაუზერებიდანაა შესაძლებელი [1-3].

ეს დანართები იქმნება .NET Framework -ის ორი სხვადასხვა პაკეტით. პირველი - Windows Forms კომპონენტებით და მეორე ASP.NET -ის საშუალებით. ორივეს აქვს თავისი უპირატესობები და ნაკლოვანებანი. კერძოდ, სამაგიდო დანართები ძალზე მოქნილი და რეაქციულია, ხოლო Web-დანართები ინტერნეტის საშუალებით იძლევა დისტანციური წვდომის საშუალებას ერთდროულად მრავალი მომხმარებლისთვის. მაგრამ თანამედროვე კომპიუტერული ტექნოლოგიების სამყაროში ამ ორი სახის აპლიკაციებს შორის საზღვრები სულ უფრო და უფრო იშლება [4].

Web-სამსახურების და WCF (Windows Communication Foundation) სერვის-ორიენტირებული არქიტექტურის აგების საშუალებების გაჩენამ განაპირობა სამაგიდო- და ვებ-დანართების ფუნქციონირების შესაძლებლობა ერთიან განაწილებულ გარემოში, სადაც მონაცემთა გაცვლა ხორციელდება როგორც ლოკალურ, ასევე გლობალურ ქსელებში. 1-ელ ნახაზზე ნაჩვენებია ეს გამაერთიანებელი პროცესი.

ნახ.1

WPF – Windows Presentation Foundation ერთ-ერთი ასეთი გამაერთიანებელი ტექნოლოგიაა და აგბს ისეთ დანართს, რომელშიც გამორიცხულია დაპირისპირება სამაგიდო აპლიკაციას და ინტერნეტს შორის.

WPF-დანართს შეუძლია ფუნქციონირება როგორც სამაგიდო აპლიკაციის, ასევე როგორც ვებ-აპლიკაციას ბრაუზერის შიგნით.

არსებობს ასევე WPF-ის შეზღუდული ვერსია, სახელით Silverlight, რომლითაც შესაძლებელია ვებ-დანართში დინამიკური მდგენელის დამატება.

WF (Workflow Foundation) ტექნოლოგია .NET-ში არის სრულიად ახალი პარადიგმა სამუშაო (ბიზნეს) პროცესებზე (workflow) ბაზირებული აპლიკაციების ასაგებად. იგი ფუნდამენტურად ახლად გააზრებული ტექნოლოგიაა. თავიდან ჩვენ გავეცნობით მარტივი სამუშაო პროცესის აგების საფუძვლებს და ძირითად ცნებებს, შემდეგ კი განვიხილავთ შედარებით რთულ ამოცანებსაც.

წინამდებარე სახელმძღვანელო, ფაქტობრივად, არის [19] წიგნის „კორპორაციული მენეჯმენტის სისტემების Windows-დეველოპმენტის WPF ტექნოლოგია“ გაგრძელება, მისი მეორე ნაწილი. იგი დეტალურად განიხილავს Workflow Foundation ტექნოლოგიის ძირითად ცნებებს, საბაზო სტრუქტურებს და მათი გამოყენების მეთოდოლოგიურ საკითხებს, ვიზუალური დაპროგრამების პრინციპების თვალსაზრისით.

ნაშრომის სამ თავში შემოთავაზებულია 15 ლაბორატორიული სამუშაო, შესაბამისი პრაქტიკული ამოცანებით და ექსპერიმენტული სავარჯიშოებით, რომელთა სირთულის მიხედვით შესაძლებელია მათი გამოყენება ბაკალავრიატის და მაგისტრატურის ჯგუფებში, სპეციალობით მართვის საინფორმაციო სისტემების პროგრამული ინჟინერია.

I თავი
ბიზნესპროცესების (Workflows) აგების ელემენტები

1.1. მარტივი Workflow-ის აგება (ლაბ.N1)

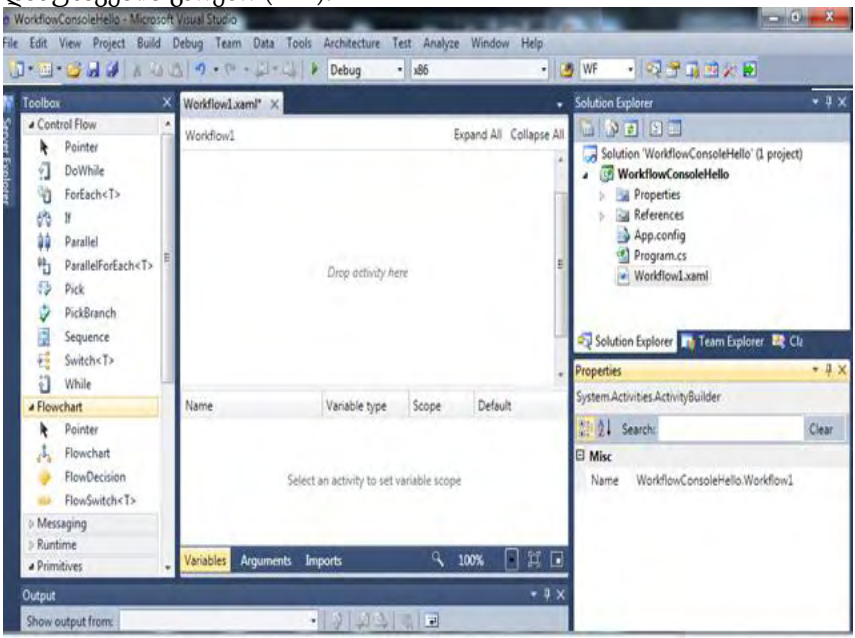
მიზანი: Visual Studio.NET პაკეტის Workflow Foundation-ის სამუშაო გარემოს გაცნობა და პირველი აპლიკაციის აგება.

დავიწყოთ მარტივი სამუშაო პროცესის (workflow-ის) შექმნა Visual Studio.NET გარემოში. შევარჩიოთ ახალი პროექტის სახელი (მაგალითად, WorkflowConsoleHello), მისი შენახვის ადგილი (C:\WF\). აგრეთვე ავირჩიოთ Template-ში Visual C# და Workflow, ხოლო შუა ფანჯრიდან Workflow Console Application (ნახ.1.1):

ნახ.1.1. ახალი workflow პროექტის შექმნა

შაბლონი (Template) აგენერირებს Program.cs ფაილს, რომელიც რეალიზებას უკეთებს კონსოლურ აპლიკაციას (დანართს). იგი ასევე აგენერირებს Workflow1.xaml ფაილს, რომელიც განსაზღვრავს ქმედებას (აქტიურობას) სამუშაო პროცესში (workflow-ში). XAML ენა გამოიყენება პროგრამული ელემენტების გამოსაცხადებლად (როგორც WPF აპლიკაციაში). ოღონდაც ლებელის, ტექსტბოქსის და ბადის ნაცვლად ეს ფაილი შეიცავს წარმოებული ელემენტების აქტიურობებს ჩვენს მიერ განსაზღვრულ სამუშაო პროცესში. Visual Studio იძლევა დიზაინერისთვის ქმედებების (აქტიურობების) გრაფიკულად ნახვის და რედაქტირების საშუალებას.

1.2 ნახაზზე ნაჩვენებია Visual Studio–ის ინტეგრირებული დამუშავების გარემო (IDE).



ნახ.1.2. Visual Studio-ს სტანდარტული IDE გარემო

მარცხნივ მოთავსებულია ინსტრუმენტების პანელი, რომელიც მოიცავს ჩადგმულ და მომხმარებლის აქტიურობებს, რომელთა გაფართოება შესაძლებელია სხვა ქმედებებითაც. Solution Explorer და თვისებათა ფანჯარა მარჯვნივაა მოთავსებული. ქვემოთ ფანჯარაში გამოიტანება შეცდომების შეტყობინებები, შუალედური შედეგები და სხვა ინფორმაცია.

WF 4.0 დიზაინერი მოთავსებულია შუაში. ქვედა მარჯვენა კუთხეში არის მასშტაბირების კომბობოქსი, სურათის გამადიდებელი და აქტიურობის (ქმედების) მოსაძებნი ღილაკები. ქვედა მარცხენა კუთხეში სამი ელემენტია: ცვლადების, არგუმენტების და იმპორტული ანაწყობის. თუ მუშა პროცესი კლასია, მაშინ ცვლადები იქნება კლასის წევრები. მათი ნახვა შეიძლება გახსნით (ნახ.1.3).

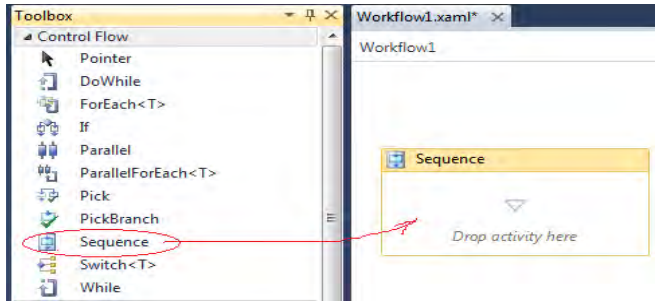
ნახ.1.3. სამუშაო პროცესის ცვლადების ნახვა

- **სამუშაო პროცესის (workflow-ის) დაპროექტება**

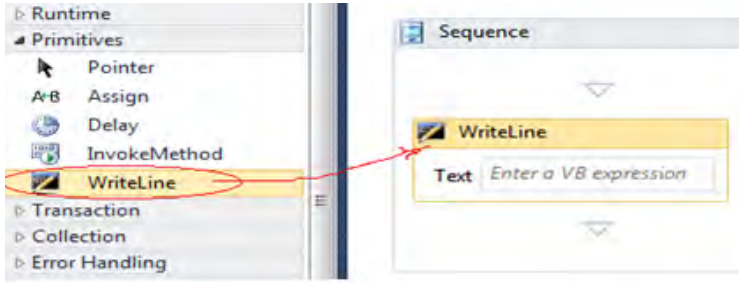
თავიდან სამუშაო ნაკადის კონსტრუქტორი ცარიელია. ინსტრუმენტების პანელიდან საჭირო აქტიურობა გადაიტანება დიზაინერის გარემოში, რითაც განისაზღვრება სამუშაო პროცესის ყოფაქცევა.

ჩვენი პროექტი თავიდან უნდა ასახავდეს მისალმებას „Hello, World !“, შემდეგ კი დაემატება სხვა პროცედურები. გადმოვიტანოთ Sequence ქმედება დიზაინერზე (ნახ.1.4). შემდეგ კი – WriteLine ქმედება ამ Sequence–ზე. სქემას ექნება 1.5 ნახაზზე ნაჩვენები სახე.

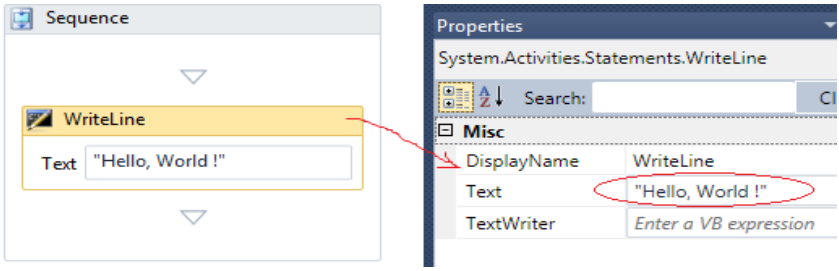
WriteLine-ის Properties-ში ტექსტის შეტანა ნაჩვენებია 1.6 ნახაზზე. სისტემის ამუშავების შემდეგ საბოლოო შედეგი მოცემულია 1.7 ნახაზზე.



ნახ.1.4. Sequence ქმედების გადმოტანა



ნახ.1.5. WriteLine ქმედების დამატება Sequence-ში



ნახ.1.6. Text –ის მნიშვნელობის შეტანა Properties-ში

ნახ.1.7. კონსოლზე გამოსული შედეგი

გავხსნათ პროგრამა Program.cs, რომელიც ამუშავებს კონსოლის აპლიკაციას (ლისტინგი_1.1):

```
// -- ლისტინგი_1.1 ----
using System;
using System.Linq;
using System.Activities;
using System.Activities.Statements;

namespace WorkflowConsoleHello
{
   class Program
   {
      static void Main(string[] args)
      {
         WorkflowInvoker.Invoke(new Workflow1());
        // Console.WriteLine("Press ENTER to exit");
        // Console.ReadLine();
      }
   }
}
```

პროგრამაში ხელითაა ჩამატებული ბოლო ორი სტრიქონი (კომენტარი მოხსენით), რათა შესაძლებელი იყოს შედეგის ნახვა კონსოლზე.

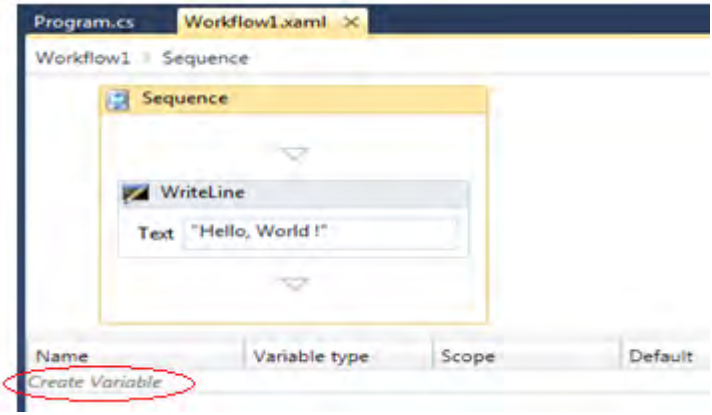
1.2. პროცედურული ელემენტები (ლაბ.N2)

მიზანი: Visual Studio.NET პაკეტის WF-ის ინსტრუმენტების პანელის ძირითადი პროცედურული ელემენტების გამოყენების გაცნობა.

WF-ს აქვს პროცედურული ელემენტები, როგორიცაა, მაგალითად, If, While, Assign, Sequence და სხვა. მათი ფუნქციონირების სადემონსტრაციოდ გამოვიყენოთ ზემოთ აგებული მისალმების კოდი. წარმოვადგინოთ ძველებური საათის მექანიზმი, რომელიც ყოველ საათზე გამოსცემს ზარს. გავხსნათ Workflow1.xaml ფაილი.

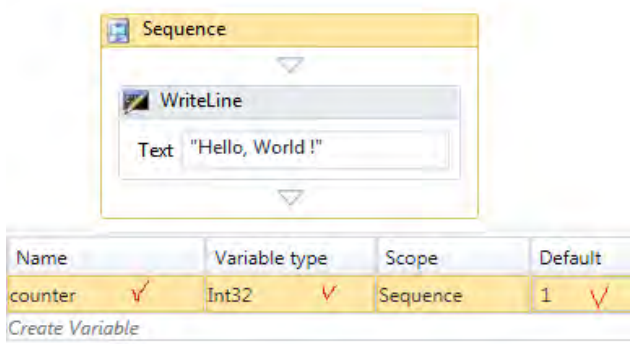
- **ცვლადების გამოყენება**

WF–ში უნდა გამოვაცხადოთ ყველა ცვლადი, რომელიც გამოიყენება მუშა ელემენტებში. ჩვენ შემთხვევაში საჭირო იქნება ორი ცვლადი: ერთი – ზარების რაოდენობისთვის, მეორე – მთვლელისთვის, რომელიც დაითვლის თუ რამდენი ზარი იქნა ამოქმედებული აქამდე. დავაჭიროთ ღილაკს Variables. თუ მასში არაა ელემენტები, ე.ი. არ მომხდარა მათი გამოცხადება. ახლა დავდგეთ მთავარ ქმედებაზე – Sequence, ცვლადების ფანჯარას ექნება 1.8 ნახაზზე ნაჩვენები სახე.



ნახ.1.8

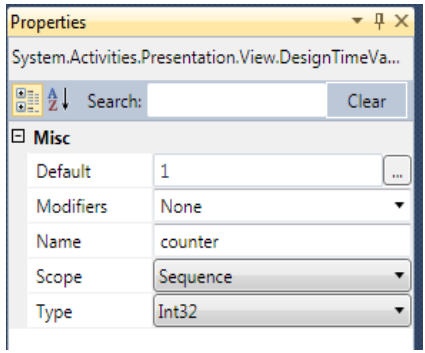
დავკლიკოთ *Create Variable* და შევიტანოთ ცვლადის სახელი, ტიპი და მნიშვნელობა (ნახ.1.9).



ნახ.1.9

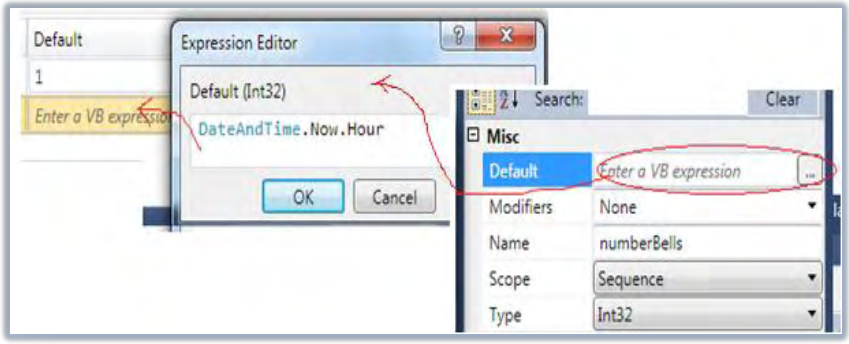
ცვლადი counter ხილვადია როგორც Sequence ქმედების, ისე მისი შვილობილი ელემენტებისთვის (მაგალითად, WriteLine). ცვლადის მნიშვნელობა შევიტანეთ 1.

1.10 ნახაზზე ნაჩვენებია ცვლადების სტრიქონის Properties-ის ფანჯარა. აქაც შესაძლებელია ცვლადების მნიშვნელობათა შეტანა, მაგალითად, Default-ში „1“.



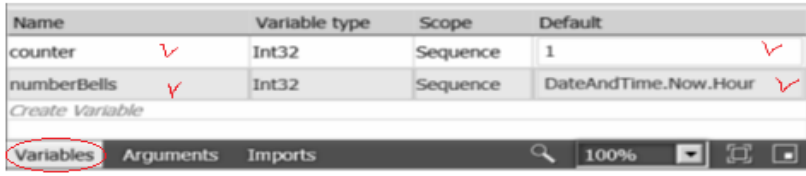
ნახ.1.10

მეორე სტრიქონში, შეიტანება ზარების რაოდენობის ცვლადის მნიშვნელობა, რომელიც გამოიყენებს „*Enter a VB expression*“-ს (ნახ.1.11).



ნახ.1.11

საბოლოო შედეგს ცვლადებისთვის ექნება 1.12 ნახაზზე ნაჩვენები სახე.



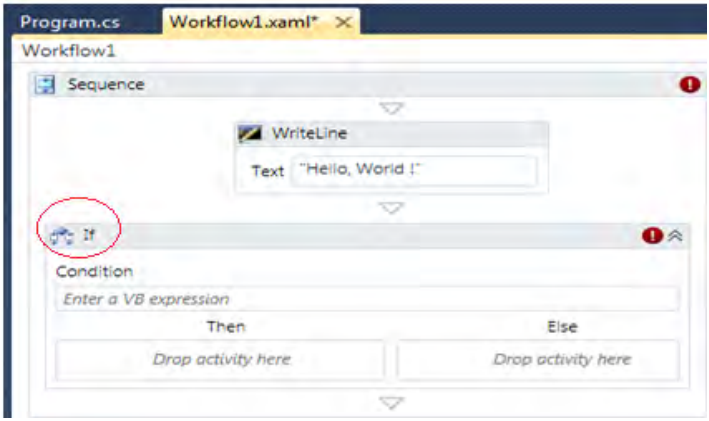
ნახ.1.12

- **If - განშტოების პროცედურა**

DateAndTime კლასის Hour-წევრი აბრუნებს დროის მნიშვნელობას 24–საათიანი ფორმატით. მაგალითად, 2 PM–თვის ის დააბრუნებს 14–ს. ამიტომაც ჩვენ უნდა ავაწყოთ სისტემა ისე, რომ ზარი იყოს 2–ჯერ და არა 14–ჯერ. ამისთვის კოდში უნდა ჩაიწეროს შემდეგი:

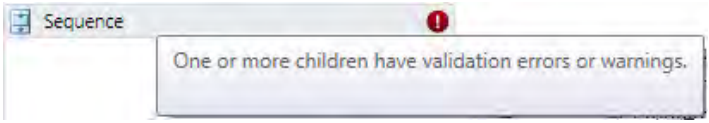
```
if (numberBells > 12)
        numberBells -= 12;
```

ამ მიზნით გამოიყენება if და Assign ქმედებები. გადმოვიტანოთ ინსტრუმენტების პანელიდან if აქტიურობა Hello აქტიურობის ოდნავ ქვემოთ. დიაგრამა მოცემულია 1.13 ნახაზზე.

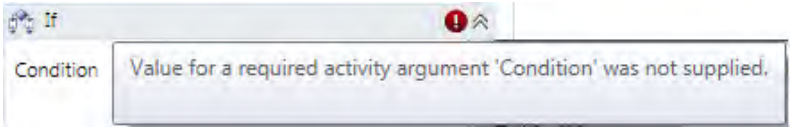


ნახ.1.13. if ქმედების დამატება

<u>შენიშვნა:</u> ნახაზზე წითელი მახილის ნიშნით მიეთითება, რომ არის შეცდომა/გაფრთხილება. მაუსის კურსორის მიტანით ჩნდება ტექსტი (ნახ.1.14, 1.15).



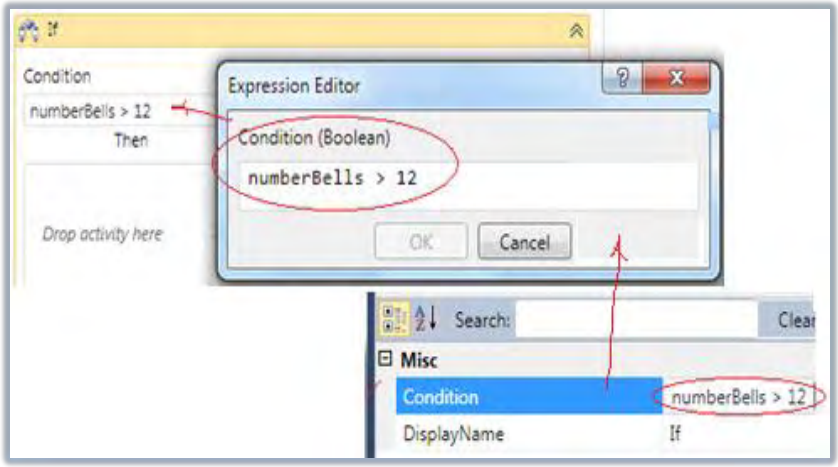
ნახ.1.14. გაფრთხილება Sequence-ში, რომ შვილობილს აქვს შეცდომა



ნახ.1.15. გაფრთხილება if-ში, რომ ქმედებას არ აქვს მითითებული პირობა

თვისებების ფანჯარაში შევცვალოთ DisplayName–მნიშვნელობა Adjust for PM–ით. if ქმედება შედგება სამი ელემენტისგან. პირობა Condition განსაზღვრავს ლოგიკას, რომელიც შეფასდება. ეს იქნება ლოგიკური მნიშვნელობა („ჭეშმარიტი“ ან „მცდარი“). იგი შეიცავს ქმედებებს, რომლებიც სრულდება, როცა პირობა

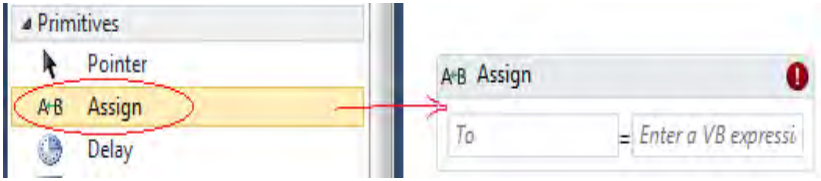
„ჭეშმარიტია“, ან სხვა ქმედებებს, როცა პირობა „მცდარია“. მხოლოდ ერთია აუცილებელი. შევიტანოთ პირობა numberBells> 12 (ნახ.1.16).



ნახ.1.16. Condition პირობის შეტანა

- **Assign - მინიჭების პროცედურა**

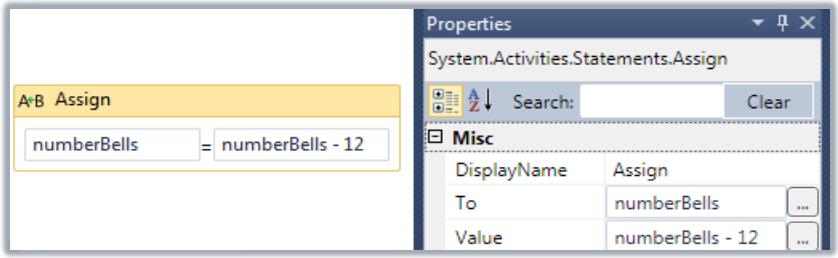
გადავიტანოთ ინსტრუმენტების პანელიდან Assign ქმედება. იგი საშუალებას იძლევა ცვლადს ან არგუმენტს მივანიჭოთ მნიშვნელობა. გრაფიკულად მიიღება ასეთი სურათი (ნახ.1.17).



ნახ.1.17. Assign ქმედების დამატება

Assign–ში To და Value, ორივე ღებულობს მნიშვნელობას. შეტანა შეიძლება უშუალოდ ველში, ან Properties-ის ფანჯარაში (“...“-ით გამოიძახება გამოსახულების შეტანის რედაქტორი). თვისებისთვის (To) შევიტანოთ numberBells. თვისების

მნიშვნელობისთვის (Value) კი: numberBells–12. თვისებათა ფანჯარას ექნება ასეთი სახე (ნახ.1.18).

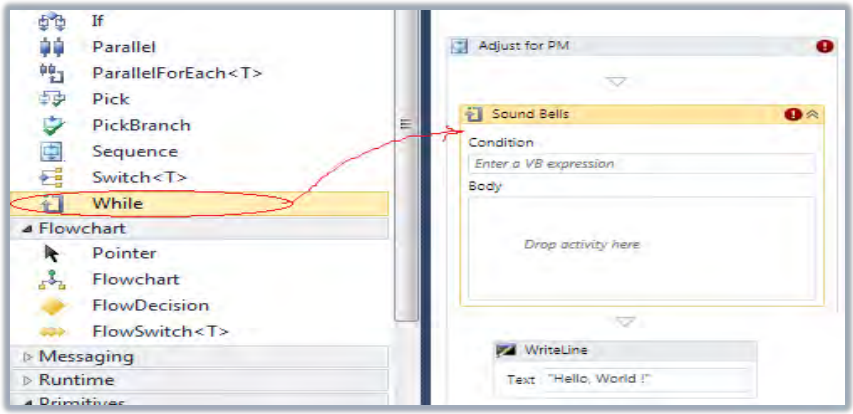


ნახ.1.18. Assign ქმედების თვისებათა ფანჯარა

მრავალი ქმედება არის შედგენილი აქტიურობა (ქმედება), რაც ნიშნავს, რომ იგი შეიძლება შეიცავდეს სხვა სახის ქმედებებს.

- **While - ციკლის პროცედურა**

დავამატოთ While ქმედება ზარის დასარეკად. გადმოვიტანოთ ინსტრუმენტების პანელიდან While აქტიურობა „Adjust for PM"–ის ქვემოთ. შევცვალოთ თვისებებში DisplayName სახელით Sound Bells (ნახ.1.19).



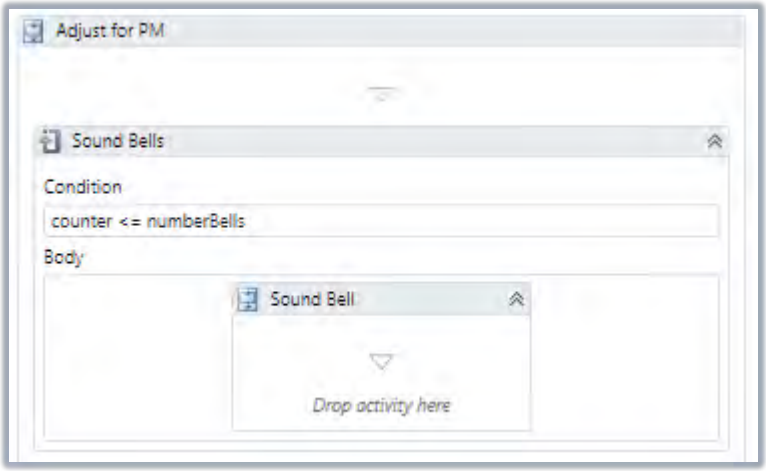
ნახ.1.19. While ქმედების დამატება Sound Bell დასახელებით

While ქმედებაში Body სექციის მოქმედება სრულდება მანამ, სანამ პირობა (Condition) ჭეშმარიტია. თავიდან პირობა მოწმდება და თუ ის არის „true", მაშინ ქმედებები სრულდება. ეს მეორდება მანამ, სანამ პირობა გახდება "false".

შენიშვნა: DoWhile ქმედება იდენტურია While ქმედების, ოღონდაც ჯერ აქტიურობა სრულდება ერთხელ და შემდეგ მოწმდება პირობა.

შევიტანოთ Condition (პირობა) **counter <= numberBells.**

გადმოვიტანოთ ინსტრუმენტების პანელიდან Sequence ქმედება Body-სექციაში. მისი DisplayName შევცვალოთ Sound Bell-ით. მივიღებთ 1.20 ნახაზზე ნაჩვენებ სურათს.



ნახ.1.20. While ქმედება შეიცავს Sequence-ს

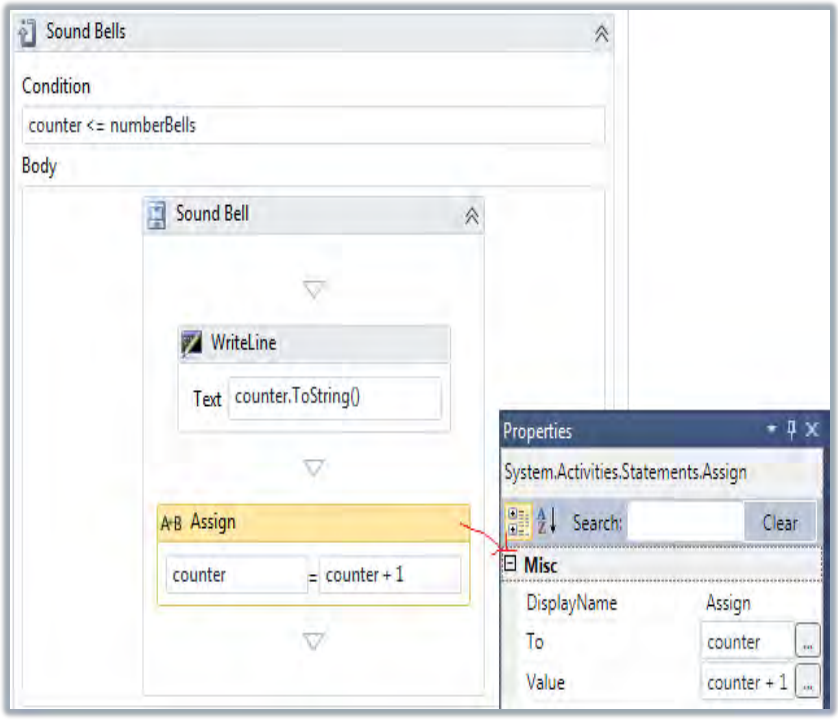
- **Sequence - პროცედურა**

გადმოვიტანოთ სამი სახის ქმედება Sequence–ზე „Sound Bell". ამ სავარჯიშოში ზარი არ დარეკავს, მაგრამ კონსოლის სტრიქონი გამოიტანს ზარის რეკვის რაოდენობას. გადმოვიტანოთ „Sound Bell"–ში WriteLine ქმედება. თვისებების Text-ში შევიტანოთ ბრძანება:

counter.ToString()

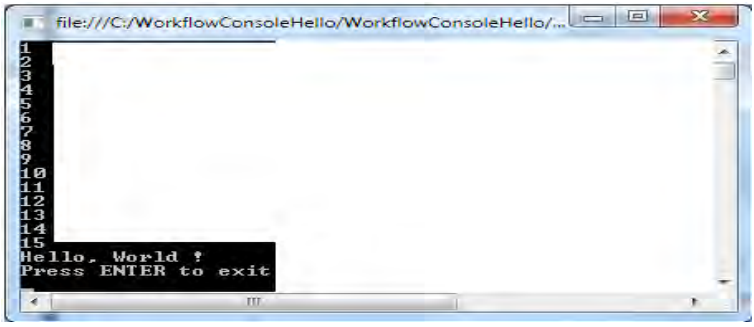
იგი გამოიტანს კონსოლზე მთვლელის მიმდინარე მნიშვნელობას.

გადმოვიტანოთ Assign ქმედება WriteLine ქმედების ქვემოთ. To თვისებისთვის შევიტანოთ counter; Value თვისებისთვის კი counter+1. ესაა მარტივი ინკრემენტის მაგალითი (ნახ.1.21).



ნახ.1.21. საბოლოო Sequence დიაგრამის კომპლექტი

სისტემის ამუშავების შემდეგ კონსოლზე გამოჩნდება ასეთი შედეგი (ნახ.1.22).



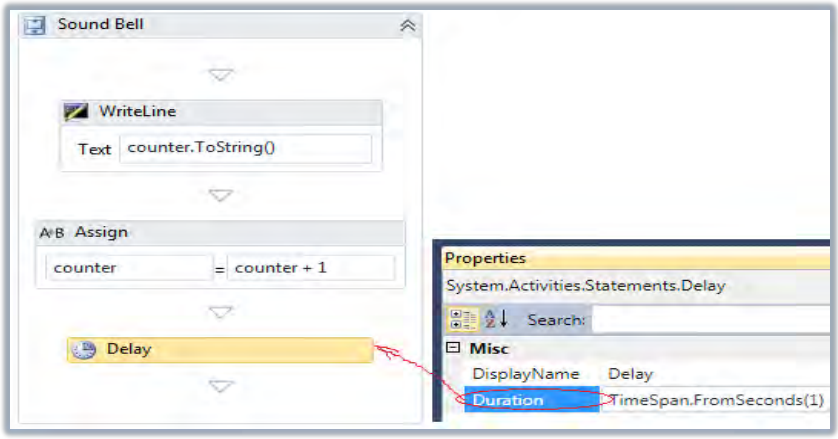
ნახ.1.22. შედეგი

- **Delay ქმედება**

გადმოვიტანოთ ინსტრუმენტების პანელიდან დაყოვნების Delay ქმედება Assign ქმედების ქვემოთ. დაყოვნების აქტიურობა განსაზღვრავს პაუზის ხანგრძლივობას. იგი მიეთითება როგორც TimeSpan კლასი. შევიტანოთ შემდეგი გამოსახულება: (ნახ.1.23).

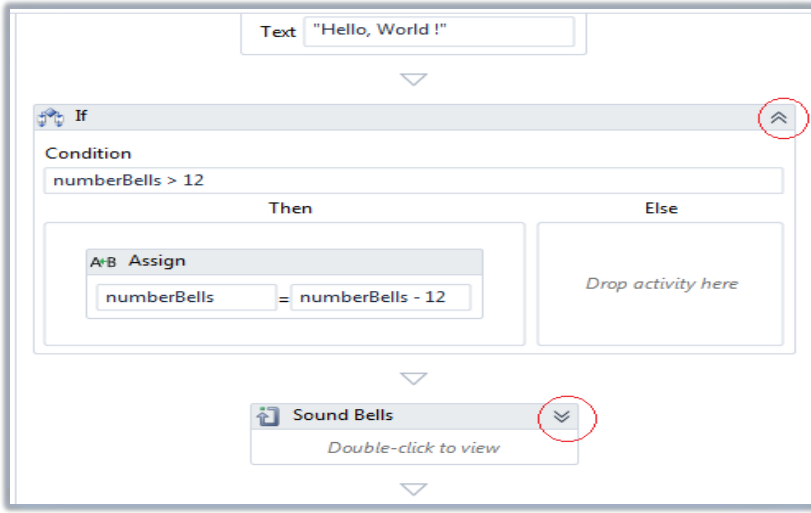
TimeSpan.FromSeconds(1)

მივიღებთ 1.23 ნახაზზე მოცემულ სურათს.



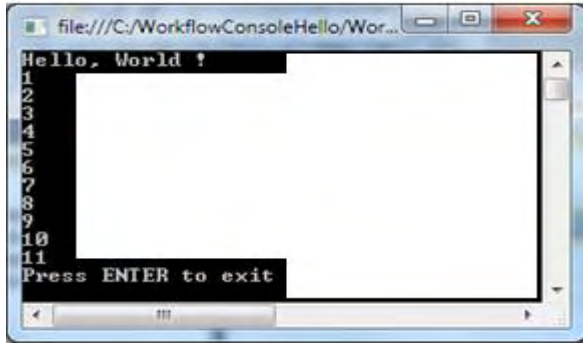
ნახ.23. დაყოვნების ქმედების დამატება Sequence-ში

1.24 ნახაზზე რგოლებით შემოხაზულია ჩაკეცვა–გაშლის ისრები. Sound Bell-სთვის (ნახ.1.23) ის ჩაკეცილია და ჩანს მხოლოდ სათაური.



ნახ.1.24

თუ ავამუშავებთ ამ სიტუაციას, მივიღებთ 1.25 სურათს.



ნახ.1.25

რიცხვები 1–11 გამოიტანება მიმდევრობით, ოდნავ დაყოვნებით.

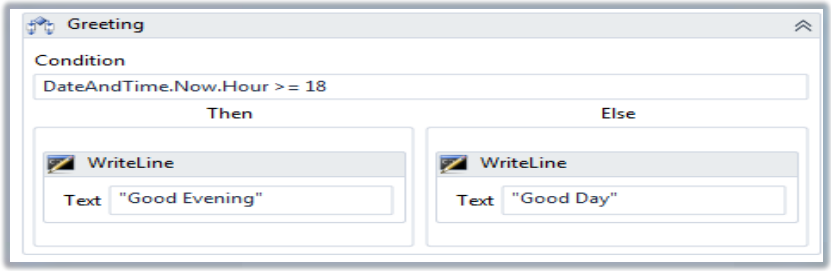
ახლა გადმოვიტანოთ WriteLine ქმედება Sound Bell-ის ქვემოთ. შევცვალოთ DisplayName სახელით Display Time. თვისებისთვის Text შევიტანოთ გამოსახულება:

„The time is: " + DateAndTime.Now.ToString()

გადმოვიტანოთ if ქმედება „Display Time"–ს ქვემოთ და DisplayName შევცვალოთ Greeting–ით. Condition პირობისთვის შევიტანოთ გამოსახულება:

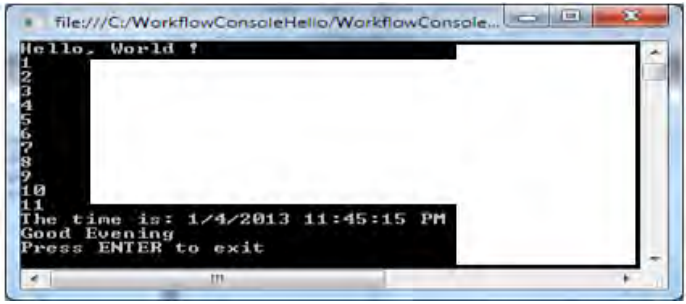
DateAndTime.Now.Hour >= 18

გადმოვიტანოთ WriteLine ქმედება ორივე Then და Else სექციებში. Then სექციისთვის Text-ში შევიტანოთ „Good Evening"; Else სექციისთვის Text-ში შევიტანოთ „Good Day". ამგვარად, „Greeting" აქტიურობა მიიღებს ასეთ სახეს (ნახ.1.26).



ნახ.1.26. Greeting ქმედება

ავამუშავოთ აპლიკაცია F5-ით.

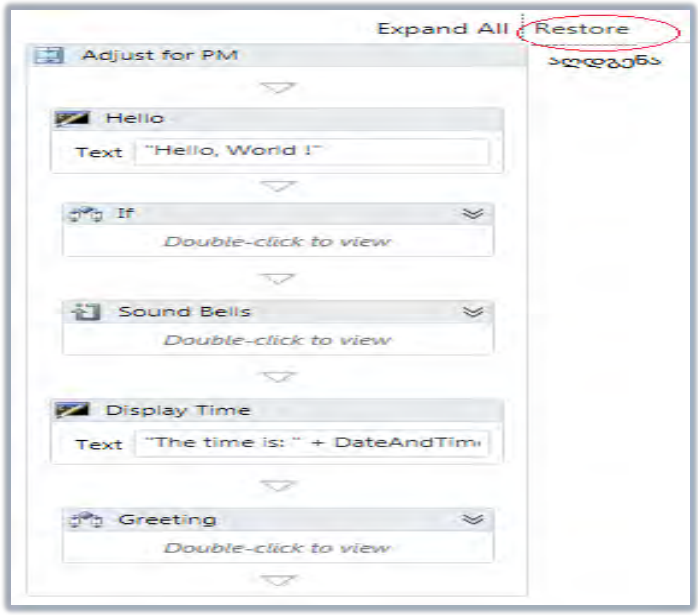


ნახ.27. შედეგი

1.3. დიზაინერის მართვა და XAML კოდი (ლაბ.N3)

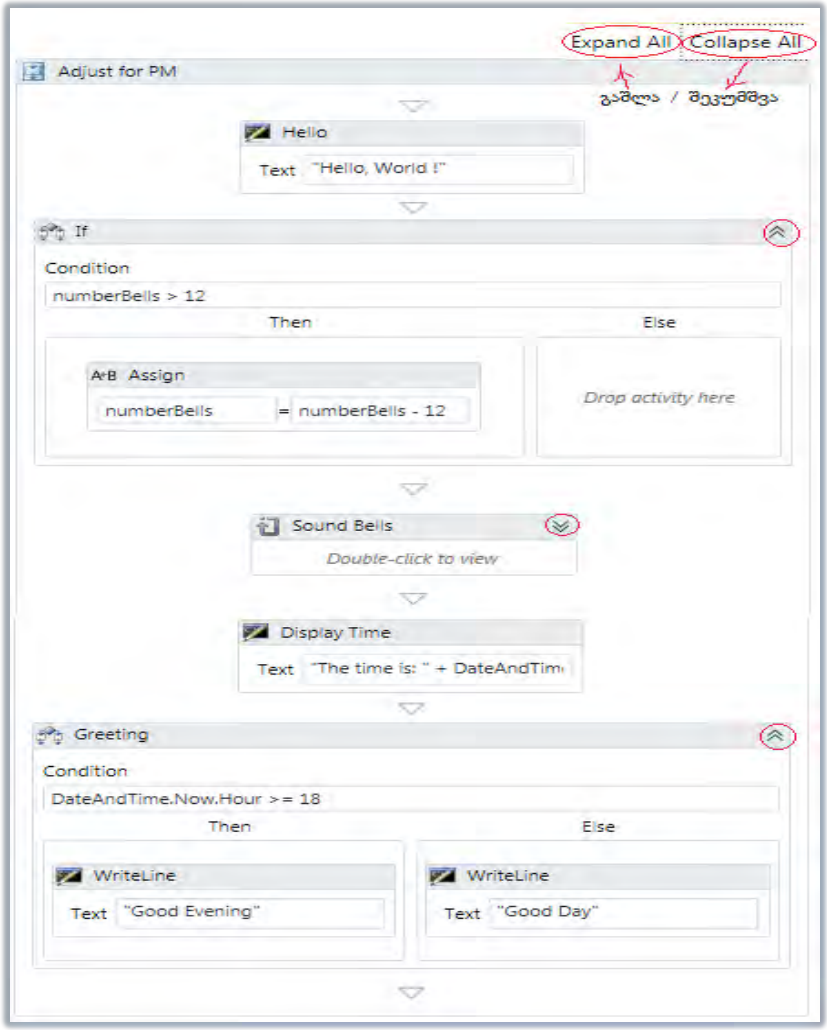
მიზანი: Visual Studio.NET პაკეტის WF-ის დიზაინერის სამუშაო გარემოში დიდი პროექტების და სქემების მართვის საშუალებების გაცნობა (Navigating the Designer).

მარტივი მაგალითიდან დავინახეთ, რომ საკმაოდ რთულია სამუშაო პროცესების დიდი სქემების მართვა და მათი მთლიანობაში წარმოდგენა. დიდი პროექტების დროს ის კიდევ უფრო რთულია. ამისათვის დიზაინერებს ეძლევათ საშუალება ჩაკეცონ სქემის სექციები, მათი მთლიანი სტრუქტურის დათვალიერების და მართვის მიზნით. მაგალითად, ჩვენი პროექტი ჩაკეცვის (Collapse All) შემდეგ ასე გამოიყურება (ნახ.1.28).



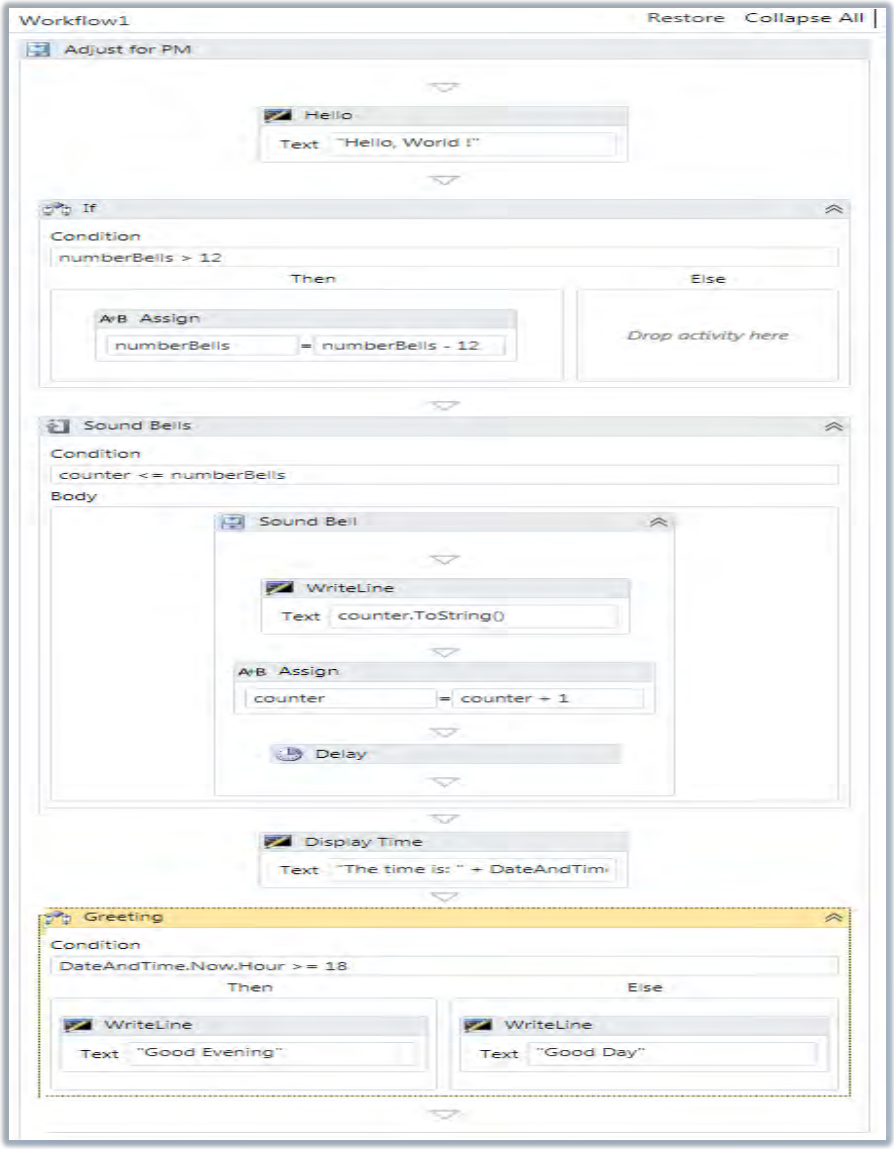
ნახ.1.28. შეკუმშული workflow დიაგრამა

თუ აღდგენის (Restore) ღილაკს გამოვიყენებთ, მაშინ მივიღებთ სურათს შეკუმშვამდე (ნახ.1.29).



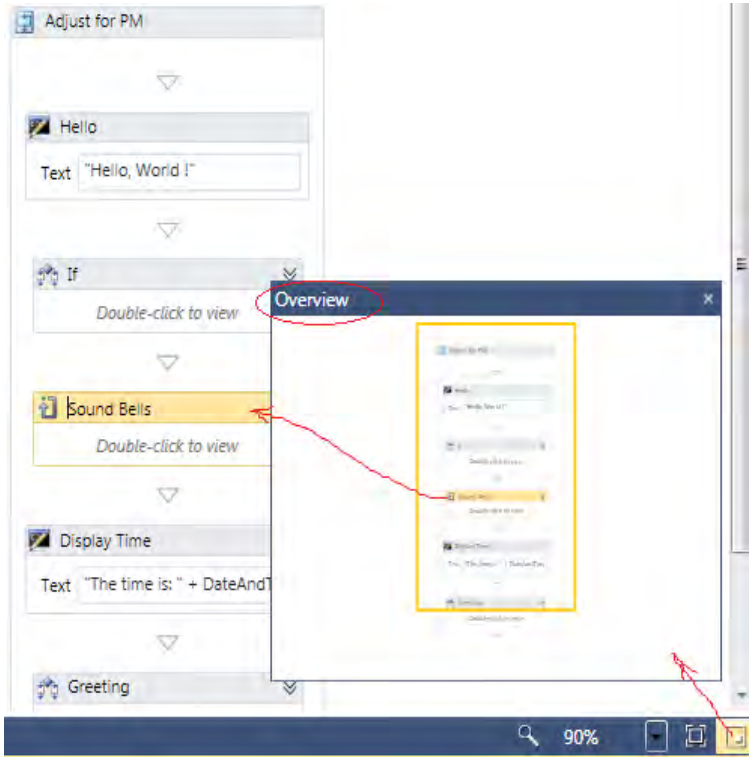
ნახ.1.29.ნაწილობრივ შეკუმშული workflow დიაგრამა

აქ რამდენიმე ბლოკი გაშლილია მთლიანად, რამდენიმე კი ჩაკეცილია (მაგალითად, Bound Bells). Expand All ღილაკით კი გაიშლება ყველა ჩაკეცილი ბლოკი (ნახ.1.30).



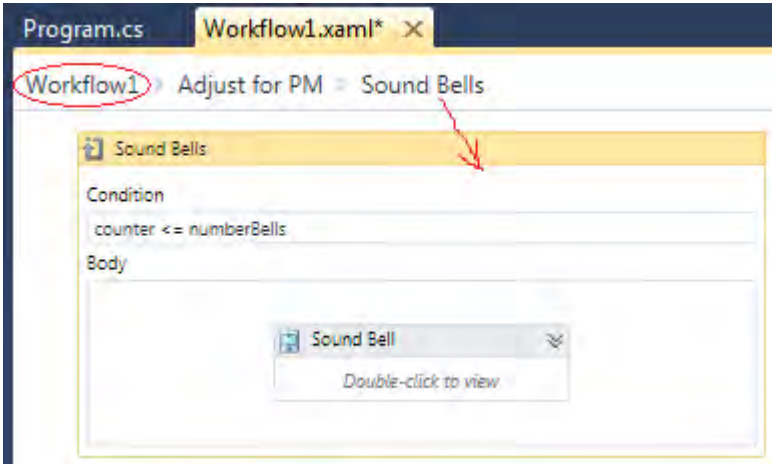
ნახ.1.30. მთლიანად გაშლილი workflow დიაგრამა

დიზაინერის ქვედა მარჯვენა კუთხეში არის Overview–ღილაკი, რომელიც ხსნის პროექტის მონიტორინგის ფანჯარას (ნახ.1.31). მოყვითალო ფერის ზოლი მიუთითებს აქტიურ ხილვად ბლოკზე. აქვეა ეკრანის ზომების ცვლილების (მასშტაბირების) ღილაკებიც.



ნახ.1.31. Overview - ღილაკი

„Sound Bell" ქმედების დაკლიკვით გამოჩნდება მხოლოდ ეს ბლოკი თავისი შვილობილი (ჩადგმული) კომპონენტებით (ნახ.1.32). აქ Window1 გადამრთველის არჩევით დიზაინერში გამოჩნდება ისევ 1.28–ე ნახაზზე ნაჩვენები სქემა.



ნახ.1.1.32. გამოყოფილი Sound Bell ბლოკი

- დიზაინერის XAML-კოდი

Solution Explorer-ში დავკლიკოთ Workflow1.xaml და დავათვალიეროთ შესაბამისი კოდი (ლისტინგი_1.2):

```
<! ---- ლისტინგი_1.2 -----  >
<Activity mc:Ignorable="" x:Class="WorkflowConsoleHello.Workflow1"
    xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:mv="clr-namespace:Microsoft.VisualBasic;assembly=System"
    xmlns:mva="clr-namespace:Microsoft.VisualBasic.Activities;assembly=
                                                        Sys
tem.Activities"
    xmlns:s="clr-namespace:System;assembly=mscorlib"
    xmlns:s1="clr-namespace:System;assembly=System"
    xmlns:s2="clr-namespace:System;assembly=System.Xml"
    xmlns:s3="clr-namespace:System;assembly=System.Core"
    xmlns:s4="clr-namespace:System;assembly=System.ServiceModel"
```

```
      xmlns:sa="clr-namespace:System.Activities;assembly=System.Activities"
      xmlns:sad="clr-namespace:System.Activities.Debugger;assembly=
                                                          System.Activities"
xmlns:sap="http://schemas.microsoft.com/netfx/2009/xaml/activities/presentation
"
      xmlns:scg="clr-namespace:System.Collections.Generic;assembly=System"
      xmlns:scg1="clr-namespace:System.Collections.Generic;assembly=
                                                      System.ServiceModel"
      xmlns:scg2="clr-
namespace:System.Collections.Generic;assembly=System.Core"
      xmlns:scg3="clr-namespace:System.Collections.Generic;assembly=mscorlib"
      xmlns:sd="clr-namespace:System.Data;assembly=System.Data"
      xmlns:sl="clr-namespace:System.Linq;assembly=System.Core"
      xmlns:st="clr-namespace:System.Text;assembly=mscorlib"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <sap:WorkflowViewStateService.ViewState>
    <scg3:Dictionary x:TypeArguments="x:String, x:Object">
      <x:Boolean x:Key="ShouldCollapseAll">True</x:Boolean>
      <x:Boolean x:Key="ShouldExpandAll">False</x:Boolean>
    </scg3:Dictionary>
  </sap:WorkflowViewStateService.ViewState>

  <Sequence DisplayName="Sequence"

sad:XamlDebuggerXmlReader.FileName="C:\WorkflowConsoleHello\WorkflowC
onsoleHello\Workflow1.xaml"
    sap:VirtualizedContainerService.HintSize="233,564">
    <Sequence.Variables>
      <Variable x:TypeArguments="x:Int32" Default="1" Name="counter" />
      <Variable x:TypeArguments="x:Int32" Default="[DateAndTime.Now.Hour]"
Name="numberBells" />
    </Sequence.Variables>
    <sap:WorkflowViewStateService.ViewState>
```

```
      <scg3:Dictionary x:TypeArguments="x:String, x:Object">
        <x:Boolean x:Key="IsExpanded">True</x:Boolean>
      </scg3:Dictionary>
    </sap:WorkflowViewStateService.ViewState>

  <WriteLine DisplayName="Hello"
sap:VirtualizedContainerService.HintSize="211,62"

Text="Hello,  World !" />
    <If Condition="[numberBells &gt; 12]"
sap:VirtualizedContainerService.HintSize="211,52">
      <sap:WorkflowViewStateService.ViewState>
        <scg3:Dictionary x:TypeArguments="x:String, x:Object">
          <x:Boolean x:Key="IsExpanded">True</x:Boolean>
          <x:Boolean x:Key="IsPinned">False</x:Boolean>
        </scg3:Dictionary>
      </sap:WorkflowViewStateService.ViewState>
      <If.Then>

        <Assign sap:VirtualizedContainerService.HintSize="291,100">
          <Assign.To>
            <OutArgument
x:TypeArguments="x:Int32">[numberBells]</OutArgument>
          </Assign.To>
          <Assign.Value>
            <InArgument x:TypeArguments="x:Int32">[numberBells -
12]</InArgument>
          </Assign.Value>
        </Assign>
      </If.Then>
    </If>
```

```
    <While DisplayName="Sound Bells"
sap:VirtualizedContainerService.HintSize="211,52">
      <sap:WorkflowViewStateService.ViewState>
        <scg3:Dictionary x:TypeArguments="x:String, x:Object">
          <x:Boolean x:Key="IsExpanded">False</x:Boolean>
          <x:Boolean x:Key="IsPinned">False</x:Boolean>
        </scg3:Dictionary>
      </sap:WorkflowViewStateService.ViewState>
      <While.Condition>[counter &lt;= numberBells]</While.Condition>
      <Sequence DisplayName="Sound Bell"
sap:VirtualizedContainerService.HintSize="438,100">
        <sap:WorkflowViewStateService.ViewState>
          <scg3:Dictionary x:TypeArguments="x:String, x:Object">
            <x:Boolean x:Key="IsExpanded">False</x:Boolean>
            <x:Boolean x:Key="IsPinned">False</x:Boolean>
          </scg3:Dictionary>
        </sap:WorkflowViewStateService.ViewState>

    <WriteLine sap:VirtualizedContainerService.HintSize="242,62"
Text="[counter.ToString()]" />
        <Assign sap:VirtualizedContainerService.HintSize="242,58">
          <Assign.To>
            <OutArgument x:TypeArguments="x:Int32">[counter]</OutArgument>
          </Assign.To>
          <Assign.Value>
            <InArgument x:TypeArguments="x:Int32">[counter + 1]</InArgument>
          </Assign.Value>
        </Assign>

   <Delay Duration="[TimeSpan.FromSeconds(1)]"
sap:VirtualizedContainerService.HintSize="242,22" />
      </Sequence>
    </While>
```

```
 <WriteLine DisplayName="Display Time"
sap:VirtualizedContainerService.HintSize="211,62"
        Text="[&quot;The time is: &quot; + DateAndTime.Now.ToString()]" />

  <If Condition="[DateAndTime.Now.Hour &gt;= 18]" DisplayName="Greeting"
        sap:VirtualizedContainerService.HintSize="211,52">
     <If.Then>
       <WriteLine sap:VirtualizedContainerService.HintSize="219,100" Text="Good
Evening" />
     </If.Then>
     <If.Else>
       <WriteLine sap:VirtualizedContainerService.HintSize="220,100" Text="Good
Day" />
     </If.Else>
   </If>
  </Sequence>
</Activity>
```

1.4. კოდირებული სამუშაო პროცესები (ლაბ.4)

მიზანი: Visual Studio.NET პაკეტის WF-ის სამუშაო პროცესის შესრულების კოდის გაცნობა.

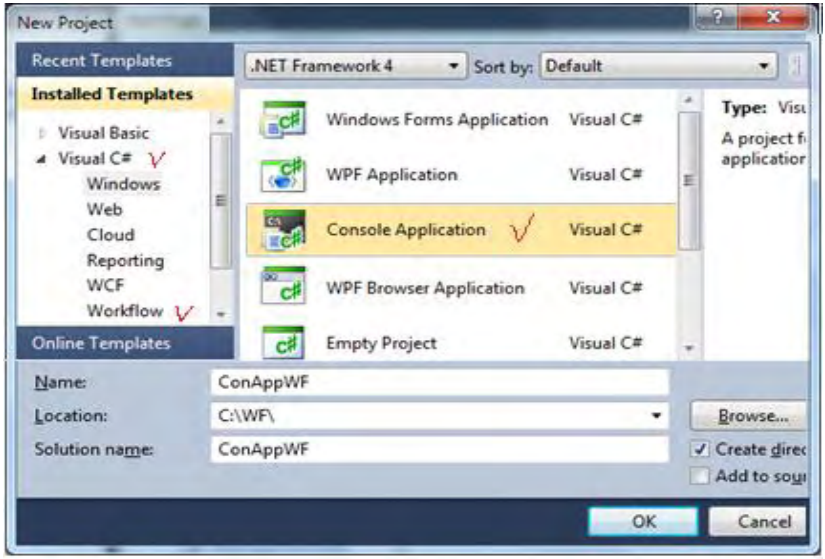
წინა ლაბორატორიებში ჩვენ ავაგეთ მარტივი სამუშაო პროცესი (workflow) დიზაინერის გამოყენებით. ახლა განვიხილოთ იგივე სამუშაო პროცესის შესრულება კოდის გამოყენებით.

ნებისმიერი მუშა პროცესი შეიძლება განხორციელდეს **კოდის ან დიზაინერის** გამოყენებით (შემსრულებლის გემოვნების საკითხია).

კოდის გამოყენება საშუალებას იძლევა უფრო კარგად გავიგოთ თუ როგორ ხდება სამუშაო პროცესის (workflow) რეალიზაცია და ფუნქციონირება.

- **კონსოლური აპლიკაციია აგება**

Visual Studio.NET გარემოში შევქმნათ ახალი კონსოლური აპლიკაცია სახელით ConAppWF (ნახ.1.33).



ნახ.1.33. კონსოლის აპლიკაციის შექმნა

დავამატოთ მიმართვა (reference) **System.Activities**. იგი საშუალებას იძლევა აპლიკაციაში გამოყენებულ იქნას სამუშაო პროცესის (workflow) ქმედებები (აქტიურობები).

შევცვალოთ Program.cs ფაილში სახელსივრცეები შემდეგი სახით (ლისტინგი_1.3).

// ------- ლისტინგი_1.3 --------

```
using System;
using System.Activities;
using System.Activities.Statements;
using System.Activities.Expressions;
namespace ConAppWF
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            WorkflowInvoker.Invoke(CreateWorkflow());
            Console.WriteLine("Press ENTER to exit");
            Console.ReadLine();
        }
    }
}
```

Main()–ის კოდი მსგავსია ჩვენ მიერ წინა მასალაში განხილული იგივე კოდისა. არის ერთი განსხვავება, რომ

```
//აქ CreateWorkflow()–ია
WorkflowInvoker.Invoke(CreateWorkflow());

// აქ new Workflow1() –ია
WorkflowInvoker.Invoke(new Workflow1());
```

Workflow1 იყო განსაზღვრული Workflow1.xaml ფაილში, რომელიც შეიქმნა Workflow Designer-ით.

CreateWorkflow () კი არის მეთოდი, რომლის რეალიზაცია ახლა უნდა მოვახდინოთ.

- **სამუშაო პროცესის (workflow) განსაზღვრა**

როგორც ზემოთ აღვნიშნეთ, სამუშაო პროცესი არის ჩადგმული კლასების და მათი თვისებების ერთობლიობა. განვიხილოთ ეს საკითხი კოდის გამოყენებით. დავამატოთ Program.cs ფაილში შემდეგი მეთოდი (ლისტინგი_1.4).

```
// ---- ლისტინგი_1.4 ----------------
static Activity CreateWorkflow()
 {
   Variable<int> numberBells = new Variable<int>()
    {
     Name = "numberBells",
```

```
        Default = DateTime.Now.Hour
      };
    Variable<int> counter = new Variable<int>()
      {
        Name = "counter",
        Default = 1
       };
  return new Sequence()
   {
   };
}
```

CreateWorkflow() მეთოდი პირველად ქმნის ორ Variable<T> კლასის შაბლონს ტიპით int, სახელით numberBells და counter. ეს ცვლადები გამოიყენება სხვადასხვა ქმედებებში.

CreateWorkflow() მეთოდი გამოიყნება იმ აქტიურობაზე დასაბრუნებლად, რომელსაც ელოდება WorkflowInvoker კლასი. იგი ფაქტობრივად აბრუნებს Sequence კლასის ანონიმურ ეგზემპლარზე.

Activity კლასი არის საბაზო, რომლისგანაც იწარმოება მუშა პროცესის ყველა ქმედება, ასევე Sequence–ც.

ასე რომ კომპილატორი აბრუნებს Sequence ეგზემპლარს და Activity–ის როგორც მის საბაზო კლასს

- **რეალიზაცია 1–ელ დონეზე**

ამგვარად, ჩვენ განვსაზღვრეთ ცარიელი Sequence ქმედება. ეს ეკვივალენტურია ახალი მუშა პროცესის შექმნის, რომელსაც აქვს Sequence, ქმედებების გარეშე. ახლა განვსაზღვროთ ქმედებები ამ Sequence–ზე return new Sequence() გამოძახებით და შეცვლით 1.5 ლისტინგზე მოცემული კოდით.

```
// ---- ლისტინგი_1.5 ------------
return new Sequence()
{
```

```
    DisplayName = "Main Sequence",
    Variables = { numberBells, counter },
    Activities =
      {
       new WriteLine()
         {
           DisplayName = "Hello",
           Text = "Hello, World!"
         },
       new If()
         {
           DisplayName = "Adjust for PM"
           // Code to be added here in Level 2
         },
       new While()
         {
           DisplayName = "Sound Bells"
           // Code to be added here in Level 2
         },
       new WriteLine()
         {
            DisplayName = "Display Time",
            Text = "The time is: " + DateTime.Now.ToString()
         },
        new If()
         {
           DisplayName = "Greeting"
           // Code to be added here in Level 2
         }
      }
};
```

ეს კოდი თავიდან განსაზღვრავს DisplayName და ობიექტის დამოკიდებულ ცვლადებს ამ ქმედებასთან. შემდეგ იგი

აინიციალიზებს წევრ–ქმედებებს, როგორც ქმედებათა კოლექციას. კერძოდ, იგი ქმნის 1.1 ცხრილში მოცემულ ქმედებებს.

ცხრ.1.1

Type	DisplayName
WriteLine	“Hello”
If	“Adjust for PM”
While	“Sound Bells”
WriteLine	“Display Time”
If	“Greeting”

WriteLine ქმედებებისთვის განსაზღვრულია Text თვისებები. სხვა ქმედებებისთვის რეალიზაცია განსაზღვრულ იქნება შემდეგ დონეზე.

- **რეალიზაცია მე–2 დონეზე**

პირველი If ქმედებისთვის შევიტანოთ შემდეგი კოდი:

```
DisplayName = "Adjust for PM",
    // მე–2 დონეზე დასამატებელი კოდი
    Condition = ExpressionServices.Convert<bool>
          (env => numberBells.Get(env) > 12),
    Then = new Assign<int>()
     {
        DisplayName = "Adjust Bells"
       // მე–3 დონეზე დასამატებელი კოდი
```

ეს კოდი განსაზღვრავს მდგომარეობას (Condition) და Then–ის თვისებებს (აქ არაა Else ნაწილი). Assign ქმედება იქნება რეალიზებული მომდევნო დონეზე. Condition თვისების განსაზღვრა საჭიროებს მცირე კომენტარს.

- გამოსახულებების შეტანა

ExpressionServices კლასის სტატიკური Convert<T>() მეთოდი გამოიყენება InArgument <T> კლასის შესაქმნელად, რასაც ელოდება მდგომარეობის (Condition) თვისება. ეს კლასები და მეთოდები იყენებს საერთო (T) ტიპს, ამიტომაც ისინი შეიძლება ნებისმიერი ტიპისთვის იქნას გამოყენებული. ამ შემთხვევაში ჩვენ უნდა გამოვიყენოთ BOOL ტიპი, ვინაიდან If ქმედების პირობის თვისება ელოდება true-ს ან false-ს.

გამოსახულების შეტანა რეალიზდება ლამბდა–გამოსახულებით (LINQ სინტაკსის ანალოგიურად), მონაცემების ამოსაღებად მუშა პროცესის გარემოდან (workflow environment). ლამბდა გამოსახულებაში „ => “ –ს უწოდებენ ლამბდა ოპერატორს. მის მარცხნივ თავსდება შემავალი პარამეტრები, ხოლო მარჯვენა მხარეს განისაზღვრება ფაქტობრივი გამოსახულება. ENV–ის მნიშვნელობა მიიღება შესრულების დროს, როცა ის ცდილობს მდგომარეობის (Condition) შეფასებას.

ფაქტობრივად, სამუშაო პროცესი მდგომარეობის გარეშეა, იგი არ ინახავს ელემენტთა მონაცემებს. კლასის ცვლადები განსაზღვრულია მარტივი მონაცემებით. იმისათვის, რომ მივიღოთ ფაქტობრივი მონაცემები კლასის ცვლადებიდან, უნდა გამოვიყენოთ საკუთარი Get () მეთოდი. ის მოითხოვს სორტის მარკერს, რომელიც არის ActivityContext კლასი. ეს საჭიროა, რათა განვასხვავოთ მუშა პროცესის კონკრეტული ეგზემპლარი სხვებისგან, რომლებიც შეიძლება ერთდროულად იქნას გაშვებული. Get (env)–ით დაბრუნებული მნიშვნელობა შეუდარდება, არის თუა არა ის მეტი 12–ზე.

შევიტანოთ შემდეგი კოდი While ქმედებისთვის:

```
DisplayName = "Sound Bells",
// მე–2 დონეზე დასამატებელი კოდი
Condition = ExpressionServices.Convert<bool>
```

```
    (env => counter.Get(env) <= numberBells.Get(env)),
Body = new Sequence()
 {
    DisplayName = "Sound Bell"
    // მე–3 დონეზე დასამატებელი კოდი
 }
```

While ქმედების Condition თვისება იდენტურია If ქმედების. ის იყენებს აგრეთვე ExpressionServices კლასს InArgument<T> კლასის შესაქმნელად. აგრეთვე bool ტიპს. შემთხვევაში ის აფასებს, არის თუ არა count <= numberBells. ამ ორივე ცვლადისთვის იგი იყენებს Get(env) მეთოდს ფაქტობრივი მნიშვნელობის მისაღებად.

მეორე If ქმედებისთვის (სახელით "Greeting") შევიტანოთ შემდეგი კოდი:

```
DisplayName = "Greeting",
// მე–2 დონეზე დასამატებელი კოდი
Condition = ExpressionServices.Convert<bool>
    (env => DateTime.Now.Hour >= 18),
  Then = new WriteLine() { Text = "Good Evening" },
  Else = new WriteLine() { Text = "Good Day" }
```

ამ პირობისთვის (Condition) შემავალი env–პარამეტრი არ გამოიყენება, მაგრამ იგი მაინც უნდა გამოცხადდეს გამოსახულებაში. ლოგიკა იყენებს მიმდინარე დროს, რათა დავინახოთ არის თუ არა ის 6:00 PM. ორივე Thet და Else თვისებისთვის იქმნება WriteLine ქმედება. ერთი ამბობს „საღამო მშვიდობისა" (Good Evening), მეორე კი – „გამარჯობათ" (Good Day).

- **რეალიზაცია მე–3 დონეზე**

პირველი If ქმედებისთვის (სახელით "Adjust for PM"), შექმნილია ცარიელი Assign ქმედება Then თვისებაში. შევიტანოთ შემდეგი ტექსტი ამ რეალიზაციაში:

```
DisplayName = "Adjust Bells",
// მე–3 დონეზე დასამატებელი კოდი
To = new OutArgument<int>(numberBells),
Value = new InArgument<int>(env => numberBells.Get(env) - 12)
```

- **Assign ქმედება (დანიშვნა, მინიჭება)**

Assign კლასი არის უნივერსალური (ზოგადი), ამიტომაც მას შეუძლია მონაცემთა ნებისმიერი ტიპის მხარდაჭერა. ამ შემთხვევაში ის ანიჭებს მთელ მნიშვნელობას, ამიტომ შექმნილია როგორც Assign<int>. თვისებები To და Value აგრეთვე იყენებს შაბლონურ კლასებს და უნდა შეიქმნას იგივე ტიპით (<int>).

To თვისება არის OutArgument კლასის, რომელიც იღებს კლასის ცვლადს კონსტრუქტორიდან. Value თვისება იყენებს InArgument კლასს. ის გამოყენებულ იქნა აქამდე Condition თვისების If და While -ში. მისი კონსტრუქტორისთვის გამოიყენება ლამბდა გამოსახულება, როგორც ეს იყო Condition თვისებისთვის.

- **Sequence ქმედება**

While ქმედებაში Execute თვისებისთვის შევქმენით ცარიელი Sequence. ის განსაზღვრავს შესასრულებელ ქმედებათა მიმდევრობას ციკლის შესრულების ყოველი მომენტისთვის. შევიტანოთ შემდეგი ტექსტი Activities თვისების შესავსებად:

```
DisplayName = "Sound Bell",
// მე–3 დონეზე დასამატებელი კოდი
```

```
Activities =
{
  new WriteLine()
   {
    Text = new InArgument<string>(env => counter.Get(env).ToString())
   },
  new Assign<int>()
   {
    DisplayName = "Increment Counter",
    To = new OutArgument<int>(counter),
    Value = new InArgument<int>(env => counter.Get(env) + 1)
   },
  new Delay()
   {
    Duration = TimeSpan.FromSeconds(1)
   }
}
```

ეს კოდი ამატებს სამ ქმედებას Sequence-ში:

- WriteLine აქტიურობას counter–ის გამოსატანად ეკრანზე;
- Assign აქტიურობას counter–ის ინკრემენტისთვის;
- Delay აქტიურობას მცირე პაუზისთვის იტერაციებს შორის.

ამ WriteLine ქმედებისთვის Text–ის თვისება არაა სიმბოლური string, როგორც სხვა პირობა იყო. ამ შემთხვევაში ეკრანზე გამოსატანი მნიშვნელობა განსაზღვრულია გამოსახულებაში.

Text–ის თვისება ელოდება string-ს, ამიტომაც იგი ქმნის InArgument<string> კლასს. ჩვენ ამ დროს ვიყენებთ ლამბდა გამოსახულებას. Get(env) მეთოდი Variable კლასისთვის უზრუნველყოფს მიმდინარე ცვლადისთვის მთელ ტიპს (integer). ToString() მეთოდი გარდაქმნის მას სტრიქონად.

Delay ქმედებისთვის ხანგრძლივობის (Duration) თვისება გადაეცემა როგორც TimeSpan კლასი, რომელიც შექმნილია FromSeconds() სტატიკური მეთოდით.

- **Running the Application (ამუშავება)**

F5 ღილაკით ავამუშავოთ აპლიკაცია. დღე–ღამის დროისგან დამოკიდებულებით შედეგები იქნება შემდეგი:

```
Hello, World!
1
2
3
4
5
6
7
The time is: 10/5/2009 7:02:41 PM
Good Evening

        Press ENTER to exit
```

რეალიზაციის პროგრამის სრული ტექსტი მოცემულია 1.10 ლისტინგში.

// ---- ლისტინგი_1.10 ----------

```
using System;
using System.Activities;
using System.Activities.Statements;
using System.Activities.Expressions;
namespace Chapter02
{
  class Program
    {
     static void Main(string[] args)
       {
          WorkflowInvoker.Invoke(CreateWorkflow());
          Console.WriteLine("Press ENTER to exit");
```

```
    Console.ReadLine();
  }
static Activity CreateWorkflow()
 {
  Variable<int> numberBells = new Variable<int>()
    {
     Name = "numberBells",
     Default = DateTime.Now.Hour
    };
  Variable<int> counter = new Variable<int>()
    {
     Name = "counter",
     Default = 1
    };
  return new Sequence()
    {
     DisplayName = "Main Sequence",
     Variables = { numberBells, counter },
     Activities =
        {  new WriteLine()
           {  DisplayName = "Hello",
              Text = "Hello, World!"
           },
          new If()
           {  DisplayName = "Adjust for PM",
            // Code to be added here in Level 2
            Condition = ExpressionServices.Convert<bool>
            (env => numberBells.Get(env) > 12),
            Then = new Assign<int>()
              {  DisplayName = "Adjust Bells",
                 // Code to be added here in Level 3
                 To = new OutArgument<int>(numberBells),
                 Value = new InArgument<int>
                 (env => numberBells.Get(env) - 12)
```

```
                }
              },
            new While()
              {  DisplayName = "Sound Bells",
                // Code to be added here in Level 2
                Condition = ExpressionServices.Convert<bool>
                (env => counter.Get(env) <= numberBells.Get(env)),
                Body = new Sequence()
                  { DisplayName = "Sound Bell",
                   // Code to be added here in Level 3
                   Activities =
                     { new WriteLine()
                         { Text = new InArgument<string>
                           (env => counter.Get(env).ToString())
                         },
            new Assign<int>()
              {
                DisplayName = "Increment Counter",
                To = new OutArgument<int>(counter),
                Value = new InArgument<int>
                (env => counter.Get(env) + 1)
              },
            new Delay()

              {
                Duration = TimeSpan.FromSeconds(1)
               }
            }
          }
      },
new WriteLine()
  {
      DisplayName = "Display Time",
      Text = "The time is: " + DateTime.Now.ToString()
  },
```

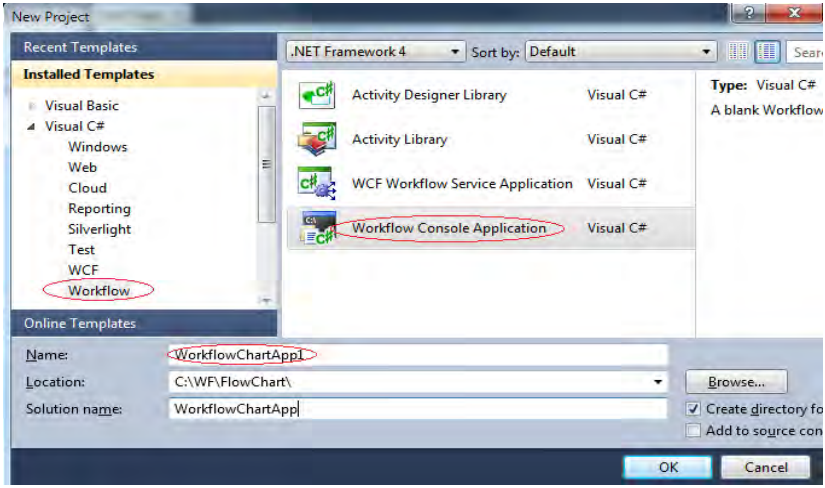
```
            new If()
             {
                 DisplayName = "Greeting",
                 // Code to be added here in Level 2
                 Condition = ExpressionServices.Convert<bool>
                 (env => DateTime.Now.Hour >= 16),
                 Then = new WriteLine() { Text = "Good Evening" },
                 Else = new WriteLine() { Text = "Good Day" }
             }
           }
         };
      }
    }
}
```

1.5. ბიზნესპროცესის დიაგრამა (ლაბ.5) (Flowchart Workflow)

ამჯერად უნდა ავაგოთ ბიზნესპროცესი, რომელიც გამოიყენებს აქტიურობათა დიაგრამას. როგორც სათაურიდან ჩანს, ქმედებათა დიაგრამა მუშაობს როგორც ბლოკ–სქემა, ქმედებათა სახეები დაკავშირებულია ერთმანეთთან გადაწყვეტილების ხეებით (decision trees). ქმედებათა მიმდევრობითობის გამოყენებით, შვილი–პროცესები სრულდება მიმდევრობით ზემოდან–ქვევით (top-down). ამისდა მიუხედავად, შვილი–ქმედებები შეიძლება შესრულდეს ნებისმიერი მიმდევრობით, გადაწყვეტილების მიმღები პირის მიერ.

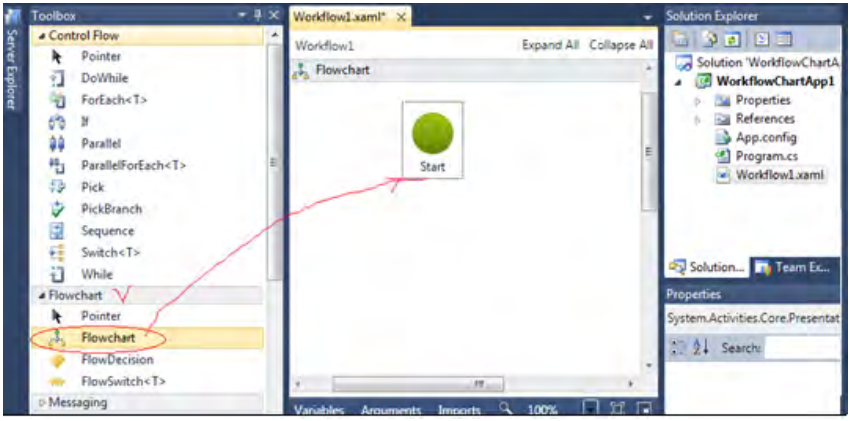
- **ბიზნესპროცესის დიაგრამის შექმნა**

შევქმნათ ახალი პროექტი (solution), ავირჩიოთ workflow და Console Application (ნახ.1.34).



ნახ.1.34

ინსტრუმენტების პანელიდან Flowchart გადმოვიტანოთ ფორმაზე Flowchart-ქმედება. საწყისი მუშა პროცესის დიაგრამა სასტარტო მწვანე წრით მოცემულია 1.35 ნახაზზე. ცარიელი სივრცე მის ქვეშ გამოიყენება ასაგები ბიზნესპროცესის ქმედებების დასამატებლად.

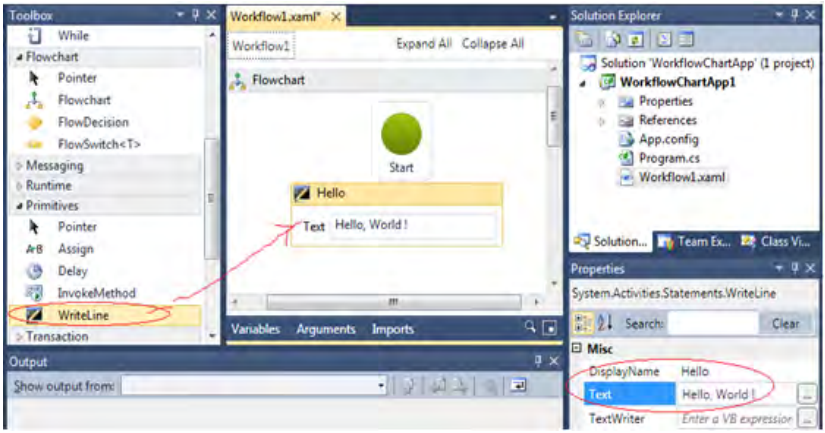


ნახ.1.35

ძირითადი განსხვავება Flowchart ქმედებასა და Sequence ქმედებას შორის ისაა, თუ როგორაა დაკავშირებული შვილი–აქტიურობები. როგორც 1–ელ თავიდან ვიცით, ქმედებების დამატებისას მიმდევრობითობაში ისინი სრულდება დაღმავალი (top-down) რიგითობით. შესაძლებელია მიმდევრობის კონტროლი (მართვა), ქმედებათა გადაადგილებით, ოღონდაც ისინი ყოველთვის გასწორებულია ვერტიკალში და განაწილებულია თანაბრად. ისრები ქმედებებს შორის კი აგებულია ავტომატურად.

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

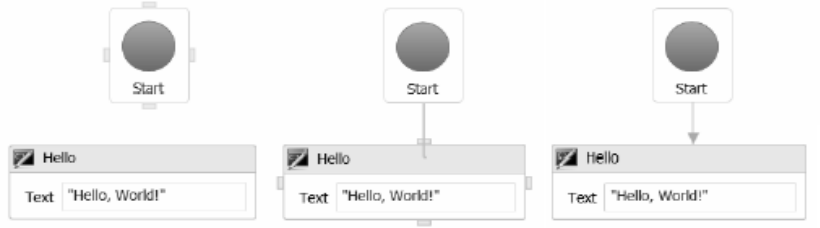
ჩვენ აპლიკაციაში დისპლეიზე უნდა გამოვიტანოთ მისალმებები, შესაბამისად დღე–ღმის დროისა. დავიწყოთ სტანდარტული მისალმების ასახვით „Hello, World". ამისათვის გადმოვიტანოთ WriteLine ქმედება ინსტრუმენტების პანელიდან მწვანე წრის ქვეშ. დავაყენოთ DisplayName-ში "Hello" და Text-ში "Hello, World !" (ნახ.1.36).



ნახ.1.36

- **მიერთების განსაზღვრა**

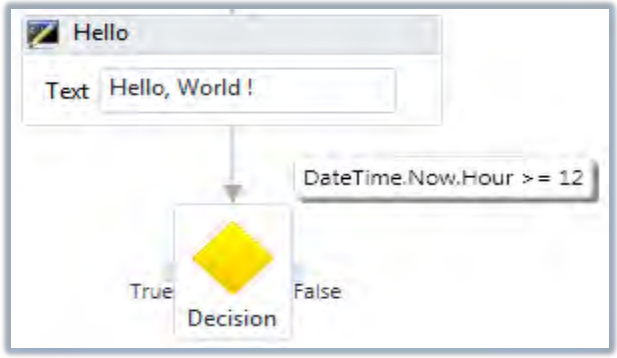
მაუსით მოვნიშნოთ სასტარტო მწვანე წრე, გავატაროთ კავშირი და ავტომატურად შეიქმნება ისარი ორ ქმედებას შორის (ნახ.1.37).



ნახ.1.37. მიერთების საწყისი და ბოლო წერტილები

- **გადაწყვეტილების ნაკადი (FlowDecision)**

გადმოვიტანოთ სქემაზე FlowDecision ქმედება "Hello" ქმედების შემდეგ. FlowDecision ქმედება გამოიყურება ყვითელი ალმასის სახით, როგორც განშტოების სიმბოლო ნორმალურ ბლოკ–სქემაში, თვისებათა ფანჯარაში შევიტანოთ მდგომარეობა (Condition) DateTime.Now.Hour> = 12. მაუსის კურსორის მიტანით FlowDecision ქმედებაზე გამოჩნდება ასეთი სურათი (ნახ.1.38).



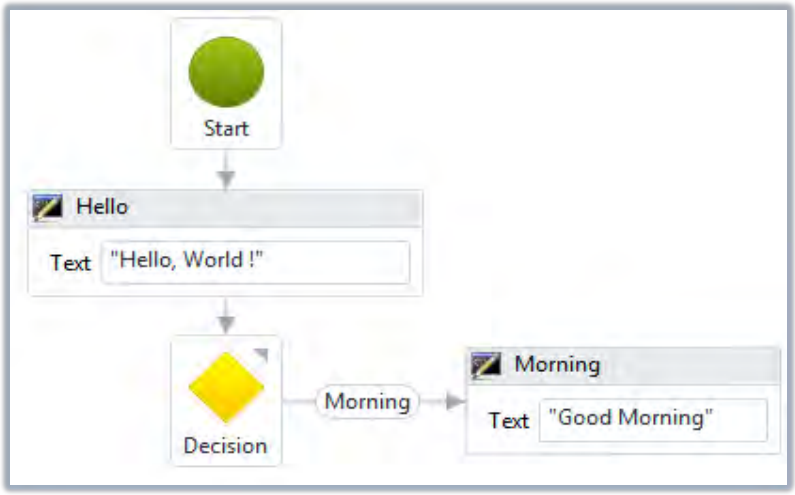
ნახ.1.38

ქმედებას აქვს მარცხნიდან მიერთების წერტილი true შტოსთვის, ხოლო მარჯვნიდან – false შტოსთვის. ზედა მარჯვენა კუთხეში პატარა სამკუთხედით ჩაირთვება პირობის გამოსახულების მუდმივად გამოსაჩენი თვისება (უმაუსოდ).

შეიძლება ტექსტის შეცვლა true/false შტოებისთვის. მაგალითად, FalseLabel–თვის შევიტანოთ Morning, და TrueLabel-თვის Afternoon. მიიღება სურათი:

ნახ.1.39

შევაერთოთ FlowDecision ქმედება მარჯვნიდან WriteLine ქმედებას, რომელშიც ჩაწერილი გვაქვს Morning/”Good Morning” (ნახ.1.40).



ნახ.1.40

FlowDecision ქმედებას არ აქვს DisplayName თვისება, მაგრამ შეუძლია პირობის ჩვენება და true/false შტოების შემოწმება და კორექტირება. ამგვარად ამ აქტიურობის მიზანი და ფუნქციონალობა ცალსახად ცხადია.

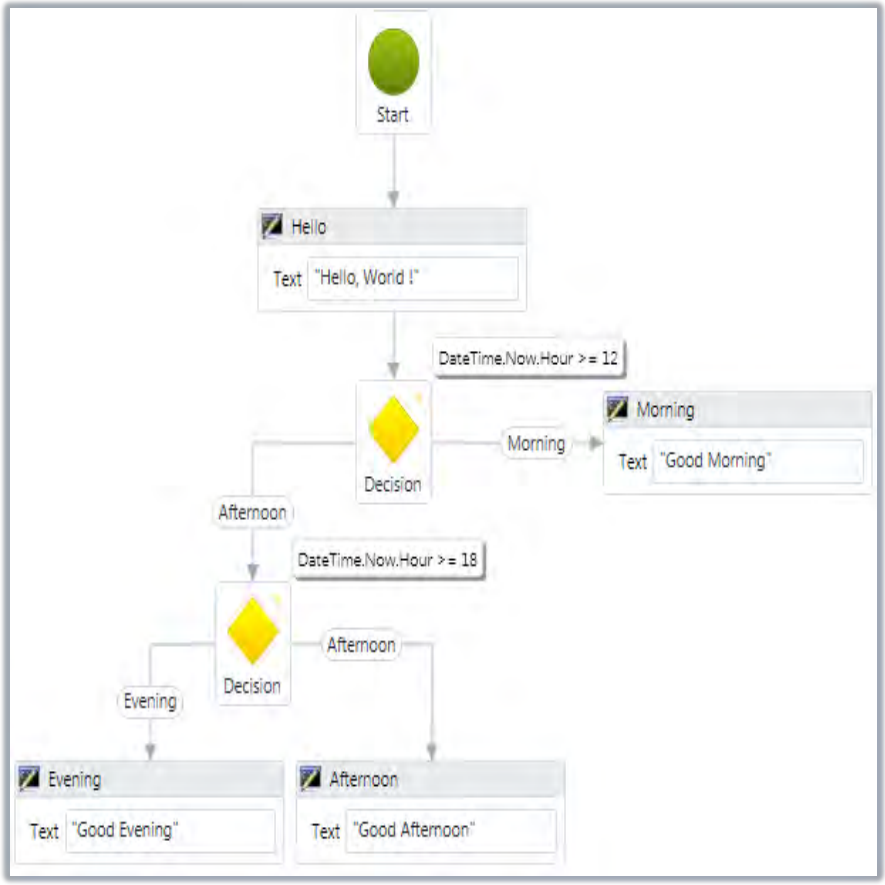
დავამატოთ FlowDecision ქმედების მარცხენა true შტოს ახალი FlowDecision ქმედება, პირობის გამოსახულებით DateTime.Now.Hour >= 18. მისი მარცხენა TrueLabel შტო იყოს Evening, ხოლო მარჯვენა – ისევ Afternoon. ორივეს მივუერთოთ ახალი WriteLine ქმედებები: Evening და Afternoon. 1.41 ნახაზზე მოცემულია ეს სურათი.

სანამ ავამუშავებთ მიღებულ აპლიკაციას, გავხსნათ Program.cs ფაილი. ჩავამატოთ აქ შემდეგი კოდი WorkflowInvoker კლასის გამოძახების შემდეგ. ის დაგვანახვებს შედეგებს პროგრამიდან გამოსვლამდე.

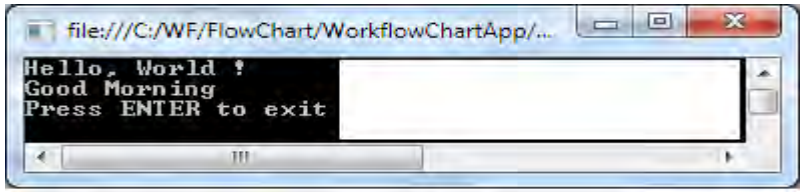
```
Console.WriteLine("Press ENTER to exit");
Console.ReadLine();
```

აპლიკაციის ამუშავება: F5 (ნახ.1.42)

შედეგი დამოკიდებულია იმაზე თუ დღის რომელ მონაკვეთში ავამუშავებთ აპლიკაციას.



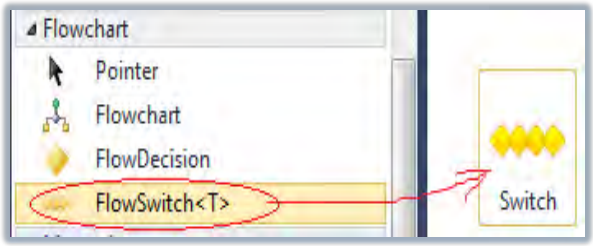
ნახ.1.41



ნახ.1.42. შედეგები

- **პროცესის გადამრთველი (Flow Switch)**

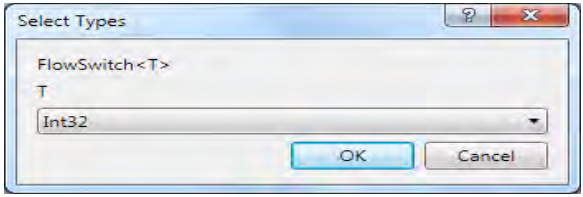
FlowSwitch ქმედება ფუნქციონირებს როგორც FlowDecision იმ გამონაკლისით, რომ არაა შეზღუდვა მხოლდ true/false ორი განშტოებით. შესაძლებელია შტოების ნებისმიერი რაოდენობის განსაზღვრა ისე, როგორც ეს იყო C# ენაში. 1.43 ნახაზზე ნაჩვენებია FlowSwitch აქტიურობის პიქტოგრამა ინსტრუმენტების პანელზე.



ნახ.1.43. პიქტოგრამა

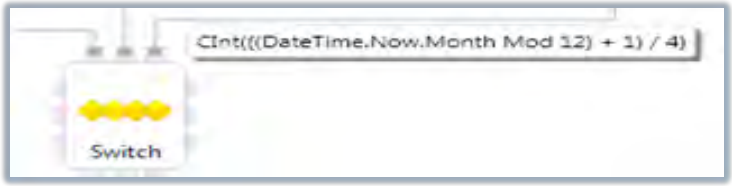
- **FlowSwitch ქმედების დამატება**

გადმოვიტანოთ ინსტრუმენტების პანელიდან FlowSwitch აქტიურობა ჩვენი ბოლო პროექტის სქემის ქვეშ. იგი არის <T> კლასის შაბლონი და უნდა მიეთითოს ტიპი. ჩვენ შემთხვევაში ესაა Int32, რომელიც ავტომატურადაა განსაზღვრული. ავირჩევთ OK-ს (ნახ.1.44).



ნახ.1.44. ტიპის განსაზღვრა

შევაერთოთ სქემის Morning, Evening და Afternoon ქმედებები FlowSwitch ქმედებას, რომელსაც აქვს ერთი თვისება გამოსახულებისთვის და იგი განსაზღვრავს გადართვის შტოს ცვლადის მნიშვნელობას (ნახ.1.45). ჩვენ პროექტში განვახორციელებთ გადამრთველის რეალიზებას მისალმების მიზნით წელიწადის დროის მიხედვით.



ნახ.1.45

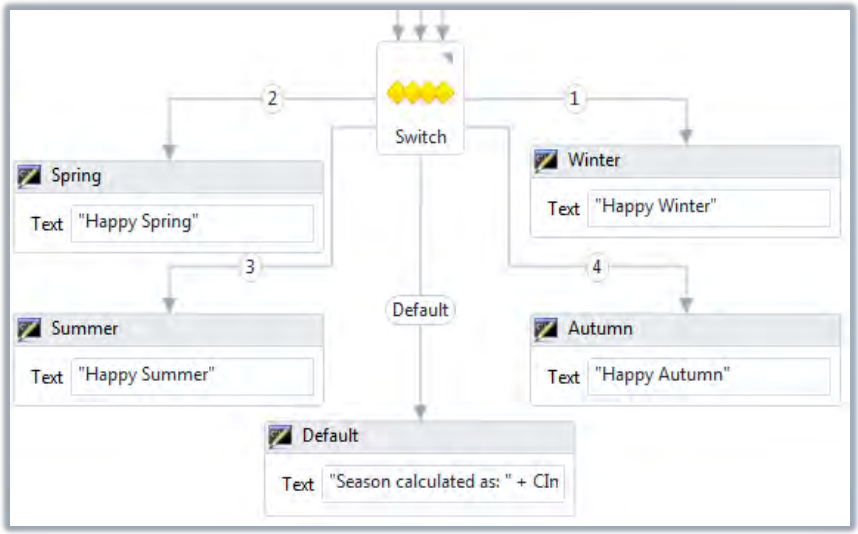
შენიშვნა: გამოსახულების ჩაწერის ფორმა არაა დამოკიდებული გამოყენებული ენის სინტაკსზე. აქ მიღებულია, რომ ეს ფორმა იყოს Visual Basic ენის სინტაკსის მსგავსი.

ჩვენი გამოსახულება განსაზღვრავს წელიწადის დროის (ზამთარი, გაზაფხული, ზაფხული, შემოდგომა) სეზონს. მაგალითად, თუ მიმდინარე (ახლანდელი) თვე არის დეკემბერი, იანვარი ან თებერვალი, მაშინ გამოსახულება გვაძლევს 0–ს. ანალოგიურად მარტი, აპრილი და მაისი მოგვცემს 1–ს და ა.შ. გადამრთველის ოთხ შტოსთვის უნდა განვსაზღვროთ 0,1,2 და 3, ანუ სეზონები.

- **FlowStep ქმედების დამატება**

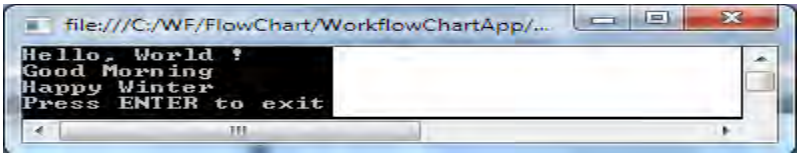
FlowSwitch აქტიურობის ყოველი შტო იძახებს FlowStep ქმედებას. ის Toolbox-ში არაა, ამიტომ მისი ცხადი სახით შექმნა არ შეიძლება. სქემას გადამრთველის გამოსასვლელზე უნდა დაემატოს რამდენიმე ქმედება, მაგალითად, ხუთი WriteLine და მათი შეერთების დროს მოხდება შინაგანად FlowStep–ის მნიშვნელობის განსაზღვრა. WriteLine ქმედებებისთვის ჩავწეროთ DisplayName-ში:

Winter, Spring, Summer, Autumn და Default, აგრეთვე შესაბამისი მისალმებები (ნახ.1.46).



ნახ.1.46

პროგრამის ამუშავებით(F5) მივიღებთ შედეგს (ნახ.1.47).



ნახ.1.47. FlowSwitch აპლიკაციის შედეგები

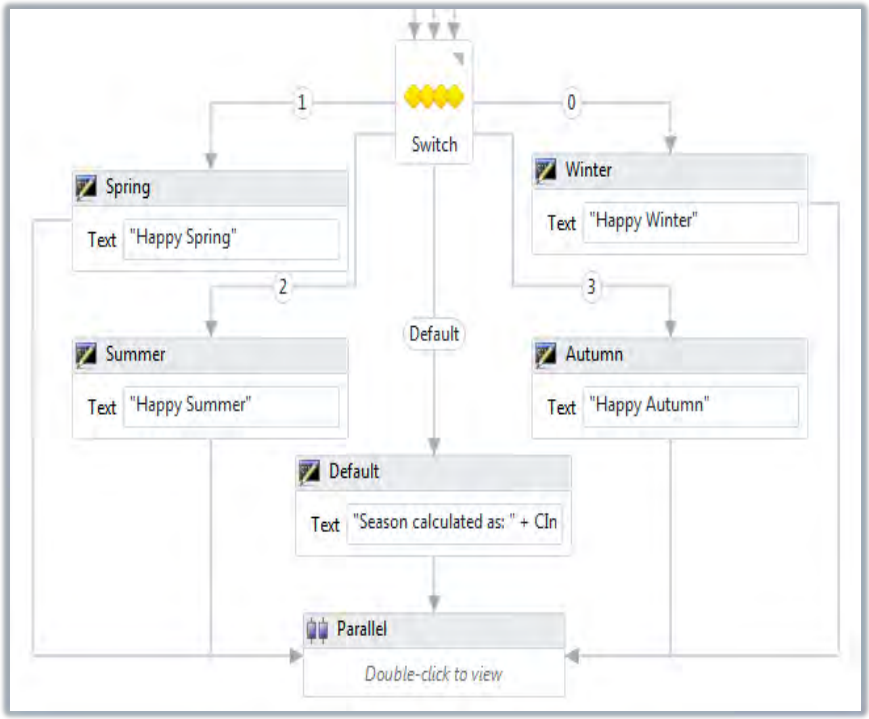
- **პარალელური პროცესები (Parallel)**

ჩვენი პროექტის ფარგლებში განვიხილოთ პარალელური პროცესების საკითხი, Parallel ქმედების მაგალითზე, რომელიც

საშუალებას იძლევა განვსაზღვროთ Sequence ქმედებათა რაოდენობა, რომლებიც სრულდება პარალელურად (ერთ-დროულად). ამ პროექტში ყოველი შტო გამოიტანს ინფორმაციის თავის ნაწილს. გამოტანის რიგითობას არ აქვს არსებითი მნიშვნელობა, მთვარია, რომ ისინი მოთავსებულია ერთ Parallel აქტიურობაში და ყველა სრულდება ერთდროულად.

- **Parallel ქმედების დამატება**

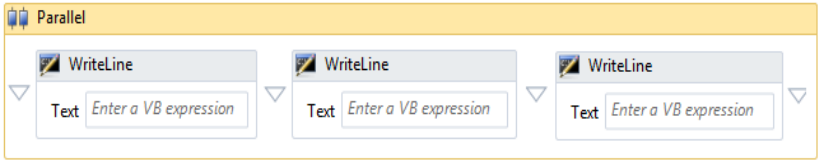
გადმოვიტანოთ Parallel აქტიურობა ჩვენი მუშა პროცესის ბოლოში. ხუთივე WriteLine-დან გავავლოთ კავშირი Parallel ქმედებასთან (ნახ.1.48).



ნახ.1.48

- **შტოების დამატება**

ორჯერ დავკლიკოთ Parallel ქმედება და მის ზედაპირზე გადმოვიტანოთ სამი WriteLine ქმედება (ნახ.1.49).



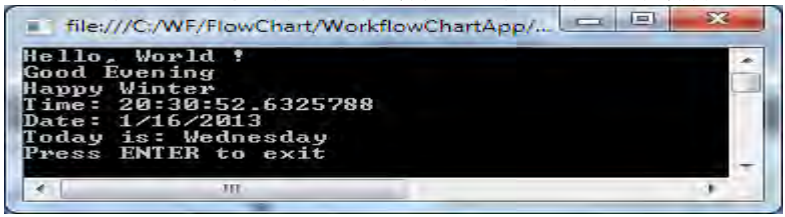
ნახ.1.49

ერთმა უნდა გამოიტანოს მიმდინარე თარიღი, მეორემ – დრო და მესამემ კვირის დღე. მათ Text-ში გამოსახულებებისათვის (expression) შევიტანოთ შემდეგი:

"Date: " + DateTime.Now.Date.ToShortDateString()
"Time: " + DateTime.Now.TimeOfDay.ToString()
"Today is: " + DateTime.Now.ToString("dddd")

<u>შენიშვნა:</u> Parallel ქმედება ნებას იძლევა მხოლოდ ერთი ქმედება განხორციელდეს ერთ შტოში, რაც ჩვენ მაგალითზე კარგად მუშაობს. თუ გვინდა მულტი–ქმედებების გამოყენება თითოეულ შტოში, მაშინ უნდა ვიხმაროთ Sequence ქმედება. აქ შეიძლება მის შიგნით დავამატოთ ქმედებათა გარკვეული რაოდენობა.

პროგრამის ამუშავების შემდეგ მიიღება ასეთი შედეგები:



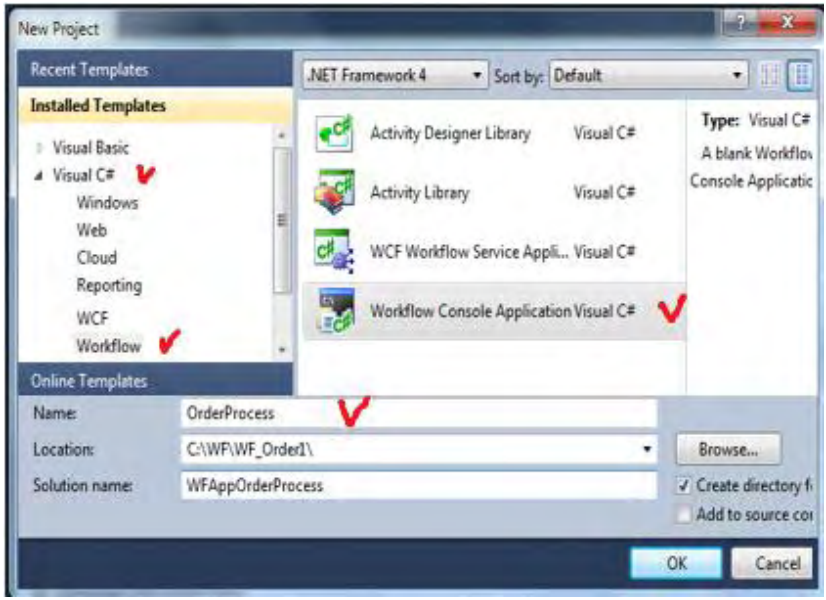
ნახ.1.50

II თავი
ბიზნესპროცესების დაპროექტება

2.1. რთული ბიზნესპროცესის არგუმენტები (ლაბ.N6)

მიზანი: Visual Studio.NET პაკეტის WF-ის სამუშაო გარემოში რთული ბიზნესპროცესის შემავალ-გამომავალი არგუმენტების გადაცემის ხერხების გაცნობა მუშა პროცესსა და ჰოსტ-აპლიკაციას შორის.

შევქმნათ ახალი ბიზნეს–პროცესის კონსოლური აპლიკაცია პროექტის სახელით OrderProcess, და Solution-ის სახელით WFAppOrderProcess (ნახ.2.1).



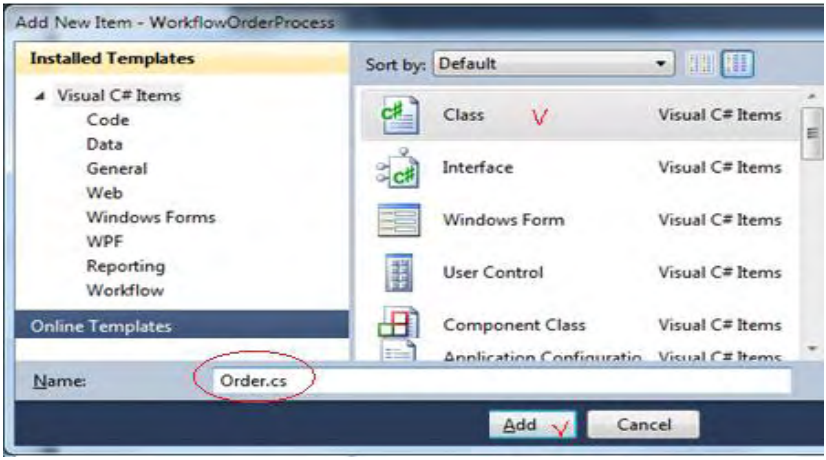
ნახ.2.1. ახალი კონსოლური აპლიკაციის პროექტის შექმნა

პროექტში უნდა განვსაზღვროთ შეკვეთა გარკვეულ პროდუქციაზე და ამ შეკვეთის გადაცემა ბიზნესპროცესში. ეს მუშა

პროცესი (workflow) გამოითვლის შეკვეთის სრულ ღირებულებას და დააბრუნებს მას აპლიკაციაში.

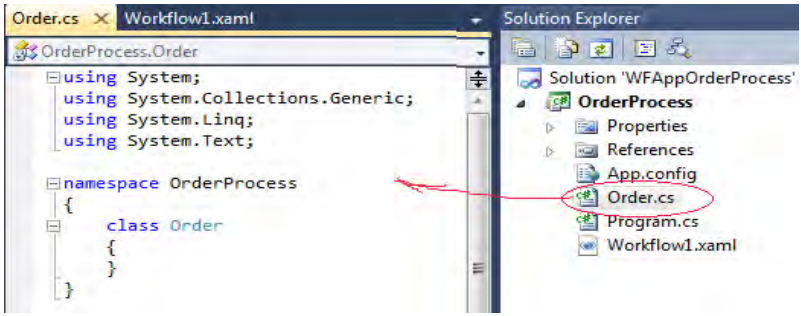
- **შეკვეთის კლასის განსაზღვრა (Defining the Order Class)**

პირველ ბიჯზე უნდა განისაზღვროს მონაცემთა სტრუქტურა, რომელიც უნდა შეიცავდეს შეკვეთის დეტალებს. Solution Explorer–ში დავამატოთ პროექტს ახალი კლასი: Add -> Class, სახელით Order.cs (ნახ.2.2).



ნახ.2.2. Order.cs კლასის შექმნა

მიიღება Solution Explorer ახალი კლასით (ნახ.2.3).



ნახ.2.3

კლასისი განსაზღვრისათვის Order.cs ფაილში შევიტანოთ შემდეგი კოდი (ლისტინგი_2.1).

```
//—— Order.cs ——  ლისტინგი_2.1 ————————
using System;
using System.Collections.Generic;
namespace OrderProcess
{
  public class OrderItem
   {
       public int OrderItemID { get; set; }
       public int Quantity { get; set; }
       public string ItemCode { get; set; }
       public string Description { get; set; }
   }
public class Order
   {
        public Order()
         {
           Items = new List<OrderItem>();
         }
       public int OrderID { get; set; }
       public string Description { get; set; }
       public decimal TotalWeight { get; set; }
       public string ShippingMethod { get; set; }
       public List<OrderItem> Items { get; set; }
    }
}
```

Order კლასი შეიცავს რამდენიმე ღია წევრს (OrderID, Description, TotalWeight და მეთოდს ShippingMethod). აგრეთვე OrderItem კლასის კოლექციას. ეს დეტალები ესაჭიროება ბიზნეს–პროცესს რათა გაითვალოს შეკვეთის ღირებულება.

ავაგოთ გადაწყვეტილება (solution) **F6 –ით (!!!)**, ეს შექმნის Order კლასს, რომელიც შემდეგ ბიჯებზე იქნება მისაწვდომი.

- **ბიზნესპროცესის რეალიზაცია (Implementing)**

Solution შაბლონი ქმნის სამუშაო პროცესის ფაილს Workflow1.xaml სახელით. შევცვალოთ ეს სახელი (Rename-თი) OrderWF.xaml სახელით.

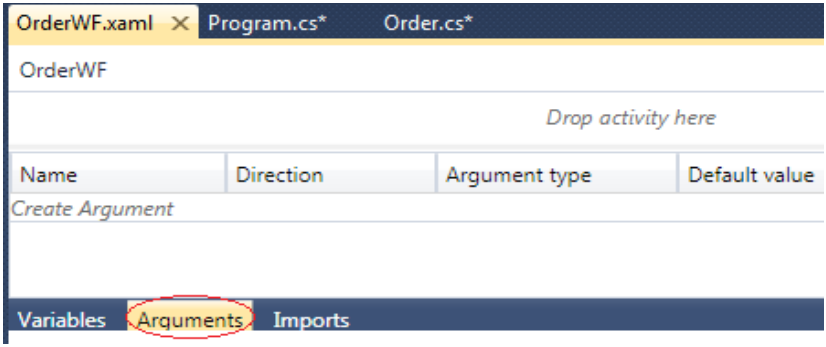
OrderWF.xaml ფაილში შევასწოროთ პირველი სტრიქონი:

x:Class="OrderProcess.OrderWF"

გავხსნათ Program.cs კოდი და `new Workflow1()` შევცვალოთ ახლით - `new` OrderWF()-ით.

- **არგუმენტების განსაზღვრა**

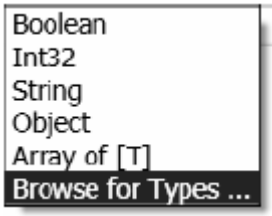
გავხსნათ OrderWF.xaml ფაილი და დიზაინის რეჟიმში. ახლა უნდა განვსაზღვროთ (შემავალი და გამომავალი) არგუმენტები ბიზნეს–პროცესისთვის. დავკლიკოთ Argument ღილაკი ფანჯრის ქვედა მარცხენა ნაწილში. გამოჩნდება არგუმენტების ცარიელი კოლექცია (ნახ.2.4).



ნახ.2.4. ცარიელი არგუმენტების სია

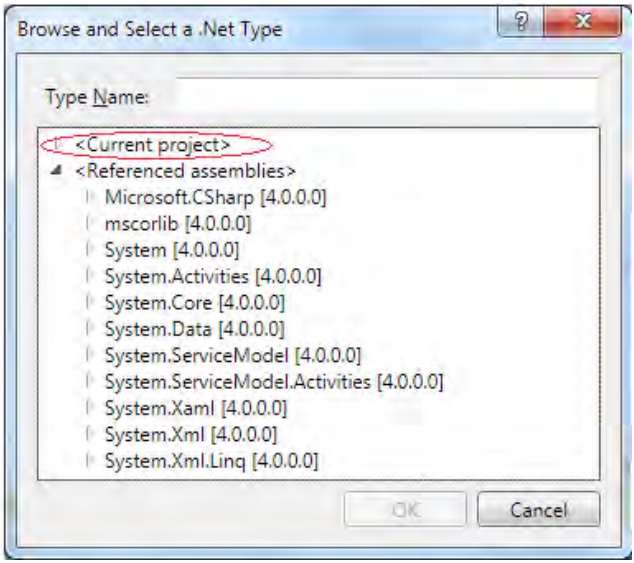
შენიშვნა: ცვლადები განისაზღვრებოდა მთელი პროცესისთვის, ან რომელიმე უბანზე და მათი შვილებისთვის. არგუმენტები ყოველთვის მთელი პროცესისთვისაა.

Create Argument -ით შევიტანოთ Name=OrderInfo, Direction=In და ArgumentType–სთვის ავირჩიოთ Browse for Types:



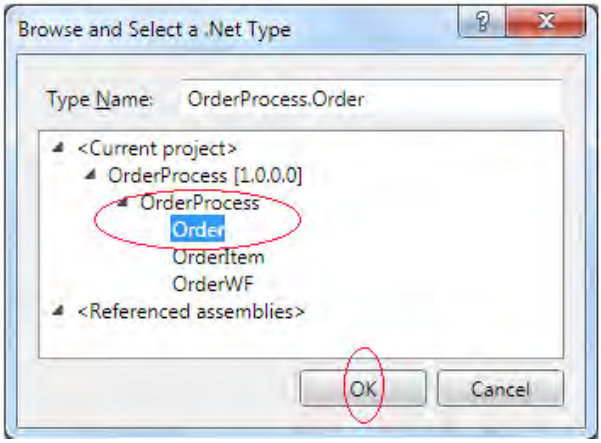
ნახ.2.5

გამოვა 2.6 ნახაზზე ნაჩვენები შედეგი <Current project>-ით [F6-ით კომპილირება წინა ეტაპზე აუცილებელია ამისთვის !!!].



ნახ.2.6

შევიტანოთ: TypeName=OrderProcess.Order, მივიღებთ ფანჯარას (ნახ.2.7).



ნახ.2.7

შემდეგ გავხსნათ OrderProcess და ავირჩიოთ Order კლასი და ღილაკი OK.

კიდევ ავირჩიოთ არგუმენტის შექმნა და შევიტანოთ სახელი TotalAmount. მდგომარეობა Direction უნდა იყოს **Out**, ასევე ArgumentType უნდა იყოს Decimal. (Type Name–ში შეიძლება System.Decimal მითითება უშუალოდ, ძებნის გარეშე). მიიღება:

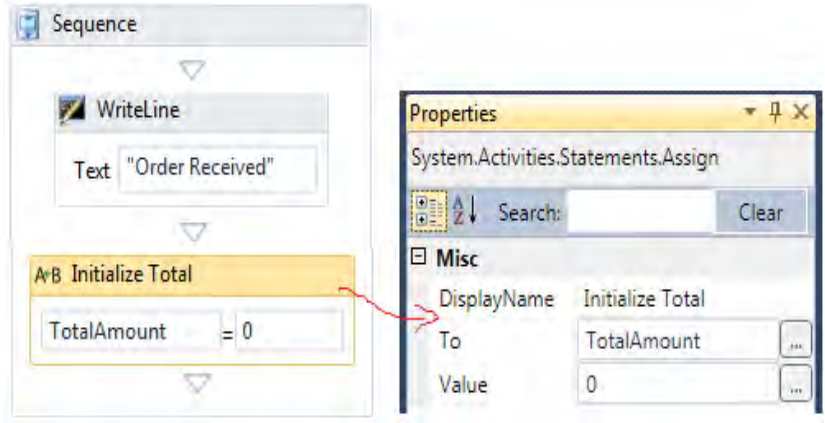
Name	Direction	Argument type	Default value
OrderInfo	In	Order	*Enter a VB expression*
TotalAmount	Out	Decimal	*Default value not supported*
Create Argument			

ნახ.2.8

- **ბიზნეს–პროცესის დაპროექტება (დიზაინი)**

ახლა შეიძლება განსაზღვროს ქმედებები, რომლებიც დაამუშავებენ შეკვეთას, რომელიც გადმოიცა. დავიწყოთ Sequence

ქმედების გადმოტანით მუშა პროცესის დიაგრამაზე. შემდეგ გადმოვიტანოთ WriteLine ქმედება. Text–სთვის ჩავწეროთ “Order Received”. გადმოვიტანოთ Assign ქმედება WriteLine–ს ქვემოთ. DisplayName დავაყენოთ Initialize Total. თვისებისთვის TotalAmount მნიშვნელობა შევიტანოთ 0.

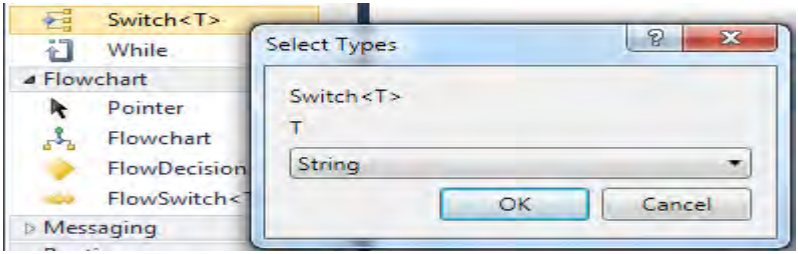


ნახ.2.9. საწყისი ინიციალიზაცია TotalAmount=0

- **Switch ქმედება**

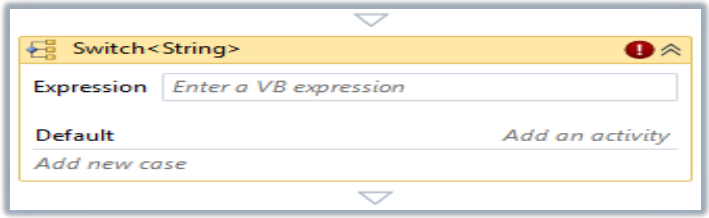
Switch ქმედება მუშაობს ისე, როგორც Switch ოპერატორი C# ენაში. ის საშუალებას იძლევა შესრულდეს sequence ქმედებები გამოსახულებათა ანალიზის საფუძველზე. Switch აქტიურობა გამოიყენება ShippingMethod მეთოდის განსაზღვრისთის, ბარგის დამუშავების ღირებულების დადგენის მიზნით.

ეს ნიშნავს, რომ იგი განსაზღვრულია როგორც კლასის შაბლონი და მუშაობს მონაცემთა სხადასხვა ტიპებთან. გადმოვიტანოთ ინსტრუმენტების პანელიდან Switch ქმედება Assign ქმედების ქვემოთ, შევარჩიოთ გამოსახულების (Expression) ტიპი. (ნახ.2.10). გამოჩნდება ფანჯარა; ShippingMethod მეთოდი String ტიპისაა.



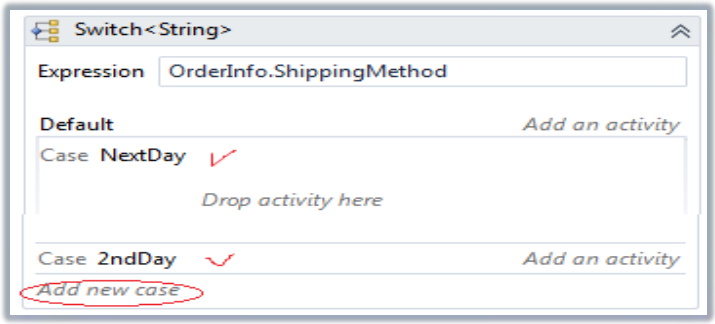
ნახ.2.10.

DisplayName–ში ჩავწეროთ Handling Charges. მივიღებთ:



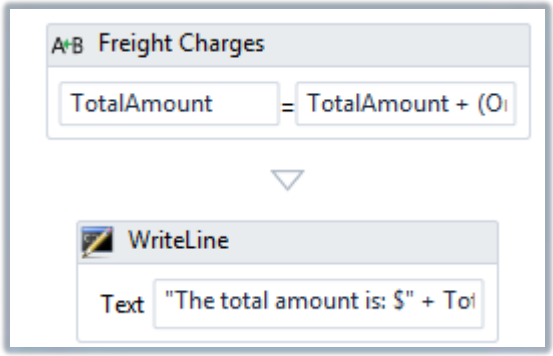
ნახ.2.11

Switch ქმედებას აქვს Expression თვისება, მიმდინარე ბლოკი და რამდენიმე მომხმარებლის მიერ სპეციფირებული case ბლოკი. შევიტანოთ გამოსახულება **OrderInfo.ShippingMethod.** აგრეთვე Ad new case–ში შევიტანოთ ჯერ NextDay, შემდეგ 2ndDay. შედეგები 2.11 ნახაზზეა მოცემული.



ნახ.2.12

დავამატოთ ახალი Assign და WriteLine (ნახ.2.13).



ნახ.2.13

Properties-ში შევიტანოთ გამოსახულებები: TotalAmount და TotalAmount + (OrderInfo.TotalWeight * 0.5D) – პირველში, "The total amount is: $" + TotalAmount.ToString() – მეორეში.

Program.cs –ის კოდი მოცემულია 2.2 ლისტინგში.

```
// --- ლისტინგი-2_2 ------
using System;
using System.Activities;
using System.Activities.Statements;
using System.Collections.Generic;
namespace OrderProcess
{
    class Program
    {
        static void Main(string[] args)
        {
            Order myOrder = new Order
            {
                OrderID = 1,
                Description = "Need some stuff",
                ShippingMethod = "2ndDay",
                TotalWeight = 100
            };
```

```
  // create dictionary with input arguments for the workflow
       IDictionary<string, object> input =
                     new Dictionary<string, object>
        {
          { "OrderInfo" , myOrder }
        };
        // execute the workflow
      IDictionary<string, object> output =
           WorkflowInvoker.Invoke(new OrderWF(), input);

      // Get the TotalAmount returned by the workflow
       decimal total = (decimal)output["TotalAmount"];
  Console.WriteLine("Workflow returned ${0} for my
                                  order total", total);
      Console.WriteLine("Press ENTER to exit");
      Console.ReadLine();
     }
   }
}
```

OrderWF.xaml ფაილის კოდის ფრაგმენტი <Switch . . . > -თვის:

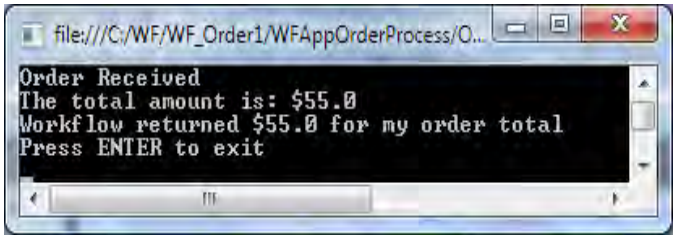
```
     . . .
<Switch x:TypeArguments="x:String" DisplayName="Handling
     Charges" Expression="OrderInfo.ShippingMethod"

    sap:VirtualizedContainerService.HintSize="476,264">
       <Switch.Default>
        <Add x:TypeArguments="x:Decimal, x:Decimal,
            x:Decimal" DisplayName="Add 5"
          sap:VirtualizedContainerService.HintSize="458,100"
          Left="[TotalAmount]" Result="[TotalAmount]"
              Right="[5.0D]" />
       </Switch.Default>
       <sap:WorkflowViewStateService.ViewState>
        <scg3:Dictionary x:TypeArguments="x:String,
                  x:Object">
           <x:Boolean x:Key="IsExpanded">True</x:Boolean>
           <x:Boolean x:Key="IsPinned">False</x:Boolean>
```

```
        </scg3:Dictionary>
      </sap:WorkflowViewStateService.ViewState>
  <Add x:TypeArguments="x:Decimal, x:Decimal, x:Decimal"
       x:Key="NextDay" DisplayName="Add 15"
    sap:VirtualizedContainerService.HintSize="456,100"
    Left="[TotalAmount]" Result="[TotalAmount]"
          Right="[15.0D]" />

  <Add x:TypeArguments="x:Decimal, x:Decimal, x:Decimal"
       x:Key="2ndDay" DisplayName="Add 10"
   sap:VirtualizedContainerService.HintSize="456,100"
   Left="[TotalAmount]" Result="[TotalAmount]"
          Right="[10.0D]" />
</Switch>
      ...
```

მუშაობის შედეგები:



ნახ.2.14

2.2. რთული ბიზნესპროცესის გამოსახულებათა ქმედებები (ლაბ.N7)

მიზანი: Visual Studio.NET პაკეტის WF-ის სამუშაო გარემოში რთული ბიზნესპროცესის გამოსახულებათა ქმედებების (Expression Activities) გაცნობა.

1. თეორიული ნაწილი

წინა ლაბორატორიულ ამოცანაში ჩვენ განვსაზღვრეთ გადამრთველის ორი case-ბლოკი NextDay და 2ndDay, პლუს ერთი ბლოკი ავტომატურად გამოყენებადი ShippingMethod–ის მიერ, როცა არაა წინა ორი მითითებული. ამჯერად უნდა მიეთითოს ქმედებები (მაგალითად, Sequence), რომლებიც უნდა შესრულდეს თითოეული შემთხვევის დროს. ჩვენი პროექტისთვის გამოვიყენოთ Add ქმედება.

შენიშვნა: System.Activities.Expressions სახელსივრცე შეიცავს ქმედებებს, რომლებიც გამოიყენება ბიზნეს–პროცესებში, მათ შორის Add, Subtract, Multiply და Divide. იგი შეიცავს აგრეთვე ისეთ ჩაშენებულ ქმედებებს, როგორიცაა Equal, GreaterThan, And და Or, რომლებიც გამოიყენება გამოსახულებათა შესაფასებლად. მათი გამოყენება შესაძლებელია უშუალოდ მუშა პროცესებში.

2. პრაქტიკული ნაწილი

სამწუხაროდ, ინსტრუმენტების პანელზე არაა Add ქმედება. ჩვენ უნდა შევიდეთ XAML კოდის რედაქტირების არეში. შევინახოთ პროექტი.

Solution Explorer–ში OrderWF.xaml–ზე მარჯვენა ღილაკით ავირჩიოთ Code view. სისტემა მოითხოვს არსებული ფანჯრის დახურვას, „დიახ“.

Switch ქმედება განისაზღვრება შემდეგი კოდით, რომლის ლისტინგი 2.3 მოცემულია ქვემოთ.

```
<!------ ლისტინგი_2.3 ---------->
<Switch x:TypeArguments="x:String" DisplayName="Handling Charges"
        Expression="OrderInfo.ShippingMethod"
        sap:VirtualizedContainerService.HintSize="476,158">
    <sap:WorkflowViewStateService.ViewState>
     <scg3:Dictionary x:TypeArguments="x:String, x:Object">
      <x:Boolean x:Key="IsExpanded">True</x:Boolean>
      <x:Boolean x:Key="IsPinned">False</x:Boolean>
     </scg3:Dictionary>
    </sap:WorkflowViewStateService.ViewState>
    <x:Null x:Key="NextDay" />
    <x:Null x:Key="2ndDay" />
  </Switch>
```

საყურადღებოა, რომ x: Null ატრიბუტები NextDay და 2ndDay ბლოკებში ნიშნავს, რომ ჯერ არაა ქმედებები მათთვის განსაზღვრული. ამიტომ შევცვალოთ ეს სტრიქონები შემდეგით:

```
<Add x:TypeArguments="s:Decimal, s:Decimal, s:Decimal"
          x:Key="NextDay"
      DisplayName="Add 15" Left="[TotalAmount]"
           Result="[TotalAmount]"
                       Right="[15.0D]" />
<Add x:TypeArguments="s:Decimal, s:Decimal, s:Decimal"
          x:Key="2ndDay"
    DisplayName="Add 10" Left="[TotalAmount]"
           Result="[TotalAmount]"
                       Right="[10.0D]" />
```

დავამატოთ შემდეგი კოდი უშუალოდ წინა კოდის წინ, რათა განისაზღვროს ავტომატურად არსებული შემთხვევის ვარიანტი:

```
<Switch.Default>
    <p:Add x:TypeArguments="s:Decimal, s:Decimal, s:Decimal"
                                                DisplayName="Add 5"
     Left="[TotalAmount]" Result="[TotalAmount]" Right="[5.0D]" />
</Switch.Default>
```

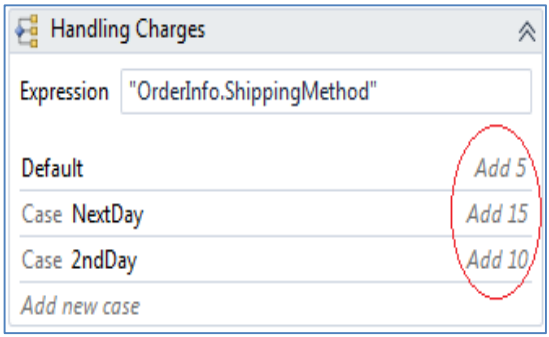
Add ქმედებას აქვს სამი თვისება: Left, Right და Result. ცვლადის მნიშვნელობა Right თვისებიდან ემატება Left–ის მნიშვნელობას, ხოლო ჯამი ინახება Result–ში. Left და Result თვისებებისთვის ყენდება TotalAmount არგუმენტი. Right თვისება განისაზღვრება როგორც სტატიკური ცვლადი, რომელიც განსხვავებულია ყოველი კონკრეტული case -სთვის.

Switch ქმედების საბოლოო განსაზღვრა დაკომპლექტებულია 2.4 ლისტინგში.

```
<!—— ლისტინგი 2.4 ———————————— —>
<Switch x:TypeArguments="x:String" DisplayName="Handling Charges"
        Expression="OrderInfo.ShippingMethod"
        sap:VirtualizedContainerService.HintSize="476,158">
    <Switch.Default>
    <Add x:TypeArguments="s:Decimal, s:Decimal, s:Decimal"
DisplayName="Add 5"
                    Left="[TotalAmount]" Result="[TotalAmount]"
Right="[5.0D]" />
  </Switch.Default>
   <Add x:TypeArguments="s:Decimal, s:Decimal, s:Decimal"
x:Key="NextDay"
       DisplayName="Add 15" Left="[TotalAmount]"
Result="[TotalAmount]"
        Right="[15.0D]" />
```

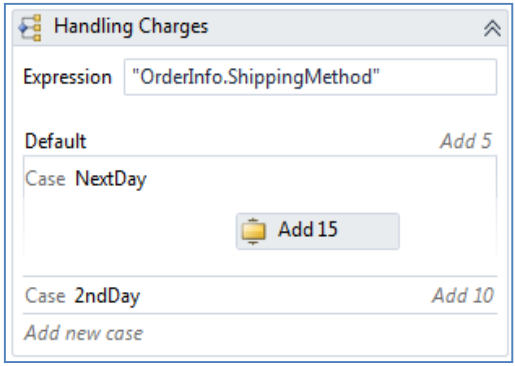
```
    <Add x:TypeArguments="s:Decimal, s:Decimal, s:Decimal"
x:Key="2ndDay"
        DisplayName="Add 10" Left="[TotalAmount]"
Result="[TotalAmount]"
        Right="[10.0D]" />
</Switch>
```

შევინახოთ პროექტი და Solution Explorer–დან გამოვიტანოთ OrderWF.xaml ფაილი დიზაინერში. Switch ქმედება ასე გამოიყურება (ნახ.2.15):



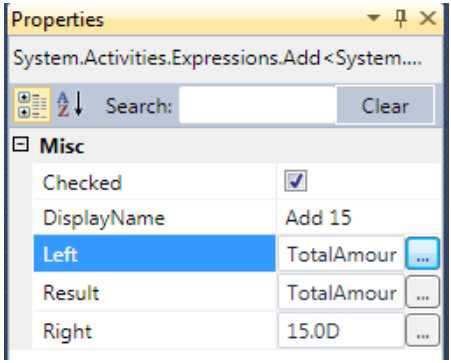
ნახ.2.15

როდესაც ShippingMethod არის NextDay, მაშინ TotalAmount–ს ემატება $ 15, თუ არის 2ndDay, მაშინ ემატება $ 10. მიწოდების ყველა სხვა მეთოდისთვის ავტომატურად ემატება $ 5. დიაგრამა ასახავს ყველა case შემთხვევას და DisplayName–ს შესრულებადი ქმედებისთვის შესაბამის case-ში. თუ დავდგებით Switch ქმედების რომელიმე case-ბლოკზე, მაშინ გააქტიურდება ეს ბლოკი (ნახ.2.16):



ნახ.2.16. Switch ქმედების გაფართოება

Add ქმედებაზე დაკლიკვით გამოვა თვისებების ფანჯარა (ნახ.2.17). აქ შესაძლებელია თვისებების შეცვლა, საჭიროების შემთხვევაში.



ნახ.2.17. Add ქმედების თვისებების ფანჯარა

შენიშნა: ათობითი მნიშვნელობა შეიტანება 15.0D ფორმატით(VB-ენა). C# -ის დროს, მაგალითად, 15.0m მოგვცემს შედომას. უნდა გვახსოვდეს, რომ გამოსახულებები შეიტანება VB–სინტაკსით !!!

გადმოვიტანოთ სქემაზე შემდეგი Assign ქმედება Switch-ის ქვემოთ და დავაყენოთ DisplayName სატრანსპორტო ხარჯი (Freight Charges). To თვისებისთვის შევიტანოთ TotalAmount. ხოლო Value თვისებისთვის შევიტანოთ:

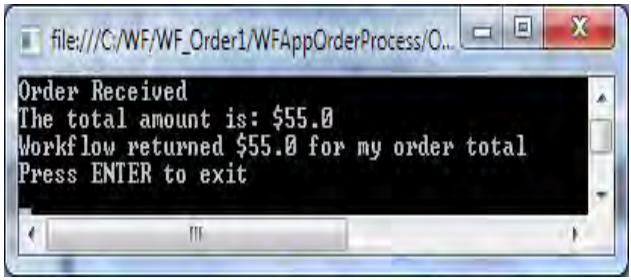
TotalAmount + (OrderInfo.TotalWeight * 0.50D)

ეს ფორმულა მიწოდების ხარჯებისთვის დაამატებს $ 0,50 –ს ყოველ ფუნტზე (წონა).

გადმოვიტანოთ ახალი WriteLine ქმედება „Freight Charges“–ს შემდეგ და Text თვისებაში ჩავწეროთ:

"The total amount is: $" + TotalAmount.ToString()

ეკრანზე გამოიტანება გაანგარიშებული შეკვეთის საერთო ღირებულება. საბოლოო ბიზნეს–პროცესის სქემა ასე გამოიყურება (ნახ.2.18).



ნახ.2.18

2.3. განმეორებადი ქმედებები (ლაბ.N8)

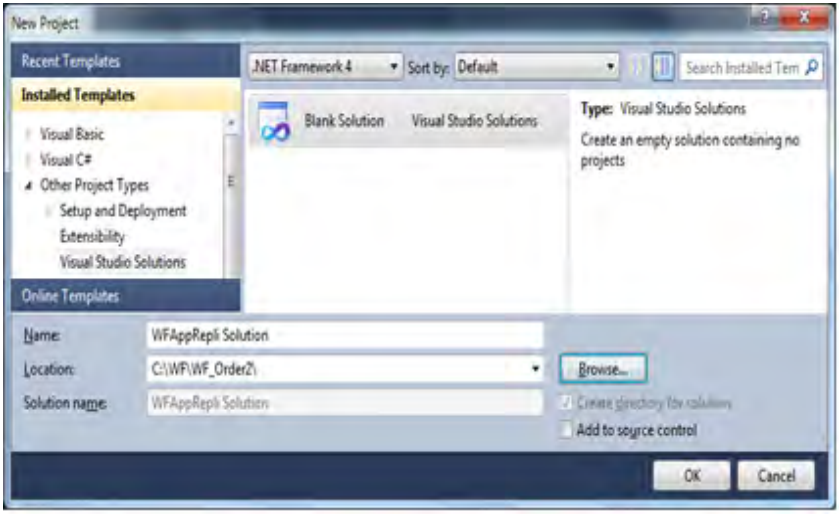
მიზანი: Visual Studio.NET პაკეტის WF-ის სამუშაო გარემოში რთული ბიზნესპროცესის შესრულების გაცნობა განმეორებადი ქმედებების საფუძველზე.

1. თეორიული ნაწილი

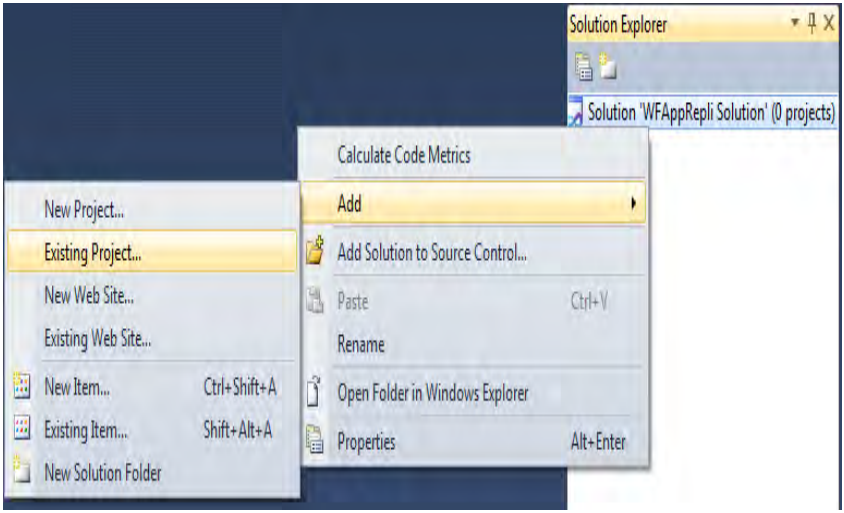
წინა ლაბორატორიის ამოცანაში შექმნილი ბიზნესპროცესი ითვლიდა შეკვეთის საერთო ღირებულებას და გადასცემდა მას არგუმენტის სახით. ის შედგებოდა მხოლოდ დამუშავების და მიწოდების ხარჯებისგან. ამ თავში დავამატებთ შეკვეთის თითოეული პოზიციის ღირებულებას. ამისათვის კი საჭირო იქნება ქმედების შესრულება ყოველი ელემენტისთვის.

2. პრაქტიკულ ნაწილი

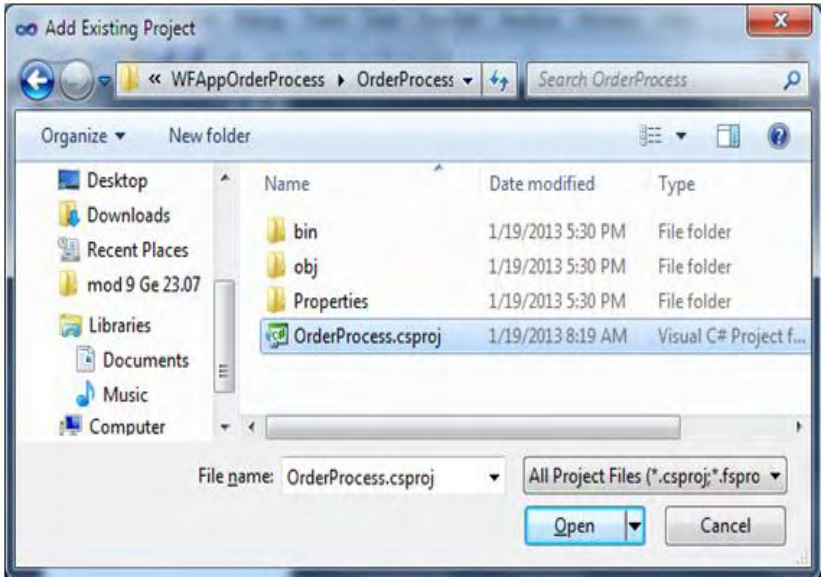
წინა პროექტის გამოყენება: ავამუშავოთ Visual Studio, შევქმნათ ახალი პროექტი WFAppRepli Solution სახელით (ნახ.2.19).



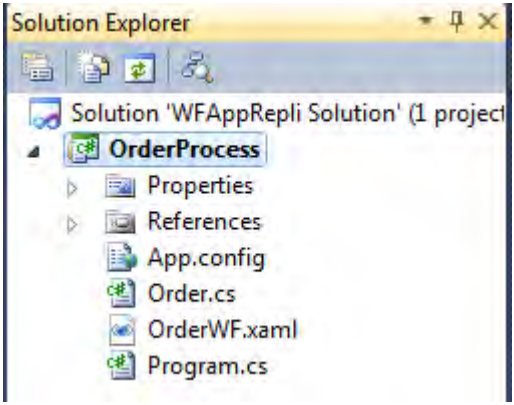
ნახ.2.19. Blank Solution პროექტის შექმნა



ნახ.2.20. წინა პროექტის გადმოტანა ახალში



ნახ.2.21



ნახ.2.22. შედეგი Solution Explorer-ში

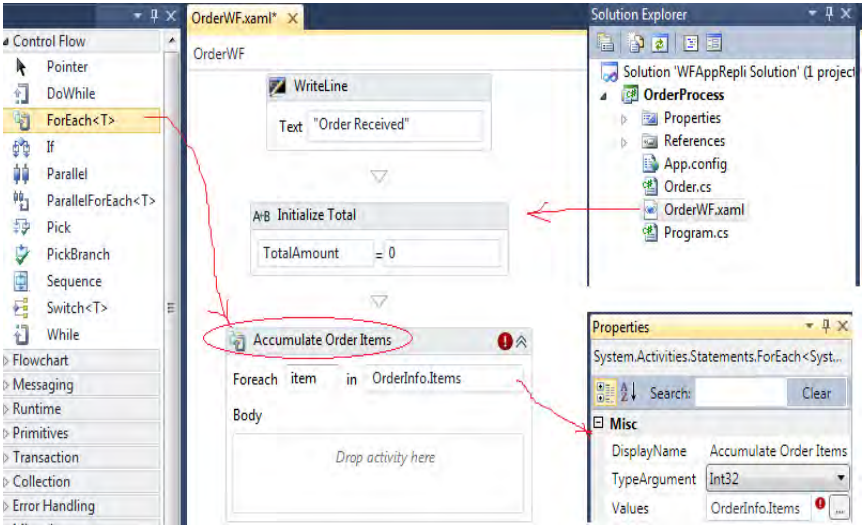
3. შეკვეთის ელემენტების დამუშავება
(Adding OrderItem Processing)

ახლა პროგრამაში უნდა დავამატოთ ბიჯი შეკვეთის ცალკეული კომპონენტების ღირებულების გასაანგარიშებლად.

- **ForEach ქმედება**

აქტიურობა ForEach ასრულებს ქმედებას (ან ქმედებათა მიმდევრობას) ყოველი ელემენტისთვის კოლექციაში. ესაა სწორედ ის ქმედება, რომელიც მოცემულ პროექტში გამოდგება.

გავხსნათ OrderWF.xaml ფაილი დიზაინის რეჟიმში. გადმოვიტანოთ ForEach <T> ქმედება “Initialize Total” ქმედების ქვემოთ. DisplayName შევცვალოთ Accumulate Order Items–ით. ქმედება შეიძლება იყოს შეკუმშული. თუ ასეა, მაშინ გააფართოვეთ (ღილაკი ზედა მარჯვენა კუთხეში). მიიღება (ნახ.2.21).



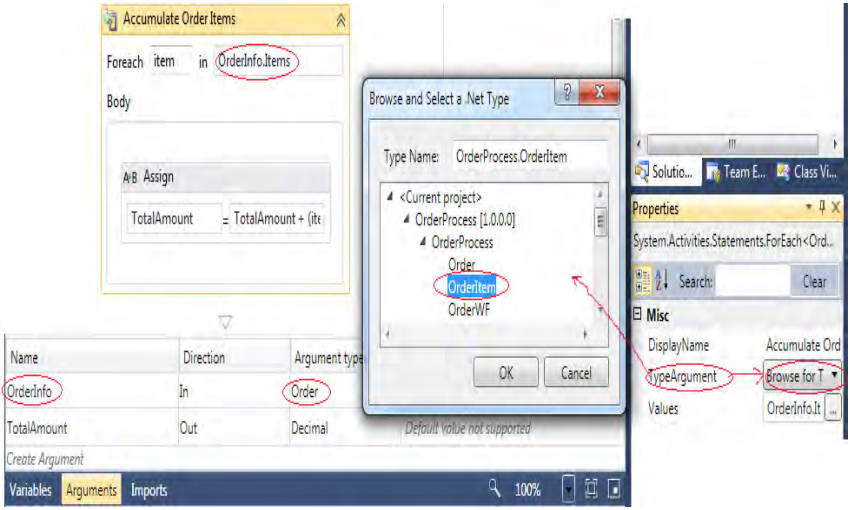
ნახ.2.2.21. ForEach ქმედების საწყისი მდგომარეობა

Expression ველში ჩავწეროთ OrderInfo.Items.

ქმედებაში <T>–ს არსებობა მიუთითებს იმაზე, რომ იგი არის უნივერსალური (generic) კლასი და საჭიროა ტიპის განსაზღვრა, რომელიც კოლექციაში იქნება. Propertries-ში ავტომატურად მიეთითება Int32 ტიპი. წითელი წრე კუთხეში გვამცნობს, რომ შეცდომაა, შეტანილი გამოსახულება (OrderInfo.Items) არ შეიცავს მთელ რიცხვებს.

OrderInfo.Items არის OrderItem ობიექტების ერთობლიობა Order შეკვეთიდან, რომელიც მიღებულ იქნა სამუშაო პროცესში. Properties–ის TypeArgument–ის კომბობოქსის სიაში ავირჩიოთ ტიპი Browse-თი. OrderProcess–ში ავირჩიოთ OrderItem (ნახ.2.2.22).

ამ პროექტში ჩვენ უნდა მივიღოთ მარტივად შეკვეთის თითოეული ელემენტის ფიქსირებული ფასი. რეალურ სცენარით ასეთი თვისებები ამოიღება მონაცემთა ბაზიდან. ახლა გადმოვიტანოთ სქემაზე Assign ქმედება ForEach აქტიურობაზე.



ნახ.2.22. OrderItem კლასის არჩევა

ForEach ქმედებაზე შეიძლება მხოლოდ ერთი ქმედების გადმოტანა. თუ საჭიროა ერთზე მეტი ქმედების გამოყენება, მაშინ უნდა გადმოვიტანოთ ჯერ Sequence აქტიუროზა და შემდეგ მასზე დავდოთ რამდენიმე სხვა ქმედება.

Assign ქმედების თვისებებში შევიტანოთ:

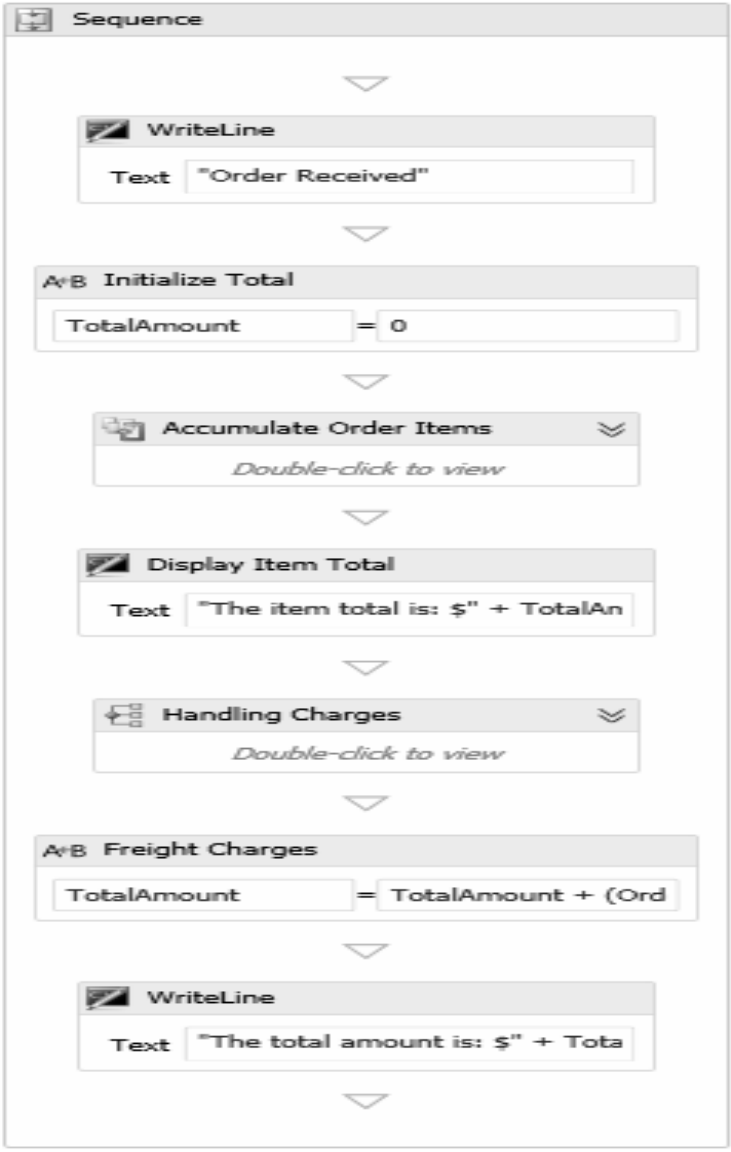
To–თვის TotalAmount და Value–თვის TotalAmount + (item.Quantity * 10.0D). ეს ნიშნავს, რომ $10 ემატება ყოველ ელემენტზე.

Assign ქმედების ქვემოთ გადმოვიტანოთ WriteLine აქტიურობა, რომლისთვისაც ჩავწეროთ სახელი Display Item Total. მისიText თვისებისთვის კი შევიტანოთ გამოსახულება:

```
"The item total is: $" + TotalAmount.ToString()
```

საბოლოოდ, 2.23 ნახაზზე მოცემულა მთლიანი პროცესის workflow სქემა.

აქ შეკუმშული სახითაა მოცემუ;ი ქმედებათა რიგი, რომელთა გაშლა შესაძლებელია მარჯვენა ზედა კუთხეში არსებული ისრით.


Sequence
WriteLine
Text
"Order Received"
Initialize Total
TotalAmount
=
0
Accumulate Order Items
Double-click to view
Display Item Total
Text
"The item total is: $" + TotalAn
Handling Charges
Double-click to view
Freight Charges
TotalAmount
=
TotalAmount + (Ord
WriteLine
Text
"The total amount is: $" + Tota

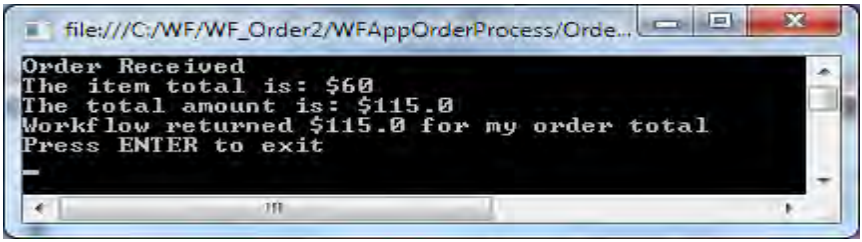
ნახ.2.23

- შეკვეთის ელემენტების დამატება

ForEach ფუნქციის შემოწმებამდე საჭიროა აპლიკაციაში ცვლილების შეტანა, რათა Order კლასში შესაძლებელი იყოს ზოგიერთი OrderItem ობიექტის დამატება. გავხსნათ Program.cs ფაილი და ჩავამატოთ უშუალოდ Order კლასის შემდეგ ასეთი კოდი:

```
// Add some OrderItem objects
            myOrder.Items.Add(new OrderItem
            {
                OrderItemID = 1,
                Quantity = 1,
                ItemCode = "12345",
                Description = "Widget" } );
            myOrder.Items.Add(new OrderItem
            {
                OrderItemID = 2,
                Quantity = 3,
                ItemCode = "12346",
                Description = "Gadget" } );
            myOrder.Items.Add(new OrderItem
            {
                OrderItemID = 3,
                Quantity = 2,
                ItemCode = "12347",
                Description = "Super Widget" } );
```

პროგრამის ამუშავებით მივიღებთ შედეგებ (ნახ.2.24).



ნახ.2.24

- **ParallelForEach ქმედება**

ნაცვლად ForEach ქმედებისა შესაძლებელია ParallelForEach ქმედების გამოყენება. იგი კონფიგურირებულია ზუსტად ForEach – ით. ერთადერთი განსხვავება არის თუ როგორ სრულდება აქტიურობა. როგორც მისი სახელი ვარაუდობს, ParallelForEach აქტიურობა სრულდება „შვილი პროცესის“ პარალელურად, ხოლო ForEach ქმედება ახორციელებს ამ პროცესებს მიმდევრობით.

ჩვენი პროექტისთვის ეს ეფექტი არჩანს, მაგრამ თუ რთული მიმდევრობითი პროცესებია გასაშვები, მაშინ მათი პარალელურად შესრულება უფრო მისაღებია. მაგალითად, თუ უნდა გაიგზავნოს შეტყობინება და დაველოდოთ პასუხს, შეიძლება პარალელურად გაიშვას სხვაპროცესი, Wait დრო არ არის დაიკარგება.

ამის შესამოწმებლად ჩვენ პროექტში შეიძლება წაიშალოს ForEach აქტიურობა და დავდოთ ParallelForEach ქმედება იგივე ადგილზე. კონფიგურირებით და ამუშავებით ვნახავთ, რომ გვაქვს იგივე შედეგი.

2.4. გამონაკლისების დამუშავება (ლაბ.N9)
Exception Handling

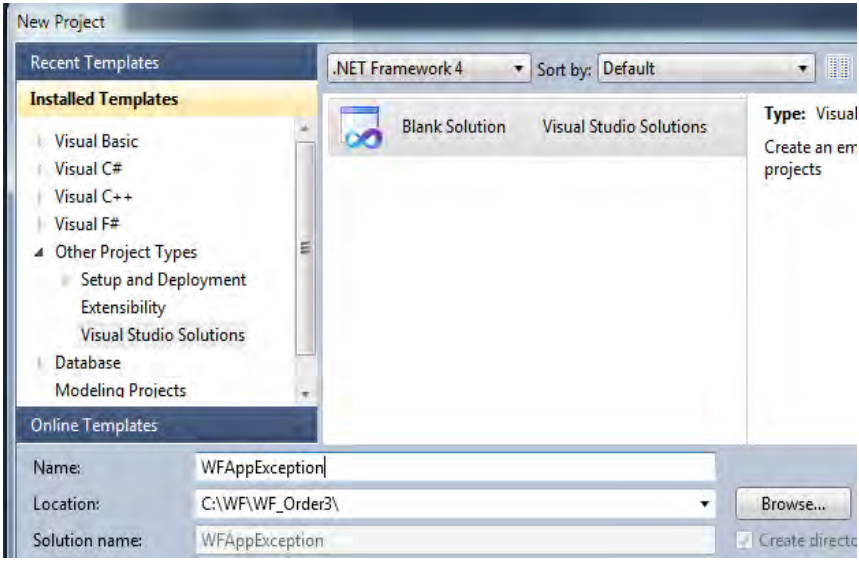
მიზანი: გამონაკლისების (გამოსარიცხი პროცესების) ანალიზის ჩატარების და მათი აღმოფხვრის ხერხების შესწავლა.

1. თეორიული ნაწილი

უნდა დავამატოთ ლოგიკა, რათა მოხდეს გადაამოწმება, რომ შეკვეთის თითოეული ელემენტი არის საწყობში. ამისათვის, იტარეაციულად შეკვეთის თითოეული ელემენტისთვის, როგორც ეს გაკეთდა წინა თავში. თუ პროდუქცია (ელემენტი) არაა საწყობში, უნდა გამოვრიცხოთ ეს გამონაკლისი შემთხვევა, რომელიც დააფიქსირა workflow პროცესმა.

2. პრაქტიკული ნაწილი

ახალი პროექტის შესაქმნელად გამოვიყენოთ წინა პროექტი:



ნახ.2.25

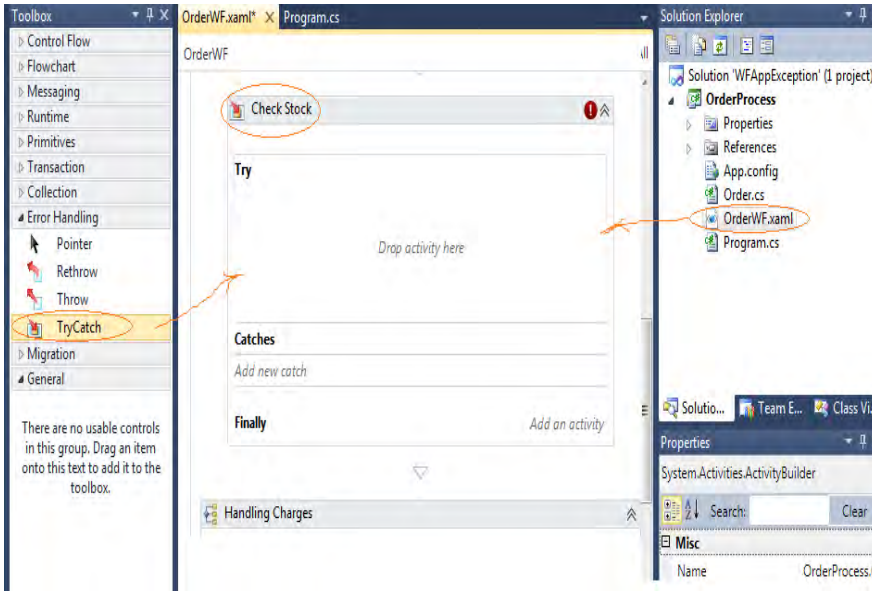
Solution Explorer–დან დავამატოთ Add-> Existing Project ჩვენი წინა პროექტის ფაილი

ეს ფაილი ინახება OrderProcess ფოლდერში.

- მარაგის შემოწმების ქმედება (Check Stock Activity)

დავამატოთ ჩვენ პროექტს ლოგიკა, რომელიც დაადგენს არის თუ არა ელემენტების საკმარისი მარაგი შეკვეთისთვის.

გავხსნათ OrderWF.xaml ფაილი დიზაინის რეჟიმში. გადმოვიტანოთ TryCatch ქმედება სქემაზე "Handling Charges" აქტიურობის შემდეგ. შევცვალოთ DisplayName სახელი Check Stockით. მიიღება (ნახ.2.26).



ნახ.2.26. TryCatch ქმედება

TryCatch შედგება სამი ნაწილისგან:

- Try ნაწილში მოთავსდება იმ ქმედებათა მიმდევრობა, რომელთაც პოტენციურად შეუძლია გამონაკლისი სიტუაციების გენერირება;
- Catch ნაწილში განისაზღვრება ერთი ან რამდენიმე Catch–ობიექტი. ყოველი Catch ობიექტი ამუშავებს კონკრეტულ გამონაკლისს, ამიტომ მისი ყოველ ტიპისთვის უნდა დაიწეროს ცალკე დამუშავების პროცედურა;
- Finally ნაწილი, არაა სავალდებულო. აქ უნდა მოთავსდეს იმ ქმედებათა მიმდევრობა, რომელიც სრულდება Try ქმედების შემდეგ.

- გამონაკლისების განსაზღვრა

ახლა განვსაზღვროთ გამონაკლისი სიტუაცია, როცა შეკვეთისთვის საჭირო ელემენტი არაა საწყობში. გავხსნათ Order.cs ფაილი და ჩავამატოთ შემდეგი კოდი, რომელიც განსაზღვრავს OutOfStockException კლასს. Order კლასის განსაზღვრის შემდეგ OrderProcess სახელსივრცის შიგნით პროგრამის ლისტინგს ასეთი სახე ექნება:

```
// -- ლისტინგი-2.4 --------
using System;
using System.Collections.Generic;
namespace OrderProcess
{
    public class OrderItem
    {
        public int OrderItemID { get; set; }
        public int Quantity { get; set; }
        public string ItemCode { get; set; }
        public string Description { get; set; }
    }
```

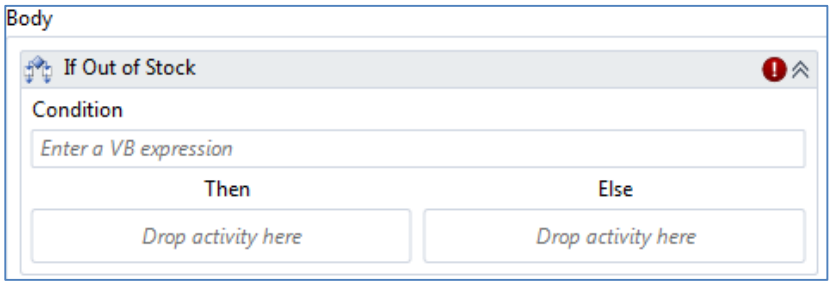
```
    public class Order
    { public Order()
        {
            Items = new List<OrderItem>();
        }
        public int OrderID { get; set; }
        public string Description { get; set; }
        public decimal TotalWeight { get; set; }
        public string ShippingMethod { get; set; }
        public List<OrderItem> Items { get; set; }
    }
    //---------------------------------------------
    // გამონაკლისი სიტუაციის განსაზღვრა, როცა შეკვეთისთვის
    // საჭირო ელემენტი არაა საწყობში
    //---------------------------------------------
    public class OutOfStockException : Exception
    {
        public OutOfStockException() : base()
        {
        }
        public OutOfStockException(string message) :
base(message)
        {
        }
    }
}
```

- **ForEach ქმედება**

გადმოვიტანოთ ForEach ქაქტიურობა Try ნაწილში და DisplayName შევცვალოთ სახელით Check Each Item. გავშალოთ ბლოკი და Expression ველში ჩავწეროთ OrderInfo.Items. თვისებებში შევცვალოთ TypeArgument ტიპი (როგორც წინა თავში) და ავირჩიოთ OrderItem კლასი.

- **If ქმედება**

გადმოვიტანოთ If ქმედება Try ნაწილში და DisplayName შევცვალოთ to If Out of Stock სახელით.



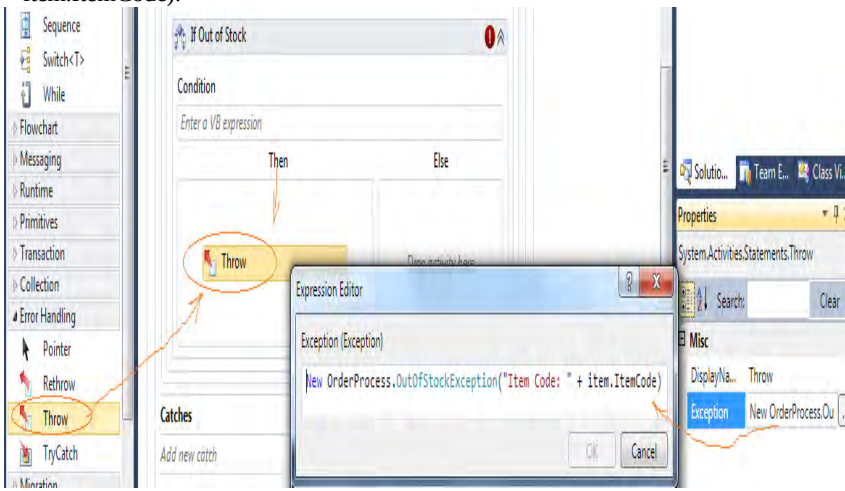
ნახ.2.27

Condition თვისებაში შევიტანოთ გამოსახულება: item.ItemCode = "12346"

შენიშვნა: რეალურ სისტემაში „არსებული ელემენტები“ უნდა შემოწმდეს მოპნაცემთა ბაზაში. აქ, პროექტის გადასამარტივებლად, გამოყენებულია ItemCode.

- **Throw ქმედება**

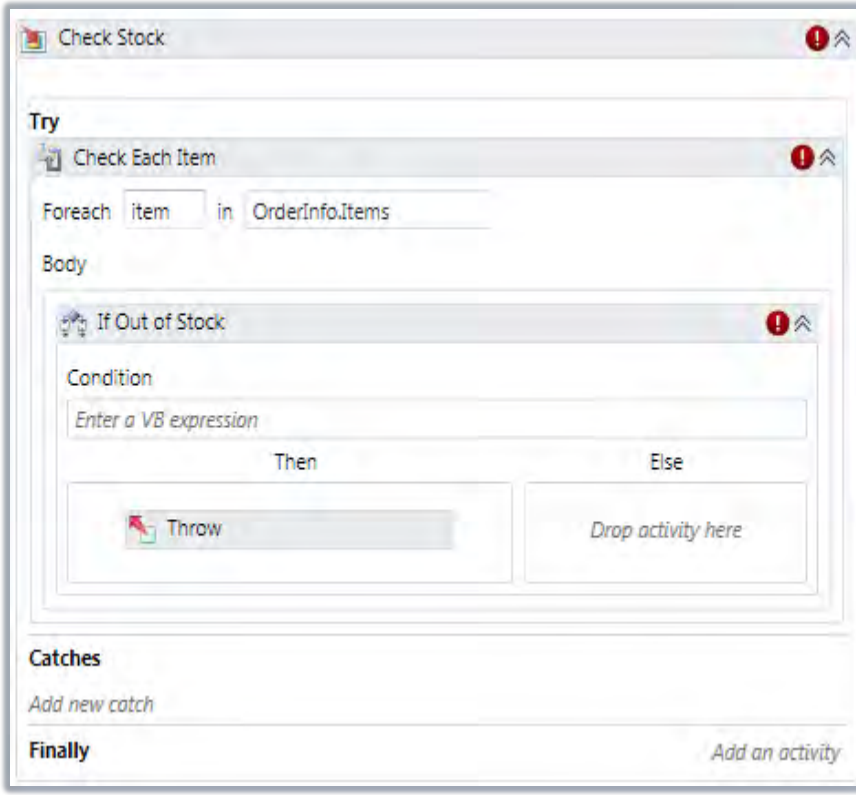
გადმოვიტანოთ Then ბლოკში Throw ქმედება. თვისებაში Exception შევიტანოთ New OrderProcess.OutOfStockException("Item Code: " + item.ItemCode).



ნახ.2.28

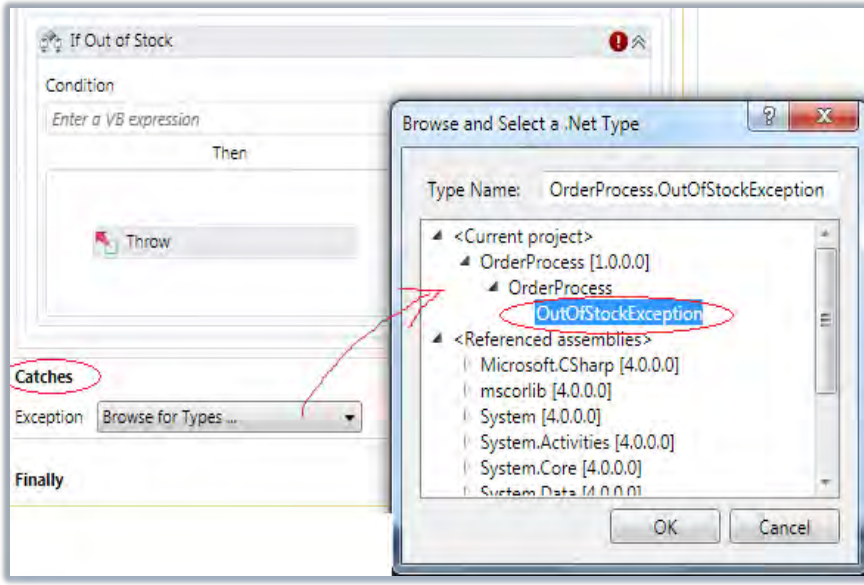
- **Catch ქმედება**

  “Check Stock” ქმედება ასე გამოიყურება:

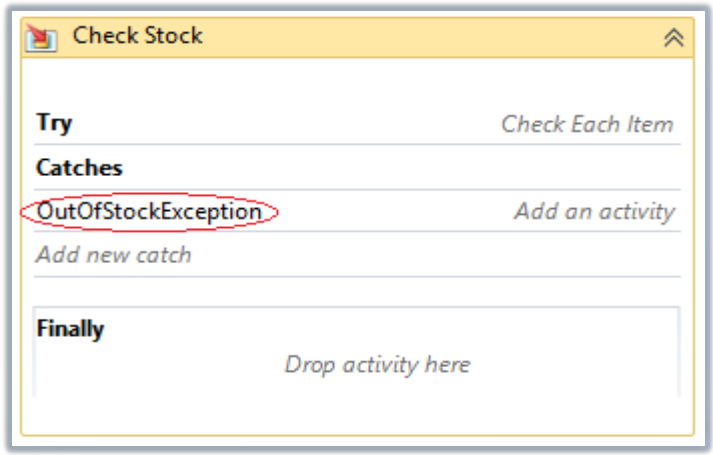


ნახ.2.29. TryCatch ქმედების სტრუქტურა

დავკლიკოთ *Add new catch* ღილაკი.



ნახ.2.30. OutOfStockException –ის არჩევა

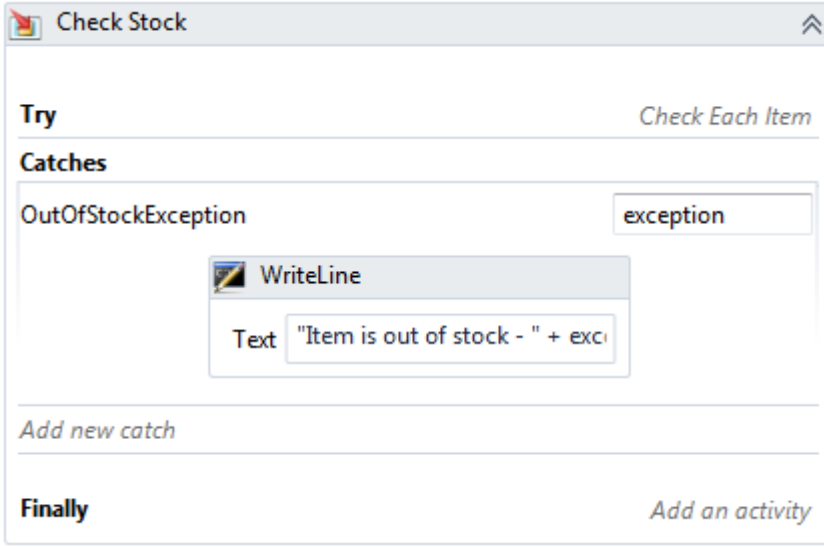


ნახ.2.31. Catch ქმედების ბლოკის დაკომპლექტება

ყოველი გამონაკლისი შემთხვევისთვის უნდა განისაზღვროს შესაბამისი ქმედება და ჩაიდოს ბლოკში. ჩვენი პროექტისთვის უნდა გამოვიტანოთ შეტყობინება, როცა არაა პროდუცია. გადმოვიტანოთ WriteLine ქმედება Catch ნაწილში და შევიტანოთ Text-ში:

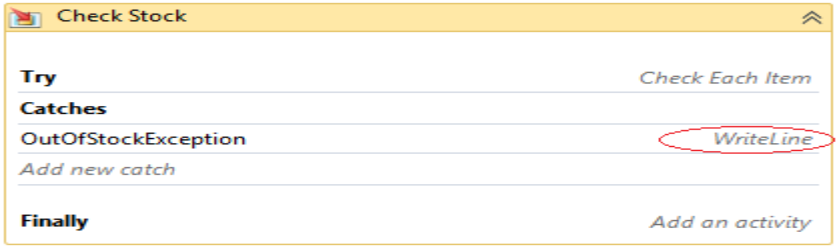
"Item is out of stock - " + exception.Message

საბოლოოდ “Check Stock” ბლოკი ასე გამოიყურება:



ნახ.2.32

Expand-Collapse (გაფართოება/შეკუმშვის) გამოყენების შემდეგ წინა სქემა მიიღებს ასეთ სახეს.

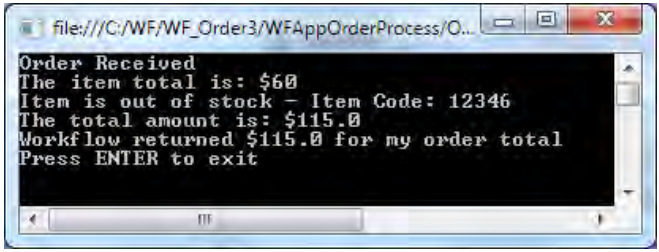


ნახ.2.33

Try, Catches და Finally ნაწილები შეჯამებულია. Try გვიჩვენებს, რომ არსებობს ქმედება სახელით "Check Each Item" (შემოწმდეს თითოეული ელემენტი). Catch –ში ჩამოთვლილია გამონაკლისები, რომლებიც მუშავდება WriteLine ქმედებით (ჩვენთან ასეთი ერთია OutOfStockException). ვინაიდან Finaly–ში არაა ქმედებები, აქ არის Add an Activity ღილაკი, რომლითაც შესაძლებელია დამატებები.

- აპლიკაციის ამუშავება

  პროგრამის მუშაობის შედეგები ასეთია (ნახ.2.34).



ნახ.2.34

- გამონაკლისები (Exceptions)

გამონაკლისების (ან გამოსარიცხი პროცესების) დამუშავება მნიშვნელოვანი ფრაგმენტია პროგრამული სისტემების მუშაობის

დროს გაუთვალისწინებელი მოვლენების გამო პროგრამების ავარიული დამთავრების პრევენციისათვის.

გამონაკლისების დამუშავების პროცედურა TryCatch აქტიურობით რეალიზდება. კერძოდ მთლიანი მუშა პროცესი თავსდება Try ნაწილში.

მაგალითისათვის მოვიტანოთ შემდეგი კოდის ლისტინგი:

```
// --- ლისტინგი-2.7 ---- TryCatch1---
Sequence
{
  CheckStock (Try)
   {
     Check Each Item (ForEach)
      {
        If Out of Stock (If)
          {
            Throw Exception (Then)
          }
      }
   }
   Catch
     {
      }

     Remaining Activities
}

// --- ლისტინგი-2.8 ---- TryCatch2---
Sequence
{ Check Each Item (ForEach)
     {
       (Try)
          {
            If Out of Stock (If)
```

```
                    {
                      Throw Exception (Then)
                    }
                }
              Catch
                {
                }
        }
}

// --- ლისტინგი-2.9 ---- TryCatch3---
 (Try)
{  Sequence
     {
        Check Each Item
           {
              If Out of Stock
                 {
                   Throw Exception
                 }
           }
         Remaining Activities
     }
 }
 Catch
 {
  }
```

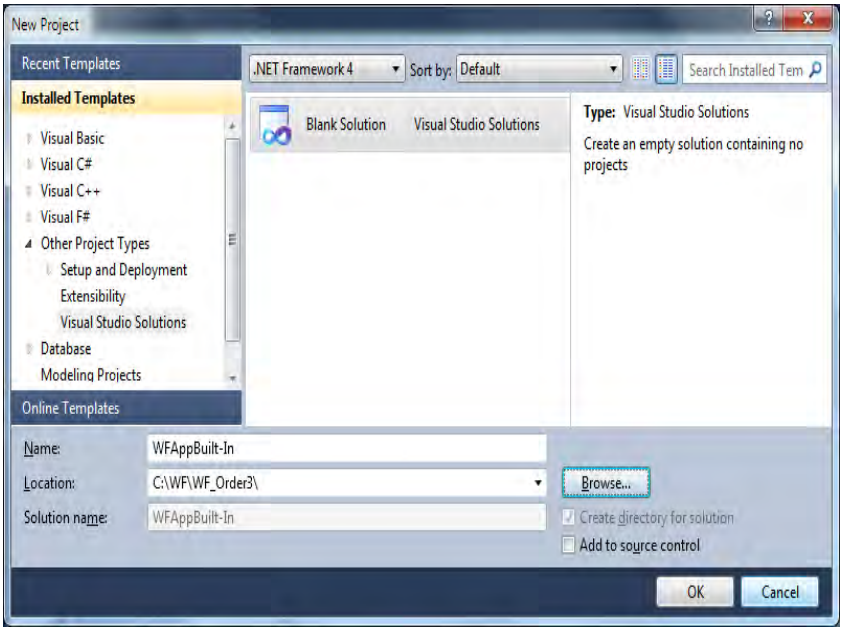
2.5. Built-In ქმედების გაფართოება

(ლაბ.N10)

მიზანი: წინა ლაბორატორიის პროექტისთვის შეკვეთის ფასწარმოქმნის წესების დაზუსტება. გაფართოების ორი ხერხის შესწავლა built-in ქმედებისთვის: მომხმარებლის ქმედებების შექმნა და InvokeMethod ქმედების გამოყენება.

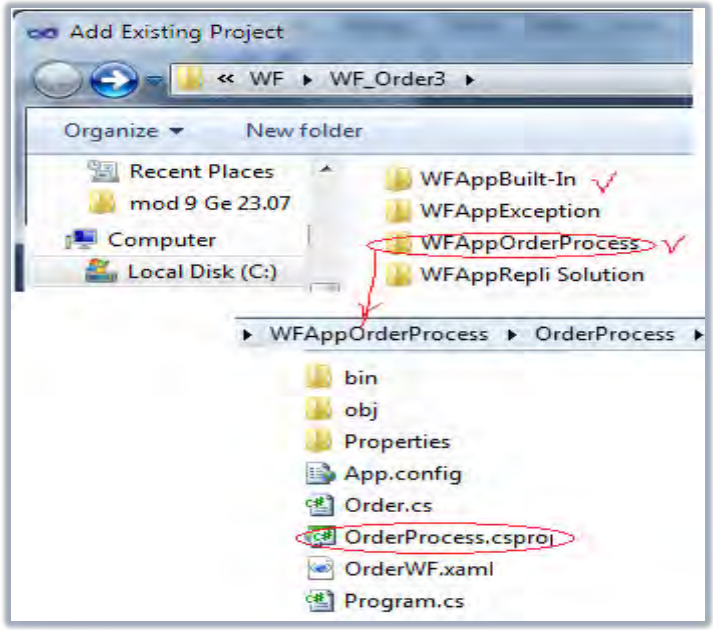
1. პრაქტიკული ნაწილი

გადმოვაკოპიროთ წინა პროექტი WFAppBuilt-In სახელით:



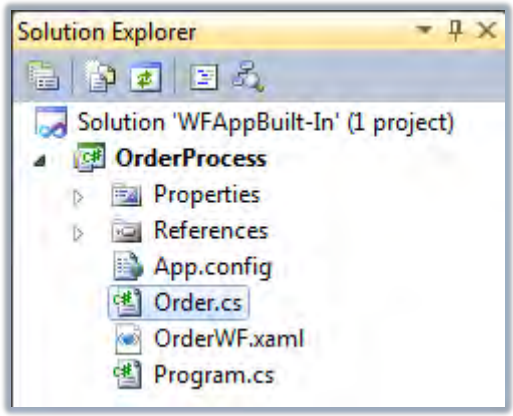
ნახ.2.35

WFAppOrderProcess–დან OrderProcess–ში ავირჩიოთ ფაილი OrderProcess.csproj ;



ნახ.2.36

მიიღება Solution Explorer-ის საწყისი შემადგენლობა:



ნახ.2.37

2. მომხმარებლის ქმედებათა გამოყენება (Custom Activities)

ჩვენი პროექტი იყენებს ფიქსირებულ ფასს ($ 10) ყველა ელემენტისთვის. შეიძლება ფასის დაზუსტება მომხმარებლის ქმედების შექმნით, რომელიც მოიძიებს ყველა ელემენტის ფასს ItemCode თვისებით. გავხსნათ Order.cs ფაილი და ჩავამატოთ შემდეგი კლასის განსაზღვრება:

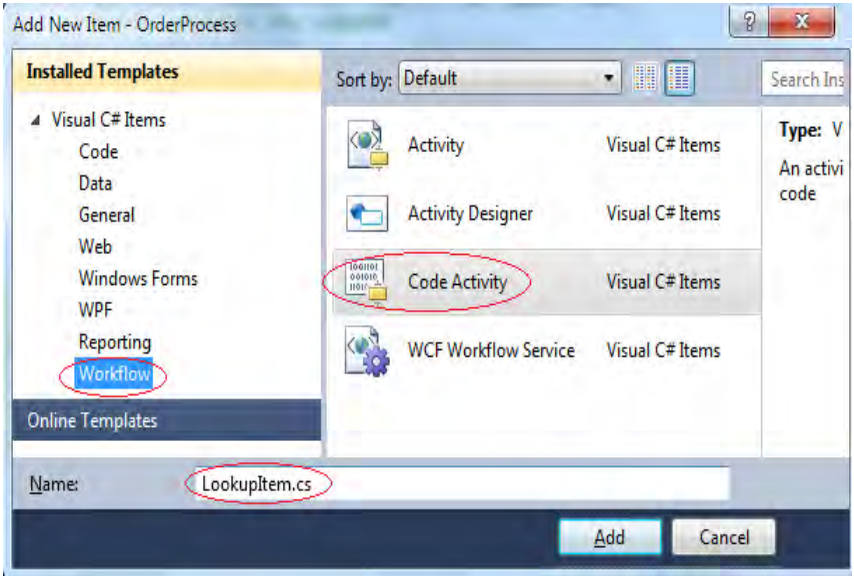
```
//-------------------------------------------
// სტრუქტურის განსაზღვრა, რომელიც ბრუნდება
// LookupItem მომხმარებლის ქმედებით
//-------------------------------------------
public class ItemInfo
{
  public string ItemCode { get; set; }
  public string Description { get; set; }
  public decimal Price { get; set; }
}
```

შენიშვნა:

WF 4.0–დან CodeActivity არის უკვე აბსტრაქტული კლასი, რომელიც გამოიყენება როგორც საბაზო კლასი მრავალი ჩაშენებული built-in ქმედებისთვის (და გამოყენებადი როგორც საბაზო მომხმარებლის ქმედებებისთვის). მაგრამ მისი უშუალო გამოყენება სამუშაო პროცესში არ შეიძლება. WF 4.0–ში მოგვიხდება არაერთი ასეთი, მომხმარებლის ქმედების დაწერა. საბედნიეროთ, ეს მარტივია, რასაც ქვემოთ ვნახავთ.

3. მომხმარებლის ქმედების რეალიზაცია (Implementing a Custom Activity)

Solution Explorer-ში OrderProcess პროექტზე მარჯვენა ღილაკით ავირჩიოთ Add->NewItem. შემდეგ Workflow კატეგორიაში ავირჩით Code Activity შაბლონი. შევიტანოთ კლასის სახელი LookupItem.cs, როგორც ნახაზზეა ნაჩვენები:



ნახ.2.38

ამ კლასის იმპლემენტაცია ნაჩვენებია 2.10 ლისტინგში.

// – – – ლისტინგი 2.10 – – –

```
using System;
using System.Activities;
namespace OrderProcess
{
  public sealed class LookupItem : CodeActivity
```

```
    {
     public InArgument<string> ItemCode { get; set;
}
     public OutArgument<ItemInfo> Item { get; set;
}
     protected override void
    Execute(CodeActivityContext context)
     {
      ItemInfo i = new ItemInfo();
      i.ItemCode =
            context.GetValue<string>(ItemCode);
     switch (i.ItemCode)
      {
       case "12345":
                  i.Description = "Widget";
                  i.Price = (decimal)10.0;
                  break;
       case "12346":
                  i.Description = "Gadget";
                  i.Price = (decimal)15.0;
                  break;
        case "12347":
                  i.Description = "Super Gadget";
                  i.Price = (decimal)25.0;
                  break;
      }
      context.SetValue(this.Item, i);
     }
    }
}
```

ისევე, როგორც სამუშაო პროცესებს შეიძლება ჰქონდეს შემავალი და გამომავალი არგუმენტები, ასევე ქმედებასაც

შეიძლება ისინი ჰქონდეს. ფაქტობრივად, თვისებები, რომლებიც იქნა დაყენებული built-in კლასისითვის (მაგ., Text თვისება WriteLine ქმედებისთვის), არის ქმედების არგუმენტები. თქვენი მომხმარებლის ქმედება (LookupItem) ღებულობს ItemCode–ს როგორც შემავალ არგუმენტს და აბრუნებს უკან ItemInfo კლასს, როგორც გამომავალ არგუმენტს.

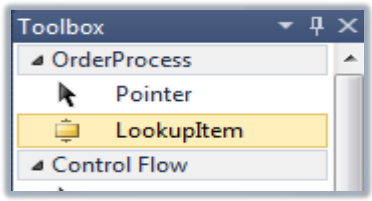
LookupItem კლასი არის წარმოებული CodeActivity საბაზო კლასიდან და უცვლის განსაზღვრებას (გადატვირთავს, overrides) Execute მეთოდს. ეს მეთოდი ქმნის ItemInfo კლასს და ინახავს შიგ ItemCode–ს. საყურადღებოა, რომ იგი უნდა იყენებდეს context.GetValue() მეთოდს, რათა მიიღოს ItemCode მნიშვნელობა, ვინაიდან არგუმენტის მონაცემების მხარდაჭერა ხდება უშუალოდ თვით სამუშაო პროცესის მიერ. CodeActivityContext–ს მიწოდება Execute() მეთოდი, როცა იგი გამოიძახება.

Execute() მეთოდი აკონკრეტებს აღწერის და ფასის თვისებებს ItemCode–ს ბაზაზე (რეალურ აპლიკაციაში ის უნდა ვეძებოთ მონაცემთა ბაზაზში). და ბოლოს, იგი გამოიძახებს context.SetValue() მეთოდს ItemInfo კლასის შესანახად გამომავალ არგუმენტში.

F5–ით ავამუშავოთ აპლიკაცია.

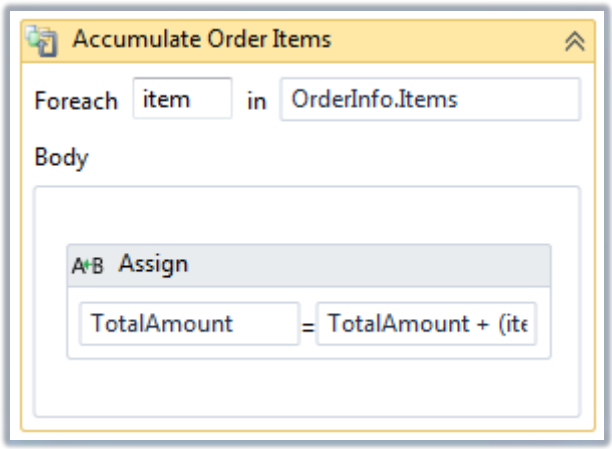
4. LookupItem ქმედების გამოყენება

გავხსნათ OrderWF.xaml ფაილი დიზაინის რეჟიმში. საყურადღებოა, რომ LookupItem ქმედება დამატებული იყო Toolbox–დან:



ნახ.2.39

workflow სქემაზე გავშალოთ "Accumulate Order Items" ქმედება:



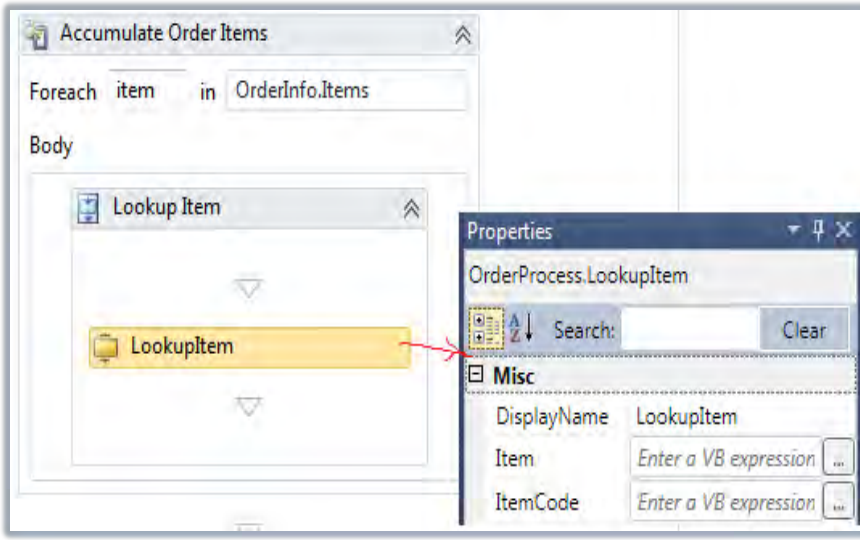
ნახ.2.40

ის შეკვეთის ელემენტებს მარტივად ჩამოთვლის და ასრულებს Assign ქმედებას, რომელიც ამატებს OrderTotal–ში ყოველი ელემენტისთვის $10–ს. ავირჩიოთ Assign აქტიურობა და Delete–თი წავშალოთ იგი სქემიდან. მის ადგილას გადმოვიტანოთ Sequence ქმედება უშუალოდ Body სექციაში. DisplayName

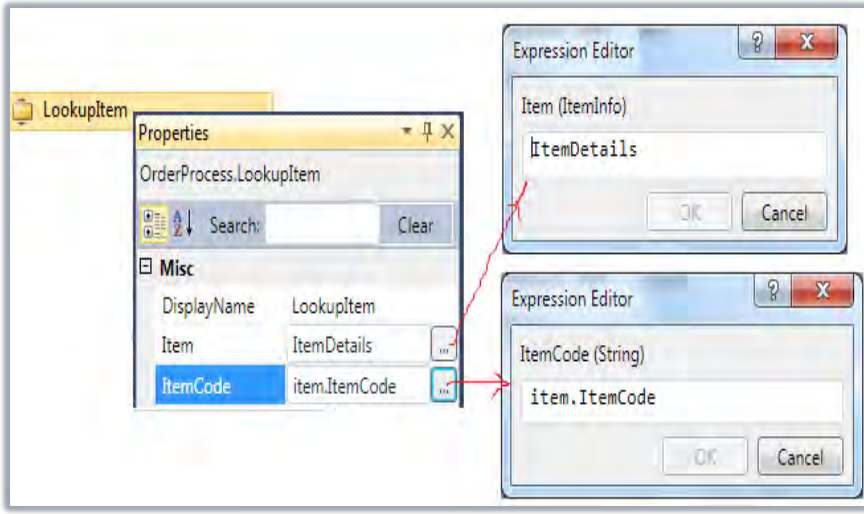
თვისებაში ჩავწეროთ Lookup Item და გავშალოთ. გადმოვიტანოთ Lookup Item ქმედება ამ Sequence–ზე (ნახ.2.41).

საყურადღებოა, რომ თვისებათა ფანჯარა შეიცავს არგუმენტების ItemCode–ს და Item–ს, რომლებიც ჩვენ ამ ქმედებისთვის განვსაზღვრეთ. არგუმენტის DisplayName იყო განსაზღვრული CodeActivity საბაზო კლასში.

ItemCode თვისებისთვის შევიტანოთ გამოსახულება: item.ItemCode (ნახ.2.42).

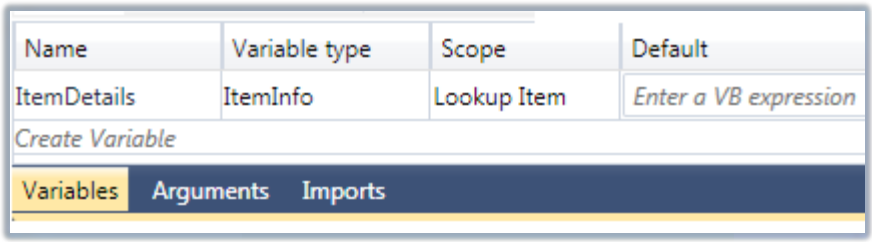


ნახ.2.41

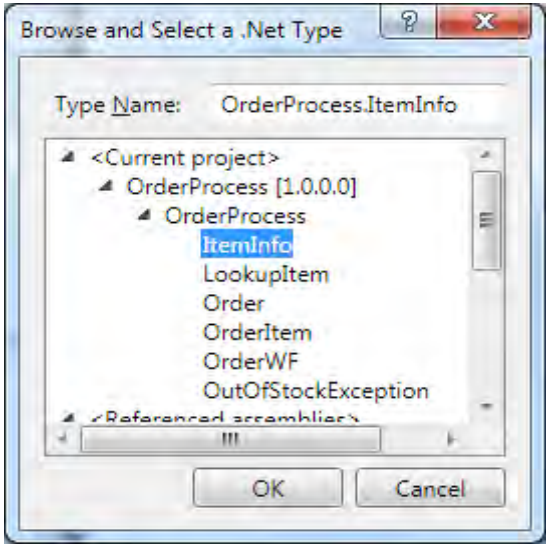


ნახ.2.42

უნდა განვსაზღვროთ ცვლადი (Variables) ItemInfo კლასის შესანახად, რომელიც აბრუნებს უკან მნიშვნელობას. დიზაინერის ქვედა მარცხენა კუთხეში ჩავრთოთ Variables და შევიტანოთ სახელი ItemDetails (ნახ.2.43). ტიპის ასარჩევად გამოვიყენოთ Browse და ავირჩიოთ ItemInfo კლასი (ნახ.2.44). განსაზღვრის არე (Scope) მიმდინარე Sequence ქმედებისთვის უნდა იყოს (“Lookup Item”).



ნახ.2.43

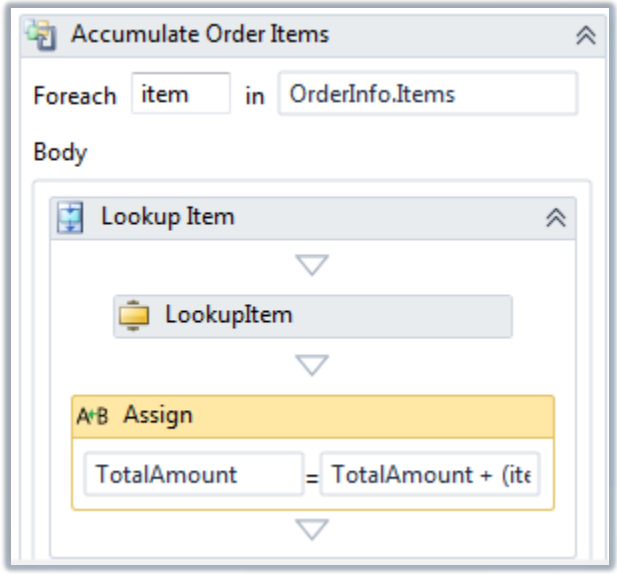


ნახ.2.44

ახლა ავირჩიოთ LookupItem ქმედება და მივუთითოთ Item თვისებაში ItemDetails. იგი აიღებს ItemInfo კლასისთვის მომხმარებლის ქმედებით დაბრუნებულ მნიშვნელობას და შეინახავს ItemDetails ცვლადში. გადმოვიტანოთ Assign ქმედება LookupItem–ის ქვემოთ. To თვისებისთვის შევიტანოთ TotalAmount, ხოლო Value თვისებისთვის გამოსახულება:

TotalAmount + (item.Quantity * ItemDetails.Price).

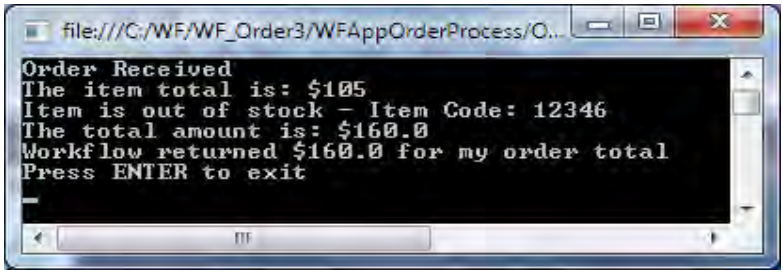
მივიღებთ 2.45 ნახაზზე ნაჩვენე სურათს.



ნახ.2.45

- აპლიკაციის ამუშავება

F5-ით ავამუშავოთ აპლიკაცია და მივიღებთ შედეგებს (ნახ.2.46).



ნახ.2.46

შდეგების სისწორე ($105) შეიძლება ხელით გადაანგარიშებით გადავამოწმოთ (ნახ.2.47).

ItemCode	Quantity	Price	Ext. Price
12345	1	$10	$10
12346	3	$15	$45
12347	2	$25	$50
Total			$105

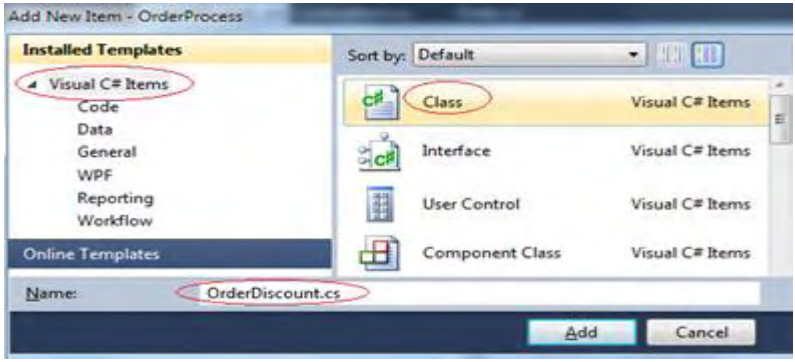
ნახ.2.47

5. InvokeMethod ქმედება

აქტიურობა InvokeMethod არის კიდევ ერთი სასარგებლო ხერხი კოდის რეალიზაციისათვის სტანდარტული built-in ქმედების გარეთ. ამ ქმედების გამოყენება შესაძლებელია კლასის მეთოდის გამოსაძახებლად. კლასი არ უნდა იყოს სამუშაო პროცესის ნაწილი ან იყენებდეს სამუშაო პროცესის საბაზო კლასებს.

ამ პროექტში რეალიზებულია კლასი, რომელიც ანგარიშობს ფასდაკლების ზომას სხვადასხვა წესების საფუძველზე. ეს კლასი გამოიძახება (invoked) მუშა პროცესის მიერ ფასდაკლების საანგარიშოდ მითითებული შეკვეთისთვის.

- ფასდაკლების კლსის შექმნა (Discount Class)



ნახ.2.48

OrderDiscount.cs ფაილის ლისტინგი მოცემულია ქვემოთ:

```
// – – – ლისტინგი 2.11 – – –
using System;
using System.Collections.Generic;

namespace OrderProcess
{
  public static class OrderDiscount
   {
     public static decimal ComputeDiscount(Order o, decimal
total)
      {
       // შეკვეთილი ელემენტების რაოდენობის ანგარიში
       int count = 0;
       foreach (OrderItem i in o.Items)
         {
           count += i.Quantity;
         }
       // ფასდაკლების პროცენტის განსაზღვრა
       decimal pct = 0;
       if (total > 500)
           pct = (decimal)0.20;
       if (total > 200)
```

```
            pct = (decimal)0.15;
        if (total > 100)
            pct = (decimal)0.10;
        // ფასდაკლების ჯამის ანგარიში
        decimal discount = total * pct;
        // დოლარის დაკლება ყოველ შეკვეთილ ელემენტზე
        discount -= (decimal)count;
        // შემოწმება, რომ ის არაა ნულზე ნაკლები
        if (discount < 0)
            discount = 0;
        Console.WriteLine("გაანგარიშებული ფასდაკლება: ${0}",
discount.ToString());
        return discount;
       }
   }
}
```

ComputeDiscount() მეთოდი ღებულობს ორ პარამეტრს: Order კლასი და item total (ელემენტების ჯამი). იგი აბრუნებს უკან ფასდაკლების რაოდენობას, რომელიც გამოიყენება მოცემული შეკვეთისთვის. ფასდაკლების ლოგიკა აქ გამოყენებულია რაიმე წესების გარეშე, თავისუფლად. ალგორითმულად ჯერ განისაზღვრება ფასდაკლების პროცენტი შეკვეთის საერთო ჯამიდან. შემდეგ იგი აკლებს 1 $–ს ყოველი ელემენტისათვის.

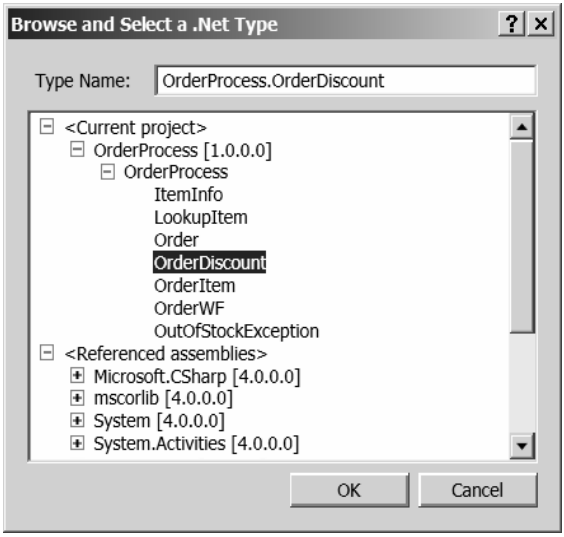
F6-ით დავაკომპილიროთ პროგრამა.

- **InvokeMethod ქმედების გამოყენება**

ჩვენ პროექტში გამოვიყენოთ InvokeMethod ქმედება ComputeDiscount() მეთოდის შესასრულებლად. გადმოვიტანოთ InvokeMethod აქტიურობა სამუშაო პროცესის სქემაზე, უშუალოდ “Check Stock” ქმედების წინ. DisplayName თვისებაში შევიტანოთ Calculate Discount.

- მიზნობრივი ობიექტის განსაზღვრა (Specifying the Target Object)

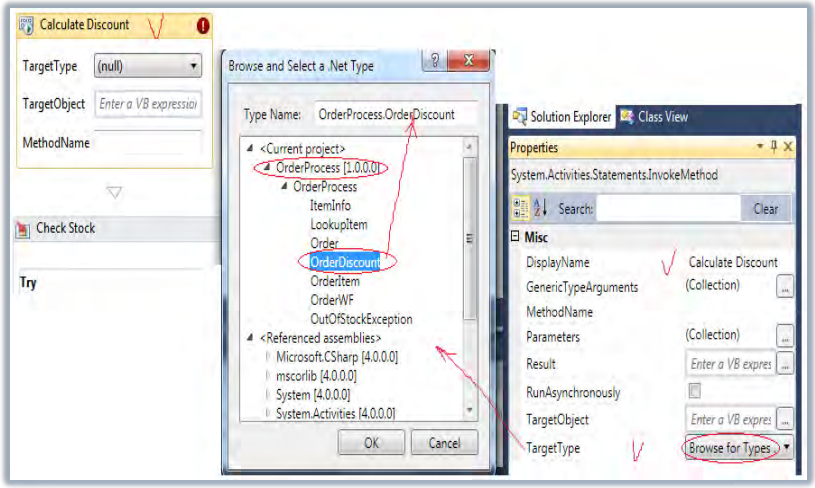
TargetType თვისება განსაზღვრავს კლასს, რომელიც შეიცავს გამოსაძახებელ მეთოდს. ჩამოშლად სიაში ტიპისათვის ავირჩიოთ Browse. დიალოგურ ფანჯარაში გავხსნათ OrderProcess ნაკრები და ავირჩიოთ OrderDiscount კლასი (ნახ.2.49).



ნახ.2.49. OrderDiscount კლასის არჩევა

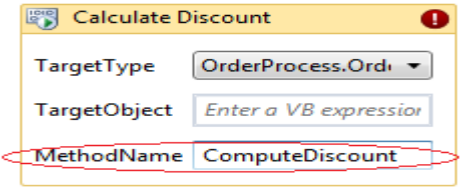
<u>რჩევა:</u> შეიძლება ორი მეთოდის გამოყენება, იმის მისათითებლად, რომ ობიექტი შეიცავს მეთოდს, რომელიც იქნება გამოძახებული.

1. თუ მეთოდი სტატიკურ კლასშია, მაშინ შეიძლება მიეთითოს მხოლოდ TargetType თვისება, როგორც ჩვენ მაგალითში გავაკეთეთ (ნახ.2.50);



ნახ.2.50

2. თუ კლასი არაა სტატიკური, მაშინ უნდა განისაზღვროს ამ ობიექტის კონკრეტული ეგზემპლარი. ამისთვის უკეთესი ხერხია კლასის ტიპის ცვლადის განსაზღვრა. მაშინ TargetObject თვისებაში უნდა მიეთითოს ცვლადის სახელი. თუ არსებობს რამდენიმე ეგზემპლარი და საჭიროა კონტროლი, თუ რომელია გამოყენებაში, მაშინ შეიძლება ცვლადის დაყენება ან Assign ქმედებთ ან მომხმარებლის ქმედებით (custom activity) მანამ, სანამ შესრულდება InvokeMethod ქმედება. თუ მიეთითება TargetObject თვისება, მაშინ აღარაა საჭირო TargetType თვისების მითითება. MethodName თვისებისთვის შევტანოთ ComputeDiscoun (ნახ.2.51).



ნახ.2.51

III თავი

კორპორაციის პროგრამული სისტემის დაპროექტება UML/2 და Workflow ტექნოლოგიებით

3.1. ბიზნესპროცესების მოდელირება UML2 ტექნოლოგიის Enterprise Architect ინსტრუმენტით (ლაბ.N11)

მიზანი: საინფორმაციო სისტემების დაპროექტებისა და მოდელირების საკითხების გაცნობა UML და Workflow ტექნოლოგიების ერთობლივი გამოყენებით.

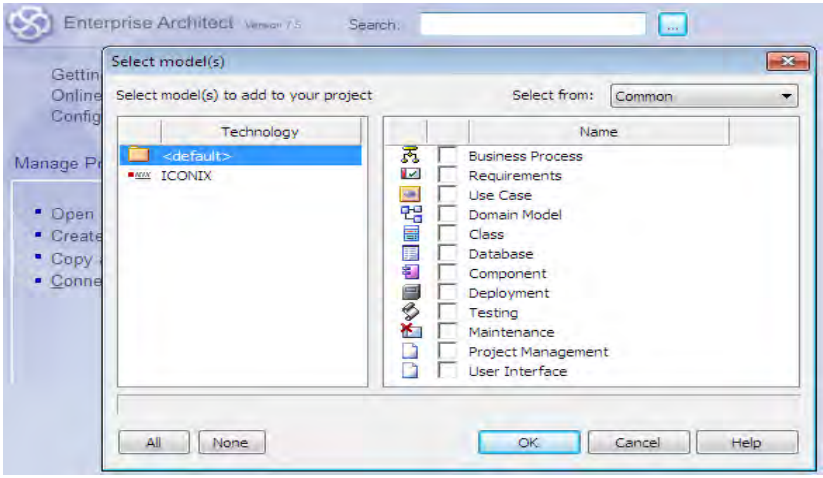
1. თეორიული ნაწილი

განხილულია საინფორმაციო სისტემების დაპროექტებისა და დამუშავების თანამედროვე ტექნოლოგია (CASE – Computer Aided System Engineering), რომელსაც საფუძვლად უდევს უნიფიცირებული მოდელირების ენის სხვადასხვა ვერსიები. ჩვენ შემთხვევაში გამოყენებულია ახალი UML2 ვერსია [8]. ამ ტექნოლოგიაზე დაყრდნობით დამუშავებულია ბიზნეს-პროექტების მართვის სისტემის ტექნოლოგიური პროცესის მოდელი (ინსტრუმენტული საშუალებების გამოყენებით საპრობლემო არის ვიზუალური მოდელირება და ამ მოდელის ანალიზი სისტემის დამუშავების ყველა ეტაპზე).

მართვის ავტომატიზებული სისტემების სრულყოფილი, საიმედო და მოქნილი პროგრამული უზრუნველყოფის (Software Engineering) სწრაფად დაპროექტება, რეალიზაცია, დანერგვა და შემდგომი თანხლება სისტემის დამკვეთ ორგანიზაციაში მეტად მნიშვნელოვანი ამოცანაა. მისი ეფექტურად გადაწყვეტა ბევრადაა დამოკიდებული როგორც საპროექტო-დეველოპმენტის გუნდის

შემადგენლობა გამოცდილებაზე, ასევე IT-ინფრასტრუქტურასა და CASE-ინსტრუმენტებზე [10].

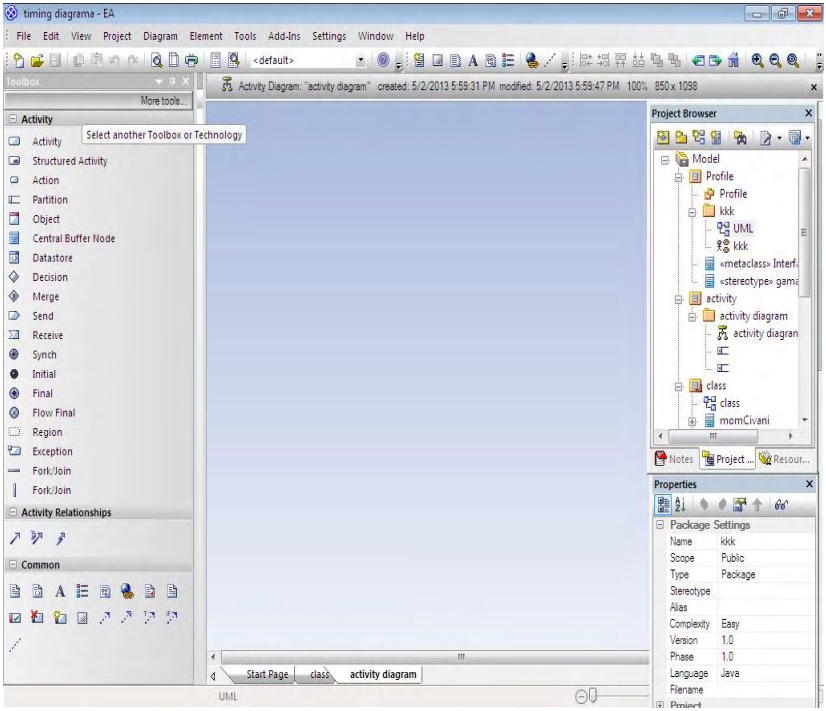
Enterprise Architect სისტემაში ახალი პროექტის შექმნისას, სასტარტო გვედზე იხსნება მოდელების არჩევის ფანჯარა (ნახ. 3.1), რის მიხედვითაც შემდგომში იგება შესაბამისი მოდელები. გამოყენებაშია როგორც UML1 და UML2 ინსტრუმენტების ერთობლიობა, ისე ბიზნესპროცესების მოდელირების თანამედროვე სტანდარტები.



ნახ.3.1.

ფანჯარაში “New diagram”, ველში “Name” იწერება “Project Browser” პანელში (ნახ.3.2) მონიშნული მოდელის სახელი, დიალოგის მარცხენა ნაწილიდან “Select Form” ხდება მოდელის საჭირო ფორმის არჩევა, რომლის გააქტიურებით მარცხენა ნაწილში “Diagram types” იხსენება მოდელის შესაბამისი დიაგრამის ტიპები (ნახ. 3.1). ღილაკით “OK” არჩეული დიაგრამა ჯდება მითითებულ მოდელში (“project browser” პანელში) ხოლო,

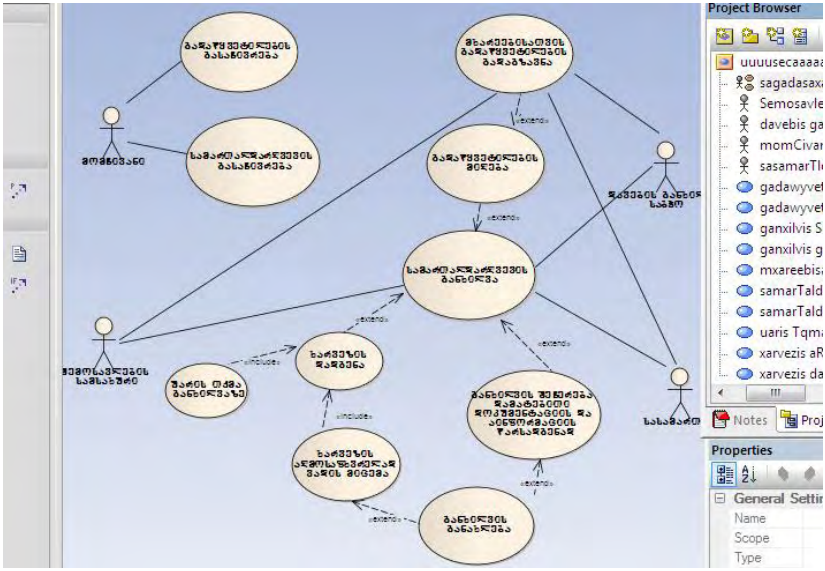
ინსტრუმენტების პანელში “Toolbox”, ჩნდება დიაგრამის შესაბამისი ელემენტები.



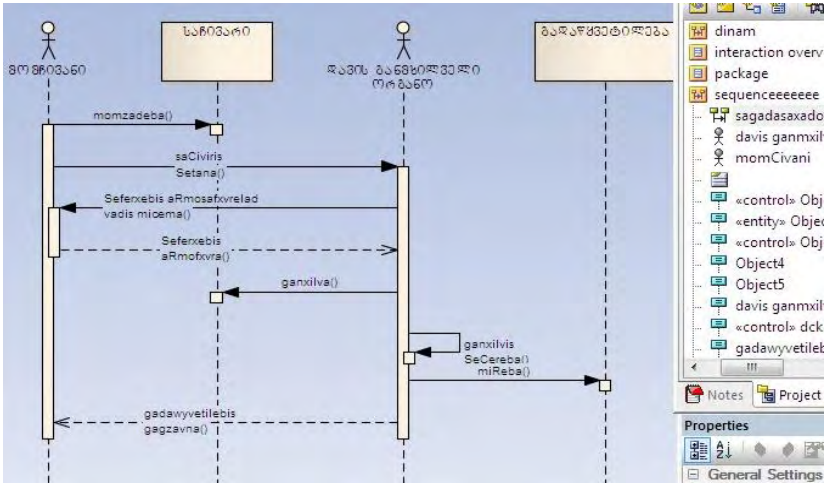
ნახ.3.2. Enterprise Architect სისტემის სასტარტო გვერდი

2. პრაქტიკული ნაწილი

Enterprise Architect პროგრამული პაკეტის დახმარებით ავაგეთ ის დიაგრამები (UseCase, Sequence), რომლებიც შემდგომში სისტემის (საგადასახადო დავის წარმოების) ფრაგმენტის ინფორმაციული უზრუნველყოფის შექმნის წინმსწრებ ეტაპზე დაგვჭირდება (ნახ.3.3-3.4).

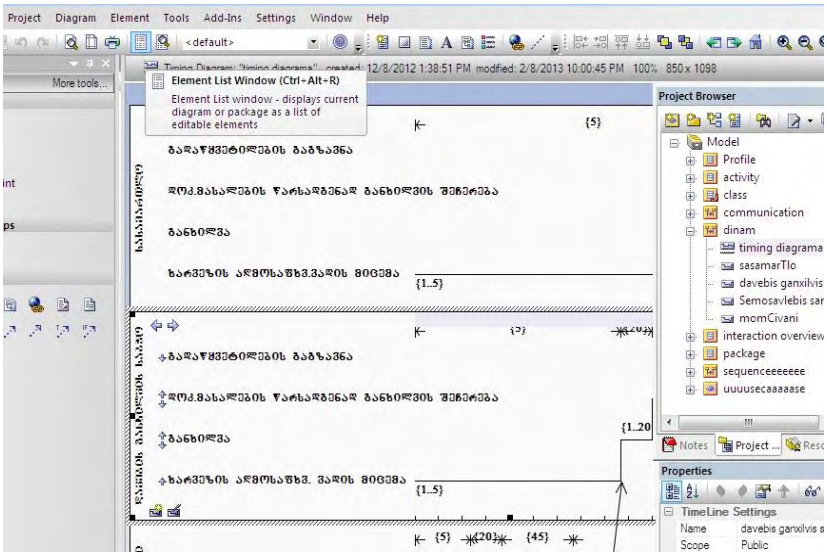


ნახ.3.3. Use Case - დიაგრამა



ნახ.3.4. Sequence – დიაგრამა

სინქრონიზაციის დიაგრამა მიმდევრობითობის დიაგრამის ალტერნატიული ვარიანტია. დიაგრამა განსაზღვრავს სხვადასხვა მდგომარეობათა ცვლილებებსა და ობიექტების ქცევას დროითი შკალის საზღვრებში. შესაძლებელია გამოყენებულ იქნას, დროის მიხედვით მართვადი ბიზნესპროცესებისათვის. მაგალითად საგადასახადო დავის წარმოების პროცესების დროში ასაღწერად (ნახ.3.5).



ნახ.3.5. სინქრონიზაციის დიაგრამა

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

ამგვარად, UML-ის საფუძველზე განისაზღვრა ასაგები სისტემის მოთხოვნილებები და მომხმარებლის მუშაობის სცენარი.

3.2. საპრობლემო სრეფოს სისტემის დაპროექტება WF ტექნოლოგიით .NET პლატფორმაზე (ლაბ.N11)

მიზანი: საპრობლემო სფეროს ბიზნესპროცესების აქტიურობათა დიაგრამის მოდელირების ხერხის გაცნობა UML-ენის ბაზაზე და მისი შემდგომი პროგრამული რეალიზაციისთვის ახალი ტექნოლოგიების საფუძველზე.

1. თეორიული ნაწილი

შემოთავაზებულია Workflow Foundation ტექნოლოგიის გამოყენება ბიზნესპროცესების და ბიზნესწესების ვიზუალური დაპროგრამებისთვის. საპრობლემო სფეროს კონკრეტული მაგალითი განიხილება საბაჟო სამართალდარღვევის ბიზნეს-პროცესებისთვის [8].

შემოსავლების სამსახურის აუდიტის და საბაჟო დეპარტამენტების ფუნქციებიდან ერთ-ერთი მნიშვნელოვანი საკითხია საგადასახადო სამართალდარღვევების ბიზნესპროცესების მართვა, კერძოდ, მათი გამოვლენა და საგადასახადო დავის წარმოება [10]. იმისდა მიხედვით თუ მომჩივანის მიერ საჩივარი დავის განმხილველ რომელ ორგანოში შეიტანება, საგადასახადო სამართალდარღვევის ოქმი და თანდართული მასალები შეიძლება შემოსავლების სამსახურის გარდა ფინანსთა სამინისტროს დავების განხილვის საბჭომ ან სასამართლომ განიხილოს [8].

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

- მოქალაქეს შეაქვს საჩივარი დავის განმხილველ ორგანოში (ათვლის თარიღი ფიქსირდება);
- თუ საჩივარი არ აკმაყოფილებს პროცედურულ მოთხოვ-

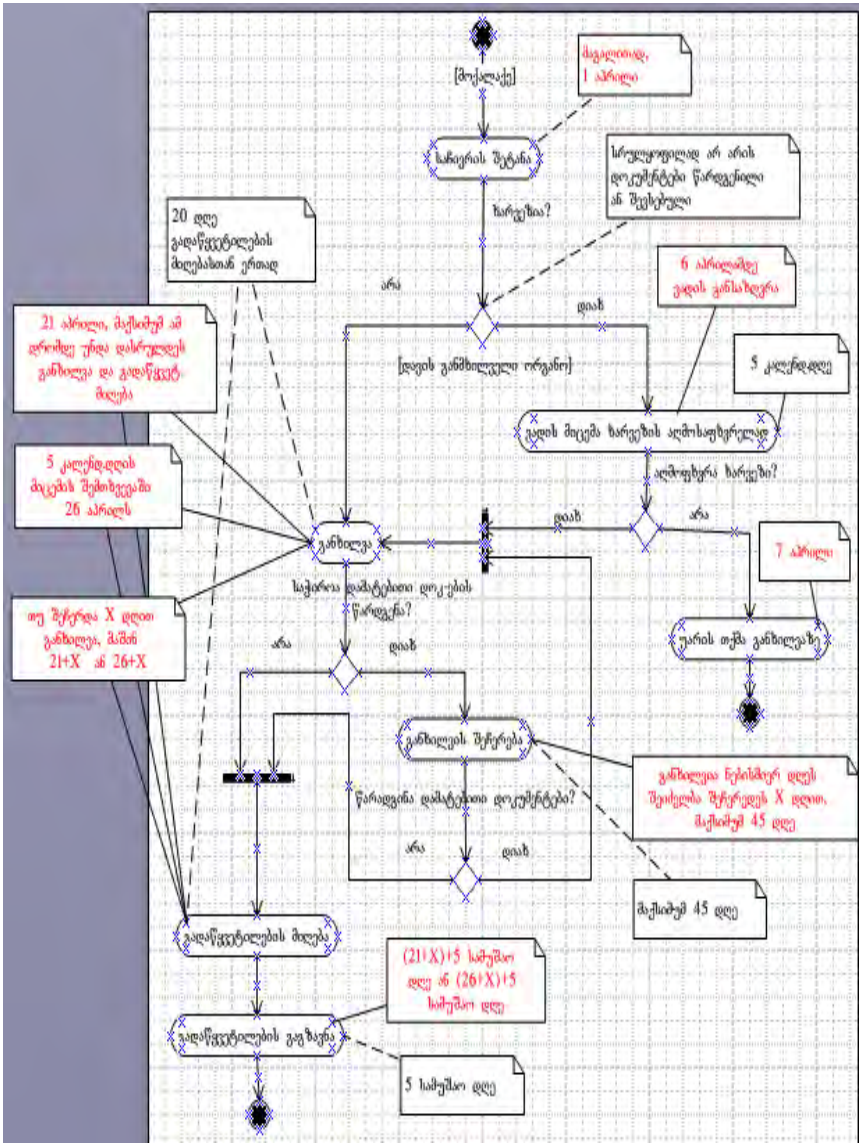
ნებს, მომჩივანს წერილობით ეცნობება ამის შესახებ და მიეცემა არანაკლებ 5 დღე არსებული ხარვეზის გამოსაწორებლად;

- დავის განმხილველ ორგანოს უფლება აქვს მომჩივანის მოტივირებული მოთხოვნის შემთხვევაში გააგრძელოს ხარვეზის გამოსასწორებლად მიცემული ვადა;
- დავის განმხილველი ორგანო საჩივარს განიხილავს 20 დღის ვადაში;
- დამატებითი ინფორმაციის ან/და დოკუმენტაციის მოპოვების საფუძვლით საცივრის განხილვის შეჩერების საერთო ხანგრძლივობა არ უნდა აღემატებოდეს 45 დღეს;
- დავის განმხილველი ორგანოს გადაწყვეტილების დამოწმებული ასლი, მისი მიღებიდან 5 სამუშაო დღის ვადაში ეგზავნება მხარეებს.

ნაშრომში განიხილება აღნიშნული ამოცანების მოდელირების, დაპროექტების და პროგრამული რეალიზაციის საკითხები უნიფიცირებული მოდელირების ენის (UML) და ახალი Workflow Foundation ტექნოლოგიის გამოყენებით [12,13].

Workflow Foundation ტექნოლოგია .NET Framework 4.0/4.5 - ში არის სრულიად ახალი პარადიგმა სამუშაო პროცესებზე (workflow) ბაზირებული აპლიკაციების ასაგებად. იგი ფუნდამენტურად ახლად გააზრებული ტექნოლოგიაა. აქ ჩვენ განვიხილავთ კონკრეტული ამოცანის, კერძოდ საბაჟო სამართალდარღვევების სამუშაო პროცესის აქტიურობათა დიაგრამიდან პროგრამული კოდის დაპროექტების პროცესს.

3.6 ნახაზზე ნაჩვენებია სამართალდარღვევის სადავო საკითხების ბიზნეს-პროსეცების და ბიზნეს-წესების აქტიურობათა დიაგრამა, აგებული MsVisio ინსტრუმენტის გამოყენებით [13]. სქემაზე გარდა ბიზნესპროცესების და ცალკეულ ქმედებათა შესრულების მიმდევრობის აღწერისა, ნაჩვენებია ბიზნეს-წესების კომენტარებიც, რომლებშიც კარგად ჩანს საქმის წარმოების კანონით არსებული ვადების დროითი რეგლამენტი.


[მოქალაქე]
მაგალითად, 1 აპრილი
საბერის შეტანა
სრულყოფილად არ არის დოკუმენტები წარდგენილი ან შევსებული
ხარვეზია?
20 დღე გადაწყვეტილების მიღებასთან ერთად
არა
დიახ
6 აპრილამდე ვადის განსაზღვრა
21 აპრილი, მაშინაც ამ დროიდე უნდა დასრულდეს განხილვა და გადაწყვეტ მიღება
[დადეს განმხილველი ორგანო]
5 კალენდ.დღე
ვადის მიცემა ხარვეზის აღმოსაფხვრელად
5 კალენდ.დღის მიცემის შემთხვევაში 26 აპრილს
აღმოფხვრა ხარვეზი?
დიახ
არა
7 აპრილი
განხილვა
საჭიროა დამატებითი დოკ-ების წარდგენა?
უარის თქმა განხილვაზე
თუ შეჩერდა X დღით განხილვა, მაშინ 21+X ან 26+X
არა
დიახ
განხილვის შეჩერება
განხილვა ნებისმიერ დღეს შეიძლება შეჩერდეს X დღით, მაგრამ 45 დღე
წარადგინა დამატებითი დოკუმენტები?
არა
დიახ
მაგრამ 45 დღე
გადაწყვეტილების მიღება
(21+X)+5 სამუშაო დღე ან (26+X)+5 სამუშაო დღე
გადაწყვეტილების გაგზავნა
5 სამუშაო დღე

ნახ.3.6

ახლა განვიხილოთ აგებული აქტიურობის დიაგრამის ასახვა Workflow Foundation ტექნოლოგიის სამუშაო გარემოში [5]. ესაა ჰიბრიდული (Windows+Web) სისტემა, რომლის კონსტრუირება (დიზაინი) ხდება შესაბამისი Workflow – ინსტრუმენტის ვიზაულური ელემენტებით (XAML ენაზე), ხოლო პროგრამის მუშაობის ლოგიკა, ჩვენს შემთხვევაში C#.NET კოდით.

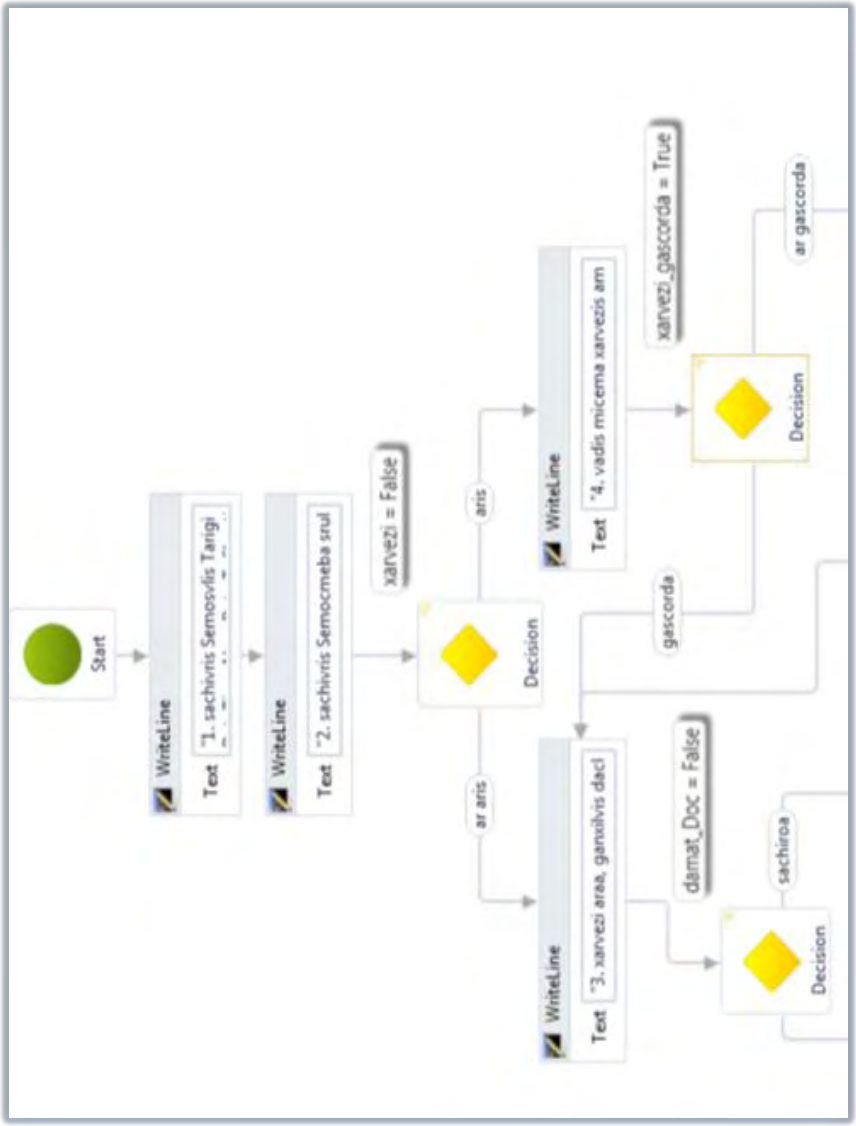
3.7 ნახაზზე ნაჩვენებია დასმული ამოცანის გადაწყვეტის ალგორითმის ერთი ვარიანტი Workflow-ინსტრუმენტით. სქემაზე განთავსებულია შემდეგი ქმედებები (აქტიურობები): Flowchart, FlowDecission, If, Switch, Sequence, Assign და სხვ. [13]

ბიზნესპროცესის დიაგრამა (Flowchart Workflow) გამოიყენებს აქტიურობათა დიაგრამას. ქმედებათა ეს დიაგრამა მუშაობს როგორც ბლოკ–სქემა, ქმედებათა სახეები დაკავშირებულია ერთმანეთთან გადაწყვეტილების ხეებით (decision trees). ქმედებათა მიმდევრობითობის გამოყენებით, შვილი–პროცესები სრულდება მიმდევრობით ზემოდან-ქვევით (top-down). ამისდა მიუხედავად, შვილი-ქმედებები შეიძლება შესრულდეს ნებისმიერი მიმდევრობით, გადაწყვეტილების მიმღები პირის მიერ. ძირითადი განსხვავება Flowchart ქმედებასა და Sequence ქმედებას შორის ისაა, თუ როგორაა დაკავშირებული შვილი-აქტიურობები. ქმედებათა დამატებისას მიმდევრობითობაში ისინი სრულდება დაღმავალი (top-down) რიგითობით.

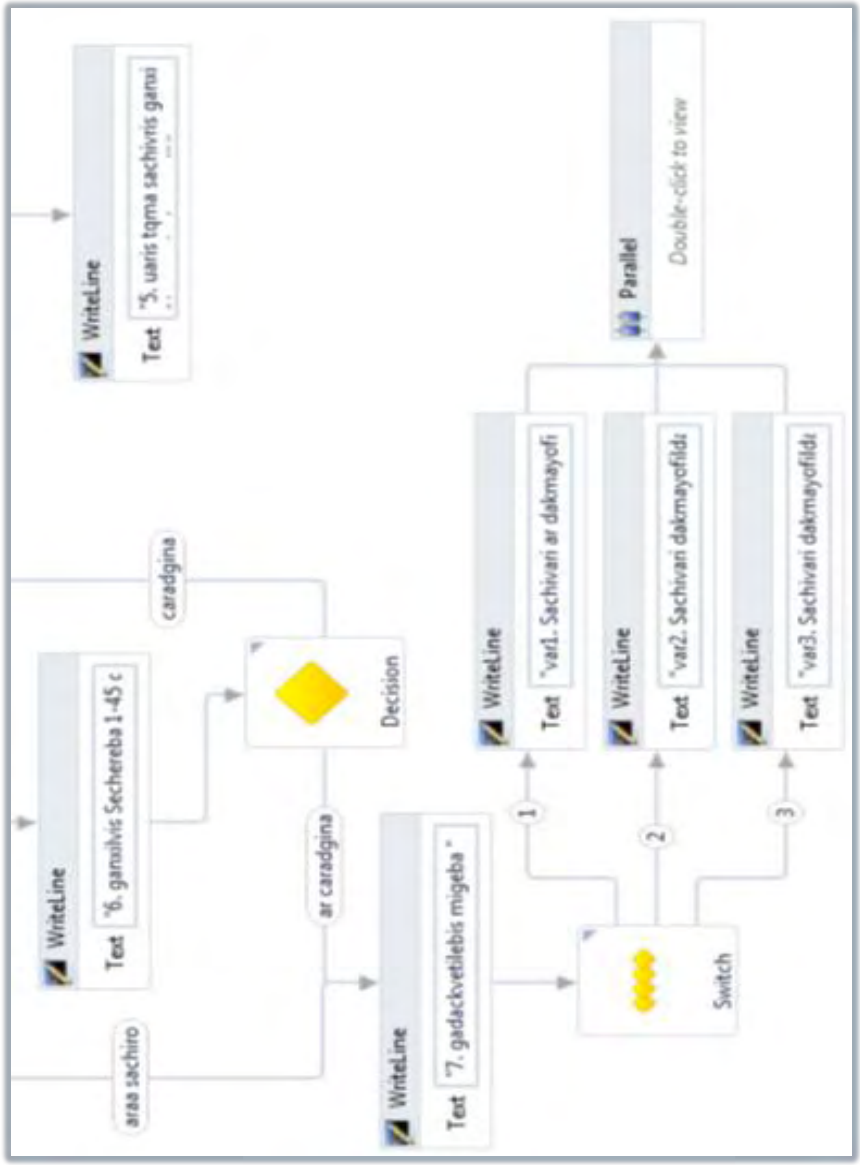
აქ შესაძლებელია მიმდევრობის კონტროლი (მართვა),

ქმედებათა გადაადგილებით, ოღონდაც ისინი ყოველთვის გასწორებულია ვერტიკალში და განაწილებულია თანაბრად. ისრები ქმედებებს შორის კი აგებულია ავტომატურად. Flowchart აქტიურობით შესაძლებელია ქმედებათა განთავსება პალიტრის ნებისმიერ ადგილას. მნიშვნელოვანია ისიც, რომ აქ ისრები ხელით უნდა გაკეთდეს და შეერთება დასაშვებია უკან, წინა ქმედებასთანაც.

გადაწყვეტილების ნაკადის C ქმედება სქემაზე გამოიყურება „ყვითელი ალმასის“ სახით, როგორც განშტოების სიმბოლო ნორმალურ ბლოკ–სქემაში, Properties-ფანჯარაში შეიტანება მდგომარეობა (Condition), მაგალითად, “ხარვეზი = False”. მაუსის კურსორის მიტანით FlowDecision ქმედებაზე გამოჩნდება ეს წარწერა.



ნახ.3.7. Workflow -დიაგრამის ფრაგმენტი
Visual Studio .NET Framework 4.0 გარემოში



ნახ.3.7. გაგრძელება

3.3. საპრობლემო სფეროს სისტემის პროგრამული რეალიზაცია WF ტექნოლოგიით - /გაგრძელება/ (ლაბ.N13)

WF 4.0 აქვს რიგი პროცედურული ელემენტების: If, While, Assign, Sequence და სხვა. მაგალითად, ხარვეზების გასწორების პროცესის შედეგი განშტოვდება „გასწორდა“ ან „არ გასწორდა“:

```
if(xarvezi_gascorda = True) t2=t1+5;
        else Console.WriteLine("Sachivari ar ganixileba");
```

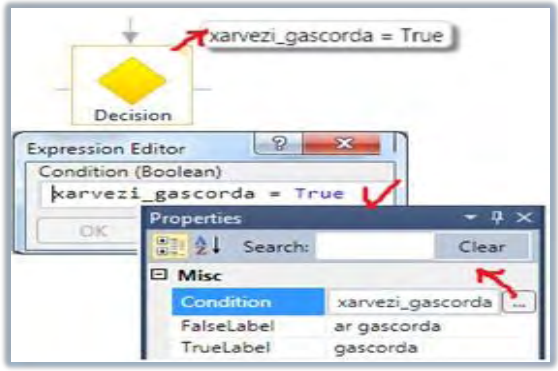
3.8 ნახაზზე ნაჩვენებია ცვლადების ცხრილი ჩვენი ალგორითმისთვის, ხოლო 3.9 ნახაზზე პირობის FlowDecision ქმედებაში მდგომარეობის (Condition) მნიშვნელობის განსაზღვრა.

Name	Variable type	Scope	Default
xarvezi	Boolean	Flowchart	False
xarvezi_gascorda	Boolean	Flowchart	True
damat_Doc	Boolean	Flowchart	False
Varianti	Int32	Flowchart	1
p1	Int32	Flowchart	0
p2	Int32	Flowchart	0
p3	Int32	Flowchart	0
p4	Int32	Flowchart	0
Create Variable			

Variables | Arguments | Imports

ნახ.3.8. Workflow -ცვლადების აღწერა

Switch ქმედება მუშაობს ისე, როგორც Switch ოპერატორი C# ენაში. ის საშუალებას იძლევა შესრულდეს sequence ქმედებები გამოსახულებათა ანალიზის საფუძველზე. Switch აქტიურობა გამოიყენება გადაწყვეტილების მიღების ვარიანტების განსაზღვრისთვისაც.



ნახ.3.9. FlowDecision -ის პირობის განსაზღვრა

Workflow სქემის აგების ამ ეტაპზე კონსოლის აპლიკაციის ამუშავებით მიიღება 3.10 ნახაზზე მოცემული შედეგები, რომელთა შემდგომი დაზუსტება და გაფართოებაა შესაძლებელი.

Workflow Foundation ტექნოლოგიის გამოყენებით ბიზნეს-პროცესების და ბიზნესწესების დაპროგრამება ხორციელდება ეფექტურად, შესაბამისი ვიზუალური კომპონენტების საფუძველზე. შედეგად საგრძნობლად მცირდება სისტემის დაპროექტების და მისი პროგრამული რეალიზაციის დრო.

ნახ. 3.10. შედეგები

3.4. მარკეტინგული პროცესების მოდელირება პეტრის ქსელებით და რეალიზაცია WF ტექნოლოგიით (ლაბ.N14)

მიზანი: მარკეტინგული პროცესების მენეჯმენტის, მისი მოდელური და პროგრამული რეალიზაციის საკითხების გაცნობა პეტრის ქსელების, Workflow, WPF და WCF ტექნოლოგიების ბაზაზე კლიენტ-სერვერ არქიტექტურით.

1. თეორიული ნაწილი

ძირითადი საწარმოო-ეკონომიკური მაჩვენებლები, როგორიცაა პროდუქციის წარმოების, რეალიზაციის, თვითღირებულების გეგმის შესრულება, რენტაბელობა, პროდუქციის ხარისხი, შრომის ნაყოფიერება, საშუალო ხელფასი და ა.შ. წარმოადგენს იმ მონაცემებს, რომლებიც მოქცეულია ხელმძღვანელობის კონტროლის ქვეშ და რომელთა განსაზღვრული მნიშვნელობების მისაღწევად წარიმართება მათი ყოველდღიური საქმიანობა. მენეჯერი თავის ფუნქციებს შეასრულებს კარგად იმ შემთხვევაში, თუ იგი სრულად ფლობს სიტუაციას, თუ გააჩნია უტყუარი ინფორმაცია საწარმოო საქმიანობის შესახებ [14].

საქმიანი პროცესების ავტომატიზაციის ტექნოლოგია (workflow) არის საწარმოს მართვის პროცესების პროგრამული მხარდაჭერა. იგი აერთიანებს რამდენიმე საინფორმაციო ტექნოლოგიას, როგორიცაა ელექტრონული ფოსტა, პროექტების მართვის სისტემა, მონაცემთა ბაზების მართვის სისტემა, ობიექტ-ორიენტირებული პროგრამირება და CASE-ტექნოლოგიები [10].

2. პრაქტიკული ნაწილი

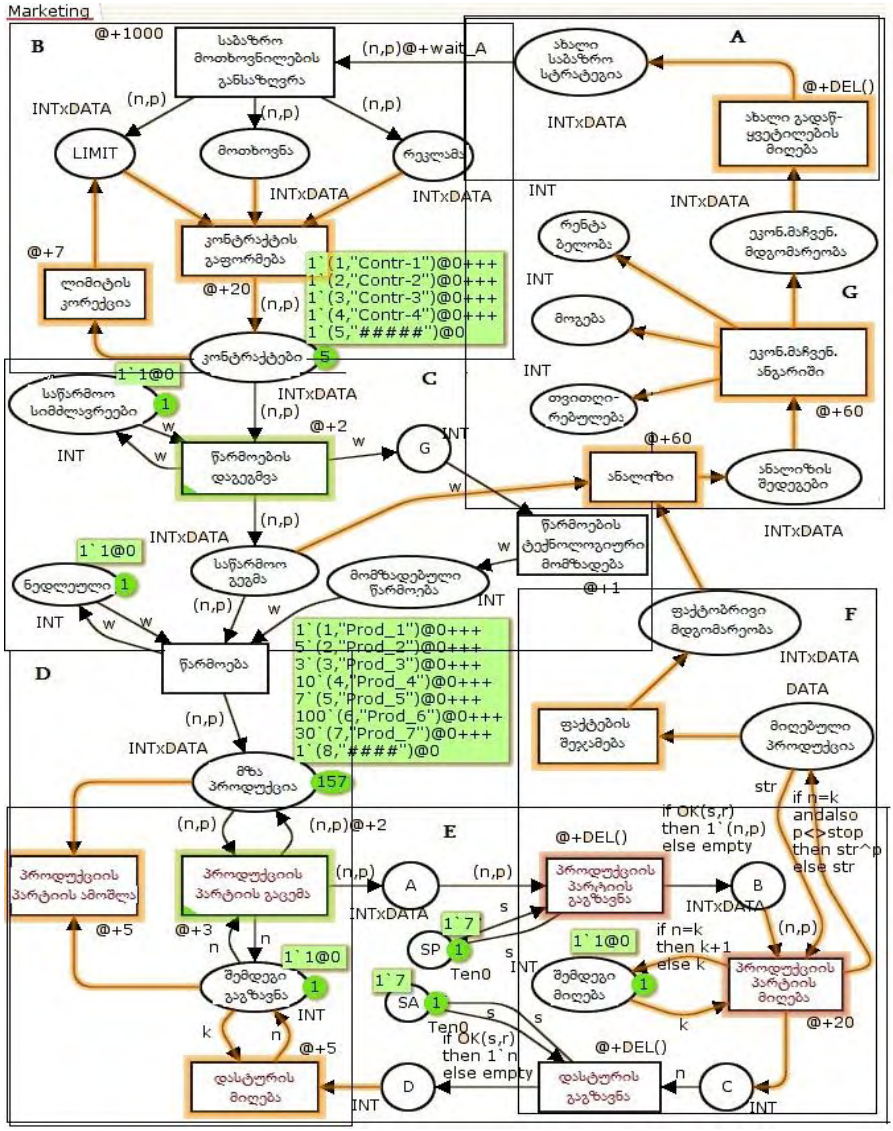
განვიხილოთ მარკეტინგული პროცესების მოდელირების საილუსტრაციო მაგალითი პეტრის ფერადი ქსელების (CPN-ინსტრუმენტის) გამოყენებით [14, 15] .

პროდუქციის საწარმოო ფირმის მარკეტინგული პროცესების მოდელირებისათვის გვაქვს შემდეგი ძირითადი იერარქიული მოდულები (ნახ.3.11): ახალი საბაზრო სტრატეგიის ფორმირება (A), საბაზრო მოთხოვნების განსაზღვრა (B); პროდუქციის წარმოების დაგეგმვა (C); წარმოების ტექნიკური მომზადება და პროდუქციის წარმოება (D); პროდუქციის გაცემა (სასაწყობო მეურნეობა) და პროდუქციის გადაგზავნა (ტრანსპორტირება) (E), პროდუქციის მიღება და დამკვეთის ინფორმირება (F); ფაქტობრივი მდგომარეობის აღრიცხვა, საწარმოო და სარეალიზაციო გეგმების შესრულების ანალიზი, ეკონომიკური მაჩვენებლების ანგარიში და ანალიზი (G); გადაწყვეტილების მიღება ახალი საბაზრო სტრატეგიისთვის (A) და ა.შ. ციკლურად.

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

3. ექსპერიმენტული ნაწილი

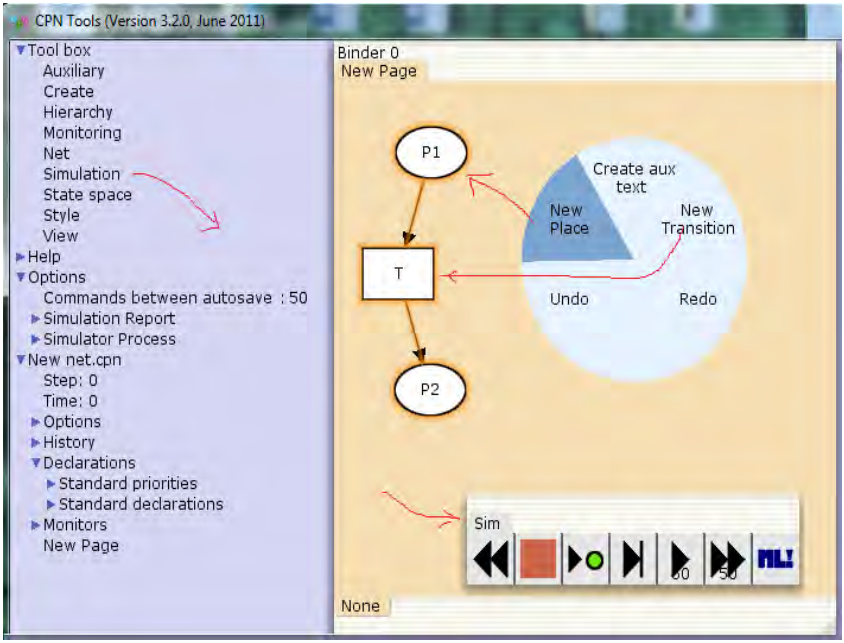
დანიელი მეცნიერ-ინჟინრების მიერ აარჰუსის უნივერსიტეტში დამუშავდა პეტრის ქსელის გაფართოებული ვარიანტი - პეტრის ფერადი ქსელი (Coloured Petri Net), ML ენითა და CPN-ინსტრუმენტით [15]. იგი იყენებს გრაფო-ანალიზური, ობიექტ-ორიენტირებული, ვიზუალური დაპროგრამების პრინციპებს. CPN ML ენა საშუალებას იძლევა აღიწეროს ქსელის ფერადი კომპონენტები (მარკერები), ცვლადები, კონსტანტები და თვით პოზიციების, გადასასვლელებისა და რკალების ტექსტური აღწერები, რაც ერთგვარ კომფორტს ქმნის ქსელის წასაკითხად და გასაგებად [14].



ნახ.3.11. მარკეტინგული პროცესების
პეტრის ქსელის მოდელი

ამჯერად მოკლედ განვიხილოთ CPN ინსტრუმენტის სამუშაო გარემო, მისი დახმარებით მოდელის აგების და შემდეგ ბიზნესპროცესის იმიტაციური რეჟიმის ამუშავება.

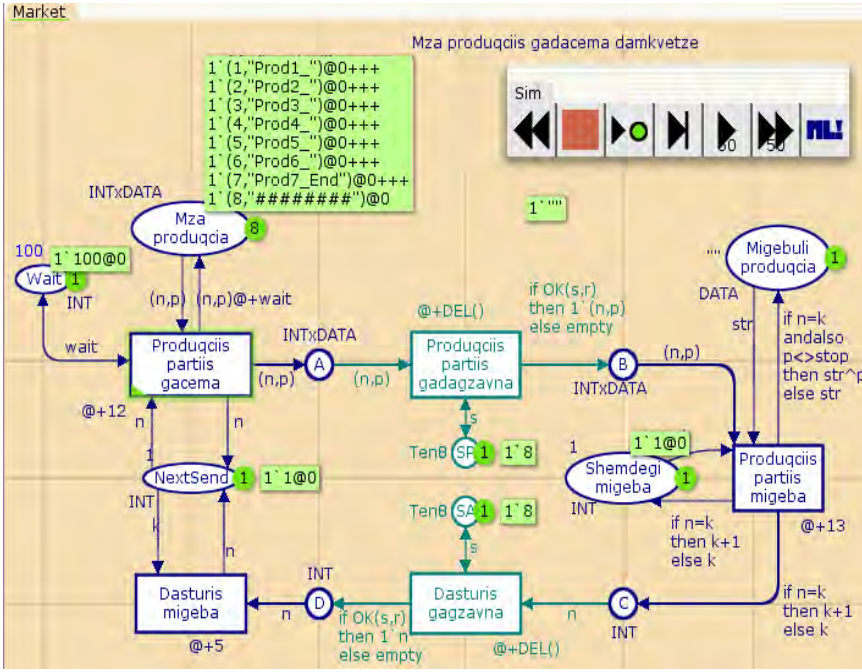
3.12 ნახაზზე ნაჩვენებია CPN-ის მომხმარებლის საწყისი ინტერფეისი (ვუშვებთ, რომ CPN პაკეტი დაინსტალირებულია კომპიუტერში [16]. იგი უფასო ვერსიაა და ფართოდ გამოიყენება აშშ, ევროპის, ჩინეთის და სხვა ქვეყნების უნივერსიტეტებში).



ნახ.3.12. CPN სამუშაო გარემო

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

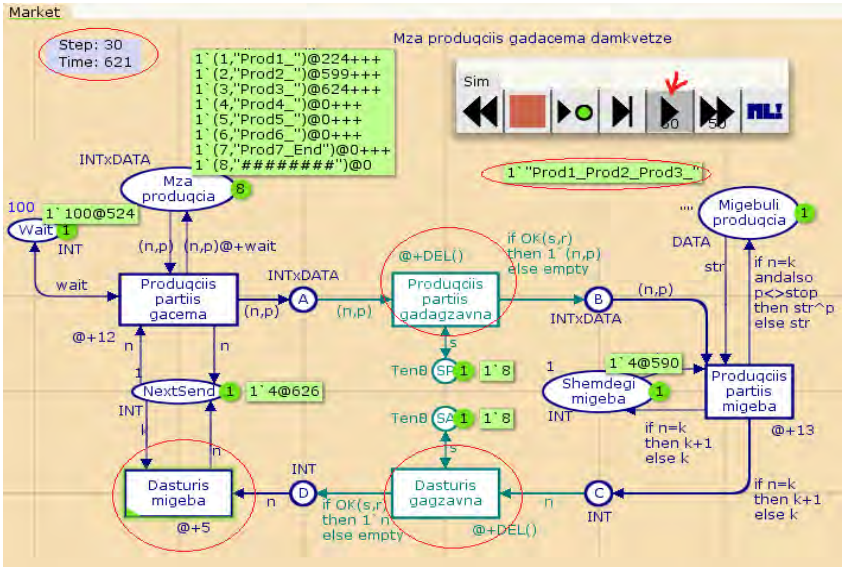
3.13 ნახაზზე ნაჩვენებია აგებული E-ბლოკის მაგალითი.



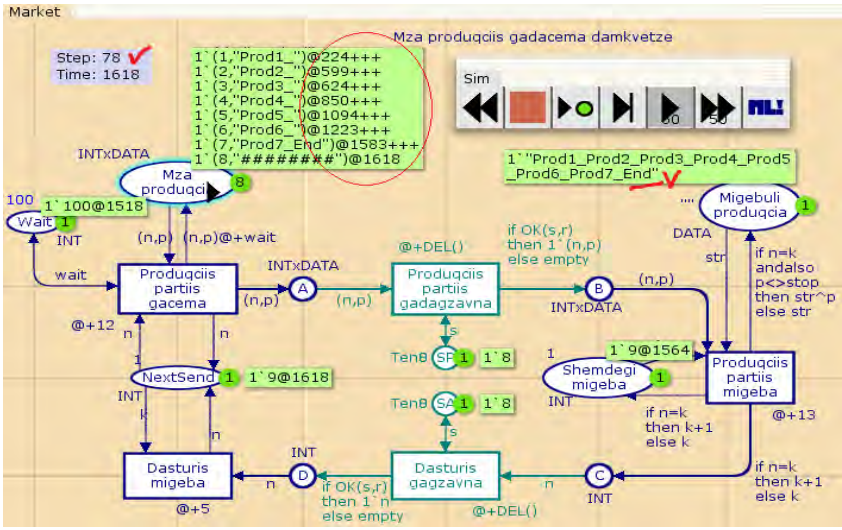
ნახ.3.13. „პროდუქციის მიწოდების“ ბიზნესპროცესის პეტრის ქსელის ფრაგმენტი (საწყისი პოზიცია)

სიმულატორის ამუშავებით (ნახ.3.14) ჩანს პროცესის დინამიკა, ანუ მიმდევრობით ხდება „ფერადი მარკერების“ (სხვადასხვა სახის პროდუქციის) გაგზავნა დამკეთებზე. წითელი წრეხაზებით აღნიშნული გვაქვს პარალელურად შესრულებადი პროცესები (გადასასვლელების გახნა). სქემაზე ჩანს ასევე დროის მომენტები და დამკვეთთან უკვე მისული პროდუქციის დასახელებები.

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



ნახ.3.14. იმიტაციური პროცესი (შუალედური ბიჯი = 30).



ნახ.3.15. პროცესი დასრულდა (ბიჯი = 78).

3.5. მარკეტინგული პროცესების პროგრამული რეალიზაცია WF ტექნოლოგიით /გაგრძელება/ (ლაბ.N15)

მიზანი: მარკეტინგული ბიზნესპროცესების პროგრამული რეალიზაციის შესწავლა Workflow, WPF და WCF ტექნოლოგიების საფუძველზე, კლიენტ-სერვერ არქიტექტურის პრინციპის განხორციელებით.

1. თეორიული ნაწილი

აქ წარმოდგენილი ამოცანა არის წინა ლაბორატორიაზე განხილული ამოცანის გაგრძელება, ანუ როგორ დავაპროგრამოთ მარკეტინგული მენეჯმენტის ის ბიზნესპროცესები, რომელთა მოდელი ავაგეთ პეტრის ქსელის CPN ინსტრუმენტით.

კლიენტ-სერვერული ბიზნეს–პროცესების შესრულებისას ერთ–ერთი მნიშვნელოვანი საკითხია კომუნიკაცია „მიმწოდებელ-დამკვეთ" აპლიკაციებს შორის, ან კლიენტებსა და სერვერებს შორის. ასეთი კავშირების რეალიზაცია შესაძლებელია ბიზნეს–პროცესებსა და ჰოსტ–დანართებს შორის. ქვემოთ განვიხილავთ ჩვენი მაგალითის შესაბამისი პროგრამების ფრაგმენტებს.

აპლიკაციის მაგალითის სახით ვიხილავთ პროექტის აგებას პროდუქციის მიმწოდებელ ფირმასა და დამკვეთ ორგანიზაციას შორის. კერძოდ, მიმწოდებელი (Firm) აგზავნის პროდუქციის პარტიას (Product), შესაბამისი თანმხლები დოკუმენტით, ფაქტურით (Invoice) დამკვეთთან.

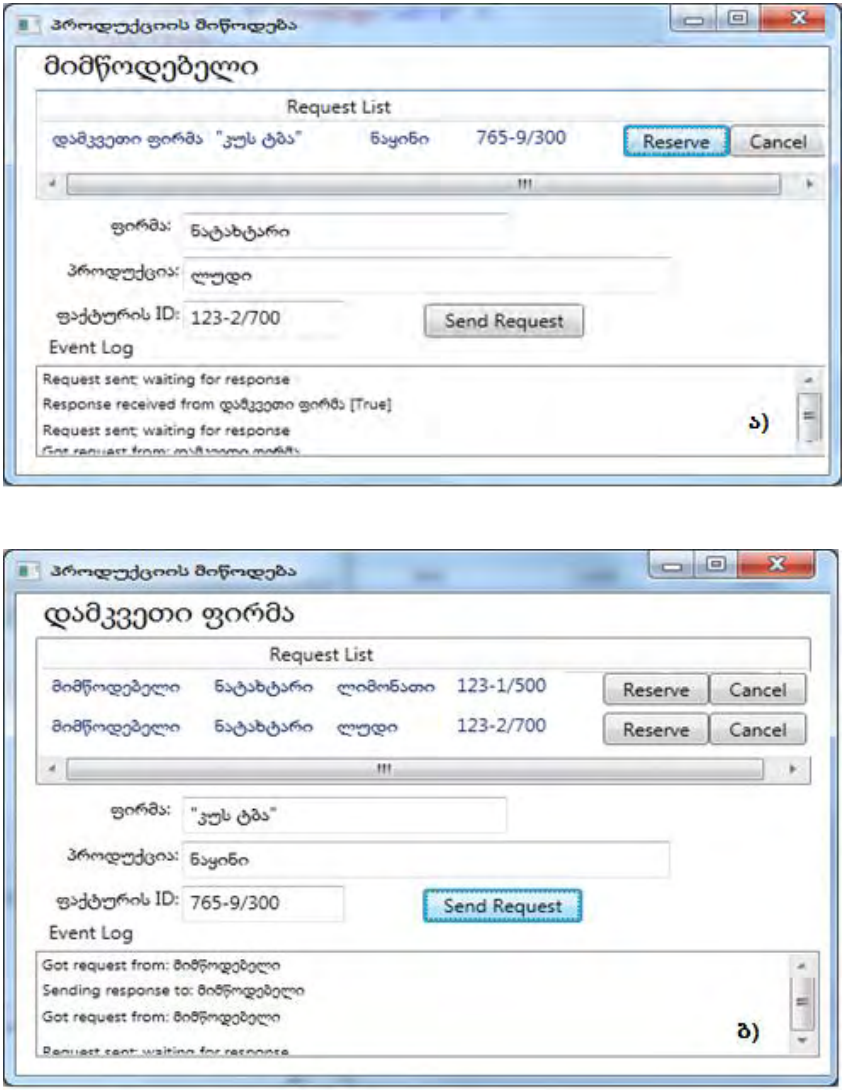
როდესაც დამკვეთი ფირმა მიიღებს პროდუქციას, იგი საპასუხო შეტყობინებას უბრუნებს მიმწოდებელს, რითაც ეს „ტრანზაქცია" დასრულებულად ითვლება. მიმწოდებლის იგივე აპლიკაციას შეუძლია მოთხოვნის გაგზავნა სხვა დამკვეთთან და ასევე საპასუხო შეტყობინების მიღება მათგან. თუ საპასუხო შეტყობინება არ დაბრუნდა მიმწოდებელთან, ეს ნიშნავს, რომ შეკვეთა არაა შესრულებული და შესაძლებელია პროდუქციის იგივე პარტია კვლავ გაიგზავნოს დამკვეთთან.

2. პრაქტიკული ნაწილი

მთავარი ქმედებები, რომლებიც კომუნიკაციისთვის გამოიყენება არის Send და Receive ქმედებები (და მათი ვარიაციები: SendReply და ReceiveReply). ეს ქმედებები გამოიყენებს Windows Communication Foundation (WCF) ტექნოლოგიას შეტყობინებათა გადასაცემად და სამეთვალყურეოდ [17,18]. ჩვენ ავაგებთ მარტივ WPF-აპლიკაციას (Windows Presentation Foundation), რომელიც გამოიყენებს კომუნიკაციას, მაგალითად ორ სხვადასხვა აპლიკაციის (მიმწოდებელი და დამკვეთი) ბიზნეს–პროცესებს შორის.

3.16-ა,ბ ნახაზებზე მოცემულია საილუსტრაციო ფრაგმენტები „მიმწოდებელ-დამკვეთ" აპლიკაცებს შორის, თუ როგორი შეიძლება იყოს მათი ინტერფეისები. მაგალითად, ფირმა „ნატახტარი" აგზავნის შეკვეთილი „ლიმონათის პარტიას", ფაქტურით „123-1/500", ამოქმედდა „Send Request" ღილაკი და დამკვეთი ფირმის ინტერფეისზე „Request List"-ში გამოჩნდა სტრიქონი ამის შესახებ. ეს ნიშნავს პროდუქციის ადგილზე მიტანას (ჩვენ მაგალითში ეს მყისიერად მოხდა, თუმცა შესაძლებელია გარკვეული დაყოვნების დროის გამოყენებაც, შემთხვევით რიცხვთა გენერატორის დახმარებით, რადგან პროცესი სტოქასტურია) [1]. გარკვეული დროის შემდეგ, მიმწოდებელი აგზავნის მეორე შეკვეთას, „ლუდის პარტიას", ფაქტურით „123-2/700" და ა.შ.

დამკვეთი ფირმა, პროდუქციის პარტიის მიღების შემთხვევაში, იყენებს „Reserve" ღილაკს, რაც უზრუნველყოფს მიმწოდებლის ინფორმირებას პროდუქციის ამ პარტიის მიღების შესახებ. ეს შეტყობინება მიმწოდებლის ინტერფეისზე აისახება მოვლენათა რეგისტრაციის, „Event Log" ლისტბოქსში. 3.1 ლისტიგნში მოცემულია „მიმწოდებლის" ინტერფეისის პროგრამული აპლიკაციის კონფიგურაციის ფაილი (App.config). აქ ყურადსაღებია პორტის ნომრები (Address, Request Address), რომლებიც „დამკვეთი ფირმისთვის" იგივეა, ოღონდ შებრუნებული.



ნახ.3.16-ა,ბ. „მიმწოდებელი-დამკვეთი“ აპლიკაციის ინტერფეისები

```
<-- ლისტინგი–3.1: --- App.config ---
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="Branch Name" value="მიმწოდებელი"/>
    <add key="ID" value="{43E6DADD-4751-4056-8BB7-
7459B5C361AB}"/>
    <add key="Address" value="8730"/>
    <add key="Request Address" value="8000"/>
  </appSettings>
</configuration>
```

ორივე პროგრამული აპლიკაცია, გაიშვება „Run as administrator" რეჟიმში. ერთ კომპიუტერზე (ექსპერიმენტისათვის) ერთდროულად ჩანს ორი ფანჯარა (ნახ.3.16-ა,ბ). ინფორმაციის მომზადება და გადაცემა, ასევე შეტყობინების გაგზავნა შესაძლებელია ორივე მიმართულებით. 3.2 და 3.3 ლისტინგებში ნაჩვენებია

```
// -- ლისტინგი_3.2--- CreateRequest.cs ---
using System;
using System.Activities;
using System.Configuration;
namespace ProductDelivery  // პროდუქციის მიწოდება
{ public sealed class CreateRequest : CodeActivity
    {  public InArgument<string> Product { get; set; }
        public InArgument<string> Firm { get; set; }
        public InArgument<string> InvoiceID { get; set; }
public OutArgument<ReservationRequest> Request {get; set;}
     public OutArgument<string> RequestAddress {get; set;}

protected override void Execute(CodeActivityContext
                                                  context)
   {
     // config ფაილია გახსნა და Request Address-ის მიწოდება
```

```
Configuration config = ConfigurationManager
       .OpenExeConfiguration(ConfigurationUserLevel.None);
   AppSettingsSection app =
  AppSettingsSection)config.GetSection("appSettings");
// ReservationRequest კლასის შექმნა და მისი შევსება არგუმენტებით
      ReservationRequest r = new ReservationRequest
        (
          Product.Get(context),
          Firm.Get(context),
          InvoiceID.Get(context),
          new Branch
           {
          BranchName = app.Settings["Branch Name"].Value,
          BranchID = new Guid(app.Settings["ID"].Value),
             Address = app.Settings["Address"].Value
           },
          context.WorkflowInstanceId
         );

       // მოთხოვნის შენახვა OutArgument-ში
         Request.Set(context, r);

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

// -- ლისტინგი_3.3 --- CreateResponse. cs ---

```
using System;
using System.Activities;
using System.Configuration;
namespace ProductDelivery
 {
```

```
  public sealed class CreateResponse : CodeActivity
   {
   public InArgument<ReservationRequest> Request {get; set;}
   public InArgument<bool> Reserved { get; set; }
public OutArgument<ReservationResponse> Response {get;
set;
}

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

(AppSettingsSection)config.GetSection("appSettings");

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

3.4 ლისტინგში მოცემულია კლიენტის სერვისის კლასის კოდის ფრაგმენტი.

```
// -- ლისტინგი–3.4 --- ClientService.cs   ---
using System;
using System.ServiceModel;

namespace ProductDelivery
{
  public class ClientService : IProductDelivery
   {
    public void RequestProduct(DeliveryRequest request)
      {
         ApplicationInterface.RequestProduct(request);
      }

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

ლიტერატურა:

1. სურგულაძე გ. ვიზუალური დაპროგრამება C#_2010 ენის ბაზაზე. სტუ, თბ., 2011

2. Мак-Дональд М. WPF: Windows Presentation Fundation в .NET 3.5 с примерами на C# 2008 для профессионаюв. 2-е издание: Пер. с англ. - М. : ООО "И.Д. Вильяме". 2008

3. Petzold Ch. Applications=Code+Markup. A Guide to the MicroSoft Windows Presentation Foundation. St-Petersburg. 2008

4. Уотсон К.,Нейгел К., Педерсен Я., ХаммерР., Джон Д., Скиннер М., Уайт Э. Visual C# 2008: базовый курс. : Пер. с англ. - М. : ООО "И.Д. Вильяме", 2009

5. Долженко А.И. Разработка приложений на базе WPF и Silverlight. –М., 2011, www.intuit.ru/department/se/dawpfsl/

6. Снетков В.М. Практикум прикладного программирования на C# в среде VS.NET 2008. www.intuit.ru/department/se/prcsharp08/

7. Eberhardt C. WPF DataGrid Practical Examples. 2009. http://www.codeproject.com/Articles/30905/WPF-DataGrid-Practical-Examples.

8. ბიტარაშვილი მ., სურგულაძე გ. საგადასახადო სამართალდარღვევის საქმის წარმოების სისტემის ბიზნესპროცესების მოდელირება UML2 ტექნოლოგიით. სტუ-ს შრ.კრ. „მართვის ავტომატიზებული სისტემები“. N2(13), თბილისი, 2012, გვ. 217–222.

9. სურგულაძე გ., ოხანაშვილი მ., კაშიბაძე მ., ნეფარიძე მ. თანამედროვე საინფორმაციო ტექნოლოგიები მარკეტინგული პროცესების და წარმოების მენეჯმენტში. სტუ-ს შრ.კრ. „მართვის ავტომატიზებული სისტემები“. N1(17), თბილისი, 2014, გვ. 64-71.

10. თურქია ე. ბიზნეს–პროექტების მართვის ტექნოლოგიური პროცესების ავტომატიზაცია. მონოგრ., სტუ. თბ., 2010

11. საქართველოს საგადასახადო კოდექსი. თბ., 17 სექტ., 2010

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

13. სურგულაძე გ., ბიტარაშვილი მ. ბიზნესპროცესების UML-მოდელირება და პროგრამული რეალიზაცია Workflow Foundation ტექნოლოგიით საგადასახდო დავების სისტემის მაგალითზე. სტუ-ს შრ.კრ. „მართვის ავტომატიზებული სისტემები“. N1(14), თბილისი, 2013, გვ. 229–233.

14. სურგულაძე გ., ოხანაშვილი მ., სურგულაძე გ. მარკეტინგის ბიზნეს_პროცესების უნიფიცირებული და იმიტაციური მოდელირება. მონოგრ., სტუ. თბ., 2009

15. Jensen K., Kristensen M.L., Wells L. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. University of Aarhus. Denmark. 2007

16. CPN Tools. www.daimi.au.dk/CPNTools/

17. Collins M.J. Beginning WF: Windows Workflow in .NET 4.0. ISBN-13 (pbk): 978-1-4302-2485-3 Copyright © 2010. USA. http://www.ebooks-it.net/ebook/beginning-wf.

18. Уотсон К.,Нейгел К., Педерсен Я., ХаммерР., Джон Д., Скиннер М., Уайт Э. Visual C# 2008: базовый курс. : Пер. с англ. - М. : ООО "И.Д. Вильямс", 2009

19. სურგულაძე გ. კორპორაციული მენეჯმენტის სისტემების Windows დეველოპმენტი (WPF ტექნოლოგია). სტუ, თბ., 2014.

გადაეცა წარმოებას 15.03.2015 წ. ხელმოწერილია დასაბეჭდად 15.04.2015 წ. ოფსეტური ქაღალდის ზომა 60X84 1/16. პირობითი ნაბეჭდი თაბახი 8,25. ტირაჟი 100 ეგზ.

Verba volant, scripta manent

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

ISBN 978-9941- 0-7520-9

გია სურგულაძე

gsurg@gmx.net

სტუ-ს „პროგრამული ინჟინერიის“ დეპარტამენტის სრული პროფესორი, ტექნიკის მეცნიერებათა დოქტორი. IT-კონსალტინგის სამეცნიერო ცენტრის ხელმძღვანელი, საერთაშორისო სამეცნიერო ჟურნალის „მართვის ავტომატიზებული სისტემები“ რედაქტორი.

აქვს 300-ზე მეტი სამეცნიერო ნაშრომი, მათ შორის 60 წიგნი, 40 ელექტრონული სახელმძღვანელო მართვის საინფორმაციო სისტემების და მონაცემთა ბაზების დაპროექტების და აგების სფეროში.

არის გერმანიის DAAD-ის მრავალგზის გრანტის მფლობელი. ბერლინის ჰუმბოლდტის, პასსაუს, მაგდებურგის და ნიურნბერგ-ერლანგენის უნივერსიტეტების მიწვეული პროფესორი 1974-2014 წლებში.