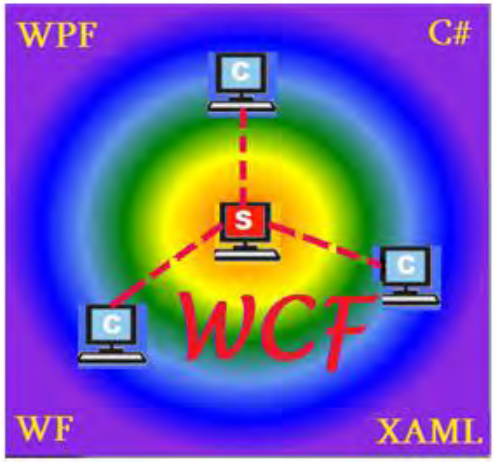
გია სურგულაძე

კორპორაციული მენეჯმენტის სისტემების Windows დეველოპმენტი: WCF ტექნოლოგია



„IT-კონსალტინგის ცენტრი“

საქართველოს ტექნიკური უნივერსიტეტი

გია სურგულაძე

კორპორაციული მენეჯმენტის სისტემების Windows დეველოპმენტი: WCF ტექნოლოგია

(ლაბორატორიული პრაქტიკუმი, ნაწ.3)

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 4.0/4.5 ინტეგრირებულ გარემოში ორგანიზაციული მართვის კომპიუტერული სისტემების (Windows- და Web-აპლიკაციების) დაპროგრამების ინსტრუმენტული საშუალება Windows Communication Foundation (WCF) ტექნოლოგიის ბაზაზე. იგი ეფუძნება XAML (სისტემის დიზაინის ნაწილი) და C# (სიტემის ლოგიკური ნაწილი) ენების კომპლექსურ გამოყენებას. წარმოდგენილია ლაბორატორიული პრაკტიკუმის ამოცანები და მეთოდური ინსტრუქციები ასეთი სისტემების კომპონენტების დასაპროგრამებლად (WPF და Workflow ტექნოლოგიებთან ერთად).

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

რეცენზენტები:

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

პროფ. გ. სურგულაძის რედაქციით

ISBN 978-9941-0-7103-4 (ყველა ნაწილის)

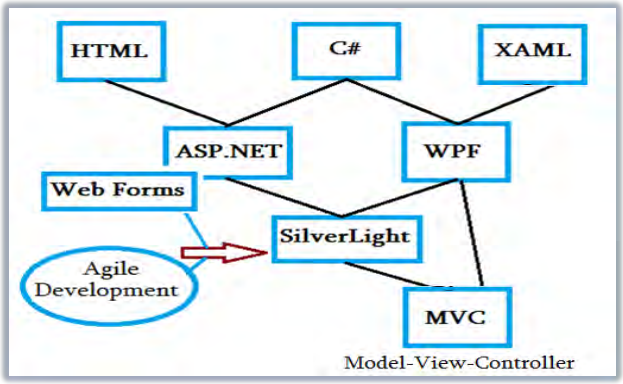
ISBN 978-9941-0-7878-1 (მესამე ნაწილის)

შინაარსი

შესავალი

აპლიკაციების (დანართების) ორი ნაირსახეობაა ცნობილი: „დესკტოპ“ (ვინდოუსის) სისტემები, რომელთაც ასევე სამაგიდო აპლიკაციებს უწოდებენ და ვებ-აპლიკაციები, რომელთა გამოყენებაც ინტერნეტ ბრაუზერებიდანაა შესაძლებელი [1-5].

ეს დანართები იქმნება .NET Framework -ის ორი სხვადასხვა პაკეტით. პირველი - Windows Forms კომპონენტებით და მეორე ASP.NET -ის საშუალებით [3,4]. ორივეს აქვს თავისი უპირატესობები და ნაკლოვანებანი [7]. WPF და ASP.NET ინსტრუმენტების ბაზაზე კი განვითარდა ახალი „მსუბუქი“ ტექნოლოგიები, როგორიცაა ASP Silverlight, ASP MVC და ა.შ. 1-ელ ნახაზზე ნაჩვენებია „მაიკროსოფთის“ კორპორაციის მიერ ვებ-აპლიკაციების განვითარების სქემა Visual Studio.NET Framework-ის ბაზაზე [7,8].



ნახ.1. Web-აპლიკაციების ტექნოლოგიების ევოლუცია

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 [9].

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.

WPF – Windows Presentation Foundation ტექნოლოგია აგებს მაღალეფექტიან სამომხმარებლო დიზაინის მქონე ისეთ პროგრამულ დანართებს, რომლებშიც გამორიცხულია დაპირის-პირება სამაგიდო აპლიკაციასა და ინტერნეტს შორის [1,6,9,10,12]. WPF-დანართს შეუძლია ფუნქციონირება როგორც სამაგიდო აპლიკაციის, ასევე როგორც ვებ-აპლიკაციას ბრაუზერის შიგნით.

WF - Workflow Foundation ტექნოლოგია .NET-ში არის სრულიად ახალი პარადიგმა სამუშაო (ბიზნეს) პროცესებზე (workflow) ბაზირებული აპლიკაციების ასაგებად. იგი ფუნდამენტურად ახლად გააზრებული ტექნოლოგიაა [2,5,11].

WCF - Windows Communication Foundation ტექნოლოგია. ამ წიგნში, რომელიც ფაქტობრივად, არის „კორპორაციული მენეჯმენტის სისტემების Windows-დეველოპმენტის“ მესამე ნაწილი, დეტალურად განიხილება WCF ტექნოლოგიის ძირითადი ცნებები, საბაზო სტრუქტურები და მათი გამოყენების მეთოდოლოგიური საკითხები, დაპროგრამების პრინციპების თვალსაზრისით.

ნაშრომის სამ თავში შემოთავაზებულია 16 ლაბორატორიული სამუშაო, შესაბამისი პრაქტიკული ამოცანებით და ექსპერიმენტული სავარჯიშოებით, რომელთა სირთულის მიხედვით შესაძლებელია მათი გამოყენება ბაკალავრიატის და

მაგისტრანტების ჯგუფებში, სპეციალობით მართვის საინფორმაციო სისტემების პროგრამული ინჟინერია.

ბიზნესპროცესების შესრულებისას ერთ–ერთი მნიშვნელოვანი ასპექტია ურთიერთობა (კომუნიკაცია) აპლიკაციებს შორის, კლიენტებსა და სერვერებს შორის, აგრეთვე მუშა–პროცესებსა და ჰოსტ–დანართებს შორის. სახელმძღვანელოში გავეცნობით თუ როგორ შეიძლება ბიზნესპროცესების გამოყენებით გამარტივდეს და კოორდინაცია გაეწიოს კომუნიკაციათა სხვადასხვა სცენარებს.

პირველ ცხრილში მოცემულია წიგნში განხილული საკითხების გამოყენების მაგალითები სხვადასხვა საპრობლემო სფეროებში. აპლიკაციების საკომუნიკაციო ამოცანების მაგალითებზე ხდება კლიენტ-სერვერული და სერვის ორიენტირებული სისტემების დაპროგრამების შესწავლა.

პროექტში განხილული საპრობლემო სფეროების სია ცხრ.1

N	საპრობლემო სფერო	სერვერი	კლიენტი	[ლიტ]
1	უნივერსიტეტი	1. ტრეინინგ ცენტრი. 2. სასწავლო ნაწილი	ფილიალი დეკანატი	[13]
2	ბიბლიოთეკა	ცენტრალური ბიბლიოთეკა	ბიბლიოთეკის ფილიალები	[11]
3	საწარმოო ფირმის მარკეტინგი	1. დამკვეთი. 2. მიმწოდებელი	მიმწოდებელი. დამკვეთი	[14,15]
4	ელ-საარჩევნო სისტემა	1. ცენტრალური სს. 2. რეგიონი	რეგიონი. ოლქი	[16,17]
5	საფინანსო-საკრედიტო სისტემა	1. კომერც-ბანკი. 2. კრედიტორი	კრედიტორი. კომერც-ბანკი	[18]

აპლიკაციის საილუსტრაციო მაგალითის სახით განიხილება პროექტების აგება უნივერსიტეტის სასწავლო პროცესის,

ტრეინინგ-ცენტრის, ბიბლიოთეკის, წარმოების, ბიზნესის და სხვა სფეროს მაგალითებისთვის, რომლებშიც ინფორმაციის („მოთხოვნა-პასუხის“) გადაცემა მათ ფილიალებს ან სტრუქტურულ ერთეულებს შორის ხორციელდება კლიენტ-სერვერული ან სერვის-ორიენტირებული არქიტექტურის ტექნოლოგიებით და პრინციპებით.

საილუსტრაციო მაგალითის სახით პირველ თავში ჩვენ ავაგებთ უნივერსიტეტის ტრეინინგ-ცენტრისა და მისი ფილიალების აპლიკაციებს კონსოლის რეჟიმში. პროგრამის დუბლირებით და მისი გარკვეული პარამეტრების კორექტირებით შევქმნით ფილიალის აპლიკაციას. შემდეგ კი ამ აპლკაციებს შორის დავამყარებთ კომუნიკაციას და გავცვლით ინფორმაციას.

მეორე თავში ეს ამოცანა გადაწყდება ვიზუალურ რეჟიმში WPF/WF/WCF ტექნოლოგიებით.

მესამე თავში კი განხილულ იქნება Web-სერვისების შექმნის საკითხები ბიზნესპროცესების ავტომატიზებული დამუშავების აპლიკაციების ასაგებად, მათი ტესტირების ამოცანის განხილვით.

I თავი

კომუნიკაცია აპლიკაციებს შორის WCF-ის ბაზაზე

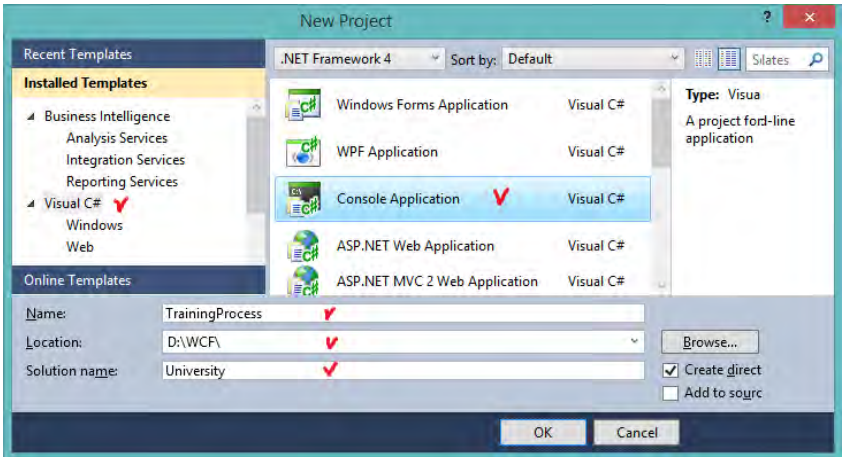
1.1. ინფორმაციის „გადაცემის“ და „მიღების“ ქმედებები (ლაბ-1)

მიზანი: Visual Studio.NET პაკეტის WCF-ის სამუშაო გარემოს გაცნობა და პირველი აპლიკაციის აგება. Send და Receive ქმედებების გაცნობა და დაპროგრამება.

მთავარი ქმედებები (Activities), რომლებიც კომუნიკაციისთვის გამოიყენება არის Send (გაგზავნა) და Receive (მიღება). ეს ქმედებები (და მათი ვარიაციები: SendReply და ReceiveReply) გამოიყენებს Windows Communication Foundation (WCF) ტექნოლოგიას შეტყობინებათა გადასაცემად და მონიტორინგის პროცესისთვის.

1.1.1. ახალი პროექტის შექმნა კონსოლის რეჟიმში

შევქმნათ ახალი პროექტი TrainingProcess სახელით, Solution name: University და Windows -> Console Application რეჟიმში.

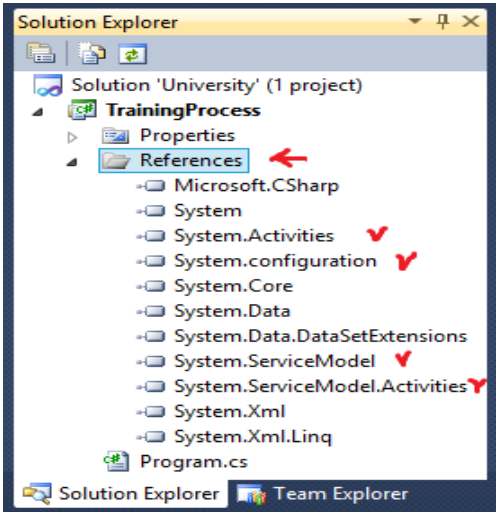


ნახ.1.1

TrainingProcess პროექტის Solution Explorer–ში მაუსის მარჯვენა ღილაკით ავირჩიოთ Add Reference და .NET tab-დან დავამატოთ შემდეგი სტრიქონები.

- System.Activities
- System.Configuration
- System.ServiceModel
- System.ServiceModel.Activities

მიიღება:



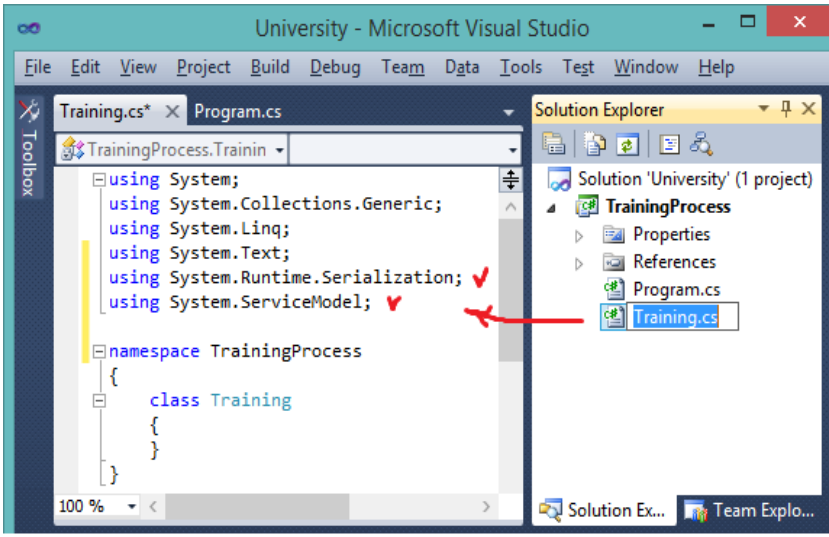
ნახ.1.2

1.1.2. შეტყობინებათა განსაზღვრა

საჭიროა შეიქმნას კლასი, რომელიც განსაზღვრავს შეტყობინებებს აპლიკაციებს შორის. Solution Explorer-იდან Add -> Class და ჩავწეროთ კლასის სახელი Training.cs. ამ ფაილში დავამატოთ ორი სახელსივრცე (namespaces):

```
using System.Runtime.Serialization;
using System.ServiceModel;
```

მიიღება 1.3 ნახაზზე ნაჩვენები სურათი.



ნახ.1.3. Training.cs კლასის შექმნა

Training.cs ფაილში განვსაზღვროთ (ჩავამატოთ) სამი კლასი:

- Branch: განსაზღვრავს **მონაცემებს** ტრეინინგ-ცენტრის ფილიალის **მდებარეობის შესახებ;**
- TrainingRequest: განსაზღვრავს ფილიალის **მოთხოვნას** სასწავლო საგნის (კურსის დისციპლინის) მითითებით;
- TrainingResponse: განსაზღვრავს ტრეინინგ-ცენტრის პასუხს მოთხოვნის შესაბამისი ფილიალისთვის.

Branch კლასის განსაზღვრა მოცემულია 1.1 ლისტინგში. შევიტანოთ იგი namespace TrainingProcess-ის შიგნით. წავშალოთ ცარიელი კლასი Training, რომელიც წინა ბიჯზე შეიქმნა.

```
// --- ლისტინგი_ 1 .1 ---
// --- ფილიალის მონაცემთა სტრუქტურის განსაზღვრა ---
```

```
public class Branch
{
  public String BranchName { get; set;}
  public String Address { get; set; }
  public Guid BranchID { get; set; }
  #region Constructors
  public Branch()    {   }
  public Branch(String name, String address)
    {
      BranchName = name;
      Address = address;
      BranchID = Guid.NewGuid();
    }
 public Branch(String name, String address, Guid id)
   {
     BranchName = name;
     Address = address;
     BranchID = id;
   }
public Branch(String name, String address, String id)
   {
     BranchName = name;
     Address = address;
     BranchID = new Guid(id);
   }
#endregion Constructors
}
```

Branch კლასს აქვს სამი დასამახსოვრებელი წევრი“: სახელი, ქსელის მისამართი და უნიკალური იდენტიფიკატორი. ზოგიერთი კონსტრუქტორი შემოტანილია გამოყენების გასამარტივებლად. აქ

დამატებულია რეგიონის მარკერები კონსტრუქტორის ირგვლივ, შესაძლებელი რომ იყოს კოდის შეკუმშვა მისი კითხვადობის გასაუმჯობესებლად.

ახლა დავამატოთ TrainingRequest კლასის განსაზღვრება:

```
//---ლისტინგი_1.2  ---
// --- მოთხოვნის შეტყობინების განსაზღვრა: TrainingRequest ---
[MessageContract(IsWrapped = false)]
public class TrainingRequest
    {
        private String _Jgupi;
        private String _Sagani;
        private String _Lector;
        private Guid _RequestID;
        private Branch _Requester;
        private Guid _InstanceID;

        #region Constructors
        public TrainingRequest() { }

public TrainingRequest(String sagani, String lector,
                           String jgupi, Branch requestor)
    {
       _Sagani = sagani;
       _Lector = lector;
       _Jgupi = jgupi;
       _Requester = requestor;
       _RequestID = Guid.NewGuid();
    }
public TrainingRequest(String sagani, String lector,
                  String jgupi, Branch requestor, Guid id)
    {
       _Sagani = sagani;
       _Lector = lector;
       _Jgupi = jgupi;
       _Requester = requestor;
       _RequestID = id;
     }
     #endregion Constructors
```

```
    #region Public Properties
    [MessageBodyMember]
    public String sagani
      {
           get { return _Sagani; }
           set { _Sagani = value; }
      }
    [MessageBodyMember]
     public String Jgupi
      {
           get { return _Jgupi; }
           set { _Jgupi = value; }
      }
     [MessageBodyMember]
     public String Lector
      {
           get { return _Lector; }
           set { _Lector = value; }
      }
     [MessageBodyMember]
     public Guid RequestID
       {
           get { return _RequestID; }
           set { _RequestID = value; }
       }
     [MessageBodyMember]
     public Branch Requester
       {
           get { return _Requester; }
           set { _Requester = value; }
       }
     [MessageBodyMember]
     public Guid InstanceID
       {
           get { return _InstanceID; }
           set { _InstanceID = value; }
       }
       #endregion Public Properties
}
```

TrainingRequest კლასი შედგება Jgupi, Sagani და Lector წევრებისაგან მოთხოვნილი ტრეინინგის კურსის აღსაწერად. იგი შეიცავს ასევე Branch კლასს, რომელიც ასახავს მოთხოვნილი ტრეინინგის განმახორციელებელ ფილიალს.

1.1.3. კონტრაქტის შეტყობინება

ვინაიდან TrainingRequest კლასი გამოყენებულ უნდა იყოს გამომავალი შეტყობინების განსაზღვრისთვის, MessageContract ატრიბუტი მიუთითებს, რომ ეს კლასი ჩართულ იქნება SOAP ბარათში. SOAP-ის გამოყენების დროს შეტყობინებები გადაიცემა XML-ის მსგავსი ფორმატირებადი ენით. ეს უზრუნველყოფს კლიენტებსა და სერვერს შორის მაღალი ხარისხის ურთიერთ-ქმედების პლატფორმას. SOAP არის სტანდარტული პროტოკოლი, რომლის მხარდაჭერაც აქვს WCF-ს.

არსებობს აგრეთვე MessageBodyMember ატრიბუტი მის ყოველ პაბლიკ–თვისებაზე. ეს აუცილებელია WCF-ფენისთვის, რათა სწორად დაფორმატდეს SOAP შეტყობინება.

ახლა შევიტანოთ რეალიზაციის კოდი TrainingResponse კლასისითვის, რომელიც 1.3 ლისტინგზეა მოცემული.

```
//--- ლისტინგი_1.3 ---
// პასუხის შეტყობინების განსაზღვრა: ReservationResponse ---
[MessageContract(IsWrapped = false)]
public class ReservationResponse
  {
    private bool _Reserved;
    private Branch _Provider;
    private Guid _RequestID;
   #region Constructors
    public ReservationResponse()  {  }
```

```
public ReservationResponse(ReservationRequest request,
                                    bool reserved,Branch provider)
      {
        _RequestID = request.RequestID;
        _Reserved = reserved;
        _Provider = provider;
       }
    #endregion Constructors
    #region Public Properties

 [MessageBodyMember]
 public bool Reserved
  {
    get { return _Reserved; }
    set { _Reserved = value; }
  }
 [MessageBodyMember]
  public Branch Provider
    {
      get { return _Provider; }
      set { _Provider = value; }
    }
  [MessageBodyMember]
   public Guid RequestID
    {
      get { return _RequestID; }
      set { _RequestID = value; }
    }
    #endregion Public Properties
}
```

TrainingResponse კლასი შეიცავს ლოგიკურ ელემენტს (Reserved), რომელიც მიუთითებს იყო თუ არა სატრეინინგო მოთხოვნა ხელმისაწვდომი. ის შეიცავს ასევე Branch კლასს, რომელიც ასახავს იმ ფილიალს, რომელმაც შესარულა ეს მოთხოვნა.

1.1.4. სერვისის კონტრაქტი

WCF–ის ბოლო წერტილის განსაზღვრისთვის არსებობს ინფორმაციის სამი პორცია, რომლებიც მითითებულ უნდა იქნას: მიერთება (binding), მისამართი და კონტრაქტი.

მიერთება მიუთითებს იმ პროტოკოლს, რომელიც გამოიყენება (მაგალითად, HTTP, TCP ან სხვ.).

მისამართი მიუთითებს თუ სად უნდა ვიპოვოთ ბოლო წერტილი და მისამართის ტიპს, რომლის გამოყენება დამოკიდებულია მიერთებაზე. მაგალითად, HTTP-მიერთებისას უნდა მიეთითოს URL, ხოლო TCP-თვის მისამართი იქნება სერვერის სახელი ან IP-მისამართი.

კონტრაქტი განისაზღვრება ServiceContract-ით, რომელიც არის ინტერფეისი. იგი განსაზღვრავს მეთოდებს, რომლებიც მიწვდომადია ბოლო წერტილში.

ამგვარად, ჩვენ განვსაზღვრეთ შეტყობინებები, რომლებიც გადაიცემა სერვის–მეთოდების მიერ პარამეტრების სახით.

ახლა დავამატოთ ინტერფეისის განსაზღვრება, რომელიც 1.4 ლისტინგზეა მოცემული იმავე Training.cs ფაილში.

```
//--- ლისტინგი_1.4 ---
//--- სერვისის კონტრაქტის განსაზღვრა: ITrainingProcess ---
// შეიცავს ორ მეთოდს: RequestTraining() and RespondToRequest()
[ServiceContract]
    public interface ITrainingProcess
    {
        [OperationContract(IsOneWay = true)]
```

```
        void RequestTraining(TrainingRequest request);

        [OperationContract(IsOneWay = true)]
        void RespondToRequest(TrainingResponse response);
    }
```

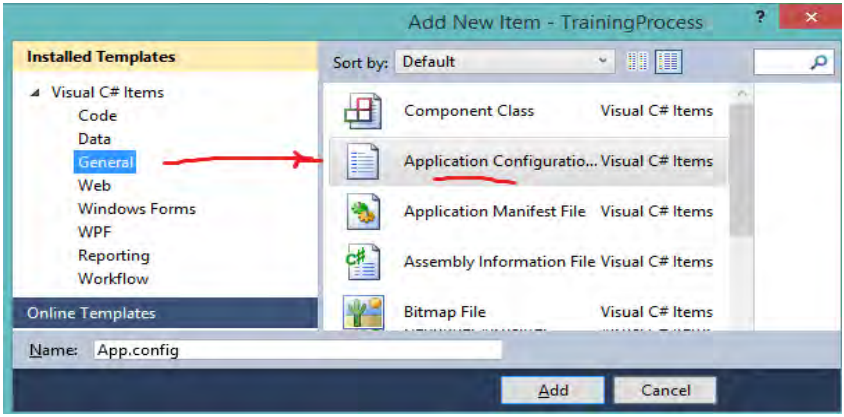
RequestTraining() მეთოდი გამოძახებულ იქნება კლიენტის მიერ TrainingRequest შეტყობინების გადასაცემად სერვერზე. ამასთანავე RespondToRequest() მეთოდი გააგზავნის TrainingResponse შეტყობინებას უკან – კლიენტისკენ.

დავაკომპილიროთ პროექტი (F6) და გავმართოთ კოდი.

1.1.5. აპლიკაციის კონფიგურაცია

როგორც ადრე აღვნიშნეთ, ჩვენ გვექნება აგებული აპლიკაციის რამდენიმე დუბლი (კოპიო), რომლებიც ასახავს სხვადასხვა ფილიალების განთავსებას. ფილიალთა მონაცემები ინახება კონფიგურაციის ფაილებში. **აპლიკაციის ყოველ კოპიოს ექნება თავისი კონფიგურაციის ფაილი, რომელიც შეიცავს თავის სპეციფიკურ ატრიბუტებს**.

TrainingProcess პროექტის Solution Explorer–დან ვირჩევთ Add New->Item. დიალოგურ ფანჯარაში General ჯგუფში ვირჩევთ Application Configuration File (ნახ.1.4).



ნახ.1.4. აპლიკაციის კონფიგურაციის შექმნა

ფაილის სახელი ავტომატურად არის App.config (). კონფიგუ-რაციის ფაილში შევიტანოთ საჭირო მონაცემები:

```
<!— ლისტინგი_1.5 ———— –>
<?xml version="1.0" encoding="utf-8" ?>
  <configuration>
    <appSettings>
      <add key="Branch Name" value="Central Library"/>
      <add key="ID" value="{43E6DADD-4751-4056-8BB7-
7459B5C361AB}"/>
      <add key="Address" value="8000"/>
      <add key="Request Address" value="8730"/>
   </appSettings>
</configuration>
```

<appSettings> სექციას აქვს მნიშვნელობები ფილიალის სახელი, ID (უნიკალური იდენტიფიკატორი) და მისამართი (პორტის ნომერი, რომელსაც აპლიკაცია იყენებს).

მოთხოვნის მისამართი განსაზღვრავს პორტის ნომერს, საითაც იქნება მოთხოვნები გაგზავნილი. შესაძლებელია სხვა პორტების გამოყენებაც, თუ ეს საჭიროა.

1.1.6. ბიზნესპროცესების განსაზღვრა

შემდეგი ბიჯი არის კლიენტის და სერვერის სამუშაო პროცესების განსაზღვრა. მათ შორის კომუნიკაცია ხორციელდება Send და Receive ქმედებათა გამოყენებით. ამ პროექტში ჩვენ გამოვიყენებთ Workflow-ის კოდირებას, ნაცვლად დიზაინერისა .xaml ფაილის გენერაციისათვის.

TrainingProcess პროექტის Solution Explorer-ში დავამატოთ კლასი: Add->Class, სახელით TrainingWF.cs და ჩავამატოთ სახელსივრცეები:

```
using System.Activities;
using System.Activities.Statements;
```

```
using System.ServiceModel.Activities;
using System.ServiceModel;
```

1.1.7. კლიენტი – მოთხოვნების გაგზავნა

პირველ რიგში უნდა განისაზღვროს სამუშაო პროცესი (workflow), რომლითაც მოთხოვნა გადაეცემა სხვა ფილიალს. TrainingWF კლასის კოდი შევცვალოთ 1.6 ლისტინგის მიხედვით.

```
//--- ლისტინგი_1.6 ------
public sealed class SendRequest : Activity
  {
    // შესატან და გამოსატან არგუმენტთა განსაზღვრა
    public InArgument<string> Title { get; set; }
    public InArgument<string> Author { get; set; }
    public InArgument<string> ISBN { get; set; }
    public OutArgument<ReservationResponse> Response { get; set; }

    public SendRequest()
    {
      // ცვლადების განსაზღვრა ამ პროცესისთვის
      Variable<ReservationRequest> request =
                  new Variable<ReservationRequest> { Name = "request"
    };
      Variable<string> requestAddress =
                        new Variable<string> { Name = "RequestAddress"
    };
    // გაგზავნის Send ქმედების განსაზღვრა
    Send submitRequest = new Send
     {
       ServiceContractName = "ILibraryReservation",
       EndpointAddress = new InArgument<Uri>
```

```
                (env => new Uri("http://localhost:" +
        requestAddress.Get(env) +
                    "/LibraryReservation")),
            Endpoint = new Endpoint
              {   Binding = new BasicHttpBinding()   },
            OperationName = "RequestBook", Content =
        SendContent.Create
                        (new InArgument<ReservationRequest>(request))
          };
          // SendRequest პროცესის განსაზღვრა
        }
    }
```

ამ ბიზნესპროცესს აქვს სამი შემავალი არგუმენტი: Sagani, Lector და Jgupi, რომლებიც განსაზღვრავს მოთხოვნილ სატრეინინგო კურსს. აგრეთვე აქვს გამომავალი არგუმენტი (Response), რომელიც არის საპასუხო შეტყობინება და იგი ადრე იყო განსაზღვრული TrainingResponse კლასით.

კონსტრუქტორი განსაზღვრავს აგრეთვე ზოგიერთ ლოკალურ ცვლადებს. მათი გადაცემა არ ხდება ბიზნესპროცესიდან (ან –ში), ისინი გამოიყენება შიგა სამუშაო პროცესების ქმედებებით. მოთხოვნილი ცვლადი შეიცავს გამომავალ შეტყობინებას, რომელიც არის TrainingRequest კლასი. RequestAddress ცვლადი შეინახავს პორტის ნომერს, რომელიც იმ ფილიალისაა, რომელიც ელოდება მოთხოვნის მიღებას.

შენიშვნა: კოდირებულ მუშა პროცესისთვის ყველა ქმედება შეიქმნა როგორც ჩადგმული ანონიმური კლასი. ეს კარგად მუშაობს ხშირ შემთხვევებში. Send ქმედება აქ შეიქმნა როგორც სახელმინიჭებული კლასი, ვინაიდან იგი საჭიროა მიმართვისათვის ReceiveReply ქმედებიდან. იგი არ განსხვავდება არგუმენტებისა და ცვლადებისგან, რომლებიც შევქმენით. ისინი

იქმნება სახელდებული კლასების სახით, რომლებიც მოგვიანებით იქნება გამოყენებული.

1.1.8. გაგზავნის ქმედება

submitRequest (მოთხოვნის გაგზავნის) ეგზემპლარი განისაზღვრება როგორც Send ქმედება. იგი იყენებს WCF-ს, რათა გადასცეს შეტყობინება მითითებულ ბოლო წერტილში.

ინფორმაციის სამი ნაწილი გამოიყენება ბოლო წერტილის მისათითებლად:

- ServiceContractName მიეთითება როგორც ITrainingProcess;
- EndpointAddress მიეთითება როგორც URL ცვლადით (პორტის ნომერი);
- Binding მიეთითება BasicHttpBinding კლასით.

ამასთანავე არსებობს კიდევ რამდენიმე თვისება, რომლებიც უნდა განისაზღვროს. OperationName მიუთითებს სერვისის კონტრაქტის სპეციფიკურ მეთოდზე, რომელიც გამოძახებულ უნდა იქნას დანიშნულების ადგილას შეტყობინების მიღების დროს.

Content თვისება შეინახავს მიმთითებელს შეტყობინებაზე (TrainingRequest კლასი), რომელიც უნდა გაიგზავნოს.

1.2. მომხმარებლის ქმედების დაპროგრამება (ლაბ-2)

მიზანი: მომხმარებლის ქმედების (activity) შექმნის პროცესის შესწავლა. ეს ქმედება ააგებს შეტყობინებას მოთხოვნაზე ბიზნეს–პროცესით გადმოცემული არგუმენტების საფუძველზე.

ჩვენ ვაგრძელებთ მუშაობას წინა პარაგრაფში შექმნილ აპლიკაციის კოდთან.

TrainingProcess პროექტის Solution Explorer–ში დავამატოთ კლასი: Add->Class სახელით **CreateRequest.cs**, რომლის რეალიზაცია მოცემულია 1.7 ლისტინგში.

```
//——— ლისტინგი_1.7 ——————————
using System;
using System.Activities;
using System.Configuration;
namespace TrainingProcess
{
 // ეს მომხმარებლის ქმედება ქმნის TrainingRequest კლასს შესატანი
 // პარამეტრებით (Sagani, Lector და Jgupi). ეს უზრუნველყოფილია
 // გამომავალი პარამეტრის მოთხოვნაში. იგი ასევე აბრუნებს ქსელის
 // მისამართს ფილიალისთვის, რომელსაც  უნდა გაეგზავნოს მოთხოვნა.
  public sealed class CreateRequest : CodeActivity
  {
    public InArgument<string> Sagani { get; set; }
    public InArgument<string> Lector { get; set; }
    public InArgument<string> Jgupi { get; set; }
    public OutArgument<TrainingRequest> Request { get; set; }
    public OutArgument<string> RequestAddress { get; set; }

    protected override void Execute(CodeActivityContext context)
      {
        // config ფაილის გახსნა და Request Address მიღება (get)
```

```
Configuration config = ConfigurationManager
        .OpenExeConfiguration(ConfigurationUserLevel.None);
AppSettingsSection app =
          (AppSettingsSection)config.GetSection("appSettings");
//  TrainingRequest კლასის შექმნა და მისი შევსება შესატანი
 // არგუმენტებით
TrainingRequest  r = new TrainingRequest
    (
     Sagani.Get(context),
      Lector.Get(context),
      Jgupi.Get(context),
      new Branch
        {
          BranchName = app.Settings["Branch Name"].Value,
          BranchID = new Guid(app.Settings["ID"].Value),
          Address = app.Settings["Address"].Value
        }
       );
  // მოთხოვნის მოთავსება OutArgument – ში
  Request.Set(context, r);
  // address–ის მოთავსება OutArgument –ში
RequestAddress.Set(context, app.Settings["Request
    Address"].Value);
     }
    }
  }
```

Execute() მეთოდი პირველად ხსნის აპლიკაციის კონფიგ–ფაილს ფილიალის დეტალების დასადგენად, რომელსაც ესაჭიროება მოთხოვნის ფორმატირება. შემდეგ იგი ქმნის TrainingRequest კლასს ერთ-ერთი ჩვენ მიერ უზრუნველყოფილი კონსტრუქტორით. Branch კლასი, რომელიც არის კონსტრუქტორის

ერთ-ერთი პარამეტრი, შექმნილია როგორც ანონიმური კლასი BranchName, BranchID და Address თვისებების მითითებით. შემდეგ TrainingRequest კლასი შეინახავს მოთხოვნის გამომავალ პარამეტრში.

კონფიგურაციის ფაილში Request Address შეიცავს ფილიალის მისამართს (პორტის ნომერს), რომელმაც უნდა მიიღოს მოთხოვნა. იგი შეინახება RequestAddress გამომავალ არგუმენტში, ვინაიდან ის დასჭირდება სამუშაო პროცესს.

დავუბრუნდეთ TrainingWF.cs ფაილს. დავამატოთ ბიზნეს–პროცესის განსაზღვრება 1.8 ლისტინგის შესაბამისად. ეს უნდა ჩაჯდეს იქ სადაც მითითებულია (// SendRequest სამუშაო პროცესის განსაზღვრა).

```
// ——— ლისტინგი_1.8 ———
// Define the SendRequest workflow
this.Implementation = () => new Sequence
{
  DisplayName = "SendRequest", Variables = { request,
requestAddress},
Activities =
     {
     new CreateRequest
      {
       Sagani = new InArgument<string>(env => Sagani.Get(env)),
       Lector = new InArgument<string>(env => Lector.Get(env)),
       Jgupi = new InArgument<string>(env => Jgupi.Get(env)),
       Request = new OutArgument<TrainingRequest>
            (env => request.Get(env)),
       RequestAddress = new OutArgument<string>
            (env => requestAddress.Get(env))
```

```
        },
      new CorrelationScope
        {
          Body = new Sequence
            {
             Activities = {  submitRequest,
              new WriteLine
                 {
                   Text = new InArgument<string>
                     (env => "Request sent; waiting for response"),
                 },
            new ReceiveReply
              {
               Request = submitRequest,
               Content = ReceiveContent.Create
                    (new OutArgument<ReservationResponse>
                       (env => Response.Get(env)))
              }
             }
          }
        },
       new WriteLine
        {
          Text = new InArgument<string>
             (env => "Response received from " +
                  Response.Get(env).Provider.BranchName),
        },
      }
};
```

ქმედების კლასის Implementation თვისება (მითითებით როგორც this. Implementation) შეიცავს შვილ ქმედებას. ამ შემთხვევაში იგი განისაზღვრება როგორც Sequence ქმედება, რომელიც შედგება შვილ ქმედებათა ერთობლიობისგან. ამ ქმედებათა მიერ გამოყენებული ცვლადები უნდა იყოს გამოცხადებული. ესაა მოთხოვნა და requestAddress ცვლადები, რომლებიც განისაზღვრა კონსტრუქტორში.

პირველი ქმედება არის მომხმარებლის CreateRequest ქმედება, რომელიც ახლახანს ავაგეთ. მივაქციოთ ყურადღება, რომ როგორც ჩვენ მივუთითეთ თვისება, ეს იცის Intellisense–მ (შესატანი და გამოსატანი არგუმენტები, რომლებიც განვსაზღრეთ ჩვენს კლასში). Sagani, Lector და Jgupi არის სამუშაო პროცესის შემავალი არგუმენტები. Request და RequestAddress გამომავალი არგუმენტები ინახება სამუშაო პროცესის ცვლადებში.

CorrelationScope ქმედება ამატებს შემდეგს (next), რომელიც შედგება ქმედებათა მიმდევრობისგან. კერძოდ, ის შეიცავს Send და ReceiveReply ქმედებებს, რომელთა მოთავსებით CorrelationScope ქმედებაში შესაძლებელი ხდება შემოსული მოთხოვნისა და საპასუხო შეტყობინების კორელაციის განსაზღვრა მოცემული ბიზნესპროცესისთვის.

WriteLine ქმედება ემატება Send ქმედების შემდეგ, რათა მიეთითოს, რომ მოთხოვნა იქნა გაგზავნილი.

1.2.1. პასუხის მიღების ქმედება

ReceiveReply ქმედება ასოცირდება გაგზავნის Send ქმედებასთან. იგი ელოდება პასუხს შეტყობინებაზე, რომელიც გაიგზავნა Send ქმედების მიერ. ამის რეალიზაციის მიზნით Request–ის (მოთხოვნის) თვისებას აქვს Send ქმედების სახელმინიჭებული ეგზემპლარის მნიშვნელობა (submitRequest).

თვისების შინაარსი განსაზღვრავს თუ სად ინახება საპასუხო შეტყობინება (TrainingResponse კლასი). ამით ყენდება სამუშაო ნაკადის Response გამომავალი არგუმენტი. იგი მისაწვდომი იქნება ჰოსტ–აპლიკაციისთვის როცა ვორკფლოვ–პროცესი დასრულდება.

1.2.2. სერვერი – მოთხოვნის დამუშავება

განვსაზღვროთ სამუშაო პროცესი, რომელიც სრულდება სრევერზე ფილიალიდან მიღებული მოთხოვნის დასამუშავებლად.

TrainingWF.cs ფაილში დავამატოთ კლასის განსაზღვრა:

```
// ––– ლისტინგი_1.9 ––––––––––––––––––––––––––
public sealed class ProcessRequest : Activity
{
   public ProcessRequest()
   { // ცვლადების განსაზღვრა ამ workflow–ისთვის
      Variable<TrainingRequest> request =
                  new Variable<TrainingRequest> { Name = "request"
      };
      Variable<TrainingResponse> response =
               new Variable<TrainingResponse> { Name = "response"
      };
Variable<bool> reserved = new Variable<bool> { Name = "reserved"
      };
Variable<CorrelationHandle> requestHandle =
        new Variable<CorrelationHandle> { Name = "RequestHandle"
      };
      // Receive ქმედების შექმნა
      Receive receiveRequest = new Receive
          {
              ServiceContractName = "ITrainingProcess",
              OperationName = "RequestTraining",
```

```
                CanCreateInstance = true,
                Content = ReceiveContent.Create
                    (new OutArgument<TrainingRequest>(request)),
                CorrelatesWith = requestHandle
            };
        // ProcessRequest workflow–ის განსაზღვრა
    }
}
```

ამ სამუშაო პროცესს არ აქვს შემავალი და გამომავალი არგუმენტები. აქვს ოთხი ცვლადი, რომლებიც განსაზღვრულია. მოთხოვნის (request) ცვლადი ინახავს შემავალ შეტყობინებებს (TrainingRequest კლასი), ხოლო პასუხის (response) ცვლადი ინახავს გამომავალ პასუხებს (TrainingResponse კლასი). დარეზერვებული ცვლადი მიუთითებს შეიძლება თუ არა კურსის საგანი (sagani) იყოს დაჯავშნული, თუ არა. requestHandle გამოიყენება შემოსული მოთხოვნის და პასუხის კორელაციისთვის.

1.2.3. მოთხოვნის მიღების ქმედება

Receive ქმედების სახელმინიჭებული ეგზემპლარი (receiveRequest – მოთხოვნის მიღება) განსაზღვრულია. არაა საჭირო მიერთების ან მისამართის მითითება WCF შეტყობინების მიღების ბოლოს. მაგრამ უნდა განისაზღვროს სერვისის კონტრაქტი. ServiceContractName მიუთითებს, რომ ITrainingProcess სერვისის კონტრაქტი უნდა იქნას გამოყენებული და OperationName თვისება განსაზღვრავს RequestTraining() მეთოდს.

CanCreateInstance დაყენებულია true -ში, ვინაიდან როცა ეს ქმედება სრულდება, იგი შექმნის პროცესის ახალ ეგზემპლარს. იგი მოითხოვს, რომ ეს ქმედება იყოს პირველი სამუშაო პროცესში. Content თვისება უნდა შეიცავდეს შემავალ შეტყობინებას და

კონფიგურირდება ისე, რომ შეინახოს იგი მოთხოვნის ცვლადში. CorrelatesWith თვისება იყენებს requestHandle ცვლადს.

1.2.4. მომხმარებლის ქმედება – პასუხის შექმნა

სანამ განვსაზღვრავთ სამუშაო პროცესის ქმედებებს, საჭიროა მომხმარებლის ქმედება TrainingResponse კლასის შექმნისათვის. TrainingProcess პროექტის Solution Explorer–დან დავამატოთ ახალი კლასი სახელით CreateResponse.cs, რომლის რეალიზაცია მოცემულია 1.10 ლისტინგზე.

```
// —— ლისტინგი_1.10 ——————
using System;
using System.Activities;
using System.Configuration;
namespace TrainingProcess
{
// ეს custom ქმედება ქმნის ReservationResponse კლასს. საწყისი
// მოთხოვნა უზრუნველყოფილია როგორც InArgument, ასევე
// ლოგიკური (boolean) მითითებით, რომ მოთხოვნა დაკმაყო-
ფილდა
// თუ არა. კლასი უზრუნველყოფილია Response–ში OutArgument-
ით
public sealed class CreateResponse : CodeActivity
{
  public InArgument<TrainingRequest> Request { get; set; }
  public InArgument<bool> Reserved { get; set; }
  public OutArgument<TrainingResponse> Response { get; set; }

  protected override void Execute(CodeActivityContext context)
   {
    // config ფაილის გახსნა
```

```
Configuration config = ConfigurationManager
      .OpenExeConfiguration(ConfigurationUserLevel.None);
AppSettingsSection app =
      (AppSettingsSection)config.GetSection("appSettings");
// TrainingResponse კლასის შექმნა და შევსება
TrainingResponse r = new TrainingResponse
   (   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);
       }
      }
   }
```

CreateResponse ქმედება ძალზე ჰგავს CreateRequest-ეს. ჯერ იგი ხსნის აპლიკაციის კონფიგ–ფაილს, რათა მიიღოს ფილიალის შესახებ დეტალები. შემდეგ TrainingResponse კლასი იქმნება ერთ-ერთი მოცემული კონსტრუქტორით და შეინახება Response გამომავალ არგუმენტში.

დავბრუნდეთ TrainingWF.cs კლასში შევიტანოთ სამუშაო პროცესის განსაზღვრება, როგორც 1.11 ლისტინგშია (იგი უნდა ჩაემატოს //---სამუშაო პროცესის განსაზღვრა ProcessRequest--- ში).

```
// ——— ლისტინგი_1.11 ——————————
// ProcessRequest workflow-ის განსაზღვრა
```

```
this.Implementation = () => new Sequence
{   DisplayName = "ProcessRequest",
    Variables = { request, response, reserved, requestHandle },
    Activities =
    { receiveRequest,
      new WriteLine
       {
       Text = new InArgument<string>(env => "Got request from: " +
            request.Get(env).Requester.BranchName),
       },
      new WriteLine
       {      Text = new InArgument<string>(env => "Requesting: " +
              request.Get(env).Title),
        },
      new Assign
       {     To = new OutArgument<Boolean>(reserved),
             Value = new InArgument<Boolean>(env => true)
       },
      new Delay
       {
              Duration = TimeSpan.FromSeconds(2)
       },
      new CreateResponse
        {    Request = new InArgument<ReservationRequest>(env =>
                  request.Get(env)),
             Response = new OutArgument<ReservationResponse>
                  (env => response.Get(env)),
             Reserved = new InArgument<bool>(env => reserved.Get(env)),
       },
      new WriteLine
```

```
    {    Text = new InArgument<string>(env => "Sending response to: " +
             request.Get(env).Requester.BranchName),
    },
    new SendReply
     {   Request = receiveRequest,
         Content = SendContent.Create
             (new InArgument<ReservationResponse>(response))
      }
    }
  };
```

ეს ოთხი ცვლადი, რომლებიც კონსტრუქტორშია განსაზღვრული, გამოცხადებულია სამუშაო პროცესის კოდის ტანში. Receive ქმედება (receiveRequest) არის პირველი ქმედება მუშა პროცესში. იგი, ორ WriteLine ქმედებასთან თანხლებით: პირველს ეკრანზე გამოაქვს ფილიალის სახელი, რომელმაც მოთხოვნა გამოაგზავნა, და მეორე გვიჩვენებს საგნის დასახელებას (sagani), რომელიც მოითხოვება.

Assign ქმედება უბრალოდ აყენებს დარეზერვებულ ცვლადებს true-ში. ამ მაგალითში ჩვენ დავუშვით, რომ საგნის დასახელება არის მიწვდომადი. Delay ქმედება აყოვნებს სამუშაო პროცესს 2 წამით (დამუშავების პროცესის იმიტაცია). შემდეგ მომხმარებლის CreateResponse ქმედება სრულდება TrainingResponse კლასის შექმნის მიზნით, რომელიც შენახულ იქნება response ცვლადში. ბოლო WriteLine ქმედება მიუთითებს, რომ პასუხი გაიგზავნა.

1.2.5. SendReply ქმედება

SendReply ქმედება ასოცირდება (კავშირშია) Receive ქმედებასთან. ეს ხორციელდება Request თვისების მითითებით როგორც მაჩვენებლისა Receive ქმედებაზე (receiveRequest).

თვისების შინაარსი განსაზღვრავს შეტყობინებას, რომელიც იქნება გაგზავნილი უკან საპასუხოდ. ესაა პასუხის ცვლადის მნიშვნელობა.

ჩვენი სამუშაო პროცესი დასრულდა. საფინალო რეალიზაცია TrainingWF.cs ფაილისთვის მოცემულია 1.12 ლისტინგში.

// ── ლისტინგი_1.12 ──────────

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Activities;
using System.Activities.Statements;
using System.ServiceModel.Activities;
using System.ServiceModel;

namespace TrainingProcess
{
  public sealed class SendRequest : Activity
   {
     // შესატან და გამოსატან არგუმენტთა განსაზღვრა
    public InArgument<string> Sagani { get; set; }
    public InArgument<string> Lector { get; set; }
    public InArgument<string> Jgupi { get; set; }
     public OutArgument<TrainingResponse> Response
{get;set;}

  public SendRequest()
  {
   // ცვლადების განსაზღვრა ამ პროცესისთვის
   Variable<TrainingRequest> request = new
           Variable<TrainingRequest> { Name = "request" };
   Variable<string> requestAddress = new Variable<string>
{
           Name = "RequestAddress" };

   // გაგზავნის Send ქმედების განსაზღვრა
    Send submitRequest = new Send
      {
```

```
 ServiceContractName = "ITrainingProcess",
 EndpointAddress = new InArgument<Uri>
       (env => new Uri("http://localhost:" +
     requestAddress.Get(env) + "/TrainingProcess")),
 Endpoint = new Endpoint
    {
        Binding = new BasicHttpBinding()
    },
 OperationName = "RequestTraining",
 Content = SendContent.Create(new
          InArgument<TrainingRequest>(request))
};
//  SendRequest პროცესის განსაზღვრა
// ----- ლისტინგი_1.8 ----
this.Implementation = () => new Sequence
 {
  DisplayName = "SendRequest",
  Variables = { request, requestAddress },
 Activities = { new CreateRequest {
     Sagani = new InArgument<string>(env =>
                 Sagani.Get(env)),
     Lector = new InArgument<string>(env =>
                 Lector.Get(env)),
     Jgupi = new InArgument<string>(env =>
                 Jgupi.Get(env)),
 Request = new OutArgument<TrainingRequest> (env =>
                 request.Get(env)),
     RequestAddress = new OutArgument<string>(env =>
            requestAddress.Get(env))
 },
new CorrelationScope
{
  Body = new Sequence
   {
    Activities = { submitRequest,  new WriteLine
     {
       Text = new InArgument<string>(env =>
           "Request sent; waiting for response"),
     },
new ReceiveReply
```

```
        {
          Request = submitRequest,
          Content = ReceiveContent.Create(new
                      OutArgument<TrainingResponse>
                (env => Response.Get(env)))
        }
       }
      }
    },
    new WriteLine
     {
      Text = new InArgument<string>
      (env => "Response received from " +
             Response.Get(env).Provider.BranchName),
         },
       }
      };
     }
   }
public sealed class ProcessRequest : Activity
 {
  public ProcessRequest()
  { // ცვლადების განსაზღვრა ამ workflow-ისთვის
    Variable<TrainingRequest> request =
       new Variable<TrainingRequest> { Name = "request" };
   Variable<TrainingResponse> response =
      new Variable<TrainingResponse> { Name = "response"};
   Variable<bool> reserved =
         new Variable<bool> { Name = "reserved" };
   Variable<CorrelationHandle> requestHandle =
         new Variable<CorrelationHandle> { Name =
                "RequestHandle" };

// Receive ქმედების შექმნა
Receive receiveRequest = new Receive
 {
   ServiceContractName = "ITrainingProcess",
   OperationName = "RequestTraining",
   CanCreateInstance = true,
   Content = ReceiveContent.Create
```

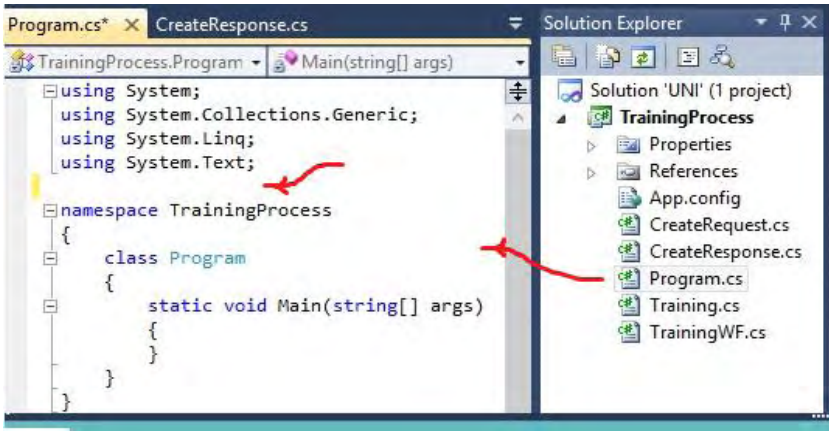
```
            (new OutArgument<TrainingRequest>(request)),
   CorrelatesWith = requestHandle
 };

 //  ProcessRequest workflow-ის განსაზღვრა
this.Implementation = () => new Sequence
{
 DisplayName = "ProcessRequest",
 Variables = { request, response, reserved, requestHandle
},
 Activities =
  {
    receiveRequest,
    new WriteLine
     {
Text = new InArgument<string>(env=> "Got request from: " +
                  request.Get(env).Requester.BranchName),
     },
    new WriteLine
      {
    Text = new InArgument<string>(env => "Requesting: " +
                    request.Get(env).Sagani),
     },
  new Assign
   {
     To = new OutArgument<Boolean>(reserved),
     Value = new InArgument<Boolean>(env => true)
   },
  new Delay
   {
     Duration = TimeSpan.FromSeconds(2)
   },
  new CreateResponse
   {
     Request = new InArgument<TrainingRequest>(env =>
                  request.Get(env)),
     Response = new OutArgument<TrainingResponse>
                  (env => response.Get(env)),
     Reserved = new InArgument<bool>(env =>
              reserved.Get(env)),
    },
```

```
  new WriteLine
    {
Text=new InArgument<string>(env=>"Sending response to: "+
                request.Get(env).Requester.BranchName),
    },
  new SendReply
    {
      Request = receiveRequest,
      Content = SendContent.Create
            (new InArgument<TrainingResponse>(response))
               }
             }
          };
       }
    }
 }
```

1.3. Host-აპლიკაციის რეალიზაცია (ლაბ-3)

მიზანი: ასაგები სისტემის ჰოსტ–აპლიკაციის რეალიზაციის საფუძვლების შესწავლა.

ბოლო ბიჯი პროექტის ასაგებად არის ჰოსტ–აპლიკაციის რეალიზაცია. უნდა გამოვიყენოთ კონსოლის აპლიკაცია (Program.cs), რომელიც გენერირებულ იქნა შაბლონის (template) სახით (ნახ.1.5).



ნახ.1.5

აპლიკაცია ასრულებს მოთხოვნის ინიციალიზებას და დამუშავებას, ამიტომ საჭირო იქნება ორივესთვის ლოგიკის დაწერა. ჯერ უნდა აეწყოს აპლიკაცია მოთხოვნების მისაღებად და დასამუშავებლად, შემდეგ კი უნდა მოხდეს ახალი მოთხოვნის ინიცირება და გაგზავნა სხვა აპლიკაციისთვის.

დავამატოთ Program.cs ფაილში რამდენიმე სახელსივრცე:

```
using System.ServiceModel;
using System.ServiceModel.Activities;
using System.ServiceModel.Activities.Description;
using System.ServiceModel.Description;
```

```
using System.Activities;
using System.Xml.Linq;
using System.Configuration;
```

1.3.1. ServiceHost კლასი და WorkflowServiceHost –ის შექმნა

შემავალი მოთხოვნების მისაღებად გამოიყენება ServiceHost კლასი, რომელიც WCF ტექნოლოგიაშია. WF 4.0 უზრუნველყოფს WorkflowServiceHost კლასს, რომელიც ახდენს ServiceHost-ის რეალიზაციას, ოღონდ აინიცირებს სამუშაო პროცესს, როცა შეტყობინება მიღებულია.

შვიტანოთ 1.13 ლისტინგის კოდი როგორც main() ფუნქციის რეალიზაცია Program კლასისთვის.

```
// --ლისტინგი 1-13. ნაწილობრივი რეალიზაცია  main() ფუნქციის –
//  config ფაილის გახსნა და  ფილიალის სახელის (name) და
// ქსელის მისამართის (adress) მიღება
Configuration config = ConfigurationManager
      .OpenExeConfiguration(ConfigurationUserLevel.None);
AppSettingsSection app =
(AppSettingsSection)config.GetSection("appSettings");

string adr = app.Settings["Address"].Value;
Console.WriteLine(app.Settings["Branch Name"].Value);

// სერვისის შექმნა შემოსული მოთხოვნების დასამუშავებლად
WorkflowService service = new WorkflowService
 {
   Name = "LibraryReservation",
   Body = new ProcessRequest(),
   Endpoints =
     {
```

```
        new Endpoint
        {
            ServiceContractName = "ILibraryReservation",
            AddressUri = new Uri("http://localhost:" + adr +
                "/LibraryReservation"),
            Binding = new BasicHttpBinding(),
        }
    }
};
// WorkflowServiceHost –ის შექმნა შემოსული მოთხოვნების მისაღებად
System.ServiceModel.Activities.WorkflowServiceHost wsh =
        new
System.ServiceModel.Activities.WorkflowServiceHost(service);
wsh.Open();
```

ეს კოდი ჯერ ხსნის აპლიკაციის კონფიგურაციის ფაილს და იღებს მისამართის პარამეტრებს. ის განსაზღვრავს პორტის ნომერს, რომლითაც აპლიკაცია იღებს შემავალ მოთხოვნას. აგრეთვე ღებულობს ფილიალის სახელს, რომელიც ეკრანზე გამოიტანება. ვინაიდან ჩვენ საქმე გვაქვს რამდენიმე აპლიკაციასთან, ეს მექანიზმი საშუალებას მოგვცემს თვალყური ვადევნოთ თუ რომელი რომელია.

1.3.2. სერვისი - WorkflowService კლასი

1.13 ლისტინგის ბოლოში ნაჩვენებია WorkflowService კლასის შქმნის კოდი. Body თვისებისთვის ის იყენებს ProcessRequest კლასის ახალ ეგზემპლარს, რომელიც განსაზღვრავს პროცესს შემავალი მოთხოვნების დასამუშავებლად.

1.3.3. ბოლო წერტილი

სერვისის კლასი განსაზღვრავს აგრეთვე ბოლო წერტილს (Endpoint), კონტრაქტის ITrainingProcess სერვისის გამოყენებით, URL–ით, რომელიც შეიცავს პორტის ნომრის ცვლადს და BasicHttpBinding კლასს. დასასრულ, WorkflowServiceHost კლასი კონკრეტიზირდება განსაზღვრული სერვისკლასის გამოყენებით. შემდეგ ის იხსნება Open() მეთოდის გამოძახებით. ამ მომენტში აპლიკაცია უთვალთვალებს შემავალ შეტყობინებებს. როცა ის მიღებულ იქნება, ProcessRequest მუშა პროცესის ეგზემპლარი ამუშავდება მოთხოვნის დასამუშავებლად.

1.3.4. სამუშაო პროცესის გამომძახებელი [WorkflowInvoker]

ახლა საჭიროა კოდის დამატება მოთხოვნის ინიციალიზებისათვის. შევიტანოთ 1.14 ლისტინგის კოდი, ოღონდაც wsh.Open() მეთოდის გამოძახების შემდეგ.

```
// ლისტინგი 1-14.  main() ფუნქციის რეალიზების დარჩენილი ნაწილი – – –
Console.WriteLine
  ("Waiting for requests, press ENTER to send a
request.");
Console.ReadLine();

//ლექსიკონის შექმნა შემავალი არგუმენტებით სამუშაო
პროცესისთვის-            IDictionary<string, object> input =
            new Dictionary<string, object>
    {
      { "Sagani" , "Distributed DBS Management" },
      { "Lector", "Gia Surguladze" },
      { "Jgupi", "151100108350" }
    };

//  SendRequest სამუშაო პროცესის გამოძახება
IDictionary<string, object> output =
         WorkflowInvoker.Invoke(new SendRequest(), input);
TrainingResponse resp =
        (TrainingResponse)output["Response"];
```

```
// პასუხის ეკრანზე გამოტანა
Console.WriteLine("Response received from the {0} branch",
                  resp.Provider.BranchName);
Console.WriteLine();
Console.WriteLine("Press ENTER to exit");
Console.ReadLine();
// WorkflowServiceHost-ის დახურვა
wsh.Close();
```

ეს კოდი იცდის, სანამ მომხმარებელი არ დააჭერს Enter ღილაკს, რომელიც მოგვცემს დროს, რათა მივიღოთ რამდენიმე მუშა კოპიო და თვალყური ვადევნოთ შემოსულ შეტყობინებებს.

ჯერ იქმნება ლექსიკონი შემავალი არგუმენტებისთვის. შემდეგ ის იყენებს WorkflowInvoker კლასის Invoke() მეთოდს, რათა დაწყებულ იქნას SendRequest სამუშაო პროცესის ახალი ეგზემპლარი.

Response პასუხის გამომავალი არგუმენტები ამოიღება ლექსიკონიდან, რომელიც დაბრუნდება უკან როცა მუშა პროცესი დასრულდება. დასასრულ, WorkflowServiceHost დაიხურა მუშა პროცესის დასრულებამდე.

ქვემოთ მოცემულია მთლიანი Program.cs კოდის ლისტინგი:

```
// – – – ლისტინგი_1.15 – – – Program.cs  კოდი – – –
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.ServiceModel;
    using System.ServiceModel.Activities;
    using System.ServiceModel.Activities.Description;
    using System.ServiceModel.Description;
    using System.Activities;
    using System.Xml.Linq;
   using System.Configuration;
```

```
// config ფაილის გახსნა და ფილიალის სახელის (name) და
// ქსელის მისამართის (adress) მიღება
Configuration config = ConfigurationManager
        .OpenExeConfiguration(ConfigurationUserLevel.None);
AppSettingsSection app =
         (AppSettingsSection)config.GetSection("appSettings");
string adr = app.Settings["Address"].Value;
Console.WriteLine(app.Settings["Branch Name"].Value);
// სერვისის შექმნა შემოსული მოთხოვნების დასამუშავებლად
  WorkflowService service = new WorkflowService
    {
      Name = "TrainingProcess",
      Body = new ProcessRequest(),
      Endpoints =
       {
       new Endpoint
        {
        ServiceContractName="ITrainingProcess",
          AddressUri = new Uri("http://localhost:" + adr +
                              "/TrainingProcess"),
          Binding = new BasicHttpBinding(),
        }
       }
      };
//  WorkflowServiceHost –ის შექმნა შემოსული მოთხოვნების მისაღებად
System.ServiceModel.Activities.WorkflowServiceHost wsh =
      new System.ServiceModel.Activities.WorkflowServiceHost(service);

wsh.Open();

Console.WriteLine
        ("Waiting for requests, press ENTER to send a request.");
Console.ReadLine();
// ლექსიკონის შექმნა შემავალი არგუმენტებით სამუშაო პროცესისთვის
 IDictionary<string, object> input = new Dictionary<string, object>
```

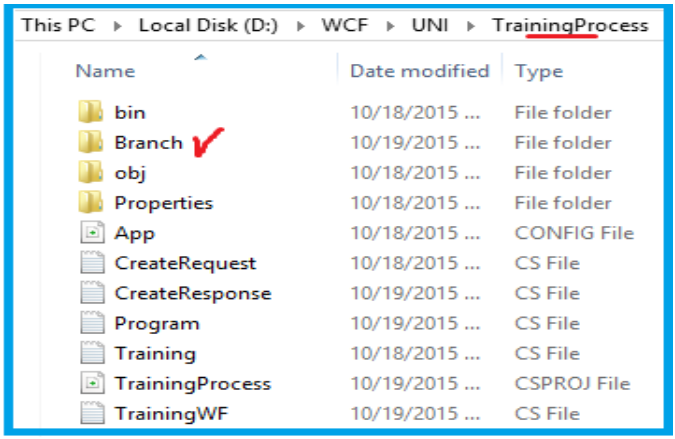
```
 {
    { "Sagani", "Distributed DBS Management" },
    { "Lector", "Gia Surguladze" },
    { "Jgupi", "151100108350" }
 };
 // SendRequest სამუშაო პროცესის გამოძახება
IDictionary<string, object> output =
            WorkflowInvoker.Invoke(new SendRequest(), input);
TrainingResponse resp=
(TrainingResponse)output["Response"];

 // პასუხის ეკრანზე გამოტანა
 Console.WriteLine("Response received from the {0} branch",
                   resp.Provider.BranchName);
 Console.WriteLine();
 Console.WriteLine("Press ENTER to exit");
 Console.ReadLine();
   // WorkflowServiceHost დახურვა
  wsh.Close();
 }
 }
}
```

1.3.5. აპლიკაციათა ამუშავება

F6-ით ავამუშავოთ აპლიკაცია და გავმართოთ კოდი. ჩვენ შგვიძლია აპლიკაციის ორი ეგზემპლარის ამუშავება და მათ დასჭირდება განსხვავებული კონფიგურაციის ფაილები. ამიტომაც შეიძლება მათთვის შერჩეულ იქნას სხვადასხვა პორტის ნომრები.

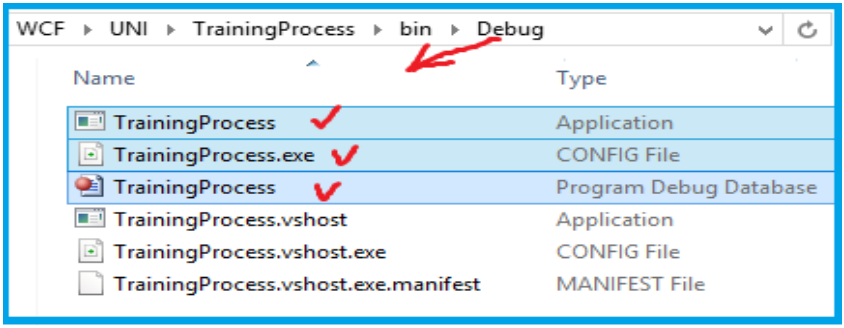
1.3.6. ტრეინინგ-ცენტრის ფილიალის კონფიგურირება

გავხსნათ Windows Explorer და ჩვენი პროექტის TrainingProcess ფოლდერში ჩავამატოთ ახალი ქვეკატალოგი Branch (ფილიალი) - იხ. ნახ.1.6.



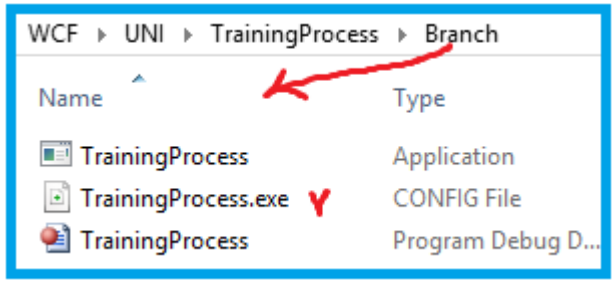
ნახ.1.6

Debag-კატალოგში მოვნიშნოთ სამი ფაილი (ნახ.1.7) და კოპირებით გადავიტანოთ Branch ფოლდერში.



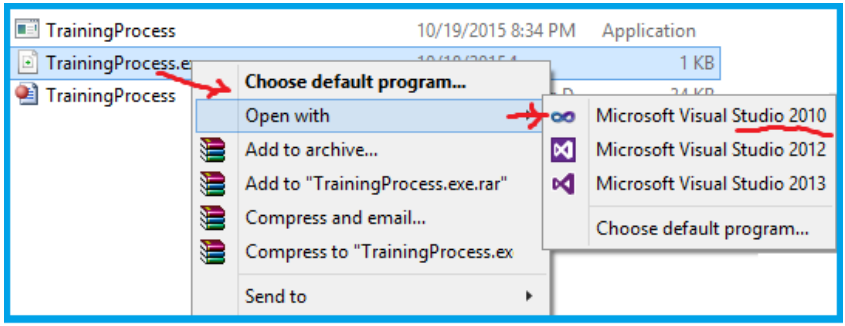
ნახ.1.7

კოპირების შემდეგ Branch–ში მივიღებთ:



ნახ.1.8

გავხსნათ TrainingProcess.exe.config ფაილი რომელიმე ტექსტურ რედაქტორში, მაუსის მარჯვენა ღილაკით Open with… და მაგალითად, Visual Studio 2010 – ში (ნახ.1.9).



ნახ.1.9

```
<!– ლისტინგი_1.16 ——— –>
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
 <appSettings>
   <add key="Branch Name" value="TrainingCenter"/>
   <add key="ID" value="{43E6DADD-4751-4056-8BB7-
                                        7459B5C361AB}"/>
    <add key="Address" value="8000"/>
    <add key="Request Address" value="8730"/>
  </appSettings>
```

```
</configuration>
```

შევცვალოთ კონფიგურაციის ფაილის ტექსტი შემდეგი კოდის 1.17_ლისტინგით.

```
<!– ლისტინგი_1.17 ——— –>
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="Branch Name" value="TrainingBranch_GTU"/>
    <add key="ID" value="{43E6DADD-4751-4056-8BB7-
                                        7459B5C361AB}"/>
    <add key="Address" value="8730"/>
    <add key="Request Address" value="8000"/>
  </appSettings>

</configuration>
```

დავაკვირდეთ, რომ პორტის ნომრები მისამართისა და მოთხოვნისთვის წესრიგშია (ტრაინინგ-ცენტრის და ფილიალის პორტის ნომრები განსხვავებულია). ავამუშავოთ .exe ფაილი. კონსოლზე მივიღებთ ასეთ ტექსტს:

```
TrainingBranch_GTU
Waiting for requests, press ENTER to send a request.
```

1.3.7. აპლიკაციათა მუშაობის მოსალოდნელი შედეგები

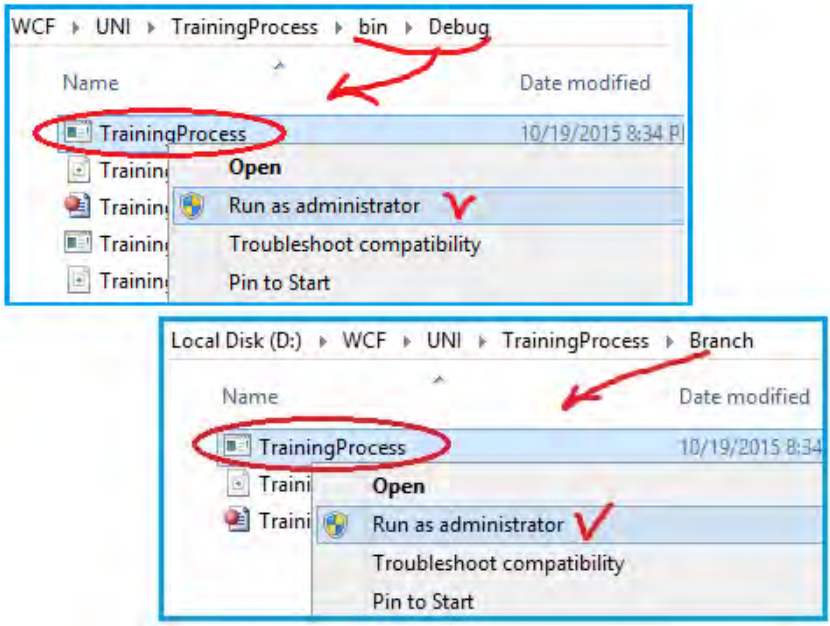
Visual Studio-დან F5–ით ავამუშავოთ აპლიკაცია. ამ დროს უნდა გაიხსნას მეორე კონსოლის ფანჯარა ასეთი ტექსტით:

```
TrainingCenter
Waiting for requests, press ENTER to send a request.
```

შესაძლებელია ფანჯრების გადაადგილება ისე, რომ ორივე ჩანდეს. ავირჩიოთ ერთ-ერთი და დავაჭიროთ Enter-ს. რამდენიმე წამის შემდეგ უნდა გამოჩნდეს შედეგები.

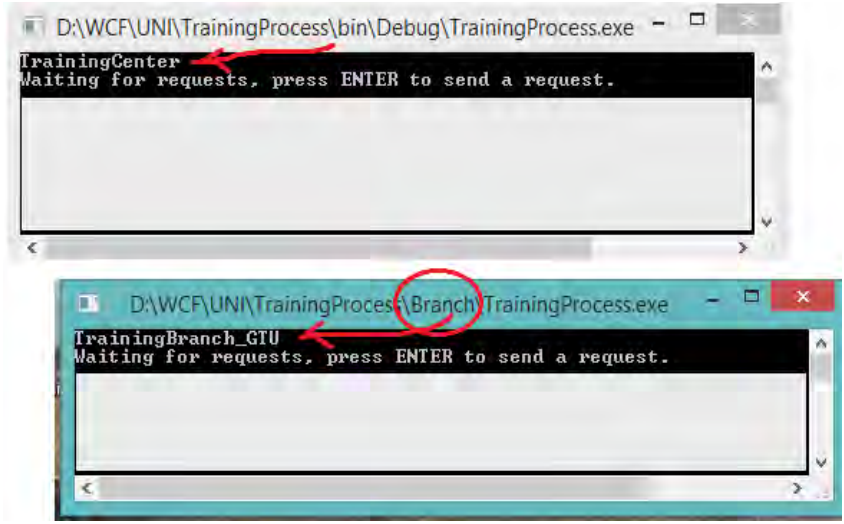
თუ შედეგები არ მიიღება და პროგრამა ავარიულად ჩერდება, შესაძლებელია მოისინჯოს ორივე აპლიკაციის (TrainingCenter და Branch) ამუშავება ადმინისტრატორის უფლებით.

მაგალითად, 1.10 ნახაზზე ნაჩვენებია ტრეინინგ-ცენტრის და მისი ფილიალის მხრიდან, შესაბამისად, bin->Debag და Branch-კატალოგებში ადმინისტრატორის უფლებით ორივე აპლიკაციის ამოქმედება.



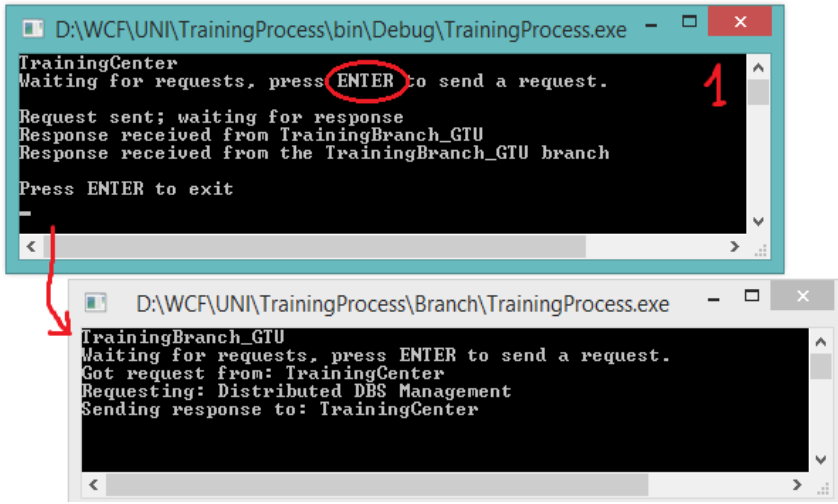
ნახ.1.10. ადმინისტრატორის უფლების გამოყენება

1.11-ა ნახაზზე ნაჩვენებია ორივე აპლიკაციის საწყისი მდგომარეობა. პირველ სტრიქონში ჩანს ფანჯრის მიკუთვნება ცენტრზე ან ფილიალზე.



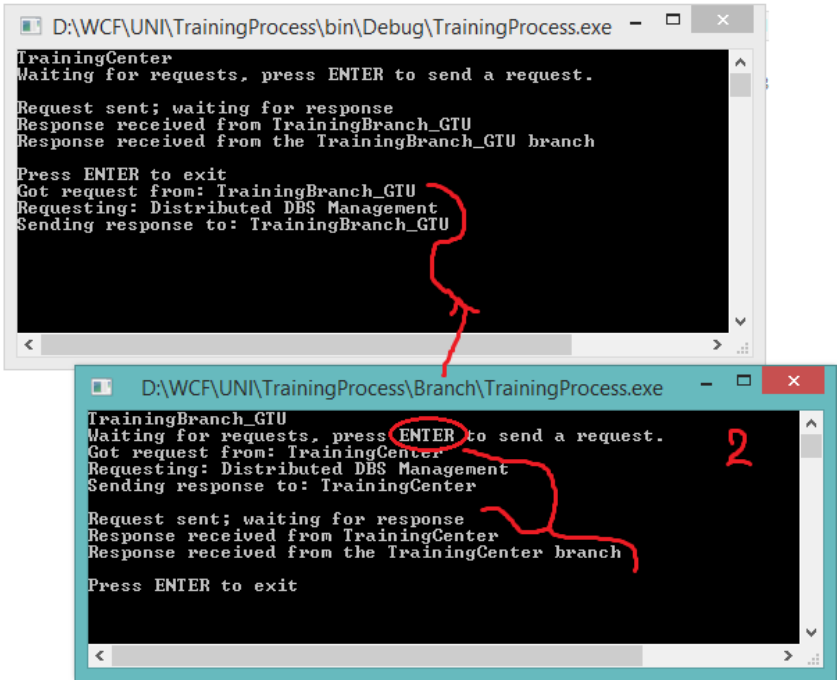
ნახ.1.11-ა

პირველ ფანჯარაში Enter–ღილაკის დაჭერით მივიღებთ ასეთ შედეგს ტექსტებით:



ნახ.1.11-ბ. აქტიურია 1-ფანჯარა (TrainingCenter)

ახლა მეორე ფანჯარაში ვაჭერთ Enter–ღილაკს, რომლის შედეგები ასეთია:



ნახ.1.11-გ. აქტიურია 2-ფანჯარა (TrainingBranch_GTU)

როგორც ვხედავთ, განხორციელდა ინფორმაციის გაცვლა ცენტრსა და მის ფილიალს შორის.

Enter–ღილაკით შეიძლება დაიხუროს აპლიკაციები.

II თავი
კომუნიკაცია Host–აპლიკაციასთან WPF-ის ბაზაზე

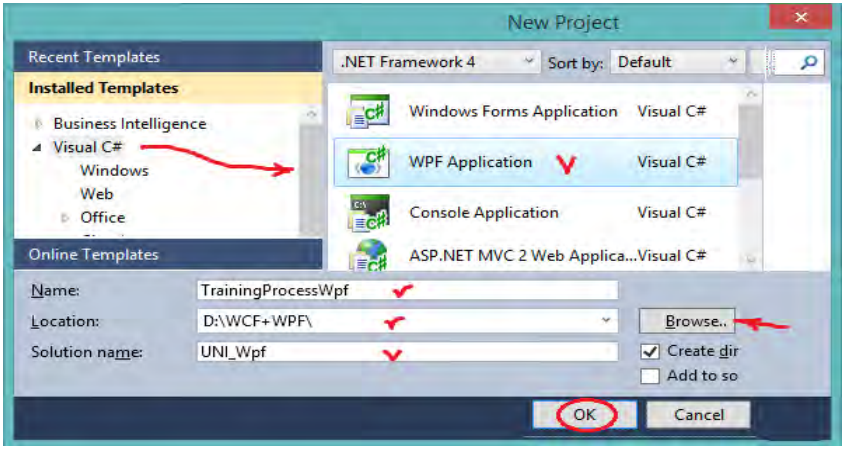
ამ თავში ნაჩვენები იქნება 1-ელი თავის პროექტის მსგავსი გადაწყვეტა, ოღონდაც კონსოლის რეჟიმის ნაცვლად გამოყენებულ იქნება Windows Presentation Foundation (WPF) აპლიკაცია.

აქამდე ჩვენ მიერ აგებულ პროექტში ჰოსტი მარტივად იძახებდა მუშა პროცესს და დამთავრების შემდეგ ასახავდა შედეგებს. ახალი პროექტისთვის საჭირო იქნება უფრო მეტი კავშირი მუშა პროცესსა და ჰოსტ-აპლიკაციას შორის. საბედნიეროდ, WF 4.0 აქვს კარგი შესაძლებლობები ამ მიზნის მისაღწევად.

2.1. WPF პროექტის შექმნა (ლაბ-4)

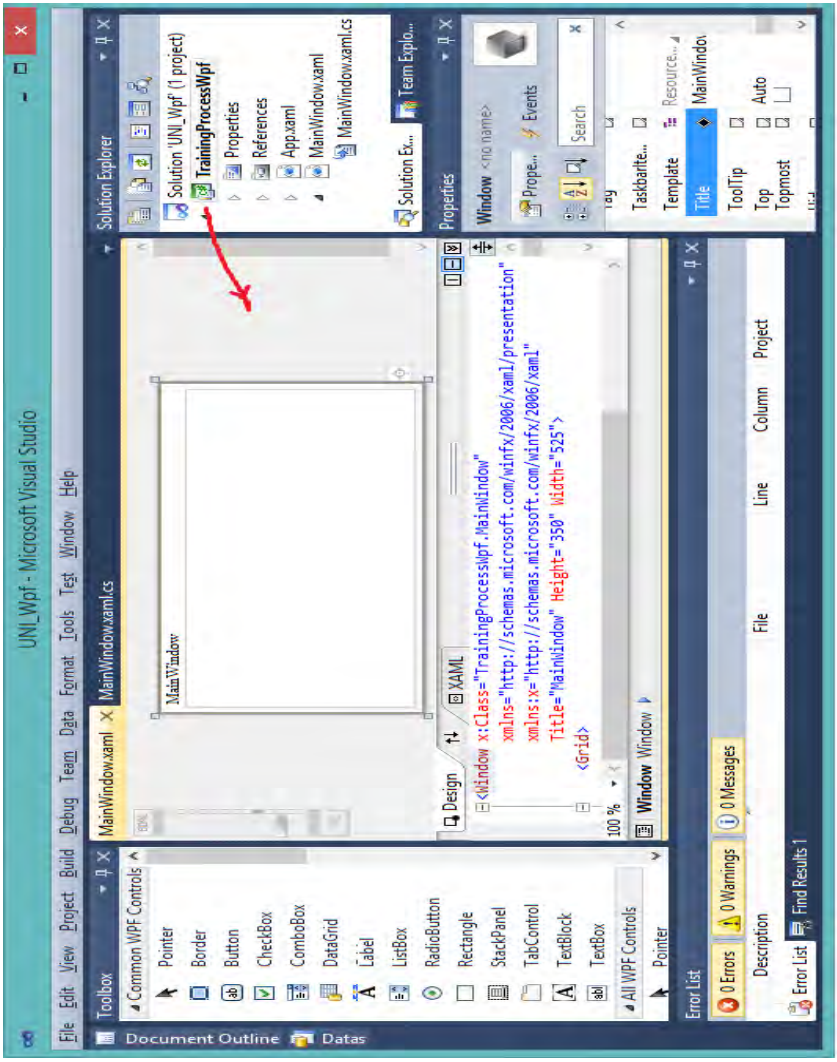
მიზანი: აპლიკაციათაშორისი კომუნიკაციების დაპროგრამების პროცესის შესწავლა ვიზუალური კომპონენტებით (WPF).

1. შევქმნათ ახალი პროექტი WPF აპლიკაციის სახით. პროექტის სახელი იყოს TrainingProcessWpf და Solution-ის Uni_Wpf.



ნახ.2.1. WPF აპლიკაციის შექმნა ტრეინინგ-ცენტრისთვის

შედეგი მოცემულია 2.2 ნახაზზე, სადაც ჩანს აპლიკაციის აგების სამუშაო გარემო მისი ყველა კომპონენტით [1].

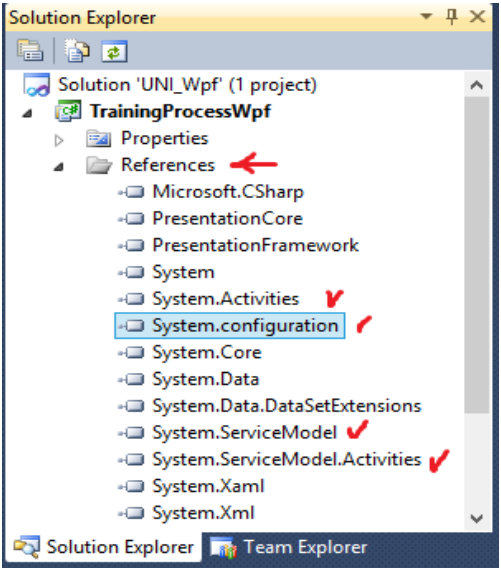


ნახ.2.2. Wpf ტექნოლოგიის სამუშაო გარემო

2. Solution Explorer-ში TrainingProcessWpf პროექტზე მაუსის მარჯვენა ღილაკით ავირჩიოთ Add Reference და .NET ცხრილიდან დავამატოთ შემდეგი კავშირები:

- System.Activities
- System.Configuration
- System.ServiceModel
- System.ServiceModel.Activities

Solution Explorer-ში მივიღებთ შემდეგ სურათს (ნახ.2.3).



ნახ.2.3.

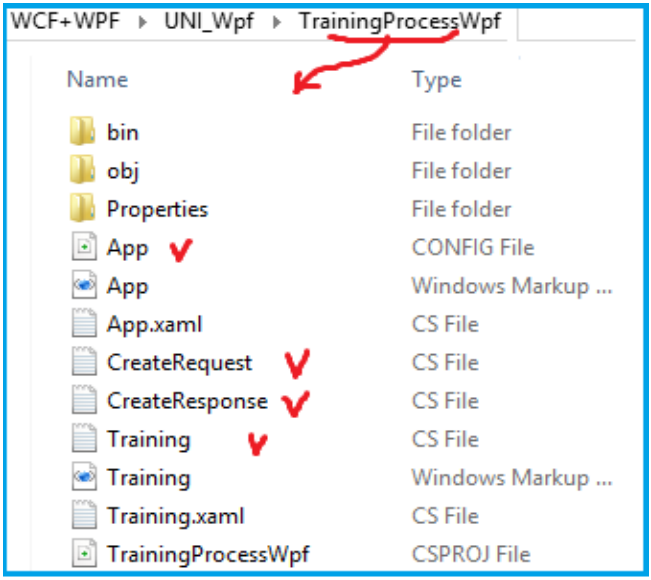
3. Solution Explorer-ში (ნახ.2.2) გენერირდება ვინდოუსის ფაილი სახელით MainWindow.xaml, რომელიც Training.xaml - ით შევცვალოთ.

App.xaml ფაილი განსაზღვრავს ვინდოუსის startup-ს. ესაა ახლა Windows1 ფაილი, რომელიც უნდა შევცვალოთ ასე:

StartupUri="Training.xaml"

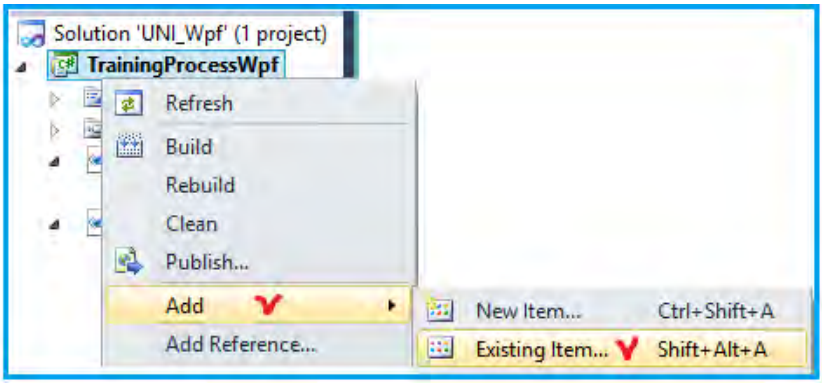
2.1.1. არსებული კლასების გამოყენება ახალ პროექტში

წინა თავში აგებული პროექტის აპლიკაციის TrainingProcess ფოლდერიდან: App.config, CreateRequest.cs, CreateResponse.cs და Reservation.cs ფაილები კოპირებით გადმოვიტანოთ მე-2 თავის Uni_Wpf-ის TrainingProcessWpf ფოლდერში (ნახ.2.4).

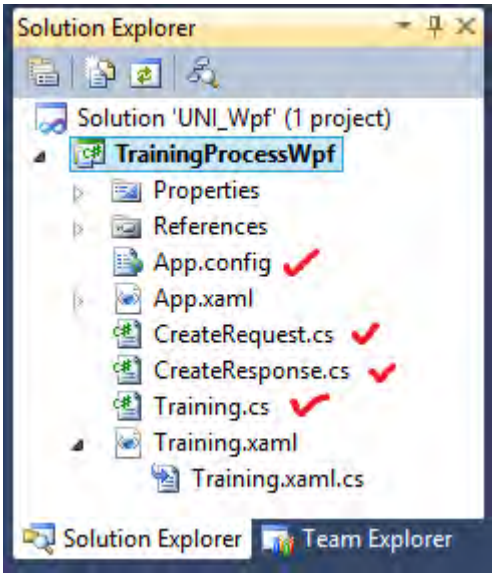


ნახ.2.4. ფაილების ჩამატება ახალი პროექტის კატალოგში

შემდეგ Solution Explorer-ის TrainingProcessWpf –ზე მაუსის მარჯვენა ღილაკით Add -> Existing Item და პროექტს მივუერთოთ კოპირებული ფაილები (ნახ.2.5). საბოლოოთ მივიღებთ 2.6 ნახაზზე ნაჩვენებ მდგომარეობას.



ნახ.2.5. ფაილების მიერთება TrainingProcessWpf-პროექტზე



ნახ.2.6. ახალ პროექტზე მიერთებული ფაილები

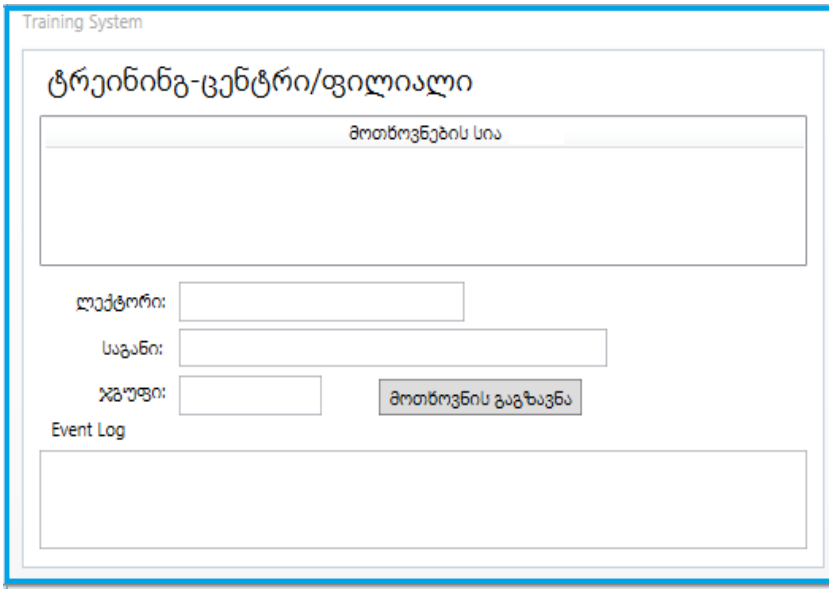
2.1.2. Window Form –ის განსაზღვრა

გავხსნათ Training.xaml ფაილი და ავირჩიოთ XAML tab. ჩავანაცვლოთ კოდი შემდეგი 2.1 ლისტინგის ტექსტით:

```
<!– – ლისტინგი_2.1 – – – –>
<Window x:Class="TrainingProcessWpf.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/
                                    xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Training System" Height="480" Width="650">
Loaded="Window_Loaded" Unloaded="Window_Unloaded">
<Grid>
   <Label Height="40" HorizontalAlignment="Left" Margin="12,0,0,0"
     Name="lblBranch" FontSize="20" VerticalAlignment="Top" Width="399"
     FontStretch="Expanded">ტრეინინგ-ცენტრი/ფილიალი</Label>
     <ListView x:Name="requestList" Margin="12,42,12,5" Height="150"
     VerticalAlignment="Top" ItemsSource="{Binding}">
       <ListView.View>
         <GridView>
           <GridViewColumn Header="მოთხოვნების სია" Width="610">
             <GridViewColumn.CellTemplate>
               <DataTemplate>
                 <StackPanel Orientation="Horizontal">
                   <TextBlock
                     Text="{Binding Requester.BranchName}" Width="100"/>
                   <TextBlock Text="{Binding Author}" Width="95"/>
                   <TextBlock Text="{Binding Title}" Width="180"/>
                   <TextBlock Text="{Binding ISBN}" Width="90"/>
                   <Button Content="Reserve" Tag="{Binding InstanceID}"
                       Click="Reserve" Width="65"/>
                   <Button Content="Cancel" Tag="{Binding InstanceID}"
                    Click="Cancel" Width="60"/>
                 </StackPanel>
               </DataTemplate>
             </GridViewColumn.CellTemplate>
```

```
            </GridViewColumn>
          </GridView>
        </ListView.View>
      </ListView>
      <Label Height="30" Margin="25,0,0,210" Name="label5"
        VerticalAlignment="Bottom" HorizontalAlignment="Left" Width="80"
        HorizontalContentAlignment="Right">ლექტორი:</Label>
      <Label Height="30" Margin="43,0,0,180" Name="label2"
        VerticalAlignment="Bottom" HorizontalAlignment="Left" Width="60"
        HorizontalContentAlignment="Right">საგანი:</Label>
      <Label Height="30" Margin="45,0,0,152" Name="label3"
        VerticalAlignment="Bottom" HorizontalAlignment="Left" Width="60"
        HorizontalContentAlignment="Right">ჯგუფი:</Label>
      <TextBox Height="25" Margin="0,0,318,213" Name="txtLector"
        VerticalAlignment="Bottom" HorizontalAlignment="Right" Width="200"
/>
      <TextBox Height="25" Margin="110,0,0,184" Name="txtSagani"
        VerticalAlignment="Bottom" HorizontalAlignment="Left" Width="300" />
      <TextBox Height="25" Margin="110,0,0,153" Name="txtJgupi"
        VerticalAlignment="Bottom" HorizontalAlignment="Left" Width="100" />
      <Button Height="23" Margin="250,0,0,153" Name="btnRequest"
        VerticalAlignment="Bottom" HorizontalAlignment="Left" Width="142"
        Click="btnRequest_Click" Content="მოთხოვნის გაგზავნა"></Button>
      <Label Height="27" HorizontalAlignment="Left" Margin="15,0,0,131"
       Name="label4" VerticalAlignment="Bottom" Width="76">Event
Log</Label>
      <ListBox Margin="12,0,12,12" Name="lstEvents" Height="119"
        VerticalAlignment="Bottom" FontStretch="Condensed" FontSize="10" />
     </Grid>
</Window>
```

შემდეგ ავირჩიოთ Design tab. მომხმარებლის ინტერფეისის ფორმას უნდა ჰქონდეს შემდეგი სახე (ნახ.2.7).



ნახ.2.7. მომხმარებლის ინტერფეისის დიზაინი

ფორმის ზედა ნაწილში „მოთხოვნების სია“ (Request List) ასახავს შემოსულ მოთხოვნებს, რომლებიც მოქმედებაშია. მოთხოვნის გასაგზავნად სხვა ფილიალში გამოიყენება ველები ინტერფეისის ფორმის შუაში. აქ მიეთითება ლექტორი, საგნის_დასახელება და ჯგუფი, შემდეგ ღილაკი „მოთხოვნის გაგზავნა“.

Event Log (მოვლენების რეგისტრაციის ჟურნალი) ქვედა მარცხენა კუთხეში ასახავს სამუშაო პროცესის შეტყობინებებს ისე, როგორც კონსოლის რეჟიმშია.

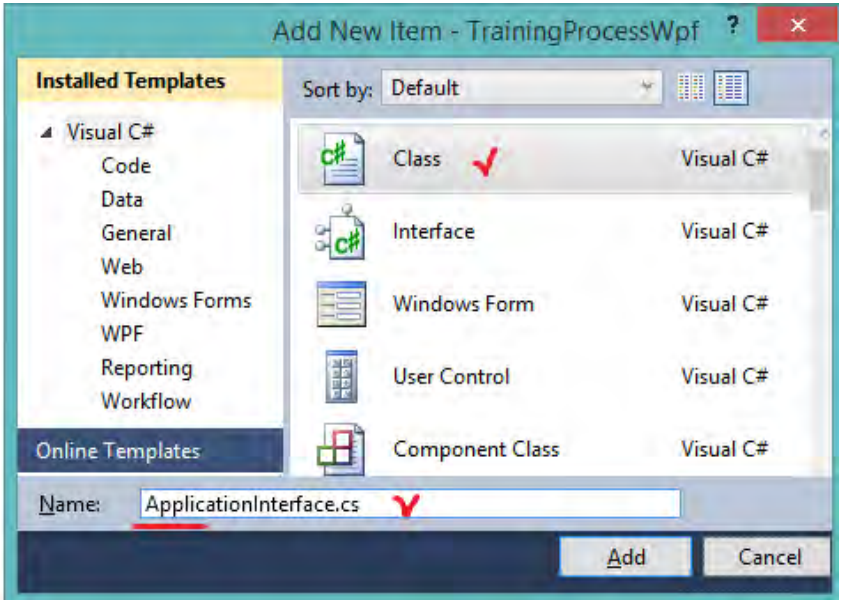
2.1.3. ტექსტის ჩამწერის რეალიზაცია (ლაბ-5)

მიზანი: TextWriter კლასის გაცნობა ფორმაზე მონაცემების გამოსატანად.

WriteLine ქმედებისთვის, რომელსაც ადრე ვიყენებდით, არ დაგვიყენებია თვისება TextWriter. კონსოლის რეჟიმში ტექსტი ავტომატურად გამოიტანებოდა ეკრანზე. ახლა კი ფორმაზე გამოსატანად უნდა გავეცნოთ TextWriter კლასს.

ა) აპლიკაციის სტატიკური მიმთითებლის უზრუნველყოფა.

თავიდან უნდა **შეიქმნას სტატიკური კლასი,** რომელიც უზრუნველყოფს აპლიკაციის ფანჯარასთან წვდომას. Solution Explorer-ში TrainingProcessWpf-ზე მაუსის მარჯვენა ღილაკით ვირჩევთ Add->Class. კლასის სახელია ApplicationInterface.cs, რომლის პროგრამული რეალიზაცია მოცემულია 2.2 ლისტინგში.



ნახ.2.8. ApplicationInterface კლასის შექმნა

```
// – – ლისტინგი_2.2 – – ApplicationInterface.cs  – –
using System;
using System.Windows.Controls;
using System.Activities;
namespace TrainingProcessWpf
 {
 public static class ApplicationInterface
{
    public static MainWindow _app { get; set; }
    public static void AddEvent(String status)
       {
         if (_app != null)
            {
         new
ListBoxTextWriter(_app.GetEventListBox()).WriteLine(status);
             }
        }
    }
  }
```

ApplicationInterface კლასს აქვს სტატიკური მიმთითებელი (_app) აპლიკაციის ფანჯარაზე (MainWindow კლასი). სტატიკური AddEvent() მეთოდი ქმნის ListBoxTextWriter კლასის ეგზემპლარს, რომელიც შემდგომში იქნება რეალიზებული და იძახებს მის WriteLine() მეთოდს.

გავხსნათ Training.xaml.cs ფაილი და დავამატოთ შემდეგი სახელსივრცეები:

```
using System.ServiceModel;
using System.ServiceModel.Activities;
using System.ServiceModel.Activities.Description;
using System.ServiceModel.Description;
```

```
using System.ServiceModel.Channels;
using System.Activities;
using System.Xml.Linq;
using System.Configuration;
```

დავამატოთ კონსტრუქტორის შემდეგი კოდი:

```
ApplicationInterface._app = this;
```

this აინიციალიზებს _app მიმართვას ApplicationInterface კლასში. ვინაიდან იგი სტატიკური კლასია, მასში იქნება მხოლოდ ერთი ეგზემპლარი, რომელსაც ექნება მიმართვა MainWindow კლასზე. დავამატოთ შემდეგი მეთოდები Training.xaml.cs ფაილში.

```
public ListBox GetEventListBox()
  {
    return this.lstEvents;
  }
private void AddEvent(string szText)
  {
     lstEvents.Items.Add(szText);
  }
```

GetEventListBox() მეთოდი აბრუნებს უკან მიმთითებელს ListBox–ის ფაქტობრივი კონტროლისთვის, რომელმაც უნდა ასახოს ეს მოვლენები. ამ მეთოდს იყენებს ApplicationInterface კლასი. AddEvent () მეთოდს იყენებს აპლიკაცია მაშინ, როცა მას სჭირდება მოვლენის დამატება.

ბ) ListBoxTextWriter–ის რეალიზაია

დავამატოთ პროექტს კლასი ListBoxTextWriter.cs, რომლის რეალიზაცია 2.3 ლისტინგშია მოცემული.

```
// – – ლისტინგი_2.3 – –
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Windows.Controls;
namespace TrainingProcessWpf
 {
  public class ListBoxTextWriter : TextWriter
    {
      const string textClosed = "This TextWriter must be opened before use";

     private Encoding _encoding;
     private bool _isOpen = false;
     private ListBox _listBox;

     public ListBoxTextWriter()
     {
          // Get the static list box
          _listBox = ApplicationInterface._app.GetEventListBox();
          if (_listBox != null)
          _isOpen = true;
       }

     public ListBoxTextWriter(ListBox listBox)
     {
            this._listBox = listBox;
            this._isOpen = true;
     }

      public override Encoding Encoding
      {
        get
         {
            if (_encoding == null)
```

```
            {
                _encoding = new UnicodeEncoding(false, false);
             }
          return _encoding;
       }
}

public override void Close()
{
   this.Dispose(true);
}

protected override void Dispose(bool disposing)
  {
  this._isOpen = false;
  base.Dispose(disposing);
  }

 public override void Write(char value)
  {
      if (!this._isOpen)
           throw new ApplicationException(textClosed) ;  ;

      this._listBox.Dispatcher.BeginInvoke
             (new Action(() =>
      this._listBox.Items.Add(value.ToString())));
  }

  public override void Write(string value)
   {
      if (!this._isOpen)
            throw new ApplicationException(textClosed) ;  ;
```

```
            if (value != null)
                this._listBox.Dispatcher.BeginInvoke
                    (new Action(() => this._listBox.Items.Add(value)));
        }

    public override void Write(char[] buffer, int index, int count)
      {
          String toAdd = "";

          if (!this._isOpen)
                throw new ApplicationException(textClosed) ;  ;

          if (buffer == null || index < 0 || count < 0)
              throw new ArgumentOutOfRangeException("buffer");

          if ((buffer.Length - index) < count)
              throw new ArgumentException("The buffer is too small");
          for (int i = 0; i < count; i++)
              toAdd += buffer[i];

          this._listBox.Dispatcher.BeginInvoke
               (new Action(() => this._listBox.Items.Add(toAdd)));
       }
     }
}
```

ListBoxTextWriter კლასი არის მიღებული აბსტრაქტული TextWriter კლასიდან და უზრუნველყოფს Write() მეთოდის განხორციელებას, რომელიც ამატებს სტრიქონს ListBox-ში (თუ თქვენ გსურთ განახორციელოთ Write() მეთოდის სამი გადატვირთვა, იმისთვის რომ იყოს მიღებული, როგორც char ან string ან char [] მასივი.)

არსებული კონსტრუქტორი იყენებს ApplicationInterface სტატიკურ კლასს MainWindow-ის lstEvents კონტროლის მისაღებად. იგი ასევე უზრუნველყოფს კონსტრუქტორს, რომელშიც ListBox შეიძლება შესრულდეს. ეს კონსტრუქტორი გამოიყენება ApplicationInterface კლასის AddEvent () მეთოდით.

ListBox-ის Add() მეთოდი ხორციელდება აპლიკაციის შესრულების ნაკადის (thread) შემდეგაც. იგი აკეთებს ამას Dispatcher-ის BeginInvoke() მეთოდის გამოყენებით, რომელიც ასოცირდება lstEvents მართვის ელემენტთან. ეს საშუალებას აძლევს მეთოდს იმუშაოს სხვადასხვა ნაკადებიდან გამომახების დროსაც.

იმის გამო, რომ ListBoxTextWriter კლასი არის მიღებული TextWriter-დან, შეიძლება მისი მითითება როგორც TextWriter თვისებისა ნებისმიერი WriteLine ქმედებისთვის. და სტატიკური ApplicationInterface კლასის გამო, ListBoxTextWriter კლასს შეუძლია წვდომა lstEvents ელემენტზე აპლიკაციის გარედანაც კი.

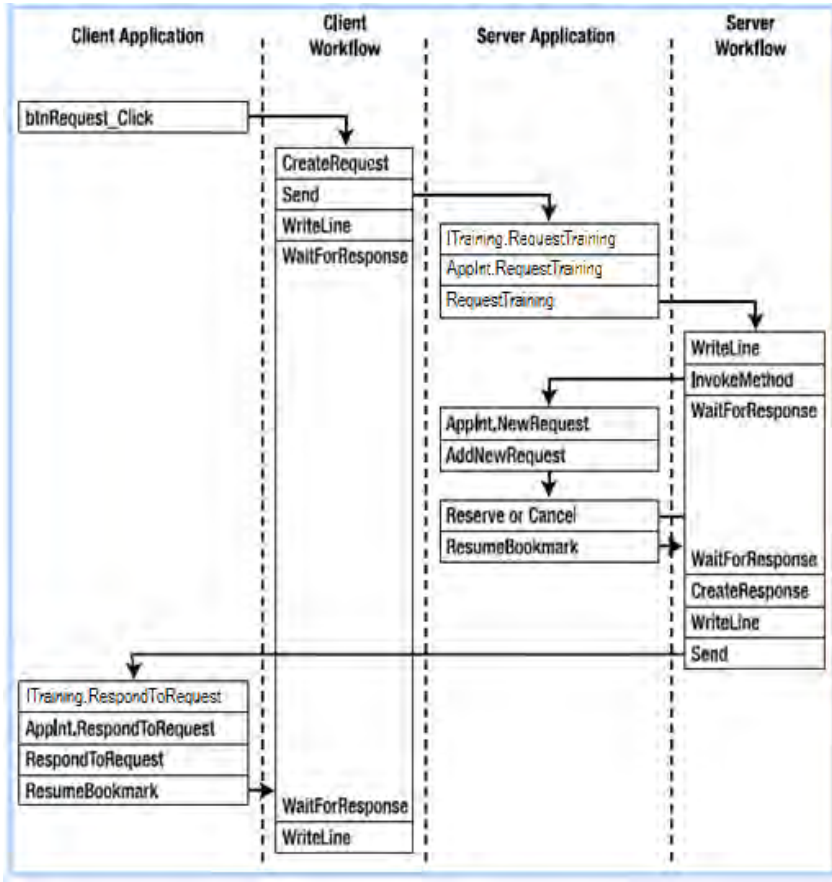
ასე რომ, არსებობს სამი გზა lstEvents მართვის ელემენტში ტექსტის დასამატებლად:

- აპლიკაციის შიგნიდან, გამოიყენება ლოკალური AddEvent() მეთოდი;
- აპლიკაციის გარედან, გამოიყენება ApplicationInterface კლასის AddEvent () მეთოდი;
- WriteLine ქმედებიდან, მიეთითება TextWriter თვისება ListBoxTextWriter-ზე.

2.1.4. Workflow პროცესების რეალიზაცია (ლაბ-6)

მიზანი: აპლიკაციის მიერ შეტყობინების მიღების და მის საფუძველზე სამუშაო პროცესის რეალიზაციის საკითხების გაცნობა.

2.9 ნახაზზე ნაჩვენებია ზოგადი ლოგიკა და შეტყობინებათა ნაკადი. ამ სქემის ელემენტები მოგვიანებით იქნება ახსნილი, ამჯერად კი განხილულ იქნება ძირითადი კონცეფციები.



ნახ.2.9. შეტყობინებათა ნაკადის მოძრაობის მოდელი

ეს მცირეთი განსხვავდება პროცესებისგან, რომლებიც წინა თავში განვიხილეთ. შესამჩნევი განსხვავება მდგომარეობს იმაში, რომ არ არსებობს არავითარი Receive (მიღების) ქმედება. მის მაგივრად აპლიკაცია მიიღებს შემავალ შეტყობინებებს, შემდეგ კი გამოიძახებს (ან აღადგენს) მუშა პროცესს.

ა) შეტყობინებათა მიღება

2.9 ნახაზზე სერვერული აპლიკაცია იღებს შეტყობინებას და მასთან დაკავშირებული ელემენტი დიაგრამეზე არის ლებელი ILibrary.RequestBook. გარდა ამისა, კლიენტის აპლიკაცია იღებს შეტყობინებას და ელემენტი აღნიშნულია ILibrary.RespondToRequest-ით. ესაა მეთოდები სერვისული კონტრაქტის, რომლებიც რეალიზებული იყო მე-8 თავში.

გავხსნათ Reservation.cs ფაილი, რომელშიც გამოჩნდება ინტერფეისის შემდეგი განსაზღვრება:

```
    [ServiceContract]
public interface ILibraryReservation
{
    [OperationContract]
    void RequestBook(ReservationRequest request);

    [OperationContract]
    void RespondToRequest(ReservationResponse response);
}
```

საჭიროა მცირე ცვლილების ჩატარება. კერძოდ OperationContract-ს უნდა დაემატოს (IsOneWay = true). ქვემოთ ნაჩვენებია ეს:

```
[ServiceContract]
public interface ITrainingProcess
  {
    [OperationContract(IsOneWay = true)]
     void RequestTraining(TrainingRequest request);

     [OperationContract(IsOneWay = true)]
     void RespondToRequest(TrainingResponse response);
  }
```

შეტყობინება იგზავნება მუშა პროცესთან ერთად, მაგრამ პასუხი მიიღება ServiceHost-ით აპლიკაციის შიგნით. ასე, რომ ეს არაა ტქნიკური ორმხრივი საუბარი. არსებობს შეტყობინებები ორივე მიმართულებით. რადგან გაგზავნის და მიღების საბოლოო წერტილები სხვადასხვაა, WCF ამას აფიქსირებს როგორც ცალკე ერთმიმართულებიან შეტყობინებებს.

ბ) სერვისის კონტრაქტის რეალიზაცია

სერვისის კონტრაქტი განსაზღვრავს მხოლოდ ხელმისაწვდომ მეთოდებს, იგი არ უზრუნველყოფს მათ იმპლემენტაციას (რეალიზაციას). წინა თავში მუშა პროცესი უზრუნველყოფდა რეალიზაციას.

ჩვენი პროექტისთვის აუცილებელია ამ საკითხის გადაწყვეტა. ამიტომაც, Solution Explorer-ში TrainingProcessWpf-ზე მარჯვენა ღილაკით ვირჩევთ Add Class. მივუთითებთ კლასის სახელს ClientService.cs. მისი კოდის რეალიზაცია ნაჩვენებია 2.4 ლისტინგში.

```
// ===== ლისტინგი 2.4 =========
using System;
using System.ServiceModel;

namespace TrainingProcessWpf
```

```
{
    public class ClientService : ITrainingProcessWpf
    {
        public void RequestTraining(TrainingRequest
request)
        {
            ApplicationInterface.RequestTraining(request);
        }

     public void RespondToRequest(TrainingResponse
response)
        {

ApplicationInterface.RespondToRequest(response);
        }
    }
}
```

ეს რეალიზაცია იყენებს ApplicationInterface სტატიკურ კლასს, რომელიც უკვე შექმნილია ჩვენს მიერ. ყოველი მეთოდი უბრალოდ იძახებს ApplicationInterface კლასის შესაბამის მეთოდს. გავხსნათ ApplicationInterface.cs ფაილი და დავამატოთ შემდეგი მეთოდები:

```
public static void RequestTraining(TrainingRequest request)
        {
          if (_app != null)
              _app.RequestTraining(request);
        }
public static void RespondToRequest(TrainingResponse response)
        {
            if (_app != null)
                _app.RespondToRequest(response);
        }
```

ეს მეთოდები თავის მხრივ იძახებს შესაბამის მეთოდებს აპლიკაციაში სტატიკური მიმთითებლის გამოყენებით. საჭირო ინება ამ მეთოდების რეალიზება Training.xaml.cs ფაილში, რასაც მოგვიანებით დავუბრუნდებით.

გ) ServiceHost -ის რეალიზაცია

აპლიკაციისთვის აუცილებელია ServiceHost-ის რეალიზაცია შემავალი შეტყობინებების მისაღებად (მოსასმენად). გავხსნათ Training.xaml.cs ფაილი და დავამატოთ შემდეგი კლასის წევრები:

private ServiceHost _sh;

იგი უნდა მოთავსდეს კონსტრუქტორის წინ. ასე:

```
public partial class MainWindow : Window
  {
      private ServiceHost _sh;

      public MainWindow()
      {
          InitializeComponent();
          ApplicationInterface._app = this;
      }
```

იწყება ServiceHost მაშინ, როცა ფანჯარა ჩატვირთულია და იხურება, როცა ფანჯარა ამოტვირთულია. მეთოდების დამატება ნაჩვენებია 2.5 ლისტინგში MainWindow კლასისთვის ჩატვირთვის და ამოტვირთვის მოვლენათა დამმუშავებლების სარეალიზაციოდ.

```
//== ლისტინგი 2.5 ======The Loaded and Unloaded Event Handlers ===
private void Window_Loaded(object sender, RoutedEventArgs e)
 { // გაიხსნას config ფაილი და მიეცეს ფილიალის სახელი და
   // მისი ქსელური მისამართი
 Configuration config =  ConfigurationManager.OpenExe
             Configuration(ConfigurationUserLevel.None);
  AppSettingsSection app = (AppSettingsSection)config.
                              GetSection("appSettings");
 string adr = app.Settings["Address"].Value;
   //  ფილიალის სახელის გამოტანა ფორმაზე
```

```
lblBranch.Content = app.Settings["Branch Name"].Value;

        // ServiceHost-ის შექმნა
        _sh = new ServiceHost(typeof(ClientService));

        // დასასრულის წერტილის (Endpoint) დამატება
        string szAddress = "http://localhost:" + adr +
                "/ClientService";
    System.ServiceModel.Channels.Binding bBinding = new
             BasicHttpBinding();
       _sh.AddServiceEndpoint(typeof(ITrainingProcessWpf),
              bBinding, szAddress);

  // ServiceHost-ის გახსნა შეტყობინებების მისაღებად (listen)
         _sh.Open();

 // ListBoxTextWriter -ის ტესტირება
 //ListBoxTextWriter lbtw = new ListBoxTextWriter();
 //lbtw.Write("ეს არის ტესტი - This is a test");
    }

private void Window_Unloaded(object sender, RoutedEventArgs e)
        {
            // service host-ის დატოვება
            _sh.Close();
        }
```

მოვლენის დამმუშავებელი Loaded ხსნის კონფიგურაციის ფაილს და ათავსებს ფილიალის სახელს lblBranch - მართვის ელემენტში, ამიტომაც ფორმა ასახავს ლოკალური ფილიალის სახელს. შემდეგ იქმნება ServiceHost თანამგზავრი (passing) ClientService კლასისა, რომელიც ახლახანს შევქმენით როგორც მისი რეალიზაცია. შემდეგ იგი აკონფიგურირებს დასასრულის

წერტილს ServiceHost-თვის, იყენებს რა ცნობილი მისამართის, მიბმის და კონტრაქტის სამეულს.

Unloaded მოვლენის დამმუშავებელი უბრალოდ ხურავს ServiceHost-ს, ასე რომ მეტი აღარ მოხდება შეტყობინებების მიღება.

2.2. სანიშნეები და Workflow-პროცესის რეალიზაცია (ლაბ-7)

მიზანი: სამუშაო პროცესების მართვის წესების ათვისება. მაგალიტად, როგორ შეაჩეროს ვორკფლოვ პროცესი, დაიმახსოვროს შეჩერების წერტილი შემდგომი აღდგენის მიზნით და ა.შ.

სანიშნეები (Bookmarks) იძლევა საშუალებას შეაჩეროს მუშა პროცესი და შეინახოს მარკერი ისე, რომ შემდგომ შეიძლებოდეს მისი იმავე წერტილიდან განახლება. ისინი დამუშავებულია მონაცემთა მისაღებად შემდგომი აღდგენის მიზნით. ამ პროექტში, მაგალითად, როცა მოთხოვნა მიღებულია, აპლიკაციას გამოაქვს ეკრანზე ეს მოთხოვნა და ელოდება მომხმარებლისგან პასუხის (კი/არა) შეტანას. შემდეგ მუშა პროცესი გრძელდება ამ პასუხის გავლით.

სანიშნეები იქმნება მომხმარებელთა აქტიურობით. აქ ჩვენ შევქმნით ზოგად ქმედებას, რომელიც შემდგომში გამოყენებადი იქნება ყველგან, სადაც სანიშნე მოითხოვს. Solution Explorer-ში TrainingProcessWpf-ზე მარჯვენა ღილაკით დავამატოთ კლასი WaitForInput.cs, რომლის 2.6 ლისტინგი მოცემულია ქვემოთ.

```
// ლისტინგი 2.6 ======
using System;
using System.Activities;

namespace TrainingProcessWpf
{
```

```
 public sealed class WaitForInput<T> :
NativeActivity<T>
    {
        public WaitForInput() : base()
        {
        }

        public string BookmarkName { get; set; }
        public OutArgument<T> Input { get; set; }

        protected override void
Execute(NativeActivityContext context)
        {
           context.CreateBookmark(BookmarkName,
                 new BookmarkCallback(this.Continue));
        }

void Continue(NativeActivityContext context, Bookmark
                    bookmark, object obj)
        {
            Input.Set(context, (T)obj);
        }
protected override bool CanInduceIdle { get { return true; } }
    }
}
```

= = = =

განმარტება: sealed class -ის შესახებ (http://www.techopedia.com/definition/25637/sealed-class-c):

ეს არის **დაცული** კლასი C# -ში, რომელიც არ შეიძლება მიღებულ იქნს მემკვიდრეობით ყველა კლასის მიერ, მაგრამ მისი შექმნა შესაძლებელია.

დაცული კლასის დიზაინერული ჩანაფიქრი ისაა, რომ იქნას მითითებული, რომ ის სპეციალიზებულია და არაა აუცილებელი

მისი გაფართოება, მემკვიდრეობით მისთვის დამატებითი ფუნქციონალობის გადაცემა, მისი ყოფაქცევის გადასატვირთად.

დაცული კლასი ხშირად გამოიყენება ლოგიკის ინკაფსულაციისთვის, რომელიც პროგრამამ უნდა გამოიყენოს ყოველგვარი ცვლილებების გარეშე.

დაცული კლასი გამოიყენება ძირითადად უსაფრთხოების მიზნით. მას არ შეუძლია საბაზო კლასის ფორმირება. დაცული კლასების გამოძახება უფრო სწრაფად ხდება, რადგან ისინი უზრუნველყოფს შესრულების გარკვეულ ოპტიმიზაციას, მაგალითად, ვირტუალური ფუნქცია-წევრების გამოძახება დაცული კლასის შემთხვევაში არავირტუალური გამოძახებისას.

.NET Framework ბიბლიოთეკის ზოგიერთი საკვანძო კლასები შესრულებულია დაცული კლასების სახით, ძირითადად მათი გაფართოების შეზღუდვის მიზნით.

= = = =

მომხმარებლის ეს ქმედება გამოიყენებს NativeActivity საბაზო კლასს (CodeActivity-ის ნაცვლად), იმიტომ რომ იგი აძლევს მას NativeActivityContext-თან მიმართვის საშუალებას, რომელიც აუცილებელია სანიშნის შესაქმნელად. იგი ასევე იყენებს შაბლონის ვერსიას (კლასში <T> აღნიშვნაა). შემავალი არგუმენტი ასახავს მონაცემებს, რომლებიც გადაეცემა მუშა პროცესს, როცა ის განახლდება. შაბლონური ვერსიის დახმარებით ეს ქმედება შესაძლებელია გამოყენებულ იქნას განმეორებით მონაცემთა ნებისმიერ ტიპთან.

Execute() მეთოდი იძახებს NativeActivityContext-ის CreateBookmark() მეთოდს, მიუთითებს რა სანიშნის სახელს და მიმთითებელს უკუ-გამოძახების მეთოდზე, სახელით Continue(). როდესაც მუშა პროცესი აღდგენილია, მაშინ ეს უკუ-გამოძახების მეთოდი სრულდება. ყურადსაღებია, რომ უკუ-გამოძახების მეთოდი ობიექტს ღებულობს მესამე პარამეტრის სახით. ესაა

მონაცემები, წარმოდგენილი აპლიკაციით. იგი ინახება შემავალ არგუმენტში, რაც ხელმისაწვდომია მუშა პროცესისთვის.

აქტიურობები (ქმედებები), რომლებიც იყენებს სანიშნეებს, უნდა იქნას გადატვირთული (override), რომ CanInduceIdle თვისება აბრუნებდეს true მნიშვნელობას. ეს კი უზრუნველყოფს მუშა პროცესს, რომ გადავიდეს ლოდინის მდგომარეობაში მანამ, სანამ სანიშნით მოხდება მისი აღდგენა.

2.2.1. SendRequest სამუშაო პროცესის რეალიზაცია

ახლა შევასრულოთ მუშა პროცესების რეალიზაცია. Solution Explorer-ის TrainingProcessWpf-ზე მარჯვენა ღილაკით ავირჩიოთ Add->Class. სახელი TrainingWF.cs. კოდის რეალიზაცია მოცემულია 2.7 ლისტინგში.

```
// ლისტინგი 2.7======
using System;
using System.Activities;
using System.Activities.Statements;
using System.ServiceModel.Activities;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.Runtime.Serialization;
using System.Xml.Linq;
using System.IO;

namespace TrainingProcessWpf
 {
  // This file contains the definition of two
workflows:
  // SendRequest initiates a new request
  // ProcessRequest handles incoming requests
  public sealed class SendRequest : Activity
    {
```

```
      // Define the input and output arguments
        public InArgument<string> Sagani { get; set; }
        public InArgument<string> Lector { get; set; }
        public InArgument<string> Jgupi { get; set; }
        public InArgument<TextWriter> Writer {get;set;}
    public OutArgument< TrainingProcessWpf>
           Response{get;set;}

    public SendRequest()
      {
       // Define the variables used by this workflow
       Variable<TrainingRequest> request =
   new Variable<TrainingRequest> { Name = "request" };
       Variable<string> requestAddress =
   new Variable<string> { Name = "RequestAddress" };
       Variable<bool> reserved =
   new Variable<bool> { Name = "Reserved" };

      // Define the SendRequest workflow
    this.Implementation = () => new Sequence
     {
       DisplayName = "SendRequest",
       Variables = {request,requestAddress,reserved },
       Activities =
        {
         new CreateRequest
          {
Sagani = new InArgument<string>(env=>Sagani.Get(env)),
Lector = new InArgument<string>(env=>Lector.Get(env)),
Jgupi  = new InArgument<string>(env=>Jgupi.Get(env)),
Request = new OutArgument<TrainingRequest>
                            (env => request.Get(env)),
RequestAddress = new OutArgument<string>
                   (env => requestAddress.Get(env))
```

```
        },
    new Send
       {
        OperationName = "RequestTraining",
        ServiceContractName = "ITrainingProcessWpf",
        Content = SendContent.Create
         (new InArgument<ReservationRequest>(request)),
        EndpointAddress = new InArgument<Uri>
              (env => new Uri("http://localhost:" +
            requestAddress.Get(env) + /ClientService")),
        Endpoint = new Endpoint
         {
           Binding = new BasicHttpBinding()
         },
        },
       new WriteLine
        {
         Text = new InArgument<string>
          (env => "Request sent; waiting for response"),
         TextWriter = new InArgument<TextWriter>
            (env => Writer.Get(env))
         },
        new WaitForInput<TrainingResponse>
         {
           BookmarkName = "GetResponse",
           Input = new OutArgument<TrainingResponse>
                       (env => Response.Get(env))
         },
        new WriteLine
         {
          Text = new InArgument<string>
             (env => "Response received from " +
    Response.Get(env).Provider.BranchName + " [" +
Response.Get(env).Reserved.ToString() + "]"),
```

```
    TextWriter = new InArgument<TextWriter>
          (env => Writer.Get(env))
       },
      }
     };
    }
   }
   // Add the ProcessRequest workflow here
}
```

ამ მუშა პროცესის უმეტესი ნაწილი იდენტურია წინა თავში განხილული რეალიზაციის, ამიტომაც აქ იგი დეტალურად აღარ აიხსნება. მოცემულ იქნება მხოლოდ ის, რაშიც განსხვავებაა. უნდა აღვნიშნოთ, რომ ყოველ WriteLine ქმედებას აქვს დამატებითი თვისება:

TextWriter = new ListBoxTextWriter()

ის მიუთითებს იმაზე, რომ ახალი კლასი ListBoxTextWriter, რომელიც იქნა რეალიზებული, უნდა იქნას გამოყენებული ამ ტექსტის დისპლეიზე გამოსატანად. ეს გამოიწვევს ტექსტის ასახვას lstEvents მარტვის ელემენტში.

სხვა განსხვავება იმაშია, რომ მომხმარებლის ქმედება WaitForInput გამოიყენება Receive ქმედების ნაცვლად. აპლიკაცია მიიღებს საპასუხო შეტყობინებას უშუალოდ (პირდაპირ). როცა მიღებულ იქნება პასუხი, მაშინ აპლიკაცია აღადგენს მუშა პროცესს, რომელიც მიმდინარეობს ReservationResponse კლასში. ყურადსაღებია, რომ მომხმარებლის ქმედება განისაზღვრება როგორც WaitForInput <TrainingResponse>, მიუთითებს რა, რომ გადასაცემი მონაცემები იქნება TrainingResponse კლასის.

2.2.2. ProcessRequest სამუშაო პროცესის რეალიზაცია

ProcessRequest მუშა პროცესი განსაზღვრება მოცემულია 2.8 ლისტინგში. ჩავამატოთ ეს კოდი TrainingWF.cs ფაილში.

```
// ლისტინგი 2.8 ==========
public sealed class ProcessRequest : Activity
{
 public InArgument<TrainingRequest> request{get;set;}
 public InArgument<TextWriter> Writer { get; set; }
 public ProcessRequest()
  {
    // Define the variables used by this workflow
   Variable<TrainingResponse> response =
     new Variable<TrainingResponse> {Name="response"};
   Variable<bool> reserved = new Variable<bool>
         { Name = "Reserved" };
   Variable<string> address = new Variable<string>
         { Name = "Address" };

   // Define the ProcessRequest workflow
   this.Implementation = () => new Sequence
    {
      DisplayName = "ProcessRequest",
      Variables = { response, reserved, address },
      Activities =
       {
         new WriteLine
          {
            Text = new InArgument<string>
               (env => "Got request from: " +
               request.Get(env).Requester.BranchName),
                TextWriter = new InArgument<TextWriter>
                      (env => Writer.Get(env))
```

```
    },
   new InvokeMethod
   {
     TargetType = typeof(ApplicationInterface),
      MethodName = "NewRequest",
      Parameters =
       {
         new InArgument<TrainingRequest>
             (env => request.Get(env))
       }
     },
     new WaitForInput<bool>
      {
       BookmarkName = "GetResponse",
       Input = new OutArgument<bool>
               (env => reserved.Get(env))
      },
  new CreateResponse
   {
     Request = new InArgument<TrainingRequest>
                   (env => request.Get(env)),
     Reserved = new InArgument<bool>
                   (env => reserved.Get(env)),
  Response = new OutArgument<TrainingResponse>
             (env => response.Get(env))
   },
new WriteLine
 {
   Text = new InArgument<string>
             (env => "Sending response to: " +
   request.Get(env).Requester.BranchName),
 TextWriter = new InArgument<TextWriter>
         (env => Writer.Get(env))
},
```

```
        new Send
         {
          OperationName = "RespondToRequest",
          ServiceContractName = "ITrainingProcessWpf",
          EndpointAddress = new InArgument<Uri>
                 (env => new Uri("http://localhost:" +
                 request.Get(env).Requester.Address +
                        "/ClientService")),
          Endpoint = new Endpoint
           {
             Binding = new BasicHttpBinding()
           },
           Content = SendContent.Create
          (new InArgument<TrainingResponse>(response))
         }
        }
       };
      }
    }
```

ეს მუშა პროცესი განსხვავდება წინა თავში რეალიზებული ვერსიისგან. იმის მაგივრად, რომ დაწყება იყოს Receive ქმედებით, რათა მიღებულ იქნას შემავალი მოთხოვნა, TrainingRequest გადასცემს მუშა პროცესს შემავალი არგუმენტის გამოყენებით. WriteLine ქმედება, რომელიც მოსდევს მას, ცნობს შემავალ მოთხოვნას.

InvokeMethod ქმედება გამოვიყენოთ მონაცემთა გადასაცემად აპლიკაციაში. ApplicationInterface კლასი მოხერხებულადაა შესრულებული ამ მიზნით. იგი უზრუნველყოფს სამუშაო პროცესს, რათა განხორციელდეს გამოძახება აპლიკაციაში. InvokeMethod ქმედება იძახებს

ApplicationInterface კლასის NewRequest() მეთოდს TrainingRequest კლასში გადასაცემად.

გავხსნათ ApplicationInterface.cs ფაილი და დავამატოთ მეთოდი, რომელიც უბრალოდ იძახებს AddNewRequest ()-ს აპლიკაციაში:

```
public static void NewRequest(TrainingRequest request)
    {
        if (_app != null)
            _app.AddNewRequest(request);
    }
```

შემდეგი აქტიურობაა მომხმარებლის WaitForInput ქმედება, რომელიც გამოიყენებოდა SendRequest მუშა პროცესში.

ამჯერად იგი ელოდება Bool-შესატანი მითითება იყო თუ არა დაჯავშნული დასახელება (საგანი). CreateResponse და WriteLine ქმედებები იგივეა, რაც იყო წინა თავში. აქ გამოიყენებოდა SendReply ქმედება, ვინაიდან იგი იყო დაკავშირებული საწყის Receive ქმედებასთან.

ამ პროექტში, ვინაიდან არაა არავითარი Receive ქმედება, ჩვენ გამოვიყენებთ Send ქმედებას. საყურადღებოა, რომ EndpointAddress აწყობილია მისამართის გამოყენებით (პორტის ნომერი), რომელიც გათვალისწინებულია შესატან მოთხოვნაში.

2.3. აპლიკაციის რეალიზაცია (ლაბ-8)

მიზანი: პროექტის რეალიზაცია აპლიკაციის სახით მოვლენათა დამმუშავებლების დაპროგრამების ასათვისებლად.

არსებობს მოვლენათა რამდენიმე დამმუშავებელი (event handlers), რომელთა რეალიზაცია აუცილებელია, აგრეთვე მეთოდები, რომლებიც გამოიძახება სტატიკური ApplicationInterface კლასით.

2.3.1. მხარდაჭერა მუშა პროცესების ეგზემპლარებისთვის

აპლიკაცია თვალყურს უნდა ადევნებდეს მუშა პროცესის ეგზემპლარებს, ამიტომაც მას შეუძლია განაახლოს სწორი ეგზემპლარი. ამის შესრულება შესაძლებელია მარტივად ობიექტის ლექსიკონით.

გავხსნათ Training.xaml.cs ფაილი და დავამატოთ კლასის წევრები მომხმარებლის ServiceHost-ის ქვემოთ:

```
private IDictionary<Guid, WorkflowApplication>

_incomingRequests;
private IDictionary<Guid, WorkflowApplication>

_outgoingRequests;
```

ისინი იყენებს სამუშაო პროცესის ეგზემპლარის იდენტიფიკატორს, როგორც ლექსიკონის გასაღებს და WorkflowApplication ობიექტს, როგორც მნიშვნელობას. ვინაიდან აპლიკაცია ამუშავებს ორივე მუშა პროცესს SendRequest და ProcessRequest, ამიტომაც საჭირო იქნება ლექსიკონის ორი ობიექტი.

დავამატოთ MainWindow() კონსტრუქტორში კოდი ამ ობიექტების ინიციალიზებისთვის:

```
_incomingRequests = new Dictionary<Guid,
                                   WorkflowApplication>();
_outgoingRequests = new Dictionary<Guid,
                                   WorkflowApplication>();
```

საჭიროა კიდევ ერთი მცირე ცვლილება მომხმარებლის CreateRequest ქმედებაში. მუშა პროცესის ეგზემპლარის ID გამოყენებულ უნდა იქნას როგორც TrainingRequest კლასის RequestID ველი. აპლიკაცია მას გამოიყენებს პროცესის განახლების დროს.

გავხსნათ CreateRequest.cs ფაილი და შევცვალოთ გამოძახება, რომელიც ქმნის TrainingRequest კლასს, ალტერნატიული კონსტრუქტორის გამოსაყენებლად, რომელიც ღებულობს მეხუთე პარამეტრს RequestID -თვის. დავამატოთ მუქი სტრიქონი კოდის შემდეგ ტექსტში:

```
// TrainingRequest კლასის შექმნა და
// მისი შევსება შესატანი არგუმენტებით

TrainingRequest r = new TrainingRequest
    (
      Sagani.Get(context),
      Lector.Get(context),
      Jgupi.Get(context),
     new Branch
      {
       BranchName = app.Settings["Branch Name"].Value,
       BranchID = new Guid(app.Settings["ID"].Value),
```

```
        Address = app.Settings["Address"].Value
      },
      context.WorkflowInstanceId   // !!!
    );
```

2.3.2. მოვლენათა დამმუშავებელი (Event Handlers)

ახალი მოთხოვნის შესაქმნელად მომხმარებელი შეავსებს ***ლექტორის, საგნის სათაურის, ჯგუფის*** ველებს და აამოქმედებს SendRequest ღილაკს. ამ მოვლენის ღილაკის რეალიზება მოცემულია 2.9 ლისტინგში, Training.xaml.cs ფაილში.

```
// --- ლისტინგი 2.9 -----Click Event -ის რეალიზება -----
private void btnRequest_Click(object sender, RoutedEventArgs
e)
 {
  // Setup a dictionary object for passing parameters
   Dictionary<string, object> parameters =  new
                          Dictionary<string, object>();
      parameters.Add("Lector", txtLector.Text);
      parameters.Add("Sagani", txtSagani.Text);
      parameters.Add("Jgupi", txtJgupi.Text);
      parameters.Add("Writer", new
                       ListBoxTextWriter(lstEvents));

     WorkflowApplication i = new WorkflowApplication
                (new SendRequest(), parameters);
     _outgoingRequests.Add(i.Id, i);
     i.Run();
  }
```

ამ მეთოდის პირველი ნაწილი ჩვენთვის ნაცნობია. იგი იყენებს ობიექტის ლექსიკონს შემავალი არგუმენტების შესანახად, რომლებიც უნდა გადაეცეს მუშა პროცესს. შემდეგ იგი ქმნის

WorkflowApplication-ს, რომლის კონსტრუქტორსაც გადაეცემა პარამეტრები:

- მუშა პროცესების დეფინიცია;
- ობიექტის ლექსიკონი, რომელიც შეიცავს შემავალ არგუმენტებს.

WorkflowApplication შემდეგ ემატება _outgoingRequests კოლექციას.

ბოლოს, ეგზემპლარი გაიშვება Run () მეთოდით.

- - - - - - - -

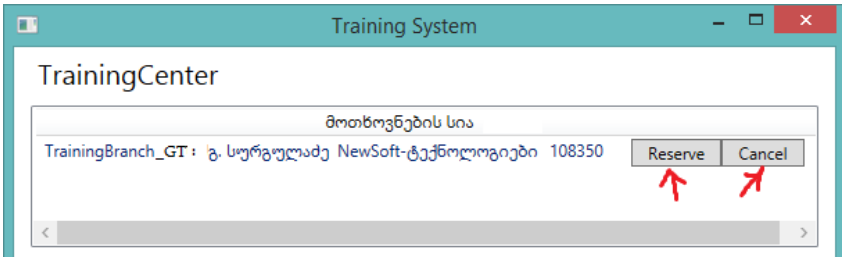
რეკომენდაცია:

წინა პროექტებში ჩვენ ვიყენებდით WorkflowInvoker კლასის Invoke () მეთოდს, რათა დაწყებულიყო WorkflowApplication.

ეს მიდგომა იწყებს მუშა პროცესს სინქრონულად; მუშა პროცესი შესრულდება გამომძახებლის შესრულების ნაკადში (caller's thread). ეს ნიშნავს, რომ აპლიკაცია ბლოკირებულია მანამ, სანამ მუშა პროცესი არ გადავა ლოდინის რეჟიმში. მაგრამ ეს ის არაა, რაც ჩვენ გვინდა ამ პროექტში. ჩვენ გვინდა, რომ მუშა პროცესი დაიწყოს თავის საკუთარ შესრულების ნაკადში, მაშინ როცა აპლიკაციას შეუძლია გააგრძელოს რეაგირება მოვლენებზე (და შემავალ შეტყობინებებზე). Run () მეთოდის გამოყენება წყვეტს სწორედ ამ ამოცანას.

- - - - - - - - -

მოთხოვნების სიის ფორმაზე მოთავსებულია ღილაკები Reserve და Cancel, რომელთაც იყენებს მომხმარებელი იმის მისათითებლად, თუ რომელი ელემენტი (true, false) იყო გამოყენებული (ნახ.2.10).



ნახ.2.10.

2.10 ლისტინგი აღწერს ამ ღილაკებისთვის მოვლენათა დამმუშავებლების რეალიზაციას. დავამატოთ ეს მეთოდები Training.xaml.cs ფაილში.

```
// -- ლისტინგი 2.10 -- Reserve და Cancel ღილაკების რეალიზაცია ---
-
// Handle the Reserve button click event
 private void Reserve(object sender, RoutedEventArgs e)
  {
    // Get the instanceID from the Tag property
    FrameworkElement fe = (FrameworkElement)sender;
    Guid id = (Guid)fe.Tag;
    ResumeBookmark(id, true);
  }
//Handle the Cancel button click event
 private void Cancel(object sender, RoutedEventArgs e)
  {
    // Get the instanceID from the Tag property
    FrameworkElement fe = (FrameworkElement)sender;
    Guid id = (Guid)fe.Tag;
    ResumeBookmark(id, false);
  }
 private void ResumeBookmark(Guid id, bool bReserved)
  {
     WorkflowApplication i = _incomingRequests[id];
```

```
    try
     {
       i.ResumeBookmark("GetResponse", bReserved);
     }
     catch (Exception e)
       {
          AddEvent(e.Message);
       }
}
```

მოვლენის დამმუშავებლები იღებს მუშა პროცესის ეგზემპლარის ID-ს ღილაკის Tag თვისებიდან. შემდეგ იძახებენ ResumeBookmark() მეთოდს, მიაწოდებს true-ს ან false-ს, იმისდა მიხედვით, თუ რომელი ღილაკი იყო ამოქმედებული.

ResumeBookmark() მეთოდი მიიღებს WorkflowApplication-ს _incomingRequests-კოლექციიდან და გამოიძახებს მის ResumeBookmark () მეთოდს. გადაეცემა სანიშნის სახელი (bookmark name) და მნიშვნელობა, რომელშიც ეგზემპლარი განახლედება (resumed).

2.4. ApplicationInterface მეთოდები (ლაბ-9)

მიზანი: აპლიკაციის ინტერფეისის კლასის მეთოდების პროგრამული რეალიზციის შესწავლა.

ჩვენ განვსაზღვრეთ ApplicationInterface კლასის სამი მეთოდი. ახლა უნდა უზრუნველვყოთ მათი რეალიზაცია MainWindow კლასში (Training.xaml.cs ფაილში). ამ მეთოდების რეალიზაცია მოცემულია 2.11 ლისტინგში.

```
// ლისტინგი_2.11-ApplicationInterface.cs მეთოდების რეალიზაცია-
public void RequestTraining(TrainingRequest request)
 {
```

```
   // Setup a dictionary object for passing parameters
   Dictionary<string, object> parameters = new
          Dictionary<string, object>();
   parameters.Add("request", request);
   parameters.Add("Writer", new
                ListBoxTextWriter(lstEvents));
   WorkflowApplication i = new WorkflowApplication
            (new ProcessRequest(), parameters);

             request.InstanceID = i.Id;
             _incomingRequests.Add(i.Id, i);
             i.Run();
    }

public void RespondToRequest(TrainingProcessWpf response)
    {
       Guid id = response.RequestID;
       WorkflowApplication i = _outgoingRequests[id];
       try
        {
          i.ResumeBookmark("GetResponse", response);
        }
        catch (Exception e2)
           {
               AddEvent(e2.Message);
           }
     }

   public void AddNewRequest(TrainingRequest request)
      {
        this.requestList.Dispatcher.BeginInvoke
            (new Action(() =>
                  this.requestList.Items.Add(request)));
      }
```

RequestTraining() მეთოდი ანალოგიურია btnRequest_Click() მეთოდის. იგი გამოიძახება მაშინ, როცა შემავალი შეტყობინება მიღებულია ServiceHost-დან და სერვის კონტრაქტის RequestTraining მეთოდი მითითებულია. ის აგებს ობიექტის ლექსიკონს ერთი არგუმენტის შესანახად, ქმნის WorkflowApplication-ს, ამატებს მას _incomingRequests კოლექციაში, ხოლო შემდეგ აამოქმედებს მუშა პროცესს.

RespondToRequest() მეთოდი ასევე გამოიძახება ServiceHost-დან მიღებული შეტყობინებით. იგი გამოიძახება მაშინ, როცა RespondToRequest მეთოდია მითითებული. ეს ხდება მაშინ, როცა სხვა ფილიალები აგზავნის უკან პასუხს შემოსულ მოთხოვნაზე. იგი ღებულობს WorkflowApplication-ს _outgoingRequests კოლექციიდან და აღადგენს სანიშნეს, გამავალს TrainingResponse კლასში.

AddNewRequest() გამოიძახება ProcessRequest მუშა პროცესით, როცა მიიღება ახალი შეტყობინება. ეს ხდება InvokeMethod ქმედების დახმარებით. იგი უბრალოდ დაამატებს ჩანაწერს ListView-კონტროლის RequestList-ელემენტში. ვინაიდან ის გამოძახებულ უნდა იქნას მუშა პროცესის შესრულებად ნაკადში, Dispatcher კლასი გამოიყენებს შესასრულებლად Add() მეთოდს main window-ის შესრულებადი ნაკადით. Training.xaml.cs-ის სრული რეალიზაცია მოცემულია 2.12 ლისტინგში.

```
// --- ლისტინგი 2.12 ---
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
```

```
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.ServiceModel;
using System.ServiceModel.Activities;
using System.ServiceModel.Activities.Description;
using System.ServiceModel.Description;
using System.ServiceModel.Channels;
using System.Activities;
using System.Xml.Linq;
using System.Configuration;

namespace TrainingProcessWpf
{
  // Interaction logic for MainWindow.xaml
  public partial class MainWindow : Window
  {
    private ServiceHost _sh;
    private IDictionary<Guid, WorkflowApplication>
              _incomingRequests;
    private IDictionary<Guid, WorkflowApplication>
              _outgoingRequests;
    public MainWindow()
     {
       InitializeComponent();
       ApplicationInterface._app = this;

       _incomingRequests = new Dictionary<Guid,
              WorkflowApplication>();
       _outgoingRequests = new Dictionary<Guid,
              WorkflowApplication>();
        }
```

```
private void Window_Loaded(object sender, RoutedEventArgs e)
    {
     // Open the config file and get the name for this branch
     // and its network address
     Configuration config = ConfigurationManager
        .OpenExeConfiguration(ConfigurationUserLevel.None);
     AppSettingsSection app =

(AppSettingsSection)config.GetSection("appSettings");
     string adr = app.Settings["Address"].Value;

     // Display the Branch name on the form
     lblBranch.Content = app.Settings["Branch Name"].Value;

     // Create the ServiceHost
     _sh = new ServiceHost(typeof(ClientService));

     // Add the Endpoint
    string szAddress = "http://localhost:" + adr +
              "/ClientService";
    System.ServiceModel.Channels.Binding bBinding = new
               BasicHttpBinding();

_sh.AddServiceEndpoint(typeof(ITrainingProcessWpf),
                 bBinding, szAddress);

     // Open the ServiceHost to listen for messages
    _sh.Open();

     // Test the ListBoxTextWriter
     //ListBoxTextWriter lbtw = new ListBoxTextWriter();
     //lbtw.Write("This is a test");
  }

private void Window_Unloaded(object sender,RoutedEventArgs e)
  {
    // Terminate the service host
     _sh.Close();
  }
```

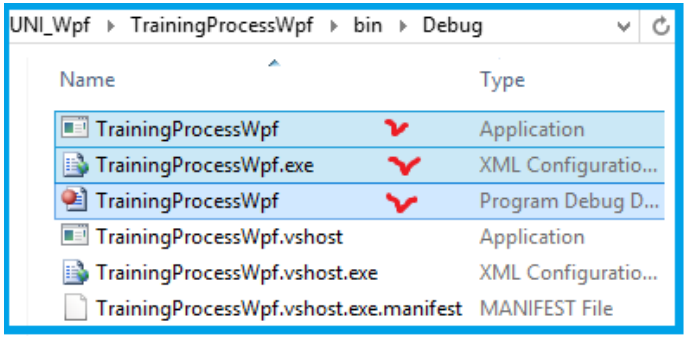
```
private void btnRequest_Click(object sender,RoutedEventArgs e)
  {
   // Setup a dictionary object for passing parameters
    Dictionary<string, object> parameters = new
          Dictionary<string, object>();
    parameters.Add("Lector", txtLector.Text);
    parameters.Add("Sagani", txtSagani.Text);
    parameters.Add("Jgupi", txtJgupi.Text);
    parameters.Add("Writer", new
           ListBoxTextWriter(lstEvents));
   WorkflowApplication i =
    new WorkflowApplication(new SendRequest(),parameters);

    _outgoingRequests.Add(i.Id, i);
     i.Run();
   }

   // Handle the Reserve button click event
private void Reserve(object sender, RoutedEventArgs e)
   {
     // Get the instanceID from the Tag property
     FrameworkElement fe = (FrameworkElement)sender;
     Guid id = (Guid)fe.Tag;
     ResumeBookmark(id, true);
   }

   // Handle the Cancel button click event
private void Cancel(object sender, RoutedEventArgs e)
   {
     // Get the instanceID from the Tag property
       FrameworkElement fe = (FrameworkElement)sender;
       Guid id = (Guid)fe.Tag;
       ResumeBookmark(id, false);
   }

private void ResumeBookmark(Guid id, bool bReserved)
   {
     WorkflowApplication i = _incomingRequests[id];
     try
```

```
      {
        i.ResumeBookmark("GetResponse", bReserved);
      }
      catch (Exception e)
         {
            AddEvent(e.Message);
         }
    }

public void RequestTraining(TrainingRequest request)
   {
   // Setup a dictionary object for passing parameters
    Dictionary<string, object> parameters = new
              Dictionary<string, object>();
    parameters.Add("request", request);
    parameters.Add("Writer", new
             ListBoxTextWriter(lstEvents));
WorkflowApplication i =
  new WorkflowApplication(new ProcessRequest(),
parameters);
     request.InstanceID = i.Id;
    _incomingRequests.Add(i.Id, i);
     i.Run();
 }
public void RespondToRequest(TrainingResponse response)
  {
    Guid id = response.RequestID;
    WorkflowApplication i = _outgoingRequests[id];
    try
     {
       i.ResumeBookmark("GetResponse", response);
     }
    catch (Exception e2)
      {
         AddEvent(e2.Message);
      }
   }
public void AddNewRequest(TrainingRequest request)
  {
    this.requestList.Dispatcher.BeginInvoke
```

```
        (new Action(() =>
             this.requestList.Items.Add(request)));
  }
public ListBox GetEventListBox()
  {
     return this.lstEvents;
  }
private void AddEvent(string szText)
        {
            lstEvents.Items.Add(szText);
        }
    }
}
```

2.5. აპლიკაციის ამუშავება

როგორც წინა თავის შემთხვევაში, აქაც საჭიროა აპლიკაციის რამდენიმე კოპიოს ერთად გაშვება, თითოეული თავისი კონფიგურაციის ფაილის ვერსიით. თავიდან საჭიროა F6 კლავიშის ამოქმედება solution-ის აღსადგენად და კომპილატორის შენიშვნების აღმოსაფხვრელად. შევქმნათ ახალი Branch-ფოლდერი UNI_Wpf-კატალოგის ქვეშ (TrainingProcessWpf-თან), რომელიც იძახებს ფილიალებს. შემდეგ დავაკოპიროთ ფილიალის Branch-ფოლდერში ფაილები TrainingProcessWpf-ის bin->Debug კატალოგიდან, რომლებიც 2.11 ნახაზზეა ნაჩვენები.



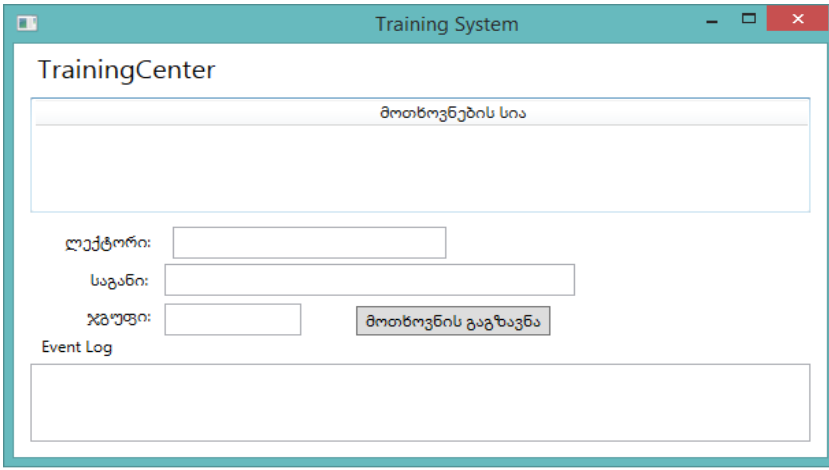
ნახ.2.11

გავხსნათ TrainingProcessWpf.exe.config ფაილი (ფილიალის ქვეფოლდერში) და შევასწოროთ შემდეგნაირად:

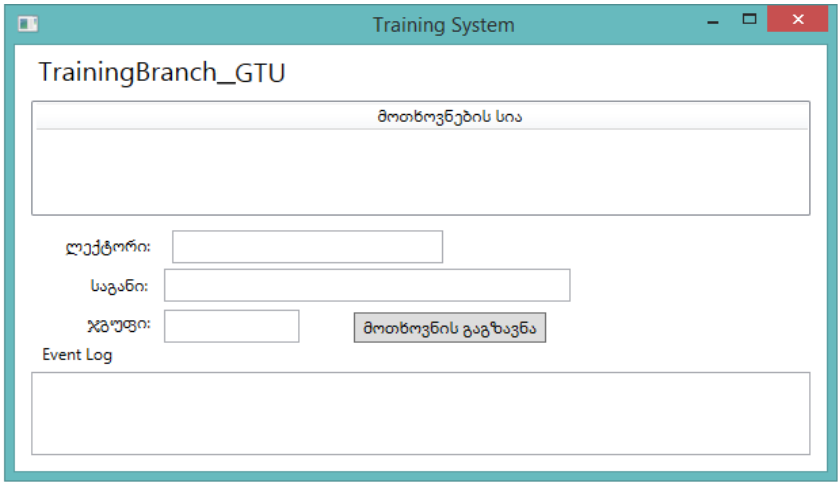
```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="Branch Name" value="TrainingBranch_GTU"/>
    <add key="ID" value="{43E6DADD-4751-4056-8BB7-
7459B5C361AB}"/>
    <add key="Address" value="8730"/>
    <add key="Request Address" value="8000"/>
  </appSettings>

</configuration>
```

შენიშვნა: თუ შედეგი შეცდომითაა მიღებული, უნდა ვცადოთ აპლიკაციის გაშვება *ადმინისტრატორის უფლებებით* (იხ. თავი 1).

ადმინისტრატორის უფლებით ავამუშავოთ ორივე აპლიკაცია. ტრეინინგ-ცენტრის (ნახ.2.12) და მისი ფილიალის (ნახ.2.13). ფანჯრები განვაცალკევოთ ხედვის გაუმჯობესების მიზნით.

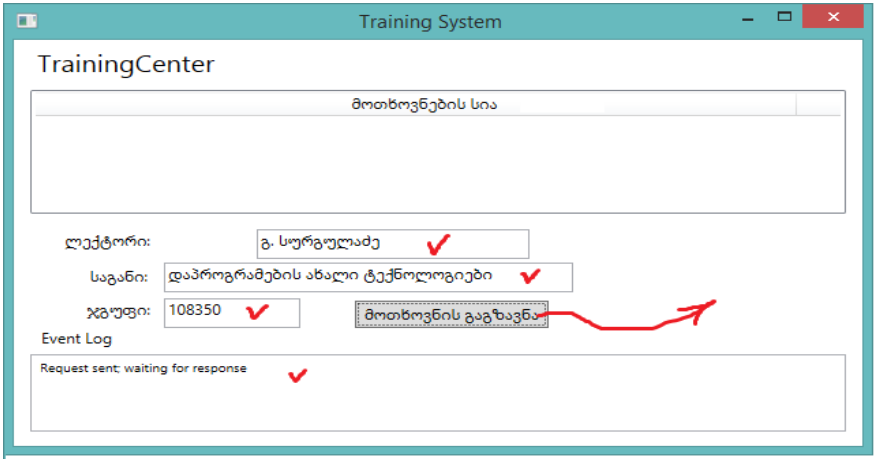


ნახ.2.12. ტრეინინგ-ცენტრის მომხმარებლის ინტერფეისი

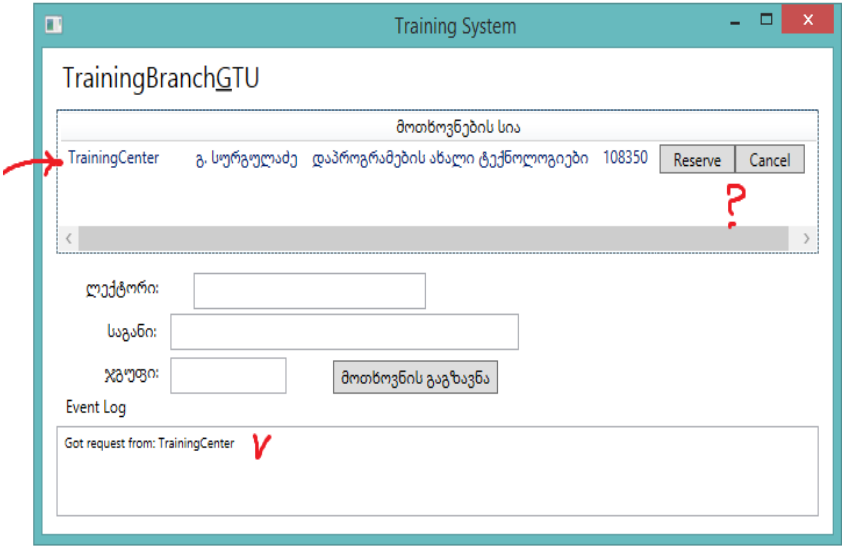


ნახ.2.13. ტრეინინგ-ფილიალის მომხმარებლის ინტერფეისი

ერთ-ერთ აპლიკაციაში, მაგალითად, ტრეინინგ-ცენტრისაში, შევიტანოთ ლექტორი, საგანი და ჯგუფი და ავამოქმედოთ ღილაკი **„მოთხოვნის გაგზავნა“**. მოთხოვნა უნდა გამოჩნდეს მეორე ფანჯრის **„მოთხოვნების სიაში“** (ნახ.2.14).

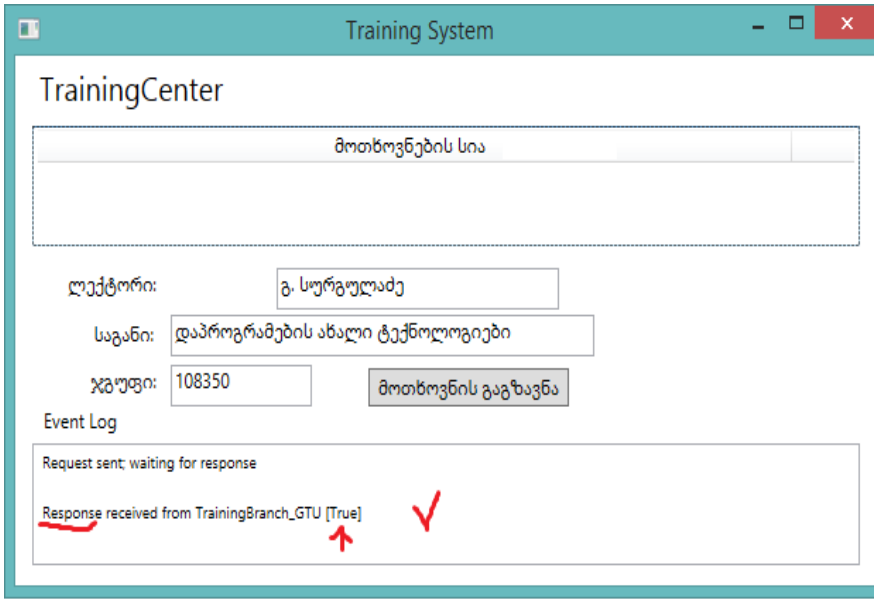


ნახ.1.14-ა. ცენტრიდან გაიგზავნა მოთხოვნა ფილიალში



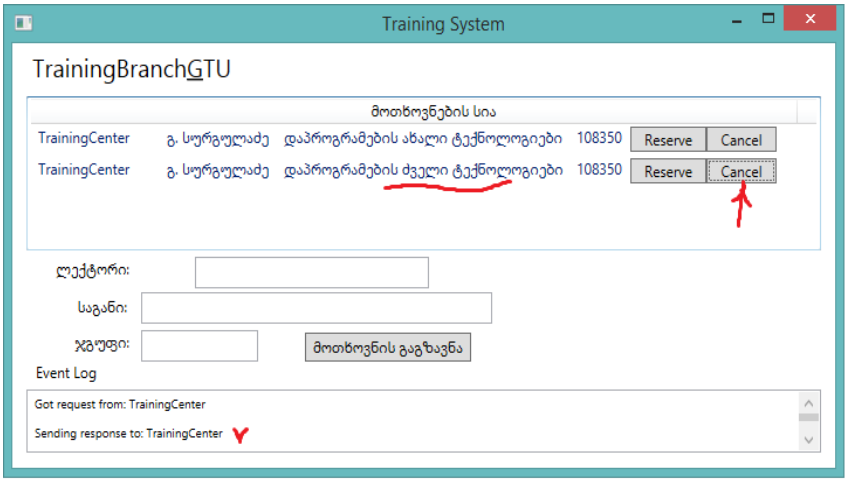
ნახ.1.14-ბ. ფილიალში მიღებულია მოთხოვნა ცენტრიდან

ფილიალის სასწავლო პროცესის მენეჯერმა უნდა აირჩიოს ღილაკი Reserve (true-დაჯავშნულია) ან Cancel (false - უარყოფილია) მოთხოვნა. ტრეინინგ-ცენტრი ელოდება პასუხს (ინფორმაცია Event Log-ში). დავუშვათ, რომ ფილიალმა დადებითად გადაწყვიტა და დააჭირა Reserve ღილაკს. ამის შემდეგ ფილიალის აპლიკაციიდან გაიგზავნება თანხმობის შეტყობინება, რომელიც გამოჩნდება ტრეინინგ-ცენტრის ქვედა, Event Log-ის ფანჯრის მოვლენების ჟურნალში, რომ პასუხი მიღებულია (true).

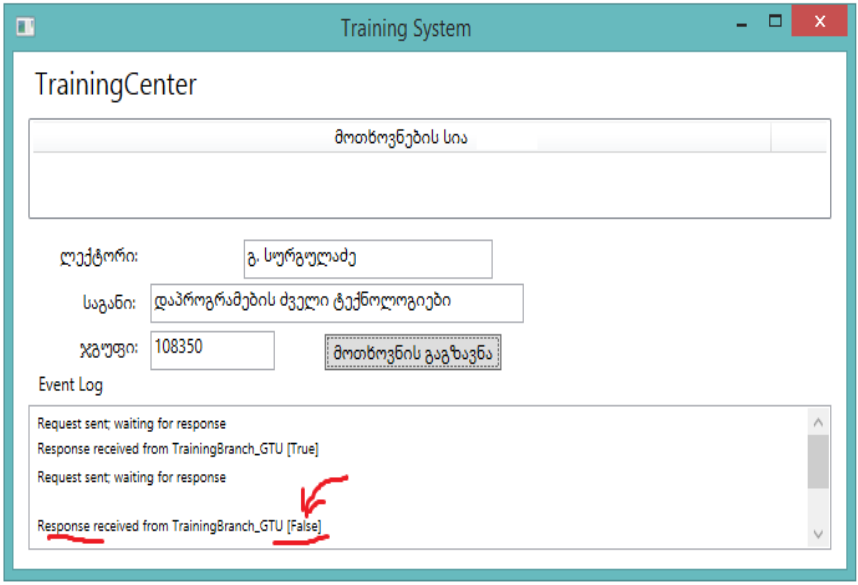


ნახ.2.15. დადებითი (true) შეტყობინების მიღება ტრეინინგ-ცენტრში

მეორე მოთხოვნაზე, რომელიც 2.16-ა,ბ ნახაზზეა ნაჩვენები, მიღებულია ფილიალიდან უარყოფითი პასუხი (false).



ნახ.2.16-ა. ფილიალში მოხდა Cancel ღილაკის არჩევა



ნახ.2.16-ბ. ცენტრში მიღებულია [false] უარყოფითი პასუხი

III თავი
Web-სერვისების დაპროგრამება

სამუშაო პროცესები (Workfkow-სერვისები, ბიზნესპროცესები, დოკუმენტბრუნვა) შეიძლება განთავსდეს ვებ-სერვისში, რომელიც უზრუნველყოფს იდეალურ საშუალებას მუშა პროცესის გადაწყვეტილების მისაწოდებლად არა-მუშა პროცესის კლიენტებისთვის, როგორიცაა ვებ-აპლიკაციები [17,21].

ვებ-სერვისი იღებს მოთხოვნას, ასრულებს მის სათანადო გადამუშავებას და აბრუნებს პასუხს. ეს, ბუნებრივია, სრულდება Receive და Send ქმედებებით, რომლებიც ჩვენ წინა თავებში განვიხილეთ. ვინაიდან ეს აქტიურობები ინტეგრირებულია Windows Communication Foundation (WCF) -თან, ჩვენ შეგვიძლია ადვილად შევქმნათ WCF სერვისები.

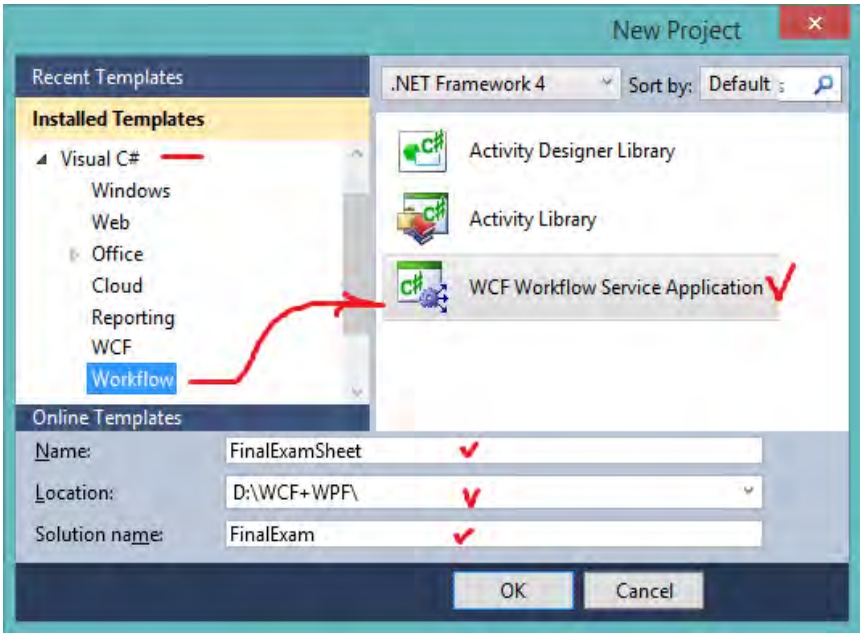
ამ თავში განიხილება საუნივერსიტეტო მართვის საინფორმაციო სისტემის შექმნა ან არსებული სისტემის მოდიფიკაცია ინტეგრაციის პრინციპების საფუძველზე მოითხოვს ამ საპრობლემო სფეროს ობიექტ-ორიენტირებული, პროცეს-ორიენტირებული და სერვის-ორიენტირებული მიდგომების კომპლექსურ გამოყენებას [1,2,19].

3.1. სამუშაო პროცესის (Workflow-) სერვისის შექმნა (ლაბ-10)

მიზანი: სამუშაო პროცესის Web-სერვისის შექმნის პროცედურების შესწავლა **W**orkflow და WCF ტექნოლოგიების საფუძველზე.

ვიხილავთ ამოცანას უნივერსიტეტის სასწავლო პროცესის საპრობლემო სფეროსთვის. კერძოდ, მომხმარებელთა ინტერფეისის დაპროგრამებას „საგამოცდო უწყისების“ მართვის სისტემისთვის.

ავამუშავოთ Visual Studio 2010 (13/15) და შევქმნათ ახალი პროექტი WCF Workflow Service Application template გამოყენებთ. შევიტანოთ პროექტის სახელი FinalExamSheet და Solution-ის სახელი FinalExam.

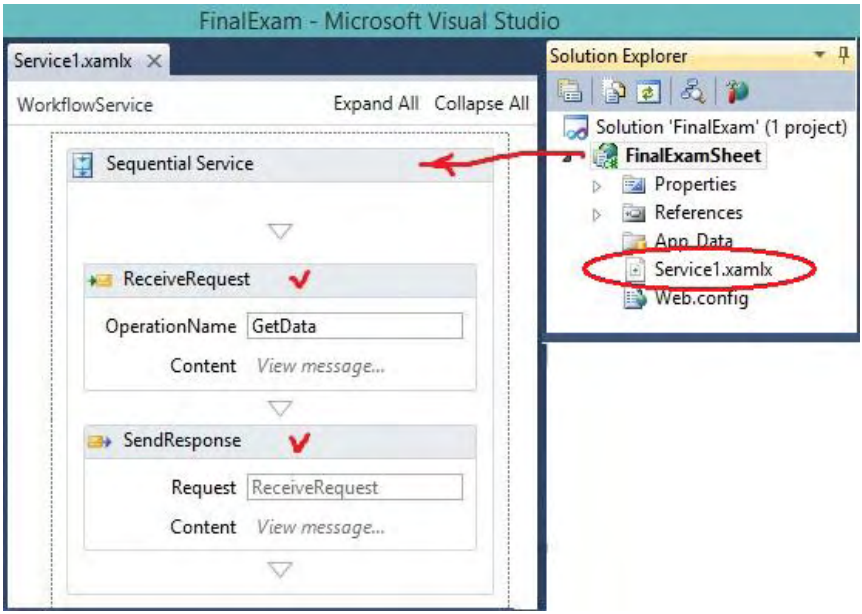


ნახ.3.1. WCF Workflow Service Application პროექტის შექმნა

შეიქმნება საინიციალიზაციო სამუშაო პროცესი Sequence, რომელიც შეიცავს Receive და SendReply ქმედებებს, როგორც 3.2 ნახაზზეა ნაჩვენები: Sequential Service (სერვისების მიმდევრობა), Receive Request (მოთხოვნის მიღება), Send Response (პასუხის გაგზავნა).

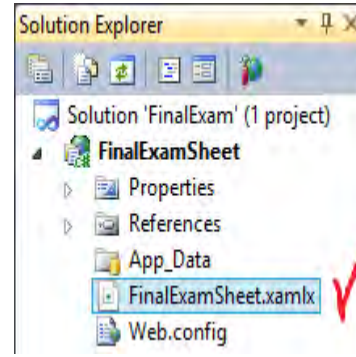
თავიდან საჭიროა ამ ქმედებების კონფიგურირება სერვისის კონტრაქტის განსაზღვრის მიზნით, რომელსაც ისინი დააკმაყოფილებს. შემდეგ უნდა დაემატოს დამუშავების სამუშაო პროცესი,

რომელიც განხორციელდება Receive და SendReply ქმედებებს შორის.



ნახ.3.2. საწყისი სამუშაო პროცესის თანამიმდევრობა

შაბლონი შექმნის საწყის მუშა პროცესს ფაილში სახელით Service1.xamlx. შევცვალოთ Solution Explorer-ში ეს სახელი FinalExamSheet.xamlx -ით (ნახ.3.3).



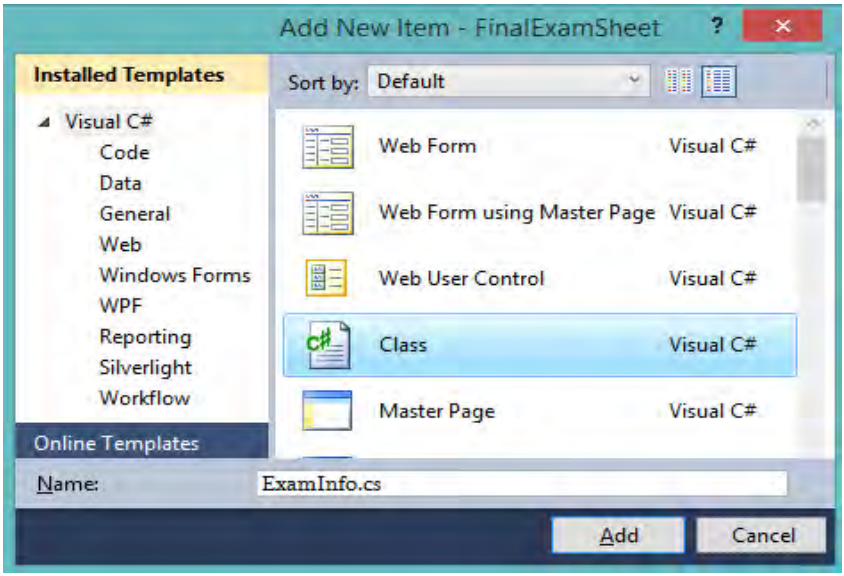
ნახ.3.3

სერვისი, რომელიც მაგალითის სახით უნდა შევქმნათ, საფინალო გამოცდისთვის ძებნის შესაბამის უწყისებს მითითებული აკადემიურ ჯგუფის, ლექტორის და აკადემიური საგნის მიხედვით.

ელექტრონული საგამოცდო უწყისები ეკუთვნის ტრეინინგ-ცენტრს (ან ფაკულტეტის დეკანატს), ხოლო მათი დამუშავება ხდება ცენტრის ან ფილიალების (ან კათედრების) ლექტორების მიერ.

3.2. სერვისის კონტრაქტის განსაზღვრა

Solution Explorer-ში, მარჯვენა ღილაკით FinalExamSheet პროექტზე ავირჩიოთ: Add->Class სახელით BookInfo.cs (ნახ.3.4), ამ კლასის კოდის ტექსტი მოცემულია 3.1 ლისტინგში.



ნახ.3.4. ExamInfo კლასის შექმნა

```
// ----- ლისტინგი 3.1 -----
using System;
using System.Collections.Generic;
using System.Web;
using System.Runtime.Serialization;
using System.ServiceModel;
```

```
namespace FinalExamSheet
{
// სერვისის კონტრაქტის განსაზღვრა  IFinalExamUckisi,
რომელიც
/ შედგება ერთი მეთოდისგან  -  LookupUckisi () ------
    [ServiceContract]
    public interface IFinalExamUckisi
    {
        [OperationContract]
        UckisiInfoList LookupUckisi(UckisiSearch request);
    }
//-- მოთხოვნის შეტყობინების განსაზღვრა, UckisiSearch ---
    [MessageContract(IsWrapped = false)]
    public class UckisiSearch
    {
      private String _Jgupi;
      private String _Sagani;
      private String _Lector;
      public UckisiSearch() { }
      public UckisiSearch(String sagani, String lector,
                  String jgupi)
        {
           _Sagani = sagani;
           _Lector = lector;
           _Jgupi = jgupi;
        }
      #region Public Properties
      [MessageBodyMember]
      public String Sagani
        {
            get { return _Sagani; }
            set { _Sagani = value; }
        }
      [MessageBodyMember]
      public String Lector
        {
            get { return _Lector; }
            set { _Lector = value; }
        }
      [MessageBodyMember]
```

```
    public String Jgupi
      {
          get { return _Jgupi; }
          set { _Jgupi = value; }
      }
      #endregion Public Properties
}

// --- ExamInfo კლასის განსაზღვრა -----
[MessageContract(IsWrapped = false)]
public class ExamInfo
{
  private Guid _ExamUckisiID;
  private String _Jgupi;
  private String _Sagani;
  private String _Lector;
  private String _Status;
  public ExamInfo() { }
  public ExamInfo(String sagani, String lector,
            String jgupi, String status)
    {
       _Sagani = sagani;
       _Lector = lector;
       _Jgupi = jgupi;
       _Status = status;
       _ExamUckisiID = Guid.NewGuid();
    }
  #region Public Properties
  [MessageBodyMember]
  public Guid ExamUckisiID
    {
        get { return _ExamUckisiID; }
        set { _ExamUckisiID = value; }
    }
  [MessageBodyMember]
  public String Sagani
    {
        get { return _Sagani; }
        set { _Sagani = value; }
    }
  [MessageBodyMember]
```

```
        public String Lector
          {
              get { return _Lector; }
              set { _Lector = value; }
          }
        [MessageBodyMember]
        public String JGUPI
          {
              get { return _Jgupi; }
              set { _Jgupi = value; }
          }
        [MessageBodyMember]
        public String status
          {
              get { return _Status; }
              set { _Status = value; }
          }
        #endregion Public Properties
    }
//--- საპასუხო შეტყობინების განსაზღვრა: UckisiInfoList ---
  [MessageContract(IsWrapped = false)]
  public class UckisiInfoList
    {
       private List<ExamInfo> _UckisiList;
       public UckisiInfoList()
        { _UckisiList = new List<ExamInfo>(); }
       [MessageBodyMember]
       public List<ExamInfo> UckisiList
        { get { return _UckisiList; } }
    }
}
```

სერვისის კონტრაქტი IFinalExamUckisi შეიცავს ერთადერთ მეთოდს LookupBook(). იგი მონაცემებს გადასცემს UckisiSearch კლასს, რომელსაც აქვს სხვადასხვა თვისებები, საჭირო უწყისის მოსაძებნად, მაგალითად ლექტორი და საგნის დასახელება. ის აბრუნებს უკან UckisiInfoList კლასს, რომელიც შეიცავს ExamInfo კლასების კოლექციას. შესაძლებელია პირველი თავის ნახვა, სადაც

ახსნილია ServiceContract, MessageContract და MessageBodyMember ატრიბუტების გამოყენება.

F6 -ით ავამოქმედოთ პროექტია და მივიღოთ მუშა solution.

გამეორება:

MessageContract ატრიბუტი მიუთითებს, რომ ეს კლასი ჩართულ იქნება SOAP ბარათში. SOAP–ის გამოყენების დროს შეტყობინებები გადაიცემა XML–ის მსგავსი ფორმატირებადი ენით. ეს უზრუნველყოფს კლიენტებსა და სერვერს შორის მაღალი ხარისხის ურთიერთქმედების პლატფორმას. SOAP არის სტანდარტული პროტოკოლი, რომლის მხარდაჭერაც აქვს WCF –ს.

არსებობს აგრეთვე MessageBodyMember ატრიბუტი მის ყოველ პაბლიკ–თვისებაზე. ეს აუცილებელია WCF–ფენისთვის რათა სწორად დაფორმატდეს SOAP შეტყობინება.

WCF–ის ბოლო წერტილის განსაზღვრისთვის არსებობს ინფორმაციის სამი პორცია, რომლებიც მითითებულ უნდა იქნას: მიერთება (binding), მისამართი და კონტრაქტი.

მიერთება მიუთითებს იმ პროტოკოლს, რომელიც გამოიყენება (მაგალითად, HTTP, TCP ან სხვ.).

მისამართი მიუთითებს თუ სად უნდა ვიპოვოთ ბოლო წერტილი, და მისამართის ტიპს, რომლის გამოყენება დამოკიდებულია მიერთებაზე. მაგალითად, HTTP–მიერთებისას უნდა მიეთითოს URL, ხოლო TCP–თვის მისამართი იქნება სერვერის სახელი ან IP–მისამართი.

კონტრაქტი განისაზღვრება ServiceContract–ით, რომელიც არის ინტერფეისი. იგი განსაზღვრავს მეთოდებს, რომლებიც მიწვდომადია ბოლო წერტილში.

ამგვარად, ჩვენ განვსაზღვრეთ შეტყობინებები, რომლებიც გადაიცემა სერვის–მეთოდების მიერ პარამეტრების სახით.

3.3. Receive და SendReply კონფიგურირება (ლაბ-11)

მიზანი: სამუშაო პროცესის სერვისისთვის მოთხოვნის მიღების (ReceiveRequest) და პასუხის გაგზავნის (SendResponse) ქმედებების დაპროგრამების შესწავლა.

FinalExamSheet პროექტში გავხსნათ FinalExamSheet.xamlx ფაილი და ავირჩიოთ "ReceiveRequest" ქმედება (ნახ.3.5-ა).

თვისებათა ფანჯარაში ServiceContract თვისებას აქვს default-მნიშვნელობა {http://tempuri.org/}IService. შევცვალოთ IService სტრიქონი IFinalExamUckisi -ით. შევიტანოთ OperationName როგორც LookupUckisi (ნახ.3.5-ბ).

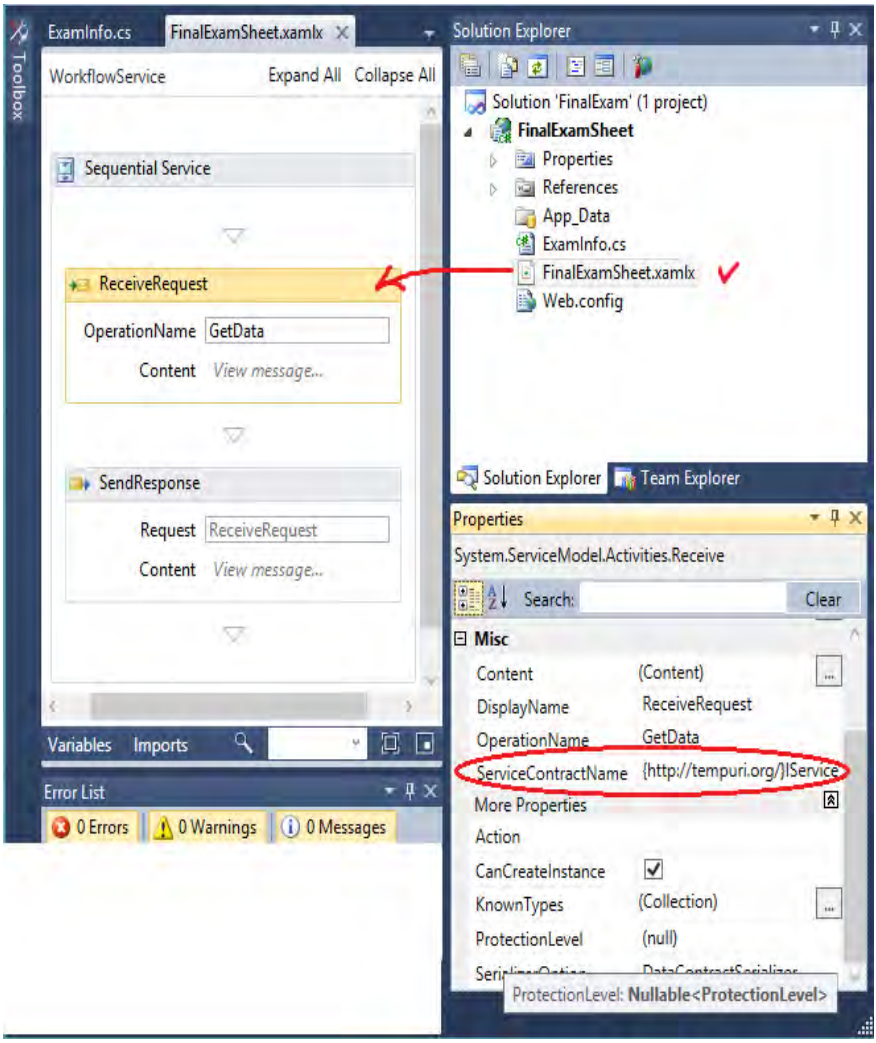
ეკრანზე WorkflowService-დიზაინერში ქვედა მარცხენა კუთხეში დავკლიკოთ Variables ღილაკი. გამოჩნდება შაბლონი ორი ცვლადის შესაქმნელად (ნახ.3.6).

გადასამუშავებელი ცვლადი (handle variable) გამოიყენება პასუხის კორელაციისთვის იმ ეგზემპლართან, რომელმაც გააგზავნა მოთხოვნა.

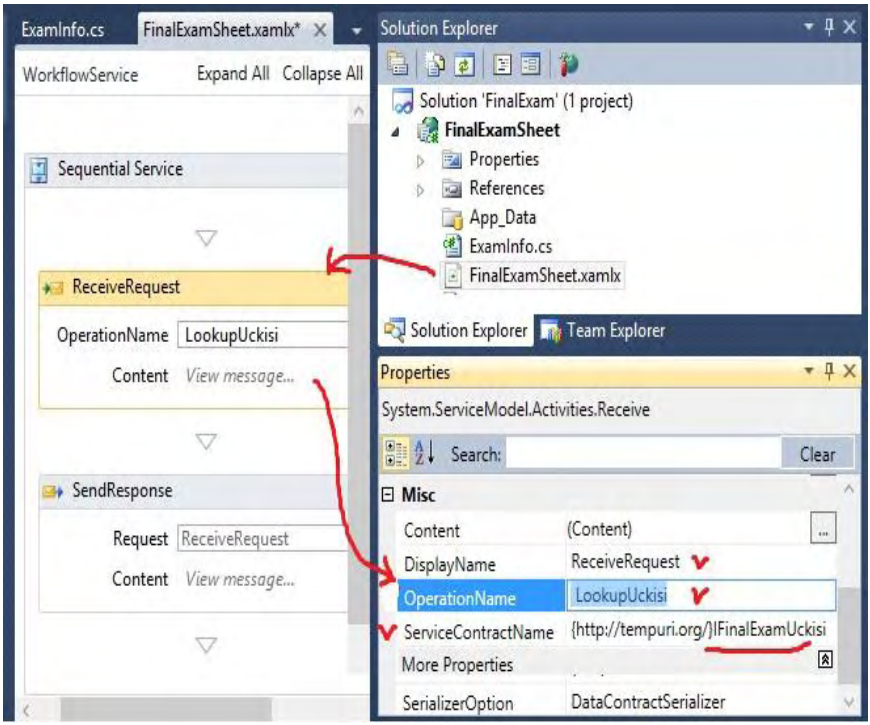
მონაცემთა ცვლადი იქმნება გადასაცემი მონაცემებისთვის (ინფორმაციისთვის). გავასუფთავოთ ცვლადების არე (data) და შევქმნათ ორი ახალი ცვლადის სახელი.

პირველისთვის Name-ში ავირჩიოთ სახელი search და ტიპი - Browse for Typs. ახალ დიალოგურ ფანჯარაში FinalExamSheet ნაკრები გავაფართოვოთ და ავირჩიოთ UckisiSearch კლასი OK-ით (ნახ.3.7).

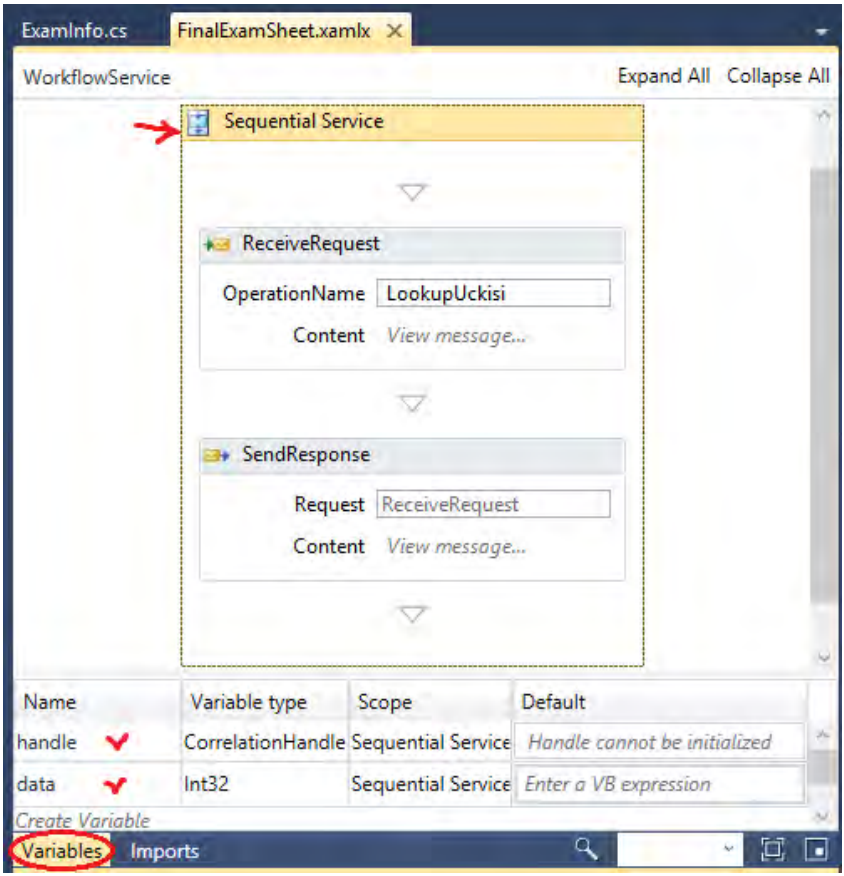
მეორე ცვლადისთვის შეიტანეთ Name: result. ტიპი შეირჩევა Browse-დან, UckisiInfoList კლასით. მიიღება 3.8 ნახაზზე მოცემული შემთხვევა.



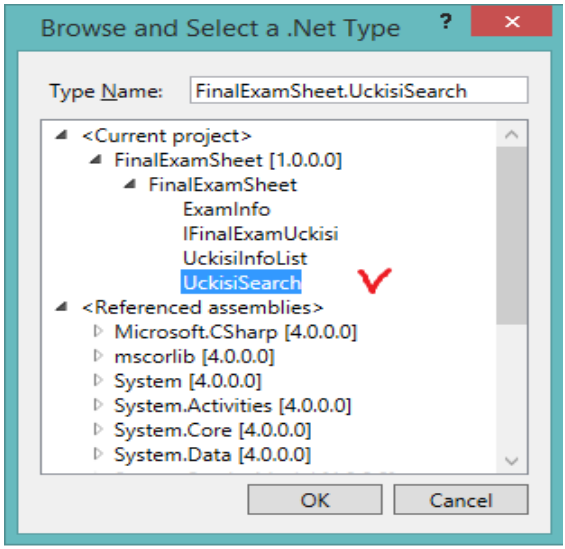
ნახ.3.5-ა. საწყისი მდგომარეობა



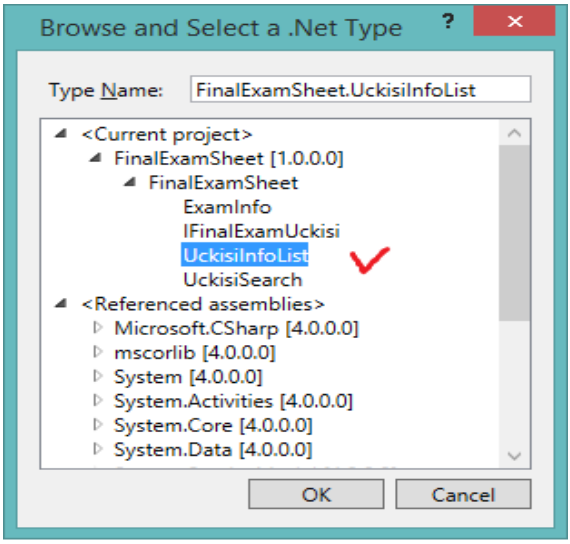
ნახ.3.5-ბ. საბოლოო მდგომარეობა


ExamInfo.cs
FinalExamSheet.xamlx
WorkflowService
Expand All
Collapse All
Sequential Service
ReceiveRequest
OperationName
LookupUckisi
Content
View message...
SendResponse
Request
ReceiveRequest
Content
View message...
Name
Variable type
Scope
Default
handle
CorrelationHandle
Sequential Service
Handle cannot be initialized
data
Int32
Sequential Service
Enter a VB expression
Create Variable
Variables
Imports

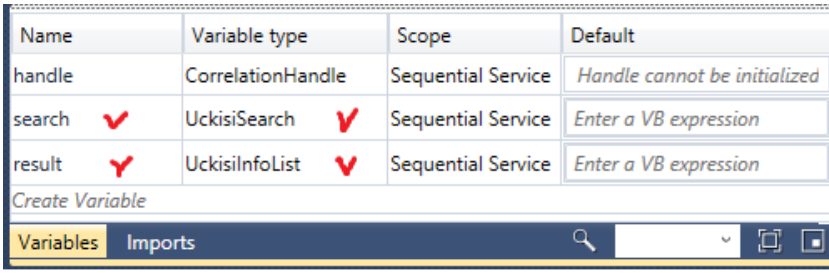
ნახ.3.6.



ნახ.3.7. UckisiSearch ცვლადის ტიპის არჩევა



ნახ.3.8. UckisiInfoList ცვლადის ტიპის არჩევა



ნახ.3.9. ცვლადები განისაზღვრა სამუშაო პროცესისთვის

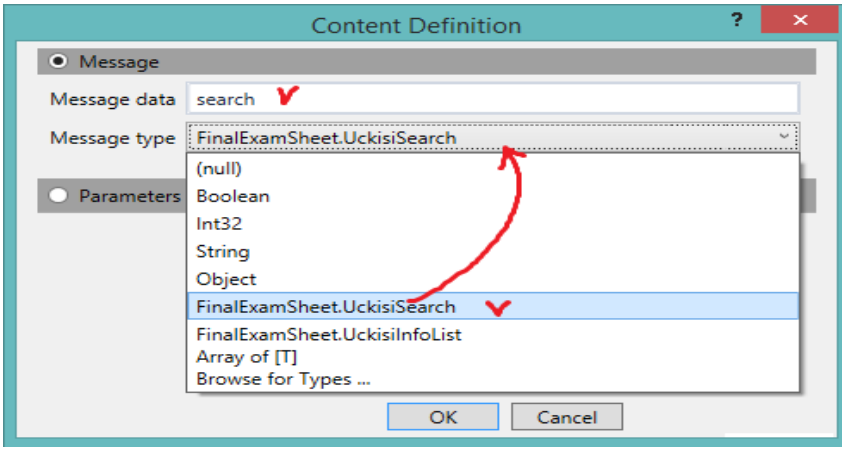
მუშა პროცესების დიზაინერში „ReceiveRequest“ (მოთხოვნების მიღების) ქმედებას აქვს view message (შეტყობინების ნახვის) ლინკი Content (შინაარსის) თვისებისთვის. მისი ამოქმედებით იხსნება დიალოგური ფანჯარა შემავალი შეტყობინების დასადგენად (შეიძლება ასევე სამ-წერტილიანი ღილაკის გამოყენებაც, თვისების გვერდით). შესასვლელი განისაზღვრება ორი ხერხით: შეტყობინებით ან პარამეტრების ერთობლიობით (მოგვიანებით ჩვენ განვიხილავთ მეორე ხერხსაც).

ახლა დავაკვირდეთ, რომ რადიობუტონის გადამრთველი შეტყობინებისთვის სწორადაა არჩეული.

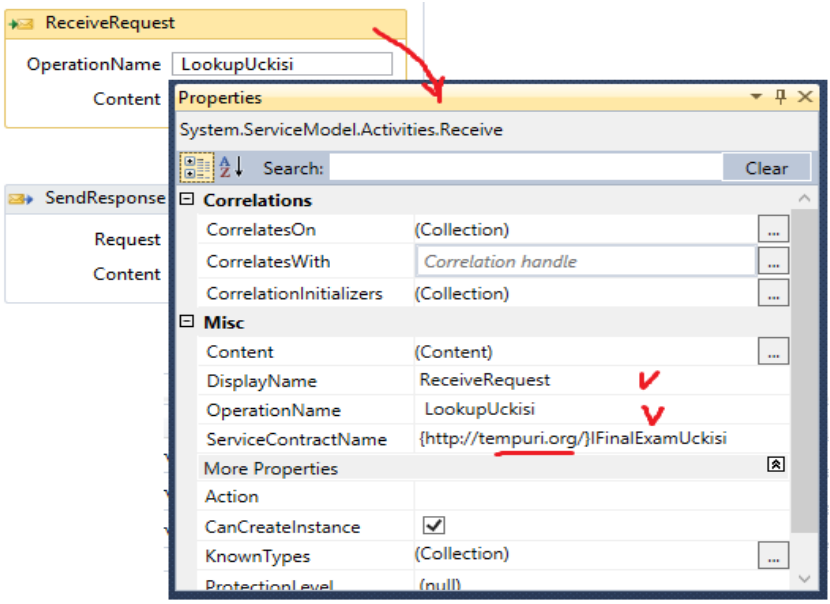
Message data თვისებისთვის შევიტანოთ search. ის მიუთითებს, რომ შემომავალი შეტყობინება უნდა ინახებოდეს search ცვლადში. Message ტიპისთვის კი FinalExamSheet.UckisiSearch-ს ვირჩევთ. დიალოგური ფანჯარა მოცემულია 3.10 ნახაზზე.

თვისებების ფანჯარა უნდა გამოიყურებოდეს 3.11 ნახაზზე ნაჩვენები სახით.

შემდეგ ვირჩევთ ქმედებას „SendResponse“ და ავამოქმედებთ view message ლინკს. კვლავ შევამოწმოთ, რომ არჩეულია შეტყობინების გადამრთველი.

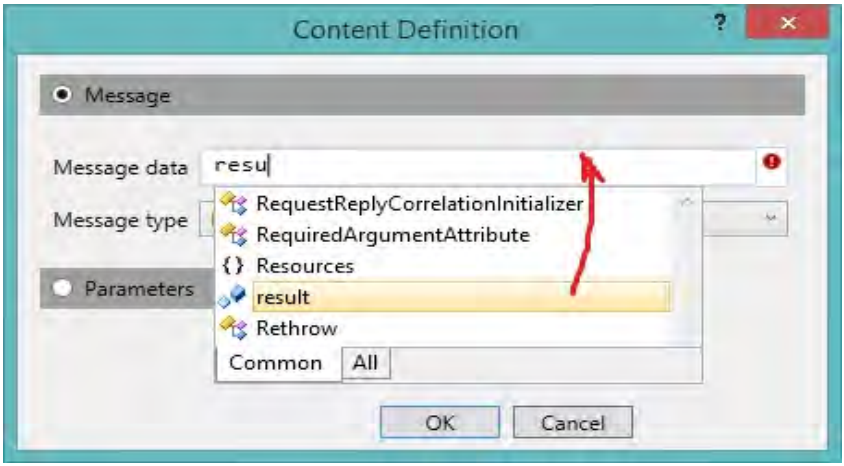


ნახ.3.10. შემავალი შეტყობინების განსაზღვრა

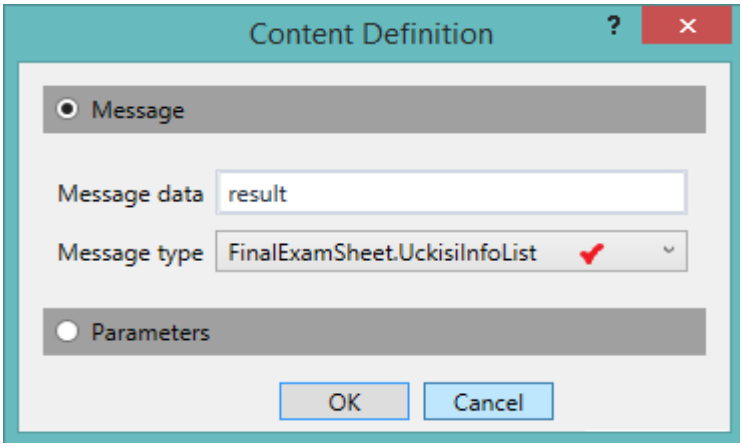


ნახ.3.11. Receive ქმედების თვისებათა ფანჯარა

Message data თვისებისთვის შევიტანოთ result (ნახ.3.12). Message type თვისებისთვის ვირჩევთ FinalExamSheet.UckisiInfoList კლასს (ნახ.3.13).



ნახ.3.12. SendResponce ქმედების Content ფანჯარაში სახელის არჩევა



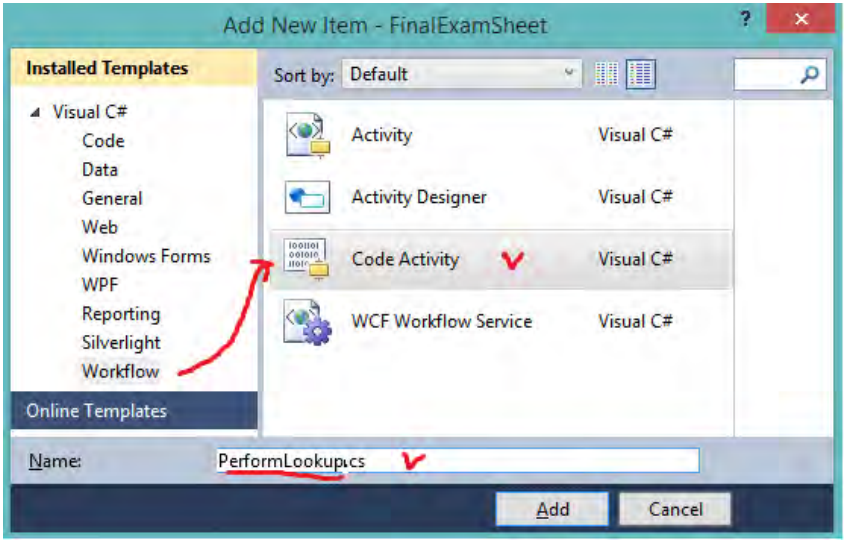
ნახ.3.13. SendResponce ქმედების Content ფანჯარაში ცვლადის ტიპის არჩევა

3.4. PerformLookup აქტიურობის შექმნა (ლაბ-12)

მიზანი: აპლიკაციის პროექტისთვის მომხმარებლის ქმედების PerformLookup (აქტიურობის-Activity) შექმნა "lookup"-ის (ძებნის) შესასრულებლად.

ფაქტობრივად, ხისტად-კოდირებულ მონაცემთა დაბრუნება (ცვლადების დონეზე) მარტივად სრულდება. რეალურ სიტუაციაში კი საჭირო მონაცემების მისაღებად პროგრამამ უნდა შეასრულოს ძებნის მოთხოვნა მონაცემთა ბაზაში.

Solution Explorer-ში FinalExamSheet პროექტზე მარჯვენა ღილაკით ვირჩევთ Add-➤NewItem და დიალოგში Workflow კატეგორიისთვის ვირჩევთ Code Activity-ს. Name-ში შევიტანთ PerformLookup.cs სახელს (ნახ.3.14).



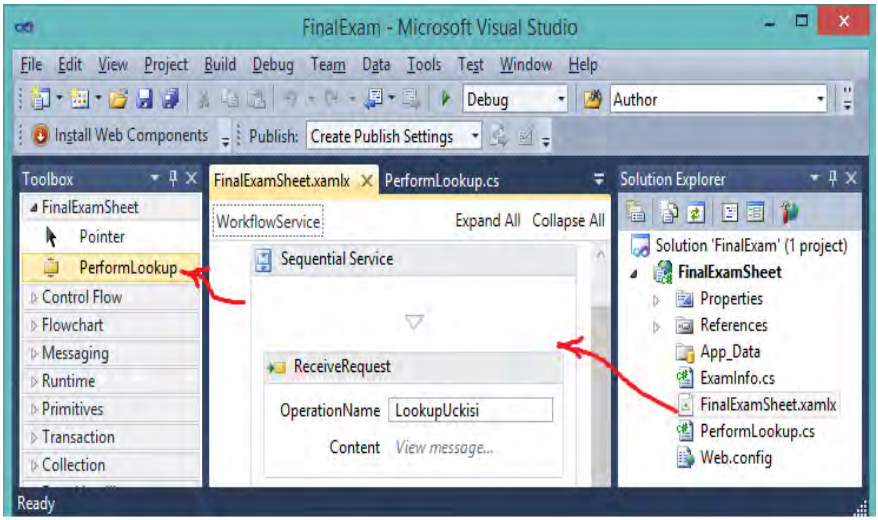
ნახ.3.14. მომხმარებლის ქმედების შექმნა

შევიტანოთ PerformLookup ქმედების რეალიზაციისთვის 3.2 ლისტინგში მოცემული კოდი.

```
// ლისტინგი 3.2 ----------------------------------
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Activities;
namespace FinalExamSheet
{
  public sealed class PerformLookup : CodeActivity
  {
    public InArgument<UckisiSearch> Search { get; set; }
    public OutArgument<UckisiInfoList> UckisiList {get;set;}

protected override void Execute(CodeActivityContext context)
  { string lector = Search.Get(context).Lector;
    string sagani = Search.Get(context).Sagani;
    string jgupi  = Search.Get(context).Jgupi;

    UckisiInfoList l = new UckisiInfoList();

    l.UckisiList.Add(new ExamInfo(sagani, lector,  jgupi, "Available"));
    l.UckisiList.Add(new ExamInfo(sagani, lector,  jgupi, "CheckedOut"));
    l.UckisiList.Add(new ExamInfo(sagani, lector,  jgupi, "Missing"));
    l.UckisiList.Add(new ExamInfo(sagani, lector,  jgupi, "Available"));
     UckisiList.Set(context, l);       }
  }
}
```

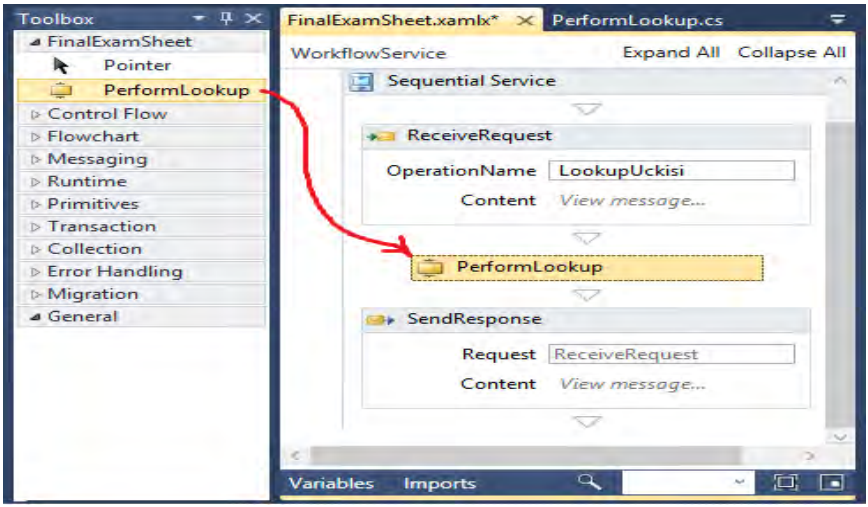
F6-ით განვახორციელოთ აპლიკაციის აღდგენა. გავხსნათ `FinalExamSheet`.xamlx ფაილი.

გასათვალიწინებელია, რომ მომხმარებლის PerformLookup ქმედება მოთავსდა ToolBox-ზე (ნახ.3.15).



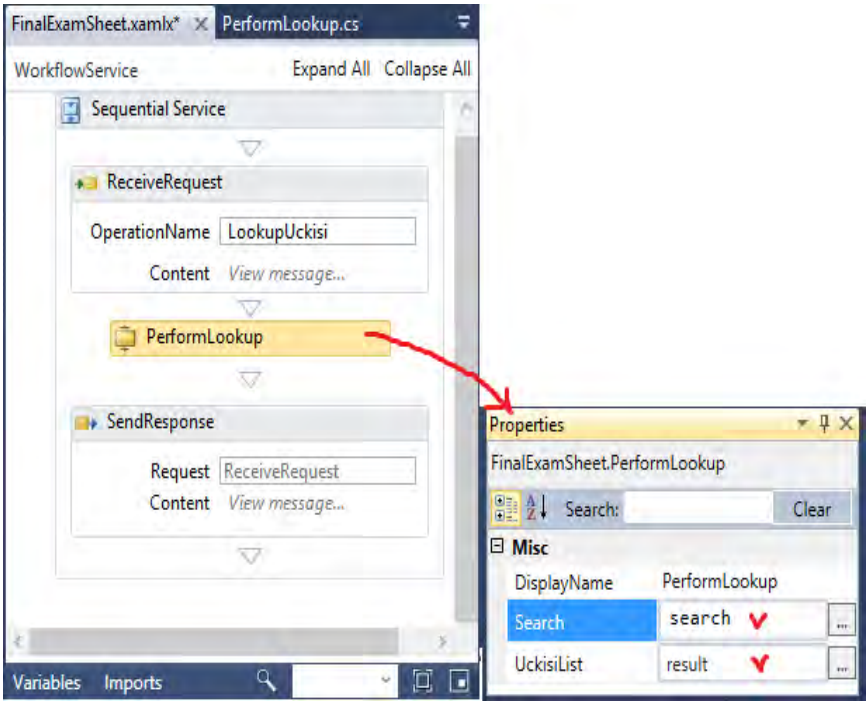
ნახ.3.15

გადმოვიტანოთ PerformLookup ქმედება „ReceiveRequest” და „SendResponse” ქმედებებს შორის (ნახ.3.16).



ნახ.3.16

ავირჩიოთ PerformLookup ქმედება. Properties-ის ფანჯარაში, UckisiList თვისებისთვის შევიტანოთ result, ხოლო Search თვისებისთვის კი search (ნახ.3.17).

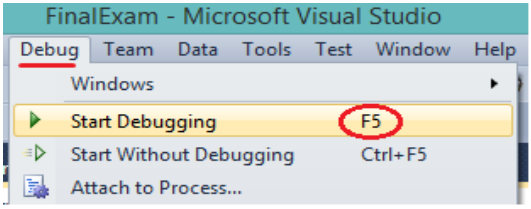


ნახ.3.17

3.5. სერვისის ტესტირება (ლაბ-13)

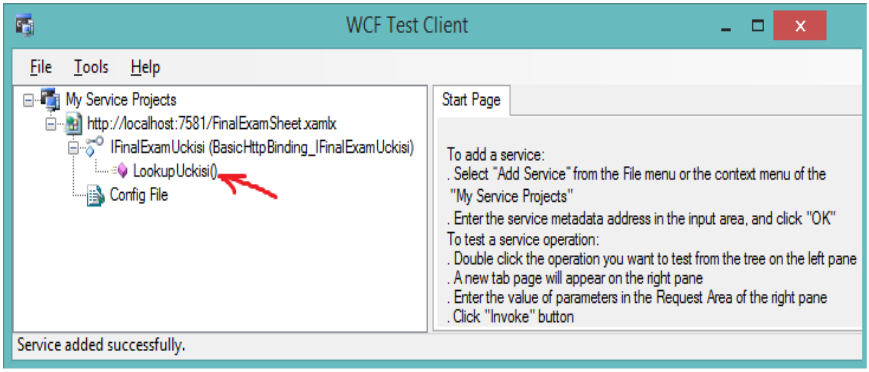
მიზანი: აპლიკაციის ვებ-სერვისის ტესტირების პროცესის შესწავლა.

წინა პარაგრაფში აგებული აპლიკაციის პროექტისთვის განვახორციელოთ ტესტირების პროცესი. F5-ით ჩავატაროთ სერვისის გამართვა (Debug), როგორც 3.18 ნახაზზეა ნაჩვენები.



ნახ.3.18

ვინაიდან ეს Web-სერვისია, Visual Studio ავტომატურად აამუშავებს WCF Test Client-ს. ეს მეტად მოსახერხებელი უტილიტაა. იგი ჩატვირთავს Web-სერვისებს და აღმოაჩენს მეთოდებს, რომლებიც გათვალისწინებულია. ისინი ჩანს 3.19 ნახაზის მარცხენა პანელზე.



ნახ.3.19. WCF Test Client ინიციალიზაციის ფანჯარა

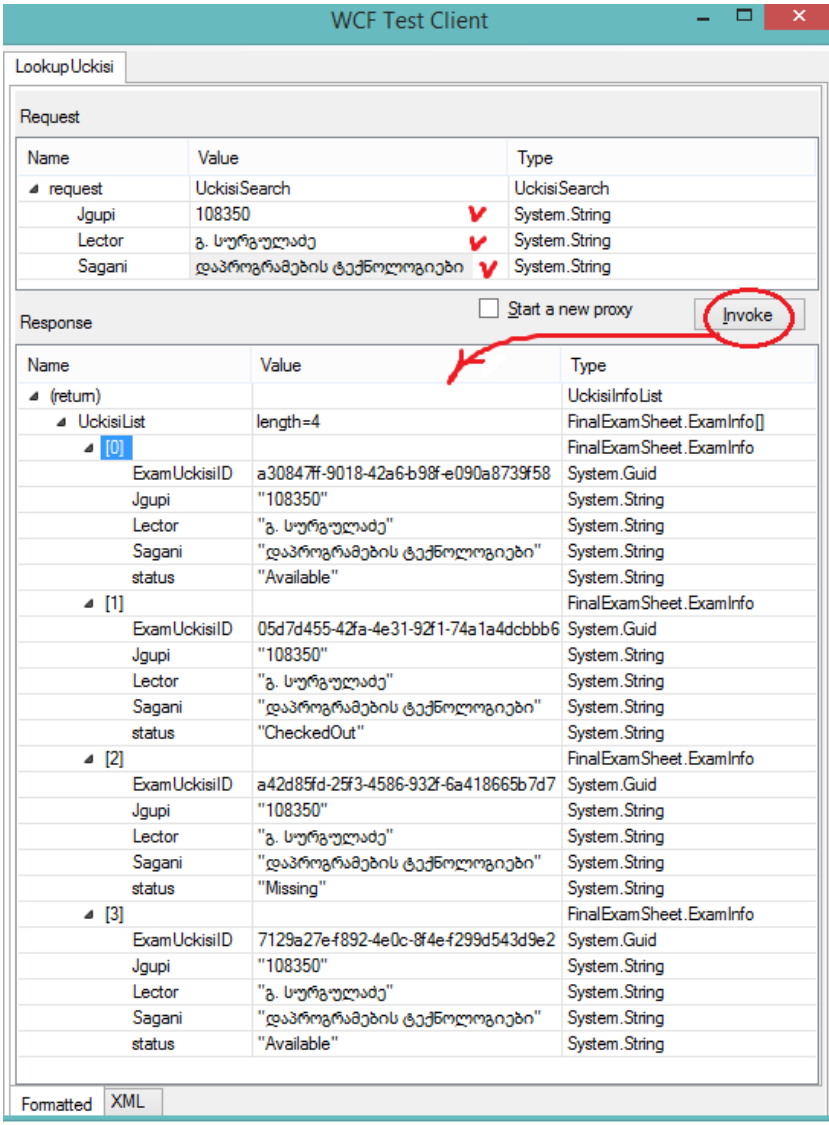
LookupUckisi() მეთოდზე მაუსის 2-ჯერ დაჭერით მარჯვენა პანელის ზედა ნაწილში გამოიყოფა ადგილი შემოსული შეტყობინების განსათავსებლად. იგი მზადაა რთული შეტყობინებებისთვისაც, რომლებიც შეიცავს კლასების და თვისებების კოლექციებს.

შევიტანოთ ლექტორის გვარი, ჯგუფის-ნომერი, საგნის დასახელება. შემდეგ ავამოქმედოთ Invoke ღილაკი. გამოჩნდება შედეგები, რომლებიც ანალოგიურია 3.20 ნახაზის.

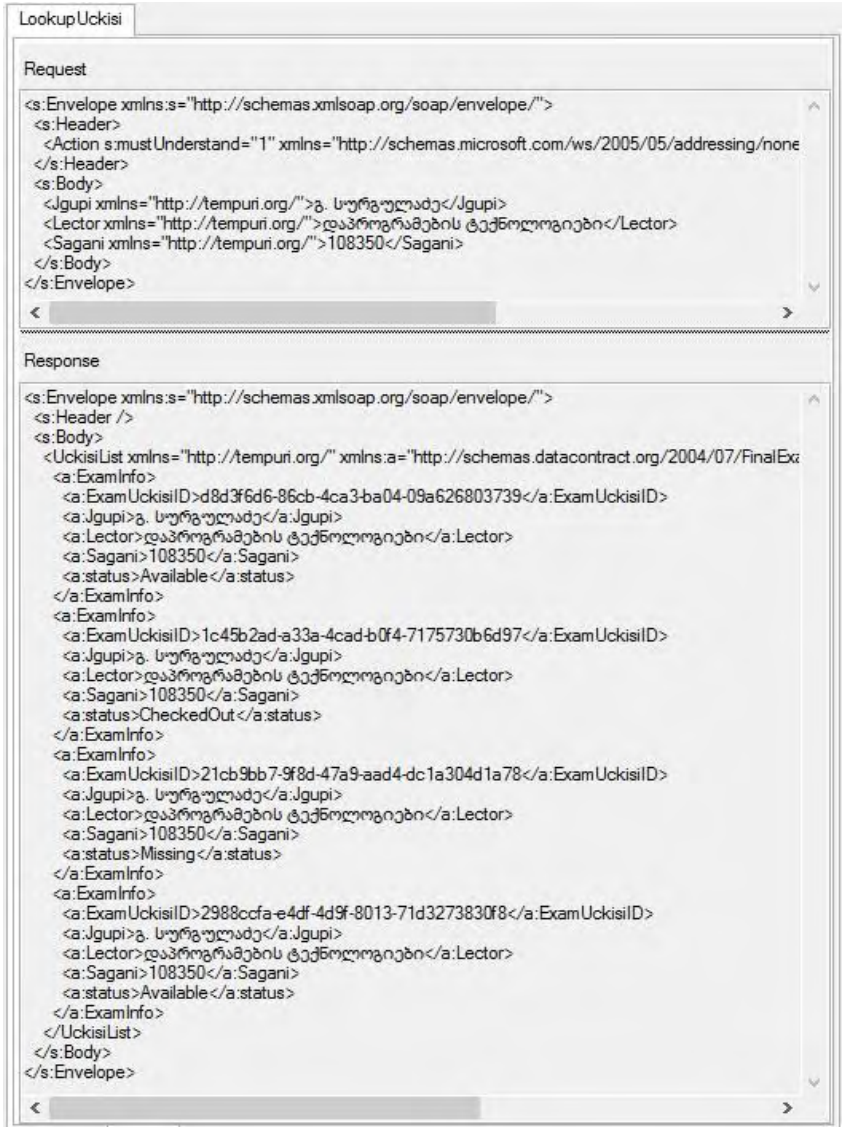
სერვისი აბრუნებს ExamInfo-ს ოთხ კლასს. მაგალითად, ნახაზზე მესამე ჩანაწერი გვიჩვენებს, რომ მოცემულ კონკრეტულ ელემენტს აქვს სტატუსი Missing (დაკარგული, ამოვარდნილი).

3.21 ნახაზზე ნაჩვენებია შედეგების შესაბამისი XML ტექსტები.

შენიშვნა: თუ .xamlx ფაილი არის მიმდინარე ფაილი Visual Studio-ში, როცა F5-ვაჭერთ, მაშინ WCF Test Client გაიშვება, როგორც აქაა ნაჩვენები. თუ სხვა ფაილია აქტიური, მაშინ გამოჩნდება შესაბამისი კატალოგი და შედეგები. ის უნდა დაიხუროს და გააქტიურდეს ჩვენთვის საჭირო .xamlx ფაილი.



ნახ.3.20. WCF Client ტესტის სერვისის შედეგების ნახვა



ნახ.3.21. შესაბამისი XML კოდი

3.6. პარამეტრების გამოყენება (ლაბ-14)

მიზანი: სერვისის შექმნის პროცესის შესწავლა, რომელიც გამოიყენებს პარამეტრებს შეტყობინებების მაგივრად.

წინა პროექტის მიხედვით ვებ-სერვისში შესვლა განისაზღვრებოდა როგორც კლასი MessageContract ატრიბუტით. ესაა ტიპური გზა WCF სერვისის გამოსაძახებლად. მიუხედავად ამისა, იმის მაგივრად, რომ შეიქმნას შეტყობინების ერთი კლასი, რომელიც ყველა შემავალ მონაცემს შეიცავდეს, შესაძლებელია მათი გადაცემა მუშა პროცესების სერვისებისათვის ცალკეული პარამეტრების სახით. ამის სადემონსტრაციოდ შევქმნათ მეორე იდენტური სერვისი, რომელიც გამოიყენებს პარამეტრებს შეტყობინებათა მაგივრად.

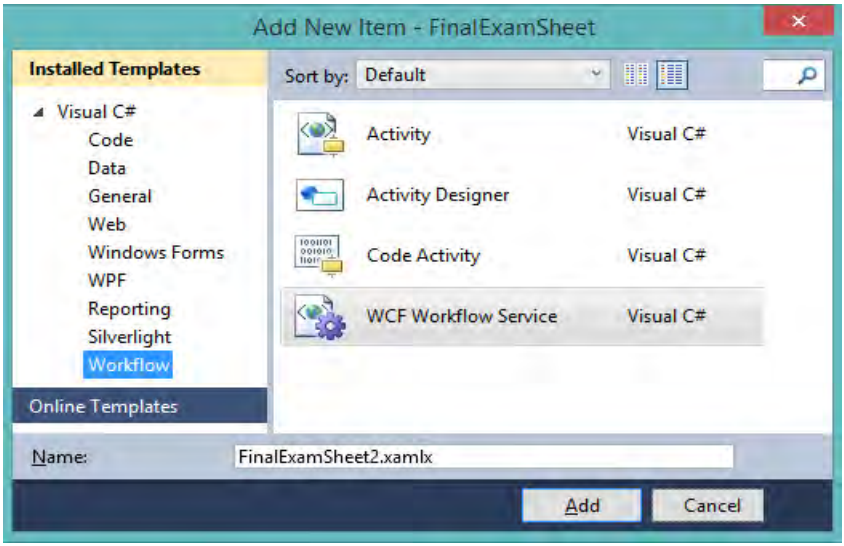
3.6.1. მეორე სერვისის შექმნა

Solution Explorer-ში მარჯვენა ღილაკს ვაჭერთ FinalExamSheet პროექტზე და ვირჩევთ Add -➢ New Item.

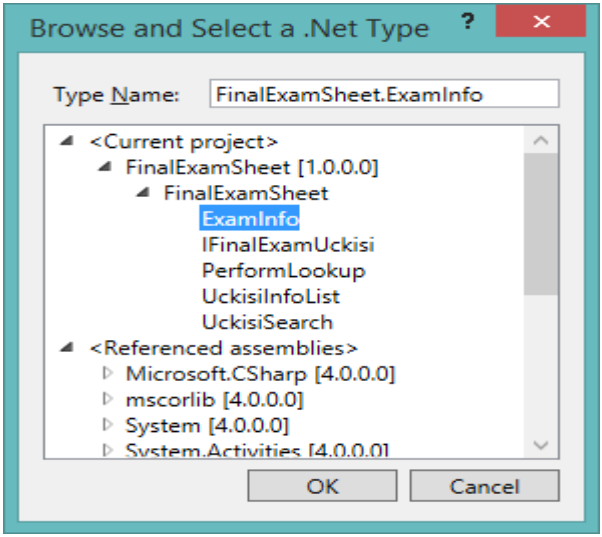
ამ დიალოგში ვირჩევთ WCF Workflow Service შაბლონს,, რომელიც Workflow კატეგორიაშია. 3.22 ნახაზზე ნაჩვენებია ეს, სახელით Name: **FinalExamSheet2.xamlx**.

მუშა პროცესის დიზაინერში ქვემოთ მარცხნივ ავამოქმედოთ ცვლადების ღილაკი Variables. რამდენიმე ცვლადი უკვე შექმნილია პირველი სერვისის შექმნის დროს. წავშალოთ data ცვლადები და შევქმნათ ახალი ცვლადი, სახელით result. ცვლადის ტიპისთვის (type) ავირჩიოთ ArrayOf<T>. გამოჩნდება დიალოგური ფანჯარა <T> ტიპის ასარჩევად. ვირჩევთ Browse-ს და FinalExamSheet კრებულიდან ExamInfo კლასს (ნახ.3.23).

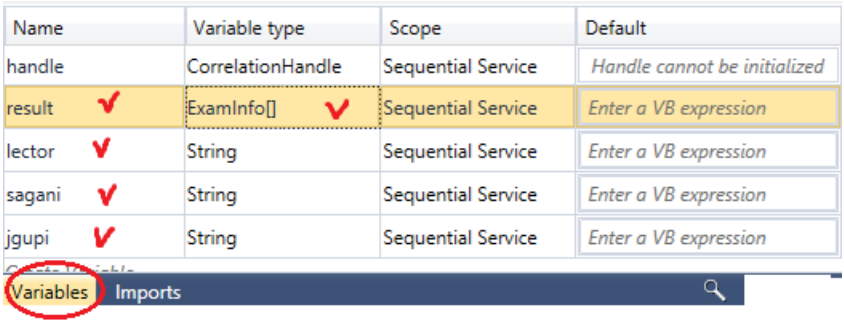
კიდევ დავამატოთ სამი String ტიპის ცვლადი სახელებით: lector, sagani და jgupi. 3.24 ნახაზზე ნაჩვენებია ცვლადების სია.



ნახ.3.22. WCF მუშა პროცესის სერვისის შექმნა



ნახ.3.23



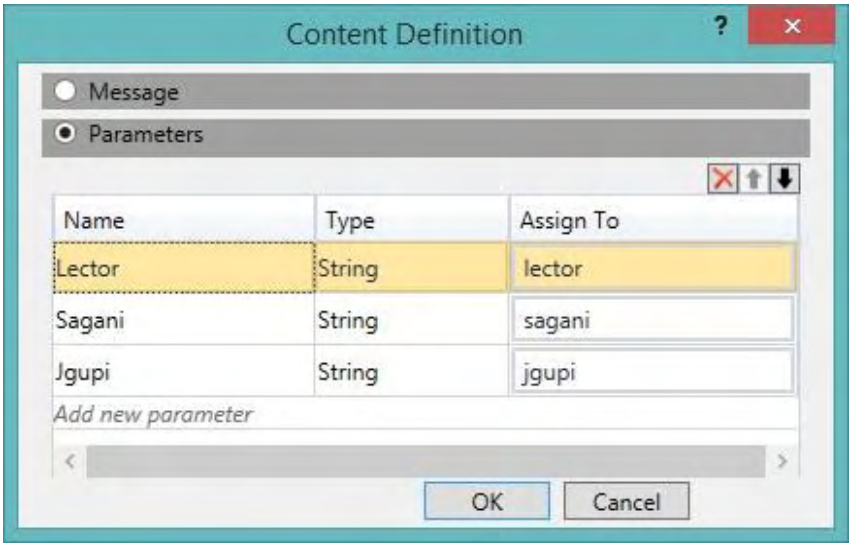
ნახ.3.24. ცვლადების სია

Properties-ის ფანჯარაში ServiceContract თვისებისთვის, შევცვალოთ IService კონტრაქტი IFinalExamUckisi. ოპერაციის სახელში ჩავწეროთ LookupUckisi2.

შენიშვნა: პირველი სერვისის შექმნისას CanCreateInstance თვისება დაყენდა true-ში შაბლონით. მეორე სერვისისთვის იგი არის false-ში. შეამოწმეთ და დარწმუნდით, რომ ის გადაყვანილია true-ში.

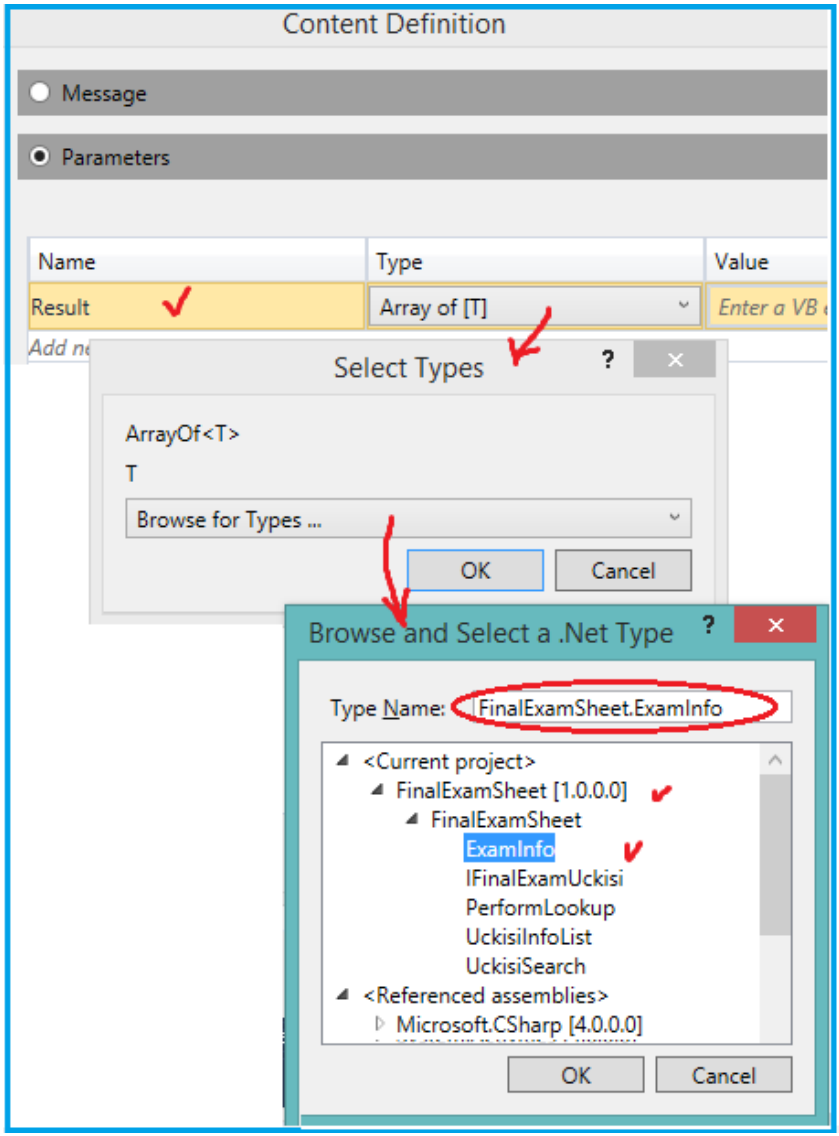
ავირჩიოთ „ReceiveRequest" ქმედება (მოთხოვნის მიღება) და ავამოქმედოთ Content ლინკი. ამჯერად დავაყენოთ გადამრთველი პოზიციაში „პარამეტრები".

პარამეტრები ყენდება ცვლადების და არგუმენტების ანალოგიურად. ავამოქმედოთ ლინკი „Add new parameter". შევიტანოთ Name როგორც Lector და დავაყენოთ Assign lector. დავამატოთ კიდევ ერთი პარამეტრი, სახელით Sagani და Assign sagani. დავამატოთ მესამე პარამეტრი სახელით Jgupi და Assign - jgupi. შევსებული გვერდს ექნება ასეთი სახე (ნახ.3.25).



ნახ.3.25. ReceiveRequest პარამეტრების სია

დავაჭიროთ „SendResponse” ქმედების Content ლინკს. ავირჩიოთ გადამრთველი პარამეტრები და დავაჭიროთ ლინკს „Add new parameter“. შევიტანოთ Name როგორც Result; ტიპისთვის ავირჩიოთ FinalExamSheet.ExamInfo[] ჩამოსაშლელი სიიდან. დიალოგურ ფანჯარას აქვს 3.26 ნახაზზე ნაჩვენები სახე.



ნახ.3.26. SendResponse პარამეტრების სია

3.6.2. მოდიფიცირებული PerformLookup ქმედების შექმნა

მომხმარებლის PerformLookup ქმედება, რომელიც ჩვენ შევქმენით პირველი სერვისისთვის, იყენებს UckisiSearch კლასს როგორც შესასვლელ არგუმენტს და აბრუნებს უკან UckisiInfoList კლასს. ახლა ჩვენ უნდა შევქმნათ სხვა სამომხმარებლო ქმედება, რომელიც იყენებს ცალკეულ პარამეტრებს.

Solution Explorer-ში მაუსის მარჯვენა ღილაკით ვაჭერთ FinalExamSheet პროექტზე და ვირჩევთ Add-➤NewItem. შემდეგ ვირჩევთ შაბლონიდან Code Activity-ს და სახელისთვის Name: PerformLookup2.cs. ამ ქმედების რეალიზაცია ნაჩვენებია 3.3 ლისტინგში.

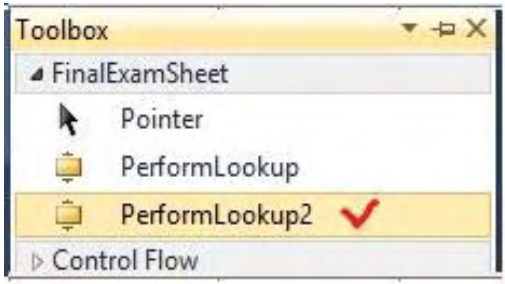
```
// ლისტინგი -- 3.3 ----PerformLookup2.cs. რეალიზაცია----
using System;
using System.Collections.Generic;
using System.Activities;

namespace FinalExamSheet
{
 // This custom activity creates a BookInfo array and
 // uses the input parameters to "lookup" the matching
 // items. The BookInfo array is returned in the output
 // parameter.
public sealed class PerformLookup2 : CodeActivity
 {
  public InArgument<String> Sagani { get; set; }
  public InArgument<String> Lector { get; set; }
  public InArgument<String> Jgupi  { get; set; }
  public OutArgument<ExamInfo[]> UckisiList { get; set;
}
```

```
  protected override void Execute
                      (CodeActivityContext context)
        {
          string lector = Lector.Get(context);
          string sagani = Sagani.Get(context);
          string jgupi  = Jgupi.Get(context);

            ExamInfo[] l = new ExamInfo[4];

 l[0]=new ExamInfo(sagani, lector,jgupi,"Available");
 l[1]=new ExamInfo(sagani, lector,jgupi,"CheckedOut");
 l[2]=new ExamInfo(sagani, lector, jgupi,"Missing");
 l[3]=new ExamInfo(sagani, lector, jgupi,"Available");
          UckisiList.Set(context, l);
        }
    }
}
```

ეს კოდი მუშაობს ისევე, როგორც პირველი, იმისგან განსხვავებით, რომ შემავალი არგუმენტები მიეწოდება ცალ-ცალკე, და შედეგები ბრუნდება მასივად, და არა კლასში.

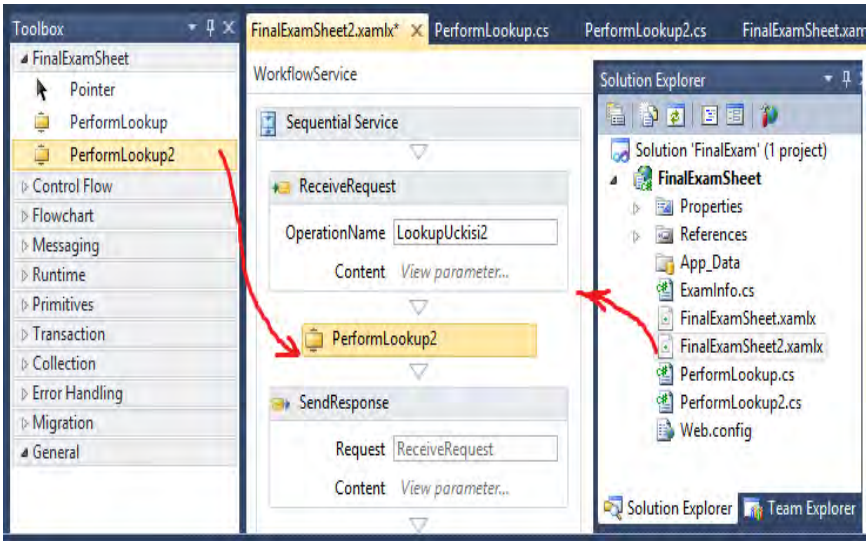
F6-ის დაჭერით განვაახლოთ solution.

ინსტრუმენტების პანელზე გამოჩნდა PerformLookup2.



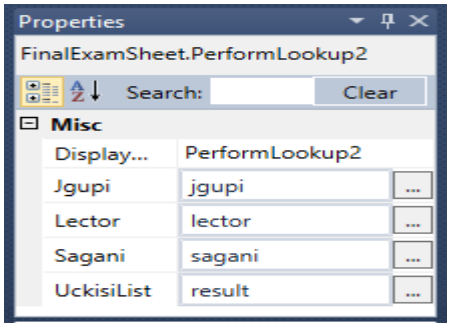
ნახ.3.27

ავირჩიოთ FinalExamSheet2.xamlx ფაილი და გადმოვიტანოთ PerformLookup2 ქმედება ინსტრუმენტების პანელიდან „Receive-Request" და „SendResponse" ქმედებებს შორის (ნახ.3.28).



ნახ.3.28

თვისებების ფანჯარაში შევიტანოთ შესაბამისი მნიშვნელობები, როგორც 3.29 ნახაზზეა ნაჩვენები.

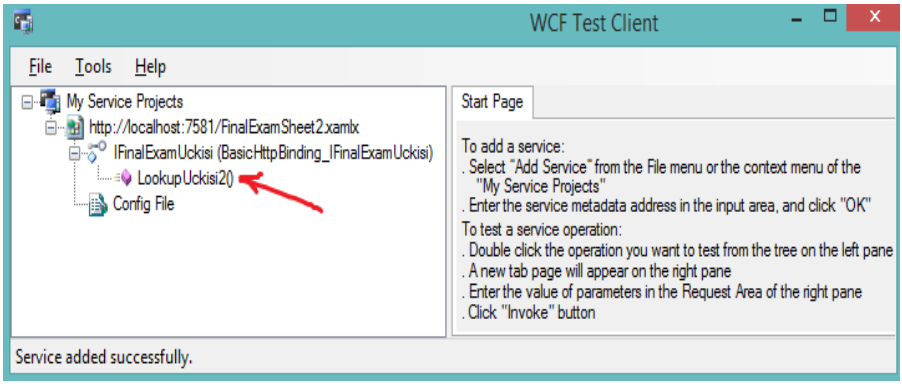


ნახ.3.29. PerformLookup2 ქმედების Properties ფანჯარა

3.6.3. სერვისის ხელახალი ტესტირება

დავრწმუნდეთ, რომ FinalExamSheet2.xamlx ფაილი არის აქტიური Visual Studio-ში და F5-ით გავუშვათ დებაგის პროცესი კოდის გასამართად.

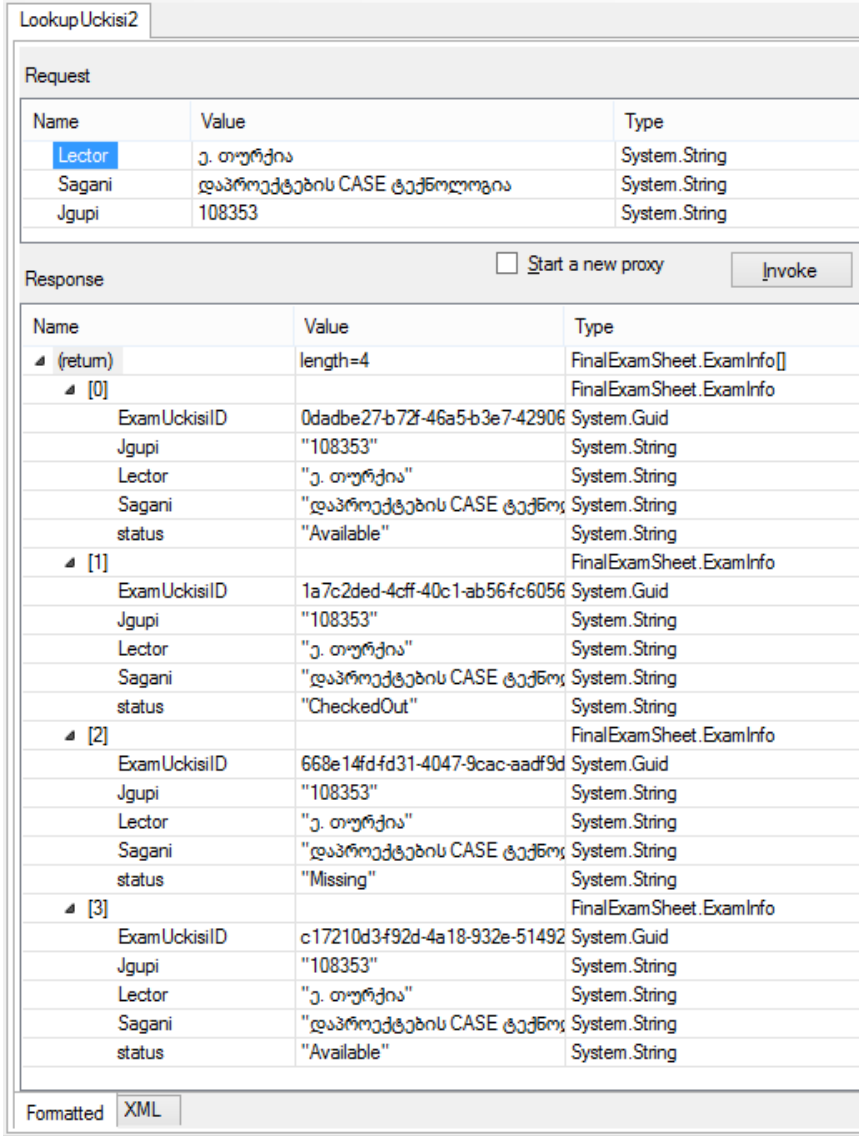
WCF Test Client უნდა ამოქმედდეს ისე, როგორც პირველი სერვისის შემთხვევაში (ნახ.3.30).



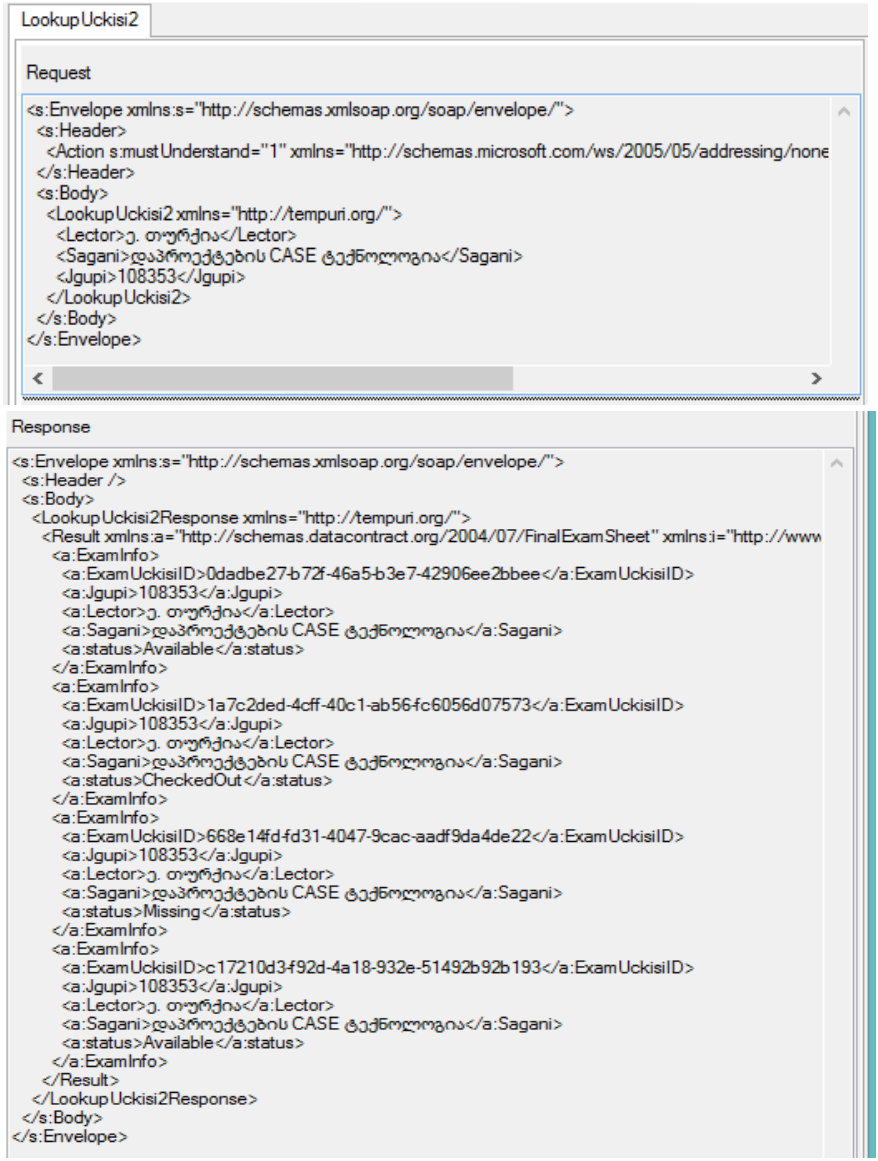
ნახ.3.30

2-ჯერ დავკლიკოთ LookupBook2() მეთოდი, შევიტანოთ მოთხოვნის მონაცემები და დავაჭიროთ Invoke ღილაკს. შედეგებს ექნება 3.30 ნახაზზე ნაჩვენები სახე.

3.31 ნახაზზე ნაჩვენებია შეაბამისი XML ტექსტები.



ნახ.3.30. WCF Test Client



ნახ.3.31. XML ტესტები

ფორმატი მცირედით განსხვადება პირველი სერვისისგან, მაგრამ ფუნქციონირებს ძირითადად მის მსგავსად. მაგალითად, ნახაზზე გაფართოებული მეორე ჩანაწერი გვიჩვენებს, რომ შემოწმებულ იქნა ეს ეგზემპლარი.

მეორე სერვისისთვის არ შეგვიქმნია კონტრაქტი მომსახურე-ბისთვის. ჩვენ მხოლოდ განვსაზღვრეთ პარამეტრები, რომლებიც გადასცა და უკან დაიბრუნა სერვისმა. კონტრაქტი სერვისის მიწოდებისთვის იქმნება ავტომატურად.

შენიშვნა: Receive/SendReply წყვილის განსაზღვრისას, გვეძლევა არჩევანის უფლება: შეტყობიმებები ან პარამეტრები. ამ ორი ვარიანტის შერევა დაუშვებელია. თუ ვიყენებთ პარამეტრებს ქმედების მისაღებად, მაშინ არ შეიძლება შეტყობინების გამოყენება SendReply ქმედებისთვის.

ამავდროულად, პარამეტრების გამოყენებისას ტიპებს არ უნდა ჰქონდეს ატრიბუტი MessageContract. წინააღმდეგ შემთხვევაში ფიქსირდება საკმაოდ ხანგრძლივი განსაკუთრებული შემთხვევა შესრულების პროცესში.

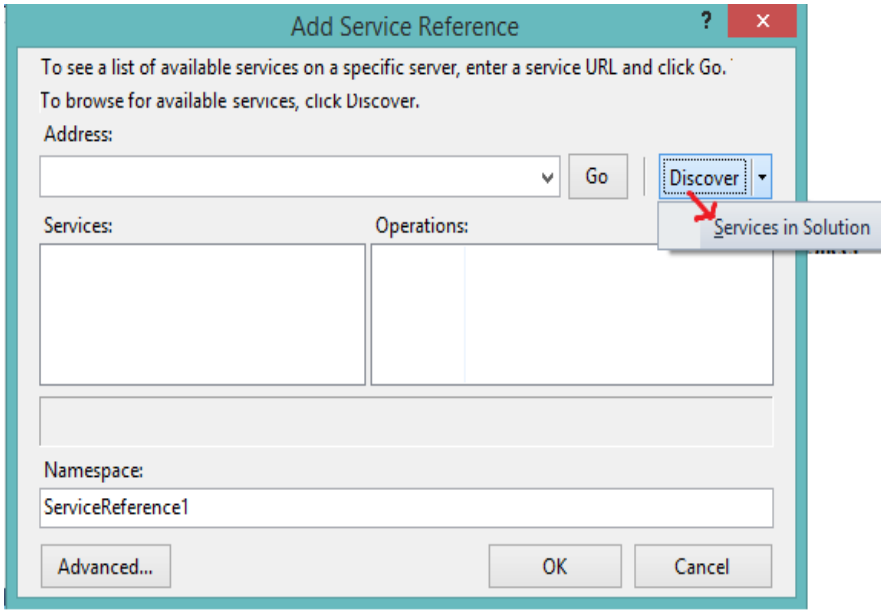
3.7. კლიენტის სამუშა პროცესის შექმნა (ლაბ-15)

მიზანი: კლიენტის აპლიკაციისთვის სამუშაო პროცესის დაპროგრამების და ვებ-სერვისის გამოძახების პროცედურების შესწავლა.

ახლა უნდა შევქმნათ Client Workflow , რომელიც გამოიძახებს Web-სერვისებს. Solution Explorer-ში Solution 'FinalExam'-ზე მარჩვენა ღილაკით ვირჩევთ Add-➤New Project. შემდეგ ავირჩევთ Workflow Console Application და შეგვაქვს პროექტის სახელი: ExamLookup (ნახ.3.32).

ნახ.3.32. ExamLookup ახალი პროექტის დამატება Solution FinalExam-ში

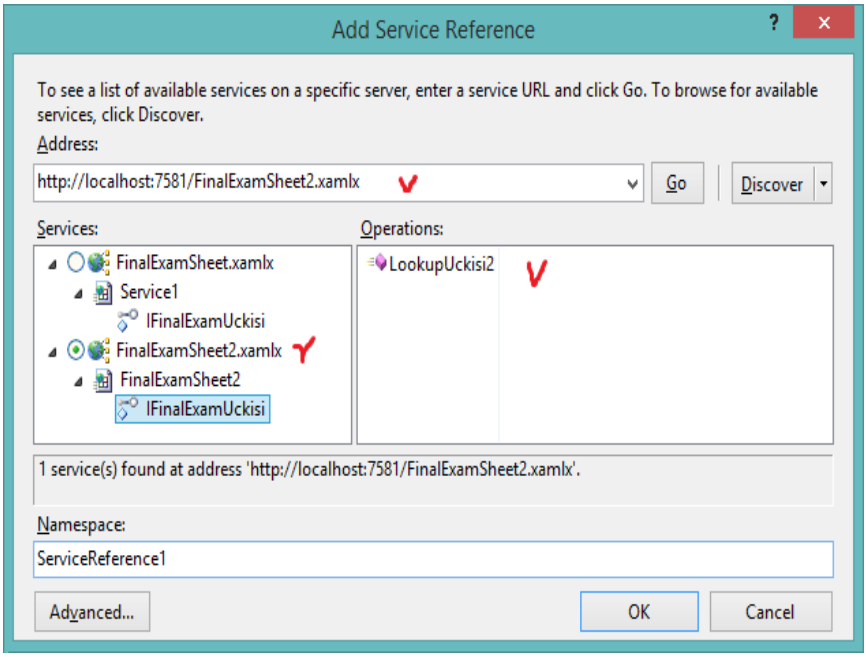
Solution Explorer-ში ExamLookup project-ზე მარჯვენა ღილაკის დაჭერით და Add Service Reference არჩევით, უნდა მივიღოთ 3.33-ა ნახაზი.



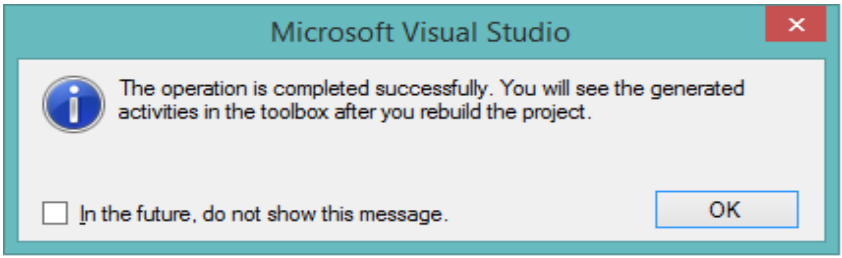
ნახ.3.33-ა. არსებული სერვისების ძებნა

დავაჭიროთ Discover ლინკის ღილაკს და ავიჩიოთ სერვისები Solution-ში. 3.33-ბ ნახაზზე დიალოგის სიაში ჩანს ორი ჩვენ მიერ შექმნილი სერვისი FinalExamSheet პროექტში.

შესაძლებელია ამ სერვისების გაფართოვება, რათა დანახულ იქნას მეთოდები, გათვალისწინებული თითოეულში. ავირჩიოთ მეორე, FinalExamSheet2.xamlx და OK. რამდენიმე წამის შემდეგ უნდა გამოჩნდეს დიალოგური ფანჯარა (ნახ.3.34).



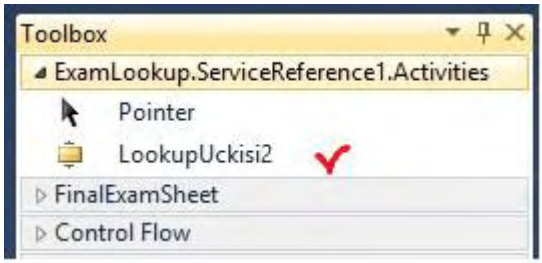
ნახ.3.33-ბ. სასურველი სერვისის მოძებნა



ნახ.3.34. დასრულებული დიალოგის პროცესი

ეს გაგვაგებინებს, რომ მიმართვა სერვისზე იქნა დამატებული პროექტში. F6-ით აღდგენილ იქნება Solution. ფაილი Window1.xaml უნდა იქნას ასახული. თუ არა, მაშინ გავხსნათ იგი.

ინსტრუმენტების პანელის ზედა ნაწილში უნდა იყოს 3.35 ნახაზის მაგვარი.



ნახ.3.35. განახლებული Toolbox სერვისებით

ინსტრუმენტების პანელზე ExamLookup.ServiceReference1 .Activities სახელსივრცე შეიცავს მომხმარებლის ქმედებას ყოველი მეთოდისთვის სერვისში. ჩვენ შემთხვევაში არის მხოლოდ ერთი: LookupUckisi2.

Solution Explorer-ში მაუსის მარჯვენა ღილაკით ExamLookup პროექტზე დავაჭიროთ და ავირჩიოთ Set as Startup Project.

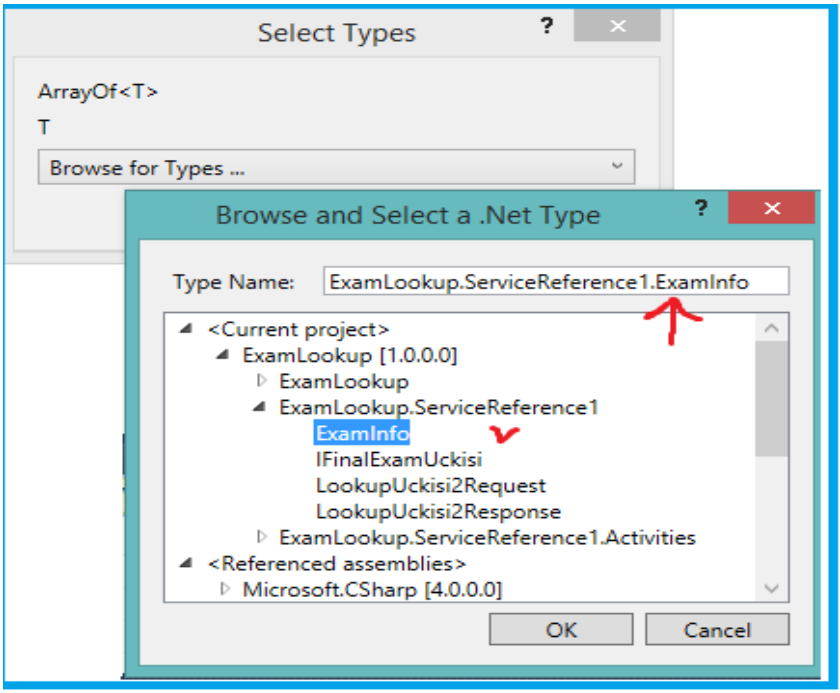
3.7.1. Workflow-პროცესის განსაზღვრა

გადავიტანოთ LookupBook2 ქმედება მუშა პროცესზე. შემდეგ საჭიროა არგუმენტების დაყენება რათა შესაძლებელი იყოს საძებნი კრიტერიუმების გადაცემა მუშა პროცესში და შედეგების დაბრუნება. დააჭირეთ Arguments მართვის ელემენტს.

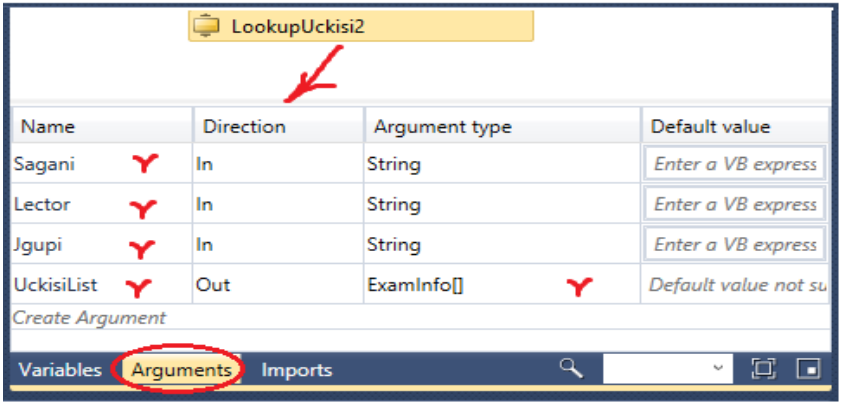
დავამატოთ სამი String შემავალი არგუმენტი სახელით Sagani, Lector და Jgupi. დავამატოთ გამომავალი არგუმენტი სახელით UckisiList. არგუმენტის ტიპისთვის ავირჩიოთ Array Of[T]. გამოსულ დიალოგურ ფანჯარაში ვირჩევთ Browse for Types და ავირჩევთ ExamInfo კლასს

ExamLookup.ServiceReference1.FinalExamSheet

ნაკრებიდან (ნახ.3.36). არგუმენტების სია მოცემულია 3.37 ნახაზზე.

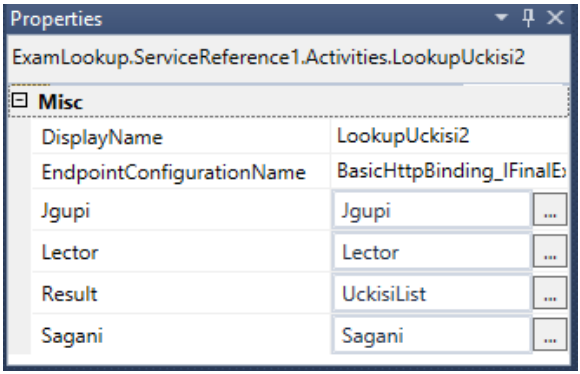


ნახ.3.36. ტიპის არჩევა



ნახ.3.37. მუშა პროცესის არგუმენტები

ავირჩიოთ “LookupBook2” ქმედება; Properties ფანჯარაში შევიტანოთ თვისებების values , როგორც 3.38 ნახაზზეა ნაჩვენები.



ნახ.3.38. LookupBook2 თვისებები

3.7.2. Host-აპლიკაციის რეალიზაცია

გავხსნათ Program.cs ფაილი ExamLookup პროექტში. ამ ფაილის რეალიზაცია ნაჩვენებია 3.4 ლისტინგში.

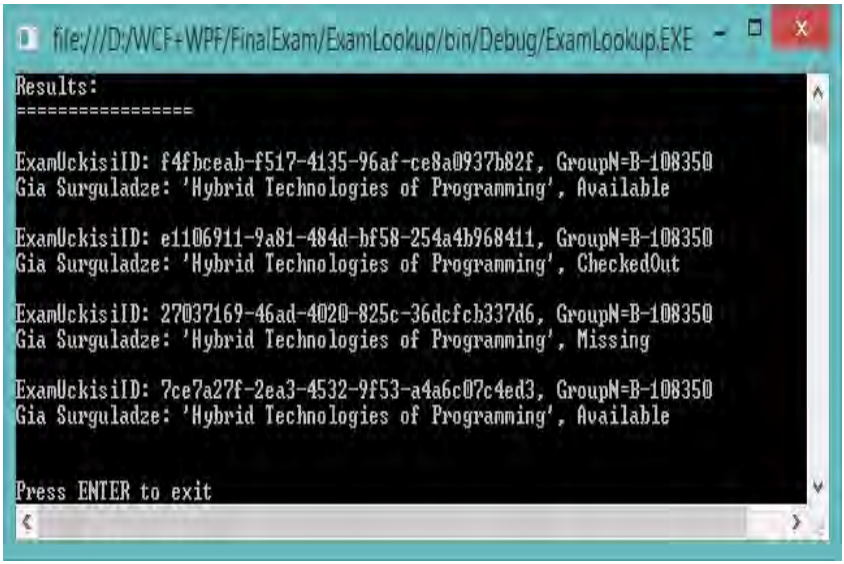
```
//- ლისტინგი 3.4 ---   Program.cs ფაილი ------------
using System;
using System.Linq;
using System.Activities;
using System.Activities.Statements;
using System.Collections.Generic;
using ExamLookup.ServiceReference1;
namespace BookLookup
{
 class Program
 {
   static void Main(string[] args)
  { // ლექსიკონის შექმნა არგუმენტების შეტანით
    // სამუშაო პროცესისთვის
```

```
  IDictionary<string, object> input = new
                    Dictionary<string, object>
  {
     {"Lector" , "Gia Surguladze" },
     {"Sagani","'Hybrid Technologies of Programming'"},
     {"Jgupi" , "B-108350" }
  };
   // workflow პროცესის შესრულება
      IDictionary<string, object> output =
         WorkflowInvoker.Invoke(new Workflow1(), input);

    ExamInfo[] l = output["UckisiList"] as ExamInfo[];
    if (l != null)
    {
     Console.WriteLine("Results:\n================\n");
      foreach (ExamInfo i in l)
      {
       Console.WriteLine("ExamUckisiID: {0}, GroupN={1}",
                      i.ExamUckisiID, i.Jgupi);
       Console.WriteLine("{0}: {1}, {2}\n", i.Lector,
                    i.Sagani, i.status);
      }
    }
    else
      Console.WriteLine("No items were found");
      Console.WriteLine("Press ENTER to exit");
      Console.ReadLine();
     }
   }
}
```

ეს კოდი გადასცემს Lector-, Sagani- და Jgupi-ში არგუმენტებს ობიექტის ლექსიკონის საშუალებით [17]. სამუშაო პროცესი აბრუნებს უკან ExamInfo ობიექტების მასივს. კოდს გამოაქვს ეკრანზე ამ მასივის შინაარსი.

3.7.3. აპლიკაციის ამუშავება

F5-ით გაიშვება პროგრამა. შედეგები ნაჩვენებია 3.39 ნახაზზე.



ნახ.3.39. შედეგები

3.8. არჩევის (Pick) გამოყენება (ლაბ-16)

მიზანი: Pick ქმედების დაპროგრამების შესწავლა, რომელიც ახორციელებს სამუშაო პროცესებში ტრიგერების და აქციების მართვას.

WF 4.0 უზრუნველყოფს ქმედებას, სახელით Pick, რომელსაც აქვს რამდენიმე არჩევის შტო (PickBranch). ყოველი შტო შეიცავს ტრიგერის თვისებას და აქციის თვისებას. თითოეული მათგანი ასრულებს ქმედებას (ან ქმედებათა მიმდევრობას). როდესაც Pick ქმედება შესრულდა, ტრიგერის ყველა ქმედება დაწყებულია.

როგორც კი ამათგან ერთი ქმედება დასრულდება, მისი შესაბამისი აქციები შესრულებულია და ყველა სხვა შტო გაუქმებულია.

ეს სასარგებლოა შესაბამისი აქციების განსაზღვრისათვის რომელიმე მოვლენის საფუძველზე. მაგალითად, შეიძლება მონაცემთა მიღების (Receive) ქმედების გამოყენება ტრიგერისთვის.

ყოველ შტოს შეიძლება ჰქონდეს Receive ქმედება, რომელიც ელოდება სხვა შეტყობინებას. რომლის საფუძველზეც მიღებულია შეტყობინება, იმის შესაბამისი აქცია შესრულდება.

ამ პროექტისთვის გამოყენებულ იქნება Pick ქმედება, რათა უზრუნველყოფილ იქნას ტაიმ-აუტის ფუნქცია მუშა პროცესისთვის.

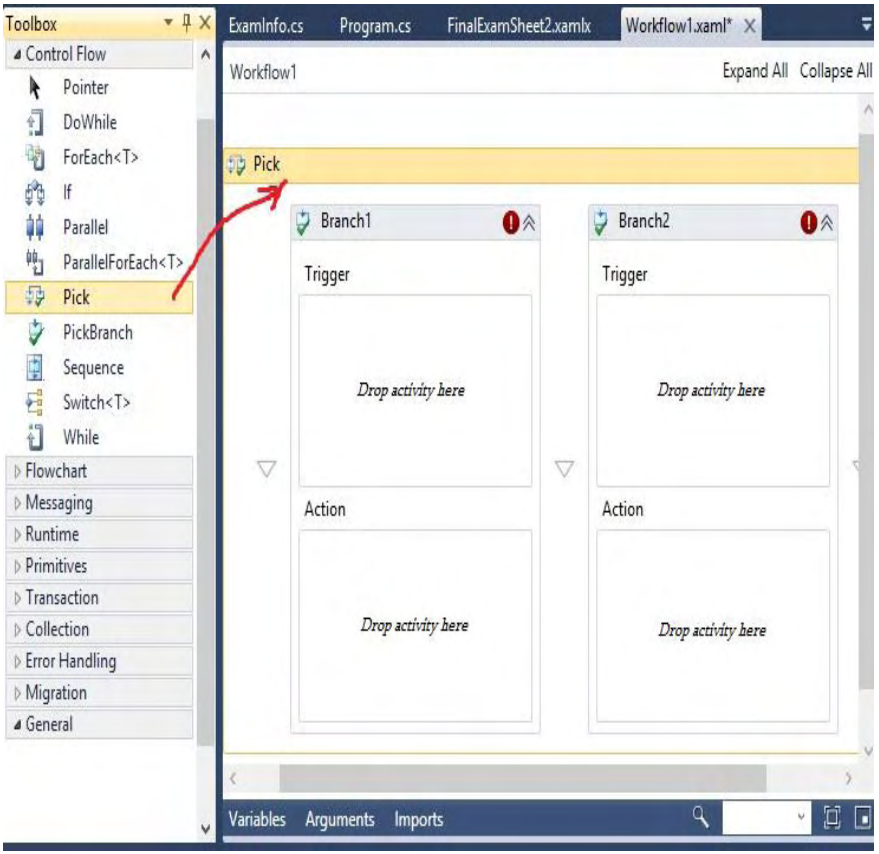
რჩევა. თუ თქვენ იცნობთ მუშა პროცესის წინა ვერსიებს, ეს ექვივალენტურია, მაგალითად, Listen ქმედებისა.

გავხსნათ Window1.xaml ფაილი, მაუსის მარჯვენა ღილაკით „LookupBook2” ქმედებაზე დავაჭიროთ და ავირჩიოთ Cut (ნახ.3.40).

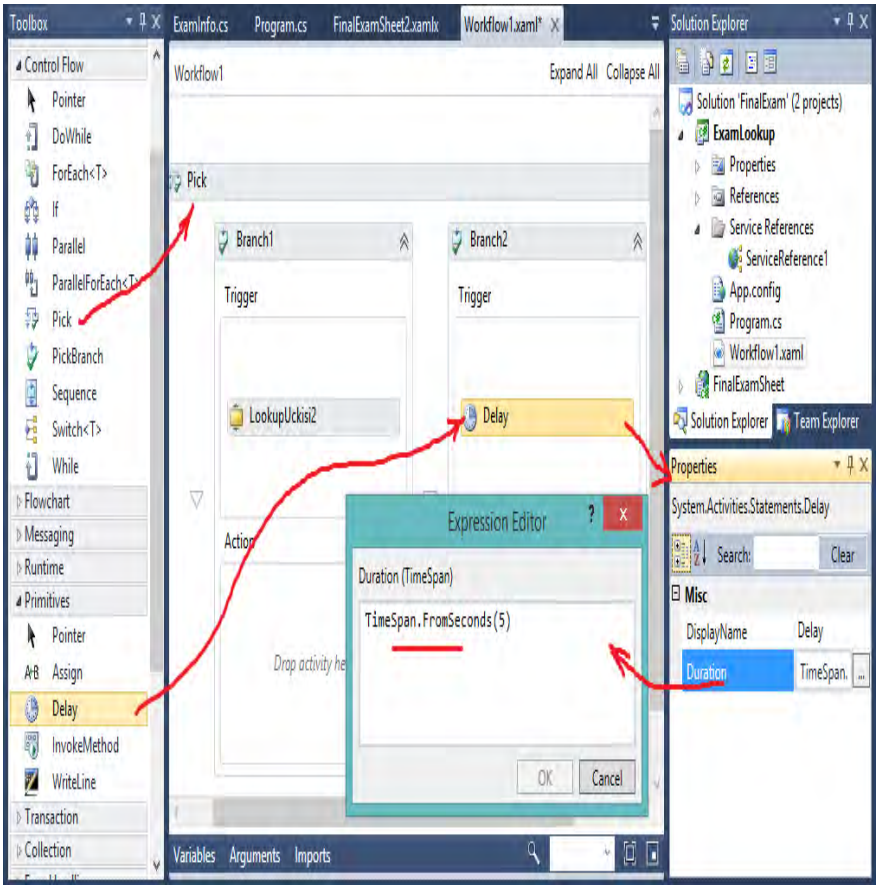
გადავიტანოთ Pick ქმედება მუშა პროცესზე (ნახ.3.41).

მაუსის მარჯვენა ღილაკით დავაჭიროთ პირველი განყოფილების Trigger-ს და ავირჩიოთ ბრძანება Paste (ჩასმა).

გადავიტანოთ დაყოვნების ქმედება (Delay) მეორე შტოს Triggeg-ის განყოფილებაში. დააყენეთ მისი ხანგრძლივობის (Duration) თვისება TimeSpan.FromSeconds(5), როგორც 3.42 ნახაზზეა ნაჩვენები.

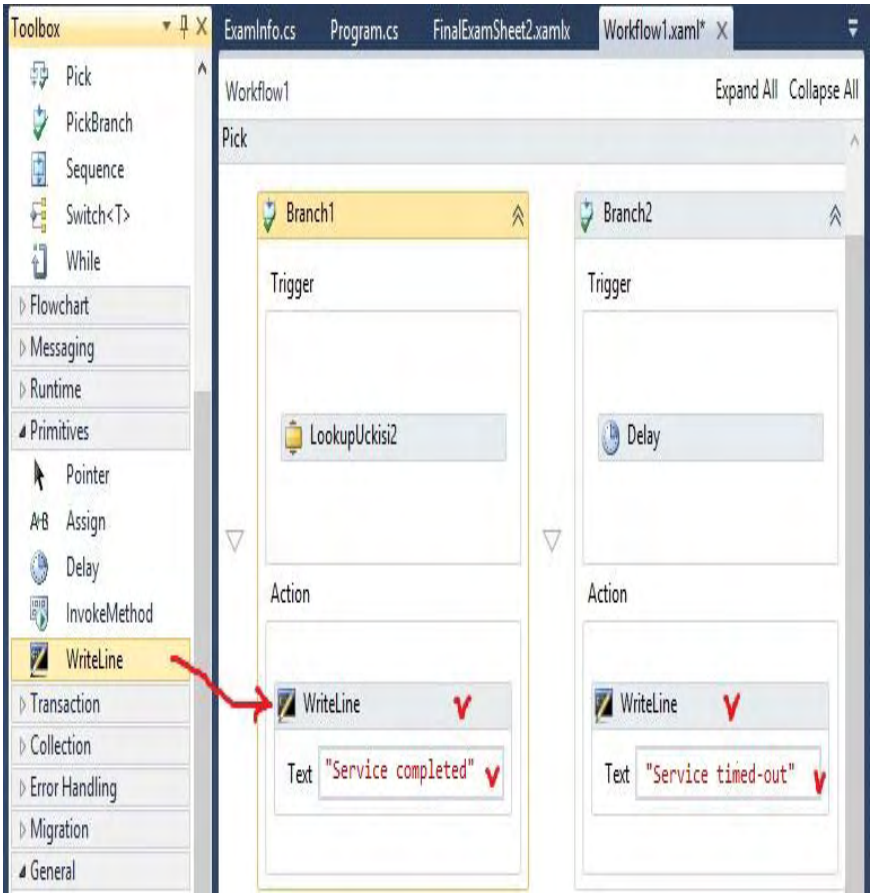


ნახ.3.41. Pick-ქმედების გადმოტანა



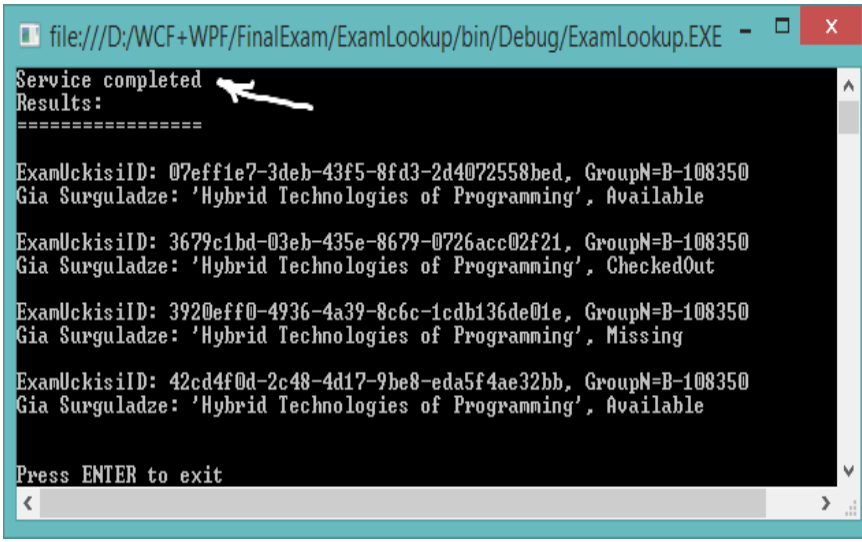
ნახ.3.42. მუშა პროცესი timeout ლოგიკით

გადავიტანოთ WriteLine ქმედება ყოველი აქციის სექციაში და მივცეთ Text-თვისება „სრული სერვისი“ და „სერვისი ტაიმ-აუტით“. მუშა პროცესი უნდა გამოიყურებოდეს 3.43 ნახაზზე მოცემული სახით.



ნახ.3.43. WriteLine ქმედების ჩამატება ორივე აქციაში

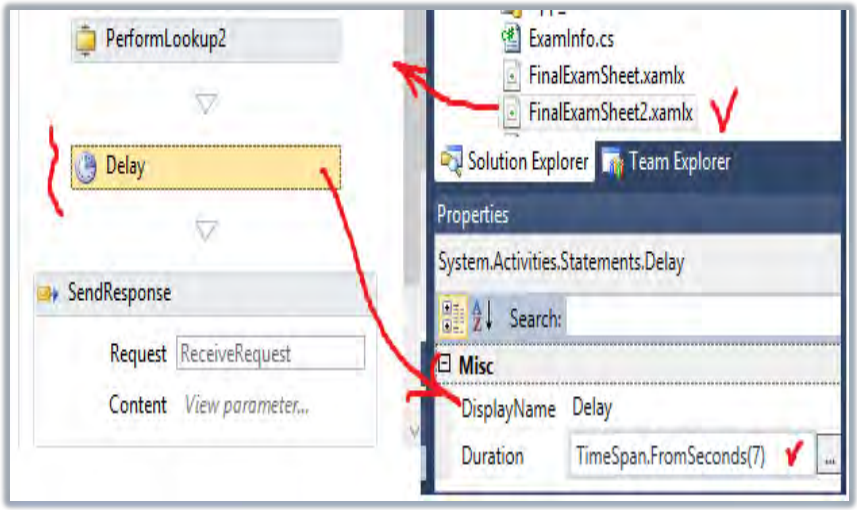
F5-ით ავამუშავოთ აპლიკაცია. შედეგები უნდა იყოს იდენტური პროგრამის ბოლო გაშვებისა გამორიცხვის დამატებული შეტყობინებით („სრული სერვისი“).



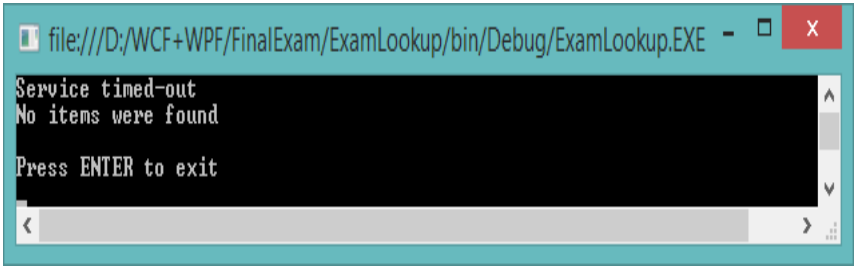
ნახ.3.44. შედეგები "Service completed"-ით

ტაიმ-აუტის ფუნქციის შესამოწმებლად გავხსნათ ფაილი FinalExamSheet2.xamlx. გადმოვიტანოთ დაყოვნების Delay ქმედება SendResponse ქმედების დაწყების წინ და დავაყენოთ მისი ხანგრძლივობის Duration თვისება TimeSpan.FromSeconds(7).

F5-ით ავამუშავოთ აპლიკაცია. 7 წამის დაყოვნების შემდეგ შედეგებს ექნება 3.46 ნახაზზე მოცემული სახე.



ნახ.3.45



ნახ.3.46. შედეგი "Service timed-out" - ით

დასკვნა

სამუშაო პროცესები (ან ბიზნესპროცესები, დოკუმენტ-ბრუნვის პროცესები) ხშირად განაწილებულია სხვადასხვა აპლიკაციებში და სხვადსხვა სერვერებზეც კი. ამიტომაც კომუნიკაცია არის პროექტში მუშა პროცესის დიზაინის მნიშვნელოვანი ნაწილი.

Send და Receive ქმედებები (აგრეთვე ReceiveReply და SendReply) უზრუნველყოფს შეტყობინებათა მოსახერხებელი ფორმით გადაცემას და მიღებას. ეს ქმედებები ეყრდნობა WCF-ს შეტყობინებების გადასაცემად და შეუძლია გამოიყენოს რიგი პროტოკოლებისა, როგორიცაა HTTP ან TCP/IP. მიუხედავად ამისა, ჰოსტ-აპლიკაციას შეუძლია ასევე უშუალოდ მიიღოს WCF-შეტყობინება.

Web-სერვისების გამოყენება ხდება დაპროექტებისთვის სულ უფრო პოპულარული მიდგომა. საქმე შეიძლება დაწყებულ იქნას კონტრაქტიდან მომსახურების მიღებაზე, ან უბრალოდ განისაზღვროს შემავალი და გამომავალი პარამეტრები, მუშა პროცესების კონსტრუქტორის გამოყენებით.

სამუშაო პროცესი შეიძლება გამოყენებულ იქნას როგორც სერვისის შესაქმნელად, ასევე მათ გამოსაყენებლად. მეთოდები, რომლებიც უზრუნველყოფილია Web-სერვისებით, ხდება ტრადიციული, სტანდარტული ქმედებები, რომელთა გამოყენება შესაძლებელია თანამედროვე პროგრამული აპლიკაციების ბიზნეს-პროცესებში.

გამოყენებული ლიტერატურა:

1. სურგულაძე გ. (2014). კორპორაციული მენეჯმენტის სისტემების Windows დეველოპმენტი: WPF ტექნოლოგია (ნაწ.1). სტუ, თბილისი, 200 გვ.

2. სურგულაძე გ. (2015). კორპორაციული მენეჯმენტის სისტემების Windows დეველოპმენტი: Workflow ტექნოლოგია (ნაწ.2). სტუ, თბილისი. 136 გვ.

3. სურგულაძე გ. (2011) .ვიზუალური დაპროგრამება C#_2010 ენის ბაზაზე. სტუ, თბილისი. -445 გვ.

4. სურგულაძე გ., ბულია ი., თურქია ე. (2009). Web-აპლიკაციების დამუშავება მონაცემთა ბაზების საფუძველზე (ADO.NET, ASP.NET, C#). სტუ, თბილისი. -172 გვ.

5. 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

6. 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

7. სურგულაძე გ., გულიტაშვილი მ., კივილაძე ნ. (2015). Web-აპლიკაციების ტესტირება, ვალიდაცია და ვერიფიკაცია. სტუ. თბილისი. -205 გვ.

8. კივილაძე ნ., სურგულაძე გ., გულიტაშვილი მ. (2013). Microsoft-ის ტექნოლოგიების ანალიზი და განვითარების ტენდენციები საინფორმაციო სისტემებისთვის. სტუ-ს შრ.კრ. „მართვის ავტომატიზებული სისტემები“, N 1(17), გვ.199-202.

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

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

11. Collins M.J. (2010). Beginning WF: Windows Workflow in .NET 4.0. ISBN-13 (pbk): 978-1-4302-2485-3 Copyright © 2010. USA.

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

13. სურგულაძე გ., თოფურია ნ., ბაკურია კ., ლომიძე მ. (2014). საინფორმაციო სისტემის დაპროექტება ობიექტ-როლური მოდელირების და სერვის-ორიენტირებული არქიტექტურის ბაზაზე. სტუ შრ.კრებ.: „მას“-N1(17). თბილისი, გვ. 35-42

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

15. სურგულაძე გ., ოხანაშვილი მ., სურგულაძე გ. (2009). მარკეტინგის ბიზნეს_პროცესების უნიფიცირებული და იმიტაციური მოდელირება. მონოგრ., სტუ. თბილისი. -170 გვ.

16. მეიერ-ვეგენერო კ., სურგულაძე გ., ბასილაძე გ. (2014). საინფორმაციო სისტემების აგება მულტიმედიური მონაცემთა ბაზებით. სტუ, თბილისი. -345 გვ.

17. სურგულაძე გ., თოფურია ნ., ბასილაძე გ., კივილაძე ნ., ნეფარიძე მ. (2014). ელექტრონული საარჩევნო სისტემა მულტიმედიური მონაცემთა ბაზებით და კლიენტ-სერვერ არქიტექტურით. GESJ: Computer Science and Telecommunications, 2014, N2(42). გვ. 39-86

18. სურგულაძე გ., ფხაკაძე ც., კაიშაური თ., კეკენაძე ა. (2015). ორგანიზაციული მართვის ბიზნეს-პროცესების მოდელირება, დაპროეტება და პროგრამული რეალიზაცია. VII საერთ. სამეცნ. პრაქტ. კონფ. „ინტერნეტი და საზოგადოება“. აკ.წერეთლის სახ.უნივ. ქუთაისი. გვ. 92-96

19. სურგულაძე გ., ბულია ი. (2012). კორპორაციულ Web-აპლიკაციათა ინტეგრაცია და დაპროექტება. სტუ. თბ., -324 გვ.

ISBN 978-9941- 0-7878-1

გია სურგულაძე

gsurg@gmx.net

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

300-ზე მეტი სამეცნიერო ნაშრომის ავტორი, მათ შორის 60 წიგნი და 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.

გადაეცა წარმოებას 10.10.2015 წ. ხელმოწერილია დასაბეჭდად 20.10.2015 წ. ოფსეტური ქაღალდის ზომა 60X84 1/16. პირობითი ნაბეჭდი თაბახი 9.5. ტირაჟი 100 ეგზ.

Verba volant, scripta manent

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