Developing Windows Services Succinctly

by José Roberto Olivas Mendoza

Syncfusion | Technology Resource Portal

Developing Windows Services Succinctly

By

José Roberto Olivas Mendoza

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Important licensing information. Please read.

Technical Reviewer: Stephen Haunts
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click" or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the author

I'm an IT businesses entrepreneur, software developer, and huge technology fan. My company went to market in 1990, focused basically in custom software development. Since then, we've been using Microsoft's technologies. We started with COBOL as our main programming language, evolving along these years up to DotNET and Office technologies.

I've been programming since 1988 (just a couple of years before my company was founded) in many languages, such as COBOL, Quick Basic, PASCAL, Clipper, FoxPro, Visual FoxPro, Visual Basic, Clarion, and C#. VBA for Applications (included in Microsoft Office products) and various database platforms such as Microsoft SQL Server, Postgres, Firebird, and MySQL have also have been part of my projects.

My company has worked in more than 100 projects for the last 25 years. These projects have been oriented to small and mid-sized businesses the most, including Grain Purchasing and Selling Software, ERP solutions, point of sale applications for general retail businesses, hardware stores, mobile devices stores, auto parts stores, and fast food restaurants. We also developed a .NET API oriented to electronic invoicing, according to Mexico's Revenue Service (known as SAT) regulations. This API is used by about 5,000 users.

My work with Windows Services started five years ago. Our technical support department needed to monitor usage of a POS application, which was deployed in a remote server and accessed over the internet. Since the monitoring process was intended to run in background mode with no human interaction at all, automation was one of the main requirements. The process needed to run indefinitely, as long as the server was up and running, even if there was no user logged on. The solution was to develop a Windows Service, which was intended to write a log entry every time a user logs on or logs off the application. The data would be stored in an MS SQL Database, so we could get statics about usage time and who was working with the application.

Today, programming Windows Services helps us to improve our products. As a result, we can offer our customers more value for their businesses and cut our technical support costs.

Who is this book for?

This book is being written primarily for .NET developers who want to improve their applications by using Windows Services or who want to start in Windows Services development.

The book starts with a brief explanation about what Windows Services are and how they are managed in Windows. Then, it shows how to create a basic Windows Service project type using Visual Studio.

The rest of the book is intended to build a practical Windows Service project, step by step, and show the code needed for that purpose. Basic concepts about Windows Services and Windows log events are discussed too.

Finally, it explains how to deploy the service application created in the computer in which the service will be running, and gives suggestions about making the deployment process easier.

For the purposes of this book, all the sample code was written in C# and requires Visual Studio 2010 and .NET framework 3.5 minimum. The entire project discussed in this book can be downloaded here.

I hope that by the time they finish reading this book, developers can create Windows Services taking advantage of Visual Studio and C# capabilities and benefit from using them in their projects.

Introduction

What is a Windows Service?

A Windows Service, formerly known as an NT Service, is an executable application that runs in its own Windows session and doesn't show a user interface. It operates in the background and runs as long as Windows is running. It can also be configured to start when the operating system is started, and alternatively, can be started manually or by an event. Because a Windows Service operates in the context of its own dedicated user account, it can operate when a user is not logged on.

Windows Services administration

Windows administrators can manage services in several ways. These include the Sc.exe command line tool, Windows Power Shell, and the Services snap-in, which is found under Administrative Tools in the Windows Control Panel and shown in the following figure.

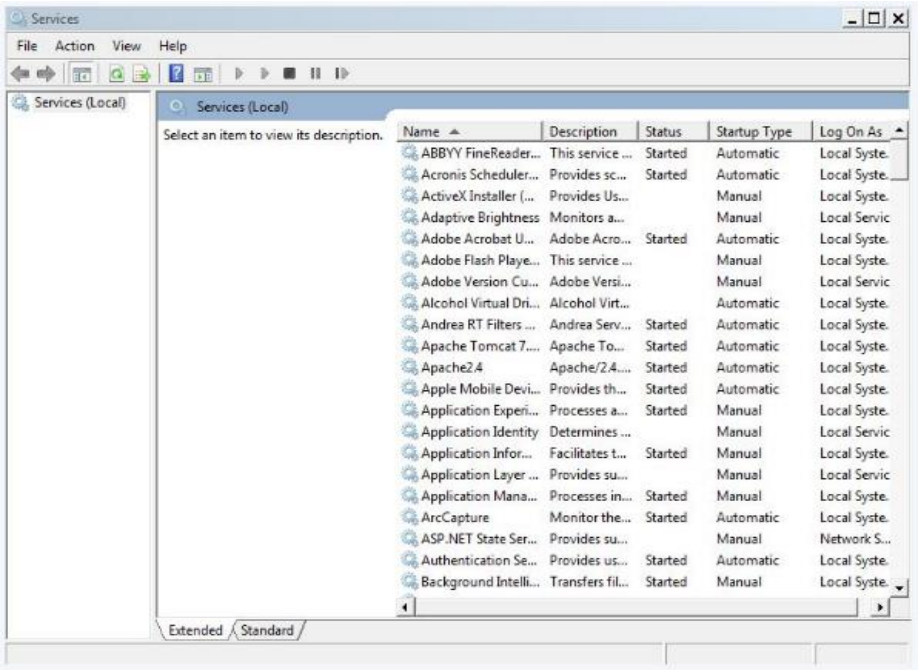

Figure 1: The Services Snap-in

The Services snap-in can connect to the local computer or a remote computer on the network, enabling users to do the following:

- View a list of all installed services including service name, description, and configuration
- Start, stop, pause, or restart services

- Change the startup type. Acceptable types are:
  - Automatic: The service starts at system startup.
  - Automatic (delayed): The service starts a short while after the system has finished starting up.
  - Manual: The service starts only when explicitly summoned.
  - Disabled: The service is disabled and will not run.
- Change the user account context in which the service works.
- Configure actions that should be taken if a service fails.
- Inspect service dependencies.
- Export the list of services to a text file.

Note: The Automatic (delayed) option was introduced in Windows Vista in an attempt to reduce the boot-to-desktop time. However, not all services support delayed start.

Developing Windows Services

Windows Services can be developed using Visual Studio. In order to be a Windows Service, a program needs to be coded in a particular way using a supplied template for this purpose. That is, it must handle start, stop, and pause messages from the Service Control Manager, the component of Windows that is responsible for starting and stopping services.

Tip: Services must be created in a Windows Service application project under Visual Studio.

Chapter 1 Windows Services Development with .NET

Getting started

How to create the project in Visual Studio

The first step in developing a Windows Service, after loading Microsoft Visual Studio, is to create a Windows Service type project. The following figure shows this kind of project dialog in Visual Studio.

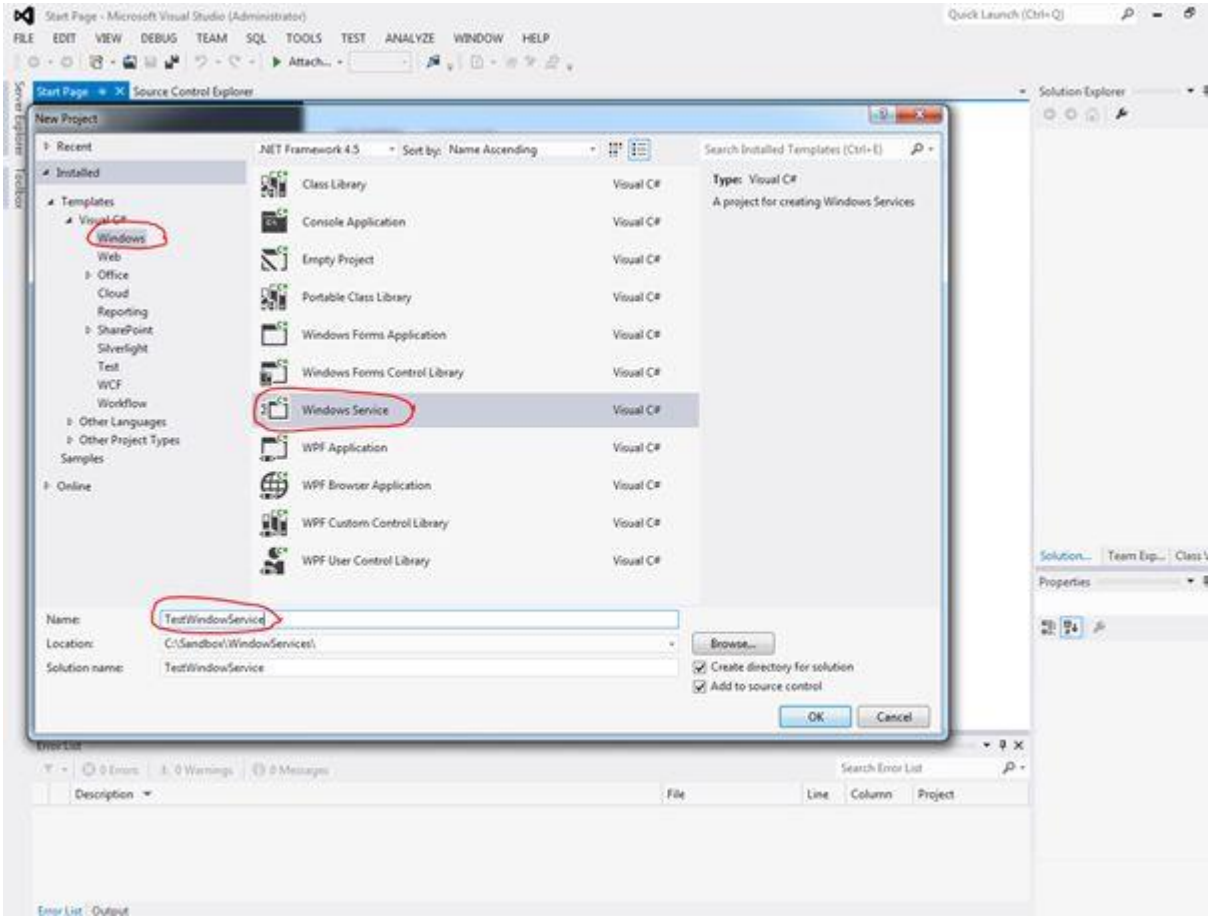

Figure 2: The Visual Studio Windows Service project type

Windows Service project base line

The base line for a Windows Service project consists of two programs; one of them, called **Program.cs**, manages the application entry point. The other one, **Service1.cs**, encapsulates the service class definition. A developer can add as many programs as it needs in order to easily maintain code or to add special features to the project.

Customizing project base line

The project base line can be customized in order to fit a developer's own needs. To do it in such way, you must rename the **Solution**, **Project**, **Program.cs**, and **Service1.cs** nodes. For the purposes of this book, the Windows Service project will be named **monitorservice.** The following figure shows the Visual Studio Solution Explorer before and after customization.

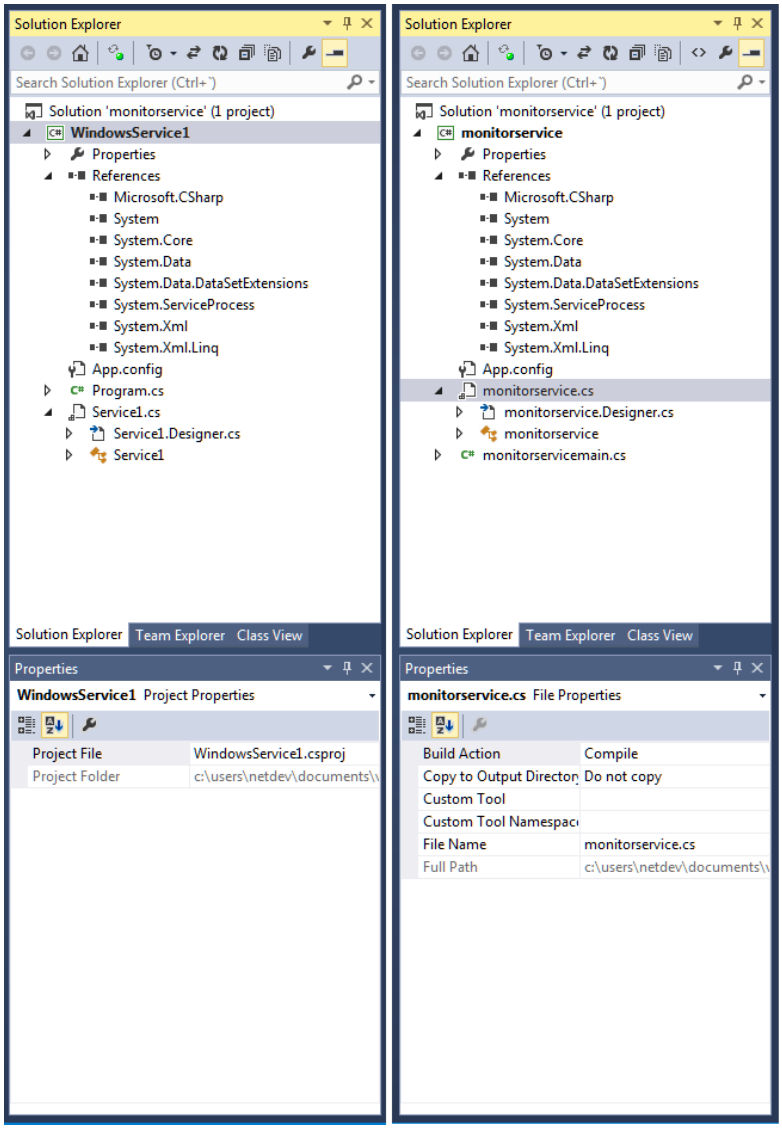

Figure 3: Solution Explorer before and after project customization

Tip: You can quickly rename elements in the Solution Explorer tree by right-clicking the desired node and applying the Rename command, from the Context pop-up menu. After you rename the action, the Visual Studio Refactoring Code Tool is fired.

Application entry point

Like many Visual Studio applications, a Windows Service needs an entry point in order to be executed. The following code sample shows the entry point for the project.

Code Sample 1

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.ServiceProcess;
using System.Text;
using System.Threading.Tasks;

namespace monitorservice
{
    static class monitorservicemain
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main()
        {
            ServiceBase[] ServicesToRun;
            ServicesToRun = new ServiceBase[]
            {
                new monitorservice()
            };
            ServiceBase.Run(ServicesToRun);
        }
    }
}
```

This program creates an array named **ServicesToRun**. This array is based on the **ServiceBase** .NET class, and stores an instance of the **monitorservice** custom class. **Monitorservice** is also derived from the **ServiceBase** .NET class, and will manage the Windows Service that is being developed.

Once the array is created, the program calls the **Run** method of **ServiceBase**, passing the array as a parameter, and service execution starts.

ServiceBase .NET class

This class is part of the **System.ServiceProcess** namespace, and provides a base class for a service that will exist as part of a service application. **ServiceBase** must be derived from creating a new service class for a service application. As mentioned previously, the derived class that will manage the service is named **monitorservice**, and can be found in the **monitorservice.cs** file.

Remarks

Any useful service overrides the **OnStart** and **OnStop** methods. For additional functionality, **OnPause** and **OnContinue** can be overridden with specific behavior in response to changes in the service state. This can be useful if a user interface needs to be provided to change service behavior, because the service can be notified about this change using these methods.

By default, services run under the System account, which is not the same as the Administrator account. The rights of the System account can't be changed.

When a service is started, the system locates the executable and runs the **OnStart** method for that service, contained within the executable. However, running the service is not the same as running the executable. The executable only loads the service. The service is accessed (for example, started, and stopped) through the Service Control Manager.

ServiceBase derived class definition

The following code sample shows the **monitorservice** derived class definition.

Code Sample 2

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Linq;
using System.ServiceProcess;
using System.Text;
using System.Threading.Tasks;

namespace monitorservice
{
    public partial class monitorservice : ServiceBase
    {
        private System.Timers.Timer serviceTimer = null;

        public monitorservice()
        {
```

```
            InitializeComponent();
        }

        protected override void OnStart(string[] args)
        {
        }

        private void timer_Elapsed(object sender, System.Timers.ElapsedEventArgs e)
        {
        }

        protected override void OnStop()
        {
        }

    }
}
```

When the project was created, Visual Studio generated this code automatically. Basically this code does the following:

1. When an instance of the class is created, the constructor calls the **`InitializeComponent()`** method in order to place all the code needed to perform when the instance is created.
2. Overrides the **`OnStart()`** event in order to place all the code needed to perform when the service starts its execution.
3. Overrides the **`OnStop()`** event in order to place all the code needed to perform when the service stops its execution.

This is the base code from which the entire service application will be developed. In order to add more features to the application, we may need to override other events. I suggest sticking to service lifetime, which is explained in the following section.

Service lifetime

A service goes through several internal states in its lifetime. First, the service is installed onto the system on which it will run. This process executes the installers for the service project and loads the service into the Services Control Manager for that computer. The Services Control Manager is the utility provided to administer services.

After the service has been loaded, it must be started. Starting the service allows it to begin functioning. A service can be started from the Services Control Manager, from Server Explorer, or from code by calling the **`Start`** method. The **`Start`** method passes processing to the application's **`OnStart`** method and processes any code you have defined there.

A running service can exist in this state indefinitely until it is either stopped or paused, or until the computer shuts down. A service can exist in one of three basic states: Running, Paused, or Stopped. The service can also report the state of a pending command: **ContinuePending**, **PausePending**, **StartPending**, or **StopPending**. These statuses indicate that a command has been issued (such as a command to pause a running service), but has not yet been carried out. You can query the Status to determine what state a service is in, or use **WaitForStatus** to carry out an action when any of these states occurs.

You can pause, stop, or resume a service from the Services Control Manager, from Server Explorer, or by calling methods in code. Each of these actions can call an associated procedure in the service (**OnStop**, **OnPause**, or **OnContinue**), in which you can define additional processing to be performed when the service changes state.

OnStart method

When a service is started, the system locates the executable and runs the **OnStart** method for that service, contained within the executable. However, running the service is not the same as running the executable; the executable only loads the service. The service is accessed (for example, started and stopped) through the Service Control Manager.

The executable calls the **ServiceBase** derived class's constructor the first time **Start** is called on the service. The **OnStart** command-handling method is called immediately after the constructor executes. The constructor is not executed again after the first time the service has been loaded, so it is necessary to separate the processing performed by the constructor from that performed by **OnStart**. Any resources that can be released by **OnStop** should be created in **OnStart**, since creating resources in the constructor prevents them from being created properly if the service is started again after **OnStop** has released the resources.

The following code sample shows how the resources needed are created.

Code Sample 3

```
protected override void OnStart(string[] args)
{
    this.serviceTimer = new System.Timers.Timer(300);
    this.serviceTimer.AutoReset = true;
    this.serviceTimer.Elapsed += new System.Timers.ElapsedEventHandler(this.timer_El
apsed);
    this.serviceTimer.Start();
}
```

In this case a **Timer** object is instantiated and started. Every 300 milliseconds, the **Elapsed** event of the timer will be fired and the **timer_Elapsed** method will be executed. The **OnStop** event will release this resource.

OnStop method

The **OnStop** method executes every time a **Stop** command is sent to the service, since the **OnStop** event is fired. The **Stop** command could be sent by the Service Control Manager (SCM). In similar way to the **OnStart** method, we can use the **OnStop** method to perform any task needed at service stopping, such as releasing resources no longer needed, as shown in the following code sample.

Code Sample 4

```
protected override void OnStop()
{
    this.serviceTimer.Stop();
    this.serviceTimer.Dispose();
    this.serviceTimer = null;
}
```

This code stops the **Timer** object execution and disposes it before service execution stops.

Chapter summary

Windows Services can be developed with .NET and Visual Studio using the Windows Service template provided for this purpose. This template automatically creates the code baseline for development.

The **ServiceBase** .NET class provides a base class for a service that will exist as part of a service application, and must be derived from creating a new service class for a service application.

When the service code baseline is created, the **OnStart()** and **OnStop()** methods are overridden in order to execute actions when the service starts or stops its execution.

Chapter 2 The Windows Event Log

Windows Services have no interaction with the user, so it doesn't have an interface. Whatever output is needed to be produced is typically written to some sort of log, such as a database. One good place to log to is the Windows Event Log.

The Windows Event Log is a record of a computer's alerts and notifications. Microsoft defines an event as "any significant occurrence in the system or in a program that requires users to be notified or an entry added to a log."

The Windows operating system classifies events by type. For example, an *information event* describes the successful completion of a task, such as installing an application. A *warning event* notifies the administrator of a potential problem, such as low disk space. An *error message* describes a significant problem that may result in a loss of functionality. A *success audit event* indicates the completion of an audited security event, such as an end user successfully logging on. A *failure audit event* describes an audited security event that did not complete successfully, such as an end user locking himself out by entering incorrect passwords.

Each event in a log entry contains the following information:

- **Date:** The date the event occurred
- **Time:** The time the event occurred
- **User:** The name of the user who was logged on when the event occurred
- **Computer:** The name of the computer
- **Event ID:** A Windows identification number that specifies the event type
- **Source:** The program or component that caused the event
- **Type:** The type of event (information, warning, error, security success audit, or security failure audit)

The Windows Event Viewer

The Windows Event Viewer is a tool that displays detailed information about significant events (for example, programs that don't start as expected or updates that are downloaded automatically). The Windows Event Viewer can be helpful when troubleshooting problems and errors with Windows and other programs, such as Windows Services. The Windows Event Viewer can be found in the Administrative Tools section of the Control Panel.

Entries in the Windows Event Log can be viewed through Windows Event Viewer. This can be used to debug service code. In fact, this is the only way to do this, since a Windows Service has no user interface. The Windows Event Viewer's main window is shown in the following figure.

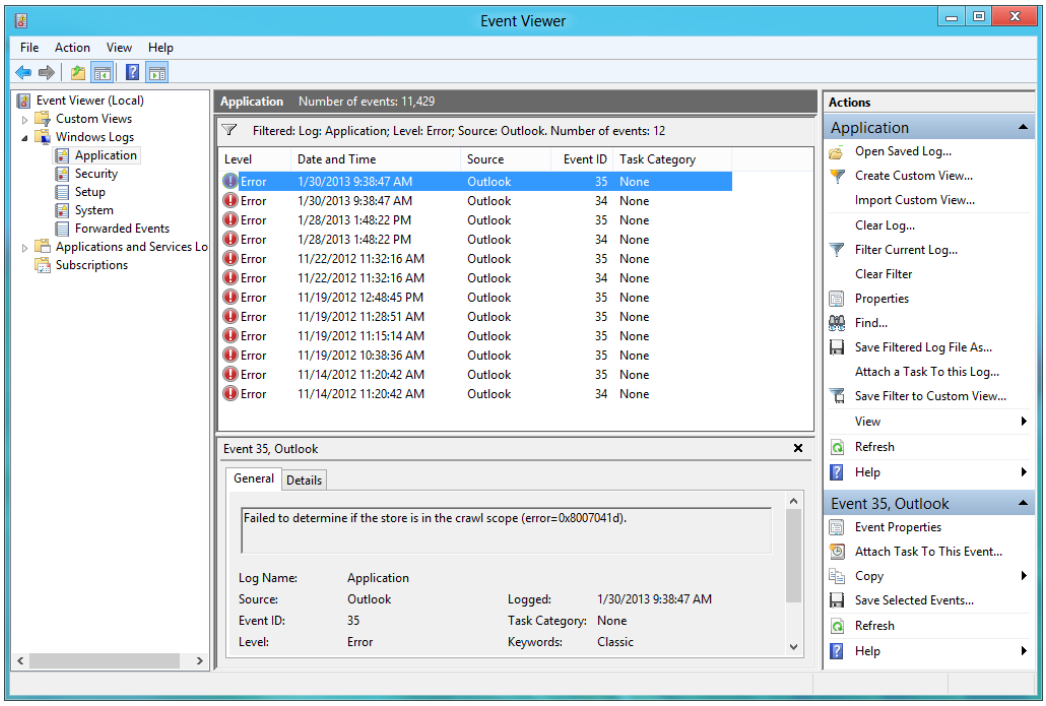

Figure 4: Windows Event Viewer

Bounding the service to the Windows Event Log

The service needs to be bounded to the Windows Event Log in order to write entries in it. To accomplish that, the following code needs to be placed in the **`OnStart`** method.

Code Sample 5

```
if (!System.Diagnostics.EventLog.SourceExists("MonitorService"))
    System.Diagnostics.EventLog.CreateEventSource("MonitorService", "Application");
```

As shown, the service first inquires the Event Log asking if an event source for **`MonitorService`** has been created previously. If not, the event source is created with the **`CreateEventSource`** method specifying that every log sent by the service will be written in the **`Application`** type log.

Now, the code for the **`OnStart`** method looks like this:

Code Sample 6

```
protected override void OnStart(string[] args)
{
    if (!System.Diagnostics.EventLog.SourceExists("MonitorService"))
        System.Diagnostics.EventLog.CreateEventSource("MonitorService", "Application
```

```
");


    this.serviceTimer = new System.Timers.Timer(300);
    this.serviceTimer.AutoReset = true;
    this.serviceTimer.Elapsed += new System.Timers.ElapsedEventHandler(this.timer_El
apsed);
    this.serviceTimer.Start();
}
```

Writing events to the Windows Event Log

The service needs to write into the Windows Event Log in order to communicate with users. To simplify the code, a method to log events will be written in class definition code. This method will receive two parameters: a string containing the message that will be written into the log, and another one that will indicate the type of event that's being saved.

Event types

As previously mentioned, the Windows Event Log allows you to specify what type of event is being saved. The **`EventLogEntryType`** enumeration will be used to that purpose.

The types allowed are the following:

- Information
- Warning
- Error
- Security success audit (SucessAudit): This type of event occurs when a user successfully logged on to a network or a computer.
- Security failure audit (FailureAudit): This type of event occurs when a user fails to log on to a network or a computer.

The LogEvent method

The code for this method is shown in the following sample.

Code Sample 7

```
private void LogEvent(string message, EventLogEntryType entryType)
{
    System.Diagnostics.EventLog eventLog = new System.Diagnostics.EventLog();

    eventLog = new System.Diagnostics.EventLog();
    eventLog.Source = "MonitorService";
    eventLog.Log = "Application";
    eventLog.WriteEntry(message, entryType);
```

```
}
```

As previously shown, every time the method is executed, it creates an **EventLog** instance. To perform log entry writing, the **Source** and **Log** properties need to store the event source and the log section in which the entry will be written. For this case, **MonitorService** will be the source of the entry and **Application** the log section. Once the values are stored in their respective properties, the **WriteEntry** method writes the entry in the Windows Event Log.

How class definition code looks so far

At this point, the code for service class definition looks like this:

Code Sample 8

```
public partial class monitorservice : ServiceBase
{
    private System.Timers.Timer serviceTimer = null;

    public monitorservice()
    {
        InitializeComponent();
    }

    protected override void OnStart(string[] args)
    {
        if (!System.Diagnostics.EventLog.SourceExists("MonitorService"))
            System.Diagnostics.EventLog.CreateEventSource("MonitorService", "Applicat
ion");

        this.LogEvent(String.Format("MonitorService starts on {0} {1}", System.DateTi
me.Now.ToString("dd-MMM-
yyyy"), DateTime.Now.ToString("hh:mm:ss tt")), EventLogEntryType.Information);

        this.serviceTimer = new System.Timers.Timer(300);
        this.serviceTimer.AutoReset = true;
        this.serviceTimer.Elapsed += new System.Timers.ElapsedEventHandler(this.timer
_Elapsed);
        this.serviceTimer.Start();
    }

    private void timer_Elapsed(object sender, System.Timers.ElapsedEventArgs e)
    {

    }

    protected override void OnStop()
    {
        this.serviceTimer.Stop();
        this.serviceTimer.Dispose();
        this.serviceTimer = null;
```

```
        this.LogEvent(String.Format("MonitorService stops on {0} {1}", System.DateTim
e.Now.ToString("dd-MMM-
yyyy"), DateTime.Now.ToString("hh:mm:ss tt")), EventLogEntryType.Information);

    }

    private void LogEvent(string message, EventLogEntryType entryType)
    {
        System.Diagnostics.EventLog eventLog = new System.Diagnostics.EventLog();

        eventLog = new System.Diagnostics.EventLog();
        eventLog.Source = "MonitorService";
        eventLog.Log = "Application";
        eventLog.WriteEntry(message, entryType);

    }

}
```

Notice that the **OnStart** and **OnStop** methods write to the Windows Event Log, notifying the date and time when each of them were fired. In this case, it's been considered an Information log entry type for both of them.

Tip: Log entry type should be used every time a program writes to the Windows Event Log, in order to clarify the reason the program is writing.

Chapter summary

The Windows Event Log is a record of a computer's alerts and notifications. An event can be defined as "any significant occurrence in the system or in a program that requires users to be notified or an entry added to a log." The Windows operating system classifies events by type. An *information event* describes the successful completion of a task; a *warning event* notifies the administrator of a potential problem; an *error message* describes a significant problem that may result in a loss of functionality; a *success audit* event indicates the completion of an audited security event, such as an end user successfully logging on; and a *failure audit event* describes an audited security event that did not complete successfully, such as an end user locking himself out by entering incorrect passwords.

Entries written in the Windows Event Log can be viewed through the Windows Event Viewer, which can be found in the Administrative Tools section of Control Panel. The Windows Event Viewer is a tool that displays detailed information about significant events (for example, programs that don't start as expected or updates that are downloaded automatically) on the computer, and can be helpful when troubleshooting problems and errors with Windows and other programs, such as Windows Services.

Since a Windows Service has no interface, writing entries in the Windows Event Log is a preferred way to communicate with users. In order to write these entries, the service needs to be bounded to the Windows Event Log. This can be accomplished using the **`CreateEventSource()`** method of the **`System.Diagnostics.EventLog`** namespace.

It's suggested to write a separate method to deal with entries writing activity. The **`EventLogEntryType`** enumeration must be used in order to clarify why each entry was written by the program.

Chapter 3 Service Installer

Creating a Windows Service project is a bit different from other kinds of projects. In order to be correctly deployed, Visual Studio ships installation components that can install resources associated with service applications. Installation components register an individual service on the system to which it is being installed and let the Services Control Manager know that the service exists.

Adding a Service Installer

Working with a service application allows you to automatically add the appropriate installers to the project. You can also accomplish this by double-clicking on the **monitorservice.cs** file name (where service base class is stored) in the Solution Explorer tree, and the Service Designer screen (shown in the following figure) will appear.

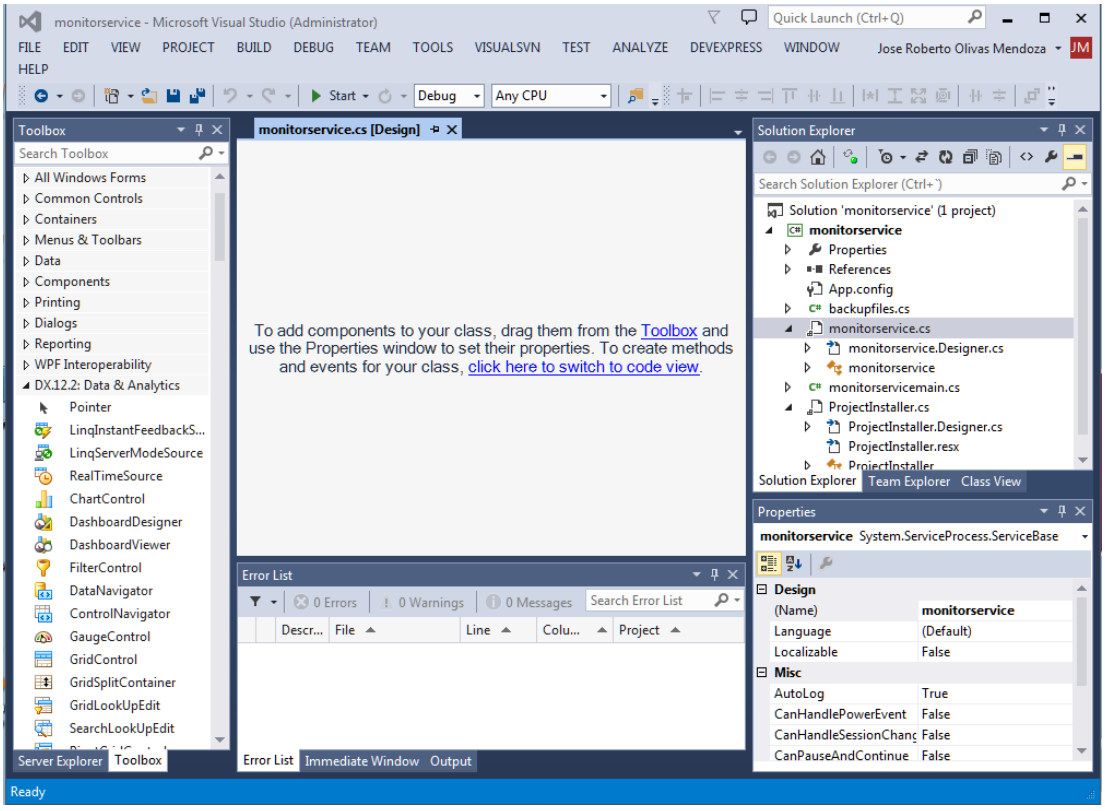

Figure 5: Service designer screen

Right-clicking on the gray area will bring up the Service Designer context menu. To automatically add the proper installer code to the project, click **Add Installer**.

The installer code

Two files, **ProjectInstaller.cs** and **ProjectInstaller.Designer.cs**, will be added to the project. All properties needed for service installation will be set here. The following code sample shows the contents of **ProjectInstaller.cs**.

Code Sample 9

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;
using System.Configuration.Install;
using System.Linq;
using System.Threading.Tasks;

namespace monitorservice
{
    [RunInstaller(true)]
    public partial class ProjectInstaller : System.Configuration.Install.Installer
    {
        public ProjectInstaller()
        {
            InitializeComponent();
        }
    }
}
```

The **`RunInstaller`** attribute of the **`ProjectInstaller`** class tells that Visual Studio's Custom Action Installer or the InstallUtil.exe will be invoked when the assembly is installed. InstallUtil.exe will be discussed later.

A constructor method is the only one created by the IDE, and this method calls the **`InitializeComponent()`** method in order to setup the values needed to install the service correctly.

Establishing service installation properties

To establish service installation properties, right-click the **ProjectInstaller.cs** file name in the monitorservice project tree, and then choose **View Designer** from the context menu that appears.

The designer window for **ProjectInstaller.cs** will appear with two icon buttons in it. One of these is linked to an instance of a **`serviceProcessInstaller`** object, and the other one to an instance of a **`ServiceInstaller`** object. These two instances contain the properties needed to make the Windows service installation successful.

These properties are:

- Account – Indicates the account type under which the service will run

- Description – Indicates the service's description (a brief comment that explains the purpose of the service)
- DisplayName – Indicates the friendly name that identifies the service to the user
- ServiceName – Indicates the name used by the system to identify this service
- StartType – Indicates how and when this service is started (as discussed in the Introduction)

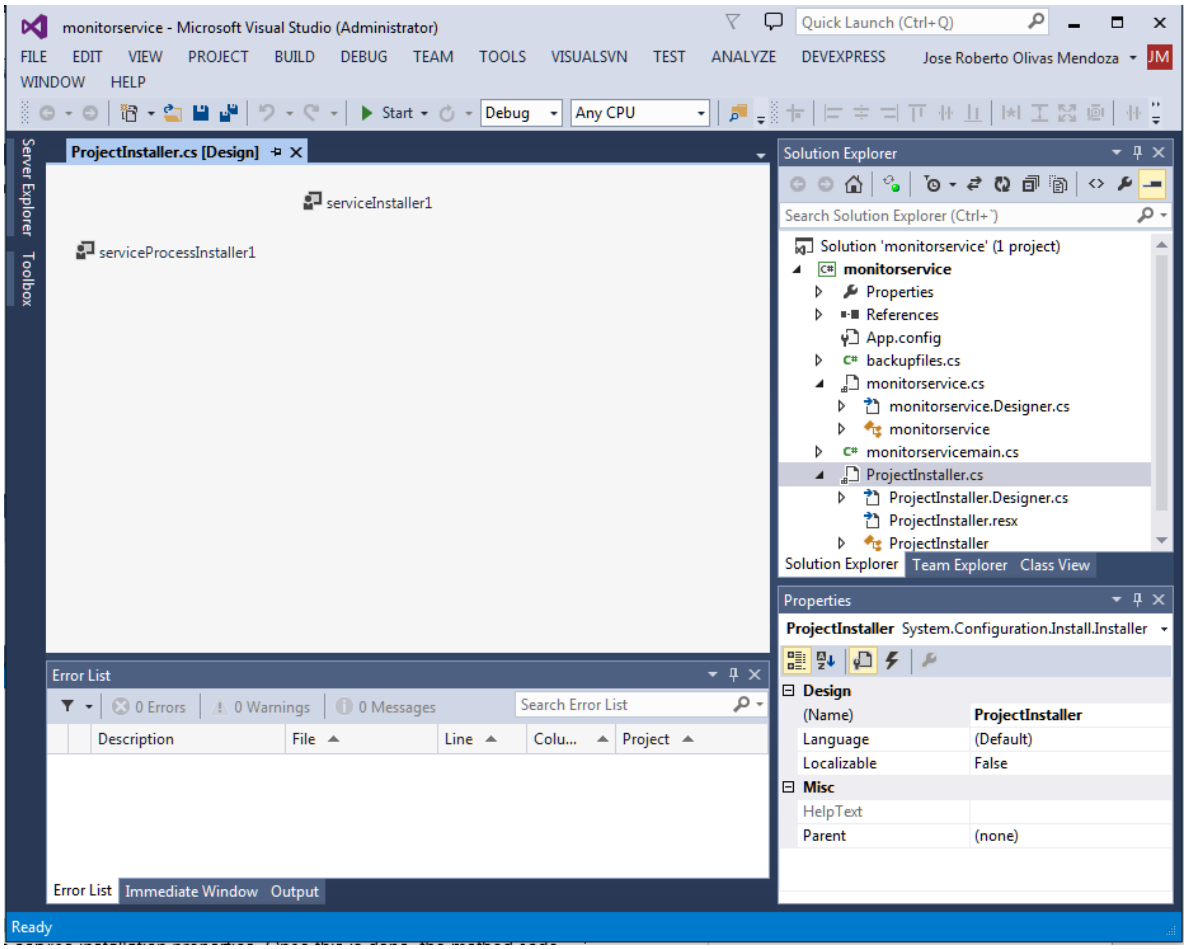

Figure 6: ProjectInstaller.cs designer screen with iconic buttons

Clicking on each button allows you to change the properties listed previously. To accomplish this task, it is necessary to use the Properties Window corresponding to each button, and enter the proper values in the corresponding textbox's Properties window.

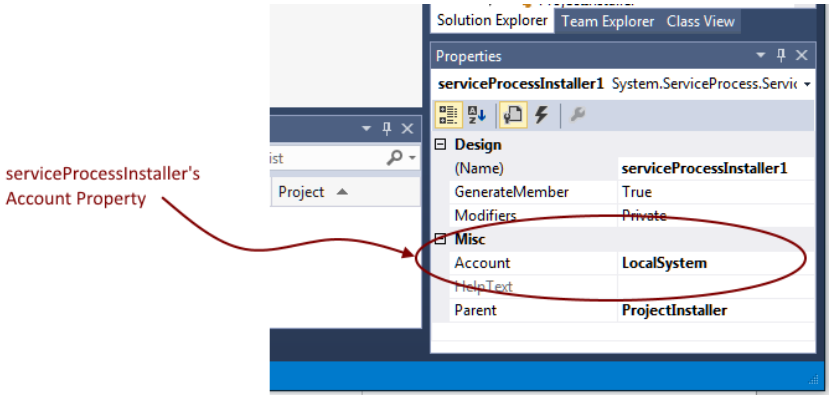

Figure 7: Figure 7: serviceProcessInstaller Properties window

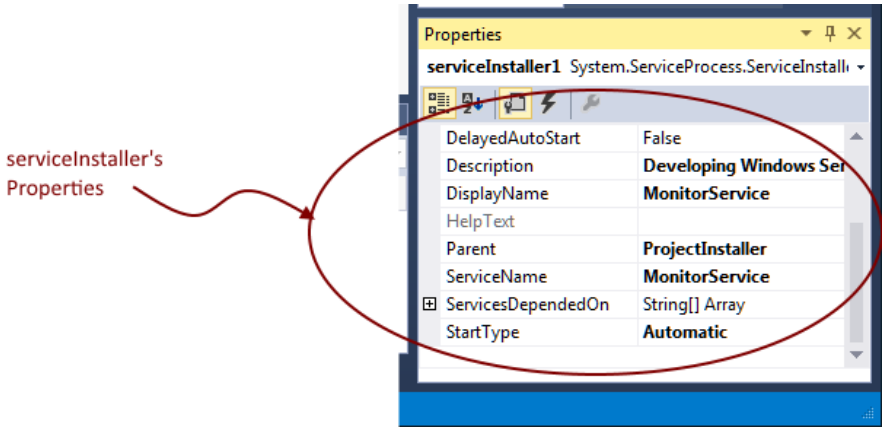

Figure 8: serviceInstaller Properties window

Once this is done, the **`InitializeComponent`** method's code will look like the following sample.

Code Sample 10

```
private void InitializeComponent()
{
    this.serviceProcessInstaller1 = new System.ServiceProcess.ServiceProcessInstaller
();
    this.serviceInstaller1 = new System.ServiceProcess.ServiceInstaller();
    //
    // serviceProcessInstaller1
    //
    this.serviceProcessInstaller1.Account = System.ServiceProcess.ServiceAccount.Loca
lSystem;
    this.serviceProcessInstaller1.Password = null;
    this.serviceProcessInstaller1.Username = null;
    //
    // serviceInstaller1
    //
```

```
    this.serviceInstaller1.ServiceName = "MonitorService";
    this.serviceInstaller1.DisplayName = "MonitorService";
    this.serviceInstaller1.Description = "Developing Windows Services Succinctly Tuto
rial";
    this.serviceInstaller1.StartType = System.ServiceProcess.ServiceStartMode.Automat
ic;
    //
    // ProjectInstaller
    //
    this.Installers.AddRange(new System.Configuration.Install.Installer[] {
    this.serviceProcessInstaller1,
    this.serviceInstaller1});

}
```

The code creates both an instance of **ServiceProcessInstaller** class and an instance for **ServiceInstaller** class. Then, it stores the name of the account that will be in charge of managing the service in the Account property of the **ServicesProcessInstaller** instance. In this case, the installer will use the **LocalSystem** account.

The **StartType** property of the **ServiceInstaller** instance will tell the installer that service will start automatically. The name that will be displayed in the Windows Services snap-in is stored in the **DisplayName** property, and a description for the service is stored in the **Description** property.

Now, the entire code for project installer will look like the following sample.

Code Sample 11

```
//ProjectInstaller.cs
using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;
using System.Configuration.Install;
using System.Linq;
using System.Threading.Tasks;

namespace monitorservice
{
    [RunInstaller(true)]
    public partial class ProjectInstaller : System.Configuration.Install.Installer
    {
        public ProjectInstaller()
        {
            InitializeComponent();
        }
    }
}

//ProjectInstaller.Designer.cs
```

```
namespace monitorservice
{
    partial class ProjectInstaller
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed; oth
erwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Component Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.serviceProcessInstaller1 = new System.ServiceProcess.ServiceProcessI
nstaller();
            this.serviceInstaller1 = new System.ServiceProcess.ServiceInstaller();
            //
            // serviceProcessInstaller1
            //
            this.serviceProcessInstaller1.Account = System.ServiceProcess.ServiceAcco
unt.LocalSystem;
            this.serviceProcessInstaller1.Password = null;
            this.serviceProcessInstaller1.Username = null;
            //
            // serviceInstaller1
            //
            this.serviceInstaller1.ServiceName = "MonitorService";
            this.serviceInstaller1.DisplayName = "MonitorService";
            this.serviceInstaller1.Description = "Developing Windows Services Succinc
tly Tutorial";
            this.serviceInstaller1.StartType = System.ServiceProcess.ServiceStartMode
.Automatic;
            //
            // ProjectInstaller
            //
            this.Installers.AddRange(new System.Configuration.Install.Installer[] {
```

```
            this.serviceProcessInstaller1,
            this.serviceInstaller1});

        }

        #endregion

        private System.ServiceProcess.ServiceProcessInstaller serviceProcessInstaller
1;
        private System.ServiceProcess.ServiceInstaller serviceInstaller1;
    }
}
```

At this point, the process of adding a Service Installer is complete.

Chapter summary

A Windows Service project is a bit different from others. To deploy it correctly, Visual Studio ships installation components that register an individual service on the target system and let the Services Control Manager know about its existence.

The Service Designer screen of the service base class file is used to add the appropriate installers to the project. To show this screen, you double-click on the file name in the Solution Explorer tree. Right-clicking on the gray area will bring up a context menu where the Add Installer item can be found. Clicking on it will automatically add the proper installer code.

To successfully deploy the service in the target computer, it's necessary to establish values for some properties in the service installation code. These properties are: **`Account`**, which indicates the account type under which the service will run; **`Description`**, which indicates a brief comment that explains the purpose of the service; **`DisplayName`**, which indicates the friendly name that identifies the service; **`ServiceName`**, which indicates the name used by the system to identify it; and **`StartType`**, which indicates how and when this service is started. The Designer View of the project installer code file (**ProjectInstaller,cs**) is used to set these values.

Chapter 4 Backup Files Service

So far, all code pieces needed to build a Windows Service project have been seen. Now, the most important thing is to give a purpose to the service that will be created. In this case, a backup files service will be developed.

Defining requirements

As explained previously, the purpose of the service will be to back up a set of files. The requirements for this service are the following.

- The set of files to be backed up will be in a particular folder in the target computer.
- The backup process needs to be done out of working time.
- All files in the folder will be added to a zip file.
- The zip file will be copied to a specific folder in the target computer.

Task list

In order to fulfill the previous requirements list, the following tasks need to be accomplished:

- Create a configuration file that specifies the folder to be backed up, the backup file destination folder, and the date and time in which the backup process will be executed. The file will be in XML format.
- Create a method that will read the parameters stored in the configuration file and let them be visible for the entire service application.
- Create a separate class that will be in charge of the backup process.
- Execute the backup process every time the condition established in the parameter's file is met.

For the purpose of this book, a Windows Service with specific function will be developed. In this case, a backup files service will be written using C# as a programming language, and it will make a compressed copy of a specific folder in the target system.

Creating the XML configuration file

As discussed previously, Windows Services has no user interaction capabilities. So, the only communication method available is using configuration files, whether these configuration files are written in a text editor or by using a desktop application built for that purpose.

The XML configuration file for Backup Files Windows Service will look like the following.

Code Sample 12

```
<?xml version="1.0" encoding="utf-8"?>
<Parameters>
<Backup source="C:\Documents" destination="D:\Backups"dayofweek="0"
hour="04:40:00"/>
</Parameters>
```

The root node of XML is called **`Parameters`**, and is intended to hold any kind of action the service would need to perform. Each action that will be executed by the service is stored as a child node. In this case, **`Backup`** child node attributes stores all parameters for backing up files.

Backup node attributes

The attributes of the **Backup** node are the following:

- Source **–** The folder that contains the files to be backed up
- Destination – The folder that will store the zip backup file
- Dayofweek – The day of the week, starting with 1 for Sunday, in which the backup process will be executed; 0 (zero) means every day
- Hour – The time of the day for executing the backup process

Creating the method that will read the parameters

Once the XML parameters file is created, the service will need the ability to read it and store these parameters for using them later. A method with this purpose will be added to the service definition class. The code will look like the following sample.

Code Sample 13

```
private void check_parameters()
{
    if (!System.IO.Directory.Exists(this.HomeDir + "\\parameters"))
    {
        System.IO.Directory.CreateDirectory(this.HomeDir + "\\parameters");
        this.LogEvent(String.Format("MonitorService: parameters file folder was just
been created"), EventLogEntryType.Information);
        this.IsReady = false;

    }
    else
    {
        if (System.IO.File.Exists(this.HomeDir + "\\parameters\\srvparams.xml"))
        {
            Boolean docparsed = true;
            XmlDocument parametersdoc = new XmlDocument();

            try
            {
```

```
                parametersdoc.Load(this.HomeDir + "\\parameters\\srvparams.xml");
            }
            catch (XmlException ex)
            {
                docparsed = false;
                this.IsReady = false;
                this.LogEvent(String.Format("Parameters file couldn't be read: {0}",
ex.Message), EventLogEntryType.Error);
            }

            if (docparsed)
            {
                XmlNode BackupParameters = parametersdoc.ChildNodes.Item(1).ChildNode
s.Item(0);
                this.source_path = BackupParameters.Attributes.GetNamedItem("source")
.Value.Trim();
                this.destination_path = BackupParameters.Attributes.GetNamedItem("des
tination").Value.Trim();
                this.dayofweek = Convert.ToInt32(BackupParameters.Attributes.GetNamed
Item("dayofweek").Value.Trim());
                this.time = BackupParameters.Attributes.GetNamedItem("hour").Value.Tr
im();
                 this.IsReady = true;

                this.LogEvent(String.Format("Backup Service parameters were loaded"),
 EventLogEntryType.Information);
            }

            parametersdoc = null;
        }
        else
        {
            this.LogEvent(String.Format("Backup Service parameters file doesn't exist
"), EventLogEntryType.Error);
            this.IsReady = false;
        }
    }
}
```

A **HomeDir** property is added to the class definition, in order to store the folder name in which service application will be installed. Also, an **IsReady** property is added to tell the service if the working parameters have been loaded from the XML file.

The method checks to see if a folder named **parameters** exists in the **HomeDir** folder. If not, the **System.IO.Directory.CreateDirectory** method is used to create it, and an entry is written in the Windows Event Log. The value of the **IsReady** property is set to **false** so the service won't execute the backup process, since there's no working parameters loaded.

Otherwise, the method looks for a **srvparams.xml** file in the **parameters** folder in order to load the service working parameters. If the file doesn't exist, the method just writes an entry in the Windows Event Log, and the **`IsReady`** property value is set to **`false`**. If the file does exist, the **`check_parameters()`** method tries to parse the content of the XML file. If parsing fails, an entry in the Windows Event Log is written and the **`IsReady`** property is also set to **`false`**. Otherwise, the working parameters are stored in their respective properties into the class definition and the **`IsReady`** property is set to **`true`**.

The **`check_parameters()`** method is called when service execution starts. This will occur when the **`OnStart`** event is triggered and its associated method is executed.

Now, the entire service class definition code looks like the following sample.

Code Sample 14

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Linq;
using System.ServiceProcess;
using System.Text;
using System.Threading.Tasks;
using System.Xml;

namespace monitorservice
{
    public partial class monitorservice : ServiceBase
    {
        private System.Timers.Timer serviceTimer = null;
        private string HomeDir = (new System.IO.DirectoryInfo(System.AppDomain.Curren
tDomain.BaseDirectory)).FullName.Trim();
        private string source_path = "";
        private string destination_path = "";
        private int dayofweek = 0;
        private string time = "";

        public monitorservice()
        {
            InitializeComponent();
        }

        protected override void OnStart(string[] args)
        {
            if (!System.Diagnostics.EventLog.SourceExists("MonitorService"))
                System.Diagnostics.EventLog.CreateEventSource("MonitorService", "Appl
ication");

            this.LogEvent(String.Format("MonitorService starts on {0} {1}", System.Da
teTime.Now.ToString("dd-MMM-
```

```
yyyy"), DateTime.Now.ToString("hh:mm:ss tt")), EventLogEntryType.Information);

            this.check_parameters(); //Need to load service behavior parameters

            this.serviceTimer = new System.Timers.Timer(300);
            this.serviceTimer.AutoReset = true;
            this.serviceTimer.Elapsed += new System.Timers.ElapsedEventHandler(this.t
imer_Elapsed);
            this.serviceTimer.Start();
        }

        private void timer_Elapsed(object sender, System.Timers.ElapsedEventArgs e)
        {

        }

        protected override void OnStop()
        {
            this.serviceTimer.Stop();
            this.serviceTimer.Dispose();
            this.serviceTimer = null;

            this.LogEvent(String.Format("MonitorService stops on {0} {1}", System.Dat
eTime.Now.ToString("dd-MMM-
yyyy"), DateTime.Now.ToString("hh:mm:ss tt")), EventLogEntryType.Information);

        }

        private void LogEvent(string message, EventLogEntryType entryType)
        {
            System.Diagnostics.EventLog eventLog = new System.Diagnostics.EventLog();

            eventLog = new System.Diagnostics.EventLog();
            eventLog.Source = "MonitorService";
            eventLog.Log = "Application";
            eventLog.WriteEntry(message, entryType);

        }

        private void check_parameters()
        {
            if (!System.IO.Directory.Exists(this.HomeDir + "\\parameters"))
            {
                System.IO.Directory.CreateDirectory(this.HomeDir + "\\parameters");
                this.LogEvent(String.Format("MonitorService: parameters file folder w
as just been created"), EventLogEntryType.Information);
            }
            else
            {
                if (System.IO.File.Exists(this.HomeDir + "\\parameters\\srvparams.xml
"))
                {
                    Boolean docparsed = true;
                    XmlDocument parametersdoc = new XmlDocument();
```

```
                    try
                    {
                        parametersdoc.Load(this.HomeDir + "\\parameters\\srvparams.xml");
                    }
                    catch (XmlException ex)
                    {
                        docparsed = false;
                        this.LogEvent(String.Format("Parameters file couldn't be read: {0}", ex.Message), EventLogEntryType.Error);
                    }

                    if (docparsed)
                    {
                        XmlNode BackupParameters = parametersdoc.ChildNodes.Item(1).ChildNodes.Item(0);
                        this.source_path = BackupParameters.Attributes.GetNamedItem("source").Value.Trim();
                        this.destination_path = BackupParameters.Attributes.GetNamedItem("destination").Value.Trim();
                        this.dayofweek = Convert.ToInt32(BackupParameters.Attributes.GetNamedItem("dayofweek").Value.Trim());
                        this.time = BackupParameters.Attributes.GetNamedItem("hour").Value.Trim();

                        this.LogEvent(String.Format("Backup Service parameters were loaded"), EventLogEntryType.Information);
                    }

                    parametersdoc = null;
                }
                else
                {
                    this.LogEvent(String.Format("Backup Service parameters file doesn't exist"), EventLogEntryType.Error);
                }
            }
        }

    }
}
```

Creating a class for the backup process

In order to keep project maintenance easy, the file backup process will be coded in a separate class definition. To accomplish this, a class type item needs to be added into project. Right-click on the project name node in the Solution Explorer tree, and click on the **Class** item of the **Add** sub-menu. This will bring up the Add New Item dialog box.

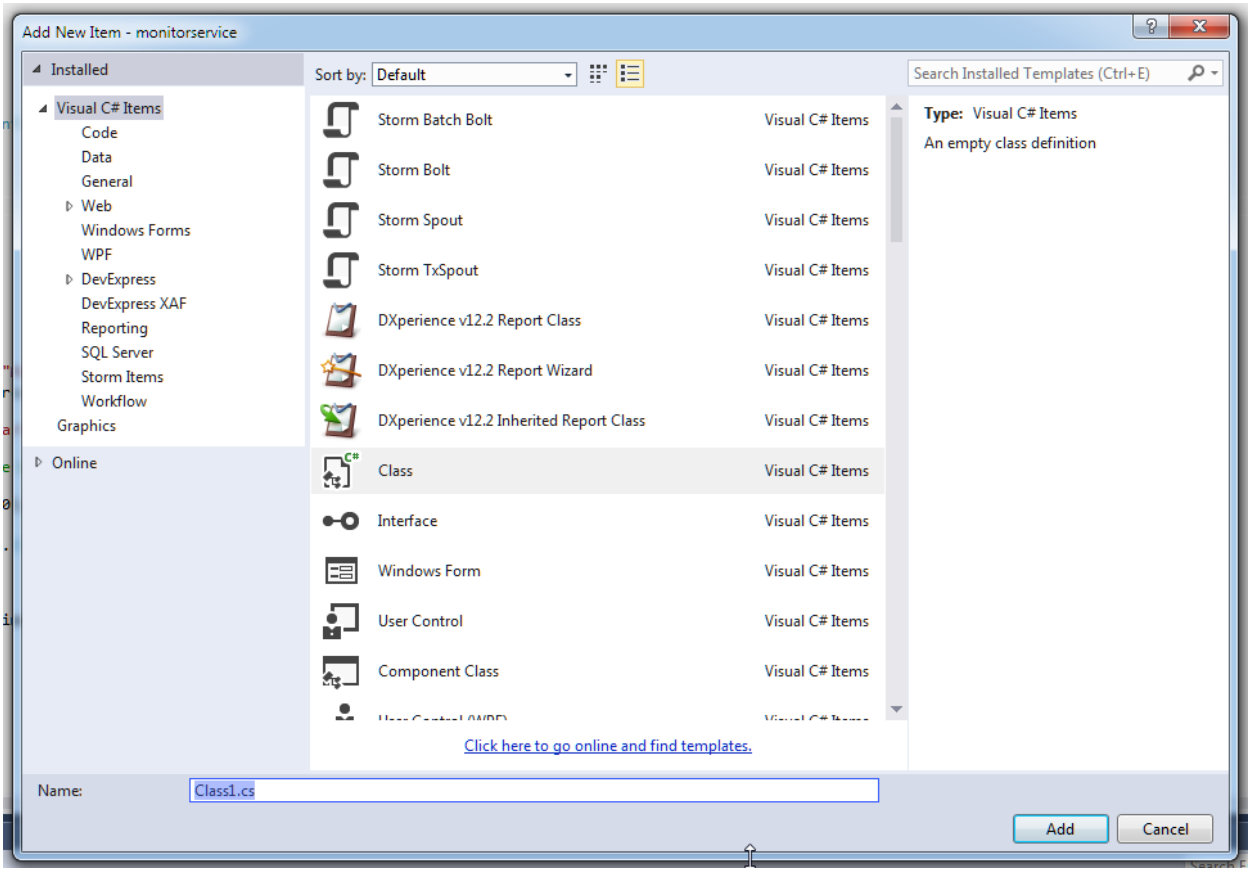

Figure 9: Add New Item dialog

Type the class name on the proper textbox, and click **Add** to add the following code into the project.

Code Sample 15

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace monitorservice
{
    class backupfiles
```

```
    {
    }
}
```

The entire class will be written up on the baseline code added to the project. One thing that must be taken into account is that the requirements previously mentioned dictate that backup must be stored in a ZIP file. The **Ionic.Zip** library will be used for this purpose, and can be downloaded here. After downloading, the library files need to be copied into the project folder and added into the project References node.

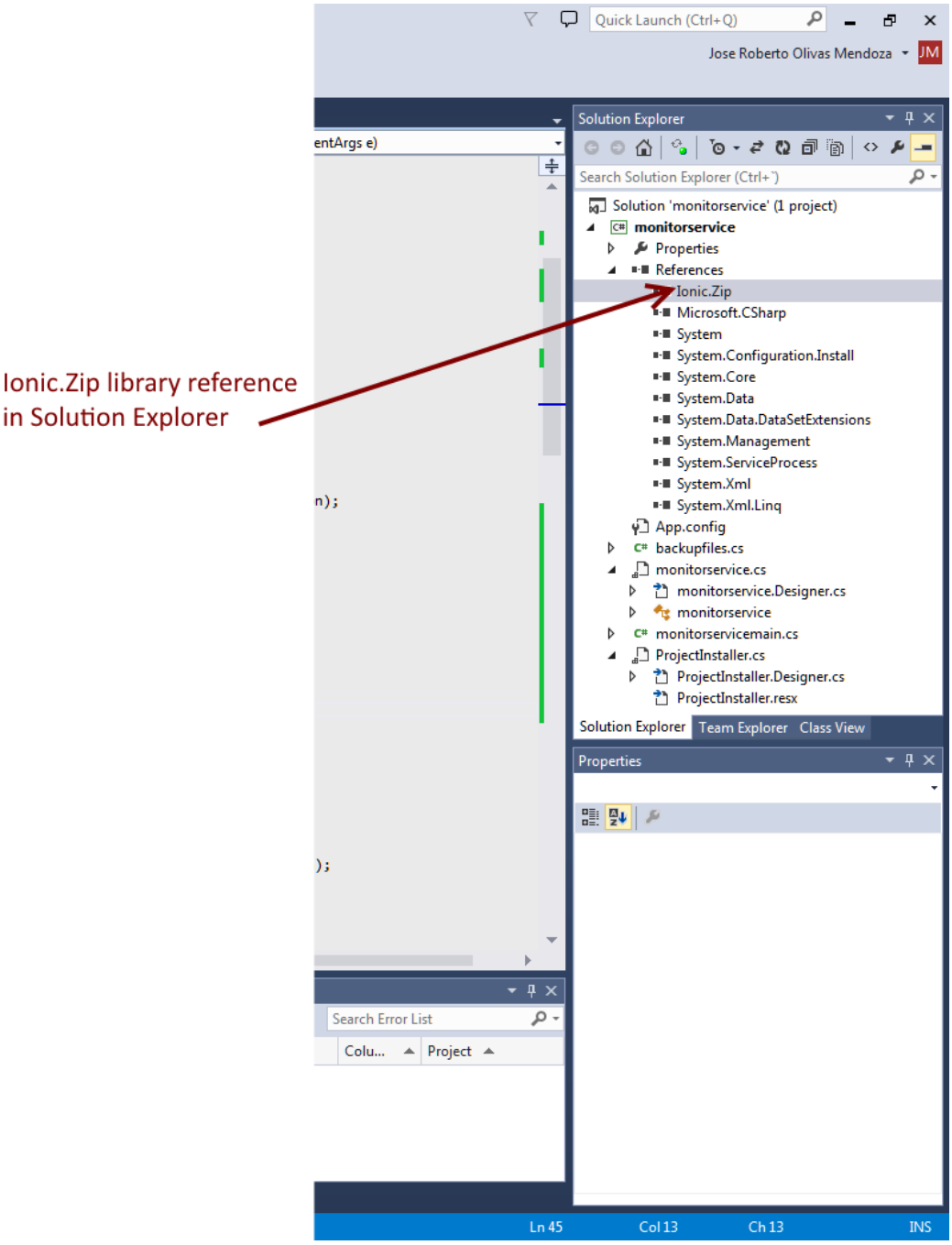

Figure 10: Ionic.Zip library added to References node

The entire code for the **`backupfiles`** class is shown in the following snippet.

Code Sample 16

```
using Ionic.Zip;
```

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace monitorservice
{
    public class backupfiles
    {
        public string source_path = "";
        public string destination_path = "";
        public string error_message = "";

        public Boolean DoBackup()
        {
            Boolean result = default(Boolean);

            string destFileName = this.destination_path + "\\backup_" + System.DateTi
me.Now.ToString("MMM-dd-yyyy") + "-
" + System.DateTime.Now.ToString("hh:mm:ss").Replace(":","-")+".zip";

            using (ZipFile zipFile = new ZipFile())
             {
                    string[] fileList = new string[1];
                    result = true;

                    this.error_message = "";

                    try
                    {
                        fileList = Directory.GetFiles(this.source_path + "\\");
                    }
                    catch (Exception exception)
                    {
                        this.error_message = String.Format("MonitorService: Folder fi
le list can't be read: {0}",exception.Message);
                        result = false;
                    }
                    finally
                    {
                        if (result)
                        {
                            zipFile.Encryption = EncryptionAlgorithm.WinZipAes256;
                            zipFile.AddProgress += (this.zipFile_AddProgress);
                            zipFile.AddDirectory(this.source_path);

                            zipFile.Save(destFileName);
                        }

                    }
             }
```

```
            return (result);
        }

        void zipFile_AddProgress(object sender, AddProgressEventArgs e)
         {
            switch (e.EventType)
                {
                case ZipProgressEventType.Adding_Started:
                    break;
                case ZipProgressEventType.Adding_AfterAddEntry:
                    break;
                case ZipProgressEventType.Adding_Completed:
                    break;
                }

         }
    }
}
```

This class has only two methods. The **DoBackup()** method performs the backup process and stores the files from the source folder in a compressed ZIP file. The **zipFile_AddProgress()** method is a delegate for the **AddProgress** event of the **ZipFile** class. This event is fired every time there is a change in compression work progress.

Executing the backup process

As previously described, the service definition class contains a **Timer** object, which is programmed to fire an **Elapsed** event every 300 milliseconds. Every time the **Elapsed** event is triggered, the method **timer_Elapsed()** is executed. So, the backup process will be executed within this method.

Checking that backup conditions are met

First, the **timer_Elapsed()** method needs to check to see if the backup weekday and time read from the XML configuration file match the current weekday and time at the moment when the method is executed. This can be done with the following code.

Code Sample 17

```
if (this.weekday != 0) //Need to know if current weekday matches parameter's weekday
{
    if (((int)System.DateTime.Now.DayOfWeek) + 1 != this.weekday)
    {
        return;
    }
}

if (System.TimeSpan.Parse(this.time) >
DateTime.Now.TimeOfDay) //If current daytime is earlier than defined in parameters, t
he process is stoped
```

```
{
    return;
}
```

The first thing the method does is to check if weekday parameter is zero. If it is, it means that the backup process must be performed every day. Otherwise, the method needs to check if the weekday parameter matches the current weekday. This is done by comparing weekday parameter value against **System.DateTime.Now.DayOfWeek** property value. Since the **DayOfWeek** property is zero-base indexed, it's needed to add 1 before comparison, because the parameter established in the XML file is one-base indexed. If comparison gets **true** as a result, the execution continues. Otherwise, the method returns control to the calling process.

The next thing to do is to check if backup time parameter value matches current time value. The first thing the code does is to parse the time string parameter value, in order to transform it in a **TimeSpan** value. Then, compares the result against the **TimeOfDay** property of **DateTime.Now**, and if the values don't match, the method returns control to the calling process. Otherwise, the method execution continues.

Running the backup process

If parameter conditions are met, the backup process is executed. The following code accomplishes this.

Code Sample 18

```
this.BackupEngine.source_path = this.source_path;
this.BackupEngine.destination_path = this.destination_path;
this.BackupEngine.DoBackup();
```

At this point, a **BackupEngine** property was added to the service class definition, in order to keep an instance of the **backupfiles** class available while the service is running. The values of **source_path** and **destination_path** service class properties are passed to the respective properties in the **BackupEngine** instance, and the **DoBackup()** method is executed to start the backup process.

An issue to be solved

The **timer_Elapsed()** method is executed every 300 milliseconds while the service is running, and this method performs the backup process if the parameter conditions are met. As mentioned, an instance of the **backupfiles** class is available along service lifetime. So, the first time that backup conditions are met, **DoBackup()** method is executed and the control returns outside the **timer_Elapsed()** method.

What will happen if, next time, the **timer_Elapsed()** method is executed and backup conditions are met? The **DoBackup()** method will be executed again, and if a previous backup is started, it's likely that the service will crash. To avoid this situation, a property named **IsBusy** will be added to the **backupfiles** class in order to flag when the backup process is in progress. Now, the code for the **backupfiles** class will look like the following snippet.

Code Sample 19

```
using Ionic.Zip;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace monitorservice
{
    public class backupfiles
    {
        public string source_path = "";
        public string destination_path = "";
        public string error_message = "";
        public Boolean IsBusy = false;

        public Boolean DoBackup()
        {
            Boolean result = default(Boolean);
            this.IsBusy = false;

            string destFileName = this.destination_path + "\\backup_" + System.DateTi
me.Now.ToString("MMM-dd-yyyy") + "-
" + System.DateTime.Now.ToString("hh:mm:ss").Replace(":","-")+".zip";

            using (ZipFile zipFile = new ZipFile())
             {
                    string[] fileList = new string[1];
                    result = true;

                    this.error_message = "";

                    try
                    {
                        fileList = Directory.GetFiles(this.source_path + "\\");
                    }
                    catch (Exception exception)
                    {
                        this.error_message = String.Format("MonitorService: Folder fi
le list can't be read: {0}",exception.Message);
                        result = false;
                    }
                    finally
                    {
                        if (result)
                        {
                            this.IsBusy = true;

                            zipFile.Encryption = EncryptionAlgorithm.WinZipAes256;
                            zipFile.AddProgress += (this.zipFile_AddProgress);
```

```
                            zipFile.AddDirectory(this.source_path);

                            zipFile.Save(destFileName);

                            this.IsBusy = false;
                        }

                    }
                }

            return (result);
        }

        void zipFile_AddProgress(object sender, AddProgressEventArgs e)
         {
            switch (e.EventType)
                {
                case ZipProgressEventType.Adding_Started:
                    break;
                case ZipProgressEventType.Adding_AfterAddEntry:
                    break;
                case ZipProgressEventType.Adding_Completed:
                    break;
                }

         }
    }
}
```

Every time the backup process starts, the **IsBusy** property is set to the value of **true**, indicating that the method won't be allowed to execute until the current process finishes. When backup process finishes, **IsBusy** is set to **false**.

The entire code for the timer_Elapsed() event

Now, the code for the event will look like the code shown in the following sample.

Code Sample 20

```
private void timer_Elapsed(object sender, System.Timers.ElapsedEventArgs e)
{
    if (!this.IsReady)
    {
        return;
    }

    if (this.weekday != 0) //Need to know if current weekday matches parameter's week
day
    {
        if (((int)System.DateTime.Now.DayOfWeek) + 1 != this.weekday)
        {
            return;
        }
    }

    if (DateTime.Now.TimeOfDay < System.TimeSpan.Parse(this.time)) //If current dayti
me is earlier than defined in parameters, the process is stoped
    {
        return;
    }

    if (this.BackupEngine.IsBusy)  //If backup process was previously started we do n
othing
    {
        return;
    }

    this.BackupEngine.source_path = this.source_path;
    this.BackupEngine.destination_path = this.destination_path;
    this.BackupEngine.DoBackup();

}
```

After backup conditions are checked, the property **`IsBusy`** of the **`BackupEngine`** instance is checked. If the value for the property is **`true`**, the method stops its execution and returns the control to the calling process. Otherwise, the backup process is executed.

The puzzle has been assembled

Now, all the pieces have been gathered together, and the service class code looks like the following snippet.

Code Sample 21

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Linq;
using System.ServiceProcess;
using System.Text;
using System.Threading.Tasks;
using System.Xml;

namespace monitorservice
{
    public partial class monitorservice : ServiceBase
    {
        private System.Timers.Timer serviceTimer = null;
        private string HomeDir = (new System.IO.DirectoryInfo(System.AppDomain.Curren
tDomain.BaseDirectory)).FullName.Trim();
        private string source_path = "";
        private string destination_path = "";
        private int weekday = 0;
        private string time = "";
        private Boolean IsReady = false;
        private backupfiles BackupEngine = new backupfiles();

        public monitorservice()
        {
            InitializeComponent();
        }

        protected override void OnStart(string[] args)
        {
            if (!System.Diagnostics.EventLog.SourceExists("MonitorService"))
                System.Diagnostics.EventLog.CreateEventSource("MonitorService", "Appl
ication");

            this.LogEvent(String.Format("MonitorService starts on {0} {1}", System.Da
teTime.Now.ToString("dd-MMM-
yyyy"), DateTime.Now.ToString("hh:mm:ss tt")), EventLogEntryType.Information);

            this.check_parameters(); //Need to load service behavior parameters

            this.serviceTimer = new System.Timers.Timer(300);
            this.serviceTimer.AutoReset = true;
            this.serviceTimer.Elapsed += new System.Timers.ElapsedEventHandler(this.t
imer_Elapsed);
```

```
            this.serviceTimer.Start();
        }

        private void timer_Elapsed(object sender, System.Timers.ElapsedEventArgs e)
        {
            if (!this.IsReady)
            {
                return;
            }

            if (this.weekday != 0) //Need to know if current weekday matches paramete
r's weekday
            {
                if (((int)System.DateTime.Now.DayOfWeek) + 1 != this.weekday)
                {
                    return;
                }
            }

            if (DateTime.Now.TimeOfDay < System.TimeSpan.Parse(this.time)) //If curre
nt daytime is earlier than defined in parameters, the process is stoped
            {
                return;
            }

            if (this.BackupEngine.IsBusy)  //If backup process was previously started
 we do nothing
            {
                return;
            }

            this.BackupEngine.source_path = this.source_path;
            this.BackupEngine.destination_path = this.destination_path;
            this.BackupEngine.DoBackup();

        }

        protected override void OnStop()
        {
            this.serviceTimer.Stop();
            this.serviceTimer.Dispose();
            this.serviceTimer = null;

            this.LogEvent(String.Format("MonitorService stops on {0} {1}", System.Dat
eTime.Now.ToString("dd-MMM-
yyyy"), DateTime.Now.ToString("hh:mm:ss tt")), EventLogEntryType.Information);

        }

        private void LogEvent(string message, EventLogEntryType entryType)
        {
            System.Diagnostics.EventLog eventLog = new System.Diagnostics.EventLog();

            eventLog = new System.Diagnostics.EventLog();
```

```
            eventLog.Source = "MonitorService";
            eventLog.Log = "Application";
            eventLog.WriteEntry(message, entryType);

        }

        private void check_parameters()
        {
            if (!System.IO.Directory.Exists(this.HomeDir + "\\parameters"))
            {
                System.IO.Directory.CreateDirectory(this.HomeDir + "\\parameters");
                this.LogEvent(String.Format("MonitorService: parameters file folder w
as just been created"), EventLogEntryType.Information);
                this.IsReady = false;
            }
            else
            {
                if (System.IO.File.Exists(this.HomeDir + "\\parameters\\srvparams.xml
"))
                {
                    Boolean docparsed = true;
                    XmlDocument parametersdoc = new XmlDocument();

                    try
                    {
                        parametersdoc.Load(this.HomeDir + "\\parameters\\srvparams.xm
l");
                    }
                    catch (XmlException ex)
                    {
                        docparsed = false;
                        this.IsReady = false;
                        this.LogEvent(String.Format("Parameters file couldn't be read
: {0}", ex.Message), EventLogEntryType.Error);
                    }

                    if (docparsed)
                    {
                        XmlNode BackupParameters = parametersdoc.ChildNodes.Item(1).C
hildNodes.Item(0);
                        this.source_path = BackupParameters.Attributes.GetNamedItem("
source").Value.Trim();
                        this.destination_path = BackupParameters.Attributes.GetNamedI
tem("destination").Value.Trim();
                        this.weekday = Convert.ToInt32(BackupParameters.Attributes.Ge
tNamedItem("dayofweek").Value.Trim());
                        this.time = BackupParameters.Attributes.GetNamedItem("hour").
Value.Trim();

                        this.IsReady = true;

                        this.LogEvent(String.Format("Backup Service parameters were l
oaded"), EventLogEntryType.Information);
```

```
                    }

                    parametersdoc = null;
                }
                else
                {
                    this.LogEvent(String.Format("Backup Service parameters file doesn
't exist"), EventLogEntryType.Error);
                }
            }
        }

    }
}
```

Now it's time to build the solution and get the application executable file.

Chapter summary

To build the backup files service, the first thing that must be done is to create a XML file to store all the parameters needed to control service behavior. The root node of the XML file will be called **`Parameters`**, and will be intended to hold all the actions the service will perform. Each action will be stored as a child node, and for backup service, a Backup child node will be stored with the following attributes: **`Source`**, which indicates the folder in which the files to be backed up are contained; **`Destination`**, which indicates the folder that will store the zip backup file; **`Dayofweek`**, which indicates the day of the week, starting with 1 for Sunday, in which the backup process will be executed (zero means every day); and **`Hour`**, which indicates the time of the day for executing backup process.

A method called **`check_parameters()`** is created for reading the XML file, using the **`XmlDocument`** .NET class. This method is called when service execution starts, and stores all the parameters in a set of corresponding properties defined in the service class definition.

A class named **`backupfiles`** is created to deal with the backup process. This class uses the **IonicZip** library to compress the backup. This library must be added as a reference into the Visual Studio project, and can be downloaded here. The service class definition holds an instance for this class in a property called **`BackupEngine`**.

Every time the **`timer_Elapsed()`** method of the service class definition is executed, it checks to see if the backup conditions are met. If they are, the method inquires if a previous backup process is being performed, by using a property named **`IsBusy`** that belongs to the **`backupfiles`** class, and comparing its value to **`true`**. If it is, the method execution stops. Otherwise, the backup process starts. When it is complete, the value of **`IsBusy`** is set to **`false`**, in order to allow the execution of a new backup process.

At the end, an executable file is obtained when the solution is built.

Chapter 5 Deploying the Service

Once the application executable file is built, it needs to be deployed in the computer where it will work—either a workstation or server. Trying to build an msi installation file could be the most reliable way to do this. But there's also a simpler way to accomplish the deployment process: creating a .BAT file in which the installutil.exe tool will be used.

Installer tool

The Installer tool is a command-line utility that allows you to install and uninstall server resources by executing the installer components in specified assemblies. This tool works in conjunction with classes in the **`System.Configuration.Install`** namespace.

This tool is automatically installed with Visual Studio. To run the tool, use the Developer Command Prompt (or the Visual Studio Command Prompt in Windows 7).

At the command prompt, type the following:

Code Sample 22

```
installutil [/u[ninstall]] [options] assembly [[options] assembly] ...
```

Table 1: Parameters

Argument	Description
Assembly	The file name of the assembly in which to execute the installer components. Omit this parameter if you want to specify the assembly's strong name by using the **/AssemblyName** option.

Table 2: Options

Option	Description
h[elp] -or-	Displays command syntax and options for the tool.

Option	Description
/?	
/help assembly -or- /? assembly	Displays additional options recognized by individual installers within the specified assembly, along with command syntax and options for InstallUtil.exe. This option adds the text returned by each installer component's Installer.HelpText property to the help text of InstallUtil.exe.
/AssemblyName "assemblyName ,Version=major.minor.build.revision ,Culture=locale ,PublicKeyToken=publicKeyToken"	Specifies the strong name of an assembly, which must be registered in the global assembly cache. The assembly name must be fully qualified with the version, culture, and public key token of the assembly. The fully qualified name must be surrounded by quotes. For example, "myAssembly, Culture=neutral, PublicKeyToken=0038abc9deabfle5, Version=4.0.0.0" is a fully qualified assembly name.
/InstallStateDir=[directoryName]	Specifies the directory of the .InstallState file that contains the data used to uninstall the assembly. The default is the directory that contains the assembly.
/LogFile= [filename]	Specifies the name of the log file where installation progress is recorded. By default, if the **/LogFile** option is omitted, a log file named *assemblyname*.InstallLog is created. If *filename* is omitted, no log file is generated.
/LogToConsole ={true|false}	If true, displays output to the console. If false (the default), suppresses output to the console.
/ShowCallStack	Outputs the call stack to the log file if an exception occurs at any point during installation.

Option	Description
/u [ninstall]	Uninstalls the specified assemblies. Unlike the other options, /u applies to all assemblies regardless of where the option appears on the command line.

Remarks

.NET Framework applications consist of traditional program files and associated resources, such as message queues, event logs, and performance counters that must be created when the application is deployed. You can use an assembly's installer components to create these resources when your application is installed, and to remove them when your application is uninstalled. Installutil.exe detects and executes these installer components.

You can specify multiple assemblies on the same command line. Any option that occurs before an assembly name applies to that assembly's installation. Except for **/u** and **/AssemblyName**, options are cumulative but can be overridden. That is, options specified for one assembly apply to all subsequent assemblies unless the option is specified with a new value.

If you run Installutil.exe against an assembly without specifying any options, it places the following three files into the assembly's directory:

- InstallUtil.InstallLog – Contains a general description of the installation progress
- *assemblyname*.InstallLog – Contains information specific to the commit phase of the installation process
- *assemblyname*.InstallState – Contains data used to uninstall the assembly

Installutil.exe uses reflection to inspect the specified assemblies and to find all Installer types that have the **`System.ComponentModel.RunInstallerAttribute`** attribute set to **true**. The tool then executes either the **`Installer.Install`** or the **`Installer.Uninstall`** method on each instance of the Installer type. Installutil.exe performs installation in a transactional manner; that is, if one of the assemblies fails to install, it rolls back the installations of all other assemblies. Uninstall is not transactional.

BAT installation file

This file can be created using a text editor like Notepad. Once the file is created, it should look like the following sample.

Code Sample 23

```
@ECHO OFF
CLS
ECHO Installing Windows Service
```

```
INSTALLUTIL.EXE monitorservice.exe
ECHO Service has been installed
PAUSE
```

BAT uninstall file

Likewise, the uninstall process can be performed using a .BAT file, and should look like the following snippet.

Code Sample 24

```
@ECHO OFF
CLS
ECHO Uninstalling Windows Service
INSTALLUTIL.EXE /U monitorservice.exe
ECHO Service has been uninstalled
PAUSE
```

Both files look almost the same. The only difference is the /U option used for INSTALLUITL.EXE in the uninstall file.

Note: To ensure the service will be properly deployed, Installutil.exe must be included in the service distribution package.

Service distribution package

To deploy the service executable, a distribution package is needed. This package will contain the necessary files to make a successful installation in the target computer. The following files must be included in the service distribution package:

- Installutil.exe (shipped with Visual Studio)
- Monitorservice.exe (the service executable file)
- Ionic.Zip.dll (library used for ZIP creation)
- The XML file with service execution parameters
- BAT installation file
- BAT uninstall file

Tip: For an easy distribution, the package can be shipped in a zip file, which can be decompressed in the target computer at install time.

Chapter summary

An msi file could be a reliable way to deploy the service executable file in the target computer, but using a .BAT command file with the Installer tool is an easy way to do it.

The Installer tool (Installutil.exe) is a command-line utility that allows you to install and uninstall server resources. This tool works in conjunction with the **`System.Configuration.Install`** namespace. This tool is automatically installed with Visual Studio.

Installutil.exe uses reflection to inspect the specified assemblies and to find all Installer types that have the **`System.ComponentModel.RunInstallerAttribute`** attribute set to **`true`**. The tool then executes either the **`Installer.Install`** or the **`Installer.Uninstall`** method on each instance of the **`Installer`** type.

A couple of .BAT files must be included in the distribution package; one for installing the service executable file, and another one to perform uninstallation of the executable from the target computer.

Chapter 6 Creating a User Interface to Configure the Service

Overview

Throughout this book it has been stated that a Windows Service has no interface, and that a configuration file is the proper way to control the behavior of a service. Usually, this configuration file can be written in a text editor and saved to disk, but it looks more professional if a program with a user interface is built for this purpose. Many known services (such as Filezilla FTP Server) make these programs available to the user, in order to create or modify its own configuration files in an intuitive and easy way.

This chapter is intended to build a Windows Forms program named MonitorServiceGUI, which will deal with creating and editing the configuration file needed by the service that was built previously.

Creating the solution in Visual Studio

The first step is to create a Windows Forms project named **MonitorServiceGUI** using Visual Studio. The programming language to use will be C# and the target framework will be .NET 4.

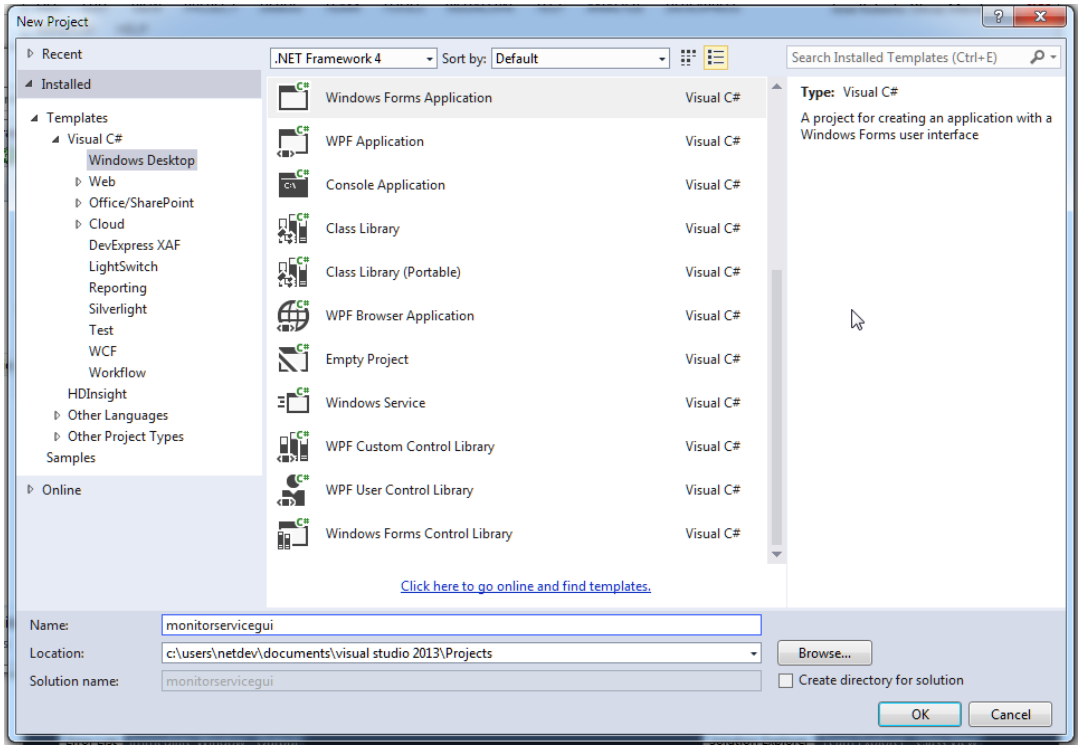

Figure 11: Figure 11: MonitorServiceGUI project dialog

When Visual Studio ends project creation, two files named **Form1.cs** and **Program.cs** can be found in the Solution Explorer's tree. For clarity, these files will be renamed to **mainform.cs** and **mainprogram.cs**, respectively. Now, the Solution Explorer will look like the following figure.

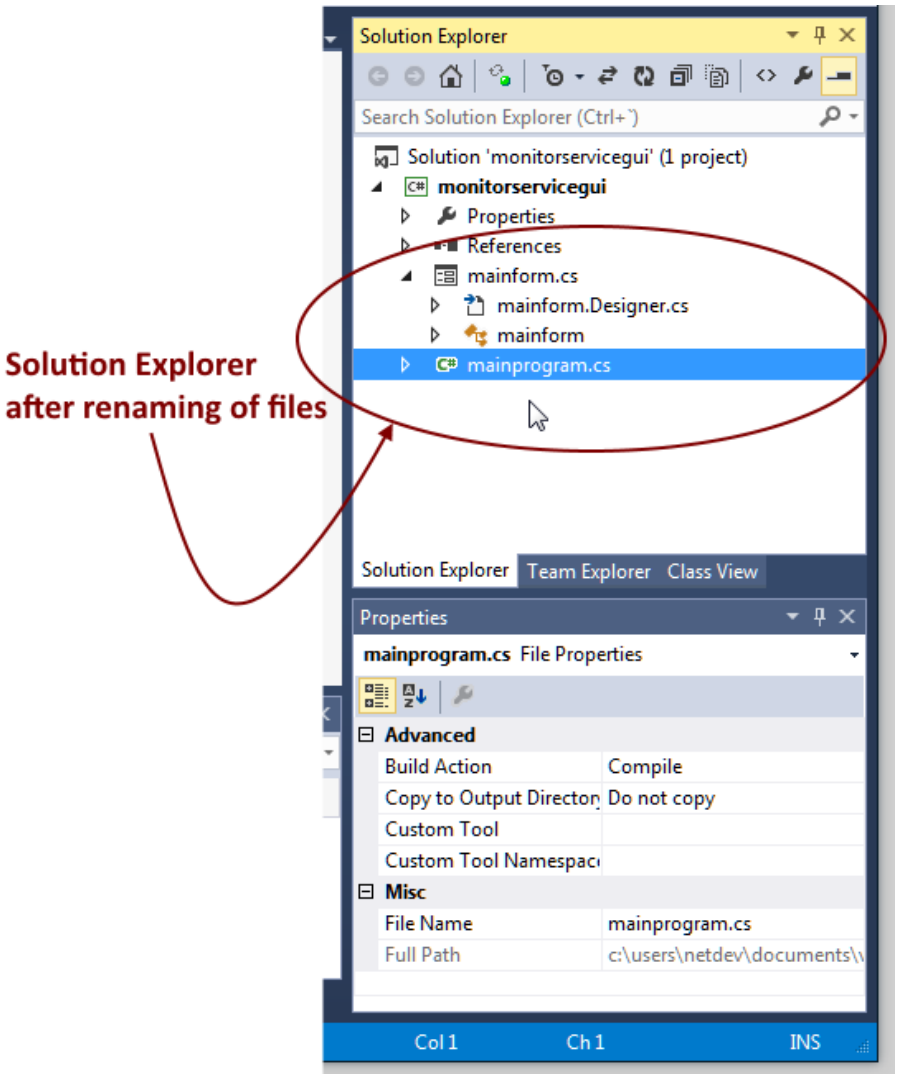

Figure 12: Figure 12: Solution Explorer for MonitorServiceGUI project

The **mainprogram.cs** file contains the application's entry point, which is handled by a static class named **`mainprogram`**. This class has one method named **`Main`**, which sets the application environment, creates an instance of the **`mainform`** class (defined in **mainform.cs**), and shows it on the screen. It can be seen in the following code sample.

Code Sample 25

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace monitorservicegui
{
    static class mainprogram
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new mainform());
        }
    }
}
```

The **mainform** will be the only form in the project. This will contain all GUI elements in order to set up the service behavior parameters, and will deal with creation of the configuration file needed for the service to run.

Setting up the project's main form

The main form needs only a few graphic elements to accomplish its purpose. These elements are:

- A combo box to select the day of the week in which the backup will be done. The **DropDownStyle** property for this element must be set to **DropDownList**. Using the Designer, the item list will be filled with the names of all week days. The first name for this list will be *Every day*.
- A label placed to the left of the combo box, with the legend *Day of Week*
- A masked text box to enter the hour in which the backup will be done
- A label placed to the left of the masked text box, with the legend *Hour*
- A textbox to enter the path in which the files to be backed up are
- A label placed to the left of the previous textbox, with the legend *Source Path*
- A textbox to enter the path in which the zipped backup will be stored
- A label placed to the left of the previous textbox, with the legend *Destination Path*
- A Save button to store the parameters in an XML file
- A Cancel button to close the form and ignore the changes that could be made

Besides the elements mentioned previously, the following properties need to be set to finish the **mainform** design:

- **Text** – This property will be set to "Backup Service Interface."

- **`FormBorderStyle`** – This property will be set to FixedSingle.
- **`MaximizeBox`** – This property will bet set to False.
- **`CancelButton`** – This property will be set to the Cancel button graphic element mentioned previously.

When all graphic elements are placed in the form, and the values for the properties mentioned previously are set, the Designer View for the **`mainform`** will look like the following figure.

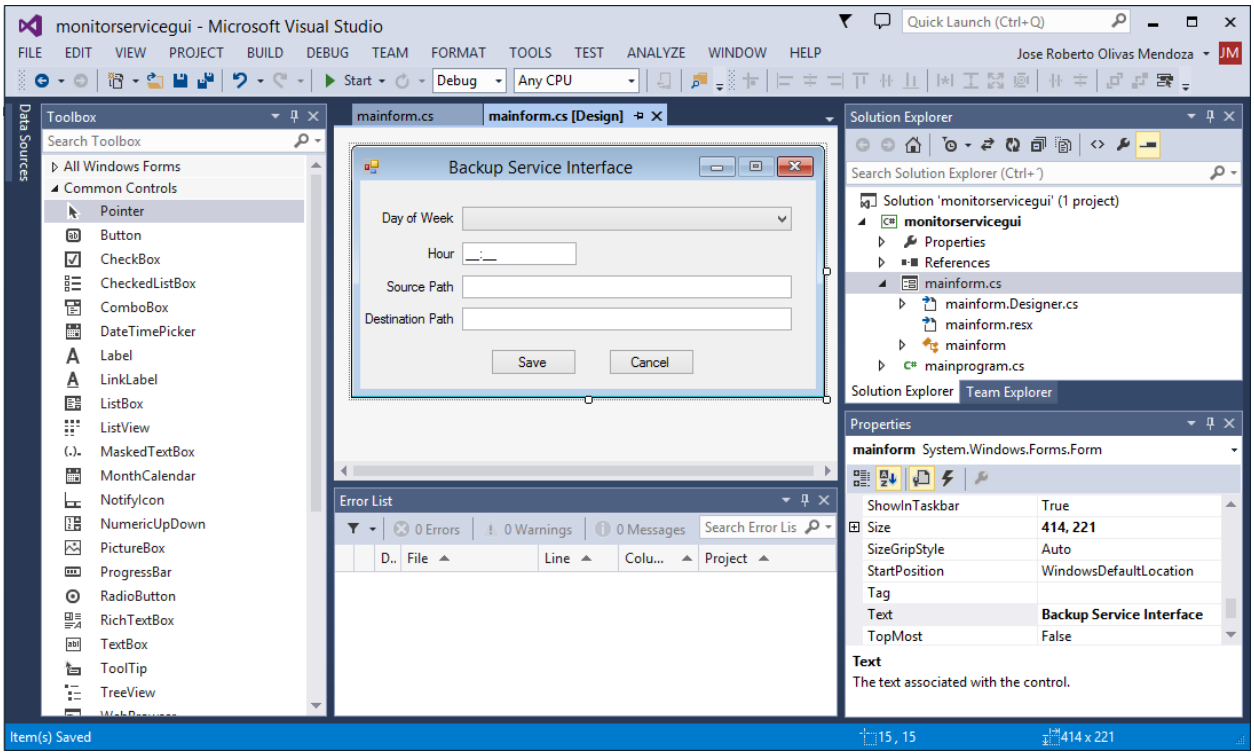

Figure 13: Mainform.cs Designer View

Looking for a previous XML parameters file

The first thing that must be done is to check if a XML parameters file already exists. If it does, all values stored in the file need to be passed to the graphic elements in the form, in order to show them to the user. If the file doesn't exist, the program must pass a set of initial values to the graphics elements in the form and show them. To do this, we'll use the **`Load`** event of **`mainform`**, as shown in the following code sample.

Code Sample 26

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
```

```
using System.Drawing;
using System.IO;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Xml;

namespace monitorservicegui
{
    public partial class mainform : Form
    {
        string HomeDir = Path.GetDirectoryName(Application.ExecutablePath).Trim();
        public mainform()
        {
            InitializeComponent();
        }

        private void mainform_Load(object sender, EventArgs e)
        {
            if (!this.check_parameters())
            {
                this.comboBox1.SelectedIndex = 0;
                this.comboBox1.Refresh();
                this.maskedTextBox1.Text = "00:00";
                this.maskedTextBox1.Refresh();
                this.textBox1.Text = "";
                this.textBox1.Refresh();
                this.textBox2.Text = "";
                this.textBox2.Refresh();
            }
        }

        private Boolean check_parameters()
        {
            Boolean result = default(Boolean);

            if (!System.IO.Directory.Exists(this.HomeDir + "\\parameters"))
            {
                System.IO.Directory.CreateDirectory(this.HomeDir + "\\parameters");
                result = false;
            }
            else
            {
                if (System.IO.File.Exists(this.HomeDir + "\\parameters\\srvparams.xml
"))
                {
                    result = true;
                    XmlDocument parametersdoc = new XmlDocument();

                    try
                    {
                        parametersdoc.Load(this.HomeDir + "\\parameters\\srvparams.xm
l");
                    }
```

```
                    catch
                    {
                        result = false;

                    }

                    if (result)
                    {
                        XmlNode BackupParameters = parametersdoc.ChildNodes.Item(1).C
hildNodes.Item(0);
                        this.textBox1.Text = BackupParameters.Attributes.GetNamedItem
("source").Value.Trim();
                        this.textBox1.Refresh();
                        this.textBox2.Text = BackupParameters.Attributes.GetNamedItem
("destination").Value.Trim();
                        this.textBox2.Refresh();
                        this.comboBox1.SelectedIndex = Convert.ToInt32(BackupParamete
rs.Attributes.GetNamedItem("dayofweek").Value.Trim());
                        this.comboBox1.Refresh();
                        this.maskedTextBox1.Text = BackupParameters.Attributes.GetNam
edItem("hour").Value.Trim();
                        this.maskedTextBox1.Refresh();
                    }

                    parametersdoc = null;
                }
                else
                {
                    result = false;
                }
            }

            return (result);
        }

    }
}
```

A separate method called **`check_parameters()`** is created with the purpose of inquiring the existence of the XML file. If the file exists, the method uses the **`XmlDocument`** object in order to parse the file and get all parameter values. Then, these values are passed to their corresponding graphic elements in the form. A value of **`true`** is returned in order to indicate that the file was found and all the parameters were properly loaded. Else, if the XML file is not found or can't be parsed, a value of **`false`** is returned to indicate that the parameter values were not available.

The **`Load`** event checks for the value returned from the **`check_parameters()`** method. If a value of **`false`** is returned, all graphic elements are filled with initial values and refreshed to show these values to the user.

Dealing with data entry

At this point, the user interface for the backup service checks for XML file existence and loads the parameter values if this file is present.

Now, it's time to control data entry to prevent storage of wrong values in the XML file that can cause the service malfunction. The following tasks must be performed:

- Avoid time values out of 00:00 to 23:59 range.
- Verify the existence of the source and destination path.

The value of **DayofWeek** is controlled by the **combobox** automatically, because the **SelectedIndex** property value will be between 0 and 7, depending on which day is selected by the user, including the *Every day* option.

Validating time values

For time values validation, you'll use the **TypeValidationCompleted** event of the **maskedTextbox** control. The following code sample shows how this is accomplished.

Code Sample 27

```
private void maskedTextBox1_TypeValidationCompleted(object sender, TypeValidationEven
tArgs e)
{
    if (!e.IsValidInput)
    {
        e.Cancel = true;
    }
}
```

The method checks for the value of the **IsValidInput** property that belongs to the **TypeValidationEventArgs** parameter passed. If this value is false, the **Cancel** property of the parameter is set to **true**. This avoids passing the focus to another control, including the Cancel button and the Close button, in the form.

Checking existence of source and destination paths

Source and destination paths must exist in the disk in order to ensure the service will work. The design of **mainform** has two textboxes in which these paths can be entered. To validate the existence of these folders, the **Validating** event for both paths will be used, as in the following code snippet.

Code Sample 28

```
private void textBox1_Validating(object sender, CancelEventArgs e)
```

```
{
    if (this.textBox1.Text.Trim().Length == 0)
    {
        e.Cancel = true;
    }
    else
    {
      if (!System.IO.Directory.Exists(this.textBox1.Text.Trim()))
       {
          MessageBox.Show("The Source Path entered doesn't exist.", "Backup Service I
nterface");
          e.Cancel = true;
       }
    }
}



private void textBox2_Validating(object sender, CancelEventArgs e)
{
    if (this.textBox2.Text.Trim().Length == 0)
    {
        e.Cancel = true;
    }
    else
    {
        if (!System.IO.Directory.Exists(this.textBox2.Text.Trim()))
        {
            MessageBox.Show("The Destination Path entered doesn't exist.", "Backup Se
rvice Interface");
            e.Cancel = true;
        }
    }

}
```

For both methods, if there's no input in the textbox, the method stores **`true`** in the **`Cancel`** property of the **`CancelEventArgs`** parameter. This prevents the textbox from losing the focus, and the cursor remains in it. Else, the method checks if the entry in the textbox corresponds to a valid path in the system. If the path doesn't exist, the method shows an error message dialog and sets the **`Cancel`** property of the **`CancelEventArgs`** parameter to the value of **`true`**, in order to make sure the cursor remains in the textbox.

Saving the parameters in the XML file

Once all the parameters values are entered, the last step is to store these values in the XML file that the service will use to work properly. The following task list needs to be completed to succeed:

- Create an **`XmlDocument`** object with the proper attributes to hold the parameters' values.
- Use the **`Save`** method of the **`XmlDocument`** to store the file.
- Tell the service that the parameters were changed, in order to make the service change its behavior.

Using the XmlDocument object

A method called **`Save_Parameters`** will be created to perform the XML file creation. This method will be called from the **`Click`** event of the **`Save`** button, as seen in the following code sample.

Code Sample 29

```
private void button1_Click(object sender, EventArgs e)
{
    this.Save_Parameters();
}

private void Save_Parameters()
{
    XmlDocument oparamsxml = new XmlDocument();

    XmlProcessingInstruction _xml_header = oparamsxml.CreateProcessingInstruction("xml", "version='1.0' encoding='UTF-8'");

    oparamsxml.InsertBefore(_xml_header, oparamsxml.ChildNodes.Item(0));

    XmlNode parameters = oparamsxml.CreateNode(XmlNodeType.Element, "Parameters", "");
    XmlNode backup = oparamsxml.CreateNode(XmlNodeType.Element, "Backup", "");

    XmlAttribute attribute = oparamsxml.CreateAttribute("source");
    attribute.Value = this.textBox1.Text.Trim();
    backup.Attributes.Append(attribute);

    attribute = oparamsxml.CreateAttribute("destination");
    attribute.Value = this.textBox2.Text.Trim();
    backup.Attributes.Append(attribute);

    attribute = oparamsxml.CreateAttribute("dayofweek");
    attribute.Value = this.comboBox1.SelectedIndex.ToString("00");
    backup.Attributes.Append(attribute);

    attribute = oparamsxml.CreateAttribute("hour");
    attribute.Value = this.maskedTextBox1.Text.Trim();
    backup.Attributes.Append(attribute);

    parameters.AppendChild(backup);
    oparamsxml.AppendChild(parameters);
```

```
    if (!Directory.Exists(this.HomeDir + "\\parameters"))
    {
        Directory.CreateDirectory(this.HomeDir + "\\parameters");
    }

    oparamsxml.Save(this.HomeDir + "\\parameters\\srvparams.xml");
}
```

First, the method creates an **XmlDocument** object and the **Parameters** root node. Then, the **Backup** child node is created along with all its attributes. Each attribute corresponds to a parameter needed for the service to work, and its proper value is taken from the graphics elements placed in the form for that purpose.

At the end, the **Save** method of the **XmlDocument** object stores the file in the disk.

Notifying the service that the parameters were changed

The action of saving the parameters in the disk means that the service behavior needs to change. To notify the service, the program needs to perform the following tasks:

- Check to see if the service is installed in the target system.
- Stop the execution of the service.
- Start the execution of the service in order to make it load the new parameters.

A method named **Notify_Changes** will be created to execute the previous tasks, and will be called from the **Click** event of the **Save** button, just after the calling of the **Save_Parameters** method discussed in the previous section. Now, the code will look like the following.

Code Sample 30

```
private void button1_Click(object sender, EventArgs e)
{
    this.Save_Parameters();
    this.Notify_Changes();
    this.Close();
}

private void Save_Parameters()
{
    XmlDocument oparamsxml = new XmlDocument();

    XmlProcessingInstruction _xml_header = oparamsxml.CreateProcessingInstruction("xm
l", "version='1.0' encoding='UTF-8'");

    oparamsxml.InsertBefore(_xml_header, oparamsxml.ChildNodes.Item(0));

    XmlNode parameters = oparamsxml.CreateNode(XmlNodeType.Element, "Parameters", "")
;
```

```
    XmlNode backup = oparamsxml.CreateNode(XmlNodeType.Element, "Backup", "");

    XmlAttribute attribute = oparamsxml.CreateAttribute("source");
    attribute.Value = this.textBox1.Text.Trim();
    backup.Attributes.Append(attribute);

    attribute = oparamsxml.CreateAttribute("destination");
    attribute.Value = this.textBox2.Text.Trim();
    backup.Attributes.Append(attribute);

    attribute = oparamsxml.CreateAttribute("dayofweek");
    attribute.Value = this.comboBox1.SelectedIndex.ToString("00");
    backup.Attributes.Append(attribute);

    attribute = oparamsxml.CreateAttribute("hour");
    attribute.Value = this.maskedTextBox1.Text.Trim();
    backup.Attributes.Append(attribute);

    parameters.AppendChild(backup);
    oparamsxml.AppendChild(parameters);

    if (!Directory.Exists(this.HomeDir + "\\parameters"))
    {
        Directory.CreateDirectory(this.HomeDir + "\\parameters");
    }

    oparamsxml.Save(this.HomeDir + "\\parameters\\srvparams.xml");
}

private void Notify_Changes()
{

    ServiceController controller = ServiceController.GetServices().FirstOrDefault(s =
> s.ServiceName == "MonitorService");

    if (controller!=null) //The service is installed
    {
       if (controller.Status == ServiceControllerStatus.Running) //The service is run
ning, so it needs to be stopped and started again to reload the parameters
       {
           controller.Stop(); //Stops the service
           controller.WaitForStatus(ServiceControllerStatus.Stopped); //Waits until t
he service is really stopped
           controller.Start(); //Starts the service and reload the parameters
       }
    }

}
```

The *ServiceController* class

A **`ServiceController`** component allows us to access and manage Windows Services running on a machine. The **`ServiceController`** class can be used to connect to and control the behavior of existing services. When an instance of the **`ServiceController`** class is created, its properties can be set to interact with a specific Windows service. The class can be used to start, stop, and otherwise manipulate the service.

After an instance of **`ServiceController`** is created, two properties must be set within it to identify the service with which it interacts: the computer name, and the name of the service you want to control.

Note: By default, MachineName is set to the local computer, so you don't need to change it unless you want to set the instance to point to another computer.

Adding a *System.ServiceProcess* Reference

A **`ServiceController`** represents a Windows Service and is defined in the **`System.ServiceProcess`** namespace. Before this namespace can be imported, you must add a reference to the **`System.ServiceProcess`** assembly.

To add a reference to an assembly, right-click on the project name in Visual Studio and select **Add Reference**, and then browse the assembly you need to add to your application.

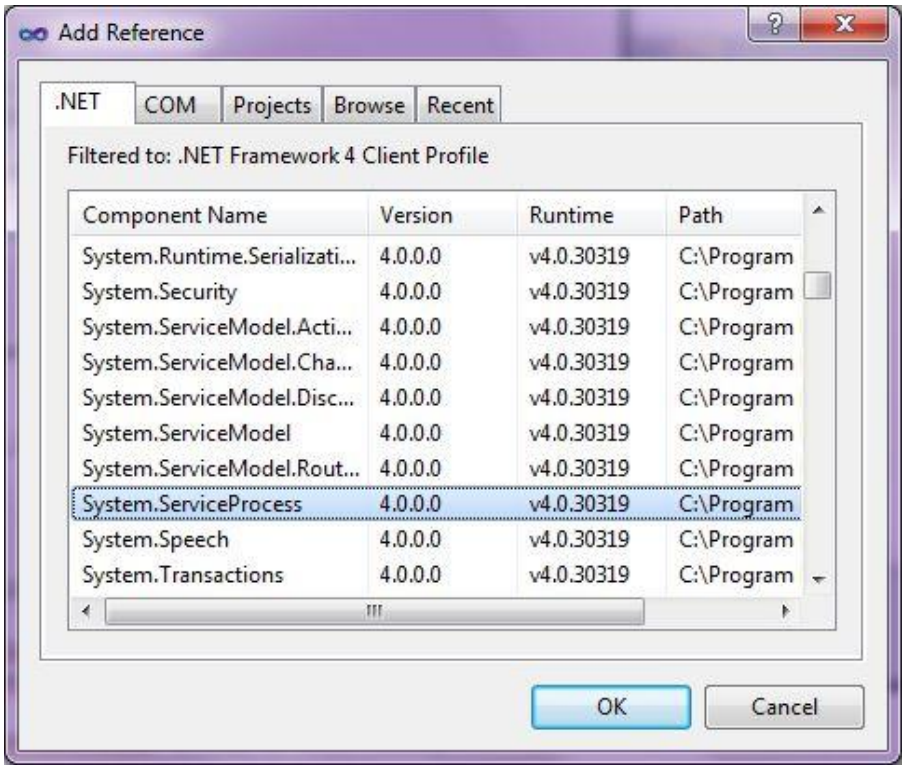

Figure 14: Add Reference dialog with the System.ServiceProcess assembly

What does *Notify_Changes* code do?

The **ServiceController.GetServices** static method returns the list of all services running on the computer. Along with this method, the **Enumerable.FirstOrDefault** associated method is used to scan the entire list, seeking a service named **MonitorService**. If the name is not found, **FirstOrDefault** returns a null value; otherwise, it returns an instance of a **ServiceController** object associated to the **MonitorService**.

If an instance associated to the **MonitorService** is returned, the program inquires for the value stored in the **Status** property of the instance. The possible values that can be stored in the property are:

- **ServiceControllerStatus.ContinuePending** – The service continue is pending. This corresponds to the Win32 **SERVICE_CONTINUE_PENDING** constant, which is defined as 0x00000005.
- **ServiceControllerStatus.Paused** – The service is paused. This corresponds to the Win32 **SERVICE_PAUSED** constant, which is defined as 0x00000007.
- **ServiceControllerStatus.PausePending** – The service pause is pending. This corresponds to the Win32 **SERVICE_PAUSE_PENDING** constant, which is defined as 0x00000006.
- **ServiceControllerStatus.Running** – The service is running. This corresponds to the Win32 **SERVICE_RUNNING** constant, which is defined as 0x00000004.
- **ServiceControllerStatus.StartPending** – The service is starting. This corresponds to the Win32 SERVICE_START_PENDING constant, which is defined as 0x00000002.
- **ServiceControllerStatus.Stopped** – The service is not running. This corresponds to the Win32 SERVICE_STOPPED constant, which is defined as 0x00000001.
- **ServiceControllerStatus.StopPending** – The service is stopping. This corresponds to the Win32 SERVICE_STOP_PENDING constant, which is defined as 0x00000003.

In this case, the method needs to perform actions only if the value of **Status** is **ServiceControllerStatus.Running**. This means that the service is currently running in the computer and needs to be stopped in order to reload the parameters. The **Stop** method is used to perform this action.

Ensuring that the service is really stopped

To start the service execution again, the program needs to be sure that it is really stopped. The time consumed by the service to stop depends on how many dependencies it has. The **WaitForStatus** method is used to delay program execution until the service reaches the **Stopped** status. Now, the program is sure that the service is not running.

Reloading service parameters

Since the parameters file is read every time the service starts its execution, once the service is stopped, the program executes the **Start** method of the **ServiceController** instance. This action causes the parameters to be reloaded from the XML file.

The *mainform.cs* entire code

Now, the **mainform.cs** code looks like the following sample.

Code Sample 31

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.ServiceProcess;
using System.Xml;

namespace monitorservicegui
{
    public partial class mainform : Form
    {
        string HomeDir = Path.GetDirectoryName(Application.ExecutablePath).Trim();
        public mainform()
        {
            InitializeComponent();
        }

        private void mainform_Load(object sender, EventArgs e)
        {
            if (!this.check_parameters())
            {
                this.comboBox1.SelectedIndex = 0;
                this.comboBox1.Refresh();
                this.maskedTextBox1.Text = "00:00";
                this.maskedTextBox1.Refresh();
                this.textBox1.Text = "";
                this.textBox1.Refresh();
                this.textBox2.Text = "";
                this.textBox2.Refresh();
            }
        }

        private Boolean check_parameters()
        {
            Boolean result = default(Boolean);

            if (!System.IO.Directory.Exists(this.HomeDir + "\\parameters"))
            {
                System.IO.Directory.CreateDirectory(this.HomeDir + "\\parameters");
                result = false;
            }
```

```
            else
            {
                if (System.IO.File.Exists(this.HomeDir + "\\parameters\\srvparams.xml
"))
                {
                    result = true;
                    XmlDocument parametersdoc = new XmlDocument();

                    try
                    {
                        parametersdoc.Load(this.HomeDir + "\\parameters\\srvparams.xm
l");
                    }
                    catch
                    {
                        result = false;

                    }

                    if (result)
                    {
                        XmlNode BackupParameters = parametersdoc.ChildNodes.Item(1).C
hildNodes.Item(0);
                        this.textBox1.Text = BackupParameters.Attributes.GetNamedItem
("source").Value.Trim();
                        this.textBox1.Refresh();
                        this.textBox2.Text = BackupParameters.Attributes.GetNamedItem
("destination").Value.Trim();
                        this.textBox2.Refresh();
                        this.comboBox1.SelectedIndex = Convert.ToInt32(BackupParamete
rs.Attributes.GetNamedItem("dayofweek").Value.Trim());
                        this.comboBox1.Refresh();
                        this.maskedTextBox1.Text = BackupParameters.Attributes.GetNam
edItem("hour").Value.Trim();
                        this.maskedTextBox1.Refresh();
                    }

                    parametersdoc = null;
                }
                else
                {
                    result = false;
                }
            }

            return (result);
        }

        private void button2_Click(object sender, EventArgs e)
        {
            this.Close();
        }

        private void maskedTextBox1_TypeValidationCompleted(object sender, TypeValida
```

```
tionEventArgs e)
        {
            if (!e.IsValidInput)
            {
                e.Cancel = true;
            }
        }

        private void textBox1_Validating(object sender, CancelEventArgs e)
        {
            if (this.textBox1.Text.Trim().Length == 0)
            {
                e.Cancel = true;
            }
            else
            {
              if (!System.IO.Directory.Exists(this.textBox1.Text.Trim()))
               {
                  MessageBox.Show("The Source Path entered doesn't exist.", "Backup S
ervice Interface");
                  e.Cancel = true;
               }
            }
        }

        private void textBox2_Validating(object sender, CancelEventArgs e)
        {
            if (this.textBox2.Text.Trim().Length == 0)
            {
                e.Cancel = true;
            }
            else
            {
                if (!System.IO.Directory.Exists(this.textBox2.Text.Trim()))
                {
                    MessageBox.Show("The Destination Path entered doesn't exist.", "B
ackup Service Interface");
                    e.Cancel = true;
                }
            }

        }

        private void button1_Click(object sender, EventArgs e)
        {
            this.Save_Parameters();
            this.Notify_Changes();
            this.Close();
        }

        private void Save_Parameters()
        {
            XmlDocument oparamsxml = new XmlDocument();
```

```
            XmlProcessingInstruction _xml_header = oparamsxml.CreateProcessingInstruc
tion("xml", "version='1.0' encoding='UTF-8'");

            oparamsxml.InsertBefore(_xml_header, oparamsxml.ChildNodes.Item(0));

            XmlNode parameters = oparamsxml.CreateNode(XmlNodeType.Element, "Paramete
rs", "");
            XmlNode backup = oparamsxml.CreateNode(XmlNodeType.Element, "Backup", "")
;


            XmlAttribute attribute = oparamsxml.CreateAttribute("source");
            attribute.Value = this.textBox1.Text.Trim();
            backup.Attributes.Append(attribute);

            attribute = oparamsxml.CreateAttribute("destination");
            attribute.Value = this.textBox2.Text.Trim();
            backup.Attributes.Append(attribute);

            attribute = oparamsxml.CreateAttribute("dayofweek");
            attribute.Value = this.comboBox1.SelectedIndex.ToString("00");
            backup.Attributes.Append(attribute);

            attribute = oparamsxml.CreateAttribute("hour");
            attribute.Value = this.maskedTextBox1.Text.Trim();
            backup.Attributes.Append(attribute);

            parameters.AppendChild(backup);
            oparamsxml.AppendChild(parameters);

            if (!Directory.Exists(this.HomeDir + "\\parameters"))
            {
                Directory.CreateDirectory(this.HomeDir + "\\parameters");
            }

            oparamsxml.Save(this.HomeDir + "\\parameters\\srvparams.xml");
        }

        private void Notify_Changes()
        {

            ServiceController controller = ServiceController.GetServices().FirstOrDef
ault(s => s.ServiceName == "MonitorService");

            if (controller!=null) //The service is installed
            {
               if (controller.Status == ServiceControllerStatus.Running) //The servic
e is running, so it needs to be stopped and started again to reload the parameters
               {
                   controller.Stop(); //Stops the service
                   controller.WaitForStatus(ServiceControllerStatus.Stopped); //Waits
 until the service is really stopped
                   controller.Start(); //Starts the service and reload the parameters
               }
```

```
            }

        }

    }
}
```

Now the project is ready to build the executable file. For a context where people are just reading and not necessarily following along in a code editor, the following screenshots show the finished program running.

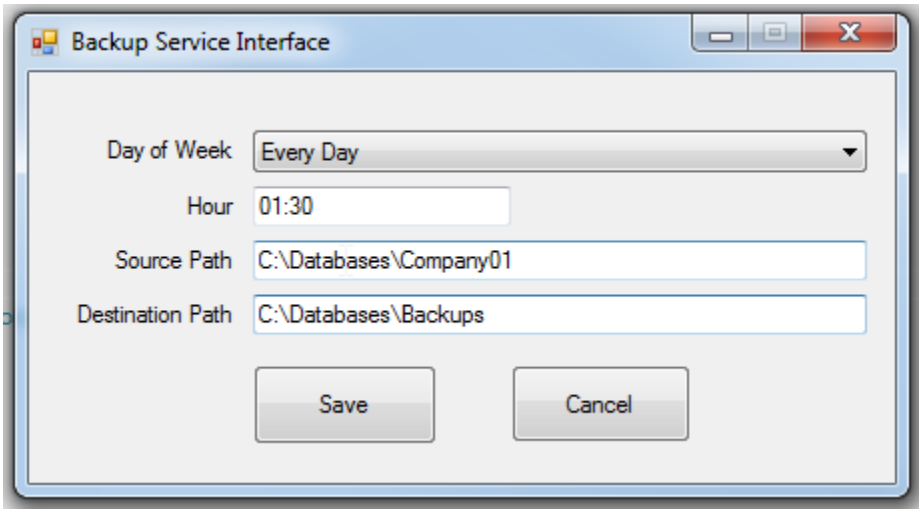

Figure 15: Backup Service Interface mainform filled with data

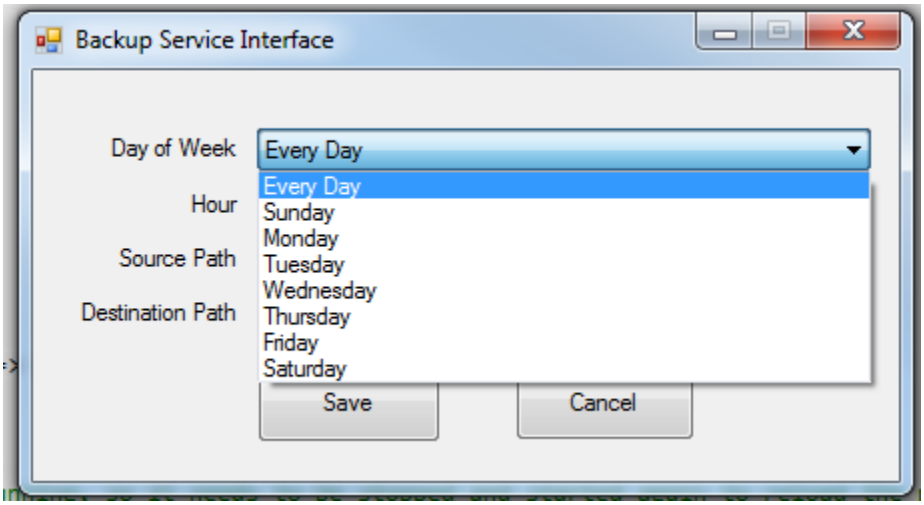

Figure 16: Backup Service Interface mainform showing the list with all week and weekend days

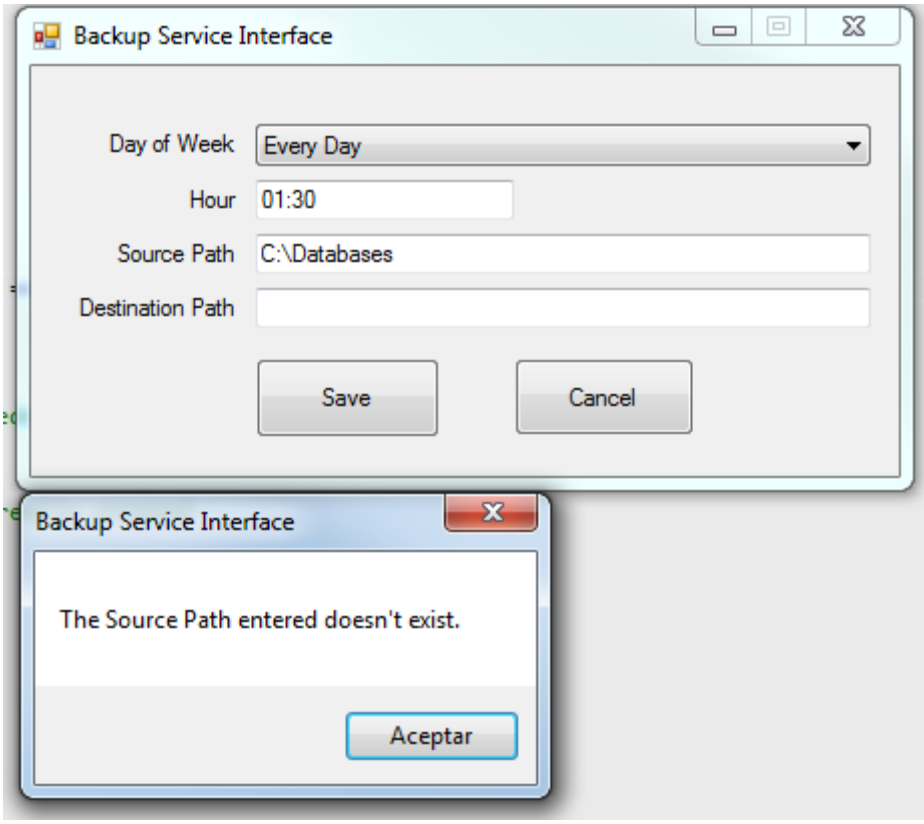

Figure 17: Backup Service Interface with a wrong Source Path entered

Deploying the user interface executable

At the end, the user interface project consists of only one executable file. It can be deployed along with the service executable distribution package, and must be copied in the same folder in which the service executable will be installed. The distribution package must include the following files:

- Installutil.exe (shipped with Visual Studio)
- Monitorservice.exe (the service executable file)
- Monitorservicegui.exe (the service user interface executable file)
- Ionic.Zip.dll (the library used for ZIP creation)
- The XML file with service execution parameters
- BAT installation file
- BAT uninstall file

Chapter summary

Since a Windows Service has no interface, a configuration file is the proper way to control its behavior. A text editor can be used to create or edit this file, but it's more professional to provide a program with a user interface to handle the configuration file. Besides, this is an easier way to accomplish this task.

This chapter explained how to create a Windows Forms program, which will provide the user interface to the **`MonitorService`** discussed here. The program will have only one form, in which all the parameter values needed will be entered.

The program will write the parameter values in a XML file, which will be used by the service executable program. The program will be able to read the parameters values if the XML file previously exists in the computer.

Finally, the program will communicate with the service in order to notify it when the parameter values change. The **`ServiceController`** class will be used to accomplish this task. In order to use the **`ServiceControlle`***r* class, a reference to the **`System.ServiceProcess`** assembly needs to be added in the project.

The **`ServiceController`** class allows a program to communicate with and control any service installed in the computer. Using this class, a service can be stopped, paused, or started. The program can inquire for the existence of a specific service and if this service is currently running or stopped.

Once the executable file is created, it must be deployed with the service distribution package and copied into the folder in which the service will run in the target computer.

General Summary

A Windows Service is an executable program that runs in its own Windows sessions. These services don't show any user interface, and can run even if no user is logged on the computer on which they are running.

Windows Services can be managed by administrators using the Services snap-in located in the Administrative Tools section of Control Panel.

Since Windows Services have no user interface, the Windows Event Log is the suggested way to communicate with administrative users. The Windows Event Log is a record of a computer's alerts and notifications that can be classified in warning, information, error, security success audit, or security failure audit types. The entries written in the Windows Event Log can be viewed in the Windows Event Viewer.

Windows Services can be created using Visual Studio with the Windows Service project type, which creates the service code baseline automatically. The **`ServiceBase`** .NET base class is used to create a derived custom class in order to develop the service. Then, the code baseline can be customized to fit the specific project needs.

The **`OnStart`** and **`OnStop`** events are used to trap the moment when the service stars its execution, and when the service execution stops, respectively.

The use of a **`Timer`** class instance is suggested if a continuous event monitoring is desired along service execution.

The service executable file can be deployed using the .NET InstallUtil tool. The InstallUtil tool is a command-line utility that allows you to install and uninstall server resources, and is shipped with Visual Studio. BAT files for installation and un-installation should be created, and the InstallUtil tool must be shipped in the installation package with BAT files and service executable file. I suggest the ZIP file format for the distribution package, in order to make delivery an easy step.

If a Windows Service requires parameters to work, the easiest and most common way to provide them is using a text file. Different formats can be used, but the most reliable is XML. A text editor program is commonly used to create or edit the file. However, a more professional way to perform this task is to provide a program with user interface.

A Windows Forms program can be developed to manage the service parameters. This program must allow you to enter the parameter values and then save them in a file. The program must use the **`ServiceController`** class to communicate the changes made to the service.

Finally, the user interface program made to manage the service parameters must be delivered with the service deploy package, and copied into the same folder where the service executable file is stored.

Conclusion

Windows Services have been present since Windows NT release, back in 1993. Formerly, they were known as NT Services, and have been improved along all these years.

There are many practical uses for a service. One main practical use is the interaction between UI and service programs, which, in this case, is the difference between a client and a server. A server receives requests, processes the requests, and usually sends back a reply. Thinking about Microsoft SQL Server or IIS, a client usually calls a service by sending requests to the server and then displaying or processing the reply. Examples of this are a data entry application and a web application. The request processing application is always installed as a service in Windows, and the client is usually just a normal application with a user interface.

There are more uses of a service. In fact, Windows functionality relies on many of them—Printer Spooler, Audio Manager, DNS Client, and DHCP Client are some examples of that fact.

Anything that can be done without relying on a person to start an application and click a button is a good candidate for a service. This could be useful to improve software products adding value either to customers or to the organization internally.