Internetanwendungstechnik

Microsoft .NET

Gero Mühl
Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Kommunikations- und Betriebssysteme (KBS)
Einsteinufer 17, Sekr. EN6, 10587 Berlin
Why Microsoft .NET?

- Component-based software development still is complex
  - Different programming paradigms
  - Different programming languages
  - Different runtime environments
  - A variety of end devices and interfaces

- In addition there is the Internet
  - Web Clients
  - Web Services
.NET is …

> ... a programming language and platform independent framework for building distributed applications
  > Classical Windows applications
  > Web-applications
  > Web Services

> … a marketing strategy of Microsoft

> … a set of services and tools (e.g. Passport)

> Main Objective: Simplify development
.NET vs. JAVA

> Java Slogan:
Write once (in Java) – run everywhere (on JRE)

> .NET Slogan:
Write in any language – run under .NET (on Windows?)
.NET Overview

VB  C++  C#  JScript  ...

Common Language Specification

ASP.NET  Windows Forms

Data and XML

Base Class Library

Common Language Runtime

Windows  COM+ Services

Visual Studio.NET
Common Language Runtime (CLR)

Source code
- VB
- C#
- C++

Managed code
- Assembly IL Code
- JIT Compiler
- Native Code

Unmanaged Component

Operating System Services
Compile and Execute

Compilation

Source Code → Language Compiler → Code

Metadata

Execution

Native Code → JIT Compiler → Code

Before a method is called or optionally before deployment
Common Language Runtime (CLR)

- Runtime environment for .NET assemblies (managed code)
- Programming language independent
- Responsible for
  - Type-safety
  - Garbage collection
  - Exception handling
  - Sandboxing
  - Versioning
  - …
- Major components: Common Type System, Memory Managers, JIT Compiler, etc.
.NET Programming Languages

- Visual Basic
- C++
- C#
- Python
- COBOL
- Eiffel
- Haskell
- ML
- JScript
- Ada
- APL
- Pascal
- Perl
- SmallTalk
- Oberon
- Scheme
- Mercury
- Objective Caml
- Oz
Common Language Infrastructure (CLI)

- Subset of the CLR
  - ECMA standard (335)

- Standardized parts
  - Semantics for metadata
  - Microsoft Intermediate Language (MSIL)
  - Parts of the .NET base class library (except ADO.NET, ASP.NET)

- C#
  - ECMA standard (334)
  - ISO/IEC standard (23270)
Common Type System (CTS)

- A single type system shared by compilers, tools and the CLR
- Supports the types and operations found in many programming languages
- Enables interoperability between different programming languages through uniform type system
- A string in C# and VB.NET are identical!
Common Type System (CTS)

> Two different types
  > Value types (stack allocation)
    > Byte, Char, Int, Boolean, Structure, Single, Double, Enum
  > Reference types (heap allocation)
    > Class, Interface, Array, String, Delegate

> Single inheritance for objects

> Multiple inheritance for interfaces

> A VB.NET class can inherit from a C# class

> Software written for the CLR is referred to as managed code
Common Language Specification (CLS)

- Subset of the CTS
- A set of specifications that language and library designers need to follow to be compliant with the .NET Framework
- Contract between language designers and the .NET Framework to ensure interoperability
MS Intermediate Language (MSIL)

> Common target language for all .NET compilers → programming language independence

> High-level, strongly-typed, stack-based assembler language

> Supports CTS

> Integrated support for complex data types and object as well as inheritance and polymorphism

> MSIL code is never interpreted

> JIT Compiler translates on demand MSIL code into native code for actual target platform → platform independence
MSIL Tools

- MSIL Assembler (`ilasm.exe`)
  - Generates executable from MSIL

- MSIL Disassembler (`ildasm.exe`)
  - Creates MSIL code from executable

- Native Image Generator Tool (`ngen.exe`)
  - Compiles MSIL to machine code in lieu of JIT compiler
Assemblies

> .NET applications consist of assemblies which are a logical unit of functionality

> .NET implementation of the component concept

> All .NET compilers generate assemblies

> An assembly contains all necessary information in one .exe or .dll file

> Security and versioning checks by the CLR are based solely on assemblies

> Private vs. shared assemblies
Assemblies

- System Header (6 Bytes)
  - just calls _CorExeMain() or _CorDllMain()

- Manifest
  - Assembly name and version number
  - List of all assembly modules (files)
  - List of external (referenced) assemblies
  - Security and versioning information
  - Exported types
  - …

- Metadata
  - Information about types, interfaces, method signatures, fields, properties, events …
Base Class Library (BCL)

> Common for all languages

> Accessible from all languages

> Powerful API for
  > Collections
  > Threading
  > File I/O
  > Reflection
  > Serialization
  > Security
  > Graphical User Interface (GUI)
  > XML/SOAP
  > …
Interoperability Example

C#:
```csharp
using System;
public class App {
    public static void Main() {
        Console.WriteLine("Hi");
    }
}
```

Oberon, Zonnon:
```onton
IMPORT System;
MODULE App;
BEGIN
    System.Console.WriteLine('Hi');
END App.
```

Base Class Library:
```onton
namespace System
{
    ...
    class Console
    {
        ...
        public static WriteLine(...);
    }
    ...
}
```
.NET Remoting

> .NET Remoting = middleware part of .NET

> Features
  > very modular, customizable
  > inherently component-based
  > many interceptor points
  > support for meta-data and reflection
  > contexts for application objects

> Comparable to CORBA, Java RMI, DCOM etc.
.NET Remoting Architecture

- **Messages**: What?
- **Formatter**: How?
- **Channels**: Whereto?
- **Proxy**: Generates messages from client’s method calls
- **Dispatcher**: Generates method invocations out of messages
.NET Remoting Message Path

Client
- Proxy
- Sink Chain
- Formatter
- Sink Chain

Transport Channel

messages

serialized messages

Server Object
- Stackbuilder
- Sink Chain
- Formatter
- Sink Chain
Message

> Messages are objects
  > implement `IMessage` interface
  > simple value tables {Key, Value}

> .NET message types
  > Constructor calls
  > Method calls
  > Pre-defined types have pre-defined values in value table

> Calls
  > Synchronous: request / response
  > Asynchronous: delayed response or no response
Channel

- Channels carry messages

- TCP channel
  - for LAN communications
  - permanent socket connection

- HTTP channel
  - for Internet communications
  - no permanent connection necessary

- Custom channels
  - IPX, Pipes, QoS enabled (TU Berlin) …

- Channels may implement customized sinks, e.g.
  - for monitoring and logging
  - for extended security checking
  - for message compression
Formatter

- Formatter serialize .NET Objects into a specific wire format
  - .NET formatters: SOAP and binary formatter
  - other formatters: IIOP, RMI, ORPC, …
- dynamically invoked by channels
- selection of channel and formatter depends on message context
Kinds of Remote Objects

> Server-activated objects (SAO)
  > Singleton (well-known object)
    > only one object instance for all clients
    > object created at server start
  > Single-call
    > each invocation generates a new object which is afterwards deleted
    > prevents unwanted resource consumption
  > Published (like Singleton but created manually)

> Client-activated objects (CAO)
  > each client has its own object; this might not scale!
## RMI vs. .NET

<table>
<thead>
<tr>
<th>Java RMI</th>
<th>.NET Remoting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object and proxy must have common interface</td>
<td>transparent proxy, type conversion on demand</td>
</tr>
<tr>
<td><strong>RemoteException</strong> need to be declared and caught</td>
<td>not needed</td>
</tr>
<tr>
<td>Explicit registration of object at <em>rmiregistry</em></td>
<td>not needed</td>
</tr>
<tr>
<td>not supported</td>
<td>remote constructor</td>
</tr>
<tr>
<td></td>
<td><strong>... and many more...</strong></td>
</tr>
</tbody>
</table>
Example: Remote Object Implementation

```csharp
public class HelloWorld : MarshalByRefObject
{
    public HelloWorld() {} 

    public string SayHello() 
    {
        return "Hello World!";
    }
}
```
Example: Client Mainline

```csharp
public class Client
{
    public static void Main(string args[])
    {
        RemotingConfiguration.Configure("client.xml");

        HelloWorld hello = new HelloWorld();

        Console.WriteLine(hello.SayHello());
    }
}
```
Example: Server Mainline

```csharp
public class Server
{

    public static void Main(string args[])
    {

        RemotingConfiguration.Configure("server.xml");

        Console.ReadLine();

    }

}
```
Example: Client Configuration

```xml
<configuration>
  <system.runtime.remoting>
    <application>
      <client url="tcp://localhost:4711/HelloWorld">
        <activated type="HelloWorld, server"/>
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```
Example: Server Configuration

<configuration>
  <system.runtime.remoting>
    <application>
      <channel>
        <channel ref="tcp" port="4711"/>
      </channel>
      <service>
        <activated type="HelloWorld, server"/>
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
How it works

> Client side proxy created automatically
  > No IDL
  > No IDL-compiler, …
  > Uses reflection and reflection emit at runtime

> Generic server side proxy
  > Stackbuilder dispatches method invocations
  > Uses reflection

> Almost everything configured using XML-Files

> But: may need interface definition
  > Don’t want to ship implementation with client
Microsoft .NET

ASP.NET
ASP.Net Overview

> ASP.NET is a server-side technology

> ASP.NET provides services for creation (i.e. debugging), deployment and execution of

  > Web applications
  > Web services

> Web applications are built using Web Forms

  > Creation should be as easy as building Visual Basic applications
Key Features of ASP.NET

- Web Forms
- Web Services
- Built on .NET Framework
- Simple programming model
- Complete object model
- Maintains page state
- Multibrowser support
- XCOPY deployment
- XML configuration
- Session management
- Caching
- Debugging
- Extensibility
- Separation of code and UI
- Security
- ASPX, ASP side by side
- Simplified form validation
- Cookieless sessions
Architecture

> ASP.NET is built upon
  > .NET Framework
  > Microsoft Internet Information Server (IIS)

> MONO builds a Apache binding
  > ASP.NET 1.0 & 1.1 are completely supported
  > ASP.NET 2.0 is supported (except WebParts)
  > Currently working on ASP.NET AJAX
Example: HelloWorld.aspx

```html
<html>
  <%@ Page language="c#" %>
  <head></head>

  <script runat="server">
    public void B_Click(object sender, System.EventArgs e)
    {
      Label1.Text = "Hello, the time is " + DateTime.Now;
    }
  </script>

  <body>
    <form method="post" runat="server">
      <asp:Button onclick="B_Click" Text="Push Me" runat="server" />
    </form>
  </body>
</html>
```
Programing Model

- Server-side programming model
- Based on controls and events
  - Just like Visual Basic
  - Not “data in, HTML out”
- Higher level of abstraction than ASP
- Requires less code
- More modular, readable, and maintainable
Controls and Events

Browser

ASP.NET

Event handlers

Button
List
Text

Button code ...
List code ...
Text code ...

G. Mühl
ASP.NET Object Model

> Code executes on the web server in page or control event handlers

> Controls are objects, available in server-side code
  > Derived from `System.Web.UI.Control`
  > Similar to `System.Windows.Forms.Control`

> The web page is an object, too
  > Derived from `System.Web.UI.Page` which is a descendant of `System.Web.UI.Control`
  > A page can have methods, properties, etc.
Postback

> **Page** object generates an HTML form

> Upon user action (e.g. button pressed) the event is posted back to the server
>   > Not all possible events are posted back
>   > For example, mouse move or key press are not posted back because of performance issues

> In ASP and other server-side technologies the state of the page is lost upon postback...
>   > Unless you explicitly write code to maintain state
>   > This is tedious, bulky and error-prone
Postback

> By default, ASP.NET maintains the state of all server-side controls during a postback
  > Can use `method="post"` or `method="get"`

> Server-side control objects are automatically populated during postback
  > No state stored on server
  > Works with all browsers
Postback

1. HTML page including state encoded in <form> tag
2. Postback of <form> data
3. Load post data
4. Raise events

Browser

ASP.NET

Event handlers
Server Side Controls

> Multiple sources to obtain controls
  > Built-in
  > 3\textsuperscript{rd} party
  > User-defined

> Controls range in complexity and power
  > Button
  > Text
  > Drop down
  > Calendar
  > Data grid
  > …

> Can be populated via data binding
Automatic Browser Compatibility

- Controls can provide automatic browser compatibility

- Can target uplevel or downlevel browsers
  - Uplevel browsers support additional functionality, such as JavaScript and DHTML
  - Downlevel browsers support HTML 3.2
Automatic Browser Compatibility

- Mozilla
  - Button
  - Menu
  - Text

- Netscape
  - Button
  - Menu
  - Text

- IE 5.5
  - Button
  - Menu
  - Text

- IE 6
  - Button
  - Menu
  - Text

Button Control
- Button code
  - ...

Menu Control
- Menu code
  - ...

Text Control
- Text code
  - ...

ASP.NET
Event handlers
Code-behind Pages

> Two styles of creating ASP.NET pages
  > UI and code in .aspx file
  > UI in .aspx file, code in code-behind page
    > Supported in Visual Studio.NET

> Code-behind pages allow you to separate the user interface design from the code
  > Allows programmers and designers to work independently
  > Direktive

    <%@ Codebehind="WebForm1.cs"
      Inherits="WebApplication1.WebForm1" %>

Automatic Compilation

> Just edit the code and hit the page (edit, save, and run)

> ASP.NET will automatically compile the code into an assembly

> Compiled code is cached in the CLR Assembly Cache

> Subsequent page hits use compiled assembly

> If the text of the page changes then the code is recompiled
Automatic Compilation

Internet Explorer → Parser → Compiler → Assembly Cache

Client

Output Cache

Server

Memory

Assembly IL

Execute

HTTP Runtime

hello.aspx
Page Syntax

> The most basic page is just static text
> Any HTML page can be renamed .aspx

> Pages may contain:
> Directives: `<%@ Page Language="C#" %>`
> Server controls: `<asp:Button runat="server">`
> Code blocks: `<script runat="server">..</script>`
> Server side comments: `<%- - - %>`
> Render code: `<%=`
> Use is discouraged; use `<script runat="server">` with code in event handlers instead
Page Directive

> `<%@ Page Language="C#" ... %>`

> Lets you specify page-specific attributes, e.g.
>  > AspCompat: Compatibility with ASP
>  > Buffer: Controls page output buffering
>  > CodePage: Code page for this .aspx page
>  > ContentType: MIME type of the response
>  > ErrorPage: URL if unhandled error occurs
>  > Inherits: Base class of Page object
>  > Language: Programming language
>  > Trace: Enables tracing for this page
>  > Transaction: COM+ transaction setting

> Only one page directive per .aspx file
Server Code Blocks

> Server code lives in a script block marked `runat="server"`

```c#<script language="C#" runat="server"></script>
<script language="VB" runat="server"></script>
<script language="Jscript" runat="server"></script>
```

> Script blocks can contain
  > Variables, methods, event handlers, properties
  > They become members of a custom `Page` object
Page Import Directive

> Adds code namespace reference to page
> Avoids having to fully qualify .NET types and class names
> Equivalent to the C# using directive

```csharp
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Net" %>
<%@ Import Namespace="System.IO" %>
```
Page Class

> The Page object is always available when handling server-side events

> Provides a large set of useful properties and methods, including:

> Application, Cache, Controls, EnableViewState, ErrorPage, IsPostBack, IsValid, Request, Response, Server, Session, Trace, User, Validators

> DataBind(), LoadControl(), MapPath(), Validate()
Server Control Syntax

> Controls are declared as HTML tags with `runat="server"` attribute

```html
<input type="text" id="text2" runat="server" />
<asp:calendar id="myCal" runat="server" />
```

> Tag identifies which type of control to create
>   Control is implemented as an ASP.NET class

> The `id` attribute provides programmatic identifier
>   It names the instance available during postback
>   Just like Dynamic HTML
Server Control Properties

> Tag attributes map to control properties

\[
\text{<asp:button id="c1" Text="Foo" runat="server">}
\]
\[
\text{<asp:ListBox id="c2" Rows="5" runat="server">}
\]

> Tags and attributes are case-insensitive

> Control properties can be set programmatically

\[
c1. \text{Text} = "Foo";
\]
\[
c2. \text{Rows} = 5;
\]
Maintaining State

> Controls maintain their state across multiple postback requests by default
  > Implemented using a hidden HTML field: __VIEWSTATE
  > Works for controls with input data (e.g. TextBox, CheckBox), non-input controls (e.g. Label, DataGrid), and hybrids (e.g. DropDownList, ListBox)

> Can be disabled per control or entire page
  > Set EnableViewState="false"
  > Lets you minimize size of __VIEWSTATE
Events

> Controls reacts to events
>   Enables clean code organization
>   Avoids the “Monster IF” statement
>   Less complex than ASP pages

> Code can respond to page events
>   e.g. Page_Load, Page_Unload

> Code can respond to control events
>   Button1_Click
>   TextBox1_Changed
Event Lifecycle

1. Initialize
2. Restore Control State
3. Load Page

Control Events
1. Change Events
2. Action Events

Save Control State
Render
Unload Page

Page_Init
Page_Load
Textbox1_Changed
Button1_Click
Page_Unload
Page Loading

> Page_Load fires at beginning of request
> Controls are already initialized
> Input control values already populated

```csharp
protected void Page_Load(Object s, EventArgs e)
{
    message.Text = textbox1.Text;
}
```
Page Loading

> **Page_Load** fires on every request
> **Use** `Page.IsPostBack` to execute conditional logic
> **If** a Page/Control is maintaining state then only initialize it when `IsPostBack` is false

```csharp
protected void Page_Load(object s, EventArgs e)
{
    if (!Page.IsPostBack)
    {
        // Executes only on initial page load
        Message.Text = "initial value";
    }

    // Rest of procedure executes on every request
}
```
Server Control Events

> **Change Events**
> - By default, these execute only on next action event
> - E.g. `OnTextChanged`, `OnCheckedChanged`
> - Change events fire in random order

> **Action Events**
> - Cause an immediate postback to server
> - E.g. `OnClick`

> **Works with any browser**
> - No client script required, no applets, no ActiveX® Controls
How Postback works on Client Side

<form name="Form1" method="post" action="WebForm1.aspx" id="Form1">
<a href="javascript:__doPostBack('Firstcontrol1','dec')">Decrease Number</a>
</form>

<script language="javascript">
    <!--
    function __doPostBack(eventTarget, eventArgument) {
        var theform
        theform = document.Form1;
        theform.__EVENTTARGET.value = eventTarget;
        theform.__EVENTARGUMENT.value = eventArgument;
        theform.submit();
    }
    // -->
</script>
Wiring Up Control Events

> Control event handlers are identified by the tag

```html
<asp:button onclick="btn1_click" runat="server">
<asp:textbox onchanged="text1.changed" runat="server">
```

> Event handler code

```csharp
protected void btn1_click(Object s, EventArgs e)
{
    Message.Text = "Button1 clicked";
}
```
Event Arguments

> Events pass two arguments
  > The sender, declared as type `Object`
    > Usually the object representing the control that generated the event
    > Allows you to use the same event handler for multiple controls

> Arguments, declared as type `EventArgs`
  > Provides additional data specific to the event
  > `EventArgs` itself contains no data; a class derived from `EventArgs` will be passed
  > Read the fine manual to find out which subclass of `EventArgs` you have to expect
Change Events & AutoPostBack

> Usually only action events trigger a postback

> Some controls can send postbacks for change events
  > CheckBox
  > ListControl
  > TextBox

> Postback is sent after the user
  > 1. changed a value
  > 2. tabbed out of the control

> Example
  > Automatically sum up values entered in text boxes
Change Events & AutoPostBack

```csharp
<%@ Page Language="C#" AutoEventWireup="True" %>

<html>
<head>
    <script runat="server">
        protected void Page_Load(Object sender, EventArgs e)
        {
            int Answer = Convert.ToInt32(Value1.Text) +
                          Convert.ToInt32(Value2.Text);
            AnswerMessage.Text = Answer.ToString();
        }
    </script>
</head>
<body>
<form runat="server">
    <asp:TextBox ID="Value1" AutoPostBack="True" runat="server"/>
    <asp:TextBox ID="Value2" AutoPostBack="True" runat="server"/>
    <asp:Label ID="AnswerMessage" runat="server"/>
</form>
</body>
</html>
```
Page Unloading

- Page_Unload fires after the page is rendered
  - Don’t try to add to output

- Useful for logging and clean up

  ```csharp
  protected void Page_Unload(Object s, EventArgs e)
  {
      MyApp.LogPageComplete();
  }
  ```
Bibliography


Fragen?