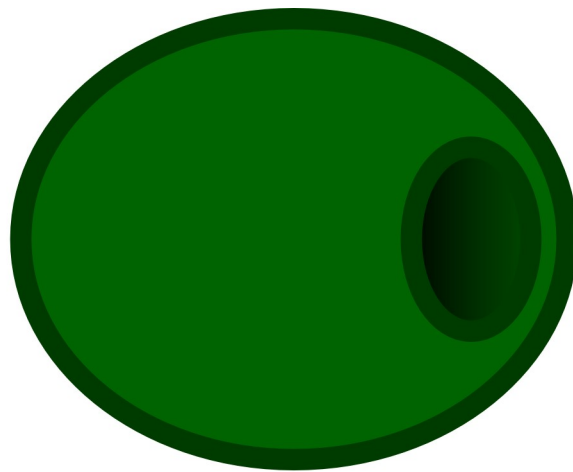


# **Mono Olive**

## **Introducing Windows Communication Foundation**

Marcos Cobeña Morían



**mono olive**

© 2007, Marcos Cobeña Morián

© This edition:

- 2007, Marcos Cobeña Morián

<http://www.youcannoteatbits.org/>

[contact@youcannoteatbits.org](mailto:contact@youcannoteatbits.org)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.





*To my Mom and Dad, María José and José Antonio, for always  
filling my cortex with beautiful patterns*



# Contents

<b>Foreword</b> .....	<b>11</b>
<b>Preface</b> .....	<b>13</b>
1. Motivation.....	13
1.1. Mono Project: Olive.....	13
1.2. Google Summer of Code.....	14
1.3. Goals.....	14
2. Structure of This Document.....	15
3. Acknowledgment.....	16
<b>Chapter 1. Service-oriented Architecture (SOA)</b> .....	<b>1</b>
1. Brief Definition.....	1
2. Service-oriented vs. Object-oriented.....	1
3. The Four Tenets.....	3
3.1. Boundaries Are Explicit.....	3
3.2. Services Are Autonomous.....	3
3.3. Services Share Schemas and Contracts, Not Classes and Types.....	4
3.4. Compatibility Is Policy-Based.....	5
4. Service-oriented Example.....	6
5. Summary.....	7
<b>Chapter 2. Getting Olive Running</b> .....	<b>9</b>
1. Build Olive from Sources.....	9
1.1. Get Olive Sources.....	10
1.2. Build 3.0 Assemblies.....	10
2. Hello World Service.....	11
2.1. Sever Side.....	11
2.2. Client Side.....	13
2.3. Build and Test.....	14
3. Summary.....	14
<b>Chapter 3. WCF Fundamentals</b> .....	<b>17</b>
1. Architectural Concepts.....	17
1.1. Endpoints.....	18

1.2. Messages.....	19
1.3. Channels.....	22
2. Example.....	23
3. Summary.....	25
<b>Chapter 4. Contracts.....</b>	<b>27</b>
1. Service Contracts.....	27
2. Data Contracts.....	29
2.1. When To Use Them and When Not.....	30
2.2. The Important Thing Is at The Shell.....	31
3. Message Contracts.....	32
4. Summary.....	33
<b>Chapter 5. Mono Olive in Depth.....</b>	<b>35</b>
1. Moonlight.....	35
2. How To Collaborate.....	36
2.1. Step 0: Things You Should Know Before.....	36
2.2. Step 1: Get in Touch.....	37
2.3. Step 2: Write Tests.....	37
2.4. Step 3: Ready, Steady, Go!.....	39
3. Summary.....	39
<b>Conclusion.....</b>	<b>41</b>
<b>Bibliography.....</b>	<b>43</b>
<b>Appendix A: Full Examples.....</b>	<b>45</b>





# Foreword

The Mono Project is a huge effort to develop and run .NET applications on Unix-like systems. It comprises, among other things, a C# 2.0 compiler, an almost full .NET 2.0 profile, and its own stack of libraries exposing existing APIs into a managed world. Codename Olive was born to bring new 3.0 and 3.5 profiles in to the Mono's scene.

During April 2007, I was selected from both Google and The Mono Project organization, with twenty-three more students around the world, to work inside the huge Mono community as part of Google Summer of Code program. My main goal was to implement a subset of Microsoft Windows Communication Foundation (WCF) for Olive.

This document was written thinking on those who want to get briefly introduced into both WCF and Mono Project worlds, as well as how to become an active contributor.

One of the background goals from Summer of Code program is to *help open source projects identify and bring in new developers and committers*, and I honestly feel I've tried to do my best to achieve this.

These writings are my way to say thanks.

Marcos Cobeña Morián  
Seville (Andalusia, Spain), September 2007



# Preface

We live in a distributed world. Nowadays, it's difficult to think of disconnected pieces of software. Even further, it's difficult to imagine us going into low-level programming for building distributed applications when high level components as Windows Communication Foundation (WCF) exists.

WCF is *Microsoft's unified framework for building secure, reliable, transacted, and interoperable distributed applications.*

## 1. Motivation

Since Microsoft publicly announced .NET Framework by 2001, a new effort from Open Source community came up with the main objective of improving development platform for Unix systems: the Mono Project.

### 1.1. Mono Project: Olive

As time goes by, the community grew up and Mono programmers decided to expand their libraries to support Microsoft ones, resulting in a powerful runtime for supporting cross-platform applications, making it easier to port Windows ones to Unix and vice versa.

Nowadays, Mono almost fully supports .NET 2.0 API, including ASP.NET 2.0, .NET Remoting and Windows Forms, among other ones. It also ships with a complete C# 2.0 compiler.

On November 2006, Microsoft announced the final release of their new 3.0 profile: a set of new libraries that run on top of an existing 2.0 runtime.

This runtime includes WCF. Some weeks earlier, on October 24, Atsushi Eno and Miguel de Icaza, during the Mono User and Developers Meeting (you can have a look to schedule and slides at <http://www.go-mono.com/meeting/schedule.aspx>), announced Olive, a new effort for implementing Windows CardSpace (WCS) through WCF.

From the end of September 2005, when Olive was started, more than 120,000 lines of code within close to 10,000 classes have made it a serious and reliable option to contribute to. However, it's still not mature enough nor stable for a final release.

## **1.2. Google Summer of Code**

Since 2005, Google has supported Open Source projects by offering summer scholarships for students who spend the whole summer programming for their favorite projects. You can have a look to past years results at <http://code.google.com/soc/>.

As part of 2007 program, the author of this notebook was selected to contribute towards implementing peer-to-peer (P2P) bits within Olive.

## **1.3. Goals**

Everything has an end, and so did this project. The project began with a specification and a set of goals with the project organization.

Implementing P2P communication was the main objective. Microsoft WCF ships, by default, with predefined classes for making P2P connections in an easy way, making it possible to build services which host a mesh of nodes (a node represents a potential client), taking care of everything that is needed to achieve this goal:

registration of new nodes, message flows over the mesh, and more.

The main goals schedule which was started to work with contained (extracted from <http://groups.google.com/group/mono-soc-2007/web/on-wcf-netpeertcpbinding-and-peerresolvers>):

- Hello, world, with Mono Olive module, SVN repository, tests and so on.
- CustomPeerResolverService must be almost finished (among tests).
- NetPeerTcpBinding finished, and working coupled with CustomPeerResolverService.
  - Chat application working.

## 2. Structure of This Document

This notebook will cover some of the fundamental aspects related to WCF, as well as how to help Mono effort to improve their work.

Following chapters are organized in this way:

- *Chapter 1. Service-oriented Architecture (SOA):* a guided visit within the wonderful world of distributed software.
- *Chapter 2. Getting Olive Running:* how to set up your environment to test this platform.
- *Chapter 3. WCF Fundamentals:* must-known keys for creating distributed solutions using WCF.
- *Chapter 4. Contracts:* how to expose operations, data and everything in between.
- *Chapter 5. Mono Olive in Depth:* a short guide for anyone interested in contributing to this project.
- *Conclusion*
- *Bibliography*
- *Appendix A: Full Examples*

### 3. Acknowledgment

I've followed the Mono Project since the very beginning, when the project didn't even exist. There were only a handful of entries at Miguel de Icaza's Blog (<http://tirania.org/blog/>) talking about a new C# compiler he was working on. Over the years, the project was completed. Nowadays, hundreds of people around the world contribute to the effort of porting .NET, and to increase its library stack.

It's been exciting to feel me part of this large mesh of people that contribute to the Mono effort. Just feeling someone located in somewhere around the world is using code that I've written makes one happy. And there's no money that can buy this.

There's so much people I'd like to thank... Without a defined order (just the one my cortex applies): thanks to my family and friends, those I keep sharing time with and also those I used to, for daily understanding I love computers since I'm child, and for keeping close after days and days of *I'm sorry, I can't go out, I have no time*, specially to my soul mate Luis Serrano Domínguez; thanks to my .NET Club mates, who have been crucial for me in both my personal and professional life, both changed when all this started, I love all of you; thanks to Microsoft whole support team, who I worked with as if a dream it was, specially to Alejandro Campos Magencio, David Salgado Bermejo, Ignacio Alonso Portillo, Javier Melero Gallego, Sacha Arozarena Valladares and Distributed Services Europe Core Team (*We're Service Oriented!*), where Dominique Pochat introduced me into COM, COM+ and DCOM debugging precious world, I'll never forget all of you.

Thanks to Rafael Corchuelo Gil, my tutor, for his unfordable help on this document, his superb daily conferences during software engineering class hours and his back-end job on students like me who love what we do, I wish there were more teachers like him; thanks to María José Hidalgo Doblado and José Antonio Alonso Jiménez, from Computer Science and Artificial Intelligence Department at University of Seville, for letting me visit their offices during non-working hours, they've always let me their door open; thanks to David Guerrero Martos and Paulino Ruiz de Clavijo, from Electronic Technology Department at University of Seville, for their help

since my first years at High School.

I'd like to thank those who have helped me checking this document: Miguel de Icaza and Teresa Matamoros Casas. They surprisingly took my writings and sent me back a handful of corrections. Thanks guys.

Finally, I'd like to thank Google for such a beautiful program as Summer of Code is, and Atsushi Enomoto, Miguel de Icaza and every Mono programmer for their support during this summer. Huge, huge, huge thanks.

It's been thrilling, I'm pretty sure that I'll never forget this summer.



# Chapter 1. Service-oriented Architecture (SOA)

It's highly probable you've already heard of this acronym. Within a world full of acronyms as software design is, SOA plays an important role.

As in many other scenarios, if you ask for the meaning of SOA you'll get, almost sure, different definitions. Within this chapter, you'll possible go through a new one, but who said it wasn't fine to have different points of view?

## 1. Brief Definition

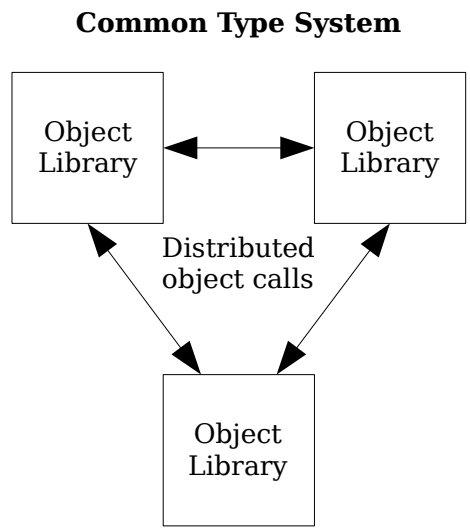
*A loosely-coupled architecture designed to meet the business needs of the organization.* Simple, isn't it? Well, it isn't at all, it'd be worth to discuss some points.

SOA isn't something new. If you have some background on Service Orientation (SO) you'll have heard of technologies like CORBA or DCOM, which share some conceptual keys with SOA. However, SOA isn't a component, nor a final product, nor an existing technology you can simply reuse. SOA is more about philosophy.

## 2. Service-oriented vs. Object-oriented

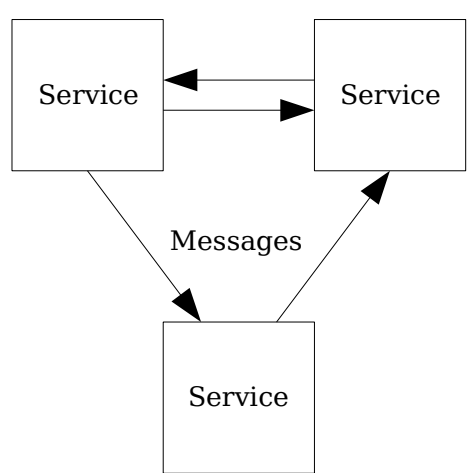
Do I need to change programming model in order to think about SO? No, you don't. Even more, SO relies on Object Orientation (OO), so it's likely you can reuse current knowledge.

Within the OO .NET approach, distributed calls are always made inside a sandbox, known as Common Type System (CTS), which increases dependency of each object library with the rest, resulting in a highly coupled scenario, as Figure 1-1 shows.



**Figure 1-1.** OO distributed approach

On the other side (not opposite), SO provides a more open solution, composed of autonomous and platform-independent services, which communicate in a loosely coupled way. Figure 1-2 represents this.



**Figure 1-2.** SO distributed approach

## 2 2. Service-oriented vs. Object-oriented

Above scenario is built also upon OO. Every service is coded following OO, so SO just provides a plus to make distributed solutions.

### **3. The Four Tenets**

As OO ones (inheritance, modularity, polymorphism and encapsulation), there are also some principles which define a SO scenario:

1. Boundaries are explicit.
2. Services are autonomous.
3. Services share schemas and contracts, not classes and types.
4. Compatibility is policy-based.

If your solution follows those four points, it's a SO one! Now, let's have a brief look to what all this means.

#### **3.1. Boundaries Are Explicit**

The way two or more services talk one to each other is based on messages. Messages are sent across boundaries, independently on however services are internally implemented, and these are formally specified. As long as you respect boundaries, there should be no problem.

It can be possible crossing boundaries increases communication time, memory consumed, etc., so it's very important to make just the necessary calls, as excessive consumption may degrade performance.

#### **3.2. Services Are Autonomous**

In a SO scenario, topology is expected to change over the time. As a distributed system, it's possible not every part is always available to be accessed. Tasks such maintenance, upgrading, etc., can make a specific service to be unavailable, but the rest must keep working, including those which directly depend on not working one.

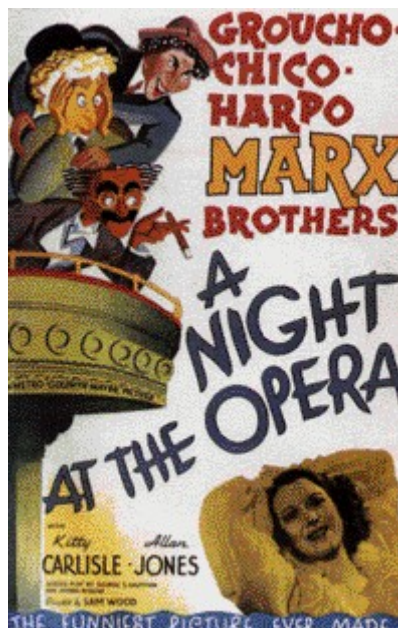
Even more, there's no oracle which controls which services are up

or not, emergence takes care of this. Each service is treated as an independent piece, and must work independently of the rest.

### 3.3. Services Share Schemas and Contracts, Not Classes and Types

As Figure 1-1 showed, inside an OO world every call is made within a CTS (for example, `Add(int a, int b)` must return a new 32 bits integer in .NET, which is a well-known type). It isn't the same on SO.

On the contrary, as there couldn't be just a single CTS involved, previous call to `Add` should be formally specified using a schema for data (remember, 32 bits integer returned) and a contract for its behavior (yes, easily expected for this example:  $a+b$ ). It's important to keep stable schemas and contracts over time.



**Figure 1-3.** A Night at the Opera Poster (from [http://en.wikipedia.org/wiki/A\\_Night\\_at\\_the\\_Opera\\_\(film\)](http://en.wikipedia.org/wiki/A_Night_at_the_Opera_(film)))

Do you remember The Marx Brothers' unforgettable contract skit in A Night at the Opera (Figure 1-3)?

**Groucho Marx:** *Now pay particular attention to this first clause, because it's most important. There's the party of the first part shall*

#### 4 3. The Four Tenets

*be known in this contract as the party of the first part. How do you like that, that's pretty neat eh?*

**Chico Marx:** *No, that's no good.*

**Groucho Marx:** *What's the matter with it?*

**Chico Marx:** *I don't know, let's hear it again.*

**Groucho Marx:** *So the party of the first part shall be known in this contract as the party of the first part.*

**Chico Marx:** *Well it sounds a little better this time.*

Well, Groucho remained contract stable over time (for now), but just Chico didn't hear it fine. He was likely paying more attention to something else.

### **3.4. Compatibility Is Policy-Based**

SO separates interaction services can have from its constraints. Policies make possible to change determined service's aspects without changing its behavior. In order to communicate with a specific service, you (well, not you, but your machine) previously reads its policies to check if there's a mutual match.

Following with previous conversation:

[...]

**Chico Marx:** *Oh sure. You bet. Hey wait, wait. What does this say here, this thing here?*

**Groucho Marx:** *Oh that? Oh that's the usual clause, that's in every contract. That just says, it says, 'If any of the parties participating in this contract are shown not to be in their right mind, the entire agreement is automatically nullified.'*

**Chico Marx:** *Well, I don't know.*

**Groucho Marx:** *It's all right, that's in every contract. That's what they call a sanity clause.*

**Chico Marx:** *You can't fool me, there ain't no sanity clause.*

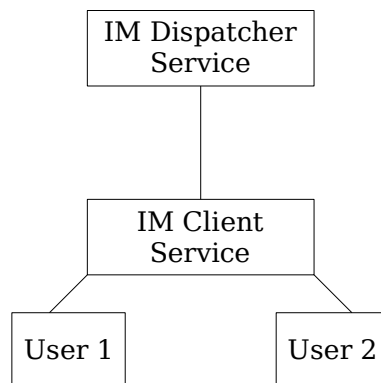
You can understand policies as Groucho understands what an *usu-*

*al clause* is.

## 4. Service-oriented Example

Almost sure you've ever used an Instant messaging (IM) service: ICQ, IRC, Jabber, Windows Live Messenger, Yahoo! Messenger, and so forth. If you have, did you ask your-self how does it work? Don't worry if the answer is no, you'll briefly see how now. If you did, remember additional information takes almost no space on your brain.

Probably not 100% nowadays available IM services follow the four SO tenets studied before, but it'd be possible to have one which does. Having the minimum structure needed, it must have at least a service side, and one or more clients which connect that service (more than just one would be fine, as talking alone isn't funny), as Figure 1-4 tries to show.



**Figure 1-4.** IM architectural diagram

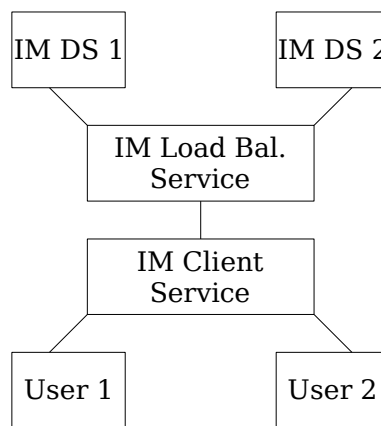
User 1 logs into an application which communicates with IM Client Service. This services checks if user exists establishing a new communication channel with IM Dispatcher Service. Same for User 2. If User 1 wants to send a message to 2 one, he writes down a text which IM Client Service will route through IM Dispatcher Service to User 2, if this last is the final one. Pretty simple.

After some months working, you notice your new IM service has registered thousand of new users. Maybe, there could be no prob-

### 6 4. Service-oriented Example

lem for IM Client Service, as it simply sends and receive single messages but, will IM Dispatcher Service support this new high workload? Imagine this scenario: one thousand users sending messages at the same time, IM Dispatcher Service will have to check which destination each message has, check if the user is on-line and, if he is, to route the message to his client. It seems like Figure 1-4 approach will need some tuning, it just doesn't scale.

A known solution for high workload could be to add a Load Balancer (LB) as a proxy between IM Client and Dispatcher services. This way, LB will take care of how many connections are made to each Dispatcher Service (DS) routing the message to the less loaded one. Figure 1-5 shows this new approach.



**Figure 1-5.** IM architectural diagram plus high workload support

As you've seen, scenario has changed over time, being necessary to deploy new services, modify existing ones, enlarge topology, etc.

## 5. Summary

Within this chapter you've been briefly introduced into SO world. You've learned a, surely, new definition for SOA, and which four principles it defends, known as the four tenets:

1. Boundaries are explicit.
2. Services are autonomous.
3. Services share schemas and contracts, not classes and types.

4. Compatibility is policy-based.

Don't forget them.

# Chapter 2. Getting Olive Running

Right now, Mono packages don't ship with new 3.0 assemblies. By default, installing Mono from official distribution won't let you work with WCF, as it's still considered as experimental software, and has no support. But, wait, not everything is lost.

As you already know, Mono programmers maintain source code on-line using Subversion (SVN), a version control system. This one, lets anonymous users check-out last changes and build the entire framework by their-selves. You can access it through the browser at <http://anonsvn.mono-project.com/> and, for example, have a look at current System.ServiceModel namespace implementation at <http://anonsvn.mono-project.com/source/trunk/olive/class/System.ServiceModel/>.

## 1. Build Olive from Sources

In order to build olive module, you previously must have built, at least, mono and mcs ones. Depending on your OS, you'll have to follow different steps for getting this done. You can get more information on build steps at [http://www.mono-project.com/Compiling\\_Mono](http://www.mono-project.com/Compiling_Mono). If your OS matches Windows, you can go for two different approaches listed at [http://www.mono-project.com/Compiling\\_Mono](http://www.mono-project.com/Compiling_Mono).

It's assumed you've successfully built mono and mcs, and you're just waiting to go for olive one.

## 1.1. Get Olive Sources

Independently on your OS, you'll have to check-out latest olive sources, which can be done with the following command line:

```
$ svn co svn://anonsvn.mono-project.com/source/trunk/olive
```

If you're a Windows user and you prefer working with a Graphic User Interface (GUI), there are several SVN clients which can make these tasks easier. It's worth to try TortoiseSVN, which can be downloaded from <http://tortoisesvn.tigris.org/>.

## 1.2. Build 3.0 Assemblies

Once you have the sources locally, just type these command lines in order to build and install 3.0 assemblies:

```
$ make PROFILE=net_3_0
$ make install
```

If everything went fine, you have everything you need but, for being sure every step has completed successfully, check if 3.0 assemblies are currently installed on the GAC as follows:

```
$ gacutil -l | grep 3.0
System.Xml.Linq, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a
3a
System.Workflow.ComponentModel, Version=3.0.0.0, Culture=neutral,
PublicKeyToken
=31bf3856ad364e35
System.Workflow.Activities, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31b
f3856ad364e35
System.Workflow.Runtime, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf38
56ad364e35
WindowsBase, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35
PresentationFramework, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=0738eb9f
132ed756
System.Runtime.Serialization, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=b
77a5c561934e089
System.IdentityModel, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561
```

## 10 1. Build Olive from Sources

```
934e089
System.IdentityModel.Selectors, Version=3.0.0.0, Culture=neutral,
PublicKeyToken
=b77a5c561934e089
System.ServiceModel, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c5619
34e089
PresentationCore, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364
e35
System.ServiceModel.Web, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c
561934e089
agclr, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35
```

As olive is in early stages, don't worry if your output differs from this one, but you'll need at least `System.ServiceModel`, among its dependencies, in order to work along next chapters.

## 2. Hello World Service

It's for sure professional training, but hackers always try to understand new APIs by using Hello World examples. Here it won't be less, so pick your computer up and get ready for writing some source code.

The next example is divided into two different pieces: client and server sides. You'll go through both of them, getting in detail in each one.

### 2.1. Sever Side

As almost every distributed application nowadays, it's needed a well-known piece of software which hosts a defined number of services, accessed by one or more clients.

In this example application, you'll create a server which publishes an interface `IHelloWorld`, a service contract, with a single operation `SayHello()`, an operation contract, which returns a string.

**Example 2-1.** Examples\HelloWorldService\Main.cs

```
[ServiceContract]
public interface IHelloWorld
{
    [OperationContract]
    string SayHello();
}
```

Don't worry if you still don't know what `ServiceContract` and `OperationContract` attributes mean, you'll do in the following chapters.

OK, so now you have *what* your service does. Next step is to decide *how* your service will behave when a client requests its operation. In order to achieve that, you must create a new class `HelloWorldService`, implementing `IHelloWorld`.

**Example 2-1.** Examples\HelloWorldService\Main.cs (continued)

```
public class HelloWorldService : IHelloWorld
{
    public string SayHello()
    {
        return "Hello, World!";
    }
}
```

It just returns that famous message, quite simple (you can change it to match your taste).

You've done almost everything, but there's still a last question before your service can be accessed: *where*.

**Example 2-1.** Examples\HelloWorldService\Main.cs (continued)

```
public static void Main(string[] args)
{
    using (ServiceHost sh = new ServiceHost(typeof(HelloWorldService)))
    {
        Binding binding = new BasicHttpBinding();
        ServiceEndpoint se = sh.AddServiceEndpoint(typeof(IHelloWorld),
            binding,
            new Uri("http://localhost:8080/HelloWorldService"));

        sh.Open();
    }
}
```

## 12 2. Hello World Service

```

        Console.WriteLine("Service ready. Press ENTER to shut down
            server...");
        Console.ReadLine();

        sh.Close();
    }
}

```

Briefly, as you'll go in detail later on, your service will host `HelloWorldService`, and it will be accessed through HTTP at `http://localhost:8080/HelloWorldService`.

Carrying on, you'll need a client to test if everything worked fine.

## 2.2. Client Side

You know *what* your service offers, and *where* it's located, so let's go with some code for accessing it.

### Example 2-2. Examples\HelloWorldClient\Main.cs

```

public static void Main(string[] args)
{
    BasicHttpBinding binding = new BasicHttpBinding();
    EndpointAddress ea = new EndpointAddress(
        new Uri("http://localhost:8080/HelloWorldService"));
    IHelloWorld proxy = ChannelFactory<IHelloWorld>.CreateChannel(
        binding, ea);

    try
    {
        Console.WriteLine("Response: {0}", proxy.SayHello());
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }

    Console.WriteLine("Press ENTER to shut down client...");
    Console.ReadLine();
}

```

It simply creates a proxy for the service, and lets treating its operation `SayHello()` as if it was available locally.

## 2.3. Build and Test

As final step, you must compile both client and server code, following next command lines. You'll start building and running server side:

```
$ gmcs -pkg:olive Main.cs -out:Server.exe
$ mono Server.exe
Service ready. Press ENTER to shut down server...
```

If everything went fine, you should see message shown above. Your service is now waiting for incoming requests, which will be done by client side (you'll need a different command window):

```
$ gmcs -pkg:olive Main.cs -out:Client.exe
$ mono Client.exe
Response: Hello, World!
Press ENTER to shut down client...
```

If your response matches "Hello, World!" (or your custom message), hurray, it worked! Just for fun, you can try executing assemblies on CLR and/or Mono (mixing platforms), and check if it also works.

In addition, a large bunch of examples are available at */samples/services/* under *olive* module. Inside each one of them, you'll find Makefiles for compiling.

## 3. Summary

If previous steps went OK, you've everything to enjoy the rest of chapters, independently on the OS you use. A brief summary on what you've done:

- If you weren't familiar with building Mono from sources, now you are. It's exciting to have updated sources daily!
- You've built Olive assemblies from sources, which's needed if you finally decide to contribute with this beautiful project.
- If you've never put your hands on WCF before, you've created and tested your first WCF distributed application.

Congratulations, you can continue then, you're prepared to get introduced into the wonderful world of Distributed Services!



# Chapter 3. WCF Fundamentals

As every distributed piece of software currently available, WCF shares some roots with existing distributed technologies like ASP.NET Web Services, CORBA, Java RMI, etc.

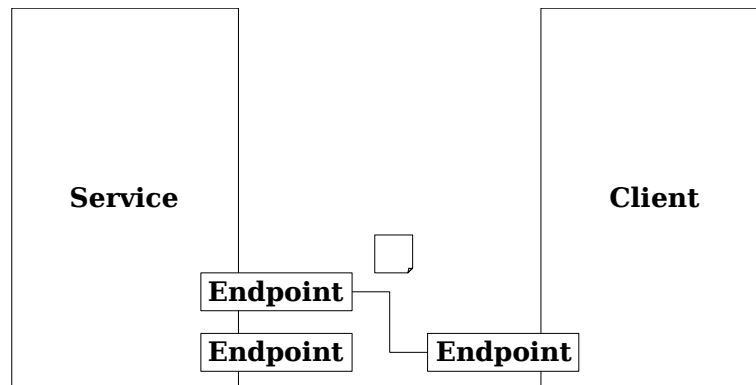
Within this chapter, you'll go through a brief overview of WCF model, as well as some must-known concepts such like addresses, bindings, contracts (the ABC, remember it), and so on.

## 1. Architectural Concepts

On a distributed scenario, client and server sides exchange information. In order to achieve this, it's needed an interface which communicates your application with the outside world, independently of the side you're located.

WCF model distinguishes between clients, those which active the communication, and services, those which await incoming clients to start a conversation.

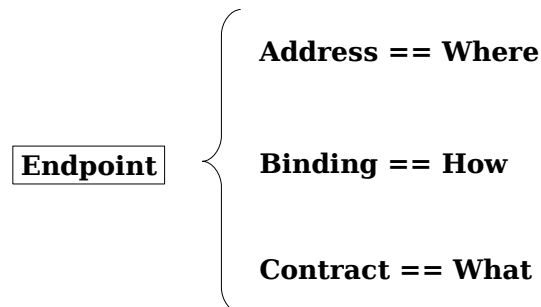
Information exchanged by clients and services is known as messages, and these are sent between endpoints. Figure 3-1 shows a common scenario.



**Figure 3-1.** Brief client-service communication overview

## 1.1. Endpoints

An endpoint is built upon three really important concepts: addresses, bindings and contracts (this last refers to syntactic interfaces, not semantic nor other non-functional requirements are expressed). You can easily remember them as ABC, as you can appreciate on Figure 3-2.



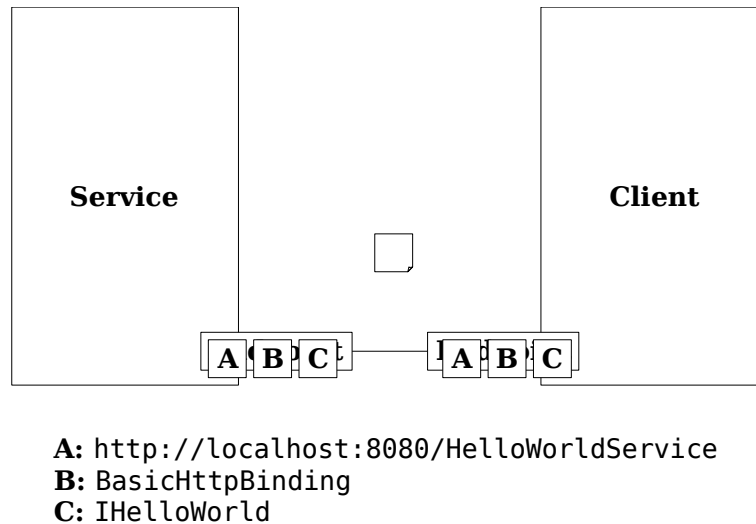
**Figure 3-2.** Endpoint configuration

Each one of them provides an answer to paired question:

- *Where*: an address makes and endpoint uniquely addressable, based on an URI, for example *http://localhost:8080/HelloWorldService*.
- *How*: a binding specifies the way communication will go through, for example HTTP, TCP, etc.
- *What*: a contract determines the external view of a service, such as data representation (data contracts), message composition (message contracts) and exposed operations (service

contracts). Note that here contracts are just syntactic artifacts; the semantics behind them is assumed to be implicit or documented elsewhere.

Putting it all together, and based on Hello World example from previous chapter, a more detailed scenario looks like Figure 3-3.

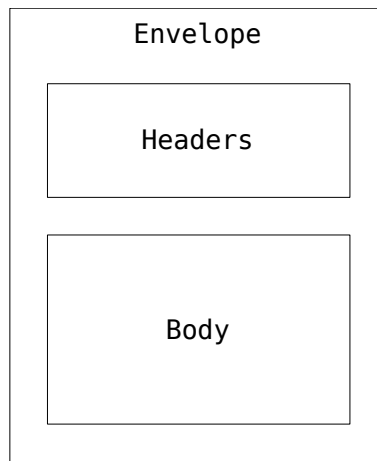


**Figure 3-3.** Hello World sample scenario

## 1.2. Messages

Now you've learned what an endpoint is, it's time to go for what endpoints exchange: messages. You can think of messages in the same way as an ordinary mail (e-mail also) is built.

Before existing e-mail, almost every letter you wanted to send needed to be inside an envelope. This also happens here. Inside the envelope, there are two important parts: headers and body. Figure 3-4 shows an overview.



**Figure 3-4.** Message components

Above structure's built up using XML. SOAP, known at the beginning (version 1.0) as the acronym for Simple Object Access Protocol, is a XML-based protocol for exchanging messages.

Following with example from previous chapter, you'll see underlying message exchange when this line is executed:

```
Console.WriteLine("Response: {0}", proxy.SayHello());
```

As above line is claiming to, a response message must be returned after the request, which must contains your specified string. Summing up, two messages take part: one requesting SayHello() call, and a response one with previous call result.

Going back to above line, when it's executed underlying calls are told to build an outgoing message, which for this example looks similar to Figure 3-5.

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Header>
    <Action s:mustUnderstand="1"
      xmlns="http://schemas.microsoft.com/
      ws/2005/05/addressing/none">
      http://tempuri.org/HelloWorld/SayHello
    </Action>
  </s:Header>
  <s:Body>
    <SayHello xmlns="http://tempuri.org/"></SayHello>
  </s:Body>
</s:Envelope>
```

## 20 1. Architectural Concepts

```
</s:Body>
</s:Envelope>
```

**Figure 3-5.** Request SOAP message

It's human-readable. Above message contains a single header where it specifies which operation is requesting, and an empty operation tag, inside Body one, which means there's no parameter. Message travels down to its final destiny. Once service side notices arrival, it proceeds to interpret what the message is requesting, and it returns a response, as Figure 3-6 shows.

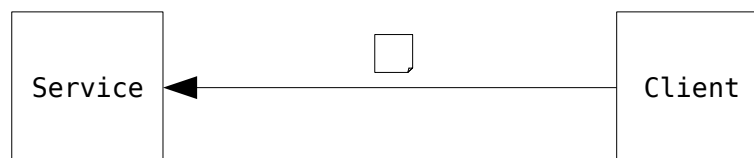
```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Header></s:Header>
  <s:Body>
    <SayHelloResponse xmlns="http://tempuri.org/">
      <SayHelloResult>Hello, World!</SayHelloResult>
    </SayHelloResponse>
  </s:Body>
</s:Envelope>
```

**Figure 3-6.** Response SOAP message

As previous Figure 3-5, it's easy to decode what message contains, so this work is let for you.

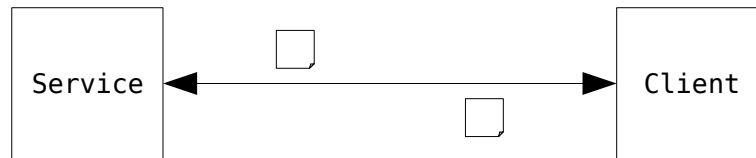
Above example follows one of the existing messaging patterns, more specifically request-reply one, but there are some more worth to mention depending on scenario topology:

- Simplex: just request messaging. There's no response at a SOAP transporting level, it's left to lower layer's hands.



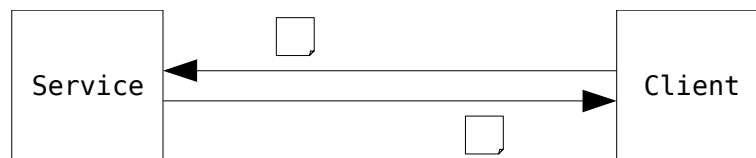
**Figure 3-7.** Simplex flow

- Duplex: asynchronous send-receive messaging.



**Figure 3-8.** Duplex flow

- Request-reply: synchronous request-reply messaging.



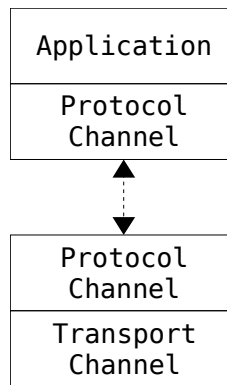
**Figure 3-9.** Request-reply flow

### 1.3. Channels

Now you know how a message is, next step before going ahead with a final sample is to have a look at WCF channels model.

On the contrary as spoken humans conversations take place, where there's no need to specify a concrete channel to communicate through (well, it'd be fine both to speak the same language; if not, you always can add an intermediate translator service, but this' up to you), within a WCF scenario it's needed a channel where all the messages will travel along.

WCF channel model is composed of stacks configured depending on topology constrains. Figure 3-10 shows a brief schema.



**Figure 3-10.** WCF channel stack

Messages flow up and down through above stack. In the middle, there can be protocols such like reliable messaging, which operate with messages modifying its headers, for example. At the bottom, a transport piece's in charge of transforming message object into a stream to be sent to its destiny.

Channels are created from bindings specifications, as these last ones determine channel stack in terms of Figure 3-10. This notebook doesn't cover bindings depth-along, but it's definitely a must to study before seriously working with WCF. You'll find an useful guide at <http://msdn2.microsoft.com/en-us/library/ms733027.aspx>.

## 2. Example

It's time for a final exam. Wait, wait! Well, not really, but a chance to play with things learned on this chapter.

WCF ships with a custom binding which lets you configure yourself a channel pipeline. That's what you'll work with in the following example.

### **Example 3-1.** Examples\CustomBindingService\Program.cs

```

public static void Main(string[] args)
{
    CustomBinding cb = new CustomBinding();
    cb.Elements.Add(new TextMessageEncodingBindingElement());
    cb.Elements.Add(new TcpTransportBindingElement());
}
  
```

```

IChannelListener<IDuplexSessionChannel> cl =
    cb.BuildChannelListener<IDuplexSessionChannel>(
        new Uri("net.tcp://localhost/CustomBindingService"),
        new BindingParameterCollection());
cl.Open();
Console.WriteLine("Waiting for clients...");
IDuplexSessionChannel dsc = cl.AcceptChannel();
Console.WriteLine("Waiting for incoming messages...");
dsc.Open();
Message m1 = dsc.Receive();
Console.WriteLine("Message received: {0}", m1.Headers.Action);
m1.Close();
dsc.Close();
cl.Close();
}

```

Above code relies on low-level messaging, in terms of manually working with them, instead of a higher layer as contract is. CustomBinding allows you to configure your own message pipeline, and heres it comprises TextMessageEncodingBindingElement, which will manage SOAP messages in a text format, and TcpTransportBindingElement, which will make use of TCP for transporting messages.

As you can appreciate, messaging pattern now is duplex one, which allows to send and receive messages asynchronously. Your service will simply wait for incoming connections, and once there'll be one, it'll wait for a new message.

### **Example 3-2.** Examples\CustomBindingClient\Program.cs

```

public static void Main(string[] args)
{
    CustomBinding cb = new CustomBinding();
    cb.Elements.Add(new TextMessageEncodingBindingElement());
    cb.Elements.Add(new TcpTransportBindingElement());
    IChannelFactory<IDuplexSessionChannel> cl =
        cb.BuildChannelFactory<IDuplexSessionChannel>(
            new Uri("net.tcp://localhost/CustomBindingService"),
            new BindingParameterCollection());
    cl.Open();
    IDuplexSessionChannel dsc = cl.CreateChannel(
        new EndpointAddress("net.tcp://localhost/CustomBindingService"));
    Console.WriteLine("Creating outgoing message...");
    dsc.Open();
    Message m1 = Message.CreateMessage(MessageVersion.Default, "Hello",
        "MyBody");
    dsc.Send(m1);
    m1.Close();
    dsc.Close();
}

```

## **24 2. Example**

```
        cl.Close();  
    }
```

As expected, client side uses the same binding as service one. It tries to create a channel and, if succeeded, it sends a new message through it.

Building and executing both assemblies in the same way as you did on previous chapter, you should see following output on service one:

```
$ mono CustomBindingService.exe  
Waiting for clients...  
Waiting for incoming messages...  
Message received: Hello
```

### 3. Summary

If WCF relies on fundamentals concepts, you've gone ahead with those along this chapter:

- You've learned the meaning of an endpoint, and what it's composed of: addresses, bindings and contracts (ABC).
- You've learned endpoints exchange messages, and how messages flow look like.
- You've learned what's the channel stack, and how it can be customized thanks to custom bindings.

There's still plenty of information concerning concepts seen on this chapter, as well as the whole set of available built-in bindings, but now you have a brief shot for keeping with following chapter.



# Chapter 4. Contracts

As you learned from Groucho Marx at Chapter 1, it's all about contracts. Contracts are the master piece of the puzzle, as they'll let you exchange information between endpoints.

WCF comes with three different built-in types of contracts: service contracts, data contracts and message contracts. Every of them have a specified purpose, as you'll see on the following lines.

## 1. Service Contracts

Even if this notebook is your first connection to WCF, you've already worked with them. Remember Example 2-1, where you defined a simple interface, `IHelloWorld`, to perform a simple operation, `SayHello()`.

**Example 2-1.** `Examples\HelloWorldService\Main.cs`

```
[ServiceContract]
public interface IHelloWorld
{
    [OperationContract]
    string SayHello();
}
```

Service contracts then specify what explicit operations a service can perform, and as the rest of contract types it's done using OO tools: classes and interfaces, with behavior attributes. Minimum requirements for a WCF service is to include at least one service contract, but it's currently possible to expose more than one. It's up to you.

Usual steps to define a service contract is to start up with an interface, marked with `ServiceContract` and `OperationContract` attributes. Following interface, Figure 4-1, contains a single service contract, `ICalculator`, with four operations contracts.

```
[ServiceContract]
public interface ICalculator
{
    [OperationContract]
    int Add(int n1, int n2);

    [OperationContract]
    int Divide(int n1, int n2);

    [OperationContract]
    int Multiply(int n1, int n2);

    [OperationContract]
    int Subtract(int n1, int n2);
}
```

**Figure 4-1.** `ICalculator` contract

Once you have an interface which specifies your service contract, next step is to keep up with an implementation for your interface, as Figure 4-2 shows.

```
public class CalculatorService : ICalculator
{
    public int Add(int n1, int n2)
    {
        return n1 + n2;
    }

    public int Divide(int n1, int n2)
    {
        return n2 != 0 ? n1 / n2 : 0;
    }

    public int Multiply(int n1, int n2)
    {
        return n1 * n2;
    }

    public int Subtract(int n1, int n2)
    {
        return n1 - n2;
    }
}
```

**Figure 4-2.** CalculatorService contract inheriting ICalculator

Done. However, if you want to write less lines of code (even less!) you can avoid interface and go ahead directly to annotating class with attributes. Figure 4-3 claims this.

```
[ServiceContract]
public class CalculatorService
{
    [OperationContract]
    public int Add(int n1, int n2)
    {
        return n1 + n2;
    }

    [OperationContract]
    public int Divide(int n1, int n2)
    {
        return n2 != 0 ? n1 / n2 : 0;
    }

    [OperationContract]
    public int Multiply(int n1, int n2)
    {
        return n1 * n2;
    }

    [OperationContract]
    public int Subtract(int n1, int n2)
    {
        return n1 - n2;
    }
}
```

**Figure 4-3.** CalculatorService built-in contract

Finally, ServiceContract and OperationContract attributes contain properties to specify, for instance, name and namespace, but you won't go in detail here. You can find wide information at <http://msdn2.microsoft.com/en-us/library/system.servicemodel.servicecontractattribute.aspx> and <http://msdn2.microsoft.com/en-us/library/system.servicemodel.operationcontractattribute.aspx>.

## 2. Data Contracts

Within previous CalculatorService sample, every data type used

was simple, just integers. What happens when you want to exchange an object which type is `People` (imagine it's made up of an integer `Age`, and a string `Name`, for instance), a complex one? Well, then you have to make use of data contracts.

## 2.1. When To Use Them and When Not

Before going ahead, you must know the difference between explicit and implicit data contracts. `CalculatorService`'s based on implicit ones, as it uses a simple type. Simple types include implicit data contracts in the way they automatically know how to be serialized, unlike complex ones, where you must define an explicit data contract for them.

Following list shows which types are automatically de/serialized, it means, those with implicit data contracts (extracted from <http://msdn2.microsoft.com/en-us/library/ms731923.aspx>):

- Collection types, such regular arrays of types, or specific collections like `ArrayList` and `Dictionary`
- Enumeration types
- .NET primitive types:
  - `System.Boolean`
  - `System.Byte`
  - `System.Char`
  - `System.Decimal`
  - `System.Double`
  - `System.Int16`
  - `System.Int32`
  - `System.Int64`
  - `System.Object`
  - `System.SByte`
  - `System.Single`
  - `System.String`
  - `System.UInt16`
  - `System.UInt32`

- System.UInt64
- Other primitive ones:
  - System.DateTime
  - System.Guid
  - System.TimeSpan
  - System.Uri
  - System.Xml.XmlQualifiedName
  - Arrays of Byte
- Types marked with Serializable attribute, any ISerializable one
- Types representing plain XML (System.Xml.XmlElement, System.Xml.XmlNode and System.Xml.Serialization.IXmlSerializable types) and ADO.NET data structures

## 2.2. The Important Thing Is at The Shell

You don't need to use exactly the same data contract on both sides, client and service. Each one of them can be internally different, with different field names, different access modifiers, etc., depending on different requirements.

In order to get a data contract compatible with two sides, it must match name, namespace and list of members.

Following with the same scenario as previously, imagine you know there's a service you want to communicate with which exposes data contract shown on Figure 4.4.

```
[DataContract(Namespace="http://mono-olive-notebook.com")]
public class People
{
    [DataMember]
    public int Age;
    [DataMember]
    public string Name;
}
```

**Figure 4-4.** Service side's data contract

Imagine now you're coding client side and you decide that due to coding guideline used at your hypothetical company class fields names must be lowercase and preceded by an underscore, and class name must contain "Class" string at the end. You also decide to save ID card for client-only purpose. No problem. You'd go for an approach similar to Figure 4.5.

```
[DataContract(Name="People", Namespace="http://mono-olive-notebook.com")]
public class PeopleClass
{
    [DataMember(Name="Age")]
    private int _age;
    [DataMember(Name="Name")]
    private string _name;
    private string _id_card;
}
```

**Figure 4-5.** Client side's data contract

That's all. Your new data contract will work without issues against service's one. Please note that access modifiers are also different on both data contracts, but as long as you mark those fields with DataMember attribute, from data contract point of view there's no difference. You can also change data types, but make sure there's an implicit conversion available.

For further information on data contracts, please refer to <http://msdn2.microsoft.com/en-us/library/ms733127.aspx>.

### 3. Message Contracts

By default, WCF framework decides SOAP structure when serializing data: this field goes with message headers, this other one inside message body, and so on. Message contracts let you modify by-default behavior and specify where to add each thing.

Keeping imagination up, imagine now there's a service for notifying new bugs into your product. Figure 4-6 shows a simple service contract for that purpose.

```

[ServiceContract]
public interface IBugTracking
{
    [OperationContract]
    public void OpenNewBug(BugNotificationRequest message);
}

```

**Figure 4-6.** Really simple bug tracking service contract

Now, you can specify `BugNotificationRequest` as a message contract, and tell it where to include each field, as Figure 4-7 exposes.

```

[MessageContract]
public class BugNotificationRequest
{
    [MessageHeader]
    public string BugID;
    [MessageBodyMember]
    public string Summary;
}

```

**Figure 4-7.** `BugNotificationRequest` message contract

What you've done above is known as a typed message, and it avoids to work directly at raw XML level. A typed message can be used both on requests and replies.

There are other scenarios where it's possible to use message contracts, as well as specific properties to used attributes, so you can find them at <http://msdn2.microsoft.com/en-us/library/ms730255.aspx>.

## 4. Summary

You've learned the most important aspects on WCF contracts. You've briefly studied:

- Service contracts: they specify which operations a service can perform.
- Data contracts: how to use them within complex data types scenarios, as well as the difference between explicit and im-

PLICIT data contracts.

- Message contracts: where to put specific fields inside a SOAP message.

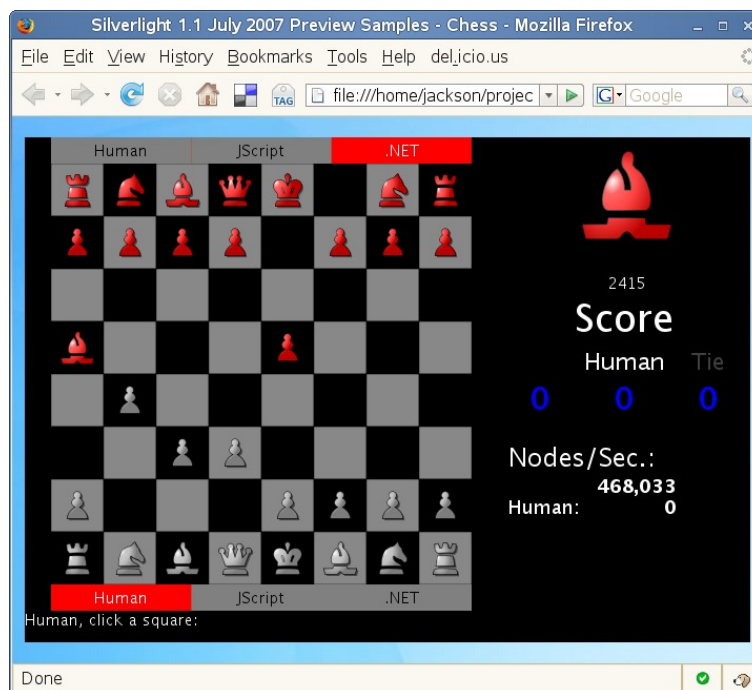
Contracts are one of the most important keys on SOA world (remember third tenet at Chapter 1).

# Chapter 5. Mono Olive in Depth

Olive doesn't just cover WCF, it also includes 3.0, and incoming 3.5, profiles. Even further, it comprises codename Moonlight, Mono's Silverlight implementation to port existing browser plug-in into Unix world.

## 1. Moonlight

Twenty-one days. That's the time a bunch of Mono developers group took to implement Microsoft's Silverlight for Mono to a state where it started to be usable. Twenty-one days after the first commit, it was able to render existing demos like rotating videos, complex animations and so forth. Figure 5-1 shows a shot from a demo working on Moonlight.



**Figure 5-1.** Silverlight 1.1 chess demo running on Moonlight

If you feel you'd like to help improving this effort, following points will also help you as Olive shares some roots with this one. However, you can look for current status at <http://www.mono-project.com/Moonlight>.

## **2. How To Collaborate**

Would you like to take part into this project? If you would, great, thanks in advance! Here you'll find some steps for getting started based on author experience. Following is just a recommendation, so don't hesitate to ask if you need further help, or similar.

### **2.1. Step 0: Things You Should Know Before**

If you've finally decided to contribute in a development way, there's one topic you should take care of before getting started.

Mono is an implementation of Microsoft's .NET. Current .NET license for class libraries don't allow them to be reused within Mono, so that's the main reason why it's being implemented from zero.

However, .NET assemblies can be disassembled, so you can get almost original source code for every managed piece. If you've ever done this before, sorry, but you can't then contribute as developer.

Don't worry, you still can go for other ways such like writing documentation, which are listed at <http://www.mono-project.com/Contributing>.

Mono source code follows a defined coding guideline. This helps to maintain clear and coherent source code within a huge project as Mono is, and keep similar pieces independently on where it comes from. You'll find a detailed list and best practices at [http://www.mono-project.com/Coding\\_Guidelines](http://www.mono-project.com/Coding_Guidelines).

## 2.2. Step 1: Get in Touch

Currently Olive contributors get synchronized mostly by a distribution list located at <http://groups.google.com/group/mono-olive>. You can join it your-self, and have a look to existing topics. If you're interested on 3.0 and 3.5 profiles development, this is your place.

There are also some interesting distribution lists worth to join. You can find all of them listed at [http://www.monoproject.com/Mailing\\_Lists](http://www.monoproject.com/Mailing_Lists), but here's a brief summary on those more important for your daily use:

- *mono-devel-list*: This' the main resource for whoever contributes towards Mono. Developers without access to development Subversion account send patches here, which are revised and applied by people from Mono team. Summing up, you must definitely join this list.
- *mono-svn-patches*: Every commit done to SVN repository, independently on the project where it's done, is mirrored here. It's really recommended to join, but without receiving copies to your e-mail. Use this list as a resource for looking for specific patches and so on.

Internet Relay Chat (IRC) is another must. It exists a main channel at <irc.gimp.net>, #mono, where you'll find people who will try to help answering questions. Write your question once, someone almost sure will read it and help you but, if there's no answer, it's likely everyone is busy with other tasks.

## 2.3. Step 2: Write Tests

Once you know which piece you'd like to implement, it's highly recommended to write unit tests before. There are some reasons why writing tests is a good way: it lets you learn how to use API you're going to work with, and also it's useful as regression test to ensure Mono behavior is the same as .NET one.

If you've never gotten in touch with NUnit (<http://www.nunit.org/>) before, the framework used for unit tests, the main idea is to write a test class and mark it, and its methods, with some specific attrib-

utes. Figure 5.1 shows a possible template.

```
//
// ClassNameTest.cs
//
// Author:
//     Name Surname (e-mail)
//
// Copyright Info
//

using System;
// FIXME: To include here namespace you'll work with.

using NUnit.Framework;

namespace MonoTests.Full.Namespace.Path
{
    [TestFixture]
    public class ClassNameTest
    {
        // You can declare here the class you'll test against.
        private ClassToBeTested c;

        [SetUp]
        protected void Setup ()
        {
            c = new ClassToBeTested ();
        }

        [Test]
        public void Method1Test ()
        {
            c.Method1 ();
        }

        [Test]
        [ExpectedException (typeof (InvalidOperationException))]
        public void Method2Test ()
        {
            c.Method2 ();
        }
    }
}
```

**Figure 5.1.** NUnit possible template

Briefly, Nunit will execute every method contained in your class, and before each method execution, it'll invoke `Setup()` for creating a new clean object before each test. There are multiple ways of writing unit tests, but those won't be covered here. It's really encouraged to visit [http://www.mono-project.com/Test\\_Suite](http://www.mono-project.com/Test_Suite) for more in depth information.

## 38 2. How To Collaborate

Once you have a nice test suite, send it to *mono-devel-list*. Someone will check it and proceed to commit your contribution. Really thanks in advance!

## **2.4. Step 3: Ready, Steady, Go!**

OK, so you're ready to go a step further. Olive WCF has still a lot of unsupported pieces, and these are tracked down with class libraries status pages, as the ones you can find at <http://www.youcanteatbits.org/Projects/Olive/>.

If you've already decided which part you'd like to contribute to, superb. Keep Olive developers updated through its distribution list on your progress, you can receive useful feedback during your work.

If you don't know what to start with, don't hesitate to ask in the distribution list also, they'll likely guide you on preferred working areas and so on.

## **3. Summary**

This chapter has briefly guided you through Olive effort, which parts it comprises and some basic steps in order to become a contributor.

On contributing, you should remember these key points:

- Don't look at .NET original sources. Due to license matters, if you do that you couldn't contribute to in a developer way, but you could in some other areas.
- Follow coding guidelines. It helps to have source code consistent.
- Write unit tests. Independently on your decision on going further with a specific implementation or similar, try to start always with unit tests.
- Keep in touch. Feedback is a really important point which can help you to achieve your goals through the proper way. Mem-

bers help members.

If you finally decide to help improving Mono, really thanks in advance on behalf of the whole community. Welcome to this amazing project.

# Conclusion

If you've arrived here after spending some time reading previous chapters, you likely have noticed there have been many aspects uncovered. There's a handful of books on WCF written by professional programmers which cover all this left information (in a, almost sure, better way), and some of them are listed at Bibliography.

The main goal for this developer's notebook was different. It was more about community, like the one which has accompanied the author to spend a superb Summer of Code collaborating with The Mono Project and Olive.

You're now in a wonderful position to help this project in the process of growing, like if a baby it was. So now, it's up to you.

Honestly, we'd like to hear from you. Thanks for your time.



# Bibliography

1. Microsoft Corporation, ed. "Channel Model Overview." <<http://msdn2.microsoft.com/en-us/library/ms729840.aspx>>.
2. Dumbill, Edd, and Niel M. Bornstein. "Mono: a Developer's Notebook". First ed. O'Reilly Media, Inc., 2004.
3. "Object-Oriented Programming." Wikipedia, the Free Encyclopedia. 9 Aug. 2007 <[http://en.wikipedia.org/wiki/Object-oriented\\_programming](http://en.wikipedia.org/wiki/Object-oriented_programming)>.
4. Pallmann, David. "Programming Indigo". Beta ed. Microsoft Press, 2005.



# Appendix A: Full Examples

## Example 2-1. Examples\HelloWorldService\Main.cs (full)

```
//
// Main.cs
//
// Author:
//   Marcos Cobena (marcoscobena@gmail.com)
//
// Copyright 2007 Marcos Cobena (http://www.youcannoteatbits.org/)
//

using System;
using System.Collections.Generic;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;

namespace HelloWorldService
{
    class MainClass
    {
        public static void Main(string[] args)
        {
            using (ServiceHost sh = new
ServiceHost(typeof(HelloWorldService)))
            {
                Binding binding = new BasicHttpBinding();
                ServiceEndpoint se =
sh.AddServiceEndpoint(typeof(IHelloWorld),
                                                                    binding,
                                                                    new
Uri("http://localhost:8080/HelloWorldService"));

                sh.Open();

                Console.WriteLine("Service ready. Press ENTER to shut
down server...");

                Console.ReadLine();

                sh.Close();
            }
        }
    }
}
```

```

[ServiceContract]
public interface IHelloWorld
{
    [OperationContract]
    string SayHello();
}

public class HelloWorldService : IHelloWorld
{
    public string SayHello()
    {
        return "Hello, World!";
    }
}
}

```

### Example 2-2. Examples\HelloWorldClient\Main.cs (full)

```

//
// Main.cs
//
// Author:
//   Marcos Cobena (marcoscobena@gmail.com)
//
// Copyright 2007 Marcos Cobena (http://www.youcannoteatbits.org/)
//

using System;
using System.Collections.Generic;
using System.ServiceModel;

namespace HelloWorldClient
{
    class MainClass
    {
        public static void Main(string[] args)
        {
            BasicHttpBinding binding = new BasicHttpBinding();
            EndpointAddress ea = new EndpointAddress(new
Uri("http://localhost:8080/HelloWorldService"));
            IHelloWorld proxy =
ChannelFactory<IHelloWorld>.CreateChannel(binding, ea);

            try
            {
                Console.WriteLine("Response: {0}", proxy.SayHello());
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }

            Console.WriteLine("Press ENTER to shut down client...");
            Console.ReadLine();
        }
    }
}

```

```

    [ServiceContract]
    public interface IHelloWorld
    {
        [OperationContract]
        string SayHello();
    }
}

```

### Example 3-1. Examples\CustomBindingService\Program.cs (full)

```

//
// Program.cs
//
// Author:
//   Marcos Cobena (marcoscobena@gmail.com)
//
// Copyright 2007 Marcos Cobena (http://www.youcannoteatbits.org/)
//

using System;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;

namespace CustomBindingService
{
    class Program
    {
        public static void Main(string[] args)
        {
            CustomBinding cb = new CustomBinding();
            cb.Elements.Add(new TextMessageEncodingBindingElement());
            cb.Elements.Add(new TcpTransportBindingElement());
            IChannelListener<IDuplexSessionChannel> cl =
                cb.BuildChannelListener<IDuplexSessionChannel>(
                    new
Uri("net.tcp://localhost/CustomBindingService"),
                    new BindingParameterCollection());
            cl.Open();
            Console.WriteLine("Waiting for clients...");
            IDuplexSessionChannel dsc = cl.AcceptChannel();
            Console.WriteLine("Waiting for incoming messages...");
            dsc.Open();
            Message m1 = dsc.Receive();
            Console.WriteLine("Message received: {0}", m1.Headers.Action);
            m1.Close();
            // FIXME: It isn't currently supported. Your collaboration is
welcome. :-))
            //   Message m2 = Message.CreateMessage(MessageVersion.Default,
"Bye", "MyBody");
            //   dsc.Send(m2);
            //   m2.Close();
            //   dsc.Close();
            //   cl.Close();
        }
    }
}

```

```
}
```

### **Example 3-2.** Examples\CustomBindingClient\Program.cs (full)

```
//  
// Program.cs  
//  
// Author:  
//     Marcos Cobena (marcoscobena@gmail.com)  
//  
// Copyright 2007 Marcos Cobena (http://www.youcannoteatbits.org/)  
//  
  
using System;  
using System.ServiceModel;  
using System.ServiceModel.Channels;  
using System.ServiceModel.Description;  
  
namespace CustomBindingClient  
{  
    class Program  
    {  
        public static void Main(string[] args)  
        {  
            CustomBinding cb = new CustomBinding();  
            cb.Elements.Add(new TextMessageEncodingBindingElement());  
            cb.Elements.Add(new TcpTransportBindingElement());  
            IChannelFactory<IDuplexSessionChannel> cl =  
                cb.BuildChannelFactory<IDuplexSessionChannel>(new  
Uri("net.tcp://localhost/CustomBindingService"),  
                new BindingParameterCollection());  
            cl.Open();  
            IDuplexSessionChannel dsc = cl.CreateChannel(new  
EndpointAddress("net.tcp://localhost/CustomBindingService"));  
            Console.WriteLine("Creating outgoing message...");  
            dsc.Open();  
            Message m1 = Message.CreateMessage(MessageVersion.Default,  
"Hello", "MyBody");  
            dsc.Send(m1);  
            m1.Close();  
            // FIXME: It isn't currently supported. Your collaboration is  
welcome. :-)  
            //            Message m2 = dsc.Receive();  
            //            Console.WriteLine("Message received: {0}", m2.Headers.Action);  
            //            m2.Close();  
            dsc.Close();  
            cl.Close();  
        }  
    }  
}
```