Using SAP Enterprise Service with Microsoft's Windows Communication Foundation: The Handling of GDT's

Applies to:

SAP Enterprise Service, Global Data Types, Microsoft WCF, Healthcare Industries Patient Administration

For more information, visit the Interoperability - .NET homepage.

Summary

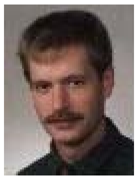
Integrating web services is not always an easy task, especially when different technologies are involved. In this article we address the consumption of SAP Enterprise Services (SAP ES) with Microsoft's Windows Communication Foundation (WCF). Every web service technology has the ability of building reusable types that can be used in different services. In SAP ES these types are called Core Data Types (CDT) or Global Data Types (GDT) and published in the "Data Type Catalogue - Definitions of Global Data Types and Core Data Types". If your task is only the integration of only one service in your WCF application, the standard Microsoft NET 3.5 tool svcutil.exe (Service Model Metadata Utility Tool) works well. But if your task is the integration of many services, you will not import the same basic type n times in your project. So this paper gives you some hints for solving this problem.

Author: Andreas Ascher

Company: Meierhofer AG

Created on: 19 October 2008

Author Bio



Andreas Ascher received his degree (Dipl.-Inform. Univ.) In computer science 1995 from the University of Passau. Since 2000 he is working as senior developer for the **MEIERHOFER AG**, the software producer of the medical information system **MCC**. His main interest is the application integration in health care enterprises.

Acknowledgement

The result of this paper was inspired by the PCDG Healthcare Industries Patient Administration v.2 between SAP AG and MEIERHOFER AG. Claudius Metze, Bettina Liske, Christian Stark and Dirk Riemer deserve credit for making this paper possible, through their feedback and advice. Thanks go out to Nektarios Dourdas and other peoples from SAP behind the stage, who supplies us with additional material. Last but not least, special thanks go to the MEIERHOFER AG, especially Markus Rathmayer for giving me the opportunity to write this article.

Table of Contents

The Problem

Consuming SAP Enterprise Services

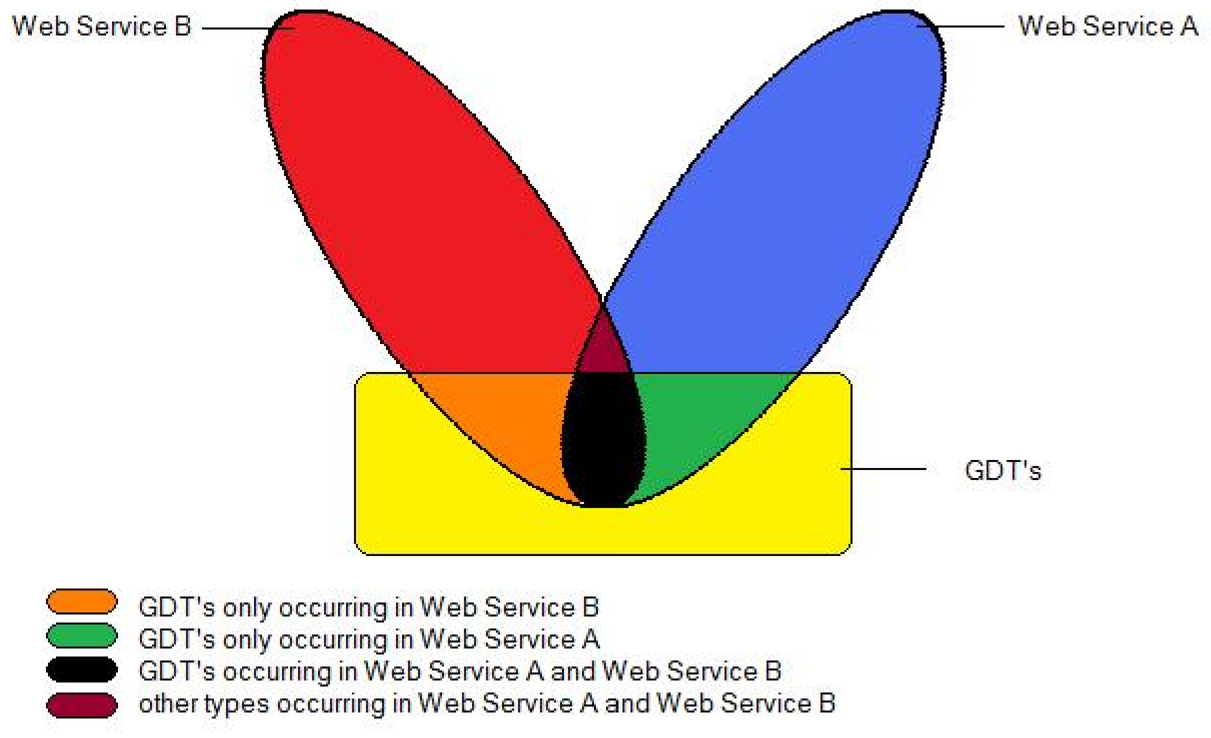
In [1] is basically shown how you can obtain and consume SAP Enterprise Services in a WCF Development Environment. Note there is an important difference comparing the WCF approach, we state as premise 1:

Note: The metadata of a SAP Enterprise Service is distributed in one flat WSDL file, containing all information's that are necessary to generate code.

Secondly, if you concern in detail with SAP ES, you will notice, that SAP has build a big type system that is used throughout Enterprise Services.

Note: SAP has developed a type system, of types that are common used in SAP ES and are published in the **Data Type Catalogue - Definitions of Global Data Types and Core Data Types**[1].

As an example we give the distribution of types of two services in a venn-diagram.


Web Service B
Web Service A
GDT's
GDT's only occurring in Web Service B
GDT's only occurring in Web Service A
GDT's occurring in Web Service A and Web Service B
other types occurring in Web Service A and Web Service B

The GDT Import Problem

If your job is the implementation of an integration between your application and an SAP Business Solution by SAP ES, normally you will have to consume not one but n services. The aim, when importing n WSDL files, is to generate code for a GDT only once, because nobody wants to code the logic of mapping between your internal types to global data types more than once.

Regrettably, the WCF on board remedy svcutil.exe (Service Model Metadata Utility Tool) works not adequate for this task. Let us a have a short look why.

[1] With the term GDT's in this paper we often reference GDT's and CDT's.

WCF advocates an explicit contract based approach in building web service applications. This includes

- The type declaration is normally done in a programming language via special attributes (e.g. DataContract, DataMember).
- When mapping a data contract into xml schemas, only a subset is used for improving the interoperability between WCF applications (the process of building metadata from code and vice versa).
- The serialization is called data contract serialization[2].

When svcutil recognize data contracts in wsdl metadata it is a powerful tool. So you can e.g. exclude data contract types that are already in a referenced assembly.

Naturally the tool does not recognize data contracts in wsdl files of SAP ES, hence many advanced options are not possible. So we research for another solution.

The Approach

Fortunately, the NET 3.5 framework ships with classes, which gives you the opportunity to write your own code generator when importing WSDL files.

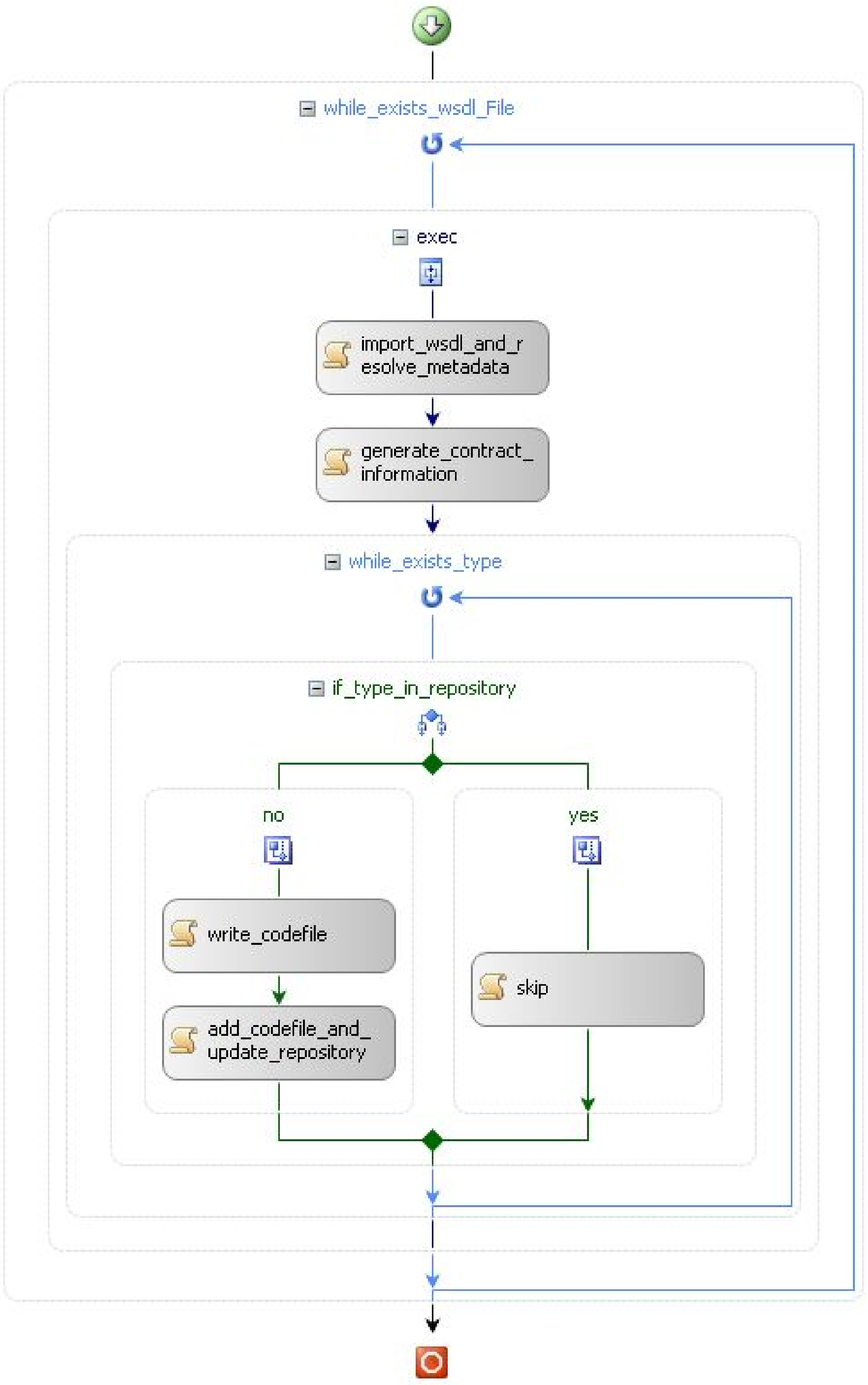
The Strategy

First we give a short diagram of our strategy[3].

We assume, we have to integrate n SAP Enterprise services given in n WSDL files in our project. Design your own repository[4] that holds information about types you have already imported.

The first task is to import the WSDL files and resolve the metadata generating contract information for all types. Then ask your repository "is the type already imported". If so skip the type, otherwise generate the code file for the type and update your repository.

[2] The standard serialization when working with arbitrary WSDL metadata is called xml serialization.
[3] Note that we handle all WSDL-types in this manner, not only GDT's
[4] As repository we simply assume a place, where information about imported types is persistent. Actually we keep this information only in our project files, but we take also into account to use a database for documenting additional information.


while_exists_wsdl_File
exec
import_wsdl_and_resolve_metadata
generate_contract_information
while_exists_type
if_type_in_repository
no
yes
write_codefile
add_codefile_and_update_repository
skip

Steps Generating Code

Now have a look at some code. We show only the basic ideas, Please contact Microsoft's MSDN for further information.

Read WSDL and Get The Contracts

In the .NET Framework the class MetadataSet represents a serializable collection of service metadata in XML form. The XML representation is imported (class WSDLImporter) and contracts (class ContractDescription) are identified. From these contracts contract types – the input of the code generator - are build (class ServiceContractGenerator).

See the following Listing:

```
private ServiceContractGenerator ResolveImports(string url)
{
    // Represents a serializable collection of service metadata in XML form
    MetadataSet metadataSet = new MetadataSet();

    // using wsdl file instead of online metadata exchange
    // complicates the import a little bit
    // we use here the DiscoveryClientProtocol class
    DiscoveryClientProtocol dcp = new DiscoveryClientProtocol();
    dcp.Credentials = CredentialCache.DefaultCredentials;
    dcp.AllowAutoRedirect = true;
    dcp.DiscoverAny(url);
    dcp.ResolveAll();

    // now build the MetadataSet
    foreach (object osd in dcp.Documents.Values)
    {
        if (osd is System.Web.Services.Description.ServiceDescription)
        {
            MetadataSection mds =
                MetadataSection.CreateFromServiceDescription(
                    (System.Web.Services.Description.ServiceDescription) osd );
            metadataSet.MetadataSections.Add(mds);
            continue;
        }
        if (osd is XmlSchema)
        {
            MetadataSection mds = MetadataSection.CreateFromSchema((XmlSchema)osd);
            metadataSet.MetadataSections.Add(mds);
            continue;
        }
    }

    // Import the MetadataSet
    WsdlImporter importer = new WsdlImporter(metadataSet);

    // and get the ContractDescription of Windows Communication Foundation Contracts
    Collection<ContractDescription> contracts = importer.ImportAllContracts();

    // The class ServiceContractGenerator helps us to generate the types
    ServiceContractGenerator generator = new ServiceContractGenerator();

    foreach (ContractDescription contract in contracts)
    {
        // now generate service contract types
        // these type are the base for generation code
        generator.GenerateServiceContractType(contract);
    }

    return generator;
}
```

Generate the Code

In the next step we have to generate the code files. The necessary classes are in the namespace System.CodeDom.Compiler.

We give two short listings. The first listing shows how to iterate above all type and the usage of a possible repository.

```
    private void GenerateCodeFromServiceContractGenerator(
        ServiceContractGenerator generator,
        YourRepository repository )
    {
        // iterate above all types
        foreach (CodeNamespace cn in generator.TargetCompileUnit.Namespaces)
        {
            foreach (CodeTypeDeclaration ctd in cn.Types)
            {
                // query your Respository concerning the type
                if ( !repository.ExistsType( cn, ctd ))
                {
                    // support a function that generates unique filenames for a type
                    string fileName =
                        repository.BuildUniqueCodeFileName( cn, ctd );

                    // update your Respository with rhe type info
                    repository.InsertTypeInfo(cn, ctd, fileName);

                    // and generate the source code file
                    this.GenerateCodeFromTypeDeclaration(fileName, cn, ctd);
                }
            }
        }
    }
```

The second listing generates the source code file.

```
private void GenerateCodeFromTypeDeclaration(string filename,
    CodeNamespace cn, CodeTypeDeclaration ctd)
{
    // We want to generate C# code
    System.CodeDom.Compiler.CodeGeneratorOptions options =
        new System.CodeDom.Compiler.CodeGeneratorOptions();
    options.BracingStyle = "C";
    System.CodeDom.Compiler.CodeDomProvider codeDomProvider =
        System.CodeDom.Compiler.CodeDomProvider.CreateProvider("C#");

    // Create a TextWriter
    System.CodeDom.Compiler.IndentedTextWriter methodWriter =
         new System.CodeDom.Compiler.IndentedTextWriter(
             new System.IO.StreamWriter(filename));

    // we add some comments to the generated source code file
    // namespace and name of the type, and the source wsdl file name
    CodeCommentStatement ccs1 =
        new CodeCommentStatement(
            "Namespace:" + cn.Name +
            "." + ctd.Name);
    CodeCommentStatement ccs2 =
        new CodeCommentStatement("Source:" +
            this.CurrentSourceWSDLFileName);
    ctd.Comments.Add(ccs1);
    ctd.Comments.Add(ccs2);

    // write using and namespace declaration
    foreach (CodeNamespaceImport cni in cn.Imports)
    {
        methodWriter.WriteLine("using " + cni.Namespace + ";");
    }

    if (this.DefaultNameSpace != string.Empty)
        methodWriter.WriteLine("using " + this.DefaultNameSpace + ";");

    // when no namespace is given in the wsdl,
    // we use a default namespace from this.DefaultNameSpace
    bool useNamespace = ( cn.Name != string.Empty ||
        this.DefaultNameSpace != string.Empty );

    if (useNamespace)
    {
        methodWriter.WriteLine("namespace " +
            (cn.Name != string.Empty ? cn.Name : this.DefaultNameSpace));
        methodWriter.WriteLine("{");
    }

    // generate the code for the type
    codeDomProvider.GenerateCodeFromType(ctd, methodWriter, options);

    // close } when using namespaces
    if (useNamespace)
    {
```

```
                methodWriter.WriteLine("}");
            }

            methodWriter.Close();
        }
```

A Sample Result

As conclusion we give a sample result. The GDT type Note may have the following encoding in your WSDL-file.

```
    <xsd:complexType name="Note">
        <xsd:annotation>
         <xsd:documentation xml:lang="EN">
          <ccts:RepresentationTerm>Note</ccts:RepresentationTerm>
         </xsd:documentation>
        </xsd:annotation>
        <xsd:simpleContent>
         <xsd:extension base="xsd:string">
          <xsd:attribute name="languageCode" type="LanguageCode"/>
         </xsd:extension>
        </xsd:simpleContent>
      </xsd:complexType>
    No have a look at the generated code file:
    using YourNamespace;
    namespace YourNamespace
    {
    /// <remarks/>
    // Namespace:.Note
    //
    Source:C:\Projekt\MCC.NET\Eai\PatientManagement\SAPEnterpriseService\SAPEnterpriseSer
    vice\WSDL\Create_Patient_Encounter.wsdl
    [System.CodeDom.Compiler.GeneratedCodeAttribute("Mag.Mcc.Tools.WCFHelper",
    "1.0.0.0")]
    [System.SerializableAttribute()]
    [System.Diagnostics.DebuggerStepThroughAttribute()]
    [System.ComponentModel.DesignerCategoryAttribute("code")]
    [System.Xml.Serialization.XmlTypeAttribute(Namespace="http://sap.com/xi/IS-
    H/Global2")]
    public partial class Note
    {

        private string languageCodeField;

        private string valueField;

        /// <remarks/>
        [System.Xml.Serialization.XmlAttributeAttribute(DataType="language")]
        public string languageCode
        {
            get
            {
                return this.languageCodeField;
            }
            set
            {
                this.languageCodeField = value;
            }
        }

        /// <remarks/>
        [System.Xml.Serialization.XmlTextAttribute()]
        public string Value
        {
            get
            {
                return this.valueField;
            }
            set
            {
                this.valueField = value;
            }
        }
    }
    }
```

Related Content

[1] Boris Mueller, SAP AG, How to Consume Enterprise Services with Microsoft .NET 3.0 and Visual Studio 2005

[2] Steve Resnick, Richard Crane, Chris Bowen, Essential Windows Communication Foundation (WCF) (2008)

[3] Microsoft MSDN .NET Framework 3.5 Home Page

For more information, visit the Interoperability - .NET homepage.

Copyright