Creating DAL Components Using Custom ASP.NET Build Providers And Compiler Techniques

Jürgen Bäurle

September 2006

Parago Media GmbH & Co. KG

Introduction

There are many articles on the internet that are dealing with the creation and usage of a Data Access Layer (DAL) and the comprising components, also known as DALC (DAL Components). There is nothing new in the process of creation a DAL. You can actually use Typed DataSets, Microsoft's Enterprise Library (DAAB) or you may use one of the many third party tools to implement a comprehensive DAL system.

The main objectives of this composition are to show how to create and use ASP.NET build providers and also to explain how easy you can analyze a small self-defined description language to declare DALCs or anything else. To implement a full-blown DAL killer application which you can use for the rest of your programming life is not the objective of this article. Normally you would not define your own description language to declare DALCs, you would instead use a XML-based description of the components and analyze them using the feature-rich XML classes that comes with the .NET framework.

I just wanted to implement a lexical analyzer (a.k.a. scanner or tokenizer), some parsing techniques and dynamic code generation using the .NET's CodeDOM. There a lot of situations in daily work where it would be handy to develop some kind of parser (even a very small and simple one) to come up with an acceptable and elegant solution. In fact, for one of my customers I defined a description language to automate the extension of a web application.

To show an example of the final result of a dynamically generated DAL using the DALComp application lets assume we have a database table called Articles. For this table the DAL (see section "DALC Description Language" below) respectively the build provider will automatically create a class called Article, containing private member fields and public poperties that correspond to the table column names. Nullable type are created for value types.

In addition the system generates static methods (also defined within a .dal file) to select the requested data. The data are returned as a generic list of type Article (in C# List<Article>) and can be used as follows:

Sample 1:

```
foreach(Article article in Article.SelectAll())
  Console.WriteLine(article.Title);
```

Sample 2:

```
ArticlesGridView.DataSource=Article.SelectAll();
ArticlesGridView.DataBind();
```

Sample 3:

```
<asp:ObjectDataSource ID="ArticlesDS" TypeName="Parago.DAL.Article" SelectMethod="SelectAll"
  runat="server" />
```

So, now lets start!

Build Providers

Build providers are new to ASP.NET and the .NET framework version 2.0. They basically allow you to integrate yourself within the ASP.NET compilation process and build environment. That means you can define a new file type for which you can generate source code based on arbitrary file content. The source code (provided for instance as a CodeCompileUnit) will then be built into the final compiled website assemblies. In our case, we will define a new file type .dal and the file content will be of type Text containing our own little description language to define DALCs.

In fact, the ASP.NET framework is basically doing the exact same thing for file types like .apsx and .ascx as well as for many more. The corresponding build providers are defined in the global web.config configuration file. For instance the file type .apsx is handled by a framework class called PageBuildProvider. If you are interested in how the ASP.NET-Team has implemented this provider, you can either use ILDASM or Lutz Roeder's ".NET Reflector" to disassemble the code.

To use build providers in your web applications, you have to activate new file types within the local web.config. The file type .dal for the DALComp application is defined in the configuration file as follows:

```
<compilation>
  <buildProviders>
    <add extension=".dal" type="Parago.DALComp.DALCompBuildProvider, DALComp.BuildProvider"/>
  </buildProviders>
</compilation>
```

The class DALCompBuildProvider is handling from now on all files with the extension .dal. The class extends the abstract base class BuildProvider and overrides the GenerateCode method. ASP.NET is calling this method during the compilation and build process of the website and is passing an instance of type AssemblyBuilder to the method. Code can then be added by calling the AddCodeCompileUnit method of the AssemblyBuilder.

CodeCompileUnits represent containers for CodeDOM program graphs. Basically they are an internal image of the source code. Each .NET language that supports the code provider model can create source code in its own language based on a CodeCompileUnit.

Creating a clean language-independent CodeDOM program graph is an annoying and somewhat cumbersome task. You have to create it if you want to generate source code for different languages. The CodeGen class for DALComp generates DAL source code for currently C# and VB.NET.

The BuildProvider class also provides a method called OpenReader to read the source code (a file with the extension .dal). The next steps are to tokenize, parse and generate a CodeDOM program graph which we can turn over to the ASP.NET build process:

```
Tokenizer tokenizer=new Tokenizer(source);
Parser parser=new Parser(tokenizer);
CodeGen codeGen=new CodeGen(parser);

builder.AddCodeCompileUnit(this, codeGen.Generate());
```

In the next section we first of all take a look at a sample source code to see what kind of language we want to analyze and generate code for.

DALC Description Language

The description "language" to formally describe DAL components uses a very simple syntax. The following shows a sample DAL definition contained in the file Sample.dal (stored in the special folder App_Code):

```
Config {
  Namespace = "Parago.DAL",
  DatabaseType = "MSSQL",
  ConnectionString = "Data Source=.\SQLEXPRESS;…"
}

//
// DAL component for table Articles
//
DALC Article ( = Articles ) {
  Mapping {                          // Map just the following fields, leave others
    ArticleID => Id,
    Text1 => Text
  }
  SelectAll()
  SelectByAuthor(string name[CreatedBy])
  SelectByCategory(int category[Category])
}

DALC Category( = "Categories" ) {
  SelectAll()
}
```

The syntax for the language is defined using the extended Backus-Naur form (EBNF), which is an extension of the basic Backus-Naur form (BNF) metasyntax notation, a formal way to describe languages. The following syntax-rules illustrate the definiton of the DALC description language:

```
digit
  = "0-9"
letter
  = "A-Za-z"
identifier
  = letter { letter | digit }
string
  = '"' string-character { string-character } '"'
string-character
  = ANY-CHARACTER-EXCEPT-QUOTE | '""'
dal
  = config dalc { dalc }
config
  = "Config" "{" config-setting { "," config-setting } "}"
config-setting
  = ( "Namespace" | "DatabaseType" | "Connectionstring" ) "=" string
dalc
  = "DALC" identifier [ dalc-table ] "{" [ dalc-mapping ] dalc-function { dalc-function } "}"
dalc-table
  = "(" "=" ( identifier | string ) ")"
dalc-mapping
  = "Mapping" "{" dalc-mapping-field { "," dalc-mapping-field } "}"
dalc-mapping-field
  = ( identifier | string ) "=>" identifier
dalc-function
  = identifier "(" [ dalc-function-parameter-list ] ")"
dalc-function-parameter-list
  = dalc-function-parameter { "," dalc-function-parameter }
dalc-function-parameter
  = ( "string" | "int" ) identifier "[" identifier | string "]"
```

The next section explains how to scan and parse the syntax above shown.

Compiler Techniques

The DALComp application is using compiler techniques in a very basic kind. Implementing a real-world compiler can be a very complicated task. It involves techniques like syntax error recovering, variable scoping or bytecode (IL) generation and many more.

The first step in the implementaion is to create a tokenizer. A tokenizer is analyzing the input character by character and is trying to split it into so-called tokens. Tokens are categorized blocks of texts. The categories may be language keywords like the C# loop statement "for", comparison operators like "==" or whitespaces. DALComp is defining a class named Token to represent a single token as well as an enumeration called TokenKind to define the categories of tokens:

```
public enum TokenKind {
        KeywordConfig,
        KeywordDALC,
        KeywordMapping,
        Type,
        Identifier,
        String,
        Assign,               // =>
        Equal,                // =
        Comma,                // ,
        BracketOpen,          // [
        BracketClose,         // ]
        CurlyBracketOpen,     // {
        CurlyBracketClose,    // }
        ParenthesisOpen,      // (
        ParenthesisClose,     // )
        EOT                   // End Of Text
}

public class Token {

        public TokenKind Type;
        public string Value;

        public Token(TokenKind type) {
```

```
                Type=type;
                Value=null;
        }

        public Token(TokenKind type, string value) {
                Type=type;
                Value=value;
        }

}
```

The class Tokenizer is doing the tokenizing by analyzing character by character of the input stream. The Tokenizer constructor initializes the object instance with the input text to scan, creates a generic queue of type Token and calls the method Start to do the work:

```
public Tokenizer(string text) {

  // To avoid index overflow append new line character to text
  this.text=(text==null?String.Empty:text)+"\n";

  // Create token queue (first-in, first-out)
  tokens=new Queue<Token>();

  // Tokenize the text!
  Start();

}
```

The constructor will also add an additional character ("\ n") to the input text to avoid an index overflow. The Start method looks similar to the folowing:

```
void Start() {

  int i=0;

  // Iterate through input text
  while(i<text.Length) {

    // Analyze next character and may be the following series of characters
    switch(text[i]) {

      // Ignore whitespaces
      case '\n':
      case '\r':
      case '\t':
      case ' ':
        break;

      // Comment (until end of line)
      case '/':
        if(text[i+1]=='/')
          while(text[++i]!='\n') ;
            continue;

      …

      case '{':
         tokens.Enqueue(new Token(TokenKind.CurlyBracketOpen));
         break;

      case '}':
         tokens.Enqueue(new Token(TokenKind.CurlyBracketClose));
         break;

      // '=' or '=>'
      case '=':
        if(text[i+1]=='>') {
          i++;
          tokens.Enqueue(new Token(TokenKind.Assign));
        }
        else
          tokens.Enqueue(new Token(TokenKind.Equal));
        break;

      …
```

As you can see it is simple and straightforward to implement a tokenizer. There two more methods of the Tokenizer class, PeekTokenType to lookahead for the next token type in the queue and GetNextToken to actually return the next token from the queue (also removes the token from the queue):

```
public TokenKind PeekTokenType() {
  // Always return at least TokenKind.EOT
  return (tokens.Count>0)?tokens.Peek().Type:TokenKind.EOT;
}

public Token GetNextToken() {
  // Always return at least Token of type TokenKind.EOT
  return (tokens.Count>0)?tokens.Dequeue():new Token(TokenKind.EOT);
}
```

Both methods are called from the parser, the next step in compiling source code. Parsing is the process of analyzing the sequence of tokens in order to determine its grammatical structure with respect to a given formal grammar. An instance of the Tokenizer class will be handed over to the Parser class. The Parser class is producing a structure that is representing a semantically correct source code, called an abstract syntax tree (AST). The name is used by mistake, since the structure is not a tree of any kind. The AST used in this context is just a generic list of DALC class objects and settings (the Config part).

The Parser class implementation is also simple and straightforward. There is no time for me to explain in detail how parsing and the concepts behind are working. This is a huge area of computing. The source code for the Parser class is self-explanatory and easy to understand.

The parser is basically just implementing the syntax-rules defined above by the extended Backus-Naur form, see also the section "DALC Description Language":

```
/// <summary>
/// dal = config dalc { dalc }
/// </summary>
void ParseDAL() {
  ParseConfig();
  do {
    ParseDALC();
  } while(Taste(TokenKind.KeywordDALC));
  Eat(TokenKind.EOT);
}

/// <summary>
/// config = "Config" "{" config-setting { "," config-setting } "}"
/// </summary>
void ParseConfig() {
  Eat(TokenKind.KeywordConfig);
  Eat(TokenKind.CurlyBracketOpen);
  ParseConfigSetting();
  while(true) {
    if(!Taste(TokenKind.Comma))
      break;
    Eat();
    ParseConfigSetting();
  }
  Eat(TokenKind.CurlyBracketClose);
  …
}
```

The parsing methods are using helper functions to "eat" the tokens of the queue by calling the mentioned method GetNextToken of the Tokenizer class or simply aborting the parsing process. Here an example:

```
/// <summary>
/// Looks ahead the token line and returns the next token type.
/// </summary>
bool Taste(TokenKind type) {
  return tokenizer.PeekTokenType()==type;
}

/// <summary>
/// Returns the next token.
/// </summary>
string Eat() {
  token=tokenizer.GetNextToken();
  return token.Value;
}
```

```
/// <summary>
/// Returns the next token of type, otherwise aborts.
/// </summary>
string Eat(TokenKind type) {
  token=tokenizer.GetNextToken();
  if(token.Type!=type)
    Abort();
  return token.Value;
}

/// <summary>
/// Returns the next token of any of the passed array of types, otherwise aborts.
/// </summary>
string EatAny(TokenKind[] types) {
  token=tokenizer.GetNextToken();
  foreach(TokenKind type in types)
    if(token.Type==type)
      return token.Value;
  Abort();
  return String.Empty;
}
```

The third phase is using the structure generated by the Parser class and transforms it into a CodeDOM structure that can be used to create C# or VB code. This phase is called the code generation phase. Language compilers targeting the .NET framework are usually generating Intermediate Language (IL) code. The DALComp application is not generating any IL code, instead it generates a CodeDOM graph that can be used e.g. by ASP.NET to compile into web assemblies.

The method Generate of the CodeGen class is creating first of all a container for the CodeDOM structure, is adding a namespace unit to it and tries to connect to the defined database using the connection string that is specified within the DAL defintion file (see Sample.dal):

```
// Create container for CodeDOM program graph
CodeCompileUnit compileUnit=new CodeCompileUnit();

try {

  // If applicable replace the value '|BaseDirectory|' with the current
  // directory of the running assembly (within the connection string)
  // to allow database access in DALComp.Test.Console
  string connectionString=dal.Settings["CONNECTIONSTRING"]
    .Replace("|BaseDirectory|", Directory.GetCurrentDirectory());

  // Define new namespace (Config:Namespace)
  CodeNamespace namespaceUnit=new CodeNamespace(dal.Settings["NAMESPACE"]);
  compileUnit.Namespaces.Add(namespaceUnit);

  // Define necessary imports
  namespaceUnit.Imports.Add(new CodeNamespaceImport("System"));
  namespaceUnit.Imports.Add(new CodeNamespaceImport("System.Collections.Generic"));
  namespaceUnit.Imports.Add(new CodeNamespaceImport("System.Data"));
  namespaceUnit.Imports.Add(new CodeNamespaceImport("System.Data.SqlClient"));

  // Generate private member fields (to save public property values)
  // by analyzing the database table which is defined for the DALC
  SqlConnection connection=new SqlConnection(connectionString);
  connection.Open();

  // Generate a new public accessable class for each DALC definition
  // with all defined methods
  foreach(DALC dalc in dal.DALCs) {

    // Generate new DALC class type and add to own namespace
    CodeTypeDeclaration typeUnit=new CodeTypeDeclaration(dalc.Name);
    namespaceUnit.Types.Add(typeUnit);

    // Generate public empty constructor method
    CodeConstructor constructor=new CodeConstructor();
    constructor.Attributes=MemberAttributes.Public;
    typeUnit.Members.Add(constructor);

    // Get schema table with column defintions for the current DALC table
    DataSet schema=new DataSet();
    new SqlDataAdapter(String.Format("SELECT * FROM {0}", dalc.Table), connection)
        .FillSchema(schema, SchemaType.Mapped, dalc.Table);
```

```
    // Generate for each column a private member field and a public
    // accessable property to use
    foreach(DataColumn column in schema.Tables[0].Columns) {

      // Define names by checking DALC mapping definition
      string name=column.ColumnName;
      string nameMapped=
        dalc.Mapping.ContainsKey(name.ToUpper())?dalc.Mapping[name.ToUpper()]:name;

      // Generate private member field with underscore plus name; define
      // member field type by checking if value type and create a
      // nullable of that type accordingly
      CodeMemberField field=new CodeMemberField();
      field.Name=String.Format("_{0}", nameMapped);
      field.Type=GenerateFieldTypeReference(column.DataType);
      typeUnit.Members.Add(field);

      // Generate public accessable property for private member field,
      // to use for instance in conjunction with ObjectDataSource
      CodeMemberProperty property=new CodeMemberProperty();
      property.Name=nameMapped;
      property.Type=GenerateFieldTypeReference(column.DataType);
      property.Attributes=MemberAttributes.Public;
      property.GetStatements.Add(
        new CodeMethodReturnStatement(
          new CodeFieldReferenceExpression(
            new CodeThisReferenceExpression(),
            field.Name
          )
        )
      );
      property.SetStatements.Add(
        new CodeAssignStatement(
          new CodeFieldReferenceExpression(
            new CodeThisReferenceExpression(),
            field.Name
          ),
          new CodePropertySetValueReferenceExpression()
        )
      );
      typeUnit.Members.Add(property);

    }

    …

  }

}
```

Based on the DAL specification file the method reads in each schema table of a DALC table, builds up a new class type and adds all columns as private member fields and public properties to it. If a column data type is a value type then it will create a nullable version of that value type as follows:

```
CodeTypeReference GenerateFieldTypeReference(Type columnType) {

  // If column data type is not a value type return just return it
  if(!columnType.IsValueType)
    return new CodeTypeReference(columnType);

  // Type is a value type, generate a nullable type and return that
  Type nullableType=typeof(Nullable<>);
  return new CodeTypeReference(nullableType.MakeGenericType(new Type[] { columnType }));

}
```

For example if a column is of type “int”, this helper function will generate a “int?” or “System.Nullable<int>”. Here a sample of auto-generated C# code:

```
public class Article {

  private System.Nullable<int> _Id;
  private string _Title;
  private string _Text;
  private string _Text2;
```

```
  private string _Language;
  private System.Nullable<int> _Category;
  private string _CreatedBy;
  private System.Nullable<System.DateTime> _CreatedOn;

  public Article() {
  }

  public virtual System.Nullable<int> Id {
    get {
      return this._Id;
    }
    set {
      this._Id = value;
    }
  }

  …
  // Helper method to query data
  public static List<Article> SelectData(string sql) {
    List<Article> result;
    result = new List<Article>();
    System.Data.SqlClient.SqlConnection connection;
    System.Data.SqlClient.SqlCommand command;
    System.Data.SqlClient.SqlDataReader reader;
    connection = new System.Data.SqlClient.SqlConnection("Data Source=...");
    connection.Open();
    command = new System.Data.SqlClient.SqlCommand(sql, connection);
    reader = command.ExecuteReader();
    for (; reader.Read(); ) {
      Article o;
      o = new Article();
      if (Convert.IsDBNull(reader["ArticleID"])) {
        o.Id = null;
      }
      else {
        o.Id = ((System.Nullable<int>)(reader["ArticleID"]));
      }
      …
      result.Add(o);
    }
    reader.Close();
    connection.Close();
    return result;
  }

  // DALC function
  public static List<Article> SelectAll() {
    string internalSql;
    internalSql = "SELECT * FROM Articles";
    return SelectData(internalSql);
  }

  …

}
```

For more detail information please refer to source code.

Summary

The DAL itself is a basic implementation and shows the concepts of creating dynamic code. To be accurate, the DALComp compiler is actually more a source-to-source translator then a compiler. The current version is only generating methods to select data, no updates or inserts. As you can see there is plenty of room for extending the DAL by augmenting the description language and generating more dynamic code to make the DAL productive.

For more information regarding building compilers and virtual maschines, I recommend Pat Terry's "Compiling with C# and Java" book. Another way to study compiler techniques in practise is to take a look at the sources of the .NET implementation of Python, IronPython. The source code is available as download on the CodePlex website.

For real-world compiler development there are plenty of utility tools available such as Coco/ R, a scanner and parser generator, or the ANTLR compiler tools (used by the Boo compiler). You also find a lot of information on the web site of Microsoft Research, e.g. the F# compiler. Another interesting topic is the Phalanger project ("The PHP Language Compiler for the .NET Framework") on CodePlex.com.

The source code of DALComp is available for download on my homepage.

Contact Information

If you have any feedback or suggestions, please feel free to contact me:

Jürgen Bäurle
jbaurle@parago.de
http://www.parago.de/jbaurle

Parago Media GmbH & Co. KG
Im Wengert 3 | 71336 Waiblingen, Germany | Phone +49.7146.861803 | Internet http://www.parago.de