Metadata Driven Code Generation Using .NET Framework

Ivo Damyanov Nick Holmes

Abstract: *In this paper incorporating manual and automatic code generation is discussed. A solution for automatic metadata-driven code generation is presented illustrated with multi tier Enterprise Resource Planning System. We intend to make our solution available to public in order to encourage investigation of code generation and schema-driven tools for .NET Framework.*
Key words: *Metadata, Custom Attributes, .NET Framework, Code Generation.*

INTRODUCTION

The software is embodiment of a system's design goals. Changes in business requirements reflect on changes in goals. Software development needs to be a fast process. Software technologies rapidly evolve and improve. Recently application development has successfully incorporated automatic code generation. Modern integrated development environments start to provide useful wizards and code generators. Many researchers propose different approaches of incorporating manual and automatic code generation. Automatic code generation for design patterns has been observed in [2], frame oriented programming is investigated in [3, 8, 10]. In this paper we present our approach using Microsoft .NET Framework. The .NET framework is a programming model for developing, deploying, and running XML Web services that targets all types of applications [1]. In fact, .NET framework is a rather radical approach to make the development of Windows more component-oriented [9]. The .NET framework enables building open applications. Every .NET-aware language is compiled to intermediate language code which is executed in Common Language Runtime (CLR). Custom attributes are one of the most innovative features of the .NET framework [7]. They allow everyone to define information which can be applied to classes, methods, parameters, etc.

We divide the development process in steps; an incremental development approach. On each step we use metadata provided from the previous step. As a significant innovation Custom Attributes first appear in .NET Framework. As a way to create metadata accessible at runtime using reflection we found this very useful for reducing effort of code writing and making the process of automatic code generation much easier.

We illustrate our approach with a multi-tier Enterprise Resource Planning System (ERP). In our solution we use different code generator models such as *mixed code generators, partial class generators* and *tier generators* [4]. As a result of the whole development process we get 100% automatically generated stored procedures and triggers (DB tier), 90% of data access layer (DAL) and what we call ether tier and 100% of the client facade.

CODE GENERATOR MODELS AND METADATA SOURCES

Let us first recall some definitions of different code generation models that we use in our solution.

Mixed Code Generators take source code files as input, modify the contents and replace the original file. We use mixed code generator to partially update of data access layer (see figure 1) to reflect changes in database schema or stored procedures. To enable this we have introduced source code meta-tags that define connections between data access layer code and the database.

Partial class generators generate a set of base classes to implement specified designs. We use partial class generators in the so called *ether tier*, which is code that is shared between the business logic layer and the client facade. Mainly we create here

some validation and recalculation methods which were overridden, if necessary, in derived classes. The actual benefits of using ether tier are reducing round trips to the server and in-depth validation to ensure data correctness.

Tier generators build and maintain an entire tier within the application. We use tier generators in two stages – creation of the data tier (stored procedures and triggers) and in the generation of the client facade.

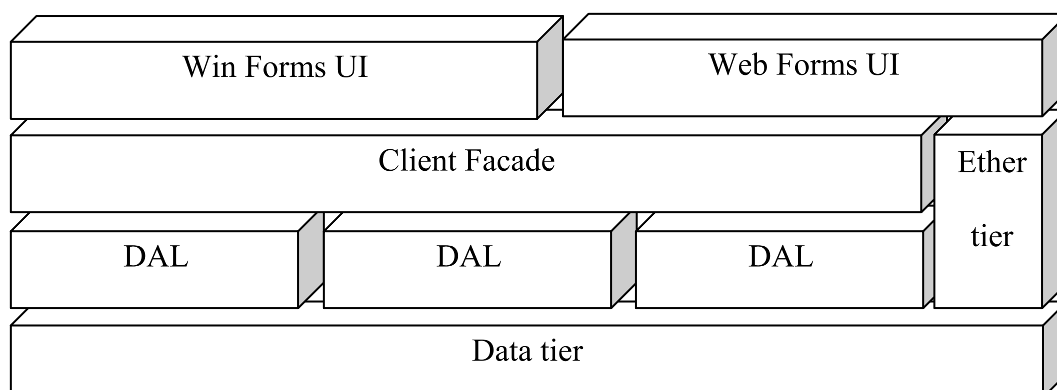

Figure 1
Multi tier model of Enterprise Resource Planning System

The code generator's work is driven by metadata. For the data tier code generator, an essence of UML model information is stored in a simple XML model file (figure 2), which is used in generation process.

DAL code generation use schema information queried from the database system catalogs, as well as model XML file.

Automatic generation of the client facade relies on knowledge of provided and required services of a software component. Traditionally they are provided in terms of signatures; however this information is not enough to give a detailed understanding of component interactions, underlying database or security issues.

Custom attributes are part of the metadata of every managed module. The framework provides functionality to gather information from the metadata of loaded modules. This mechanism is known as *reflection* [6]. The DAL Code generator sets necessary metadata for client facade generator using .NET custom attributes. The Client facade code generator queries modules and extracts this data using reflection.

One of the major doctrines of software engineering is to keep objects loosely coupled. With the many classes and subsystems we use, it is important to isolate our software. The *Facade Pattern* [5] is intended to provide a unified interface to a set of interfaces in a subsystem. The Facade defines a higher-level interface that makes the subsystems easier to use. There are several benefits to Facade. First, it provides developers with a common interface to the subsystem, leading to more uniform code. Second, it isolates your system and provides a layer of protection from complexities in your subsystems as they evolve. Third, this protection makes it easier to replace one subsystem with another because the dependencies are isolated.

The main disadvantage of Facade is that it tends to limit your flexibility. You are free, however, to instantiate objects directly from the subsystems.

The facade pattern is not so often applied to time critical software development. It is very important to provide developers with ability to "write" the facade tier together with development of system backend – which we achieve in our approach.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT model (table)+>
<!ELEMENT table (list*, foreignKey*,
                        validation*, dataSet*)>
<!ATTLIST table
        name CDATA #IMPLIED
        logEnabled CDATA #IMPLIED
        hasCompactDS CDATA #IMPLIED
        updateRequiresEntireCollection
                CDATA #IMPLIED
        allowOnlyParentUpdates
                CDATA #IMPLIED>
<!ELEMENT list (additionalFilter*, parameter*)>
<!ATTLIST list
        name CDATA #IMPLIED
        manualSP CDATA #IMPLIED
        dataSet CDATA #IMPLIED>
<!ELEMENT foreignKey (filter?)>
<!ATTLIST foreignKey
        table CDATA #IMPLIED
        buildAsCollection CDATA #IMPLIED
        dataSet CDATA #IMPLIED
        name CDATA #IMPLIED
        foreignAttribute CDATA #IMPLIED>
<!ELEMENT validation (col*)>
<!ELEMENT dataSet (column*)>
<!ATTLIST dataSet
        name CDATA #IMPLIED
        viewBased CDATA #IMPLIED>
<!ELEMENT additionalFilter (#PCDATA)>
<!ATTLIST additionalFilter
        op CDATA #IMPLIED>
<!ELEMENT parameter EMPTY>
<!ATTLIST parameter
        name CDATA #IMPLIED
        type CDATA #IMPLIED>
<!ELEMENT filter (#PCDATA)>
<!ELEMENT col EMPTY>
<!ATTLIST col
        name CDATA #IMPLIED
        minLength CDATA #IMPLIED
        minValue CDATA #IMPLIED
        maxValue CDATA #IMPLIED>
<!ELEMENT column EMPTY>
<!ATTLIST column
        name CDATA #IMPLIED>
```

Figure 2

DTD of model XML file used for Data tier code generation

CODE GENERATION PROCESS

The primary measure of success for a software project is delivering a bug free, maintainable application in a timely manner within an allotted budget. An analysis done in the design stage of Enterprise Resource Planning System (ERP) has led us to implementation of set of code generators.

ERP is a multi-layer application that uses components as data access layer (DAL). DAL is simple stored procedure invoker that runs as serviced components (COM+). We designed DAL to be stateless. Several benefits come from stateless DAL. With stateless COM+ objects we can have Just-in-time activation (to keep server overload low), application level transactions and good scalability.

There is one to one mapping between objects (tables) in the database and components in DAL. Requested information was returned from DAL as Datasets. Even using Typed Datasets the provided interface to the frond end developers is complex. The major weakness is dealing with relational objects. For example dealing with Product information we need to be able to reach full Manufacturer information without going through a second query based on Products.ManufacturerFK or using a JOIN query to gather all the information at once. That's why we use a Client Facade pattern to simplify access to relational business objects. We hide querying Manufacturer information behind dot notation, i.e. we have on a client facade classes that wraps DataRows and provides methods and properties to access related objects easily.

The following figure illustrates metadata streams and process steps of ERP development (manual and automatic code generation).

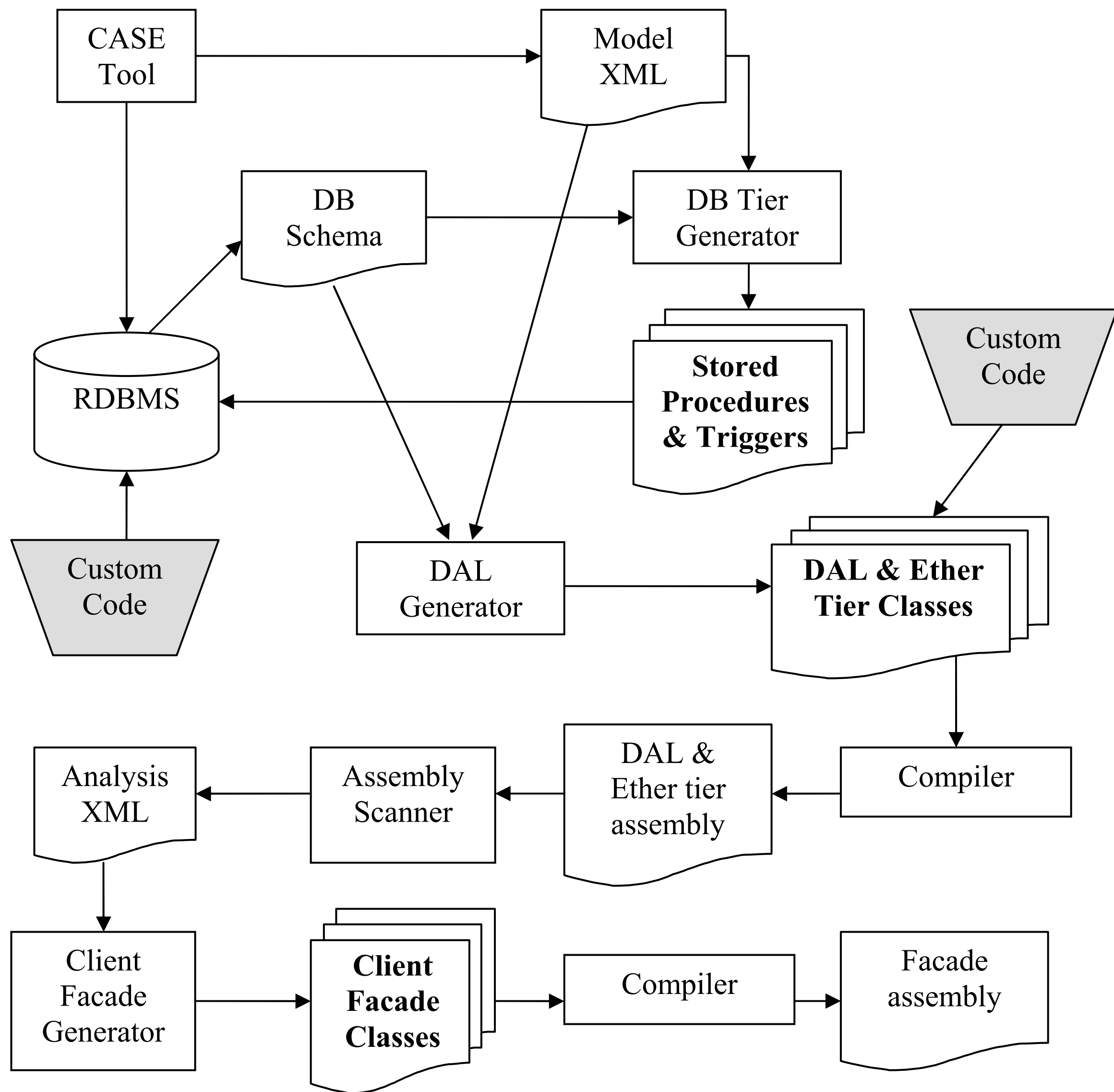

Figure 3
Code generation process

Using reflection we can query DAL assemblies and discover classes and their interfaces. What is missing is some additional information like database relations, type of the class methods, i.e. Primary selector, Update/Insert or Delete, collection selector (which will act on client facade as static methods because will return more than one row usually). To provide this extra information a set of 9 custom attributes was defined.

Assembly level attributes:

- *ClientFacadeNamespace* – defines namespace used for client facade code

Class level attributes:

- *ClientFacade* – determine exposing class behavior via client facade
- *ClientFacadeUnderlyingDatabaseTable* – define underlying DB table
- *ClientFacadeClassProperty* – set properties that will be exposed
- *ClientFacadeAllowOnlyParentUpdates* – in parent-child related objects determine who will do the update.

Method level attributes:

- *ClientFacade* – determine exposing method via client facade
- *ClientFacadePrimarySelector* – mark method as primary selector

- *ClientFacadeUpdate* – mark method as update/insert/delete method
- *ClientFacadeStatic* – mark method as client facade static (i.e. return list rather single object)

Each of them defines structural or behavioral metadata. This metadata are extracted and stored in intermediate XML file which is used in next stage – client facade generation. DTD of this intermediate file is represented bellow.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT assemblies (assembly)+>
<!ELEMENT assembly (path?, type*)>
<!ATTLIST assembly
   clientFacadeNamespace CDATA #IMPLIED
>
<!ELEMENT path (#PCDATA)>
<!ELEMENT type (propertiesAsEnum?,
        foreignKeys*, allowConstructionFrom*,
        methods*,  typedDataset*,  properties*,
        primaryKeys*)>
<!ATTLIST type
        name CDATA #IMPLIED
        namespace CDATA #IMPLIED
        underlyingTable CDATA #IMPLIED
        createDeleteMethod CDATA #IMPLIED
        allowOnlyParentUpdates CDATA
                #IMPLIED>
<!ELEMENT propertiesAsEnum (#PCDATA)>
<!ELEMENT foreignKeys (relation*)>
<!ELEMENT allowConstructionFrom
        (constructionType*)>
<!ELEMENT methods (method*)>
<!ELEMENT typedDataset (table*)>
<!ATTLIST typedDataset
        type CDATA #IMPLIED>
<!ELEMENT properties (property*)>
<!ATTLIST properties
        rowType CDATA #IMPLIED>
<!ELEMENT primaryKeys (primaryKey*)>
<!ELEMENT relation EMPTY>
<!ATTLIST relation
        fkColumnName CDATA #IMPLIED
        pkColumnName CDATA #IMPLIED
        pkTableName CDATA #IMPLIED>
<!ELEMENT constructionType EMPTY>
<!ATTLIST constructionType
        name CDATA #IMPLIED
        namespace CDATA #IMPLIED>
<!ELEMENT method (params*)>
<!ATTLIST method
        name CDATA #IMPLIED
        serverName CDATA #IMPLIED
        update CDATA #IMPLIED
        getChanges CDATA #IMPLIED
        returns CDATA #IMPLIED
        primarySelector CDATA #IMPLIED
        static CDATA #IMPLIED>
<!ELEMENT table EMPTY>
<!ATTLIST table
        type CDATA #IMPLIED
        namespace CDATA #IMPLIED
        master CDATA #IMPLIED
        serverObject CDATA #IMPLIED>
<!ELEMENT property EMPTY>
<!ATTLIST property
        name CDATA #IMPLIED
        type CDATA #IMPLIED
        allowNulls CDATA #IMPLIED
        defaultProperty CDATA #IMPLIED
        formatting CDATA #IMPLIED
        initValue CDATA #IMPLIED>
<!ELEMENT primaryKey EMPTY>
<!ATTLIST primaryKey
        name CDATA #IMPLIED
        type CDATA #IMPLIED
        autoGenerated CDATA #IMPLIED>
<!ELEMENT params (param*)>
<!ELEMENT param EMPTY>
<!ATTLIST param
        name CDATA #IMPLIED
        type CDATA #IMPLIED
        serializedDataSet CDATA #IMPLIED>
```

Figure 4
DTD of assembly analysis file

Another significant feature in .NET Framework is the System.CodeDom namespace. The System.CodeDom namespace defines an abstract language. This language has no defined concrete representation; a program in the language is represented by a tree of CodeDom objects, which corresponds to a parse tree in a compiler for a conventional language. The System.CodeDom.Compiler namespaces defines classes and interfaces for using CodeDom trees. A CodeDom generator is a translator/compiler which converts the abstract program represented by a CodeDom tree into some concrete language; such as

C#, Visual Basic, or JSharp. The interface ICodeGenerator defines the methods for converting CodeDom trees to concrete language code. We exploit these facilities to generate our Client Facade code.

CONCLUSIONS AND FUTURE WORK

In this paper we present a consistent approach used to generate software components such as stored procedures, triggers, data access layer and facade wrappers. Using a rich set of mixed code generators, partial class generators and tier generators absorbing data model changes is mitigated in most cases to a very minor issue, preserving manually written custom code.

.NET Framework provides rich set of language extensions (like Custom Attributes) and namespaces (like System.CodeDom) that make possible automatic code generation. Our approach was proved by developing a complete ERP system with: entirely auto generated client facade – about 36,000+ lines of C# code, partially generated (90%) Server and Ether tier code – about 66,800+ lines of C# code and about 11,800+ lines of SQL code (stored procedures and triggers). We have used DB Schema and .NET reflection mechanism to gather the necessary information.

As a fairly new technology .NET Framework has some bottlenecks. One of them is inability of CodeDom to load source code and produce Document Object Mode. We think that resolving this CodeDom will become a very powerful approach for automatic code generation and manipulation.

REFERENCES

[1] An Introduction to Microsoft .NET. White Paper, Microsoft Corporation, 2001.

[2] Burginsky, F., M. Finnie, P. Yu, Automatic Code Generation from Design Patterns, IBM Systems Journal, Volume 35, #2, 1996.

[3] Bassett, P., Framing software reuse – lessons from real world, Yourdon Press, Prentice Hall, 1997.

[4] Code Generation Network – http://www.codegeneration.net

[5] Gamma, E., R. Helm, R. Johnson, J. Vilssides, Design Patterns: Elements of Reusable Object Oriented Software, Addison Wesley Inc., 1998

[6] Lumpe, M., On the representation and use of Metadata, Second International Workshop on Composition Languages, Málaga, Spain, June 2002

[7] Richter, J., Applied Microsoft .NET Framework Programming, Microsoft Press, 2002.

[8] Sauer, F., Metadata driven multi-artifact code generation using Frame Oriented Programming, OOPSLA 2002.

[9] Troelsen, A., C# and the .NET Platform, Apress, 2001.

[10] Wong, T., S. Jarzabek, M. Soe, R. Shen, H. Zhang, XML Implementation of Frame Processor, Symposium of Software Reusability, SSR'01, Toronto, Canada, May 2001, pp. 164-172.

ABOUT THE AUTHORS

Ivo Damyanov, Department of Computer Sciences, South-West University, Blagoevgrad, Phone: +359 73 889 132, E-mail: damianov@aix.swu.bg.

Nick J. Holmes, Coyote Software GmbH, Luxembourg, E-mail: nickh@coyote-software.com.