Type-safe data binding on modern object-oriented platforms

István Albert

Budapest University of Technology and Economics
Department of Automation and Applied Informatics
H-1111 Budapest, Goldmann György tér 3, Hungary
E-mail: ialbert@aut.bme.hu

ABSTRACT

Most object-oriented platforms support run-time type information to provide access to class members like fields and methods. These solutions are often based on strings, textual names of types and members. Such approach makes the systems very fragile and sensitive to modification of names and to other changes. This paper illustrates an elegant and highly efficient solution for this problem which is also type-safe thanks to compile-time type checking. The introduced new language construct supports access to class members through multiple parameterized one-to-many associations. It can also be used in many languages and platforms which makes it an ideal candidate to be used in real world systems.

Keywords

Programming Tools and Languages, reflection, association, data binding, C#.

1. INTRODUCTION

Today's most wide-spread and most heavily used programming paradigm is object-oriented paradigm with imperative languages, like C++, Java or C# [8, 9, 10]. While the core concepts are quite solid, there are numerous possible ways to improve the quality of software. There are several current techniques to customize this approach. In C++ language, environment macros and templates [12] are heavily used constructs. Java and .NET are introducing generics [11, 18, 1, 16, 14, 7] (a kind of template implementation for parameterized types) and we are well aware of Design by Contract [4], as well as aspect-oriented approach and other extremely useful concepts. Many of these, although still under research, are leaking into the world of applied software technology [19].

One of the main goals of these enhancements is to make the language and environment more type-safe which would result in more stable applications with less run-time errors.

This paper introduces an elegant and efficient way to use typed reflection and so type-safe data binding.

The next two sections introduce reflection and data-binding. After getting familiar with the problem, a new language construct called navigation expression is introduced. Its features are discussed in detail, including multiple associations. The next section compares navigation expressions with a similar concept of delegates. Finally, an implementation plan is suggested and a formal definition of C# language changes is also proposed in the appendix.

.NET Technologies'2005 *conference proceedings,*
ISBN 80-86943-01-1

2. RELATED WORK

There are reflection scenarios where programs use strings to identify type members like methods and fields. In some cases a more type-safe method can be used. One of these is data binding on the CLI platform.

Reflection: Accessing Type Information at Run-time

Reflection mechanism provides objects that encapsulate modules, types, methods, fields, etc [6]. With these constructs a program can examine the structure of types, create instance of types, and

invoke methods, access fields and properties. Similar language and virtual machine support exists in the Java platform [5] (reflection API); it is called RTTI (Run-Time Type Information) in C++ [12]. According to the current C++ Standard [12], RTTI has far less features than the Java or .NET implementations: only type names, type equality and inheritance hierarchy can be determined at run-time, but no method list, method invocation, object creation, field access, etc. are allowed. But there are some currently researched theories and proof-of-concept implementations of a full-fledged reflection mechanism API in C++ [20, 21].

These solutions are based on string literals to refer to member variables or methods. This highly flexible approach is necessary but makes the systems very fragile and sensitive to modification of names.

This paper illustrates an extension to the current reflection models which could be very useful in certain scenarios. We are using the "data binding" scenario throughout this paper to analyze the problem and the way the new language construct solves it.

Introduction to Data Binding

Consider the following example: we have a generic component that displays data, and a program that uses this component. The configuration of the component determines which data is to be displayed; it also defines its format. The data to be displayed is called *data source* and is provided by the application. After configuring the component and *binding* it to a data source the application uses it to show the data to the user.

This concept is called *data binding* in .NET and it is very flexible and frequently used. Here is an example:

```
public class DataVisualizer {
    public object DataSource;
    public string DataMember;
    public void Render() {
        Console.WriteLine( DataSource.GetType().
      GetField( DataMember ).GetValue( DataSource ) );
} }
public class Person { public string Name; }

public class MyApp {
    public static void Main() {
        DataVisualizer vis = new DataVisualizer();
        Person p = new Person();
        p.Name = "Stephen Albert";
        vis.DataSource = p;
        vis.DataMember = "Name";
        vis.Render();
} }
```

Figure 1

The Render method uses reflection to extract data from the data source object (an instance of the Person class) which is based on DataMember holding a textual reference to the Name field of the Person class.

Although it may not be a good idea to use strings to identify members, there are many examples where this flexibility is quite useful. Reflection is often used by generic frameworks and algorithms where type information is not known or cannot be expressed at compile-time. The most well-known platform feature which uses reflection is serialization [6, 17]. During this process an entire graph of objects is written to a stream or created from a stream. Other typical frameworks using this technology are object persistency layers (both in J2EE and .NET [7, 13, 2]), workflow engines, data access layers or data binding components. This paper uses the data binding as an illustration but the idea can be used in many other frameworks as well. The samples are in C# on the .NET platform but the main concept can be easily transferred to another language or platform.

Open Problems

The problem with string based member access is twofold. Since it uses strings, it is very easy to make a typographic error (1), which is mostly discovered only at run-time when Render() method is called (2).

The reason for the errors also seems to be twofold. Firstly, the programmer could misspell the string and give a wrong identifier, hence the reflection mechanism cannot find the appropriate member by name. This causes a *run-time error*.

Secondly, there can be a type mismatch between DataSource and DataMember: the first one is the object which is being read, the second one is the expression which refers to a member. If the DataSource is an object without a "Name" field, it also causes a *run-time error*. This paper addresses both issues.

With a suitable language construct the programmer can get a *compile-time error* which is preferred to *run-time error* [15, 22].

1. NAVIGATION EXPRESSIONS

The main purpose of DataMember is to traverse the object hierarchy graph along associations and to provide access to member variables (which can be fields or properties). DataSource is the root of the object graph. The example in Figure 1 shows only one hop, but certainly it can take more hops to get to the target member. A new language construct called *navigation* is defined in the next sample as follows (Figure 2):

```
public sealed class DataVisualizer {
    public Navigation DataSource;
    public void Render() {
```

```
        Console.WriteLine(  DataSource.ToString() ); } }
public class MyApp {  public static void Main() {
        DataVisualizer vis = new DataVisualizer();
        Person p = new Person();
        p.Name = "Steve Albert";
        vis.DataSource = new Navigation( p.Name );
        vis.Render(); } }
```

Figure 2

Navigation construct aggregates data source and data member in one object and provides a run-time evaluation of the expression with type safety.

Navigation instance has a strict root type at which the traversal begins – in this case class Person. It contains a dot-separated list of association names – type members. The object graph is traversed through these associations.

The navigation expression can be not only in the right side of an equation, but in left side as well – it can be an lvalue – which makes it possible to use bi-directional data binding. In this case the expression is used to set field and property values.

Fields, Properties, Indexers

A referenced type member can be a field, a property or an indexer. Properties are named groups of accessor method definitions that implement the named property [6,23]. Indexers are parameterized properties. The properties enable field-like access, whereas indexers enable array-like access [3].

Multiple Associations

In many cases an association refers to multiple objects and navigation expression must support it. To be able to navigate through one-to-many associations, parameters should be passed to the navigation object at all those points where collections of objects are referenced.

A one-to-many association must be an array or an indexer (parameterized property), a technique widely used in the CLI platform [6].

Each association may have zero or more parameters, depending on its type. Field and property accessors have no parameter at all, arrays have as many signed integer parameters as the rank of the array, and also indexers can have any number of parameters of any type.

The parameter list of the navigation expression is the concatenation of those parameters and can be derived by examining a particular navigation expression and the referenced members. Since indexers can be overloaded with different parameter lists [6, 23], one expression can actually refer to more than one parameter list. Expressions must also contain named parameters with types for unambiguous member traversal.

A short sample for using navigations with one-to-many associations (Figure 3):

```
... string [] myStrings = new string [] { "a", "ab", "abc" };
NavigationArray nav1 = new NavigationArray(
                                myStrings[int].Length);
for( int i = 0; i < myStrings.Length; i++ )
Console.WriteLine(myStrings[i]+':'+nav1[i].ToString());
```

Figure 3

In the above sample (Figure 3) a navigation object is constructed with a string array being the root object. This refers to multiple strings and, for usability, an additional parameter should be supplied to choose from the collection of referenced strings. In this particular case only one parameter is

necessary: a signed integer. In a more complex case more parameters could be used.

Cast operators

This version of navigation construct does not support casting members. This will be discussed in a separate paper. Navigation expression must be in pure format of member names separated by dots, with optional parameter lists like in Figure 4.

```
// compiles, no parameters
root.Member1.Member2
// compiles, with parameters
list[int].Column[string, State].Member
// does not compile with cast operator
((DataColumn) root.Member1).Member2
```

Figure 4

Root object ambiguity

The root of navigation expressions could be ambiguous for object member access. Examining the first code expression in Figure 4, the root object (the root of the path) could be a reference to "root" or "root.Member1" (both are references). To avoid this situation, navigation expressions always use the first object reference as root reference.

These syntax rules ensure that navigation is not an expression evaluated at run-time but rather a compile-time appearance of the object hierarchy path.

2. NAVIGATION TYPE DEFINERS

Reflection is most often used when type information of parameters and objects is not known at compile-time but can be acquired at run-time. In this way the component and the application development can be totally separated, which is crucial for generic frameworks and scenarios like data binding. Although strict type information is not known, the way an object is handled is very often hardcoded in the component.

For example a component that displays matrix data uses data source as a two-dimensional array. A component which displays a table uses data source as a list and each column refers to a specific data member. In these scenarios the data source must satisfy the demands of the component, preferably checked at compile-time.

To support this requirement, navigation expressions are strictly typed.

The component that uses the navigation as a data source determines the parameters and also the return type of the expression. The type declarations for *Navigation* and *NavigationArray* with respect to the above samples (Figure 2 and Figure 3) are as follows:

```
navigation object Navigation;
navigation int NavigationArray( int i );
```

Figure 5

Navigation declaration and instantiation with navigation expression are depicted in Figure 6. However, a more formal definition can be found in Appendix A: Formal C# language definition:

```
navigation type TypeName( formal-parameter-list );
TypeName var = new TypeName(
                        navigation-expression );
```

Figure 6

These types are generated automatically by the compiler from the navigation declaration. Variables of these types can only hold a reference to navigation instances which have the same number of parameters and the type of each parameter is the same or inherited from the appropriate type in the navigation declaration. The return type expression must also match the type in the declaration with equality or inheritance.

In this way the component can safely use data source which conforms to its requirements and forms a matrix, a list, etc. A client application is verified *at compile-time* to check whether it supports the appropriate data source with type safe member references.

All this results in a type-safe data binding.

Inheritance and Access Modifiers

The type where the navigation object is created must have access to the referenced members. Private fields, properties, indexes can be used only when the class itself declares a navigation to its own members. Protected members can be used in derived classes, internal members [6] in the same compilation unit (assembly) accordingly. Public members can be used anywhere.

A navigation type declaration can be public, internal, protected or private just like a class declaration. These modifiers define the visibility of the navigation type just like class visibility does. Once a navigation type is instantiated, it can be used by any class. If a method of class A receives a navigation object as a parameter, the method can use it to access the referenced member independently of whether class A has access to the member referenced by the expression or not.

The compiler checks, for all but the last of properties and fields and indexers in the association list, whether they are readable and all are accessible by the declaring class which creates the navigation object. No write-only members are allowed through the association path except the last one. An expression is read-only if the last member is a read-only member for the instantiating class, write-only if it is write-only, and normal otherwise.

Comparison to Delegates

In CLI delegates are used as "object-oriented type-safe function pointers" [6, 3]. They share common ideas with navigation expressions. In both cases a special language element is used for type definer which allows type-safety by identifying methods to invoke or members to be accessed later. The syntax is quite similar, too [23, 3] (Figure 7):

```
void PrintInt( int i ) { Console.WriteLine( i );  }
delegate void MethodDelegate( int a );
MethodDelegate del =
        new MethodDelegate( this.PrintInt );
del( 42 );
navigation int myNavigation( int );
string [] myStrings = new string [] { "a", "ab", "abc" };
myNavigation nav1 = new myNavigation(
                                myStrings[int].Length );
Console.WriteLine( nav1[2] );
```

Figure 7

The difference between the language constructs is that the delegates are applicable to methods but not to fields or properties (even though properties are implemented as methods in CIL). Moreover, delegates do not support navigation in the object hierarchy; they only have a reference to a class instance and a handle referencing a method of that type. Navigations hold an entire reference path to navigate through the object hierarchy and reach the addressed field or property through multiple associations. Data binding on .NET platform uses properties and not methods for member access. Hence in that case delegates are not applicable and cannot be used for data binding.

3. COMPILER IMPLEMENTATION

A "compiler only" solution can be provided if only one language is taken into consideration. After checking syntax (see Section 5) and type consistency the compiler generates extra code in place of *declaration*, *instantiation* and *usage* (see Appendix B).

Each *navigation declaration* is a type creator syntax element (similar to *class*, *interface*, *delegate* and *array sign* ('[ ]') [6, 23, 3]). The abstract type (*class A*) is constructed by the compiler and is unique for each navigation declaration. For each object hierarchy path, a unique class (*class B* : A) is generated by the compiler which finally derives from type created for navigation declaration. *Class A* contains two abstract methods for reading and writing members (GetValue and SetValue methods). Parameter lists are generated according to the navigation declaration. Derived *Class B* provides implementation for these abstract methods, using strict type information.

Using reflection, dynamic navigation creation can also be supported but it is not recommended, since it ensures no type safety at all. In this scenario a program can create navigation expression instances at runtime, based on strings.

To measure performance impact we have modified the Mono C# compiler. The compiler-generated type safe navigation expressions are 10 to 50 times faster than a reference solution with reflection.

The advantage of this "compiler only" approach is that the runtime environment remains unchanged. Only language compilers should be extended to provide the new functionality. Similarly to delegates, a navigation declaration is also a type declaration and this type could be a basis for language interoperability which is essential on the CLI platform.

4. CONCLUSION

In this paper we have introduced a new C# language construct that provides more type-safe solution with compile-time errors rather than run-time errors. The new language construct called *navigation* supports access to class members through multiple parameterized one-to-many associations and similarly to delegates, a navigation declaration is also a type declaration.

This solution is not only more type-safe but can also provide a huge gain of performance in many application scenarios.

5. APPENDIX A : FORMAL C# LANGUAGE DEFINITION

The following list is the extension to C# language grammar [23, Appendix A].

A.1.7 Keywords, Keyword: `navigation`

A.2.2 Types

Reference type: *navigation-type*

Navigation-type: *type-name*

A.2.4 Expressions

Primary-no-array-creation-expression:
navigation-creation-expression

Expression: navigation-creation-expression:
`new` *navigation-type* (*navigation-expression*)

Navigation-expression:

expression

navigation-expression . *identifier*

navigation-expression . *identifier* [*type-list*]

Type-list: *type* | *type-list* , *type*

A.2.5. Statements

Type-declaration: navigation-declaration

A.2.13. Navigations, navigation-declaration:
$attributes_{opt}$ $navigation\text{-}modifiers_{opt}$
`navigation` *type identifier(* $fixed\text{-}parameters_{ot}$ *)*

navigation-modifiers:

navigation-modifier

navigation-modifiers navigation-modifier

navigation-modifier:

`new | public | protected | internal | private`

6. APPENDIX B : ILLUSTRATION OF THE COMPILER GENERATED CODE

Navigation declaration:

```
public navigation string gridNavigation(
                                int row, int column );
```

Generated code:

```
public abstract class gridNavigation
:BaseNavigation
{
    public abstract void SetValue(
                    int row, int column, string value );
    public abstract string GetValue(
                                int row, int column );
}
```

Navigation instantiation:

```
string [][] birthData = new string [][] { new string [] {
      "Blaise Pascal", "1623-1662", "Clermont" },
    new string [] {
   "Sir Isaac Newton", "1642-1727", "Woolsthorpe" },…
}; …
gridNavigation nav1 = new gridNavigation(
                              birthData [int][int] );
```

Generated code:

```
… public sealed class gridNavigation_nav1 :
                                    gridNavigation {
    String [][] rootObj;
    public myNavigation_1( string [][] root )
    {
        rootObj = root; }
    public override string GetValue(
                              int row, int column) {
        return rootObj[row][column]; }
    public override void SetValue(
                int row, int column, string value) {
        rootObj[row][column] = value;
} }
…gridNavigation nav1 =
                new gridNavigation_nav1( birthData );
```

Navigation usage:

```
public class DataGrid {
    public gridNavigation DataSource;
    public int RowNumber, ColumnNumber;
    public void Render() {
        for( int r = 0; r < RowNumber; r++ ) {
            for( int c = 0; c < ColumnNumber; c++ ) {
                Console.Write( DataSource[r, c] );
                if( c < ColumnNumber – 1 )
                    Console.Write( ", " ); }
            Console.WriteLine(); } } }
```

Generated code:

```
…  for( int c = 0; c < ColumnNumber; c++ ) {
            Console.Write(DataSource.GetValue( r, c ) );
        if( c < ColumnNumber – 1 )Console.Write( ", " );
} …
```

7. REFERENCES

[1] A. Kennedy and D. Syme.: Design and implementation of generics for the .NET Common Language Runtime. ACM SIGPLAN, PLDI, pages 1–12, Snowbird, Utah, June 2001.

[2] A. Homer, D. Sussman, M. Fussell: First Look at ADO.NET and System.Xml v.2.0 (Addison Wesley, 2003)

[3] A. Hejlsberg, S. Wiltamuth, P. Golde, The C# Programming Language (Addison Wesley, 2003)

[4] B. Meyer, Eiffel: the language (Prentice Hall, New York, NY, first edition, 1992)

[5] B. Joy, G. Steele, J. Gosling, G. Bracha, The Java Language Specification, Second Edition (Addison-Wesley, 2000)

[6] ECMA-335 Common Language Infrastructure (CLI), ECMA, December 2001. http://www.ecma.ch/

[7] ECMA TC39-TG2/2004/14, C# Language Specification, Working Draft 2.7, Jun, 2004

[8] Gartner Inc. (Michael J. Blechar): The Impact of Web Services Architecture on Application Development, 26 August 2002

[9] Gartner Inc.: J2EE and .NET Will Vie for E-Business Development Efforts, 28 April 2003

[10] Gartner Inc. (Joseph Feiman): The Gartner Programming Language Survey, 1 October 2001

[11] G. Bracha, M. Odersky, D. Stoutamire: Making the future safe for the past: Adding genericity to the programming language. OOPSLA, ACM, Oct. 1998.

[12] International Standard: Programming Languages - C++. ISO/IEC. 2003. Number 14882:2003 (E) in ASC X3, ANSI, New York, NY, USA.

[13] JSR 12: JavaTM Data Objects (JDO) Specification. http://jcp.org/en/jsr/detail?id=12, 2003

[14] JSR-14, Add Generic Types To The Java Programming Language. Available on line at http://jcp.org/en/jsr/detail?id=014, 2004

[15] K. B. Bruce. Typing in object-oriented languages: Achieving expressibility and safety. Technical report, Williams College, 1996.

[16] M. Lucia Barron-Estrada, R. Stansifer: A Comparison of Generics in Java and C#, 41st ACM Southeast Regional Conference, March 2003

[17] M, Hericko, M, B. Juric, I. Rozman, S. Beloglavec, A. Zivkovic, Object serialization analysis and comparison in Java and .NET., SIGPLAN Notices 38(8): 44-54 (2003)

[18] O. Agesen, S. Frølund, and J. C. Mitchell.: Adding parameterized types to Java. In Object-Oriented Programming: Systems, Languages, Applications, pages 215-230. ACM, 1997.

[19] R. Bruce Findler, M. Latendresse, M. Felleisen, Behavioral contracts and behavioral subtyping, ACM SIGSOFT Software Engineering Notes, Volume 26 , Issue 5, September 2001

[20] S. Chiba, A Metaobject Protocol for C++, ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, 1995

[21] S. Roiser, Reflection in C++, CERN, February 2004 (Available on-line at http://doc.cern.ch//archive/electronic/cern/others/LHB/internal/lhcb-2003-116.pdf)

[22] R. Finkel: Advanced Programming Language Design (Addison Wesley, 1995)

[23] Standard ECMA-334 C# Language Specification, ECMA, December 2001. Available on-line at http://www.ecma.ch/