An Approach to the Metadata Driven Programming in .Net Framework

Pavel Abduramanov, Sergey Kalistratov and Yuri Okulovsky
Institute of Mathematics and Computer Science, Ural Federal University, Turgeneva 4, Yekaterinburg, Russia

Keywords: Programming Patterns, Data Driven Approach, .Net Framework.

Abstract: We describe a metadata driven approach to development of stereotyped business accounting software. We modify the model-view-controller pattern by placing all the application's logic into a model, and automatically building controllers and views. The work has two essential parts. The first one is developing a way to define metadata, that can be altered in runtime, can depend on context and can store actions as well as data. The second is designing of the software as a data server, which stores a model and its metadata, and modify it by requests from various clients, e.g. web pages or windows applications. This approach was implemented in the Thornado framework and was used for creation of various applications. We justify the easiness and elegance of our implementation of the metadata driven development, and discuss advantages of the approach, such as cross-platformness, scalability and testability.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

In this article we discuss an original approach to rapid development of a special kind of software, data manipulation systems (DMS). The primary aim of such systems is to collect various data and then use them to generate reports. The typical example of DMS is accounting systems, which store information about employees and clients, schedule events, handle budgets, route documents for internal approvement, print agreements, and so on. Such accounting systems are the main consideration for us, although other software types can fall into the DMS category as well.

We define two characteristics of DMS. The first is a complex business logic for data manipulation, which may vary significantly even among the enterprises of the same field, due to differences in the enterprises' business process. This is the most distinctive peculiarity of DMS, since it brings the requirement to the development process. While creating DMS, it is especially important to rapid develop and then implement the entangled business model, and be able to further improve it. The second characteristic is an absence of an interactive and intelligent graphic user interface. Typically, such systems only require an access to data manipulation, without any "whistles and bells".

We consider the implementation of DMS in .NET framework. This framework provides many convenient features for the modern development, such as LINQ, delegates, generics, etc. .NET framework provides ASP.NET MVC framework for web development, and Windows Presentation Foundation (WPF) for Windows programming. Considering the general problem of an universal framework that could be used for creation of any possible application, it is probably hard to obtain a significant improvement in comparison with these two. However, in the particular cases, when we only consider applications, which follow some pattern, the peculiarities of the pattern can be exploited to get an approach for even more effective development of such stereotyped software.

In our case of DMS, the main source of the improvements lays in analysis of Model-View-Controller pattern (Krausner and Pope, 1988) in ASP.NET MVC, and its descendant Model-View-ViewModel (Smith, 2009) in WPF. In general case all three components of Model-View-Controller pattern play important roles in the software. But in DMS, the attention's focus shifts significantly to the model. Controllers in DMS only call methods from the model. Due to the absence of the rich GUI, views are stereotyped and can be generated from a model.

Both ASP.NET MVC and WPF provide a basic solution for such stereotyped views through metadata driven development (MDDD). In this paradigm, metadata are associated with models' fields by the C# attributes, and then are used to generate views. However, this approach has a limited applicability. In some cases, the developer needs to change metadata

Abduramanov P., Kalistratov S. and Okulovsky Y..
An Approach to the Metadata Driven Programming in .Net Framework.
DOI: 10.5220/0004584102390243
In *Proceedings of the 8th International Joint Conference on Software Technologies* (ICSOFT-EA-2013), pages 239-243
ISBN: 978-989-8565-68-6
Copyright © 2013 SCITEPRESS (Science and Technology Publications, Lda.)

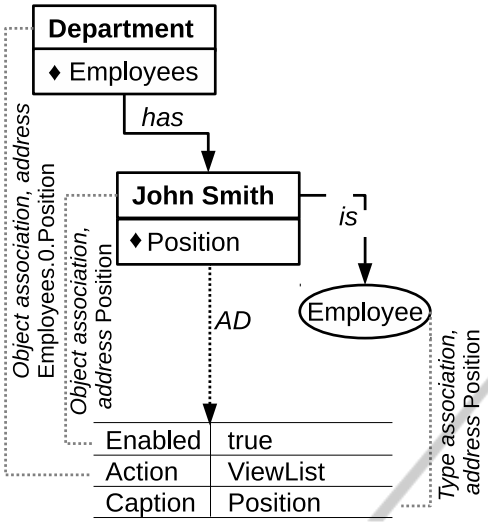

Figure 1: An example of various associated data.

in the runtime, to provide, for example, a different visibility of fields depending on the runtime-known circumstances, such as the user's role. But the attributes' values cannot be changed in runtime, since they are stored in the assembly, and therefore the developer need to switch from the MDDD to other methods. In section 1, we provide a description for the extended metadata, which we call *associated data* (AD). AD can be changed in the runtime, it is possible to attach different sets of AD to an object depending on its context, and methods can be stored in AD as well as primitive values.

In section 2, we discuss the architecture of an application as a *data server*, which interacts with the environment via requests and replies. The data server contains a model and its associated data, including the information about methods, available for call. Data server acts as a controller. It analyzes the packages, sets the model's fields and calls its methods. Then it provides a current state of the model so the view can be automatically generated.

This approach is implemented in the Thornado library, and in section 3 we discuss our experience about it, and its main advantages. Namely, the approach allows creation of cross platform software: the very same data servers can be used to create windows- and web-based interfaces without any additional efforts. Our approach is also testable and scalable. Minor disadvantages of the approach are also discussed.

2 ASSOCIATED DATA

This section explains associated data (AD), the most fundamental feature of the Thornado framework. AD are small parts of data, binded to objects' fields. Consider an example of `Employee` class with `Position` string property on Figure 1. We may want associate the `Enabled` flag with the property `Employee.Position`, which indicates whether the field is editable by the user. Since the `string` type does not provide any means to do that, the flag should be represented as associated data. Another example of associated data is the `Caption`, which corresponds to the explanation of the field's meaning and is rendered in the Label control or the column's header in the DataGrid.

In many cases, AD is shared between all instances of a class. For example, `Caption` AD is unlikely to be changed for some particular employee. Therefore, `Caption` should be associated with the type `Employee`, and therefore is a classic metadata. The type, however, may have many properties, and we should specify the address of the property we are working with. So, the definition of the caption for the `Position` field is associating a string value with the type `Employee` and the address `Position`. This association is performed by the hashtable with corresponding keys. Association can be established by instructions, placed in the static constructor or other methods, or C# attributes.

Some AD varies from one object to another. For the security reasons, editing of the `Position` property may be turned off in some application's areas, but enabled in others. Therefore, `Enabled` AD cannot be associated with the type `Employee`, and should be bound to the particular `Employee` object.

Not only Boolean or string values can be associated with a property, but arbitrary C# objects. The important special case is associating action with fields. For example, a button can be placed near the `Position` text box, which shows a dialog box and allows picking a position from a list. Since actions can be decoupled from methods as delegates, they can be represented as AD as well.

Data may be associated with properties of a class, or properties of properties, or with arbitrary long addresses, or with every objects in a collection: for example, if several `Employee` objects are placed in a list, we may associate the same `Remove` action with all of them. Since there are several sources of associatied data, conflicts are possible, and the way of aggregating multiple AD values into one needs to be defined. The logic of the aggregation depends on the type of AD: captions cannot be aggregated and therefore the last defined caption should be chosen to display, while actions are to be accumulated and displayed as a stack of buttons. In the case of `Enabled` field, more complex logic may be used: the field may be declared en-

abled by default, which can be provided by the type-assotiation, but later overriden for the particular object by assotiation of the corresponded AD with this. Therefore, the priority of the AD should be defined: for example, object assotiation is almost always more important that the type assotiation.

Our implementation of AD which guarantees correctness of usage in compilation time. Consider the following example of AD settings:

```
employee.Set(e=>e.Position,
             AD.Caption,"text")}
```

Here the address is specified by LINQ Expression, and it is impossible to assotiate data with a non-existing field. The name of AD is provided via singleton object `AD.Caption`, which also defines the type of the stored value. The type of the third `"text"` argument is defined by `AD.Caption`, and therefore guarantees the type correctess of the value that is being associated. This effect is reached by the extensive use of .NET generic-methods and LINQ expressions. Type-associated data can also be defined in attributes, and such definitions are placed into hashtable with other AD at the first time the type is used. `AD.Caption` also defines the logic of AD aggregation.

Summarizing, the introduced metadata system is:

1. Flexible. It allows storing actions in metadata, which is vital to the procedure of metadata-driven validation. It also allows changing metadata in runtime, and defining context-dependent metadata. All these features are new in comparison with traditional .NET metadata.
2. Type-safe, and discloses more errors in the compilation time in comparison with storing metadata value in hash-tables.
3. Expandable, because the addition of new AD requires only the definition of a singleton object, similar to `AD.Caption`

3 DATA SERVER

This section describes the development of a data-driven application in the Thornado framework. The most important part of the application is a model, which represents data. We demand a firm correspondence between a hierarchy of objects in model, and hierarchy of controls in a page. That means we must provide a data structures for `MainWindow` and other controls. Such structures play the role of a view model in MVVM pattern. For example, `MainWindow` is a small class with one property `Content`, which represents a top-level control in a window. To display data in `MainWindow`, the `Content` property should be assigned to some object, and so various methods like `ShowEmployee` or `ShowDepartments` are defined in `MainWindow` inheritant and are bound with it by AD. Other auxiliary classes, like `StackPanel`, `Grid` and `MasterDetailTemplate`, are also provided by the Thornado framework. It is important that these classes *are* data structures, *are not* widgets, and *do not* have any references to windows, HTML code builders or other platform-specific code.

The data structures can be displayed in the `MainWindow` simply by setting `Content` property to, for example, `Employee` or `List<Employee>` object. The way how data structures are displayed are defined in various ADs, including `Caption` and `Enabled`. The data server (DS) provides the interaction between data structures and the user, by acting like a controller and creating views with the aid of the associated data.

Data server follows the algorithm, shown in the Figure 2. At the first call DS goes through the tree of objects in the model, gathers all the additional data and prepares a tree-like `NEW` reply with a detailed description of each node in the GUI. The reply is passed to the Client, which actually builds and processes graphic user interface. Depending on the client's type, it may create WPF widgets and subscribe to their events, or send an HTML page and then reroute Ajax requests from the page to the data server in a proper format.

The most important requests are `SET` and and `PUSH`. The `SET` request contains an address of a field and a new value. The data server processes the request by settings the field to the received value, and calling the notifications delegates associated with the field, as well as validation prodecures that may be defined in the model. The `PUSH` request contains the address of an element and a name of the button, which was associated with the element. After receiving a `PUSH` request, the DS invokes the corresponding delegate from an object at the specified address. Methods, called by the data server, alter the model and AD according to the business logic. After the methods are completed, DS creates a `UPD` reply, which contains the updated values for changed nodes.

If the invoked method requires a dialog window or a message box to be shown, it calls the corresponding method from the data server. In this case, the server forms a `NEW` reply that corresponds to the new window and processes events of the window. Execution of the old window's methods are suppressed, and the control will return to them when the dialog box is closed.

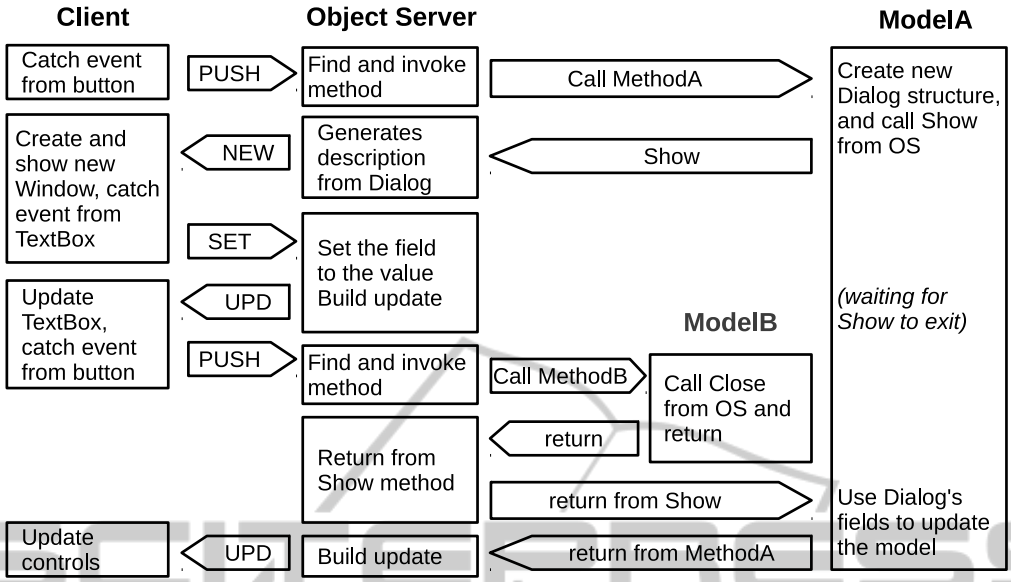

Figure 2: Workflow in a data-driven Thornado application.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

4 DISCUSSION ON THORNADO FRAMEWORK

The described approach was implemented and used for the development of circa 40 small accounting systems. By our observation, the data-driven framework seems very logical for developers. They also find it very simple: this is a pure C# programming, without involvement of XAML, Razor, HTML or other languages, and does not require any considerations about HTTP requests, event models or structural patterns. The only thing that needs to be created is the business-logic.

Thornado framework is cross-platform, because the model does not contain any references to platform-specific libraries. When the model is created, it can be used with various clients, and therefore presented as a windows application, a web application or an application for mobile device. Moreover, the model can be represented as a SOA service (Bell, 2009), which accepts request and sends replies as XML packages. That makes Thornado applications scalable, because the services can be distributed on multiple servers behind the gate server that just keeps a correspondence between users and services, and redirects packages to appropriate service.

Testability is another important feature of Thornado application. It is easy to create a client, which sends a test requests and checks features in the received replies. Therefore, testing scenario that emulates user input can be developed in addition to usual unit tests.

Thornado may be used for the prototyping of interfaces, since we should develop only data structures to get a preliminary view of the final application. This prototype can later evolve in the application itself, which is impossible when different frameworks are used for the prototyping and the development.

Of course, Thornado framework also has disadvantages, the primary being the limited graphic user interface. Inside the program, the developer does not have access to the GUI descriptors, and therefore cannot make use of many of the controls' features. In case of need to access the specific feature, the develope may introduse new AD into the framework and then update the client to make it process the new AD. We should stress, however, that the Thornado framework is not designed for the applications with particularly complex and intelligent interfaces. We should also note, that in some cases limitation to the GUI is in fact an advantage that forces developers to use the consistent user interface through the all application.

Let us briefly consider the applicability of the presented approach outside of the .NET Framework. Data servers does not bring any limitations to the programming environment, and can be implemented in any object-oriented language. However, the data servers rely on the additional data. AD can be implemented in other languages. However, LINQ Expressions, which are used to define addresses of associated data, is missed in many object-oriented languages. When implemented, for example, in Java, the

addresses will probably have to be defined as string constants. That would significantly reduces the value of the AD, because its correctness could not be established in the compilation time. Also, in the absence of the delegates, storing actions in metadata will require writing a class for each action, which is not as comfortable as using delegates. These inconveniences would probably make AD so hard and verbose, that they would be near to useless.

5 CONCLUSIONS

At this point, all the described ideas are implemented and tested. Thornado framework also includes numerous routine features, like serialization and deserialization in human-readable XML and INI formats, simple ORM over the disconnected ADO.NET layer, and so on. We have also implemented many stereotyped subsystems, such as:

1. Events' scheduling, typically used in software for educational centers;
2. Documents routing, which propagates documents through the chain of approving persons;
3. Budgeting subsystem, which allows planning estimated incomes and expenses, and track the actual ones.

This justifies the viability of the described approach, and its applicability to DMS development.

The research is generally finished. There are some issues with AD conflicts: when several values are bound to the same field of the same class, is it sometimes hard to decide, how exactly they should be aggregated. We try to use different strategies, for example, different schemes of priorities, or timestamps to define which value was assotiated later or earlier than others. Another potential way of research is a general approach to generic containers in the objects' hierarchy, which presents some technical challenges. However, these branches of research are unlikely to be applicable outside of the Thornado framwork.

We hope, however, that the ideas of the framework will find their ways in other developing software. The described approach to the metadata definition may appear useful for other rapid application development frameworks, as well in the other areas that require metdata handling. We also believe that the presented way to decouple the business model and the graphic user interface via data server can be applied in cases, when several GUI types are working with the same model, and therefore there is a need for a coherent way to access the data.

ACKNOWLEDGEMENTS

The work is partially supported by the Fund for Assistance to Small Innovative Enterprises in Science.

REFERENCES

Bell, M. (2009). *SOA Modeling Patterns for Service-Oriented Discovery and Analysis*. Wiley & Sons.

Krausner, G. E. and Pope, S. T. (1988). A description of the model-view-controller user interface paradigm in the smalltalk-80 system. Technical report, ParcPlace Systems, Inc.

Smith, J. (2009). Wpf apps with the model-view-viewmodel design pattern. *MSDN Magazine*.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS