Aspect.NET: Aspect-Oriented Programming for Microsoft.NET in Practice

Aspect-oriented programming for .NET

BY
VLADIMIR SAFONOV &

DMITRY GRIGORIEV

This article is the third part of the series of articles in this journal to describe our Aspect.NET project – an aspect-oriented programming (AOP) framework for Microsoft.NET based on a number of new ideas (for the first two articles, see #4 and 5 in the References section). This article analyzes the advantages of using AOP, presents the current status of Aspect.NET – its design principles, its working version – and gives some practical examples.

The evolution of software and the customers' for it, as well as the needs of the everyday fixing, update, and enhancement of software often lead to poorly understandable and tangled code, with fragments of implementation of quite different functionalities unclearly intermixed in the same software modules.

The AOP approach allows you to define the implementation of crosscutting concerns in separate modules and to describe the approach's weaving into the target code by high-level metalanguage. In particular, this relates to our Aspect.NET project.

In this article we'll demonstrate how Aspect.NET helps to develop well-structured and clearly implemented applications, using the power of AOP as part of Visual Studio.NET 2005 (Whidbey).

First we'll consider the general principles of the AOP technology, then we'll describe the design and implementation principles of Aspect.NET, and finally we'll show the advantages of Aspect.NET by way of an example of developing a software system of order processing.

AOP Background

AOP is one of the most prospective approaches among state-of-the-art software technologies. This technology offers simple and elegant ways of developing complicated software systems. Such systems usually consist of the code to implement business logic and the code that implements a variety of the other crosscutting concerns such as transaction mechanisms, security checks, profiling, and tracing. Crosscutting concerns penetrate several layers of abstraction and appear to be intermixed with each other and with the basic business logic code. Object-oriented programming allows you to clearly define the concepts of the problem domain, but for solving most of the tasks combining several objects and updating several abstract layers is required. For example (see #1 in the References section), to add to an operating system the new functionality of prefetching data from memory, it should be implemented at the virtual memory layer, file system layer (local and remote), and disk layer, and runtime context information on prefetching should be passed between those abstract layers. Design and code patterns afford the ability to combine several objects at the same abstract layer for solving some task. However first, to apply them one should manually make changes to the existing classes, since the patterns cannot be automatically extracted and reapplied to some other projects. Second, the implementation of patterns penetrates a number of modules, which are crosscutting concerns themselves.

Aspect-oriented programming allows you to define the implementation of a crosscutting concern as a single module (referred to as an aspect), together with the rules of their weaving into the target code – *pointcuts*, and with the *actions* to be inserted into some definite points of the target code – *joinpoints*. It helps to both better understand the functionality implemented as an aspect definition (since all of the aspect's code is located in one place), and to make it reusable (since, in order to weave the aspect into different joinpoints, it is enough to change its pointcuts).

It should be noted that AOP does not replace OOP, but that it complements it in quite a natural way. The implementation of patterns by means of AOP increases their efficiency (see #2 in the References section). All of the AOP systems follow this general approach. Currently the most common AOP tools are Java oriented, and AOP implementations for Microsoft.NET are still at the experimental stage.

The most widespread and powerful AOP tool is AspectJ (www.aspectj.org) implemented as an extension of the Java programming language by aspect definition metalanguage. For implementing such an approach, all of the architecture should be developed from scratch and changed every time the language specification is updated As compared to this approach, developing aspect declarations in a separate metalanguage allows for a more flexible way of defining aspect and its weaving rules, and improves readability of aspects.

There are a number of alternative approaches to aspect definition that allow for the declaration of aspects in an already existing language. They are based on aspect declarations represented as attributes (AspectWerkz [http://aspectwerkz.codehaus.org/] for Java, AOP.NET [M. Blackstock. Aspect Weaving with C# and .NET. www.cs.ubc.ca/~michael/publications/AOPNET5.pdf]) or XML files (JBoss AOP [www.jboss.org/products/aop], Spring AOP [www.springframework.org/docs/reference/aop.html] for Java, Weave.NET [www.dsg.cs.tcd.ie/sites/Weave.NET.html]). The main advantage of these approaches is the opportunity to use already existing tools for application development. The shortcoming of aspect representation by attributes is as follows: complicated aspect definitions in this format are more difficult to read and understand. There is an additional shortage of aspect representation by XML files – specification

of aspects is stored separately from their implementation.

Aspect weaving into a target application can be performed *dynamically* (JBoss AOP, Spring AOP, AOP.NET or interceptor approach [see #3 in the References section]) or *statically* (AspectJ, CCC for .NET [Y. Xiong, F. Wan. CCC: An Aspect Oriented Intermediate Language on .NET Platform: www.fit.ac.jp/~zhao/waosd2004/pdf/Xiong.pdf]). In the former case, the virtual machine invokes the appropriate actions of the aspect when the related conditions are satisfied at runtime. In this case, the overhead for aspect weaving can dramatically decrease the performance. Besides, with runtime aspect weaving, the set of joinpoints that can be tracked is somewhat limited and some kinds of events just cannot be caught, e.g., access to or update of some field. On the contrary, with static aspect weaving, the aspects woven and the target application are joined together, at the class file/bytecode level for Java or at assembly/MSIL code level for .NET. In this case, there is actually no overhead related to invoking the actions of the aspects, and, in addition, the result of weaving is more understandable and can be viewed by the user.

Finally, an important reason why AOP tools are not yet widespread for .NET is the lack of adequate GUI in the current tools. For this reason, features of visual debugging and visualizing aspects are lacking also. In their turn, AOP tools for Java are available via a popular integrated development environment – Eclipse.

Now let's consider how the above and a number of other issues are successfully resolved in Aspect.NET (see #4, 5, and 6 in the References section).

Aspect.NET Design Approach

Aspect.NET is a language-agnostic visual environment for developing aspect-oriented applications for Microsoft.NET that was implemented as an add-in to Microsoft Visual Studio.NET 2005 (Whidbey). Also supported is functioning in an SSCLI (Rotor) environment. Using Aspect.NET, the user can define and weave aspects and assess the results of the weaving in his or her projects.

Aspect.NET uses two approaches to aspect definition: AOP metalanguage and attribute specifications. An aspect definition in Aspect.NET is mapped to a special kind of Whidbey project (Aspect) that can be either converted to the aspect's implementation class with the appropriate attribute annotations, or compiled straight to the aspect's assembly. Then, the compiled aspect's assembly is used in aspect weaving.

Detailed description of our AOP meta-language can be found in the References section (4-6). In short, an *aspect definition* consists of:

- *header* (with optional formal *parameters*)
- *data (optional)*
- *modules (classes, methods, functions, etc.)*
- *actions (method calls* or just *statements)*
- *weaving rules* for each action in Aspect.NET metalanguage

Weaving rules (or pointcuts in terms of the more common AOP tools) are suggestions for the weaver pertaining to where and how to join the action to the target application or to its module subject to the weaving operation. For example, the construct

```
%before %call *OrderSystem.
StartOrderProcessing
```

means the following: insert the aspect's action before calling the *StartOrderProcessing* method of the *OrderSystem* class. Listing 1 shows an example of aspect for implementing profiling functionality for an order processing system.

An aspect can use information in the context of a joinpoint to which its action is woven. In the current version, this context information can be passed as arguments to the action. In the example in Listing 1, the string argument *methname* is the full qualified name of the method that the action is applied to, e.g., *Orders.OrderSystem.StartOrderProcessing*. If the second argument of the action is of type *Object*, it denotes a reference to the target object to whose member the action was applied, or *null*, if the member is static. The Aspect.NET tool has the following three main components:

- ***Aspect editor*:** Allows you to add new aspects, browse, and select (or unselect) the potential joinpoints found in the source of a user project (see Figure 1), weave aspects to the compiled target assembly of the project, and visualize the resulting application (see Figure 2).
- ***Weaver*:** Statically weaves the selected aspects to the selected joinpoints of the target assembly. The process of joining the aspect with the target assembly consists of the two phases:
  - *Scanning* – searching the appropriate (potential) join points within the target assembly, based on the weaving rules defined in the aspect. The aspect editor allows the user to browse these potential joinpoints in the source code and agree or disagree with any of them.
  - Weaving the aspect actions to the joinpoints found by the scanning phase and finally clarified by the user.

Our implementation of aspect weaving, instead of using the somewhat limited features of .NET reflection, is based on the state-of-the-art multitargeted compiler development environment Microsoft Phoenix (see Reference #7). Phoenix, in particular, provides powerful and comfortable features of analyzing, updating, and creating .NET assemblies and is considered by Microsoft as a basis for all of its own and third-party oncoming compiler-related projects. Our group was the first outside of Microsoft to start using Phoenix in September 2003. Before Phoenix had been made available within the framework of Microsoft Early Adoption and Academic Programs, a lot of researchers had to develop their own different and limited tools for handling .NET assemblies.

- ***Converters:*** Convert from Aspect.NET AOP metalanguage to attribute annotations (currently implemented for C#.NET and VB.NET).

The process of the user's work with Aspect.NET can be described as follows. Assemblies with aspects are created as the result of compiling classes with AOP attribute annotations or as the result of compiling new special kinds of Whidbey projects (aspect template projects) written in the Aspect.NET metalanguage. Upon loading a user project into Whidbey, by using the Aspect.NET Editor, the assemblies with aspect definitions can be selected to weave into the current project, and the weaver's scanning mode can be activated for finding the potential joinpoints (Figure 1). A mouse click on a joinpoint selects the appropriate point in the project source. To unselect weaving into a joinpoint, it should be unchecked in the list of the

AUTHOR BIOs:

Vladimir Safonov is a professor of computer science and the head of the computer science laboratory at St. Petersburg University. He received his Masters degree in computer science there in 1977 and has been with the university for 27 years. Vladimir has published five books and his research interests include AOP, Microsoft.NET, Java, programming technologies, compilers, and knowledge management.

v_o_safonov@mail.ru

Dmitry Grigoriev received his Masters degree in computer science at St. Petersburg University in 2003, and he is currently attending the university as PhD student. His research interests include aspect-oriented programming and knowledge management.

gridmer@mail.ru

potential joinpoints. After clarifying the set of joinpoints for weaving by the user, the weaving process is invoked by pressing an appropriate button, and the user gets the resulting assembly, with all of the selected aspects woven. The visualization tab provides the visual representation of a module with the aspects woven (Figure 2).

Example of Aspect.NET Use

Let's consider an example of applying Aspect.NET for business software development. We've taken a simple example and will not explain the details of implementation in order to emphasize the advantages of applying AOP at the user level.

Suppose a customer needs to develop a simple software system of processing orders that supports debit and credit operations for the clients. In addition, the system is to collect time statistics of the use of it by all the clients.

In such system, the *Client* class could be designed as shown on the Listing 2. The ordering system of the customer (*OrderSystem*) could use it in the way shown in Listing 3.

Analysis of customer projects has shown that collecting time statistics can be implemented by the total time of the *StartOrderProcessing()* method execution, since that method is responsible for processing orders of the clients and related calculations.

If we apply to the customer projects the *AspectProfiler* aspect (see Listing 1), we'll get the desirable implementation of collecting time statistics (we simplify in this example what may be actually needed in real projects). That aspect prescribes to invoke the *ActionBeforeCall()* and *ActionAfterCall()* actions, accordingly, before and after any call of *StartOrderProcessing()*. On weaving the *AspectProfiler* aspect into the *OrderSystem* customer project as described above, when executing the generated assembly *OrderSystem.exe*, we can get on our console the following output:

```
AspectProfiler: Executing
time of Orders.OrderSystem.
StartOrderProcessing: 4015 mil-
liseconds
Bob: account=500
John: account=-150
```

Suppose the number of customer's clients has increased to such an extent that the ordering system can no longer cope with their service. As the result of analyzing the time statistics, to improve performance, we decide to run each call of the *ProcessOperation()* method of the *Client* class in a separate thread, since it contains time-consuming requests to databases, so their concurrent execution could dramatically improve the performance. To preserve the application logic, on finishing the order processing method, we should wait for all such threads to terminate.

The *AsyncCallAspect* aspect (see Listing 4) can perform this task very well. It keeps the pool of threads started in its *WrapMethod WithAsyncCall()* action and waits for each of them to terminate in its *TotalJoining()* action. The *WrapMethodWithAsyncCall()* action is performed instead of the *ProcessOperation()* method and uses, for invoking the method in a separate thread, the reference to the target object of the *Client* class (e.g., *John* or *Bob*) and the name of the method to be called (e.g., *Orders.Client.ProcessOperation*).

If we now run the generated assembly after weaving the *AspectProfiler* and *AsyncCallAspect* aspects, our output to console will be different:

```
AspectProfiler: Executing
time of Orders.OrderSystem.
StartOrderProcessing: 2012 mil-
liseconds
Bob: account=500
John: account=-150
```

Once the customer noticed that a negative balance is shown on the accounts of his clients he decided to change his credit policy. Now we are to implement a mechanism of transactions. The execution of the *StartOrderProcessing()* method will be considered as a separate transaction that should be rolled back in the result of one of its credit operations *CreditOperation()* because the balance on the client's account is negative. One of the possible solutions of this problem can be weaving two more aspects – *CreditAbilityAspect* and *TransactionalAspect*. The *CreditOverflowAssertion* action of the *CreditAbilityAspect* action is invoked upon finishing *CreditOperation()* and throws an exception in case, as the result if its execution, the *Client.Account* field gets negative. The *AroundTransaction* action of the *TransactionalAspect* aspect is performed instead of the *StartOrderProcessing* method and calls the method inside, after wrapping it into a try/catch block for handling a possible exeption thrown by the credit operation. This action also implements all of the transactional mechanisms, from saving the state of the system before starting a transaction, till rollback whan it failed (see Listing 5).

Please note how the *CreditAbilityAspect* aspect accesses the context of weaving, and and how the *AroundTransaction* action of the *TransactionalAspect* aspect performs the call of the method to which the action is woven.

Now the execution of the resulting

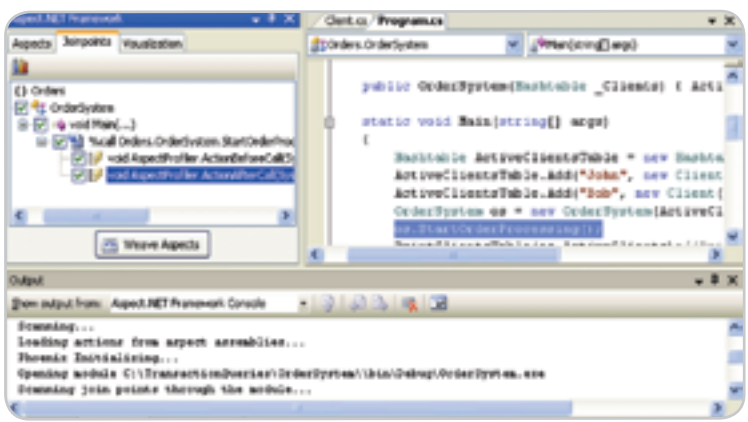


Figure 1: *Aspect.NET at work*

assembly leads to the following result:

```
Rolling back transactions...
AspectProfiler: Executing time of Orders.
OrderSystem.StartOrderProcessing: 2032
milliseconds
Bob: account=250
John: account=100
```

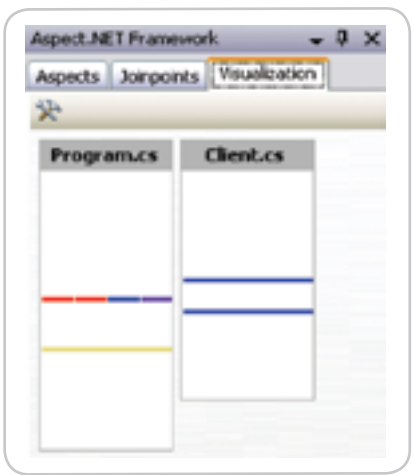
Figure 2 shows the result of weaving of all of the four aspects into *OrderSystem* project. Each aspect is visualized by its own color.

This is just a simple example of how the requirements to a software system can evolve in the process of its development. In this case, more and more requirements were added that hadn't been taken into account in the initial design, and that were not related to the business logic of the application. We managed to take all of these requirements into account without adding any line of code into the initial project that remained as simple as it was and that remained open to all subsequent changes. What we got is a library of aspects that could be successfully applied to other projects, just by clarifying the behavior and changing the weaving rules.

As another example, the design-by-contract software technology (Eiffel: www.eiffel.com) is intended to improve software quality, but its straightforward use may decrease readability of the source codes. To avoid the latter shortcoming, one can just develop a couple of design-by-contract aspects or configure some ready ones, if any.

Advantages of Using AOP and Aspect.NET and Its Perspectives

After the given introduction to the Aspect.NET technology, let's summarize some of its advantages, along with to those already mentioned above.



f2 ***Figure 2:*** *Visualizing (coloring) aspects*

- ***Aspect.NET makes creating new aspects easier.*** Due to powerful Whidbey refactoring features, it becomes possible to automatically extract the crosscutting concerns into separate classes and, by adding attribute annotations to them, turn them into aspects. Nevertheless, we started to develop an *aspectizer* (aspect mining tool) as part of our Aspect.NET system.
- ***Aspect.NET allows you to use all of the existing tools for .NET.*** Aspects in Aspect.NET are separate compilation units and store all necessary information in their assemblies. So, one can work with them as with ordinary .NET assemblies, e.g., use NUnit tool (www.nunit.org) for testing.
- ***Aspect.NET enables readability and understanding the results of weaving.*** One can use the resulting assembly to construct a UML diagram, use decompilers, and visualize the joinpoints as parts of the source of the project. It makes testing the correctness of aspect weaving easier. Further work in this direction may lead to adding formal specifications to aspect definitions and to target assemblies (as custom attributes) and implementing a formal verifier (a tool to prove formal correctness of weaving) as part of Aspect.NET.
- ***Aspect.NET facilitates the development of versions of software systems intended for various kinds of users.*** It does so by updating, adding, or deleting the appropriate aspects with the Aspect.NET aspect editor.
- ***Aspect.NET helps the user to make the weaving correct and consistent.*** Generally speaking, it is not an easy task to determine the pointcuts for many complicated software projects, to correctly describe all joinpoints. Thus we think the user should be allowed to control the process of weaving. Aspect.NET shows the potential joinpoints and allows the user to select and unselect them.

Conclusion

Due to the development of the Microsoft.NET platform, the developers get more and more opportunities that were unavailable before. AOP lets a software developer concentrate mostly on the business logic of his or her project, by implementing auxiliary features as special kinds of reusable modules – aspects. Interaction of aspects to the core code of the project is defined in a high-level metalanguage that does not depend on the project implementation language. These loosely coupled software components can be developed independently. Keeping in mind the tendency toward distributed software system development, using AOP thusly can greatly decrease the cost of software products. Whereas for the Java platform there are a lot of powerful AOP tools, we expect that, due to further development of Microsoft Phoenix and Visual Studio.NET, more and more AOP systems will be also developed for .NET and integrated to the oncoming versions of Visual Studio.

Our Aspect.NET tool is one of the first such systems, discovering for .NET software developers the magic world of AOP.

The working prototype of the Aspect.NET tool and its documentation is available upon request from the authors and will soon be available at the Microsoft Developer's Network Curriculum Repository (www.msdn-aa.net/curriculum). In order to function, Aspect.NET requires that Whidbey beta2 and the Phoenix RDK (Microsoft Phoenix home page: http://research.microsoft.com/phoenix – available from Microsoft under the terms and conditions of the Phoenix Academic Program) be preinstalled.

References

1. Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. *Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code.* In: Proceedings of the Joint European Software Engineering Conference (ESEC), September 2001: www.cs.uvic.ca/~ycoady/papers/fse.pdf
2. J. Hannemann and G. Kiczales. *Design Pattern Implementation in Java AspectJ.* OOPSLA, November 2002: www.cs.ubc.ca/labs/spl/papers/2002/oopsla02-patterns.html
3. D. Shukla, S. Fill, and D. Sells. "Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse." *MSDN Magazine*, March 2002.
4. Vladimir O. Safonov. "Aspect.NET – A New Approach to Aspect-Oriented Programming." *.NET Developer's Journal.* April 2003.
5. Vladimir O. Safonov. "Aspect.NET: concepts and architecture." *.NET Developer's Journal.* October 2004.
6. Vladimir O. Safonov and Dmitry A. Grigoriev. "Aspect.NET – an aspect-oriented programming tool for Microsoft.NET." In: Proceedings of IEEE Regional Conference 2005, St. Petersburg, 2005.
7. *Mik Kersten. AOP tools comparison:* www.ibm.com/developerworks/java/library/j-aopwork1/ ◉

Listing 1: An example of a profiling aspect. Metalanguage constructs are in bold. Please note that, for weaving the aspect into different kinds of join points and according to different kinds of rules, the %action declaration should be updated only.

```
%aspect AspectProfiler
public class AspectProfiler
{

%modules
     static private DateTime StartTime;
     static private DateTime EndTime;

     public static void StartTimeUpdate()
     {
           StartTime = DateTime.Now;
     }

     public static void EndTimeUpdate()
     {
           EndTime = DateTime.Now;
     }

%rules
     %action %before %call *OrderSystem.
StartOrderProcessing
     public static void ActionBeforeCall(string
methname)
     {
           AspectProfiler.StartTimeUpdate();
     }

     %action %after %call *OrderSystem.
StartOrderProcessing
     public static void ActionAfterCall(string
methname)
     {
           AspectProfiler.EndTimeUpdate();
        if (StartTime.Ticks != 0)
        {
         TimeSpan tsp = EndTime.
Subtract(StartTime);
             System.Console.WriteLine("AspectProfi
ler: Executing time of
{0}: {1} milliseconds", methname, tsp.
Ticks/10000);

        }
     }

}//AspectProfiler
```

Listing 2: The *Client* class of the ordering system

```
//Client.cs
class Client
{
   public string Name;
   public double Account;

   public Client(string _Name, double _Account)
   {
         Name = _Name; Account = _Account;
   }

   public void CreditOperation(double Amount)
   {
         Account -= Amount;
  ProcessOperation(); // Perform time consuming
further processing
   }

   public void DebitOperation(double Amount)
   {
      Account += Amount;
  ProcessOperation(); // Perform time consuming
further processing
   }

   public void ProcessOperation()
   {
         Thread.Sleep(2000); // Emulation of pro-
cessing
   }

}//Client
```

Listing 3: The *OrderSystem*_class of the ordering system

```
//Program.cs
class OrderSystem
{
public Hashtable ActiveClients;

  public OrderSystem(Hashtable _Clients)
{ActiveClients = _Clients;}

  static void Main(string[] args)
  {
         Hashtable ActiveClientsTable = new
Hashtable();
         ActiveClientsTable.Add("John", new
Client("John", 100));
         ActiveClientsTable.Add("Bob", new
Client("Bob", 250));
         OrderSystem os = new OrderSystem(ActiveC
lientsTable);
         os.StartOrderProcessing();
//Printing somehow ActiveClients table on the
console...
  }

  public void StartOrderProcessing()
  {
         Client John = (Client)ActiveClients["Joh
n"];
```

```
        Client Bob = (Client)ActiveClients["Bob"];
        John.CreditOperation(250);
        Bob.DebitOperation(250);
}
}// OrderSystem
```

Listing 4: The *AsyncCallAspect*

```
%aspect AsyncCallAspect
public class AsyncCallAspect
{
%modules
    static private ArrayList ThreadsPool = null;

%rules
    %action %instead %call *Client.ProcessOperation
    static public void WrapMethodWithAsyncCall
                        (string FullMethodName,
Object target)
    {
        if (ThreadsPool == null)
        ThreadsPool = new ArrayList();

    string ShortMethodName =
      FullMethodName.Substring(FullMethodName.
LastIndexOf(".")+1);

        MethodInfo mi = target.GetType().GetMethod(
ShortMethodName);
  ThreadStart ts =
(ThreadStart)ThreadStart.CreateDelegate(typeof(Thre
adStart),
target, mi);
    Thread thread = new Thread(ts);
    ThreadsPool.Add(thread);
    thread.Start();
   }

   %action %after %call *OrderSystem.
StartOrderProcessing
   static public void TotalJoining()
   {
        if (ThreadsPool != null)
    {
        foreach (Thread t in ThreadsPool)
         t.Join();
    }
   }
}// AsyncCallAspect
```

Listing 5: *CreditAbilityAspect* and *TransactionalAspect*

```
%aspect CreditAbilityAspect
public class CreditAbilityAspect
{
%rules
    %action %after %call *Client.CreditOperation
    public static void CreditOverflowAssertion
                        (string FullMethodName,
Object target)
    {
        Type TargetType = target.GetType();
  double ClientAccount =
      (double) TargetType.GetField("Account").
GetValue(target);
    if(ClientAccount < 0)
        throw new Exception("Account exceeded!");
    }
}// CreditAbilityAspect

%aspect TransactionalAspect
public class TransactionalAspect
{
%modules
  static protected void BackupSystemState(Object
target)
  {
          //Store somehow this.ActiveClients table
  }

  static protected void RecoverSystemState(Object
target)
  {
     //Copy previously stored ActiveClients to
this.ActiveClients
  }

%rules
%action %instead %call *OrderSystem.
StartOrderProcessing
  public static void AroundTransaction(string
FullMethodName,
    Object target)
  {
    string ShortMethName =
FullMethodName.Substring(FullMethodName.
LastIndexOf(".")+1);
      Type TargetType = target.GetType();
      // ... Saving state of the system for further
recovering
      BackupSystemState(target);
      try
      {
          TargetType.GetMethod(ShortMethName).
Invoke(target, null);
            //...Commit transaction
      }
      catch (Exception e) //...Rolling back transac-
tions

      {
          Console.WriteLine("Rolling back transac-
tions...");
           RecoverSystemState(target);
       }
   }
}// TransactionalAspect
```