Aspect Oriented Programming under .NET

Ramaswamy Krishnan-Chittur

Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering

Advisor:

Dr. James Fawcett

Department of Electrical Engineering and Computer Science
Syracuse University

Table of Contents:

1. Aspect-Oriented Programming - 1
    1.1. What are aspects? - 5
    1.2. What is Aspect-Oriented Programming? - 5
    1.3. Thesis outline - 6
2. Technical support for AOP in .NET - 8
    2.1. Introduction - 8
    2.2. Messages - 8
        2.2.1. The IMessage interface - 9
        2.2.2. Message invocation - 11
        2.2.3. Message sinks - 13
        2.2.4. Message wrappers - 15
    2.3. Channel architecture in .NET - 16
        2.3.1. Channel Sinks - 18
        2.3.2. Message Processing in the Channel Sink Chain - 20
        2.3.3. Default Sinks - 21
            2.3.3.1. Formatter Sinks - 21
            2.3.3.2. Transport Sinks - 22
        2.3.4. Custom Channel Sinks - 22
        2.3.5. Replacing the Default Formatter: - 24
        2.3.6. Sink providers - 25
    2.4. Contexts - 27
        2.4.1. Cross-context communication - 28
        2.4.2. Context attributes - 31
        2.4.3. Context properties - 31
        2.4.4. Context Sinks (Interceptors) - 31
        2.4.5. Reflection - 33
    2.5. Summary - 33
3. Method Synchronization - 34
    3.1. Introduction - 34
        3.1.1. Prior Work - 35

3.2. AOP-based Method Synchronization - 35
3.3. Achieving Method-Synchronization - 40
3.3.1. .NET features that support Method-Synchronization - 40
3.3.2. Requirements - 41
3.4. Theory - 41
3.5. Architecture and Implementation - 48
3.6. Analysis - 51
3.7. Conclusion - 52
4. Debugging - 53
4.1. Introduction - 53
4.2. Developing debugging applications using AOP - 54
4.2.1. Overview of AOP in .NET - 55
4.2.2. Requirements - 56
4.3. Theory - 58
4.4. Conclusion - 74
5. Code Profiling - 75
5.1. Introduction - 75
5.2. Code Coverage - 75
5.2.1. Theory - 76
5.2.1.1. .NET features that support code profiling - 76
5.2.1.2. Prerequisite - 76
5.2.1.3. Functioning - 77
5.3. Time profiling - 80
5.3.1. Theory - 80
5.3.1.1. .NET features that support code profiling - 80
5.3.1.2. Prerequisite - 81
5.3.1.3. Functioning - 81
5.4. Conclusion - 83
6. Prioritization - 84
6.1. Introduction - 84
6.2. Background - 84

6.3. Client Prioritization - 85

6.3.1. Client Prioritization architectural overview - 86

6.3.2. Theory - 87

6.3.2.1. Inserting the sink into the remoting chain - 95

6.4. Message Prioritization - 96

6.4.1. Architectural overview - 96

6.4.2. Theory - 97

6.5. Conclusion - 100

7. Analysis - 101

7.1. Introduction - 101

7.2. Analysis - 104

7.3. Conclusion - 123

8. Conclusion - 126

- Summary - 127
- Future Work - 131

Appendix - 132

References - 136

List of Diagrams:

2.1: Inter-AppDomain communication demands message passing - 9

2.2: Types of messages - 10

2.3: Overriding message invocation using custom proxies - 12

2.4: Channel-Sink architecture in .NET - 17

2.5: Implementing custom channel sinks - 24

2.6: Contexts - 27

2.7: Objects and contexts - 28

2.8: Context-Agile objects - 29

2.9: Context-Bound objects - 30

2.10: Cross-context message passing - 30

2.11: Class diagram for a typical inter-context interception library - 32

3.1: Registering ID via attributes - 42
3.2: Synchronization via locking at the interceptor sink - 45
3.3: Achieving method synchronization by AOP - 47
3.4: Class layout for the method synchronization library - 50

4.1: Context-bound object activation - 60
4.2: Class layout for the interception library - 61
4.3: The custom function debugger displaying the metadata information of a method call - 63
4.4: Editing the method call's argument value - 66
4.5: User-Interface waiting for user input to trigger post-processing - 68
4.6: Custom debugger printing out the stack trace - 69
4.7: Top level Activity diagram for the custom debugger - 72

6.1: Module layout for the client prioritizer - 87
6.2: Class diagram for the client prioritizer - 94
6.3: Module layout for the message prioritizer - 97

7.1: Class diagram for non-AOP File-Center application - 114
7.2: Class diagram for AOP-based File-Center application - 116

Code Listings:

2.1(a), (b): Message Invocation - 12

3.1: A queue-handler class that does not employ AOP - 36
3.2: Queue-handler class employing AOP - 37
3.3: Two methods owned by two separate types that require to be synchronized - 39
3.4: Synchronizing methods belonging to two different types using AOP-based method synchronization - 39
3.5: Registering ID via attributes - 43
3.6: Tracing ID, querying for the lockee - 44

3.7: Synchronization via locking at the interceptor sink - 46

4.1: Enabling interception - 56
4.2: Programmatically setting a break point - 57
4.3: A client class enabled for interception - 58
4.4: The executive spawning an instance of a context-bound object - 59
4.5: Tracing metadata information using reflection - 64
4.6: Custom processing at the intercepting sink - 65
4.7: Manipulating a method call message at the sink stack using message wrappers- 67
4.8: The actual method call - 67
4.9: Post-processing the message - 68
4.10: Dumping the stack trace - 70
4.11: Manipulating a return message using method return message wrapper - 71
4.12: Restarting the application - 73

5.1: Processing functions at the interceptor sinks - 78
5.2: Code-coverage method called at the interceptor sink - 79

6.1: Serialization method of the custom formatter class - 88
6.2: Deserialization method of the custom formatter class - 89
6.3: Custom serialization of messages at the client interceptor - 90
6.4: Tracing the client’s Uri - 91
6.5: Context interception code for IP validation - 91
6.6: Context property class doing IP validation - 92
6.7: Priority assignment for clients - 93
6.8: Configuration file for client NOT employing interceptors - 95
6.9: Configuration file for client employing interceptors - 95
6.10: Priority assignment for messages - 99

7.1: Non-AOP implementation of the Geometry_FindSectorArea method - 105
7.2: AOP implementation of the Geometry_FindSectorArea method - 105

7.3: A queue-handler class that does not employ AOP - 109

7.4: Queue-handler class employing AOP - 110

7.5: DeleteFile method in the non-AOP File-Center application - 115

7.6: DeleteFile method in the AOP-based File-Center application - 115

7.7: A type that is enabled for interception by the debugger-interception library - 120

7.8: Executive spawns a new instance of ClosedFigure type,
and invokes its method - 121

7.9: A code-profiling application - 122

List of Tables:

7.1: Lists the aspects that were dealt with in this thesis work, and whether or not quantitative analysis was performed on a set of applications based on the aspect - 103

7.2: Analysis results for the range-checker application - 107

7.3: Analysis results for the Queue-Synchronization application - 112

7.4: Analysis results for the File-Center application - 117

7.5: Integrated analysis results - 118

ABSTRACT

In commonly employed code, there are elements that are orthogonal to the primary functionality of the code. These elements, though extraneous to the primary purpose of the code, are vital to its proper execution. Further more, they tend to be scattered throughout the system so that they contribute to the complexity of the code. These elements are called aspects. Examples of aspects include processing to support 'security', 'fault-tolerance' and 'synchronization'. Aspect-oriented programming (AOP) isolates aspects into separate modules so that the resulting client-code is more purpose-specific, more reusable, and less complex.

In this research, we analyze the efficacy of AOP under the .NET[1] environment. We identify several applications where aspect-oriented approach can be applied. Aspects that are dealt with in this thesis work are Method-Synchronization, Debugging, Code-Profiling, Prioritization, and Validation. The technique used here for implementing AOP is interception of method calls. For each of the aspect-oriented applications, we evaluate the impact of AOP on code complexity and performance. Finally, we have summarized our contributions and conclusions, and have made some suggestions for future work along similar lines.

[1] A technology specific, at the current time, to Microsoft Windows platforms. This however is changing due to work on cross-platform implementations, most notably the Mono project, supporting most of the .Net platform on the Unix and Linux platforms.

1. ASPECT-ORIENTED PROGRAMMING

In the early 1990's, Gregor Kiczales, et al, published a seminal paper on what they called "*the young idea* of Aspect Oriented Programming" (AOP) at Proc-Europe Conference on object-oriented programming [1]. A decade later, AOP continues to generate considerable interest among programmers and developers. AOP programmers and developers claim that aspect-oriented approach for programming has "immense potential for improving non-functional aspects of components such as resource usage, timing behavior, fault tolerance and security" [2]. In this research, we have analyzed different applications where aspect-oriented approach can be applied, and have evaluated the impact of AOP on code complexity and performance. Currently, work is being done on aspect oriented programming with both Java and .NET[2]. We have used .NET for our work.

To illustrate the concept of AOP, consider the example of a server operating in a large LAN (or a web server for that matter). Suppose that the server needs to trace the number of clients trying to connect to it at different points of time, in addition to processing their requests.

Here the client-tracing aspect is extraneous to the message processing logic of the server system. We use the term orthogonal to define functionalities, client-tracing, that are independent of the primary purpose of the application.

[2] The reference section of this thesis work lists 31 publications (papers, books), many of which deal with AOP in Java and .NET

1. Aspect oriented programming

Conventional style of programming would add code for client tracing in the main body of the code, which would result in the client tracing code being mixed with the message-passing logic of the server system. From this point forward, we use the term 'tangling' to refer to the combination of primary functionality of the code with secondary elements.

Kiczales states "if we modularize some aspects of our program, it may not be possible to modularize others at the same time. When code is scattered in different fragments throughout a program, it is hard to see its structure and hard to get a good view of the apparent tangling of the code. It's hard to change such code efficiently and hard to find all the cases that have to be changed." [1] As discussed before, in commonly employed code, there are elements that are secondary to the primary functionality of the code. These elements though non-primary are vital to the proper execution of the code. Furthermore, they may be so scattered throughout the system that they contribute to the complexity of the code. These elements are called aspects. Examples of aspects include 'security', 'fault-tolerance' and 'synchronization'. Aspect-oriented programming tries to isolate aspects into separate modules so that the resultant client-code is more purpose-specific, more reusable, and less tangled. It accomplishes this by a process of interception – intercepting function calls and managing their execution, as described in chapter 2.

Consider another example with a method in a class called *Geometry* that finds the area of a sector of a circle. We pass the *radius* and the *angle* of the sector as parameters; the returned value of area should be half of the square of radius times angle. Again, the method has to make sure that the radius and angle passed as parameters are non-negative; here, we make an

assumption that the angle is *between 0 and 2 PI*. Otherwise, the method has to throw an exception. Let us see how we implement the method in the conventional way:

```
public double Geometry_FindSectorArea(double radius, double angle)
{
  if (radius < 0.0)
      throw new ArgumentOutOfRangeException("radius", "The radius is out of range.");
  if ( (angle < 0.0) || (angle > 6.2832) )
      throw new ArgumentOutOfRangeException("angle", "The angle is out of range");
  return (radius * radius * angle / 2);
}
```

Here, the actual *function* of the method is just:

```
return (radius * radius * angle / 2);
```

The remaining piece of code is orthogonal to the primary purpose of the method. Isolating the *range-checking* aspect from the code would, clearly, make the code more readable and purpose-specific, and less complex.

This is nicely achieved by the AOP code, which is shown below:

```
[method: Range(true, Lower = 0.0)]
public double Geometry_FindSectorArea
              ([Range(true, Lower = 0.0)] double radius,
               [Range(true, Lower = 0.0, Upper = 6.28)] double angle)
{
  return (radius * radius * angle / 2);
}
```

The code ***clearly describes*** its function. The parameter *radius* has a range-checker attribute, which specifies its lower range as 0. The *angle* parameter has a range that lies between 0 and 2 PI. Again the method attribute to the method restricts the *lower range of the return value* to 0. The AOP-based method is more readable, and captures the purpose of the code better. Again, all that the function body does is its function, namely calculating the area of the sector of the circle.

In the following chapters, we discuss how this is achieved using AOP. We also present performance and code analysis for various applications based on AOP.

In a general sense, the advance made by AOP has been its potential in complimenting OOP to develop an improved system. Certain aspects, like thread synchronization constraints and failure handling, are systemic in nature and present across components, which implies that they get tangled with the main functionality when developers employ conventional OOP approaches alone [1]. AOP successfully 'captures' these aspects, and isolates them as a separate entity. In the end, what results is a system that interweaves both OOP and AOP, and which is clearer and more focused on both the primary functionality and the aspects of its performance.

1.1. What are aspects?

Kiczales classifies components and aspects this way:

"With respect to a system and its implementation using a general procedure based language, a property that must be implemented is: **A component** or **an Aspect**

- **A component**, if it can be cleanly encapsulated in a generalized procedure (i.e., object, procedure, method, API). By cleanly, it is meant – well localized, and easily accessed and composed as necessary. Components tend to be units of the system's functional decomposition, such as image filters, bank accounts and GUI widgets.
- **An Aspect,** if it cannot be cleanly encapsulated in a generalized procedure. Aspects tend not to be units of the system's functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways. Examples of aspects include memory access patterns and synchronization of concurrent objects." [1]

1.2. What is Aspect-Oriented Programming?

Again, in commonly employed code, there are elements that are orthogonal to the primary functionality of the code. These elements, though extraneous to the primary purpose of the code, are vital to its proper execution. Further more, they tend to be scattered throughout the system that they contribute to the complexity of the code. Aspect-oriented programming aims at isolating aspects into separate modules so that the resultant client-code is more purpose-specific, more reusable, more readable, and less tangled.

The purpose of this thesis is to identify several applications that can be effectively developed using AOP, and to evaluate the impacts of AOP on code complexity and performance.

Performance evaluation is based on metrics concerned with cohesion[3], line count, and cyclomatic complexity[4] of the code.

1.3. Thesis outline:

In the following section, we outline fundamental parts of this thesis work:

1. Aspect-Oriented Programming needs support from the language used to develop the program. Currently, there are very few languages / platforms that support AOP. Java and .NET are two that do. We have chosen .NET for our research. AOP support is provided by the .NET platform, and is available for any language that targets the Common Language Runtime (CLR). We begin by studying technical support provided by .NET for AOP. Chapter 2 discusses a message-passing mechanism, channel architecture and the execution model used in .NET. We describe how these facilities support interception in .NET. Interception is the process of intercepting method calls, and letting some custom code execute before or after the call, or, perhaps, instead of it.

[3] Please refer to Appendix 1.(a)

[4] Please refer to Appendix 1.(b)

1. Aspect oriented programming

2. In chapters 3, 4, 5, 6 and 7, we have identified some common applications that have secondary aspects, and have implemented them using AOP. The aspects that were successfully isolated in these chapters are: "Method Synchronization", "Debugging", "Code-Profiling", "Prioritization", and "Validation". We discuss in detail AOP techniques to successfully isolate these elements from application code, and benefits and drawbacks of aspect-oriented approach[5] for each of them.

3. For most systems, to perform a quantitative assessment of implementation efficacy, the criteria for performance evaluation will have to be chosen after selecting the system for study. For the systems consider in chapters 3 through 7, we have chosen 'code complexity' and 'cohesion' as criteria with which to compare an AOP implementation and a parallel Non-AOP implementation[6]. We present the results at the end of each chapter, and in a final sumarization.

4. Finally, we draw our conclusions, list our contributions, make suggestions for future work along similar lines, and suggest a few existing and prospective applications that could be implemented efficiently using AOP.

---

[5] We chose not to explore 'security' as we believe that it has been well researched already.

[6] These criteria, as we realized the hard way during our research, did not lend themselves to quantitative definition. For instance, "How readable a code is", is subjective to a certain degree.

2. TECHNICAL SUPPORT FOR AOP IN .NET

2.1. Introduction:

Aspect-Oriented Programming is a relatively new paradigm; one unique characteristic of AOP is that AOP needs the support of the language. Currently, there are not many languages that support AOP. Java, with the support of AspectJ, and .NET are a couple of environments that support AOP. In this chapter, we look through the .NET features that support AOP and focus on the technicalities that support AOP by means of interception.

Interception of method calls is the most important technique in .NET that enables Aspect-Oriented Programming. Interception can be inter-application domain or inter-context, explained in the next section, where, we look at .NET support for interception.

2.2. Messages:

Each programming environment has a fundamental scope of execution. In .NET the fundamental scope of execution is an *AppDomain*, Application Domain [25]. While ***application domains*** form the fundamental domain of execution in .NET, ***contexts*** are a collection of execution attributes that function as boundaries as well. For communication between different contexts, message passing is enforced, provided the objects are context-bound. It is the process of handling messages that introduces the possibility of (message) interception.

Messages are the fundamental units of data transfer across .NET boundaries. Data is transferred across these AppDomain and context boundaries by Message objects. The

System.Runtime.Remoting.Messaging namespace contains classes used to create and transmit messages. The remoting infrastructure uses messages to communicate with remote objects. Messages are used to transmit remote method calls, to activate remote objects, and to communicate information. A message object carries a set of named properties, including action identifiers, envoy information, and parameters [31].

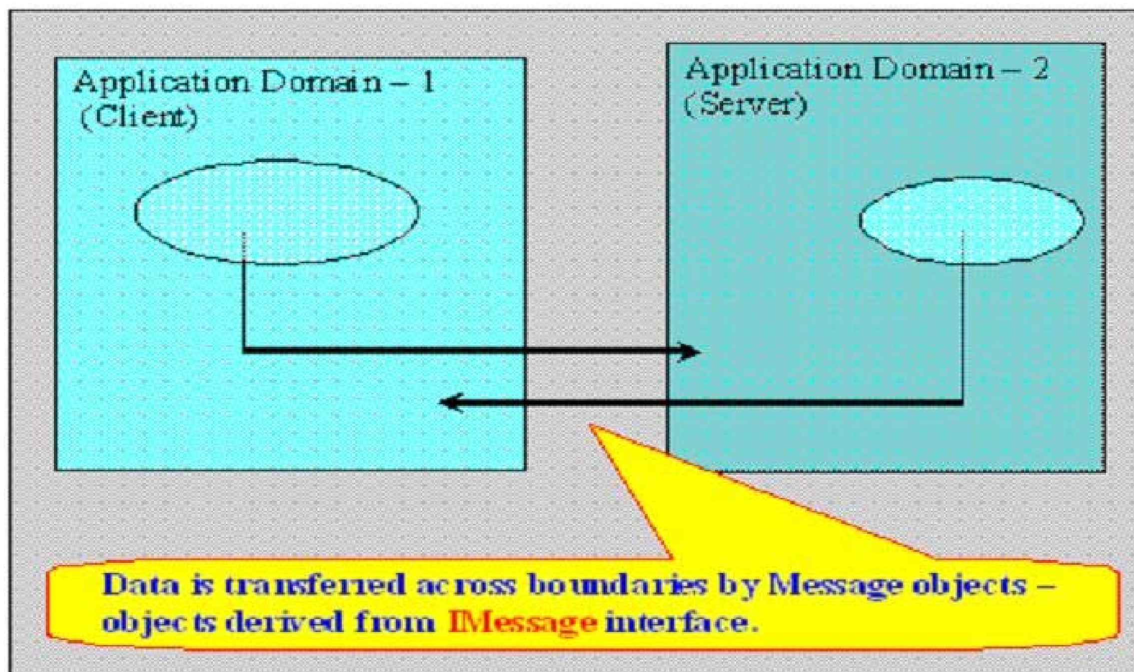

Figure – 2.1: Inter-AppDomain communication demands message passing

2.2.1. The IMessage interface:

Messages are instances of classes implementing the IMessage interface. An object that implements the **IMessage** interface meets the minimum qualifications to be considered as a message object. The following diagram describes the types of messages, and their hierarchy.

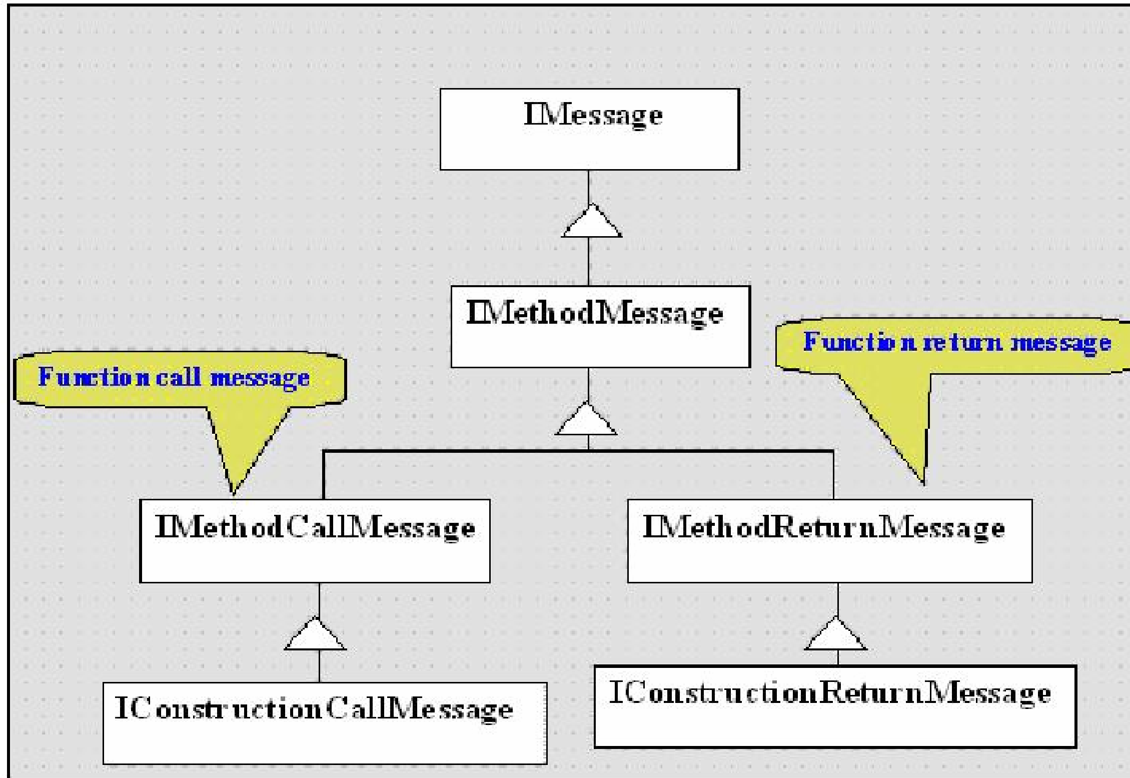

Figure - 2.2: Types of messages

- **IMethodMessage**: A method message is used to send information to and from remote methods. Messages used for remote method calls implement the **IMethodMessage** interface.

- **IMethodCallMessage**: An **IMethodCallMessage** is generated as a result of a method called on a remote object, and is used to transport details about the remote method call to the server side.

- **IMethodReturnMessage**: A method call return message represents the response to a method call on an object at the end of the message sink. An **IMethodReturnMessage** is

generated as a result of a method called on a remote object, and is used to return the results of the method call back to the caller.

- **IConstructionCallMessage**: When the user creates an instance of a new client-activated object by calling **new** or **Activator.CreateInstance** and before the thread returns to the user code, an **IConstructionCallMessage** is sent to the remote application. When the construction message arrives at the remote application, it is processed by a remoting Activator (either the default one, or one that is specified in the Activator property) and a new object is created. The remoting application then returns an **IConstructionReturnMessage** to the local application. The **IConstructionReturnMessage** contains an instance of ObjRef, which is a package of information about the remote object. The remoting infrastructure converts the **ObjRef** instance into a proxy to the remote object, which is returned to the user code.

- **IConstrcutionReturnMessage**: A message implementing **IConstructionReturnMessage** returns the result of the construction request sent with the **IConstructionCallMessage**.

2.2.2. Message invocation:

Messages are invoked whenever a construction call or a method call occurs. The response to the method call and the construction call also occur as response messages.

```
// invokes a construction call message.
MyClass M = new MyClass( );
```

Code Listing – 2.1(a): Message Invocation

```
// invokes a method call message.
M.TestFunc( );
```

Code Listing – 2.1(b): Message Invocation

We cannot invoke a method calls explicitly. We can, however, override the invocation functionality by customizing the proxies, and making the object context bound. SRC??

```
namespace AOP ProxyTest
{
    public class MyProxy : RealProxy
    {
        MarshalByRefObject _target;

        public MyProxy(Type type, MarshalByRefObject target) : base(type)
        {
            _target = target;
        }

        public override IMessage Invoke(IMessage msg)
        {
            if (msg is IConstructionCallMessage)
            {
                IConstructionReturnMessage crm = EnterpriseServicesHelper.CreateConstructio
                return crm;
            }
            else
            {
                MethodCallMessageWrapper mcm = new MethodCallMessageWrapper((IMethodCallMes
                mcm.Uri = RemotingServices.GetObjectUri((MarshalByRefObject)_target);
                return RemotingServices.GetEnvoyChainForProxy((MarshalByRefObject)_target).
            }
        }
```

Figure – 2.3: Overriding message invocation using custom proxies

There is just one system class – **ReturnMessage**, which lets us create message objects explicitly. *ReturnMessage* objects hold the return message in response to a method call on a remote object. This class is defined in the namespace System.Runtime.Remoting.Messaging

It has two constructors:

1. Initializing a new instance of the **ReturnMessage** class, which holds an exception along with it.

public ReturnMessage(Exception, IMethodCallMessage);

2. Initializing a new instance of the **ReturnMessage** class with all the information returning to the caller after the method call.

public ReturnMessage(object, object[], int, LogicalCallContext, IMethodCallMessage);

2.2.3. Message sinks:

We can intercept messages in message sinks, classes derived from the IMessageSink interface. Messages carry all the information about the method call, the parameters, the return value, info about the declaring type, the custom attributes (if any), and many other useful metadata. We can expose these metadata using reflection by intercepting the messages in a message sink.

Message sinks that use **IMessage** can be placed in the client sink chains or in the server sink chains. A message object is passed from message sink to message sink through the chain, and carries a set of named properties.

An object that implements the **IMessage** interface meets the minimum qualifications to be considered a message object. The exact object received by a message sink does not have to be passed on to the next sink, but this is often the case.

Although the objects in the property bag do not have to be serializable, the message sink implementer needs to consider this, since the properties that flow out of the application domain must be serializable.

When a method call is made on the proxy, the remoting infrastructure provides the necessary support for passing the arguments to the actual object across the remoting boundaries, calling the actual object method with the arguments, and returning the results back to the client of the proxy object.

A remote method call is a message that goes from the client end to the server end and possibly back again. As it crosses remoting boundaries on the way, the remote method call passes through a chain of **IMessageSink** objects. Each sink in the chain receives the message object, performs a specific operation, and delegates to the next sink in the chain. The proxy object contains a reference to the first **IMessageSink** it needs to use to start off the chain.

For asynchronous calls, at the time of delegation, each sink provides a reply sink (another **IMessageSink**) that will be called by the next sink when the reply is on its way back.

Different types of sinks perform different operations, depending on the type of message object received. For example, one sink could cause a lock to be taken, another could enforce call security, another could perform flow call control and reliability services, and yet another could transport the call to a different AppDomain, process, or computer. Two or more message sinks in the chain can interact with each other in regard to each specific action.

2.2.4. Message wrappers:

We can manipulate a message in the message sink using *message wrappers*. .NET provides two classes, **MethodCallMessageWrapper** and **MethodReturnMessageWrapper**, which facilitate manipulating messages in the sink stack. These classes are defined in the **System.Runtime.Remoting.Messaging** namespace.

2.3. Channel architecture in .NET:

As discussed in the previous section, messages function as the fundamental units of data transfer across .NET boundaries. When we construct a proxy and make a method call, we generate a *message* object, which travels through a series of sinks as it makes round-trips to the server. The *Message* class, which is referenced through the *IMessage* interface within the sinks, contains a dictionary that holds the data representing the call. The remoting framework exposes a number of IMessage implementing classes, which represent different messages passed along either during construction (Construction Call, Construction Response) or during a method call (Method Call, Method Response).

Inter-AppDomain communication involves message passing through channels. In the following section, we will see the channel architecture in .NET.

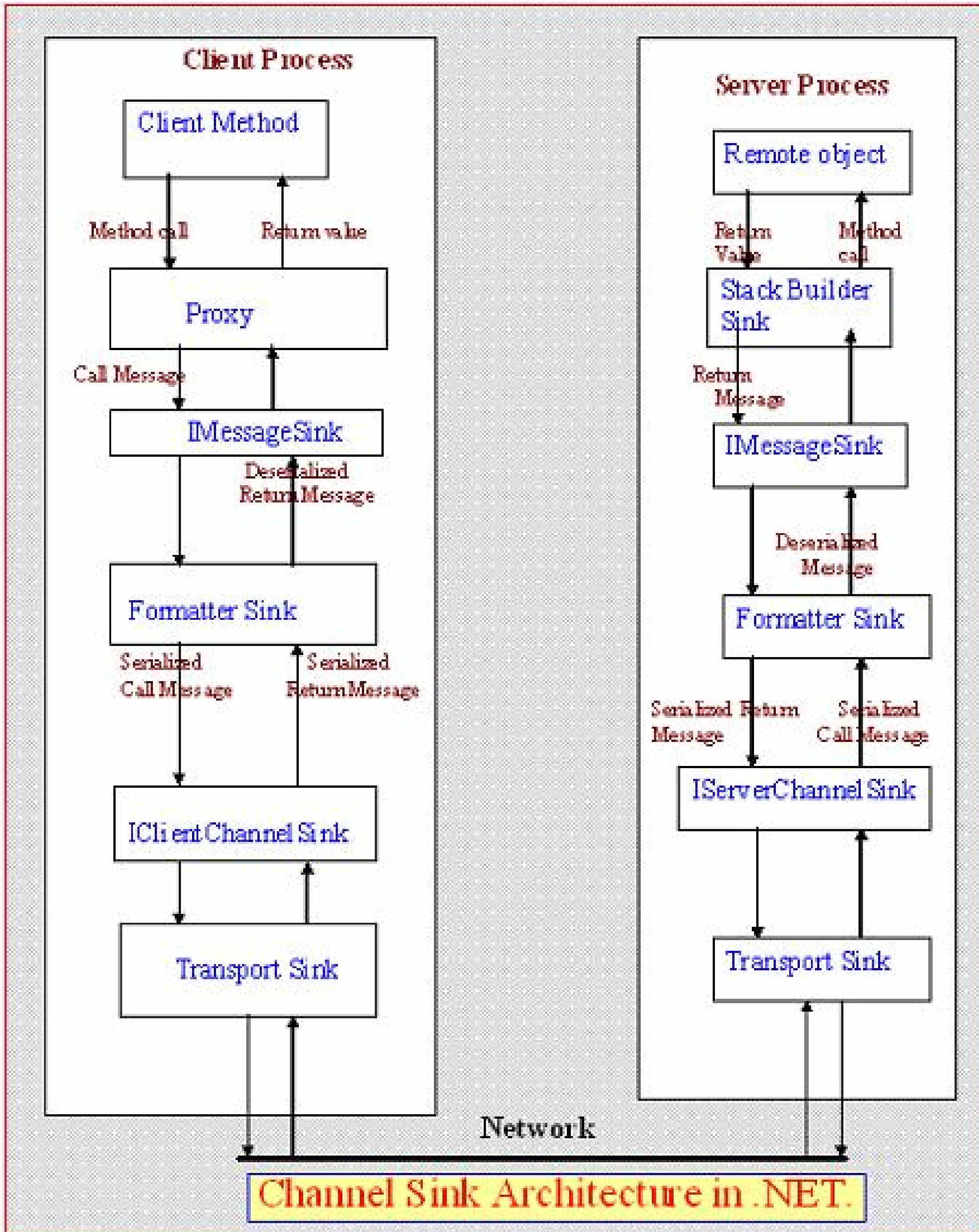

Figure - 2.4: Channel-Sink architecture in .NET

2.3.1. Channel Sinks:

Channels send each message along a chain of channel sink objects prior to sending or after receiving a message. This sink chain contains sinks required for basic channel functionality, such as formatter, transport, or stack builder sinks, but we can customize the channel sink chain to perform special tasks with a message or a stream.

Channel sinks provide a plug-in point that allows access to the underlying messages flowing through the channel as well as the stream used by the transport mechanism to send messages to a remote object. Channel sinks are linked together in a chain of channel sink providers, and all channel messages flow through this chain of sinks before the message is serialized and transported.

Each channel sink implements either **IClientChannelSink** or **IServerChannelSink**. The first channel sink on the client side must also implement **IMessageSink**. It typically implements IClientFormatterSink (which inherits from both **IMessageSink**, **IchannelSinkBase**), and **IClientChannelSink**, and is called a formatter sink because it transforms the incoming message into a stream (an IMessage object).

The channel sink chain processes any message that is sent to or from an application domain. At this point, all we have is the message, but we are able to do anything we want with that message, and subsequent processing will use the message that we return to the system after processing. This is a natural place to implement a logging service, any sort of filter, or perhaps encryption or other security measures on the client or the server.

Each channel sink processes the stream and then passes the stream to the next channel sink, which means that objects before or after our sink should know what to do with the stream being passed to them.

Channel sink providers (objects that implement the IClientChannelSinkProvider, IClientFormatterSinkProvider, or IServerChannelSinkProvider interface) are responsible for creating the channel sinks through which remoting messages flow. When a remote type is activated, the channel sink provider is retrieved from the channel; and the **CreateSink** method is called on the sink provider to retrieve the first channel in the sink from the chain.

Channel sinks are responsible for transporting messages between the client and the server. Channel sinks are also linked together in a chain. When the **CreateSink** method is called on a sink provider, it should do the following:

- Create its own channel sink.
- Call **CreateSink** on the next sink provider in the chain.
- Ensure that the next sink and the current one are linked together.
- Return its sink to the caller.

Channel sinks are responsible for forwarding all calls made on them to the next sink in the chain and should provide a mechanism for storing a reference to the next sink.

Channel sinks have great flexibility in what they send down the sink chain. For example, security sinks that want to negotiate authentication before sending the actual serialized original message can hold onto the complete channel message, replace the content stream with their own content, and send it down the sink chain and on to the remote application domain. On the return journey,

the security sink can intercept the reply message, creating a conversation with the corresponding security sinks in the remote application domain. Once an agreement is reached, the originating security sink can send the original content stream on to the remote application domain.

2.3.2. Message Processing in the Channel Sink Chain:

Once the .NET remoting system locates a channel that can process the IMethodCallMessage implementation, the channel passes the message to the formatter channel sink by calling IMessageSink.SyncProcessMessage (or IMessageSink.AsyncProcessMessage) method. The formatter sink creates the transport header array and calls IClientChannelSink.GetRequestStream on the next sink. This call is forwarded down the sink chain, and any sink can create a request stream that will be passed back to the formatter sink. If **GetRequestStream** returns a null reference, the formatter sink creates its own sink to use for serialization. Once this call returns, the message is serialized and the appropriate message processing method is called on the first channel sink in the sink chain.

Sinks cannot write data into the stream but can read from the stream or pass a new stream along where required. Sinks can also add headers to the header array (if they have not previously called **GetRequestStream** on the next sink) and add themselves to the sink stack before forwarding the call to the next sink. When the call reaches the transport sink at the end of the chain, the transport sink sends the headers and serialized message over the channel to the server where the entire process is reversed. The transport sink (on the server side) retrieves the headers and serialized message from the server side of the stream and forwards these through the sink chain until the formatter sink is reached. The formatter sink deserializes the message and forwards it to the

remoting system where the message is turned back into a method call and is invoked on the server object.

When multiple channel sink providers are provided in a configuration file, the remoting system chains them together in the order in which they are found in the configuration file. The channel sink providers are created when the channel is created during the RemotingConfiguration.Configure call.

2.3.3. Default Sinks:

The two important built-in sinks are the **Formatter Sink** and the **Transport Sink**.

2.3.3.1. Formatter Sinks:

Formatter sinks serialize the channel message into the message stream as an object that implements **IMessage**. Some formatter sink implementations use the system-provided formatter types (BinaryFormatter and SoapFormatter). Other implementations can use their own means to transform the channel message into the stream.

The function of the formatter sink is to generate the necessary headers and serialize the message to the stream. After the formatter sink, the message is forwarded to all sinks in the sink chain through the IMessageSink.ProcessMessage or IMessagesink.AsyncProcessMessage calls. At this stage the message has already been serialized and is provided as information only.

Sinks that need to create or modify the message itself must be placed in the sink chain prior to the formatter. This is easily achieved by implementing **IClientFormatterSink**, thereby fooling

the system into believing that it has a reference to the formatter sink. The real formatter sink can then be placed later in the sink chain.

On the return journey, the formatter sink transforms the message stream back into the channel message elements (return message). The first sink on the client side must implement the **IClientFormatterSink** interface. When **CreateSink** returns to the channel, the reference returned is cast to an **IClientFormatterSink** type so the **SyncProcessMessage** of the **IMessage** interface can be called. If the cast fails, the system raises an exception.

2.3.3.2. Transport Sinks:

The transport sink is the last sink in the chain on the client side, and the first sink in the chain on the server side. Besides transporting the serialized message, the transport sink is also responsible for sending the headers to the server and retrieving the headers and the stream when the call returns from the server. These sinks are built into the channel and cannot be extended.

2.3.4. Custom Channel Sinks:

On the client side, custom channel sinks are inserted into the chain of objects between the formatter sink and the last transport sink. Inserting a custom channel sink in the client or server channel enables you to process the **IMessage** at one of two points:

- During the process by which a call represented as a message is converted into a stream and sent over the wire.
- During the process by which a stream is taken off the wire and sent to the **StackBuilderSink** object (the last message sink before the remote object on the server) or the proxy object (on the client).

Custom sinks can read or write data (depending if the call is outgoing or incoming) to the stream and add additional information to the headers where desired. At this stage, the message has already been serialized by the formatter and cannot be modified. When the message call is forwarded to the transport sink at the end of the chain, the transport sink writes the headers to the stream and forwards the stream to the transport sink on the server using the transport protocol dictated by the channel.

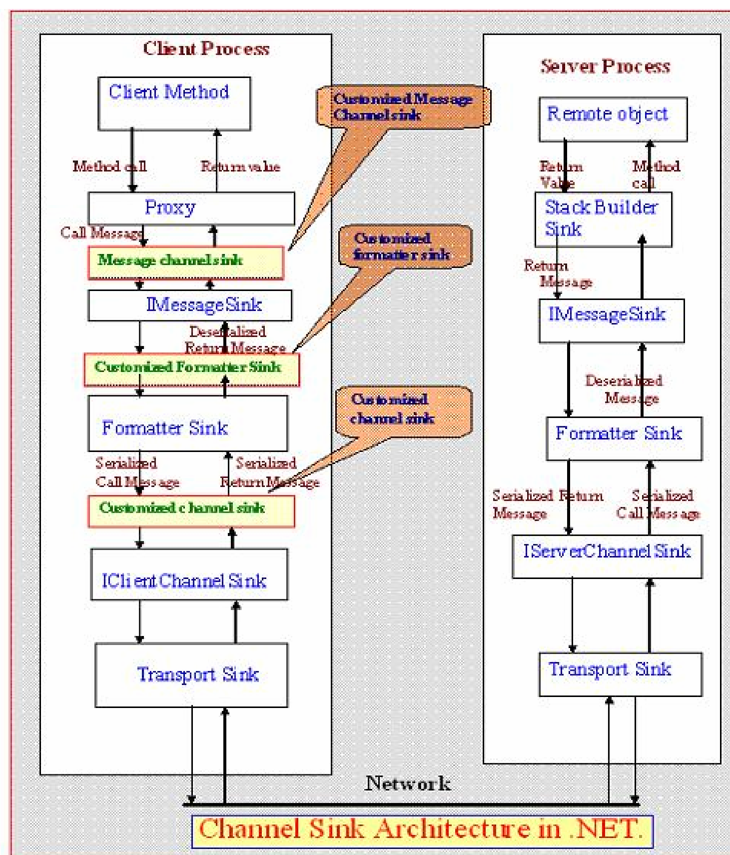

Figure – 2.5: Implementing custom channel sinks

2.3.5. Replacing the Default Formatter:

Because a channel is an abstract networking mechanism, we can configure the .NET remoting system to combine a system-implemented channel with any formatter we choose. We can do this using the channel constructor that takes an IDictionary implementation of channel properties, a server-side formatter, and a client-side formatter. We can also specify the formatter in a configuration file.

2.3.6. Sink providers:

Channel sinks are connected to a server channel through implementations of the **IServerChannelSinkProvider** interface. All the remoting server channels provide constructors that take an **IServerChannelSinkProvider** as a parameter.

Channel sink providers are stored in a chain, and the user is responsible for chaining all channel sink providers together before passing the outer one to the channel constructor. **IServerChannelSinkProvider** provides a property called Next for this purpose.

Channel sinks are connected to a client channel through implementations of the **IClientChannelSinkProvider** interface. All the remoting client channels provide constructors that take an **IClientChannelSinkProvider** as a parameter.

Channel sink providers are stored in a chain, and the user is responsible for chaining all channel sink providers together before passing the outer one to the channel constructor. **IClientChannelSinkProvider** provides a property called Next for this purpose.

When multiple channel sink providers are specified in a configuration file, the remoting infrastructure will chain them together in the order they are found in the configuration file. The channel sink providers will be created when the channel is created during the RemotingConfiguration.Configure call.

After the IMethodCallMessage is generated, .NET Framework searches through the list of registered channels to find one that can process the call. Once an appropriate channel has been found, the channel sink is retrieved from the channel, and the **IMethodCallMessage** is forwarded to the sink for processing.

As discussed, a sink provider implements IClientChannelSinkProvider for creating client-side channel sinks and implements IServerChannelSinkProvider for creating server-side channel sinks. The provider chain is initialized when the configuration file is read and the channel initialized. The order in which the providers are placed in the chain depends on the order in which they are defined in the configuration file. The CreateSink method of the provider is called when the proxy is created. It should forward the call to the next provider in the provider chain, and then chain the next sink (obtained after its own channel sink). This order sets up the sink chain through which the message passes. Using a configuration file, we can override the sink chain by providing a reference to custom sink providers.

2.4. Contexts:

As discussed in section 2.2, many programming technologies and environments define their own unique models for scoping the execution of code and the ownership of resources:

- For Java VM, it is based on Class loaders
- For IIS and ASP, the scoping model is Virtual directory.
- *For the CLR*, the fundamental scope is an *AppDomain*

Application Domains are divided into one or more contexts. In fact, every CLR application is divided into one or more contexts. Contexts are themselves objects that are instances of *System.Runtime.Remoting.Contexts.Context* type.

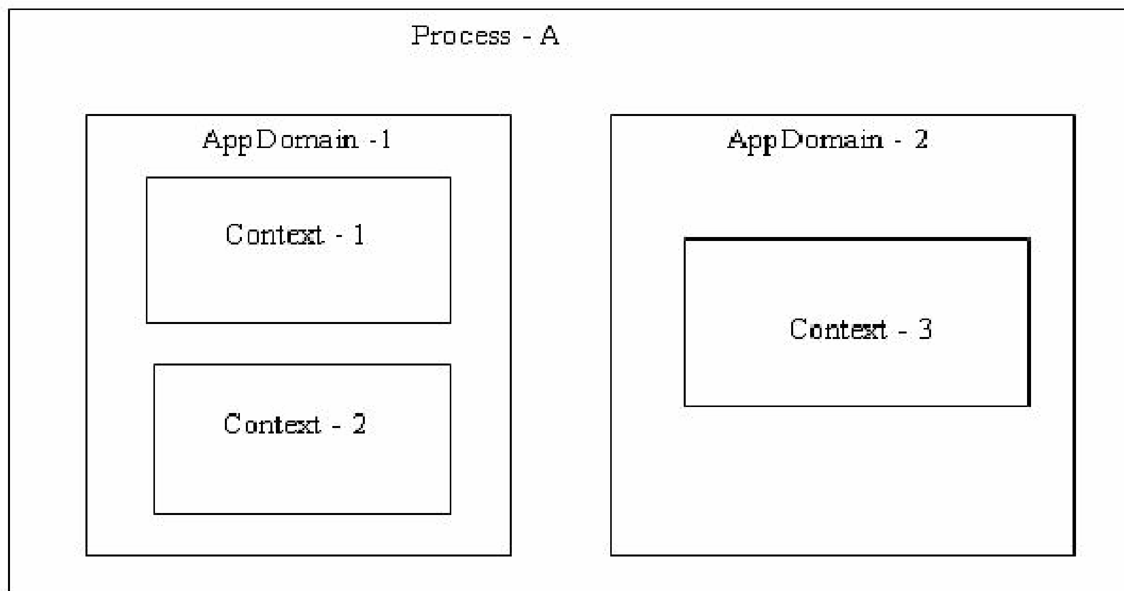

Figure – 2.6: Contexts

A context is an ordered sequence of properties that defines an environment for the objects within it. Contexts are created during the activation process for objects that are configured to require

certain automatic services such synchronization, transactions, just-in-time (JIT) activation, security, and so on. Multiple objects can live inside a context. Contexts help to group together objects that have similar execution requirements. Normally, the runtime creates contexts as needed. Whenever a new object is created, the .NET Framework finds a compatible context or creates a new context for the object. The **System.Runtime.Remoting.Contexts** namespace contains objects that define the contexts all objects reside within.

2.4.1. Cross-context communication:

Objects in an AppDomain are either Context-agile or Context-bound.

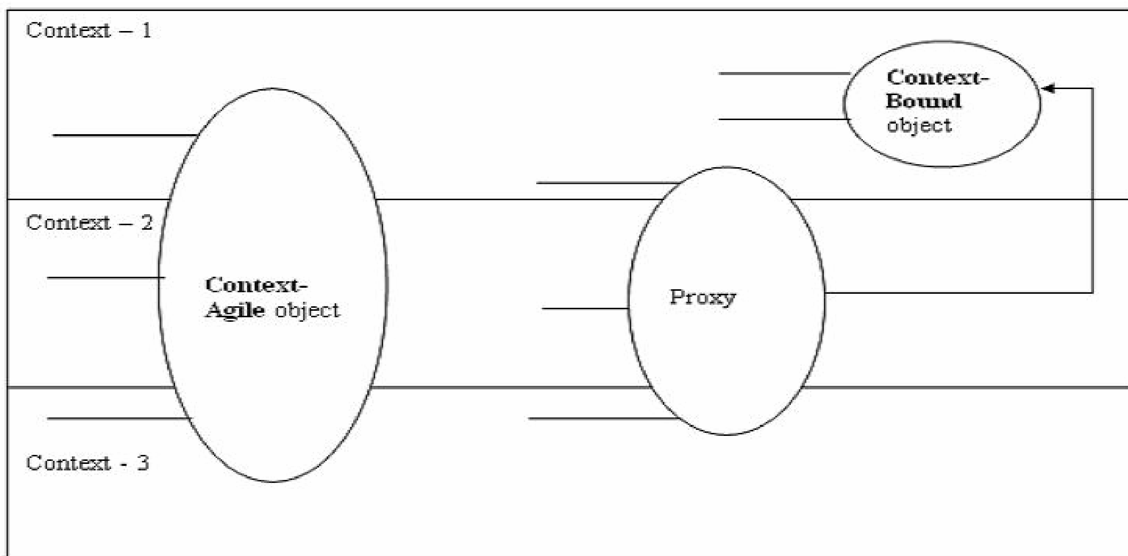

Figure 2.7: Objects and contexts

Context-Agile objects:

Context-agile objects can be accessed directly from anywhere in an AppDomain. Types derived from MarshalByRefObject are context-agile. 'Context-agile' implies pass-by-ref, if between AppDomains.

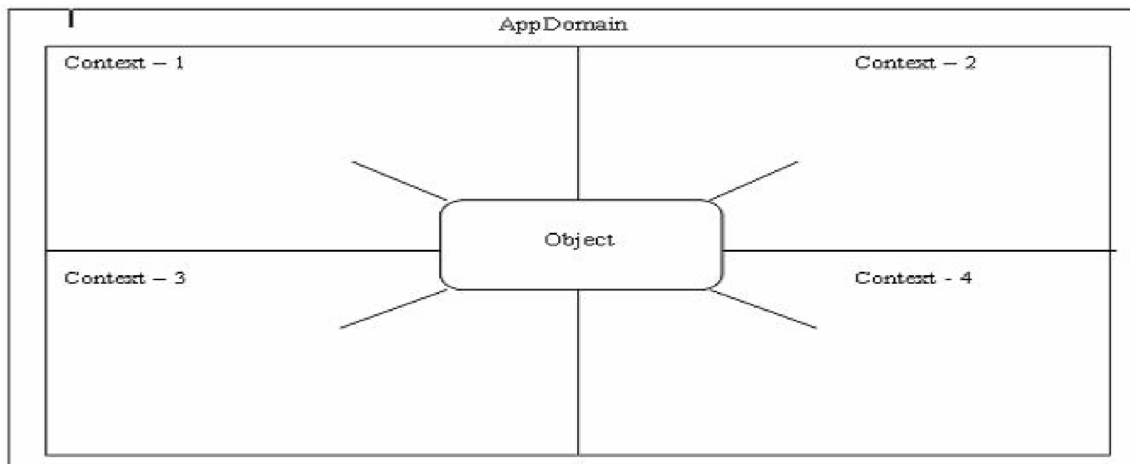

Figure 2.8: Context-Agile objects

Context-Bound objects:

Context-bound objects are accessed through proxies from outside the home context. Types derived from ContextBoundObject are context-bound. Context-bound implies pass-by-ref between contexts.

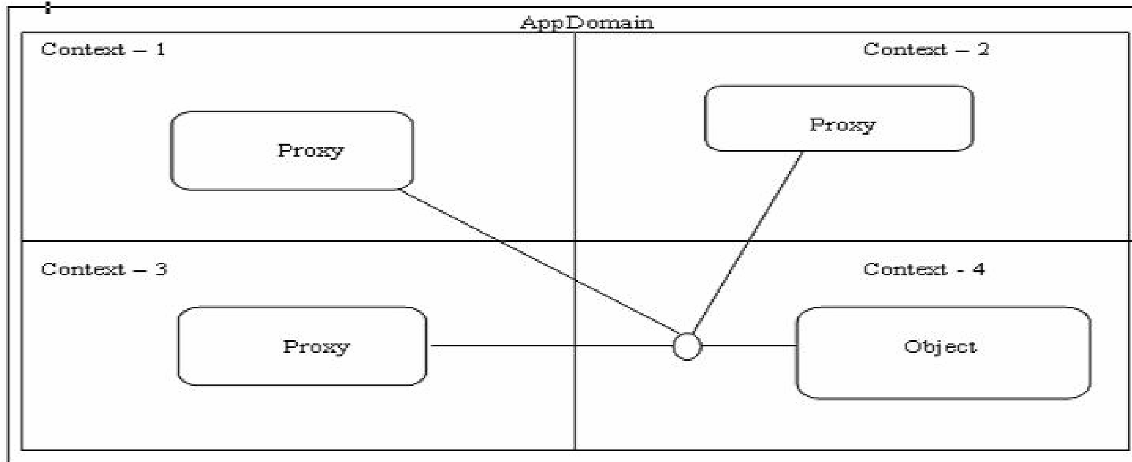

Figure 2.9: Context-Bound objects

Cross-context member access is message – based. Inter-context communication supports interception, and is completely pluggable. Cross-Context access requires a proxy

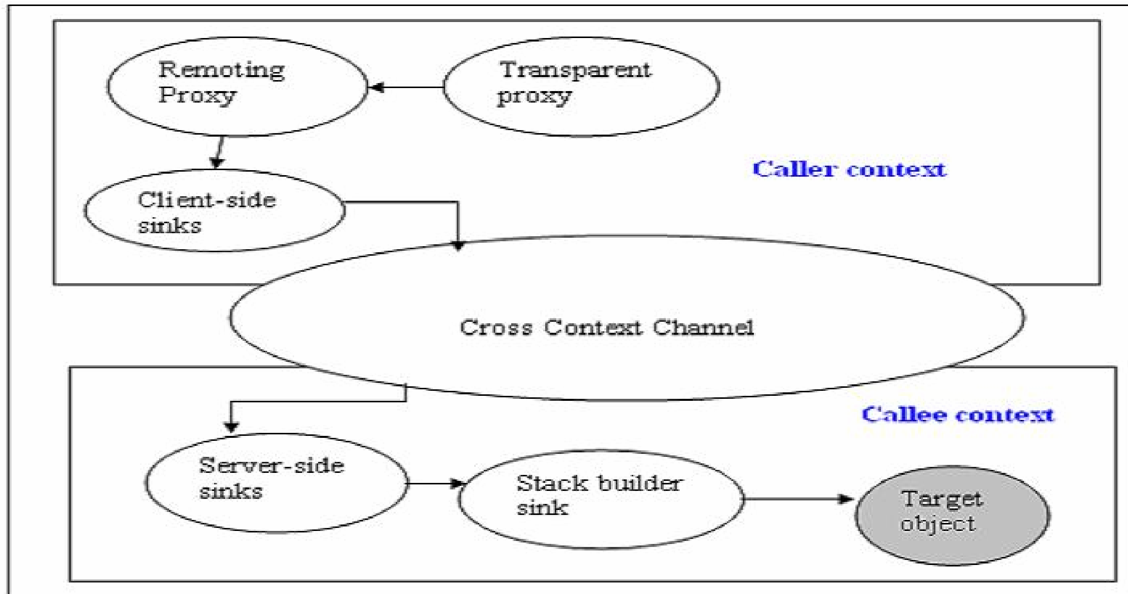

Figure 2.10: Cross-context message passing

The facility to customize proxies, and to intercept inter-context message passing are the primary features that facilitate AOP in .NET.

2.4.2. Context attributes:

Context attributes provide a facility to explicitly request context-based services. They are instantiated at the runtime, just before the type that is using it is instantiated. Context attributes are derived from System.Runtime.Remoting.Contexts.ContextAttribute class. Context attributes participate in the decision to create a new context (or not); at instantiation time, they install context properties and context sinks to provide application-specific service.

2.4.3. Context properties:

Context properties are instances of types implementing System.Runtime.Remoting.Contexts.IContextProperty interface. Context properties provide on-demand services to the developers. They can contribute the instantiation of interceptors (sinks), that can intercept the messages intended to the target object. Context properties can be programmatically accessed from within a type's methods. They provide an interface that allows developers to control sink behavior.

2.4.4. Context Sinks (Interceptors):

Context-properties can be programmed to provide sinks, on-demand. Sinks are instances of types implementing System.Runtime.Remoting.Messaging.IMessageSink interface. Sinks provide services by intercepting calls into and out of the context. .NET supports context-wide and object-specific interception. Method calls on context-bound object types are converted into messages that pass through sinks, where interception can be employed if needed. By employing interception at the sinks, we can modify the messages, or can even create a return-value thereby "short-circuiting" the method call.

The following diagram shows a typical interception library:

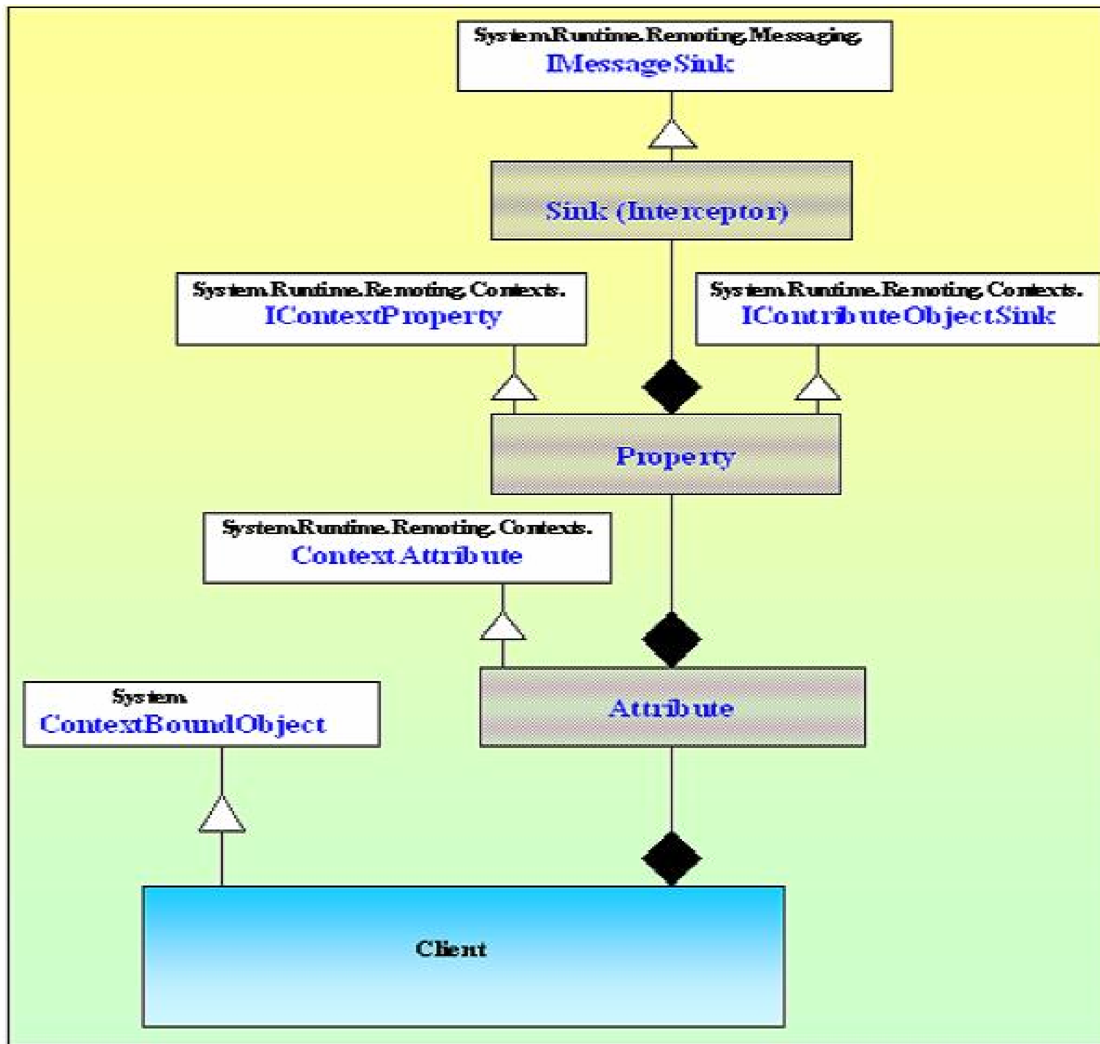

Figure 2.11: Class diagram for a typical inter-context interception library

2.4.5. Reflection:

Reflection is s a powerful feature of .NET, which Provides access to type-metadata embedded in each library and executable assembly. Reflection, among many other features, supports late binding. It even provides facility to create types at run-time! We will discuss more about reflection in the subsequent chapters.

2.5. Summary:

In this chapter, we discussed the .NET features that support interception. Interception is the primary technique in .NET that supports AOP. This chapter mainly serves as a reference for the .NET facilities for AOP. In subsequent chapters, we identify different aspects, in several applications, and isolate them into separate modules. We several different aspects and AOP techniques to isolate them. Finally, we make a qualitative analysis of the AOP code versus conventional non-AOP code, and draw conclusions about that.

3. METHOD SYNCHRONIZATION

Abstract:

In this chapter we present a new approach for method synchronization in .NET. Our approach is based on aspect-oriented programming. Here, we attempt to isolate the method synchronization aspect from client code involving method synchronization, into a separate module. As we show in this chapter, our method helps to develop client code that is more purpose-specific, and less tangled.

3.1. Introduction:

Consider the following operations: (1) *writing to a file*, and (2) *enqueue, dequeue operations on a queue*. These are two classical cases that require synchronization. In the first case, we make sure that one, and only one thread writes to a file at any time, and that it completes the required operation on the file before letting any other threads access the file. The other threads may be kept blocked during this time. Synchronization, in this example, is orthogonal to the purpose of the operation, namely writing to the file; but vital as well.

Synchronization is usually achieved via object synchronization, i.e., by locking the file in this case. In this chapter, we explore "Method Synchronization via AOP, in .NET".

3.1.1. Prior Work:

An internet search on http://www.google.com with the key words – "Method Synchronization" + "Aspect-Oriented Programming" (as of February 27, 2005) lists, among others, the following links:

http://www.ccs.neu.edu/home/lieber/courses/csg260/f03/lectures/lec6-csg260-f03.ppt , http://www.cs.wustl.edu/~schmidt/PDF/monitor.pdf. The articles on these web pages talk about Method Synchronization using Aspect-Oriented Programming. The Method Synchronization that these articles discuss is a way of synchronizing multiple threads.

Method Synchronization as dealt with in this chapter, though, is actually a way of synchronizing methods (belonging to the same or different types).

3.2. AOP-based Method Synchronization:

In this section, we discuss the advantages of an AOP-based approach for method synchronization as compared to conventional object synchronization approach. AOP-based method synchronization isolates the synchronization aspect from primary logic which results in a simpler implementation of the application.

As discussed in the previous section, synchronization in a queuing operation is totally extraneous to the method's primary purpose. Let us see the typical code for a class that handles queuing operations.

```
public class NonAOP_Q
{
	private Queue Q = new Queue(); // queue

	public int CountOfQ           // Count of the queue
	{
		get
		{
			int count = 0;
			Monitor.Enter(this.Q);
			try
			{
				count = this.Q.Count;
			}
			finally
			{
				Monitor.Exit(this.Q);
			}
			return count;
		}
	}

	public void EnQ(string msg)   // Enqueue a string to the queue
	{
		Monitor.Enter(this.Q);
		try
		{
			this.Q.Enqueue(msg);
		}
		finally
		{
			Monitor.Exit(this.Q);
		}
	}

	public string DeQ()           // Dequeue a string from the queue
	{
		string RetVal = null;
		Monitor.Enter(this.Q);
		try
		{
			if (this.Q.Count == 0)
				throw new Exception();
			else
				RetVal = (string)this.Q.Dequeue();
		}
		finally
		{
			Monitor.Exit(this.Q);
		}
		return RetVal;
	}
}
```

Code Listing 3.1: A queue-handler class that does not employ AOP

The class NonAOP_Q supports two public functions, EnQ and DeQ that provide interfaces for safe enqueuing and dequeueing to the private member queue. The class also supports a property, CountOfQ, which returns the size of the queue. The functions and properties ensure synchronization by locking the queue before their desired functionality is executed.

Extracting the synchronization element from the body of a function would, clearly, help in generating a less-complex code, and one that would capture the purpose of the code better. Shown below is the AOP-code that we have developed for this operation:

```
[EnableInterception(true)]
public class AOP_Q : System.ContextBoundObject
{
	private Queue Q = new Queue(); // queue

	[Sync("Q")]
	public int CountOfQ		// Count of the queue
	{
		get
		{
			return this.Q.Count;
		}
	}

	[Sync("Q")]
	public void EnQ(string msg)	// Enqueue a string to the queue
	{
		this.Q.Enqueue(msg);
	}

	[Sync("Q")]
	public string DeQ()		// Dequeue a string from the queue
	{
		if(this.Q.Count == 0)
			throw new Exception();
		else
			return (string)this.Q.Dequeue();
	}
}
```

Code Listing 3.2: Queue-handler class employing AOP

3. Method synchronization

The AOP-based code has managed to remove monitor-based locking from the function body. Again, AOP-based code captures the purpose of the code better: the methods do just their functions, enqueuing / dequeuing, without worrying about the synchronization factor, which as discussed, is an aspect, and has been successfully extracted into a separate module by the AOP-based code. Towards the end of this chapter, we demonstrate the efficacy of the AOP-based code through a quantitative analysis.

Synchronizing methods that belong to different types:

Object synchronization is usually taken care of by the member methods; i.e., the member methods of a type, in their function body, employ code for synchronizing their member objects; as illustrated in the locking of the Q by the EnQ and the DeQ methods in the non-AOP code discussed above. This is referred to as self-synchronization.

If the methods that require to be synchronized belong to different types, synchronization becomes more difficult. (This is not to say that it is impossible). There are privacy issues concerned with the object being locked – the declaring type of the method that requires to be synchronized has to let the foreign method access its member object, or provide an equivalent get-set method.

Synchronization of any two methods using AOP, whether they belong to the same type or different types, is simple. A method a1 ( ) of class X, can be quite easily synchronized with a method a2 ( ) of class Y.

```
class X { public void WriteNamesToDatabase ( ) {….} }

class Y { public bool WriteSomeStuffToTheSameDatabase ( ) {….} }
```

Code Listing 3.3: Two methods owned by two separate types that require synchronization

These methods need only register with the same ID to be mutually synchronous. The synchronized methods would look like:

```
[EnableInterception (true)]
class X : System.ContextBoundObject
{
      [Sync("DatabaseX")]
      public void WriteNamesToDatabase( ) {….}
}


[EnableInterception (true)]
class Y : System.ContextBoundObject
{
      [Sync("DatabaseX")]
      public bool WriteSomeStuffToTheSameDatabase( ) {….}
}
```

Registering with the same ID for synchronization, “DatabaseX” being the same ID in this case

Code Listing 3.4: Synchronizing methods belonging to two different types using AOP-based method synchronization

Again, there are times when we would want to synchronize methods that do not share a common object. Some display applications, for instance, require this. The lack of a common object makes it difficult to employ object-synchronization for such applications. Method synchronization does not demand the existence of a common object, and hence could prove to be an ideal choice for such purposes.

AOP-based method synchronization clearly captures and isolates the non-functional aspect of synchronization. The methods that need to be synchronized simply need to employ the custom synchronization attribute with a common ID. The synchronization element, which would otherwise be spread through the client code, is now extracted and taken out into a separate module. The client code now looks more purpose-specific, and less complex.

3.3. Achieving Method-Synchronization:

In the following section, we discuss how method-synchronization is achieved in a .NET environment.

3.3.1. .NET features that support Method-Synchronization:

We have employed interception of method-calls at run-time to achieve method synchronization. The .NET features that facilitate this purpose are facilities to:

1. *Develop and register custom metadata using attributes*.
2. *Trace metadata using reflection*.
3. *Intercept method calls using message sinks.*

3.3.2. Requirements:

To synchronize two or more methods using our current approach:

1. The parent class(es) must employ the EnableInterception[7] attribute; also it must derive from System.ContextBoundObject, to be context-bound in order to enable interception.
2. The methods must employ the Sync attribute, with an ID.

   If a method, say WriteIntsToFile(), needs to be synchronized with another method say WriteStringsToFile(), then they both must register to the Sync attribute using the same ID.

3.4. Theory:

In this section, we discuss how AOP-based method synchronization is achieved.

(1) Registration via Attributes:

The Synchronization attribute class, ***Sync***, maintains an ID storage. The Sync attribute is made applicable to methods only. When a method registers to the Sync attribute with an ID, the Sync class in turn checks if the ID is already present or not. If the ID is not present, the Sync class adds the ID to its ID store.

[7] EnableInterception and Sync are custom attributes that we have developed for synchronization applications.

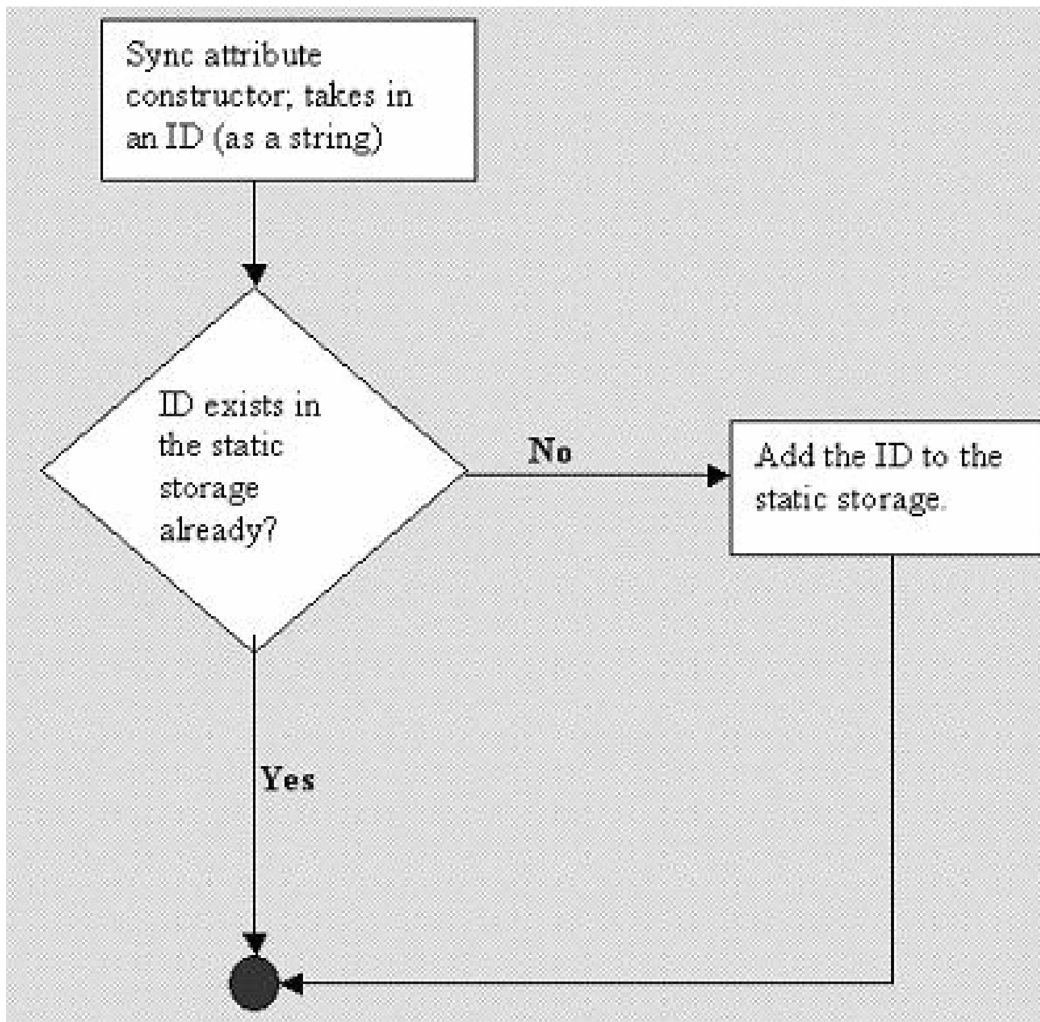

Figure – 3.1: Registering ID via attributes

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Property,
      Inherited = false, AllowMultiple = true)]
public class Sync : Attribute
{
      // The Id.
      private string Id;

      // The Id Store.
      private static StringCollection IdStore;

      // The static constructor. Initialize the Id Store.
      static Sync()
      {
            Sync.IdStore = new StringCollection();
      }

      // "Sync" attribute's constructor. pass the ID.
      public Sync(string Identity)
      {
            this.Id = Identity;            // set Id.
            this.InsertIfNewId(Identity); // If new Id, add to Id Store.
      }

      // Insert the id into the table, if the ID is NOT present already.
      private bool InsertIfNewId(string Identity)
      {
            foreach(string s in Sync.IdStore)
            {
                  if (s == Identity)
                        return false;
            }
            Sync.IdStore.Add(Identity);
            return true;
      }

      // ---- other functions…
}
```

Code Listing 3.5: Registering ID via attributes

(2) Tracing ID via Reflection:

As stated earlier, the parent class of the method has to employ the ***EnableInterception*** attribute to enable interception. When the method call is intercepted, the interceptor sink employs reflection, and traces the custom attributes of the method from its metadata.

(3) Querying for the lockee:

If the method has a locking attribute, ***Sync***, the interceptor sink gets its ID through reflection. The interceptor sink then queries the ***Sync*** attribute class with the ID of the current method. If the queried ID matches with any ID in the storage, the Sync attribute class hands over to the interceptor sink, a reference to that ID object in the storage. Please note that the ID object in the storage serves as a pseudo lock as it is not the actual object that requires to be locked. The term lockee refers to this pseudo-lock.

```
// Gets the object with the MethodCall's id.
// This object when locked, functions as a MutEx
private object getMutEx(IMethodCallMessage MsgCall)
{
      // Get the custom attributes.
      object[] attrs = MsgCall.MethodBase.GetCustomAttributes(false);
      Sync s = null;                  // The Sync attribute.
      object MutEx = null;      // The object to be returned.

      // Iterate through the custom attributes.
      foreach(Attribute attr in attrs)
      {
            s = attr as Sync; // Is it a Sync attribute?
            if (s != null)
            {
                  break;
            }
      }
      if (s != null)
      {
            // Overridden ToString func; returns the Id.
            string identity = s.ToString();

            // Get the MutEx object mapping to this identity.
            MutEx = Sync.GetMutEx(identity);
      }
      return MutEx;
}
```

Code Listing 3.6: Tracing ID, querying for the lockee

(4) Synchronization via locking at the interceptor sink:

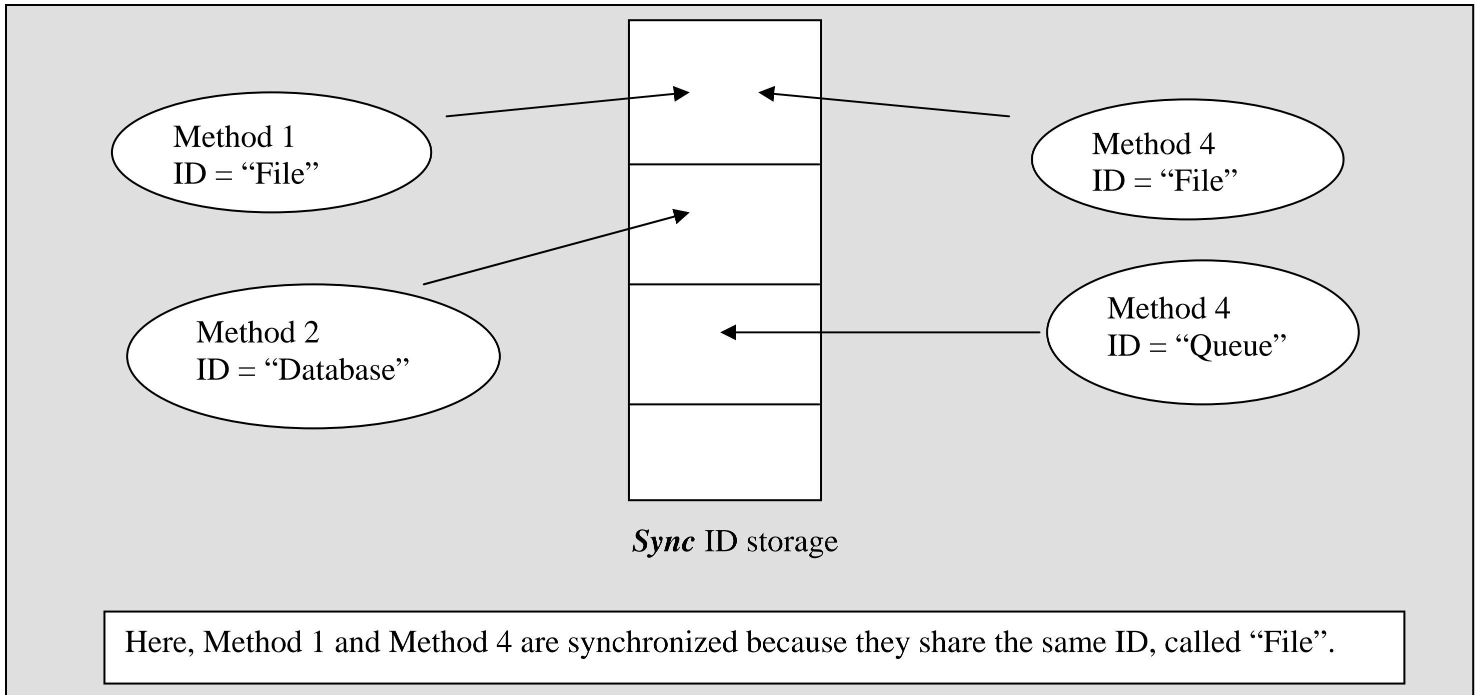

Figure 3.2: Synchronization via locking at the interceptor sink

Once the interceptor gets a reference to an object in the ID store, it locks that object, executes the method call, releases the lock, and returns the processed message. If in between, another method call with the same ID comes in, the ***Sync*** attribute hands to the method's interceptor sink, the same reference in its ID storage. The sink processing for this new method call tries to lock this object. But as it is already locked, it waits until the lock is removed.

```
public IMessage SyncProcessMessage(IMessage msg)
{
      // if NOT a method call message, return after default processing.
      IMethodCallMessage MsgCall =  msg as IMethodCallMessage;
      if(MsgCall == null)
      {
            return _nextSink.SyncProcessMessage(msg);
      }
      else
      {
            IMessage retMsg = null;                   // return message.
            object MutEx = this.getMutEx(MsgCall);    // get the MutEx object.
            if (MutEx != null)
            {
                  Monitor.Enter(MutEx);               // lock the MutEx object.
                  try
                  {                                   // default processing.
                        retMsg = _nextSink.SyncProcessMessage(msg);
                  }
                  finally
                  {
                        Monitor.Exit(MutEx);          // release monitor.
                  }
            }
            if (retMsg == null)
                  retMsg = _nextSink.SyncProcessMessage(msg);
            return retMsg;
      }
}
```

Code Listing 3.7: Synchronization via locking at the interceptor sink

3. Method synchronization

The accompanying diagram shows how method synchronization is achieved at the interceptor sink.

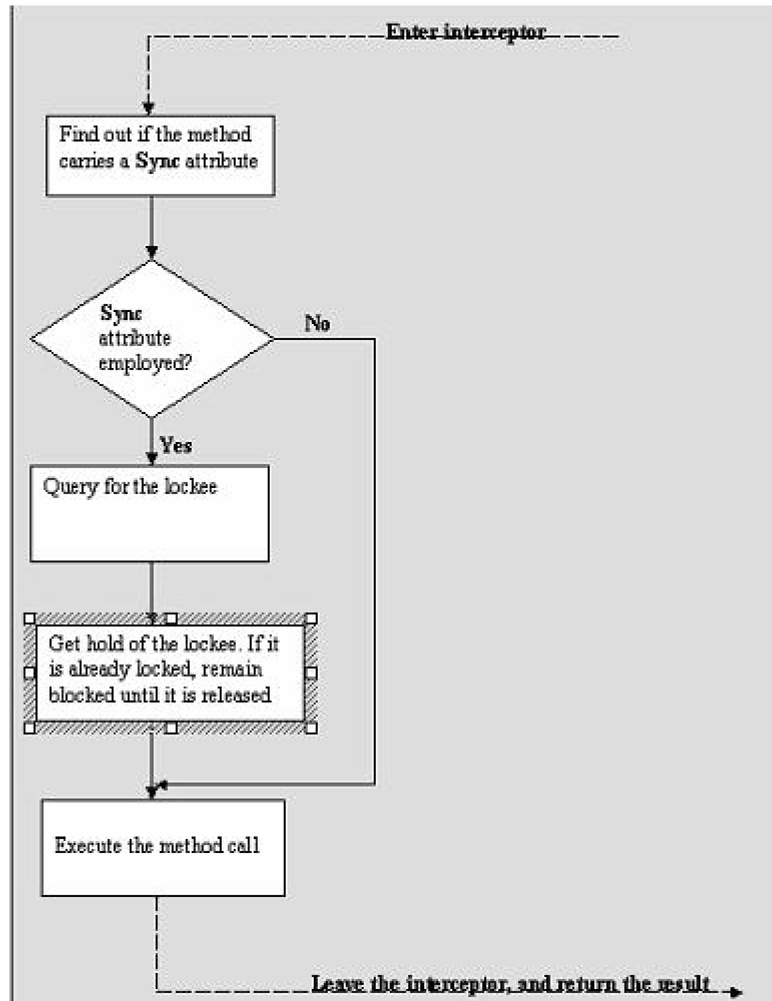

Figure 3.3: Achieving method synchronization by AOP

3.5. Architecture and Implementation:

As mentioned above, the Aspect-Oriented approach used here involves run-time interception of method-calls; this demands that the client class be context-bound, and hence enable interception. In addition, the client has to employ the services of the custom interception library.

The custom interception library consists of the following types:

- A custom context-attribute class called the **EnableInterceptionAttribute**. The custom context attribute achieves the following:
  - Creates a new context, if specified, for an instance of a type that employs this attribute. Since the instance is in an isolated context, cross-context calls can be intercepted if an interceptor sink is employed.

```
public override bool IsContextOK
      (Context MyContext, IConstructionCallMessage MyConstructor)
{
  // return false to create a new context. Else return true.

  return (false);
}
```

  - Provides an option to add context properties for the context.

```
public override void GetPropertiesForNewContext
      (IConstructionCallMessage MyConstructor)
{
  MyConstructor.ContextProperties.Add(new SyncContextProperty());
}
```

- A custom context-property class called **SyncContextProperty**. The custom context-property class lets us bind a custom interception sink to the context that would intercept incoming and outgoing calls.

```
public override void GetPropertiesForNewContext
      (IConstructionCallMessage MyConstructor)
{
  MyConstructor.ContextProperties.Add(new SyncContextProperty());
}
```

- A custom interceptor sink class called **SyncSink**. The custom interceptor sink performs the synchronization operation for every method that is intercepted and is tagged with the ***Sync*** attribute.

Any client type whose method(s) require(s) synchronization using the custom AOP synchronization library should meet the following requirements.

- The Type should inherit from `System.ContextBoundObject` class.
- The Type should apply `EnableInterceptionAttribute` attribute with true as the constructor's parameter.
- Any method(s) of the above-said type(s) that require to be synchronized with each other should apply the `Sync`[8] attribute with the same ID, the ID being of type string.

[8] Please refer to section 3.4.(1)

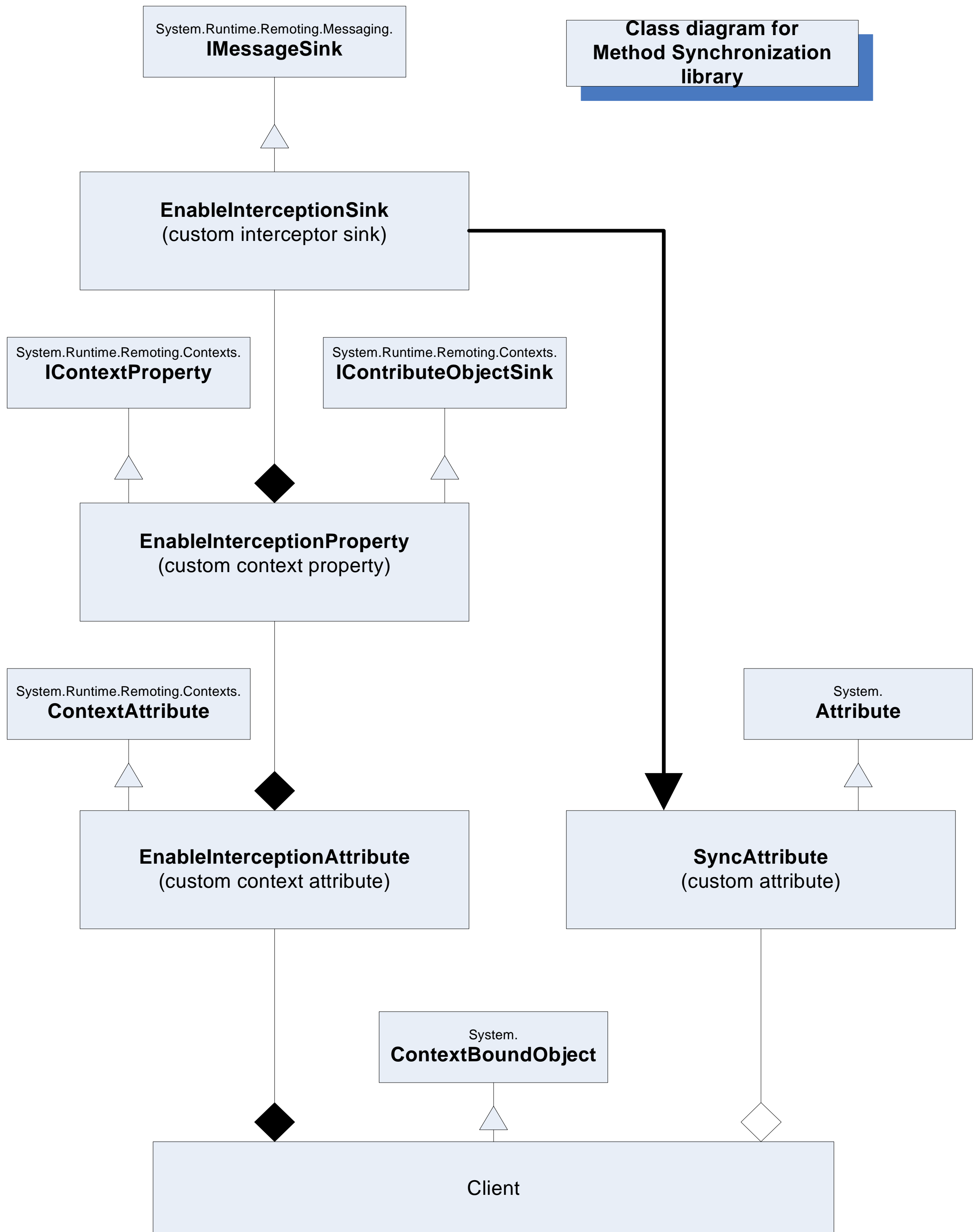

Figure 3.4: Class layout for the method synchronization library

3.6. Analysis:

To evaluate our approach, we have done code complexity analysis on a sample application. First, we created a conventional queue-handler class (Code Listing – 3.1), which does not employ AOP. We then developed an AOP-based queue handler class (Code Listing – 3.2). In the following section, we list the results that we obtained from the code complexity analysis.

The Non-AOP class which supports 3 functions (includes 2 methods and a property) bore a cyclomatic complexity of 7, and required 46 lines of function code. The AOP-based class, which supports similar functions, on the other hand, required just 18 lines of function code, with an aggregate cyclomatic complexity of just 4.

	AOP	Non-AOP
Lines of function code (Including properties)	18	46
Cyclomatic complexity	4	7

The AOP library, which we have developed, needs to be employed by the AOP-based classes to achieve method synchronization. This though is a *one-time* effort, that will be amortized over every application that users our AOP style Method Synchronization. We list here the analysis results for the Method-synchronization AOP library as well.

Lines of function code (Including properties)	130
Cyclomatic complexity	24

3.7. Conclusion:

The primary contribution of this chapter is the demonstration of a new way of achieving Method-Synchronization in a .NET environment. Our approach is based on Aspect-oriented programming.

We have developed a .NET library that enables client code developers to insert method synchronization aspect into their application, simply by including attributed before each method definition. The library helps client code developers remove synchronization elements from the client code, and plug it in as an attribute. This reduces complexity of the client code, and captures better its purpose.

Method-synchronization, as discussed here, isolates the synchronization aspect into a separate module. Method synchronization can be nicely encapsulated as Aspect-Oriented code. All that the client code developer needs to do is to employ the Synchronization attribute, and turn it on where synchronization is required. Isolation of the synchronization aspect results in less-complex, and more cohesive (purpose-specific) code. Method synchronization won't be as widely applied as object synchronization. Nevertheless, there are places where the former might prove to be more efficient than the latter.

4. DEBUGGING

Abstract:

Gregor Kiczales et al in their seminal paper on "Aspect-Oriented Programming" define an aspect as *"a property that tends not to be unit of a system's functional decomposition, but rather one that affects the performance or semantics of its components in systemic ways."* [1] In this chapter, we propose that debugging is an aspect, and that debugger applications can be effectively developed using aspect-oriented programming. For illustration, we use a 'custom debugger' that we have developed using AOP.

4.1. Introduction:

There are lots of commercial debuggers available today, each targeted for different languages and platforms. In this chapter, we propose a new way to develop such applications. Our approach is based on aspect-oriented programming. We propose that 'debugging' is an aspect, and that employing AOP is an effective way to develop debugger applications. Developing debugger applications via AOP enables the developer to programmatically interact with the debugger. For illustration, we use a 'custom debugger' that we have developed in .NET using AOP.

A debugger application helps a client-code developer to functionally interact with the code under execution. It should have at least some basic features, like capability to programmatically set break points, to restart the process being debugged, to print out a

stack trace, and to manipulate argument values. The custom debugger application that we have developed using AOP has facilities to:

1. Programmatically set break points
2. Manipulate input parameter values from sink stack
3. Print out the stack trace in case of an exception
4. Manipulate the return value
5. Ignore a return-exception, and substitute it with a dummy return value
6. Restart the application, all over again, at any point during execution

In the following sections, we discuss the theory behind the development of debugger applications using AOP.

4.2. Developing debugging applications using AOP:

The initial step in developing an application using AOP is identifying the aspect in the application. Within this paper, we propose that debugging is an aspect, and proceed to isolate the 'debugging' aspect into a separate module. As we show here, 'debugging' can nicely be segregated from client-code and developed into a separate module. The approach we have employed to isolate the debugging aspect involves run-time interception of method calls. In the following section, we discuss how .NET supports AOP.

4.2.1. Overview of AOP in .NET:

AOP is a relatively new method of programming, and not many languages support AOP. Serious research is being done on AOP with AspectJ and .NET. We intend to use .NET for our analysis. In this section, we discuss some .NET features that facilitate debugging with AOP.

1. Facility to develop and register custom metadata using attributes: .NET supports metadata. Metadata allows .NET languages to describe themselves automatically in a language-neutral manner, unseen by both the developer and the user [2]. Additionally, metadata is extensible through the use of attributes. Custom attributes let custom information be added to .NET types.
2. Facility to trace metadata, using reflection: Reflection is a technique provided by .NET that lets clients access metadata and type information of .NET types and assemblies. In addition, reflection supports mechanisms to dynamically build and bind types during run-time.
3. Facility to intercept method calls using message sinks: The fundamental domain of execution in .NET is the *Application Domain* [25]. Each process consists of at least one application domain, the default one. Application domains in turn are composed of one or more contexts. Calls across the context boundary involve serialization and message passing and are handled with lightweight proxies. The process of intercepting these cross-context calls at run-time is known as *interception*, and functions as the fundamental technique in our approach to aspect-oriented programming.

4. Facilities to manipulate messages in a sink stack using method call wrappers: .NET provides the two classes, **MethodCallMessageWrapper** and **MethodReturnMessageWrapper**, which facilitate manipulating messages in the sink stack. Message wrappers enable us to provide a feature that lets a user manipulate method call argument values, or method return value from the sink stack.
5. Facility to kill and start processes using process handles: .NET declares a **Process** class that enables the user to acquire handles to the applications on the system. In addition it helps create processes. This facility is used in our custom function debugger to stop the application being debugged, on user demand, and to restart the application.

4.2.2. Requirements:

(1) For a method call to be intercepted, the declaring type:

1. Must inherit from *System.ContextBoundObject*.
2. Must employ the *CustomDebuggerAttribute*, a custom attribute that we have developed to support debugging, and turn it on by passing a true as its constructor parameter.

```
[DebuggerContext(true)]
public class Closed : System.ContextBoundObject
{
  // Closed figure class.
}
```

Code listing - 4.1: Enabling interception - the client class has to inherit from System.ContextBoundObject, and employ the debugger attribute

(2) For setting break points at specific method calls:

1. The methods that need be intercepted must also employ the *CustomDebuggerAttribute*, and switch it on by passing a true to its constructor. To disable the break point, the attribute can be switched off, by passing a *false* to its constructor.

```
[DebuggerContext(true)]
public class Closed : System.ContextBoundObject
{
  [DebuggerContext(true)]
  public double ClosedCalcArea(double Area)
  {
      // Closed figure class
  }
}
```

Code listing - 4.2: Programmatically setting a break point; the method to be debugged has to employ the debugger attribute

4.3. Theory:

Consider the following steps involved in the operation of the custom-debugger:

(1) Object activation in a new context:

To illustrate the functioning of the custom-debugger, consider this sample client-code:

```
[DebuggerContext (true)]
public class Closed : System.ContextBoundObject
{
  [DebuggerContext (true)]
  public double ClosedCalcArea(double Area)
  {
    Console.WriteLine ("Inside Closed class's ClacArea function.");
    if (Area < 0)
      throw new ArgumentOutOfRangeException ("Area", "Area cannot be negative.");
    Console.WriteLine ("Input parameter is: " + Area + " ;\tMy area is: " + Area);
    return Area;
  }
}
```

Code listing - 4.3: A client class enabled for interception

The client-code employs a custom-debugger attribute, and also inherits from *System.ContextBoundObject,* and therefore qualifies for interception.

```
class Executive
{
  // Main.
  static void Main (string [] args)
  {
    Circle C = new Circle ();
    double CirArea = C.CircleCalcArea (1.0);
    Console.WriteLine ("\nThe returned value is " + CirArea.ToString ());
  }
}
```

Code listing – 4.4: The executive spawning an instance of a context-bound object

When an instance of a context-bound object that employs the debugger-attribute is created, a new context is spawned for that object. All the methods calls to that object, now involve inter-context message passing, and can be intercepted using message sinks.

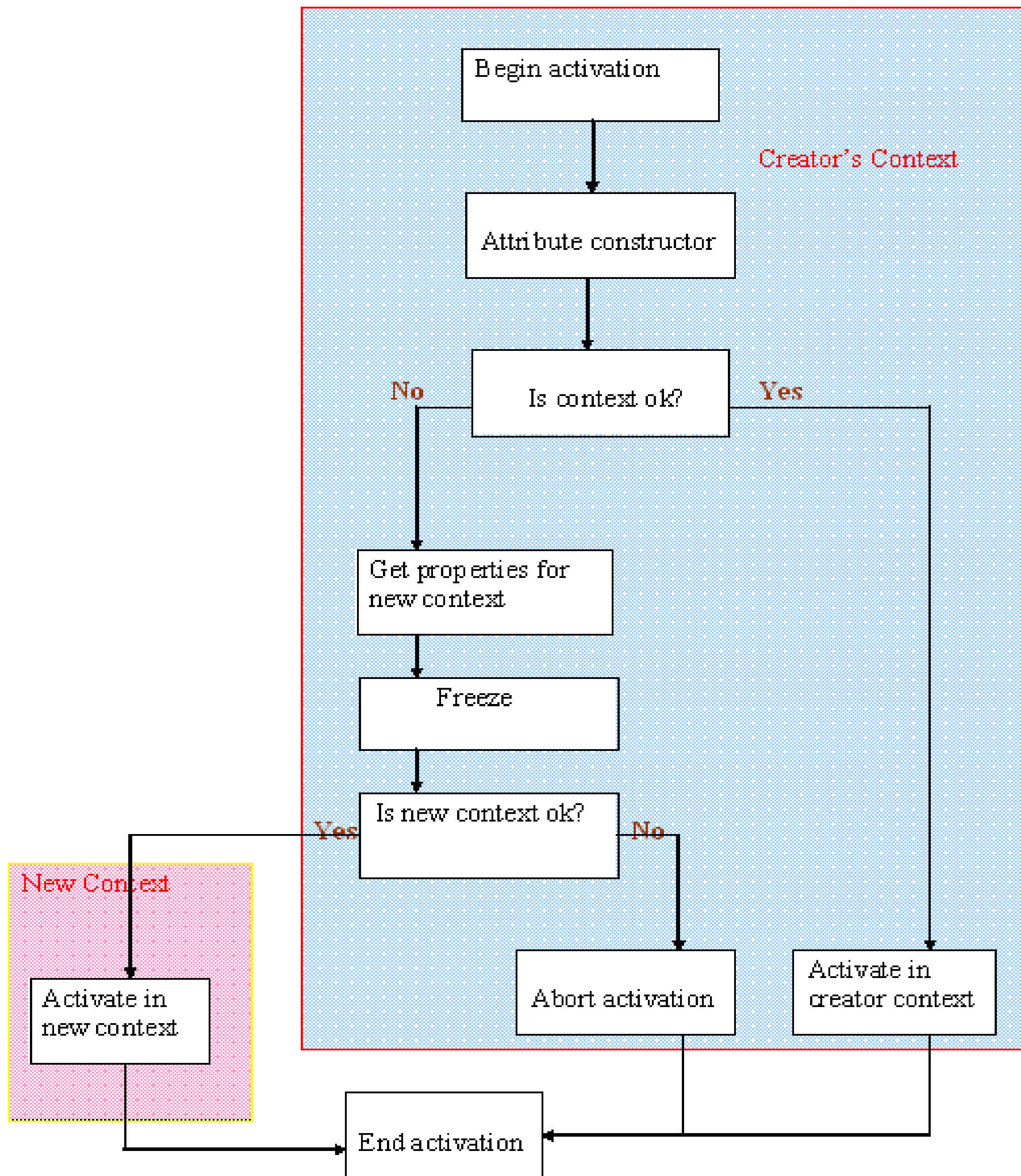

Figure 4.1: Context-bound object activation

(2) The interception library:

Once the object is ready for interception, all the calls to the object pass through the custom-sink that we have developed for this application. Shown below is the class layout for the interceptor library.

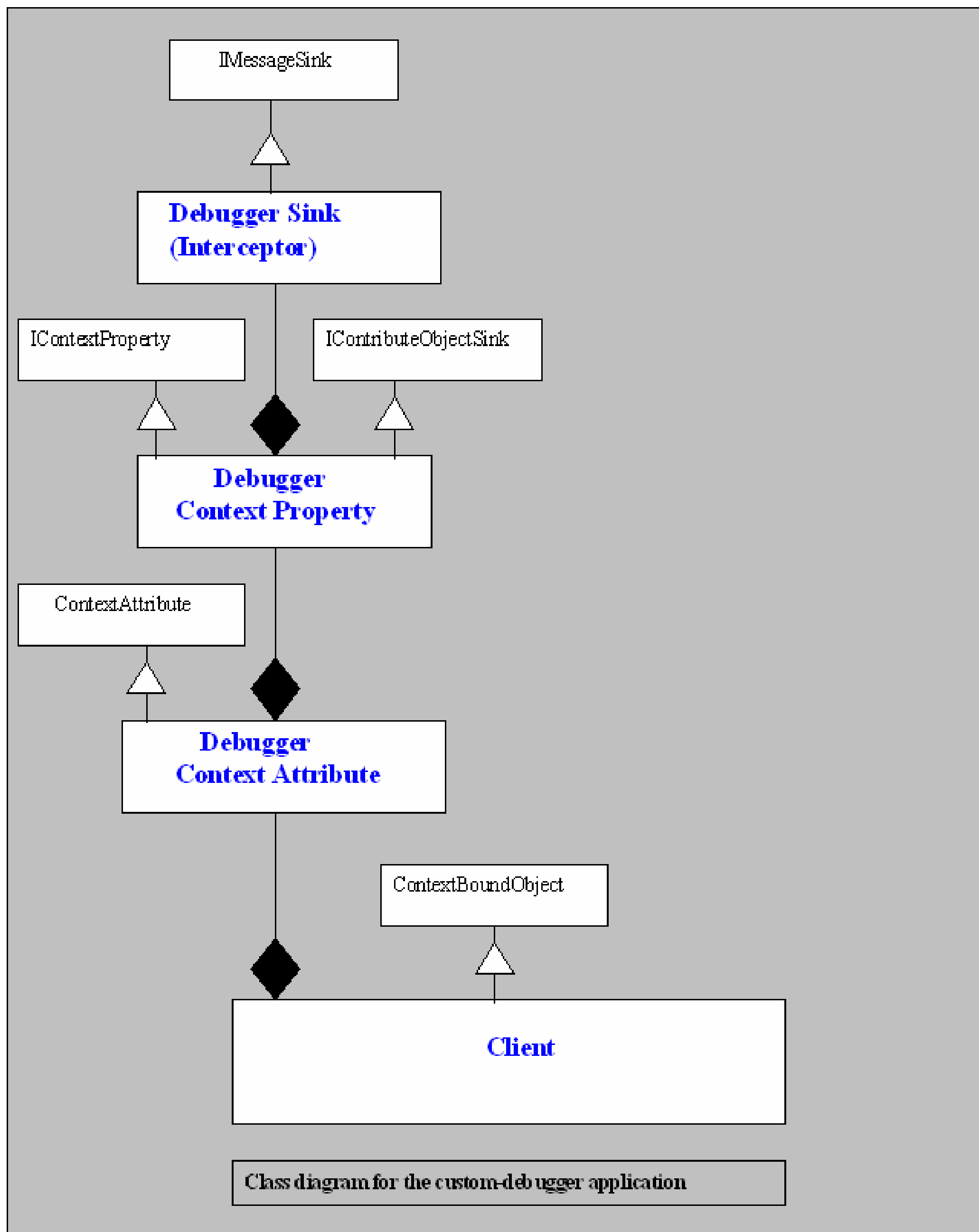

Figure 4.2: Class layout for the interception library

(3) Checking for a break point:

At the interceptor sink, the current method is searched for "CustomDebuggerAttribute", (not on its declaring type, but on the method itself). If the attribute is not present, or has *false* as its flag, the sink assumes that a break point is not specified, and returns the method call without any custom processing (with the default processing alone). If not, i.e., if the method carries the attribute with a *true* flag, the sink assumes the existence of a break point, and the method call qualifies for custom processing.

(4) Tracing and displaying metadata information:

Once the intercepting sink identifies a break point in a method call, it uses reflection to trace out its metadata information.

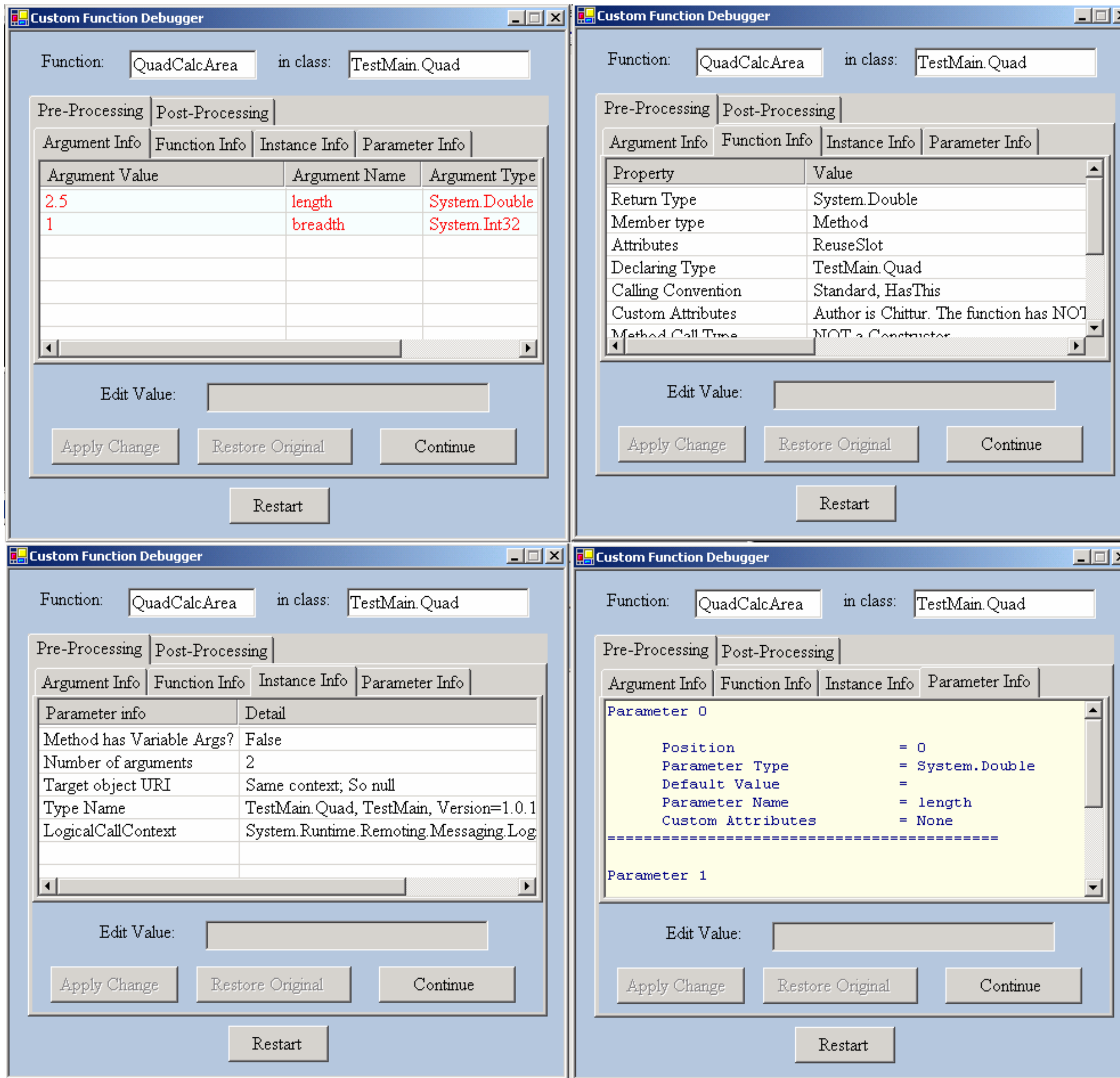

Figure 4.3: The custom function debugger displaying the metadata information of a method call

The intercepting sink makes use of .NET reflection to inquire about the metadata information of the method call and its declaring type. . Net objects carry metadata information with them; . Net reflection makes it available to any client upon querying. The code listing shows the code that employs reflection to trace the method info of a method call.

```
System.Reflection.MethodInfo MyMethodInfo =
MyType.GetMethod((string)MsgCall.Properties["__MethodName"], types);
```

Code listing – 4.5: Tracing metadata information using reflection

(5) Pre-processing:

Pre-processing is a major feature of interception. Pre-processing is the custom processing done on a method call at the intercepting sink, just before the call is invoked on the target object.

A lot of useful processing can be done at the pre-processing spot. The custom debugger provides the facility 'to edit the argument values to a method call', and it employs pre-processing to implement this feature.

Code employed for pre-processing is shown on the next page:

```
public IMessage SyncProcessMessage(IMessage msg)
{
  IMessage TempRetMsg;   // the temporary return message.
  IMessage TheMsg;       // processed message.

  do
  {
      // Pre-process the message.
      TheMsg = DebuggerProcess(msg);

      // Actual processing.
      TempRetMsg = _nextSink.SyncProcessMessage(TheMsg);

      // Post-process the message.
      TheMsg = ProcessRetMsg(TempRetMsg, (IMethodCallMessage)TheMsg);

  }while (this.ToContinueOrToGoBack);

  return TheMsg; // return the ReturnMessage.
}
```

Code listing - 4.6: Custom processing at the intercepting sink

The method **DebuggerProcess(msg)** does this custom pre-processing. The primary pre-processing implementation in this custom-debugger is the facility to edit input argument values. Manipulation of messages at the sink stack is done by using message-wrapping facility provided by .Net, as shown in code-listing 4.7.

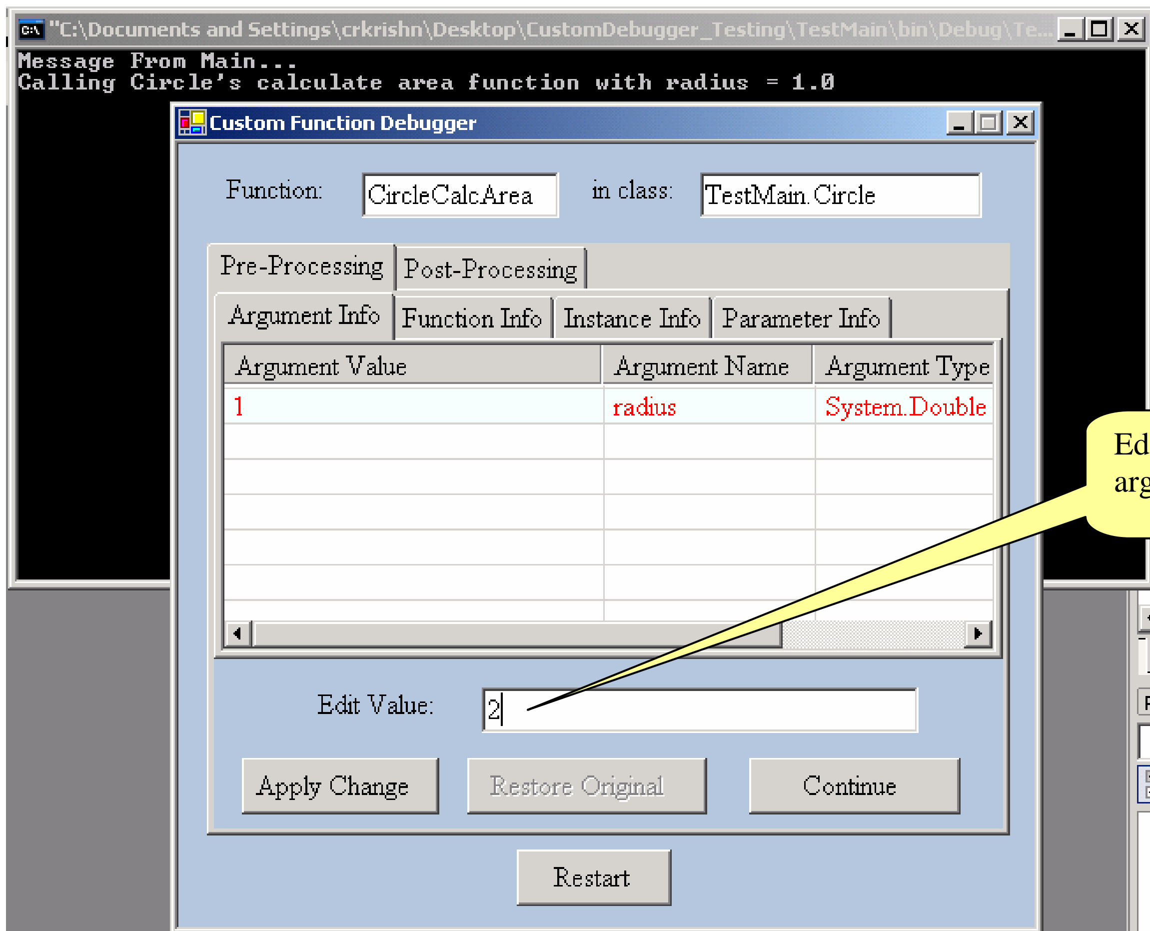

Figure 4.4: Editing the method call's argument value

. NET provides *message-wrappers* to manipulate messages from the sink stack. The custom-debugger employs *method call message wrapper* to facilitate pre-process editing. The method call then goes into the actual processing sink with the modified argument set.

Code for method call message manipulation at the sink stack is shown on the next page:

```
private IMessage MakeMessage(IMethodCallMessage MsgCall, ArrayList L)
{
  MethodCallMessageWrapper MyWrapper = new MethodCallMessageWrapper(MsgCall);
  object[] methodArgs = new object[L.Count];
  for(int count = 0; count < L.Count; count++)
  {
    methodArgs[count] = L[count];
  }
  MyWrapper.Args = methodArgs;
  return (IMessage)MyWrapper;
}
```

Code listing – 4.7: Manipulating a method call message at the sink stack using message wrappers

(6) Actual-processing:

After the custom pre-processing is done, the method call message is passed on to the default sink chain for actual processing.

```
      // Actual processing.
      TempRetMsg = _nextSink.SyncProcessMessage(TheMsg);
```

Code listing – 4.8: The actual method call

(7) Post-processing:

Post-processing is an equally useful feature of interception as pre-processing. *Post-processing is custom processing done on a method-call at the intercepting sink, just after the call returns from the target object and before it is returned to the initial invoker*. The intercepting sink traps the message on its return, and keeps it open for editing.

```
// Post-process the message.
TheMsg = ProcessRetMsg(TempRetMsg, (IMethodCallMessage)TheMsg);
```

Code listing – 4.9: Post-processing the message

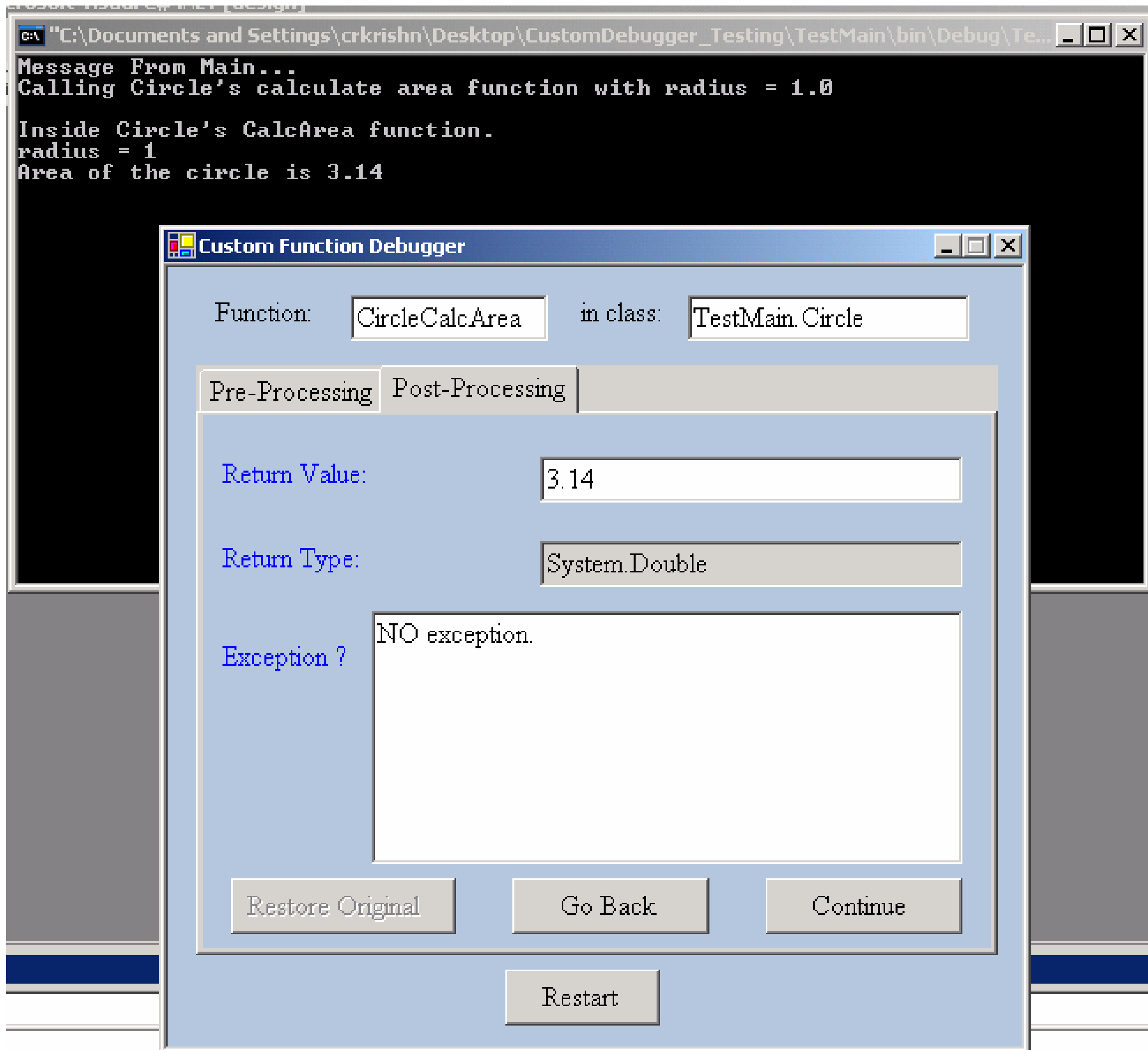

Figure 4.5: User-Interface waiting for user input to trigger post-processing

Post-processing can be put to use to do some effective manipulations, like exception-handling:

(7.1) Exception-handling:

Let us suppose that the method invocation resulted in an exception. The exception gets trapped in the intercepting sink, and is held over there for the user to manipulate. The user can choose to let the exception propagate to the upper level back to the invoker, or to negate it using a dummy value.

(7.2) Monitoring the stack trace:

The custom debugger emits the stack trace in case of an exception.

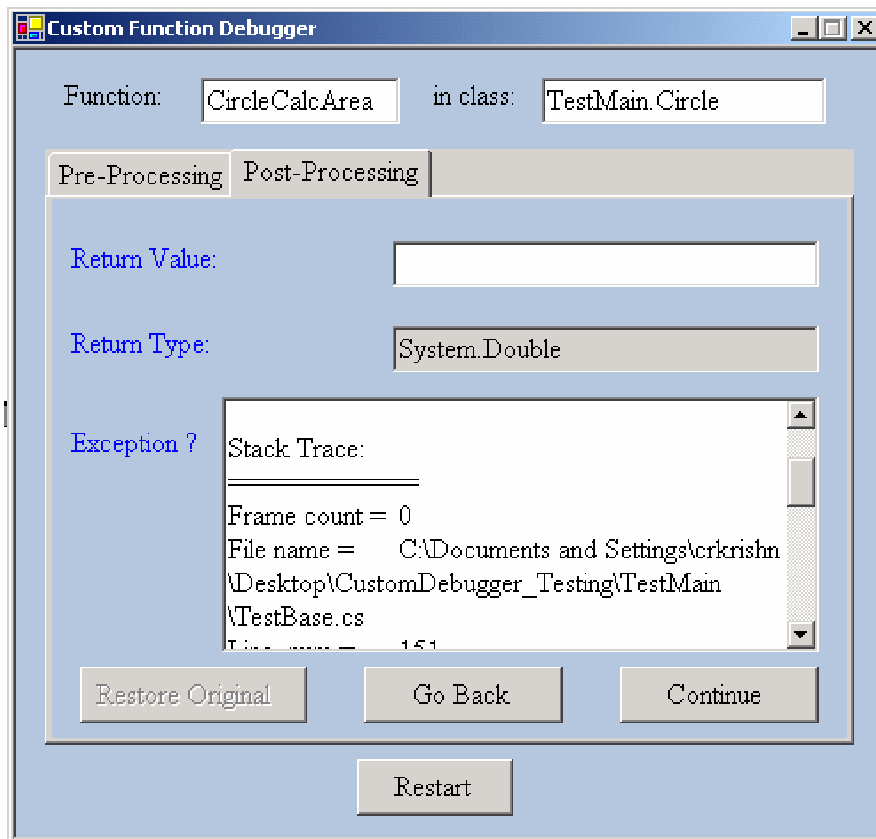

Figure 4.6: Custom debugger printing out the stack trace

```
string Excep;                  // Exception, if any
if (msg.Exception != null)
{
  Excep = "Exception Message:\n";
  // get 'stack-trace' details.
  System.Diagnostics.StackTrace ST = new
System.Diagnostics.StackTrace(msg.Exception, true);
  for(int count = 0; count < ST.FrameCount; count++)
  {
      System.Diagnostics.StackFrame SF = ST.GetFrame(count);
      Excep += "Frame count =\t" + count.ToString();
      Excep += "\nFile name =\t" + SF.GetFileName();
      Excep += "\nLine  num =\t" + SF.GetFileLineNumber().ToString();
      Excep += "\nIL offset =\t" + SF.GetILOffset().ToString();
      Excep += "\nNative offset =\t" + SF.GetNativeOffset().ToString();
      Excep += "\n-------------------------------\n";
  }
}
```

Code listing - 4.10: Dumping the stack trace

(7.3) Negation of a return exception:

The custom debugger provides a facility for negating a return exception. It uses *method return message wrapper*s to nullify the return exception, and to pass a dummy value in its place. While, substitution of an exception with a dummy value may not be the most desirable way to go, it helps to prevent the propagation of the exception out of the sink back to the initial invoker.

```
private IMethodReturnMessage MakeRetMessage(IMethodReturnMessage MsgCall,
IMethodCallMessage RequestMsg, Object newVal)
{
  if (MsgCall.Exception == null)
  {
      MethodReturnMessageWrapper MyWrapper = new MethodReturnMessageWrapper(MsgCall);
      MyWrapper.ReturnValue = newVal;
      return (IMethodReturnMessage)MyWrapper;
  }
  else
  {
      return new ReturnMessage(newVal, null, 0, null, RequestMsg);
  }
}
```

Code listing - 4.11: Manipulating a return message using

method return message wrapper

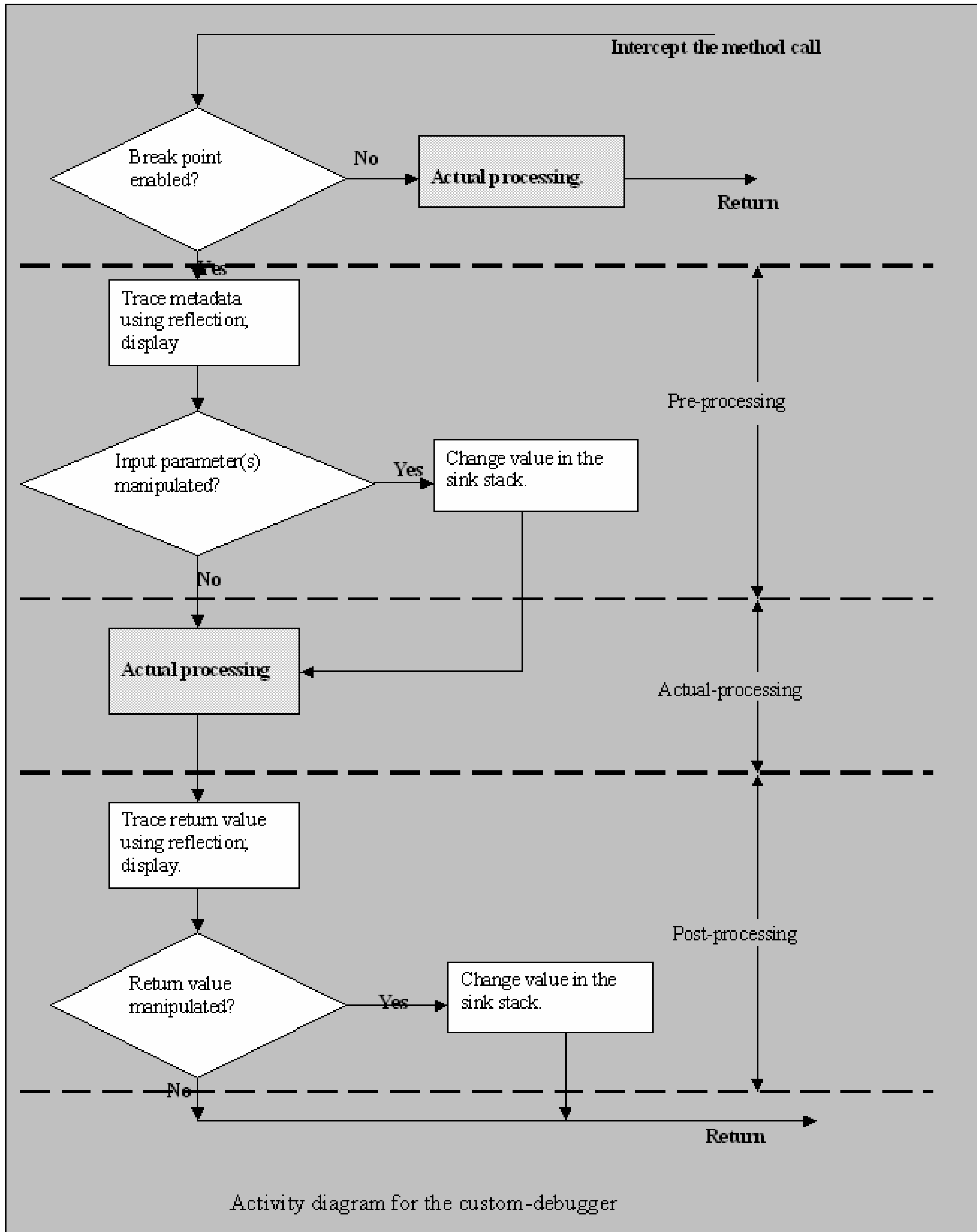

Figure 4.7: Top level Activity diagram for the custom debugger

(8) Restarting the application:

The custom function debugger provides a facility to programmatically restart the application, at any point of time during debugging.

```
try
{
  System.Diagnostics.Process P = Process.GetCurrentProcess();
  ProcessStartInfo PSI = new ProcessStartInfo(P.ProcessName + ".exe");
  Process NewP = Process.Start(PSI);
  P.Kill();
}
catch(System.Exception excep)
{
  // Could not restart process.
}
```

Code listing – 4.12: Restarting the application

4.4. Conclusion:

Debugging proves to be a viable aspect. The *custom function debugger* discussed in this chapter is a debugging application that has been developed using the Aspect-oriented approach and provides a reasonably rich set of debugging facilities.

Our contribution has been in identifying that debugging is an aspect. We have also developed a custom interception library which performs debugging services by intercepting method calls. The debugger library is a really useful tool for the client code developer. Break points can be turned on or switched off programmatically, by simply enabling or disabling the debugger-attribute. The AOP debugger provides facilities

a. To programmatically set break points,
b. To trace the metadata information of a method call and of its declaring type,
c. To manipulate messages from the sink stack, to modify method arguments.
d. To print out the stack trace of a method call,
e. To negate a return exception,
f. To restart the application.

5. CODE PROFILING

5.1. Introduction:

In a broad sense, Aspect-Oriented Programming is a way of programming that intends to assist a client code developer by isolating aspects into separate modules, which can then be plugged into the main application code to attain a code that is more purpose-specific, and one that is more reusable. In the previous chapter, we discussed the development of debugger applications using AOP. In this chapter, we propose that code-profiling and optimizing applications can be developed effectly using AOP. We discuss an Aspect-Oriented approach to develop code coverage applications. We also illustrate a execution-time profiler application developed using AOP.

5.2. Code Coverage:

In this chapter, we define *code coverage analysis as the process of discovering methods in a program that are not examined using a set of test cases*. Code coverage analysis provides a quantitative measure of code coverage, which is an indirect measure of the quality of the tests.

Test drivers are test executive codes that test functioning of a program. As far as OOP is considered, good test drivers test the methods in a class thoroughly. The two primary goals of a test driver are to test all the *methods* in a program, and to test if they handle probing test cases supplying both good and bad inputs gracefully and as desired. Code

coverage analysis is a means to help the test driver designer achieve the first purpose, namely finding out if *all* the methods in the program have been tested. In the following section, we discuss how code coverage can be achieved by AOP. We use the code coverage analyzer that we have built for the purpose of illustration.

5.2.1. Theory:

Our code coverage analyzer uses run-time interception of method calls to perform code coverage analysis. In addition, the analyzer uses .NET reflection to trace out custom attributes on a method to see if it should be included or exempted from coverage analysis.

5.2.1.1. .NET features that support code profiling:

The .NET features that aid code coverage are facilities to:

- **Develop and register custom metadata using attributes.**
- **Trace metadata using reflection**.
- **Intercept method calls using message sinks**.

5.2.1.2. Prerequisite:

For executing code-coverage analysis on program to be tested, classes should employ the ***Profile*** attribute, a custom attribute developed for this application. In addition, they have to be context-bound to enable context-interception, and hence must derive from *System.ContextboundObject* class. The methods that require to be profiled should employ the ***Profile*** attribute as well.

In order to employ the services of the AOP profiling library, the client code developer needs to do the following:

- Any method that requires profiling should employ the ***`Profile`*** attribute, with the Boolean value **true** passed as its constructor parameter.
- The declaring types of the methods must inherit from the type ***`System.ContextBoundObject`***.
- The declaring types of the methods must also employ the ***`profile`*** attribute, with the Boolean value **true** passed as its constructor parameter.

5.2.1.3. Functioning:

Let us see how the aspect-oriented interception library achieves code coverage. When an instance of a client class, inheriting from *ContextBoundObject*, and employing the Profile attribute is created, the object becomes context-bound and is ready for interception by the custom interceptor sink.

Once the interceptor sink traps a method call, it traces out all the member methods of its declaring type. The interceptor sink maintains a list of the parent type, and its member methods. The sink then tracks if all methods of the class were invoked at least once.

```
/////////////////////////////////////////////////////////
// Processes Synchronous method calls.
public IMessage SyncProcessMessage(IMessage msg)
{
      this.Analyze(msg);
      return _nextSink.SyncProcessMessage(msg);
}

/////////////////////////////////////////////////////////
// Processes Asynchronous method calls. Returns the processed message crtl.
public IMessageCtrl AsyncProcessMessage(IMessage msg, IMessageSink
replySink)
{
      this.Analyze(msg);
      return _nextSink.AsyncProcessMessage(msg, replySink);
}
```

Code listing 5.1: Processing functions at the interceptor sinks

```
////////////////////////////////////////////////////////////
// Analyze the method call message.
private void Analyze(IMessage msg)
{
IMethodCallMessage M = msg as IMethodCallMessage;
if (msg == null)
      return;

// Get the type that declares this method.
System.Type T = M.MethodBase.DeclaringType;
if( ! (Depot.MainStore.ContainsKey(T.ToString())))
{
      System.Reflection.MethodInfo[] MIarray = T.GetMethods();
      Hashtable temp = new Hashtable();
      foreach(MethodInfo MI in MIarray)
      {
            object[] attrs = MI.GetCustomAttributes(false);
            foreach(object a in attrs)
            {
                  ProfileAttribute PA =  a as ProfileAttribute;
                  if ( ! (PA == null))
                  {
                        if (PA.Apply)
                        {
                              if (MI.ToString() == M.MethodBase.ToString())
                                    temp.Add(MI.ToString(), "1");
                              else
                                    temp.Add(MI.ToString(), "0");
                        }
                  }
            }
            //Console.WriteLine(MI.ToString());
      }
      Depot.MainStore.Add(T.ToString(), temp);
}
else
{
      Hashtable temp = Depot.MainStore[T.ToString()] as Hashtable;
      if (temp.ContainsKey(M.MethodBase.ToString()))
      {
            temp[M.MethodBase.ToString()] = "1";
      }
      Depot.MainStore[T.ToString()] = temp;
}
```

Code listing 5.2: Code-coverage method called at the interceptor sink. *Note: The code listing for the data structure class Depot is not shown here.*

5.3. Time profiling:

In this chapter, *we define time profiling as the process of finding out the time of execution of a method call with a specified set of parameters*. The time profiler collects data about how much time is consumed by the different methods in the program, with any specified argument list. This information enables the programmer to find out if the execution times of the various functions are acceptable. The approach used here is run-time *time profiling*, and may help the programmer to develop code meeting its performance specifications.

5.3.1. Theory:

The time profiling interception library also uses run-time interception of method calls to perform timing analysis.

5.3.1.1. .NET features that support code profiling:

We have employed interception of method-calls at run-time to achieve time profiling. The .NET features that facilitate this purpose are facilites to:

- ***Develop and register custom metadata using attributes.***
- ***Trace metadata using reflection.***
- ***Intercept method calls using message sinks***.

5.3.1.2. Prerequisite:

For executing time-profiling analysis on code, the code's classes should employ the *Profile* attribute, a custom attribute developed for this application. In addition, they have to be context-bound objects to enable context-interception, and hence must derive from *System.ContextboundObject* class. The methods that require to be profiled should employ the profile attribute as well.

In order to employ the services of the AOP profiling library, the client code developer needs to ensure the following conditions:

- Any method that requires profiling should employ the ***`Profile`*** attribute, with the Boolean value true passed as its constructor parameter.
- The declaring types of the methods must inherit from the type ***`System.ContextBoundObject`***.
- The declaring types of the methods must also employ the ***`profile`*** attribute, with the Boolean value true passed as its constructor parameter.

5.3.1.3. Functioning:

As discussed above the time profiling library uses run-time interception of method calls. This is very similar to the code coverage application. There are quite a few differences in the functioning of the custom sink though. We have detected some critical issues in time profiling, and have proposed the solutions as well:

1. Issue: Accuracy of time measurement:

Solution - The interceptor sink uses a high-resolution timer (least count = 1 micro second) to determine the time of execution of any intercepted method call.

2. Issue: Inter-context message passing:

For the sake of execution-time measurement of the method calls, we make the declaring objects context-bound. Now the method calls involve inter-context message passing. At the time of actual execution, we disable the context-boundedness of the object, and let it be context-agile. This would mean that while taking the measurements the method calls involve inter-context message passing, while at time of actual execution there is no message- passing as the objects are context-agile.

Solution – To reduce the inaccuracy associated with the difference between the two scenarios, we make dummy inter-context method calls which involve no actual processing – just the making of the method call and the return; we call this time period, the *null interval*. Then for each inter-context method call made, we subtract the null interval from the measured value. This provides an accurate result. [*Please see appendix 3*.]

5.4. Conclusion:

Our contribution has been to identify that AOP supports code-coverage applications very well. All that the client-code developer needs to enable profiling is to decorate the class / methods with custom attributes. Code-coverage is hidden very well by the aspect-oriented interception library. A time-profiling application illustrated in the chapter goes well with the aspect-oriented approach as well. There are a couple of issues related to the accuracy of the timing measurement though. The issues have been discussed, and the solutions proposed. The solutions are satisfactory, and answers the questions associated with any inaccuracy in time-measurement.

6. PRIORITIZATION

6.1. Introduction:

Distributed computing, usually, involves message passing through channels. Sockets and .Net remoting are built on these byte-oriented channels to simplify the communication interface. Though these techniques have managed to achieve their purpose, there is a possibility that some elements, though *less important*, get hidden during the simplification process. We have explored .NET remoting to investigate some of the aspects involved in distributed computing, to isolate them, and to provide the client code developer with an interface that captures these elements.

This chapter discusses one of the services that can be potentially implemented effectively using Aspect Oriented Programming, namely **Prioritization**, or **Priority Assignment**. Two specific cases of Prioritization that are considered here are **Client prioritization** and **Message Prioritization**.

6.2. Background:

Consider a typical remoting system, where many clients talk to one server. Due to its limited processing potential, a single-server environment requires effective and well-defined priority criteria – for its clients, and for the messages it needs to process. Prioritization, hence, may be a crucial issue in a single-server environment, or generally under any system where the server's workload is quite heavy compared to its processing power.

On the other hand, Priority assignment of its messages is *not* the actual business of the clients. The client machine's primary functionality is to send request messages across to the server for processing, and to receive the processed message. Similarly, prioritizing its clients is *not* the actual business of the server, though the ideal situation would be that the server has a provision for processing the high priority clients first. Further, it would be usefult if all remoting systems had a provision for prioritizing messages and clients, though each new developer would prefer not to have to produce code developed solely for this purpose.

These considerations lead us to believe that **prioritization** is indeed an ***aspect*** of a remoting system – "*a property of the system that tends not to be a unit of the system's functional decomposition, but rather one that affects the performance or semantics of the components in systemic ways"* [1]. In the following section, we discuss how client prioritization can be achieved through aspect-oriented programming.

6.3. Client Prioritization:

A server might need to handle requests from various clients; of these some clients might require higher priority than the others. *Client prioritization is the feature that enables the server to process messages from different clients based on the priority levels of those clients*. For instance, in the case of a simple company LAN, client prioritization may mean that the requests from an administrator be given higher priority than requests from staff personnel.

In the following sections, we discuss how prioritization can be achieved in a .NET Remoting system.

6.3.1. Client Prioritization architectural overview:

The goal of our client prioritization library is to let the server process clients based on their priorities; a server interceptor makes sure that messages from high priority clients are sent to a high priority queue, and those from low priority clients go to a low priority queue.

The client prioritizer, server, client, and remotable object library, also contain a client interceptor, a server interceptor and a custom formatter library. The priorities of the clients are set in an XML file residing at the server.

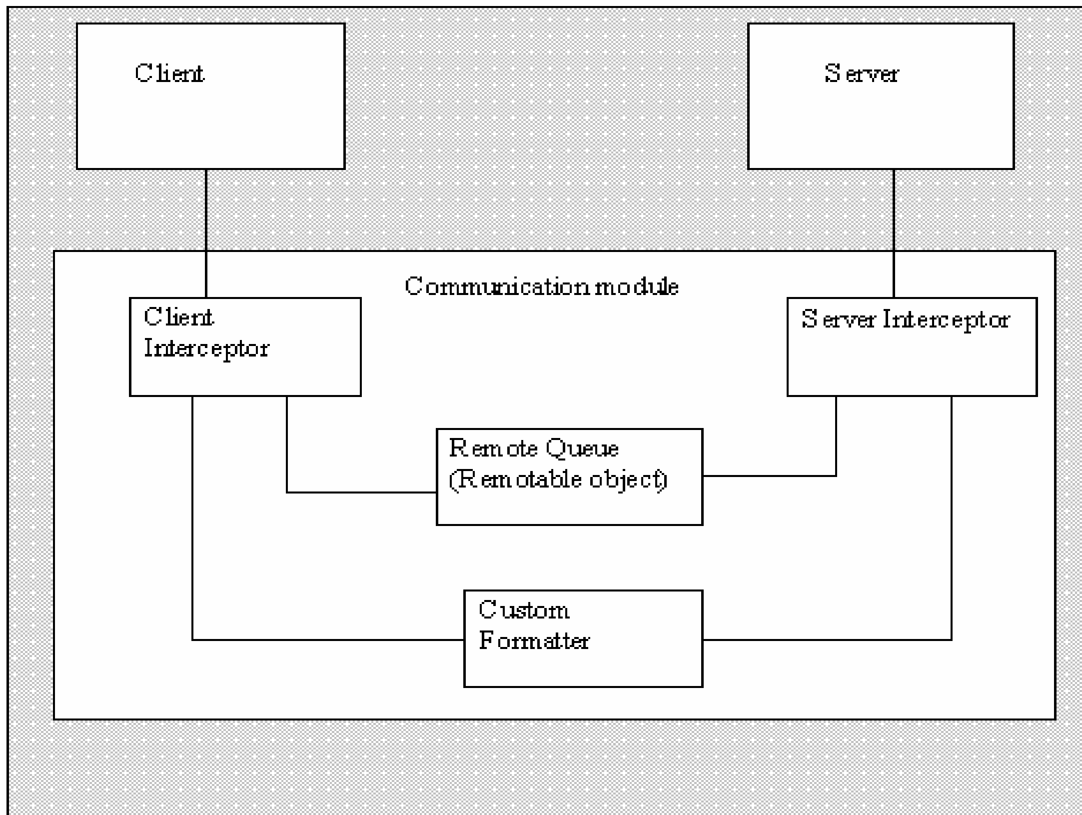

Figure 6.1: Module layout for the client prioritizer

6.3.2. Theory:

As shown in the module layout, the communication library employs custom formatters and interceptors in addition to a remotable object library. Next, we show how interceptors capture the client prioritization aspect.

(1) Custom formatter:

The custom formatter[9] defines custom serialization and deserialization methods, which let us transfer custom information along with the actual message. The custom formatter lets us wrap custom data in special headers, which can be unwrapped at the receiver end. Code for the custom serialization and deserialization functions is shown below.

```
public override void Serialize(Stream serializationStream,
      object graph)
{
      _writer = new StreamWriter(serializationStream);

      // first, schedule the graph object for serialization
      // Schedule just registers the object with the object id
      // generator if it has not already seen the object and places
      // it in a queue of objects for later serialization.
      //.we'll retrieve the objects in the queue via GetNext()
      Schedule(graph);

      // get the next object to be serialized and serialize it
      object oTop;
      long topId;
      while( (oTop = GetNext(out topId)) != null)
      {
            // execution of the WriteObject method will likely result
            // in the sceduling of more objects for serialization
            WriteObject(oTop, topId);
      }

      _writer.Flush();
}
```

Code listing 6.1: Serialization method of the custom formatter class

[9] The custom formatter that we use in this application was developed by **Scott McLean, James Naftel, and Kim Williams**, and is listed in the book **Microsoft .NET Remoting (Microsoft Press)** [29].

```
public override object Deserialize(System.IO.Stream serializationStream)
{
	//
	// create an object manager to help with deserialization
	_om = new ObjectManager( _surrogateselector, _streamingcontext );

	_reader = new StreamReader(serializationStream);

	// read objects until end of stream
	while( _reader.Peek() != -1 )
	{
		ReadObject();
	}

	//
	// now we can do fixups and get the top object
	_om.DoFixups();

	// return top most object
	return _om.GetObject(1);
}
```

Code listing 6.2: Deserialization method of the custom formatter class

(2) Client interceptor:

Client prioritization is primarily a server-oriented aspect. Therefore, the client interceptor does not play a major role, but is involved in formatting messages to support decoding at the server end.

```
public IMessage SyncProcessMessage ( IMessage msg )
{
      //
      // serialize message to a stream using the custom formatter, MyFormatter.
      TransportHeaders    requestHeaders = new TransportHeaders();

      Stream requestStream = _NextChannelSink.GetRequestStream(msg,
            requestHeaders);
      if ( requestStream == null )
      {
            requestStream = new System.IO.MemoryStream();
      }

      RemotingSurrogateSelector rem_ss = new RemotingSurrogateSelector();
      MyFormatter fm = new MyFormatter();
      fm.SurrogateSelector = rem_ss;
      fm.Context = new StreamingContext( StreamingContextStates.Other );

      // serialize a MethodCall message to the stream
      MethodCall mc = new MethodCall(msg);
      fm.Serialize(requestStream, mc);

      // let sink chain process the message
      ITransportHeaders   responseHeaders = null;
      System.IO.Stream    responseStream = new System.IO.MemoryStream();

      this._NextChannelSink.ProcessMessage(    mc,
            requestHeaders, requestStream, out responseHeaders, out responseStream);

      // use our version of IMessage to deserialize
      fm.SurrogateSelector = null;

      // the formatter special cases IMessage types and
      // deserializes them as MyMessage types.
      MyMessage mr = (MyMessage)fm.Deserialize(responseStream);
      return mr.ConvertMyMessagePropertiesToMethodResponse(mc);
}
```

Code listing 6.3: Custom serialization of messages at the client interceptor

(3) Server interceptor:

The server interceptor is the most important library in the client prioritizer. First of all, when a message comes in, the server interceptor extracts the IP address of the client.

```
string uri = (string)mymsg.Properties["__Uri"];
```

Code listing 6.4: Tracing the client's Uri

The interceptor then employs *context interception* to validate the IP. *Validating the IP* means discovering the priority of the IP by checking an xml file residing at the server end. Context interception code for IP validation is shown below.

```
// Priority validation service.
public bool GetPriority(string ClientIP)
{
	Context ctx = Thread.CurrentContext;
	ContextLogAndValidateProperty Validate =
(ContextLogAndValidateProperty)ctx.GetProperty("ContextLogAndValidateProperty");

	string RetVal = Validate.getValue(ClientIP);

	if (RetVal == "1")
		return true;
	else
		return false;
}
```

Code listing 6.5: Context interception code for IP validation

The interceptor employs a context attribute to create a new context. The context attribute in turn aggregates a context property, which does the IP validation.

```
// logging and validation class
[AttributeUsage(AttributeTargets.Class)]
class ContextLogAndValidateProperty : Attribute, IContextProperty
{
	bool FileOK; // If the validation file could be opened and read from.
	ArrayList Clients; // the clients.
	ArrayList Priority; // their priority.

	public ContextLogAndValidateProperty()
	{
		Clients = new ArrayList();
		Priority = new ArrayList();
		try
		{
			// load the file into DOM.
			XmlDocument doc;
			doc = new XmlDocument();
			doc.Load("Validate.xml");
			XmlNodeList RootList = doc.GetElementsByTagName("Client");
			foreach (XmlNode Root in RootList)
			{
				string val = Root.InnerText;
				char[] sep = {':'};
				string[] AfterSplit = val.Split(sep, 2);
				Clients.Add(AfterSplit[0]); // add the cleints.
				Priority.Add(AfterSplit[1]); // add their priority.
			}
			FileOK = true;
		}
		catch
		{
			FileOK = false;
		}
	}
	// other functions
	//-----
}
```

Code listing 6.6: Context property class doing IP validation

Once the server interceptor extracts the IP's priority, it directs the message to the high priority queue or low priority queue based on client's priority.

```
// check if the call is the dummy function.
if ((string)mymsg.Properties["__MethodName"] == "enQRecvQ")
{
	if (ClientPriority) // check if high priority client
	{
		// enqueue in the high priority queue.
		mymsg.Properties["__MethodName"] = "enQHighPQ";

		// using the logging service.
		this.Logger("Client IP = " + ClientIP + " ; Client Priority = High.");
	}
	else // low priority client
	{
		// enqueue in the low priority queue.
		mymsg.Properties["__MethodName"] = "enQLowPQ";

		// using the logging service.
		this.Logger("Client IP = " + ClientIP + " ; Client Priority = Low.");
	}
}
```

Code listing 6.7: Priority assignment for clients

The server interceptor thus involves “Inter- Application Domain” interception to trace the client IP, and also “Inter-Context” interception to check client priority.

Figure 6.2: Class diagram for the Client prioritizer

6.3.2.1. Inserting the sink into the remoting chain:

Inserting the custom interceptor sinks into the sink chain is done by making some simple changes to the configuration file as shown below:

```
<configuration>
	<system.runtime.remoting>
		<application>
			<channels>
				<channel ref = "http" >
				</channel>
			</channels>
		</application>
	</system.runtime.remoting>
</configuration>
```

Code listing 6.8: Configuration file for client NOT employing interceptors

```
<configuration>
	<system.runtime.remoting>
		<application>
			<channels>
				<channel ref = "http" >
					<clientProviders>
<formatter type="ClientInterceptor.MyFormatterClientSinkProvider, ClientInterceptor"/>
					</clientProviders>
				</channel>
			</channels>
		</application>
	</system.runtime.remoting>
</configuration>
```


Code listing 6.9: Configuration file for client employing interceptors

6.4. Message Prioritization:

In a distributed computing system, clients send different types of messages to the server. Of these, some require higher priority than the others. *Message prioritization is the feature that enables a client to set a priority level to the messages being sent to the server*. For instance, a 'file transfer' message might be of lower priority an error message.

6.4.1. Architectural overview:

The modular structure for the message prioritizer is essentially the same as that for the client prioritizer. In message prioritization, the client interceptor ensures the messages leaving a client are assigned definite priorities. In that sense, message prioritization is a client-oriented aspect. The client interceptor is the primary library that captures the message prioritization aspect.

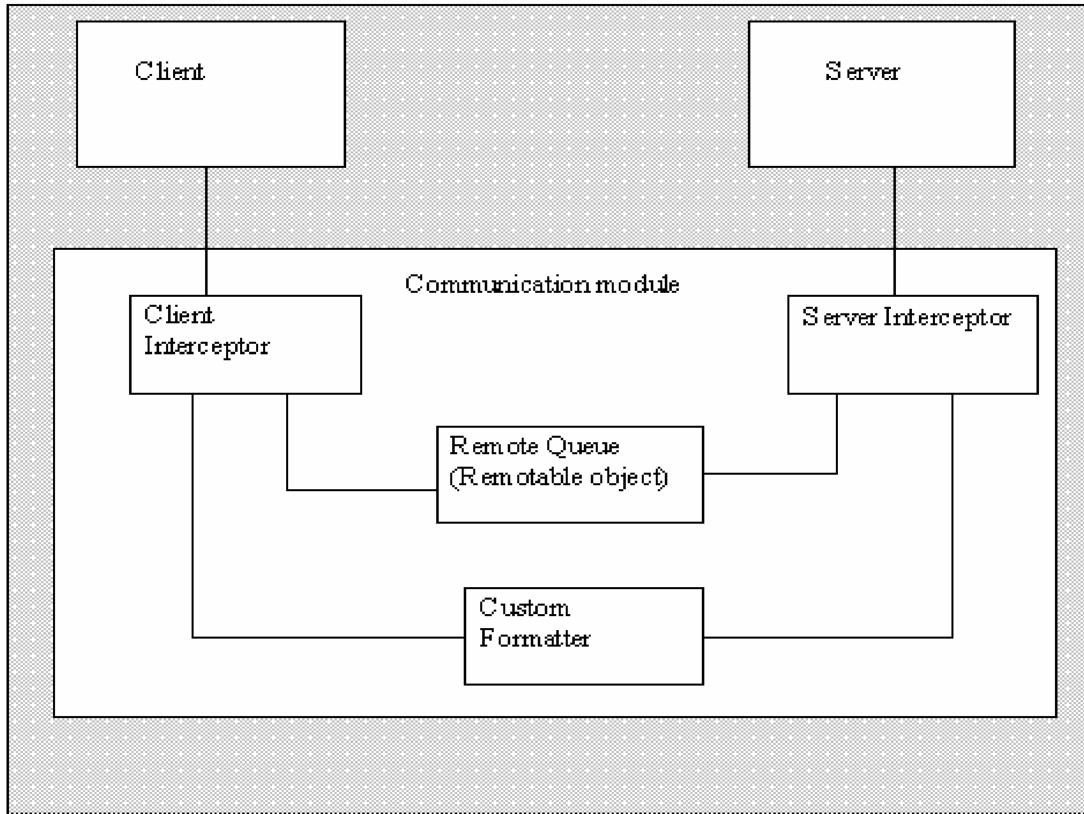

Figure 6.3: Module layout for the message prioritizer

6.4.2. Theory:

As shown in the module layout, the communication library employs custom formatters and interceptors besides the remotable object library. The custom formatter, as discussed above, provides methods for custom serialization and deserialization. Let us see how the client prioritizer captures the message-prioritization aspect.

The client interceptor maintains a 'validation file' containing the priorities associated with different methods. When an outgoing call is intercepted, the client sink reads the designated priority for the particular message, from the validation file, and appends the priority tag. Each message at the client egress thus has its priority assigned. The server interceptor doesn't have a significant role over here; it just needs to decode the message, using the custom formatter instead of the default way.

Message prioritization as implemented here is less aspect-oriented in that, the interception library has to be aware of all the messages types, (or the client code developer might need to have the priorities entered in a file). That, aspect-oriented code needing to know client code particulars, makes message prioritization less aspect-oriented than client prioritization.

There is a proposed enhancement for the client prioritizer. Instead of analyzing message specifics, the interception library can use the priority level of the thread that generated the outgoing call, and accordingly make the priority assignment. There is a certain level of inexactitude associated with this technique as well because, the priority level of the parent thread might not always be the same as the desired priority level of the message.

```
public IMessage SyncProcessMessage ( IMessage msg )
{
      // serialize message to a stream using the custom formatter, MyFormatter.
      // --- Code for that, omitted from this listing... ---

      /****************************************/
      // Implementing Message prioritization...
      MyMessage TempMsg = new MyMessage(msg);
      MethodCall mCall = new MethodCall((IMessage)TempMsg);

      // get the Xml message sent by the worker function of the client
      string xmlMessage = (string)mCall.Args[0];

      // get the priority.
      if (this.GetPriority(xmlMessage))
      {
            TempMsg.Properties["__MethodName"] = "enQHighPQ";
      }
      else
      {
            TempMsg.Properties["__MethodName"] = "enQLowPQ";
      }

      // since the MyMessage object, TempMsg, was changed
      //          make the new method call message again.
      MethodCall mc = new MethodCall((IMessage)TempMsg);

      /***************************************/
      // Serialize the message.
      fm.Serialize(requestStream, mc);

      /******************************************/
      // adding run time information...
      // Encoding the Thread priority by creating our own header tag...
      requestHeaders["Thread_Priority"] =
                        Thread.CurrentThread.Priority.ToString();

      //
      // let sink chain process the message
      // --- Code for that, omitted from this listing... ---

      // use our version of IMessage to deserialize
      // the formatter special cases IMessage types and
      // deserializes them as MyMessage types.
      // --- Code for that, omitted from this listing... ---
}
```

Code listing 6.10: Priority assignment for messages

6.5. Conclusion:

This chapter discusses 'prioritization' service implemented effectively using Aspect Oriented Programming. Our primary contribution in this chapter has been in identifying that prioritization can be classified as an aspect of a distributed computing system, and can be successfully isolated into a separate module using AOP. We have explored AOP-based client prioritization and message prioritization as two specialized cases of priority assignment.

In this chapter, the architectural and design details of the interception implementation have been discussed - Client prioritization using AOP involves server-side inter-Application Domain interception, and inter-context interception; Message prioritization using AOP involves client-side inter-Application Domain interception. The isolation of the prioritization aspect into the interceptor code is also explained. The client-priority settings reside in a database file at the server, and can be conveniently modified when necessary.

7. QUANTITATIVE ANALYSIS

7.1. Introduction:

In the previous chapters, we identified different *aspects*, isolated them into separate modules, and came up with various AOP-based code examples. In the current chapter, we make a quantitative analysis of several AOP examples based on code complexity and cohesion.

Let us quickly review the aspects that were dealt with in the previous chapters: In the first chapter, we explained a *Range-Checker* application, involving the "Range-Checking (validation)" aspect. The second chapter discusses technical support for AOP in .NET, and does not talk about any aspect in particular. The third chapter explores the "Method Synchronization" aspect. Chapter four explores "Debugging" as an aspect, and explains a *Debugger* application developed using AOP. In the fifth chapter, we talk about AOP-based "Code-Profiling" applications. The sixth chapter explores "Prioritization" and "Validation" as aspects of a remoting system.

In the following sections we perform quantitative analysis on three different sets of applications - a Range-Checker application, (based on "Range-Checking" aspect described in the first chapter), a Queue-Synchronization application (based on the "Method Synchronization" aspect explained in the third chapter), and a File-Center application (based on "Prioritization/Validation" aspect described in the Sixth chapter).

7. Quantitative analysis

We have developed, for each of the three applications described above, an AOP version and a corresponding non-AOP version for analysis of code-complexity and cohesion.

An interesting outcome of our exploration of the *Debugging* and *Code-Profiling* aspects was the difficulty that we faced in developing non-AOP codes for these applications; we have not performed quantitative analysis on these two aspects – we explain why toward the end of this chapter.

The different aspects that were dealt with in the previous chapters and whether or not quantitative analysis was performed on a set of applications based on those aspects are listed in the following table:

Chapter	Aspect dealt with in chapter	Whether or not quantitatively analyzed
1	Range-Checking (Validation)	Performed
2	-	-
3	Method Synchronization	Performed
4	Debugging	Not performed Reason: Difficult to develop non-AOP code
5	Code Profiling	Not performed Reason: Difficult to develop non-AOP code
6	Prioritization	Performed

Table 7.1: Lists the aspects that were dealt with in this thesis work, and whether or not quantitative analysis was performed on a set of applications based on the aspect.

7.2. Analysis:

(a) Range-Checker application: In chapter 1, for the purpose of illustration, we discussed a range-checker application using AOP. We have developed a corresponding Non-AOP application, and have conducted analysis on the two sets of code. A short description of the application design is given below:

We developed a sample class that contains three methods.

- ***Geometry_FindSectorArea*** **-** finds the area of a sector of a circle, given the radius and the angle of the sector.
- ***Banking_FindSimpleInterest*** **-** finds the simple interest given the principal, the number of years, and the rate of interest.
- ***Trignometry_FindCosine*** **-** finds cosine of an angle in a triangle, given the adjacent side and the hypotenuse.

Let us quickly review how these AOP and non-AOP codes implement the methods, revisiting the description provided in chapter 1 – (The description is repeated here for clarity):

Consider a method in a type called *Geometry* that finds the area of a sector of a circle. We pass the *radius* and the *angle* of the sector as parameters; the returned value of area should be half of the square of radius times angle. Again, the method has to make sure that the radius and angle passed as parameters are non-negative; here, we make an

assumption that the angle is *between 0 and 2 PI*. Otherwise, the method has to throw an exception. Let us see how we implement the method in the conventional way:

```
public double Geometry_FindSectorArea(double radius, double angle)
{
  if (radius < 0.0)
      throw new ArgumentOutOfRangeException("radius", "The radius is out of range.");
  if ( (angle < 0.0) || (angle > 6.28) )
      throw new ArgumentOutOfRangeException("angle", "The angle is out of range");
  return (radius * radius * angle / 2);
}
```

Code Listing 7.1: Non-AOP implementation of the Geometry_FindSectorArea method

Here, the actual *function* of the method is just:

```
return (radius * radius * angle / 2);
```

The remaining piece of code is orthogonal to the primary purpose of the method. Isolating the *range-checking* aspect from the code would, clearly, make the code more readable, more purpose-specific, and less complex.

This is nicely achieved by the AOP code, which is shown below:

```
[method: Range(true, Lower = 0.0)]
public double Geometry_FindSectorArea([Range(true, Lower = 0.0)] double radius,
                        [Range(true, Lower = 0.0, Upper = 6.28)] double angle)
{
  return (radius * radius * angle / 2);
}
```

Code Listing 7.2: AOP implementation of the Geometry_FindSectorArea method

The code ***clearly describes*** its function. The parameter *radius* has a range-checker attribute, named `Range`, which specifies its lower bound as 0. The *angle* parameter has a range that lies between 0 and 2 PI. Again the method attribute to the method restricts the *lower bound of the return value* to 0.

The AOP-based method is more readable, and clearly captures the purpose of the code. The AOP-based implementation of the method provides better cohesion – All that the method body does is to find the area of the sector as described by its method name, `Geometry_FindSectorArea`. On the contrary, the non-AOP based implementation of the method has poorer cohesion in that in addition to serving its primary function, which is finding the area of the sector, the method does range-checking on its input parameters.

Code complexity and cohesion analysis were done on the other methods described above as well. The analysis results are listed below:

The Non-AOP code:

Lines of function code = 20
Cyclomatic complexity = 11
Cohesion = 6 tasks

The AOP client-code:

Lines of function code = 3
Cyclomatic complexity = 3
Cohesion = 3 tasks

The AOP server-code (interception library code):

Lines of function code = 204
Cyclomatic complexity = 33

Range-Checker Interception library code

Lines of common AOP function code = 67
Cyclomatic complexity of common AOP code = 7

AOP code applicable to any context-interception library

Lines of application specific AOP function code = 137
Cyclomatic complexity of application specific AOP code = 26

AOP code specific to the *Range-Checker* application

Table 7.2: Analysis results for the range-checker application

In the table above, the non-AOP code used three functions that had two tasks each for a total of 6 tasks. The AOP code used three functions that had one task each for a total of 3 tasks.

(b) Queue-Synchronization application: In chapter 3, we explored Method-Synchronization using AOP. For quantitative analysis, we developed two versions of a synchronization application involving a queue-handler class – one without using AOP, and the other employing AOP.

Chapter 3 describes two types NonAOP_Q, and AOP_Q each of which contain two methods, one for enqueuing a string into the member queue, and the other to dequeue a string from the member queue; the types also support a property for getting the count of the member queue. Queuing and dequeuing operations generally require synchronization – the methods and the property are synchronized, with the type NonAOP_Q using conventional non-AOP methods for synchronization, and the type AOP_Q employing the services of an interception library that takes care of synchronization.

The technique to achieve AOP-based method synchronization and the theory behind it has been explained in chapter 3. In the following section, the program codes for the two types are listed followed by the analysis results (The program code listing and description are repeated here for clarity):

```
public class NonAOP_Q
{
	private Queue Q = new Queue(); // queue

	public int CountOfQ		// Count of the queue
	{
		get
		{
			int count = 0;
			Monitor.Enter(this.Q);
			try
			{
				count = this.Q.Count;
			}
			finally
			{
				Monitor.Exit(this.Q);
			}
			return count;
		}
	}

	public void EnQ(string msg)	// Enqueue a string to the queue
	{
		Monitor.Enter(this.Q);
		try
		{
			this.Q.Enqueue(msg);
		}
		finally
		{
			Monitor.Exit(this.Q);
		}
	}

	public string DeQ()		// Dequeue a string from the queue
	{
		string RetVal = null;
		Monitor.Enter(this.Q);
		try
		{
			if (this.Q.Count == 0)
				throw new Exception();
			else
				RetVal = (string)this.Q.Dequeue();
		}
		finally
		{
			Monitor.Exit(this.Q);
		}
		return RetVal;
	}
}
```

Code Listing 7.3: A queue-handler class that does not employ AOP

The class NonAOP_Q supports two public functions, EnQ and DeQ that provide public interfaces for safe queuing and dequeuing to the private member queue. The class also supports a property, CountOfQ, which returns the size of the queue. The methods and properties ensure synchronization by locking the queue before their desired functionality is executed.

Extracting the synchronization element from the body of the functions would, clearly, help in generating a less-complex code, and one that would capture the purpose of the code better. Shown below is the AOP-code that we have developed for this purpose:

```
[EnableInterception(true)]
public class AOP_Q : System.ContextBoundObject
{
	private Queue Q = new Queue(); // queue

	[Sync("Q")]
	public int CountOfQ		// Count of the queue
	{
		get
		{
			return this.Q.Count;
		}
	}

	[Sync("Q")]
	public void EnQ(string msg)	// Enqueue a string to the queue
	{
		this.Q.Enqueue(msg);
	}

	[Sync("Q")]
	public string DeQ()		// Dequeue a string from the queue
	{
		if(this.Q.Count == 0)
			throw new Exception();
		else
			return (string)this.Q.Dequeue();
	}
}
```

Code Listing 7.4: Queue-handler class employing AOP

Let us specifically consider the “DeQ” method – In the non-AOP application, the method “DeQ” suffers from poor cohesion; the method has to take care of three tasks –

1. Lock the queue – **secondary task**
2. Check if the number of elements in the queue is zero; if so, raise an exception – **secondary task**
3. Perform the dequeue operation, and return the dequeued string – **primary task**

In the AOP implementation of the method, cohesion is improved – the method synchronization aspect is isolated and delegated to the interception library. The method now has to perform only two tasks –

1. Check if the count of the queue is non-zero; if not, raise an exception – **secondary task**
2. Perform the dequeue operation, and return the dequeued string – **primary task**

The AOP-based code has managed to remove monitor-based locking from the function bodies. Again, AOP-based code captures the purpose of the code better: the methods do just their functions, enqueuing / dequeuing, without worrying about the synchronization factor, which as discussed, is an aspect, and has been successfully extracted into a separate module by the AOP-based code.

The following table lists the analysis results based on the two versions of the synchronization application.

The Non-AOP code:

Lines of function code = 37
Cyclomatic complexity = 12
Cohesion = 7 tasks

The AOP client-code:

Lines of function code = 9
Cyclomatic complexity = 6
Cohesion = 4 tasks

The AOP server-code:

Lines of function code = 120
Cyclomatic complexity = 26

Queue-Synchronization Interception library code

Lines of common AOP function code = 67
Cyclomatic complexity of common AOP code = 7

AOP code applicable to any context-interception library

Lines of application specific AOP function code = 53
Cyclomatic complexity of application specific AOP code = 19

AOP code specific to the *Queue-Synchronization* application

Table 7.3: Analysis results for the Queue-Synchronization application

(c) File Center application: In chapter 6, we explored “Prioritization”, and “Validation” aspects. We have developed a “File-Center” application based on prioritization, and validation aspects. The “File Center” application is a simple distributed file management system where users with different privilege levels can access files residing at a centralized server.

The different operations that can be done to the files at the server are

- GetFile – check out a file from the server
- PutFile – check in a file to the server
- DeleteFile – delete a file residing at the server

Users fall into three categories:

- Administrator – has the privilege to carry out any of the above three operations
- Manager – has the privilege to check in or check out a file; does not have the privilege to delete a file
- Staff – has the privilege of checking out a file; does not have the privilege to check in a file or to delete any.

We have developed two versions of the application – one without employing AOP, and one based on AOP. We have performed analysis on the two sets of codes. The application design is described here followed by the analysis results:

The Non-AOP file center application consists of a server, a client, a remotable library, a priority validator class, and a file handler class. The following diagram shows the class diagram for the Non-AOP file center application:

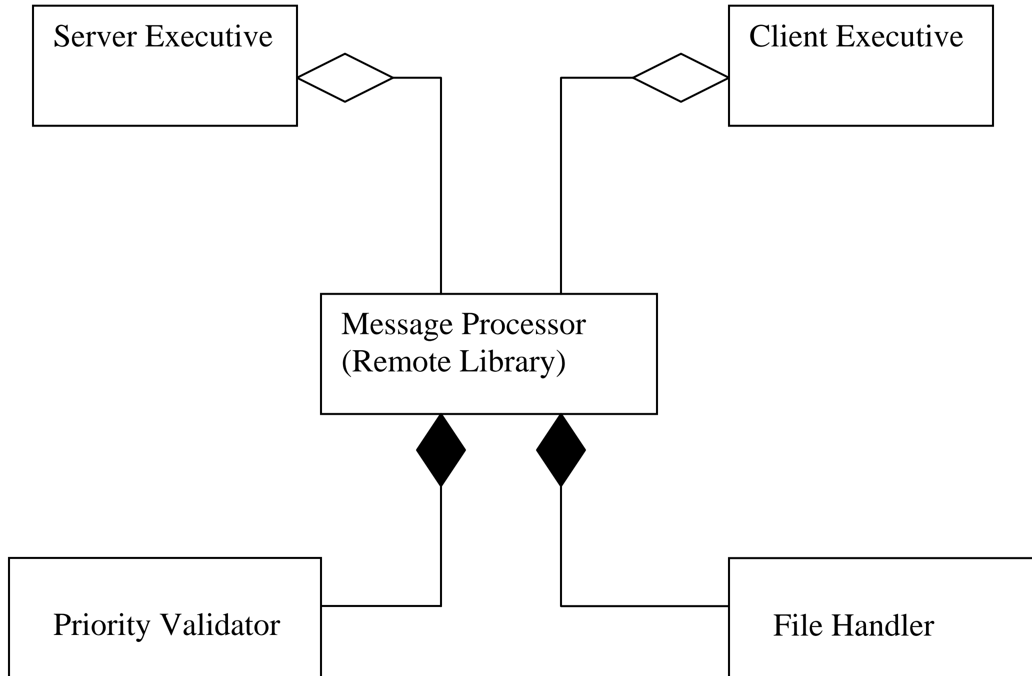

Figure 7.1: Class diagram for non-AOP File-Center application

The `MessageProcessor` class is the remotable type. It performs `DeleteFile`, `GetFile` and `PutFile` operations discussed above, employing the services of the File Handler class. In addition, it does priority validation using the Priority Validator class.

Let us see the code for `DeleteFile` operation in the Non-AOP File-Center application:

```
public void DeleteFile ( string IP )
{
      // Priority Validation - ORTHOGONAL TASK
      if ( this.mValidator.GetPriority ( IP ) < 2 )
            throw new Exception ( "Permission: DENIED.");

      // File Deletion - PRIMARY TASK
      System.IO.File.Delete ( MessageProcessor.cFileName );
}
```

Code Listing 7.5: DeleteFile method in the non-AOP File-Center application

As we can see, the method performs two functions – file deletion, which is its primary task, and priority validation, which is a secondary task. Clearly, the method has poor cohesion.

In the corresponding AOP code for File-Center application, we isolate the priority validation aspect into the server interceptor; as a result, the methods in `MessageProcessor` class are more cohesive and more purpose-specific. Let us see the code for `DeleteFile` operation in the AOP-based File-Center application:

```
public void DeleteFile ( )
{
      // File Deletion - PRIMARY TASK
      System.IO.File.Delete ( MessageProcessor.cFileName );
}
```

Code Listing 7.6: DeleteFile method in the AOP-based File-Center application

Comparing the code listings for the AOP-based and Non-AOP methods for `DeleteFile` operation, we can see the improvement in cohesion achieved by AOP. The AOP-based `DeleteFile` operation performs only its primary task – file deletion; the priority validation aspect is taken care of by the server interceptor. The class diagram for the AOP-based file center application is shown below:

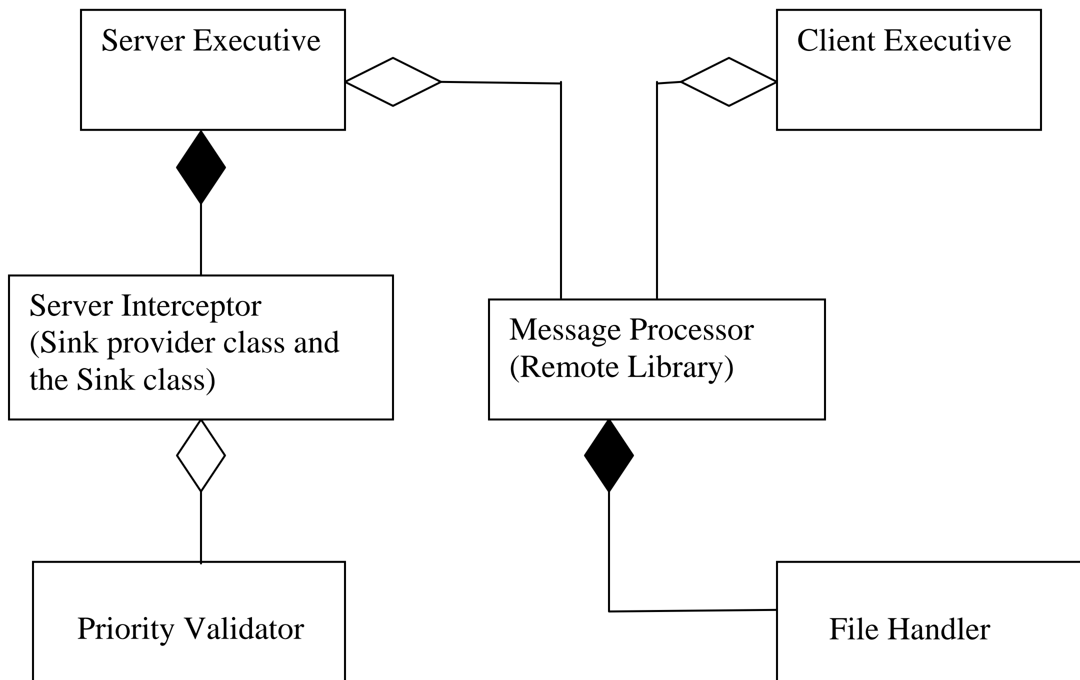

Figure 7.2: Class diagram for AOP-based File-Center application

The analysis results for the non-AOP and the AOP-based File-Center applications are listed below:

The Non-AOP code in the remotable library:

Lines of function code = 11
Cyclomatic complexity = 6
Cohesion = 5 tasks

The AOP client-code in the remotable library:

Lines of function code = 7
Cyclomatic complexity = 4
Cohesion = 3 tasks

The AOP server-code:

Lines of function code = 79
Cyclomatic complexity = 16

File-Center Interception library code

Lines of common AOP function code = 44
Cyclomatic complexity of common AOP code = 6

AOP code applicable to any Inter-AppDomain interception library

Lines of application specific AOP function code = 35
Cyclomatic complexity of application specific AOP code = 10

AOP code specific to the *File-Center* application

	Non-AOP	AOP
Total lines of function-code for the *File-Center* application	447	512

Table 7.4: Analysis results for the File-Center application

(d) Summary:

In the following table, we integrate the analysis results of the three applications described in sections 7.2.(a), 7.2.(b), and 7.2.(c). The integrated analysis results list the details on the lines of function code, cyclomatic complexity, and cohesion, adding up the values of the corresponding columns listed in Table 7.3, Table 7.4, Table 7.5.

- Number of sets of applications considered = 3 (Each set of applications has two sets of code – one based on AOP, and the other without using AOP)
- Number of Functions (Methods + Properties) considered = 9

	Without AOP	With AOP
Lines in the body of the function (for the maintainable code)	**68**	**19**
Cyclomatic complexity (for the maintainable code)	**29**	**13**
Cyclomatic complexity of maintenance-requiring library code	**55**	**-**
Cohesion	**18 tasks**	**10 tasks**

Table 7.5: Integrated analysis results

(e) Performance penalty of interception-based AOP:

As discussed in the previous chapters, interception-based AOP requires that the caller and the callee be in separate .NET contexts. In some cases, method synchronization for example, we explicitly create separate contexts for the caller and the callee, which would not have been necessary for conventional non-AOP execution. To enable interception, we force inter-context message passing which would insert overheads and affect performance. Again, the messages are intercepted at run-time; this, as well, would negatively affect performance. As a minimal test for calculating the over-head of interception, a simple experiment was conducted. The experiment and the results are described in Appendix 1.(b)

In section 7.1, we described that we had difficulty in developing non-AOP codes for Debugging and Code-Profiling. In this section, we describe why:

Let us quickly recollect how *debugging* was achieved in the debugger application described in chapter 4, with the help of a sample code:

```
[DebuggerContext (true)]
public class ClosedFigure : System.ContextBoundObject
{
  [DebuggerContext (true)]
  public double ClosedCalcArea(double Area)
  {
    Console.WriteLine ("Inside Closed class's ClacArea function.");
    if (Area < 0)
      throw new ArgumentOutOfRangeException ("Area", "Area cannot be negative.");
    Console.WriteLine ("Input parameter is: " + Area + " ;\tMy area is: " + Area);
    return Area;
  }
}
```

Code listing 7.7: A type that is enabled for interception by the debugger-interception library

The type ClosedFigure inherits from **System.ContextBoundObject** class, and employs the **DebuggerContext** attribute, a custom attribute that we have developed for the debugger application, with a true passed to its constructor – this qualifies the type for interception. When a client invokes a new instance of the type **ClosedFigure**, a separate context is created for the instance, and a custom interceptor sink is bound to it. All calls to and from the context is intercepted by the custom intercepted sink.

```
class Executive
{
  // Main.
  static void Main (string [] args)
  {
    ClosedFigure C = new ClosedFigure ();
    double CirArea = C.ClosedCalcArea (1.0);
    Console.WriteLine ("\nThe returned value is " + CirArea.ToString ());
  }
}
```

Code listing 7.8: Executive spawns a new instance of ClosedFigure type, and invokes its method

In the code listing above, the command

```
ClosedFigure C = new ClosedFigure ();
```

generates a new context-bounded instance of **ClosedFigure**, with a custom interceptor sink bound to it. Any further method call to the instance, like

```
C.ClosedCalcArea (1.0)
```

whose method definition is marked with the breakpoint, **DebuggerContext** with a true passed to its constructor, is intercepted and debugged.

Note that the entire application debugger is implemented as an aspect. We therefore conclude that non-AOP code cannot be developed for the "debugger" application

described in chapter 4; hence "debugging" aspect has not been considered for quantitative comparison.

Let us consider a code-coverage application involving the "code-profiling" aspect as described in chapter 5:

```
[Profile(true)]
public class MyClass : System.ContextBoundObject
{
      [Profile(true)]
      public void TestFunc1()
      { Console.WriteLine("\n\tMyClass.TestFunc1 called."); }

      [Profile(true)]
      public void TestFunc2()
      { Console.WriteLine("\n\tMyClass.TestFunc2 called.");  }

      [Profile(true)]
      public int TestFunc3(double d)
      { Console.WriteLine("\n\tMyClass.TestFunc3 called.");
        return Convert.ToInt32(d); }
}
class Executive
{
      [STAThread]
      static void Main(string[] args)
      {     MyClass M = new MyClass();
            M.TestFunc1();
            M.TestFunc3(4.5);

            MyClass N = new MyClass();
            N.TestFunc2();
            M.TestFunc1();

            //Interceptor library code-NOT listed in here
            Depot.DisplayResult();
            System.IO.StreamWriter S = new
            System.IO.StreamWriter("Output.txt", false);
            Depot.DisplayResult(S);
            S.Close();
      }
}
```

Code listing 7.9: A code-profiling application; *Note*: interception library code NOT listed here.

Code listing 7.9 provides the program code for a class that performs code-coverage to list the methods in a type that were tested, and those that weren't tested. A method is considered tested if it is called by a client at least once. Otherwise, it is considered untested. We believe that intercepting method calls would be the ideal technique to find out the methods in a specified type that are tested – finally a bit of metadata analysis on the type to find out all the methods in it would help in exposing the methods that remain untested as well. For such code-profiling applications also, we do not find an elegant way to develop a corresponding non-AOP application. We consequently do not include "Code-Profiling" aspect for quantitative analysis.

7.3. Conclusion:

From the analysis, we draw the following conclusions:

- + AOP reduces client code complexity:

From the analysis results, we see that the cyclomatic complexity of a total of 9 client functions (methods + properties) without using AOP is 29 [Table 7.6]. This is reduced to 13 by the AOP-based client code, which is a striking advantage. The AOP-server code, though, suffers from a high cyclomatic complexity. This is tolerable, since the isolation of the aspect is already done now, and any more addition of methods won't increase the cyclomatic complexity of the server code further.

- + AOP improves cohesion:

An advantage of AOP, as apparent from the analysis, is its potential in improving the cohesion of methods. Over a set of 9 functions ranging across three different sets of applications, a cohesion of 18 tasks by the non-AOP codes was brought down to 10 tasks by AOP-based codes [Table 7.6].

- + AOP captures the purpose of the code better:

AOP captures the functionality of the code; also, the code is more readable and maintainable. Consider the *Range-Checker* application described in 7.2.(a):

```
[Range (Lower = 0.0)] double radius
```

at the function declaration is more explicit and elegant than

```
if (radius < 0.0)
     throw new ArgumentOutOfRangeException ("...");
```

inside the body of the function.

- + AOP enhances abstraction and encapsulation:

AOP-based codes isolate and encapsulate aspects into separate modules. The client code, hence, is made more maintainable, and less tangled. For instance, in the *Range-Checker* application described in section 7.2.(a), the validation aspect is isolated into a separate module. Any change to validation check handling can be made at this **ONE** place.

- + AOP promotes reusability:

AOP-based code promotes reusability. For example, in the *Range-Checker* application described in section 7.2.(a), The range-checker attribute, named `Range`, provides a common point where the out-of-range exceptions can be handled. The range-checker attribute can be applied to a variety of classes, which need range-checks, like: class Geometry that deals with geometric calculations, or a fictitious class called Stock_Market that processes stock accounts.

- - Interception-based AOP comes with performance penalty:

Interception-based AOP requires that the caller and the callee be in separate .NET contexts. To enable interception, we force inter-context message passing which would insert overheads and so hamper performance. Again, the messages are intercepted at run-time; this, as well, would negatively affect performance[10].

[10] Please refer to Appendix 1.(b) for experimental measurement of performance penalty of AOP

8. CONCLUSION

In this thesis research, we have evaluated the impacts of AOP on code complexity and performance. We have used .NET for our work. The focus has not only been on identifying means to achieve AOP but also on analyzing different scenarios where AOP would be the right choice. The need for isolating aspects to develop a more purpose-specific and less complex client code has been emphasized.

The technique used in this research to achieve AOP is run-time interception of method calls. The second chapter of this thesis discusses the .Net messaging system, channel architecture, and execution contexts. In subsequent chapters (chapters 3, 4, 5, 6 and 7), we discuss some applications where AOP has been employed to isolate secondary aspects, and to develop more purpose-specific and less complex client code. The different techniques that were discussed in the previous chapters are summarized here:

- **Summary:**

1. **Method synchronization**:

The primary contribution of this chapter is the demonstration of a new way of achieving Method-Synchronization in a .NET environment. Our approach is based on Aspect-oriented programming.

We have developed a .NET library that enables client code developers to plug in the method synchronization aspect into their application. The library helps the client code developer to dispose of synchronization elements from the client code, and plug them in as attributes. This reduces the complexity of the code, and captures the purpose of the code better.

Method-synchronization, as discussed in this research, tries to isolate the synchronization aspect into a separate module. Method synchronization can be encapsulated as Aspect-Oriented code. All that the client code developer needs to do is to employ the Synchronization attribute, and turn it on where synchronization is required. Isolation of the synchronization aspect results in a less-complex, and more cohesive (purpose-specific) code.

2. **Debugging**:

In this chapter, we propose a new way to develop debugger applications. Our approach is based on aspect-oriented programming. Our contribution has been in identifying that

debugging is an aspect, and that employing AOP is an effective way to develop debugger applications. We have also developed a custom interception library which performs debugging services by intercepting method calls. The debugger library is a really useful tool for the client code developer. Break points can be turned on or switched off programmatically, by simply enabling or disabling the debugger-attribute.

The debugger application helps a client-code developer to functionally interact with the code under execution. The custom debugger application that we have developed using AOP supports the following features:

- Facility to programmatically set break points
- Facility to manipulate input parameter values from sink stack
- Facility to dump the stack trace in case of an exception
- Facility to manipulate the return value
- Facility to ignore a return-exception, and substitute it with a dummy return value
- Facility to restart the application, all over again, at any point during execution

The chapter discusses the theory behind the development of debugger applications using AOP, and the advantages of employing AOP for the same.

3. **Code-profiling**:

In this chapter, we propose that certain code-profiling and optimizing applications can be developed efficiently using AOP. Our contribution has been in identifying that AOP approach suits code-coverage applications very well. All that the client-code developer needs to enable the testing is to decorate the class / methods with custom attributes. Code

coverage is hidden very well behind by the aspect-oriented interception library. The time-profiling application illustrated in the chapter goes well with the aspect-oriented approach as well. Firstly, we discuss an Aspect-Oriented approach to develop code coverage applications. We then illustrate a time profiler application. The time-profiler functions as a code optimizer.

4. **Prioritization**:

This chapter discusses 'prioritization' service implemented efficiently using Aspect Oriented Programming. Our primary contribution in this chapter has been in identifying that prioritization can be classified as an aspect of a distributed computing system, and can be successfully isolated into a separate module using AOP. We have explored AOP-based client prioritization and message prioritization as two specialized cases of priority assignment. The architectural and design details of the interception implementation are discussed. The isolation of the prioritization aspect into the interceptor code is explained.

5. **Validation**:

In chapter 7 of this thesis work, we have made a quantitative analysis of AOP. We have chosen two sample application involving Priority Validation, and Range-Checking, and have implemented them with and without using AOP separately. We have then analyzed the code features like line count, code complexity and cohesion. Based on the analysis results, we have drawn the following conclusions:

- AOP helps reduce client code complexity
- AOP captures the purpose of the code better
- AOP enhances abstraction and encapsulation
- AOP promotes reusability
  - Run-time interception-based AOP comes with a performance penalty

- **Future Work**

This thesis provides several opportunities for future work on AOP. A list of prospective applications that can be explored using AOP is given below:

1. **Loading-unloading of DLLs**: It is a common scenario where the client code developer requires the services of a DLL for a small part of his/her program, but not for the entire lifetime of the application. One workaround for this in .NET is the to load an application domain, load the DLL into the AppDomain, and after use, unload the whole AppDomain. We believe that this process can be achieved efficiently using an AOP.

2. **Load-balancing**: Load-balancing is a critical issue in a multi-server environment. We believe that an interception-based AOP mechanism implemented at the gateway of the server cluster would be a good way to achieve load balancing.

3. **Clustering technology**: Clustering technology aims to provide continuous high availability of service to clients despite processor / communication failures at the backend machines. We believe that clustering technology can be potentially implemented effectively using AOP.

Techniques for AOP:

In this research, we have used run-time interception-based implementation of AOP. As part of a future project, we plan to explore AOP using customized proxies.

APPENDIX

1. Definition of key words:

a) Cohesion of a method / property: The number of tasks performed by the method / property. The lower the cohesion for a method / property, the better; ideal case for the cohesion of a method / property is one – in which case the method / property performs only its primary task, specified by the method's / property's name ideally.

b) Cyclomatic complexity of a method / property: The number of independent loops through the method. Cyclomatic complexity of a method / property can be calculated by the following formula:

1 + number of "if" loops in the Method / Property

+ number of "else" loops

+ number of "for / foreach" loops

+ number of "while" loops

+ number of "do" loops

+ number of "case" statements

+ number of "break" statements

+ number of "continue" statements

2. Execution time measurement of AOP and Non-AOP method calls:

As a minimal experiment to measure the performance penalty of employing AOP, a simple application was developed. The program design is as follows. Two types, called AOP_Class and NonAOP_Class are developed, each with one function called Test. The type AOP_Class is derived from System.ContextBoundObject, and applies a ProfileAttribute that adds a custom interceptor sink to the new context that is created for any instance of the AOP_Class type, i.e., inter-context method calls to instances of type AOP_Class are intercepted. The interceptor sink though does not have any custom processing in this example. The type NonAOP_Class is not enabled for interception.

The code listing for the two types, AOP_Class and NonAOP_Class are added below:

```
[Profile(true)]
class AOP_Class : System.ContextBoundObject
{
      public void Test ( )
      {
            return ;
      }
}

class NonAOP_Class
{
      public void Test ( )
      {
            return ;
      }
}
```

Code Listing Appendix.2.1: Code listing for types AOP_Class and NonAOP_Class

The methods by name Test, of both the types, are called from the executive to measure the time of execution of the methods. The experiment result is added below:

Time for execution of the method call **once** (averaged from 100000 trials) is:

Non-AOP	**1.06201** Microseconds
AOP	**10.36437** Microseconds

Table Appendix.2.1: Execution time measurement of AOP and Non-AOP method calls

This time is simply the time required to make a function call to an empty body with and without interception, needed for AOP. Usually that will be amortized over the time required to do processing within the function, so the impact of interception may often be negligible.

3. Null interval measurement:

To measure the inaccuracy caused by the overheads of inter-context method calls, we make dummy inter-context method calls which involve no actual processing – just the making of the method call and the return; we call this time period, the *null interval*. The following code listing shows the code at the interceptor sink for measuring the null interval:

```
// The interceptor sink class.
internal class ProfileSink : IMessageSink
{
  // Next message sink in the chain.
  private IMessageSink _nextSink;

  // High resolution timer - least count = 1 microsecond
  private HRTimer.HiResTimer H = new HRTimer.HiResTimer();

  // Other member methods not listed here

  // Processes Synchronous method calls.
  public IMessage SyncProcessMessage(IMessage msg)
  {
    ulong resultAOP = 0 ;
    const int trials = 100000 ;
    IMessage Ret = null ;
    for ( int count = 0 ; count < trials ; ++count )
    {
      H.Start();
      Ret = this._nextSink.SyncProcessMessage ( msg ) ;
      H.Stop();

      resultAOP += H.ElapsedMicroseconds ;
    }

    // Null interval (in microseconds)
    //       = (double)resultAOP / (double)trials.

    return Ret ;
  }
}
```

Code Listing Appendix.3.1: code at interceptor sink for null interval measurement

REFERENCES

[1] **Aspect-Oriented Programming**; *Gregor Kiczales, John Lamping, Anurag Mendhekar, ChrisMaeda, Cristina Videira Lopes, Jean-Marc Loingtier andJohn Irwin - Xerox Palo Alto Research Center*; European Conference on Object-Oriented Programming (ECOOP), 1997.

- The seminal paper on AOP; discusses the fundamental ideas behind AOP; illustration using a black and white image processing system.
- This paper, being a classic paper on AOP, served as a base for this thesis research work.

[2] **Aspect-Oriented Programming with C# and .NET**; *WolfGang Schul, Andreas Polz – Hasso-Plattner-Institute at University Potsdam, Germany*; Fifth IEEE international symposium on object-oriented real-time distributed computing (ISORC 2002) : Proceedings : 29 April – 1 May, 2002, Washington, D.C.

- Discusses the usage of AOP techniques in context of the .NET framework; focuses on the fault-tolerance aspect and discusses the expression of non-functional component properties (aspects) as C# custom attributes.
- This paper discusses .NET support for AOP, and was instrumental in guiding our research on AOP under .NET.

[3] **Aspect-Oriented Programming of Sparse Matrix Code**; *John Irwin, Jean-Marc Loingtier, John R. Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar, Tatiana Shpeisman - Xerox Palo Alto Research Center*; Proceedings International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE), Marina del Rey, CA, December 1997

- Discusses an AOP-based implementation of a system for sparse matrix computation that deals with aspects like execution time, data representation and numerical stability
- This paper presents an illustration of AOP, though the example discussed is not directly related to this thesis work.

[4] **Improving Dynamic Data Analysis with Aspect-Oriented Programming**; *Thomas Gschwind, Johann Oberleitner*; Seventh European Conference on Software Maintenance and Reengineering; Proceedings; 26-28 March, 2003, Benevento, Italy

- Presents a new instrumentation approach based on AOP to reverse engineer a given software application, which provides support for dynamic feature analysis
- This illustrations discussed in this paper are not directly related to this thesis work

[5] **Applying Aspect-Oriented Programming Concepts to a component-based Programming Model**; *Thomas Eidson, Jack Dongarra, Victor Eijkhout;* International Parallel and Distributed Processing Symposium : Proceedings; April 22-26, 2003, Nice, France

- Discusses how aspect-oriented concepts can be applied to support the reduction of intertwined code related to different programming concerns; talks about two specific cases – mixing I/O with a numerical computation, and the use of behavioral metadata
- The illustrations discussed in this paper are not directly related to this thesis work

[6] **Implementing collaboration-based Designs using Aspect-Oriented Programming**; *Elke Pulvermuller, Andreas Speck, Awais Rashid*; 34th International Conference on Technology of Object-Oriented Languages and Systems; TOOLS 34; July 30 – August 4, 2000, Santa Barbara, California

- Presents a new approach based on AOP to realize a collaboration-based design
- The illustration discussed in this paper is not directly related to this thesis work

[7] **Aspect-Oriented Technology for Business Applications: A Case Study in Stock Trading**; *Faisal Akkawi, Atef Bader, Tzilla Elrad*; 12th International Workshop on Database and Expert Systems Applications; Proceedings ; 3 – 7 September, 2001, Munich, Germany

- Discusses an AOP-based approach to achieve adaptability with regard to the life cycle of software systems; presents a stock-trading example
- The stock-trading example discussed in this paper is not directly related to this thesis work

[8] **Aspect-Oriented Programming in C#/.NET**; *Edward Garson, Dunstan Thomas Consulting, http://consulting.dthomas.co.uk*;

- Provides a broad overview of Aspect-Oriented Programming under the .NET platform
- Provides illustration of a 'event logging' application using AOP

[9] **Contexts in .NET - Decouple Components by Injecting Custom Services into Your Object's Interception Chain**; *Juval Lowy*; MSDN Magazine; http://msdn.microsoft.com/msdnmag/issues/03/03/ContextsinNET/

- Explains contexts in .NET, custom context attributes, interception architecture
- Served as guide in learning about the context architecture, channel model, and interception technique under the .NET platform

[10] **Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse;** *Dharma Shukla, Simon Fell, and Chris Sells*; http://msdn.microsoft.com/msdnmag/issues/02/03/AOP/default.aspx

- Explains context architecture, interception, and how AOP can be achieved under a .NET environment
- Served as a guide in learning about AOP using interception under .NET, and some advantages of the aspect-oriented approach.

[11] **AspectC++: an aspect-oriented extension to the C++ programming language;** *Olaf Spinczyk, Andreas Gal, Wolfgang Schröder-Preikschat*; Proceedings of the Fortieth International Confernece on Tools Pacific: Objects for internet, mobile and embedded applications - Volume 10, February 2002

- Presents AspectC++ (Gal & Spinczyk 2001), a new language extension to C/C++ that provides language elements to facilitate AOP
- This topics discussed in this paper are not directly related to this thesis work

[12] **The Convergence of AOP and Active Databases: Towards Reactive Middleware**; *Mariano Cilia, Michael Haupt, Mira Mezini, Alejandro Buchmann, Department of Computer Science, Darmstadt University of Technology - Darmstadt, Germany*; Proceedings of the second international conference on Generative programming and component engineering, September 2003;

- Analyzes the convergence of techniques from aspect oriented programming, active databases and asynchronous notification systems to form reactive middleware
- The illustration discussed in this paper is not directly related to this thesis work

[13] **On adaptable middleware product lines**; *Wasif Gilani, Nabeel Hasan Naqvi, Olaf Spinczyk*; Proceedings of the 3rd workshop on Adaptive and reflective middleware, October 2004

- Proposes a family-based approach based on aspect-oriented programming (AOP) for the implementation of middleware product lines which are highly configurable and adaptable
- The illustration discussed in this paper is not directly related to this thesis work

[14] **An approach for supporting aspect-oriented domain modeling**; *Jeff Gray, Ted Bapty, Sandeep Neema, Douglas C. Schmidt, Aniruddha Gokhale, Balachandran Natarajan*; Proceedings of the second international conference on Generative programming and component engineering, September 2003

- Describes a technique for improving separation of concerns at the level of domain modeling
- The illustration discussed in this paper is not directly related to this thesis work

[15] **DAOP-ADL: an architecture description language for dynamic component and aspect-based development**; *Mónica Pinto, Lidia Fuentes, Jose María Troya*; Proceedings of the second international conference on Generative programming and component engineering, September 2003

- Describes DAOP-ADL, a component- and aspect-based language to specify the architecture of an application in terms of components, aspects and a set of plug-compatibility rules between them
- The illustration discussed in this paper is not directly related to this thesis work

[16] **MADAPT: managed aspects for dynamic adaptation based on profiling techniques**; *Robin Liu, Celina Gibbs, Yvonne Coady*; Proceedings of the 3rd workshop on Adaptive and reflective middleware, October 2004

- Makes the case for using aspect-oriented programming (AOP) as a means to achieve adaptive middleware based on fine-grained, customizable, profiling techniques
- The illustration discussed in this paper is not directly related to this thesis work

[17] **A Study on Exception Detection and Handling Using Aspect-Oriented Programming**; *Martin Lippert Cristina Videira Lopes*; Proceedings of the 22nd International Conference on Software Engineering, Limmerick, Ireland. ACM Press, 2000

- Reports on a study to investigate AOP's ability to ease tangling related to exception detection and handling
- Served as a guide in developing the 'exception handling' feature of the 'custom debugger' application discussed in chapter 4 of this thesis work

[18] **An Initial Assessment of Aspect-oriented Programming**; *Robert J.Walker, Elisa L.A. Baniassad and Gail C. Murphy, Dept. of Computer Science, University of British Columbia, Vancouver, Canada*; Proceedings of the 21st International Conference on Software Engineering , 16–22 May 1999, Los Angeles, CA, USA

- Explores the efficacy of AOP in modularizing design decisions that cross-cut a functionally-decomposed program

- Served as an initial guide for learning about the proposed merits of AOP

[19] **Aspect-Oriented Programming is Quantification and Obliviousness**; *Robert E. Filman Daniel P. Friedman*; Workshop on Advanced Separation of Concerns, OOPSLA, 2000

- Analyzes AOP systems with respect to three critical dimensions: the kinds of quantifications allowed, the nature of the actions that can be asserted, and the mechanism for combining base-level actions with asserted actions
- The illustrations discussed in this paper are not directly related to this thesis work

[20] **RG: A Case-Study for Aspect-Oriented Programming**; *Anurag Mendhekar, Gregor Kiczales, John Lamping*; Technical report SPL97-009 P9710044 Xerox Palo Alto Research Center. February 1997

- A case study for AOP using RG, an image processing system that allows sophisticated image processing operations to be defined by composing primitive image processing filters
- The illustration discussed in this paper is not directly related to this thesis work

[21] **Role Model Designs and Implementations with Aspect-oriented Programming**; *Elizabeth A. Kendall, Departments of Computer Science and Computer Systems Engineering, Royal Melbourne Institute of Technology, Melbourne, Australia*; 14th ACM SIGPLAN conference on Object-oriented programming, 1999

- Describes research in applications of aspect-oriented programming (AOP) as captured in the AspectJ™ language
- The illustration discussed in this paper is not directly related to this thesis work

[22] **Declarative aspect-oriented programming**; *Ralf Lammel, Department of Computer Science, University of Rostock, Rostock, Germany*; Proceedings PEPM, 1999

- Investigates the suitability of functional meta-programs to specify aspects and to perform weaving
- The illustration discussed in this paper is not directly related to this thesis work

[23] http://eclipse.org/aspectj/; AspectJ™ (aspect-oriented extension to Java) homepage

[24] **Recent Developments in AspectJ™**; *Cristina Videira Lopes and Gregor Kiczales Xerox Palo Alto Research Center, Palo Alto, CA*; ECOOP'98 Workshop Reader

- Summarizes the latest developments in AspectJ™

[25] **Essential .NET, Volume I: The Common Language Runtime**; *Don Box, Chris Sells*; Addison-Wesley Professional, 1st edition (November 4, 2002), ISBN: 0201734117

- Provides a good overview of the scope of execution in .NET, and how .NET provides support for context-interception, and customization of proxies
- Served as a guide in learning about interception mechanism in .NET.

[26] **Programming .NET Components**; *Juval Lowy*; O'Reilly 1st edition (April, 2003), ISBN: 0596003471

- Discusses AOP; context-interception
- Has illustrations on interception-based AOP in .NET

[27] **Advanced .NET Remoting (C# Edition)**; *Ingo Rammer*; Apress, 1 edition (April 5, 2002), ISBN: 1590590252

- Explains context-interception, remoting interception, channel architecture, and techniques for implementing custom channel sinks
- Has illustrations on inter-context interception

[28] **Remoting with C# and .NET: Remote Objects for Distributed Applications**; *David Conger*; Wiley, 1 edition (January 3, 2003), ISBN: 047127352X

- Explains channel architecture in .NET; techniques to plug custom channel sinks at server-side and client-side sink chains in a remoting system.
- Served as a guide for remoting interception, and architecture and implementation of channel sinks in .NET

[29] **Microsoft .NET Remoting**; *Kim Williams, James Naftel, Scott McLean*; Microsoft Press, 1st edition (September 25, 2002), ISBN: 0735617783

- Discusses .NET Remoting architecture in depth and provides examples in C# that demonstrate how to extend and customize .NET Remoting

- Served as a guide in learning about real and transparent proxies, channel model, and message passing in .NET

[30] **Applied .Net Attributes**; *Tom Barnaby, Jason Bock*; Apress, 1st edition (October 1, 2003), ISBN: 1590591364

- Discusses AOP; interception, usage of custom attributes
- Served as a guide in learning about inter-context interception, and AOP.

[31] MSDN documentation

http://msdn.microsoft.com/library/default.asp

- Served as a guide in learning about the context-architecture in .NET, and the in-built library types of .NET that support message passing and interception, as discussed in chapter 2 of this thesis work