Aspect Oriented Software Design Approach and Implementations

Gergely Z. Tóth[*] and Éva Hajnal[**]
[*] evosoft Hungary Ltd, Budapest, Hungary
[**] University of Óbuda/AREK, Székesfehérvár, Hungary
gergely.toth@evosoft.com
eva.hajnal@arek.uni-obuda.hu

***Abstract*—Nowadays we can face a new software crisis in Object Oriented Programming (OOP). Advantages of Aspect Oriented Programming (AOP) against the OOP method is really glaring in the development of complex software systems. The handling of crosscutting is hard in OOP, especially if a component or an object uses the services of another component or object for the execution of its own work. Software design patterns are mitigating this problem, for suggesting templates to the handling of the most frequent interactions, but do not give general solutions. With the enlargement of the complexity of the software, the numbers of implicit and explicit dependencies are increasing. The consequence of it, the maintenance of the system and the integration of new functionality became more difficult. The new paradigm of AOP shows a competitive methodology for solving these problems.**

I. Introduction

How Ada Augusta Byron could be surprised if she could nowadays study the science of computer programming created by her. Almost each area of our life is affected by some kind of computing device. And these devices are controlled exclusively by the descendants of her first instructions sequences represented by punch cards: complex software systems.

In the beginning there were “magicians” creating computer programs using their intuition to figure out the proper order of machine instructions in order to solve mathematical problems. The early 50’s have raised the abstraction of the functionality – as the code gurus created the FORTRAN and other “high level” languages for themselves. To the end of 70’s these “Wizards of OS” were absolutely extinct: they were replaced by craftsmen of structured programming manufacturing the code instead of perceiving it. Their knowhow was founded by Dijkstra and Wirth, and their hammer was given them by Ritchie. With their common methodology and tools they were able to construct more complex software systems that even earlier did.

The wheels were turning that a new abstraction was involved: the paradigm of Object Oriented design. The spotlight was moved from the algorithms to the architecture, and the new hero – ser Object – can win over even more headed dragons of problems to solve.

However it was only a single victory in the heroic war against the complexity. Each battle won covered more complex tasks to be solved, and the long-standing weapons had to be replaced with more advanced ones. Ser Object has taken the Shield of Design Patterns, he put on the Armor of Java and .Net Environments, and finally he mounted the Stallone of Object Relation Mapping.

As the complexity was grown we had to be faced to a new problem – called “crosscutting”. Despite of its excellent tools and methodology, there were more and more problems the Object Oriented approach had difficulties to manage with. Uncovering the structure of the problem and recognizing the artifacts supporting the final solution is the good start to construct the system. However, fulfilling the requirements step by step may lead to unclear, unmanageable code with unpredictable internal impacts between the components, because the requirements simply refuse to be independent.

This is one of the actual problems of the software manufacturing – probably the biggest one. Nowadays huge effort is done to find new tools, methodologies or even a new approach to manage these interdependencies. Fortunately we have a promising direction to reach these goals: the Aspect Oriented Software Design.

II. Aspect Oriented Approach

A. *Why Aspect Oriented Design*

To understand why the Aspect Oriented Design was raised we should start studying its ancestor, the Object Oriented Approach. It is widely popular because of the ease of its use. OO programming came to solve two serious problems: designer modularity and data abstraction. OO programming presented tools and methodology to better decomposition of the problem – like classes, objects, encapsulation, inheritance, polymorphism, etc.

However, decomposing the problem into classes may easily lead to forgetting general concerns impacting each of the atomic assemblies or at least most of them. For example, while the specific services got separated into distinct classes, general tasks – like logging or authorizing – could be repeatedly implemented in each class addressing them. Actually, multiple inheritances can solve these problems, but because the huge amount of ambiguity they can involve, they are banned from the most object oriented languages.

Object oriented programming also has difficulty dealing with global information. Functionality that requires the involvement of several different objects results in interdependency between those objects/components. This makes the application susceptible to the implementation changes of a dependent object/component.

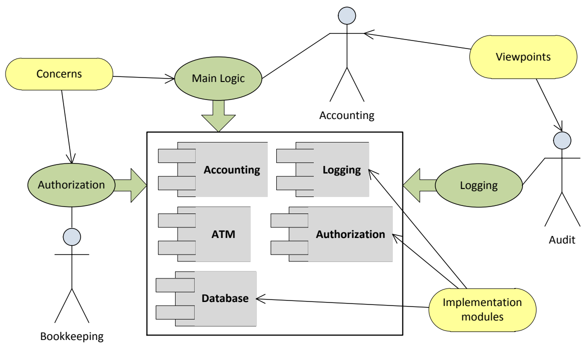

Figure 1: A typical application

Maintenance and enhancement are also problems, as the interactions between these objects/components are typically hard coded within the containing object.

Messy object architectures occur when something that the application needs to do requires the involvement of many different objects. Figure 1 introduces such an application. Logging and Persistence are not business logic requirements, but those are internal or system level requirements. So if you think in a concern point of view system has 3 concerns:

1. Business logic concern (core concern)
2. Logging concern
3. Authorization concern.

For example if Logging is to be added to Accounting, ATM and Database components, this involves changes in all three components. Since there are Client-Server interactions between these modules and the Logging module, the traditional approach dictates the server to be invoked by the clients as shown in Figure 2.

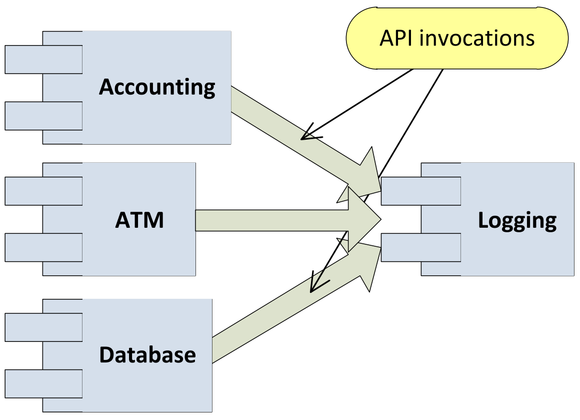

Figure 2: Traditional implementation

B. *Crosscutting concerns, aspects and tangled code*

According to Kiczales [6], our problem is lies in the existence of *crosscutting concerns*. That is, concerns which cannot be constrained easily into modular form. These concerns destroy the modularity that we strive for in our OO programs. They introduce related or even duplicated code into one or more modules.

Examples of crosscutting concerns are not hard to find in large systems. A group of crosscutting concerns that seem common to many programs already exists. In the example application of Figure 1 Logging and Persistence are such kind of concerns. Some other examples of these are performance, synchronization, communication, graphics manipulation and debugging.

According to the IEEE definition [1][6] a *concern* should be "... those interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders". A simple concern, non-crosscutting concern is also known as *core concern* or *requirement*. In the example of Figure 1 Main Logic is a core concern. A core concern can be easily encapsulated into a generalized procedure – i.e. a requirement can be implemented by a classic OOP component.

Crosscutting concerns affect several subsystems. On Figure 1 Logging and Authorization are crosscutting concerns as they are affects several components. Crosscutting concerns can be implemented by *aspects*. An aspect then is a representation of a crosscutting concern. A component is a modular unit of functional decomposition, which addresses a specific concern or function of the system. An aspect is similar, in that it addresses a concern of the system, but it cannot be cleanly decomposed into a component.

As Figure 2 shows, in a typical OOP approach, cross-cutting behaviors are factored into separate classes, which are then instantiated and called directly by business logic code. This forces business logic code to be intimately aware of these cross-cutting concerns and creates numerous problems for the architecture and code quality of an OOP application.

1. The number of lines of code increase. Cross-cutting concerns tend to accumulate: a business logic method that ought to be focused on calculating shipping for an online order may end up saddled with boilerplate code dedicated to creating an audit trail, authenticating the user, notifying the UI that a change has occurred, and executing its task on a worker thread.
2. An enormous amount of code is duplicated, as these boilerplate calls are often copied and pasted into new methods, which can lead to duplication of bugs and increases the difficulty of refactoring the cross-cutting concerns. In other words, OOP applications end up committing one of the very evils that OOP was designed to prevent: code duplication.

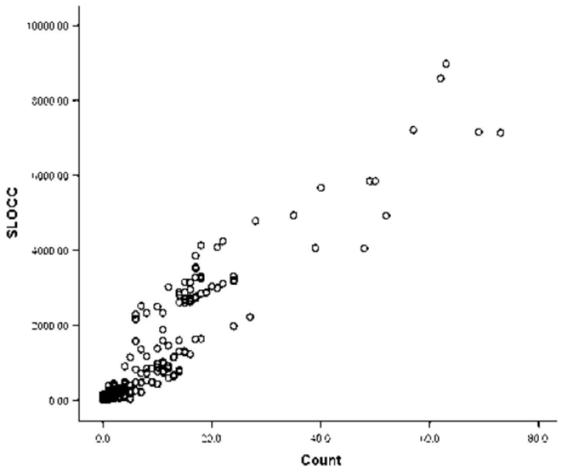

Figure 3: The number of defects in a feature is proportional to the size of the component. [2]

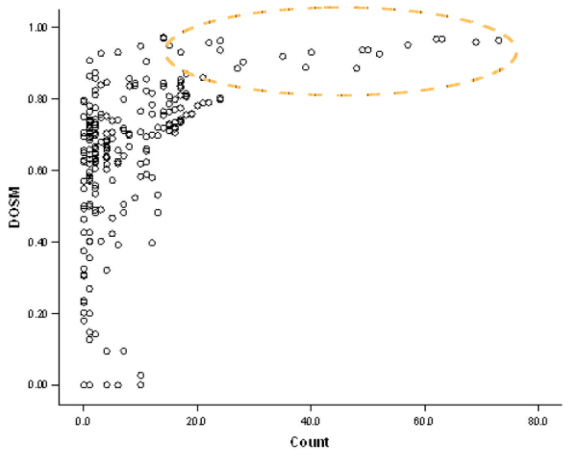

Figure 4: As a feature becomes more scattered through the code base, the odds that it will suffer from more defects increase dramatically. [2]

3. Business logic code becomes entangled with cross-cutting code, making it harder to find the actual parts of the routine dedicated to implementing business rules. This makes applications harder to understand, debug and modify.
4. Business logic code is tightly coupled to crosscutting concerns. If a change to transaction handling requires a change to the public interfaces of the transaction handling component, then potentially all of the code that uses that component must change as well.

The end result of this approach to crosscutting concerns is a code base that is hard to author, hard to maintain, and hard to debug. A study published in IEEE Transactions on Software Engineering [2] demonstrated that the number of defects in a feature is correlated to two factors:

1. the number of lines of code implementing that feature; and (see Figure 3)
2. the scattering of that feature throughout numerous source code artifacts. (see Figure 4)

The study, based on the analysis of three widely used open-source projects, also showed that cross-cutting concerns – whose implementation is always very scattered – have more defects than classical business features. Using a rank-order correlation coefficient, the authors found a strong correlation between a feature's bug count and that feature's diffusion across multiple classes and methods. "Crosscutting concerns" in traditional software, noted the researchers "are hard to find, understand, and work with."

C. ...And how AOP deals with them

Now that the nature of the problem is understood, the next logical step is to provide a solution. The Aspect Oriented Paradigm simply reverses the invocation routes as seen in Figure 5.

As you can see in Figure 5, the main difference with the traditional approach, that *AOP moves the responsibility of the invocation to the server side*. Rather than the clients should know about the aspects concerning them, the aspect itself have to “connect” the clients it support.

The structure is very similar to the *Observer* design pattern described in [3]. However, to avoid unnecessary impact to the client modules – they should not be “know”

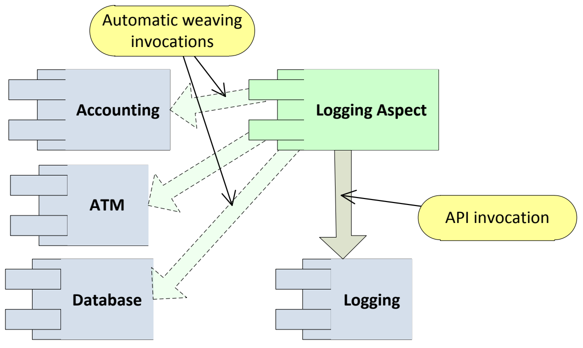

Figure 5: Implementation by AOP

about a new aspect was implemented – they should not send notifications to the server/observer. So the aspect itself should “inject” the invocations of itself into the clients’ code.

Kiczales’ solution [6] is to provide support in the language (or provide another language) for defining aspects along with the already present support for defining components. This new approach to programming is known as Aspect Oriented Programming (AOP) and is still in its infancy. AOP puts a greater focus on crosscutting concerns than is present in OO or many other language paradigms. It allows aspects to be cleanly separated and placed into modules that can be composed with other components (including other aspects) in the system.

D. AOP Development Stages

In many ways, developing a system using AOP is similar to developing a system using other methodologies: identify the concerns, implement them, and form the final system by combining them. The AOP research community typically defines these three steps in the following way:

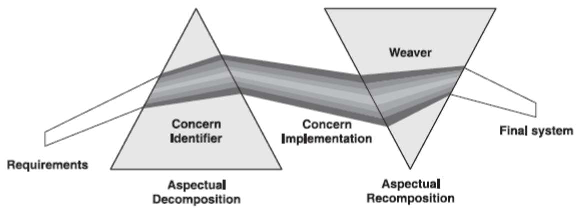

Figure 6: AOP development stages. [8]

1. *Aspectual Decomposition* – In this step, you decompose the requirements to identify crosscutting and core concerns. This step separates core-level concerns from crosscutting, system-level concerns. For example, in the Typical Application example (see Figure 1), suppose the following concerns: core business logic, logging, cache management, thread safety, contract enforcement, persistence, and authorization. Of these, only the core business logic is the core concern of Typical Application. All other concerns are system wide concerns that will be needed by many other modules and therefore are classified as crosscutting concerns.
2. *Concern Implementation* – In this step, you implement each concern *independently*. Using the previous example, developers would implement the business logic unit, logging unit, authorization unit, and so

forth. For the core concern of a module, you can utilize procedural or OOP techniques as usual.

For example, let's look at authorization. If you are using OOP techniques, you may write an interface for the authorization, a few concrete implementations for it, and perhaps a class to abstract the creation of the authorization implementation used in the system. Understand that the term "core" is a relative term. For the authorization module itself, the core concern would be mapping users to credentials and determining if those credentials are sufficient to access an authorized service. However, for the business logic module, the authorization concern would be a peripheral concern and so would not be implemented in the module at this time.

3. *Aspectual Recomposition* – In this step, you specify the recomposition rules by creating modularization units, or *aspects*. The actual process of recomposition, also known as weaving or integrating, uses this information to compose the final system. For our example, you would prescribe that each operation must first ensure that the client has been authorized before it proceeds with the business logic.

III. AspectJ: An AOP Language

AspectJ is a general-purpose, aspect-oriented extension to the Java programming language. Given that AspectJ is an extension to Java, every valid Java program is also a valid AspectJ program.

AspectJ consists of two parts: the language specification and the language implementation. The language specification part defines the language in which you write the code; with AspectJ, you implement the core concerns using the Java programming language, and you use the extensions provided by AspectJ to implement the weaving of crosscutting concerns.

The language implementation part provides tools for compiling, debugging, and integrating with popular integrated development environments (IDEs).

In AspectJ, the implementation of the weaving rules by the compiler is called crosscutting; the weaving rules cut across multiple modules in a systematic way in order to modularize the crosscutting concerns. AspectJ defines two types of crosscutting: static crosscutting and dynamic crosscutting.

Dynamic crosscutting is the weaving of new behavior into the execution of a program. Most of the crosscutting that happens in AspectJ is dynamic. Dynamic crosscutting augments or even replaces the core program execution flow in a way that cuts across modules, thus modifying the system behavior.

For example, if you want to specify that a certain action be executed before the execution of certain methods or exception handlers in a set of classes, you can just specify the weaving points and the action to take upon reaching those points in a separate module.

Static crosscutting is the weaving of modifications into the static structure – the classes, interfaces, and aspects – of the system. By itself, it does not modify the execution behavior of the system. The most common function of static crosscutting is to support the implementation of dynamic crosscutting. For instance, you may want to add new data and methods to classes and interfaces in order to define class-specific states and behaviors that can be used in dynamic crosscutting actions. Another use of static crosscutting is to declare compile-time warnings and errors across multiple modules.

A *Join Point* is an identifiable point in the execution of a program. Join cuts are defined by languages so different languages support different join cuts. AspectJ supports 11 different join points. It could be a call to a method or an assignment to a member of an object. In AspectJ, everything revolves around join points, since they are the places where the crosscutting actions are woven in.

A *pointcut* is a program construct that selects join points and collects context at those points. For example, a pointcut can select a join point that is a call to a method, and it could also capture the method's context, such as the target object on which the method was called and the method's arguments.

An *advice* is the code to be executed at a join point that has been selected by a pointcut. Advice can execute before, after, or around the join point. Around advice can modify the execution of the code that is at the join point, it can replace it, or it can even bypass it. Using an advice, we can log a message before executing the code at certain join points that are spread across several modules. Pointcuts and advice together form the dynamic crosscutting rules. While the pointcuts identify the required join points, the advice completes the picture by providing the actions that will occur at the join points.

The introduction is a static crosscutting instruction that introduces changes to the classes, interfaces, and aspects of the system. It makes static changes to the modules that do not directly affect their behavior. For example, you can add a method or field to a class.

The compile-time declaration is a static crosscutting instruction that allows you to add compile-time warnings and errors upon detecting certain usage patterns.

The *aspect* is the central unit of AspectJ, in the same way that a class is the central unit in Java. It contains the code that expresses the weaving rules for both dynamic and static crosscutting. Pointcuts, advice, introductions, and declarations are combined in an aspect. In addition to the AspectJ elements, aspects can contain data, methods, and nested class members, just like a normal Java class.

A. *Traditional Application example in AspectJ*

```
public class Logging {
  public void Trace(){
      System.out.println("Logging.Trace" );
  }
}

public class ATM {
  public void foo( int number, String name ){
      System.out.println( "ATM.foo" );
  }
}

public aspect Log{
  pointcut callPointcut() :
      call( void ATM.foo( int, String ) );
  before() : callPointcut() {
      System.out.println( "Log aspect" );
      Logging logger = new Logging();
      logger.Trace();
  }
}
```

The example code above illustrates the implementation model of Figure 5. The Logging and ATM modules are traditional Java classes not knowing about each other. There is a Log aspect declared in the 13th—20th lines to link these classes together. In the 14th and 15th lines it declares a pointcut with the name of *callPointcut*. This pointcut will be in effect whenever a call will be made to the foo method of the ATM class. (The full signature should be presented to distinguish the overrides of the same function.) Then the 16th..20th lines specify an advice to be executed just before a join point matching to the pointcut *callPointcut* is reached.

IV. PostSharp: An AOP Framework Implementation

As an approach to software development, aspect-oriented programming can be implemented through different technologies: source code weaving, bytecode/MSIL weaving, or dynamic proxies. Many tools are available on the market for most programming languages. Although all tools support basic features, they largely differ in their ability to scale to complex projects and large teams.

PostSharp relies on post-compilation, also called static weaving or MSIL/bytecode rewriting, which enhances the output of the compiler by producing a new executable program that includes additional behaviors added by aspects.

Instead of introducing a new programming language, PostSharp rather extends the existing ones with the capability of specifying aspects to resolve the problem of crosscutting. It uses static and dynamic pointcuts as AspectJ does, but PostSharp let the programmers to use their comfort programming languages in .Net environment.

A. *"Hello World" example with PostSharp*

1) *Create core component*

The "core component" is as simple as possible: it writes out the classic "Hello World" message, end exits. To see the effects of AOP logging, the component leaves traces also when it was created and disposed.

```
using System;

class Program : IDisposable
{
    static void Main(string[] args)
    {
        new Program(args).Run();
        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }

    public Program(string[] args)
    {
        Console.WriteLine("Program created");
    }

    public void Run()
    {
        Console.WriteLine("Hello World!");
    }

    public void Dispose()
    {
       Console.WriteLine("Program disposed");
    }
}
```

2) *Create a traditional attribute class*

To extend the program with logging capabilities, you should first add a reference to the PostSharp.dll component.

Your aspect will take the form of a .NET attribute that is applied to your code using declarative syntax. Create a new class ending in the suffix Attribute, such as TraceAttribute:

```
using System;

public sealed class TraceAttribute : Attribute
{
    private readonly string _category;

    public TraceAttribute( string category )
    {
        this._category = category;
    }

    public string Category
    {
        get { return _category; }
    }
}
```

3) *Derive from OnMethodBoundaryAspect*

To make this class an aspect, it must derive from an aspect parent class defined by PostSharp. Import the namespace *PostSharp.Aspects*, and declare that your aspect will derive from *OnMethodBoundaryAspect*.

A last thing: the aspect must be made *serializable*.

```
using PostSharp.Aspects;

[Serializable]
public sealed class TraceAttribute : OnMethodBoundaryAspect
```

4) *Implement Advices*

Advices are the notifications provided when you enter and exit a method. Add handlers for the *OnEntry* and *OnExit* events to capture these notifications. These events are where your aspect will provide its services.

```
public override void OnEntry(MethodExecutionArgs args)
{
    Console.WriteLine("{0}: Entering {1}.{2}.", Category,
            args.Method.DeclaringType.Name,
            args.Method.Name);
}

public override void OnExit(MethodExecutionArgs args)
{
    Console.WriteLine("{0}: Leaving {1}.{2}.", Category,
            args.Method.DeclaringType.Name,
            args.Method.Name);
}
```

5) *Apply Aspects to Methods*

A .NET attribute declaration is all it takes to wire your new aspect into one or more existing class methods. Add your aspect to an entire class to add your new behavior to all the class methods, or apply it only to individual methods.

```
[Trace("Trace Message")]
class Program : IDisposable
```

6) *Multicasting attributes: a "real pointcut"*

Using attributes to mark the join point in the source is a very elegant feature; however it involves some changes in the core component when a new aspect is inserted into the system. There is another way to specify the scope of an aspect without touching other parts of the code: applying the aspect attribute to the assembly itself. You only should insert a following code before any class code in the aspect module:

```
[assembly: Trace("Trace Message",
AttributeTargetTypes="Prog*")]
```

V. Conclusion

Though AOP/AOSD approach has a more than twelve year old past, it is hardly can be seemed to be widely spread. It cannot be obvious concerning the height of the problem it is targeted to solve.

Crosscutting concerns could be the most frequented source of the lack of maintainability of the code and leaving hidden defeats in it.

May be the thin language support was the reason why AOP was not accepted widely in software manufacturing. But today, AOP is supported in most languages and platforms. AspectJ, the reference AOP implementation, is the leading tool for Java. For Microsoft .NET, PostSharp is by far the most advanced and mature framework.

The best way for teams to learn AOP is to begin with simple aspects and add them to non-critical applications. For instance, developers can use diagnostic aspects during development and remove them from the production release, all without any impact on code. Then, as developers become more confident in AOP, they can begin to add more complex aspects in more critical applications.

By embracing aspect-oriented programming today, engineers are moving toward software of higher quality and lower complexity.

References

[1] M. Clifton “Aspect Oriented Programming / Aspect Oriented Software Design”, *Code Project Publication*, April 2003
Available online at http://www.codeproject.com/ Articles/ 4039/ Aspect-Oriented-Programming-Aspect-Oriented-Softwa#2

[2] Eaddy M., Zimmerman T., Sherwood K., Garg V., Murphy G., Nagappan N., Aho A. “Do Crosscutting Concerns Cause Defects?” *IEEE Transactions on Software Engineering, Vol. 34, No. 4*, July/August, 2008.
Available online at http:// citeseerx.ist.psu.edu/ viewdoc/ download? doi=10.1.1.164.7117 &rep=rep1 &type=pdf

[3] G. Fraiteur “Producing High-Quality Software with Aspect-Oriented Programming (Technical Whitepaper)”, *SharpCrafters s.r.o.*, July 2011

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides “Design Patterns: Elements of Reusable Object-Oriented Software” *Addison-Wesley Professional*, November 1994.

[5] T.J. Highley, M. Lack and P. Myers “Aspect Oriented Programming – A critical analysis of a new programming paradigm”, *University of Virginia Department of Computer Science, Programming Languages – CS655 Semester Project*

[6] IEEE “1471Conceptual Framework”,
Available online at http://www.iso-architecture.org/ 42010/ cm/ cm-1471-2000.html

[7] G. Kiczales, J. Lamping, A. Mendhekar, et al., “Aspect-Oriented Programming”, *Xerox PARC, Palo Alto, CA.* June, 1997.

[8] R. Laddad “AspectJ in Action”, *Manning Publications Co.*, 2003

[9] K. Ch. Ravipati “Introduction to Aspect Oriented Programming”, *M.Tech IInd Year. School of Info. Tech.*, August 2005

[10] Xerox’s AspectJ Team “The AspectJ™ Programming Guide”, *Xerox Corporation, 2002-2003*,
Available online at http://www.eclipse.org/ aspectj/ doc/ next/ progguide/ index.html