DSAW
A Dynamic and Static Aspect Weaving Platform

Luis Vinuesa, Francisco Ortin, José M. Félix and Fernando Álvarez
Computer Science Department, University of Oviedo, Calvo Sotelo s/n – 33007 Oviedo, Spain

Keywords: Aspect Oriented Programming, static, dynamic, reflection, code instrumentation, performance, flexibility.

Abstract: Aspect Oriented Software Development is an effective realization of the Separation of Concerns principle. A key issue of this paradigm is the moment when components and aspects are weaved together, composing the final application. Static weaving tools perform application composition prior to its execution. This approach reduces dynamic aspectation of running applications. In response to this limitation, dynamic weaving aspect tools perform application composition at runtime. The main benefit of dynamic weaving is runtime adaptability; its main drawback is runtime performance.
Existing research has identified the suitability of hybrid approaches, obtaining the benefits of both methods in the same platform. Applying static weaving where possible and dynamic weaving when needed provides a balance between runtime performance and dynamic adaptability. This paper presents DSAW, an aspect-oriented system that supports both dynamic and static weaving homogeneously over the .Net platform. An aspect can be used to adapt an application both statically and dynamically, without needing to modify its source code. Moreover, DSAW is language and platform neutral, and source code of neither components nor aspects is required.

1 INTRODUCTION

Aspect Oriented Software Development AOSD (Kiczales, 1997) is a concrete approach of the Separation of Concerns (SoC) principle. AOSD offers a direct support to modularize different concerns that cut across the main functionality of applications. Separating the application functional code from its crosscutting aspects, the application source code would not be tangled, being easy to debug, maintain and modify (Parnas, 1972). Typical examples of cross-cutting concerns are persistence, authentication, logging and tracing (Ortin, 2004).

The process of integrating the aspects into the main application code is called *weaving* and a tool called *aspect weaver* performs it. The weaving process can be performed statically or dynamically at runtime.

Most programming environments that offer AOSD employ static weavers. Once the final application has been weaved, compiled and executed, it is not possible to add new aspects, or remove existing ones, at runtime. However, there are specific scenarios when it is necessary to adapt running applications in response to runtime emerging requirements (Popovici, 2001), (Zinki, 1997), (Matthijs, 1997), (Segura–Devillechaise, 2003). In these cases, dynamic weavers are a powerful tool to create runtime adaptable software.

The main benefit of static weaving is runtime performance. Since the combination of components and aspects is performed prior to application execution, there is little performance cost when compared to traditional object-oriented development (Böllert, 1999), (Haupt and Mezini, 2004). On the other hand, dynamic weaving AOSD tools imply a performance penalty.

Dynamic weaving provides higher flexibility in the development of software. While developing aspect-oriented applications, the dynamic adaptation mechanism is preferable because it facilitates incremental weaving and makes application debugging easier (Ortin and Cueva, 2002). Upon deployment, aspects that do not need to be adapted at runtime should be woven statically for performance reasons.

Another limitation of existing runtime weavers is a more reduced join-point set, because runtime adaptation is more complicated to be implemented (Blackstock, 2004).

Previous work has identified the appropriateness of integrating both static and dynamic weaving in the same development environment (Blackstock, 2004), (Böllert, 1999), (Gilani et al, 2007). Benefits of both approaches would be obtained in the software development process. This approach is the result of applying the separation of the weaving-time concern to AOSD. This process should be performed transparently, reducing the impact of changing this concern in the application's source code. As an example of this benefit, the programmer could use non-invasive dynamic weaving for agile development. Once the application has been tested, static weaving, where possible, could be applied to obtain a better runtime performance.

In this paper we present DSAW, an aspect platform that support both static and dynamic weaving. DSAW offers the better performance of static weaving and the agile interactive development of dynamic weaving in a transparent way. Moreover, DSAW is language and platform independent and it has a set of join-points wider to most dynamic weaving tools.

The rest of this paper is structured as follows. In the next section we present a classification of AOSD systems based on when the weaving is done and describe some requirements on AOSD. We introduce the basic architecture of the system presented in section 3. Section 4 comments the benefits we obtain with our system. Section 5 gives an overview of some tools that offer static and dynamic weaving and the conclusions are presented in Section 6.

2 ASPECT WEAVING

A classification of AOSD is based on when the weaver is executed. Static tools perform weaving prior to application execution. A well-know example of static weaving is AspecJ (Kiczales, 2001). There are AOSD tools that offer some kind of dynamic adaptation of aspects, being PROSE (Popovici et al, 2001) (Nicoara and Alonso, 2005) and JBoss AOP (JBoss, 2008) two well-known examples of this approach.

Most dynamic AOSD systems are not totally adaptable at runtime. The majority of them require the static specification of which aspects are going to be weaved at runtime. Others offer dynamic addition of new aspects but they do not support dynamic deletion. In addition, the set of join-points they offer is significantly smaller than static ones (Blackstock, 2004) (Vinuesa and Ortin, 2004).

Although static weaving offers undeniable performance benefits, it also involves some limitations. If the weaving process is only static, AOSD is not especially suitable for rapid prototyping development. If we take logging or testing aspects as an example, it is needed to weave, compile, run and debug the application. Runtime contexts should be reproduced and all the information generated by the aspects analyzed. If some error occurs, the application should be modified, re-weaved, re-compiled and re-executed (Böllert, 1997), (Eaddy, 2007c).

In case a dynamic weaver is used, aspects could be added in the exact execution point indicated by the user, and subsequently removed. Moreover, application execution should not be stopped if we want to modify an aspect, and debugging is easier because of the non-intrusive weaving approach.

These different scenarios motivate the use of static weaving where possible and dynamic weaving when needed (Gilani et al, 2007), (Böllert, 1999), (Schröder-Preikschat et al., 2006). A tool that supports both techniques should define aspects independently of when they are weaved. This approach will benefit from both static and dynamic weaving in the same system. The result is the separation of the weaving-time aspect in the AOSD process (Gilani et al, 2007).

Apart from the dynamism an aspect platform would benefit from a language-neutrality feature. Language independence permit the adaptation of applications by aspects developed in a different programming language. This promotes component and aspect reutilization. Furthermore, if platform neutrality is also achieved, aspect-oriented systems could be run over any platform.

Another important characteristic of AOSD systems is the adaptation of binary modules, where source code is not required to adapt the application.

Taking into account all these requirements an AOSD platform must support to obtain the benefits previously described, we have developed the DSAW (Dynamic and Static Aspect Weaving) platform. This system offers both static and dynamic weaving, a wide set of join-points, language and platform neutrality, and an executable-level aspect injection.

3 DSAW

DSAW is a homogeneous static and dynamic weaving aspect-oriented platform. Its main aim is to achieve language and weaving-time neutrality. Not only is it possible to weave aspects both statically and dynamically, but also the same implementation of components and aspects is maintained in both scenarios. The application need not to be changed if we need to make static a dynamic aspect and vice versa. This way, it is possible to use aspect-orientation for rapid prototyping (dynamic weaving) and later optimize the released application (static weaving) without performing any change to its source code. The programmer should finally indicate the trade-off between performance and flexibility, regarding to the system requirements (Gilani et al, 2007).

One of the features that have been considered in the design of DSAW is the set of join-points to be provided. The collection of join-points DSAW offers is similar to the one supplied by AspectJ (Kiczales, 2001). We currently support the following static and dynamic join-points: method and constructor execution, method and constructor call, field and property read and write, field and property write, and exception handling. We also provide before, after, and around aspects.

The design of the platform has been performed following the .Net standard reference (ECMA, 2006), without modifying, neither extending, the semantics of the platform. This fact guarantees complete platform independence, permitting the deployment of our system over any .Net implementation (such as Mono, SSCLI and DotGNU).

DSAW performs software adaptation at the intermediate language (IL) of the virtual machine – executable files. This means that our weaver does not require source code, and it is not language dependent either.

Finally, point-cuts are specified by means of XML documents that specify the mapping between join-points and advices. The schema of this XML documents is an evolution of the one used on the Weave.Net platform (Weave.Net, 2008). We have modified this original schema to provide wide set of point-cut, similar to the one supplied by AspecJ.

This separation between point-cuts and advices improves the reutilization of aspects. In fact, aspects can be also treated as components. They may be adapted by other aspects, statically or dynamically, regardless of its programming language.

3.1 ReadyAOP

DSAW is the enhancement of a previous AOSD tool called Ready (Really Dynamic) AOP (Vinuesa, 2007) (Vinuesa and Ortin, 2004). ReadyAOP is a language and platform neutral system that offers a real separation of aspects at runtime. It is possible to add new aspects (and remove existing ones) to a running application, even if the aspects were developed later than the application. There is no static coupling between components and aspects.

The system was implemented over the standard .Net platform specification, obtaining full language independence. The weaving process is performed at the virtual machine level, without requiring applications source code. Its join-point set is similar to the one offered by AspectJ. Both ReadyAOP and DSAW are freely available at http://www.reflection.uniovi.es.

DSAW extends the core of ReadyAOP to provide the programmer a transparent static and dynamic weaving, maintaining all the advantages of ReadyAOP. This way, it is possible to mix dynamic and static aspects in the same application. It is also possible to change in any moment of the software development process if an aspect must be static or dynamic. The programmer manages the trade-off between performance and flexibility, following the separation of the weaving-time concern.

3.2 System Architecture

Our system is composed of three main elements: 1) the JoinPoint Injector (JPI) performs both static and dynamic weaving; 2) the application server that coordinates components with dynamic aspects; and 3) the aspect framework that consists of a set of interfaces to integrate all the elements of the system in dynamic weaving scenarios.

In our system, there are two stages in applications life cycle. The first one is prior to program execution. Once the application has been compiled, the JoinPoint Injector (JPI) manipulates the program, weaving static aspects and including code to make possible a future dynamic aspect weave.

The second stage of application life cycle is about application execution. The running program is launched together with its statically-weaved aspects. The application server provides the dynamic adaptation of the program at runtime. Any aspect, making use of the interfaces defined in the aspect framework, will be capable of adapting running applications.

The application server is the *Mediator* component of the system (Gamma, 1994). This server works as a registry of running applications, making them capable of being adapted by the aspects. The application server offers the aspects the list of running applications, facilitating their adaptation at runtime.

The aspect framework provides a set of interfaces that orchestrates all the elements of the system at runtime. These interfaces are implemented by the applications in order to be able to be adapted at runtime. Applications do not implement these interfaces explicitly. It is the JPI the responsible for adding this implementation at weaving time.

In DSAW aspects could also be processed by the JoinPoint Injector to become fully-functional applications in our system –i.e. components.

3.3 Static Weaving and JoinPoint Injection

Before to its execution, the application to be adapted is processed by the JPI. The JPI receives the application, aspects to be statically weaved, and the XML document mapping join-points to advices. The JPI generates a new program that includes the main functionality plus the aspects that have been statically weaved.

The JPI also instruments the application with code that allows its dynamic adaptation by aspects at runtime. Concretely, what is added is a Meta-Object Protocol (MOP) that offers dynamic computational reflection (Ortin, 2002). A MOP is a technique that permits the dynamic adaptation of running applications. This behavior is called computational reflection.

This MOP modifies virtual machine semantics such as the message passing mechanism or field access. This way, it will be possible to adapt running applications with dynamic aspects. Finally, the JPI adds other functionalities of the platform such as application registration at startup, intra-application introspection, and application deregistration at exit (Figure 1).

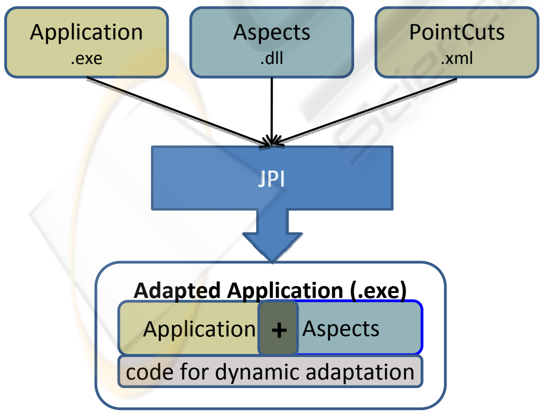

Figure 1: Static weaving and JoinPoint injection.

The JPI performs the following operation:

1. The JPI takes the compiled application, its static aspects, and the XML file that specifies the join-point / advice mapping.
2. Analyzing the IL code, the JPI detects the application join-point shadows -the mapping between join-points and the points in the program text where the compiler actually operates- (Masuhara and Kiczales, 2003) .
3. When a join-point shadow is selected by any point-cut described in the XML document, a call to the selected advice is included into the program's IL code.
4. Besides the code added to the application described in the previous point, a reflective MOP is included in each join-point shadow. This MOP will make the application capable of being adapted at runtime by new runtime aspects. In order to obtain a better runtime performance this MOP injection could be avoided in certain join-point shadows –this is described in the XML file. This is part of the trade-off between performance and runtime adaptability.
5. The JPI adds new code to make the application register in the application server at startup. It is also included a deregistration routine at exit, and the publication of .Net reflective information to permit runtime aspects inspect application's structure.
6. The final application is generated.

3.4 Application Execution

The application, once processed by the JPI, has been weaved together with its static aspects. These aspects are included in the application until its execution is over. If new static aspects should be added, or any existing one removed, the JPI must re-process the application.

However, if the application needs to be adapted at runtime, our system will facilitate the dynamic addition or deletion of aspects. Notice that this behavior is obtained after the instrumentation of the application by the JPI.

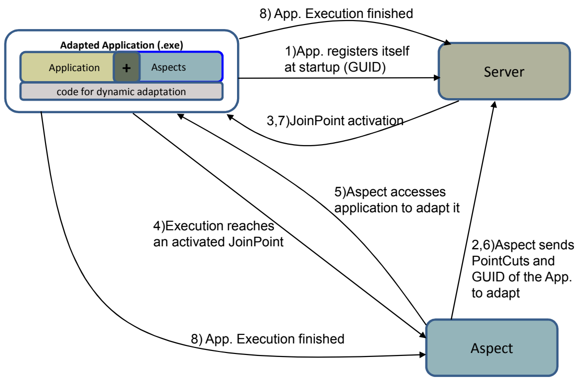
8) App. Execution finished
Adapted Application (.exe)
Application
+
Aspects
code for dynamic adaptation
1)App. registers itself at startup (GUID)
Server
3,7)JoinPoint activation
5)Aspect accesses application to adapt it
4)Execution reaches an activated JoinPoint
2,6)Aspect sends PointCuts and GUID of the App. to adapt
8) App. Execution finished
Aspect

Figure 2: Dynamic adaptation.

These are the steps followed at runtime to dynamically adapt an application with dynamic aspects (Figure 2):

1. At startup the application is registered into the application server with a GUID. This GUID was generated by the JPI during code instrumentation and is used to identify the application in the system.
2. Once the application is running, an aspect may be used to dynamically adapt it. If so, the aspect calls the application server via .Net remoting. The information passed is a reference to the aspect, a point-cut XML document, and the GUID of the application to be adapted.
3. The application server parses the XML document finding the point-cuts required by the aspect. These point-cuts are activated in the application join-points by means of the MOP added in the JoinPoint injection phase. This activation implies the precise call to the aspect at runtime, making use of all the framework services. The result of this process is a dynamic weaving.
4. When the dynamically weaved application execution reaches an activated join-point, the system calls the aspects that have been previously weaved with the application. Aspects should implement the specific interfaces described in the application framework (Vinuesa and Ortin, 2004) to make this happen. This way, the application will send the aspect information regarding the join-point and the own application (including a reference to the application). The aspect has now the opportunity to adapt the application behavior at runtime.
5. The aspect may use the reference to the application to access the latter by means of the publication of .Net reflective services added by the JPI. The operations offered by these reflective services are inspection of the application structure, field values modification, and method invocation.
6. A runtime weaved aspect could have the necessity of changing the point-cuts at runtime. This operation is also offered by the application server. The aspect should send a new XML document specifying the new set of point-cuts. The application server will then analyze this XML file, activating new join-points and deactivating old join-points in the running application.
7. If a dynamic aspect does not want to adapt the application any more, it notifies so to the application server. The application server deactivates the application join-points previously turned on by the aspect. Therefore, the aspect will not receive future notifications.
8. When the application execution finishes, the code added by the JPI notifies all the registered aspects and the application server that the application has exited.

The application server acts as a mediator between aspects and applications. This mediation is only performed when join-points are activated or deactivated. Once these operations have been performed, the application and the aspect interact directly one with the other. The application calls the aspect when an activated join-point is reached. The aspect may inspect the application when a join-point notification is received.

With this design, applications do not need to know the aspects that might be weaved at runtime. At the same time, aspects can be applied to any application, or even aspects, without any static dependency. This behavior reduces coupling and promotes aspect and component reutilization.

We have used .Net remoting to communicate the application server, the aspects and the applications. .Net remoting is a standard service over the .Net platform and is channel independent. It could be even used in distributed environments.

3.5 Aspect Conflict Resolution

DSAW provides the weaving of multiple aspects in the same application join-point. This feature involves the establishment of aspect coordination strategies. The programmer could use this coordination mechanism to control the order and priority of aspect execution.

The coordination mechanism offered by DSAW is based on the classification of aspect in static and dynamic, plus the utilization of aspect priorities.

Since dynamic aspects are aimed at adapting applications to runtime emerging requirements, we have granted dynamic aspects a higher precedence than the static ones.

Static aspects are weaved following a priority strategy that the application developer could change while the application is being developed. Therefore, the execution of static aspects registered in the same join-point will depend on their priority level. The JPI is the responsible for adding this coordination behavior while the application is being statically weaved.

In the case of dynamic aspects, DSAW gives precedence to dynamic aspects. However, conflicts between dynamic aspects are solved following the same priority-based strategy than the static ones. The main difference is that this resolution of dynamic aspects is performed by the application server, whereas static conflicts are solved by the JPI. Apart from that, dynamic aspects may modify their priority at runtime, depending of their specific requirements.

3.6 Runtime Performance

The main drawback of dynamic weaving is runtime performance (Böllert, 1999) (Haupt and Mezini, 2004). A statically-weaved aspect-oriented program may be almost as efficient as it was developed without an AOSD approach (Böllert, 1999). However, the process of adapting an application at runtime, as well as the use of reflection, induces a certain overhead at the execution of an application (Popovici et al., 2003).

In our previously developed ReadyAOP system, runtime performance penalties are produced by two key features: 1) the MOP injected by the JPI to dynamically enable or disable join-points, and 2) the communication between different modules of our system.

The first feature implies a dynamic check of join-point activation. It is also performed some verifications that manipulate the state of the stack, implying a performance cost. The average performance cost of this primitive is 70%.

The second penalization of runtime performance is derived from the communication between aspects, applications, and the application server. We have used .Net remoting for this purpose. The performance cost is linear with respect to the amount of information exchanged. This inter-process communication implies the highest performance cost at runtime. This is the reason why DSAW is an improvement of ReadyAOP. One of the benefits of static weaving is the performance benefit obtained.

While applications are being developed, DSAW offers a dynamic-weaving approach to facilitate the creation of AOSD applications. This facilitates the incremental development and the application of the agile “fix-and-continue” debugging scheme (Ortin and Cueva, 2002) (Dmitriev, 2002). Prior to the deployment of the application, aspects that do not require dynamic weaving could be statically injected to the final application. This way, runtime performance of the whole application will be notably improved. This is the main benefit obtained from separating the weaving-time concern in our system.

4 SYSTEM BENEFITS

The AOSD platform presented in this paper provides the following benefits:

- Both static and dynamic weaving is supported in a homogeneous way. Aspects could be injected in an application statically or at runtime without any modification.
- Separation of the weaving-time concern. The homogeneity described in the previous point provides the translation between static and dynamic weaving without changing the source code of the application, neither the aspects.
- Combination of dynamic and static adaptation. Not only it is possible to statically and dynamically aspectize applications, but it is also viable to combine both approaches in the same program.
- Really dynamic aspect weaving. Applications need not to know anything of future aspects to be weaved. Moreover, aspects could be added to or removed from any running application.
- Language neutrality. DSAW works at the intermediate language (IL) of the .Net platform. Components and aspects could be implemented in any programming language.
- IL instrumentation. The weaving process is performed by means of IL instrumentation, once applications and aspects have been compiled. This feature is important when we need to adapt components developed by third parties and its source code is not provided.
- Platform neutrality. The use of the .Net platform makes DSAW capable of being executed over any standard .Net implementation.

- Wide join-point set. Our system offers an ample set of join-points, similar to the one offered by AspectJ. These join-points are provided for both dynamic and static weaving.
- Aspect and components reutilization. The non-invasive weaving technique followed and IL-level instrumentation promotes aspect (and component) reutilization. This is an effect of the null coupling between aspects and components.
- Aspects aspectation. Aspects could be considered as components in DSAW. This involves the aspect adaptation by other aspects.

5 RELATED WORK

There exist many static weaving AOSD tools, and there are also some dynamic ones. However, there are few that offer both approaches.

Wicca is one example of a dynamic and static aspect-oriented system (Eaddy, 2007a). Wicca has been developed over the .Net platform making use of the Phoenix framework –a back-end compiler infrastructure (Phoenix, 2008). Wicca performs static weaving by means of code instrumentation. However, dynamic weaving is achieved using the debugging API of the CLR. The dynamic weaving is released in an alpha version, and it does not support dynamic aspect deletion. The static and the dynamic weaver are not equivalents. This means that static weaving is more expressive than the dynamic one (Eaddy, 2007b). Moreover, Wicca makes use of the debugging API specific of a single implementation of .Net (the CLR), losing the platform neutrality. Runtime performance is poor because applications should be executing in debugging mode, enabling the edit-and-continue support of the CLR (Eaddy, 2007b).

AOP.NET, also known as NAop, is another dynamic and static weaving proposal –no implementation has been released (Blackstock, 2004). Its design follows a proxy-based component decoration. This proxy is used in both static and dynamic scenarios. The weaver uses a proxy class instead of any class in a component. The proxy is capable of adapting the behavior of its decorated class. Depending on the point-cuts, the proxy delegates its functionality on the original class or it calls the registered aspects. The static weaver performs this process prior to the application execution; the dynamic weaver does it at runtime.

The LOOM.NET project provides dynamic and static weaving over the same core implementation, using the .Net platform (Schult, 2003). Rapier LOOM.NET is the dynamic weaver. Point-cuts in aspects are expressed by means of .Net's custom attributes. At load-time, the application is weaved together with its aspects. Applications and aspects should be linked, prior to their execution. It is currently being developed a static weaver, called Gripper-LOOM.NET, which is in alpha version (Köhne et al, 2005) (Schult, 2008). The syntax of the point-cut language description is not the same in both weavers. This makes it difficult to convert static aspects into dynamic ones. The dynamic weaver requires the source code of applications and provides a reduced set of join-points (Schult, 2003) (Köhne et al, 2005); it does not support dynamic aspect deletion either (Frei et al, 2004).

6 CONCLUSIONS

AOSD is an effective approach to obtain the benefits of the Separation of Concerns principle. In this paradigm, final applications are built from the main functionally of the application plus cross-cutting aspects.

There are many tools that support AOSD. Some of them offer application weaving prior to its execution, while others provide this adaptation at runtime. Although the static approach is suitable in most cases, dynamic adaptation may also be required when the application should respond to runtime emerging contexts and requirements. Static weaving supports efficient AOSD, whereas dynamic weaving involves runtime application adaptation.

This paper describes the architecture of DSAW, a dynamic and static weaving platform that provides language and platform neutrality, and the separation of the weaving-time concern.

DSAW has been designed over the .Net platform, benefiting from its features. Both weavers are built by means of IL code instrumentation. This makes DSAW language neutral, permits the adaptation of legacy applications, and promotes aspects and components reutilization.

Both dynamic and static weavers supply the same rich set of join-points. An aspect may be weaved statically or dynamically without changing its source code, not the component's one. This facilities the fix-and-continue development at first stages of software development, and the later efficient static weave when the application is about to be released. That is, DSAW completely separates the weaving-time concern. The programmer may modify the flexibility / performance trade-off during the development life cycle.

Finally, it is possible to create applications with aspects that have been statically weaved, together with aspects that are later added at runtime. The conflict resolution mechanism is based on making dynamic aspects have precedence over static ones. Resolution between aspects of the same kind follows a priority-base strategy.

REFERENCES

Blackstock, M. Aspect Weaving with C# and .NET. 2004. *www.cs.ubc.ca/ michael/ publications/ AOPNET5.pdf*

Böllert, K. On Weaving Aspects. 1999. *Proceedings of the Workshop on Object-Oriented Technology*, p.301-302.

Dmitriev, M., 2002. Applications of the Hotswap Technology to Advance Profiling. In *ECOOP 2002 International Conference*.

Eaddy, M., 2007a. Wicca 2.0: Dynamic Weaving using the .Net 2.0 Debugging API. In *AOSD 2007*.

Eaddy, M., 2007b. *Wicca.* [Online] http://www1.cs.columbia.edu/~eaddy/wicca/ [accessed January, 15, 2008].

Eaddy, M., Aho, A., Hu, W., McDonald, P., Burger, J. 2007c. Debugging Aspect-Composed Programs. *Software Composition, 2007*. Portugal.

ECMA, 2006. *Standard ECMA-335: Common Language Infrastructure (CLI).* [Online] Available from http://www.ecma-international.org/publications/ standards/Ecma-335.htm [accessed December 2007].

Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley

Frei, A., Grawehr, P. & Alonso, G., 2004. A Dynamic AOP-Engine for .NET. *Technical Report 445*, Department of Computer Science, ETH Zürich.

Gilani, W., Scheler, F., Lohmann, D., Spinczyk, O., Schröder-Preikschat, W. 2007. Unification of Static and Dynamic AOP for Evolution in Embedded Software Systems. 2007. *6th International Symposium on Software Composition* Braga,Portugal.

Haupt, M. & Mezini, M., 2004. Micro-Measurements for Dynamic Aspect-Oriented Systems *Proc. of Net.ObjectDays 2004 (NODe)*, LNCS 3263.

JBoss AOP homepage, 2008. [Online] http://labs.jboss.com/jbossaop/ [February 28, 2008]

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W., 2001. An Overview of AspectJ. *Proceedings of the ECOOP 2001*.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M. & Irwin, J., 1997. Aspect Oriented Programming. *Proceedings of ECOOP 1997*, vol. 1241 of LNCS.

Köhne, K., Schult, W. & Polze, A., 2005. Design by contract in .NET Using Aspect Oriented Programming.

Masuhara, H., Kiczales, G., 2003. A Compilation and Optimization Model for Aspect-Oriented Programs. *Compiler Construction (CC2003), LNCS 2622.*

Matthijs, F., Joosen, W., Vanhaute, B., Robben, B. & Verbaten, P., 1997. Aspects should not die. In: *ECOOP Workshop on Aspect-Oriented Programming.*

Nicoara, A. & Alonso, G., 2005. Dynamic AOP with PROSE. *Proceedings of ASMEA 2005 in conjunction with CAiSE 2005*, Porto, Portugal.

Ortin, F., Cueva, J. 2002. Implementing a real computational–environment jump in order to develop a runtime–adaptable reflective platform. *ACM SIGPLAN Notices, Volume 37, Issue 8.*

Ortin, F., Lopez, B., Perez-Schofield, J.B.G. 2004. Separating Adaptable Persistence Attributes through Computational Reflection. *IEEE Software, Volume 21, Issue 6*.

Parnas, D, 1972. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, Vol. 15, No. 12.

Phoenix, 2008. (Online). research.microsoft.com/phoenix (accessed February, 28, 2008)

Popovici, A., Gross, T. & Alonso, G., 2001. Dynamic Homogenous AOP with PROSE. *Technical Report*, Dept. of Computer Science, ETH Zürich.

Popovici, A., Alonso, G. & Gross, T., 2003. Just in Time Aspects: Efficient Dynamic Weaving for Java. *AOSD 2003 Proceedings*.

Schult, W., Trögger, P., 2003. Loom.NET – an Aspect Weaving Tool. *Workshop on Aspect-Oriented Programming, ECOOP'03,* Darmstadt.

Schult, W., 2008. Documentation Gripper-LOOM.Net. [Online] http://www.gripper-loom.net [accessed January, 15, 2008].

Schröder-Preikschat, W., Lohmann, D., Gilani, W., Schele & F., Spinczyk, O., 2006. Static and dynamic weaving in System Software with AspectC++. In *HICSS '06 Mini-Track on Adaptive and Evolvable Software Systems, IEEE*, January.

Segura-Devillechaise, M., Menaud, J., Muller, G.& Lawall, J., 2003. Web Cache Prefetching as an Aspect: Towards a Dynamic-Weaving Based Solution. *AOSD 2003 Proceedings*, pp: 110-119.

Vinuesa, L. & Ortín, F., 2004. A Dynamic Aspect Weaver over the .NET Platform. *Metainformatics International Symposium, MIS 2003*. LNCS 3002.

Vinuesa, L, 2007. Separación Dinámica de Aspectos independiente del Lenguaje y Plataforma mediante el uso de Reflexión Computacional. Ph. D. Dissertation. University of Oviedo.

Weave.Net, 2008. Weave.NET homepage http://www.dsg.cs.tcd.ie/index.php?category_id=193. [Accessed on February, 25, 2008].

Zinky, J., Bakken D. & Schantz, R., 1997. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems (TAPOS).*