

# Benefits and Challenges of a Class-Based Design for Dynamic Aspects in DAC++

Sufyan Almajali  
Illinois Institute of Technology  
3300 S. Federal St.  
Chicago, IL 60616, USA  
almasuf@iit.edu

Tzilla Elrad  
Illinois Institute of Technology  
3300 S. Federal St.  
Chicago, IL 60616, USA  
elrad@iit.edu

## ABSTRACT

This paper presents an impact analysis of using a class-based design aspect on the different software development properties of dynamic weaving systems. Class-based design aspect allows the use of object oriented (OO) capabilities when defining aspects. The impact is analyzed with respect to code modularity, reusability, adaptability, evolvability, supportability, and comprehensibility. The analysis is based on two different perspectives, that of application developer and compiler designer. Dynamic Aspect C++ (DAC++) is used as a case study for this analysis.

## 1. INTRODUCTION

Dynamic weaving systems enable application developers to design applications that can adapt at runtime to different application and system situations. This flexibility gives the application user and the developer the ability to evolve their application and adapt to new scenarios with minimum work in terms of programming, compilation, installation, and application deployment.

The three main classes of dynamic weaving in AOP are: Pre-Run-Time Instrumentation, Run-Time Event Monitoring, and Run-Time Weaving. Examples of Pre-Run-Time instrumentation are: EAOP[7], JAC[9], JBoss AOP [10], PROSE 2[12], and DAO C++[1,2]. These approaches either modify the application classes before compilation using a preprocessor (PROSE 2,DAO C++) at application loading time (JAC, JBoss AOP) or as their byte code about to be compiled by the just-in-time compiler (PROSE). An example of Run-Time Event Monitoring is PROSE 1 [11]. In this category, program execution is intercepted by the system at the possible joinpoints, and an advice code is executed if the aspect is activated for that joint point. PROSE 1 depends on JVM debugging capabilities to accomplish the runtime weaving. The last category of these systems is Run-Time Weaving systems. Examples of this category are: Steamloom [5], a .NET-based approach [13 ], DAC++ [1,2,4], and AspectS [8]. In these systems, the actual weaving and unweaving are performed at runtime.

In general, all the above mentioned systems follow one of two approaches in defining an aspect. The first approach is to have a unique syntax and semantic for an aspect by adding new language constructs to define the aspect and the different aspect components. Examples of systems use new syntax for aspect definition are AspectWerkz [15] and PROESE[11].

Many other systems have adopted the idea of using regular class definition to define an aspect and given it a different runtime

semantic during program execution. Examples of these systems are: Jboss [10], JAsCo [14], JAC [9], and DAC++ [1,2]. Using a class-based design for the aspect means that you can use object oriented (OO) capabilities when dealing with aspects. The OO built-in capabilities include class composition, class inheritance, member variables, member methods, method overloading, and virtual function support. Making all or some of these capabilities available during aspect definition gives the application extra flexibility and power. This extra power, however, is not free from complexities and side effects in terms of language semantic and compiler support.

This paper discusses the impact of using class-based dynamic aspects from application developer and compiler designer perspectives. The paper is structured as follows: Section 2 briefly discusses DAC++ compiler and its aspect syntax. Section 3 presents an analysis of the positive impact of using class-based aspects. Section 4 discusses the challenges associated with such a design approach. Section 5 concludes the paper.

## 2. Case Study: DAC++ Compiler

DAC++ [1,2] is a research compiler that supports a runtime weaving system for the C++ language. DAC++ enables aspect to be woven and unwoven at runtime. A new aspect definition (advice or pointcut expressions) can be introduced or changed at runtime. In addition, DAC++ supports the notions of network aspects [3] and inter-process aspects [4] that are useful for networking and distributed computing applications.

Figure 1 shows an aspect definition example using DAC++ syntax. Lines 1 through 7 show the definition of one aspect "Authentication aspect," to be used with a web browser application. The web browser application sends two types of web requests, HTTP and HTTPS. The aspect is defined by using regular class definition in C++ syntax. No new syntax is included, but new semantic is supported for aspect classes. To define an aspect, the new aspect class from a base class "Aspect". DAC++ compiler treats all classes inherited from "Aspect" class as an aspect and enables them to be woven and unwoven at runtime. According to DAC++, the aspect class definition includes all aspect information (advice, aspect member variables, and aspect member methods) except pointcut expression information and weaving specification. Lines 9 through 19 show how to define the aspect pointcut expression for the authentication aspect. To define a pointcut designator, first, you need to define the value for one or more pointcut dimensions (method, class, class instance, application, thread, variable, or network). In our example, Line 9

shows the pointcut method dimension being used. All web client sockets httpCS and httpsCS will be targeted by the authentication aspect. httpCS represents a user-defined client socket to send and receive http traffic, while httpsCS is the same, but for https traffic. Second, a subpointcut designator (SPCD) is required to join the multiple dimensions you selected in your definition. In Figure 1, lines 10 and 11 show an example of defining a SPCD for the authentication aspect. Then, a pointcut designator (PCD) is defined by joining one or more SPCD to it. An example is shown in lines 12 and 13.

```

1  class Authentication: public Aspect {
2  Authentication ();
3  void advice() {
4      Authenticate();
5      }
6  }
7
8  int main() {
9  MethodPC http_requests(Exact,"
10 httpCS.<<,httpsCS.<<");
11 SPCD http_spdc;
12 http_spdc.setmethodpc(http_requests);
13 PCD httpPC;
14 httpPC.add(http_spdc);
15 WeaveSpecs w1;
16 w1.setweavetarget(Method_W);
17 w1.setweavetype(Pre_T);
18 Authentication auth_aspect;
19 auth_aspect.setweavespecs(w1);
20 auth_aspect.setpcd(httpPC);

```

**Figure 1. Aspect Definition in DAC++**

The WeavingSpecs defined in lines (14 to 16) is the last aspect component that needs to be defined. The weaving specifications include the weaving target and its type. The three main targets are: Method\_W, Variable\_W, and Application\_W. The type values that can be assigned to each one are: (Before\_T, After\_T, and Around\_T), (Before\_T, and After\_T), and (Start\_T and Finish\_T), respectively. For example, to make an aspect target a specific method or methods using Pre-weaving mode, the weave type is set to be Method\_W and the weaving type is set to be Pre\_T. In addition, DAC++ compiler has a set of AOPEngine APIs that is used to weave, unweave, and load new aspects. The AOPEngine::Weave(“aspect instance name”) function is used to weave aspects at runtime. AOPEngine::Unweave(“aspect instance name”) function is used to unweave aspect. DAC++ enables defining new aspect at runtime. Programmers can compile aspects alone. This generates the aspect metadata along with the object code of the aspect. To introduce a new aspect at runtime, AOPEngine::AddNewAspect(“ path to meta data file”, “aspect name”) function is used. The AOPEngine::CreateAspectInstance(“aspect name”, PCD, WeaveSpecs) function enables programmers to create an aspect instance that can be woven as any other aspect defined at initial application deployment time.

### 3. Benefits of Using a Class-Based Design for Dynamic Aspects

#### 3.1 Modularity and Code Reusability

As shown in authentication aspect example in figure 1, class-based aspects enable programmers to structure the aspect as a set

of separate components that are integrated to form the overall aspect. According to our aspect example, the components are: aspect body (advice and internal members), pointcut expression, and weaving specifications. Each one of these components is represented as a regular C++ class. To build an aspect instance, a composition among these components is performed using aspect instantiation as shown Figure 1 (lines 17-19). This separation results in a more modular aspect representation. In addition, programmers can define their pointcut expressions and/or weaving specs to be reusable by multiple aspect types or aspect instances. Figure 2 shows an example of reusing the pointcut expression (PCD) defined in figure 1 by two different aspects.

```

1  Authentication auth_aspect;
2  Encryption enc_aspect;
3  auth_aspect.setpcd(httpPC);
4  enc_aspect.setpcd(httpPC);

```

**Figure 2. PCD Reusability**

Another example of code reusability is shown in Figure 3. Two different PCDs are defined, httpPC1 and httpPC2. Each one of these PCDs is associated with different Authentication aspect instance ( Lines 7 and 8). The first instance is supposed to authenticate using server 1, while the second instance is supposed to use server2. The same aspect advice is used, the only difference is that the Authenticate method authenticates according to the server name defined for that aspect instance. The OO constructor mechanism simplified the design of aspect reusability.

```

1  class Authentication: public Aspect {
2  Authentication (char *);
3  private
4      char * server_name;
5  void advice() { Authenticate(); ... } }
6  ...
7  Authentication auth_aspect1("s1");
8  Authentication auth_aspect2("s2");
9  auth_aspect1.setpcd(httpPC1);
10 auth_aspect2.setpcd(httpPC2);

```

**Figure 3. Code Reusability by Aspect Instantiation**

#### 3.2 Code Adaptability

Class-based aspect design improves application adaptability by enabling different aspect components to change at runtime and adapt to a new situation. Figure 4 shows an example of how the authentication aspect PCD adapts from httpPC1 to httpPC2 at runtime.

```

1  auth_aspect.setpcd(httpPC1);
2  Weave(&auth_aspect);
3  ..... // do some work
4  Unweave(&auth_aspect);
5  auth_aspect.setpcd(httpPC2);
6  Weave(&auth_aspect);

```

**Figure 4. PCD Runtime Adaptation Example**

Line 1 shows how the “auth\_aspect” is set to use “httpPC1” PCD. To start using the aspect, line 2 shows how weaving an aspect is performed. To adapt the running “auth\_aspect” to a new PCD, lines 4 through 6 show how we can associate a new PCD “httpPC2” with “auth\_aspect”. Unweaving and reapplying the weaving are necessary. Another example that shows how class-based aspect can help in improving code adaptability is presented in figure 5. The example shows how we can create two different versions of the same aspect and make the system switch between these two versions at runtime according to the user needs.

```

1 class Authentication: public Aspect {
2   Authentication (char *);
3   private
4     char *   server_name;
5   void advice() { Authenticate(); .. }
6   ...
7   Authentication   auth_aspect1("s1");
8   Authentication   auth_aspect2("s2");
9   auth_aspect1.setpcd(httpPC1);
10  auth_aspect2.setpcd(httpPC1);
11  Weave(&auth_aspect1);
12     ..... // do some work
13  Unweave(&auth_aspect1)
14  Weave(&auth_aspect2);

```

Figure 5. Aspect Instantiation and Runtime Adaptation

### 3.3 Code Comprehensibility

Making the source language the aspect language, leads to an easier programming design, overall. Programmers can apply the concepts of AOP in their systems using one-language syntax for the program components, classes and aspects. Class-based design for aspects enables programmers who are familiar with OO languages to learn quickly how to define aspects. In addition, the modularity introduced by the class-based design for aspects leads to better understanding of the AOP part of the application. This can be valuable, especially when multiple aspects are defined in the application. On the other hand, using one-language syntax to implement two different semantics (Aspects and Classes) may introduce confusion for new programmers. Programmers still need to distinguish between Aspects and Classes as a two different program components that are used differently and support different semantics.

### 3.4 Code Evolvability

For the changeability/extensibility of the application, the system may change in two general areas– the aspect itself may need to be modified, or the core system may need to be modified. Class-based aspects enable modular design of the aspect. Aspect modularity makes the design more flexible toward new changes or extensions. For example, a new target (pointcut expression) can be designed for the authentication aspect. Implementing this will not require editing the aspect body. The programmer simply needs to define a new PCD object and use it with an already defined aspect body. The same thing happens if a new aspect body (advice for example) must be implemented. There is no need to edit the aspect PCD object(s). The aspect body alone needs to be edited or implemented.

In addition, the improvement gained in code modularity, code adaptability, and understandability reduces the overall design complexity of the application. This simplifies the process of evolution and increases the pace of upgrading and adopting new application designs to serve new solutions.

## 4. Challenges of Using a Class-Based Design for Dynamic Aspects

Although there are multiple advantages of using class-based design for dynamic aspects, this design approach is not free from challenges and difficulties. In this section, the main problem faced in our self-designed compiler DAC++ is addressed.

### 4.1 Compiler Supportability

To support dynamic C++ aspects of a class-based structure, multiple things had to be implemented. As language Syntax, there was no addition, but as class semantic, the interpretation differs depending on the type of the class, aspect class or regular core class. Figure 6 shows the general structure of the DAC++ compiler.

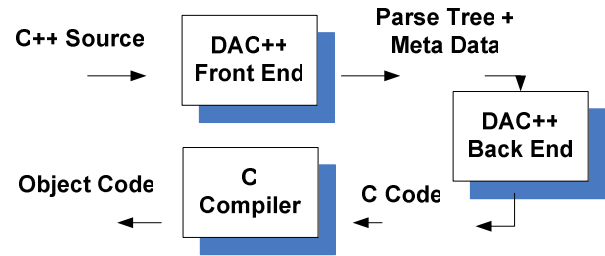


Figure 6. DAC++ Compiler

The front end of the DAC++ compiler generates metadata about the source program and its structure using OpenC++ system. The purpose of the metadata is to enable the AOP engine to keep track of the different components needed to weave the aspect at runtime. DAC++ generates part of the metadata by the front end of the compiler. The rest of the metadata is generated at loading, linking, and runtime. Metadata information changes during program runtime to reflect the aspect weaving and unweaving status. The back end generates the object code of the program using two stages of translations. The first stage includes translating the intermediate representation to C code. The second stage uses the C compiler to generate the final object code of the compiler. One of the main tasks associated with the back end is the generation of the runtime linking information needed during program running time. In addition to this, a runtime AOP engine runs as a part of the final compiled application to enable runtime weaving and unweaving of aspects using the gathered metadata information: classes, aspects, linking, and others.

This implementation enabled support to runtime weaving for C++ using class-based dynamic aspects. Another main future goal is to establish the basic compiler infrastructure for Intro Aspects support. To accomplish this task, major work was required on the

backend of the compiler in addition to defining new semantic of the aspect classes. This extra compiler complexity was required to support runtime weaving for a strong typed language like C++.

## 4.2 Complete OO Support

Class-based aspects mean that our aspect can be defined using the full power of OO class design. Unfortunately, supporting ALL OO features for dynamic AOP systems will increase the complexity of the compiler design and require a lot of research to define the new semantic behaviors that will appear in these cases. For example, the inheritance feature of OO will improve code modularity and reusability, but it will complicate the runtime weaving process. Suppose aspect must be defined as an inheritance from another aspect, then supporting runtime weaving/unweaving has multiple semantic alternatives that can lead to different results. For example, if there are base and extended aspects and both are woven at runtime, what happens if the base class changes? What is the impact of this on the extended aspect? Furthermore, what about multiple-inheritance supported by C++? Which advice must be followed? How should the virtual function support in these cases be handled?

Due to the complexity of this issue with runtime systems, DAC++ supports a subset of OO capabilities for aspect definition.

## 4.3 Introduction Aspects

Introduction aspects enable adding new aspect members like member variables and functions to other classes in the application. From language syntax and programmer perspective, class-based aspect design supports better Intro Aspects. From implementation perspective, this adds extra complexity to the compiler design. In this class, the compiler is supposed to support runtime class composition. In addition, maintaining application consistency when weaving and unweaving intro aspects will be a complicated issue.

## 5. Conclusion

This paper presented an impact analysis of class-based aspect design on dynamic AOP systems. Class-based aspect design enables programmers to use OO capabilities in their design of the aspect structure. DAC++ compiler was used as a case study in this paper. In general, class-based aspect design allows programmers to use one-language syntax to implement modular, adaptable, and evolvable applications. Supporting complete OO design capabilities for runtime aspects is not available yet, and requires further research to define the semantic and behavior of the different programming cases.

## 6. ACKNOWLEDGMENTS

This work is partially supported by CISE NSF grant No. 0137743.

## 7. REFERENCES

[1] Almajali, S. and Elrad, T. A Dynamic Aspect Oriented C++ using MOP with Minimal Hook Weaving Approach. *In*

*Dynamic Aspect Workshop*, Lancaster, England. March 2004.

- [2] Almajali, S. and Elrad, T., Coupling Availability and Efficiency for Aspect Oriented Runtime Weaving Systems , Proceedings of the 2005 Dynamic Aspects Workshop (DAW'05) as part of AOSD'05 (Chicago, USA, March 2005) <http://aosd.net/2005/workshops/daw/AlmajaliElrad.pdf>
- [3] Almajali, S. and Elrad, T., Dynamic Network Policies using Aspect Oriented Network Framework, Proceedings of the IEEE Fifth International Conference on Networking (ICN'06), Mauritius, April 2006.
- [4] Almajali, S. and Elrad, T., An Object Model for Inter-Process Aspects, Submitted to European Conference on Object-Oriented Programming (ECOOP'06) ( Nantes, France, July 2006). <http://www.iit.edu/~almasuf/papers/ecoop06.pdf>
- [5] Bockisch, C., Haupt, M., Mezini, M. and Ostermann, K. Virtual Machine Support for Dynamic Join Points. In AOSD 2004 Proceeding. ACM Press, 2004
- [6] DAC++ Home Page. <http://www.iit.edu/~almasuf/dacpp.html>
- [7] Douence, R. and Sudholt, M. A Model and a Tool for Event-Based Aspect-Oriented Programming (EAOP). Technical Report 02/11/INFO, Ecole des Mines de Nantes, 2002.
- [8] R. Hirschfeld. AspectS -- Aspect-Oriented Programming with Squeak. In M. Aksit, M. Mezini, R. Unland, editors, Objects, Components, Architectures, Services, and Applications for a Networked World, LNCS 2591, pp. 216-232, Springer, 2003
- [9] JAC Home Page. <http://jac.aopsys.com>
- [10] JBoss AOP Home P. <http://www.jboss.org>
- [11] Popovici, A., Gross, T. and Alonso, G. Dynamic Weaving for Aspect Oriented Programming. In AOSD 2002 Proceedings. ACM press, 2002.
- [12] Popovici, A., Gross, T. and Alonso, G. Just-in-Time Aspects. In AOSD 2003 Proceeding. ACM Press, 2003
- [13] W. Schult, A. Polze: Dynamic Aspect-Weaving with .NET. Workshop zur Beherrschung nicht-funktionaler igschaften in Betriebssystemen und verteilten Systemen, TU Berlin, Germany, 7-8 November 2002.
- [14] Vaderperren, W. and Suvee, D. Optimizing JAsCo dynamic AOP through HotSawp and Jutta. *In Dynamic Aspect Workshop*, Lancaster, England. March 2004.
- [15] Vasseur, A. Dynamic AOP and Runtime Weaving for Java-How does AspectWerkz address it? *In Dynamic Aspect Workshop*, Lancaster, England. March 2004.