Designing an Aspect-Oriented Framework in an Object-Oriented Environment

Constantinos A. Constantinides¹, Atef Bader¹, Tzilla H. Elrad¹, Mohamed E. Fayad², P. Netinant¹

¹ Concurrent Programming Research Group
Department of Computer Science
Illinois Institute of Technology
{conscon, elrad, netipan} @charlie.cns.iit.edu
bader@delta.csam.iit.edu

² University of Nebraska, Lincoln
fayad@cse.unl.edu

Abstract

Separation of concerns is at the heart of software development, and although its benefits have been well established, the core problem remains how to achieve it. For complex software systems the solution is still debatable and it is a major research area. Object Oriented Programming (OOP) works well only if the problem at hand can be described with relatively simple interface among objects. Unfortunately, this is not the case when we move from sequential programming to concurrent and distributed programming. The September 1993 CACM issue was devoted to the problematic marriage between OOP and Concurrency [Cohen 93]. Since then, numerous workshops, articles and books have attempt to tackle the problem. The core complexity is that concurrent and distributed systems manifest over more than one dimension. Features such as scheduling, synchronization, fault tolerance, security, testing and verifications are all expressed in such a way that they tend to cut across different objects. Hence, simple object interfaces are violated and the traditional OOP benefits no longer hold. One of the current attempts to resolve this issue is the Aspect Oriented Software Architecture. To address this multi-dimensional structure of concurrent systems we distinguish between components and aspects. Aspects are defined as properties of a system that do not necessarily align with the system's functional components but tend to cut across functional components, increasing their interdependencies, and thus affecting the quality of the software. Although not bound to OOP, Aspect-Oriented Programming (AOP) is a paradigm proposal that retains the advantages of OOP and aims at achieving a better separation of concerns. In this paper we provide an assessment of AOP and we discuss the architecture of an aspect-oriented framework. The goals of our proposal is to achieve an improved separation of concerns in both design, and implementation, to provide adaptability, and to support the complex interaction among non-orthogonal aspects.

1. The “Code Tangling” Problem

The traditional approach for organizing software systems has been based on some form of functional decomposition. A problem is broken down into sub-problems that can be addressed relatively independently. Current programming languages and paradigms support implementation,
and composition of sub-parts into whole systems through the availability of some modular unit of functionality, or component. The concept of component is well supported by existing programming technologies with a large collection of design practices, and notations. In the Object Oriented Programming (OOP) paradigm, components can be represented as classes and objects. At the same time, many systems have properties that do not necessarily align with the functional components of the system. In the OOP paradigm, these properties are not localized to objects. These properties do not arise randomly, but tend to constitute emergent entities, arising during execution. Example properties include synchronization, scheduling, resource allocation, performance optimizations, failure handling, persistence, communication, replication, coordination, memory management, and real-time constraints.

Manually programming these properties into the system’s functionality using current languages or methodologies results in these properties being spread throughout the code. Aspects were originally introduced by [Kiczales et al. 97a, b] and are defined as system properties that tend to cut across functional components, increasing their interdependencies, and resulting in what was coined as “the code-tangling problem.” This code tangling makes the source code difficult to develop, understand and evolve by destroying modularity and reducing software quality.

The major goals of OOP are abstraction, modularity, and code reuse. On the other hand, the inadequacy of the OOP paradigm to address a complete separation of concerns was also addressed by [Ossher and Tarr 99], where it was advocated that OOP provides only one dimension along which concerns can be separated. This was coined the “tyranny of dominant decomposition”.

![Figure 1. Cross-cutting between components and aspects. The same aspect may interact with all components (e.g. Am) or only some of them (e.g. A1), and vice versa (e.g. C2, and C1). (Adapted from [Bardou98])](image)

2. Aspect-Oriented Programming: Addressing Complete Separation of Concerns

Although aspects can be analyzed and designed relatively separately from the basic functionality of the system, at the implementation level components and aspects must be combined (weaved) together. In [Kiczales et al. 97b] the authors addressed the problem of code tangling, and proposed a new approach to express each of the system's aspects of concern in a separate linguistic construct. Their proposal would automatically combine aspect and component descriptions into a
final executable form using automatic tools. This approach was coined Aspect-Oriented Pro-gramming (AOP). Although not bound to OOP, the AOP paradigm retains the advantages of OOP and aims at avoiding the tyranny of dominant decomposition. In [Mens et al. 97, Lorenz98] AOP is viewed as a general modeling mechanism, which applies to all phases of the life cycle of the software.

3. Architectural Issues

One difference in the proposals for supporting AOP resides in the way in which aspects are weaved across the functional components of the system. In this section we discuss issues related with current AOP architectures.

3.1 Language Support for AOP

Current languages and paradigms support a number of modular representations such as procedures, and objects. They further support composition of modules into whole systems. In [Mens et al. 97] it was argued that current languages do not provide the right abstraction for the description of aspects. The authors also addressed the importance of having appropriate languages for the expression of aspects and it was argued that aspect languages make aspect code more concise and easier to understand. If aspects are expressed in domain-specific languages, one needs an aspect language for every type of aspect and an automatic weaver tool would implement one (or more) aspect languages. Linguistic approaches automatically weave aspects into the functional behavior of the system in order to produce the overall system. Additionally there are proposals of specific languages for the support and implementation of AOP versus extensions to general-purpose languages. Examples of the former include COOL, RIDL [Lopes97], D²-AL, IL and TyRuBa [De-Volder98]. Examples of the latter include AspectJ which extends the Java language [Gosling et al. 96] with aspect support, Replication-Framework [Fabry98], JST [Seinturier99], Luthier-MOP and Kava [Welch and Stroud 99]. [Czarnecki98] mentions three approaches for the implementation of aspects: 1) using libraries, 2) using domain-specific aspect languages and 3) designing language extensions for the support of aspects. A comparison between languages or language extensions on the one hand and libraries on the other is shown in Table 1. Further, language extensions seem to have a number of advantages over domain-specific languages: 1) they are more scalable, and 2) they allow for the reuse of the compiler infrastructure and language implementation.

<table>
<thead>
<tr>
<th>LANGUAGES OR LANGUAGE EXTENSIONS</th>
<th>LIBRARIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declarative representation: Requirements directly translated into the language.</td>
<td>Less direct representation.</td>
</tr>
<tr>
<td>Simpler analysis and reasoning: language constructs capture programmer’s intentions.</td>
<td>Analysis often not practicable.</td>
</tr>
<tr>
<td>Forces capture of all important design information.</td>
<td>Design information gets lost.</td>
</tr>
</tbody>
</table>

Table 1. Comparison between languages, language extensions and libraries for the implementation of aspects. (Adapted from [Czarnecki98]).
3.2 Static and Dynamic weaving

One difference in the proposals for supporting an Aspect-Oriented Software Architecture (AOSA) resides in the way in which aspects are weaved across the functional components of the system. One important issue is whether the weavng is static or dynamic. In automatic weaver implementations, the weaving process is static. Aspects reference the classes of those objects whose behavioral additions describe, and define at which join points additions should be made. Static weaving means to modify the source code of a class by inserting aspect-specific statements at join points. In other words, aspect code is inlined into classes. The result is highly optimized woven code whose execution speed is comparable to that of code written without AOSA. The aspect weaver delivered with AspectJ is an example for a static weaver. However, static weaving makes it difficult to later identify aspect-specific statements in woven code. As a consequence, adapting or replacing aspects dynamically during run-time can be time consuming or not possible at all. An example where this would be beneficial is given by [Matthijs et al. 97] where a load balancing aspect could replace the load distribution strategy woven before with a better one depending on the current load of managed servers. Currently, automatic weaving technology cannot support aspects in a dynamic environment. Dynamic weaving facilitates incremental weaving and makes debugging easier. On the other hand, static weaving has advantages over performance [Böllert 99]. Ideally, an implementation should support both. Another issue is whether the weaving is static or dynamic. Example architectures that impose static weaving include D [Lopes97], AspectJ, D²-AL [Becker98], and IL [Berger et al. 98]. Architectures that make use of reflective technologies allow dynamic weaving. Examples include Luthier-MOP [Pryor and Bastán 99] and AOP/ST [Böllert99].

3.3 Core Transformation

Another issue is whether there is source-to-source code transformation, from separate constructs to an intermingled source code. Technologies that rely on automatic weavers produce code transformation. Examples include D, AspectJ, D²-AL and IL. Reflective technologies will typically not have code transformation.

3.4 Level of weaving

As weaving is considered a general mechanism through which one can achieve composition of concerns, a difference in the AOSA proposals is in the level of weaving of these implementations. The level of weaving defines the point up to which one manages to achieve separation of concerns in the software system. One level is pre-compile which is associated with two phases of compilation: one for the weaver to produce ‘woven’ (intermingled source) code, and another for the final compilation into an executable code. An example of a pre-compile weaving technology is AspectJ. Another level of weaving is at compile-time, where the intermingled code exists only at the binary level. An example of compile-level weaving architecture is the Aspect Moderator Framework [Constantinides et al. 99a, b] discussed in the next section. The level of weaving can also be run-time, as in the case of reflective architectures. As a result, the interpretation of Figure 2 will depend on the level on weaving in discussion.

3.5 Open versus Closed Implementations

There are tradeoffs between using a language and adopting an open language. A language is ready to program but it is limited to the facilities that it provides. Clearly, opening a language can
be considered a risky approach, as the semantics of the extension mechanisms should balance openness with protection and security.

Figure 2. Weaving aspects and components together.

4. The Design of an Aspect-Oriented Framework

Our work concentrates in Concurrent Object-Oriented Programming. In its simplest form, we view a concurrent (shared) object as being decomposed into a set of abstractions that form a cluster of cooperating objects: a functional behavior, synchronization, and scheduling. The behavior of a concurrent object can be reused or extended. We view synchronization and scheduling as aspects, and we focus on the relationships between these abstractions within the cluster. We can shift the responsibility of an automatic weaver to an object, the aspect moderator that would coordinate aspects and components together (Figure 3). In this framework, a proxy object controls access to the functionality class. Aspects are created using the factory method pattern. The proxy would use the aspect factory in order to create aspects, and it would also use the moderator object to evaluate the aspects for every method of the functionality class. Before invocation, the proxy would call the moderator to evaluate the aspect(s) associated with this invocation. We believe that this approach provides the flexibility to the programmer to retain the definition of aspects by using current programming languages. In this framework, the semantic interaction between the components and the aspects is cleanly defined. Part of the semantics is the order of activation of the aspects.

4.1. Framework Architecture

A sequential object is comprised of functionality control and shared data. Access to this shared data is controlled by synchronization and scheduling abstractions. Synchronization controls enable or disable method invocations for selection. The synchronization abstraction is composed of guards and post-actions. Every method is executed within a preActivation and postActivation phase, which are defined by the AspectModerator class. The preActivation will evaluate the precondition of the method’s corresponding synchronization aspect (methodSyncAspect). During the precondition phase, guards will validate the synchronization constraints of the invoked method, returning RESUME upon success. After successfully executing the precondition
phase, the `aspectModerator` will return a `RESUME` to the `FunctionalityProxy` that will activate the method in the sequential object (`Functionality`). The completion of the method execution will initiate a call by the `FunctionalityProxy` to the `AspectModerator`’s `postActivation` phase. During `postActivation`, there is a call to the `notification` method of `methodSyncAspect`. During `notification`, synchronization variables are updated upon method completion. The proxy-moderator object pair coordinate functional and aspectual behavior by handling their interdependencies. We stress the fact that the activation order of the aspects is the most important part in order to verify the semantics of the system. Synchronization has to be verified before scheduling. A possible reverse in the order of activation may violate the semantics. There are other issues that might also be involved. If authentication is introduced to a shared object for example, it must be handled before synchronization.

**Figure 3.** A concurrent object as a cluster of components and aspects within the Aspect Moderator Framework.

### 4.2 The Use of Assertions to Support Software Quality

A major component of quality in software is reliability: a system’s ability to perform its job according to the specification (correctness) and to handle abnormal situations (robustness) [Meyer]. The concept of “Design by Contract” was introduced in the context of the Eiffel programming language [Meyer92]. Under this theory, a software system is viewed as a set of communicating components whose interaction is based on precisely defined specifications of the mutual obligations known as contracts. These contracts govern the interaction of the element with the rest of the world. The Aspect Moderator Framework adopts this approach in a slightly different context: defining assertions (preconditions and postconditions) as a set of design principles. Another important issue is the one of the verification of components and aspects in isolation from each other. One must be able to test the functionality of a component as well as being able to test that an aspect will align nicely with the functional component. Otherwise, there can be no guarantee that
components and aspects will cooperate. In other words, one must test and verify the collaboration of components and aspects. This would constitute an important phase in the design process.

### 4.3 Adaptability

Adaptability is an important quality factor in software systems and the issue of it being explicitly engineered into a system is stressed in [Fayad and Cline 96]. Incremental adaptability means coping with changing requirements without modifying previously defined software components. The conventional object-oriented model supports adaptability through composition, encapsulation, message passing and inheritance mechanisms. In general, lack of support of dynamic adaptability might lead to re-engineering the whole software system. In [Sanchez et al. 98] it is argued that concurrent OO languages do not provide enough support for the development of true adaptable software either because aspects are mixed in the functional components, or because once components are woven the resulting piece of software is too rigid to be adapted or reconfigured at run-time.

The general architecture of the framework allows reusability and ensures adaptability of components [Fayad et al. 99a, 99b, Fayad and Johnson 00] and aspects as both are designed relatively separately from each other. This framework hooks components and aspects together, defining their semantic interaction. The use of design patterns in order to provide axes of adaptability is suggested in [Fayad and Cline 96]. One advantage is that the aspect moderator class is extensible in order to make the overall system adaptable to addition of new aspects. If a new aspect of concern would have to be added to the system, we do not need to modify the moderator class. For static adaptability we can simply create a new class to inherit and re-define it, and reuse it for a new behavior. The inherited class can handle all previous aspects, together with the newly added aspect. Adaptability is also applied to components. In this framework, the moderator object has the capability to activate or drop aspects on the fly. For dynamic adaptability, we can create aspects and register them with the AspectModerator object.

The Aspect Moderator Framework does not require some new syntactic structure for the representation of new aspects, but simply a new class for the new aspect. This technique makes it easy for an existing aspect to be removed from the overall system. Further, the semantic interaction between components and aspects in the framework is defined by a set of principles. Part of this semantic interaction is the order of activation of the aspects thus providing a criterion for aspect ordering. The order of execution can also be altered on the fly. This concept is not feasible with automatic weaver technologies. In this framework, components and aspects are designed relatively separately and they remain separate entities that may access each other freely without code transformation. In fact, functional components do not need to know about the aspect components in advance (before run-time) but only after an aspect has been created and registered by the moderator class. As a result, components and aspects discover each other at run-time if necessary. The interaction of newly added aspects with the rest of the system is handled in a similar manner as the implementor must specify the contract that binds a new aspect to the rest of the system rather than having to re-engineer the whole system. On the other hand, automatic weavers must rely on language constructs that are hard coded into aspect code to provide the contact (join) points.

In [Matthijs et al. 97] the authors stressed the importance of aspect manifestation in every stage of development. The issue that in some cases aspects should remain run-time entities was also discussed in [Kenens et al. 98], and [Böllert99] where it is argued that with static weaving it might be impossible to adapting or replacing aspects dynamically. The framework manages to achieve the manifestation of aspects at run-time. We argue that it is important for a framework to provide
for dynamic aspect evolution in order to achieve maximum flexibility. Ideally, a framework would support both static and dynamic behaviors [Fayad and Schmidt 97]. As an example where this would be necessary, an aspect such as scheduling or load balancing might need to adapt itself based on run-time information. On the other hand an aspect such as synchronization can be statically dealt with.

4.4 Composition of aspects

In the Adaptive Arena [Bader and Elrad 98b] the functional part of a system is separated from the synchronization code, but it still remains in the same class. The separation of functional and aspectual code in the Aspect Moderator Framework results in program code that is more modular. Furthermore, the framework follows a general-purpose approach in order to achieve composition of concerns. This way, it is not confined to certain aspects but can address a number of aspects as long as their interactions can be well defined through a set of preconditions and postconditions. It is also language neutral.

5. Comparisons with current technologies

The framework puts the system under one compilation phase where an executable code is produced. Intermingled code exists only at the binary (executable) level. On the other hand, a technology such as AspectJ requires two phases of compilation, one for the weaver to produce an intermingled source code and another for the final compilation into an executable code. The level of weaving defines the point up to which one manages to achieve separation of concerns in the software system.

Both automatic weaver and the aspect moderator approaches provide the elegance of the original clean code during the analysis and design of the system.

The very reason that guided research towards AOP has resulted in the avoidance of the problem of inheritance anomaly. With the Aspect Moderator Framework we manage to avoid the problem of inheritance anomaly since components and aspects are pure objects and can be therefore reused. This problem is solved in automatic weaver technologies as well.

The concurrency facilities of the Java language provide a good choice to for the framework implementation. Since AOP is not restricted to programming only (and thus it is not restricted to one paradigm or a particular language) the framework manages to remain language neutral and work is under way to identify other candidate languages. Particularly beneficial is the ability to express components and aspects in the same language as large-scale software systems are built based on COTS technology rather than domain specific languages.

A comparison between this framework and AspectJ is essentially a demonstration of the tradeoffs between a language and a framework. A language is ready to program but it is limited to the facilities that it provides. This framework can be viewed as an open implementation since the moderator provides a mechanism to support an open language. On one hand, a language implementor can always hard code a set of constructs to support a number of pre-defined aspects. Perhaps it would be impossible to predict all possible aspects that might come up and it would thus be impossible to predict their syntax and semantics. A language implementor would need to have the syntax in advance. On the other hand the framework provides a general aspectual capability to the system which is language independent. The Aspect Moderator Framework is an architecture that allows for an open language where new aspects (specifications) can be added and their semantics can be delivered to the compiler through the moderator. In essence the moderator is a program
that extends the language itself. Our approach has a good chance to reduce possible inconsistencies, although it cannot guarantee correctness.

6. Conclusion

We believe that AOP should be considered a discipline for general programming and should not confine itself in one application or a range of applications. It should not confine itself in a domain-specific language either. In this position paper we presented some preliminary work on Aspect Oriented Framework for concurrent object-oriented systems. The overall behavior is made up of a functional behavior, concurrency aspects and a moderator class that coordinates the interaction between components and aspects while observing the overall semantics. Our approach partitions a system into a collection of cooperating classes in order to promote code reusability and make it easier to validate the design and correctness of these systems. This framework can provide for an adaptable model with ease of modification. The framework approach is promising, as it seems to be able to address a large number of aspects (and applications) as long as the relationships of components and aspects (as well as the aspect interrelationships) are clearly defined. A clear definition of these inter-relationships is achieved through the use of pre-conditions and post-conditions. In general we argue that a framework has a longer life span than a language (one that is not constantly extended). Further, we believe that a language with a large set of constructs is generally undesired. A framework can therefore be viewed as providing a mechanism to address future needs with the minimum cost (in regards to time, financial and complexity cost). Clearly, opening a language can be considered a risky approach, as the semantics of the extension mechanisms should balance openness with protection and security. In our framework the introduction of a new specification (aspect) must be accompanied by a set of rules that will ensure the integrity of the semantics of the system. These rules are expressed as pre-conditions, postconditions, and the order of activation of aspects.

7. References


