Composition of Concerns

Constantinos A. Constantinides, Atef Bader, Tzilla Elrad, Paniti Netinant
Concurrent Programming Research Center
Department of Computer Science
Illinois Institute of Technology
10 W. 31 St. Chicago IL 60616, U.S.A
+1 312 567.5150 (phone) +1 312 567.5067 (fax)
{conscon, elrad, netipan} @charlie.cns.iit.edu, bader@delta.csam.iit.edu
www.iit.edu/~concur

Abstract

As software systems become larger, the interaction of their components becomes more complex. This interaction may limit reuse, and make it difficult to validate the design and correctness of the system. As a result, re-engineering of these systems might be inevitable to meet future requirements. There is a general feeling that OOP promotes reuse and expandability by its very nature. This is a misconception as none of these issues is enforced. Rather, a software system must be specifically designed for reuse and expandability. In this paper we describe an aspect-oriented framework where both functional components and aspects are designed relatively separately from each other. This separation of concerns allows for reusability and enables the building of software systems that are stable and adaptable. Our work concentrates on the decomposition of concurrent object-oriented systems and our goal is to achieve an improved separation of concerns in both design, and implementation.

Keywords: Aspect-Oriented Programming, Reusability, Concurrent Programming, Open Implementation.

1. Introduction

There are numerous benefits from having an important concern of a software system being expressed well localized in a single code section. We can more easily understand it, analyze it, modify it, extend it, debug it, reuse it etc. The need for dealing with one important issue at a time was coined as the principle of separation of concerns [Parnas72]. To date, the primary idea for organizing software systems has been based on software decomposition, where a problem is broken down into subproblems that can be addressed relatively independently. Software decomposition and programming languages have been mutually supportive. Current languages and paradigms support a number of modular representations such as procedures, and objects. They further support composition of modules into whole systems. At the same time many systems have properties that do not necessarily align with the functional components and cannot be localized to modular units. Example properties include performance optimizations, failure handling, synchronization, coordination and scheduling policies. Aspects are defined as properties that cut across groups of functional components. While these aspects can be thought about and analyzed relatively separately from the basic functionality, at the implementation level they must be combined together. Programming them manually into the system’s functionality using current component-oriented languages results in aspects being spread throughout the code. This code tangling makes the source code difficult to develop, understand, and evolve by destroying modularity and reducing software quality [Lopes97, Kiczales et al. 97]. The major goals of Object-Oriented Programming (OOP) are abstraction, modularity, and code reuse. On the other hand, OOP provides only one dimension along which concerns can be separated. This was coined the “tyranny of dominant decomposition” by [Ossher and Tarr 99]. Although not bound to OOP, Aspect-Oriented Programming (AOP) is a paradigm proposal that 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. In fact, [Luna97, Tekinerdogan and Aksit 98] encourage aspectual decomposition from the very beginning. Current AOP approaches view components and aspects as two separate entities where the aspects are automatically weaved into the functional behavior of the system in order to produce the overall system. In [Mens et al. 97] it was argued that generalized procedure languages do not provide the right abstraction for the description of aspects. The importance of having appropriate languages for the expression of aspects was also addressed 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. On the other hand [VanHilst97] views weaving as a more general process that corresponds to component composition rather than merging. In [Ossher and Tarr 99] the authors argue that AOP does not go far enough with respect to tackling the tyranny of
dominant decomposition as AOP permits only one decompo-
sition. Instead, the concept of multi-dimensional separation 
of concerns is proposed.

2. Statement of the problems

One difference in the proposals for supporting AOP resides in 
the way in which aspects are woven across the functional 
components of the system: One issue is whether the weaving is 
static or dynamic. Example architectures that impose static 
weaving include D [Lopes97], AspectJ [Lopes and Kiczales 
98], D3-AL [Becker98], and IL [Berger et al. 98]. Other archi-
tectures that make use of reflective technologies allow dynamic 
weaving. Examples include Luthier-MOP [Pryor and Bastán 
99] and AOP/ST [Böllert99]. Another issue is whether there is 
code transformation. Technologies that rely on automatic 
weavers produce code transformation. Examples include D, 
AspectJ, D3-AL and IL. Reflective technologies will typically 
not have code transformation. Additionally there are proposals 
of specific languages for the support and implementation of 
AOP versus extensions to general-purpose languages. Exa-
amples of the former include COOL, RIDL [Lopes97], D3-AL, IL 
and TyRuBa [DeVolder98]. Examples of the latter include As-
pectJ, Replication-Framework [Fabry98], JST [Seinturier99], 
Luthier-MOP and Kava [Welch and Stroud 99]. Further, there 
are differences in the level of abstraction of these implementa-
tions. Our position is that automatic weaver implementations 
and aspect languages impose a number of restrictions which we 
discuss in this section.

2.1 Increased complexity of a general-purpose aspect language. 
Where a general-purpose aspect language is implemented, the 
introduction of new types of aspects will require the language 
to be extended in order to provide new constructs for their rep-
resentations. As the number of these aspects increases, the 
complexity of (both implementation and use of) the language 
would also increase.

2.2 Expressiveness of general-purpose aspect language. Can a 
general-purpose aspect language really be general purpose? We 
yet have to see real examples of expressing different kinds of 
aspects. On the other hand, the language designers must have a 
graham specification in advance. As it is impossible to predict 
the syntax of possible future aspects we argue that an aspect 
language can really be general purpose if it is constantly ex-
panded.

2.3 Restrictions in specific aspect language implementations. 
[Beugnard99] argues that AspectJ imposes a restriction by en-
forcing the explicit reference to the code in the aspect, making 
it reusable only for that purpose. The author proposes a separa-
tion of the aspect description in two parts: the semantics of the 
aspect itself, and the join points.

2.4 Restrictions imposed by static weaving, and lack of support 
of dynamic weaving. In a number of current implementations, 
the weaving process is static. Aspects reference the classes of 
those objects whose behavioral additions describe, and define 
the points at which 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 a highly optimized woven 
code whose execution speed is comparable to that of code 
written without AOP. However, static weaving makes it diffi-
cult to later identify aspect-specific statements in woven code. 
As a consequence, adapting or replacing aspects dynamically 
can be time consuming or not possible at all. An example 
where dynamic weaving would be beneficial is given by [Mat-
thijs et al. 97] where a load balancing aspect could replace the 
load distribution strategy woven before with a better one de-
pending on the current load of managed servers. Another ex-
ample is given by [Böllert98b] where a particular malfunction 
in a software system should deploy a tracing aspect to be 
 woven and run without having to restart the software system. 
Currently, automatic weaving is not a viable technology for 
implementing aspects in a dynamic environment. Static weav-
ing has advantages over performance [Böllert99] whereas dy-
namic weaving facilitates incremental weaving and makes de-
bugging easier. Ideally, an implementation should support both 
static and dynamic weaving. A feasible approach to handle 
dynamic weaving is to implement aspects using meta-objects. 
Proposals that address dynamic weaving through the use of 
meta-programming are [Lunau98], AOP/ST and Luthier-MOP.

2.5 Increased complexity of an automatic weaver. The weaver 
confines itself to the nature of the aspects and the constructs 
provided by the aspect language(s) it implements. The weaver 
must contain a specific interpreter for each aspect description 
language. If a new type of aspect of concern were to be added 
to the model, the weaver would have to be modified (extended) 
to adapt to the new model. As different aspect description lan-
guages address different types of aspects, an increase in the 
number of aspect languages would result in an increase in the 
complexity of the weaver.

2.6 Aspect inter-relationships. An open problem remains the 
issue of aspect-aspect interaction. We can break this down into 
two areas: 1) relationship of orthogonal aspects and 2) the ex-
istence of non-orthogonal aspects. The relationship of orthogo-
 nal aspects relies on their order of activation and by their vali-
dation and verification. Current aspect-oriented architectures 
have addressed models where aspects are orthogonal and there-
fore present a flat structure. There are cases where aspects are 
not orthogonal, but they can cut across each other. In these 
cases the overall structure is not flat but rather hierarchical. The 
issue of non-orthogonal aspects still remains an open problem, 
which we believe can be attacked in either of two ways: 1) by 
language design and 2) by implementation.

2.7 Debugging and level of weaving. In several implementa-
tions that use automatic weavers, weaving is done before com-
pile time. Of great interest is the issue of debugging an aspect 
program. In proposals such as AspectJ, the only way of think-
ing about and therefore debugging a program is to examine the 
 woven code. We argue that this should not be the case. Today 
a programmer debugs at the level of Java [Gosling et al. 96], and
C++ [Stroustrup86] and not assembly code. The implementor should not be constrained by the requirement that there exist woven code that is readable by the programmer.

2.8 Lack of criteria of aspect ordering. How does an automatic weaver cope with multiple aspects that must be woven into the same class, or the same method? Does the ordering have to be done manually, or should the weaver sort aspects according to some criteria?

3. The proposed aspect moderator 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 1). The aspect moderator is a framework that uses patterns for software stability. A proxy object controls access to the functionality class. The proxy uses the aspect factory in order to create aspects (figure 2), and it also uses the moderator object to evaluate the aspects for every method of the functionality class. Before invocation, the proxy calls the moderator to evaluate its associated aspects. The aspect moderator class is extensible in order to make the overall system adaptable to addition of new aspects. We also believe that this approach provides the flexibility to the programmer to retain the definition of aspects by current programming languages. It also provides the basis for a design framework that would make use of patterns whose importance within the AO technology was addressed in [Lorenz98]. The aspect moderator class defines the semantic interaction between the components and the aspects. Further, the semantics of the model define the order of activation of the aspects.

4. Architecture of the framework

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. During the Precondition phase, guards will validate the synchronization conditions. In the Notification phase, post-actions will update the synchronization variables. The scheduling abstraction allows the specification of scheduling restrictions and terminate-actions. At the Precondition phase, scheduling restrictions use scheduling counters to form the scheduling condition for each method. At the Notification phase, terminate actions update the scheduling counters. During the Precondition phase, the synchronization constraints of the invoked method are evaluated. If the current synchronization condition evaluates to RESUME the scheduling constraints are then evaluated. After executing the Precondition phase, the moderator will activate the method in the sequential object. During Notification, synchronization variables and scheduling counters are updated upon method completion. The aspect moderator object coordinates 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 1. A concurrent object as a cluster of components and aspects within the aspect moderator framework.](image-url)

4.1 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] introduces the concept of “design by contract” 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 importance of assertions is also stressed in [Jézéquel and Meyer 97] where it is described how the absence of specifications caused the disaster associated with the European Ariane 5 launcher. The aspect moderator framework adopts this approach in a different context: defining assertions (preconditions and postconditions) as a set of design principles. Meyer argues that assertion monitoring yields to a productive approach to debugging, testing and quality assurance, in which the search for errors is not blind but based on consistency conditions provided by the developers themselves. As a result, reliability should be a built-in component in software development, not an afterthought. None of Java, Ada [DoD80] or CORBA [OMG98] has any built-in support for design by contract. In [Jézéquel and Meyer 97] the authors ar-
gue that without specification it is probably safer to redo rather than to reuse. 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 co-operate. In other words, one must test and verify the collaboration of components and aspects. This would constitute an important phase in the design process.

4.2 Adaptability

There is a general feeling that OOP promotes reuse and expandability by its very nature. We argue that this is a misconception as none of these issues is enforced. Rather, a software system must be specifically designed for reuse and expandability. 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 re-configured at run-time.

The general architecture of the framework allows reusability and ensures adaptability of components and aspects as both are designed relatively separately from each other. The aspect moderator is a design pattern that 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 of the advantages of the aspect moderator framework is that if a new aspect of concern would have to be added to the system, we do not need to modify the moderator class. 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. 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. In this framework, the moderator object has the capability to activate or drop aspects on the fly. 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]. [Böllert98a] also stressed this issue by arguing that much like conditional compilation, aspects must be woven to the program on-demand. In technologies that rely on automatic weaving, aspects manifest in the model and in the program code, but neither in object code (byte code in the case of Java) nor in executable (binary) code. [Böllert99] argues 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 is important that in order to achieve maximum flexibility a framework must provide for dynamic aspect evolution and ideally support both static and dynamic behavior. As an example, 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.

4.3 Composition of aspects

In ESP [Dempsey and Cahill 97] and 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. It is also language neutral. With the exception of AspectJ, current technologies are confined in domain specific languages. We introduce the concept of an aspect bank, where the moderator of a cluster initially needs to collect and register all the required aspects from. The aspect bank provides a hierarchical two-dimensional composition of the system in terms of aspects and components.

5. Comparisons with current technologies

The aspect moderator 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, technologies such as AspectJ require two
phases of compilation, one for the weaver to produce an inter-
mingle 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.

A comparison between this framework and AspectJ is essen-
tially 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 lan-
guage implementor can always hard code a set of constructs to 
support a number of predefined aspects. Perhaps it would be 
possible to predict all possible aspects that might come up 
and it would thus be impossible to predict their syntax and se-
manitics. A language implementor would need to have the syn-
tax in advance. On the other hand the framework provides a 
general aspectual capability to the system which is independent 
of a language. The aspect moderator is an architecture that al-

dows for an open language where new aspects (specifications) 
can be added and their semantics can be delivered to the com-
piler through the moderator. In essence the moderator is a pro-
gram that extends the language itself. This 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 ap-
lication or a range of applications. It should not confine itself 
in a domain-specific language either. As software systems be-
come larger, the interaction of their components becomes more 
complex. This interaction may limit reuse, and make it difficult 
to validate the design and correctness of the system. As a re-

sult, re-engineering of these systems might be inevitable to 
meet future requirements. There is a general feeling that OOP 
promotes reuse and expandability by its very nature. This is a 
 misconception as none of these issues is enforced. Rather, a 
software system must be specifically designed for reuse and 
expandability. In this paper we describe an aspect-oriented 
framework where both functional components and aspects are 
designed relatively separately from each other. This separation 
of concerns allows for reusability. Our work concentrates on 
the decomposition of concurrent object-oriented systems and 
our goal is to achieve an improved separation of concerns in 
both design, and implementation. Our approach partitions a 
system into a collection of cooperating classes in order to pro-
mote code reusability and make it easier to validate the design 
and correctness of these systems. We achieve composition of 
concerns through a class that coordinates the interaction of 
components and aspects while preserving the semantics of the 
overall system. We argue that is important that in order to 
achieve maximum flexibility a framework must provide for 
dynamic aspect evolution and ideally support both static and 
dynamic behavior. Based on run-time information, an aspect 
such as scheduling or load balancing might need to adapt itself. 
Our design framework provides an adaptable model 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. The interaction of newly added aspects 
is specified by a contract that binds a new aspect to the rest of 
the system rather than having to re-engineer the whole system. 
In essence the aspect moderator is a program that extends the 
language itself. The framework approach is promising, as it 
seems to be able to address a large number of aspects (and ap-
lications) as long as the inter-relationships of components and 
aspects (as well as the aspect-to-aspect relationships) are 
cleanly defined. The framework approach is promising, as it 
seems to be able to address a large number of aspects (and ap-
lications) as long as the inter-relationships of components and 
aspects (as well as the aspect interrelationships) are cleanly 
defined. A clean definition of these interrelationships is 
achieved through the use of preconditions and postconditions.
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 large language 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 risky, 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 preconditions, postconditions, and order of activation of aspects. The most notable advantages of this framework is the decoupling of functional and aspect components, the reduction of development cost, the ease of change of the overall behavior, and a complete control over the interaction behavior. A drawback of our proposal is that there is an increase in the number of classes.

7. References


