# De-constructing and Re-constructing Aspect-Orientation

William Harrison
Department of Computer Science
Trinity College
Dublin 2, Ireland*
(+353) 1-896 8556

Bill.Harrison@cs.tcd.ie

## ABSTRACT

Through its decade-and-a-half long evolution, the aspect-oriented software community has occasionally struggled with its identity – revisiting the question "What kinds of technologies make up aspect-oriented software and who should be interested in it?" We attempt to de-construct "aspect-oriented" into several issues making up its foundation, believing that the community is inclusive and that work exploring or exploiting any of these concepts fits within the community. Although their historical setting contributes somewhat to the understanding of why different authors have emphasized one or more of these issues, we analyze them from an intrinsic point-of-view, to highlight broader or deeper issues that may lie behind the constructs currently made available, in the hope that "aspect-oriented" software technologies can be extended to provide an even stronger basis for software than they do today.

## Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: *Data abstraction, Information hiding, Languages*

D.3.3 [**Language Constructs and Features**]: *Modules, Packages, Concurrent programming structures*

## General Terms

Languages, Design

## Keywords

Aspect-oriented, separation-of-concerns, encapsulation, software-composition, event-flow, broadcast, obliviousness, complex-event-processing, specification, modularity, malleability.



Seventh International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2008), April 1, 2008, Brussels, Belgium.



## 1. INTRODUCTION

Much discussion has taken place about Aspect-oriented Programming (AOP) and Aspect-oriented Software Development (AOSD), both in characterization of the field as a whole and in attempts to use that characterization to evaluate its value. The adjective "aspect-oriented" came into its current use[1] to qualify "programming". As such it characterized a research view in which different "aspects" of a program, like distribution or storage layout, were addressed by different languages and language processors, with results "woven" together by an aspect weaver [18]. When a single-language focus was adopted that resulted in AspectJ, the term "aspect" was also refocused, and defined, for patent purposes, as an "aspect ... comprising: a cross-cut ... and a cross-cut action" [17]. The emphasis here was on the fact that unlike earlier work, both crosscut specification and consequent action must occur together in the aspect. Several other groups of researchers and developers were pursuing related approaches to the expression of software in which different concerns are expressed in separate artefacts linked by exchange of messages [10] [1]. These groups of researchers coalesced to form the growing community working on AOSD. In that expanded context, the characterization of AOP as "quantification and obliviousness" [7] which was correct for AspectJ is too limited for AOSD. The subtlety of the distinction between AOP and AOSD has led to an inclination by the wider software community to find aspects inappropriate or too limiting, and even to find AOSD's success to be "paradoxical" [27], so it is important to emphasize that AOSD researcher has always addressed a wider set of issues.

In fact, the wider set of issues of interest to the AOSD community is becoming ever more relevant to problems emerging in future computing environments. The computing world has undergone considerable change since the advent of object-oriented technology. From its early development, Aspect-Oriented technologies [6] have generally avoided the narrow view of the common object model in which a

1 It had been previously used in [25] for a role-like concept.

* This work is supported by a grant from Science Foundation, Ireland

message is sent to and handled by an isolated "target" object. This evasion is prominent in many emerging technologies as well. Service-Oriented Computing [32], Grid Computing [31], Ubiquitous Computing [33] and Complex-Event Processing [21] all provide a view of software in which the infrastructure re-routes messages or method calls from the apparent target to one or more real target objects. Various proxies are employed to filter or redirect messages. In some cases even, the point-to-point communication model for sending messages to objects is being replaced by a broadcast model in which messages are sent for delivery to whichever objects hold their implementation.

## 2. UNDERLYING ISSUES

Much work on aspect-oriented software deals with at least one of four issues. In roughly the order in which they emerged, these issues are:

- [aspects] representation of concern-separated artefacts and control of their interconnection or composition at points where they must join,
- [pointcuts] identification and exposure of appropriate points in concerns at which their behaviour should be joined,
- [context] characterization of complex behaviour and exploitation of inferred program state, and
- [mining] identification and possible extraction of concerns from tangled software.

Let us look in some more detail at the characteristics of the solutions proposed.

### 2.1 Aspects and Composition

Many AOSD approaches allow the state and behaviour of objects to be separated into elements that can be re-grouped into "subjects" [10] or "aspects" [16] to address separate issues of concern to the architecture or implementation. The various elements of objects in the separated concerns are put back together by a process of composition or weaving.

Several reasons have been put forward to motivate the importance of aspects:

- attachment of systemic[1] behaviour to objects, to support transactions, persistence, etc. [1][24]
- enhanced development characteristics such as simple extension and concurrent development of functional concerns like editing, verification, or display in an IDE [10]
- support for late selection and combination of functional features of product-lines [15]
- support for large-scale middleware construction [5]

In its broadest form, software aspects are generally modelled as a body of material which, when executed, produces events or cooperative method calls. In object-oriented formulations, aspects may be classes or may associate fields and methods with one or more classes. Aspects need not form complete programs, although some aspect-oriented approaches require completeness of some aspects.

Composition, or weaving, can be thought of as a form of dispatch, but one which employs more complex delivery rules than does the common target-directed model for objects, and one in which the message may result in behaviour that was defined in several aspects. While Subject-oriented Programming and Aspect-oriented Programming treat events in terms of a broadcast and focus on the end-points of message delivery, other work attempts to deal with the handling of messages in a way that allows each object along the path to influence the delivery targets [20] [11].

### 2.2 Pointcuts and Join-Points

The second prominent issue among AOSD researchers is the specification of points in an aspect at which the events or cooperative method calls originate. The call's points of origin form a set of "join-points" – the points at which aspects join. They generally provide the means by which the various aspects of an object's behaviour are woven together. In Subject-oriented Programming, the join-points were specifically identified to be method calls because they represent the points at which a software developer using subject technology might "expect" unknown or additional behaviour to take place [23]. At method-call points, subclasses may provide overridden or extended behaviour, or target objects might have extended behaviour. This restricted view avoided the breaking of encapsulation, but made development of subjects require the same careful thought about extensibility as does the development of frameworks. AspectJ [16] introduced the concept of a pointcut – a query-specified cut through a program that identifies a set of join-points. The pointcut query is associated with an advice, identifying which of the join-points originate messages to be delivered to the advice body. In AspectJ, the advices and their pointcuts lie outside the base aspect to which the queries are applied, in-effect "injecting" additional behaviour into the base. Because these join-points are injected without participation of the original developer, they have been criticised as breaking the encapsulation. But they also greatly increase the degree of "obliviousness" [7] and provide for a much wider spectrum of points at which aspects may join.

The underlying concept behind "pointcut" as a set of join points has evolved and variations are in use by authors of a number of approaches for AOSD:

[1] Often called non-functional, although most customers would prefer to reject software that does not function or provides no function.

- **Cooperative operation** [10], behaviours in aspects are attached to the set of execution points that are method calls intended to allow for extended behaviour
- **Pointcut** [16], a query identifying specified a set of points in execution, used as a clause in an aspect's advice declaration. Viewed as a query, the pointcut's variables are bound when matched to an instance. These bindings can be passed as parameters to be advice body. A **user-defined pointcut** is given a name and signature. The signature selects from the variables bound by the query. An **abstract pointcut** is a user-defined pointcut that omits the query. It may be used as a clause in an aspect's advice but must eventually be made concrete by associating it with a particular query that provides bindings for its parameters
- **Exported pointcut** [9], is a user-defined pointcut that is specified in a module of the base rather than in an attached aspect. This effectively produces a cooperative operation, but expressing it as a query over the module's content makes it useful for after-the-fact use in the module.
- **Methoid** [12] explicitly treats an exported pointcut as a call to a method. To do so, the query specifies a set of regions in the execution of material to which it is applied. The content of the region is treated as a method that can be reduced to a single join-point - a cooperative operation call to that method. The method can be extracted or materialized as to perform the behaviour identified by the query if needed to form compositions.

### 2.3 Context: Gross Program State, CFlow, and Complex-Event Processing

AspectJ's pointcuts included the ability to filter the circumstances of a join, on the basis of dynamic information much like the filters of composition filters [1]. Among the filtering criteria is the "cflow" construct. An important use of cflow is in determining the gross program state of a base so that the aspect can respond in an appropriate manner. In the example in [2], the authors employ a series of correlated pointcuts using cflow to track processing within wizards.

But cflow is a weak mechanism for attacking an important problem. The gross program state of a system is often a more complex function of its flow history than examination of the current flow stack will reveal. So it is advantageous to use an aspect follow the series of occurrences in a system that indicate a change in gross state This is done in [28], where the authors observe that "interesting states of the system can be described in terms of previous events and the ordering of them." The aspect can summarize the state in a variable used in the pointcuts of other aspects. Such an aspect can be used to perform the kind of task usually associated with complex-event processing [21].

### 2.4 Aspect Mining: Analysis, Identification and Extraction of Concerns

As the value of concern-separated software became more evident and greater tool support for its use became available, the importance of being able to deal with the concern structure of legacy software grew. Program-slicing had been of interest for a long while, but was generally viewed as a compilation or debugging technique, perhaps for lack of ability to reflect the sliced program as a proper software artefact. A good review of work in this area can be had in [3].

## 3. OPPORTUNITIES

Much service-oriented software today treats services like objects, with composite services managed as intermediate objects that route calls to the objects they use internally. Exploiting aspect-oriented constructs allowing us to treat services as behaviour attached to cooperative operations offers the opportunity to introduce a much more flexible and malleable service structure.

Software builders conventionally work as if they have a complete view of the software they are building. This point-of-view is reinforced by the constructs we use in thinking about problem decomposition. Perhaps the deepest is the "call" construct. It is traditional for a developer to look at a specification for some service and develop an algorithm for satisfying it, examining and selecting from available software components to perform services that the algorithm itself needs in turn. Traditionally, this examination looks far deeper into the component than any formal specification. The developer may look at issues like:

- the performance characteristics, perhaps determined by examining the code if not the present in the documentation,
- the malleability or extendibility of the code, perhaps reflecting the need to create subclasses or attach aspects,
- the "burden" – additional options provided that contribute to overhead but are not needed,
- the stability of the code base, reliability of its developer, etc.

This analysis takes place even if the component is a built-in element of the programming language, like those that perform arithmetic on numbers, but the characteristics become ingrained to form part of a developers' natural repertoire. Having selected a component, the developer often adapts the algorithm under development to reflect additional choices and capabilities potential in its use by, for example exploiting public or private knowledge about the logic, state representation, or class structure of the component being used. The resulting dependencies are called EEK in [29].

This is, to some extent, changing. There is an impetus to treat components as "services" [22], contracted for when software is executed, rather than when it is written. While it may seem a small change, the impact is enormous because the trade-offs described above can no longer be made by the developer, but must be made later when the "service-finder" is operative. Fully exploiting these changes requires a change in the programming languages we use to make the artefacts more malleable and the information on which they depend more manifest to the service-finding mechanisms. The Continuum programming language [30] is being used as the basis for researching both the language and the implementation issues involved in this shift.

Most aspect-oriented approaches lie somewhere in-between these extremes of early- and late-bound selection. In asymmetric approaches, binding activities are performed by the developer of the attached aspect rather than the developer of the base aspect. This effectively reverses the usual situation by having the service (aspect) developer become familiar with the details of the client (base).

Symmetric treatments of aspects forgo assigning responsibility to either component developer, and require the developer of composition rules to be familiar with both (or all) participating services. As it is for a software "service-finder," the developer's task is made simpler, or indeed possible, if the needed information is made explicit in the software rather than having to be dug out from its latent places in the code.

The following parts of this section explore how the generalisation from aspects to services provides potential for change in the way we deal with some important issues.

### 3.1 Classic vs. Co-operative Method Call

Much discussion of aspect attachment could be clarified by explicit recognition that the named, parameterized pointcut effectively defines a cooperative method call and that the rest of the mechanisms for dealing with aspects can be applied in general. Except for the issue of where the pointcuts are specified and applied, there is often no natural difference between the structures of the concerns themselves. Implementations of functionality can as easily be placed in one concern as another, in a way that reflects requirements rather than a dominant decomposition like "class". Attaching it as an aspect can yield equivalent results as keeping it in the base. In fact, it has been observed [26] that even "class" is just another dimension for separating concerns. This effectively points out that whether the developer of a class decides to put it in the base or to put it in a separate concern makes little difference to the operation of the base. Of course, depending on the AOSD approach in use, it may affect the syntactic expression.

No matter where the pointcuts are specified that expose them, the join-points in a concern effectively become points of cooperative method call [10]. From a mechanistic point-of-view, a cooperative method call can be thought of as identical to a classic method call. Classically, we think of a method call as belonging to (defined by) a client or consumer, the one who makes the call. But the cooperative method-calls in a concern are the join-points it exposes. Ordinary method-call is concerned with what the call will do for the client. But cooperative method call is concerned with what the call can do for the community as well. Behaviour provided in the originating concern or in any other concern cooperates to provide the actual behaviour associated with the cooperative call. So, from the point-of-view of system-structure, dispatch resolution, intention, specification, etc, classic method call and cooperative method call have quite the opposite conceptions even though they are mechanically identical. Potential impact on several of these areas is addressed in the following sub-sections. A common way to look at their mechanical similarity is discussed in Section 3.2. The fact that this reversal can exploit more information about intent is discussed in Section 3.3. Section 3.4 explores potential for availing of greater concurrency in the software we write, and Sections 3.5 and 3.6 discuss language support that both increases dynamicity and malleability and enables the more concurrent style.

### 3.2 Event Flow and the Dual Role Of The Base

The role of the "base" is perhaps the most vexing issue in treating AOSD [19] [13]. Virtually all approaches connect the behaviour in the separated concerns with cooperative method calls in the base, specified implicitly or via pointcuts. The base acts both as a body of code making cooperative method calls to which aspect behaviour can be attached and as an aspect providing some of the behaviour for them. In order to accommodate the diverse collection of emerging technologies that can all benefit from aspect-oriented approaches, we should separate event behaviour attachment from the description of overall event flow. In this view, the base contains no code itself. It is a specification of a sea of events on which the aspects float, each associating its behaviour with some of the events. The differentiation of cooperative method calls from classic method calls becomes the role of the base. This view intends to accommodate either view of joinpoints – that they are intended or that they are injected, as explored further in Section 3.3

Separating the abstract model of the underlying flow of behaviour from the attachment of providers' behaviour can give some insight into the role of pointcuts and their relationship to the base. Specifically, we can construct the "base" from the collection of abstract pointcuts whose behaviour is provided by the aspects. In Figure 1, we show an aspect concern both exporting some pointcuts (cooperative method calls) and providing behaviour for others. The base could be specified separately, as part of an

overall system design, or could be derived from information in the aspects making up the system. In either case, it could be thought of as a simple list of events identified as abstract pointcuts, a constrained event specification, a work-flow diagram, or as hinted at in [28], in the form of the sequence diagrams forming the system's design.

```
aspect x;
export {
   pointcut p(int a):
             call(* X.foo(int a));
}
provide {pointcut n(int a, real b);}
class Y {when[2] p(int a) {...}}
```

Figure 1- a symmetric aspect

In addition to the specification of exported and provided pointcuts, the aspect contains behaviour that is attached to the events and is subject to the exports. For an aspect to be attached to a base, its specification of the cooperative calls required must be consonant with those of the base An aspect not in the base has no exported pointcuts. A base with no advice for other aspects has no "provides". While not exploiting this syntax, Continuum's "service" construct explicitly lists the cooperative behaviours provided and describes their dependencies on earlier flow.

### 3.3 Whose Is The Specification - Join Points By Intention or Injection

In a classic call, the client developer keeps in mind the services needed (e.g. "a hash table into which objects can be put") while finding a suitable implementation for the service. In many object-oriented languages this decision is consolidated by naming the pre-existing class or interface that is associated with the selected implementation. When providing a local implementation, the class or interface created by the developer may be sketchy, or it may be well documented and meet the expected standards for reusable software. But in all cases, the focus is on what the service does or on what the client needs.

In cooperative call, there is more that needs to be said: what the client is doing in a cooperative sense. A call to hash.put(…) may have been written because of the intention "put a book into the library records". It is this intention which is the link that ties the cooperating concerns together. When written with respect to a particular base, a pointcut specification needs to supplement the call, to fill-in just this information about intention.

We say that the purpose of pointcuts, whether injected or intentionally exported, is to distinguish cooperative calls from ones that are hidden from cooperative attachment. But distinguishing the cooperative calls does not suffice to provide the specification of their intention. If we expect the cooperator to be found by a mechanical service-finder rather than by the client developer, it is clear that additional documentation must be available. In fact, the entire issue of malleability of call structures becomes more critical for cooperative calls than for hidden ones. The same method, identified by its name and signature, may be used to support many different intentions. This argues that the intention and characterization information must be separately attached to the name. In the interest of service-finding, we can employ glossary or ontology references associated with methods and their parameters to supply information concerning:

- the actual intention expressed at the cooperative call
- the separate functional expectations of a call so that they might be realised by separate aspects composed later
- the functionality provided by other aspects available
- the roles of parameters of the call in an order-free manner to give greater flexibility in matching server to client

Not all join-points need be originally written as method-calls. Those which are not must to be injected or exported, as mentioned in Section 2.2, using a pointcut to turn them into cooperative method calls. The information about intention can be supplied at the point where the pointcut is defined.

The issue of control of the specification is closely related to a phenomenon that could be termed "function bundling". Function bundling reflects the fact that many methods perform a multiplicity of functions. For example, an analysis of the Unix "sort" command was conducted and it was found to be reasonably represented as the composition of 30 concerns [4]. Bundling of this sort is often signalled by the presence of option parameters or of optional parameters – reference parameters that may be null. The bundling reflects the developer's statement of the specification as a "maximum" for the component being developed. It often contains excessive functionality contributing to the bloat of its clients [8]. An aspect-oriented realization could encourage independent functions to be presented in separate aspects, combined later by the service-finder at run-time rather than by the developer at development time.

### 3.4 Raising Concurrency with Aspects

#### *3.4.1 Attaching Aspects to Events*

The fact that we are reaching fundamental limits in increasing the speed of sequential processors indicates a growing need to increase the parallelism and asynchrony that is potentially available even in the ordinary software we write. One way to go about this is to bring the use of asynchronous events more into the mainstream by simplifying programming language constructs to encourage

[2] "When" is used as an alternative to before, after, event, around, etc. denoting behaviour to be performed sometime between before and after but not needing to be wrapped around other behaviour.

their use. The metaphor of software as aspects floating on a sea of events may offer us an opportunity to do so because it emphasizes the attachment of behaviour to events rather than the construction of sequences of control to arrange for their execution. While it is important to preserve the flexible synchronous combinators, like before, after, with, around, etc., used for aspect behaviour, not all aspect behaviour needs to complete before the cooperative method call that calls for it can continue. For example, the behaviour provided by a logging aspect can often be attached as an event since the continuation of the base does not depend on its early completion. Event attachment is irrevocably concurrent. The originating client can have no expectation about the time of its execution, which may even be deferred until the client completes.

### *3.4.2 Sending Events and Passing Commitments*

Attachment of aspects as events is a helpful first step, but does little to encourage more concurrency within the base itself. A second step forward is to permit cooperative method calls to be sent as explicit events. All recipients run concurrently with the continuation of the base. The declaration of method one in Figure 2 suggests how a commitment for the eventual invocation of an event can be passed declaratively from one method to another. The "sends" clause in the declaration of method one indicates that it is committed to the ultimate sending of event two, either on its own or by passing the commitment forward. In Figure 2 the commitment might be passed to method three (assuming its unshown declaration has a similar "sends" clause).

### *3.4.3 Future Event Handling*

People describe problem solutions sequentially, although they can break off chunks described to be done concurrently. And they seldom think of subtasks as subject to long potential delays. The ability to send method calls asynchronously is not a new construct, and like asynchronous aspect attachment yields only a small improvement in the overall concurrency behaviour of software. Both require the software developer to break the train of sequential thought, and both require the high intellectual overhead of creating new classes, methods, etc. We need to provide developers with a construct that allows them to think sequentially about activities that can be deferred or executed concurrently. We can build on the concept of passing commitment to provide it. In Figure 2, imagine that the developer knows that after doing method four, some other tasks must be performed. Perhaps method four makes a bank transfer, and a receipt must then be presented. This is a sequential thought that would generally be represented by invoking method four and then performing the receipt processing, shown as "…". We want to allow the developer to express the sequential dependency without holding up the return from method one. (If we imagine method one is called inside a loop, then allowing it to return without waiting for method four to execute means that many executions of method four are started by the loop, and all can run concurrently.) But syntactically, we avoid interrupting the developer's train of expression, by allowing the receipt processing to be written as part of the call to method four using a commitment that it will eventually send message five.

```
void one(Object x,int y)
     sends two(Object m, real z) {
  send three(this,"hello");
  send four(this, 6)
  expect five(MyClass this, int a) {...}
}
```

Figure 2 – preserving sequence without synchrony

Eventually, the commitment is met and message five is sent. Its implementation is as specified in the "expect" clause, and the receipt is printed. Note that method five cannot access any of the local state of method one, which may be long gone. But it can use its parameters to access object state as usual. The mechanisms for doing this and the meaning of the "`MyClass this`" parameter declaration are part of with the service model of aspects employed in Continuum and with its model for dynamically extending knowledge about the methods supported by classes. These are described briefly in Sections 3.5 and 3.6.

### *3.4.4 Exceptions*

Any construct like "send", that decouples future execution but still provides for satisfaction of commitments, must address the problem of exceptions and failures. Since a sent message may commit the future invocation of another event, we must define what happens if it fails to do so. This can be addressed at two levels. On the static level, a method declared to send some event must do so on all execution paths. This can be checked at compile and load time. On the dynamic level, a logic error may still prevent the future "send" from taking place by causing an exception to be thrown. When an exception is thrown, potentially unsatisfied commitments to send an event are satisfied by sending the event in such a way as to trigger the exception immediately on entry to the called method.

## 3.5 Generalizing Aspects as Services

Ever since the programming language community adopted the target-directed method invocation model for the dominant languages, software developers have been compelled to escape from it by moving dispatch from the language to the middleware. We see this escape in the all the emerging technologies mentioned above: Aspect-Oriented, Service-Oriented, Grid, Ubiquitous, and Complex Event Processing. In the infrastructure for all of these, method calls are directed to objects other than the "target" presented by the client. Widening the concept of "aspect" provides the opportunity to address this need. The

programming language Continuum makes the escape explicit by integrating with the language and VM a dispatcher whose mechanisms are constrained to assure delivery and freedom from ambiguity but are otherwise explicitly unspecified [14]. This is a step beyond the flexibility introduced by aspects, most approaches to which still hold the definition of dispatch closely.

Continuum introduces the "service" construct into the language. Its role is to provide methods with access to the state of one or more the objects they are passed as parameters. This process is called decapsulation and can only be applied to parameters of the method's class – the same ones that govern dynamic dispatch to the method itself. Like advices, the methods provided by services are added to the behaviour performed when the cooperative method is invoked. Services have several other roles. They act as encapsulation boundaries, with explicit specification of the cooperative methods and events they support and require, They also draw boundaries that contain the propagation of ambiguity so that it remains within a service. This limits the scope of the checking that must be performed when a class is added to a service. Services each have an independent representation of objects' states, so that references to objects may be represented as other than opaque pointers without losing their ability to convey information assuring support for methods. Non-opaque references can be passed between services that are distributed around the network.

### 3.6 Overcoming the Drawbacks of Obliviousness

Much work has been done on the dynamic introduction of aspects, but if the base is intended to be "oblivious," it is hard to extend this work to use aspects as general dynamic service providers – service calls are obvious rather than oblivious in the client. Dynamic service provision suggests that the client knows about the methods and expects to call them even though the methods' implementations are not available when the client is started. The service model described above provides for the dynamic introduction of services. To complement it, Continuum's assurance model is also dynamic. A class does not specify or limit the interfaces that it supports, allowing services to add to growing knowledge about which methods are supported [14].

```
void meth(Store{put(Store,Item)} store1,
          Store store2);
Store{put(Store,Item),
      boolean inStock(Item,Store)} more;
more =
 ({boolean inStock(Item,Store)}) store1;
more = store2;
boolean t = inStock(item,store2);
```

Figure 3 – dynamic knowledge about classes

In Figure 3, the first assignment statement to "more" adds the knowledge that "inStock" is assured safe to call with ***any*** object in the class Store, allowing this knowledge to be transferred later to store2. Each service may provide methods and implementations for various classes, based on the state information the service possesses. Services can even provide methods that do not require explicit knowledge of state, but simply depend on access to that state provided by other services.

Clients do not know which service provides a method, nor even which class the method is "in". This is because continuum's assurance model is symmetric, which means that a service can provide a method dispatched on a parameter other than some designated target. The assurance that the method can be safely called is passed through a generalized concept of interface, shown in Figure 3, even though the implementation need not lie in the object to whose reference the interface is attached. Hence, even though the knowledge that inStock is assured is associated with "store2", the implementation may actually reside in "item". This helps make the client structure less dependent on the implementation structure because the client does not need to know in which objects methods are implemented.

## 4. CONCLUSIONS AND DIRECTIONS

Aspect-oriented software is broadly focused on the language and support for separating the design and implementation work required for differently-motivated concerns. In conventional object-oriented software, these are often tangled within a single class or method. While the language features of some aspect-oriented approaches emphasize its use for post-facto attachment of functionality obliviously, others employ it to achieve planned separation for flexibility, as within product lines, or to achieve concurrency and event-processing. We are entering a computing environment that is radically changing to address the needs of mobility and the challenges of physical limits on processor speed, and in which broadband services are encroaching on the traditional point-to-point structures. In such an environment, there is much room for growth and exploration in the use of aspect-like constructs to adapt software to changing environments. We have outlined some issues that need to be addressed and some approaches that may address them in the hope that the community can widen the scope of its thinking about the applicability of aspects and concern-separated software as we move forward.

## 5. REFERENCES


[1] Aksit, M., Bergmans, L., Vural, S., An object-oriented language-database integration model: The composition filters approach. In *Proc. ECOOP'92, Springer Verlag, LNCS 615*

[2] Bodkin, R., Furlong, J., Gathering Feedback on User Behaviour using AspectJ. in *AOSD 2006 - Industry Track Proceedings*, Chapman, M., Vasseur, A., Kniesel, G. (Eds.), Technical Report IAI-TR-2006-3,

ISSN 0944-8535, Computer Science Department III, University of Bonn, March 2006

[3] Breu, S., Moonen, L., Bruntink, M., and Krinke, J. (Eds.), *Proceedings First International Workshop Towards Evaluation of Aspect Mining*. July 4, 2006, Nantes, France, Delft University of Technology Software Engineering Research Group Technical Report TUD-SERG-2006-012

[4] Carver, L., *Building Real-World Applications with Aspect-Oriented Modules and Hyper/J*. Master's thesis, Univeristy of Califorina, San Diego, Department of Computer Science and Engineering, June 2002

[5] Colyer, A., and Clement, C., Large-Scale AOSD for Middleware. *Proceedings of the 3rd international Conference on Aspect-Oriented Software Development* (AOSD 2004), Lancaster, UK, March 22-26, 2004, ACM, New York (2004), pp. 56-65

[6] Elrad, T, Filman, R. E., Bader, A. Aspect-oriented programming: Introduction. *Communications of the ACM, Volume 44 Issue 10, October 2001*

[7] Filman, R.E. and Friedman, D.P, Aspect-Oriented Programming is Quantification and Obliviousness. *In Position paper for the Advanced Separation of Concerns Workshop at the Conference on Object-Oriented Programming Systems, Languages, and Applications,* Minneapolis, Minnesota, October 2000

[8] Garlan, D., Allen, R., Ockerbloom, J., Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, vol. 12, no. 6, Nov., 1995

[9] Gudmundson, S., and Kiczales, G., Addressing Practical Software Development Issues in AspectJ with a Pointcut Interface. In *Proc. ECOOP 2001 Workshop on Advanced Separation of Concerns*, July 2001.

[10] Harrison, W. and Ossher, H., Subject-Oriented Programming - A Critique of Pure Objects. In *Proceedings of 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications*, September 1993

[11] Harrison, W. and Ossher, H., *Structure-bound Messages*. IBM Research Report RC 15539, March, 1990.

[12] Harrison, W., Ossher, H., Tarr, P., General Composition of Software Artifacts. *Proceedings of Software Composition Workshop 2006*, March 2006, Springer-Verlag, LNCS 4089

[13] Harrison, W., Ossher, H., Tarr, P., *Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition*. Research Report RC22685, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, December, 2002

[14] Harrrison, W., Lievens, D., Walsh, T., *Using Recombinance to Improve Modularity*. Software Structures Group Report #104, Computer Science Department, Trinity College, Dublin, March, 2007

[15] Jansen, A., Smedinga, R., van Gurp, J. and Bosch, J., First class feature abstractions for product derivation. *IEE Proceedings on Software*, Volume: 151, Issue: 4,Aug. 2004

[16] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, Jeffrey Palm and William G. Griswold, An Overview of AspectJ. *Proc. 15th European Conference on Object-Oriented Programming*, 327-353 (2001).

[17] Kiczales, G., Lamping, J., Lopes, C., Hugunin, J., Hilsdale, E., Boyapati, C., *Aspect-oriented programming*. United States Patent 6,467,086, October 15, 2002

[18] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J., *Aspect-Oriented Programming*. In Proc. ECOOP'97 (Finland, June 1997) Springer-Verlag

[19] Lamping, J., The Role of the Base in Aspect Oriented Programming. In *Proceedings ECOOP Workshops* 1999: 289-291

[20] Lieberherr, K. J., *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996

[21] Luckham, D., *The Power of Events*. Addison Wesley Professional, May 2002, ISBN: 0201727897

[22] Mendonpa, N.C.; Silva, C.F., Aspectual services: unifying service- and aspect-oriented software development. *International Conference on Next Generation Web Services Practices*, Aug. 2005

[23] Ossher, H. and Tarr, P., Operation-level composition: A case in (join) point. In *ECOOP '98 Workshop Reader*, 406–409, July 1998. Springer Verlag. LNCS 1543

[24] Rashid, A., Chitchyan, R., *Persistence as an Aspect*. Proc. of International Conference on Aspect-Oriented Software Development (AOSD 2003), March 2003, Boston, MA

[25] Richardson, J. and Schwarz, P., Aspects: Extending Objects to Support Multiple, Independent Roles. *Proc. ACM-SIGMOD Conf.,* Denver, Colorado, May 1991

[26] Tarr, P., Ossher, H., Harrison, W., and Sutton, S. M., "N degrees of separation: Multi-dimensional separation of concerns." In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99),* 107–119, IEEE, May 1999

[27] Steimann, F., The paradoxical success of aspect-oriented programming, In *SIGPLAN Notices*, Vol. 41, No. 10. (October 2006), pp. 481-497.

[28] Walker, R. and Murphy, G., Joinpoints as Ordered Events: Towards Applying Implicit Context to Aspect Orientation. *Proc. ASOC Workshop at ICSE* 2001

[29] Walker, R. and Murphy, G.. Implicit context: Easing software evolution and reuse. In *8th International Symposium on the Foundations of Software Engineering*, San Diego, CA, USA, November 2000

[30] Continuum Draft Language Specification available from *https://www.cs.tcd.ie/research_groups/ssg*

[31] Grid Computing overview available at *http://en.wikipedia.org/wiki/Grid_computing*

[32] Service Oriented Architecture overview available at *http://en.wikipedia.org/wiki/Service-oriented_architecture*

[33] Ubiquitous Computing overview available at *http://en.wikipedia.org/wiki/Ubiquitous_computing*