Aspect-Oriented Software Development (AOSD) - An Introduction

Johan Brichau
johan.brichau@vub.ac.be
and
Theo D'Hondt
tjdhondt@vub.ac.be

Aspect-Oriented Software Development (AOSD) is a relatively new software development paradigm that complements and improves on many contemporary development paradigms. To this extent, AOSD provides unique and advanced program structuring and modularization techniques. The implementation of software applications using AOSD techniques results in a better implementation structure which has an impact on many important software qualities such as enhanced reusability and reduced complexity. In turn, these software qualities lead to an improved software development lifecycle and, hence, to better software.

This document introduces readers familiar with software development to the concepts of aspect-orientation. We present why aspect-orientation is needed in modern software development and what its contributions are to the improvement of software design and implementation structure. Without delving too much into particular AO technology details, we present the various concepts of AOSD. After reading this introduction, the reader will understand what AOSD is about, know its key concepts and vocabulary and can find directions to more elaborate descriptions of concrete AOSD technologies.

1. Introduction

The development of large and complex software applications is a challenging task. Apart from the enormous complexity of the software's desired functionality, software engineers are also faced with many other requirements that are specific to the software development lifecycle. Requirements such as reusability, robustness, performance, evolvability, etc. are requirements about the design and the implementation of the software itself, rather than about its functionality. Nevertheless, these non-functional requirements cannot be neglected because they contribute to the overall software quality, which is eventually perceived by the users of the software application. For example, a better evolvability will ensure that future maintenance tasks to the implementation can be carried out relatively easily and consequently also with less errors. Building software applications that adhere to all these functional and non-functional requirements is an ever more complex activity that requires appropriate programming languages and development paradigms to adequately address all these requirements throughout the entire software development lifecycle.

To cope with this ever-growing complexity of software development, computer science has experienced a continuous evolution of development paradigms and programming languages. In the early days, software was directly implemented in machine-level assembly languages, leading to highly complex implementations for even simple software applications. The introduction of the procedural and functional programming paradigms provided software engineers with abstraction mechanisms to improve the design and implementation structure of the software and reduce its overall complexity. An essential element of these paradigms is the ability to structure the software in separate but cooperating modules (e.g. procedures, functions, etc.). The intention is that each of these modules represents or implements a well-identified subpart of the software, which renders the individual modules better reusable and evolvable. Modern software development often takes place in the object-oriented programming paradigm that allows to further enhance the software's design and implementation structure through appropriate object-oriented modeling techniques and language features such as inheritance, delegation, encapsulation and polymorphism. Aspect-oriented programming languages and the entire aspect-orientation paradigm are a next step in this ever continuing evolution of programming languages and development paradigms to enhance software development and hence, improve overall software quality.

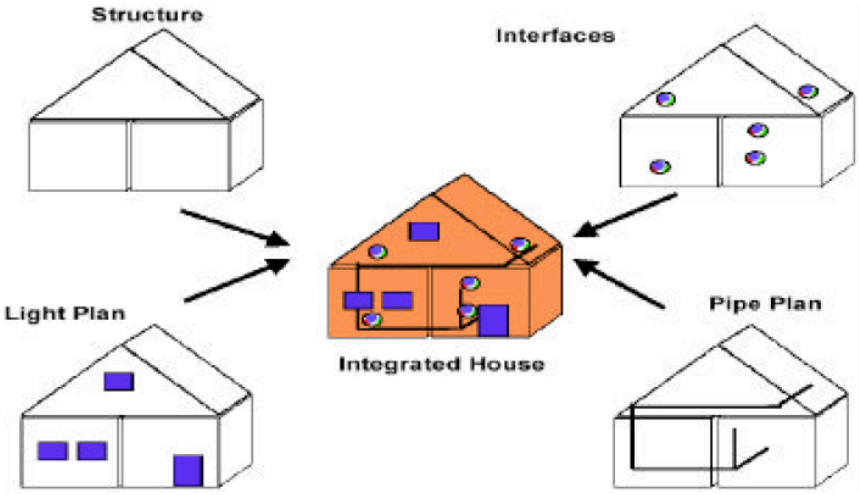
2. Separation of Concerns

The principle of separation of concerns promotes good design and reusable implementations. It is a driving factor in the continuous evolution of development paradigms because software engineers need appropriate development mechanisms to achieve it. Although the object-oriented paradigm already supports this principle quite well, aspect-orientation provides enhanced approaches and techniques which allow an even better separation of concerns in the entire development lifecycle. Let us first explain what concerns are and what their separation means to software development.

A concern is an interest which pertains to the system's development, its operation or any other matters that are critical or otherwise important to one of the stakeholders [2]. The term *separation of concerns* was originally coined by Dijkstra in [11]:

> *Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects.*
>
> *We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day […] But nothing is gained – on the contrary – by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns" […]*

The principle of separation of concerns essentially states that each concern that is relevant to the software application is best treated separately from the other concerns. This principle is not only valid in a software engineering context. To better understand what this principle is about, let us first consider it in the context of an architect's work in the design of a building.



When designing a building, architects do not make one single plan that describes the overall structure of the entire building. Instead, they use many different plans that each focus on a single part of the building: front and side views, floor plans, cross-sections, foundation, drainage system, electrical wiring, central heating, and so on. Each of these plans addresses a single concern of the building. They are separated because they are supposed to be used by different persons: clients, bricklayers, electricians, plumbers, and so on. Their separation makes them easier to understand and facilitates the modification of each single concern of the building.

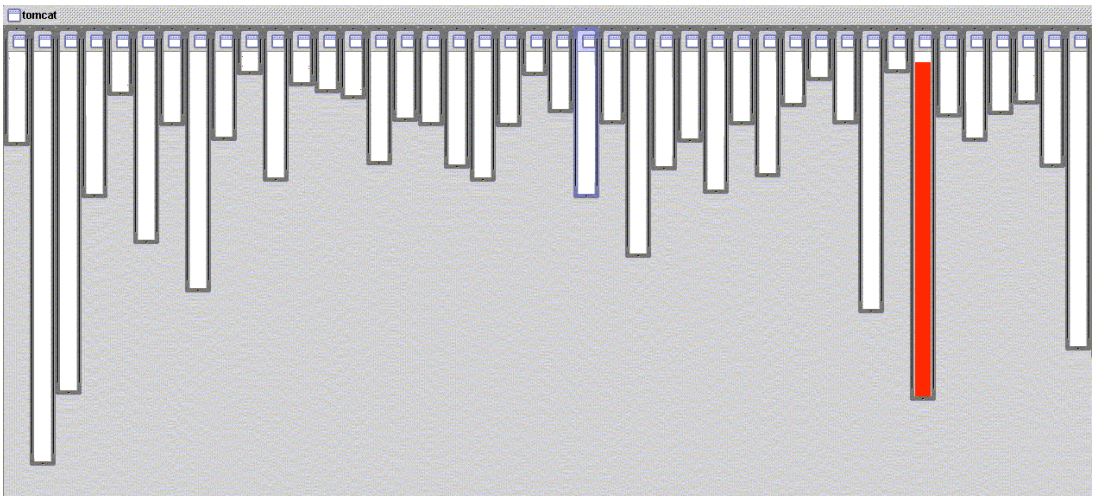
In software engineering, developers also need to identify and implement many different concerns. A concern of an application can be related to its functionality. For example, a calculator application will need to contain an implementation of the mathematical operators and an implementation of a user-interface to represent the calculator on the screen. The implementation of the operators and the user-interface are two separate concerns. Concerns can also be related to non-functional requirements such as performance and application distribution over a network. The principle of separation of concerns states that each of these concerns must be considered in isolation throughout the entire software development lifecycle. This requires that software engineers model, design and implement all concerns separately. For example, at the software implementation level, this means that each concern is implemented in its own module. Separation of concerns thereby reduces the complexity of each individual module which impacts the application's evolvability and reusability in a positive way. Concerns that are contained within a single module are relatively easy to reuse, while concerns that are contained in modules together with other concerns, are not.

2.1. Crosscutting Concerns

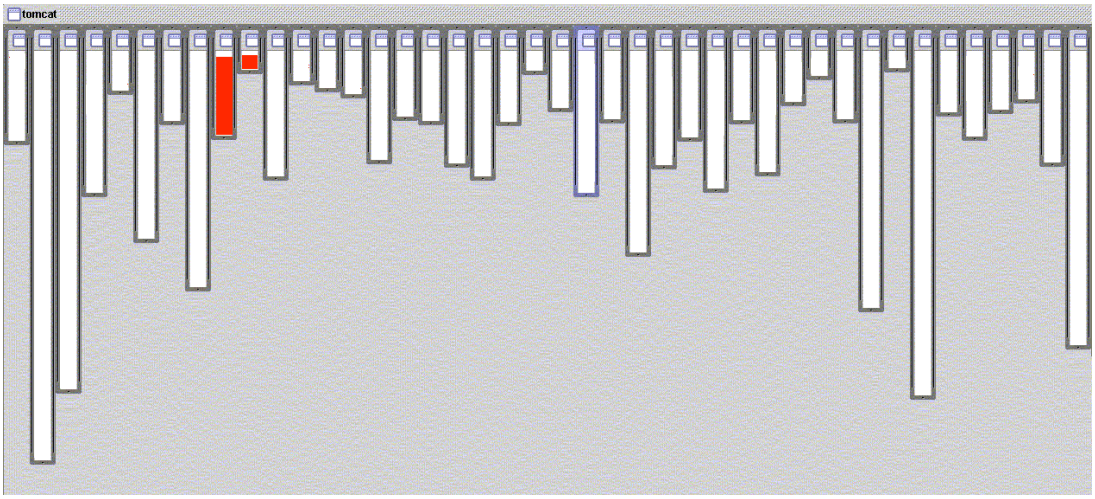
In object-orientation (which is one of today's most advanced and wide-spread development paradigms), separation of concerns is achieved by decomposing an application into individual objects. In an ideal situation, each object represents the implementation of a single concern. Object-orientation already provides significant support to achieve this through encapsulation, polymorphism, inheritance and delegation. Unfortunately, there are still concerns whose implementation will always remain distributed throughout many different objects, regardless of the chosen decomposition in objects. These concerns are said to cross-cut the other concerns. Common examples of crosscutting concerns are synchronisation policies in multi-threaded systems, error handling, enforcement of real-time constraints, fault tolerance mechanisms, and so on. Each of these crosscutting concerns re-

quires an implementation that is at least partially scattered over many other modules. Crosscutting concerns are problematic because their implementations are *tangled* with other concerns and/or *scattered* throughout the entire application. The tangling of the implementations breaks the principle of separation of concerns because a single module contains the implementation of more than one concern. This complicates the code, hampers reuse of the individual concerns and makes them complex to evolve. The scattering of the representation of a crosscutting concern throughout the entire application's design or implementation, renders it particularly hard to evolve because all modules that are crosscut need to be evolved. Even worse, the lack of an explicit representation and modularisation of crosscutting concerns makes it impossible to reuse them in other software applications. This is exactly the problem that can be tackled using aspect-orientation. Aspect-oriented software development explicitly represents crosscutting concerns as separate entities, solving many of the associated evolution and reusability problems. How aspect orientation achieves this is explained in the next section.

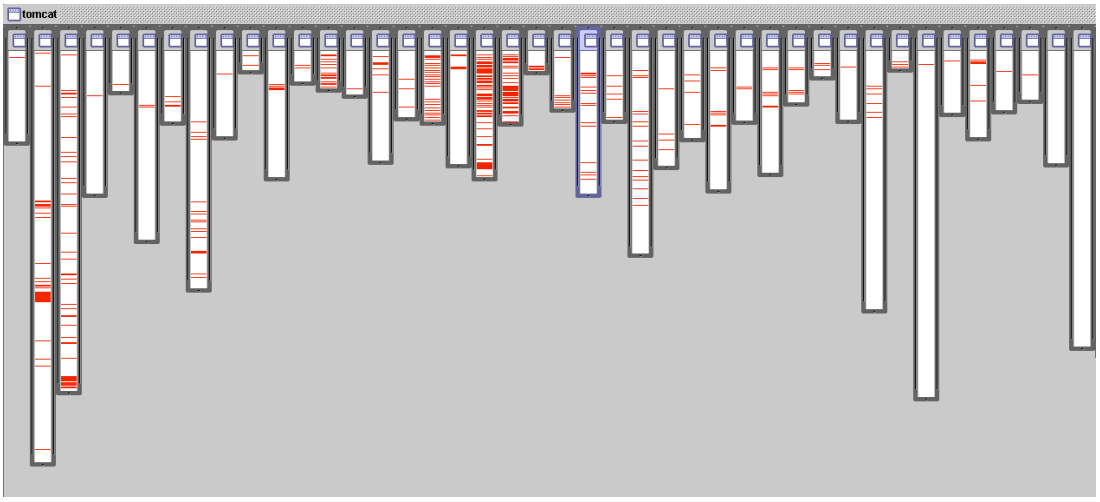
A good illustration of crosscutting concerns versus modularized concerns in a software application was given by Kiczales et al. in [7]. Consider the following three concerns in the Apache Tomcat webserver application: XML parsing (1), URL pattern matching (2) and logging (3). These concerns are respectively visualized in the following three figures (taken from [7]). The modules (classes) of the webserver's implementation are visualized as vertical bars. While the first two concerns are well modularized, the logging concern is spread across multiple implementation modules. The logging concern in this application is a good illustration of a crosscutting concern.



(1) Implementation of the XML parsing concern (shown in red).



(2) Implementation of URL pattern matching concern (shown in red).



(3) Implementation of the logging concern (shown in red).

Of course, crosscutting concerns are not limited to object-orientation. Crosscutting concerns can also be observed when the development occurs in other paradigms. Crosscutting concerns even exist in our example of the architectural design of the building. In fact, many of the mentioned plans are clearly crosscutting. For example, electrical wiring is a crosscutting concern that would be spread throughout all floor plans if it would not be drawn onto a separate plan. Crosscutting concerns are also inherent to any kind of decomposition one would make. This is referred to as the *tyranny of the dominant decomposition*. It means that there are restrictions (or tyranny) on the software engineer's ability to modularly represent particular concerns. These restrictions are imposed by the selected decomposition technique (i.e. the dominant decomposition) [2].

It is also important to stress that crosscutting concerns exist throughout the entire software development lifecycle. In other words, crosscutting concerns exist in the implementation as well as in the design and analysis artifacts of a software application. In what follows, we explain aspect-orientation with a large focus on the software implementation level (i.e. the aspect-oriented programming languages). Afterwards, we describe how aspect-oriented concepts are treated during other phases of software development (e.g. analysis and design).

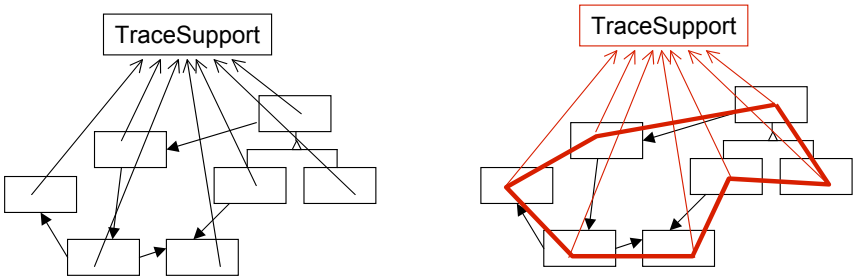
3. Aspectual Decomposition

To modularize the crosscutting concerns, software developers need a different decomposition technique. Modules in contemporary programming languages and paradigms are all based on some form of functional (de)composition (e.g. subroutines, functions, objects, components). To characterize these modules, Kiczales [5] coined the term *generalized procedure*:

> *Many existing programming languages, including object-oriented languages, procedural languages and functional languages, can be seen as having a common root in that their key abstraction and composition mechanisms are all rooted in some form of generalized procedure.*

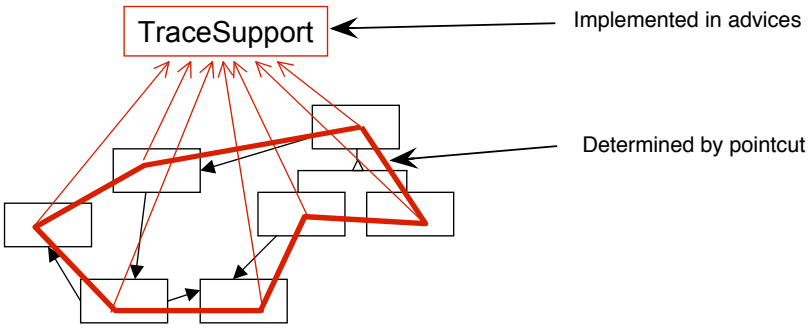
Aspect-orientation proposes a fundamentally new kind of modularisation that goes beyond generalized procedures: an *aspect*. An aspect is a module that can localize the implementation of a crosscutting concern. The key to this modularisation technique lies in its module composition mechanism. Subroutines explicitly invoke the behaviors implemented by other subroutines. In contrast, aspects have an implicit invocation mechanism. An aspect's behavior is implicitly invoked in the implementation of other modules. Consequently, the implementers of these other modules can be largely unaware of the crosscutting concern.

The difference between aspects and generalized procedures is illustrated in the figure below (adapted from [8]). The left-side of the figure shows an object-oriented implementation where the concern 'tracing support' is modularized in an object.



The right-side shows the corresponding implementation using an aspect. In the object-oriented implementation, all other objects need to consistently invoke or call the tracing support object (these calls are visualized by the arrows). Although a lot of the implementation of tracing support is modularized inside a single object, this implementation requires that all other objects implement a consistent usage of the tracing support. Adapting the tracing support concern might require an adaptation of the other objects that invoke tracing support. Removing tracing support will even require to change the implementation of all other objects. In contrast, the implementation of the trace support concern as an aspect (on the right) ensures that the consistent invocation of the trace support object is captured by the implementation of trace support itself. In the figure, the aspect implementation contains all red elements, which includes the tracing implementation and the invocation of tracing that would otherwise be spread throughout all other objects. The red line illustrates the crosscutting nature of this consistent invocation.

The implicit invocation mechanism still requires that the aspect itself specifies *where* or *when* it needs to be invoked. An aspect's implementation consequently consists of two conceptually different parts: the aspect *functionality* code and the aspect *applicability* code. The aspect's functionality code is not essentially different from 'regular' code and is executed when the aspect is invoked. This invocation of the aspect is determined by the aspect applicability code. This code contains statements that specify *where* or *when* the aspect needs to be invoked. In standard AOSD terminology [2], this aspect applicability code is referred to as a *pointcut* expression and the aspect functionality code is referred to as the aspect's *advice* code. Since a single aspect can consist of multiple different functionalities that need to be invoked from various different places in the code, an aspect's implementation can consist of several pointcuts and advice code segments.



3.1. Join point Models and Pointcuts

In all aspect-oriented programming languages, aspects can only be invoked at some well defined and principled points in the program's execution. These points are referred to as *join points* and the possible kinds of join points are described in a *join point model*. In standard AOSD terminology, the term join point is not limited to program execution but is extended to the entire software development lifecycle [2]:

> *A join point is a point of interest in some artefact in the software lifecycle through which two or more concerns may be composed.*
>
> *A join point model defines the kinds of join points available and how they are accessed and used.*

Possible kinds of join points in the execution of a program are, for example, assignment statements, method or constructor calls, variable references, etc. For each aspect-oriented programming language (in short: aspect language), a join point model describes these possible kinds of join points and how they can be determined in a pointcut. Since an aspect's behavior can only be invoked at a join point, a pointcut describes the set of join points where the aspect's advice needs to be invoked. More specifically [2]:

> *A pointcut is a predicate that matches join points. A pointcut is a relationship 'join point -> boolean', where the domain of the relationship is all possible join points.*

A pointcut is indeed often expressed using a set of dedicated predicates that reason about the entire possible set of join points in a software application to determine the correct set of join points for a specific aspect. The entire possible set of join points itself is determined by the join point model of the aspect language at hand. In our tracing example, the aspect-

oriented implementation features a pointcut definition that describes exactly those points where the object-oriented implementation contains a call to the tracing support object. Of course, in the aspect-oriented implementation, these calls are not explicitly present in the code but are captured by the aspect implementation through the pointcut definition.

3.2. Advice

We mentioned that the aspect's advice code (the aspect functionality code) is not substantially different from other code, nevertheless, there are some interesting issues. While many aspect languages feature advice code segments that basically can contain the same kind of procedural source code as standard functions or methods in that language, there are aspect languages where the advice code is expressed in a different language or even in a different paradigm. This is particularly true for domain-specific languages, that focus on the implementation of a single kind of crosscutting concern. Some of the earliest aspect languages were domain-specific aspect languages. Domain-specific languages are different from general-purpose languages because they offer dedicated constructs for the implementation of software in particular problem domains. For example, D is a domain-specific aspect language for synchronization concerns [12]. It is domain-specific because it offers constructs specific to express synchronisation. Although it is not named as such, the advice code in D consists of a specification of a synchronization policy in a dedicated notation.

In the so-called 'general-purpose' aspect languages, the implementation of advice often features aspect-specific constructs which does make it a little different from regular method or function implementations. For example, aspect languages often feature a language construct which offers explicit control over the control flow at a specific join point that is influenced by the advice. AspectJ is such a general-purpose aspect language where the statement `proceed()` can be used for that purpose. Using this construct, an advice can explicitly invoke or inhibit the execution of the original behavior defined at a certain join point. Of course, there exist more language features that are specific to advice code and specific to particular aspect languages, but these cannot be explained here.

3.3. An example: the synchronized buffer

To illustrate an implementation using aspects, consider the following code fragment taken from the implementation of a synchronized buffer. There are two major concerns here: buffer functionality and synchronization. The blue code statements are part of the implementation of the buffer functionality concern. The red code statements are part of the implementation of the synchronization concern. Obviously, the implementations of these two concerns are tangled. In this case, we identify that the synchronization concern is the crosscutting concern since it also crosscuts other modules (which are not shown here). Therefore, we will modularize the synchronization concern in an aspect.

```
class Buffer {
  char[] data;
  int nrOfElements;
  Semaphore sema;

  bool isEmpty() {
    bool returnVal;
    sema.writeLock();
    returnVal := nrOfElements == 0;
    sema.unlock();
    return returnVal;
  }}
```

The buffer implementation with tangled concerns.

From the above code fragment, one can distill that the join point of the synchronization aspect and the buffer functionality concern is the execution of the `isEmpty()` method. Indeed, whenever the method `isEmpty()` executes, we need to execute the synchronization locking code. For simplicity, the pointcuts of the aspect shown below capture only this join point and are written in red (mind that the aspect is written in pseudo-code). Obviously, the advices of the aspect contain the implementation of the synchronization code and are shown in green. First, there is a *before* advice to execute the locking and secondly, there is an *after* advice to execute the unlocking. As such, when the `isEmpty()` method executes, the aspect is implicitly invoked and the before advice executes, followed by the `isEmpty()` method body. The after advice is executed when the `isEmpty()` method execution finishes. There is no more tangling, the implementation of both concerns is cleanly modularized and separated.

```
before: reception(Buffer.isEmpty)
{ sema.writeLock();}

after: reception(Buffer.isEmpty)
{ sema.unlock(); }
```

The synchronization aspect

```
class Buffer {
  char[] data;
  int nrOfElements;

  bool isEmpty() {
    bool returnVal;
    returnVal := nrOfElements == 0;
    return returnVal;
  }
}
```

The buffer implementation

3.4. A domain-specific aspect-language example: the synchronized stack

An example of a synchronisation aspect in a domain-specific aspect language is given below, as well as an excerpt from the stack implementation code that it concerns to. The

aspect specifies that the `push` and `pop` methods are mutually exclusive and self exclusive. This means they cannot execute concurrently. The stack implementation and the synchronisation concern are clearly separated in different modules and the advice code of the synchronisation aspect is implemented in a domain-specific language for synchronisation.

```
class Stack {
    push(...) { ... }
    pop() { ... }
    ...
}
```

```
coordinator BoundedStackCoord {
    selfExclusive {pop,push};
    mutExclusive {pop,push};
}
```

3.5. Weavers

Aspect languages rely on a specific kind of compilers (or interpreters), called *weavers*, that compose the aspects' implementation with the other modules. Although one could argue that weavers are simply compilers for aspect languages and that many weavers are currently implemented as source-to-source or byte code transformations, there is an entire body of research that investigates efficient execution mechanisms for aspect languages, implemented through weavers or even aspect-aware virtual machines [16,17].

Compilers for aspect languages are called weavers because they need to *weave* the aspect code into the modules that are crosscut by the aspect. Since many of the existing aspect languages are extensions to contemporary object-oriented languages, their weavers transform the aspect language program into an object-oriented program where the aspect code is inserted (or woven) into the object-oriented implementation modules. Some weavers use source code or byte code transformation to achieve this (e.g. AspectJ), while other weavers use reflection to achieve the same result (e.g. AspectS). Apart from this technical implementation issue, research in weaving techniques focuses on the efficient implementation of aspect languages through compilation, interpretation and aspect-aware virtual machines. Similar to how the concept of virtual method tables is fundamental to the compilation of object-oriented programming languages, aspect-weaving techniques (or language implementation techniques in general, for that matter) are being researched to conceive the fundamental techniques for aspect weaving. For example, efficient weaving of aspects with complex and dynamic pointcuts or weaving of aspects that can be dynamically removed from the running software application, require sophisticated weaving techniques that are sound and that minimize the runtime performance overhead.

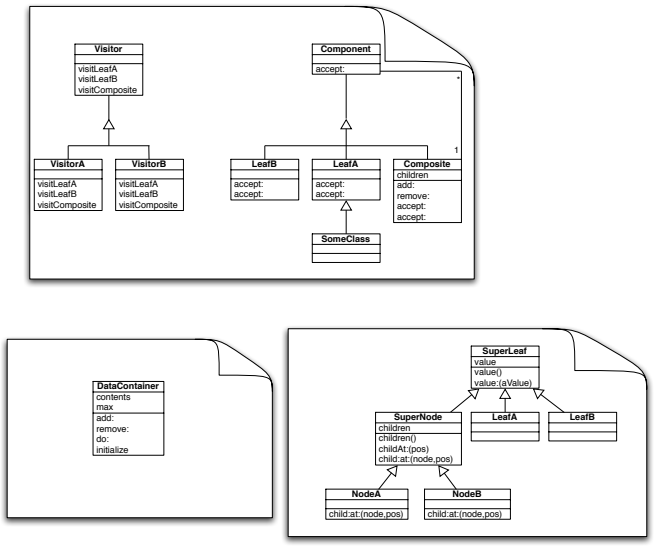
4. Symmetric vs. Asymmetric Decomposition

We have now described the aspectual decomposition technique which introduces a special module to modularize crosscutting concerns: the aspect module. This technique is an asymmetric decomposition technique because crosscutting concerns are modularized in a special kind of module (i.e. an aspect). In contrast, in symmetric decomposition, the same kind of module is used to modularize all concerns, wether the concern is crosscutting or not. A good example of such a symmetric decomposition technique that allows to modularize crosscutting concerns is *Multidimensional Separation of Concerns* [23].

In multidimensional separation of concerns, each concern of the system is modularized in a so-called 'hyperslice'. A hyperslice contains a decomposition of the program that is adequate for the implementation of a single concern. Consequently, a hyperslice again contains a number of modules. In an object-oriented context, hyperslices contain an imple-

mentation of a set of (cooperating) objects. Each of these objects contains only the required implementation elements (methods, variables, ...) for that concern. In other words, a hyperslice is a specific view on the entire software application, limited to one concern. In fact, multidimensional separation of concerns is a generalisation of Subject-Oriented Programming [26].

As an illustration, consider the following visualisation[1] of three hyperslices, each containing the implementation of a concern of some software application. Each hyperslice is visualized as a separate UML class diagram, representing the implementation of a single concern of the software application.



As one can see from this visualization, the decomposition in hyperslices very much corresponds to the different architectural design plans for the building that we presented earlier on. In essence, all modules are of the same kind (a hyperslice or a plan respectively), wether they implement a crosscutting concern or not. The entire building's design is constructed by composing all separate design plans. Similarly, the complete software application is a composition of all hyperslices. That software application thereby implements all the concerns that are implemented in each hyperslice.

Hyperslices are composed into complete applications using a set of composition rules that establishes how the different modules inside a hyperslice need to be composed with the different modules in other hyperslices. Hyperslices can easily contain the implementation of crosscutting concerns which crosscut the other hyperslices in ways specified in these (separate) composition rules. In essence, these composition rules fulfill the role of the join point model and the pointcuts of the aspects. In that respect, the definition of a join point is still adequate in the context of these symmetric approaches because it is a point in the implementation of a hyperslice where it can be composed with another hyperslice through the specification of a composition rule.

[1] The visualisation represents that a single application can be decomposed in different hierarchical decomposition. The actual contents of the diagrams inside the hyperslices is not important.

Other symmetric decomposition techniques are based on class families, mixin layers [27], class boxes [28], etc. Although the specific composition rules and other possibilities of each of these techniques can differ a lot, they are similar in the fact that they all propose a modularisation technique that is orthogonal to the (object-oriented) modularisation.

4.1. An example: the synchronized buffer revisited

Previously, we provided an example decomposition for the synchronized buffer implementation into a functionality module and a synchronization aspect. To clarify the symmetric decomposition approach, we now present the same example using a symmetric decomposition approach. Again, no particular technology is chosen and merely the concepts are clarified. Below, the modularisation of the two main concerns of the example are shown: the synchronisation and the functionality concern. The composition of these two concerns is specified through relations that compose the `lock()` method with the `isEmpty()` method such that `lock()` executes before `isEmpty()`. Similarly, the `unlock()` method needs to be composed with the `isEmpty()` method such that it executes after the body of the `isEmpty()` method. Composition can be specified in a specific composition language (e.g. in multidimensional separation of concerns) or can use standard programming language composition relations (e.g. inheritance or mixin composition).

```
class Buffer {
  char[] data;
  int nrOfElements;

  bool isEmpty() {
    bool returnVal;
    returnVal := nrOfElements == 0;
    return returnVal;
  }
```

The buffer functionality concern

```
class Synchronizer {
   Semaphore sema;

  bool lock()
  {return sema.writeLock;}

   bool unlock()
  { return sema.unlock();}
}
```

The synchronisation concern

5. AOSD

The field of Aspect-Oriented Software Development originated first at the programming level through the conception of aspect-oriented programming languages. In a sense, this led to the fact that most aspect-oriented ideas are perceived at the technology (i.e. programming) level. Recently, aspect-oriented analysis and design techniques have emerged and attention is given to formal verification techniques dealing with aspect interactions and interferences. We will now provide an overview of many of today's aspect-oriented software development techniques and technologies throughout the software development lifecycle.

5.1. Aspect-oriented Requirements Engineering

Requirements engineering techniques that explicitly recognize the importance of clearly identifying and treating crosscutting concerns are called Aspect-oriented Requirements Engineering Approaches (AORE approaches) [4]. The emergence of aspect-oriented programming languages has raised the explicit need to identify crosscutting concerns already during the analysis phase. Besides this observation, the modular representation of crosscutting requirements is a first step to ensure traceability of crosscutting concerns through all other artifacts of the software lifecycle (architecture, design and implementation). Aspect-oriented requirements engineering approaches improve existing requirements engineering approaches through an explicit representation (and modularisation) of concerns that were otherwise spread throughout other requirements artifacts (such as use cases, goal models, viewpoints, etc.).

Contemporary (non-aspect-oriented) requirements engineering approaches have been developed to primarily deal with one type of concerns. Some approaches have underlined the importance of non-functional concerns and proposed means to ensure their fulfillment in a system. Other approaches have focused on ensuring the required functionality of a system. In contrast, aspect-oriented approaches such as Cosmos and CORE explicitly acknowledge that all concerns are equally important and should be treated consistently [4]. Furthermore, AORE recognizes that all kinds of requirements (both functional and non-functional) can have a crosscutting influence on other requirements. For example, in traditional viewpoint-based analysis, security concerns are present in many viewpoints but the security requirement has no modular representation of its own. Furthermore, security and response-time requirements can influence each other.

Therefore, AORE approaches adopt the principle of separation of concerns at the analysis phase (the early separation of concerns). In other words, AORE approaches provide a representation of crosscutting concerns in requirements artifacts. For example, AORE Arcade provides the notion of crosscutting concern next to viewpoints in a requirements artifact. Besides this fact, these approaches also focus on the composition principle: it should be possible to compose each concern/requirement with the rest of the concerns/requirements of the system under construction to understand interactions and trade-offs among concerns. This composability of requirements is a central notion of AORE. It means that AORE techniques should have well-defined join point models and composition semantics. Join point models of AORE approaches identify points through which requirements can be composed. For example, join points can be individual requirements expressed in a viewpoint.

As an example, consider a security aspect at the requirements-level which specifies simple login-based authentication for all users of the system (example taken from [29]). This

top-level requirement is refined to provide more specific rights for different categories of users, e.g., managers, system administrators, etc. Let us assume that we have a viewpoint-based partitioning of our stakeholder requirements. A composition specification (based on our above example definition of a join point model) would take the following form:

```
Constrain all requirements specified in all user viewpoints by the top-level
security requirement.
     Constrain all requirements in administrator viewpoint
     by the security policy for administrators.
     Constrain all requirements in manager viewpoint
     by the security policy for managers.
     …
…
```

To draw an analogy with the aspect-oriented concepts introduced before: In the above composition specification, the green parts correspond to pointcuts, the underlined elements are the kind of advice and the red elements are the advice behavior. In this example, the requirements aspect itself is separated from its composition specification (which is provided above).

The composability of requirements allows not only reviewing the requirements in their entirety, but also detection of potential conflicts very early on in order to either take corrective measures or appropriate decisions for the next development step. Last but not least, the mapping of the concerns at the requirement level to concerns in later lifecycle stages will reveal wether the concern maps to a crosscutting artifact or wether it becomes absorbed into other artifacts.

Summarizing, AORE approaches focus on systematically and modularly treating, reasoning about, composing and subsequently tracing crosscutting concerns via suitable abstraction, representation and composition mechanisms tailored to the requirements engineering domain. Arcade, ARGM, Aspectual Use Cases, Cosmos, AOREC, etc. are such aspect-oriented requirements engineering approaches that are extensively described in [4].

5.2. Aspect-oriented Architecture

“The architecture of a program or computing system is the structure of the system, which comprise software components, the externally visible properties of those components, and the relations among them” [24]. Current software architecture design methods do not make an explicit distinction between conventional architectural concerns that can be localized using current architectural abstractions and architectural concerns that crosscut multiple architectural modules [4]. PCS, DAOP-ADL, AOGA, TranSAT, ASAAM, etc. are all new architectural design approaches that explicitly treat crosscutting architectural concerns and are described in [4].

In such aspect-oriented architectural design approaches, an architectural aspect is an architectural module that has a broad influence on a number of other architectural modules (e.g.,security module, providing authorisation, authentication and encryption/decryption functionality). Using contemporary architectural approaches, the risk is that such potential aspects might be easily overlooked during the software architecture design and remain unsolved at the design and programming level. This may lead to tangled code in the system and consequently the quality factors that the architecture analysis methods attempt to verify will still be impeded. Architectural design approaches offer explicit mechanisms to

identify and specify aspects at the architecture design level. In this sense aspectual architecture design approaches describe steps for identifying architectural aspects and their related tangled components. This information is used to redesign the given architecture in which the architectural aspects are made explicit. This is different from traditional approaches where architectural aspects are implicit information in the specification of the architecture.

5.3. Aspect-oriented Design

Aspect-oriented design focuses on the explicit representation of crosscutting concerns using adequate design languages. An aspect-oriented design language consists of some way to specify aspects, some way to specify how aspects are to be composed and a set of well-defined composition semantics to describe the details of how aspects are to be integrated [4]. In object-orientation, the UML design language became the de-facto standard. Several aspect-oriented extensions to UML, were conceived to represent aspect-oriented concepts at the design-level. Some of these UML extensions are AODM, Theme/UML, SUP, UFA, AML, etc. These and also some new (non-UML) design languages such as CoCompose are extensively described in [4]. Here we describe the main concepts of aspect-oriented design.

In the infancy of aspect orientation, developers simply used object-oriented methods and languages (such as standard UML) for designing their aspects. This proved difficult, as standard UML was not designed to provide constructs to describe aspects: Trying to design aspects using object-oriented modeling techniques proved as problematic as trying to implement aspects using objects. Without the design constructs to separate crosscutting functionality, similar difficulties in modularizing the designs occur, with similar maintenance and evolution headaches. At a high level, the main contribution of aspect-oriented design has been to provide designers with explicit means to model aspect-oriented systems, deriving software engineering quality properties as a result. In particular, this breaks down into a number of sub-contributions. Aspect-oriented design provides a means for the designer to reason about concerns (whether they are crosscutting or not) separately, and to capture concern design specifications modularly. For example, AODM [31] extends UML with a design notation for AspectJ-like aspect-oriented programs; the Theme/UML [32] approach follows the symmetric decomposition approach and provides *theme* modules that can capture concerns (crosscutting or not) in the design.

Where there is modularisation, there must also be a means to specify how those modules should be composed into the full system design. Aspect-oriented design provides a means to specify how concern modules should be composed. This includes both a means to specify how to compose concerns at a later stage of the development cycle, and also a means to compose concern design artefacts. For example, the *themes* in Theme/UML are complemented with a composition specification that describes their integration.

When composing concern designs, or specifying how concerns should be composed at a later stage of the development lifecycle, it is likely that there are points of conflict or cooperation between some concerns to be composed. Aspect-oriented design provides a means to specify how to resolve conflicts between concerns and to specify how concerns cooperate. Such conflict or cooperation specifications guide the composition process.

Another significant contribution of aspect-oriented design is the extent to which there is traceability of concerns to lifecycle stages both preceding design, and post design. Such traceability increases the comprehensibility and maintainability of the system. In addition to

traceability of concerns, aspect-oriented design provides a mapping of the constructs used in design to those used by lifecycle stages both preceding design and post design, further enhancing the traceability.

5.4. Aspect-oriented Programming

Aspect-orientation manifests itself at the programming level as aspect-oriented programming languages. Most of these aspect languages are existing (object-oriented) languages extended with aspect-oriented features to represent aspects, express pointcuts, implement advice, etc .

The most prominent and mature aspect language today is most probably AspectJ [9], which is an aspect-oriented extension to Java. Aspects in AspectJ look like regular Java class definitions but they can include advice and pointcut definitions. An advice is expressed using regular Java statements, augmented with some specific features for which we refer to [9]. An advice can influence the behavior at specific join points by executing before, after or around the execution of the join point. Join points in AspectJ are specific points in the execution of a Java program such as method calls, exception throwing, variable references, etc .

An overview of AspectJ and other important aspect languages today can be found in [10]. This includes a description of other prominent aspect languages such as JAsCo, CaesarJ, AspectS, Object Teams, HyperJ, JBOSS AOP, Compose*, DemeterJ, AspectC++, etc but also more prototypical and research-oriented aspect languages such as CARMA, OReA, AO4BPEL, Alpha, EAOP, FuseJ, AspectCOBOL, KALA, etc.

5.5. Verification of Aspect-oriented Programs

Although aspect-oriented programming languages allow software engineers to modularize crosscutting concerns, it does not mean that all concerns can be treated independently from each other. Aspects can depend on other aspects and aspects can potentially harm the reliability of a system to which they are woven, and could invalidate essential properties that already were true of the system without the aspect. Consider for example our example of the architectural design of a house. Multiple design plans are in fact modularizations of crosscutting concerns (e.g. the electrical wiring plans). Just because these plans are separated does not mean that they are independent of the other plans. Insertion of a new window may, for example, require to adapt the electrical wiring plan because wires cannot run through windows.

Aspect-orientation rises new challenges in software validation and verification techniques to ensure that the desired functionality is exerted by the system. It is necessary to show that aspects actually do add the intended cross-cutting properties to the system. To ensure the correctness of software with aspects, there is considerable research on using formal methods and testing techniques especially adapted to aspects. These are surveyed in [18].

5.6. Aspect-oriented Middleware

Although middleware is not lifecycle stage, it is an important and large application area for aspect-oriented ideas. Many software developers have adopted middleware approaches to aid in the construction of large-scale distributed systems. Middleware facilitates the development of distributed software systems by accommodating heterogeneity, hiding distribution details, and providing a set of common and domain specific services.

However, middleware itself is becoming increasingly complex; so complex in fact that it threatens to undermine one of its key aims: to simplify the construction of distributed systems. In this regard, middleware and AOSD complement each other naturally, in that each plays to one another's strengths while obviating many of the problems inherent in traditional middleware approaches. Essentially, aspect-oriented middleware offers an improvement over traditional middleware by delivering flexibility, reliability and performance in a more balanced manner. A detailed survey of aspect-oriented middleware approaches can be found in [25].

6. AOSD Timeline

Although the term Aspect-oriented Software Development (AOSD) was not coined before 2002, this was not the beginning of AOSD-related research. Aspect-oriented Programming (AOP) was first introduced in 1997 [5] and other AOSD-related techniques and technologies, were first grouped as 'Advanced Separation of Concerns' techniques. Moreover, advanced separation of concerns has been a prominent topic in the metaprogramming and reflection community. We provide a brief summary of the history of AOSD here. The interested reader can find an extensive history by C.V. Lopez in [19].

6.1 Metaprogramming and reflection: the roots of AOP.

Metalevel architectures à la Smalltalk [13,14] and CLOS [15] have clearly illustrated the potential of reflection and metaprogramming [20] to deal with separation of concerns [6,7]. In a reflective architecture, crosscutting concerns were handled at the meta level, instead of at the base level. Reflective programs were used to modularize a crosscutting concern. These programs adapt the behavior of the base level program such that it includes the behavior of the crosscutting concern. Aspect-oriented programming differs from the meta level approach because it now provides specific language constructs to modularize crosscutting concerns. For example, aspect languages include a specific feature to quantify over the program's join points to define a pointcut. It can also be argued that aspect languages provide limited and disciplined access to reflective programming for the specific purpose of modularizing crosscutting concerns.

6.2 Advanced Separation of Concerns

With the conception of aspect-oriented programming (AOP) in 1997 [5], the aspect-orientation community was born. AOP and other techniques and technologies that are focused on modularisation of crosscutting concerns, grouped into the common denominator of 'Advanced Separation of Concerns'. These other techniques include Multidimensional Separation of Concerns [23], Adaptive Programming [21] and Composition Filters [22]. These latter techniques did not originate directly from reflection and metaprogramming but rather originated directly from the need for a clear separation of concerns at the modeling as well as at the implementation level. Numerous workshops were held on this topic at various international conferences such as ECOOP, OOPSLA, ICSE, etc.

6.3 Aspect-oriented Software Development

With the advent of the first international conference on Aspect-Oriented Software Development in 2002, the community more-or-less adopted this common denominator to refer to advanced techniques and technologies for the modularisation of crosscutting concerns.

7. Acknowledgements

Many thanks to Eddy Truyen, Shmuel Katz, Kris Gybels, Johan Fabry, Ruzanna Chitchyan and Wim Vanderperren for their valuable comments on earlier versions of this text. The authors also appreciate the valuable work of the referenced authors below. Their reports and papers certainly contributed to the creation of this introduction to AOSD.

8. Bibliography

[1] T. Elrad,R. Filman, A. Bader, “Aspect-oriented Programming: Introduction”, In Communications of the ACM, Volume 44, number 10, pages 28-32, 2001.

[2] “AOSD Ontology 1.0: Public Ontology of Aspect Orientation”, Report of the EU Network of Excellence on AOSD by Klaas van de Berg, Jose-Maria Conejero, Ruzanna Chitchyan (editors), 2005.

[3] P. Cointe, H. Albin Amiot, S. Denier, From (meta) objects to aspects - From Java to AspectJ, OSCAR workshop, November 2004.

[4] “Survey of Aspect-oriented Analysis and Design Approaches”, Report of the EU Network of Excellence on AOSD by Ruzanna Chitchyan, Awais Rashid, Pete Sawyer, Allesandro Garcia, Monica Pinto Alarcon, Jethro Bakker, Bedir Tekinerdogan, Siobhan Clarke, Andrew Jackson, 2005.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, J. Irwin, “Aspect-oriented Programming”, In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Springer-Verlag, 1997

[6] Dave Thomas, “Reflective Software Engineering - From MOPs to AOSD”, In Journal of Object Technology, Volume 1, number 4, pages 17-26, October 2002.

[7] G. Kiczales, AspectJ: Aspect-Oriented Programming Using Java Technology (0.7), presented at JavaOne Conference, 2000.

[8] G. Kiczales, Tutorial on Aspect-Oriented Programming with AspectJ, FSE 2000.

[9] AspectJ, Aspect-oriented Programming in Java, http://www.aspectj.org.

[10] “Survey of Aspect-oriented Languages and Execution models”, Report of the EU Network of Excellence on AOSD by Johan Brichau and Michael Haupt (editors), 2005.

[11] E.W. Dijkstra, “A Discipline of programming”, Prentice Hall, Englewood Cliffs, NJ, 1976

[12] C.V. Lopes,” D: A Language Framework for Distributed Programming”, Ph.D. Dissertation, Northeastern University, 1997.

[13] Adele Goldberg and Dave Robson, “Smalltalk-80: the language”, Addison-Wesley, 1983.

[14] F. Rivard, “Smalltalk: A Reflective Language”, In Proceedings of Reflection’96, 1996.

[15] G. Kiczales, J.D. Rivieres, D. Bobrow, “The Art of the Metaobject Protocol”, MIT Press, Cambridge, MA (1991)

[16] Christoph Bockisch, Michael Haupt, Mira Mezini and Klaus Ostermann, “Virtual Machine Support for Dynamic Join Points”, In Proceedings of International Conference on Aspect-Oriented Software Development (AOSD'04)

[17] Michael Haupt and Mira Mezini, “Virtual Machine Support for Aspects with Advice Instance Tables”, First French Workshop on Aspect-Oriented Programming, Paris, France, Sep. 14th, 2004.

[18] “A Survey of Verification and Static Analysis for Aspects”, Report of the EU Network of Excellence on AOSD by Shmuel Katz, 2005.

[19] C.V. Lopez, “AOP: A Historical Perspective", In R. Filman et al. (editors), Aspect-Oriented Software Development, Addison-Wesley, 2004. ISBN 0321219767.

[20] P. Maes, “Computational Reflection”, PhD thesis, Dept. of Computer Science, Vrije Universiteit Brussel, Belgium, 1987.

[21] K.J. Lieberherr, I. Silva-Lepe, C. Xiao, “Adaptive Object-oriented Programming using Graph-based customization”, Communications of the ACM, 37(5), pages 94-101, May 1994.

[22] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa, “Abstracting Object Interactions using Composition Filters”, In M. Guerraoui, O. Nierstrasz, M. Riveill (editors), Object-based Distributed Processing, Springer-Verlag, LNCS, 1994.

[23] P.L. Tarr, H. Ossher, W.H. Harrison, S.M. Sutton Jr., “Multi-Dimensional Separation of Concerns in Hyperspace”, In Proceedings of the International Conference on Software Engineering (ICSE), pages 107-119, 1999.

[24] L. Bass, P. Clements, R. Kazman, “Software Architecture in Practice”, Addison-Wesley, 1998.

[25] “Survey of Aspect-oriented Middleware”, Report of the EU Network of Excellence on AOSD by N. Loughran, N. Parlavantzas, M. Pinto, L. Fuentes, P. Sanchez, M. Webster, A. Colyer (editors), 2005.

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

[27] Y. Smaragdakis, D. Batory, “Implementing Layered Designs with Mixin Layers”, In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag, LNCS, Volume 1445, pages 550-570, 1998.

[28] A. Bergel, S. Ducasse, O. Nierstrasz, “Classbox/J: Controlling the Scope of Change in Java”, In Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05), New York, NY, USA, ACM Press, 2005, To appear.

[29] J. Araujo, E. Baniassad, P. Clements, A. Moreira, A. Rashid, B. Tekinerdogan, “ Early Aspects: The Current Landscape “, Technical Report, Lancaster University, February 2005.

[30] A. Rashid, A. Moreira, J. Araujo, “Modularisation and Composition of Aspectual Requirements”, In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, ACM, Pages 11-20, 2003.

[31] D. Stein, S. Hanenberg, R. Unland, "A UML-based Aspect-Oriented Design Notation For AspectJ," presented at Aspect-Oriented Software Development (AOSD 2002), Enschede, The Netherlands, 2002.

[32] E. Baniassad and S. Clarke, "Theme: An Approach for Aspect-Oriented Analysis and Design," presented at International Conference on Software Engineering, 2004.