On The Role of Multi-Dimensional Separation of Concerns in Software Architecture

Position Paper for the OOPSLA'2000 Workshop on Advanced Separation of Concerns

Mohamed Mancona Kandé and Alfred Strohmeier
Swiss Federal Institute of Technology Lausanne
Software Engineering Laboratory, CH-1015 Lausanne, Switzerland
Email: {mohamed.kande, alfred.strohmeier}@epfl.ch

ABSTRACT
In this paper we study the need for multidimensional separation of concerns in architecture representations, including architecture-centered software development. We present a case study of a simple video surveillance system, describe its software architecture using an ADL called Wright, and we discuss the pragmatics and problems in the use of ADLs in general, compared to a concern-based approach to software architecture description.

Our position is that current ADLs provide architectural abstractions that need to be extended to achieve the major goals of software architecture. Furthermore, in order to cover all concerns of importance in a software architecture description, software architects must be able to separate various dimensions of concern and consider the system from multiple perspectives simultaneously.

Keywords: Multidimensional separation of concerns, software architecture, software architecture description, ADL, architectural viewpoints, architectural views, concern spaces.

1 INTRODUCTION

In the last ten years, software architecture has turned out to be a significant area of research and practice in software engineering. Representing architectures in an unambiguous and explicit way has been characterized as a critical issue in the design and construction of any complex software system [1]. The major goals of software architecture consist of providing blueprints for constructing software-intensive systems, helping stakeholders to understand, manage and analyze fundamental system properties, as well as providing means that allow stakeholders to communicate and reason about such system properties. These are admirable goals; however, despite important industrial and research activities in the area of software architecture, there are still many problems that remain considerable barriers to the achievement of the objectives of software architecture.

These barriers involve two categories of problems. The first category consists of problems that are related to the immaturity of software architecture in general. This category is typically characterized by some strong divergences in the field: there is still little agreement on what an architecture description language (ADL) is, and what characteristics of a software system should be specified by an ADL [4,5]. Likewise, there are still many controversies about the definition of software architecture, which continue to complicate the emergence of established architectural practices and their controlled evolution [6]. The second category consists of problems that are specific to the current trends in software architecture research and practice. These often result in certain restrictions of the expressive power of software architecture, e.g., in the inability of integrating existing ADLs with other software development artifacts [7].

Both categories of problems mentioned above are somehow related to the limitations of current architectural abstraction mechanisms (including software architecture methodologies, notations and tools [2,3,4,5,8,9,10]) to support simultaneous separation of concerns at multiple levels of abstraction. We believe that an architect, to be able to provide a software architecture description that reflects all architecturally significant aspects of a system, requires an appropriate use of the notion of multidimensional separation of concerns.

Multidimensional separation of concerns is a new field in need of attention in software engineering research and practice that was first introduced by Tarr and colleagues. These authors hypothesized that major difficulties relative to the improvement of software reuse, comprehensibility, component integration and high impact of change in software systems are due to a deficiency of separation of concerns. In addition, they argued that in spite of the presence of mechanisms to attain separation of concerns in all modern software formalisms, software artifacts still continue to exhibit properties associated with poor separation of overlapping concerns [11]. The initial work in this recent area is mainly focused on providing a support for multidimensional separation of concerns in programming languages and environments [12,13].

We have started studying the notion of multidimensional separation of concerns in the context of software architecture and examining how we can take advantage of the separation of overlapping concerns in both software architectural descriptions and architecture-centered software development. In this paper, we put emphasis on discussing the role of multidimensional separation of

concerns in the area of software architecture.

To demonstrate the need of multidimensional separation of concerns in software architecture, let us compare the architecture description we would obtain if we used a typical ADL, such as Wright, to the architecture description we would get if we used a concern-based approach to software architecture description.

2 AN ADL-BASED APPROACH TO SOFTWARE ARCHITECTURE DESCRIPTION

In this section, we briefly introduce an example of architecture description using Wright [10]. We present the Wright architectural modeling constructs and notations by illustrating the case study of a simple Video Surveillance System (VSS) [7]. As shown in Figure 1, VSS consists of a set of video cameras that interact with a control station over a communication platform. The example illustrates two kinds of software architecture constructs: the *component types* of the system and their interconnection described by a *connector type*.

The CameraDevice component type abstracts a set of geographically distributed video cameras, whereas the VideoControlStation component type abstracts the part of the system that remotely controls the cameras and continuously receives the video streams. The connector type, VSConnector, is an abstraction for the communication platform. It consists of two kinds of elements: a pair of *connection points* and the *protocol of interactions* between these connection points. Each connection point represents an interface to the connector a component can use. In order to be able to participate in a communication mediated by a connector, a component must implement this interface. The implementation might be in software or in hardware.

Conceptually, connection points can be seen as interface elements for both component and connector types. We use in Figure 1 two circles (white and black) to graphically represent the interfaces with the camera device and the video control station. Indeed, these are two connection points, conjugates of one another. As in previous work, to represent a component type in the topology of VSS, we use one of the UML extension mechanisms to define the «archComponent» stereotype, which has the properties of both a UML Class and a UML Package. The connector type is represented by a stereotype of Collaboration that contains a connector icon, shown by the black and white linked circles, at the upper right in the oval. The dashed line between the connection points is a UML binding that, in this case, conceptually represents the attachment of two connection points. (For further details on UML, see [14,15].)

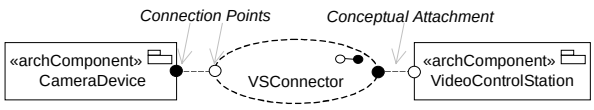

Figure 1 Topology for Video Surveillance System

Wright provides a support for separating a few dimensions (kinds) of concern: it allows us to separate the *structure* from *behavior* concerns in the architecture of a software system and it also fosters separation between the *computation*, *communication* and *configuration* concerns.

Structure of the System Architecture

In Wright the structure of a system architecture is represented as an arrangement of a set of typed components and connectors that work together. *Components* represent abstractions for independent computational entities or system-level storage units. *Connectors* abstractly represent interactions among components. Defining the structure of the system architecture using Wright consists in describing architectural styles or families of systems and declaring the configuration.

Figure 2 shows a static Wright specification of the VSS, which consists essentially of two parts: the first one represents the declaration of the architectural style (`SimpleVideoSurveillanceSystem`), while the second one declares the configuration (`SimpleSCSystem`). The *Style* introduces component and connector types, and constraints. The structure of each *Component* specification consists of a *Port* (`p`) and a *Computation* part. The *Connector* specification consists of two *Role*s (source and sink) and a *Glue*. Ports and roles describe elements of components and connector interfaces, respectively. These can be considered as Wright implementations of connection points in both components and connectors. The computation describes the entire behavior of the components, while the glue specifies the comprehensive behavior of connectors.

The configuration requires an instantiated style. Thus, it uses instances of the component and connector types that are defined in the style, to attach their ports to roles. In the example, the `SimpleSCSystem` configuration shows how both ports (`p`) of the component type instances `vcs` and `cd` are attached to the connector type instance `connector`. These attachments mean that the port `p` of the component `vcs` (instance of VideoControlStation) fills the role of type `sink` belonging to `connector`. At the same, the port `p` of the component `cd` (instance of CameraDevice) fills the role of type `source` that belongs to `connector`.

Behavior of the System Architecture

The Wright specification of the architectural behavior of the VSS describes a set of significant events that are processed by components, and the sequences in which these events occur. To describe the behavior of components and connector types, Wright allows us to specify a process for each of the following elements: port, role, computation and glue.

```
Style SimpleVideoSurveillanceSystem
 Component VideoControlStation
```

```
   Port p = videostreamrequest → ready → p ⊓ §
   Computation = streamCompute → p.videostreamrequest →
                 p.ready → Computation ⊓ §

 Component CameraDevice
   Port p = videostreamrequest → ready → p   §
   Computation = videostreamrequest → p.streamCompute →
                 p.ready → Computation   §

 Connector VSConnector
   Role sink = videostreamrequest → ready → sink ⊓ §
   Role source = videostreamrequest → ready → source ⊓ §
   Glue = source.videostreamrequest →
          sink.videostreamrequest → Glue
            source.ready → sink.ready → Glue
            §

 Constraints
  ∃! s ∈ Component,
  ∀ c ∈ Component : TypeCameraDevice (s)  ∧
  TypeVideoControlStation (c) ⇒ connected(c,s)
EndStyle

Configuration SimpleSCSystem
  Style SimpleVideoSurveillanceSystem
  Instances vcs : VideoControlStation;
         cd : CameraDevice;
         connector : VSConnector
  Attachments vcs.p as connector.sink;
         cd.p as connector.source
EndConfiguration
```

Figure 2 Static Wright Specification for Video Surveillance System

Note that the Wright notation for behavioral description indicates the direction of interactions by explicitly distinguishing initiated events (overlined) from observed events (not overlined). To make the editing work easier, we replaced these overbars by underlines in Figure 2.

The computation process specifies how to handle events arriving on any port of a component and how to send events through the ports. The `VideoControlStation` requests some video streams over and over again (by sending the `p.videostreamrequest` events) through the port `p` and waits for a response (`p.ready`) on the same port `p`; or it terminates successfully (§). In this particular case, the `VideoControlStation` decides by itself whether it makes another request or terminates. This way of taking a decision by itself is referred to as an internal choice and denoted by the ⊓ symbol. In contrast, an external choice has been used in the computation specification of the `CameraDevice`. This means that the computation process of `CameraDevice` is expected to reply to each request, and is not allowed to terminate in advance.

The process `p` assigned to the `VideoControlStation` port, defines the way this component interacts with its environment using this port. This is a local interaction protocol, which covers the same behavioral pattern as defined in the computation process mentioned above, except the internal part (specified by `streamCompute` event).

In this example, the specifications of both role processes (`source` and `sink`) are kept simple and identical to those of the ports. This makes it easier to see how instances of both component types can attach their ports to these roles to be interconnected in the configuration.

The *Glue* process specifies the interaction protocols between the roles of a connector. In our example, the `VideoControlStation` initiates an event to request video streams (`source.videostreamrequest`) that must be sent as `source.videostreamrequest` to the `CameraDevice`, and the response (`source.ready`) of the `CameraDevice` must be sent back as `sink.ready` to `VideoControlStation`.

So Wright proposes notations for abstractly and formally representing software architectures and exposing properties for analysis by promoting the independence of connectors and components and by increasing the flexibility to compose and reuse both connectors and components.

Current ADLs, including Wright, have provided a solid foundation on which one can explore some architectural abstractions that define various dimensions of concern in software architecture. However, through the pursuit of such a foundation of software architecture, we have begun to discover some inflexibility encountered by studying comparisons between existing ADLs [3,4,5].

Thus, ADLs allow architects to decompose systems along only a few dimensions of concerns. According to P. Tarr and her colleagues, these are *dominant dimensions of concern*. For instance, as the Wright notion of architecture description is essentially centered on the representation of architectural styles, we can consider styles to be the "dominant" dimension of concern in Wright-based software architectures.

3 A CONCERN-BASED APPROACH TO SOFTWARE ARCHITECTURE DESCRIPTION

Throughout this position paper, we take the premise that providing an architectural approach to large-scale software development is the right way of proceeding. However, as we discussed earlier, describing software architecture along only a restricted number of dimensions of concern is important, but often not enough. While restricting dimensions of concern facilitates the job of software architects, it makes it very difficult to express, analyze and reason about key structures of software architecture. In particular, this is the case when we admit that:

Software architecture is an abstract but fundamental "thing" that represents a bridge between requirements, program code and runtime execution environment.

The description of the structure or all the structures that constitute such a bridge is what we referred to as *software architecture description* all through this work. These structures involve numerous kinds of concerns that often need to be further refined, encapsulated, manipulated and implemented at other stages in the software life cycle.

To describe the software architecture of complex systems, software architects are concerned with the identification, representation, composition, decomposition and analysis of multiple structures. Thus, to foster the description of software architectures, we have started the *ConcernBASE* (**Concern**-**B**ased and **A**rchitecture-centered **S**oftware **E**ngineering) project. ConcernBASE is a software engineering approach that complements the abstraction mechanisms found in current ADLs, allows for simultaneous separation of overlapping concerns in software architecture description and provides a support for architecture-centered software development. Seen as an architectural approach, ConcernBASE allows us to consider the software architecture of a system as an abstract thing and to decompose it into a set of concern spaces, each of which has multiple architectural dimensions.

ConcernBASE is built around three basic abstractions: *architectural viewpoints*, *architectural concern spaces* and *architectural views*.

The relationships between these three basic elements are illustrated in Figure 3. Figure 3 presents two architectural viewpoints (the Structural Viewpoint and Architecture Analysis Viewpoint) and their corresponding concern spaces named Structural Concern Space and Architecture Analysis Concern Space.

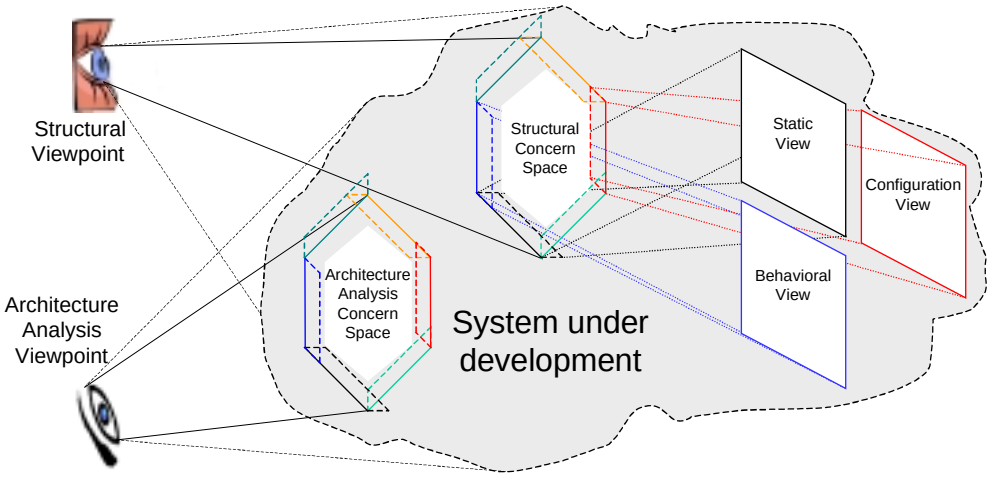

Figure 2: Relationship between Architectural Viewpoints, Concern Spaces and Views

Architectural Viewpoints

An architectural viewpoint defines a particular perspective of software architecture that establishes *rules* and *notations* for identifying architectural concerns, grouping these concerns into architectural dimensions and organizing the dimensions into one or more concern spaces. The rules determine the manner in which concerns are represented in the concern space. Rules are expressed by means of a notation that involves techniques for depicting elements on a particular architectural view. Rules also describe the conventions that guide a projection of concern space onto individual architectural views along some specific dimensions. The notations of a viewpoint describe the language elements that are needed for appropriately representing all the concerns of the software architect from the perspective defined by that viewpoint.

Architectural Concern Spaces

A concern space defines rules for representing dimensions of concern in certain architectural views. It describes a pattern or template from which to develop individual views by establishing the purposes and audience for each view and by providing techniques to create and analyze each of the views. Each architectural concern space defines a set of concerns relative to a particular perspective and encapsulates ideas, notions, elements, properties or other things that are architecturally significant and cut across multiple dimensions.

Thus, a concern space provides a mechanism for multidimensional separation of concerns, by mean of which various dimensions can be represented in an architectural view simultaneously. A concern space allows us to create, depict and analyze one or more architectural views at the same time and it specifies how to integrate and manage those architectural views.

Basically, an *architectural concern* can be an idea, notion, element, property, or any artifact that is of importance to the software architects and which can be classified according to various dimensions. Architectural concerns involve things like data, computation, signal, message, communication, synchronization, connector role, etc. In contrast, an *architectural dimension* is a set of architectural concerns that can be used to describe a specific aspect of software architecture. Each architectural dimension has structure and can be considered as a linguistic type that describes a way of looking at a set of different architectural

elements. Examples of architectural dimensions are components, connectors, configurations, etc.

We proposed in [7], a UML profile for software architecture description that supports the modeling of these dimensions by combining certain architectural concerns. For instance, we demonstrated that the component dimension could be used to encapsulate and combine the following four architectural concerns: computation, data, signal and messages.

Architectural Views

An architectural view is a specific way of presenting a software architecture description that illustrates some architectural concerns from a particular perspective. Since architectural views are incomplete architecture descriptions that reflect only a subset of a concern space, they systematically suppress all details of implementation, algorithm, and low-level data representation. An architectural view is needed to represent one or more architectural dimensions. Therefore, it is a suitable modularization mechanism that allows us to encapsulate any kind of software artifact of importance at architecture level. Architectural views can be overlapped, when they represent overlapping dimensions. ConcernBASE also allows architectural views to be nested according to the rules defined in the concern space. In this case, each nested view will be considered as a refinement of the outer view. This allows developers, for instance, to further manipulate, refine and implement various dimensions of concern that are represented in a particular architectural view.

Although it is outside the scope of this paper, the approach presented here could also be used to cover other phases of the software life cycle.

In addition the provision of explicit notations for each of the three basic abstractions, ConcernBASE allows us to combine these abstractions to provide some flexible and concern-based mechanisms for the decomposing and composing of software architectural structures using any modeling language. In our work, we choose to use an extension of the standard UML.

4 CONCLUSION

In this paper, we have discussed the limitations of existing ADLs relative to the simultaneous separation of concerns. We feel that using multidimensional separation of concerns in software architecture facilitates the representation of software architectures and provides the ability to expose the dimensions along which substantial system properties are expected to evolve, to be understood, represented, managed, reused and analyzed. Our contribution to supporting multidimensional separation of concerns in software architecture results in the ConcernBASE approach. This general approach serves as an initial starting point for future research work. The concepts described in the previous section need to be customized and detailed for specific methodologies, projects and organizations. We see two main fields that need to be clearly considered in ConcernBASE:

- *Conceptual framework issues*: characteristics of architectural abstractions, notation and tools that influence composition and decomposition mechanisms and their combination.
- *Methodological issues*: characteristics of the integration of architectural descriptions with other software development artifacts.

REFERENCES

1. Garlan, D.: *Software Architecture: A Roadmap*. In The Future of Software Engineering; Anthony Finkelstein (Ed). 22nd International Conference on Software Engineering, ICSE 2000; University of Limerick, Ireland, 4-11 June 2000.
2. Shaw, M., Garlan, D.: *Software Architecture - Perspectives on an Emerging Discipline*. Prentice-Hall, New Jersey (1996).
3. Bass, L., Clements, P., Kazman, R.: *Software Architecture in Practice*. Addison-Wesley (1998).
4. Garlan, D., Monroe, R. T. and Wile, D.: *ACME: An Architecture Description Interchange Lan-guage*. Proceedings of CASCON '97 (1997).
5. Medvidovic, N. and Taylor, R. N.: *A Classification and Comparison Framework for Software Architecture Description Languages*. IEEE Transactions on Software Engineering, Vol. 26, No.1, January 2000.
6. IEEE Architecture Working Group: *Draft Recommended Practice for Architectural Description* (Version 5, October 1999).
7. Kandé, Mohamed M. and Strohmeier, A.: *Towards a UML Profile for Software Architecture Descriptions*. To be published in the UML'2000 Conf. Proc., Stuart Kent and Andy Evans (Eds.), LNCS (2000).
8. Kruchten, P. B: *The 4+1 view model of architecture*. IEEE Software, 12(6):42-50, (1995).
9. ISO/IEC 10746-1/2/3. *Reference Model for Open Distributed Processing – Part 1: Overview/Part2: Foundations/Part3: Archictecture*. ISO/IEC (1995).
10. Allen, R. J.: *A Formal Approach to Software Architecture*. Ph.D. Thesis, Carnegie Mellon Univer-sity, School of Computer Science, available as TR# CMU-CS-97-144, May (1997).
11. Tarr, P., Ossher, H., Harrison, W. and Sutton, S.M. Jr.: "*N Degrees of Separation: Multi-Dimensional Separation of Concerns*." Proceedings of the International Conference on Software Engineering - ICSE'99 (May 1999).
12. Ossher, H. and Tarr, P.: *Multi-Dimensional Separation of Concerns using Hyperspaces*. IBM Research Report 21452 (April 1999).
13. Ossher, H. and Tarr, P.: *Multi-Dimensional Separation of Concerns and The Hyperspace Approach*. In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer (2000). (To appear.)

14. Rumbaugh, J., et al.: *The Unified Language Modeling Reference Manual*. Addison-Wesley (1999).
15. *The Unified Modeling Language Specification*. Object Management Group (On-line at http://www.omg.org), Framingham, Mas.