

# Views and Viewpoints in Software Systems Architecture\*

Rich Hilliard

~~Integrated Systems and Internet Solutions, Inc.~~

~~Concord, Massachusetts USA~~

~~rh@isis2000.com~~

~~+1 978 318 0000~~

Current email: [r.hilliard@computer.org](mailto:r.hilliard@computer.org)

## Abstract

Although the use of *multiple views* is a virtual holy grail of software and systems engineering, its status appears less secure in the field known as Software Architecture. Yet, practicing architects need views to manage the inherent complexity of the large, software-intensive systems they specify and build. This paper begins with a brief survey of the topic from its historical origins through current usage and issues, and ends with an overview of an approach to treating views as first-class entities within architectural description with respect to their usage in architectural specification, analysis and evolution.

**Keywords:** architectural description, multiple views, viewpoints

## 1 Introduction

The notion of *multiple views* has a long history in software engineering and related fields (such as requirements engineering, data engineering and systems engineering), where views are introduced to separate concerns and therefore to control descriptive complexity.

Despite these precursors, their role is less secure in the field known as Software Architecture. This appears to be the case for a couple of reasons. First, there are no coherent foundations for their use in architecture. Second, some researchers regard their introduction as problematic because multiple views introduce problems of view integration and view consistency.

Yet, practicing software systems architects routinely deploy multiple views in the description of complex systems, albeit on an informal basis. It would

be useful if there were ways to conceptualize multiple views in a manner that (1) would meet the needs of practitioners and (2) make the problems of addressing them tractable to researchers in architectural description.

The purpose of this paper is to suggest the beginnings of a theory and practice of views suitable for software systems architecture.

## 2 A Brief History of Views

*Multiple views* have a long history in software engineering dating back to at least the 1970s in work such as Ross' Structured Analysis (henceforth RSA, or SADT) [27]. The motivation for multiple views is *separation of concerns*. Thus, views were introduced as a construct for managing the complexity of software engineering artifacts (such as requirements specifications and design models). In the earliest approaches, the multiple views of a model were based upon fixed perspectives or *viewpoints* – most first-generation software engineering techniques embodied functional and data viewpoints.<sup>1</sup>

By 1980, one sees much traffic in the definition of various views and movement away from fixed viewpoints. See [4] for a cross-disciplinary, representative survey from that time.

More recently, much work on views and viewpoints appears in the biennial *International Workshop on Software Specification and Design* (IWSSD). Recognition of the importance of these notions is reflected in the occurrence of a conference, *Viewpoints'96*, focused on this topic.

The most recent and widely available incarnation of multiple viewpoint modeling is the Unified Modeling Language (UML). The UML defines eight types of diagrams, where each diagram type has an implicit

<sup>1</sup>I use the terms “view” and “viewpoint” casually until section 4 where I will distinguish and define these terms.

\*Position paper for the First Working IFIP Conference on Software Architecture (WICSA 1), San Antonio, TX, 22–24 February 1999. It is part of a larger work on architectural views and viewpoints.

viewpoint [23]. The UML also provides extension mechanisms for allowing users to define additional diagram types (and thus, indirectly, new viewpoints).

In Requirements Engineering, there is a significant trend of work on viewpoints. For a good survey with a useful bibliography, see [9]. This work is motivated by the recognition that systems have multiple stakeholders with varying concerns; a notion reflected in Ross' earliest work on Structured Analysis [27].

We conclude this brief survey with an identification of some issues raised in the use of multiple views. These are:

1. fixed v. first-class viewpoints
2. constructive v. projective views
3. view integration

Until Ross, viewpoints were regarded as fixed items – i.e., not first-class entities. In RSA, Ross associated with each model a *model orientation* which declared the purpose, context, and viewpoint of a model. A model orientation was documented in a textual form. Although informal, this could be considered a primitive form of an *extension mechanism* for viewpoints.

Recent work in Requirements Engineering, treats viewpoints as first-class entities, with associated attributes and operations [22].

The second issue reflects two different stances toward how views are created. Taking the *constructive stance*, separate views of a subject or system are individually constructed. The overall model of the system is a synthesis of those individual views. To understand the whole model is to understand the sum of the views and their interrelationships. Under the *projective stance*, various views of interest are derived from some overall model of the system of interest.

The third issue, view integration, most usually arises as a consequence of the constructive stance: if views are individually developed, they may need to be reconciled (or, *integrated*) to insure consistency and coherence of the overall model.

I return to these issues below, in the context of architecture.

### 3 Views in Architecture

**Scope.** The architectural level of concern for software-intensive systems is the subject of much interest at present in industry and the research community [14]. I use the term *software systems architecture* to convey a wide-spectrum interpretation of architecture. This is in contrast with *software architecture*. As currently represented in the literature, software

architecture is equated with “high level design” [29] – a valuable, but narrower, scope than what is discussed here. The wider interpretation of the term is motivated by the analogy with traditional (civil, building) architecture [26].

**What’s an Architect to Do?** At the architectural level, numerous system stakeholders have various concerns for a system. Typical concerns include system implementability, its development and operating costs, data management, security, fault-tolerance, distribution, ease of migration, interoperability, and maintainability, among others. Some of these clearly fall into the realm of “high level design” while others do not, although they are no less critical for the architect to decide, record, manage and communicate to others.

Such concerns ought to be accessible to architectural specification and analysis. Indeed, various views have appeared in the literature or arisen in practice to address such concerns; one finds distinct views with similar or overlapping purposes as well as similarly named viewpoints with quite different purposes. Consequently, it is difficult to discern what a given view addresses or when two viewpoints are addressing related topics. If architecture is to progress as a discipline, it would be desirable to bring some degree of order to this chaos.

**Approaches to Views.** There are a number of existing approaches utilizing multiple views in software systems architecture. Before outlining a theory intended to encompass this range of practices, it is useful to review some of these approaches for their salient characteristics.

Mesaros explains an architectural view as: “a way of looking at an architecture. Each view may have a different concept of components and relationships” [19].

Although the notion of view is not defined there, multiple views appear in one of the earliest papers on Software Architecture, Perry and Wolf’s classic [24].

Three important views in software architecture are those of processing, data, and connections ... all three views are necessary and useful at the architectural level.

These views derive from Perry and Wolf’s selection of what constitute architectural elements; one key term of their architectural “equation”.<sup>2</sup> Perry and Wolf identify three classes of Elements:

<sup>2</sup>  $SoftwareArchitecture = \{Elements, Form, Rationale\}$

*processing elements*: “those components that supply the transformation of data elements;”

*data elements*: “those that contain the information that is used and transformed;”

*connecting elements*: those “elements (which at times may be either processing or data elements, or both) are the glue that holds the different pieces of the architecture together.”

It is unclear whether these elements are intended to constitute “different concept[s] of components and relationships” for each view or have independent existence across views.

Existing approaches to architectural views reflect a range of choices relative to the issues found at the end of the previous section.

A number of approaches are based on a predefined set of fixed views that must be created to conform to the approach. [30] presents an extensive set of constructs called *columns* which appear to correspond to views. The *Reference Model for Open Distributed Processing* defines exactly five viewpoints [15].

Sometimes a set of views is taken to be a starting point, for example, [10] states an architecture description “should be comprised of alternate views, including *at least* a behavioral/operational view, a static topological view, and a dataflow view. It is important to have formal architectural notation(s) that are capable of capturing not only these views but also other views that are concerned with other stakeholder needs.” [my emphasis]

Kruchten’s 4+1 View Model takes four views as its starting point [17]. Viewpoint integration is accomplished via a “fifth view” which is a set of scenarios used to validate the other views and their interactions.

The Practical Architecture Method [7] prescribes no particular views; instead the determination of useful views is part of the architect’s work and is driven by concerns related to the specific system.

The IEEE Architecture Working Group is developing a *Recommended Practice for Architectural Description* [14] to support the range of practices described above.

Within Software Architecture, most work has been centered on structural concerns – to paraphrase Meszaros, “on a way of looking at an architecture” in terms of its structural “components and relationships.” – in terms of components such as procedures, filters, and databases, together with connectors such as procedure calls, pipes, and shared data [29].

Within Software Architecture, the structural view is often equated with *the* architecture of the software system, as suggested by the following rationale for the design of the architecture description language Acme [20]:

[A]n examination of existing ADLs reveals that there is, in fact, considerable agreement about the role of *structure* in architectural description. ... [V]irtually all ADLs take as their starting point the need to express an architectural design as a hierarchical collection of interacting components. On top of this structural skeleton different ADLs then add various kinds of additional information, such as run-time semantics, code fragments, protocols of interaction, design rationale, resource consumption, topological invariants, and processor allocations.

Conversely, we might say Software Architecture has been focused on a single *implicit* structural view. Thus, in Shaw and Garlan’s *An Introduction to Software Architecture* [29], the desiderata for architectural description do not mention views at all (in fact, “views” does not appear in the index)! This implicit structuralism is also in evidence in surveys of ADLs such as [5, 18].

For some in Software Architecture, the problems created by introducing multiple views and their subsequent integration, is sufficiently worrisome as to lead them away from consideration of full-fledged multiple views at all. For example:

Complex specifications require structure, such as different segments for different concerns. However, different concerns also lead to different notations. ... [T]his leads to a *multiple-view problem*: different specifications describe different, but overlapping issues. [28] [my emphasis]

Researchers have accommodated stakeholders’ needs for views in various ways. The two most common approaches might be described this way:

1. Treat views as projective (rather than constructive).
2. Decorate a primary representation with attributes pertaining to other concerns (rather than separate concerns).

I have described the projective stance above. The second alternative to full, constructive views, I call the *decorative stance*. In the decorative stance, a base

representation is annotated (or, *decorated*) with additional information. Adopting this stance, one needs something to decorate – within Software Architecture this base is frequently what I referred to as the *structural view* above. As suggested by the Acme quotation above, a consequence of implicit structuralism is that much current research in Software Architecture is devoted to how, when and where to capture non-structuralist information within that structuralist paradigm.

For example, [2], investigating the composition of heterogeneous styles, characterizes a style in terms of “a collection of constraints on the structure, behavior, and resource usage of the components and connectors in a software system.” While each is “viewed” separately, note the primacy of the structural – the behavioral view depends on the structural for its vocabulary, as does the resource usage view:

The *structural view* of a style describes the components and allowed interconnections between them. All possible data and control transfers are explicitly identified. The *behavioral view* places constraints on the types and order of actions which the components and connectors may engage in. The *resource usage view* places constraints on a variety of systems integration issues such as network protocols, CPU type, etc. [my emphasis]

Other recent examples include: Abd-Allah’s work on reliability [1]; Allen’s dissertation on integrating behavioral semantics with a structural formalism [3]; and much work on the topic of “dynamic architectures.” In each case, researchers are tackling the analysis of non-structural properties by adding attributes to a structural model. While there are often strong correlates between system characteristics and system structure, these system characteristics are often better specified and analyzed in their own right. I discuss this in the next section.

## 4 Practice and Theory of Views

In this section I sketch the beginnings of a theory of views and viewpoints, applicable to architecture within the context of practical use. The theory has evolved from experience with large information systems for command and control and other domains [7].

I adopt the constructive stance described above. This stance is the hardest to support, but the mechanisms proposed are equally applicable to the projective or decorative stances.

To motivate the exposition, we set it within a typical architecture development process – other processes (architectural analysis, system migration, system reengineering) could also utilize the results. In this setting, the job of the architect is to understand client and system stakeholder needs and begin to construct an architectural description (AD) which captures the key decisions the architecture embodies to meet those needs. The AD consists of one or more views.

A *view* is a description of the system relative to a set of concerns from a certain viewpoint.

A view generalizes the notion of a model, diagram, or other form of focused representation. Instead of attempting to say everything about an architecture in a single model, a view addresses a subset of the concerns for the whole system (architecture). This subset of concerns may be oriented toward a particular class of stakeholders (e.g., maintainers, thus a maintenance view) or toward specific system characteristics which may be of interest to several types of stakeholders (e.g., a reliability view for hardware suppliers, data designers, and software developers) or perhaps from other considerations or organizing principles.

A view may be characterized as follows:

**Purpose:** the concerns the view is intended to address;

**Scope:** the boundaries of what is in the view and what is not; and

**Elements:** the elements which comprise the view and the relations among them.

Traditional practice, in which an “architecture” is typically an informal block diagram, is a degenerate case of this. One way of introducing rigor into this model is to introduce well-defined languages underlying the elements and their relations. Languages support stakeholders’ shared discourse. By introducing language mechanisms such as typing of elements, we can both improve understandability and provide a basis for automation support.

In most views, there are non-trivial rules of combination among elements; e.g., joining *pipe–pipe*, or *filter–filter* are ill-formed, whereas *filter–pipe–filter* is meaningful. Much of such information is not dependent on the particular architecture being modeled, but can be reasonably factored out. Such information constitutes a *viewpoint*, which is reusable for any number of views.

A viewpoint is a way of looking at a system.

It is therefore a pattern for making particular views, just as a programming language is a pattern for making particular programs. The notions of view and viewpoint are interrelated as follows:

*view : viewpoint :: program : programming language*

A viewpoint can be thought of as a domain-specific language. The purpose of a viewpoint is to codify a specific set of concepts and relations for talking about a particular set of concerns. As our knowledge of architecture increases we might imagine a growing palette of viewpoints available to the architect.

This notion of viewpoint has much in common with the ISO/IEC *Reference Model for Open Distributed Processing* defines viewpoint this way:

*Viewpoint* (on a system): a form of abstraction achieved using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within a system. [15]

There are various ways one could characterize viewpoints, suggested by work such as [6, 22]; I will characterize a viewpoint in terms of its syntax, semantics and usage (or, pragmatics) as follows:

**Name:** an identifying name or phrase by which the viewpoint is known;

**Addressable Concerns:** the range of concerns it can address; its domain of discourse;

**Viewpoint language:** the elements or vocabulary which can occur in a view to which the viewpoint applies;

**Construction rules:** the rules by which elements are selected and combined within a view (i.e., its syntactic or well-formedness rules);

**Interpretive rules:** the rules for interpretation of (well-formed) views conforming to the viewpoint;

**Analytic techniques:** any methods or procedures associated with the viewpoint by which resulting views may be analyzed.

The *viewpoint language* may be an informal or formal language. By adding *interpretive rules*, one obtains a modeling language.<sup>3</sup> The role of the rules of interpretation is to establish a correspondance between well-formed combinations of elements and the

<sup>3</sup>For our purposes, *M* is a *model* of a subject, *S*, if *M* can be used to answer questions about *S*. This definition dates back to Ross and Minsky, circa 1971.

subject of interest. *Analytic techniques* give us ways to operate on descriptions to yield answers to further questions about a subject. These procedures may be informal, rigorous, formal, automated, deductive, etc. Most analytical techniques will be lifted from a particular, existing notation or language, although nothing precludes defining them at this level. By adding these operations on descriptions, one obtains what Curry called a *formal system* [6].

This model may be summarized as shown in the figure below (rendered in UML).

**Properties of Views.** A view is an useful level at which to impose methodological principles about a model. E.g., views might be regarded as individual artifacts of the architect, upon which to exercise version and configuration management.

One interesting principle (from traditional architecture) is the wholeness principle: a view ought to address the whole system with respect to the concerns of interest. This principle is nearly implicit in traditional architecture, perhaps because it follows from our intuitions about space. E.g., an elevation drawing does not omit every other floor. In software systems architecture, this principle provides a rudimentary bound on completeness.

Despite the large number of imaginable views, experience and intuition suggest a fairly smaller set of recurring concerns, such as those pertaining to system structure, behavior, etc. I would not suggest that this set is closed, or even finite: new implementation technologies, and new design paradigms create new concerns (recent ones include multiple threads; event-based systems led to event and behavioral views; mobile processes, ...). Sometimes new paradigms “shut down” traditional ways of viewing a system. The object-oriented paradigm appears to have had this effect on previously separate functional and data views.

#### **Example: Defining A Structural Viewpoint**

The structural viewpoint has developed in the field of Software Architecture over the past six years and is in widespread use. This viewpoint is often implicit in contemporary Architecture Description Languages. It is implicit in the earliest papers on Software Architecture such as [24] and Garlan and Shaw [11]:

The framework we will adopt is to treat an architecture of a specific system as a collection of computational components – or simply components – together with a description of the interactions between these components – the connectors. Graphically

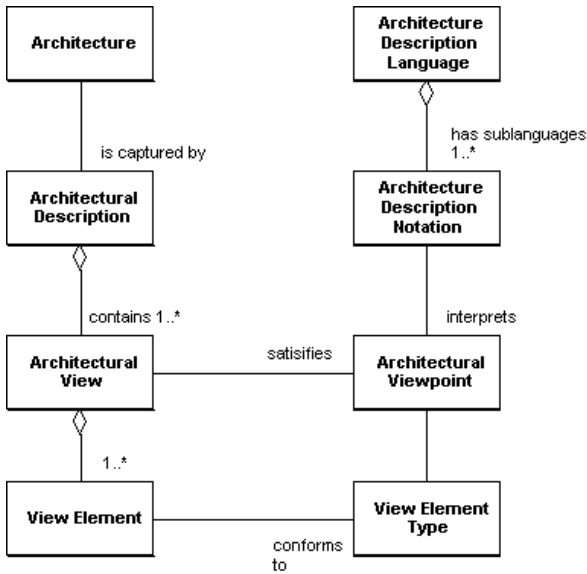


Figure 1: ADF'97 Metamodel for Architectural Description

speaking, this leads to a view of an abstract architectural description as a graph in which the nodes represent the components and the arcs represent the connectors. As we will see, connectors can represent interactions as varied as procedure call, event broadcast, database queries, and pipes.

An architectural style, then, defines a family of such systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined. These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints – say, having to do with execution semantics – might also be part of the style definition.

The structural viewpoint is characterized by the following data:

**Viewpoint Name:** Structural

**Purpose:** Define the computational elements of a system and the organization of those elements.

**Concerns:** What software elements comprise the system? What are their interfaces? How do they interconnect? What are the mechanisms for interconnection?

**Viewpoint Language:** The structural viewpoint depends on the following entities: components (individual units of computation); components have ports (interfaces); connectors (represent interconnections between components for communication and coordination); and connectors have roles (where they attach to components). Components and connectors may be typed. All of the previously listed entities may have attributes of varying kinds.

**Analytic Techniques:** The structural viewpoint supports the following kinds of checking: attachment (are connectors properly connecting components?); type consistency (are the types of components and connectors used consistent with their attachments and any style or other constraints?); and reachability. Most ADLs which support the structural viewpoint provide some kind of syntax checking of the entities above (component to component, etc.).

**A calculus of views.** Insofar as views are a construct for management of complexity of descriptions, much as modules, packages or subsystems are for software systems, we might aspire to the following:

*views : architectural description :: modules : system*

Each view is constructed as a self-contained artifact, semi-independent of the others, although inter-

related. Views are then composed or integrated to yield the overall architectural description; each distinct because their purposes differ – after all that was the whole point of separation of concerns – but overlapping because they share a common subject, the system.

To achieve this style of view construction, one needs some way to relate views on the element level. The approach we have taken is to define a basic set of primitives upon which to construct or describe any view. These primitive were documented as a part of the metamodel for the Architecture Description Framework (ADF). The metamodel represents a theory of what constitutes core architectural syntax and semantics. Each class represents a descriptive construct. Associated with each class are attributes which represent information associated with those constructs.

A view is constructed out of three primitive types of elements: components, connections, and constraints. The use of components and connections generalizes current usage in Software Architecture. It also builds upon the intuition embodied in a graph-like structure of nodes (components), edges (connections), with the addition of constraints.

The element level gives us a way to revisit the notion of a viewpoint as the reusable portion of a view. If we can characterize a viewpoint in a suitably abstract manner, we can use it as the basis for “plugging in” multiple viewpoint languages. A good example of this is the Structural Viewpoint suggested above. Many existing ADLs are “homomorphic” to this viewpoint, therefore usable for the Structural Viewpoint.

The suggested framework provides a way to formulate various problems in architectural description. One of these is a unified approach to model checking. Our definitions of view and viewpoint support various kinds of checking. E.g.,

- Is the purpose of the view consistent, or achievable given the addressable concerns?
- Does a view conform to the viewpoint language?
- If we have a well-formed view, then we may usefully apply analytic methods to it. We may use analytic techniques in various ways:
  1. Calculate properties of the architecture based on the AD. E.g., [31] provide a queuing technique for the structural viewpoint. Many techniques will arise in this way – “lifted” from existing engineering practices and applied to ADs;

2. Synthesize a design from the AD, or transform an AD to another form, e.g., to refine an AD into a possible realization [21];
3. Translate, link, or combine one or more views with new information to yield a new derived view.

Integrating multiple views is a special case of item 3 above, of particular interest. As noted, once one has multiple views, arising from semi-independent sources, one has to deal with their consistency. The viewpoints literature identifies several forms of inconsistency [8, 22]:

*Confusion* – The same concept appears with different names in distinct views.

*Conflation* – Two different concepts appear with the same names in distinct views.

*Overlap* – One view may violate the constraints of another.

To achieve a consistent set of descriptions, one must model the relations between elements of various views. Many possible relations exist and are usefully applied to architectural views. The calculus outlined above provides a basis for doing that; by introducing constructs for referring to individual elements, their types, and the viewpoints they “inhabit” one can specify a rich set of relations at the instance level, in terms of their types, or at the viewpoint level. Recent work in requirements traceability suggests an implementation approach [25, 12].

## 5 Conclusion

I have sketched the beginnings of a theory of views and their viewpoints, applicable to practical ‘architecting’ and, hopefully, suggesting a basis for further research.

The theory is based on making views first-class entities which are governed by type-like entities called viewpoints. Viewpoints are characterized in terms of a set of properties pertaining to their application, a *viewpoint language* and techniques for manipulating inscriptions in that language.

The theory sketched above was prototyped in the Architecture Description Framework (ADF) [13]. The ADF forms the basis for current work on the ARCHITECT’S STUDIO, including a XML-based document type description (DTD) for Architectural Descriptions.

The work closest in spirit to this concept of views is recent work on *aspect-oriented programming*; which

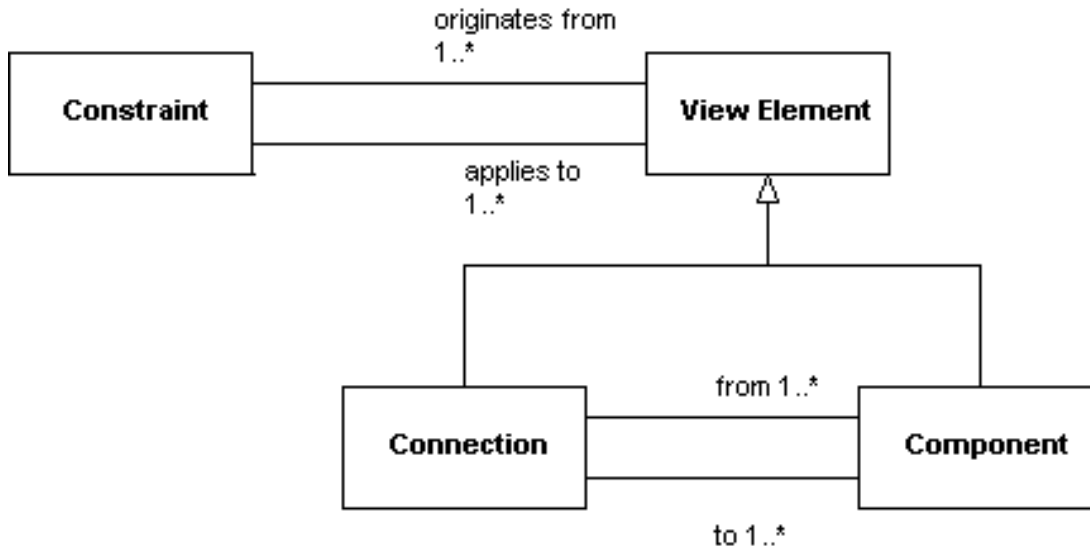


Figure 2: View Elements

begins with the observation that there are “many programming problems for which neither procedural nor object-oriented programming techniques are sufficient to clearly capture some of the important design decisions the program must implement.” These decisions are difficult to capture because “they *cross-cut* the system’s basic functionality.” Consequently, “the implementation of those design decisions [is] scattered throughout the code, resulting in ‘tangled’ code that is excessively difficult to develop and maintain.” “[A] property [of a system] that must be implemented is ... an *aspect*, if it can not be cleanly encapsulated ... Aspects tend not to be units of the system’s functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways.” [Quotations from [16].]

Aspects might be thought of as lightweight viewpoints for programming.

**Acknowledgments.** I thank the members of the IEEE Architecture Working Group for discussions of views and viewpoints: the ideas here are very much influenced by those discussions. In addition, I thank the participants in the weekly Architecture Firm meetings at MITRE (February 1996 – May 1998) for their contributions (R. Baldwin, R. Eachus, W. Farmer, M. Kokar, T. Rice, and V. Sovinsky).

## References

- [1] Ahmed Abd-Allah. Extending reliability block diagrams to software architectures. Technical Report USC-CSE-97-501, Center for Software Engineering, Computer Science Department, University of Southern California, 1997.
- [2] Ahmed Abd-Allah and Barry W. Boehm. Reasoning about the composition of heterogeneous architectures. Technical Report USC-CSE-95-503, University of Southern California, 1995.
- [3] Robert J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, May 1997. Distributed as CMU-CS-97-144.
- [4] M. L. Brodie and S. L. Zilles, editors. *Proceedings of the Workshop on data abstraction, databases, and conceptual modeling*, Published as special issue of SIGPLAN Notices 16(1) 1980.
- [5] Paul C. Clements. A survey of architecture description languages. In *Proceedings of the Eighth International Workshop on Software Specification and Design*. IEEE Computer Society Press, 1996.
- [6] Haskell B. Curry. *A theory of formal decidability*. Notre Dame, 1966.
- [7] David E. Emery, Rich Hilliard, and Timothy B. Rice. Experiences applying a practical archi-



- tectural method. In Alfred Strohmeier, editor, *Reliable Software Technologies—Ada-Europe '96*, number 1088 in Lecture Notes in Computer Science. Springer, 1996.
- [8] Gregor Engels, Reiko Heckel, Gabi Taentzer, and Hartmut Ehrig. A view-oriented approach to system modeling based on graph transformations. In Mehdi Jazayeri and Helmut Schauer, editors, *6th European Software Engineering Conference (ESEC/FSE'97)*. Springer, 1997.
- [9] A. Finkelstein and I Sommerville. The viewpoints FAQ. *Software Engineering Journal*, 11(1):2–4, 1996. Also available from <ftp://cs.ucl.ac.uk/acwf/papers/viewfaq.ps.gz>.
- [10] Cristina Gacek, Ahmed Abd-Allah, Bradford Clark, and Barry W. Boehm. On the definition of software system architecture. In *Proceedings of the First International Workshop on Architectures for Software Systems*, Seattle, WA, 1995.
- [11] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company.
- [12] O. Gotel and A. Finkelstein. Extending requirements traceability: Lessons learned from an industrial case study. In *IEEE International Symposium on Requirements Engineering*, Los Alamitos, California, January 1997. IEEE Computer Society Press.
- [13] Rich Hilliard. Representing software systems architectures or, components, connections and (why not?) first-class constraints and views. In *Joint Proceedings of the SIGSOFT '96 Workshops*, 1996.
- [14] IEEE Architecture Working Group. *Information Technology—Recommended Practice for Architectural Description (Draft 4.1)*, December 1998.
- [15] International Organization for Standardization. *ISO/IEC 10746 1–4 Open Distributed Processing – Reference Model – Parts 1–4*, July 1995. ITU Recommendation X.901–904.
- [16] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. Xerox Palo Alto Research Center, 1997.
- [17] Philippe B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 28(11):42–50, November 1995.
- [18] Nenad Medvidovic. A classification and comparison framework for software architecture description languages. Technical Report UCI-ICS-97-02, Department of Information and Computer Science, University of California, Irvine, February 1997.
- [19] Gerald Meszaros. Software architecture in BNR. In *Proceedings of the First International Workshop on Architectures for Software Systems*, 1995.
- [20] R. Monroe, D. Garlan, and D. Wile. Acme StrawManual. Available from the ACME Web site at CMU.
- [21] Mark Moriconi and Xiaolei Qian. Correctness and composition of software architectures. In *Proceedings of ACM SIGSOFT '94: Symposium on Foundations of Software Engineering*, New Orleans, LA, December 1994.
- [22] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, 1994.
- [23] Object Management Group. *Unified Modeling Language – Notation Guide (version 1.1)*, September 1997. OMG ad/97–08–05.
- [24] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), October 1992.
- [25] Francis A. C. Pinheiro and Joseph A. Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, pages 52–64, March 1996.
- [26] Eberhard Reichtin and Mark Maier. *The art of systems architecting*. CRC Press, 1996.
- [27] Douglas T. Ross. Structured Analysis (SA): a language for communicating ideas. *IEEE Transactions on Software Engineering*, SE-3(1), January 1977. Also appears in *Programming methodology : a collection of articles by members of IFIP WG2.3* edited by David Gries. New York : Springer-Verlag, 1978.

- [28] Mary Shaw and David Garlan. Formulations and formalisms in software architecture. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Development*, volume 1000 of *Lecture Notes in Computer Science*, pages 307–323. Springer-Verlag, 1996.
- [29] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an emerging discipline*. Prentice Hall, 1996.
- [30] J. F. Sowa and J. A. Zachman. Extending and formalising the framework for information systems architecture. *IBM Systems Journal*, 31(3):590–616, 1992.
- [31] Bridget Spitznagel and David Garlan. Architecture-based performance analysis. In Yi Deng and Mark Gerken, editors, *Proceedings of the 10th International Conference on Software Engineering and Knowledge Engineering*, pages 146–151. Knowledge Systems Institute, 1998.