Views as Modules*

Rich Hilliard†

February 21, 2000

Abstract

Views are a much talked about, frequently misunderstood, notion in current software architecture. Views are used to separate concerns and increase understandability of architectural descriptions, but there is a tradeoff between increased understandability and the need to integrate multiple views for consistency. This paper explores applying the metaphor *Views as modules* to this problem.

1 Introduction

Views are one of the most talked about, least understood notions in current work on software architecture. A *view* is a model which allows a stakeholder to address specific concerns about a system of interest. For a given system, there are typically multiple stakeholders, therefore multiple views are often useful. For example, an *architectural description* typically consists of a set of views to address the architectural concerns of a system.[1]

One of the tensions in the practice of architectural description is the tension between (**i**) expressivity of multiple views and (**ii**) the efficiency of establishing and maintaining consistency among those views. Branch (**i**) has been well explored: views are an essential construct for expression of architectural descriptions [7, 8, 9]. Branch (**ii**) – insuring consistency or integration of views – is an area of active investigation (see for example [3, 6]).

In this paper, I explore one approach to addressing this by using the metaphor *a view is a module*, or the analogy: *views : architectural description :: modules : program*. Read this as: *Views are to architectural descriptions as modules are to programs.* The notion of *module* has been around since the earliest days of software engineering. Modules have these characteristics: (**i**) they facilitate separation of concerns; (**ii**) they promote encapsulation; and (**iii**) module integration can defined in terms of modules' interfaces. These notions date back at least to Parnas [10].

The basic goal of modularization in programs and designs is separation of concerns. To the extent one can do this independently of other modules, each module hides or encapsulates one or more concerns that it addresses.

The analogy embodies these ideas: that views, like modules, address separate concerns; that to the extent that views can be non-interfering, those views "encapsulate" concerns; and that if we had some notion of "view interface," it might be meaningful/useful to talk about view integration via interfaces.

*Submission to the Fourth International Software Architecture Workshop (ISAW–4), 4 and 5 June 2000, Limerick, Ireland

†Integrated Systems and Internet Solutions, Inc., 150 Baker Avenue Extension, Concord, Massachusetts 01742 USA, *email:* rh@isis2000.com, *voice:* +1 978 318 0000

[1]Views are also of interest in other parts of software engineering from requirements engineering through software design. Although my interest is the use of views for architectural description, very little here is limited in applicability to architecture.

This suggests a concept of operations wherein the architect develops individual views of a system, relatively separately of other views, and then has mechanisms for integrating, or composing, them.

The remainder of the paper suggests one realization of these ideas.

Concepts. This discussion is founded upon the conceptual framework of the draft IEEE *Recommended Practice on Architectural Description* [7]. In IEEE P1471, a *view* is a particular model of a system, addressing a well-defined set of concerns. Every view is governed by a *viewpoint*. For our present purposes, the viewpoint can be thought of the "type" of the view. Like a type, it defines a predicate on the well-formedness of a view. In P1471, there is a way to *declare* (define and document) viewpoints prior to their use in constructing particular views. A viewpoint declaration specifies: concerns to be addressed by the view; the modeling vocabulary or viewpoint language to be used in its representation; and any associated analytic techniques (ATs) or methods which can be applied to the resulting representation.

2 Elaboration of the Metaphor

In this section we pursue the metaphor suggested above within the framework outlined.

Checking. Just as we unit test modules before we expend time and resources trying to integrate them, we would like to be able to maximize the stand-alone checking of views prior to attempting to integrate them into a complete architectural description. This is facilitated by the notion of *viewpoint* introduced above, as follows. A view is *well-formed* if (**i**) it addresses the concerns of its governing viewpoint; (**ii**) it conforms to the viewpoint language of its governing viewpoint; and, (**iii**) any applicable ATs have been applied. If the view is well-formed, then it is a candidate for integration.

Encapsulation. The notion of encapsulation in programming and design is based on a luxury of linguistic expression we do not necessarily have access to in generalized models such as views. In design, we encapsulate features like data structure, state, algorithm. We *export* features like invocation protocol, possible exceptions, etc. Are there principles of what makes sense to encapsulate/export for generalized models such as views? Consider a reliability view of a system. It makes decisions about many aspects of the system (processing, data structures, communications, error handling) to address a set of reliability concerns. Potentially, each of these aspects may impact other views. Is there anything to encapsulate? It seems at best we encapsulate by omission – anything not explicitly exported is encapsulated. Therefore, the interfaces of a view must be negotiated with other views, just as module interfaces are negotiated during system development.

To the extent that one view is independent of other views within an architectural description, that view *encapsulates* certain decisions. That is, it should not be affected by changes to other views.

Integration. As noted above, one of the tensions in using multiple views is inconsistencies or mismatches. Egyed and Gacek write: "Mismatches often occur because the subsystems have different characteristics for some particular features." [2]

To the extent that we can minimize the overlap of features dealt with in different views, we can minimize mismatches. There are two ways to do this: (1) attempt to make views orthogonal; (2) encapsulate decisions such

that they do not effect other views. By introducing the notion of a view's interface, we can potentially limit the computational overhead of calculating such mismatches.

In fact, we might *define* the interface of a view in this way – it is only those elements which participate in constraints with other views. If we can articulate these, then we can check them, and build the means to enforce them. Ideally, one would like to exploit "type information" – the viewpoint declaration – as the basis for interfaces. Unfortunately, this doesn't get us everything. Sometimes there are instance-level constraints that must be taken into account. In previous work, the following levels have been encountered:

- *Viewpoint-Level*: $\forall p : Process$, $\exists n : Node$ such that p **link** c (all processes must be assigned to run on a node).
- *View-Level*: everything in view V must be mapped to something in view W.
- *Element-level*: p_{123} **link** N_{839} (process p_{123} is assigned to run on node N_{839}).

The following is a very compressed example (due to length limitations) of what I have in mind.

2.1 Example of View Integration using Modules

Consider an architectural description for an web-based, "ecommerce" system consisting of three views – developed using three viewpoints defined as follows:

Capability Viewpoint. *Concerns:* What is the structural organization of the system? How do its components interact?

Viewpoint Language: The Capability Viewpoint captures **components**, **connectors** and their **dependencies**; and the **interfaces** of those components and connectors (sometimes called ports and roles, respectively). This viewpoint is widely studied in the software architecture literature (although under that name). ACME (CMU, ISI) is a cannonical implementation of the Capability Viewpoint.

Commerce Viewpoint. *Concerns:* How does the system enable commercial activities? What business models does it support? What transactions? What services are provided? Who participates in transactions, and what are their roles? How does one interact with system to conduct business?

Viewpoint Language: The primary elements of the Commerce Viewpoint are **actors** (participants in exchanges such as buyers, sellers, customers, intermediaries); **values** (resources such as products, services, money); and **value exchanges** (i.e., transactions of value among actors). These elements are inspired by [5]. A notation like UML collaboration diagrams could be appropriated to represent models in this viewpoint.

Trust Viewpoint. *Concerns:* What security and privacy principles are enforced? How are they implemented?

Viewpoint language: The elements of the Security Viewpoint include **objects** (the resources being protected), **subjects** (entities which act on objects), **policies** (rules by which subjects behave, relative to objects), and **trust domains** (ensembles of subjects and objects which interact at the same level of trust).

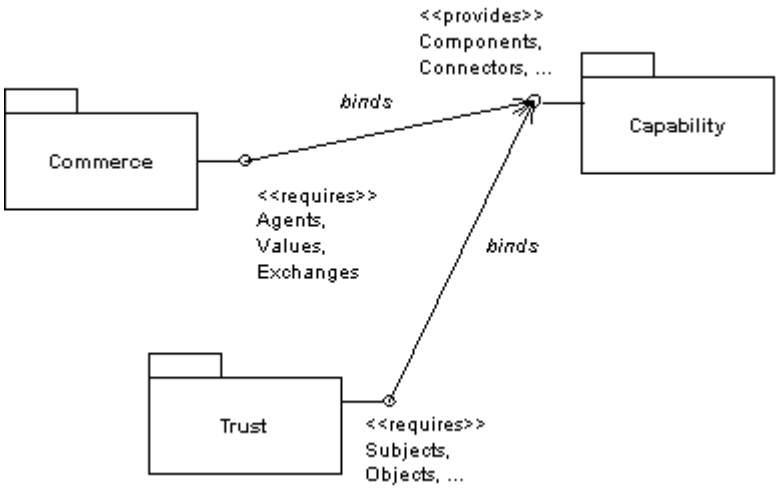

Figure 1: View Integration using Modules

View Integration.

> "... structuring a large collection of modules to form a 'system' is an essentially distinct and different intellectual activity from that of constructing the individual modules" F. DeRemer and H. H. Kron. [1]

Figure 1 depicts the architectural description in terms of three views, their interfaces and some dependencies between them. (Other relations, such as between the Trust and Commerce view are not discussed due to space limitations.) There are two kinds of interfaces: a **provides** interface exports viewpoint elements to be used by other views. A **requires** interface identifies formal parameters of the view which must be supplied by another view. The Capability View provides a set of components, connectors, and interfaces. The Commerce View requires that actors and values be instantiated from outside the view.

An interface of either kind may have an *interface theory* (**cf.** Goguen's requirement theories). For a **requires** interface, the interface theory states conditions on what may be used to instantiate the interface. For a **provides** interface, the interface theory states conditions on the ways the exported elements may be used.

Here I give a few examples (in informal text) of interface theories:

- The Trust View partitions classes into subjects and objects. Any operations owned by a component instantiating a subject or object must be compatible with its trust role.
- The Trust View partitions components into trust domains. The components and connectors used to instantiate a domain must be accessible (navigable) to one another.
- Operations on components instantiating subjects within the Trust View must not violate the specified levels of trust.

3 Conclusion

Use of multiple views in architectural description, and in other branches of software engineering, suffers the conflicting tensions of expressivity and consistency. I have outlined an approach to dealing with the latter, using the metaphor, *views as modules*, wherein views are separately constructed to address stakeholders' concerns, and then integrated, to eliminate mismatches and inconsistency, by defining a view's interfaces.

Egyed and Gacek argue for the similarity of view integration and component integration, relative to detecting feature mismatch [2]. The present position paper makes the further claim, that by introducing view interface we can further relate the two notions and reduce the complexity of view integration.

Fradet, Le Métayer and Périn demonstrate an approach to consistency checking of multiple view architectural descriptions based on treating views as typed graphs, annotated with constraints. They present a constraint-checking algorithm for checking consistency [4].

I wish to thank Alexander Egyed (USC) for helpful comments on an earlier version of this position paper.

References

[1] F. DeRemer and H. H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2:80–86, June 1976.

[2] A. Egyed and C. Gacek. Automatically detecting mismatches during component-based and model-based development. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, pages 191–198, Cocoa Beach, Florida, October 1999.

[3] Alexander Egyed. Automatically detecting modeling mismatches between heterogeneous views. Submitted to *22nd International Conference on Software Engineering*, Limerick, Ireland 2000.

[4] Pascal Fradet, Daniel Le Métayer, and Michaël Périn. Consistency checking for multiple view software architectures. In *Proceedings ESEC/FSE'99*, 1999.

[5] J. Gordijn and H van Vliet. On the interaction between business models and software architecture in electronic commerce. Case study presented at ESEC/FSE'99.

[6] Rich Hilliard. Views and viewpoints in software systems architecture. Position paper from the *First Working IFIP Conference on Software Architecture*, San Antonio, 1999.

[7] IEEE Architecture Working Group. *IEEE P1471/D5.2 Draft Recommended Practice for Architectural Description*, December 1999.

[8] International Organization for Standardization. *ISO/IEC 10746 1–4 Open Distributed Processing—Reference Model (Parts 1–4)*, July 1995. ITU Recommendation X.901–904.

[9] Philippe B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 28(11):42–50, November 1995.

[10] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15:1053–58, December 1972.