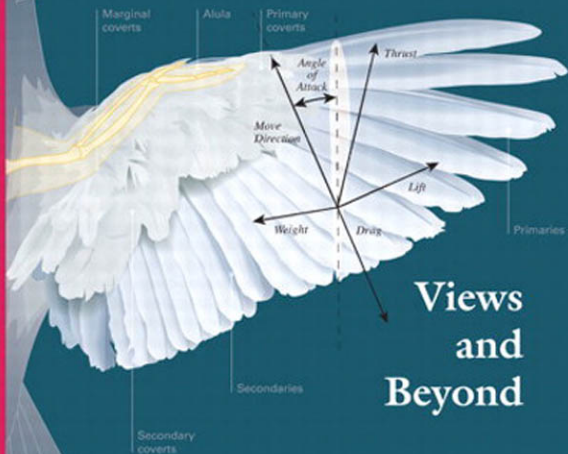




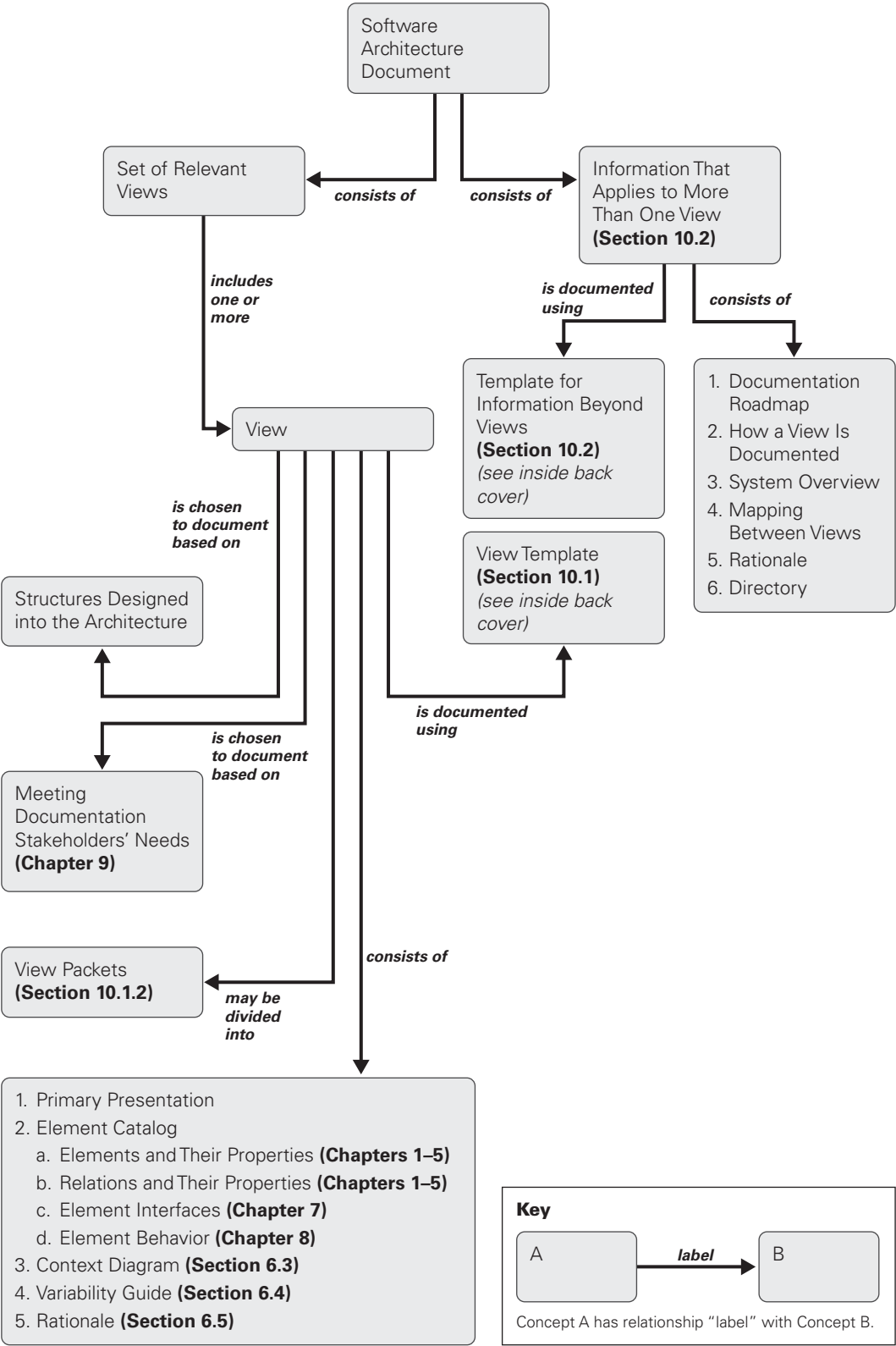
# Documenting Software Architectures



Views  
and  
Beyond

SECOND EDITION

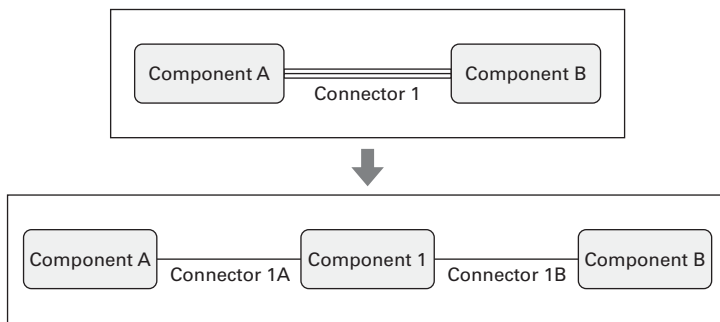
Paul Clements • Felix Bachmann • Len Bass  
David Garlan • James Ivers • Reed Little  
Paulo Merson • Robert Nord • Judith Stafford



## PERSPECTIVES

**Are Complex Connectors Necessary?**

In this book we treat connectors as first-class design elements for documenting runtime-oriented views: Connectors can represent complex abstractions; they have types and interfaces, or roles; and they require detailed semantic documentation. But couldn't one simply use a mediating component for a complex connector? For example, in Figure 3.4, the complex connector Connector 1 gets replaced by the component Component 1 and two (presumably) simpler connectors. For instance, Connector 1 might be a pipe that implements buffered data flow between components. On the other hand, Component 1 might be a buffer, and Connector 1A and Connector 1B might be simple procedure calls to read or write to the buffer.

**Figure 3.4**

A complex connector and the alternative of representing it as a component with two simpler connectors

In other words, are complex connectors needed? The answer is yes. Here's why.

First, complex connectors are rarely realizable as a single mediating component. Although most connector mechanisms do involve runtime infrastructure that carries out the communication, that is not the only thing involved. In addition, a connector implementation requires initialization and finalization code; special treatment in the components that use the connector, such as using certain kinds of libraries; global operating system settings, such as registry entries; and others.

Second, use of complex connector abstractions often supports analysis. For example, reasoning about a data flow system is greatly enhanced if the connectors are pipes rather than procedure calls or another mechanism, because well-understood calculi are available for analyzing the behavior of data flow graphs. Additionally, allowing complex connectors provides a single home where one can talk about their semantics. For example, in Figure 3.4, I could attach a single description of the protocol of interaction to the complex connector. In contrast, the lower model would require me to combine the descriptions of two connectors and a component to explain what is going on.

Third, using complex connectors helps convey an architect's design intent. When components are used to represent complex connectors, it is often no longer clear which components in a diagram are essential to the application-specific computation and which are part of the mediating communication infrastructure.

Fourth, complex connector abstractions can significantly reduce clutter in an architecture model. Few would argue that the lower of the two diagrams in Figure 3.4 is easier to understand. Magnify this many times in a more complex diagram, and it becomes obvious that clarity is served by using connectors to encapsulate details of interaction.

—D.G.



It's a good idea to provide comprehensive behavior documentation for each component (or component type). Each such model documents the possible behaviors of a component. When combined with the topological information in a C&C view, you can trace possible behaviors throughout the system, rather than just within a component.

### 3.3 What C&C Views Are For

Component-and-connector views are commonly used to show developers and other stakeholders how the system works. The C&C views (with associated behavior documentation) specify the structure and behavior of the runtime elements. In particular, these views allow you to answer questions, such as the following:

- What are the system's principal executing components, and how do they interact?
- What are the principal shared-data stores?
- Which parts of the system are replicated, and how many times?
- How does data progress through a system as it executes?
- What protocols of interaction are used by communicating entities?
- What parts of the system run in parallel?
- How can the system's structure change as it executes?

Component-and-connector views are also used to reason about runtime system quality attributes, such as performance, reliability, and availability. In particular, a well-documented view allows architects to predict overall system properties, given estimates or measurements of properties of the individual elements and interactions. For example, to determine whether a system can meet its real-time scheduling requirements, you usually need to know the execution time of each process component (among other things). Timing behavior such as this would be represented as properties of the elements. Similarly, documenting the reliability of individual elements and communication channels supports an architect when estimating or calculating overall system reliability. In some cases,

analyses such as these are supported by formal, analytical models and tools. In others, it is achieved by judicious use of rules of thumb and past experience.

## PERSPECTIVES

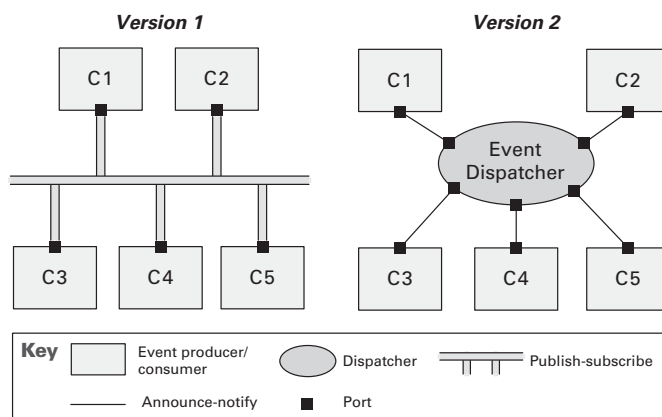
### Choosing Connector Abstractions

If you've committed to a particular C&C style, then the types of connectors to use in documenting a C&C view are already prescribed. But in other cases the architect has some freedom to determine what kinds of connectors to use and how to represent them in documentation. This choice often revolves around how much implementation structure to expose. On the one hand, a connector might be used to encapsulate a complex interaction as a single abstraction. On the other hand, a complex form of interaction can be represented as a set of components and connectors that implement it.

To illustrate, consider two ways of documenting a publish-subscribe system shown in Figure 3.5. The first version shows five components communicating through an event bus, which describes an interaction that ensures that each published event is delivered to all subscribers of that event. The second version shows the same five components communicating with the assistance of a centralized dispatcher component responsible for distributing events via procedure calls to the other components.



The publish-subscribe style is described in Section 4.4.1.



**Figure 3.5** Two potential versions of a publish-subscribe system. In Version 1, all communication takes place over an event bus; in Version 2, communication occurs with the assistance of a dispatcher component.



Refinement is discussed in Section 6.1.

There are several advantages to using the first approach:

- It simplifies the description, since there are fewer elements in the view.
- It clearly distinguishes the parts of the architecture that are used for interaction (the connectors) and the parts that are used to provide the computational functions of the system (the components).
- It permits a variety of implementations to be used to effect the event-based interactions. For instance, instead of a single dispatcher, there could be several, or alternatively each component could be responsible for sending its events to the required listeners.
- It provides a natural way to decompose documentation into multiple views, where the specific implementation would be represented in its own view as a refinement of the event bus connector.

On the other hand, the second approach has some advantages:

- It clearly indicates what kinds of mechanisms are being used to carry out event announcement.
- It may better support reasoning about runtime properties, such as delays, order guarantees, and so on, where knowledge of the specific mechanisms for dispatch is needed.
- It fits with what your chosen notation allows: For instance, because UML does not provide a way to represent rich connectors, we are forced to adopt the second approach.

Thus the choice of connector abstraction will depend on taste, needs for analysis, and the amount of implementation detail known to the architect when the architecture is documented. In practice, however, documentation usually errs on the side of putting in too much detail, using low-level communication mechanisms and additional components instead of defining the higher-level interaction abstractions that they represent.

—D.G.

## 3.4 Notations for C&C Views

### 3.4.1 Informal Notations

As always, box-and-line drawings are available to represent C&C views. Figure 3.1 is an example of a C&C diagram that uses an informal notation (explained in the diagram's notation key). Although informal notations can convey limited semantics, following some guidelines can lend rigor and depth to the descriptions. The primary guideline is to assign each component type and each connector type a separate visual form (symbol), and to list each of the types in a key.

Beyond just naming the types, however, their meaning should be specified. For example, Figure 3.1 shows a connector of type Publish-Subscribe, but the diagram does not show the connector's capacity, the type of data it can transmit, whether or not delivery is guaranteed, or a host of other important considerations. These details can be documented in the style guide in which the type is defined, or as properties in the C&C view's element catalog.

Take special care with connectors. A common source of ambiguity in most existing architecture documents is the meaning of connectors, especially ones that use arrows as their visual symbol. Make sure to say what the arrow's direction means.

### 3.4.2 Formal Notations

Most, if not all, architecture description languages (ADLs) can be used to describe component-and-connector types, constraints on topologies of component-and-connector graphs, and properties that can be associated with the elements of the graph. Tools may then process an architecture description by referring to the meanings of the types, the constraints, and the properties. For example, some ADL-associated tools can tell you if a set of processes can be scheduled so that, given the resources of the CPU, they will all meet their processing deadlines.

### 3.4.3 Semiformal Notations: UML

This section introduces some basic UML modeling constructs for representing components and connectors. Appendix A goes into more depth about using UML to represent other facets of architecture.

#### Components in UML

UML components are a good semantic match to C&C components because they permit intuitive documentation of important




Element catalogs document the architecture elements that appear in a view. They are discussed in Section 10.1.




See “Perspectives: Quivering at Arrows” on page 41, in the prologue.



Consider the following criteria if selecting an ADL: How standardized is it? What analysis or code generation does it enable? Does it lend itself only to representing certain styles, and if so, are those styles the ones you need for your architecture? Will it let you represent all of the views of the architecture that you need? Is it extensible? How robust are its tools? Is it commercially supported? Is there a large and active user community with whom you can interact?

  
Section 3.2.2 discusses types and instances of components and connectors.

  
The element catalog of an architecture view provides information about the elements in that view. Element catalogs are described in Section 10.1.

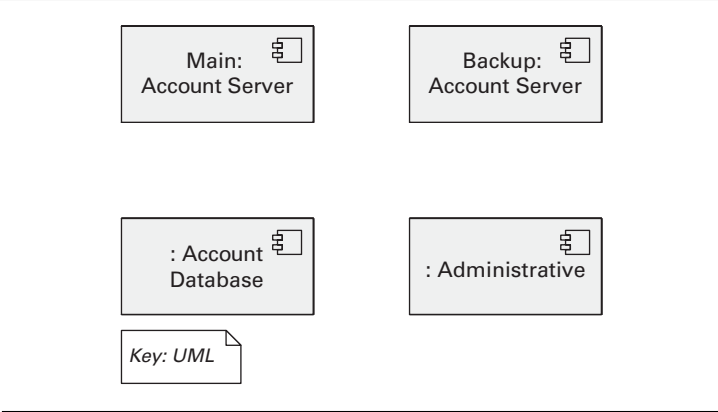
information such as interfaces, properties, and behavioral descriptions. UML components also distinguish between component types and component instances, which is useful when defining view-specific component types.

Because C&C components that appear in a view are instances, they should be represented using UML component instances, as shown in Figure 3.6. The visual distinction between UML component types and instances is found in the naming convention. Names that do not include a colon (:) are types; names that include a colon are instances, with the instance name appearing to the left of the colon. Anonymous instances, such as the instance of Account Database in Figure 3.6, are shown by starting the name with a colon.

You can define a component type in a UML diagram in a style guide you’re writing or in a view’s element catalog for a view-specific type. You should specify attributes common to all instances on the component type. If creating a view-specific type, you should link the type definition to a type defined in your style guide, such as by placing a stereotype on the type definition, as shown in Figure 3.7.

UML ports are a good semantic match to C&C ports. A UML port can be decorated with a multiplicity, as shown in the left portion of Figure 3.8, though this is typically done only on component types. The number of ports on component instances, as shown in the right portion of Figure 3.8, is typi-

**Figure 3.6**  
A UML representation of a portion of the C&C view originally presented in Figure 3.1. This fragment only shows how four components are represented in UML. Main and Backup are instances of the same component type (Account Server).



**Figure 3.7**  
A UML representation of a C&C component type. The Account Server component type is a specialization of the Server component type from the client-server style (see Section 4.3.1).

