8-2002

A Practical Method for Documenting Software Architectures

Paul C. Clements
Carnegie Mellon University

Felix Bachmann
Carnegie Mellon University

Len Bass
Carnegie Mellon University

David Garlan
Carnegie Mellon University

James Ivers
Carnegie Mellon University

See next page for additional authors

A Practical Method for Documenting Software Architectures

Paul Clements[*], Felix Bachmann*, Len Bass*, David Garlan[**],
James Ivers*, Reed Little*, Robert Nord*, Judith Stafford*
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213 USA
{clements, fb, ljb, jivers, little, rn, jas}@sei.cmu.edu, garlan@cs.cmu.edu

Abstract

A practical approach for documenting software architectures is presented. The approach is based on the well-known architectural concept of views, and holds that documentation consists of documenting the relevant views and then documenting the information that applies to more than one view. Views can be usefully grouped into viewtypes, corresponding to the three broad ways an architect must think about a system: as a set of implementation units, as a set of runtime elements interacting to carry out the system's work, and as a set of elements existing in and relating to external structures in its environment. A simple three-step procedure for choosing the relevant views to document is given, and applied to the problem of documentation for a large, complex NASA system.

1. Introduction

Software architecture has emerged as a foundational concept for the successful development of large, complex systems. Signs that the field is maturing to become an engineering discipline include textbooks on the subject such as those by Bass et al. [1], Bosch [2], and others; the work in repeatable design, exemplified by architectural styles and patterns catalogued by Buschmann et al. [3] and others; robust methods for evaluating software architectures such as ATAM [5]; international conferences such as WICSA devoted to it; and recognition of architecture as a professional practice.

Because architectures are intellectual constructs of enduring and long-lived importance, communicating an architecture to its stakeholders becomes as important a job as creating it in the first place. If the architecture cannot be understood so that others can build systems from it, analyze it, maintain it, and learn from it, then the effort put into crafting it will by and large have been wasted. Therefore, attention is now being paid to how architectural information should be captured in enduring and useful artifacts – that is, how architectures should be documented. Towards this end, the Unified Modeling Language (UML) has become a widely used notation for expressing architectural constructs (although informal box-and-line sketches communicated on viewgraphs may still be the most popular form of architectural expression). To help bring some discipline to the engineering world with respect to documenting architectures, a working group created what became ANSI/IEEE-1471-2000 ("IEEE 1471"), a recommended best practice for documenting the architectures of software intensive systems [6].

Even while benefiting from this welcome attention, however, practitioners are still left with a void. UML and box-and-line charts provide notational approaches but do not help to convey the wealth of supplementary information necessary for someone to understand an architecture. IEEE 1471 provides a philosophical foundation and a small number of guidelines but does not prescribe how to construct a usable documentation package.

Three years ago, researchers at the Software Engineering Institute and the Carnegie Mellon School of Computer Science set out to answer the question: "How should you document an architecture so that others can successfully use it, maintain it, and build a system from it?" The result of that work, summarized in this paper; is an approach we loosely call "views and beyond." The approach is more fully described elsewhere [4], where we also provide a fully-worked-out example of applying the approach to a National Aeronautics and Space Administration (NASA) system called ECS. ECS is a multi-million SLOC system that warehouses and processes enormous amounts of earth-observing satellite data in real-time, 24 hours a day, and makes that data available in a wide variety of useful forms to the earth sciences community worldwide.

---

[*] This author is with the Software Engineering Institute at Carnegie Mellon University.
[**] This author is with the School of Computer Science at Carnegie Mellon University.

2. Views and Beyond

Modern software architecture practice embraces the concept of architectural *views*. A view is a representation of a set of system elements and relations associated with them. Views are representations of the many system structures that are present simultaneously in software systems. Modern systems are more than complex enough to make it difficult to grasp them all at once. Instead, we restrict our attention at any one moment to one (or a small number) of the software system's structures, which we represent as views. Different views often carry different burdens with respect to quality attributes of the system. For instance, a view that shows a system structured as layers can be used to engineer certain kinds of modifiability (such as platform portability) into the system, and can also be used to communicate that aspect of the system's design to others. By contrast, a view showing how the same system is structured as a set of communicating processes can be used to engineer certain kinds of performance into the system, and also be used to communicate that aspect of the system's design to others.

Some authors prescribe a fixed set of views with which to engineer and communicate an architecture. Rational's Unified Process, for example, is based on Kruchten's 4+1 view approach to software [8]. The Siemens Four Views model [7] is another example. A recent trend, however, is to recognize that architects should produce whatever views are useful for the system at hand. IEEE 1471 exemplifies this philosophy; it holds that an architecture description consists of a set of views, each of which conforms to a *viewpoint*, which in turn is a realization of the concerns of one or more stakeholders.

This philosophy about views leads to the fundamental principle of the views-and-beyond approach:

> Documenting an architecture is a matter of documenting the relevant views, and then adding documentation that applies to more than one view.

3. Viewtypes, Styles, and Views

What views are available, from which the views relevant to a system can be chosen? Plenty; in fact, too many. To lend some order to an otherwise-chaotic collection of possible views, we find it extremely helpful to think about views in groups, according to the kind of information they carry. Architects carry out their creative task by thinking about the system in three different ways at once:

- How is the system to be structured as a set of code units?
- How is the system to be structured as a set of interacting run-time elements?
- How is the system to relate to non-software structures in its environment?

Thinking about views along the lines of these three broad categories helps an architect think in naturally structured terms about the system, and helps consumers of documentation discriminate among the separate concerns that an architecture manifests. We call the categories *viewtypes*. The three viewtypes are:

Module viewtype. In views belonging to the module viewtype, the elements are modules, which are units of implementation. Modules represent a code-based way of considering the system. Modules are assigned areas of functional responsibility, and are assigned to teams for implementation. There is less emphasis on how the resulting software manifests itself at runtime. Relations among modules shown in module views include is-a, is-part-of, and depends-on. Module views allow us to answer questions such as: What is the primary functional responsibility assigned to each module? What other software elements is a module allowed to use? What other software does it actually use? What modules are related to other modules by generalization or specialization (i.e., inheritance) relationships?

Component-and-connector viewtype. In views belonging to the C&C viewtype, the elements are components (which are principal units of computation) and connectors (which are the communication vehicles among components). The principle relation shown in C&C views is attachment between the components and the connectors. C&C views help answer questions such as: What are the major executing components and how do they interact? What are the major shared data stores? Which parts of the system are replicated? How does data progress through the system? What parts of the system can run in parallel? How can the system's structure change as it executes? And so forth.

Allocation viewtype. Views belonging to the allocation viewtype show the relationship between the software elements and elements in one or more external environments in which the software is created and executed. Allocation views answer questions such as: What processor does each software element executes on? In what files is each element stored during development, testing, and system building? What is the assignment of the software element to development teams?

Even within the confines of a viewtype, elements and relation can be specialized in known ways, resulting in *styles*. Styles represent known design approaches to architectures. In the C&C viewtype, many styles are well-known. By restricting the components to interact via a client-server request-reply connector, and by restricting the communication paths among the elements, *a client-server style* emerges. Or, by restricting the components to

be data repositories and data accessors that communicate via connectors that provide the appropriate communication mechanisms, a *shared-data style* emerges. Other well-known C&C styles *include peer-to-peer*, *pipe-and-filter*, and *communicating-processes*.

Many authors have catalogued C&C styles (see, for example, [10]). However, the other two viewtypes are just as rich with respect to styles. For example, by specializing the relation among modules to "allowed to use" and imposing a strict ordering on the relation, the well-known *layers* style emerges. If the relation is "uses", then we get a *uses* style, particularly helpful in carrying out incremental development. Specializing the relation to "is part of" and modules to elements that have functional responsibilities yields the module *decomposition* style. Employing the "is a" relation and other constraints yields a *generalization* style, the basis for inheritance relations in object-oriented systems.

The allocation viewtype can host various styles depending on how the software and environmental elements are specialized. Allocating modules to a development organization's structure produces the *work assignment* style. Allocating processes to processors defines the *deployment* style. And allocating modules to a development environment's file structure gives us the *implementation* style.

Viewtypes	Styles	Views
Module	Decomposition	Styles applied to particular systems
	Generalization	
	Uses	
	Layers	
Component-and-connector	Pipe-and-filter	
	Shared data	
	Communicating-processes	
	Peer-to-peer	
	Client-server	
Allocation	Work assignment	
	Deployment	
	Implementation	

Table 1. Styles are specializations of viewtypes and views are styles applied to a system.

Styles are documented in a style guide. The style guide tells what the elements and relations are of the style, and when the style might be chosen for use in a system. The style guide may also discuss documentation approaches that are specifically geared to that style. For example, documenting layers in UML is decidedly different from documenting a shared-data style in UML.

When a style is bound to a particular system, the result is a view.

Table 1 summarizes the relation between viewtypes, styles, and views.

4. Choosing the Views

Our fundamental principle cited in Section 3 implies that the first task for an architect is to decide which views are relevant. Our approach provides a simple three step procedure for choosing the views relevant to a particular project's needs. In concert with IEEE 1471, it is based upon determining the needs of the stakeholders. We'll illustrate the procedure by using ECS as an example.

Step 1. Produce a candidate view list.

For this step, begin by building a stakeholder/view table for your project. Enumerate the stakeholders for your project's software architecture documentation down the rows. Be as comprehensive as you can. For the columns, enumerate the views that apply to your system. Some views (such as decomposition, uses, and work assignment) apply to every system, while others (C&C views, the layered view) only apply to systems designed according to the corresponding styles.

Once you have the rows and columns defined, fill in each cell to describe how much information the stakeholder requires from the view: none, overview only, or detailed information. We encourage architects to hold a workshop with stakeholders or their representatives to begin a dialogue about what information they will need from the documentation.

The candidate view list consists of those views for which some stakeholder has a vested interest.

Table 2 shows a stakeholder/view table for ECS. Stakeholders for the ECS architecture include the current and future architect, developers, testers and integrators, and maintainers. But the size and complexity of ECS, plus the fact that it is a government system whose development is assigned to a team of contractors, add complicating factors. There is not one project manager, but several: one for the government, and one for each of the contractors. Each contractor organization has its own assigned part of the system to develop, and hence its own team of developers and testers. ECS relies heavily on COTS components, and so the people responsible for selecting COTS candidate components, qualifying them, selecting the winners, and integrating them into the system play a major role. We'll call these stakeholders "COTS engineers."

The important quality attributes for ECS begin with performance. Data must be ingested into the system to keep up with the rate at which it floods in from the satellites. Processing the raw data into more sophisticated and complex "data products" must also be done quickly

enough every day to stay ahead of the flow. Finally, requests from the science community for data and data analysis must be handled in a timely fashion. Data integrity, security, and availability round out the important list of quality attributes. These observations help us recognize that analysts concerned with these qualities are important architectural stakeholders.

ECS is a highly visible and highly funded project, which attracts oversight attention. The funding authorities require at least overview insight into the architecture to make sure the money over which they have control is being spent wisely. Finally, the science community using ECS to measure and predict global climate change have insight into how the system works so they can better set their expectations about its capabilities.

At least five component-and-connector views would be useful. ECS is primarily (1) a shared-data system. Its components interact in both (2) client-server and (3) peer-to-peer fashion. Many of those components are (4) communicating processes. And while the system is not actually built using pipes and filters, the (5) pipe-and-filter style is a very useful paradigm to provide a conceptual overview to some of the stakeholders.

In addition to the five C&C views, four module views (decomposition, uses, generalization, layered) apply to ECS, as do three allocation views (implementation, deployment, work assignment).

At this point, the candidate view list for ECS contains twelve views.

	Module views				Component-and-connector views					Allocation views		
Views / Stakeholders	DC	G	Uses	Layered	Pipe-and-filter	Shared-data	Client-server	Peer-to-peer	CP	Deploy-ment	Implementation	Work assign-ment
Current/future architect	d	d	d	d	s	d	d	d	d	d	s	s
Government project mgr.	d	o	o	s	o	s	o	o	o	s		d
Contractors' project mgrs.	s	o	s	s	o	s	s	s	o	d	s	d
Member of devel. team	d	d	d	d	o	d	d	d	d	s	s	d
Testers and integrators	s	s	d	s	o	d	d	d	s	s	d	
Maintainer	d	d	d	d	o	d	d	d	d	s	s	s
COTS engineers	d	s		d		d	d	d	s	d		d
Analyst for performance	d	s	d	s	o	d	d	d	d	d		
Analyst for data integrity	s	s	s	d	o	d	d	d	d	d		
Analyst for security	d	s	d	d	o	s	d	d	d	d	o	o
Analyst for availability	d	s	d	d				s	s	d		o
Funding agency	o				o	o				o		
Science community users	o				o	o				o		

Key: DC = decomposition view
G = generalization view
CP = communicating-processes view

d = detailed information, s = some details, o = overview information

Table 2. Stakeholders in the ECS architecture documentation and the views they would find useful.

Step 2. Combine views.

The candidate view list from Step 1 is likely to yield an impractical number of views. Step 2 winnows the list to a manageable size.

First, look for views in the table that require only overview depth, or that serve very few stakeholders. See if the stakeholders could be equally well served by another view having a stronger constituency.

Next, look for views that are good candidates to become *combined views.* A combined views is a view that shows information native to two or more separate views. Combined views are something of a paradox in architecture documentation. Combined views carefully considered tend to provide the most holistic insight about the overall design. However, experience shows that the most confusing documentation results from inadvertent or haphazard conflation of different views. A rule of thumb is that if there is a strong correspondence between the elements in two views then they are good candidates to be combined. For small and medium projects, the work assignment and implementation views are often easily overlaid with the module decomposition view. The decomposition view also pairs well with the layered and uses views. Where different parts of a system exhibit different component-

and-connector styles, the corresponding views might be easily overlaid -- that is, simply combined into a single view. Finally, the deployment view usually combines well with whatever C&C view shows the components that are allocated to hardware elements -- the communicating-processes view, for example.

Because of the large size of the ECS project and the number of different development organizations involved, the work assignment view (normally a good candidate for combination) would likely be kept separate. Similarly, because a large number of stakeholders interested in the module decomposition would not be interested in how the modules were allocated to files in the development environment, the implementation view would also be kept separate. However, neither of these views will bring unique information to, for example, an ATAM-based evaluation exercise, and so we can combine them with the module decomposition view (or dispense with them altogether).

Three of the C&C views would prove good candidates for combination. Augmenting the shared-data view with other components and connectors that interacted in client-server or peer-to-peer fashion allow those three views to become one. The pipe-and-filter view can be discarded; the shared-data view plus some key behavioral traces showing the data pipeline from satellite to scientist would provide the same intuitive overview to the less detail-oriented stakeholders.

Finally, recording uses information as a property of the decomposition view yields a combination of the decomposition and uses views.

After this step, six views remain:

- In the module viewtype, decomposition, layered, and generalization
- In the C&C viewtype, shared-data and communicating-processes
- In the allocation viewtype, deployment.

Step 3. Prioritize.

After Step 2 you should have the minimum set of views needed to serve your stakeholder community. At this point you need to decide what to do first. How you decide depends on the details specific to your project, but here are some things to consider:

- You don't have to complete one view before starting another. People can make progress with overview-level information, so a breadth-first approach is often the best.
- Some stakeholders' interests supersede others. A project manager or the management of a company with which yours is partnering often demand attention and information early and often.
- If your architecture has not yet been validated or evaluated for fitness of purpose, then documentation to support that activity merits high priority.

To let the ECS project begin to make progress requires putting contracts in place, which in turn requires coarse-grained decomposition. Thus, in our example, the higher levels of the decomposition view would likely receive the highest priority.

In ECS, the layering in the architecture is very coarse-grained and can be quickly described. Similarly, generalization occurs largely in only one of the three major subsystems, is also coarse-grained, and can also be quickly described. Hence, these two views might be given next priority because they can be quickly dispatched.

The shared-data, communicating-processes, and deployment views would follow, nailing down details of run-time interaction only hinted at by the module-based views. During this phase, the architect can see if the communicating processes map straightforwardly to components in the shared-data view, in which case those two views could also be combined.

The result gives us three "full-fledged" views (decomposition, shared-data/communicating-processes, and deployment), and two minor ones (layered, generalization) that stop at high levels.

5. Documenting a View

The unit of documentation for a view is a *view packet*, which is the smallest unit of information about the system you would ever want to give a stakeholder. View packets are a mechanism to "chunk" the information in a view into manageable pieces, because a single unit of documentation that portrayed all of the information in a view (especially for large and complex systems) would be unmanageably complex. A view packet can show information about a small portion of the system, or it can show information at a particular level of detail. For instance, the first view packet in a view might show the entire system, but with coarse-grained information. Subsequent view packets could show more detail about each element (such as its sub-structure). View packets let a stakeholder pan and tilt a "camera" of interest around the system in a view; he can zoom in or zoom out to/from elements of interest; and jump from view to view in an organized fashion.

No matter the view, the documentation for a view packet is placed into a standard organization or template comprising seven parts:

1. **Primary presentation** that shows the elements and relationships among them that populate the portion of the view shown in this view packet. The primary

presentation should contain the information you wish to convey about the system (in the vocabulary of that view) first. It should certainly include the primary elements and relations, but under some circumstances it might not include all of them. For example, you may wish to show the elements and relations that come into play during normal operation, but relegate error-handling or exceptional processing to the supporting documentation.

The primary presentation is usually graphical. If so, this presentation must be accompanied by a key that explains or points to an explanation of the notation used in the presentation.

Sometimes the primary presentation can be textual; tables, for example, often make superb primary presentations. If the primary presentation is textual instead of graphical, it still carries the obligation to present a terse summary of the most important information in the view packet. If that text is presented according to certain stylistic rules, the rules should be stated (or incorporated by reference), as the analog to the graphical notation key.

2. **Element catalog** detailing at least those elements depicted in the primary presentation, and perhaps others. For instance, if a diagram shows elements A, B, and C, then the catalog must contain entries that explain in sufficient detail what A, B, and C are, and their purposes or roles they play, rendered in the vocabulary of the view. In addition, if there are elements or relations relevant to this view packet that were omitted from the primary presentation, the catalog is where those are introduced and explained. Specific parts of the catalog include:
   - Elements and their properties. This section names each element in the view packet, and lists the values of its properties. For example, elements in a module decomposition view have the property of "responsibility" (an explanation of each module's role in the system), and elements in a process view have timing parameters (among other things) as properties.
   - Relations. Each view has a specific type of relation that it depicts among the elements in that view. Mostly these relations are shown in the primary presentation. However, if the primary presentation does not show all of the relations, or there are exceptions to what is depicted in the primary presentation, this is the place to record that information.
   - Element interfaces. The interface to an element is how it interacts with other entities. This section is where element interfaces are documented. An element might occur in more than one view packet, or even more than one view. Its interface should only appear once, however, and be cited elsewhere. Alternately, you may wish to extract all the element interfaces and publish them in a single document, in which case this entry becomes a pointer into that collection. A template for documenting element interfaces is given in [4].
   - Element behavior. Some elements have complex interactions with their environment, and for purposes of understanding or analysis it is often incumbent upon the architect to specify the element's behavior.
3. **Context diagram** showing how the system (or portion of the system) depicted in the primary presentation relates to its environment. It also shows the boundary between what information is included in the view packet, and what information is outside of the scope of the view packet.
4. **Variability guide** showing how to exercise any variation points that are a part of the architecture shown in this view packet.
5. **Architecture background** explaining why the design reflected in the view packet came to be. The goal of this section is to explain to someone why the design is as it is, and provide a convincing argument that it is sound. Architecture background includes:
   - Rationale. This explains why the design decisions reflected in the view packet were made and gives a list of rejected alternatives and why they were rejected. This will prevent future architects from pursuing dead ends in the face of required changes.
   - Analysis results. The architect should document the results of analyses that have been conducted, such as the results of performance or security analysis, or a list of what would have to change in the face of a particular kind of system modification.
   - Assumptions. The architect should document any assumptions he or she made when crafting the design. Assumptions generally fall into two categories: (i) assumptions about environment; and (ii) assumptions about need. Environmental assumptions document what the architect assumes is available in the environment that can be used by the system being designed. They also include assumptions about invariants in the environment. For example, a navigation system architect might make assumptions about the stability of the earth's geographic and/or magnetic poles.

Finally, assumptions about the environment can include assumptions about the development environment: tool suites available, or the skill levels of the implementation teams, for example. Assumptions about need are those that state why the design provided is sufficient for what's needed. For example, if a navigation system's software interface provides location information in a single geographic frame of reference, the architect is assuming that is sufficient, and that alternative frames of reference are not useful.

6. **Other information.** The precise contents of this section will vary according to the standard practices of each organization or the needs of the particular project. If the view is maintained as a separate document, you can use this section to record document information here. Or the architect might record references to specific sections of a requirements document to establish traceability. Information in this section is, strictly speaking, not architectural. Nevertheless, it is convenient to record such information alongside the architecture, and this section is provided for that purpose. In any case, use the first part of this section to detail the specific contents.
7. **Related view packets.** This section provides a pointer to the view packet's parent, siblings, and children (if any). In some cases, a view packet's children may reside in a different view, as when an element in one style (for example, a filter in a pipe-and-filter view) is decomposed into a set of elements in a different style (for example, a set of communicating processes).

We call items #2-#7 the *supporting documentation* that explains and elaborates the information in the primary presentation.

1. Primary presentation
2. Element catalog
   - Elements and their properties
   - Relations
   - Element interfaces
   - Element behavior
3. Context diagram
4. Variability guide
5. Architecture background
   - Rationale
   - Analysis results
   - Assumptions
6. Other information
7. Related view packets

Figure 1: Template for documenting a view

Figure 1 summarizes the template for documenting a view. Every view packet consists of a primary presentation, usually graphical, and supporting documentation that explains and elaborates the pictures. To underscore the complementary nature of the primary presentation with its supporting documentation, we call the graphical portion of the view packet an architectural *cartoon*. We use the definition from the world of fine art, where a cartoon is a preliminary sketch of the final work; it is meant to remind us that the picture, while getting most of the attention, is not the complete description but only a sketch of it. In fact, it may be considered merely an introduction to or a quick summary of the information provided by the supporting documentation.

6. Documenting information that applies to more than one view

The final piece of architecture documentation is the information that applies to more than one view and to the entire package. It ties together the views, and provides a holistic picture of the total design. Cross-or "beyond-view" documentation consists of just three major aspects, which we can summarize as "how-what-why":

- *How* the documentation is laid out and organized so that a stakeholder of the architecture can find the information he or she needs efficiently and reliably. This part consists of a documentation roadmap and a view template.
- *What* the architecture is. Here, the information that remains to be captured beyond the views themselves is a short system overview to ground any reader as to the purpose of the system; the way the views are related to each other; a list of elements and where they appear; and a glossary and acronym list that applies to the entire architecture.
- *Why* the architecture is the way it is: the background for the system, external constraints that have been imposed to shape the architecture in certain ways, and the rationale for coarse-grained large-scale decisions.

6.1. *How* the documentation is organized to serve a stakeholder

Every suite of architecture documentation needs an introductory piece to explain its organization to a novice stakeholder, and to help that stakeholder access the information he or she is most interested in. There are two kinds of "how" information to help an architecture

stakeholder: a documentation roadmap and a view template.

Documentation roadmap. The documentation roadmap is the reader's introduction to the information that the architect has chosen to include in the suite of documentation.

When using the documentation as a basis for communication, it is necessary for a new reader to determine where particular information can be found. When using the documentation as a basis for analysis, it is necessary to know which views contain the information necessary for a particular analysis. A roadmap will contain this information. A roadmap consists of just two sections.

1. Description of the parts. The roadmap begins with a brief description of each part of the documentation package. If desired, each entry can contain document information such as the author, location, and latest version. The major part of the roadmap describes the views that the architect has included in the package. For each view, the roadmap gives:
   - the name of the view
   - a description of the view's element types, relation types, and property types These descriptions can be found in the style guide from which the view was built. They let a reader begin to understand the kind of information that he or she can expect to see presented in the view.
   - a description of what the view is for. Again, this information can be found in the corresponding style guide. The goal is to tell a stakeholder whether or not the view is likely to contain information of interest. The information can be presented by listing the stakeholders who are likely to find the view of interest, and by listing a series of questions that can be answered by examining the view.
   - A description of language, modeling techniques, or analytical methods used in constructing the view.
2. How stakeholders might use the package. The roadmap follows with a section describing how various stakeholders might access the package to help address their concerns. This might include short scenarios, such as "A maintainer wishes to know the units of software that are likely to be changed by a proposed modification."

View template. A view template is the standard organization for a view. Section 5 forms a basis for a view template by defining the standard parts of a view document and the contents and rules for each part. The purpose of a view template is that of any standard organization: it helps a reader navigate quickly to a section of interest, and it helps a writer organize the information and establish criteria for knowing how much work is left to do.

6.2. *What* the architecture is

This part consists of a system overview, the mapping between views, the directory, and the project glossary and acronym list.

System overview. A system overview is a short prose description of what the system's function is, who its users are, and any important background or constraints. The purpose is to provide readers with a consistent mental model of the system and its purpose.

The system overview is, strictly speaking, not part of the architecture, but is indispensable for understanding the architecture. If an adequate system overview exists elsewhere, such as in the overall project documentation, then a pointer to it is sufficient.

Mapping between views. Since all of the views of an architecture describe the same system, it stands to reason that any two views will have much in common with each other. Helping a reader or other consumer of the documentation understand the relationship between views will help that reader gain a powerful insight into how the architecture works as a unified conceptual whole. Being clear about the relationship by providing mappings between views is the key to increasing understanding and decreasing confusions.

The mappings in a particular architecture are almost never so straightforward. For instance, each module may map to multiple run-time elements, such as when classes map to objects. Complications arise when the mappings are not one-to-one, or when run-time elements of the system do not exist as code elements at all, such as when they are imported at run time or incorporated at build or load time. These are relatively simple one- (or none-) to-many mappings. But in general, *parts* of the elements in one view can map to *parts* of elements in another view, and these relations must be captured.

Directory. The directory is simply an index of all of the elements, relations and properties that appear in any of the views, along with a pointer to where each one is defined and used.

Project glossary and acronym list. The glossary and acronym list defines terms unique to the system that have special meaning. These lists, if they exist as part of

the overall system or project documentation, may be given as pointers in the architecture package.

1. How the documentation is organized to serve a stakeholder
   - Documentation roadmap
     - Description of the parts
     - How stakeholders might use the package
   - View template
2. What the architecture is
   - System overview
   - Mapping between views
   - Directory
   - Project glossary and acronym list
3. Why the architecture is the way it is
   - Background, design constrains, and rationale

Figure 2. A template for documenting the information beyond views.

6.3. *Why* the architecture is the way it is

This section documents cross-view rationale; that is, it documents the reasoning behind decisions that apply to more than one view. Prime candidates for cross-view rationale include documentation of background or organizational constraints that led to decisions of system-wide import.

Figure 2 summarizes the template for documentation beyond view.

7. Relation to IEEE-1471

IEEE-1471 uses viewpoints as its fundamental organizing principle. A viewpoint is "a specification of the conventions for constructing and using a view, or "a pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis" [6]. Viewpoints are stakeholder-centric, and when applied to systems, yield views.

The views-and-beyond approach uses the three fundamental viewtypes as its organizing principle, each providing their own set of specializations (which we call styles) and which, when applied to systems, yield views. Thus, the views-and-beyond approach begins with the structures that are "native" to the system whose architecture is being documented, but since the choice of which views to produce depends upon satisfying stakeholder interests, the result (like that of IEEE-1471) is a set of stakeholder-centric views.

IEEE-1471 requires the architect to consider at least the following stakeholders: users, acquirers, developers, and maintainers. It encourages the architect to consider many others. So do we.

A full point-by-point comparison between our approach and IEEE-1471 is outside the scope of this paper. Documentation produced using the views-and-beyond approach will be compliant with IEEE-1471 provided that the documentation includes certain standard "boilerplate" information (such as date of issue, status, issuing organization, and the like). Other requirements of IEEE-1471 dovetail precisely with our approach: for example a glossary (required by us as well), the purpose or mission of the system (satisfied by our system overview section), and "known inconsistencies among its architectural views" and "analysis of consistency across all of its architectural views" (which becomes our mapping between views).

8. Experience, summary, and conclusions

This paper has presented a practical approach for documenting software architectures. It is built on the fundamental principle that the documentation task consists of documenting the relevant views, and then documenting the necessary information "beyond views." This approach makes the documentation task tractable, and goes beyond (but complies with) the view-based approach of IEEE-1471.

Adopting a view-based approach to documentation, and then following that approach with discipline, helps the architect design (and then communicate) along clean conceptual lines that are not haphazardly mixed. Readers will be able to digest the information quickly, and see how the system is structured into a set of well-separated but mutually-supporting design spaces.

Our approach frees the architect from the confines of a fixed set of views or having to choose from prescriptions that conflict with each other (e.g., [7] and [8]). The architect is free to choose exactly those views that are appropriate to the system under construction.

To help the architect make the choice, we have also provided a simple three-step procedure for choosing the relevant views for a system based on stakeholder concerns. This procedure (again, going beyond but staying in concert with IEEE-1471) uses the concept of combined views and prioritization to bring the view set into manageable size for real-world projects.

We have also provided a simple but powerful way to categorize views. Structuring views (and hence, architectural documentation) into the three broad categories defined by the module, component-and-connector, and allocation viewtypes provides a strong intellectual handle for producing architectural information, and understanding documentation produced by others. In this light, views can be seen to belong in

one of the three viewtypes, or be combinations (perhaps unintended) of views in different viewtypes or styles. The result is greater insight.

Recognizing three viewtypes let us expand previous notions of an architectural style to show that module and allocation styles are a consistent conceptual extension to run-time styles and provide a rich framework in which to make architectural decisions.

We have validated and gained experience with the approach in three ways. First, we applied the viewtype-style-view approach to six of the architectural case studies in [1]. The exercise revealed several instances where the architects had (perhaps unwittingly) conflated views that we were able to tease apart into simpler and more understandable views that had a strong relation to each other.

Second, we analyzed several existing view-prescribing approaches: Rational 4+1 [8], the views implied by the diagrams available in the Unified Modeling Language, the Siemens Four Views approach [7], RM-ODP [9], and others. The detailed comparisons are outside the scope of this paper, but we were able to map each view of each approach straightforwardly into one or more views using our approach, signaling compatibility, while at the same time augmenting those approaches with prescribe places for recording rationale, behavior, cross-view mappings, element and property definitions, and wealth of other information.

And third, this overall approach was used to document a significant subset of NASA's ECS system. The result is a 90-page package available in [4] that provided us with a litmus test for the documentation templates, the choosing-the-views procedure, and the framework of viewtypes, styles, and views. This application gives us confidence that the approach is practical and helpful.

9. References

[1] Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison Wesley, Boston, 1998.

[2] Bosch, J. *Design and Use of Software Architectures*, Addison Wesley, London, 2000.

[3] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, 1996.

[4] Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Boston, 2002.

[5] Clements, P., R. Kazman, and M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, Boston, 2001.

[6] IEEE 2000. IEEE Product No.: SH94869-TBR: Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Standard No. 1471-2000. Available at http://shop.ieee.org/store/.

[7] Hofmeister, C., R. Nord, and D. Soni. *Applied Software Architecture*, Addison-Wesley, Boston, 2000.

[8] Kruchten, P. *The Rational Unified Process: An Introduction*, Second Edition. Addison-Wesley, Boston, 2001.

[9] Putman, J. *Architecting with RM-ODP*. Prentice-Hall, 2000.

[10] Shaw, M., and P. Clements. 1997. "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems." *Proceedings of First International Computer Software and Applications Conference (COMPSAC97)*. IEEE Computer Society Press, pages 6–13. Also available at http://www2.cs.cmu.edu/afs/cs.cmu.edu/project/vit/www/ paper_abstracts/Boxology.html