Applying Viewpoints and Views to Software Architecture

Nick Rozanski
Marks and Spencer PLC
nick@rozanski.com

Eoin woods
Zuhlke Engineering Ltd
ewo@zuhlke.com

Abstract

Today's large information systems are often extremely complex, and can contain millions of lines of code, thousands of database tables, and hundreds of components, all running on dozens of computers. Such systems demand that their architects make an almost overwhelming number of decisions, including deciding on the system's functional structure, its internal and external interfaces, the information to be stored, managed and presented, the physical deployment environment, the development environment and how the system should be operated and supported.

A common approach to describing complex architectures is to use a single, heavily overloaded, all encompassing model, that is usually difficult to understand and maintain, and quickly becomes irrelevant to the task of building the system. An alternative, and generally more effective, approach is to partition the architectural description into a number of separate *views*, each addressing one aspect of the architecture, with the content of the views being standardised by the use of libraries of *viewpoints*.

This whitepaper introduces the view and viewpoint based approach, clearly defines its key concepts, explains its strengths and weaknesses, and presents outlines of an example set of viewpoints for information systems development.

The content of this whitepaper is based on the book "*Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*", by Nick Rozanski and Eoin Woods, published by Addison Wesley (2005).

Introduction

Today's large-scale software systems are among the most complex structures ever built by humans, containing millions of lines of code, thousands of database tables, and hundreds of components, all running on dozens of computers. This presents some formidable challenges to software development teams—and if these challenges aren't addressed early, systems are delivered late, over budget, or with an unacceptably poor level of quality.

In order to comprehend a complex computer system, you have to understand what each of its important parts actually do, how they work together, and how they interact with the world around them—in other words, its architecture. The most widely accepted definition of software architecture comes from work done in the Software Architecture group of the Software Engineering Institute (SEI) at Carnegie-Mellon University in Pittsburgh

> Definition: *The architecture of a software-intensive system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*

A related concept, used in this definition of software architecture, is that of an *architectural element*, or just "element". We use the term architectural element to refer to the pieces from which systems are built.

> Definition: *An architectural element (or just element) is a fundamental piece from which a system can be considered to be constructed.*

The nature of an architectural element depends very much on the type of system you are considering and the context within which you are considering its elements. Programming libraries, subsystems, deployable software units (e.g., Enterprise Java Beans and Active X controls), reusable software products (e.g., database management systems), or entire applications may form architectural elements in an information system, depending on the system being built. Architectural elements are often known informally as components or modules, but these terms are already widely used with established specific meaning. For this reason, we deliberately use the term element to avoid confusion.

Information systems are not created in a vacuum, but are created in order to meet specific needs that specific groups of people have. This observation leads us to consider another important software architecture concept, that of the *stakeholder*.

> Definition: *A stakeholder in a software architecture is a person, group, or entity with an interest in or concerns about the realization of the architecture.*

The *users* of a system are an obvious group of important stakeholders, but we should not limit our consideration of stakeholders to just this group. Software systems are not just used: They have to be *built and tested*, they have to be *operated*, they may have to be *repaired*, they are usually *enhanced*, and of course they have to be *paid for*. Each of these activities involves a number—possibly a significant number—of people in addition to the users. Each of these groups of people has its own requirements, interests, and needs to be met by the software system, and we need to consider all of these people as stakeholders. Understanding the breadth of the stakeholder community, and the role of the stakeholders in that community, is fundamental to the role of the architect in the development of a software product or system.

On a closely related note, it is also important for us to consider what we mean by a stakeholder *concern*. We define the term as follows.

> Definition: *A concern about an architecture is a requirement, an objective, an intention, or an aspiration a stakeholder has for that architecture.*

We use the term "concern" quite deliberately, because it is particularly appropriate to the process of software architecture. As you will see when you start to develop an architecture, you are engaged in a process of discovery as much as one of capture—in other words, when you first encounter your stakeholders, early in the system development lifecycle, they may not yet know precisely what their requirements are.

Many concerns will be common among stakeholders, but some concerns will be distinct and may even conflict. Resolving such conflicts in a way that leaves stakeholders satisfied can be a significant challenge for you in your role as a software architect.

The Problem of Architectural Description

When you start the daunting task of designing the architecture of a system, you will find that you have some difficult architectural questions to answer.

- What are the main functional elements of your architecture?
- How will these elements interact with one another and with the outside world?
- What information will be managed, stored, and presented?
- What physical hardware and software elements will be required to support these functional and information elements?
- What operational features and capabilities will be provided?
- What development, test, support, and training environments will be provided?

A major part of your role as an architect is to provide answers to these questions in a form that is comprehensible to the people who need to understand them. One of the ways that you will probably achieve this is to create an architectural description.

> Definition: *An architectural description (AD) is a set of artifacts that documents an architecture in a way its stakeholders can understand and demonstrates that the architecture has met their concerns.*

A description of an architecture has to present the essence of the architecture and its detail at the same time—in other words, it must provide an overall picture that summarizes the whole system, but it also must decompose into enough detail that it can be validated and the described system can be built. Achieving the right level of detail in an architectural description is a major challenge for any architect.

A common temptation—one you should strongly avoid—is to try to create an architectural description containing a single, heavily overloaded, all-encompassing model. This sort of model (and we've all seen them) will probably use a mixture of formal and informal notations to describe a number of aspects of the system on one huge sheet of paper: the functional structure, software layering, concurrency, inter-component communication, physical deployment environment, and so on. Let's see what happens when we try to use an all-encompassing model, by means of an example.

Example: Consider an airline reservation system to support a number of different transactions to book airline seats, update or cancel them, transfer them, upgrade them and so forth. The context of such a system is illustrated in Figure 1.

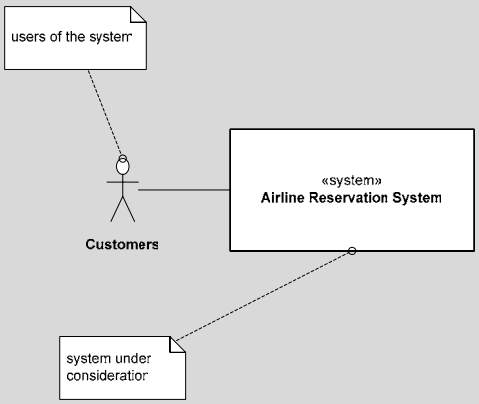

Figure 1 – Airline Reservation System Context

This system is conceptually very simple, but in practice some aspects of this system could make it very complicated indeed.

- The system's data is likely to be distributed across a number of systems in different locations.
- A number of different types of data entry devices must be supported.
- The system must be able to present some information in different languages.
- The system must be able to print tickets and other documents on a wide range of printers.
- The plethora of international regulation complicates the picture even further.

After some discussion, the architect draws up a first-cut architecture for the system, which attempts to represent all of its important aspects in a single diagram. This model includes the full range of data entry devices (including various dumb terminals, desktop PCs, and wireless devices), the multiple physical systems on which data is stored or replicated data is maintained, and some of the printing devices that must be supported (the model does not cover remote printing because it is done at a separate facility). The model is heavily annotated with text to indicate, for example, where multi-language support is required and where data must be audited, archived, or analyzed to support regulatory requirements.

However, no details of the network interfaces between the different components are included—these are abstracted out into a network icon because these are so complex. (In fact, the network design is probably the most complicated aspect of the architecture, requiring support for a number of different and largely incompatible network protocols, routing over public and private networks, synchronous and asynchronous interactions, and varying levels of service reliability and availability.) Furthermore, the model does not address any of the implications of having the same data distributed around multiple systems.

Because it is so complex and tries to address a wide mix of concerns in the same diagram, the model fails to engage any of the stakeholders. The users find it too complex and difficult to understand (particularly because of the large number of physical hardware components represented). The technology stakeholders, on the other hand, tend to disregard it because of the detail that is left out, such as the network topology. The legal team members can't use it to satisfy themselves that the regulatory aspects will be adequately handled, and the sponsor finds it completely incomprehensible.

Furthermore, the architect spends an inordinate amount of time keeping it up-to-date—every time a new type of data entry device or printer is discussed, for example, the diagram needs to be updated and reprinted on a very large sheet of paper.

Because of these problems, the diagram soon becomes obsolete and is eventually forgotten. Unfortunately, the issues that the model fails to address do not disappear and thus cause many problems and delays during the implementation and the early stages of live operation.

As we can see, this sort of description is really the worst of all worlds. Many writers on software architecture have pointed out that it simply isn't possible to describe a software architecture by using a single model. Such a model is hard to understand and is unlikely to clearly identify the architecture's most important features. It tends to poorly serve individual stakeholders because they struggle to understand the aspects that interest them. Worst of all, because of its complexity, a monolithic description is often incomplete, incorrect, or out-of-date.

> Principle: *It is not possible to capture the architecture of a complex system in a single comprehensible model that is understandable by and of value to all stakeholders*.

We need to represent complex systems in a way that is manageable and comprehensible by a range of business and technical stakeholders. A widely used approach—the only successful one we have found—is to attack the problem from different directions simultaneously. In this approach, the AD is partitioned into a number of separate but interrelated views, each of which describes a separate aspect of the architecture. Collectively, the views describe the whole system.

To help you understand what we mean by a view, let's consider the example of an architectural drawing for one of the elevations of an office block. This portrays the building from a particular aspect, typically a compass bearing such as northeast. The drawing shows features of the building that are visible from that vantage point but not from other directions. It doesn't show any details of the interior of the building (as seen by its occupants) or of its internal systems (such as plumbing or air conditioning) that influence the environment its occupants will inhabit. Thus the blueprint is only a partial representation of the building; you have to look at—and understand—the whole set of blueprints to grasp the facilities and experience that the whole building will provide.

Another way that a building architect might represent a new building is to construct a scale model of it and its environs. This shows how the building will look from all sides but again reveals nothing about its interior form or its likely internal environment.

> Strategy: *A complex system is much more effectively described by using a set of interrelated views, which collectively illustrate its functional features and quality properties and demonstrate that it meets its goals*.

Let's take a look at what this approach means for software architecture.

Architectural Views

An architectural view is a way to portray those aspects or elements of the architecture that are relevant to the concerns the view intends to address—and, by implication, the stakeholders for whom those concerns are important.

This idea is not new, going back at least as far as the work of David Parnas in the 1970s and more recently Dewayne Perry and Alexander Wolf in the early 1990s. However, it wasn't until 1995 that Phillipe Kruchten of the Rational Corporation published his widely accepted written description of viewpoints, *Architectural Blueprints—The 4+1 View Model of Software Architecture*. This suggested four different views of a system and the use of a set of scenarios (use cases) to check their correctness. Kruchten's approach has since evolved to form an important part of the Rational Unified Process (RUP).

More recently, IEEE Standard 1471 has formalized these concepts and brought some welcome standardization of terminology. In fact, our definition of a view is based on and extends the one from the IEEE standard.

> Definition: *A view is a representation of one or more structural aspects of an architecture that illustrates how the architecture addresses one or more concerns held by one or more of its stakeholders.*

When deciding what to include in a view, ask yourself the following questions.

What class(es) of stakeholder is the view aimed at? A view may be narrowly focused on one class of stakeholder or even a specific individual, or it may be aimed at a larger group whose members have varying interests and levels of expertise.

How much technical understanding do these stakeholders have? Acquirers and users, for example, will be experts in their subject areas but are unlikely to know much about hardware or software, while the converse may apply to developers or support staff.

What stakeholder concerns is the view intended to address? How much do the stakeholders know about the architectural context and background to these concerns?

How much do these stakeholders need to know about this aspect of the architecture? For non technical stakeholders such as users, how competent are they in understanding its technical details?

As with the AD itself, one of your main challenges is to get the right level of detail into your views. Provide too much detail, and your audience will be overwhelmed; too little, and you risk your audience making assumptions that may not be valid.

> Strategy: *Include in a view only the details that further the objectives of your AD—that is, those details that help explain the architecture to stakeholders or demonstrate that stakeholder concerns are being met*.

Architectural Viewpoints

It would be hard work if every time you were creating a view of your architecture you had to go back to first principles to define what should go into it. Fortunately, you don't quite have to do that.

In his introductory paper, Kruchten defined four standard views, namely, Logical, Process, Physical, and Development. The IEEE standard makes this idea generic (and does not specify one set of views or another) by proposing the concept of a *viewpoint*.

The objective of the viewpoint concept is an ambitious one—no less than making available a library of templates and patterns that can be used off the shelf to guide the creation of an architectural view that can be inserted into an AD. We define a viewpoint (again after IEEE Standard 1471) as follows.

> Definition: *A viewpoint is a collection of patterns, templates, and conventions for constructing one type of view. It defines the stakeholders whose concerns are reflected in the viewpoint and the guidelines, principles, and template models for constructing its views.*

Architectural viewpoints provide a framework for capturing reusable architectural knowledge that can be used to guide the creation of a particular type of (partial) AD.

In a relatively unstructured activity like architecture definition, the idea of the viewpoint is very appealing. If we can define a standard approach, a standard language, and even a standard metamodel for describing different aspects of a system, stakeholders can understand any AD that conforms to these standards once familiar with them.

In practice, of course, we haven't achieved this goal yet. There are no universally accepted ways to model software architectures, and every AD uses its own conventions. However, the widespread

acceptance of techniques such as entity-relationship models and of modeling languages such as UML takes us some way toward this goal.

In any case, it is extremely useful to be able to categorize views according to the types of concerns and architectural elements they present.

> Strategy: *When developing a view, whether or not you use a formally defined viewpoint, be clear in your own mind what sorts of concerns the view is addressing, what types of architectural elements it presents, and who the viewpoint is aimed at. Make sure that your stakeholders understand these as well.*

Interrelationships Between the Core Concepts

To put views and viewpoints in context, consider the conceptual model in Figure 2, which illustrates how views and viewpoints relate to the other important architectural concepts we introduced earlier.

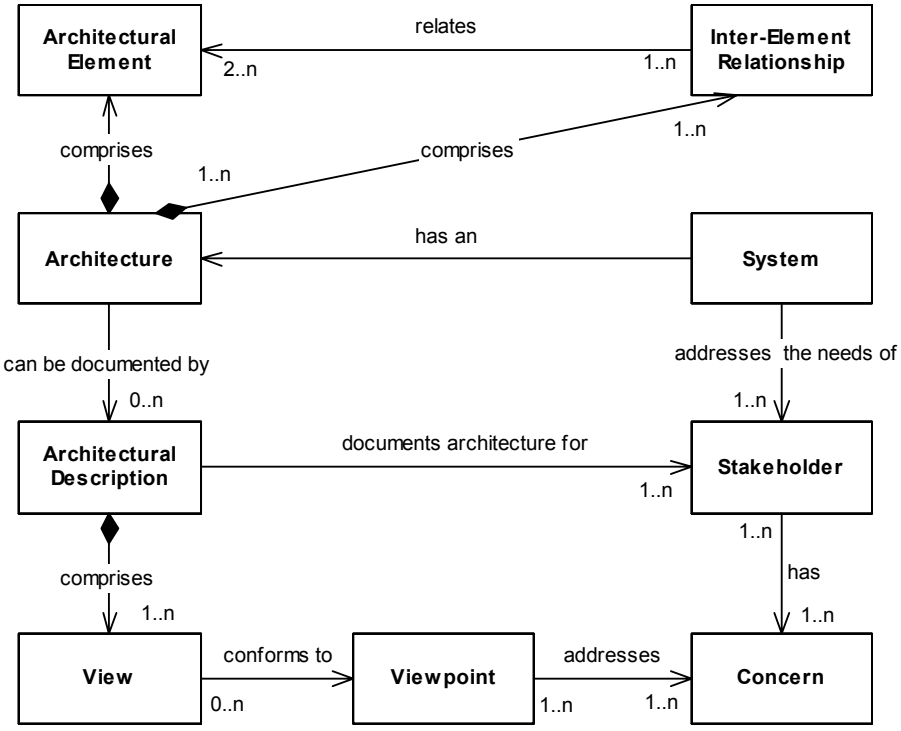

Figure 2 – Architectural Concepts and Relationships

The concepts and relationships shown in this diagram can be summarised as follows.

- A **system** is built to address the needs, concerns, goals and objectives of its **stakeholders**.
- The **architecture** of a **system** is characterized by its static and dynamic structures, and its externally-visible behavior and properties.
- The **architecture** of a **system** is comprised of a number of architectural **elements** and their inter-relationships.
- The **architecture** of a **system** can potentially be documented by an **architectural description** (fully, partly or not at all). In fact, there are many potential ADs for a given architecture, some good, some bad.
- An **architectural description** documents an architecture for its **stakeholders**, and demonstrates to them that it has met their needs.

- A **viewpoint** defines the aims, intended audience, and content of a class of **views** and defines the concerns that views of this class will address.
- A **view** conforms to a **viewpoint** and so communicates the resolution of a number of **concerns** (and a resolution of a concern may be communicated in a number of views).
- An **architectural description** comprises a number of **views**.

The Benefits of Using Viewpoints and Views

Using views and viewpoints to describe the architecture of a system benefits the architecture definition process in a number of ways.

- *Separation of concerns*: Describing many aspects of the system via a single representation can cloud communication and, more seriously, can result in independent aspects of the system becoming intertwined in the model. Separating different models of a system into distinct (but related) descriptions helps the design, analysis, and communication processes by allowing you to focus on each aspect separately.
- *Communication with stakeholder groups*: The concerns of each stakeholder group are typically quite different (e.g., contrast the primary concerns of end users, security auditors, and help-desk staff), and communicating effectively with the various stakeholder groups is quite a challenge. The viewpoint-oriented approach can help considerably with this problem. Different stakeholder groups can be guided quickly to different parts of the AD based on their particular concerns, and each view can be presented using language and notation appropriate to the knowledge, expertise, and concerns of the intended readership.
- *Management of complexity*: Dealing simultaneously with all of the aspects of a large system can result in overwhelming complexity that no one person can possibly handle. By treating each significant aspect of a system separately, the architect can focus on each in turn and so help conquer the complexity resulting from their combination.
- *Improved developer focus*: The AD is of course particularly important for the developers because they use it as the foundation of the system design. By separating out into different views those aspects of the system that are particularly important to the development team, you help ensure that the right system gets built.

Viewpoint Pitfalls

Of course, the use of views and viewpoints won't solve all of your software architecture problems automatically. Although we have found that using views is really the only way to make the problem manageable, you need to be aware of some possible pitfalls when using the view-based approach.

- *Inconsistency*: Using a number of views to describe a system inevitably brings consistency problems. It is theoretically possible to use architecture description languages to create the models in your views and then cross-check these automatically (much as graphical modelling tools attempt to check structured or object-oriented methods models), but there are no such machine-checkable architecture description languages in widespread use today. This means that achieving cross-view consistency within an AD is an inherently manual process.
- *Selection of the wrong set of views*: It is not always obvious which set of views is suitable for describing a particular system. This is influenced by a number of factors, such as the nature and complexity of the architecture, the skills and experience of the stakeholders (and of the architect), and the time available to produce the AD. There really isn't an easy answer to this problem, other than your own experience and skill and an analysis of the most important concerns that affect your architecture.

- *Fragmentation*: When you start to describe your architecture, one temptation is to create a large number of views. This can lead to your architecture being described by many independent models, each in a separate view, making the AD difficult to follow. Each separate view also involves a significant amount of effort to create and maintain. To avoid fragmentation and minimize the overhead of maintaining unnecessary descriptions, you should eliminate views that do not address significant concerns for the system you are building. In some cases, you may also consider creating hybrid views that combine models from a number of views in the viewpoint set (e.g., creating a combined deployment and concurrency view). Beware, however, of the combined views becoming difficult to understand and maintain because they address a combination of concerns.

A Viewpoint Catalogue for Information Systems

A number of viewpoint catalogues exist already, but we have found that all of them have limitations in practice, when applied to the development of architectures for large information systems. In response, we have developed a set of viewpoints for the information system architect, that build up and extend the "4+1" set, identified by Philippe Kruchten. Our catalogue contains six core viewpoints: the Functional, Information, Concurrency, Development, Deployment, and Operational viewpoints. Although the viewpoints are (largely) disjoint, we find it convenient to group them as shown in Figure 3.

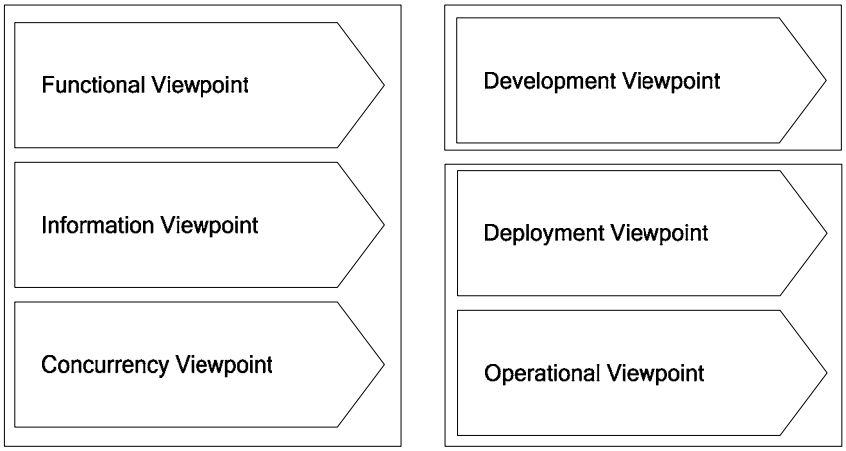

Figure 3 – Rozanski and Woods Viewpoint Groups

- The Functional, Information, and Concurrency viewpoints characterize the fundamental organization of the system.
- The Development viewpoint exists to support the system's construction.
- The Deployment and Operational viewpoints characterize the system once in its live environment.

Each of these viewpoints is briefly described in Table 1. Full descriptions of each viewpoint can be found in our book, referenced in the Further Reading section.

Viewpoint	Definition
Functional	Describes the system's functional elements, their responsibilities, interfaces, and primary interactions. A Functional view is the cornerstone of most ADs and is often the first part of the description that stakeholders try to read. It drives the shape of other system structures such as the information structure, concurrency structure, deployment structure, and so on. It also has a significant impact on the system's quality properties such as its ability to change, its ability to be secured, and its runtime performance.

Viewpoint	Definition
Information	Describes the way that the architecture stores, manipulates, manages, and distributes information. The ultimate purpose of virtually any computer system is to manipulate information in some form, and this viewpoint develops a complete but high-level view of static data structure and information flow. The objective of this analysis is to answer the big questions around content, structure, ownership, latency, references, and data migration.
Concurrency	Describes the concurrency structure of the system and maps functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently and how this is coordinated and controlled. This entails the creation of models that show the process and thread structures that the system will use and the interprocess communication mechanisms used to coordinate their operation.
Development	Describes the architecture that supports the software development process. Development views communicate the aspects of the architecture of interest to those stakeholders involved in building, testing, maintaining, and enhancing the system.
Deployment	Describes the environment into which the system will be deployed, including capturing the dependencies the system has on its runtime environment. This view captures the hardware environment that your system needs (primarily the processing nodes, network interconnections, and disk storage facilities required), the technical environment requirements for each element, and the mapping of the software elements to the runtime environment that will execute them.
Operational	Describes how the system will be operated, administered, and supported when it is running in its production environment. For all but the simplest systems, installing, managing, and operating the system is a significant task that must be considered and planned at design time. The aim of the Operational viewpoint is to identify system-wide strategies for addressing the operational concerns of the system's stakeholders and to identify solutions that address these.

Table 1 - Viewpoint Descriptions

Of course, not all of these viewpoints may apply to your architecture, and some will be more important than others. You may not need views of all of these types in your AD, and in some cases there may even be other more specialized viewpoints that you need to identify and add yourself. This means that your first job is to understand the nature of your architecture, the skills and experience of the stakeholders, and the time available and other constraints, and then to come up with an appropriate selection of views.

Summary

Capturing the essence and the detail of the whole architecture in a single model is just not possible for anything other than simple systems. If you try to do this, you will end up with a Frankenstein's monster of a model that is unmanageable and does not adequately represent the system to you or any of the stakeholders.

By far the best way of managing this complexity is to produce a number of different representations of all or part of the architecture, each of which focuses on certain aspects of the system, showing how it addresses some of the stakeholder concerns. We call these *views*.

To help you decide what views to produce and what should go into any particular view, you use *viewpoints*, which are standardized definitions of view concepts, content, and activities.

The use of views and viewpoints bring many benefits, such as separation of concerns, improved communication with stakeholders, and management of complexity. However, it is not without its pitfalls, such as inconsistency and fragmentation, and you must be careful to manage these.

In this chapter, we introduced our viewpoint catalogue, comprising the Functional, Information, Concurrency, Development, Deployment, and Operational viewpoints.

Further Reading

A lot of useful guidance on creating ADs using views (including a discussion of when and how to combine views) and thorough guidance for creating the documentation for a wide variety of types of view can be found in Clements et al. [CLEM03]. Other references that help to make sense of viewpoints and views are IEEE Standard 1471 [IEEE00] and Kruchten's "4+1" approach [KRUC95]. One of the earliest explicit references to the need for architectural views appears in Perry and Wolf [PERR92].

Some of the other viewpoint taxonomies that have been developed over the last decade or so are described in their own references, including RM-ODP [PUTM00]; the "Siemens" viewpoint set by Hofmeister, Nord, and Soni [HOFM99]; and another information systems-oriented set by Garland and Anthony [GARL03]. A workshop paper describing experiences with a number of viewpoint sets is [WOOD04].

The viewpoint set outlined in this whitepaper is fully defined in our book [ROZA05] (along with a set of complementary constructs for quality properties, that we term "perspectives").

References

[CLEM03] Clements, Paul, et al. *Documenting Software Architectures*. Boston, MA: Addison-Wesley, 2003.

[GARL03] Garland, Jeff, and Richard Anthony. *Large Scale Software Architecture*. New York: Wiley, 2003.

[HOFM00] Hofmeister, Christine, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Boston, MA: Addison-Wesley, 2000.

[IEEE00] IEEE Computer Society. *Recommended Practice for Architectural Description*. IEEE Std-1471-2000. October 9, 2000. http://standards.ieee.org/reading/ieee/std_public/description/se/1471-2000_desc.html

[KRUC95] Kruchten, Philippe. "Architectural Blueprints—The 4+1 View Model of Software Architecture." *IEEE Software*, 12(6):42–50, November 1995.

[PUTM00] Putman, Janice, *Architecting with RM-ODP*, Upper Saddle River, NJ, Prentice Hall PTR, 2000.

[ROZA05] Rozanski, Nick and Eoin Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Boston, MA: Addison-Wesley, 2005.

[PERR92] Perry, Dewayne, and Alexander Wolf. "Foundations for the Study of Software Architecture." *ACM SIGSOFT Software Engineering Notes*, 17(4), October 1992.

[WOOD04] Woods, Eoin, *Experiences Using Viewpoint for Information Systems Architecture: An Industrial Experience Report*, in F. Oquendo et. al. (Eds.): EWSA 2004, LNCS 3047, pp 182-193, 2004, Springer-Verlag, Berlin.