JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5. No. 7, September-October 2006

# On Designing Software Architectures

**Won Kim**, Samsung Electronics and SungKyunKwan University, S. Korea

**Abstract**

As the size and complexity of software has rapidly grown during the past two decades, it has become a big challenge to assure quality of software and to curb the cost of developing software. Many ways have to be brought to bear in order to meet this challenge. One of them is the design and documentation of software architecture. In this article, I discuss various aspects of designing and documenting software architecture in order to provide some practical guides to software engineers.

## 1 INTRODUCTION

Any software development process consists of eight key elements: project planning, requirements analysis, architecture design, detailed design, test planning, coding, testing, and release. The first five elements constitute the "upstream" part of the process, while the latter three elements constitute the "downstream" part. The quality of architecture design and detailed design, and the quality of documentation of architecture design and detailed design fundamentally impact the quality and reusability of software. Many books have been written on these subjects, most notably, in my opinion, [Bass et al. 2005][Clements et al. 2003]. In this article, I will review the current theory of architecture design, and provide some caveats on it and a guideline for an architecture design process.

## 2 ARCHITECTURE BASICS

The working definition of software architecture is the expression of the major elements constituting software and their layout, along with the relationships and interactions among the elements. The element may be a module, a process or a thread. A module consists of code units (functions or classes). The relationship between elements may be aggregation ("an element consisting of other elements" or "an element being a part of another element"), specialization/generalization ("an element being a kind of another element" or "an element encompassing other elements of like kind"), depends ("an element depending on the result generated by another element"), etc. Interactions among elements include data passing, control passing, communication paradigm, etc.

During the past decade, researchers in software architecture have gathered, analyzed and organized many concepts that software engineers have devised into a more coherent collection of concepts revolving around the notions of architecture views and architecture patterns. There are three basic architecture views, and each view includes a number of architectural patterns (sometimes called architectural styles). The three basic views are module view, component & connector view, and allocation view. The module view expresses the modular structure of software. It is a “static” view in that it expresses a static division of software into units. The component & connector view is a “dynamic” view, and expresses runtime entities and their interactions. Runtime entities include such “generic” ones as a process, thread, object, client, server, data store, etc., as well as domain-specific ones. Interactions among runtime entities include function calls, messages or events, data streaming, event multicast and broadcast, etc. The allocation view expresses the mapping of software or software units to the environment in which the software will run, to a software directory structure, and individual or group of software engineers who will have development responsibility for the software units.

Each view has a number of different aspects, and each aspect is expressed by an architectural pattern. In particular, the architectural patterns that express the module view include the decomposition pattern, the uses or depends (“uses/depends on the result of”) pattern, the layered pattern, the specialization/generalization pattern. There are many architectural patterns that express the component & connector view, including pipes & filters, data flow, communicating processes (also called “concurrency”), call-return, publish-subscribe, repository (also called “shared data”), peer to peer, etc. The architecture patterns that express the allocation view include the deployment pattern (mapping to the operating environment), the implementation pattern (mapping to a directory structure), and the work-assignment pattern.

I emphasize that the architecture views and patterns are needed for two different purposes. One is as aids to software architects as they think through on various aspects of the functional requirements and architecture quality attributes that they need to reflect in the architecture, subsequent detailed design and coding. For example, when a software architect has the “depends” pattern in front of him, it makes it easier, than using some other pattern, for him to think about a cluster of software modules that form a natural subset of the software that may be regarded as a unit of replacement in the future. Further, the communicating processes pattern or the repository pattern makes it easier for the architect to contemplate on such aspects of software as performance bottleneck, scheduling, reliability, etc.

A second purpose of the views and patterns is simply for documenting the architecture for review, archiving, and, of course, as a guide for developers to conduct the next steps in the development process, including detailed design, test planning, and coding.

The views are not necessarily independent of one another, in that parts of one view may be found in another view. For example, modules expressed in the decomposition pattern of the module view need to be included in, for example, the communicating processes pattern of the component & connector view. As such, often it is necessary to map one view or pattern to another. For example, a “depends” pattern may be mapped to a layered pattern – both in the module view. It is also sometimes necessary to map an architecture pattern of one view to a pattern of a different view. For example, a layered pattern of the module view may be mapped to a communicating processes pattern of the component & connector view.

## 3 CAVEATS ON ARCHITECTURE DOCUMENTATION, VIEWS AND PATTERNS

A software architecture needs to be documented as an architecture specification. An architecture specification is used for review by architects, developers, SQA/testers, project managers, and even key customers. Much is made of UML (unified modeling language) for graphically representing many aspects of software architecture. I emphasize that, while there needs to be a common set of visual notations for specifying various aspects of a software architecture, and visual notations are certainly helpful in conveying key aspects of a software architecture, in general visual notations alone are not enough to fully convey all important aspects of a software architecture. As such, proper documentation of software architectures must make use of a combination of textual descriptions and visual notations. As for the visual notations, informal notations such as boxes and lines (where a box represents a software element, and a line represents a relationship or interaction between elements) or formal notations such as UML or a combination of the two may be used, depending on the nature of software being architected, and the extent of the need for standardization.

There is an approximate agreement on the meanings and names of the architecture views and patterns. However, there are some disagreements. For example, some people regard the N-tiered client-server pattern a pattern of the component & connector view. However, some regard it as a hybrid pattern of the module view and the component & connector view. At the end of the day, such differences are not important. What is important is simply that software architects make good use of it.

The architecture views and patterns introduced in books are merely somewhat formalized versions of the views and patterns in various forms commonly found in many software architectures in practice. When situations make it necessary, software architects should feel free to create additional views and patterns. One way to create them is by combining two or more basic views or patterns.

## 4 PRINCIPLES OF GOOD SOFTWARE ARCHITECTURE AND GOOD SOFTWARE DOCUMENTATION

The architecture views and patterns are merely tools for software architects for designing and documenting software architectures. I will now discuss what it takes for a software architecture to be regarded as a good architecture.

First, a good software architecture must account for all key requirements. There are four types of requirements: user functions, non-user functions, software attributes, and architecture quality attributes. User functions are the implementation of user operations, where users may be end-users or administrators. Non-user functions are the implementation of features that are not invoked by users, such as license management, various security-related features, etc. Software attributes include such aspects as software footprint size, multi-user support, etc. Architecture quality attributes include performance, scalability, reliability, security, modifiability, extensibility, portability, etc.

Second, a good software architecture is one that anticipates future changes as much as reasonable. Future changes that should be anticipated include adding new functions and features, porting, interfacing with different third-party software or even chipsets, supporting a much larger data volume and a much larger number of simultaneous users, etc. Anticipating future changes can significantly help to reduce the scope of changes in existing software, thereby reducing development time for future upgrades.

Third, a good software architecture is one that reflects sound tradeoff considerations in all key aspects of developing complex software, including required functions, architecture quality attributes, development schedule, and available manpower and skill sets. A careful consideration of the anticipated time and people cost of developing certain features may force the software architect to decide to soften certain requirements. A performance consideration may force the architect to adopt a certain algorithm or data structure by sacrificing generality and future modifiability. A modifiability consideration, on the other hand, may force the architect to sacrifice performance and security to some extent.

Good software architecture documentation is one that reflects all key aspects of the architecture discussed above, and is easy for others to understand. To reflect all key aspects of a software architecture, a combination of textual descriptions and all three architecture views in some visual notation should be used. In my opinion, the decomposition pattern of the module view, the communicating processes pattern of the component & connector view, and the deployment pattern of the allocation view may be a reasonable minimal starting set in good software architecture documentation. Further, a software architecture specification for large complex software can itself be large and complex. As such, it is important that the principles inherent in expressing a software architecture in terms of multiple views and patterns be applied to documenting the software architecture. In other words, a software architecture specification should be organized as a collection of chunks, and each chunk in turn should be organized in a hierarchy of increasing details. Finally, a software architecture specification should employ a clear and consistent notation and format throughout. A visual description should separate out keys from the main description, thus avoiding cluttering.

## 5 A PROCESS FOR SOFTWARE ARCHITECTURE DESIGN

To the best of my knowledge, there is no universally accepted process for designing software architectures. Below, I outline an informal process as a guide to interested beginning architects and as an illustration of a few good practice ideas for designing software architectures.

1. Start with a preliminary deployment pattern of the allocation view. This will provide a picture of the operating environment for the software to be architected.
2. Work out a preliminary set of story boards on the deployment pattern. The story boards should include the flow of data and control through the operating environment, and the input, storage, and output of major types of data through the software, etc.
3. Group and prioritize the requirements. The requirements may be grouped into user functions, non-user functions, software attributes related to features to be implemented, and architecture quality attributes.
4. Select one or more architecture patterns of the module view, and reflect the requirements in successive refinements in the order of grouping and priorities determined in the previous step.
5. Select one or more architecture patterns of the component & connector view, and specify relevant requirements.
6. Select one or more architecture patterns of the allocation view, and specify relevant requirements.
7. Have the architecture design reviewed.
8. Iterate steps 1 through 7.

When iterating steps 4, 5, and 6, either the initially selected architecture patterns may be designed in greater detail, and/or additional architecture patterns may be added. Further, in step 7, care should be taken not to have all relevant people review every iteration of the architecture design. The review of a firmed-up, full design should involve as many of those who need to be involved. However, the review of rough, partial designs should involve only a small number of people, for example, another architect, a senior developer, etc.

## REFERENCES

[Bass et al. 2005] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice* (2nd Edition), Addison Wesley, 2005.

[Clements et al. 2003] Paul Clements, Felix Bachman, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, Judith Stafford. *Documenting Software Architectures: Views and Beyond*, Addison Wesley, 2003.

## About the author

**Won Kim** is Senior Advisor at Samsung Electronics and Distinguished Professor at SungKyunKwan University, Korea. He is Editor-in-Chief of ACM Transactions on Internet Technology (www.acm.org/toit). He is Global General Chair of the Human.Society@Internet International Conference. He is the recipient of the ACM 2001 Distinguished Services Award, and is an ACM Fellow.