Software Architecture Theory and Practice

Michael Tarullo, L-3 Communications

Abstract. There is often a gap between widely accepted software engineering theory and practice. This is also true for the concept of software architecture. While the concept of software architecture has been in existence for quite some time, there is still a great deal of confusion over just what software architecture actually is. Moreover, lack of a clear understanding of the concept of software architecture makes it extremely difficult to work with pragmatically. This article attempts to show how sound software architectures can be produced quite practically and documented consistently. A definition of software architecture is adopted and a model for creating software architectures by using the de-facto standard software engineering modeling tool, UML (v2.0), is introduced.

Introduction

In many fields, there is often a gap between theory and practice. Software engineering is no different. Misconceptions about software architecture, particularly by practitioners, make it difficult to communicate software architectures effectively. SEI's website [1] demonstrates the astounding diversity that exists with respect to the definition of software architecture. This website lists two modern definitions, eight classical definitions, 18 bibliographical definitions and numerous community definitions. The first three categories indicate a general agreement on the definition of the term by theoreticians and academicians. It is the wide variety of definitions held by those in the last category that is troubling, specifically because they appear to represent practitioners. And such confusion can make it difficult, if not impossible, to use the concept in a practical fashion.

This article attempts to show how sound software architectures can be produced quite practically, in a repeatable and understandable fashion, by adopting a widely held definition for the concept of software architecture, adopting a model for creating software architectures, and by using the de-facto standard software engineering modeling tool, UML (v2.0), to convey a software architecture.

Theory

Kruchten, et. al. [2] provide an excellent presentation on the history of software architecture. Their paper traces the development of software architecture theory from the time the paper was published back to its origins and before. Mary Shaw and David Garlan [3] published one of the earliest books on the subject. It is fitting that they begin their book with the question, "What is Software Architecture?"

Both the theory and practice of software architecture must be rooted in a clearly expressed and universally accepted definition of the term. What is needed is a definition that succinctly and cogently expresses the concept of software architecture. Moreover, such a definition must express the concept in such a way that it can be used practically. We turn to the myriad of definitions compiled by SEI to extract the essence of the meaning of software architecture. Many if not most of the definitions published on the SEI website have three things in common; 1) organization of a system, 2) components, and 3) relationships. While there are many other concepts conveyed, it is these three terms, or synonyms thereof, that persist throughout the definitions provided and are at the core of the theory. As a result, the definition provided by Bass, et. al. [4], that is, "The software architecture of a program or computing 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," is adopted herein since it contains each of the three common concepts cited above and complements the techniques that can be used for creating the software architectures described below. It is assumed that "structure" and "elements" in Bass's definition are synonyms for organization and components respectively, as used on SEI's website.

Also needed to build sound, practical software architectures, is a theoretical model that places software architecture within the larger context of software design. Such a model is provided by Mowbray and Malveau [5]. Their Scalability Model (SM) represents the software design continuum as a series of design levels, each representing the software under consideration at a different level of abstraction. Mowbray and Malveau describe each of these levels thus:

- The global level is concerned with the design issues that are applicable across all systems (enterprises).
- The enterprise level is focused upon coordination and communication (of systems) within a single organization.
- The system level deals with the coordination and communication across applications (and libraries) and sets of applications (and libraries).
- The application level is focused upon the organization of applications developed to meet a set of user requirements.
- The macro component level is focused on the organization and development of application frameworks.
- The micro component level is centered on the software components that solve recurring software problems.
- The classes level is concerned with the development of reusable objects and classes.

While their model was created to provide the foundation of their work in Common Object Request Broker Architecture design patterns, it is most relevant to object-oriented software development but is certainly abstract enough to be applied to

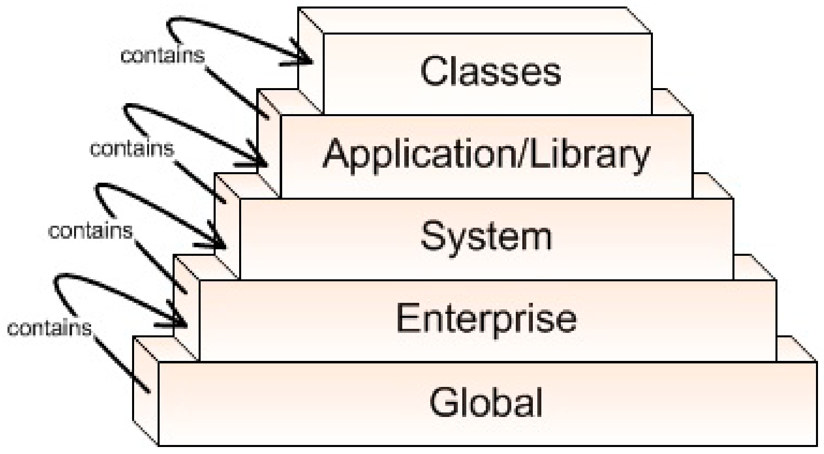


Figure 1:

other methods of software development as well. In this discussion we will use a modified version of the SM (see Figure 1) and architectural issues will focus on the global, enterprise and system levels.

Having established a sound definition of software architecture and a model for constructing such architectures, the practical application of these concepts can now be discussed.

Practice

This section will illustrate a process that can be used to build and graphically document software architectures. A discussion of the capturing of major architectural decisions is beyond the scope of this paper. For a discussion of this topic consult the work of Tyree and Akerman [6]. Also, the textual description of a software architecture that would be included in a formal software architecture document is only briefly mentioned.

This section does discuss the practical application of the modified SM shown in Figure 1. A practical application of the original model is provided by Tepfenhart [7]. This paper describes a rather strict application of UML (v2.0) to several of the concepts presented by Tepfenhart [7]. The basis for the usage of UML described here can be found in several sources [8, 9, 10].

Mowbray and Malveau [5] state that, "One of the key benefits of architecture is the separation of concerns … the SM separates concerns based upon scale of software solutions. The model clarifies the key levels inherent in software systems and the problems and solutions available at each level."

This approach fosters decomposition, a major practice used to control complexity in large (or any size for that matter) software systems. The following summarizes how the SM provides a guideline for the architectural decomposition process. This is followed by an abstract example of how UML would be used to document the global, enterprise and system levels.

In the modified SM shown in Figure 1, we treat each level as a container that holds components that are elements of the next level above. That is, the global level is a container for enterprise components, the enterprise level is a container for system components, and so on for each level of the model. It is important to note here that the term component is used throughout in both a traditional [11] (e.g. software component) and non-traditional sense. We will examine this again when describing the process used to document architectures later in this paper.

To decompose a large software system for the purpose of creating an architectural model, we start with the global level. The global level architecture is composed of enterprise components. Enterprises are the identifiable business units or organizations whose software will interact to achieve some computational goal. The business unit or organization sponsoring the software development is identified as well as business partners, customers, or suppliers—in short any business entity that may interact with the sponsoring organization.

The enterprise level consists of the organization sponsoring the software development and is composed of all the systems that will be employed to achieve the project goals and requirements. Systems identified at this level are not confined to just systems that will be developed as part of the software development effort. In-house legacy systems and COTS products, either existing or that need to be purchased as part of the present effort, are also identified. In this way no effects from unanticipated interfaces should occur during detailed design. For each system that will be developed, the architecture clearly demonstrates collaboration with other systems, either under development, already existing, or with plans to be purchased.

The next step in the process is the decomposition of the systems to be developed that were identified at the enterprise level. The components at this architectural level are either subsystems, applications or libraries. In UML v2.0 libraries are represented as artifacts. Here, we choose to represent libraries as components. Almost invariably, libraries are the manifestation of components and stereotyping components for the various levels of the SM is a natural and more than acceptable method of presentation. Applications are defined as standalone executable software components while libraries though standalone, rely on applications for their run-time execution. Libraries may be either internal or external to applications.

To graphically document a software architecture defined in this way, we use the UML component diagram. The component diagram is perfect for representing the architectural elements at each level of the SM. Furthermore, it is also a perfect companion to the component-oriented definition adopted here. Since the components at several levels of the SM are not software components as defined by the UML, we use stereotypes to indicate the components at each level. Only the components at the system level and above are software components in the sense of the UML definition. We use interfaces, direct connections, and delegation connectors to indicate the relationships between components at all levels. Direct connections are used to represent interfaces between components that may not be call level interfaces, such as shared files or non-digital medium. This is necessary because, as we have seen, not all components in the model represent software.

Figure 2 illustrates a UML component diagram at the global level for a software system under consideration. It conveys to the viewer that the global architecture consists of four enterprises. Assume that Enterprise X is the enterprise for which the software under consideration is being designed. Furthermore, assume that Enterprise W, Y and Z are not part of the same organizational unit (corporation, government agency, etc.) as Enterprise X. This component diagram clearly indicates that Enterprise X needs some functionality or data provided by Enterprise Z, and is provided by Enterprise Z through Interface Z. It also clearly indicates that Enterprise X will provide some functionality through Interface X that will be used by Enterprise Y. This component diagram also shows that Enterprise X has a relationship to Enterprise W through the direct connection XW. This component diagram shows no relationships among Enterprise W, Y and Z. This does not mean that such relationships do not exist; it only means that such relationships, if indeed they do exist, are not important to the architectural description of Enterprise X, and therefore do not need to be included.

We would now move on to a decomposition of Enterprise X into its constituent system components. The result of such a process would be a component diagram for Enterprise X like the one shown in Figure 3. This diagram conveys to the viewer that Enterprise X consists of five systems, Systems A, B, C, D and E. System A provides Interface A that is used by System C. Also, System B provides Interface B1 and Interface B2 which are used by System A and System D respectively. System A has a relationship to System E through direct connection AE. The viewer can also determine that System C is the system within Enterprise X to which access to Interface X by Enterprise Y is delegated; and that Enterprise X delegates to System D use of the external Interface Z that is provided by Enterprise Z. Furthermore, Enterprise X delegates access to Enterprise W to System E via the delegation connection to port Connection XW. It can also be seen that System A and System C are both providers and users of various interfaces, while System B is only a provider of interfaces and System D is only a user of interfaces. But more than this, the viewer knows exactly which interfaces and connections are provided by which systems and likewise which interfaces and connections are used.

Next, each system component identified at the enterprise level would be decomposed into applications and/or libraries. We will assume that analysis has shown that Systems A, B and C will be part of a new development project. Furthermore, we will assume that System D will be a COTS product and System E is a legacy system that is being retained, unchanged. For this discussion we will only describe the decomposition of System A. We also know, from the component diagram for Enterprise X, the relationships System A has to all the other systems at the enterprise level and we will discuss these as well.

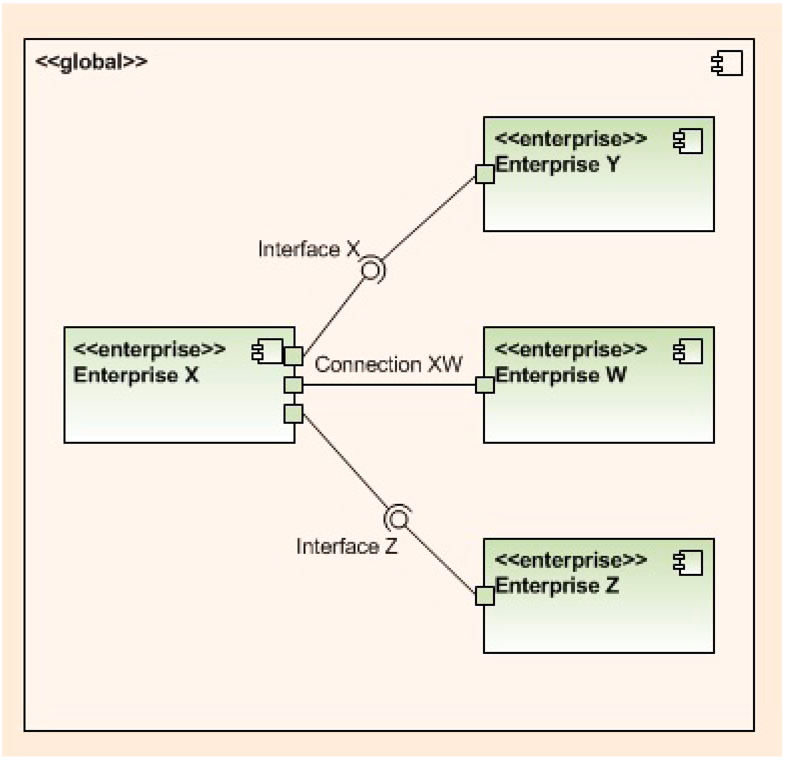

<<global>>
<<enterprise>> Enterprise Y
Interface X
<<enterprise>> Enterprise X
Connection XW
<<enterprise>> Enterprise W
Interface Z
<<enterprise>> Enterprise Z

Figure 2

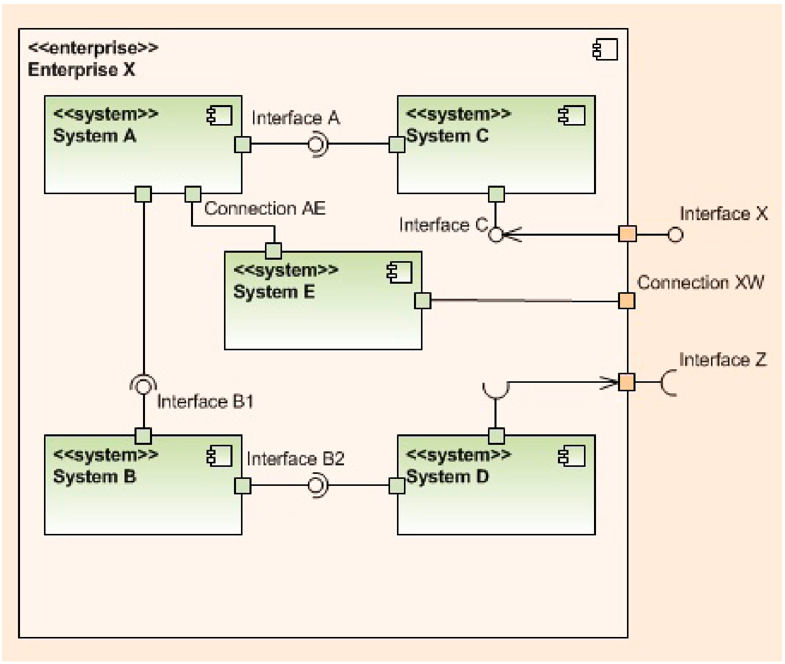


Figure 3

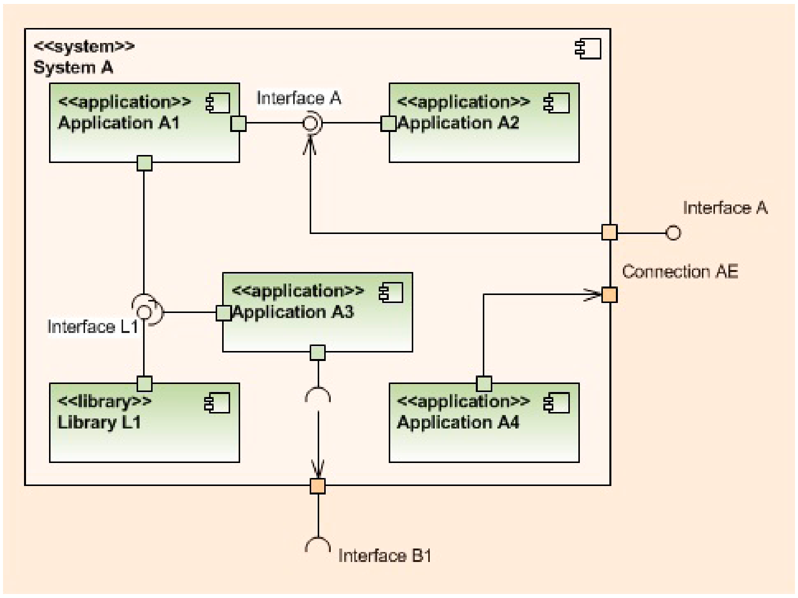


Figure 4

Figure 4 shows the decomposition of System A into its component applications and libraries. This component diagram shows that system A consists of four applications, Application A1, A2, A3 and A4 and one library, Library L1. Application A1 has a relationship to two other components that are a part of System A, that is Application A2 and Library L1. Application A1 provides Interface A that is used by Application A2. Recall that one of the system level interfaces provided by System A, and used by System C, is also Interface A. We say that System A delegates the implementation of Interface A to Application A1, as indicated in the diagram by the delegation connection that connects the external port for Interface A with the provided interface of Application A1. It should be noted that Interface A connected to the external port of System A and Interface A connected to Application A1 are in fact the same interface. Conceptually, one may think of the System A external port connection for Interface A as the access point to the interface provided by Application A1. In fact, neither System A nor Application A1 is actually capable of implementation of Interface A. Application A1 would actually delegate implementation of this interface to a specific class.

Application A1 also uses Interface L1, the provided interface of Library L1 through its required interface. Library L1 is an example of an external library component. Its implementation would be highly dependent on the programming language used to write the code for System A (e.g. in C++ on Windows it could be a Dynamic Link Library, .dll file). The concept of a library as a component more closely approximates the traditional use of the term software component used by Lau and Wang [7]. In their interpretation of the term we would build applications by assembling components, either preexisting or built specifically for the application. Libraries as components, in the context of the SM can consist of a single class or multiple classes. The internal implementation is not significant. From the user of the library's point of view, only the interfaces provided are important.

Application A3, like Application A1, also has a required interface that uses Interface L1. We know from the enterprise level that System A also interfaces to System B through Interface B1. In Figure 4 we can see that System A delegates this responsibility to Application A3.

Application A4, while having no relationship with any of the other applications in System A, does have the responsibility of providing the interface with System E. We can see that System A has delegated this responsibility to Application A4 by the delegation connection to the external port Connection AE. This is an example of a direct connection. The nature of this relationship would be described in the architecture document interfaces section or in a separate interface design document.

One very important point to note is the relationship of System A the container to System A the component of the Enterprise X container. In the Enterprise X component diagram System A has a relationship to three other systems; two are interface relationships, one provided and one required, and the other is a direct connection. System A, the container, maintains those relationships as indicated by the external ports, Interface A, Interface B1 and Connection AE.

The formal software architecture document for this software would contain these diagrams as well as detailed textual descriptions of each component, interface and connection at each level. For the interfaces these would describe the nature of the interface such as data exchange or direct program-to-program communication. It might also include a reference to any standards that might apply. As for the components, specifically the system components of the enterprise level for example, the text description would provide information about which systems will be developed and which might be COTS products. Clearly not all information can be conveyed in just the component diagrams alone. However, it has been demonstrated that a great deal of information can. More importantly, the information that is provided is exactly the kind of information that might be overlooked had the design started without any consideration of software architecture.

Concluding Remarks

While the methodology described here can go a long way to improving our software engineering design drawings and documents, further work is needed to refine the methodology. More consideration needs to be given to the application layer and the macro and micro components layers of the original model. A formal method for the validation and verification of models created with this methodology is also needed. These offer only a few areas for further research.

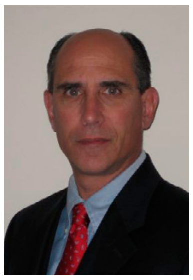
This paper has attempted to close, or at least reduce the width of, the gap between software architecture theory and practice. A methodology was described which demonstrates how to use UML component diagrams as a way to document and communicate software architectures clearly and in a reproducible fashion. This methodology leverages one of the two modern definitions of software architecture found on SEI's website and a lesser-known model for producing software architectures.

If software engineering is ever to achieve the same status as other engineering disciplines, or even approach that status, practitioners must be able to produce universally understood and reproducible design documents. The history of previous work in the area of software architecture has provided a rather stable theoretical foundation. UML, the de-facto standard for creating software engineering design diagrams provides the tools. It is up to us, the practitioners, to use these tools in the way they were intended. It is hoped that this paper demonstrates how to do just that.

Acknowledgements

The author wishes to thank Bob Nicholson, Steve Chappell, Mike Watts and Dr. Jiacun Wang for their review of this paper as well as their thoughtful comments.

ABOUT THE AUTHOR



Michael Tarullo has 29 years of experience in software design and development. Approximately half of this time was spent developing software in the mainframe environment. The remainder of his career has been spent designing and developing object-oriented software in C++, C#, and Java. He is currently an Enterprise Architect at L-3 Communications supporting System Wide Information Management, the FAA SOA initiative for the NextGen Air Traffic Management system. He also is an adjunct professor of Software Engineering at Monmouth University, where he teaches both undergraduate and graduate courses in Java Programming and Software Design. Mr. Tarullo has a Bachelors Degree in Geoscience from The New Jersey City University and a Masters Degree in Software Engineering from Monmouth University.

L-3 Communications
Contractor, FAA William J. Hughes
Technical Center
Atlantic City, NJ 08405
Phone : (609) 485-5294
E-mail: michael.ctr.tarullo@faa.gov
E-mail: michael.tarullo@l-3com.com

REFERENCES

1. <http://www.sei.cmu.edu/architecture/start/definitions.cfm>
2. Kruchten, P., Obbink, H. and Stafford, J.; The Past, Present and Future of Software Architecture; IEEE Software; March/April 2006
3. Shaw, M. and Garlan, D.; Software Architecture – Perspectives On An Emerging Discipline; Prentice Hall; 1996
4. Bass, L., Clements, P. C. and Kazman, R.; Software Architecture in Practice; Addison-Wesley; 2003, 2nd edition
5. Mowbray, T. J and Malveau, J.; CORBA Design Patterns; John Wiley & Sons; 1997
6. Tyree, J. and Akerman, A.; Architecture Decisions: Demystifying Architecture; IEEE Software; v.22, no.2, 2005
7. <http://bluehawk.monmouth.edu/~btepfenh/Courses/SE505/Sections/principlesdesign.html>
8. Booch, G., Rumbaugh, J. and Jacobson, I.; The Unified Modeling Language User Guide; Addison Wesley; 2nd Edition; 2005
9. Rumbaugh, J., Jacobson, I. and Booch, G.; The Unified Modeling Language Reference Manual; Addison Wesley; 2nd Edition; 2005
10. Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Conallen, J. and Houston, K. A.; Object-Oriented Analysis And Design With Applications; Addison Wesley; 3rd Edition; 2007
11. Lau, K. and Wang, Z.; Software Component Models; IEEE Transactions On Software Engineering; October 2007, vol. 33, no. 10