World Academy of Science, Engineering and Technology
International Journal of Computer and Systems Engineering
Vol:5, No:12, 2011

Strongly Adequate Software Architecture

Pradip Peter Dey

***Abstract*—**Components of a software system may be related in a wide variety of ways. These relationships need to be represented in software architecture in order develop quality software. In practice, software architecture is immensely challenging, strikingly multifaceted, extravagantly domain based, perpetually changing, rarely cost-effective, and deceptively ambiguous. This paper analyses relations among the major components of software systems and argues for using several broad categories for software architecture for assessment purposes: strongly adequate, weakly adequate and functionally adequate software architectures among other categories. These categories are intended for formative assessments of architectural designs.

***Keywords*—**Components, Model Driven Architecture, Graphical User Interfaces.

I. Introduction

THIS paper critically examines the current best practices in software architectural studies and suggests some strategies for improvements through formative assessments. Most engineering fields are founded on scientific disciplines. Unlike most engineering fields, software engineering is primarily based on current best practices [1]. There are some recent attempts to establish software science as a foundation of software engineering [2]. This may promote more analytical reasoning about software architecture, if it becomes popular. Software architectural design would benefit from analytical reasoning with scientific foundations. Importance of software architecture in the software design process is generally accepted among practitioners. According to Pressman [1: page 223] “One goal of software design is to derive an architectural rendering of a system”. Architectural design, detailed design and design reviews provide the most important steps in a cost effective software development process. Software engineering activities are goal directed in order to produce working software in a timely manner within some cost constraints. For complex computer based systems, software architecture plays a very important role in its success or failure. Software architecture is “the overall structure of the software and the ways in which that structure provides conceptual integrity for a system” [3]. According to Braude and Bernstein [4: page 438], "A software architecture describes the overall components of an application and how they relate to each other." Software architectural design is immensely challenging, strikingly multifaceted, extravagantly domain based, perpetually changing, rarely cost-effective, deceptively ambiguous, and perilously constrained with some exceptions [5]. Often software architecture is presented in misleading notations and incomplete or controversial descriptions. The best architectural practices are rarely published and often inferred from excellent products [5].

Pradip Peter Dey is with National University, 3678 Aero Court, San Diego, CA 92123, USA. He is now with the School of Engineering, Technology and Media (phone: 858-309-3421; fax: 858-309-3421; e-mail: pdey@nu.edu)

This paper is intended to develop some architecture evaluation ideas that may bring some clarity to architectural design specifications and their assessments. We are primarily interested in formative assessments during reviews and inspections. According to Wang [2: page 102], review and inspection is “a software engineering principle for finding and eliminating software design and implementation defects via reading and examining the work products by peer or more experienced reviewers.” This paper takes a more practical approach to reviews and inspections. The term “adequate software architecture” is often used in published articles [6] with a connotation of quality; however, the term is not properly defined. This paper suggests some architectural design contexts in which a set of related terms can be used with clear meanings and appropriate definitions.

II. Background

Controversies about software development have been profoundly ostentatious and often explicated with effective metaphors. Donald Knuth initially [7] suggested that software writing is an art. David Gries [8] argued it to be a science. Watts Humphrey [9] viewed it as a process. In recent years, practitioners have come to realize that software is engineered [1]-[2], [4], [10]-[13]. The scientific foundation of software engineering is not fully understood. That is, we do not understand it the way we understand chemistry as the scientific foundation of chemical engineering. Software architectural design is based partly on computer science and partly on behavioral sciences and intuitive judgments although there are attempts to establish “software science” [2] as the primary basis for software architecture. It is often suggested that software architectural design is creatively built from requirements analysis in an iterative process [1], [4], [9]-[17].

Model Driven Architecture (MDA) is becoming increasingly popular among the practicing software engineers [15]. MDA promotes grouping of models into three categories, according to their abstraction level; namely, Computation-Independent Models, Platform-Independent Models and Platform-Specific Models. MDA advocates Model Driven Development. In order to highlight some features, we will consider a small case presented below.

Assume that a small software project started with the following initial description of requirements: Develop a software system for computing volume of two types of storage units: box-storage and cylinder-storage. Users should be able to enter input interactively using a Graphical User Interface (GUI).

Assume that after studying requirements, software engineers discover that the system has to be web-based and it should be available 24/7. Users should be able to access the software without any login ID. It should be easy to maintain by an administrator. The software engineers then would prepare a software requirements specification (SRS) document. A modern requirements analysis is generally use case driven. A use case diagram is drawn with UML notations [18]. A use case diagram for the storage volume problem is given in Figure-1. An activity diagram can be drawn for each of the use cases in order to provide a visual representation of details of the requirements [1]. Alternatively, a use case operational diagram can be drawn for each use case; Figure-2 shows one use case operational diagram for the box-storage volume use case.

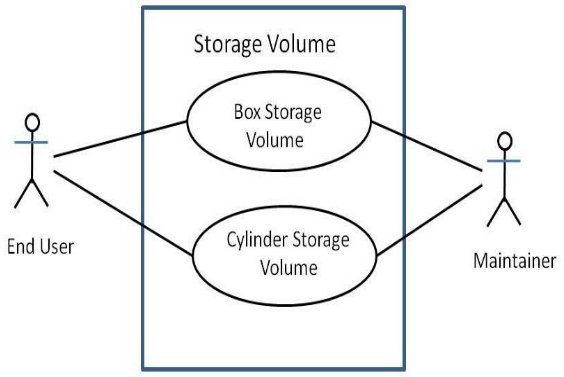

Fig. 1 A Use Case Diagram

One of the loops of the use case operational diagram, given in Figure-2, shows that user enters box-dimensions which are accepted based on some criteria and the volume is computed. Otherwise, the dimensions are rejected and an error message is generated. The notation for the use case operational diagram is similar to that of activity diagram.

In the next phase, the software architectural design is developed based on the requirements analysis according to some design approach. "In the use-case driven architecture design approach, use cases are applied as the primary artifacts for deriving the architectural abstractions" [19: page 13]. Engineers need to pay attention to details during the architectural design process , because "Architectures allow or preclude nearly all of the system's quality attributes" [20]. An elegant generic architectural framework, the Model-View-Controller (MVC) often helps software engineers in developing an architectural design for a given problem. Use case driven derivation of an instance of the MVC architecture for a specific problem allows efficient and cost effective development. Given the MVC architecture as a general guide, the domain specific computation of box-volume and cylinder-volume would be done in the Model component. The graphical user interface (GUI) elements, such as input fields, buttons, etc would be placed in the View component. The user interactions are done in the Controller. The statements about what happens when the user presses the submit button would go to the Controller component. Following the preceding logic the architecture is made ready for review.

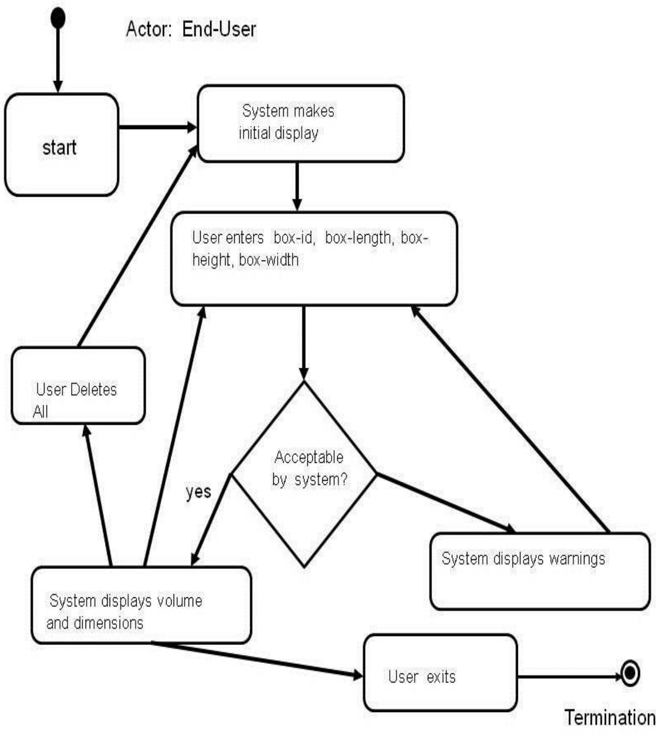
Actor: End-User
start
System makes initial display
User enters box-id, box-length, box-height, box-width
User Deletes All
Acceptable by system?
yes
System displays warnings
System displays volume and dimensions
User exits
Termination

Fig. 2 A Use Case Operational Diagram for the Box-Storage Volume Use Case

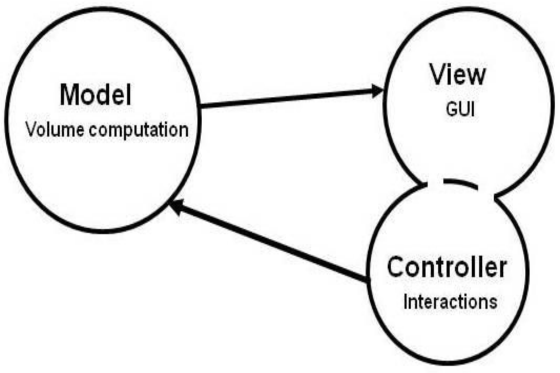

Fig. 3 A Model-View-Controller Instance for the Volume Problem

Programmer's questions are usually about the View-Controller relation. Are they tightly coupled or loosely coupled? What should be done in the review of the architecture? Following these design guidelines the software architecture is developed and a prototype is implemented in a Java applet for formative assessment. The implementation of the prototype architecture is posted at the following web site: http://www.asethome.org/soft/storage.html.

The basic architectural abstraction is presented in the instance of the MVC architecture shown in Figure-3. A significant aspect of the architecture of Figure-3 is that it merges the View and Controller together into one compound or composite unit. In other words, the View and Controller elements are put in a class that extends the Applet class of Java. The Model is a distinct reusable component where volume for the box-storage and cylinder-storage units is computed.

The classes in the Model are shown in the class diagram in Figure-4.

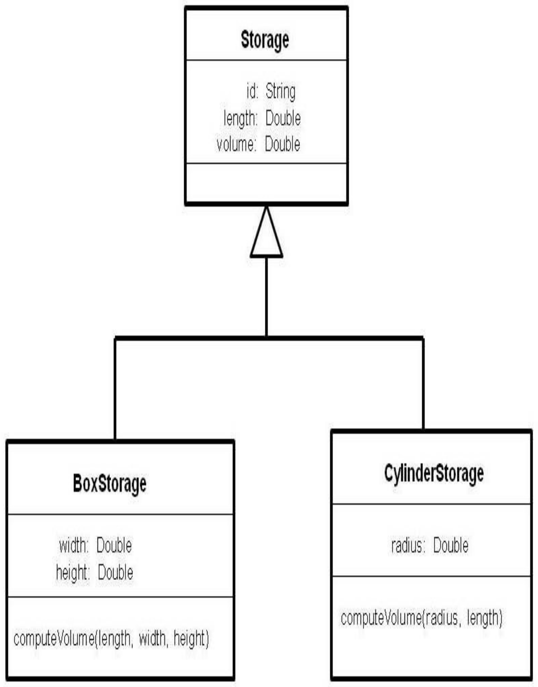

Fig. 4 A Class Diagram for the classes in the Model

After initial architectural design is completed, it is ready for a review. The review would help to assess the architecture in order to identify risks and make improvements. “An architecture evaluation does not tell you yes or no, good or bad, or 6.75 out of 10. It tells you where you are at risk.” [20, page 29]. Formative assessments are very important in an educational environment. McConnel [21] suggests a checklist of questions.

Some sample questions are:

1) Does the architecture account for all the requirements?
2) Does the whole architecture hang together conceptually?
3) Is the top-level design independent of the machine and language that will be used to implement it?

The questions such as these form a checklist for achieving high architectural quality in a convenient way. The checklists are very useful for assessment in order to avoid missing elements. In addition to checklists, some quality categories can be used especially in formative assessments.

III. Formative Assessments

The main purpose of formative assessments is to improve the quality of the architecture. Formative assessments are effective during the iterative development process. In the review process we would like to use some quality categories. Practicing software engineers suggest that loosely coupled components are more desirable than tightly coupled components, because loosely coupled components are independent reusable components and the related knowledge management is feasible [22]. However, in a well-integrated system, View-Controller relationship may be tightly coupled in well-designed implementations. Other aspects that need to be examined carefully during formative assessments are: abstraction levels, functional and non-functional requirements, security issues, architecture description language and notations. Based on these considerations the following categories are suggested for formative assessments along with detailed comments.

Strongly adequate software architecture: Software architecture is strongly adequate, if and only if, it is weakly adequate and modularity, high cohesion, low coupling, robustness, flexibility, reusability, efficiency, security and reliability are achieved at all levels of abstraction.

Weakly adequate software architecture: Software architecture is weakly adequate, if and only if, it represents solutions to all functional and non-functional requirements appropriately at least for the implementation level.

Functionally adequate software architecture: Software architecture is functionally adequate, if and only if, it represents solutions to all functional requirements at least for the implementation level.

Narrowly adequate software architecture: Software architecture is narrowly adequate, if and only if, it represents solutions to all non-functional requirements at least for the implementation level.

Marginally adequate software architecture: Software architecture is marginally adequate, if and only if, it represents solutions to a subset of functional and non-functional requirements at least for the implementation level.

Notionally adequate software architecture: Software architecture is notionally adequate, if and only if, it represents solutions to functional and non-functional requirements with an appropriate architectural description language notation.

Organizationally adequate software architecture: Software architecture is organizationally adequate, if and only if, it represents the overall organization of the software with clear definitions of all components.

Security adequate software architecture: Software architecture is security adequate, if and only if, it represents all security measures in the overall organization of the software with clear definitions. The definitions given above are general in the sense that they are not constrained by any particular programming language or problem domain. It is generally clear that the best category of all is the strongly adequate software architecture. Software developers strive to achieve this target using modern techniques. The reviewers, on the other hand, carefully review the architecture and assign the most appropriate category with appropriate comments. It should be clear that the architectural design example presented above does not belong to the strongly adequate software architecture. The software architecture for the volume problem does not support all levels of abstraction. It deals with the implementation level by merging the view and controller elements which needs to be placed preferably in loosely coupled components. It is a weakly adequate software architecture which can be improved following the guidelines generated in the formative assessment. The categories described in this section provide a viable alternative to earlier attempts in evaluating software architecture [9], [20], [23]. According to Clements, Kazman, and Klein architecture evaluation "produces answers to two kinds of questions. (1) Is the architecture suitable for the system it was designed? (2) Which two or more competing architectures is the most suitable for of the system at hand?" [20, page 27]. The first question is somewhat summative. In contrast, this paper argues for formative assessments.

IV. Conclusion

Strongly adequate software architecture is defined along with some other software quality categories which may help in formative assessments of software architecture. The architectural categories are not constrained by a particular programming language, or domain. Software engineers strive for the strongly adequate software architecture. However, software architecting is an iterative process and formative assessments guide the architects to improve the qualitative aspects in an iterative process. The categories proposed in this paper are intended to help reviewers in formative assessments. The role of formative assessments is stressed during the development process in order to produce revised architectures from initial work or working progress. Additional research is needed to measure the effectiveness of formative assessments with the proposed qualitative categories of software architecture.

Acknowledgment

The author gratefully acknowledges the help and/or encouragements received from George Beckwith, Mohammad Amin, Ali Farahani, Gordon Romney, Bhaskar Raj Sinha, and Jodi Reeves.

References

[1] R. S. Pressman, *Software Engineering: A Practitioner's Approach.* (7th ed.), McGraw-Hill, 2010.
[2] Y. Wang, *Software Engineering Foundations: A Software Science Perspective,* Auerbach Publications, 2008.
[3] M. Shaw, and D. Garlan, "Formulations and Formalisms in Software Architectures", *Computer Science Today: Recent Trends and Developments*, Springer-Verlag LNCS, 1000, 307-323, 1995.
[4] E. Braude, and M. Bernstein, *Software Engineering: Modern Approaches*, (2nd Edition), John Wiley & Sons, 2011.
[5] J. Hong, "Why is Great Design so Hard?", *Communications of the ACM,* July 2010.
[6] J. L. Azevedo, B. Cunha, and L. Almeida, "Hierarchical Distributed Architectures for Autonomous Mobile Robots: A case study", in *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation*, 2007.
[7] D. E. Knuth, *Seminumerical Algorithms: The Art of Computer Programming 2.* Addison-Wesley, Reading, Mass., 1969.
[8] D. Gries, *The Science of Programming.* Springer, 1981.
[9] W. Humphrey, *Managing the Software Process*, Reading, MA. Addison-Wesley, 1989.
[10] I. Sommerville, *Software Engineering,* 9th Edition, Addison Wesley, 2010.
[11] S. Pfleeger, and J. Atlee, *Software Engineering,* Prentice-Hall, 2010.
[12] B. Agarwal, S. Tayal and M. Gupta, *Software Engineering and Testing,* Jones and Bartlet, 2010.
[13] F. Tsui, and O. Karam, *Essentials of Software Engineering*, 2nd Ed., Jones and Bartlet, 2011.
[14] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice,* 2nd Edition Addison-Wesley, 2003.
[15] J. Miller, and J. Mujerki, Editors, MDA Guide, Version 1, OMG Technical Report. Document OMG/200-05-01, http://www.omg.com/mda, 2003.
[16] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering Using UML, Patterns, and Java* (3rd Edition), Prentice Hall, 2009
[17] Capers Jones, *Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies*, McGraw-Hill, 2009.
[18] R. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual.* (2nd Edition), Addison Wesley, 2005.
[19] B. Tekeinerdogan, and M. Aksit, "Classifying and Evaluating Architecture Design Methods", in M. Aksit (editor), *Software Architectures and Component Technology*, Kluwer Academic Publishers, 2002.
[20] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures.*Addison-Wesley, 2002.
[21] S. McConnel. *Code Complete,* Microsoft Press, 2004.
[22] M. Babar, T. Dingsoyr, P. Lago, and H. van Vliet, Editors, *Software Architecture Knowledge Management*, Springer, 2009.
[23] E. Bouwers, J. Correia, A. van Deursen, and J. Visser, "Quantifying the Analyzability of Software Architectures," Technical Report, Delft University of Technology, 2011.