FORMATIVE ASSESSMENT OF SOFTWARE DESIGN

Pradip Peter Dey
Professor, School of Engineering, Technology and Media, National University
3678 Aero Court, San Diego, CA 92123, USA

Introduction

The software engineering problems are so complex or large, that a single developer cannot solve them anymore. A team of developers work together in a collaborative manner to engineer a product in a complex process. Robert Baber (1997) explained engineering aspects of software engineering as ". . . the systematic and regular application of scientific and mathematical knowledge to the design, construction, and operation of machines, systems, and so on of practical use and, hence, of economic value. Particular characteristic of engineers is that they take seriously their responsibility for correctness, suitability, and safety of the results of their efforts." Most engineering fields are founded on scientific disciplines. Software engineering is based partly on computer science and partly on psychology, management, economics, and intuitive judgments although there are attempts to establish "software science" (Wang 2008) as the primary basis for software engineering. "Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software" (IEEE, 1990). There are many alternative definitions of software engineering. However, current best practices play an important role in software engineering (Pressman 2010). Controversies about software development have been profoundly ostentatious and often explicated with effective metaphors. Donald Knuth (1969) initially suggested that software writing is an art. David Gries (1981) argued it to be a science. Watts Humphrey (1989) viewed it as a process. In recent years, practitioners have come to realize that software is engineered (Pressman 2010; Wang 2008; Braude & Bernstein, 2011; Sommerville 2010; Pfleeger & Atlee 2010; Agarwal, Tayal & Gupta 2010; Tsui & Karam, 2011).

Most software projects start with some fuzzy requirements. After some systematic analysis of the requirements, a software requirement specification (SRS) document is produced. Based on this, software design and implementation are performed along with software testing. Finally, the software is delivered or installed at customer sites and the maintenance phase starts. It is often suggested that software design is creatively built from requirements analysis in an iterative process (Pressman 2010; Wang 2008; Braude & Bernstein, 2011; Sommerville 2010; Pfleeger & Atlee 2010; Agarwal, Tayal & Gupta 2010; Tsui & Karam 2011). This paper critically examines the current best practices in software design studies and suggests some strategies for improvements. Software design activities include architectural design, inspection or design review and detailed design. For the purpose of this paper, we focus on software architectural design or high level design. Importance of software architecture in the software design process is generally accepted among practitioners. According to Pressman (2010: 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 any complex computer based system, 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" (Shaw & Garlan 1995). According to Braude and Bernstein (2011: page 438], "A software architecture describes the overall components of an application and how they relate to each other." In practice, software architectural design is immensely challenging, strikingly multifaceted, extravagantly domain based, perpetually changing, rarely cost-effective, deceptively ambiguous, and perilously constrained. Often, software descriptions. The best architectural practices are rarely published and often inferred from excellent products (Hong 2010). This paper is intended to develop some software design 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 (2008: 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 (Azevedo, Cunha & Almeida 2007) with a connotation of quality; however, the term is not defined. This paper suggests some contexts in which a set of related terms can be used with clear meanings and appropriate definitions.

Iterative Process

It is often suggested that software design is creatively built from requirements analysis in an iterative process ((Pressman 2010; Wang 2008; Braude & Bernstein, 2011; Sommerville 2010; Pfleeger & Atlee 2010; Agarwal, Tayal & Gupta 2010; Tsui & Karam, 2011). In this process, after some initial requirements analysis, a software design representation is developed and then the requirements analysis is augmented on the basis of a combination of software design reviews, new or changed requirements or some other factors which in turn lead to a revised software design. That is, the iterative process of development or the spiral process model (Boehm 1988) is found to be one of the most productive software processes. Certain aspects of software are such that after an initial assessment, iterative refinements help. One of the greatest benefits of the iterative process is the improvements made in the development of user interfaces, in each successive iteration. The current study is based on the following iterative scheme where software design and modeling is followed by design review or evaluation.

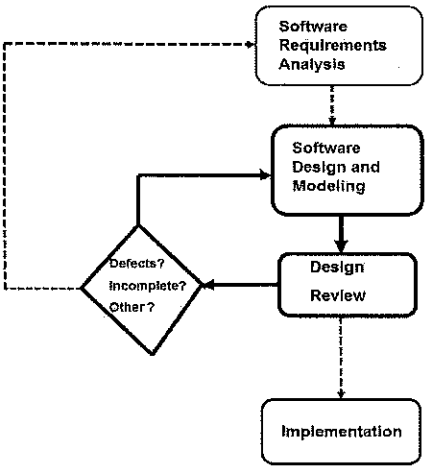

Figure 1: Iterative Design and Modeling

The iterative process of design and modeling suggested in Figure 1 allows developers to start with a highly abstract conceptual design and add details gradually in each successive iteration following the solid arrows. The dotted arrows show other viable alternatives. User interface development requires adjustments and refinements that are best done in iterations. Often defects are found during the review or evaluation process and those defects need to be corrected. The design may start with just a few elements and other elements are incrementally added.

Assume that a small software project started with the following initial requirements description: "Develop a software system for computing the volume of two types of storage units: box-storage and cylinder-storage. Users should be able to enter inputs interactively using a Graphical User Interface (GUI)". After studying the requirements, software engineers would discover that the system has to be web-based and should be available 24/7. Users should be able to access the software without any login ID. The system 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. The use case diagram for the storage volume problem is given in Figure-2 in the standard UML notations (Rumbaugh, Jacobson & Booch 2005). An activity diagram can be drawn for each of the use cases in order to provide a visual representation of details of the requirements (Pressman 2010). Alternatively, a use case operational diagram can be drawn for each use case; Figure-3 shows one use case operational diagram for the box-storage volume use case.

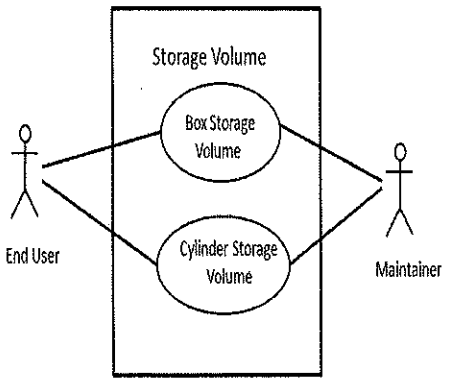

Figure 2: Use Case diagram

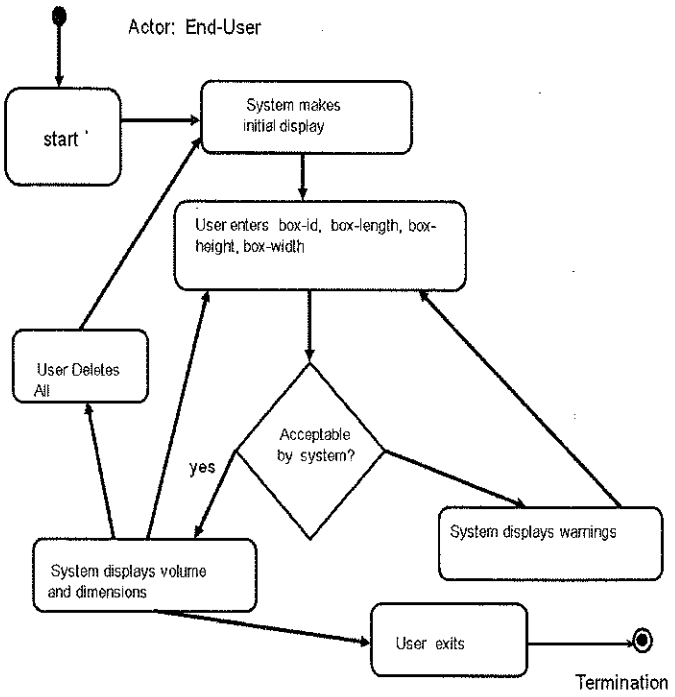

Figure 3: A Use Case Operational Diagram for Box-Storage Volume Use Case

In one of the loops of the use case operational diagram given in Figure-3 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" (Tekeinerdogan, & Aksit, 2002: 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" (Clements, Kazman, & Klein 2005). An elegant generic design framework, the Model-View-Controller (MVC) architecture, often helps software engineers in developing an architectural design for a given problem. 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 created. 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 architectural design? Following these design guidelines the software architectural design is performed and a prototype is implemented in a Java applet for formative assessment. The implementation of the architecture is posted at the following web site: http://www.asethome.org/soft/storage.html. The basic design abstractions are shown in the instance of the Model-View-Controller shown in Figure 4.

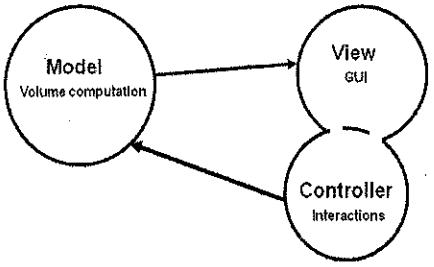

Figure 4: A Model-View-Controller Instance for the Volume Problem

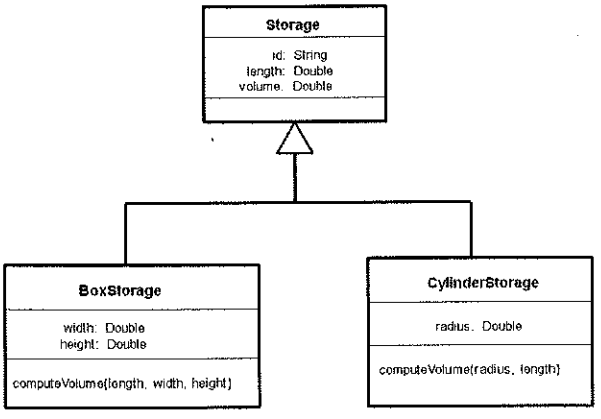

Figure 5: A Class Diagram for the classes in the Model

A significant aspect of the architecture of Figure 4 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 5. After initial architectural design, review needs to be conducted. The review would help to assess the design in order to make improvements. Formative assessments are very important in an educational environment. McConnel (2004) 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 design quality in a convenient way.

Assessment Categories

The main purpose of formative assessments is to improve the quality of the design. 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. 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 using in the formative assessments along with detailed comments.

Strongly adequate software design: Software design 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 design: Software design is weakly adequate, if and only if, it represents answers to all functional and non-functional requirements appropriately at least for one level of abstraction.

Functionally adequate software design: Software design is functionally adequate, if and only if, it represents answers to all functional requirements at least for one level of abstraction.

Narrowly adequate software design: Software design is narrowly adequate, if and only if, it represents answers to all non-functional requirements at least for one level of abstraction.

Marginally adequate software design: Software design is marginally adequate, if and only if, it represents answers to a subset of functional and non-functional requirements at least one level of abstraction.

Notionally adequate software design: Software design is notionally adequate, if and only if, it represents answers to functional and non-functional requirements with an appropriate modeling language such as UML.

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

Security adequate software design: Software design is security adequate, if and only if, it represents all security measures in the overall organization of the software with clear definitions.

It is clear that the best category of all is the strongly adequate software design. Software developers strive to achieve this target using modern techniques. The reviewers, on the other hand, carefully review the design and assign the most appropriate category. It should be clear that the example presented above does not belong to the strongly adequate software design. It is a weakly adequate software design which can be improved following the guidelines generated in the formative assessment.

Conclusions

Strongly adequate software design is defined along with some other software quality categories which may help in formative assessment of software. Software engineers strive for the strongly adequate software design. However, software design is an iterative process and formative assessment guides the professionals to improve the qualitative aspects in each iteration. The categories proposed in this paper are intended to help reviewers in formative assessments. Additional research is needed to measure the effectiveness of formative assessments with the proposed qualitative categories.

References

1. Agarwal, B., Tayal, S. & Gupta, M. (2010). Software Engineering and Testing, Jones and Bartlet.
2. Azevedo, J., Cunha, B. & Almeida, L. (2007). Hierarchical Distributed Architectures for Autonomous Mobile Robots: A case study, in Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation, 2007.
3. Babar, M, Dingsoyr, T., Lago, P. & van Vliet, H. (Editors) (2009). Software Architecture Knowledge Management, Springer.
4. Baber, R. (1997). Comparison of Electrical "Engineering" of Heaviside's Times and Software "Engineering" of our Times. IEEE Annals of the History of Computing, 19(4): 5-17.
5. Bass, L. Clements, P. & Kazman, R. (2003). Software Architecture in Practice, 2nd Edition Addison-Wesley.
6. Boehm B, (1988). A Spiral Model of Software Development and Enhancement, IEEE Computer, IEEE, 21(5):61-72, May 1988.
7. Braude, E. & Bernstein, M. (2011). Software Engineering: Modern Approaches, (2nd Edition), John Wiley.
8. Clements, P., Kazman, R. & Klein, M. (2005). Evaluating Software Architectures. Addison-Wesley
9. Gries, D. (1981). The Science of Programming. Springer.
10. Hong, J. (2010) "Why is Great Design so Hard?", Communications of the ACM, July 2010.
11. Humphrey,W. (1989). Managing the Software Process, Reading, MA. Addison-Wesley.
12. IEEE (1990). IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology.
13. Knuth, D. (1969). Seminumerical Algorithms: The Art of Computer Programming 2. Addison-Wesley.
14. McConnel, S. (2004). Code Complete, Microsoft Press.
15. Miller, J. & Mujerki, J. (Editors) (2003) MDA Guide, Version 1, OMG Technical Report. Document OMG/200-05-01, http://www.omg.com/mda.
16. Pfleeger, S. & Atlee, J. (2010). Software Engineering, Prentice-Hall.
17. Pressman, R. (2010). Software Engineering: A Practitioner's Approach. (7th ed.), McGraw-Hill.
18. Rumbaugh, R., Jacobson, I. & Booch, G. (2005). The Unified Modeling Language Reference Manual. (2nd Edition), Addison Wesley
19. Shaw, M. & Garlan, D. (1995). Formulations and Formalisms in Software Architectures, Computer Science Today: Recent Trends and Developments, Springer-Verlag LNCS, 1000, 307-323.
20. Sommerville, I. (2010). Software Engineering, 9th Edition, Addison Wesley.
21. Tekeinerdogan,B. & Aksit, M. (2002). Classifying and Evaluating Architecture Design Methods, in M. Aksit (editor), Software Architectures and Component Technology, Kluwer Academic Publishers.
22. Tsui, T. & Karam, O. (2011). Essentials of Software Engineering, 2nd Ed., Jones and Bartlet.
23. Wang, Y. (2008). Software Engineering Foundations: A Software Science Perspective, Auerbach Publications.

* * * * *