Software Architecture, Scenario and Patterns

R.V. Siva Balan[1], Dr. M. Punithavalli[2]

[1]Department of Computer Applications,
Narayanaguru College of Engineering, kanyakumari, India.

[2]Director & Head Department of Computer Applications,
Sri Ramakrishna College of Arts and Science for women, Coimbatore, India.

Abstract
The software engineering projects [22, 23] reveals that a large number of usability related change requests are made after its deployment. Fixing usability problems during the later stages of development often proves to be costly, since many of the necessary changes require changes to the system that cannot be easily accommodated by its software architectural design. This costs high for the practitioners and prevents the developers from finding all the usability requirements, resulting in systems with less than ideal usability. The successful development of a usable software system therefore must include creating a software architecture that supports the optimal level of usability. Unfortunately, no architectural design usability assessment techniques exist. To support software architects in creating a software architecture that supports usability, practicing a scenario based assessment technique that leads to successful application of pattern specification is undergone. Explicit evaluation of usability during architectural design may reduce the risk of building a system that fails to meet its usability requirements and may prevent high costs incurring adaptive maintenance activities once the system has been implemented.

Keywords: *use-case, patterns, usability, scenarios, patterns specifications*

1. Introduction

Scenarios have been gaining increasing popularity in both Human Computer Interaction (HCI) and Software Engineering (SE) as 'engines of design'. In HCI scenarios are used to focus discussion on usability [3] issues .They support discussion to gain an understanding of the goals of the design and help to set overall design objectives. In contrast, scenarios play a more direct role in SE, particularly as a front end to object oriented design. Use case driven approaches have proved useful for requirements elicitation and validation. The aim of use cases in Requirements Engineering is to capture systems requirements. This is done through the exploration and selection of system user interactions to provide the needed facilities. A use case is a description of one or more end to end transactions involving the required system and its environment. The basic idea is to specify use cases [8] that cover all possible pathways through the system functions. The concept of use case was originally proposed in Objectory [8] but has recently been integrated in a number of other approaches including the Fusion method and the Unified Modeling Language [7].

In the software design area, the concept of design patterns has been receiving considerable attention. The basic idea is to offer a body of empirical design information that has proven itself and that can be used during new design efforts. In order to aid in communicating design information, design patterns focus on descriptions that communicate the reasons for design decisions, not just the results. It includes descriptions of not only '*what*' but also '*why*'. Given the attractiveness and popularity of the patterns approach, a natural question for RE is: How can requirements guide a patterns-based approach to design? A systematic approach to organizing, analyzing, and refining nonfunctional requirements can provide much support for the structuring, understanding, and applying of design patterns during design.

2. Software architecture

The challenge in software development is to develop software with the right quality levels. The problem is not so much to know if a project is technically feasible concerning functions required, but instead if a solution exists that meets the software quality requirements, such as throughput and maintainability.

Traditionally the qualities of the developed software have, at best, been evaluated on the finished system before delivering to the customer. The obvious risks of having spent much effort on developing a system that eventually did not meet the quality requirements have been hard to manage. Changing the design of the system would likely mean rebuilding the system from scratch to the same cost. The result from the software architecture design activity is a software architecture. But, the description of that software architecture is far from trivial. A reason is that it is hard to decide what information is needed to describe software architecture, and hence, it is very hard to find an optimal description technique.

In the paper by Perry and Wolf [2] the foundations for the study of software architecture define software architecture as follows:

Software Architecture = {Elements, Form, Rationale}

Thus, software architecture is a triplet of (1) the elements present in the construction of the software system, (2) the form of these elements as rules for how the elements may be related, and (3) the rationale for why elements and the form were chosen. This definition has been the basis for other researchers, but it has also received some critique for the third item in the triplet. In [15] the authors acknowledge that the rationale is indeed important, but is in no way part of the software architecture. The basis for their objection is that when we accept that all software systems have inherent software architecture, even though it has not been explicitly designed to have one, the architecture can be recovered. However, the rationale is the line of reasoning and motivations for the design decisions made by the design, and to recover the rationale we would have to seek information not coded into software.

Software system design consists of the activities needed to specify a solution to one or more problems, such that a balance in fulfillment of the requirements is achieved. A *software architecture design method* implies the definition of two things. (i) A process or procedure for going about the included tasks. (ii) A description of the results or type of results to be reached when employing the method. From the software architecture point-of-view, the first of the aforementioned two, includes the activities of specifying the components and their interfaces, the relationships between components, and making design decisions and document the results to be used in detail design and implementation. The second is concerned with the definition of the results, i.e. what is a component and how is it described.

The traditional object-oriented design methods, e.g. (OMT [12], Booch [6], Objectory [8]) has been successful in their adoption by companies worldwide. Over the past few years the three aforementioned have jointly produced a *unified modeling language* (UML) [7] that has been adopted as de facto standard for documenting object-oriented designs.

3. Scenarios

Scenarios serve as abstractions of the most important requirements on the system. Scenarios play two critical roles, i.e. design driver, and validation/illustration. Scenarios are used to find key abstractions and conceptual entities for the different views, or to validate the architecture against the predicted usage. The scenario view should be made up of a small subset of important scenarios. The scenarios should be selected based on criticality and risk. Each scenario has an associated script, i.e. sequence of interactions between objects and between processes [13]. Scripts are used for the validation of the other views and failure to define a script for a scenario discloses an insufficient architecture.

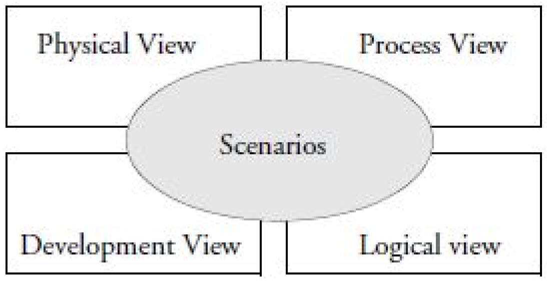

Fig. 1 4+1 View model design method

The 4+1 View Model presented in [17] was developed to rid the problem of software architecture representation. Five concurrent views (Fig. 1) are used; each view addresses concerns of interest to different stakeholders. On each view, the Perry/Wolf definition [2] is applied independently. Each view is described using its own representation, a so called *blueprint*. The fifth view (+1) is a list of scenarios that drives the design method.

4. Usability concerns

The work in this paper is motivated by the fact that this also applies to usability. Usability is increasingly recognized as an important consideration during software development; however, many well-known software products suffer from usability issues that cannot be repaired without major changes to the software architecture of these products. This is a problem for software development because it is very expensive to ensure a particular level of usability after the system has been implemented. Studies [21, 22] confirm that a significant large part of the maintenance costs of software systems is spent on dealing with usability issues. These high costs can be explained because some usability requirements will not be discovered until the software has been implemented or deployed.

5. Patterns

Software engineers have a tendency to repeat their successful designs in new projects and avoid using the less successful designs again. In fact, these different styles of designing software systems could be common for several different unrelated software engineers. This has been observed in [18] where a number of systems were studied and common solutions to similar design problems were documented as *design patterns*.

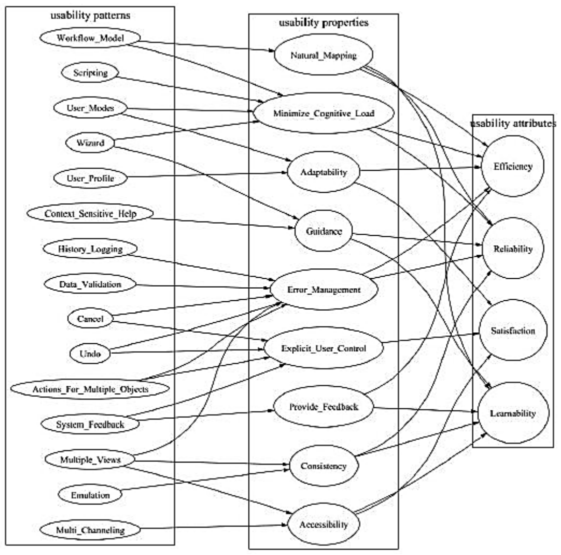

Fig. 2 Usability Framework

The concept has been successful and today most software engineers in are aware of design patterns. The concept has been used for software architecture as well. First by describing *software architecture styles* [16] and then by describing *software architecture patterns* [5] in a form similar to the design patterns. The difference between software architecture styles and software architecture patterns have been extensively debated. Two major viewpoints are; styles and patterns are equivalent, i.e. either could easily be written as the other, and the other view point is, they are significantly different since styles are a categorization of systems and patterns are general solutions to common problems.

Either way styles/patterns make up a common vocabulary. It also gives software engineers support in finding a well-proven solution in certain design situations.

The design and use of explicitly defined software architecture has received increasing amounts of attention during the last decade. Generally, three arguments for defining an architecture are used [14]. First, it provides an artifact that allows discussion by the stakeholders very early in the design process. Second, it allows for early assessment of quality attributes [29,25]. Finally, the design decisions captured in the software architecture can be transferred to other systems.

Our work focuses on the second aspect: early assessment of usability. Most engineering disciplines provide techniques and methods that allow one to assess and test quality attributes of the system under design. For example for maintainability assessment code metrics [23] have been developed. In [3] an overview is provided of usability evaluation techniques that can be used during software development. Some of the more popular techniques such as user testing [9], heuristic evaluation [10] and cognitive walkthroughs [1] can be used during several stages of development. Unfortunately, no usability assessment techniques exist that focus on assessment of software architectures. Without such techniques, architects may run the risk of designing a software architecture that fails to meet its usability requirements. To address to this problem we have defined a scenario based assessment technique (SALUTA).

The Software Architecture Analysis Method (SAAM) [20] was among the first to address the assessment of software architectures using scenarios. SAAM is stakeholder centric and does not focus on a specific quality attribute. From SAAM, ATAM [19] has evolved. ATAM also uses scenarios for identifying important quality attribute requirements for the system. Like SAAM, ATAM does not focus on a single quality attribute but rather on identifying tradeoffs between quality attributes. SALUTA can be integrated into these existing techniques.

6. Pattern Specifications

Pattern Specifications (PSs) [25, 26] are a way of formalizing the structural and behavioral features of a pattern. The notation for PSs is based on the Unified Modeling Language (UML) [26]. A Pattern Specification describes a pattern of structure or behavior and is defined in terms of roles. A PS can be instantiated by assigning modeling elements to play these roles. The abstract syntax of UML is defined by a UML metamodel. A role is a UML metaclass specialized by additional properties that any element fulfilling the role must possess. Hence, a role specifies a subset of the instances of the UML metaclass. A PS can be instantiated by assigning UMLmodel elements to the roles in the PS. A model conforms to a pattern specification if its model elements that play the roles of the pattern specification satisfy the properties defined by the roles. Pattern specifications can be defined to show static structure or dynamic behavior. Here we concern with specifications of behavior but it should be noted that any class roles participating in pattern specifications must be defined in a Static Pattern Specification (SPS), which is the PS equivalent of a class diagram.

An Interaction Pattern Specification defines a pattern of interactions between its participants. It consists of a number of lifeline roles and message roles which are specializations of the UML metaclasses *Lifeline* and *Message* respectively. The IPS in Fig. 4 formalizes the Observer pattern. Role names are preceded by a vertical bar to denote that they are roles.

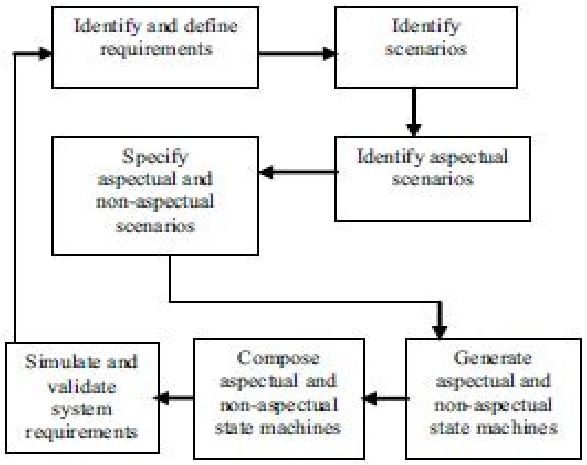
Identify and define requirements
Identify scenarios
Identify aspectual scenarios
Specify aspectual and non-aspectual scenarios
Generate aspectual and non-aspectual state machines
Compose aspectual and non-aspectual state machines
Simulate and validate system requirements

Fig. 3 Pattern Specification Process Model

Each lifeline role is associated with a classifier role, a specialization of a UML classifier. Fig. 4 shows an example of an IPS and a conforming sequence diagram.

The separation of specification concerns are maintained at the state machine level with composition of the functional need and non-functional need of requirements from the scenario level, the state machines need never be seen by the requirements engineer. Composition is specified purely in terms of scenario relationships and the composed state machine of the execution of the requirement and cancellation that are generated can be hidden. This has advantages for requirements engineers not trained in state-based techniques.

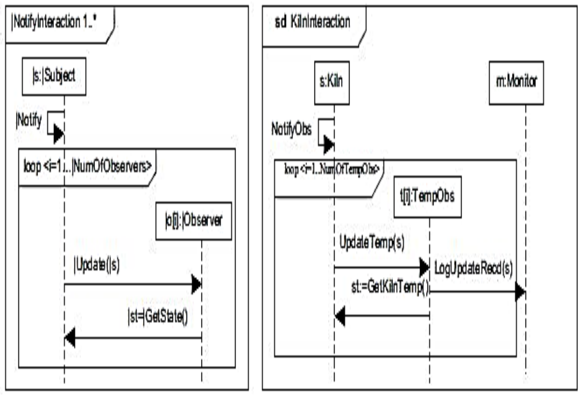

Fig. 4 Conforming Sequence Diagram

An IPS can be instantiated by assigning concrete modeling elements to the roles.

7. Functional and non-functional patterns

Non-functional requirements (NFRs) are pervasive in descriptions of design patterns. They play a crucial role in understanding the problem being addressed, the tradeoffs discussed, and the design solution proposed. However, since design patterns are mostly expressed as informal text, the structure of the design reasoning is not systematically organized. In particular, during the design phase, much of the quality aspects of a system are determined. Systems qualities are often expressed as non-functional requirements, also called quality attributes e.g. [28,29]. These are requirements such as reliability, usability, maintainability, cost, development time, and are crucial for system success. Yet they are difficult to deal with since they are hard to quantify, and often interact in competing, or synergistic ways.

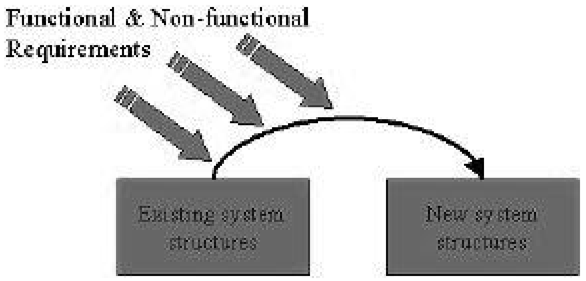
Functional & Non-functional Requirements
Existing system structures
New system structures

Fig. 5 Non-functional patterns as Requirements

During design such quality requirements appear in design tradeoffs when designers need to decide upon particular structural or behavioral aspects of the system. Applying a design pattern may be understood as transforming the system from one stage of development to the next. A good design needs the identification of architectural design decisions that improve usability, such as identification of usability patterns [29].

8. Conclusion

Use cases are a popular requirements modeling technique, yet people often struggle when writing them. They understand the basic concepts of use cases, but find that actually writing useful ones turns out to be harder than one would expect. One factor contributing to this difficulty is that we lack objective criteria to help judge their quality. Many people find it difficult to articulate the qualities of an effective use case. We have identified approximately three-dozen patterns that people can use to evaluate their use cases. We have based these patterns on the observable signs of quality that successful projects tend to exhibit. Construction guidance is based on use case model knowledge and takes the form of rules which encapsulate knowledge about types of action dependency, relationships between actions and flow conditions, properties of objects and agents, etc. Based on this knowledge rules, help discovering incomplete expressions, missing elements, exceptional cases and episodes in the use case specification through pattern specification. They support the progressive integration of scenarios into a complete use case specification.

References

[1] C. Wharton, J. Rieman, C. H. Lewis, and P. G. Polson, The Cognitive Walkthrough: A practitioner's guide., in Usability Inspection Methods, Nielsen, Jacob and Mack, R. L., John Wiley and Sons, New York, NY., 1994.

[2] D.E. Perry, A.L.Wolf, 'Foundations for the Study of Software Architecture', Software Engineering Notes, Vol. 17, No. 4, pp. 40-52, October 1992.

[3] E. Folmer and J. Bosch, Architecting for usability; a survey, Journal of systems and software, Elsevier, 2002, pp. 61-78.

[4] F. Buschmann, R. Meunier, H. Rohnert, M.Stahl, Pattern-Oriented Software Architecture - A System of Patterns, John Wiley & Sons, 1996.

[5] F. Buschmann, R. Meunier, H. Rohnert, M.Stahl, Pattern-Oriented Software Architecture - A System of Patterns, John Wiley & Sons, 1996.

[6] G. Booch, Object-Oriented Analysis and Design with Applications (2nd edition), Benjamin/Cummings Publishing Company, 1994.

[7] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Object Technology Series, Addison-Wesley, October 1998.

[8] I. Jacobson, et. al., Object-oriented software engineering. A use case approach, Addison- Wesley, 1992.

[9] J. Nielsen, Heuristic Evaluation., in Usability Inspection Methods., Nielsen, J. and Mack, R. L., John Wiley and Sons, New York, NY., 1994.

[10] J. Nielsen, Usability Engineering, Academic Press, Inc, Boston, MA., 1993.

[11] J. Bosch, Design and use of Software Architectures: Adopting and evolving a product line approach, Pearson Education (Addison-Wesley and ACM Press), Harlow, 2000.

[12] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-oriented modeling and design, Prentice Hall, 1991.

[13] K. Rubin, A. Goldberg, "Object Behaviour Analysis", Communications of ACM, September 1992, pp. 48-62.

[14] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, Addison Wesley Longman, Reading MA, 1998.

[15] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison Wesley,1998.

[16] M. Shaw, D. Garlan, Software Architecture - Perspectives on an Emerging Discipline, Prentice Hall, 1996.

[17] P.B. Kruchten, 'The 4+1 View Model of Architecture,' IEEE Software, pp. 42-50, November 1995.

[18] R. Gamma et. al., Design Patterns Elements of Reusable Design, Addison.Wesley, 1995.

[19] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, The Architecture Tradeoff Analysis Method, Proceedings of ICECCS'98, 8-1-1998.

[20] R. Kazman, G. Abowd, and M. Webb, SAAM: A Method for Analyzing the Properties of Software Architectures, Proceedings of the 16th International Conference on Software Engineering, 1994.

[21] R. S. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill, NY,1992.

[22] T. K. Landauer, The Trouble with Computers: Usefulness, Usability and Productivity., MIT Press., Cambridge, 1995.

[23] W. Li and S. Henry, OO Metrics that Predict Maintainability, Journal

of systems and software, Elsevier, 1993, pp. 111-122.

[24] R. France, D. Kim, S. Ghosh and E. Song, "A UMLBased Pattern Specification Technique", IEEE Transactions on Software Engineering, Vol. 30(3), 2004.

[25] D. Kim, R. France, S. Ghosh and E. Song, "A UMLBased Metamodeling Language to Specify Design Patterns", Proceedings of Workshop on Software Model Engineering (WiSME), at UML 2003, San Francisco, 2003.

[26] Unified Modeling Language Specification, version 2.0 January 2004, In OMG, *http://www.omg.org* [15] J. Warmer and A. Kleppe, The Object Constraint Language: Getting Your Models Ready for MDA, 2nd Edition, Addison-Wesley, 2003.

[27]Boehm BW. Characteristics of software quality. North-Holland Pub. Co., Amsterdam New York 1978.

[28]Bowen TP. Wigle GB. Tsai JT. Specification of software quality attributes (Report RADC-TR-85-37). Rome Air Development Center, Griffiss Air Force Base NY 1985.

[29] Architecting for usability; a survey, *http://segroup.cs.rug.nl*.

Prof. Dr.M.Punithavalli is currently the Director & Head of Department of Computer Applications, Sri Ramakrishna College of Arts and Science for women, Coimbatore, India. She is actively working as the Adjunct Professor in the department of Computer Applications of Ramakrishna Engineering College, India.

Lect. R.V.SivaBalan is currently working as the Lecturer in the Department of Computer Applications, Narayanaguru College of Engineering, India. He is a research scholar in Anna University Coimbatore, India.