Journal of Computer Science 7 (5): 619-628, 2011
ISSN 1549-3636
© 2011 Science Publications

Decision Based Development of Productline: A Quintessence Usability Approach

[1]R.V. Siva Balan and [2]M. Punithavalli
[1]Department of Computer Applications,
Narayanaguru College of Engineering, Kanyakumari, India
[2]Department of Computer Applications,
SNS Raja Lakshmi Engineering College, Coimbatore, India

Abstract: Problem statement: A well designed user interface is comprehensible and controllable, helping users to complete their work successfully and efficiently and to feel competent and satisfied. To improve the usability of a software system, usability patterns can be applied. However, there are short comes shows that the software architecture of a system restricts certain usability patterns from being modified after implementation. Several of these usability patterns are “architecture sensitive”, such modifications are costly to implement due through their structural impact on the system. So we practice the pattern oriented usability design with considering the dependencies between the design decisions relevant to the product line business objects which has its impact on the usability criterion. Dependencies between the rationale decisions for the architecture sensitive usability patterns can be maintained for future reference. **Approach:** While going for the usability patterns, the decisions behind the pattern selection should be specified. We address the issues by analyzing the quality based models that determines the design rationale and their dependencies. We use QDK methodology to preserve the specifications of decisions and all their inter dependencies along with the knowledge rule. **Results:** Preserving the specifications of decisions and all their inter dependencies with knowledge rules will support the evolution and maintenance of such productline systems. Explicit evaluation of usability during architectural design prevents part of the high costs incurred by adaptive maintenance activities once the system has been implemented. **Conclusion:** Capturing knowledge by this means provides the basis for justification, learning and re-uses of the knowledge rules for further design decisions.

Key words: Usability patterns, coded into software, architectural design decisions, usability attribute, comprehensive survey, architectural knowledge, complementary activities

INTRODUCTION

In the Study by (Perry and Wolf, 1992) 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. Bass *et al*. (1998) 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.

The design rationale abstracts the emergence new forces such as the controller object for the corresponding business object and the impact of the controller towards the usability. Design decision on the controller object relevant to business object can be accessed by the user through the interfaces (Fig. 1),

Corresponding Author: R.V. Siva Balan, Department of Computer Applications, Narayanaguru College of Engineering, Kanyakumari, India

when the interface is of usability pattern; the impact of design decision on this pattern is derived for maintenance.

Software architectures are typically described in one or more software architecture documents. Architecture documentation approaches provide guidelines on which aspects of the architecture should be documented and how this can be achieved (Clements *et al*., 2003). However, these approaches document only partially what an architecture is, as they lack rationale, rules, constraints and a clear relationship to the requirements (Tyree and Ackerman, 2005; Van Der Ven *et al*., 2006). This information is valued by practitioners (Tang *et al*., 2005) and helps in future design decision making (Falessi *et al*., 2006). Software architects have widely used architecture modeling tools for producing and documenting the models of their system's architecture. At present, there is lack in the documentation generated by typical architecting processes as they never record the design decisions relevant to the usability patterns that led to particular architecture focusing on quality attributes. This problem is referred in (Clements *et al*., 2003), which states the importance for recording design rationale. In the past, typical architecting tools don't include design rationale as a first class entity that has to be documented and only one of the five tools discussed in (Jansen and Bosch, 2004) provides limited support for capturing first class architectural design decisions.

Patterns from nuggets: Software engineers have a tendency to repeat their successful designs in new projects and to avoid the less successful designs. In fact, these different styles of designing software systems could be common for distinct practitioners. This has been observed in (Gamma, 2005) where a number of systems were studied and common solutions to similar design problems were documented as design patterns. They were a catalyst that propelled object oriented development into the mainstream. They helped the developers to understand the real value of inheritance and how to use it effectively. Patterns provided insight to how to construct flexible and resilient software systems. With nuggets of wisdom, such as "Favor object composition over class inheritance" and "Program to an interface, not an implementation", patterns helped a generation of software developers adopt a new adept programming paradigm.

Over the past several years, there have also been a number of object oriented design principles that have emerged. And many of these design principles are embodied within design patterns. 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

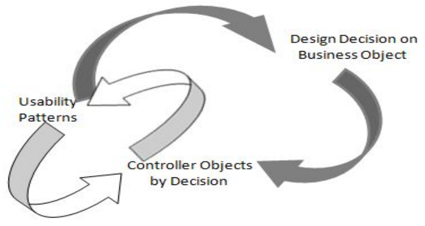
Design Decision on Business Object
Usability Patterns
Controller Objects by Decision

Fig. 1: Tracing decision correspondence for usability

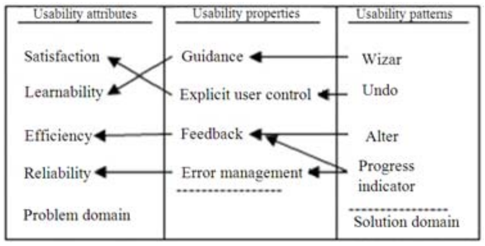

Fig. 2: Relationship between usability patterns, properties and attributes (Folmer and Bosch, 2004; ICSE, 2003)

are used (Bass *et al*., 1998). First, it provides an artifact that allows discussion by the stakeholders very early in the design process. Second, it allows for early assessment (Fig. 2) of quality attributes (Folmer and Bosch, 2004). Finally, the design decisions captured in the software architecture can be transferred to other systems.

A variety of pattern categories are recognized in software pattern community. Note, nevertheless, that a design pattern can be seen as a unique or original solution. Design patterns have become an increasingly popular choice for addressing OOD's limitations. Design patterns have a very close intact with the architectural design decisions. Abstracting the definition of design pattern, an architectural pattern can be defined as a description of the components of a design and the communication between these components to provide a solution for a usability pattern.

A comprehensive survey of the literature (Folmer and Bosch, 2004) revealed that different researchers have different definitions for the term usability attribute, but the generally accepted meaning is that a usability attribute is a precise and measurable component of the abstract concept that is usability. A well designed user interface is comprehensible and controllable, helping users to complete their work successfully and efficiently and to feel competent and satisfied. Effective user interfaces are designed based on principles of human interface design. The principles listed are consolidated from a wide range of published sources (Constantine and Lockwood, 1999; Cooper and Reimann, 2003; Jakob, 1994; Schneiderman, 1998) and are based on a long history of human-computer interaction research. Essentially, the usability properties embody the heuristics and design principles that researchers in the usability field have found to have a direct influence on system usability. These properties can be used as requirements at the design stage, for instance by specifying: "the system must provide feedback". They are not strict requirements in a way that they are requirements that should be fulfilled at all costs. It is up to the software engineer to decide how and at which levels these properties are implemented by using usability patterns of which it is known they have an effect on this usability property.

If we consider a system's architecture as a set of architectural decisions, the most significant ones concern the satisfaction of quality attributes (Jansen *et al*., 2007). The process of architectural design has been characterized as making a series of decisions that have system wide impact. Bosch (2000) notes that a decision may add components to the architecture, impose functionality on existing components, add requirements to components, or add constraints to part or all of the software architecture. The Unified Modeling Language is a very important standard for the development of software systems. UML originated from its main predecessor approaches (Booch, 1994; Jacobson, 1992). It is a graphical modeling language supporting many phases in the software development cycle by offering diagrams and language features meeting the special needs in respective phases. Many commercial tools for UML are available. Knowledge Discovery Metamodel (KDM) addresses in part the integration challenge by offering a common language-independent intermediate representation of the sources of decisions.

Architectural knowledge: The importance of design rationale in software architecture was early stated in the nineties by (Perry and Wolf, 1992), which consider the rationale as a relevant piece for understanding the design. More recently, (Kruchten *et al*., 2006) have modernized this idea as they state that Architectural Knowledge (AK) = Design Decisions + Design. Hence, the importance of design rationale that has been neglected in the past becomes now relevant for most modern architecting processes. Therefore, recording, using, managing and documenting architectural design decisions are new complementary activities (e.g.: capturing knowledge, sharing) that should be carried out in parallel to typical architecture modeling tasks. These new challenges need to deal with many obstacles in order to overcome those barriers that try to impede the transfer of implicit mental models from the architect's expertise to explicit and documented knowledge. Along with those knowledge we need to track and record the reference for the usability patterns relevant with the decision objects made up for the behavioral aspects of the decisive objects. The goal to document implicit impact of decisions has an overhead that should be taken into account if we want to estimate the potential savings in typical maintenance, as the users' requirement changes. Design rationale captures the reasons behind design decisions. They show how the system design satisfies the requirements, why certain design choices are selected over alternatives and how environmental conditions influence the system architecture.

During the process of detail design, decisions are made and justified but the justifications are often unrecorded and are lost over time (Perry and Wolf, 1992; Tyree and Ackerman, 2005). System and software architecture design often involves many implicit assumptions (Roeller *et al*., 2005) and convoluted decisions that cut across different parts of the system. A change in one part of the architecture design could affect the parts of the business objects, controller objects (by decision) and the usability pattern. A simple shift of an implicit assumption might affect seemingly disparate design objects and such change impacts could not be identified easily. This intricacy is quite different from detailed software design where usually the design or program specifications are self-explanatory. At the system and software architecture level, there are a multitude of influences that can be implicit, complex and intractable. In a survey on architecture design rationale (Tang *et al*., 2006), that is found 85% of architects agreed that the use of design rationale is important in justifying design and 80% of the respondents said they fail to understand the reasons of a design decision without design rationale support. Furthermore, 74% of respondents forget their own design decisions half the time or more often. These results indicate the need to capture the design rationale for system maintenance. The erosion of architecture design rationale can result in ill-informed decisions because the original design reasoning was missing. As a result, it may lead to inconsistent design and violations of design constraints. The impacts can be serious because architecture design is fundamental to a system. Consequently, the rectification of errors can be very costly.

Usability oriented design specification: 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. The architecture of a software system captures early design decisions. In usability-oriented requirements engineering, the relation between goals and requirements is represented explicitly. Since quality needs may conflict, this requires resolution strategies to obtain a satisfactory compromise (Lamsweerde, 2000; Mylopoulos *et al*., 2001). Representation schemas used in usability-oriented requirements engineering have also been used to represent dependencies between quality goals and architectural styles.

Unified Modeling Language (UML) is a standardized general-purpose modeling language in the field of software engineering. UML combines techniques from data modeling, business modeling, object modeling and component modeling. It can be used with all processes, throughout the software development life cycle and across different implementation technologies. Model-Driven Architecture (MDA) (Alti *et al*., 2007) is a software design approach for the development of software systems. It provides a set of guidelines for the structuring of specifications, which are expressed as models. Knowledge Discovery Metamodel (KDM) addresses in part the integration challenge by offering a common language-independent intermediate representation. Knowledge Discovery Metamodel defines the software development database format which can be used for software asset management and software asset tracking. Glossaries may also be used to specify captured detail rules. KDM also allows incremental multi-phase analysis of the same software system by multiple tools, where for the advanced analysis phases the KDM repository is both the input as well as the output of the analysis.

Quintessence of SAAM, ATAM and six sigma: A particular method (Fig. 3) for doing a scenario-based architectural analysis is (Kazman *et al*., 1996) Software Architecture Analysis Method (SAAM). SAAM was originally developed to enable comparison of competing architectural solutions. Once the scenarios have been created, we then need to classify them as direct (i.e., those that can be satisfied by executing the system being developed) or indirect (i.e. those which require a change to some of the components or connections within the architecture). The direct/indirect classification is a first indication of the fitness of an architecture with respect to satisfying a set of scenarios. When two or more indirect task scenarios necessitate changes to some component of a system, they are said to interact. Scenario interaction is an important consideration because it exposes the allocation of functionality to the product's design. In a very explicit way it is capable of showing which modules of the system are involved in tasks of different nature. High scenario interaction reveals a poor isolation of functionality in a particular component of a design, giving a clear guideline on where to focus the designer's subsequent attention (Table 1).

Thus the SAAM reveals the dependencies between the Usability Objects with the Business Objects. This depicts the need for the traceability between those two objects through the controller when it comes to the specification of design decisions. Another method (Fig. 4) ATAM (Kazman *et al*., 1998) depicts the saturation degree for the tradeoffs between the quality attributes. All design, in any discipline, involves tradeoffs; this is well accepted. What is less well understood is the means for making informed and possibly even optimal tradeoffs. Design decisions are often made for non-technical reasons: strategic business concerns, meeting the constraints of cost and schedule, using available personnel and so forth. The ATAM also serves as a vehicle for the early clarification of requirements. As a result of performing an architecture tradeoff analysis an enhanced understanding of systems' ability to meet its requirements. ATAM also have a documented rationale for the architectural choices made, consisting of both the scenarios used to motivate the attribute-specific analyses and the results of those analyses.

The international organization for standardization and the international electro technical commission ISO/IEC 9126-1 categorize software quality attributes into six categories: functionality, reliability, usability, efficiency, maintainability and portability. In the standard, usability is defined as "the capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions." The Six Sigma Quality movement takes this very much to heart. In fact, six sigma advocates believe that for many processes, there should be six sigma's between the mean and the specification limits, so that the process is only making a few bad "parts" in every million. You can, of course, do that by relaxing the specifications, but that isn't usually the way to please customers. Instead, the variation in the process needs to be driven towards zero, so that the histogram gets narrower and fits more comfortably inside the spec limits. Post-task satisfaction can be measured across multiple dimensions using semantic distance scales including the After Scenario Questionnaire. Apply the Six Sigma Continuous Method when standardizing 5-point satisfaction scale data. For example, assuming that the average post-task satisfaction for a task attempted by 20 participants is 3.6 and the standard deviation is 1.1, we can calculate the defective rate for task satisfaction as follows:

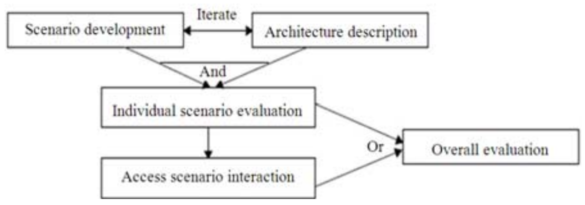

Fig. 3: SAAM method (Kazman *et al*., 1996)

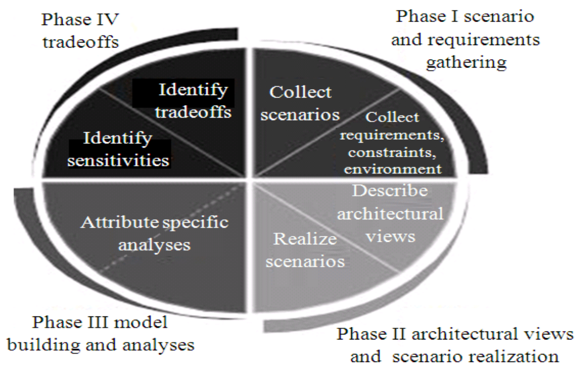
Phase IV tradeoffs
Phase I scenario and requirements gathering
Identify tradeoffs
Collect scenarios
Collect requirements, constraints, environment
Identify sensitivities
Attribute specific analyses
Describe architectural views
Realize scenarios
Phase III model building and analyses
Phase II architectural views and scenario realization

Fig. 4: Steps of the architecture tradeoff analysis method (Kazman *et al*., 1998)

(Sample Mean-Spec) / St. Dev = z-score$\rightarrow$ (3.6 – 4)/1.1 = -0.364$\rightarrow$ 36% on a standardized z-table = 36% Quality Level (64% Defective Rate).

Since the average of the sample was below the goal, the z-score is negative. The process sigma is then simply: -0.364 + 1.5 = 1.14 sigma. Now that disparate usability metrics can be expressed in standardized terms of sigma values or quality levels, there are two major benefits. First, since the standardized metrics were derived from the user-defined goals, the analyst can see which metrics are falling short and which are exceeding these goals. Second, the common scale makes reporting and ranking much easier than with the raw data.

MATERIALS AND METHODS

Design decision and knowledge-rules specification: The work in this Study is motivated by the fact that the pattern work 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. Studies (Hana and Huang, 2007) confirm that a significant large part of the maintenance costs of software systems is spent on dealing with usability issues. Generally, a motivational reason can be a requirement, a goal, an assumption, a constraint or a design object. It is important to represent motivational reasons explicitly as inputs to the decisions so that they are given proper attention in the decision making process. A design decision creates the knowledge Rules; and knowledge rules are justified by the alternatives and tradeoffs. The knowledge rule encapsulates the details of the justification. It contains a description of the issues addressed by the decision, the arguments for and against an issue/risk and the different alternatives that have been considered. Once a decision is made, the result of a decision is a design outcome or solution. A design outcome should be explicitly represented in the architecture design as an architecture element.

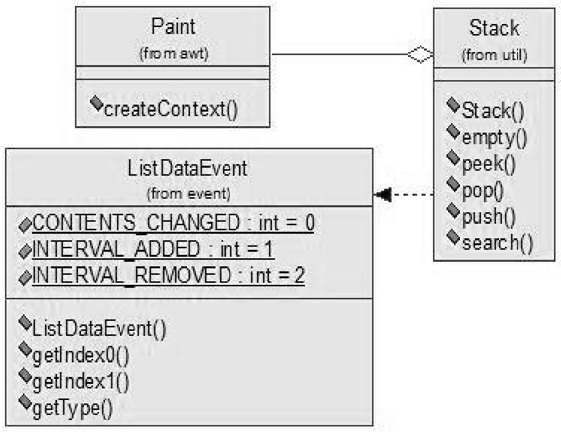

Fig. 5: Undo Command on STACK Decision

In our scenario, we intend to inculcate an undo stack (Fig. 5) as pattern-usability concern. The design decision, analytical information and the knowledge rules should be specified for the future maintenance of our software; in such context we can use the KDM metamodel which is in compliance with most of the analysis tools. But here we emphasize on the usability quality attributes on the scenario oriented development. Thus the definitions of domain specific, application-specific, or implementation-specific knowledge are specified as a comprehensive set of knowledge rules. Glossaries are also be used to specify the captured rules, constrains and context dependencies in detail as necessity.

In a study of architecture evaluations, (Bass *et al*., 2006) report that most risks discovered during an evaluation arise from the lack of an activity, not from incorrect performance of an activity. Categories of risks are dominated by oversight, including overlooking consequences of decisions. Many of the overlooked consequences are associated with quality attributes.

Table 1: Scenario evaluation from SAAM (Kazman *et al*., 1996)

Scenario	Description	Direct/indirect	Changes
3	Port to another operating system	Indirect	All components that call win31 must be modified; specifically: main, visdiff and ctrls. If the target operating system does not support OWL then either OWL needs to be ported, or all components that call OWL, specifically: main and hook. If the new operating system is not supported by Novell's software then WRCS will have to be modified to work with a new networking environment
4	Make minor modifications to the user interface	Indirect	This will require changes to one or more of those, components which call the win31 API, specifically: main diff and ctrls.
5	Change access permissions for a project	Direct	
6	Integrate with a new development environment	Indirect	This requires changes to hook, as well as the addition of a module along the lines of bcext, mcext, and cbext, which connects the new development enjoinment to hook

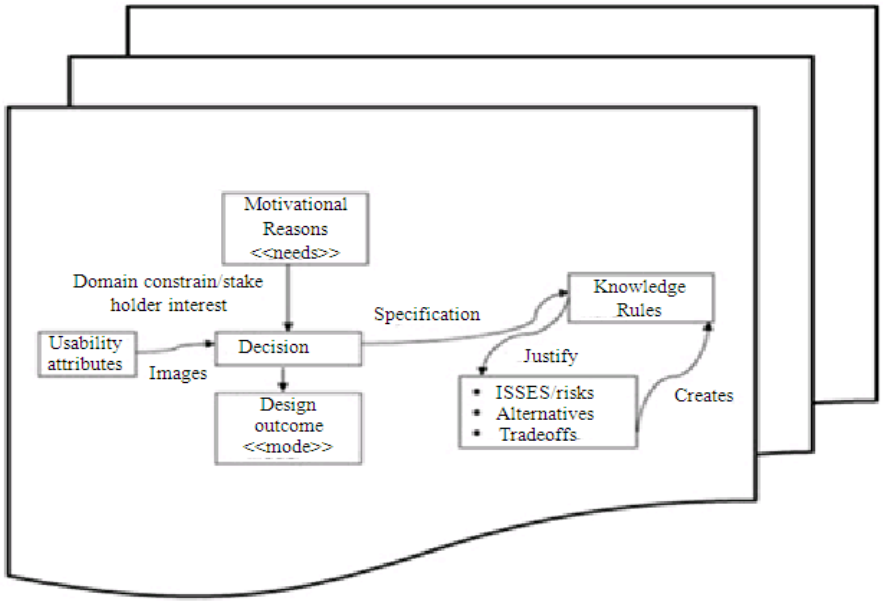

Fig. 6: Knowledge Rules conceptual model

Their top risk themes included availability, performance, security and modifiability. The iterative refinement of design decision (D), by means of the quality needs (Q) leads to the specification of (K) knowledge.

Most architectural decisions have multiple consequences; result in additional requirements to be satisfied by the architecture, which need to be addressed by additional decisions (Jansen and Bosch, 2005). Some are intended, while others are side effects of the decision. Some of the most significant consequences of decisions are those that impact the quality attributes of the system. Garlan (2000) calls them key requirements. The activity (Fig.7), QDK (Quality Needs Motivate Design decision to Knowledge-rules Specification), explores the quality-impact design decision for usability. The following steps are undergone:

- Define Release the first step in the recovery method is to define the current release of the system under consideration. A release contains the artifacts of the system at a specific moment at a time
- Specify Traceability Details to gain tacit knowledge from the explicit knowledge representation. Here tacit knowledge are the motives and ideas still in the head of the developer and the explicit knowledge are specified in various documents during previous activities (like analysis)

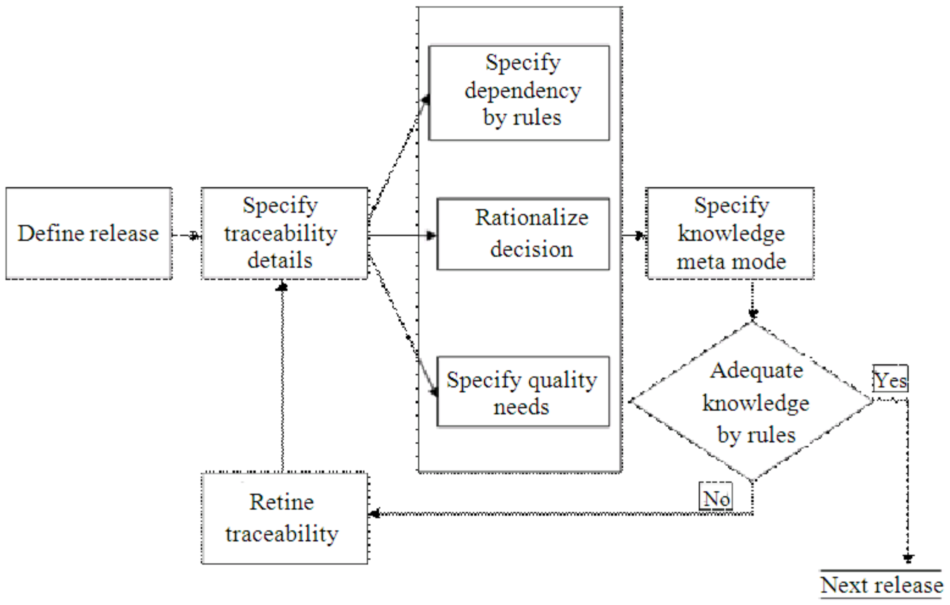
Define release
Specify traceability details
Specify dependency by rules
Rationalize decision
Specify quality needs
Specify knowledge meta mode
Adequate knowledge by rules
Yes
No
Retine traceability
Next release

Fig. 7: Quality Needs motivate Design decision to Knowledge-rules Specification' process

- Specify Dependencies by Rules Identify the dependencies among the decisions by the four-dimensional rules.
- Rationale Decision Try out the best rationale from the alternatives and specify it; quote the alternatives.
- Specify Quality Needs Specify the quality needs and the impact of the same on the rationale.
- Specify Knowledge Metamodel From the steps 3, 4 and 5 represent the Metamodel for the rationale.
- Check whether adequate decisive knowledge obtained. If not refine the traceability and go to step 2.
- Next Release Go to step 1 with new release.

Brainstorming for decisions: A choice description and the derived rationale realize one or more dependent requirements. Now the solutions are the set of prescription to the architecture. Thus this becomes a result of decision process which bride between requirement engineering and detail design. By this the usability design decisions and the relations between the elements of design decision are studied and documented adept. For each decision, the motivation (Fig. 6) and the choices lead to a decision which may become an alternative solution relying the context of application of decision.

First, collectively define the overall user experience separate from architectural or design elements to engage the entire team in defining design principles and the user experience strategy that will filter down into all design releases. This helps keep everyone on the same page throughout the entire design process and helps to make sure that the visual design reflects design principles based on adapted quality attributes.

Second, confirm all roles participate in generative design activities, if, each team member sketches out 10 ideas for a product in 5 min and then the team regroups to share ideas and discover patterns. The idea is to get as many quick ideas on the table as possible to discuss overall themes and experience approaches within the scheduled hour. Encourage all team members to participate in these activities to get input from all team members early in a project and to foster collaboration before any design decisions are solidified upon the usability patterns.

Now, set priority for each of the idea and elect the most tacit one.

Next, share early concepts with designers while they're still in progress should involve interface designers as early as possible to get their feedback about how a usability object will be structured both at a global level and at the unit (page) level. These interface designers can provide any unique perspectives in regards to information hierarchy, element placement, traceability of design and how to create overall design cohesion. This can both help to refine early prototypes as well as give practitioners a better understanding of decisions that went into releases.

Then follow to conduct user research that can influence usability design early in the process as it is fairly common for the interface designers to validate user behavior and mental models with techniques of conceptual usability tests; we often neglect to perform early techniques that can specifically influence usability design. Performing research techniques such as user-generated mood boards help and feed the usability design process and help designers start to think about ideas even before prototypes are complete.

As next step, discuss usability tradeoffs of high-level design approaches, so that before a designer craft the interfaces, the team collectively can discuss potential usability implications of various approaches. The team should also discuss how proposed style guides should be adjusted to account for things such as the usability attributes, which we intend to retain.

Finally, document the decisions, sources of motivations and interaction models that impact the interface design to explain the intended unit level interactions and any other specific details that might influence the interface design. And Monitor the effectiveness of the design to influence future iterations.

The system's documentation is the rich source of the information. Perfect documentation refers to the actual state of the decisions. The organizations culture plays big role in the process of documentation and have standard documentation policy. Information about the development process are capture in version control system; Referring to the version control may produce rich traceable dependencies of decisions.

Knowledge rules: Here we have to consider two decisions simultaneously. The First one is the design decision up on the business object and the second one is the decision on corresponding design pattern which is very traceable to it. The dependencies between the decision objects' and the usability patterns play a vital role in determining the preferences and the rules to obtain user oriented rationale. For each dependency, four-dimension rules are applicable while deriving knowledge:

- Exist-Exist: By this rule, only if the design decision live, then the decision on usability pattern will survive until the design decision is in priority list
- Exist-Emerge: By this rule, if the design decision exists to be in the priority list then the decision on pattern will emerge as the successor of it
- Emerge-Emerge: By this rule, both the decisions emerge at a time; possibly from single source of traceability
- Emerge-Exit: By this rule, if the design decision emerged, then the decision on the usability pattern is dropped out

These dependencies may be categorized into external, internal, strong and weak dependencies from the available traceable reports.

RESULTS

Usability Inspections are essential for early detection of defects in UI design, but they require sound usability knowledge. Usability Patterns are the state-of-the-art format for describing usability knowledge. Thus, it seems obvious to use them as a means for evaluating the design of user interfaces. And successful development of a usable software system therefore must include creating a software architecture that supports the right level of usability. So properly documented evidence should exist for the architectures designed focusing on usability, to support software architects in creating and maintaining the software architecture that supports usability. Explicit evaluation of usability during architectural design prevent part of the high costs incurred by adaptive maintenance activities once the system has been implemented and leads to architectures with better support for usability. Design for change is a well-known adagium in software engineering. We separate concerns, employ well-designed interfaces and the like to ease evolution of the systems we build. We model and build in changeability through parameterization and variability points as in product lines. These all concern places where we explicitly consider variability in our systems. We conjecture that it is helpful to also think of and explicitly model invariability, things in our systems and their environment that we assume will not change. In particular, we show how we can explicitly model assumptions in an existing product family. From this, we derive Meta models to document assumptions. Finally, we show how this type of modeling adds to our understanding of the architecture and the decisions that led to it.

DISCUSSION

This activity, QDK (Fig. 7), explores the quality-impact design decision for usability. Most architectural decisions have multiple consequences; result in additional requirements to be satisfied by the architecture, which need to be addressed by additional decisions. Some are intended, while others are side effects of the decision. Some of the most significant consequences of decisions are those that impact the quality attributes of the system. We call it as Discovery of Knowledge, to be recorded in Knowledge rule Specification (Ks). This impact may be the intent of the decision; for example, one may choose to use a role-based access control model in order to satisfy quality attributes. One of the key challenges in dealing with such consequences is the vast amount of knowledge required to understand their impact on all the quality attributes. Architectural design decisions are concerned with the application domain of the system, the architectural styles and patterns used in the system, components for product line and other infrastructure selections as well as other aspects needed to satisfy the system requirements.

CONCLUSION

Usability Pattern 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. Generally, a motivational reason behind usability patterns can be a requirement, a goal, an assumption, a constraint. It is important to represent motivational reasons explicitly as inputs to the decisions so that they are given proper attention in the decision making process. Quality doesn't appear without mature design decisions. In the software world, perfect information and rational consumers are two concepts that seem quite valid in the development scenario of product line or automation software. In this case, software engineers are the rational consumers need to make engineering decisions. Decisions have to be made on the basis of information, so typically there is a lot of analysis required, much of which is manual. The attributes of the quality should be recorded among with the tacit decisions of the patterns along the way designing usability oriented architecture. These activities are to be carried out manually; because when the solution space is very large, practitioners go for automated process with case tools of development and typically impossible to do enough manual analysis to cover all possibilities. Knowledge rules model process try to capture the knowledge used the architecture construction. From a knowledge system perspective, making usability oriented design decision is seen as a decision process, which decide how the architecture is maintained and controlled. Capturing knowledge by this means provides the basis for justification, learning and re-uses of the knowledge rules for further design decisions. This model explicitly model the goal of the design decision process wants to satisfy, as well as the design decision and corresponding rationale. Design decision model provide basis to capture, describe and reason about the design decisions made during design process. As a result of our specification process, the software architecture documentation not only describes what the architecture contains; the design decisions underlying the architecture provide this why. Relevant set of facts for the decision will be fertile in the future phase of maintenance and version control.

REFERENCES

Alti, A., T. Khammaci and A. Smeda, 2007. Integrating software architecture concepts into the MDA platform with UML profile. J. Comput. Sci., 3: 793-802. DOI: 10.3844/jcsssp.2007.793.802

Bass, L., P. Clements and R. Kazman, 1998. Software Architecture in Practice. 1st Edn., Addison-Wesley, USA., ISBN-10: 0201199300, pp: 452.

Bass, L., R. Nord, W. Wood and D. Zubrow, 2006. Risk themes discovered through architecture evaluations. Information for the defense community. http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA456884

Booch, G., 1994. Object-oriented analysis and design with applications. 2nd Edn., Benjamin Cummings Pub. Co., USA., ISBN: 10: 0805353402, pp: 589.

Bosch, J., 2000. Design and use of Software Architectures: Adopting and Evolving a Product-Line Approach. 1st Edn., Addison-Wesley, USA., ISBN-10: 0201674947, pp: 354.

Clements, P., D. Garlan, R. Little, R. Nord and J. Stafford, 2003. Documenting software architectures: views and beyond. Proceedings of the 25th International Conference on Software Engineering, May 3-10, IEEE Xplore, USA., pp: 740-741. DOI: 10.1109/ICSE.2003.1201264

Constantine, L.L. and LA.D. Lockwood, 1999. Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design. 1st Edn., Addison Wesley, USA., ISBN-10: 0201924781, pp: 579.

Cooper, A. and R. Reimann, 2003. About face 2.0: The Essentials of Interaction Design. 2nd Edn., Wiley, USA., ISBN-10: 0764526413, pp: 540.

Falessi, D., G. Cantone and M. Becker, 2006. Documenting design decision rationale to improve individual and team design decision making: An experimental evaluation. Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering, (ISESE'06). ACM Press, New York, USA., pp: 134-143. DOI: 10.1145/1159733.1159755

Folmer, E. and J. Bosch, 2004. Architecting for usability: a survey. J. Syst. Software, 70: 61-78. DOI: 10.1016/S0164-1212(02)00159-0

Gamma, R., 2005. Design Patterns: Elements of Reusable Object-Oriented Software. 1st Edn., Addison-Wesley, USA., ISBN-10: 0201633612, pp: 395.

Garlan, D., 2000. Software architecture: A Roadmap. Proceedings of the Conference on the Future of Software Engineering, (CFSE'00), ACM New York, NY, USA., pp: 91-101. DOI: 10.1145/336512.336537

Hana, W.M. and S.J. Huang, 2007. An empirical analysis of risk components and performance on software projects. J. Syst. Software, 80: 42-50. DOI: 10.1016/j.jss.2006.04.030

Jacobson, I., 1992. Object-Oriented Software Engineering: A Use-Case Driven Approach. 1st Edn., ACM Press, USA., ISBN-10: 0201544350, pp: 524.

Jansen, A. and J. Bosch, 2004. Evaluation of tool support for architectural evolution. Proceedings of the 19th IEEE International Conference on Automated Software Engineering, Sept. 20-24, Linz, Austria, pp. 375-378. DOI: 10.1109/ASE.2004.35

Jansen, A. and J. Bosch, 2005. Software architecture as a set of architectural design decisions. Proceedings of the Conference on Software Architecture, (CSA'05), IEEE Xplore, USA., pp: 109-120. DOI: 10.1109/WICSA.2005.61

Jansen, A., J.V.D. Ven, P. Avgeriou and D.K. Hammer, 2007. Tool support for architectural decisions. Proceedings of the Working Conference on Software Architecture, Jan. 6-9, IEEE Xplore, India, pp: 4-4. DOI: 10.1109/WICSA.2007.47

Kazman, R. G. Abowd, L. Bass and P. Clements, 1996. Scenario-based analysis of software architecture. IEEE Software, 13: 47-55. DOI: 10.1109/52.542294

Kazman, R., M. Klein, M. Barbacci, T. Longstaff and H. Lipson *et al.*, 1998.The architecture tradeoff analysis method. Technical Report, CMU/SEI-98-TR-008.

Kruchten, P., P. Lago and H.V. Vliet, 2006. Building up and reasoning about architectural knowledge. Quality Software Archit., 4214: 43-58. DOI: 10.1007/11921998_8

Lamsweerde, A.V., 2000. Requirements engineering in the year 00: A research perspective. Proceedings of the International Conference on Software Engineering, Jun. 4-11, IEEE Xplore, Ireland, pp: 5-19. DOI: 10.1109/ICSE.2000.870392

Mylopoulos, J., L. Chung, S. Liao, H. Wang and E. Yu, 2001. Exploring alternatives during requirements analysis. IEEE Software, 18: 92-96. DOI: 10.1109/52.903174

Perry, D.E. and A.L. Wolf, 1992. Foundations for the study of software architecture. Software Eng. Notes, 17: 40-52. DOI: 10.1145/141874.141884

Roeller, R., P. Lago and H.V. Vliet, 2005. Recovering architectural assumptions. J. Syst. Software, 79: 1792-1804. DOI: 10.1145/141874.141884

Tang, A., M.A. Babar, I. Gorton and J. Han, 2005. A survey of the use and documentation of architecture design rationale. Proceedings of the 5th Working Conference on Software Architecture, (WICSA'05), IEEE Xplore, USA., pp: 89-98. DOI: 10.1109/WICSA.2005.7

Tang, A., M.A. Babar, I. Gorton and J. Han, 2006. A survey of architecture design rationale. J. Syst. Software. DOI: 10.1016/j.jss.2006.04.029

Tyree, J. and A. Ackerman, 2005. Architecture decisions: demystifying architecture. IEEE Software, 22: 19-27. DOI: 10.1109/MS.2005.27

Van Der Ven, J., A.. Jansen, J. Nijhuis and J. Bosch, 2006. Design decisions: The bridge between rationale and architecture. Rationale Manage. Software Eng., 3: 329-348. DOI: 10.1007/978-3-540-30998-7_16