ASIAN JOURNAL OF PHARMACEUTICAL AND CLINICAL RESEARCH

INNOVARE ACADEMIC SCIENCES
Knowledge to Innovation

Special Issue (April)

Online - 2455-3891
Print - 0974-2441

Full Proceeding Paper

EVALUATION OF CAPTURING ARCHITECTURALLY SIGNIFICANT REQUIREMENTS METHODS

SATHIS KUMAR B*

Department of Computer Science and Engineering, VIT University, Chennai, Tamil Nadu, India. Email: sathiskumar.b@vit.ac.in

Received: 23 January 2017, Revised and Accepted: 03 March 2017

ABSTRACT

Every software development organization strives for customer satisfaction. It is universally accepted that the success of software development lies in the clear understanding of the client requirements. During requirement elicitation and analysis stage, the system analyst identifies the functional and non-functional requirements from the customer. Security, usability, reliability, performance, scalability, and supportability are the significant quality attributes of a software system. These quality attributes are also referred as non-functional requirements. Only a few functional and quality attributes requirement help to identify and shape the software architecture. A software system's architecture is the set of prime design decisions made about the system. If the requirement influences the architectural design decision then, it is referred as architecturally significant requirement (ASR). Identifying and specifying all the possible ASR are important tasks in the requirement elicitation and analysis stage. In this research, general problems that are faced while capturing and specifying ASR in requirement elicitation and analysis is studied. Among the different requirement elicitation techniques, use case diagram has been identified and enhanced to solve the problem of capturing and specifying ASR during the requirement elicitation and analysis phase**.**

Keywords: Architecturally Significant requirement method, Use case diagram.

INTRODUCTION

Every software development organization strives for customer satisfaction. It is universally accepted that the success of software development lies in the clear understanding of the client requirements. Hence requirement engineering may be applied to obtain an in depth understanding of the client requirements, and it is further divided into 4 subtasks. They are a feasibility study, requirement elicitation, and analysis, specification and validation [1]. Feasibility study assesses the usefulness of the software system in the existing environment, requirement elicitation and analysis help us in discovering and understanding the requirements from the stakeholders involved in the project. Requirement specification acts as an aid in documenting the requirements in any prescribed form. It needs to be agreed on by the requirement engineer and the customers. Thus, these subtasks serve as the foundation for the further developmental activities of the software. Finally, validation helps to ensure the correctness of the collected customer requirements. In practice, requirement engineering tasks are an iterative process.

During the requirement elicitation, the requirement engineer works closely with the customer and the end users to understand the hardware constraints and the domain of work, functional, performance, security, and other quality requirements [2]. Different types of end users and the developing team members are the important stakeholders of the software project, and they view the requirements in a different perspective [3]. Based on this perspective, the requirements are classified as business, user and system requirements. Business requirements are gathered from the top level people of the organization, and they focus on vision, constraints, objectives and scope of the project. User requirements are gathered from the end user of the system, and they describe the services demand by the user [4]. It is documented in a user requirements document (URD) using natural language and diagrams. URD documents are written to provide an insight of the software project to the customers, and it is a key input for constructing system requirements. Gathering these requirements from the user is a very complex task for the requirement engineer. In general, users fail to articulate their requirements appropriately thus leading to an incomplete and ambiguous URD.

The system requirements are complete descriptions of the software system's functionalities. Software system requirements are categorized into functional and non-functional requirements. A software system's functional requirement describes an expected functionality requested by the customer and a non-functional requirement describes the effectiveness of the function provided by the system [5]. ISO 9126 lists 22 different quality attributes such as usability, efficiency, and portability. These are also referred as non-functional requirements. For example, in an inventory management system, order processing, and stock control are important functional requirements. User interface consistencies and maximum time to complete one order are the non-functional requirements.

Various stakeholders demand their requirements in different ways [6]. Hence, it leads to requirement conflicts. These conflicts are solved in the requirement elicitation and analysis stage by analyzing the necessity of the requirements, prioritizing the critical requirement and compromising the set of requirements after the negotiation [7].

After identifying the possible requirements, it is classified and grouped into logical clusters. The logical group and their relationship are called system architecture. According to Taylor *et al.* [8] "Software systems architecture is a set of principal design decisions made about the systems." In practice, it is difficult to separate requirement engineering and architectural design activity [9].

The paper is structured as follows: Architecturally significant requirement (ASR) is described in Section 2. In Section 3, related research on ASR is described. This is followed by methods for specifying in Section 4. In Section 5, the evaluation of ASR is described. Finally, most important findings are summarized in Section 6.

ASR

The core of the software development lies in the conversion of the requirement into software design. Requirement elicitation and analysis assure that the conversion process goes on smoothly. The end product of this stage is the software architecture design of the system, which is used as the input for the further detailed design and implementation [10]. Software architecture is a general abstraction of the system, and it creates a better communication between users and the developing team to understand the system completely.

The software architect takes the design decisions of the software architecture using the key requirements. The key requirements are also termed as ASR. The ASR plays a key role in the architectural design decision. Identifying and specifying all the possible ASRs in the requirement elicitation stage is essential [11]. ASR consists of high-level functional requirements, quality attribute requirements (QAR), technical and business constraints. According to Lattanze [12] "while functional requirements describe what the system must do, QAR describe how the system must do it." A technical constraint instructs the preference of the particular hardware, software, standards, operating systems and other constraints. Business constraints do not force a particular approach or solution but may imply to apply a particular approach or a solution. For example, "in order to reduce the software development cost of y, component x should be reused." These constraints are fixed before beginning the process of architect design and are documented using natural language in all the requirement elicitation methods.

The software architect takes the design decisions of the software architecture using the key requirements. The key requirements are also termed as ASR. The ASR plays a key role in the architectural design decision. Identifying and specifying all the possible ASRs in the requirement elicitation stage is essential [11].

The key requirements are the deciding factors for the architecture of the system. The key factors are termed as architectural drivers [13]. According to Dominick *et al.* (2002), some of the requirements are important to shape the architectures, and these requirements are termed as ASR. Both the architectural drivers and ASR terms are used for the same purpose. In this research, ASR term is used to specify the key requirements.

Only a few high-level functional requirements, all the QARs, technical and business constraints are the ASR that influence the architecture design decision [12]. High-level functional requirements specify a general description of functionality. For example, for an online shopping portal the high-level functional requirements are:

- The system shall provide placing the order
- The system shall support to track the order within 45 seconds
- The system shall support to standard payment methods and protocols
- The system shall support advertisement
- The system shall support any browser.

These high-level functionality requirements are described using traditional "shall" statements [14]. There are many ambiguities in these requirements. Some of functional, quality attributes requirements and constraints are specified individually or mixed.

ISO 9126 lists 22 different quality attributes such as usability, efficiency, and portability. According to Lattanze [12] "while functional requirements describe what the system must do, QARs describe how the system must do it." In general, QAR and functional requirements are always combined. It is difficult to specify the QAR alone. For example, "shall have high performance." Specifying this performance requirement statement without functional requirement is meaningless.

Technical constraint describes the preference of the particular hardware, software, standards and operating systems, etc. Business constraints do not force a particular approach or solution but imply to apply a particular approach or solution. For example, "in order to reduce the software development cost 'x' component 'y' should be reused." In the beginning of the project, technical and business constraints are fixed. The influence of the quality attributes and functional requirements are identified at the time of architectural design. Quality attributes are the highly influencing factor of the architecture compared to functional requirements. The following section describes various methods for capturing ASR.

RELATED RESEARCH ON ASR

Dominick *et al*. introduced the term ASR. The key requirements are the deciding factors for the architecture of the system and are termed as ASR. High level functional and QARs are important ASR.

Chen *et al*. [15] conducted an empirical study to characterize ASR. ASR is difficult to define and express. It tends to be articulated vaguely, neglected initially and hidden within other requirements.

Alistair [16] conducted a survey on requirement elicitation techniques. The results of the survey show that interviews, observations, workshops, protocols, scenarios, and prototypes produce natural language ambiguity.

Lange *et al*. [17] addressed the problems with unified modeling language (UML) descriptions based on his survey with the software architect. The survey results show that design choice information are scattered over multiple views, incomplete architecture views, and inconsistency in software architecture models.

Zhu and Gorton [18] proposed UML profiles for specifying non-functional requirements in a design model. This method is applicable only in design stage and not suitable for the requirement elicitation and analysis.

Sindre and Opdahal [19] introduced a method for specifying security requirements based on use cases. Filled oval symbol is used for misuse case and filled stick men is used to specify misusers. Threaten and mitigate relationship are introduced to specify the relationship between use case and misuse cases. The mitigation flow describes steps to handle the vulnerabilities. It clearly shows that misuse case [20] diagram and its specification is purely a technical document which can only be handled by an architect and a security designer and not by the customers.

Lange *et al*. [17] proposed factor table to specify factors influencing the architecture. Factor table only focuses in specifying the generic quality attribute influence factors, and it does not hint any functional requirements.

Barbacci *et al*. [21] proposed the quality attribute scenario for describing quality attributes with respect to operational context. The drawback of the quality attribute scenario is that it is represented using natural language. Hence, chances for representing ambiguous requirement are high.

METHODS FOR SPECIFYING ASR

The ASR plays a key role in architectural design decision. Determining and specifying all the ASR are a crucial task in the requirement elicitation and analysis stage [22]. ASR consists of high-level functional requirements, QARs, technical and business constraints. It is fixed before initiating the process of architectural design. These two details are documented using natural language in all the requirement elicitation methods.

The high-level functional requirements and QARs are recorded only in the requirement elicitation stage. Documenting these requirements properly will be helpful for the architect to take quick decision of the architectural design. Different requirement elicitation methods specify the architectural requirements in different ways. In this research, comparison of four approaches for the expression of ASR is done. They are ASR specification using natural language, use case analysis, quality attribute scenarios, and factor table method [23].

Natural language specification

Natural language specification portrays the functional requirements elaborately. Architect requires only the high-level functional requirements and not a detail requirement. Reading elaborated functional requirements takes a longer time for the software architect to capture the architecturally significant functions. Since there is no guidance for writing the contents, some practitioner writes functional requirement in detail, and others write briefly. Poorly written requirements lead to ambiguous, inconsistent, and incomplete functional as well as QARs. Some of the quality requirements such as performance, safety, and security are specified separately. Other QARs are not identified explicitly which are embedded with functional requirements. Hence, capturing QARs is a difficult task for the architect. Hence, missing out the key QAR is possible, due to scattered functional and QARs.

Quality attribute scenario

Only the high-level functional requirements are not enough to decide the architecture design. A QAR with respect to operational context is worthwhile in designing the architecture. The quality attribute scenario framework is an excellent method to express the quality attributes with respect to the operational context.

The quality attribute scenario is constructed using the set of framework elements. Stimulus element is useful to bring the conditions affecting the architecture. Source element specifies the sources of stimulus. An environmental condition illustrates the operational context. The architectural elements are directly or indirectly affected by the stimulus. System response element makes vivid of how the system responds to the stimuli. The response measure element depicts the measure of how the system responds to the stimuli.

Table 1 indicates the quality attribute scenario to "place order" performance requirement. As the "place order" event affects the architecture, it is a stimulus element. When the customer initiates the stimulus, he becomes the source of stimulus. Customer placing order during busy business hours is a relevant environmental condition. The external payment gateway is an external architectural element affected by the "place order" stimulus. Hence, a payment gateway is an architectural element. The successful completion of the "place order" is a system response. The payment completed within 45 seconds is a system measure.

This method is similar to natural language specification, but it uses the structured way of representing the QARs. Each quality attributes scenario focuses on any one of the quality attributes. Reading all the quality scenarios and creating a relationship between the quality attributes is a difficult task for a software architect.

Factor table

Lange *et al.* [17] introduced the factor table to analyze factors that influence the architecture design, which are classified as organizational, technological, and product factor. A factor table is constructed using 3 columns. The first column shows the influencing factors of the architecture. Some of the factors are flexible and changes can be made by the architect after the negotiation with the customer. However, few factors cannot be changed by the architect. These flexibility and changeability issues are indicated in the second column. The third column illustrates the impact of the factor or impact of the changed factor by architect.

Table 1: Quality attribute scenario for place order

Raw quality attribute	Place order shall complete quickly
Stimulus	Customer place the order
Source (s)	Customer
Relevant environmental condition	During peak business period
Architectural elements	Payment gateway software
System response	Payment successfully completed
System measure	Payment completed within 45 seconds

Table 2 shows the online shopping portal system performance requirements specified using factor table. The factor table is an effective format to specify all influence factors. The factor table is more generic and it does not describe the factors which affect the functional requirements. Hence, it is tough for the architect to take quick design decisions.

UML use case model

UML use case model [4] is a composition of use case diagram, use case specification, and supplementary documents [24]. The use case model is a communication medium between stakeholders and developing team [25]. The usages of use case model artifacts for capturing the ASR are discussed briefly in the next section.

Use case diagram

The use case is the core of the UML because it is the driver for constructing UML diagrams. It is a collection of related scenarios to achieve a particular goal [26]. The actor may be a machine or human who invokes the system [27]. Use cases can be represented either as case diagram or in a textual form. The textual form is termed as use case specification.

Use case diagram shows a static view of the system functionality. It also depicts the use case and actor relationships. Application development standards, quality attributes of the system, legal and regulatory requirements and other requirements that do not fit naturally into the use cases are specified in the supplementary document [24].

Fig. 1 shows a simple order processing system use case diagram. The customer can place and track the orders. In this use case diagram, the customer is the actor. "Place order" and "track order" are use cases.

The <include> and <extend> are useful to create relationship between use cases. The <include> relationship is helps to remove the similar set of events repeatedly. The repeated flow of events is moved into a separate use case called addition use case, and the primary use case is called base use case. If base use case includes an addition use case, then it should invoke the additional use case at least once before it finishes the steps. Fig. 2 shows the flow of <include> relationship.

In an order processing system, whenever the customer places an order, the system first validates the customer credentials. Similarly, whenever the customer tries to track the order, the system validates the customer

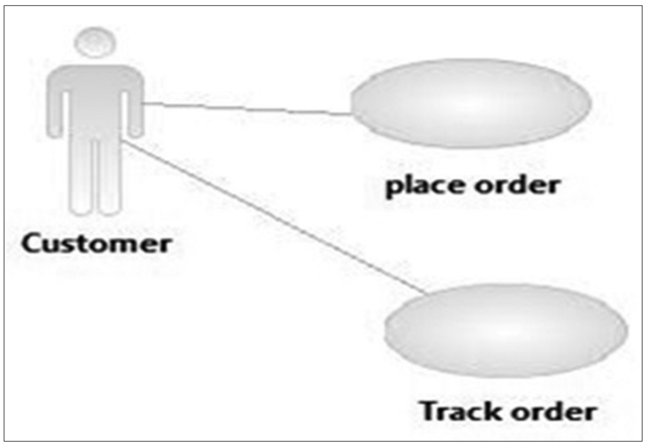

Fig. 1: Order processing system UCD

Table 2: Factor table for performance factor of the product

Product factor performance	Flexibility and changeability	Impact
More users accessing online portal at the same time	Customers demand more and more performance	There is a tightness between performance and flexibility throughout the system

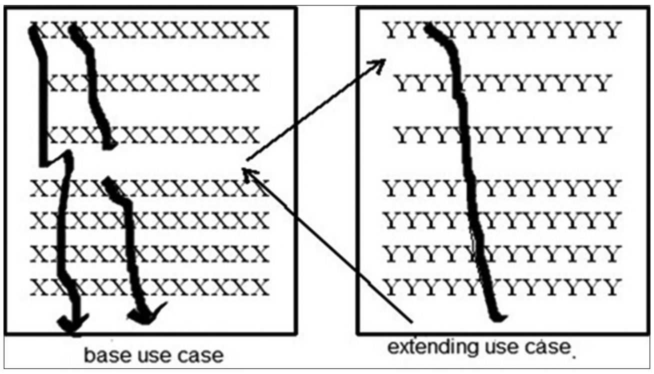

Fig 2: Control flow of <include> relationship

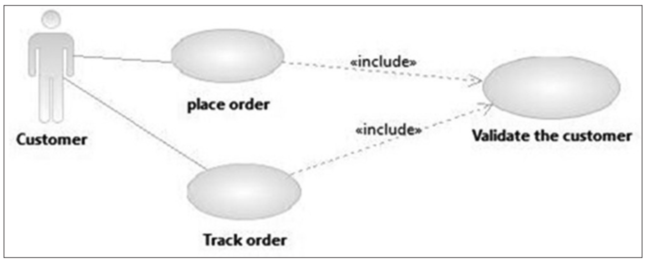

Fig. 3: <include> relationship in order processing UCD

credentials. Here, the customer credentials validation flow of events is similar for both use cases. The same flow of events are moved into a new use case called validate use case which avoids the duplication of same flow of events. Fig. 3 shows using the <include> relationship, the "place order" and "track order" use case can include the "validate customer" use case. Whenever the customer uses the "place order" and "track order" use cases, the "validate customer" use case is called compulsorily.

An <extend> relationship is useful to denote optional and exceptional events. The exceptional or optional behavior flows of events are created as a separate use case called extending use case. Whenever the base use case reaches the exception flow or optional flow condition, then the extending use case will be invoked. Fig. 4 describes the flow of <extend> relationship.

Fig. 5 depicts the usage of <extend> relationship. In the order processing system, usually the customer orders are processed sequentially, but in some exceptional cases, the orders are processed immediately. Whenever the place order use case reaches the rush order exception condition, then it will invoke the place rush order use case, otherwise, it will complete its own flow of events.

The complete flow of use case can be depicted using the <extend> and <include> relationship in the use case diagram. Most of the practitioners failed to use these relationship properly because of the similar definition of <include> and <extend> (Martin, 2000). The direction indicated in the<extend> relationship also misleads a common practitioner.

The UML use case diagram depicts only the functional requirements, and it does not hint on any. Few QARs are embedded with the functional description specified in use case specification. The next section describes the usage of use case specification.

Use case specification

A use case diagram depicts a high-level functional requirement. Each use case specified in the use case diagram is expressed in detail using a predefined structure. The use case specification template contains precondition, post condition, basic, alternative, and sub flow sections.

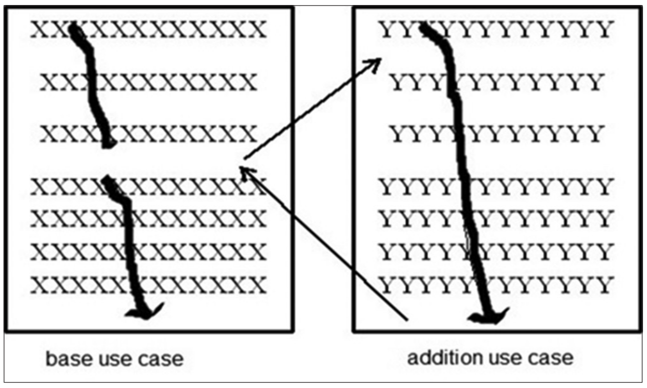

Fig. 4: Control flow of <extend> relationship

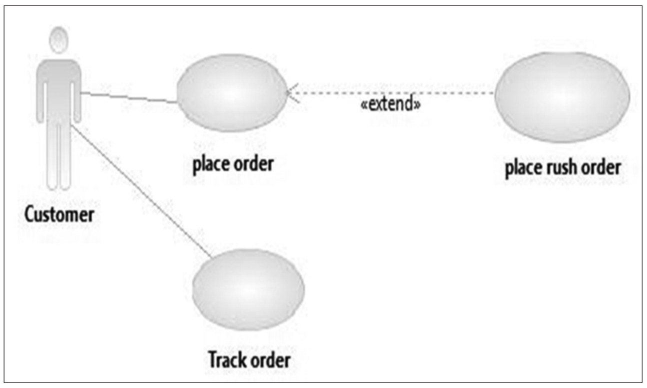

Fig. 5: Usage of <extend> relationship in order processing UCD

The basic flow section portrays the regular order of steps needed to complete the purpose of the use case. The alternative flow section brings out the occurrence of other than the regular sequence of events. Optional flows indicate the optional sequence of steps. An alternative set of steps and exceptional set of steps of the use case are described in this section. The essential conditions to be satisfied before initiating the use case are outlined in the pre-condition section. The post condition outlines the conditions to be satisfied at the time of completion of the use case, irrespective of the type of termination. The complex flows of events are divided into small flow of events called sub-flows. The sub-flow can be used to specify the repeated set of events, and this is similar to a subroutine in a programming language.

The textual form of place order use is shown in Table 3. In an online order processing system, create, delete, modify, and confirm orders are sub-flows of "place order use case." When customer selects the product which does not have a delivery to the customer location, it is an alternative flow. The chosen product cost less than the cash on delivery option; it is an alternative flow. In both alternative flows, the customer has to go back to create order option again to complete the place orders. The usage of natural language in use case specification leads to incomplete and ambiguous requirement. Capturing and tracking the requirements in use case specification is difficult. QARs are often used while describing other requirements.

Supplementary documents

A supplementary document elucidates the QARs which are important ASR of the system. For example, the place order option should be completed within 45 seconds is a performance requirement. It also specifies the technical and business constraints which are also ASR.

There is no prescribed template for supplementary documents. Hence, some practitioner describes in detail and others describe it shortly. Most of the ASRs are documented vaguely. For example, high performance

Table 3: Place order use case textual representations

Place order
Brief description
The use case explains how a customer places an order into the system
Flow of events
Basic flow
This use case begins when the customer desires to create, delete, modify and confirm the order
1. The system requests the customer to specify the function which he/she would like to perform
2. Once the customer selects the option, based on customer selection the respective sub flow is called
If the customer chooses "create order" option then the create order sub flow is called
If the customer chooses "delete order" option then the delete order sub flow is called
If the customer chooses "modify order" option then the modify order sub flow is called
If the customer chooses "confirm order" option then the confirm order sub flow is called
Create order
1. The system requests the customer to fill the product, quantity, payment mode and delivery address information
2. After entering the details, the system creates order id and these information are stored in the system
3. The system displays the stored order information with the corresponding order number
Delete order
1. The system displays the placed order list and requests the customer to choose the order number to be deleted
2. The customer chooses the order number to be deleted
3. The system displays a confirmation message to delete the order
4. If the customer confirms to delete the order then the system removes the order from the system
Modify order
1. The system displays the placed order list and requests to choose the order number to modify
2. The customer chooses the order number
3. The system displays the order information for modification
4. The customer modifies the order and select finish
5. The system displays a confirmation message to update the order
6. The customer confirms the modification
7. The system updates the order in the system
Confirm order
1. The system displays the ordered list and requests to choose the order number to confirm
2. The customer chooses the order number
3. The system displays the order information to confirm the order
4. The customer confirms the order
5. The system displays the payment options to choose
6. The customer chooses a payment option
7. The system displays the payment details
8. Once the customer fills the requested information, the system generates the bill and finally system confirms the order
9. The system generates the bill details to the customer
Alternative flows
Cash on delivery is not possible
If, in the confirm order sub flow, the customer chooses the cash on delivery option and if the order amount is <500 the system displays "no cash on delivery for <500" message and initiates the "pay now" option
No delivery location
If the customer selects a delivery location to deliver the product which does not have delivery option in create order sub flow, then the system displays that delivery is not possible
Pre-conditions
The login use case must be successfully completed
Post-conditions
If the use case is successful, the updated order information is stored. If not the same order information is maintained

and 100% uptime are vague ASRs. Highly available, fault tolerant are important QARs often used while describing other requirements. This type of embedding style will affect capturing ASR.

The term supplementary gives an impact to the practitioners that it is an additional and optional document [24]. Hence, many of them give less attention to read this document. Ignoring or missing the QAR leads to project failure. This inattention and ambiguous ASR specifications mislead the architect while designing the architecture.

EVALUATION OF ASR SPECIFICATION METHODS

The influence of the high-level functional requirements and QARs are identified in the requirement elicitation stage. Documenting these requirements properly enable the architect to take a quick decision on the architectural design. Different requirement elicitation methods specify that the ASR's are represented in different ways. In this thesis, we evaluate four approaches for the expression of ASR. They are ASR specification using natural language, use case analysis, quality attribute scenarios, and factor table method. To evaluate and compare the various architecturally significant specification methods, different criteria are used. The criteria are, effectively capturing the high level functional and QARs, better negotiability and to support quick architect design decision by the architect.

Table 4 describes architecturally significant specification methods and evaluation criteria. Natural language specification is a freeform method and it presents the functional requirements in detail. Extracting high-level functional requirement is a hard task for the architect. Some of the quality requirements such as performance,

Table 4: Architecturally significant requirement specification methods and evaluation criteria

Criteria	Natural language	Factor table	Quality attribute scenario	Use case model
Easy to capture high level functional requirements	Difficult to capture from the detailed system requirement document	Functional requirements not specified	Difficult to capture functional requirements other than quality attributes involved functions	Use case diagram depicts functional requirement visually and completely
Easy to capture quality attribute requirements	Performance, security and safety are identified explicitly others are difficult to identify from the system requirements	Capture generic quality attribute requirements and not specific to related functional requirements	Capture all quality attribute requirements	Difficult to capture from various documents
Negotiability	Difficult	Difficult	Easy for quality attribute requirements	Easy for functional requirements
Quick architect design decision	Difficult	Difficult	Difficult	Difficult

safety, and security are specified separately. Other QARs are not identified explicitly.

The quality attribute scenario [21] is constructed using a set of framework elements. Stimulus element is useful to describe the conditions affecting the architecture. Source element describes the sources of stimulus. Environmental conditions describe the operational context.

The architectural elements are directly or indirectly affected by the stimulus. System response element describes how the system responds to the stimuli. The measure of how the system responds is presented in the response measure element. Each quality attribute scenario focuses on any one of the quality attributes. Reading all the quality scenarios and creating a link between the quality attributes is a challenging task for the software architect.

Hofmeister introduced the factor table to analyze factors that affect the architecture design, and these are categorized as organizational factor, technological, and product factor. A factor table is constructed using 3 columns. First column describes the influencing factors of the architecture. Some of the factors are flexible and changes can be made by the architect after the negotiation with the customer. However, few factors are not possible to change by the architect. These flexibility and changeability issues are elaborated in the second column. Third column indicates the impact of the factor or impact of the changed factor by architect. The factor table is useful to specify all influence factors. As the factor table is more generic, it does not describe which functional requirements are affected by which factors. Hence, it is difficult task for the architect to take quick design decisions.

UML use case model is a composition of use case diagram, use case specification and supplementary documents [24]. Use case is a collection of related scenarios to achieve a particular goal. The actor may be a machine or human who invokes the system. Use cases can be represented either as a use case diagram or in a textual form. The textual form of the use case is termed as use case specification. Use case diagram shows a static view of the system functionalities. It also depicts the use case and actor relationships. Each use case is described in detail in the use case specification using predefined structure. Application development standards, quality attributes of the system, legal and regulatory requirements and other requirements that do not fit naturally into the use cases that are specified in the supplementary document [24]. The UML use case diagram depicts only functional requirements, and it does not hint any QARs. Few QARs are embedded with functional description described in use case specification. A supplementary document describes QARs of the system. It also specifies the technical and business constraints which are also a part of ASR. There is no prescribed template for supplementary documents, hence some practitioner describes in detail and others describe it shortly.

To calculate the software effort estimation using use case point method, the complexity of the use case is calculated based on use case transactions. The transactions are identified using use case specification, which is documented in natural language. The usage of the natural language results in the unstructured use case specifications which affect the use case transaction count.

CONCLUSION

Most of the requirement elicitation and analysis methods use natural language as a communication medium except use case diagram in use case model. The use case diagram is used during customer interaction to capture and negotiate the functional requirements. Visual representation of high-level functional requirements using use case diagram is considered as a best method. This use case diagram is constructed using the raw user document. This diagram is useful for discovering, analyzing, and documenting functional requirements. Using this diagram, an architect can understand the scope of the project and will be able to perform the initial decomposition of the system. It is a communicational vehicle for user and developer team. A requirement engineer can utilize this diagram to negotiate the functional requirements with the customer.

The architecture design decision depends on QARs with respect to operational context. QARs are scattered in natural language specification method and use case model. Factor table only focuses in specifying the generic quality attribute influence factors, and it does not hint any functional requirements. The quality attribute scenario is a best method for documenting quality attributes with respect to operational context. The drawback of the quality attribute scenario is that it is represented using natural language. Hence chances, for representing ambiguous requirement are high.

During the requirement elicitation stage, the customer is interested in specifying the functional requirements. Very less priority is given for conveying the QARs.

All the architectural requirement specification methods have failed to support the architect in quickly capturing the architectural design decision inputs which are used to construct the software architecture. The reason is that the functional and QAR are presented in different documents.

In this context, introducing a systematic graphical representation method for presenting functional and QAR in one diagram is a useful research. Using the proposed graphical representation method:

- Architects can capture the ASR
- System Analyst can motivate and capture more requirements from customer
- Project Manager can calculate the effort estimation easily and correctly.

To evaluate and compare the various architecturally significant specification methods, different criteria are used. The criteria are, effectively capturing the high-level functional requirements and QARs, better negotiability and to support quick architect design decision by the architect.

A visual representation of the high-level functional requirements using use case diagram highly satisfies the criteria of capturing the high-level functional requirements in a trouble free manner. The quality attribute scenario frame can easily capture QARs. A use case diagram satisfies only the functional requirement negotiability criteria. A quality attribute scenario satisfies only QAR negotiability criteria. All the architectural requirement specification methods fail to support an architect to effortlessly capture the architectural design decision and to construct software architecture. The reason is that the functional and QARs are scattered across many documents.

The use case diagram has already proven that it is an ideal communication mechanism, to describe the high-level functional requirements. Quality attribute scenario is considered as the best method for describing quality attributes with respect to the operational context. Such visual representation method improves the understanding ability and enhances the communication between the user and developers. Thus, combining use case diagram with the quality attribute scenario in one diagram aids in capturing more ASR requirements from the user and helps the software architect to effortlessly take quick design decisions.

REFERENCES

1. Davis AM. Software Requirements: Analysis and Specification. Upper Saddle River, NJ, USA: Prentice Hall Press; 1993.
2. Kulak D, Guiney E. Use Cases: Requirements in Context. Upper Saddle River, NJ: Addison-Wesley; 2012.
3. Davey B, Cope C. Requirements elicitation- What's missing? J Issues Inf Sci Inf Technol 2008;5:543-51.
4. Gottesdiener E. Good practices for developing user requirements. CrossTalk J Def Softw Eng 2008;21:13-6.
5. Huang JC, Settimi R, Zou S, Solc P. Automated classification of non-functional requirements. Requir Eng 2007;12:103-20.
6. Coughlam J, Macredie RD. Effective communication in requirements elicitation: A comparison of methodologies. Requir Eng 2002;7:47-60.
7. Herrmann A, Paech B. Practical challenges of requirements prioritization based on risk estimation. Empir Softw Eng 2009;14:644-84.
8. Taylor RN, Medvidovi N, Dashofy EM. Software Architecture Foundations, Theory, and Practice. Hoboken, NJ: John Wiley & Sons, Inc.; 2010.
9. Somerville I. Software Engineering. India: Pearson Education India; 2011.
10. Babers C. Architecture Development Made Simple. El Paso, TX: CJC Publishing; 2003.
11. de Boer RC, van Vliet H. On the similarity between requirements and architecture. J Syst Softw 2009;82:544-50.
12. Lattanze AJ. Architecting Software Intensive Systems. Boca Raton: CRC Press; 2009.
13. Base L, Clements P, Kazman, R. Software Architecture in Practice. Boston, MA: Addison-Wesley; 2003.
14. Karl W. Software Requirements: Practical Techniques for Gathering and Managing Requirements Throughout the Product Development Cycle. Redmond, WA: Microsoft Press; 2003.
15. Chen L, Babar MA, Nuseibeh B. Characterizing Architecturally Significant Requirements. IEEE Software; 2013.
16. Cockburn A. Writing Effective Use Cases. Boston: Addison-Wesley; 2001.
17. Lange CF, Chaudron MR, Muskens J. In Practice: UML Software Architecture and Design Description. IEEE Software; 2006. p. 40-6.
18. Zhu L, Gorton I. UML Profiles for Design Decisions and Non-Functional Requirements. Sharing and Reusing Architectural Knowledge - Architecture, Rationale, and Design Intent, SHARK/ADI '07, IEEE; 2007.
19. Sindre G, Opdahal AL. Eliciting Security requirements with misuse cases. Requir Eng 2005;10:34-44.
20. Booch G, Rumbaugh J, Jacobson I. The Unified Modeling Language User Guide. Amsterdam: Addition-Wesley; 2001.
21. Barbacci M, Ellison R, Lattanze A, Staffor J, Weinstock C, Wodod W. Quality Attribute Workshops (QAWs). 3rd ed. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellor University; 2003.
22. Sangwan R, Neil C, Bass M, Houda ZE. Integrating a software architecture-centric method into object-oriented analysis and design. J Syst Softw 2008;81:727-46.
23. Bass L, Bergey J, Clements P, Merson P, Ozkaya I, Sangwan R. A comparison of Requirements Specification Methods from a Software Architecture Perspective. Pittsburgh: Technical Report, Carnegie Mellon Software Engineering Institute; 2006. Available from: http:// www.sei.cmu.edu/reports/06tr013.pdf. [Last accessed on 2016 Dec ??]
24. Bittner K, Spence I. Use Case Modeling. Boston: Addison-Wesley; 2002.
25. Anda B, Sjeberg D, Jorgensen M. Quality and understandability of use case models. Proceedings of the 15th European Conference on Object-Oriented Programming; 2001. p. 402-428.
26. Cockburn A. Writing Effective Use Cases. Boston, MA: Addision- Wesley; 2000.
27. Armour F, Miller G. Advanced Use Case Modeling: Software Systems. New York: Addison-Wesley; 2000.