HQLS-PY: A New Framework to Achieve High Quality in Large Scale Software Product Development Using POKA-YOKE Principles

[1]K. K. Baseer, [2]A. Rama Mohan Reddy, [3]C. Shoba Bindu
[1]Research Scholar, JNIAS-JNTUA, Anantapuramu. kkbasheer.ap@gmail.com
[2]Professor, Dept. of CSE, SVU College of Engineering, Tirupati. ramamohansvu@gmail.com
3Associate Professor, Dept. of CSE, JNTUCEA, Anantapuramu. shobabindhu@gmail.com

Abstract—We propose a new model for large scale Software development for Products and Services with high quality expectations. It would be based on investing upfront in the Software Architecture of the system, designing with the software product monitoring and alerting logic in place, end-to-end user experience, experimentation and quality of service based on Poka-Yoke principles. The basic idea behind developing this new model is to have high quality software products and services that can be developed faster, cheaper and in better way, it can scale with demand in various scenarios, can deliver outstanding user experience and be failing safe for SDLC bottlenecks which arise in both conventional and Agile Software Development. The proposed model has the following areas:

- **Get the right Software Architecture in place**
- **Ensure high quality software is developed**
- **It is based on POKA-YOKE principles**
- **Focus is on user experience**
- **Ensuring need of the software is identified**
- **Architecture design follows 12 factor principles**

IndexTerms—Usability, Poka-Yoke, Product, Framework, Software Architecture, Quality, Product Monitor, Services

I. INTRODUCTION

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [1]. Non-Functional Requirement (NFR) approach, Quality Attribute Model approach and Intuitive Design approach. For each approach having its own bottlenecks such as decisions are not precisely determined, less modularization, no predictive model, no amenable architecture, no scale up architecture and no organizing requirements etc., shown in figure 1 [16,17]. Software must possess the qualities like Safety, Reliability, Availability, Cost, Maintainability, Performance or Response, Time, Energy consumption [5]. Usability is important not only to increase the speed and accuracy of the range of tasks carried out by a range of users of a system, but also to ensure the safety of the user. Productivity is also imperative where the software is used to control dangerous processes. Computer magazine software reviews now include usability as a ratings category [6]. To achieve non-functional requirements for any modeling software architecture still remain a difficult task as many stakeholders involved in the selection process as shown in figure 2.There are some recent attempts to establish software science as a foundation of software engineering. This may promote more analytical reasoning about software architecture, if it becomes popular. Software architectural design would benefit from analytical reasoning with scientific foundations. Importance of software architecture in the software design process is generally accepted among practitioners [7]. Below are some ideas and SE paradigms identified for better improving the Software Development Life cycle and identify the optimize some areas in SDLC so that Time To Market can be made faster and efficient, without compromising the code quality and functionality.

A. Frameworks

The key concept in using frameworks is design reuse. In contrast to past approaches that applied the term reuse to individual software functions (such as sine), the objective of frameworks is to reuse complete domain-specific units - for instance, customer records, accounts, or security accounts. In other words, we try to preserve our existing development work, such as task analysis and domain class design, by creating a skeleton frame representing, for example, the implementation of an account and its interface components.

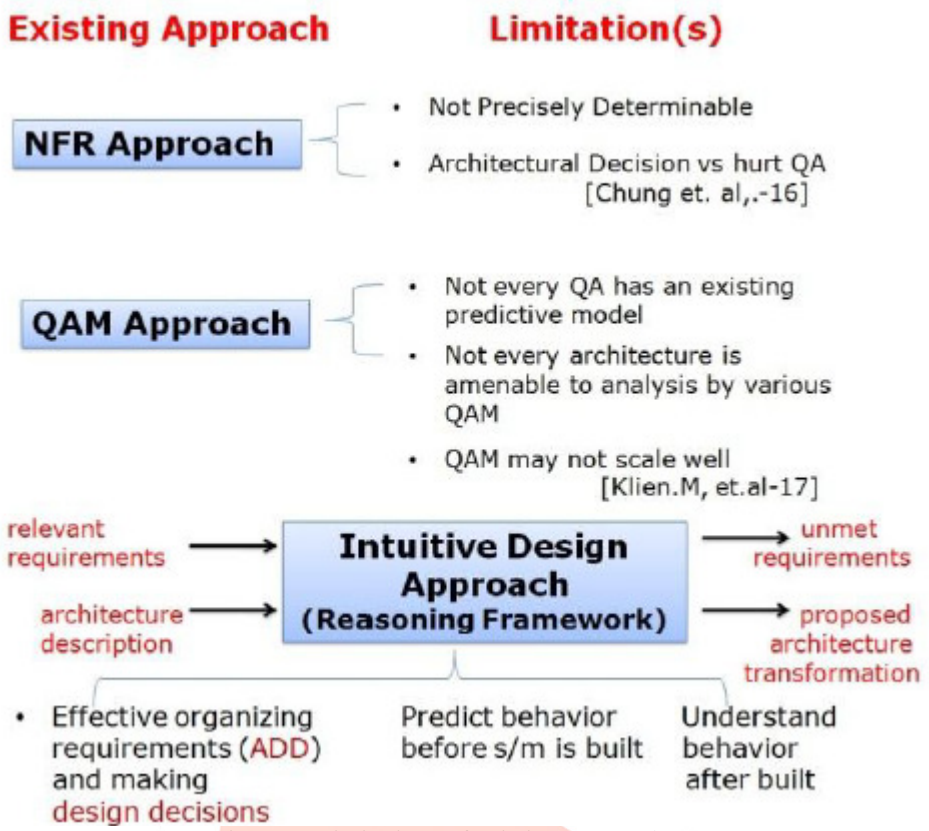


Figure 1: Limitations of existing approach

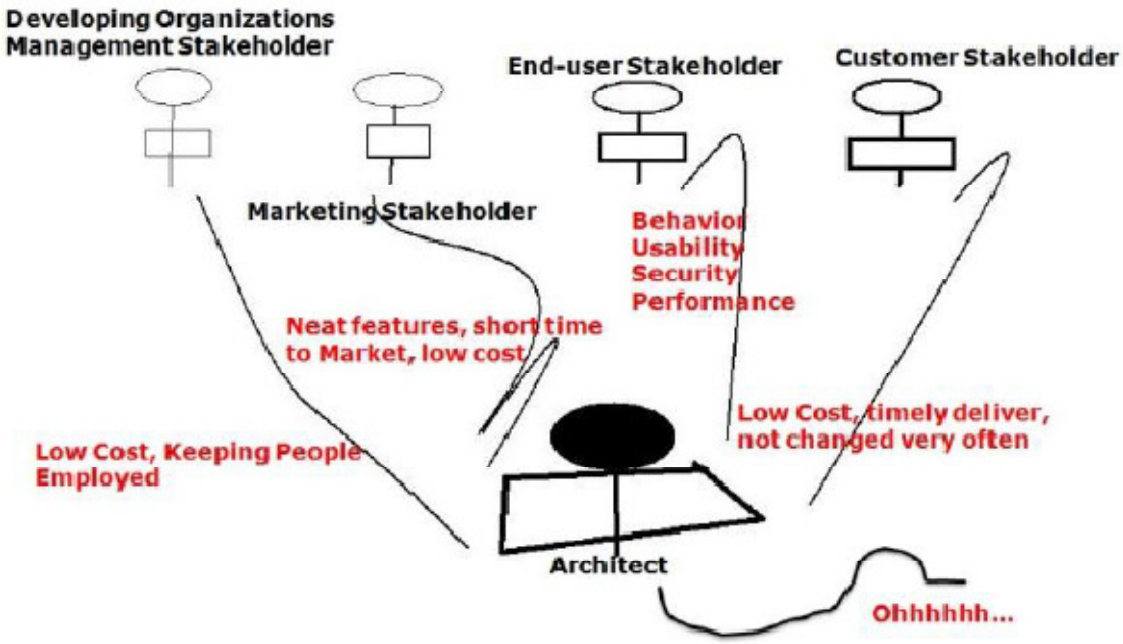


Figure 2: Influence of stakeholders on the architect [1]

Then, application programmers need only tailor the frame to the specifics of a particular application domain. To make the difference clear: In traditional approaches to software reuse, we create an application program using existing class libraries for database access, mathematical functions, and the user interface. In contrast, design reuse means that we adapt a prefabricated, fully developed hull—by sub classing, for instance. Thus, in frameworks, the program logic is already present.

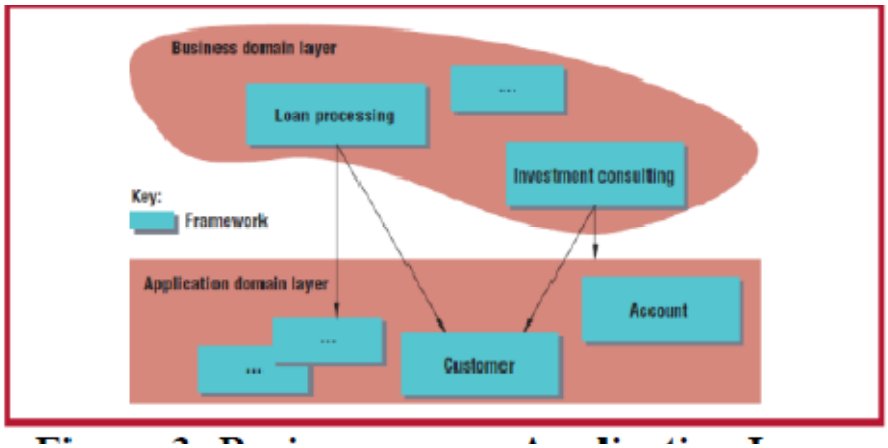


Figure 3: Business versus Application Layer

B. Sedimentation Pattern

After we ship the first small business domain and the users approve it, we gradually add software solutions for more elements of the application domain following the same method. These frameworks will overlap in part with existing ones but will also add new aspects. In simple terms, the ―art‖ of creating frameworks consists of finding exactly those implementation chunks (in object-oriented jargon, operations and attributes) that you would otherwise have to design over and over again—for example, within each implementation of a new type of account. We then extract these chunks from each implementation and represent them as frameworks in the deeper of two layers, the application domain layer (see Figure 3). This process is called sedimentation. Taking up the example of an account again, the outcome of this procedure is that all classes in which the deposit and withdrawal of money are implemented settle into the ADL. Each specific type of account (such as checking or savings) is always connected to a specific type of business activity (such as loan processing or investment consulting) and has specific features particular to it—for example, checking credit limits is a specific feature of current accounts, and progressive interest rates are a specific feature of savings accounts. All the implementation chunks that deal with such tasks are subsumed under frameworks deposited in the business domain layer (see Figure 3). All frameworks in this layer must use the basic frameworks in the ADL, thus ensuring a high degree of reuse. As application development progresses, the sedimentation of implementation chunks decreases, until all the stable frameworks have sedimented in the ADL.

C. Tool-material Design

The Tool–Material design pattern, which describes the interplay between application domain objects (such as an account) and objects for interaction purposes (such as an editor), is derived from the tool–material metaphor.1,2 This metaphor—namely, that materials can be worked on with tools- defines the interdependencies between two (technical) objects: A tool is produced for working on specific materials—for example, an editor tool might enable users to work on an account (the material, for example, for making deposits and withdrawals). Thus, a material and the user interface are not directly connected—rather, the tools themselves handle all interactions. Implementing a tool requires knowledge about the nature of relevant materials; implementing a material remains independent of specific tools. This means that when an application system‘s software architecture is in accordance with the Tool–Material pattern, user interface changes will not affect the material frameworks contained in the business design layer or the ADL. Role Pattern: The idea behind the Role pattern is that the core implementation of a customer (used by all the customer roles) is a framework in the application domain. In contrast, role-specific customer implementations are frameworks in the business domain. If a role changes, it affects only that role‘s framework. The customer core framework in the application domain as well as the frameworks of other roles remain unaffected, and—even more importantly—sedimentation is not obstructed. Also, users and software engineers can use the same terminology and refer to the same entities, the roles. Just like the Tool–Material pattern, the Role pattern is completely integrated into the development life cycle and is another excellent example of how design patterns can facilitate user participation. Style guides for framework development must take into account these two issues:

- Criteria for reuse. It is important to precisely define the situations in which developers may reuse a particular tool. A catalog, for example, might list all the tools available for reuse together with the prerequisites for using them. Otherwise, there is the danger that tools might be used outside the intended context just because they have already been implemented and are ready for reuse. Experience has shown that problems arise in such cases, because adapting tools for mid- or long-term use cancels out the benefits of reuse and destabilizes the business or even application layers.
- Usage models. A usage model is the set of all metaphors (as, for instance, the tool–material and role metaphors) that guide how to use an application. The usage model must be standardized early in the development process; otherwise, developers might unknowingly use different models as they simultaneously build tools for several business domains. At worst, this could necessitate the modification of application domain frameworks, thus slowing down or even reversing the framework maturation (sedimentation) process.

D. Architecture Principles

For building a network service (e.g. a web application), you should design it as a Twelve-Factor Application [8]. The remainder of this paper is structured as follows: In Section II, we discusson Poka-yoke principles. In Section III, we present Related Work,

Challenge and Problem Statement. In Section IV, we present our Framework which discuss the methodology, outline. Finally, in Section V, we provide some conclusion.

II. POKA-YOKE

Poka-yoke (ポカヨケ) [Poka yoke] is a Japanese term that means "mistake-proofing"[10]. A Poka-yoke is any mechanism in a lean manufacturing process that helps an equipment operator avoid (yokeru) mistakes (Poka). Its purpose is to eliminate product defects by preventing, correcting, or drawing attention to human errors as they occur. The concept was formalized, and the term adopted, by Shigeo Shingo as part of the Toyota Production System. It was originally described as baka-yoke, but as this means "fool-proofing" (or "idiot-proofing") the name was changed to the milder poka-yoke. Poka Yoke is used to eliminate the possibility or opportunity for passing on errors or making mistakes in a process. Poka Yoke is used:

• In the development or improvement of any process.
• When you want to make wrong actions impossible or more difficult to do.
• When there is a need to make it possible to reverse actions – to ―undo‖ them – or make it harder to do what cannot be reversed
• When you need to make it easier to discover that errors occur.

A. How to use it

- Identify the errors/mistakes which could be passed on
- Develop potential solutions to prevent errors
- Develop potential solutions to detect errors
- Implement solutions

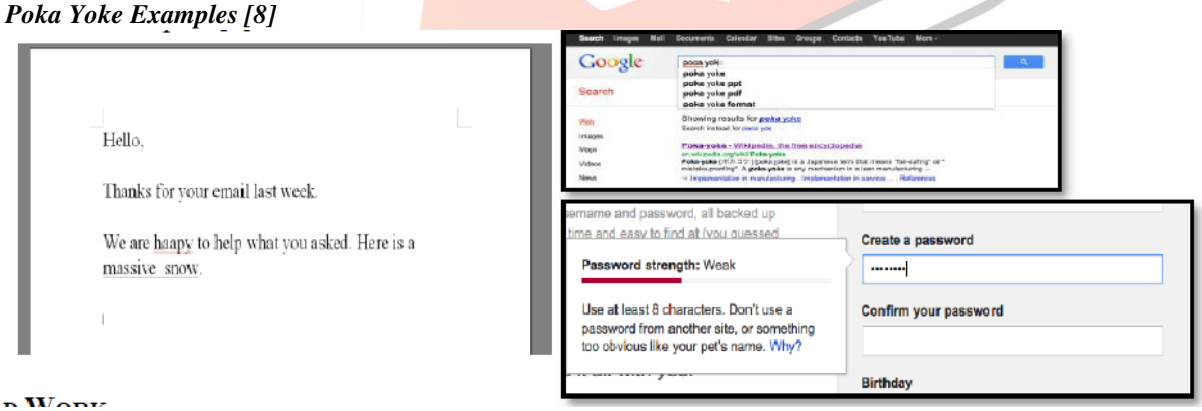
B. Benefits

- Used to develop solutions to prevent mistakes before they occur or to detect errors and make it impossible for them to be passed on to the next step of the process.
- Can be used in the development of a new process or in an existing process where rework to correct errors is hurting process effectiveness and efficiency.
- Poka Yoke ensures that mistakes are not transferred to the next step of the process.
- Poka Yoke solutions are a simple and low cost way to reduce rework

C. Background

- Mistakes are human errors that result from incorrect intentions or executing correct intentions that result in unintended outcomes.

D. Poka Yoke Examples [8]



III. RELATED WORK

In recent years, the research on applying Poka Yoke in software has received much attention [2] [9] [11] [12] [13]. Harry Robinson introduced poka-yoke (mistake-proofing) into Hewlett Packard‘s software process and he claims they have been able to prevent literally hundreds of software localization defects from reaching their customers [9]. As per Gojko Adzic, author of Impact Mapping ―software classes should not allow us to proceed and blow up when something goes wrong.Exceptions can be an effective way of giving more documentation, but the signal should be clear and unambiguous, in order not to mislead users or client-developers. Software must be designed to prevent a complete crash, even in the face of system failure. Auto-save features are a good example. It‘s not often that the power gets cut, but when it does, our users will surely appreciate that we saved most of their work‖ [14]. Much of the research focus is for ZOC, quality control, identifying defects. However, the limitation that the associated research brings is not applying Poka- Yoke in entirety. In usability a sub-characteristic is errors which is shown in figure 4.

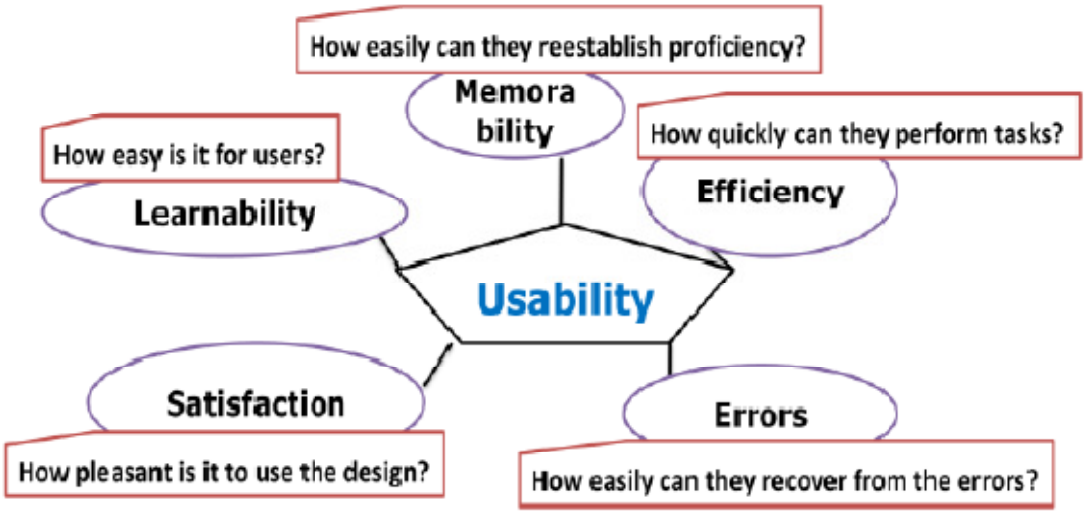


Figure 4: Usability Characteristics (Courtesy: Jakob Nielsen and Ben)

A. Challenge

The ever-increasing expansion of applications and users requirements make a steep rise in the scale and complexity of software, which results in the decrease in the software quality. So it is a great challenge in software engineering to understand, measure, manage, control, and even to low the software complexity [3]. Software Product Line engineering aims at improving productivity and decreasing realization times by gathering the analysis, design and implementation activities of a family of systems. Variabilities are characteristics that may vary from a product to another. The main challenge in the context of the Software Product Lines (PL) approach is to model and implement these Variabilities [4].

B. Problem Statements

The prior researches have not been able to uncover root cause of software quality issues. Existing methods focus on software testing and some extreme programming approaches to reduce defects. Proposed method is to focus on software architecture and building monitoring within the software product and services.

IV. FRAMEWORK

A. Methodology

We are going to examine each of the phases in software development life cycle and find out opportunities of improvements. We will explore how things can be planned and designed upfront to avoid discovering issues late in the cycle. One of the major challenges in software is quality, to understand this better we will find opportunities to inject product monitoring at the right place to capture the user experience, product quality and help us in alerting at the right time. We are going to leverage principles of Poka Yoke (a Japanese methodology primarily used in production to make the process mistake-proof, this prevent people from making mistakes and if mistakes are made, they are caught early in the cycle). We have observed that very few people have talked about this in the context of software product development (one of the references [3] have implemented this in Microsoft and have had good success with this).

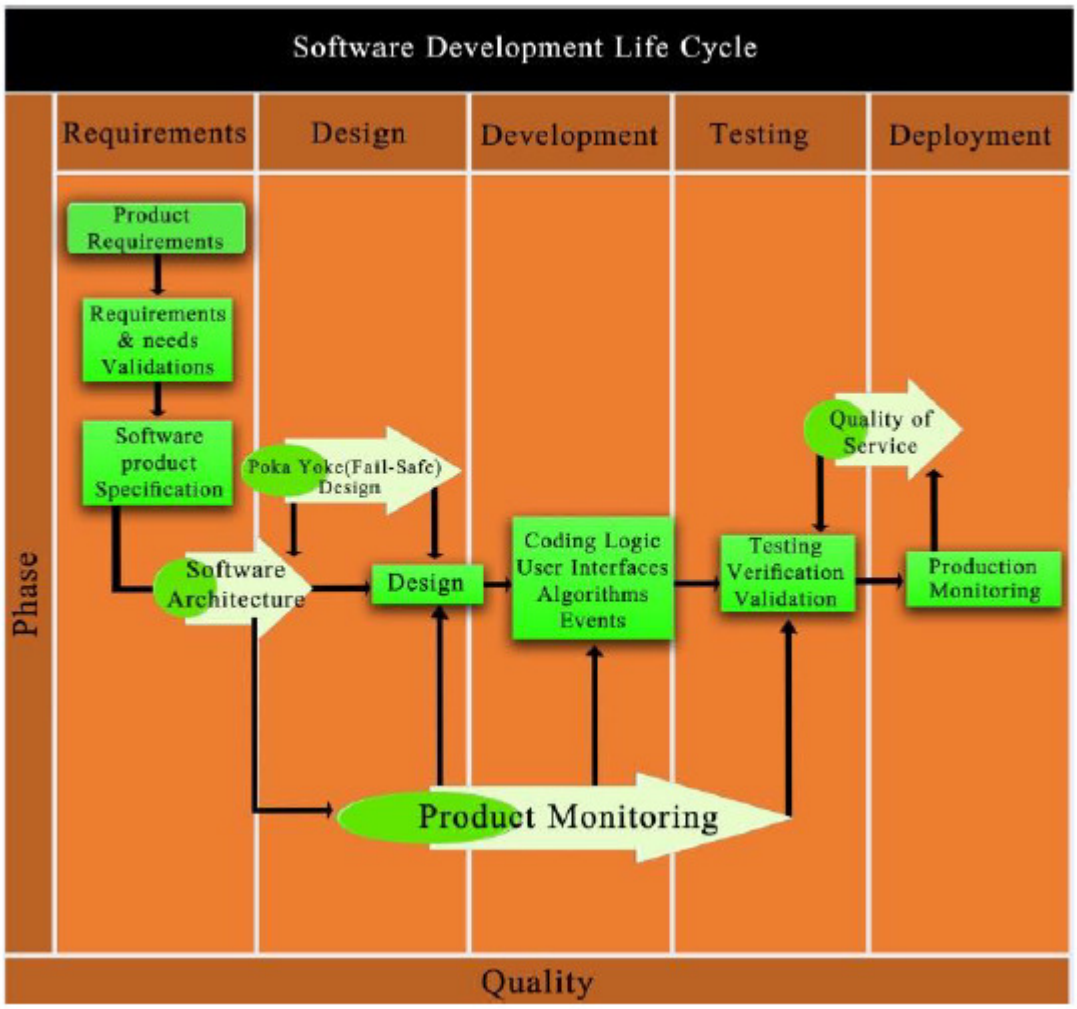


Figure 5: Framework for updated Software Development Life Cycle

B. Outline

Figure 5 shows how to provide emphasis on investing in the Software Architecture before the design is started. While developing Software Architecture, adequate attention is to be provided to the Software Reliability, Scale, User experience and making it fail safe based on Poka Yoke Principles. One of the additional outputs the software Architecture will be the details for software product monitoring. Product monitoring is something that should be thought through from day one and should not be an afterthought in the final phase of software development or deployment. Right set of monitoring helps us know how the software system is performing in various condition and can help in alerting as needed. The details and expectations of product monitoring are fed in the design and the final design needs to come out with these details. The same are ensured in all the user experience, algorithm and logic coding. In the verification and validation phase, all these monitoring needs to be validated to ensure right data are captured for the right set of events.

V. CONCLUSION

The basic idea behind developing this new model is to have high quality software products and services that can be developed faster, cheaper and in better way, it can scale with demand in various scenarios, can deliver outstanding user experience and be failing safe for SDLC bottlenecks which arise in both conventional and Agile Software Development. Proposed method is to focus on software architecture and building monitoring within the software product and services.

REFERENCES

[1] Bass, Clements, Kazman, Software Architecture in Practice, 2nd ed. Addison-Wesley, 2003
[2] Weifeng Pan,"Applying Complex Network Theory to Software Structure Analysis‖, World Academy of Science, Engineering and Technology, Vol.60, pp.1636-1642, 2011
[3] Mukesh Jain, Delivering Successful Projects with TSP(SM) and Six Sigma: A Practical Guide to Implementing Team Software Process(SM)
[4] W.M.Abdelmoez, A.H.Jalali, K.Shaik, T. Menzies and H.H. Ammar, ―Using Software Architecture Risk Assessment for Product Line Architectures", In.Proc.of.International Conference on Communication, Computer and Power (Icccp'09) Muscat, February 15-18, 2009
[5] Indika Meedeniya, ―Robust Optimization of Automotive Software Architecture", In.proc.of AutoCRC Technical Conference, 2011

[6] Ahmed Seffah, Mohammad Donyaee, Rex B. Kline and Harkirat K. Padda ,"Usability measurement and metrics: A consolidated model", Software Quality Journal,Vol.14, pp.159–178,2006
[7] Pradip Peter Dey, ―Strongly Adequate Software Architecture", World Academy of Science, Engineering and Technology, Vol.60, pp.366-369, 2011
[8] The Pokayoke Guide to Developing Software‖, http://pokayokeguide.com/
[9] Harry Robinson, ―Using Poka-Yoke Techniques for Early Defect Detection‖, Sixth International Conference on Software Testing Analysis and Review (STAR'97)
[10] Shigeo Shingo, Zero Quality Control: Source Inspection and the Poka-yoke System, Productivity Press, pp.45
[11] G. Gordon Schulmeyer, Zero Defect Software. McGraw-Hill, Inc.
[12] James Tierney, Eradicating mistakes in your software through poka yoke. MBC Video
[13] Boris Beizer, Software Testing Techniques, 2nd ed. Van Nostrand Reinhold, pp. 3
[14] Gojko Adzic, Impact Mapping: Making a big impact with software products and projects, Published by Provoking Thoughts,1 October 2012, ISBN: 978-0-9556836-4-0
[15] Revision control system‖, http://git-scm.com/
[16] Chung, L., Nixon, B., Yu, E., and Mylopoulos, J.: _Non-functional requirements in software engineering‘ (Kluwer Academic Publishers, 2000).
[17] Klein, M., Ralya, T., Pollak, B., Obenza, R., and Gonzlez Harbour, M.: _A practitioner‘s handbook for real-time systems analysis‘ (Kluwer Academic Pub, 1993)