Quality Attributes and Software Architectures Emerging Through Agile Development: Pursuit or Overlooking?

G. H. El-Khawaga *Ghada.elkhawaga@ieee.org*
Teaching Assistant, Department of information systems,
Faculty of computers and information,
Mansoura University,
Mansoura, Egypt

Prof. Dr. Galal Hassan Galal-Edeen *Galal@acm.org*
Computer Science Department,
School of Sciences & Engineering,
American University in Cairo,
Cairo, Egypt

Prof. Dr. A.M. Riad *amriad2000@mans.edu.eg*
Dean of the faculty of computers & Information,
Mansoura University,
Mansoura, Egypt

Abstract

Software architectures play an important role as an intermediate stage through which system requirements are translated into full scale working system. The idea of what a system does, what it does not, and different concerns and requirements can be negotiated and expressed clearly through the software architecture. Software architectures exist to enhance and provide quality attributes, while they are quality attributes and their required level of achievement which can offer numerous number of software architectures for a single software system.

We believe that the agile approach to architecting is problematic because of agilists' beliefs about how to architect a software system, and how critical quality attributes are to achieve a stable yet flexible architecture. Through this research we clarify these issues, and discuss consequences of agile architecting on achieved level of quality attributes. We are going to pursue the answer to how to architect to achieve required level of quality attributes, while adopting an agile process.

Keywords Quality Attributes, Software Architecting, Agile Software Development, Refactoring, Clean Architecting, Light Architecting.

1. QUALITY ATTRIBUTES AS BEING ENABLER OF SOFTWARE ARCHITECTURES VARIANCES

Are software architectures there to answer certain quality attributes-related questions? Have we got to care about arrangements and relationships between software components in response to quality attributes-related needs? Have the concept of software architectures emerged after being involved into long era of deficient software resulting from unstructured development? Do software architectures exist to enhance quality attributes of software systems, or they are quality attributes which distinguish software architectures? If the answers to these questions are all "yes", then there are more questions to ask. Do software architectures emerging through paradigms like agile software development achieve their purpose of reaching a certain level of quality attributes defined through a product's context and concerns' analysis? Can we truly offer longevity of a

software product and its ability to absorb frequent changes all over its production time without paying attention to how its architecture is formed to offer quality attributes? To find an answer, we need to begin tackling and defining the relation between a software architecture and quality attributes.

1.1 Criticality of Quality Attributes

The intent of designing the architecture for a system is to transfer system required functionality, quality attributes, business goals, and system context into an intermediate state before being transformed to full-scale developed system. Software architecture is an arrangement of software building blocks into differentiated types, or categories that are grounded in or derived from the problem domain, and the way the software might be used and later adapted as an artefact [1]. This definition mainly referred to system requirements as a main driver of an architecture. Therefore; through architecture creation, architects are supposed to elicit and understand the received requirements so as to reach clear view of what the system should do, and to begin on making decisions that shape how the system will work to achieve desired goals. However, it is emphasized that a software architecture differs from building architecture in that it can't be limited to one structure [2]. In civil engineering, structure is one category of the architecture; while in software engineering, a system can have thousands of forms, which differ in quality attributes satisfaction levels, not in the functionality associated and achieved through these forms. If it were only about functionality, a software system would have been composed of a single module with no internal structure [2]. Functionality drives the initial decomposition of a system architecture into a set of components that together perform the functions of the system [3], but it is the mapping of a system's functionality into software structures that determines the architecture's support for quality [2]. A quality attribute is a constraint on the manner in which the system implements and delivers its functionality [4]. Systems are redesigned not only due to functionality dissatisfaction, but also due to lack of consideration of quality attributes like security, performance, maintainability, and reliability [2]. Quality attributes are advanced to functionality considerations, and this can be argued for by the idea [3] that one of the motivations for creating an architectural design (addressing quality attributes) before detailed design (addressing functionality) and coding is to enable improving, measuring, observing the quality of the system, and predicting whether the system to be built will exhibit certain quality attributes while addressing risks and potential defects earlier where they are cheaper, easier, and faster to fix. At the same time, software architecting is a major strategy for enhancing quality attributes of software systems [1]. Architecture plays a central role in realizing many qualities in a system. While we believe that an architecture embodies decisions about quality priorities and tradeoffs, and represents an early opportunity to evaluate these decisions, it is argued that an architecture provides only the foundation for achieving quality; but without paying attention to the details, this foundation will be in vain [2].

1.2 Challenges Associated With Quality Attributes' Specification

Considering, expressing, and evaluating quality attributes is not an easy mission. Challenges of adopting quality attributes can be categorized into two paths, so as to enable recognizing how to consider and deal with a system's desired quality attributes. A path or a category is about *what* these quality attributes are, and the other is about *how* they are considered into a system. The first category is related to the natural characteristics of quality attributes themselves. Many quality attributes naturally have architectural and non-architectural aspects. Performance, for example, has architectural aspects like functionality allocation to components, and communication between components; while it has non-architectural aspects like the choice of algorithms to achieve functionality, and how these algorithms are coded [2]. Ignoring this confusing nature of quality attributes raises many pressures and challenges, like the difficulty of ensuring that a specific quality attribute has stemmed of nontechnical issues [4]. Much attention should be paid to architectural and non-architectural aspects of a quality attribute so as to decide how to handle it while it is affecting other attributes.

Whether positively or negatively, quality attributes affect each other. So they cannot be handled in isolation. While making an architectural design decision, interactions between quality attributes

should be put into consideration, and a decision is to be made based on affected and interacting quality attributes relative priorities. Conflicts between quality attributes should be discovered as early as possible, and desired quality attributes achievement levels should be available early so as to help make a decision about a certain quality attribute preference whenever a conflict exists. However, this depends on how a development team handles quality attributes; and this is shown through the second category of challenges in dealing with quality attributes.

Another challenge that stems of natural characteristics of quality attributes is how to measure and evaluate an architecture's achievement of certain quality attributes. This challenge is due to that many quality attributes are qualitative in nature, rather than being quantitative [3]. For example, a software system into operation can be tested for its performance by quantitative measures, while maintainability of a system should be observed and reasoned about through qualitative measures like questionnaires. Considering a qualitative or a quantitative quality attribute for assessment is critical to deciding when to carry out an evaluation phase.

The second category of challenges is related to how quality attributes are handled through the development process, and where they are located into development participants' consideration. There is a wide agreement that modelling methods are weak in representing quality attributes [3], and that architectural analysis techniques focusing on quality attributes are rare [4]. This drives software architects to deal with quality attributes with an informal process [2]. However, informal and incomplete specifications of quality attributes increase dependability on the architect to fill in blanks and mediate the conflicts, and increases possibilities of redesigning the system to meet missed quality attributes. It is confirmed that quality, cost, and schedule are not independent as poor quality affect cost, and schedule [5].

Another challenge stems from that architects and developers –especially agilists- tend to deal with quality attributes as an afterthought [4]. This was attributed to the development team's attention to business stakeholders rather than technical ones, and to the team's belief that some quality attributes don't have direct impact on the cost-benefit for a system [6]. Business stakeholders won't be able to ask questions other than those about functionality, and they won't be aware of these questions that can help in analyzing and assessing the desired system's architecture [3]. The way of handling quality attributes raises technical future risks which if not handled early, they can break the system, and consequently will impact the cost-benefit of obtaining and operating the system threatened. It is argued that the costs for maintaining and extending an application will account for most of the cost of the application over its lifetime [3].

Agilists architect software in a way that exposures resulting architectures to risks associated with the challenges defined through this section. We are going to explain this more through the coming sections.

2. THE AGILE WAY TO TACKLE QUALITY ATTRIBUTES

Agilists regard architecting in light of traditional development as being associated with heavy-weighted practices which don't yield value on the short term. Of course we are totally against these beliefs, but it is out of scope to discuss and argue about how far these claims from reality. What we are concerned about here is to discuss architecting practices that agilists use and have influence on quality attributes. The main agile techniques to tackle quality attributes are architectural spiking, and refactoring. *Architectural spiking* is about implementing a feature that the development team believe to be exposed to and affected by the highest number of architectural design decisions. We believe architectural spikes are not efficient at evaluating architecture design decisions, because those decisions were originally made to satisfy certain quality attribute concerns. Quality attributes cross-cut a software architecture, while quality attribute concerns differ across various parts of an architecture. To take a vertical slice of an architecture as a means to judge the level of achievement of a quality attribute, while knowing that this quality attribute would be heterogeneous across the whole architecture; this doesn't seem to be a viable way to evaluate an architecture's conformance to its basic role. Agilists claim

they do only practices that add value, and we strongly believe conducting architectural spikes is a practice that missed its basic value. Agilists use architectural refactoring to make high-level changes to achieve quality attributes. However, not all quality attributes such as security can be accommodated later in implementation through refactoring [9]. Some quality attributes' components and mechanisms must be designed early in the life cycle. Issues associated with the way agilists handle changes through architectural refactorings and these issues' implications are explored through the coming section.

Agilists believe in simplifying design to achieve a barely good enough design to begin with. The point here is that while software architecture is believed to be the magical work for achieving system qualities such as performance, security, and maintainability [7], agilists consider designing for system qualities to be heavy work about unforeseen changes, and this work should be eliminated if not avoided. While adopting this attitude; they ignore foreseen changes that would come up on the long term. As a consequence; agile methods are accused of ignoring quality attributes such as reliability, scalability and changeability [8]. As change is inevitable, mechanisms should be employed to enable software to smoothly be adapted to changing circumstances in the development game.

3. AGILE ARCHITECTURAL REFACTORINGS: INTENDED TO PROVIDE A CHANCE, AND RESULTING IN A THREAT

Adding quality attributes through a software system's life cycle introduces new requirements, thus it can be considered some sort of perfective changes because they introduce new requirements and they aim at non-functional optimizations [10]. Lientz et al. –as cited in [10]- reported that 60.3% of the maintenance effort was categorized as perfective. This percentage is close to the results reported by Mockus & Votta's study [11] conducted which concluded that perfective changes accounted for 45% of all the modification requests. The challenges accompanying quality attributes' accommodation -whether these challenges are in general or are attributed to the usage of agile methodologies in software development- have resulted in having perfective changes to be of the highest percentage of the total maintenance efforts. The study conducted in [11] revealed that perfective changes -as well as being the highest to add more lines of code- are more time consuming than adaptive and corrective changes.

To study a change's implications on cost and schedule; the proposed change shouldn't be attached only to the code level. Instead, and with the aid of a big picture of the system under consideration; a proposed change should be studied at a global level rather than being localized only at the code level. A proposed change to code shan't be left till it violates the principal architectural design decisions that govern the application. In the way of identifying how change can affect a system's architecture, practitioners [4] tried to borrow some architectural concepts from physical buildings' literature. They were inspired by Stewart Brand's Shearing Layers of change. Brand categorized elements that make up a building into six categories. Brand's layers of change [4]:

1.Site: the geographical setting, and legally defined lot.

2.Structure: the foundation and load-bearing elements which are expensive to change.

3.Skin: exterior surfaces; they change so frequently to keep up with technology or for repair.

4.Services: the working guts of a building like electrical wiring.

5.Space plan: the interior layout; like doors, and floors.

6.Stuff: all the things that can be changed on a daily to monthly basis.

This categorization is organized in a manner that reflects the velocity and the hardness of changing the elements classified, from the slower and harder to change to the faster and easier.

Practitioners tried to make benefit of this categorization in software [4]. According to [4], the site layer in software denotes the usage context which may be an organization; the structure layer denotes the software system architecture as it identifies a system's load-bearing elements; the skin layer denotes user interfaces; the stuff layer may denote user settings. We believe that grouping elements by their similar change rates can help separate concerns, localize changes, and hence increase a software systems' responsiveness to changes. Such categorization can aid in identifying the necessary techniques to apply a given change, as well as the time and cost to achieve it [4]. Adhering to these groupings, it can be concluded that changes to software system architecture are to be the most expensive, difficult, and complex to implement.

Besides the important role an architecture plays in preparing for how the system will change and in localizing the effects of change, the profound changes to a system's architecture are induced by quality attributes' accommodation [4]. As mentioned before, agilists use refactoring as a main technique for adding quality attributes late in development lifecycle. Refactoring to introduce or modify quality attributes can imply modifying a component's internal specification; for example, introducing new components to increase performance implies changes to connectors [4]. Therefore, the consequences of making changes that can affect architecture elements – especially those resulting from making changes to accommodate quality attributes- should be studied carefully. Quality attributes are prevailing and affecting huge portions of code and functionality, thus modifying quality attributes is believed to be costly [7]. Not accommodating these changes early in the development process is sufficient to tear down the myth of having better quality using agile methods.

Frequent non-systemic modifications to requirements can result in architectural degradation, which leads to a mismatch between the actual functions of the system and its original design [12], and subsequently upgrades and fixes become expensive to implement. This case is called architectural erosion [11]. Architectural erosion is defined as the regressive deviance of an application from its original intended architecture resulting from successive changes [4]. Architectural erosion leads to increasing resistance to change and subsequently high cost of maintenance [13]. Architectural degradation causes are mainly mapped to late-lifecycle changes which are considered to be the most crucial, risky, and expensive when they are changes to requirements [12]. Therefore, the earlier to make changes is the better, and the earlier to consider quality attributes is the best. The difficulty, the choice of suitable technique, and the cost of supporting a given change are all deeply influenced by the development level at which a change is implemented [4]. As a result, late-lifecycle refactorings affecting the architecture of a software system are considered to be the most risky and expensive changes.

Among the important triggers of architectural refactoring are architectural smells which are believed to be negatively impacting system quality [14]. Architecture violations are considered to be the main architectural smells' type for which architectural refactorings are carried out [15]. This way we can conclude that refactoring to overcome certain architecture violations is likely to produce other architecture violations, and even they can be of a greater number than the ones these refactorings were carried out to overcome. Therefore, refactoring to reduce or eliminate an architectural smell can be risky and complex [14]; as it requires decisions that seem to be local while they have broad effects and involve uncertain consequences. The problem is more complex and risky in case of the absence of a well-defined architecture, and this may be the case while adopting an agile method in software development.

Architectural refactoring effectiveness for achieving quality is another issue that rises here. Architectural refactoring is effective in increasing an application's maintainability and consequently reducing costs [15]. However, architectural refactoring's effect on other quality attributes like performance, and security should be considered as well. Also, mutual influences of quality attributes and sometimes conflicts are critical aspects to be considered. Not all quality

attributes can be achieved in the same time; their achievement is proportional and they can't be treated in isolation of each other. Thus for example, refactoring to increase performance can affect reliability negatively, and so on.

We believe that architectural refactoring can alter a product's perceived behavior whenever these refactorings are conducted to incorporate quality attributes. This claim sounds reasonable as long as the main aim of refactoring is to alter internal structure without changing external behaviour, and it also raises critical questions about the viability of refactoring –in the context of agile development- to leverage a system's architecture and alter it later to insert missed quality attributes. Refactoring to fix architectural problems was firmly emphasized to be inefficient [16]. Refactoring, as considered to be a small activity with limited effect, is almost a local activity, whereas architecture is a global concern.

The discussion above highlights two issues; the *first* is that the need for spending some time planning architecture upfront is not something to be ignored. The second issue is that depending on refactoring to bring good code structure and hoping that code units together will form a good architecture that will stand and accommodate all upcoming changes won't be a viable development strategy in most cases.

4. SALVATION THROUGH CLEAN LIGHT ARCHITECTING

It is now clear that the way software architectures developed in the context of agile development is deficient regarding how quality attributes are accommodated. Agile architecting begins with overlooking quality attributes' accommodation and ends with risky and expensive pursuit. Problems discussed through this research are the main inspiration for our suggested recipe here to achieve a framework to architect in the context of agile software development. The ingredients of the proposed recipe are clean architecting; light architecting.

- *Clean Architecting*: actually the morals of this trend are similar to those which triggered clean coding. Clean coding aims at enabling readability of code and hence backward tracing of a solution. This is exactly the same aim of clean architecting. A clear rationale of architectural decisions whenever being traceable through an architecture would guide through highlighting architecturally significant requirements (ASR)s. These ASRs include functional requirements, quality attribute requirements, design constraints, and any requirement that can influence architectural design decisions made to form an architecture. Clean coding aims at facilitating testing and discovering refactoring positions. We argue that clean architecting is about providing forward traceability of potential changes to be conducted. As changes are irresistible for an agile software system, and -as explained- changes have critical effects on architecture; there is a need to conduct change impact analysis. Change impact analysis is about analyzing potential consequences of changing a factor, component, connector, configuration upon other components, connectors, configurations, or upon the quality attribute achieved through the previous state before change. Change impact analysis also enables defining potential conflicts between various quality attributes. This way, clean architecting should also enable early evaluation of architectural design decisions; and this is aligned with agile software development mindset which encourages short feedback cycles and early changes' discovery.

- *Light Architecting*: it complements and enables clean architecting. To enable architecting while saving agile values, a light architecting process should be revolving around creating an initial minimal architecture at the preproduction or chartering level of a product development process, and leaving non-critical architectural decisions -that are more potential to changes and aren't about cross-cutting decisions- to be made incrementally and iteratively at the release and iteration levels. This highlights again the need for impact analysis to decide which decisions can affect a broader portion of software features. To eliminate the gap between customer requirements captured informally and architectures which are believed to be captured explicitly; software architects should be involved through the development life process. This way we can consider software architecting as a continuous process which is about role collaboration, and

which enables collaboration and communication among team members. Communicating "what a software product is" is a basic moral of architecting. Therefore; choosing the suitable way to share information among team members and to keep it for further usage, is purely a team free choice. This way we can consider informal diagrams on a whiteboard to be a viable document. Light architecting facilitates developing clean architectures thanks to two reasons. First, time constraints which result in dirty architecting are halted through incremental and iterative architecting. Second, when architecting becomes a shared responsibility among team members, it is easier to increase learning curve and enable making benefit of all team members' skills; therefore, there is more possibility to come up with a clean architecture.

A few approaches were suggested to overcome the absence of a mechanism to create flexible yet static architectures in the context of agile development. Most of suggested approaches revolve around systemizing and providing a context for conducting architectural refactorings. Among these approaches are developer stories writing [17], and Continuous Architectural Refactoring (CAR) & Real Architecture Qualification (RAQ) [18]. These approaches are criticized for accrediting refactoring as the only way to introduce quality attributes in resulting architectures, while ignoring the need for designing initial architectures upfront depending on careful analysis of concerns about quality attributes.

To achieve clean light architecting while planning for quality attributes in the context of agile software development, we suggest an architecting process which is comprised of a hybrid of three complementing methods. The first is *Quality Attribute Workshop (QAW),* because it facilitates capturing quality attribute requirements through collaborative brainstorming sessions, in the form of scenarios which is light enough to be placed into the product backlog. This way we argue this method is qualified to be integrated into a development process obtaining the agile mindset. The second method is *Attribute-Driven Design (ADD)*, because it enables developing an initial architecture incrementally based on quality attributes. The initial version will be based on highest priority requirements, and it will evolve through product development releases and iterations till the architecture reaches its final form. This way ADD also enables incorporation of requirements changes as they come up. ADD contains checkpoints where design is checked for being consistent with customer requirements. The third method is *Architecture Tradeoff Analysis Method (ATAM)*, which is a collaborative architecture evaluation method which early detection of architectural design decisions which are inconsistent with customer requirements. This method facilitates discovering conflicts and tradeoff points between quality attributes, and risks that can results whenever an architectural design decision is changed. This way, change impact analysis is facilitated and a team can be aware of their architectural decisions implications on various quality attributes. A proposed framework to achieve clean light architecting is under development and will be demonstrated in upcoming papers.

Considering quality attributes early while designing translates into business value, and we know that agile teams are pursuing business value in all their decisions and practices. By designing for including quality attributes right from the beginning, resulting architecture is shaped around a long term goal rather than short-sighted goals; besides, the number of architectural refactorings that would be needed over time is expected to be reduced. Agile methods would be more qualified for developing safety-critical systems, where performance and reliability are a must. Agile teams won't be able to go for large-scale products without an architecture that offers maintainability, reusability, scalability, interoperability, and other quality attributes that can be achieved through having a light clean architecture developed incrementally and iteratively.

5. CONCLUSION

Agile architects should advocate a development culture that values making architectural design decisions based on careful analysis of requirements and give a due care to quality attribute requirements in advance, especially that they do not change as rapidly as functional requirements. There is also a need for analyzing resulting architecture carefully to assess its adoption of needed quality attributes, and to deal with conflicts between several qualities at the earliest possible development level. Planning for quality attributes in advance not only prevents

problems of missed quality attributes and implications of redesigning a system to incorporate these quality attributes, but also provides a more stable basis for the architectural design as well. Planning an architecture based on quality attributes while keeping the process light and agile is not a myth. Comprising an architecting process which harmonizes both clean and light architecting is a dream that can be easily achieved if architecting and agile development morals are well-absorbed and tackled.

6. REFERENCES

[1] Galal, G. H., (1998), "Software architecting: from requirements to building blocks within an architectural style", *Workshop W2: Techniques, Tools and Formalisms for capturing and assessing Architectural Quality in Object-Oriented Software, the 12th European Conference on Object Oriented Programming (ECOOP'98),* Brussels, Belgium, 20-24 July.

[2] Bass, L., Clements, P. & Kazman, R., (2003), Software Architecture in Practice, Addison-Wesley Professional, Boston, USA.

[3] Albin, S. T., (2003), *The Art of Software Architecture: Design Methods and Techniques,* Wiley publishing, Indianapolis, Indiana, USA.

[4] Taylor, R., Medvidovic, N. & Dashofy, E., (2009), *Software Architecture: Foundations, Theory, and Practice,* Wiley publishing, Indianapolis, Indiana, USA.

[5] Barbacci, M. R., Klein, M. H. & Weinstock, C. B., (1997), "Principles for Evaluating the Quality Attributes of a Software Architecture", CMU, Software Engineering Institute, Pittsburgh, PA, USA.

[6] Mcgovern, J., Ambler, S. W., Stevens, M. E., Linn, J., Sharan, V. & JO, E. K., (2003), *A Practical Guide to Enterprise Architecture,* Prentice Hall, Upper Saddle River, New Jersey, USA.

[7] Faber, R., (2010), "Architects as service providers", *IEEE Software*, vol. 27, no. 2, pp. 33-40.

[8] Sharifloo, A. A., Saffarian, A. S. & Shams, F., (2008), "Embedding Architectural Practices into Extreme Programming", *proceedings of the 19th Australian Software Engineering Conference (ASWEC 2008),* Perth, Western Australia, Australia, 26-28 Mar., IEEE Computer Society, pp. 310-319.

[9] Barbacci, M., Ellison, R., Lattanze, A., Stafford, J., Weinstock, C. & Wood, W., (2003), "Quality Attribute Workshops (QAWs)", CMU Software Engineering Institute, Pittsburgh, PA, USA.

[10] Mohagheghi, P. & Conradi, R., (2004), "An empirical study of software change: origin, acceptance rate, and functionality vs. quality attributes", *Proceedings of the 2004 International Symposium on Empirical Software Engineering, (ISESE '04),* Redondo Beach, CA, USA, 19-20 Aug, IEEE, pp.7-16.

[11] Mockus, A. & Votta, L. G. (2000), "Identifying reasons for software changes using historic databases", *Proceedings of the International Conference on Software Maintenance,* San Jose, CA, USA, 11-14 Oct., IEEE, pp. 120-130.

[12] Williams, B. J. & Carver, J. C., (2007), "Characterizing Software Architecture Changes: An Initial Study", *proceedings of the First International Symposium on Empirical Software Engineering and Measurement, (ESEM'07),* Madrid, Spain, 20-21 Sept., IEEE, pp. 410-419.

[13] Perry, D. & Wolf, A., (1992) "Foundations for the study of software architecture", *ACM SIGSOFT Software Engineering Notes,* Vol. 17, No. 4**,** pp. 40-52.

[14] Garcia, J., Popescu, D., Edwards, G. & Medvidovic, N., (2009), "Identifying Architectural Bad Smells", *proceedings of the 13th European Conference on Software Maintenance and Reengineering, (CSMR '09),* Kaiserslautern, Germany, 24-27 March, Winter, A., Ferenc, R. & Knodel, J. (Eds.), IEEE, pp. 255-258.

[15] Bourquin, F. & Keller, R. K., (2007), "High-impact Refactoring Based on Architecture Violations", *Proceedings of the 11th European Conference on Software Maintenance and Reengineering, (CSMR '07),*Amsterdam, Holland, 21-23 Mar., IEEE, pp. 149-158.

[16] Coplien, J. O. & Bjornvig, G., (2010), *Lean Architecture: for Agile Software Development,* Wiley Publishing, Indianapolis, Indiana, USA

[17] Jensen, R. N., Moller, T., Sonder, P. & Tornehoj, G., (2006), "Architecture and Design in eXtreme Programming; Introducing Developer Stories", *proceedings of Extreme Programming and Agile Processes in Software Engineering, 7th International Conference (XP 2006),* Oulu, Finland, 17-22 June, Springer Verlag, pp. 133-142.

[18] Sharifloo, A. A., Saffarian, A. & Shams, F., (2008), "Embedding Architectural Practices into Extreme Programming", *proceedings of the 19th Australian Conference on Software Engineering (ASWEC'08)* Perth, WA, Australia, 26-28 March, IEEE, pp. 310-319.