Software Engineering
2014; 2(1): 1-14
Published online September 30, 2014 (http://www.sciencepublishinggroup.com/j/se)
doi: 10.11648/j.se.20140201.11

A framework for architecture-centric practices integration into agile software development [APIASD]

G. H. El-Khawaga[1], Galal Hassan Galal-Edeen[2], A. M. Riad[1]

[1]Information Systems Department, Faculty of Computers and Information, Mansoura University, Mansoura, Egypt
[2]Information Systems Department, Faculty of Computers and Information, Cairo University, Cairo, Egypt

Email address:
Ghada.elkhawaga@ieee.org (G. H. El-Khawaga), Galal@acm.org (G. H. Galal-Edeen), amriad2000@mans.edu.eg (A. M. Riad)

To cite this article:
G. H. El-Khawaga, Galal Hassan Galal-Edeen, A. M. Riad. A framework for architecture-centric practices integration into agile software development [APIASD]. *Software Engineering*. Vol. 2, No. 1, 2014, pp. 1-14. doi: 10.11648/j.se.20140201.11

Abstract: The need for having a clear roadmap for a software product developed using an agile method is a reasonable implication of the agilists' tendency of establishing a planning-driven process rather than a plan-driven one, and chasing and welcoming change rather than limiting it. Building an initial architecture for a product will serve as the railway for a planning process that can enable managing change accommodation rather than unmanaged change accommodation. Change accommodation –while not considering the proposed changes' effects- may serve its purpose of flexibility on the short term, but on the long term will uncover a complex, unmanageable set of relations between software components within an eroded architecture. In this paper, a framework for embedding architectural practices into an agile software development process –while avoiding problems of current agile architecting, and keeping agile development values- is presented.

Keywords: Agile Software Development, Agile Architecting, Quality Attributes, Software Architectures, Architecting Framework

1. Introduction

The agile approach has been remarked for achieving higher quality through adopting simple to apply yet productive, and people-oriented set of practices. Agilists struggled to combine a variety of practices and activities –including architecting ones- into a process, which ensures producing agile software, which can be always responding to changing requirements and in the same time achieving business value. Architecting principles and practices advocated by agile methods are believed to result in problematic architectures [1].

Agilists' view of architecting as heavy-weighted efforts associated with much more modeling and documentation is a serious problem of agile architecting. Unfortunately, to avoid traditional architecting, agilists exaggerated into simplicity [3], to the extent of underestimating the importance of certain activities such as architectures' documenting, missing the main tenet of these activities, and considering these activities as a burden, overlooking their goal. According to Hadar, not having a roadmap of the direction where the development process is going would result in opportunistic architecture [2], in which inserting some requirements would be applicable while inserting other requirements that would cause modifications to tightly coupled components may cause problems. Another problem is that quality attributes are not given sufficient care. Simplicity or simple design is achieved -from agilists' point of view- by having a barely good enough architecture [1]. Agilists tend to avoid exerting effort on designing for unforeseen changes. Therefore, agile developers may ignore functionalities, which are not specified within user stories even if they are application-related functionalities ([3] & [6]). However; these functionalities they ignore are not all always about unforeseen changes, some of them are about quality attributes or functionalities which are necessary to be added sooner or later and the customer may not be aware of their importance. As a consequence; agile methods are accused of ignoring quality attributes such as reliability, scalability and changeability ([3], [1] & [7]), which would cause architecture breakers through the project lifetime [3]. Modifying quality attributes is believed to be costly [5] and can affect system functionalities negatively [6]. Not accommodating these changes early in the development process is sufficient to tear down the myth of having better quality using agile methods.

Also, agilists used metaphors sometimes to substitute lack of architecture [6]. It was proved in many projects that chosen metaphors often result in poor architectures, because these metaphors maybe not helpful enough or even correct [8]. Irit

Hadar showed that despite creating metaphors to achieve concretion of abstract concepts, metaphors sometimes don't provide the desired clearance and cause confusion and meaning distortion [2]. Metaphors were argued to be a methodological weak point [9], because they usually don't give precise or definite meaning and they become subject to how each member in the team would understand its meaning and tenet. This practice was widely ignored [10] because it has never been completely understood. Even agilists like Fowler claimed being not able to understand what is meant by a metaphor and how to use it ([8] & [9]).

Advocating refactoring along the way is a major problem of agile architecting, as well. Agilists claim that they can accommodate any upcoming change using refactoring. They consider it an alternative to designing up front. As refactoring strongly affects the internal structure of software in an attempt to reduce complexity, it can have implications on software architecture and its quality. Kruchten emphasized several times that architecture will not gradually emerge through refactoring as many agilists misunderstand, instead refactoring won't help in architectural decisions that are hard to undo or change lately during development process [6].. Architectural refactoring effectiveness for achieving quality is another issue that rises here. Authors like Bourquin & Keller emphasized that architectural refactoring is effective in increasing an application's maintainability and consequently reducing costs [11]. However, Buschmann mentioned that refactoring is not suitable for inserting new functions or improve operational quality attributes; because such refactorings would alter a system's behavior [12]. This claim sounds reasonable as long as the main aim of refactoring is to alter internal structure without changing external behavior, 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. Another issue is that non-systemic refactorings can result in inefficiencies that the whole system may suffer from, as Sharifloo [7] claimed. Bourquin & Keller mentioned 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 [11]. If considering architectural erosion to be a result of accumulating several architecture violations, then architectural refactorings may on the long term result in architecture erosion and as a result architecture degradation.

Architecting problems have side effects that are manifested through the limitations of agile methods. Examples of these limitations are [13]: limited support of agile development for building reusable artifacts, limited support for developing safety-critical software, and the limited support for developing large, complex software. The goal of this research is to introduce a framework for architecting in the context of agile development, while not falling into the same problems of current agile architecting trends, and maximizing business value that can be reached through employing an agile software development approach. Section two of this paper includes an introduction of previous work exerted in the same field, section three is where the proposed framework is illustrated and presented; and section four includes a verification of this framework, both from the agile software development and architecting sides. And finally, this work is concluded in section five.

2. Previous Work

Agilists and methodologists worked together to increase agile methods' applicability and to overcome architecting issues and application-oriented limitations, while preserving benefits and advantages agile methods can offer. To overcome these limitations; they had tried many techniques. In the coming subsections, two trails are explained.

2.1. Refining the Design Process

In this solution path, agilists suggested replacing the whole design process with a modified sub-process that inherits agile methods' advantages accompanied with another software development approach and at the same time avoids shortcomings of the standalone approaches. They got to a design sub-process named *Agile Model Driven Development (AMDD)*. AMDD is the agile version of Model Driven Development (MDD) [4]. According to Picek & Strahonja's definition of MDD, models are used as the primary artifacts throughout the software development lifecycle [14]. While this definition contradicts with the value of agile development which confirms that working software is the primary artifact; Ambler explained that agile models are those which are barely good enough [4]. This means that AMDD's trend is not to go far in modeling, but it is to create models which are at the most effective point they could possibly be at.

In his introduction to AMDD [4], Ambler advocated providing a big picture at the beginning of each release through the envisioning activity. However, a needed big picture would be at the level of a project release. While claiming that AMDD is a critical strategy for scaling agile software development beyond the small, co-located team approach seen in the first stage of agile adoption [15], lack of a big picture at the project level would cause challenging integration problems [16], and may result in fragile architectures, which can resist further changes. It is noticeable that in his various articles and papers illustrating AMDD ([4] & [15]); Ambler avoided defining precisely which MDD model is needed or resulting from each activity, when transformations between models are held, and how to include any missed details in the transformation process so as to be present in the resulting model. Also, while MDD is referred to as a set of approaches in which code is automatically or semi-automatically generated from more abstract models [14], Ambler has argued that using modeling tools would require modeling skill set and specialized expertise [4]. Also, Ambler argued that with AMDD, a little bit of modeling is done and then a lot of coding [15]. This declaration seems to be a clear violation of the basic idea of MDD which aims at moving the development efforts from programming to the higher level of abstraction and concentrate efforts on modeling and generating needed code from these models [14].

2.2. Using a Hybrid Approach of Agile and Traditional Practice

In this solution path, process practitioners targeted mainly agile methods' ability to scale up and their applicability for large-scale systems. They used mixtures of agile practices with plan-driven practices to construct a hybrid framework, where agile practices are used where their advantages are maximized, and plan-driven practices are used where their strengths can be maximally inherited. In balancing agility and discipline, Bohem & Turner have introduced a risk-based approach to choose the right mix of agility and control in the development strategy used for a certain application [3]. In the proposed approach, a risk-based analysis is used to decide on parts where agile risks dominate and others where plan-driven risks dominate. The amount of planning, architecting, prototyping and testing needed can be considered given the analysis results. As a result, parts can be considered, where risk-based agile process and those where risk-based plan-driven process can be applied. Bohem & Turner have applied their proposed framework on an agent-based planning system for national crisis management which is a very large highly critical system [3]. This framework provided constant monitoring and evaluation to handle risks that can come up during development lifecycle. However, this framework may require experienced staff having different skills, can precisely measure possible risks, and can carefully merge agile practices with plan-driven practices in a way that best reflects the inherited advantages of both of these approaches.

3. Introducing APIASD

Through this framework- entitled APIASD (Architecting Practices Integration into Agile Software Development)- an initial architecture is achieved to serve as the base for a final form that will emerge through continuous evolution and as a result of growing and accumulated understanding of business goals and user requirements that either come up or change through the project's lifetime. APIASD is the result of integrating tenets, some activities, and some practices of GA method [17], QAW [18], ADD [19], and ATAM [20] into an agile software development process. The reason for adopting practices and activities from these methods –especially QAW, ADD, ATAM- is that they are originally complementing each other and pursuing architecture development based on quality attributes, which is the basic objective of this work. They have the same characteristics of simplicity and reliance on joint and collaborative work, besides supporting iterative and incremental development. Using practices from GA, and ATAM is expected to facilitate performing change impact analysis that would be needed to obtain clear expectations about the effect of changes to components, connectors, and their relationships. Practices from GA also facilitate collecting information about requirements from which architectural drivers and relations between them will be extracted. Using ATAM parts is helpful in figuring out conflicts between quality attributes, so as to make suitable decisions to handle these conflicts. Using practices from QAW, and ADD can result in having a roadmap to follow in slicing incremental portions of the software to be developed based on achieving both of quality attributes side-by-side with functional user requirements without getting involved in big design upfront while gaining benefits of having a big picture of the software. For more detailed of APIASD, a level-based diagram is shown through figure 1. This figure is built upon a general model of agile development presented by Layton [21]. The steps of Layton's model are shown with a faded color in dashed squares, while additions made as a result of merging APIASD's steps with the original model are shown within white boxes.

There are no prerequisites concerning project's size, type or domain to apply APIASD upon. It is assumed that if it was agreed that an agile software development method is suitable for developing a certain software product, then this project has satisfied the prerequisites for applying APIASD on. Also, there are no modifications or changes to the structure of a team adopting an agile approach while attempting to apply APIASD. If there are additions or shifts in roles as a result of applying the framework, they will be mentioned in their context. The coming subsections include an illustration of phases of APIASD.

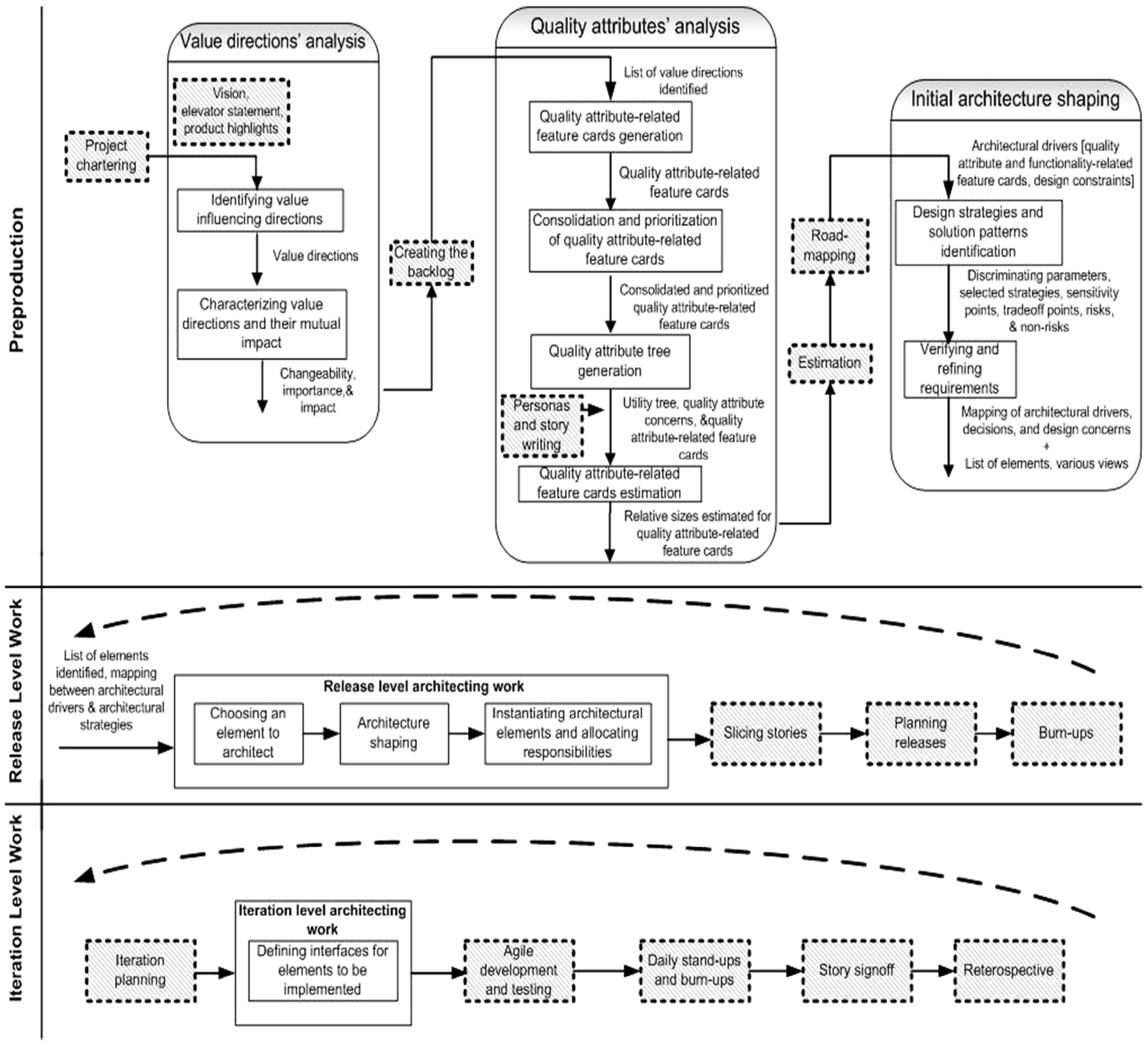
Preproduction
Value directions' analysis
Vision, elevator statement, product highlights
Project chartering
Identifying value influencing directions
Value directions
Characterizing value directions and their mutual impact
Changeability, importance,& impact
Creating the backlog
Quality attributes' analysis
List of value directions identified
Quality attribute-related feature cards generation
Quality attribute-related feature cards
Consolidation and prioritization of quality attribute-related feature cards
Consolidated and prioritized quality attribute-related feature cards
Quality attribute tree generation
Personas and story writing
Utility tree, quality attribute concerns, &quality attribute-related feature cards
Quality attribute-related feature cards estimation
Relative sizes estimated for quality attribute-related feature cards
Road-mapping
Estimation
Initial architecture shaping
Architectural drivers [quality attribute and functionality-related feature cards, design constraints]
Design strategies and solution patterns identification
Discriminating parameters, selected strategies, sensitivity points, tradeoff points, risks, & non-risks
Verifying and refining requirements
Mapping of architectural drivers, decisions, and design concerns + List of elements, various views
Release Level Work
List of elements identified, mapping between architectural drivers & architectural strategies
Release level architecting work
Choosing an element to architect
Architecture shaping
Instantiating architectural elements and allocating responsibilities
Slicing stories
Planning releases
Burn-ups
Iteration Level Work
Iteration planning
Iteration level architecting work
Defining interfaces for elements to be implemented
Agile development and testing
Daily stand-ups and burn-ups
Story signoff
Reterospective

Figure 1. *Level-based diagram of APIASD.*

3.1. Value Directions Analysis

Studying and exploring value directions that may have global effect on the entire software system is inevitable to come up with an initial idea of how a software product would look like and the rationale behind design decisions to be made about this software. This trend can help formulate architectural drivers and enable change impact analysis [17], by searching for conflicting value directions that provide more potential for changes through the software's lifecycle. Figure 2 gives an overview of basic steps of this phase.

The purpose of choosing to apply these steps after project chartering is that after defining the basic lines of a project concerning business goals, vision, team roles, and work agreements; an initial understanding of value directions affecting the product should be clarified so as to act as a guide for quality attributes, functional requirements, as well as design constraints –i.e. architectural drivers -identification. These value directions are open to modifications and changes whenever a clearer understanding of software's goals and requirements is reached. This can dispel any doubt that this step can act as a gate for inserting any heavy planning into an agile process. This phase is performed through a brainstorming session. Identifying and characterizing value directions is the first move towards identifying quality attributes explicitly in APIASD.

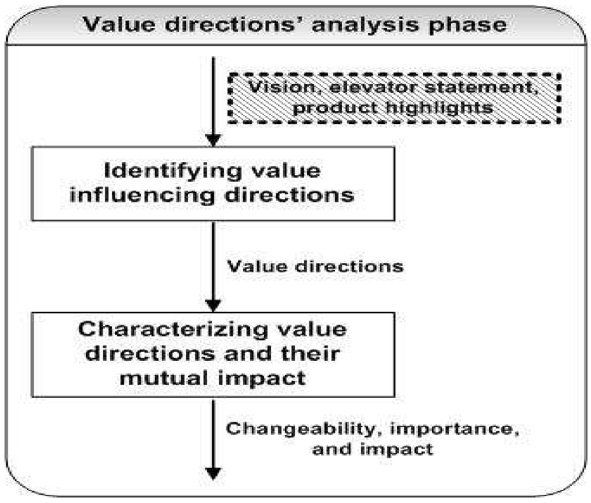
Value directions' analysis phase
Vision, elevator statement, product highlights
Identifying value influencing directions
Value directions
Characterizing value directions and their mutual impact
Changeability, importance, and impact

Figure 2. *Value directions' analysis phase.*

3.1.1. Identifying Value Influencing Directions

There is a demanding need to consider value directions coming from the context of software to be developed side-by-side with value directions resulting from understanding this software's concerns. This step prepares and supplies participants with necessary information to extract quality attributes concerns in subsequent steps. At this early stage, value directions are initial and subject to further exploration and specification. So, according to Hofmeister et al. [17], attention would be paid to identifying cross-cutting value directions that can have influence on most prospective components. Attention would be paid also to identifying value directions that may be subject to change or would be difficult to satisfy [17]. Team members should focus their efforts mainly on identifying factors representing quality attributes of the software to be built. Value directions identification is the first step towards realizing business goals.

3.1.2. Characterizing Value Directions and Their Mutual Impact

After identifying value influencing directions through the previous step, a resulting set of value directions recorded on index cards is available. Till this moment, the only relation between identified directions is that they all represent the same product, but not all of them have the same criticality for the software product to be developed. Therefore, there should be a way for differentiation between them according to their importance and their contribution to business goals achievement from stakeholders viewpoints, and their changeability and hence the software's susceptibility to change on different levels –requirements, architecture, code, or documentation levels. For each value direction defined through the previous step, development team and stakeholders should brainstorm to identify these two characteristics:

- *Changeability*. Hofmeister described this characteristic as identifying how a value direction is likely to change during or after development, and how often it will change [17]. This characteristic will be very beneficial through architecture design process, because it can help in locating components which would be more change-prone; because if a tracing is made back to requirements these components translate, there will be a direct relation with change-prone value directions.
- *Importance*. User acceptance, and business goals are the main reference for every practice, and their achievement is the main goal of every artifact resulting through any agile software development process. The intent of identifying the importance of a value direction is to define to what extent this direction is negotiable or critical from business stakeholders' viewpoint. To define this characteristic, the power of an agile team -which comprises many aspects, interests, and trends- is called here.

The product owner and the onsite customer's opinions represent directive factors while brainstorming to determine this characteristic. Also, the impact of these value directions on the expected architecture should be identified. Hofmeister suggested that characterizing a product factor's impact would be by exploring its change effect on components, other value directions, modes of operation, and other design decisions [17]. In APIASD, what matters at this stage is to identify a direction's change effect on other value directions. This will help in identifying conflicts between value directions; and based on their importance and negotiability, a decision can be made. In this early stage, it is not expected to have much information about which components will be there neither what design decisions will be made, and as it is not expected to depend on a development team members' experience.

3.2. Quality Attributes' Analysis

This phase provides a trial to convert quality attributes captured through value directions into tangible form to be addressed through and guide architecture creation. The steps of this phase are presented through figure 3. It is important to note that shaded steps are about activities held normally in the context of agile development, whether this framework is applied or not. The reason why quality attribute-related feature cards creation and shaping is suggested to be held at the stage just before story writing is that quality attributes are cross-cutting and they can have explicit effect on shaping functional requirements and as quality attributes realization should be advanced to functional requirements as it was explained before.

This phase imports practices from QAW, and ATAM, with modifications to be applicable in the context of an agile software development process. Steps and activities of this phase require presence of the architect among the whole team –which is guaranteed in teams adopting an agile process- to collaborate into discussions through which quality attributes will be made concrete. A new role of a scribe emerges. This phase is held in general through a series of brainstorming sessions, except for some practices which can be held first by individuals then these individuals join in a session to discuss their thoughts and the final output will be resulting from collaborative work.

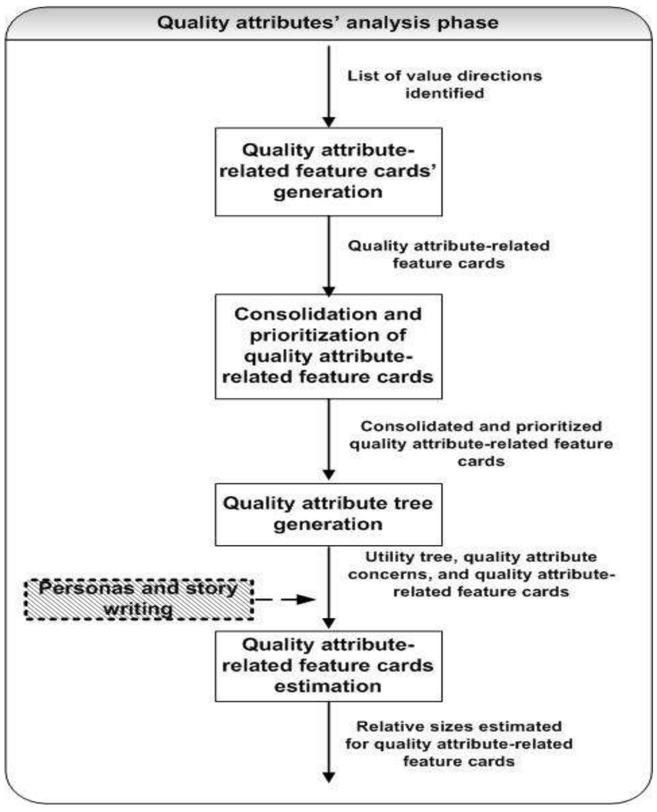
Quality attributes' analysis phase
List of value directions identified
Quality attribute-related feature cards' generation
Quality attribute-related feature cards
Consolidation and prioritization of quality attribute-related feature cards
Consolidated and prioritized quality attribute-related feature cards
Quality attribute tree generation
Utility tree, quality attribute concerns, and quality attribute-related feature cards
Personas and story writing
Quality attribute-related feature cards estimation
Relative sizes estimated for quality attribute-related feature cards

Figure 3. *Quality attributes' analysis phase.*

3.2.1. Quality Attribute Scenarios Generation

After defining software architectural drivers, they are transformed into an analyzable form that can be used to make a decision about strategies to be used to realize business goals and user requirements. Usually in agile development, quality attributes aren't explicitly addressed in feature cards neither in user stories. They can be concluded implicitly from a customer's specification of what s/he aspires to have from obtaining the software under development. A solution to this issue is to use *quality attribute scenarios*, and providing them in a form compatible with this of functional features as specified in the context of an agile software development process. Scenario brainstorming is held through at least two rounds, where each stakeholder can contribute at least two scenarios [18]. APIASD suggests supplying participants of the scenario brainstorming session with general scenarios for quality attribute requirements identified for the software to be developed so as to guide them and facilitate scenario generation. General scenarios are those which are system independent and can, potentially, pertain to any system and turned into concrete system-specific scenarios [22]. Participants in the brainstorming session will be asked to determine general scenario parts of relevance and apply them on the system in hands.

Scenario brainstorming works well with large groups that it enables creativity, facilitates communication, and helps expressing the collective minds of the participants [20], besides building group buy-in to the architecture design process. The team leader works as a facilitator, who is responsible for referring to the list of quality attributes identified and making sure that each quality attribute has at least one scenario concretizing and representing it. Another team member will act as a scribe who is responsible for filling cards with agreed upon scenarios.

Clements et al. [20] proposed some undesired properties of brainstormed scenarios like:

- Having overlapping issues and representing the same concerns. This can be handled through scenario consolidation as it will be explained later.
- Addressing issues that are unlikely to arise in the software's lifetime. This property will be overcome whenever the scenario will be prioritized according to its importance.
- Addressing issues of low priority to the development effort. Clements et al. [20] didn't clarify these issues more. However, this is unlikely to happen through APIASD, because quality attributes identified before are prioritized according to their importance, and generated scenarios will be prioritized as will be explained later.

Another property of resulting scenarios is that they may be system-specific [18]. This case demands the architect's interference to separate system-related concerns from software-concerns so as to enable software architectural decision making [18].

After finalizing scenario brainstorming session, the scribe works on transforming scenarios into a form to be compatible with feature cards embodying user stories that will be generated through story writing and persona creation sessions. A quality attribute-related feature card has some fields like:

feature ID, feature name, description, feature type – whether being technical or business-, , estimated work effort (days), story points, planned iteration, associated quality attribute concern, customer rank (H/M/L), architect rank (H/M/L), user (source of stimulus), associated features (IDs only), acceptance criteria, and associated tasks.

3.2.2. Consolidation and Prioritization of Quality Attribute-Related Feature Cards

Clements et al. [20] suggested how this step can proceed. The team leader leaves about 10 minutes for participants to review the quality attribute-related feature cards created through the previous step. Each participant takes down notes about scenarios s/he thinks should be merged. When time is out, the team leader asks participants to propose pairs of scenarios to be merged and why they think they should be merged. Then for each pair of scenarios to be merged, a decision is made through voting by raising hands. If there is a reasonable objection on the merge, then a decision would be made of leaving the addressed scenarios untouched. The agreed upon merging between scenario pairs are processed by the scribe who should read out loudly the text of remaining scenarios for participants to be assured that these scenarios are the ones they wanted to be left, while striking out the ones to be removed. Scenario consolidation helps in preventing a stakeholder from dividing his/her votes –while prioritizing scenarios- among two scenarios addressing the same issue, and subsequently eliminating their change of gaining a high priority based on their votes [20], [18].

After obtaining the final list of quality attribute scenarios, these scenarios should be prioritized. Prioritizing scenarios will help in selecting the right portion of the software to begin analyzing, based on business value, effort needed to achieve these scenarios, and impact on the architecture to be created. Scenario prioritization can enable incremental development which is the basis of agile software development. There is a need for scenario prioritization to be held again based on the relative impact of the scenario on the architecture to be developed., because differentiation based on quality attributes' preference and their impact on each other –and hence on the architecture to be developed- will affect the order in which scenarios will have solution strategies suggested for, as will be explained through the coming phase. This prioritization will be held by the architect in 10 minutes; after which, s/he should present to the whole stakeholders and team members ranked scenarios through an oral presentation of the rationale for these rankings. The architect's ranking of scenarios should be based mainly –besides his/her experience- on the quality attribute-related value directions' impact on each other and on the architecture to be created, as well as their changeability. Priorities would be in the form of (H/M/L); while "H" denotes a high rank, "M" denotes a medium rank, and L denotes a low rank.

3.2.3. Quality Attribute Tree Generation

To step towards identifying strategies to achieve the desired quality attributes, a utility tree is generated. Constructing a utility tree serves to map generated scenarios to the quality attributes they are representing, so as to ensure there is no missed user desired quality attribute not to be addressed by the architecture to be developed. In ATAM, a utility tree is used to move from general to specific, i.e. to begin with quality attributes and have team leader and architects getting together to identify scenarios representing the desired quality attributes [20]. In APIASD, quality attributes and their representing user required scenarios are already there; but still there is a need to identify –based on the identified scenarios- which aspects or quality attribute concerns really matter. Identifying needed quality attribute concerns enables figuring out commonly used strategies and patterns addressing those concerns. A utility tree enables relating several scenarios to one quality attribute; and therefore facilitates discovering whether they have conflicts together or with scenarios representing other quality attributes. In presence of a visual representation of mappings and relations between scenarios and quality attributes the way offered by a utility tree; further conflicts can be handled whenever their sources are located. Constructing a utility tree is one of the architecture-related practices which make an architecture evaluation –whenever carried out- becomes more about a confirmation exercise than being an investigatory one.

A utility tree is built upon four levels [20], [22]; utility is the root node, the second level contains quality attribute requirements specified through value directions, the third level contains quality attribute concerns which are to be specified through this step, and the forth level contains scenarios relevant to each quality attribute associated with their pair of rankings.

Quality attribute concerns –or as Wojcik et al. [19] called them design concerns- are refinements to quality attributes that bring more specification into a wide definition of a quality attribute to help facilitate identification of a suitable solution [19]. Identifying quality attribute concerns is the responsibility of an architect. The architect is the one to construct an initial version of the utility tree, before holding a meeting with the team where this initial utility tree is discussed and refined. The architect should clarify to the team why s/he thinks a scenario represents a certain quality attribute concern.

After making sure that all the identified quality attributes and their associated scenarios are represented through quality attribute-related feature cards, the team works collaboratively on identifying, prioritizing, and grouping related functionality-related feature cards. Having quality attribute-related feature cards present, while writing functionality-related feature cards, will facilitate deciding dependencies of functionalities on several quality attribute scenarios, without going through detailed design work. Identifying dependencies also will affect how functional features will be grouped into related clusters. It worth mentioning here that identifying dependencies of functionality-related features upon quality attribute-related features doesn't necessitate implementing quality attribute-related feature cards before functionality-related ones. Instead, it obligates implementing the associated quality attribute-related features as a precondition to have a functional feature gaining acceptance not as a precondition to have it operational. So, identified quality attribute-related feature cards are placed in the product backlog till functionality-related ones are prepared to have both types of cards estimated to determine their relative sizes.

3.2.4. Quality Attribute-Related Feature Cards Estimation

Scenario estimation or quality attribute-related feature cards' estimation is a practice held the same way as for functionality-related feature cards. Smith & Sidky explained estimating relevant size of feature cards using story points [23]. Each feature card is assigned a number indicating its size in relevance to other features. Numbers are derived from the Fibonacci scale. These numbers represent what is meant by story points. Numbers can be replaced with any objects the team agreed to use. Smith & Sidky [23] emphasized the usefulness of using story points –in absence of detailed information about tasks and effort estimation (in terms of days) for a certain feature- while allocating features to a release and determining an iteration within which these features are implemented. Estimating a feature size is a group effort that represents the collective mind of a team's members and increases their sense of ownership for the project they are working on.

3.3. Architecture Shaping Phase

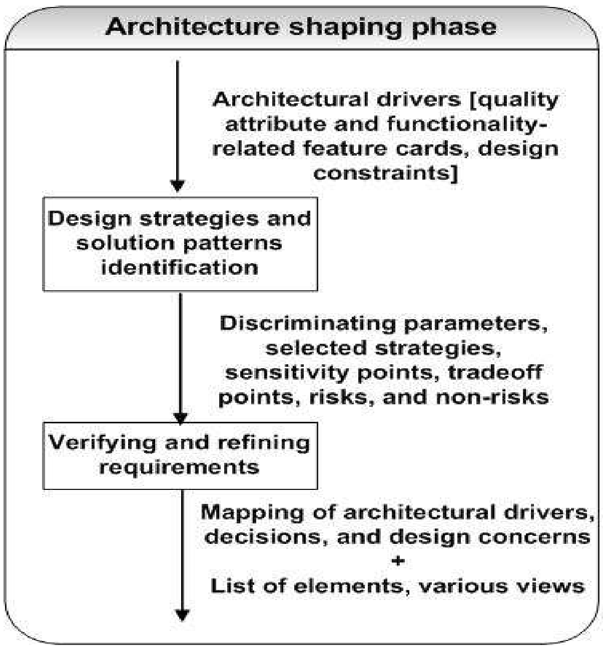
Architecture shaping phase
Architectural drivers [quality attribute and functionality-related feature cards, design constraints]
Design strategies and solution patterns identification
Discriminating parameters, selected strategies, sensitivity points, tradeoff points, risks, and non-risks
Verifying and refining requirements
Mapping of architectural drivers, decisions, and design concerns + List of elements, various views

Figure 4. *Architecture shaping phase.*

Now; there is a need for defining a roadmap for the project's components' relations and interactions, how these components are expected to satisfy quality attribute requirements, and how an integration between quality attribute and functional requirements is to be managed. This stage is needed so as not to waste a team's effort just in building structure while increasing long term costs, in absence of an architecture whose structure follows its form. Figure 4 highlights main steps of this phase. Activities and practices of this phase are mainly adopted from ADD method, while identifying tradeoffs and risks associated with strategies and architectural decisions made are inspired from ATAM practices. At this stage, the resulting initial architecture is built upon the highest priority quality attribute concerns and features, and the remaining ones are inserted into the architecture through conducting further iterations of the same phase presented in this subsection. This architecture evolves and grows to incorporate further features and quality attribute concerns as changes hit by the product to be developed. Attendance of the architect is not optional; because many activities carried out are guided by the architect's experience and knowledge as well as his/her discussions with the whole team.

3.3.1. Design Strategies and Solution Patterns Identification

This step begins by receiving the highest priority quality attribute-related as well as functionality-related features along with design constraints defined previously in value directions' identification step. These inputs serve as architectural drivers. Building the architecture based on highly prioritized quality attribute scenarios/features helps in figuring out main design concerns –by tracing these scenarios back to their related quality attribute concerns identified in the product's utility tree- besides being useful in identifying discriminating parameters to be the basis for preferring between suggested design decisions to address design concerns identified. Using functionality-related features will help in considering their related design concerns through checking scenarios upon which a feature depends and mapping these scenarios to their associated quality attribute concerns in the utility tree.

After identifying design concerns associated with selected architectural drivers, the architect defines discriminating parameters [19], which serve as conditions that should be met by selected strategies. These parameters represent comparison criteria whose values are used to select a strategy or a design decision from the candidate solution strategies and design decisions of each design concern. Discrimination parameters are derived from response measures of scenarios related to the design concerns, or from user requirements that show quality attribute-related concerns like a requirement that a software system should be able to support 100 client queries.

Defining design concerns helps in identifying and pointing out the search areas of strategies needed to satisfy these concerns. According to Bass et al. [22], an architectural strategy can be defined as *a collection of design decisions that influence the control of a quality attribute response*, while an architectural pattern represents *a package of strategies that are concerned with different quality attributes but have a basic motivation*. A list of candidate patterns to satisfy identified concerns can be obtained depending on the architect's experience and knowledge, or through commonly used and known architectural strategies for achieving quality attributes [19]. Quantitative models aren't applicable here due to the crosscutting nature of quality attributes which makes building an executable model requires having a complete architecture, which is not available at this stage. Add to this having some quality attributes which cannot be quantified like modifiability. Also, building quantitative models is not cost effective at this stage. Therefore, APIASD suggests making estimates of expected values of discriminating parameters based on architect's and development team experience. After determining discriminating parameters' values, a decision should be made about which pattern to select based on a discussion between the architect and the development team members. A decision matrix is constructed, like the one represented by table 1.

The decision column in the table below is where the pattern chosen is recorded, while the implications column is used for recording further analysis done to enable discovering expected consequences of the decision made. The purpose of analyzing consequences of an architectural decision is to provide insights into possible dependencies between decisions made and decisions to be made or to highlight possible impact of making changes to the chosen pattern or one of its components before making this change.

Table 1. *Architectural patterns' decision matrix.*

	Pattern 1	…..	Pattern M		Decision	Implications
	Discriminating Parameter 1	Discriminating Parameter (N)	Discriminating Parameter 1	Discriminating Parameter (N)		
Design concern 1 **:** **Design concern N**	Values associated with each discriminating parameter of each design concern go here				Chosen pattern's name goes here	

Analyzing consequences of an architectural decision can also highlight the extent to which a change can affect achievement of the addressed quality attribute or/and other quality attributes. Building on the agile mindset, obtaining a precise and clear analysis of an architectural decision change consequences bears uncertainty, because they are about expectations of the future. Besides, it is impossible to design an architecture that accounts for all possible evolutions in the software's requirements. However, given the requirements and their associated design decisions, it is possible to define some cases where change can cause a conflict with other attributes or might be risky to the whole software. Implications can include sensitivity points, tradeoff points, risks, and non-risks resulting from applying the chosen pattern. A sensitivity point for a certain quality attribute represents a decision with an effect on the degree to which this quality attribute can be achieved. Variations in the decision are followed by a serious variation in the resulting value of the quality attribute. Discovering sensitivity points of an architectural decision –like discovering tradeoff points, risks, and non-risks- is a collaborative activity based on asking elicitation questions. These questions can be inspired by the architect's experience.

For a software architecture planned to address many quality attributes and their concerns, it is expected to have relationships between these quality attributes, and subsequently between strategies chosen to achieve them. Franch & Carvallo characterized relations between quality attributes to be as either [24]: collaboration, increasing a quality attribute concern will lead to an increase in another; or damage, decreasing a quality attribute concern will lead to a reduction in another; or dependency, a quality attribute requires achieving some level of another quality attribute to be achieved. Identifying the relationship type between two quality attributes can provide a way to identify tradeoff points between techniques chosen. In general, tradeoff points between quantified quality attributes are easier to identify than those involving quality attributes that cannot be quantified like modifiability for example [25]. This case highlights the necessity of tracing quality attributes that cannot be quantified back to their related scenarios and value directions to get information about which value directions –and hence quality attributes- are affected. This is done through the impact characteristic associated with each product factor. Having tradeoff points affecting quality attributes of high importance can raise risks about making related architectural decisions. These risks should be highlighted. Also, a non risk is *an architectural decision that is appropriate in the context of the quality attributes that it affects* [26]. The importance of identifying non-risks lies in examining whether it remains a non-risk and represents a strength point of an architecture decision whenever this architecture decision changes.

3.3.2. Verifying and Refining Requirements

Now, a mapping should be done between selected strategies and architectural drivers. The purpose of this mapping is to ensure that no architectural driver was missed in the strategies' identification process. Moreover, visualizing which strategies should together serve to reach a specific architectural driver can help in looking for inconsistencies and decide how to resolve them. As the architecture to be developed in this stage is an initial one and is subject to evolution through the project's lifecycle, and as the chosen architectural drivers will be allocated to releases as requirements; mapping and grouping chosen strategies to architectural drivers will help in reusing decisions and implications identified in upcoming architecture-related activities through the project's releases. A skeleton for tables representing desired mappings should include architectural drivers, associated design concerns, and associated architectural decisions. The identified mapping can help in deciding how patterns relate to each other and give insights into new element types that emerge as a result of combining patterns [19].

By this stage, a list of software elements up to this moment is available. Having a list of elements available, it is time to move into more visualization of a proposed architecture. The development team gathers to construct needed views that are expected to provide a general overview of the architecture after identifying its basic lines. Types of views to be constructed are influenced by which quality attributes are highly required for the product to be developed. For example, if performance is a critical quality attribute, then a process view is required. Constructing such views enables defining integration points between functional and quality attribute requirements. Weaving the quality attribute requirements with the functional ones, according to Moreira et al. [27], can be in one of the following forms:

- *Overlapping*, where a quality attribute requirement modifies a functional one it transverses. This case is translated through having the quality attribute requirements required before or after the functional ones.
- *Overriding*, where a quality attribute requirement superposes a functional one it transverses. This case is translated through having the behavior described by a quality attribute requirement substituting the functional one's described behavior.
- *Wrapping*, where a quality attribute requirement encapsulates a functional one it transverses. This case is described by having the behavior described by a functional requirement wrapped by this described by a quality attribute requirement.

The development team should return back to update the project backlog to add feature cards about new elements to be prioritized, grouped, and estimated as was done for all other cards; and these elements should be reflected through constructed views.

3.4. Architecture-Related Release and Iteration Work

APIASD is well-formed to suit the incremental and iterative nature of agile software development. It includes work to be done at the release as well as the iteration levels, and this work should be repeated till a final version of the needed architecture emerges. Figure 5 represents steps of this phase, while faded squares represent either phase three that will be repeated; or work held normally in the context of agile development, whether or not APIASD is applied.

3.4.1. Choosing an Element to Architect (Release-Level)

Choosing and allocating suitable features to be developed through upcoming release is much more facilitated through using APIASD, because the generated initial architecture provides a roadmap of basic elements of a product to be, and all what the development team need to do is to choose an element of the system to work on in the coming release. Wojcik et al. [19] suggested that choosing an element to focus on in the coming cycle of architecture development can be subject to one of four criteria or a compromise between them all. These criteria are business ones, organizational ones, risk and difficulty ones, or current knowledge available of the architecture. These criteria are easily defined, because architectural strategies identified are mapped to their architectural drivers, which were built upon value directions that are characterized, and their associated development risks can be figured out. Add to this that strategies used were elicited to identify their sensitivity and tradeoff points besides associated risks. At this stage where the basic lines of an architecture have been already formed and it is a matter of assignment of an element to a release, risk and difficulty of developing an element can be defined, not precisely but to a limit good enough to make a decision upon.

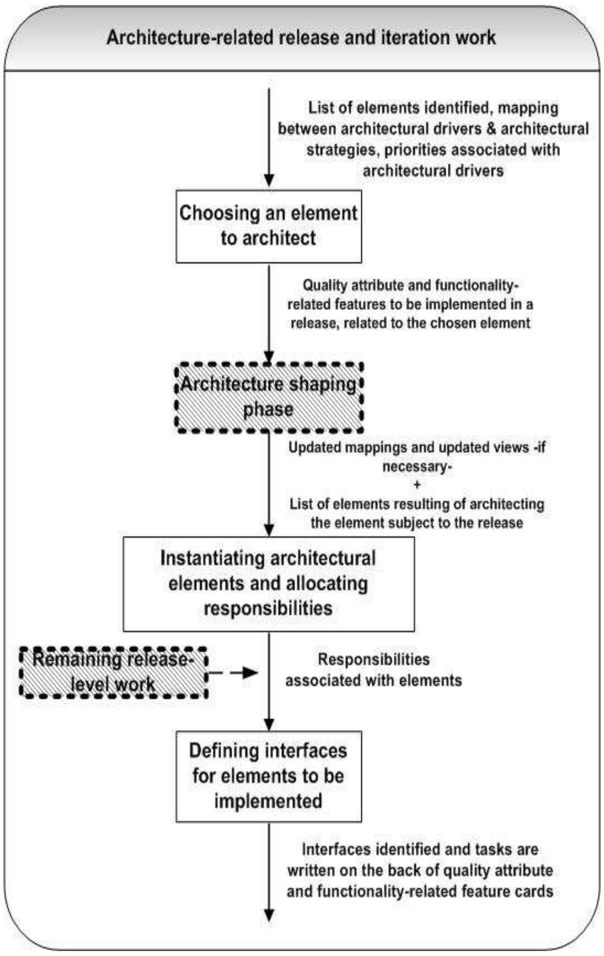
Architecture-related release and iteration work
List of elements identified, mapping between architectural drivers & architectural strategies, priorities associated with architectural drivers
Choosing an element to architect
Quality attribute and functionality-related features to be implemented in a release, related to the chosen element
Architecture shaping phase
Updated mappings and updated views -if necessary-
+
List of elements resulting of architecting the element subject to the release
Instantiating architectural elements and allocating responsibilities
Remaining release-level work
Responsibilities associated with elements
Defining interfaces for elements to be implemented
Interfaces identified and tasks are written on the back of quality attribute and functionality-related feature cards

Figure 5. *Architecture-related release and iteration.*

3.4.2. Instantiating Architectural Elements and Allocating Responsibilities (Release-Level)

This step is the same as its equivalent in ADD. Having an initial list of software elements, these elements are instantiated and assigned responsibilities according to their types [19]. Bass et al. [28] and Wojcik et al. [19] suggested some criteria that can be used to group or allocate functionality like: functional coherence, where requirements grouped together exhibit low coupling and high cohesion; or similar patterns of data or computation behavior, where responsibilities showing similar patterns of behavior, like for example, accessing database in a similar fashion; or similar levels of abstraction, where responsibilities related to hardware and others which are more abstract are separated, or locality of responsibility, where purely local responsibilities should be separated from those related to providing services to other elements.

3.4.3. Defining Interfaces for Elements to be Implemented (Iteration-Level)

In this step; interfaces between elements allocated to the current iteration and other elements –which are needed to perform features to be obtained from an iteration- should be identified. According to Wojcik et al. [19], an element's interface describes the PROVIDES and REQUIRES assumptions a software element makes about others or are made about it. Patterns and architectural decisions manifested through software elements can ease specifying interactions between them. For example, *calls*, *subscribes to*, and *notifies* are some kinds of interactions that come specific based on used patterns [28]. This step yields writing tasks associated with each feature card, and integrates well with feature card modeling activity held by the development team which is explained by Smith & Sidky [23], where team members walk through a feature to identify needed elements, interfaces and interactions between them, and decide needed tasks to implement this feature. After a team's understanding of user requirements –both quality attributes and functionalities- grows, and as the project continues to provide working software through releases; the project's software architecture continues evolving till it reaches its final form.

4. Verifying APIASD

This section aims at proving that APIASD is adhering to basic lines of its guiding umbrellas. APIASD is proved to follow software architecture design basics and achieving them. In the same time, it shouldn't violate agile software development values and principles. For achieving this purpose, a defining model of architecting methods and a guiding set of agile software development distinguishing criteria are used to identify the APIASD's level of adherence to both sides.

4.1. Architecting-Wise Verification

Several authors tried to define criteria representing commonalities between architecture design methods and to generalize them to reach a model for software architecture creation approach. Depending on the general model proposed by Hofmeister et al. [29], developing a software architecture is based on three activities; architectural analysis, architectural synthesis, and architectural evaluation; and produce some artifacts, like candidate architectural solutions, and views representing the produced architecture. The following points summarize how APIASD elements achieve and map to each criteria of the general model of software architecture design as proposed by Hofmeister et al. [29]:

- *Architectural analysis:* this activity is achieved through Identifying and characterizing value directions, then quality attribute scenarios' creation and their transformation into prioritized feature cards, and related activities; serve to elicit and form requirements based on context and concern-related needs. Through Determining highly prioritized quality attributes and functional requirements to begin with identifying architectural drivers that will shape the resulting architecture.
- *Architectural synthesis:* This activity is done incrementally and iteratively through APIASD. Architecture shaping phase encapsulates the architectural synthesis effort. Design strategies are identified and associated with them their sensitivity points and possible risks. Identified strategies are translated into responsibilities allocated to software elements. For each element, interfaces are identified at the beginning of the iteration when the element is going to be implemented.
- *Architectural evaluation:* A mapping between architectural drivers and chosen strategies is constructed to ensure satisfaction of all architectural drivers. Basing architectural decisions' making on discriminating parameters and clearly documenting this provides a chance for revising these decisions. Besides, necessary views are constructed to visualize the relations between elements and make sure that no architectural driver was missed. Also views' creation helps in integrating quality attributes and functional requirements.
- *Architectural concerns:* are handled through Value directions, functional requirements, quality attribute requirements, quality attribute concerns, design constraints
- *Context:* Team members can impose value directions –and hence quality attribute scenarios- based on domain-experience or organizational constraints, as long as the imposed value directions are aligned with business goals of the product to be developed.
- *Architecturally significant requirements***:** are handled through having Prioritized architectural drivers chosen to be the basis for strategies and patterns selection. They have the most impact on the element chosen to be the subject of a release.
- *Candidate architectural solutions:* Strategies and patterns are proposed for each quality attribute concern. Then a decision is made based on discriminating parameters' values, and team experiences and knowledge of similar cases. Architectural strategies and patterns are up to modification based on subsequent analysis and understanding in upcoming releases.
- *Validated architecture:* The mapping constructed between architectural drivers and chosen architectural decisions represents act as a joining hinge between the cause and result serving to form an initial architecture validated through associated discriminating parameters, and non-risks identified. Also, at the beginning of each release where an element is chosen to be analyzed more, its associated architectural drivers are chosen. The initial architecture is validated for satisfaction of high priority requirements and constraints with respect to the

decomposition.

- *Backlog:* APIASD emphasized the importance of placing prioritized quality attribute scenarios –i.e. representative feature cards- and functionality-related cards in the product backlog. On the other hand, creating an initial architecture and listing its basic elements can guide decomposing a software system into increments and identify quality attribute and functional requirements to be the focus of a certain release. Therefore, APIASD enables and relies on the concept of a backlog.
- *Views:* APIASD doesn't impose creating a certain set of views. There is an emphasis on creating simple views which are barely good enough to represent the created architecture and as simple as possible to communicate the design and achieve the purpose of their creation.

Also, APIASD is believed to meet the guiding principles of architecture development provided by Rozanski & Woods [30]. Among the met guidelines is APIASD's ability to integrate with any chosen agile development method, and being technology neutral. APIASD guides an architect in using solution strategies by basing the choice decision on discriminating parameters derived from quality attribute scenarios to ensure that chosen solution strategies are driven by architectural drivers. APIASD also guides the architect in his/her future use of chosen solution strategies by imposing identification of factors or decisions that can be sensitive either to the same quality attribute the strategy is addressing or to other ones and may cause a conflict or influence them negatively. APIASD suggests using practices that can facilitate communicating architectural decisions to stakeholders and collaborating during decision making process.

Falessi et al. defined nine categories of architects' needs to be this basis for evaluating software architecture design methods [31]. These criteria and how they are mapped on APIASD are as follows:

- *Abstraction and Refinement:* APIASD's ability to add details is present through further decomposition of an element in subsequent releases. Also, views' construction helps in discovering more details about elements' interactions. While removing details is shown through utility tree creation and capturing the quality attribute concern behind each scenario.
- *Empirical validation:* APIASD is based on GA, QAW, ADD, and ATAM. These methods (except for ATAM) are proposed and verified on paper only. However, a case study was used to validate APIASD, and a paper on this issue is under revision.
- *Risk management:* Risks are identified through APIASD, and handling them whenever they occur is up to the architect and development team to choose suitable strategies to mitigate risks.
- *Interaction management:* Interactions between strategies chosen are discovered and their impact change can be managed through making decisions, while considering affected quality attribute' priorities.
- *Concerns*: APIASD supports presenting different concerns and considering them. Architecture concerns are identified through early phases and the resulting architecture is presented by different views
- *Tool support:* There is not a specific tool adapted for usage by APIASD. However, through the comparison presented by Tang et al. [32], some tools were found to be able to offer some capabilities required by APIASD; examples of these tools are PAKME and ADDSS. Also, Bohem & In introduced QARCC, a tool for discovering conflicts between quality attributes [33].
- *Knowledgebase:* A few number of tools provide knowledge repository about scenarios and patterns; an example is PAKME [34].
- *Requirements management:* APIASD enables architecture development incrementally and iteratively, which facilitates responding to requirements changes and accommodating them. Also, each strategy decided for usage is analyzed to discover potential change impact whenever a change occurs.
- *Number of activities:* All classes of architecture development-related activities are covered.
- APIASD ensures the balance between being customer value-driven, while taking technical considerations into account, while prioritizing user stories.

4.2. Agile-Wise Verification

APIASD takes a minimalist approach to architecture while the highest priority architectural drivers are identified and the simplest design effort is done to achieve them. An initial version of the architecture enables project management to organize work assignments, and configuration management to setup development infrastructure; and the product builders to decide on the test strategy [28].

Having APIASD based on methods like QAW, ADD, and ATAM provides it flexibility to have barely good enough practices to reach an initial architecture early, while postponing those practices concerned with functional requirements –like instantiating elements and allocating responsibilities- to be done on the release-level and those approaching technical details and implementation-related details –like identifying interfaces between elements- to be done on the iteration-level. This development trend can enable simplicity in design, responding to changes, and waiting till more uncertainty about user requirements is reduced.

Changes in quality attributes are to be managed properly through building an architecture where insights into changes' impacts on quality attributes –if any- can be provided. Concentrating on driving the resulting architecture by quality attribute requirements is expected to reduce the need for architectural refactoring to include quality attributes. APIASD aims at finding a balance between investing in quality improvements, and feature growth so as to keep on achieving value both in the short and long terms.

Practices proposed through APIASD praise and encourage individuals and interactions among them through brainstorming sessions and joint decision making facilitated by discussions and voting. All team members are equally important and their interference and presence is adding to the effectiveness of architecture development process. Even in activities where the architect's presence is mandatory to have his/her opinion providing guidance and added value to decision making; a decision made is not made individually by the architect and just communicated through documents and diagrams to the whole team without persuasion or discussion as done in the context of traditional development methods. Customer collaboration is maximized and communication between different stakeholders is supported and facilitated through stakeholders' –especially the customer- involvement through all framework activities. This enables shorter feedback cycles and ensures always being driven by user requirements; especially architectural significant ones.

In the context of APIASD, architectural artifacts produced are kept to the minimum and don't impose exerting much documentation efforts. Documenting architectural artifacts is handled the way that enables travelling light –in agile development terminology- and helps proceeding to another software development stage, while keeping early design decisions that critically affect all subsequent development efforts. It is up to the development team to choose the best way to radiate architectural information through a project; but for storage and reusability purposes, keeping an electronic copy of the architectural artefacts produced, like views and decision tables, is advisable.

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. APIASD presents a way to develop an architecture driven by architectural drivers, especially quality attribute requirements, that serve to shape an architecture and give it the ability to survive, while absorbing and accommodating changes as they come up. APIASD can successfully bridge the gap between requirements and software architecture and yield a roadmap to be used to build a software product that can achieve maximum level of customer value as a basic goal. Through APIASD is believed to be aligned with the agile software development mindset conveyed through agile values and principles as well as architecting basic principles.

References

[1] 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.

[2] Mancl, D., Hadar, E., Fraser, S., Hadar, I., Miller, G. R. & Opdyke, B., (2009), "Architecture in an Agile World", proceedings of the *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPLSA 09),* Orlando, Florida, USA, 25-19 October, ACM, pp. 841-844.

[3] Boehm, B. & Turner, R., (2004), *Balancing Agility and Discipline: A Guide for the Perplexed,* Addison Wesley Professional, Indiana, USA.

[4] Ambler, S. W., (2007), "Agile Model Driven Development (AMDD)", *XOOTIC*, Vol. 12, No. 1, pp. 13-21.

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

[6] Abrahamsson, P., Babar, M. A. & Kruchten, P., (2010), "Agility and Architecture: Can They Coexist?", *IEEE Software*, Vol. 27, No. 2, pp. 16-22.

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

[8] Khaled, R., Barr, P., Noble, J. & Biddle, R., (2004), "Extreme programming system metaphor: A semiotic approach", *proceedings of the 7th International Workshop on Organizational Semiotics,* Setúbal, Portugal, 19-20 July, pp. 152-172.

[9] West, D., (2002), "Metaphor, Architecture, and XP", *proceedings of the third International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2002),* Sardinia, Italy, 26-30 May, pp. 101- 104.

[10] Herbsleb, J., Root, D. & Tomayko, J., (2003), "The eXtreme Programming (XP) Metaphor and Software Architecture", School of Computer Science, Carnegie Mellon, Pittsburgh, PA, USA.

[11] 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.

[12] Buschmann, F., (2011), "Gardening Your Architecture, Part 1: Refactoring", *IEEE Software,* Vol. 28**,** No. 4, pp. 92-94.

[13] Turk, D., France, R. & Rumpe, B., (2002), "Limitations of Agile Software Processes", *the third International Conference on Extreme Programming and Agile Processes in Software Engineering* (XP 2002), Sardinia, Italy.

[14] Picek, R. & Strahonja, V., (2007), "Model Driven Development – Future or Failure of Software Development?", *proceedings of the 18th International Conference on Information and Intelligent Systems (IIS2007),* Varaždin, Croatia, 12-14 September, pp. 407-414.

[15] Ambler, S. W., (2007), "Agile Model Driven Development (AMDD): The key to scaling agile software development", available at: http://www.agilemodeling.com/essays/amdd.htm , last access: 10 September 2014.

[16] Elssamadisy, A. & Schalliol, G., (2002), "Recognizing and responding to bad smells in extreme programming", *proceedings of the 24th International Conference on Software Engineering (ICSE'02),* Orlando, Florida, USA, 19-25 May, ACM, pp. 617- 622.

[17] Hofmeister, C., Nord, R. & Soni, D., (2000), *Applied Software Architecture,* Addison Wesley Professional*,* Indiana, USA.

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

[19] Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R. & Wood, B., (2006), "Attribute-Driven Design (ADD)", CMU Software Engineering Institute, Pittsburgh, PA, USA.

[20] Clements, P. C., Kazman, R. & Klein, M., (2002), *Evaluating software architectures: Methods and case studies,* Addison Wesley Professional, Indiana, USA.

[21] Layton, M. C., (2012), *Agile Project Management For Dummies,* John Wiley & Sons Inc., NJ, USA.

[22] Bass, L., Clements, P. & Kazman, R., (2003), *Software Architecture in Practice,* Addison Wesley Professional, Indiana, USA.

[23] Smith, G. & Sidky, A., (2009), *Becoming Agile in an imperfect world,* Manning Publications Co., NY, USA.

[24] Franch, X. & Carvallo, J. P., (2003), "Using quality models in software package selection", *IEEE Software,* Vol. 20**,** No. 1, pp. 34 – 41

[25] Hochmuller, H., (1999), "Towards the Proper Integration of Extra-Functional Requirements", *Australasian Journal of Information Systems,* Vol. 6**,** No. 2, pp. 98-117.

[26] Barbacci, M., Clements, P., Lattanze, A., Northrop, L. & Wood, W., (2003), "Using the Architecture Tradeoff Analysis Method (ATAM) to Evaluate the Software Architecture for a Product Line of Avionics Systems: A Case Study", CMU Software Engineering Institute, Pittsburgh, PA, USA.

[27] Moreira, A., Araujo, J. & Brito, I., (2002), "Crosscutting quality attributes for requirements engineering", *proceedings of the 14th international conference on Software Engineering and Knowledge Engineering (SEKE'02),* Ischia, Italy, 15-19 July, ACM, pp. 167-174.

[28] Bass, L., Klein, M. & Bachmann, F., (2001), "Quality Attribute Design Primitives and the Attribute Driven Design Method", CMU Software Engineering Institute, Pittsburgh, PA, USA.

[29] Hofmeister, C., Kruchten, P., Nord, R. L., Obbink, H., Ran, A. & America, P., (2007), "A general model of software architecture design derived from five industrial approaches", *Journal of Systems and Software,* Vol. 80**,** No. 1, pp. 106-126.

[30] Rozanski, N. & Woods, E., (2005), *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives,* Addison-Wesley Professional, Boston, USA

[31] Falessi, D., Cantone, G. & Kruchten, P., (2007), "Do Architecture Design Methods Meet Architects' Needs?", *proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'07),* Mumbai, India, 6-9 January, IEEE, 44-53 .

[32] Tang, A., Avgeriou, P., Jansen, A., Capilla, R. & Babar, M. A, (2010), "A comparative study of architecture knowledge management tools", *Journal of Systems and Software,* Vol. 83**,** No. 3, pp. 352–370.

[33] Boehm, B. & In, H., (1996), "Identifying quality-requirement conflicts", *IEEE Software,* Vol. 13**,** No. 2, pp. 25- 35.

[34] Babar, M. A. & Capilla, R., (2008), "Capturing and Using Quality Attributes Knowledge in Software Architecture Evaluation Process", *proceedings of the 1st International Workshop on Managing Requirements Knowledge(MARK'08),* Barcelona, Spain, 8 September, IEEE, pp. 53-62.