Towards an Evidence-Based Understanding of Emergence of Architecture Through Continuous Refactoring in Agile Software Development

Lianping Chen[12], Muhammad Ali Babar[3]
[1]Lero - The Irish Software Engineering Research Center, University of Limerick, Limerick
[2]Paddy Power PLC, Dublin, Republic of Ireland
[3]The University of Adelaide, Adelaide, Australia
lianping.chen@lero.ie, ali.babar@adelaide.edu.au

***Abstract*—The proponents of Agile software development approaches claim that software architecture emerges from continuous small refactoring, hence, there is not much value in spending upfront effort on architecture related issues. Based on a large-scale empirical study involving 102 practitioners who had worked with agile and architecture approaches, we have found that whether or not architecture emerges through continuous refactoring depends upon several contextual factors. Our study has identified 20 factors that have been categorized into four elements: project, team, practices, and organization. These empirically identified contextual factors are expected to help practitioners to make informed decisions about their architecture practices in agile software development.**

Keywords—software architecture; agile software development component; empirical study

I. INTRODUCTION

Agile software development approaches have been widely adopted in the industry [1-2] for improving a software development team's ability to easily accommodate changes and delivery working software continuously and incrementally. Despite the increasing popularity of Agile approaches, there is huge scepticism about developing large scale critical software systems because of lack of attention paid to the issues related to software architecture by Agile followers [3]. This situation has caused intense debate among software development practitioners and researchers [2].

One of the most fundamental points of the debate is "*whether or not a satisfactory architecture emerges from continuous small refactoring in agile software development*" [4]? A satisfactory architecture is one that satisfies the architecturally significant requirements [5] of the software system. This issue has been so fiercely debated because many agile approaches encourage their followers not to worry about software architecture related issues upfront under the assumption that "*a satisfactory architecture can emerge from continuous small refactoring.*"

The proponents of architecture-centric approaches warn against this assumption and quote stories of disastrous consequences for not paying sufficient attention to architecture-centric issues through a system's lifecycle. Hence, there has been significant interest in and support for finding a middle ground by integrating sound architectural practices and principles in agile approaches [2-3, 6].

However, there is not much empirical evidence in support or against the key assumption of agile approaches "*a satisfactory architecture emergences from continuous refactoring*"; and what are the factors that facilitate or inhibit the emergence of a satisfactory architecture through continuous refactoring? We assert that it is important to systematically gather and rigorously analyse empirical data to provide an evidence-based understanding about the emergence of a satisfactory architecture through continuous refactoring.

We have conducted a large-scale empirical study that explored the observations and experiences of 102 experienced practitioners with regards to the claim that "*a satisfactory architecture emergences from continuous small refactoring*." We carried out email-based semi-structured interviews and analyzed the collected data using descriptive statistics and techniques from Grounded Theory (GT) [7].

The results reveal that whether or not a satisfactory architecture can emerge depends upon contextual factors. Our study has identified 20 key factors that can support or inhibit the emergence of a satisfactory architecture through continuous refactoring in agile software development. We have classified those 20 factors into four elements that have been presented as a framework in this paper. Those four elements are: project, team, practices, and organization. This four elements framework and its 20 factors have enabled us to discuss the contexts in which a satisfactory architecture is likely to emerge through continuous small refactoring or otherwise. The findings are expected to help practitioners to make informed decisions about their architectural practices in Agile software development. We believe that this evidence-based understanding is important to address the commonly observed lack of attention paid to architecture-centric activities in Agile software development [8].

II. BACKGROUND AND MOTIVATION

Agile approaches (such as Extreme Programming (XP) [9], Crystal Clear [10] and Scrum [11]) are based on certain assumptions [12] and Agile manifesto[1]. Since Agile movement was originally promoted as a way of moving away from formalism and bureaucratic centralized way of developing software, a large number of agile software development followers started playing down the role and importance of software architecture related processes, activities, artefacts, and

[1] http:\\www.agilemanifesto.org

roles in software development projects. For example, Thapparambil argues that "*no agile methods discuss Architecture in any length*." Many practitioners have been led to believe that agile methods consider architectural design to be an optional matter. The description of the agile methods provide very little discussion on common architectural design activity types [13] such as architectural analysis, architectural synthesis and architectural evaluation, as well as the artifact types [13] associated with these activities.

Scrum, one of the most popular project management approaches, claims to support architectural practices through frequent oral communication. According to Scrum, the architecture of one-project application can always be re-factored and repackaged to components for a higher level of reuse after the release to production to implement a walking skeleton, a small end-to-end functionality of the system, at the beginning of the project.

In the Incremental Re-architecture strategy of Crystal Clear [10], the team starts from the working architectural skeleton and incrementally evolves the architecture or infrastructure in stages and in parallel with the system functionality. Two architectural work products are almost certainly needed to be produced within a Crystal Clear project: system architecture and common domain model.

Agile proponents reason that "*Refactoring is the primary method to develop Architecture in the Agile world*". The Incremental Design practice of XP [9] claims that architecture can emerge in daily design. The emergent design means that architecture relies on looking for potentially poor architectural solutions in the implemented code and making a better architecture when needed in design cycles of hours and days. According to this approach, the architecture emerges from the system rather than being imposed by some direct structuring force.

These positions of agile advocates create an impression that "*Refactoring is the primary method to develop architecture in the Agile world*" [14]. Abrahamsson and colleagues have also concluded that the proponents of Agile methods believe that architecture should emerge from continuous small refactoring [3]. The advocates of architecture-centric approaches consider software architecture design as a very important activity to be conducted in the early stage of software development life cycle [15-17]. Software architecture researchers and practitioners are very sceptical about the claim that software architecture can emerge from continuous, small refactoring without upfront thinking and appropriate design efforts. Their scepticism has been reinforced by several stories where large scale software projects failed to succeed as a result of failure in paying sufficient attention to software architecture related issues [8].

The widespread adoption of agile approaches and importance of software architecture in a large scale software development projects have been promoting the importance of paying more attention to architectural aspects in agile approaches. There is a growing interest in identifying the mechanics and prerequisites of integrating agile and architectural approaches [2]. An increased number of efforts are focusing on devising appropriate ways of incorporating architecture-centric activities into Agile software development practices [3, 6]. For example, Ali Babar conducted a case study to identify and understand architectural practices and challenges of teams using Agile approaches [18]. Falessi and his colleagues surveyed the perceptions of software developers about the potential co-existence of Agile development and software architecture [19]. Nord and Tomayko [6] have proposed a few ways of integrating some of the SEI architecture-centric methods into the XP framework.

There are some other proposals for combining the strengths of the core elements of agile and architecture-centric approaches. For example, [20-21] combine the strengths of the core elements of the risk-driven, architecture-centric Rational Unified Process (RUP) and the XP [9] process. The combinations were enabled by the facts that RUP and XP share the cornerstone of the iterative, incremental and evolutionary development [22] and that most of the core elements of RUP and XP are complementary.

None of these efforts purport to explore the role and/nature of a particular architecture activity in agile approaches. That means there has been no significant empirical effort aimed at investigating one of the most significant point of debate between agile proponents and architecture followers: *can a satisfactory architecture emerge from continuous small refactoring* [4]?

Boehm and his colleagues have reported a set of guidelines for deciding how much agility and architecting are enough in a software development project [23]. Their set of guidelines was derived from an analysis model that considers three factors: project size, criticality, and volatility. Their effort highlights the importance of considering the contextual factors in deciding the level of architectural efforts required in projects using agile approaches.

We assert that the contextual factors identified by Boehm and his colleagues can also play important role in understand whether or not a satisfactory architecture can emerge from small, continuous refactoring. At the same time, we also assert that there can be many more contextual factors whose interplay can impact the emergence of "*satisfactory architecture*" through continuous refactoring. There has been no significant effort aimed at systematically identifying the factors that characterise the context in which a satisfactory architecture emerges or otherwise.

Hence, the main goal of this research is to empirically explore the perceptions and experiences of practitioners for identifying and building a taxonomy of the factors that can influence the emergence of a satisfactory architecture through continuous refactoring. In order to achieve the research goal, we carried out a large-scale study for empirically investigating two key aspects of the role of refactoring in achieving a satisfactory architecture when using agile approaches:

- Whether or not a satisfactory architecture can emerge from continuous small refactoring;
- What can be the contextual circumstances in which a satisfactory architecture emerges or not?

Both of these aspects were considered important for our research as we were not only interested in empirically finding out whether or not a satisfactory architecture emerges from continuous refactoring based on practitioners' experiences but also intended to discover the factors that can play a role in the emergence of a satisfactory architecture through refactoring.

III. Methodological Details

In this section, we describe and discuss the research methodology, data gathering approach, and data analysis method used for the reported research.

A. *Survey*

We decided to use survey research method to explore the perceptions and opinions of practitioners about their perceptions, observations, and experiences of the role of small continuous refactoring in the emergence of satisfactory architecture. A survey research method is considered suitable for gathering self-reported quantitative and qualitative data from a large number of respondents [24]. The objective of our study was to gather self-reported qualitative data. Our survey design was a cross-sectional, case control study. Survey research can use one or a combination of several data gathering techniques such as interviews and self-administered questionnaires [25]. We decided to use interviews as the data collection instrument as it appears to be more appropriate for gathering the detailed information required to find the answers to the questions that motivated our research.

We recruited the participants from different sources based on our pre-defined criteria for inviting the potential participants in our study; some of the criteria elements were industrial experiences of working in software development using agile and architecture approaches, experience of working as software architecture or requirements engineer, and represent different regions and domains. We sought the participants through the primary researcher's personal and professional networks as well as through a professional networking site, LinkedIn.

B. *Data Collection Approach*

We conducted semi-structured interviews using open-ended questions via email. Each email thread started from a brief description of the background and objective of the study, and a few questions for kicking off the required discussion. The email thread was continued through the primary researcher's enquiring and the participants' responses for more details until it was felt that an exhaustive amount of information had been gathered from each participant. Our email-based data gathering approach not only avoided the difficulty for the participant to find a sufficient chunk of free time to fit the researcher's schedule, but also provided participants with more time for preparing and sending reflective answers.

The interview questions focused on the participants' observations and experiences on architecture practices in Agile software development. In particular, we asked about their experiences and observations of cases where a satisfactory architecture emerged or did not emerge as expected from continuous small refactoring, and their reflections on why a satisfactory architecture emerged or did not emerge in those cases. Each email thread resulted in one to five pages (A4 size) document of communication log. The communication logs with all the participants form a rich set of qualitative data.

C. *Data Analysis*

For data analysis, we used mixed methods approach. The answers to the first question needed to be analysed by quantifying the qualitative data from the participants' responses based on their experiences and observations of cases where a satisfactory architecture emerged or did not emerge as expected. The quantification simply counted the number of participants who answered "Yes" or "No" to the two particular questions we asked related to the emergence of a satisfactory architecture through refactoring. We then summarized the quantified data using descriptive statistics.

Then we analyzed the data to find answer to the second aspect of our inquiry (i.e., underlying reasons for the observed phenomenon). For this analysis, we followed the techniques described in Grounded Theory (GT) [7]. Recently, other researchers have also used GT to study different aspects of Agile software development [26]. The data analysis procedure in GT involves three types of coding: open coding, axial coding, and selective coding. During the open coding, we analyzed the data line by line for creating and assigning codes to phrases, sentences, or paragraphs. A code is a phrase that summaries the key point concisely. During the axial coding, we went through the codes, and related them to each other, via a combination of inductive and deductive thinking [7]. The data analysis was not a linear process. We went through several iterations to refine, adjust the codes, and their relationships. The codes emerged directly from the data, which is in turn collected directly from the field. Thus, the resulting findings are grounded within the context of the real world experiences and observations.

For our finding, we included only those concepts that were mentioned by at least two participants. Once the concepts had made their way into the findings, we did not discriminate between them based on their frequencies; rather, we focused on the logical relationships among the concepts as recommended by GT. We report the findings in the following sections.

IV. Participants' Demographic Information

We interviewed 102 experienced software professionals across the world. Each of them has experience in both architecture design and Agile approaches. Due to limited space, we cannot provide the complete details of the participants' demographic details, hence, we just provide a summary of their backgrounds and work experiences.

The participants were located in 13 countries from 6 continents. The distribution across these continents is summarized in Figure 1. A majority of them were from North America. They had accumulated more than 1,600 years of work experience of software development (See Table I) in over 700 companies from nearly 40 different domains. All of them had experiences in both architecture design and Agile approaches. The experience they had in architecture, and Agile approaches was: 9 years and 7 years on average respectively; 2.5 years and 2 years minimum respectively; 20 years and 30 years maximum respectively. It is worth noting that a few of them reported that they had been using Agile approaches before the term "Agile" was coined.

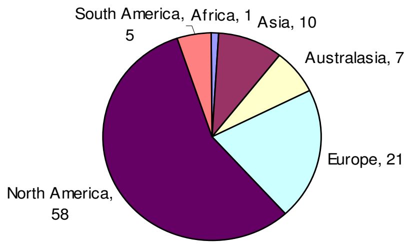

Fig. 1. Geographical distribution of participants

The Agile approaches that the participants had been using included Scrum [11], XP, Lean (Lean software development) [27], FDD (Feature Driven Development) [28], Kanban [29], and Crystal Clear [10]. The distribution over these approaches is summarized in Table II. Scrum and XP are the mostly widely used ones among the participants. It is worth noting that many of them used multiple approaches. The participants had worked for various organizations. On average, a participant had worked for 8 organizations at different stages of his/her career. The participants had worked for 737 different companies that ranged from very small (<10), small (10-49), medium (50-249), large (250-10,000), and to very large (>10,000). The ten top domains of the participants' companies are presented in Fig. 2; the domains include automotive, telecom, finance, and web-based socio-technical systems. In order to protect our participants' privacy, we refer to them by numbers P1 to P102 when presenting their quotations.

TABLE I. SUMMARY OF PARTICIPANTS' EXPERIENCE

	Minimum	Maximum	Average
Software Industry	6	37	16
Architecture[2]	2.5	20	9
Agile	2	30	7

TABLE II. AGILE APPROACHES USED BY THE PARTICIPANTS

Scrum	XP	Lean	FDD	Kanban	C. C.
82	68	10	6	4	2

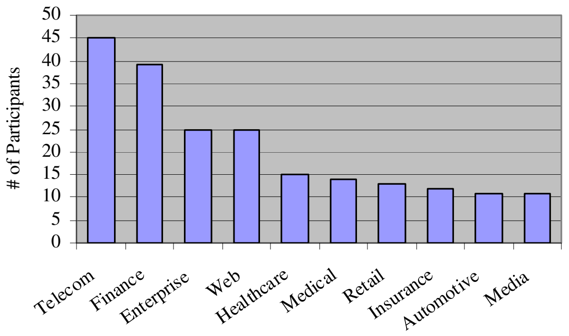

Fig. 2. Distribution of participants based on domains

[2] It was measured by the number of years that a participant has been taking the position of architect.

V. ON THE EMERGENCE OF ARCHITECTURE FROM CONTINUOUS REFACTORING

To gather evidence regarding whether a satisfactory architecture can emerge through continuous small refactoring, we particularly asked every participant two questions:

- Q1: Have you experienced or observed cases where a satisfactory architecture emerged from continuous small refactoring as expected?
- Q2: Have you experienced or observed cases where the team expected a satisfactory architecture to emerge from continuous small refactoring, but it did not?

Figure 3 shows the numbers of participants who said "YES" or "NO". A majority of the participants (68%, 69 out of 102) had experienced or observed cases where a satisfactory architecture emerged from continuous small refactoring. A slightly less than a third (32%, 33 out of 102) of the participants had not experienced or observed any such cases. It indicates that in many cases a satisfactory architecture can emerge from continuous small refactoring.

Does this mean that, in most cases, a team can go for implementation directly and let the architecture emerge? Participants' replies to Q2 do not support this claim. Figure 4 presents the participants' replies to Q2. A large majority of them (82%, 84 out of 102) had experienced the cases where a satisfactory architecture did not emerge from continuous small refactoring as expected. This result indicates that in many cases a satisfactory architecture cannot emerge from continuous small refactoring.

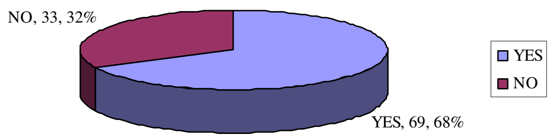

Fig. 3. Q1: Have you observed cases where a satisfactory architecture emerged as expected?

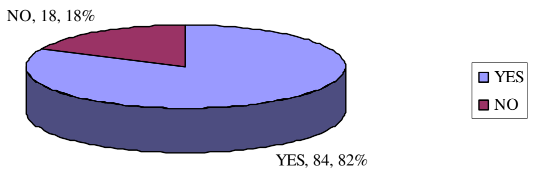

Fig. 4. Q2: Have you observed cases where a satisfactory architecture did not emerge as expected?

To gain more insights into the data, we cross-tabulated participants' answers to Q1 and Q2. Table III shows the result. The number in each cell represents the number of participants who gave the answers indicated by the column and row headings. We can see that a majority of the participants (59%, 60 out of 102) have experienced or observed both types of cases. Slightly less than a quarter of the participants (23%, 23 out of 102) have only experienced cases where a satisfactory architecture did not emerge as expected. Nearly one in ten participants (9%, 9 out of 102) has only experienced or observed cases where a satisfactory architecture emerged. A similar number of participants (9%, 9 out of 102) have not experienced or observed either type of cases. This is because they have not tried to follow the advice that architecture should emerge from continuous small refactoring. We will discuss these numbers in Section VII.

TABLE III. Q1 VS. Q2 CROSS TABULATION

	Q2.Yes	Q2.No
Q1.Yes	60 (59%)	9 (9%)
Q1.No	16 (23%)	9 (9%)

The descriptive statistical summary of the participants' answers to Q1 and Q2, as presented above, shows that a satisfactory architecture emerged from continuous small refactoring in many cases, but did not emerge in many cases as well. Thus, the answer for whether a satisfactory architecture can emerge, suggested by the data, is: It depends. But, what does it depend on? We present the findings gained from our qualitative data analysis in the next section.

VI. FACTORS THAT IMPACT THE EMERGING OF ARCHITECTURE THROUGH REFACTORING

The second key aspect of this study was to identify the contextual factors that may support or inhibit the emergence of a satisfactory architecture through continuous refactoring. For this part of the data, we applied the data analysis techniques from GT as previously stated. Our data analysis identified twenty contextual factors that perceived to have positive or negative impact on the emergence of architecture through refactoring. We systematically analysed the identified factors and place them into categories to form a framework of contextual factors that can help predict whether or not a satisfactory architecture would emerge through continuous refactoring. Figure 5 shows the framework that consists of factors: project, team, practices, and organization.

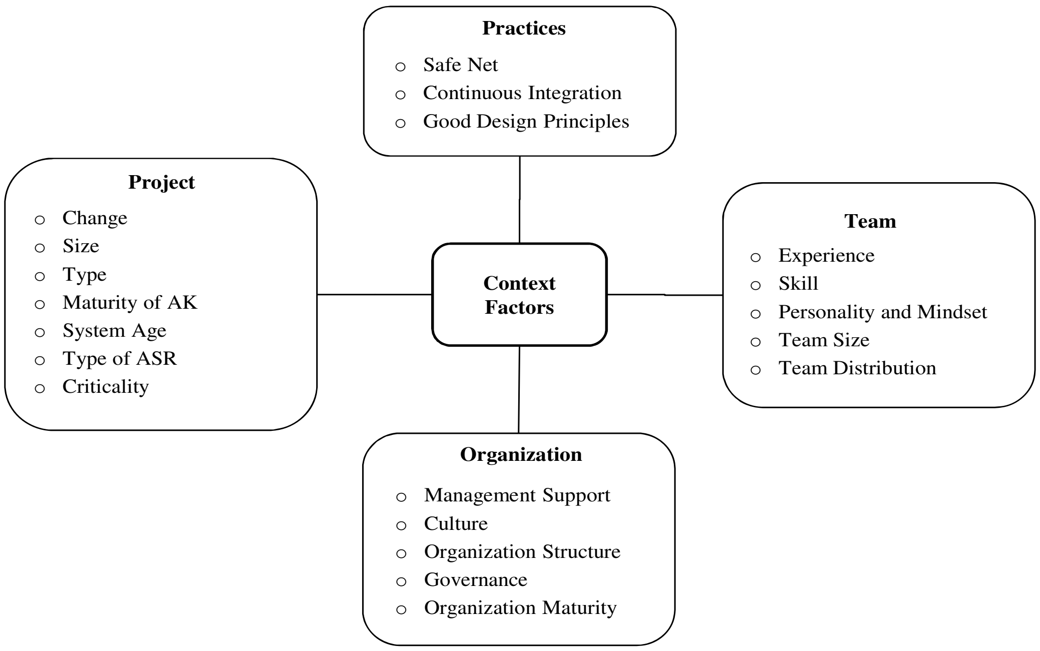

Fig. 5. A framework of factors that influence the emerging of a satisfactory architecture from continuous small refactoring

A. *Project*

This category of factors is related to different aspects of a project that can impact the emergence of architecture from refactoring. Our analysis revealed that around 35% of the participants indicated the factors that have been placed under this category. Many other studies have also mentioned that the implementation of agile approaches is customized based on a project's need. This category has 7 factors.

Rate of Change has impact on the emergence of architecture through continuous refactoring, as mentioned by many participants. If the rate of change in the requirements is quite high or a significant implementation has been done before getting to know the key non/functional requirements, it is quite unlikely that a satisfactory architecture emerges through small continuous refactoring: "*Many times string non-functional (business) requirements may appear after* the *software started to be built, and that may impact on the half-built solution you've got*", P41. On the other hand, if the rate of change is very low, continuous refactoring may become an unnecessary overhead rather than helping architecture to emerge: "*When unlike #1 [where rate of change is high]. I have seen where due to* the *constant changes and updates that the end goal either gets clouded, lost all together or is severely changed then what was first thought*", P92.

Size of A Project also plays an important role in emergence of architecture through refactoring. Like many sceptics of the claim about the emergence of architecture through refactoring, most of the participants reported that most of the time this happens for smaller projects. Interestingly, however, a few participants have experienced cases where a satisfactory architecture emerged for large projects: "*Larger projects--involving several groups-- are more prone to architectural issues, but some of this can be mitigated by focusing on loosely coupled interactions between the software components*", P3.

Type of Project is also an important consideration for this matter as for some types of projects, such as those that have only one feature and are algorithmically complex, and those that do not allow small releases, a satisfactory architecture is hard to emerge,*"sometimes you need a base with a set of minimum of functionality"* and this *"minimum"* can be *"a fairly large critical mass. Then it may not be possible to work in small steps",* P11.

Maturity of Architectural Knowledge (AK) means the amount of experience and knowledge of reusable architectural artefacts such as reference architectures, design patterns, and tactics. If a project team is mature in AK and its applications in different situations and contexts, it is likely that their knowledge and experience can lead to implicitly disciplined and mindful refactoring that takes into consideration of well-known design principles and patterns. Hence, such refactoring usually leads to the emergence of a satisfactory architecture, compared with a team consisting of architecturally immature people. With a matured AK, they are able to know where to start the architecture and have a relative clear vision on potential evolutionary paths of the architecture they started with. Without the existence of mature AK, a satisfactory architecture is unlike to emerge from continuous refactoring.

System Age also plays an important role in supporting the emergence of architecture through refactoring. Software systems designed based on contemporary design principles and technologies are likely to be more amenable to gain structural and behavioural integrity through refactoring. Generations old systems have the tendency of being monolithic and unnecessarily interdependent that can make it difficult (or impossible) for them to provide a coherent and conceptual integral architecture through refactoring. "*Failure [architecture did not emerge] is also more prevalent in cases of large pre-existing products which either had no discernable architecture to begin with or had their architecture erode away over* years *of maintenance and haphazard addition of new features by persons who didn't understand the pre-existing architecture or ignored it for one reason or another*", P85.

Type of ASRs (Architecturally Significant Requirements) and Their Criticality are known to be the key driver of architecture design activities. For refactoring, ASRs are also the key motivators. However, it all depends upon the nature of ASRs whether or not a highly coherent and integral architecture is achieved through refactoring. For example, design time ASRs such understandability may be achieved through continuous small refactoring by applying well known design principles and patterns. However, this may not be the case for another type of ASR such as security, which needs to be proactivity addressed at design stage. "*A good example is security, which in all of my years of experience, should be designed in (but can be implemented later). Especially in* terms *of cascading Web Services and such, [an] impedance mismatch between services and framework architecture is costly for rework*", P33. That means the criticality of achieving a particular ASR also plays an important role in a decision whether to do detailed design upfront or wait to get it fixed through refactoring. A project with critical ASRs hardly progresses in an expectation that a satisfactory architecture will emerge from refactoring.

B. Team

The category incorporates those factors that are directly or indirectly related to different dimensions of a project's team. Nearly half of the participants (48%, 49 out of 102) mentioned the team related factors having an impact on the emergence of architecture through continuous refactoring. "*You need to have a good Agile team that know how to go about this* process *[emerging of architecture] well. Otherwise this is a disaster and produces a lot of waste*", P99.

Experience and Skill of designing similar systems can also help support architecture through refactoring as mentioned by several participants. Researchers in other disciplines have also reported significant differences in understanding a design problem and devising solutions based on the amount of design experience [30]. That means significant experience with similar projects carried out using similar technologies in similar domains can enable designers to gain and maintain conceptual integrity of the software design through refactoring of the structure of the software being developed or evolved. "*An inexperienced developer may not have as much success since he/she may not know what architectural goals need to be achieved, which could result in absolutely messy unusable code*", P37. Skillset is another related factor mentioned by many respondents. The nature of a skill set and the amount of experience usually go together. That is why our analysis revealed that when participants indicated experience they were referring to experience of a particular skillset directly or indirectly. Lack of sufficient skills usually results in refactoring not making any noticeable contribution to have a sufficient architecture rather there may be risk of having the source code broken at various places: "*Those [teams] not having high skill in this area tend to simply make the codebase 'different', not better. Worst case they retard progress with continual rework*", P85.

Personalities and Mindset of team members is important to succeed in getting a satisfactory architecture through continuous refactoring.*"Change should be acknowledged as a part of the development process"* (P3), be *"willing to learn technology and try to adapt to them"* (P9), have *"passion"* (P9), and need *"dedication throughout the team (75% or more of its members, in my experience) to good quality design"* (P13). *"Pedantic developer personalities who are more focused on consistency of micro details rather than the overall readability, understandability, obviousness* *of the code"* (P28), *"Unwillingness from some team members to look outside of their own code (keep constant look at overall system's architecture)"* (P30), and *"personalities not inclined to work in groups"* (P55) can lead to failure.

Team Size and Distribution are also mentioned as factors. Team size is usually driven by project size, but it may not always be the case, e.g., a small project may use a large team with the expectation to finish the project faster. The participants of our study were of the view that a satisfactory architecture may be achieved through refactoring if the team size is relatively small. It was also revealed that a team's distribution nature may also impact as a satisfactory architecture can usually be achieved through refactoring in collocated teams. "*Proximity between co-workers (being able to listen and talk freely...no cubicles) [is essential]*", P24. "*2 teams* working *across geo (in different [time zones]) without an integrated approach has resulted in failure [architecture did not emerge]*", P65.

C. Practices

Since software development practices interact with each other, the participants of our study have also indicated several practices which can potentially support or inhibit the achievement of a satisfactory architecture through continuous refactoring. Due to limited space, we describe the three most frequently mentioned practices that can impact on the emergence of a satisfactory architecture.

Safe Net refers to automated testing with good coverage to form a safe net for refactoring, so that the team can have confidence that refactoring does not break a system. Many participants were of the view that continuous refactoring may fail to return a satisfactory architecture because of low coverage through automated testing. "*Your team must use constant automated unit testing and measure code coverage before practicing 'wide' refactoring. Code coverage in testing should be above 90% to give [the] team some kind of psychological confidence that changing the architecture will not break the system. Without this confidence team members will be resistant to change [the] architecture much and will rather focus on small local refactoring.*", P30. "*Without automated tests, architectural changes become more risky. They tend to take longer to implement. And so they tend to occur less often, and you don't see incremental movement toward a good architecture -- more typically you see incremental degradation of the architecture into chaos*", P13.

Continuous Integration is a key supporting practice for architecture to emerge from continuous small refactoring. *"Re-factoring without continuous integration tests to verify re-factoring hasn't broken, or re-broken the product is very, VERY dangerous […] Way before Agile and integrated tests I worked on a rule-based system [with more] than 10,000 rules. One team member would always re-factor something […]. He knew he was so good [that] his changes were 'correct'. He broke almost every build and by the time we found out he was on the road driving three hours away"*, P14.

Good Design Principles are expected to lead to high quality architecture that can ensure the achievement of desired ASRs – Agile approaches also support the use of good design principles, albeit implicitly, for example, refactoring is one way improving the internal structure of an application. Many of the participants mentioned the use of several principles: DRY (Don't Repeat Yourself) [31], SOLID (the Single responsibility principle, the Open closed principle, the Liskov substitution principle, the Interface segregation principle, and the Dependency inversion principle), and KISS (Keep It Simple and Straightforward). "*..all these practices complement each other..while eliminating the pitfalls that otherwise might be introduced by other practices. For example, unit test without refactoring will definitely introduce rigidity in architecture compounded with light up-front design could contribute to severe and rapid architectural deterioration thus compromise in quality and productivity*", P60.

D. Organization

Software development practices and their outcomes are expected to be influenced by organizational cultures and practices. The participants of our study also identified organizational factors that can support or inhibit the emergence of a satisfactory architecture through continuous refactoring. The organizational factors were identified by almost 33% of the participants.

Management support and commitment to agile and architecture is important. When lack of management support, a satisfactory architecture did not emerge. "*Of course when an architecture transformation is not a priority target and a team is not given enough time to work on the necessary changes then the new architecture will not emerge*", P5. "*You[r] team should have enough time to perform [a] 'wider' look at the system and change its architecture when local refactoring comes into conflict with overall architecture. Working under constant pressure from management to deliver releases, [the] team may not feel it's in the best interest to spend days improving architecture... even realizing it may significantly improve [the] product's maintenance and reliability. Management must be supportive on team's decision to refactor code and invest time, if necessary*", P30.

Culture can also play an important role in achieving architecture through refactoring. The cultural aspects mentioned by the participants include: good communication channels, encouragement for people to take ownership and commitment, open, and blame-free. If suitable culture is not there, a satisfactory architecture is unlikely to emerge. "*Team members must have good communication channels and discuss overall changes with each other all the time […], so everyone would know about system-wide changes […]. Bad communication between team members is another strong contributor to fail producing new architecture from small local changes*", P30. A participant referred to a failure case: "*They [the team] were used to work heads down in their cubicles for months without speaking to anybody. After that time they simply deposit hundreds of pages of useless diagrams and felt good about it. […] the sense of responsibility that comes with Agile is not there*", P24.

Organization Structure can also help or hinder in gaining a satisfactory architecture through refactoring. When describing the reasons for cases where a satisfactory architecture did not emerge, participants said: "*When the organizational structure prevents [emerging of architecture], for example, enterprise organizations often make it difficult to do continuous small refactoring.*" P29. "*Their organizational structure was based by role (architects, analysts and developers). They were not embracing the openness of the Agile approach*", P24.

TABLE IV. A CHARACTERIZATION OF CONTEXTS IN WHICH A SATISFACTORY ARCHITECTURE IS LIKELY TO EMERGE

	Factor	Success Condition
Project	*Change*	Medium to high rate of change
	Size	Small
	Type	Support small releases
	Maturity of AK	Mature Architecture Knowledge (AK)
	System Age	Green field
	Type of ASR	No demanding ASR that cannot be satisfied by refactoring
	Criticality	Low criticality
Team	*Experience*	Experienced
	Skill	Skilled
	Personality and Mindset	Willing to make change, learn, have passion, with dedication to good design
	Team Size	Small
	Distribution	Collocated
Practices	*Safe Net*	Automated testing with good coverage
	Continuous Integration	Continuous integration
	Good Design Principles	Applying good design principles such as DRY, SOLID, KISS
Organization	*Management*	Management support and commitment
	Culture	Good communication channels, encouraging for taking ownership and commitment, open, blame-free
	Structure	Embraces the openness of Agile approaches.
	Governance	Proper architecture governance
	Maturity	Certain Level of Maturity

Organizational Governance and Maturity are important factors in ensuring architecture can be achieved through refactoring. A participant said: "*Without any kind of governance, 5 different development teams implemented different code that impacted the performance of everyone on the platform*", P32. When answering the reasons for cases where a satisfactory architecture emerged, a participant noted: "*It is a success because there is good architecture governance"*, P76. Moreover, it is commonly known that sound architecture are designed upfront or achieved through refactoring in relative mature organizations. When explaining why a satisfactory architecture did not emerge as expected, P63 replied: "*CMMI level one task[s] are only accomplished by the heroic act of individuals.*"

E. *A Characterization of Contexts*

The findings from our study have resulted in a characterization of contexts in which a satisfactory architecture is likely to emerge from continuous small refactoring (see Table IV). For the projects in contexts that are different from those characterized in Table IV, relying on a satisfactory architecture to emerge instead of doing an adequate upfront architecture design is risky. For these projects, a proper upfront design by adopting architecture-centric practices is highly recommended.

Due to the qualitative and exploratory nature of this study, giving an accurate measure for each factor is beyond the scope of this study. Characterizing a factor accurately (e.g., what is the threshold for project size, what are the specific types of ASR, how mature an organization should be, how informal or formal the governance should be) would require a separate empirical program consisting of defining appropriate metrics and then validating them. Though the characterization given in Table IV could not provide accurate measurement for each of these factors, it provides useful information to assist practitioners in deciding appropriate architecture practices to be used in their particular projects, rather than following the advice that "*architecture should emerge from continuous small refactoring*" on its face value.

VII. DISCUSSION

The findings of this study has shown that like any other software development practices, the contextual factors play a significant role in supporting or inhibit the emergence of a satisfactory architecture through continuous refactoring. However, many instances have been reported when contextual aspects of a particular practice or belief may be ignored by practitioners [8]. Interestingly, our study has also unearth a reason for this phenomenon. Table III (see Section V) shows that people, who are only aware of cases where a satisfactory architecture emerged and have not observed cases where a satisfactory architecture did not emerge, do exist (9%, 9 out of 102 participants). Because they have only experienced or observed successful cases, they tend to have strong belief in their practices. Thus, they tend to say with strong conviction to other practitioners: "*it worked for me, so I do not see why it should not work for you*" [8].

The findings of our study can provide some important insights with regards to the importance and role of contextual factors that should be taken into consideration when devising strategies to deal with architectural aspects, otherwise, it can harmful. If other practitioners, who work in totally different contexts, listened and followed their generic advice, they may run into difficulties, or even cause project failures. Clarifying or even just recognizing the confusion caused by such context-regardless preaching of practices is not easy. This is evidenced by the many "*blind bigots, sometimes rabid bigots*" among followers of Agile advocates [8].

In such a situation, empirical evidence is particularly valuable. Thus, we suggest that more empirical studies should be conducted to establish an evidence-based understanding of the context factors and their impact on the effectiveness of agile practices in general and combining architecture and agile approaches. Such understanding is essential for practitioners to select and tailor their process to a particular project according to the specific contexts at hand. As described in Section II, Boehm et al. [23] and Abrahamsson et al. [3] have also indicated a set of factors that determine the needs for architecture-centric activities. Compared to Boehm et al. [23], we have identified all the three factors they considered, i.e., project size, criticality, and volatility (corresponding to change). We have also identified many more factors. This answers the question we asked in Section II about the guidance [23] they provided. The answer is: many more factors need to be considered in their guidance. So we suggest practitioners to carefully consider the extra factors we have identified when using their three-factor based guidance.

The context model presented by Kruchten [32] provides the most comprehensive list of factors. Our results have confirmed Kruchten's experience-based insights with empirical evidence. Our study has also identified several factors (e.g., experience and skill) that are not included in the Kruchten's model. Our study only focuses on a particular practice in Agile software development (i.e., emergence of architecture through continuous small refactoring), we also identified and highlighted several supporting practices as factors. So our study results corroborate and complement Kruchten's work.

VIII. THREATS TO VALIDITY

Given the qualitative nature of our research, the usual threats to validity are inconsistent. An investigator's bias is not considered a threat in GT, but a required attribute. The investigator is expected to select the participants, refine the questions, and develop the theory. When evaluating the validity of a qualitative research, the terms like quality and credibility are used. Quality concerns the question: are the findings useful [33]? Credibility concerns the question: are the findings trustworthy and do they reflect the participants' experiences with a phenomenon [33]? We used three criteria to evaluate the quality and credibility of this study.

Fit concerns with questions like, "*do the findings fit/resonate with the professionals for whom the research was intended?*" We kept all traces from each finding to the participants' replies throughout the data analysis. We can link a finding to the replies from the participants.

Applicability or Usefulness concerns with question such as "*do the findings offer new insights? Can they be used to develop policy or change practice?*" On average, each of the participants mentioned three out of the 20 factors we have identified. At maximum, an individual participant mentioned six factors. This indicates that the list of 20 factors as a whole is probably beyond an individual's knowledge. We believe that these 20 factors together with the characterization of contexts can offer new insights. The findings can also cause changes in practice. These findings can help practitioners to realize the importance of context factors, and help them to make informed decisions. Hence, they may decide to spend sufficient time and resources on architecture design in the contexts where a satisfactory architecture is unlikely to emerge.

Variation concerns with questions like, "*has variation been built into the findings?*" That means if a phenomenon is complex, the findings accurately represent that complexity. The study participants had diverse backgrounds as noted in the demographic information of the participants. Hence, the findings are expected to show that diversity.

The inherent limitation of studies like ours is that the results can only be explained in the specifically explored contexts. The identified factors are not exhaustive. They only represent those that have been experienced and observed by our participants. The selection of participants through social connections could potentially result in selection bias. We have carefully excluded persons who are likely to advocate one side of issue due to their vested interest. A very high number of variables that affect a real software engineering project make it difficult to conclusively identify the impact that one factor may have on a project.

IX. CONCLUSION

Despite continuously increasing popularity and adoption of Agile approaches, there is an increasing perplexity about the software architecture's role and importance in Agile software development [3]. One of the most fundamental points of the perplexity is the question: *Can a satisfactory architecture emerge from continuous small refactoring* [4]? Based on a large scale interview based empirical study, it can be concluded that different participants had different views about whether or not a satisfactory architecture emerges through continuous refactoring. Our study has identified twenty contextual factors that have been placed in four elements of framework: project, team, practices, and organization. We have further characterized the contexts in which a satisfactory architecture is likely to "*emerge*". These findings can be used by practitioners to make informed decisions on their architecture practices in Agile software development.

Particularly, we hope the empirical evidence reported in this study can help eliminate the commonly observed phenomenon [8]: some practitioners usually ignore architecture-centric activities with the justification that *"we are doing Agile, architecture should emerge from continuous small refactoring.*"

ACKNOWLEDGMENTS

The authors are thankful to all the participants. The first author also acknowledges the help of Bashar Nuseibeh, Klaas-Jan Stol, Soo Ling Lim, and Sajid Ibrahim Hashmi for improving an initial draft of this paper.

REFERENCES

[1] D. West and T. Grant, "Agile Development: Mainstream Adoption Has Changed Agility - Trends In Real-World Adoption Of Agile Methods," Forrester Research, Inc.2010.

[2] M. A. Babar, "Making Software Architecture and Agile Approaches Work Together: Foundations and Approaches," in *Aigle Software Architecture: Aligning Agile Processes and Software Architectures*, M. A. Babar, Brown, A. W., Mistrik, I., , Ed., ed: Morgan Kaufmann, 2014, pp. 1-22.

[3] P. Abrahamsson*, et al.*, "Agility and Architecture: Can They Coexist?," *IEEE Software,* vol. 27, pp. 16-22, 2010.

[4] H. Erdogmus, "Architecture Meets Agility," *IEEE Software,* vol. 26, pp. 2-4, 2009.

[5] L. Chen*, et al.*, "Characterizing Architecturally Significant Requirements," *Software, IEEE,* vol. 30, pp. 38-45, 2013.

[6] R. L. Nord and J. E. Tomayko, "Software architecture-centric methods and agile development," *Software, IEEE,* vol. 23, pp. 47-53, 2006.

[7] J. Corbin and A. Strauss, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, 3 ed.: Sage Publications, 2007.

[8] P. Kruchten, "Voyage in the Agile Memeplex," *ACM Queue,* vol. 5, pp. 38-44, 2007.

[9] K. Beck, *Extreme Programming Explained: Embrace Change*: Addison Wesley Longman, Inc., Reading, MA. USA, 2000.

[10] A. Cockburn, *Crystal Clear: A Human-Powered Methodology for Small Teams*: Addison-Wesley, 2004.

[11] K. Schwaber, *Agile Project Management with Scrum*: Microsoft Press, 2004.

[12] D. Turk*, et al.*, "Assumptions underlying agile software-development processes," *Journal of Database Management,* vol. 16, pp. 62-87, 2005.

[13] C. Hofmeister*, et al.*, "A general model of software architecture design derived from five industrial approaches," *Journal of System and Software,* vol. 80, pp. 106-126, 2007.

[14] P. Thapparambil. (2005) Agile architecture: pattern or oxymoron? *Agile Times*. 43-48.

[15] C. Hofmeister*, et al.*, *Applied Software Architecture*: Addison-Wesley, 1999.

[16] L. Bass*, et al.*, *Software Architecture in Practice*, 2nd ed.: Addison-Wesley, 2003.

[17] P. Kruchten, *The Rational Unified Process: An Introduction*: Addison-Wesley, 2003.

[18] M. A. Babar, "An exploratory study of architectural practices and challenges in using agile software development approaches," in *Joint Working IEEE/IFIP Conf. on Software Architecture & European Conf. on Software Architecture*, 2009, pp. 81-90.

[19] D. Falessi*, et al.*, "Peaceful Coexistence: Agile Developer Perspectives on Software Architecture," *IEEE Software,* vol. 27, pp. 23-25, 2010.

[20] S. W. Ambler, *Agile modeling: effective practices for eXreme Programming and the Unified Process*. New York: John Wiley & Sons, Inc., 2002.

[21] IBM. (2004, *RUP for Extreme Programming (XP) Plug-Ins*. Available: http://www-128.ibm.com/developerworks/rational/library/4156.html

[22] C. Larman, *Agile and iterative development: a manager's guide*. Boston, MA: Addison Wesley Professional, 2003.

[23] B. Boehm*, et al.*, "Architected Agile Solutions for Software-Reliant Systems," in *Agile Software Development: Current Research and Future Directions* T. Dingsøyr*, et al.*, Eds., 1 ed: Springer, 2010, pp. 165-184.

[24] B. Kitchenham and S. L. Pfleeger, "Principles of Survey Research, Parts 1 to 6," *Software Engineering Notes,* 2001-2002.

[25] T. C. Lethbridge, "Studying Software Engineers: Data Collection Techniques for Software Field Studies," *Empirical Software Engineering,* vol. 10, pp. 311-341, 2005.

[26] R. Hoda*, et al.*, "Organizing self-organizing teams," presented at the Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, Cape Town, South Africa, 2010.

[27] M. Poppendieck and T. Poppendieck, *Lean Software Development: An Agile Toolkit*: Addison-Wesley, 2003.

[28] S. R. Palmer and J. M. Felsing, *A Practical Guide to Feature-Driven Development*: Prentice Hall, 2002.

[29] D. J. Anderson, *Kanban*: Blue Hole Press, 2010.

[30] S. Ahmed*, et al.*, "Understanding the Difference Between How Novice and Experienced Designers Approach Design Tasks," *Journal of Research in Engineering Design,* vol. 14, pp. 1-11, 2003.

[31] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*: Addison-Wesley Professional, 1999.

[32] P. Kruchten, "Contextualizing agile software development," *Journal of Software: Evolution and Process,* vol. 25, pp. 351-361, 2013.

[33] J. W. Creswell, *Qualitative Inquiry and Research Design: Choosing among Five Approaches*, 2 ed.: Sage Publications, 2006.