A Hybrid Software Architecture Evaluation Method for FDD – An Agile Process Model

Abstract-The software development industry suffers from the delay in project completion time due to heavy documentation requirements of traditional process models. To overcome these delays, agile process models are getting a wide acceptance and popularity in the industry. The beauty of these models is light weight documentation and heavy intercommunications. Due to an emphasis of these models on rapid development, there is an ever increasing need of architecture evaluation. A single software architecture evaluation method (SAEM) capable of preserving the agility does not exist at the moment. In this paper, we have proposed a hybrid SAEM for feature driven development (FDD) agile process model. The proposed SAEM is hybrid of quality attribute workshop (QAW), architecture tradeoff analysis method (ATAM) and active review for intermediate designs (ARID).

I. INTRODUCTION

Software development process models can broadly be divided into two categories; traditional models and agile models. Traditional models are waterfall and its extensions, prototyping, rapid application development (RAD), spiral, unified process (UP) and incremental. While, agile models are extreme programming (XP), scrum, dynamic systems development methods (DSDM), feature driven development (FDD), adaptive software development (ASD) and crystal.

The prime emphasis of traditional models is on heavy documentation of software development phases, which introduces delays in overall development of the software. Moreover, these models do not support frequent communication between different stakeholders and software developers after requirement gathering phase of software development life cycle (SDLC) due to which the modifications in the deliverables are not encouraged.

On contrary, agile models being lightweight processes focus on less documentation to reduce the delays which was a major drawback of the traditional models. Another feature distinct to agile models is strong and frequent communication between stakeholders and software developers throughout the software developing process. Owing to this feature, provisions may be made for modifications in requirements during development. Agile models provide fast software development environment by overlapping some of traditional software development phases [1].

As software demand is increasing day by day there is a need to increase the pace at which software is developed, for which agile models provide a very good solution. This fact has enhanced the popularity of agile models [2]. It is necessary to devise a proper software architecture evaluation method (SAEM) [3] such that it does not affect the agility but still is capable of ensuring the validity [4] of proposed software development architecture. No single existing SAEM works properly for agile process model due to SAEM’s requirements of heavy documentation. Either an SAEM is to be modified [5] for a particular agile process model or a hybrid SAEM is to be developed which is the focus of research presented in this paper.

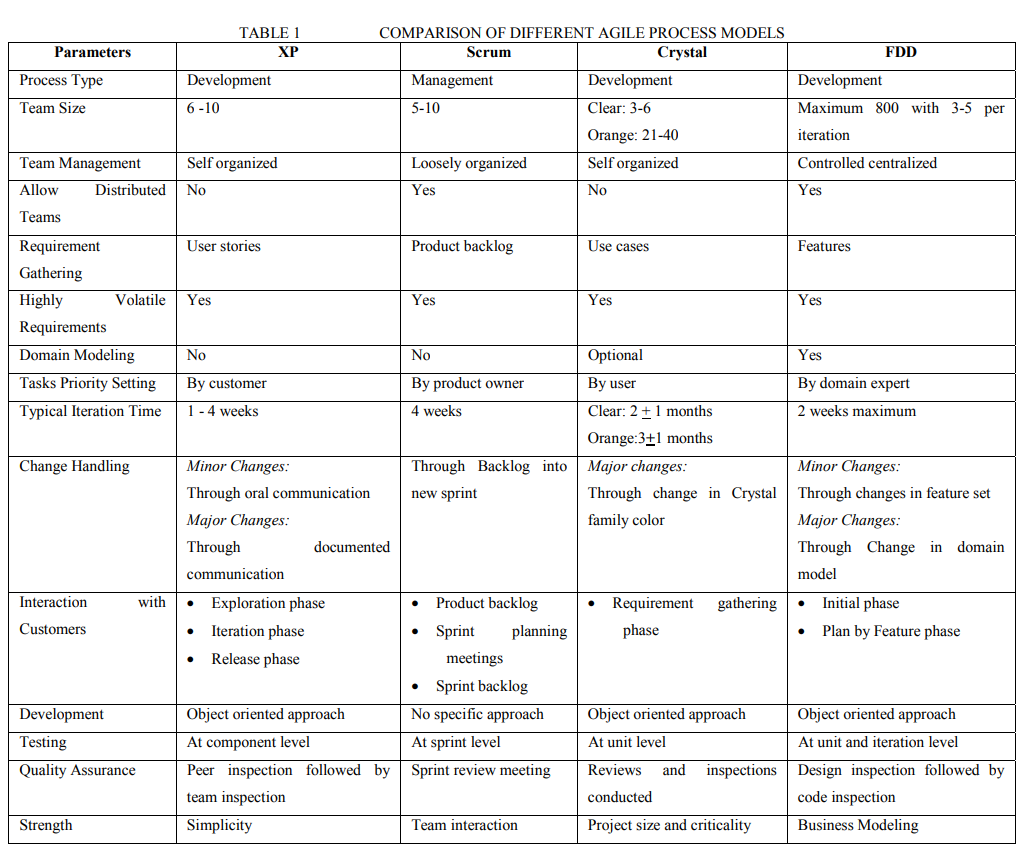
In this paper, we have proposed a hybrid SAEM for FDD process model. The proposed SAEM is hybrid of quality attribute workshop (QAW), architecture tradeoff analysis method (ATAM) and active review for intermediate design (ARID). The rest of the paper is organized as follows:

A comparison of agile process models is given in section 2 while that of SAEM is provided in section 3. Section 4 is dedicated to the proposed hybrid SAEM for FDD agile model. The discussion is summarized in section 5.

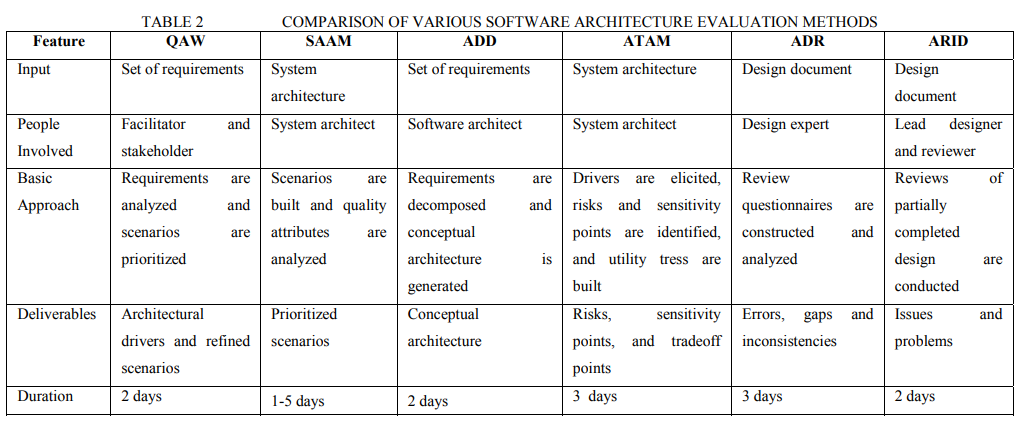
II. AGILE PROCESS MODELS – A COMPARISON

With the increased development and popularity of agile process models, it is necessary to evaluate the correctness and suitability of these models for software development. Taromirad & Ramsin [6] proposed various evaluation criteria and a framework to serve the purpose. Zuser et. al. [7] investigated the suitability of agile models as compared to that of traditional models in terms of software quality. Another study [8] determines the degree of agility for different agile process models. Various comparative analyses [9, 10] have already been carried out to explore the strengths and weaknesses of different agile process models, but those are limited to a fewer comparison parameters.

Our comparison of various agile process models shows that crystal clear is best suited for the projects where a minimum team size is required however, it demands more iteration time. XP outperforms all other models due to its simplicity. Scrum supports a heavy team interaction to make sure rapid change handling. FDD is most suitable for the projects with large team size and low iteration time. Moreover, FDD is very effective in business modeling of the projects. The discussion is summarized on a comprehensive comparison as given in Table 1.



III. SAEM – A COMPARISON Before we proceed with the description of our hybrid SAEM, it is very important to have a thorough and in-depth knowledge of existing SAEMs [11]. In this section, we have presented an exhaustive comparison of the existing SAEMs which is more comprehensive as compared to that of previous comparisons [12, 13, 14], as our comparison is not limited to scenario based SAEMs only (see Table 2).



The comparison shows that QAW and SAAM works only upto the analysis phase while, ADD and ATAM can handle design phase, as well. ADR and ARID are typically used for the reviewing purposes. Moreover, ADD is not a scenario based SAEM.

IV. A Hybrid SAEM for FDD

FDD consists of five major phases with each phase having a set of related activities (see Table 3). A brief description of these phases is given in the sections A through E. Section F details the integration of architecture evaluation activities with FDD.

A. Develop an Overall Model

The major emphasis of this phase is on requirement gathering and analysis. The requirements mainly include the functional requirements. Team building, and functional requirement documents are the major output of this phase.

B. Build a Feature List

The core of FDD process model comprises of the extraction of features on the basis of which the software development plan is prepared. This phase is dedicated to the building of a list consisting of these features.

C. Plan by Feature

The main emphasis of this phase is on tasks assignments to relevant team members. This phase also includes project scheduling against features.

D. Design by Feature

This phase concentrates the detailed design of functional requirements of the project. The design built in phases 1 and 2 is refined and finalized. The phase also includes the formalization of design specifications in the form of classes.

E. Build by Feature

The main orientation of this phase is the implementation of design specifications produced in the previous phase. Testing is also conducted in this phase.

F. Integration of Architecture Evaluation Activities with FDD - A Hybrid Approach

In this section, we have proposed a hybrid architecture evaluation approach for FDD. In FDD, architecture is developed in phase 1 and 2. Our approach is based on QAW, ATAM and ARID.

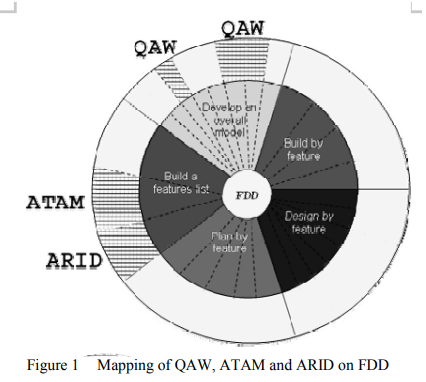
For phase 1 of FDD, we have investigated that functional as well as non-functional requirements gathering activities should be executed in parallel to ensure the development of proper architecture without affecting the agility. For that, QAW is a very good choice as the major concentration of this SAEM is on determining the quality attributes which establish the non-functional requirements of the project.

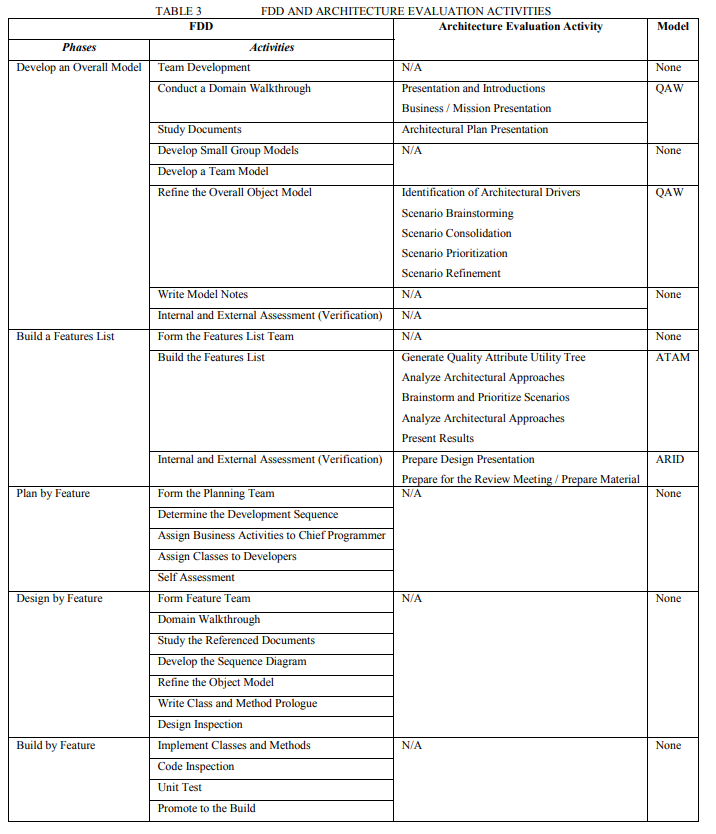
For phase 2 of FDD, there are two sub activities which need architecture evaluation.

While building the features list; utility trees, sensitivity points and tradeoffs should also be determined to develop a proper architecture. Utility trees, sensitivity points and tradeoffs are the inherent features of ATAM [15,16]. We have proposed to execute ATAM during this sub activity for obvious reasons.

For the assessments (verifications) ARID is to be executed as it is primarily developed for review activities.

The above proposed approach is illustrated in Figure 1.





V. SUMMARY

In this paper, we have proposed a hybrid SAEM for FDD so that architecture evaluation activities can be performed and yet the agility is not compromised. Our hybrid model is composed of three SAEMs i.e.; QAW, ATAM and ARID. Our approach can be extended for other agile process models, as well.

Adopting Logical Architectures within Agile Projects

Abstract— Agile Software Development (ASD) is a movement that emerged in the late 1990’s and since then has been increasingly adopted by organizations. They are successful in projects characterized by rapid changes, small teams and small cycles. Implementing the same agile philosophy but with an increase of complexity, i.e., large-scale agile (LSA) development, are sometimes criticized of not being as effective as typical ASD, requiring additional concerns and the need to address new research challenges. This PhD research proposes an approach for embedding architecture design in agile development, first in early analysis and then during the iterations and by properly providing the mechanisms to deal with the changes in requirements. The main result is a method for adopting logical architectures, Four-StepRule-Set (4SRS), properly adapted to ASD contexts, to be used as complementary approaches to an ASD lifecycle, so that ASD teams can use that information as input for delivering the working software.

I. INTRODUCTION

Agile methodologies are based in small development cycles and in continuous communication with customers with low needs on modeling formalism for requirements elicitation and documentation. However, there are projects whose context requires formal modeling and documentation of requirements, in order to raise and manage critical issues from the very beginning of the project. Agile implementation teams may experience difficulties in understanding key information from requirements artifacts (e.g., Unified Modeling Language – UML [1]) because the gathered information typically is not yet ready to be passed on to the implementation teams [2], where requirements are based in User Stories rather than models.

The use of these models is argued when projects have well-stated requirements at the very beginning, thus performing the so called “Big Design Up Front” (BDUF). On the other hand, practitioners of agile methods generally see architecture-centric methods as heavy documenting effort (misaligned with one of the eventually effort that is afterwards disregarded if requirements meanwhile change and resulting in so called “You Aren’t Going to Need It” (YAGNI) elements, and that the architecture should emerge gradually sprint after sprint, as a result of successive small refactoring [3]. On the other hand, [4] present a study where agile developers perceive software architecture as relevant in aspects such as communication among team members, inputs to subsequent design decisions, documenting design assumptions, and evaluating alternatives.

As in any kind of software projects, good software architecture is critically important since it is the key framework for all technical decisions [5]. In an agile context, [6] argue that architecture are used to provide the requirements, both functional and technical. Moreover, when developing large-scale applications, many have reported that agile methods must be adapted to include more kinds of architectural information [7].

This PhD’s research question is: “How to adopt logical architectures in agile large-scale projects?”. This PhD thesis will propose the inclusion of logical architecture design in small and iterative development cycles that occur in Scrum [8], an agile development process. The logical architecture design is supported by some methods and approaches. The Four-Step-Rule-Set (4SRS) method [9] is object of study in this PhD thesis. This method supports the functional decomposition of requirements (using UML use cases) and, based on them, deriving a logical architecture composed by UML components that relate to each functionality. Besides assuring the alignment between the requirements and the proposed architecture, the method also allows to identify missed requirements. Additionally, the method is scientifically validated (citations are not included due to paper size limits) for deriving logical architectures in complex projects and in contexts with difficulties in eliciting requirements, allowing the derivation of logical architectures at large scale and with distributed complexity with accurate specifications. For these reasons, the 4SRS method is fully capable of being included in this PhD thesis. The method requires an adaption in order to ease its adoption on agile projects, because it is structured to be used in BDUF contexts. Thus, the 4SRS method will base its incremental innovation by iterative development in small steps. This paper is structured as follows: section II presents results from the literature review in software architectures and ASD, coexistence of both approaches and LSA; section III presents the thesis’ research objectives and method; section IV preliminary results; section V future work; and in section VI the conclusions.

II. STATE OF THE ART

A. Software Architectures and ASD

The concept of software architecture is defined as the “fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution” [10]. Software architectures are useful artifacts for development teams, especially for enterprise integration and interoperability, which gave origin to a plethora of frameworks and references [11]. The concept of software architecture is considered as a distinct discipline, however still tied closely to other disciplines and communities, such as software design (in general), software reuse, systems engineering and system architecture, enterprise architecture, reverse engineering, requirements engineering, and quality [12].

Agile software development (ASD) methodologies are characterized by iterative and incremental development, customer collaboration, frequent delivery, light and short development cycle, quick adaptation to rapidly changing business requirements and self-organizing teams [13]. Most popular ASD frameworks are eXtreme Programming (XP), Scrum, Kanban, Agile Unified Process (AUP), Crystal Methodologies, Dynamic Systems Development Model (DSDM), and Feature-Driven Development (FDD).

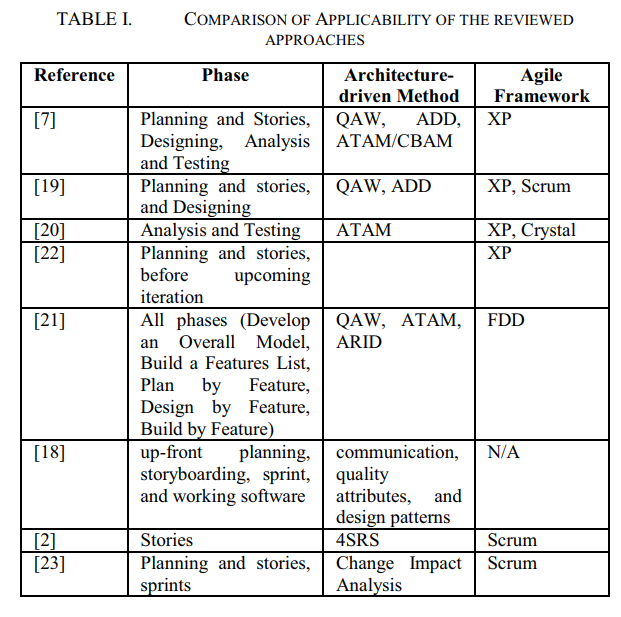
In software projects, agile methodologies are based in small development cycles and in continuous communication with customers with low needs on modeling formalism for requirements elicitation and documentation. However, the most perceived problem with agile development processes is whether these methods scale to larger software teams [14]. If large teams are to produce lots of software functionality quickly, the agile methods must scale to meet the task [15]. It is commonly stated using architecture and ASD frameworks simultaneous within the same project is inadequate [16], mainly because proponents of agile approaches usually see little value in BDUF, namely because of resulting YAGNI features [3]. The Agile Manifesto values “Working software over comprehensive documentation”, especially when future requirements are largely unpredictable [17]. Some kinds of architectural information in agile methods are evidenced in the zero-feature release, the architectural spike, and agile practices that recognize the architect role [7].

B. Potential Coexistance of Both Approaches

Madison’s approach [18], called agile architecture, advocates the coexistence of agile and architecture as complementary approaches by appropriately applying suitable combinations of architectural functions (such as communication, quality attributes, and design patterns) and architectural skills in the development life cycle (up-front planning, storyboarding, sprint, and working software). Literature shows other research works, namely presenting framework proposals that explore the relationship and synergies between architecture-centric design and analysis methods from the Software Engineering Institute (SEI) that regard architecture design and review within agile frameworks. Jeon et al. [19] propose a customized Quality Attribute Workshop (QAW) and Attribute-Driven Design (ADD) is used in Scrum projects, [20] uses Architecture Trade-off Analysis Method (ATAM) to validate architectures in a Crystal project, [7] propose an approach based in these previous mentioned works to use QAW, ADD, ATAM and Cost-Benefit Analysis Method (CBAM) within XP activities, and [21] propose a hybrid of QAW, ATAM and Active Review for Intermediate Designs (ARID) method for FDD.

Among the works that do not rely in architecture methods from SEI, Costa et al. propose a technique for deriving software requirements compliant with Scrum teams from architectures that resulted from the 4SRS method execution [2].

The reviewed literature encompass applicability of software architecture methods within any of the phases in an agile project, from sprint planning, user stories, backlogs, development, testing, etc. Additionally, the aforementioned approaches relate to a diversity of agile methodologies, from XP to Crystal, FDD or Scrum (as shown in TABLE I.



The aforementioned approaches showed that there is an opportunity to improve software development while maintaining a balance between agility and the architectural approach. Therefore, several approaches to integrate and embed software architecture and agile methods are proposed. These works all have in common the fact that architecture methods must perform in parallel with common agile methodologies, and the architecture itself must possess agility and flexibility enough to rapidly respond to changes.

After analyzing the approaches, they do not provide per si an integrated process for using architectures in LSA, but each one includes techniques that are valuable inputs to be incorporated in our proposed approach. These approaches do not include thorough requirements specification and a logical architecture able to be used as basis for the development like 4SRS does, that becomes necessary in large-scale projects as argued, for instance, by [7]. Additionally, the initial backlog should include both functional and quality (non-functional) requirements (typically quality ones only emerge during development), where 4SRS supports their identification by using the Model/View/Controller (MVC) pattern. It is expected that the proposed approach uses these “strengths” of 4SRS and adapt them to agile context, but also to include concerns that the presented approaches provide (change impact analysis, architecture review and assessment, and others).

C. Large-scale Agile Development

Although recent, agile methodologies have proven successful in projects characterized by rapid changes, small teams and development cycles. When developing large-scale applications, many have reported that agile methods must be adapted to include architectural information [7]. Alistair Cockburn recognizes that different project characteristics need different methods [24]. The term ‘large-scale agile development’ (LSA) has been used to describe agile development in everything from large teams to large multi-team projects to making use of agile development principles in a whole organization [25], [26]. Main research challenges within LSA may be [25]: “Organization of large development efforts”, “Interteam coordination” and “Release planning and architecture”.

This PhD thesis proposes an approach where these challenges can all be adequately faced, by allowing to depict information from the architecture that may be used within agile product line engineering, coordination of work between teams in LSA development, coordination and prioritization of functional and nonfunctional requirements, continuous delivery, or minimizing technical debt. LSA must also be seen as an opportunity to explicit and improve the architect role as well as agile practices.

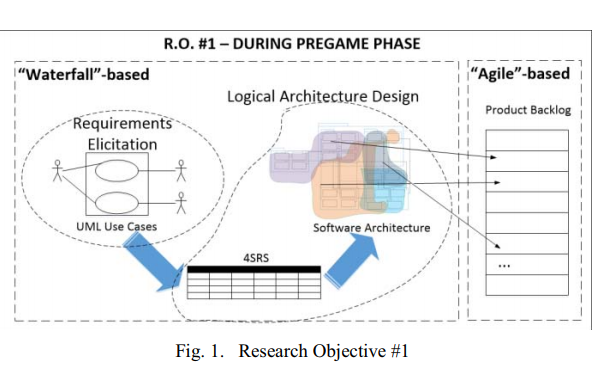
III.RESEARCH OBJECTIVES AND METHODOLOGICAL APPROACH

A. Research Objectives

Architectural design can be improved in agile methods by: (1) agile architectural modelling by using an incremental, customer-involved process; and (2) an initial vision of the system including initial design is created during the first iteration of the development [27]; (3) there are several iterations for designing the system, thus a more detailed design followed further on is created [27]; and (4) continuous iterative design where design is embedded into agile development and architectural artifacts are updated regularly [27]. This topic must be addressed in the very beginning of the agile process, and afterwards the research must aim at providing the architecture with the required flexibility during the iteration-cycles of the process. Thus, these two project phases require distinct approaches, each addressed by one objective:

O1: To develop an approach capable of deriving logical architectures in order to establish the initial requirements that are passed on to agile development teams.

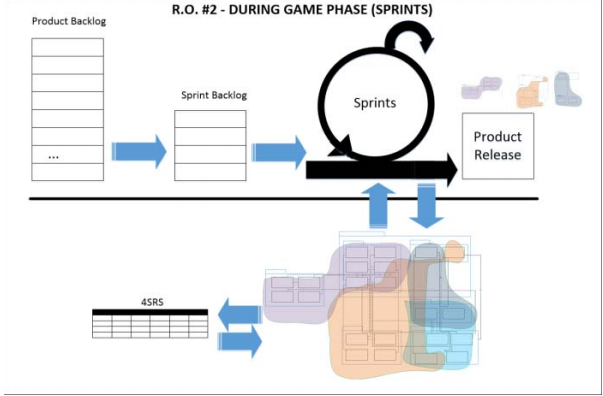
Using logical architectures for establishing initial requirements allows to combine requirements from backlogs (that focus only on functional features) with the quality attributes of the software [19], as depicted in Fig. 1. The research aims at including some upfront design in the set-up phase of the project (by some we do not mean BDUF, rather using existing research related to the sufficient amount of information for architecture design [28]), and to use the architecture as input (back to requirements again) to build almost the totality of the Product Backlog.



The 4SRS method allows to derive logical architectures aligned with the corresponding and previously elicited and modeled user requirements in large-scale projects, but demands high quantity of information (use cases functionally decomposed and textual descriptions), which is often time consuming and, in every way, misaligned with the general paradigm adopted by agile methodologies. In this PhD thesis, the 4SRS method [9] will be extended for adopting logical architectures in a typical ASD. The first approach was developed in [2], where logical architectural elements were mapped to User Stories statements (in the “As a … I want to … in order to…” format). In this PhD thesis, the steps of the 4SRS method will be modified, having in mind that the Product Backlog requires items relating to functional and quality stories, defects, technical work and knowledge acquisition.

O2: To adopt flexibility and agility mechanisms in the refinement of logical architectures throughout the iterations of ASD teams.

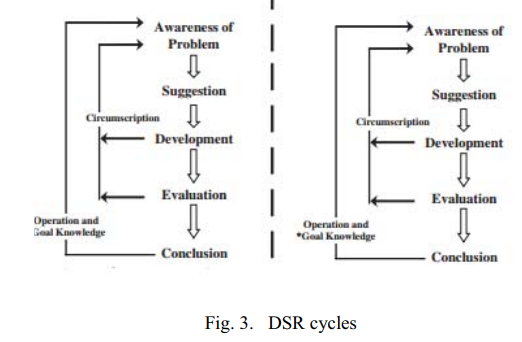
The adoption of a logical architecture implies that it is used as a complementary approach to agile in the development life cycle [18]. Considering the change-driven environment where ASD is performed, the resulting artifact from the previous objective must imperatively be able to respond to those changes without losing information and not being subjected to unnecessary refactoring efforts.



This objective relies in adding the 4SRS method with mechanisms to refine the requirements from pregame phase (addressed in O1) but also to respond to changes during the ASD iterations, as depicted in Fig. 2. This research will aim in using the 4SRS to trace the changes to the stated requirements and deriving the architecture according to the changes. Then, the impact of the changes are assessed (using approaches like [7], [21], [23] as input) before being implemented.

B. Research Method

The process to conduct this PhD thesis is Kuechler’s and Vaishnavi’s approach for Design Science Research (DSR) [29], in this case, structured as two sub-processes that relate to two DSR cycles. The research objectives are addressed sequentially, i.e., objective O1 is fulfilled after performing as many iterations as required to conduct an entire DSR cycle and, posteriorly, the resulting knowledge is used as input for conducting another set of iterations of a DSR cycle to fulfill objective O2. This sequence is presented in Fig. 3.



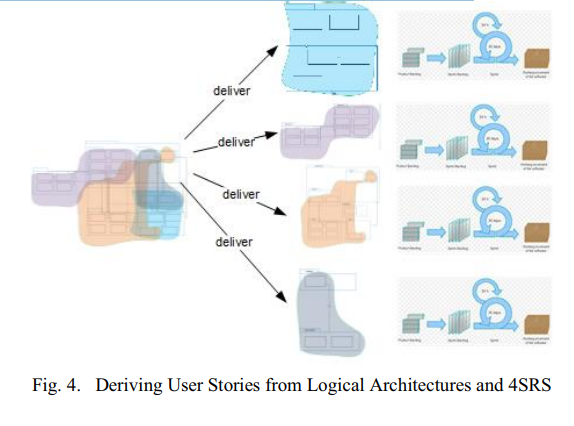
The performed research strategies throughout the research process are similar in both DSR cycles, namely: (1) a combined task of literature review and knowledge from past projects (see section III.A) within the Awareness phase; and (2) case study, as proposed by [30], aiming to experiment and evaluate of the method applicability (using the Context, Setting the Hypothesis, Planning, Validating and Analyzing the Results like [30]). Within the project, we will focus in developing an artifact, namely a process for using 4SRS and architectures within ASD. The process’ applicability will be measured by its resulting architecture and its input for establishing the Product Backlog. Its validation is assessed upon the process’s method, inputs and outputs successful use in ASD.

III. PAST WORK AND PRELIMINARY RESULTS

A. 4SRS together with Scrum

Before starting the PhD thesis, during a R&D project involving academia and software companies, namely the ISOFIN project [31], a technique for deriving User Stories from logical architectures that resulted from the 4SRS method execution (the same method that will be applied in this PhD thesis) is proposed in [2] and depicted in Fig. 4. The purpose was not to build the entire Product Backlog, but rather to develop a technique for deriving the User Stories from information provided by previous 4SRS method execution.

Moreover, the need to divide work by distributed [32] teams in the ISOFIN project resulted in the need to identify some contact points, where there is a need for synchronization within distributed Scrums and effort dependencies.



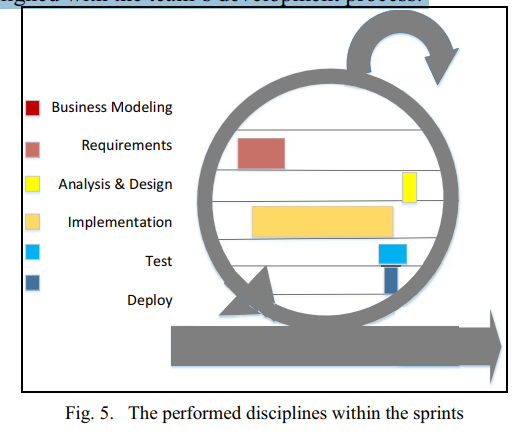
B. When Models are Complementary to Scrum

During the first year of the PhD thesis, a research and development (R&D) software project called iFloW [33], resulted in an approach where Scrum was adapted by a newly-formed team to develop a software system. This R&D project was performed in a collaboration between industry and university, namely University of Minho and Bosch Car Multimedia. Project management commonly follow plan-centric processes. However, conducting such projects in an agile framework is expected to decrease the risks from unstable and emerging requirements from industry side. In this type of collaborative R&D projects, it is often advantageous to include some artifacts, like UML models, which are not commonly used in agile processes like Scrum.

The R&D context required domain research tasks in an initialization phase, as well as technological-related research and third-party collaboration during the implementation phase. This phase was performed in form of Sprints (small cross-functional development cycles and are used in ASD frameworks like XP and Scrum). Software development projects with important integration and interoperability issues require additional concerns when compared with typical ASD, since implementation requires prior studies related to the technologies involved (except if the team has solid experience with those technologies) and collaboration with third-party entities. The process is composed of three phases: Initialization, Implementation, and Deployment.

The process is described in [34]. Within the initialization phase, the objective was to develop a product backlog that, due to the perceived complexity of the project, was delivered together with widely accepted forms of requirements documentation. For implementing each use case, the team performed tasks involving several software engineering disciplines. In this paper, we use the terminology from RUP’s disciplines (only for demonstration purposes) to depicts the type of effort involved within the Sprints. Occasionally, the team performed spikes (originally defined within XP), a technique used for activities such as research, innovation, design, investigation and prototyping. With spikes, one can properly estimate the development effort associated with a requirement or even to better understand a requirement. The use of spikes in the iFloW project justifies the inclusion of the Requirements discipline in the each Sprint, as shown in Fig. 5. These spikes were, in their majority, originated from middleware-based use cases. Within the remaining use cases, the Requirements discipline was not required. Thus, in comparison with the disciplines included in Fig. 5, the Sprint performed the remaining disciplines like illustrated with exception of Requirements. In all sprints, the need for updates to the logical architecture was assessed (within the Analysis & Design). Afterwards, the typical disciplines were carried out within the sprints: Implementation, Testing and Deploy. The Business Modeling discipline is included in the figure only for maintaining the intended comparison with RUP disciplines, since the discipline is not performed during the Sprints.

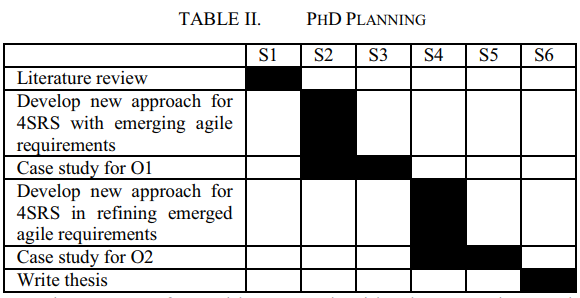
The implementation of some middleware-related use cases may involve third-party collaboration, which must be managed from a distributed team perspective. The work presented in [35] faced some complex integration issues, and showed that the implementation effort by third-parties must be properly aligned with the team’s development process.



IV. FUTURE WORK AND EXPECTED RESULTS

The preliminary results show the possibility to use models together with Scrum in projects with some perceived complexity, at this point with no support from a architecture design method. The next steps of this PhD will be towards using models in parallel with Scrum typical cycles, but where the adaptation of 4SRS to ASD cycles will be developed. In order to do so, work will be structured to address first O1 and then O2, in separate case studies. The future work regarding this PhD thesis is planned as showed in TABLE II.

Considering that changes in requirements are frequent (and easily addressed) in agile environments, the resulting artifact from O1 must imperatively be able to respond to those changes without losing information and not being subjected to unnecessary refactoring efforts. A research opportunity arises, where some existing knowledge and concepts will be taken in consideration, namely: Assessing the impact of changes in features within the architecture [23] as well as evaluating the architecture at the end of every cycle [21]; as the requirements are being developed and refined, the architect should identify architecturally significant requirements (ASR), feature- (or functionally-) oriented requirements, and their dependencies so necessary elements are present in upcoming iterations [6].



The outputs from this research objective may be used in contexts where: substantial changes to the software architecture need to be explored [20]; Given a change in features (adding, deleting or updating), the traceability-based algorithm determines a set of ordered design decisions [23]; “small rocks” (in opposition to the previously stated “big rocks”) are handled as they appear during the project [36].

Additionally, for this research objective to be fulfilled, the method must be able to be used in a typical software agile development and in contexts of LSA projects. For scope definition of the work, LSA is characterized by having more than one team, when the number of team members is larger than the typically suggested limit (7 or 8 elements), or when large quantities of user stories (or requirements) or lines of code are required. For both contexts (but especially in LSA projects), some existing knowledge and concepts will be taken in consideration, among them: prioritizing requirements and depicting dependencies between features [2], coordinating and synchronizing distributed teams [25].

Results from contexts like the ones from the research objectives are may be worthy of communication and adequate to scientific conferences in information systems, computer science and software engineering (products and processes) fields. Papers will be submitted to indexed conferences as:

* O1 results - in conferences and journals from software architectures and requirements artifacts fields, like European Conference on Software Architecture (ECSA), or Elsevier Computers in Industry;
* O2 results - in conferences and journals from software engineering process and process improvements fields, like International Conference on Product-Focused Software Process Improvement (PROFES), ACM Transactions on Information Systems, and Special issues, or book chapters, in agile architecting;
* O1 and O2 results that aim LSA projects - in conferences and journals from information systems and LSA fields, like International Conference on Advanced Information Systems Engineering (CAiSE), International Conference on Agile Software Development (XP), or special issues, or book chapters, in LSA.

V. CONCLUSIONS

At first sight, logical architectures and agile development approaches are not suitable for simultaneous use. However, the literature review showed that the possible coexistence between both is feasible and that there is an opportunity for research. There are already some approaches for using architecture methods in different agile stages. Some of them used architecture in early analysis, others used during the small cycle and/or at the end of the small cycle. It is more agreed that architecture is useful in the contexts of LSA projects. Additionally, it is also in these contexts that the architecture role has much to improve the current agile processes Past experiences showed that the 4SRS method is suitable for large-scale projects. This PhD proposes an adaptation of the 4SRS method for ASD.

Past projects between University of Minho and software companies, and afterwards with Bosch Car Multimedia, showed contexts where (architectural) models may be complementary to Scrum framework. In the future, another collaboration between University of Minho and Bosch Car Multimedia will allow to perform case studies in order to develop and validate the 4SRS in ASD projects.

Agile Architecture IS Possible – You First Have to Believe!

Abstract

In early 2007, ChannelAdvisor undertook an ambitious project to rearchitect one of its core product offerings. Flush with early success from rapid customer adoption, the foundation for this product had grown well beyond its initial design. As defects grew in frequency and severity, pressure mounted to “fix it yesterday”, but consensus on exactly how proved elusive. “Fix it right” ideals battled with “fix it now”, ultimately resulting in a new architecture that was inefficient, incomplete, and, after 3 long months of “working harder, not smarter”, cancelled. In this experience report I will recap my experience as a product development manager sponsoring this project; revealing how the project went awry, what the team learned, and how the utilization of the Scrum process not only created a scalable, reliable architecture, but also greatly improved the ongoing productivity and morale of the product team.

1. The Problem

Introduced about 2 years prior to this project, ChannelAdvisor offered a new product that at the time had undergone a massively successful launch with over 300% yearly growth in both number of customers as well as data footprint. The architecture that supported this product resided on 2 core database servers:

* The first server was a “transaction” server dedicated to processing millions of near-real time data transactions originating from external ecommerce partners every day. This server was online and processing transactions 24/7.
* The second server was a “reporting” server dedicated to aggregating the raw transactional data from the first server into numerous composite tables specifically optimized for query performance and utilized for the product’s reporting and data mining needs. Additionally, the data on this server provided the inputs to a large scale automated system that controlled both pricing and cost amounts for millions of ecommerce transactions every day. This server commonly managed hundreds of millions of records and was also functioning in a 24/7 environment.

In addition to these 2 core database servers, several home-grown processes existed to synchronize the data between those 2 systems. As more and more customers were added to the platform, it was not uncommon for this synchronization process to take 12 hours or more every day. Additionally, as load increased so did the occurrence of system timeouts, data inconsistencies (due to synchronization “race conditions”), and numerous other “at scale bugs” that would result in missing or duplicated data in the reporting tables.

From the customer’s point of view, this meant:

* Inconsistent reports: report A would show one data value, while report B would show that same data with a different value.
* Misleading reports: report totals reported higher or lower then what really happened.
* Incorrect pricing and cost amounts on customer’s ecommerce transactions, resulting in overspending and/or underselling.

This legacy architecture also suffered from reliance on an inherently expensive “Scale Up” design – as more and more customers were to be added to the platform, more memory, CPUs, and storage space was required to handle the increasing load. As the primary servers would need to become bigger and bigger, the hosting costs would in turn would grow exponentially larger.

2. The Response

As the bugs grew in severity and frequency, so did the cost: customers were losing money and faith in the product. External and internal pressures mounted – something needed to be done yesterday!

The response was as you may expect: round up the smartest people in the room, throw in some pizzas, and lock the door until a solution was found. Clever ideas rolled out left and right, but quick consensus on approach was not easily achieved between 2 primary schools of thought: “fix it right” versus “fix it now.”

2.1 “Fix It Right”

“This system was not built for this load, if we do not redesign the core architecture once and for all, we will hit these same problems again and again!”

This was a common argument from the senior architects and engineers on the project. A true enough statement to be sure, but came attached with a large price tag. The proposed solution:

* Combine all tables on the “transaction” and “reporting” servers into 1 master database server, eliminating the need to synchronize data, and thus reducing the cause of many of the data inconsistencies.
* Duplicate this consolidated architecture in whole on multiple servers, and update all client logic to partition all queries based on customer ID. Customers are assigned a specific server, and as you add more customers, you add more servers. (“scale out”).
* Partition large tables further by date, and limit queries to be directed to specific date tables. This in turn will optimize query performance and reduce timeout failures.

The design approach was deemed sound, but the plan of attack proved its undoing:

* 3-4 weeks needed to prototype this approach to make sure it would indeed solve the problems.
* 3-6 months implementation time by 3 “core engineers”, to be done in parallel with ongoing feature work for the product
* Perform massive code merge at end of project, as it was “too hard” to merge these large changes any earlier.
* All existing data had to be transformed into the new architecture via lengthy “reprocessing” task that would take days or weeks to complete due to the sheer volume of data.
* When complete, historical reports may produce different results since new architecture “fixed the bugs” that caused incorrect numbers in the past.

The net result was a proposal for a 3-6 month project fraught with risk and no easy rollback. Although the end result was attractive (IF delivered as promised), the business could not afford to wait so long for relief, something had to be done sooner.

2.2 “Fix It Now”

“Why do we have to rebuild the whole system? Let’s just fix the bugs!”

This was the common response from the business and customer representatives, but was this approach enough? The proposed solution:

* We don’t have time for a full rearchitecture, not everything can be bad. Let’s instead just focus on the top bugs and apply fixes to the existing architecture.
* Once the major bugs are fixed, we will revisit the “fix it right” plan.

This counterproposal received solid support, especially given the cheaper price tag (results in days instead of months), and ultimately was pursued with some short term success. Within 2-3 weeks, though, it was soon discovered that this approach would not be good enough. Efforts had degenerated into a “WhackA-Mole” bug hunt (fix one bug, create another) and customer problems did not seriously abate after the initial flurry of success.

“Fix It Right” again got its day in the sun, but with some bounds…

3. The Failed Project

Three months was simply too long to wait for a fix, but “just in time” bug fixes weren’t helping either. As a result, the architect team was given the green light to pursue their proposed design with some controls in place, namely:

* The product could not survive long enough to wait for the full solution, the team must deliver incremental improvements every 6 weeks or less.
* The team could have access to whatever people were deemed critical to the project, but this effort could not come at the expense of no competitive feature movement for 3 months. (“all will be for nothing if we can’t sell this product.”).
* Priority focus would be on data integrity issues first, scalability concerns second, and performance impediments third.

The project started with enthusiasm and optimism (“finally we are getting to build this the way it should have been built!”), but soon turned down a dark path. A lack of project management discipline coupled with “crisis fatigue” within the organization led to a mentality of “don’t concern me with the details, just fix it.” The architects were left alone to make things right, with little accountability or direction for the path they were taking.

For the first 6 week iteration scope initially was simple: reduce the possibility of data duplication by reducing the number of aggregated data tables. As time marched on scope creep set in (“while we’re fixing this, let’s also fix that”), purely done at the engineer level without business or project oversight. The project was construed as “an architecture project” and the architects were “handling it.”

As days turned into weeks, 6 weeks turned into 8, and then again to 10. Consistently we were “just 1 week away” from being done, but the train never left the tunnel. Morale plummeted and 80 hour work weeks settled in. Team members were added, but this only slowed things down further (a phenomenon well explained in the classic computer science book, The Mythical Man Month, by Fred Brooks).

After 3 months with no end in sight, the plug was pulled. Time to pause, reset, and rethink the approach. Three months down the drain, and the original problems were only getting worse.

4. What Went Wrong

Numerous contributing factors led to the project demise, but arguably they all traced back to poor project management. The urgency of problems coupled with crisis fatigue resulted in a relatively unchallenged blank check for the architects to “fix the problem.” When delays set in, the technical details were too obscure to be challenged, a leap of faith was made that the architecture team was doing all it could.

To be fair, the team was admirably working hard to “do the right thing” for the product, but classic “waterfall” training led to cultural biases that favored monolithic redesign over incremental value wins – without appropriate guidance and mentoring in place, there was simply an inability to understand how to deliver such a large project in smaller incremental wins.

Among the mistakes made:

* The original project mandate of 6 week iterations, although well intentioned and agile inspired, was not accompanied by the appropriate project management needed to make it happen. A 6 week iteration milestone soon became a 6 week deadline to complete a 3 month project.
* The highly technical nature of the solution led to a lack of accountability to the architects. The details of the project were difficult to understand to the layman, thus scope increased unchallenged and time delays were accepted simply because “we have to do this.”
* Time pressure caused the team to skip crucial research and prototype steps, resulting in solutions that were not identified as failures until weeks later.
* Scope creep pervaded – while fixing one thing, why not fix another? Estimates were not tracked and customer prioritization was not considered. As deadlines neared and the original goals remained incomplete, working hours increased while morale decreased. Too much time had been spent “working on the wrong things” and the team could never quite catch up.
* Adequate communication did not occur between the architecture team and the product team working on features in parallel. Features were being developed against an architecture that was radically changing, causing developers to shoot at a moving target. Additionally, the architecture team did not always fully understand the repercussions their design changes would have on existing and new product features.

5. Enter Scrum

Flush with humility from a failed project, the team was receptive to a new approach. Scrum had already been utilized by other ChannelAdvisor projects for several months prior. It was now time to ignore calls that “scrum can’t work for architecture” – it was time to give Scrum a chance!

As part of the introduction of Scrum to this project, numerous changes were made:

* A business oriented Product Owner was assigned to the project, providing clear guidance and accountability to customer needs.
* The Product and Architecture teams were reintegrated, both holding a stake in the success of the project. Coding was done in a common branch in the spirit of “continuous integration.”
* Sprint iterations were shortened to 1 month or less. Backlog scope was clearly identified prior to the sprint and team was held accountable to working with the product owner to stay focused on the backlog priorities.
* Research “spikes” and prototypes were allowed and accounted for with time-boxed restrictions. Time was made to “figure out next steps”, but not for open ended research walkabouts. An initial vision and roadmap was put in place prior to diving into the first sprint.

The addition of the loose structure of Scrum proved to be just the right amount of project management. Within the span of 3 months (the same timeframe as the first “failed” project), the following accomplishments were achieved:

1. Sprint 1 (first month): Prototyped a “consolidated single server” architecture where functional pieces of the application that originally were distributed across the 2 core database servers were now to be consolidated in one place on a single server, which in turn could be cloned to copies used for different sets of customer data. In other words, where all customers would access certain features on certain servers before, now some customers would access all features on certain servers instead. This approach would solve 2 core problems:
   1. Eliminates the data synchronization problems across servers since a single customer would not need to access data on any server other then their own
   2. Provides a linear “scale out” solution where more servers could be added at incremental cost to accommodate additional customers by simply cloning the configuration of that first server again and again.
2. Sprint 2 (second month): Implemented consolidated server technology researched in phase one, and further reduced number of aggregated data tables (tables that include copies of data from other tables) , in turn reducing the possibility of data integrity problems since there are less places for the data to get out of synch.
3. Sprint 3 (third month): Prototyped approach to improve performance by partitioning tables at date level. Further reduced number of composite tables to reduce data integrity problems further.
4. And onward…

Within the same period of time of time, the data integrity bugs reduced greatly, the product was much better prepared to scale out, and morale on the team increased significantly. The Product Owner kept priorities grounded in business realities, but the architects were also given their chance to set things right. Scope as well managed to facilitate “work smarter not harder” – the death marches were over!

6. One Year Later

Scrum was the secret sauce that allowed this product to break through its glass ceiling. The architecture is now reliable and scalable, and the team has never been more productive. Among the accomplishments in the last year:

* Data integrity problems have been all but eliminated through a simpler, more elegant design. The number of composite tables has been reduced from 16 to 4, and there is no more need for the failure-prone synchronization process that had caused so many of the failures in the past.
* The product is now truly a “scale out” solution, well partitioned across 4 servers and ready to handle ever increasing customer load simply by purchasing and plugging in more low cost servers.
* The product has more then doubled the number of active customers, with a corresponding doubling of data footprint. Over 180 million records are processed daily representing well over a terabyte of active data.
* The scrum team has held over 12 successful sprints since this project, releasing hundreds of enhancements and fixes over consistent monthly releases.
* Morale is much higher, and team is working sustainable 40(ish) hour weeks by “working smarter.”

7. Lessons Learned

Although a failed project is never desirable, if the lessons learned are heartfelt and a true “buy in” to improvement occurs, great opportunity can be built from the experience. Such was the case with this project and ChannelAdvisor as a whole, and the resultant positive cultural evolution paid back the cost tenfold.

Among the lessons learned:

7.1 Make the Time for Research

A major rearchitecture should not be rushed, but it should also not run unbounded. Spend the time to put a vision and roadmap in place at the start of your project, but expect change and plan for course corrections along the way. Do not allow for weeks of unbounded research, as even the best laid designs will need adjustment as time passes and the business environment changes.

In general we like to use a series of chained research “spikes” (the results of each drive the requirements for the next), each 2 days or less with clearly defined acceptance criteria. Acceptance criteria may include a flushed out high level design, a list of further research topics, or other tangible and reviewable artifacts. An output of a research tasks should never be as vague as “Yep it’ll work for us” – get specifics and hold the team accountable to next steps based on those specifics.

Additionally, If an engineer says something will take a week to research, they should be challenged to break those tasks into smaller components. Avoid chasing rabbit holes, leverage the product owner’s perspective to keep the team on priority focus as it matters to the customer.

7.2 Avoid The Monolithic Project

Confronted with the large scope of changes needed to be accomplished, the architecture team had originally presented the following arguments:

1. This will take us at least 6 months to deliver.
2. Architecture just can’t be done incrementally.
3. It will take longer to do in “parts” then all at once.

Experience on this project had proven these arguments to be naive, and in some cases flat out incorrect. Let’s take a look at each in turn:

7.2.1 “This will take us at least 6 months to deliver.”

A statement like this is sure to create sticker shock in any business sponsor. If you are facing problems today, you often can’t wait 6 months to fix them – you need something quicker. Where this project went wrong, though, was a misunderstanding that “quicker” meant “implement the same scope in less time.” Such an approach is unrealistic, fraught with risk and is not sustainable in the long run. Instead, the team learned to break a large task into incremental deliverables, and provide steady “production ready” value to the business along the way.

In time, the team learned the value in avoiding the “all of your problems will be fixed in 6 months” trap and instead focusing on “we will improve the highest priority issue in 4 weeks, revisit the next priority and improve that 4 weeks later, and repeat…”

7.2.2 “Architecture just can’t be done incrementally.”

Assuming the level of granularity mentioned is not considered to be trivial, I assert this statement can almost always be refuted. The misconception of the team on this project was that all changes had to be completely finalized and ready for mass consumption at release. A focus on how your customers truly work can help provide guidance on what really is needed when. Here are some strategies we commonly employed to provide incremental architectural wins:

* Roll out the new architecture “offline” and in parallel to the legacy architecture. Beta test select customers on the new architecture or side by side with the existing architecture. If things go wrong, pull the plug on the new architecture and revert back to the old.
* Given an idea on where you want the ultimate design to be, it is not always necessary that all components are refactored at the same time. Divide and conquer constituent components of the design in iterations, using temporary scaffolding when appropriate (proxy classes, interface adapters, etc.) to link the old components to the new architecture. When all old systems have been converted, destroy the old scaffolding and move on with the new.
* Leave it alone. Often the temptation to redesign an old system is due less to inherent design flaws then it is a lack of familiarity with how the old code works. You wouldn’t rebuild your house because you didn’t like the original color of the paint. Always keep the business perspective in mind when you determine what should and should not be refactored. “Joel On Software” Spolsky wrote what I consider the classic text on this point in his blog “Things You Should Never Do, Part I [Spolsky2000].”

7.2.3 “It will take longer to do in “parts” then all at once.”

This is only true if you do it 100% right on the first time, but how many of us can claim to do that? The value of incremental deliveries is when you can learn and adjust along the way. The sum of the individual parts may be greater if you are priced for perfection, but as weeks go into months it is inevitable that some factor will change, internally or externally, that will cause you to have to rethink your design to some degree. Incremental thinking gives you the ability to checkpoint and adjust along the way, rather then committing to a path that will not truly solve the problem when you finally get there.

7.3 Don’t Work in a Vacuum!

The architecture team worked independently from the product team to minimize disruptions to the feature work already underway. In the end this proved to be disastrous as the changing designs were too radically different to easily be merged back into the functional code branch without a major collaboration effort. In the end it was learned that the entire team needed to be involved in the communication and decision process, and constant merging through Continuous Integration reduced the complexity and risk considerably.

8. In Closing

This rearchitecture project certainly encountered its challenges and was a valuable lesson in learning that solid project management was just as important as a clever architectural design. More importantly, it proved to a skeptical audience that the old ways of “rebuild from the ground up” have become outdated and inadequate in the modern Internet business climate. Refactoring a large scale architecture is not only possible using agile methods, in this project it proved instrumental to its success! Don’t let the architects sell you on a full redesign, Agile Architecture IS possible – you first have to believe!