Scientific Research Publishing

Journal of Computer and Communications, 2018, 6, 171-189
http://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

Software Architectural Design in Agile Environments

Mehdi Mekni, Gayathri Buddhavarapu, Sandeep Chinthapatla, Mounika Gangula

Department of Computer Science and Information Technology, St. Cloud State University, St. Cloud, Minnesota, USA
Email: mmekni@stcloudstate.edu

How to cite this paper: Mekni, M., Buddhavarapu, G., Chinthapatla, S. and Gangula, M. (2018) Software Architectural Design in Agile Environments. *Journal of Computer and Communications*, **6**, 171-189.
https://doi.org/10.4236/jcc.2018.61018

Received: September 12, 2017
Accepted: December 26, 2017
Published: December 29, 2017

Abstract

In this paper, we propose a novel methodology to guide and assist practitioners supporting software architecture and design activities in agile environments. Software architecture and design is the skeleton of a system. It defines how the system has to behave in terms of different functional and non-functional requirements. Currently, a clear specification of software architectural design activities and processes in agile environments does not exist. Our methodology describes in detail the phases in the agile software design process and proposes techniques and tools to implement these phases.

Keywords

Agile Methodology, Software Development Life-Cycle, Software Architectural Design

1. Introduction

Software development projects seeking rapid, sustainable delivery are combining agile and architecture practices to manage competing goals of speed in the short term and stability over the long term [1] [2] [3]. A software development life-cycle is essentially a series of steps, or phases including requirement specification; software design; software construction; software verification and validation; and software deployment. These phases provide a model for the development and management of software [4].

Software architectural design is the process of applying various techniques and principle for the purpose of defining a module, a process, or a system in sufficient detail to permit its physical coding. The conventional approach to the software design process focuses on partitioning a problem and its solution into detailed pieces up front before proceeding to the construction phase. These up

front software architecture efforts are critical and leave no room to accommodate changing requirements later in the development cycle. Some of the issues faced by organizations involved in up front software design efforts are [5], [6]:

- Requirements evolve over time due to changes in customer and user needs, technological advancement and schedule constraints.
- Changes to requirements systematically involves modifying the software design, and in turn, the code.
- Accommodating changing software design is an expensive critical activity in the face of rapidly changing requirements.
- Clear specification of activities in the agile software design process is missing and there is a lack of a set of techniques for practitioners to choose from [7].

There is an obvious need for a software architectural design approach in agile environments. To the best of our knowledge, no well-established software design methodology has been proposed in any literature. These are issues of software architecture while fully supporting the fundamentals of agile software development methods. The rest of the paper is organized as follows: Section 2 provides an overview of existing agile methods. Section 3 details the software architecture design phase as a key part of the software development life-cycle. Section 4 presents the proposed software architectural design methodology in agile environments. Section 5 discusses the outcomes and limits of the proposed methodology. Finally, Section 6 concludes and presents the future perspectives of this work.

2. Agile Development Methods

The goal of agile methods is to allow an organization to be agile, but what does it mean to be Agile. Agile means being able to "Deliver quickly"; "Change quickly and often" [8].

While agile techniques vary in practices and emphasis, they follow the same principles behind the agile manifesto [9]:

- Working software is delivered frequently (weeks rather than months).
- Working software is the principal measure of progress.
- Customer satisfaction by rapid, continuous delivery of useful software.
- Late changes in software requirements are accepted.
- Close daily cooperation between business people and software developers.
- Face-to-face conversation is the best form of communication.
- Projects are built around motivated individuals who should be trusted.
- Continuous attention to technical excellence and good design.

Agile development methods have been designed to solve the problem of delivering high quality software on time under constantly and rapidly changing requirements and business environments. Agile methods have a proven track record in the software and IT industries. The main benefit of agile development software is allowing for an adaptive process—in which the team and development react to and handle changes in requirements and specifications, even late

in the development process. **Figure 1** illustrates an abstract view of the evolutionary map of main agile development methods.

Through the use of multiple working iterations, the implementation of agile methods allows the creation of quality, functional software with small teams and limited resources.

The proponents of the traditional development methods criticize the agile methods for the lightweight documentation and inability to cooperate within the traditional work-flow.

The main limitations of agile development are: agile works well for small to medium sized teams; also agile development methods do not scale, *i.e.* due to the number of iterations involved it would be difficult to understand the current project status; in addition, an agile approach requires highly motivated and skilled individuals which would not always be available; lastly, not enough written documentation in agile methods leads to information loss when the code is actually implemented. However, with proper implementation agile methods can complement and benefit traditional development methods. Furthermore, it should be noted that traditional development methods in non-iterative fashions are susceptible to late stage design breakage, while agile methodologies effectively solve this problem by frequent incremental builds which encourage changing requirements. We will now describe some common agile methods from a requirements engineering perspective.

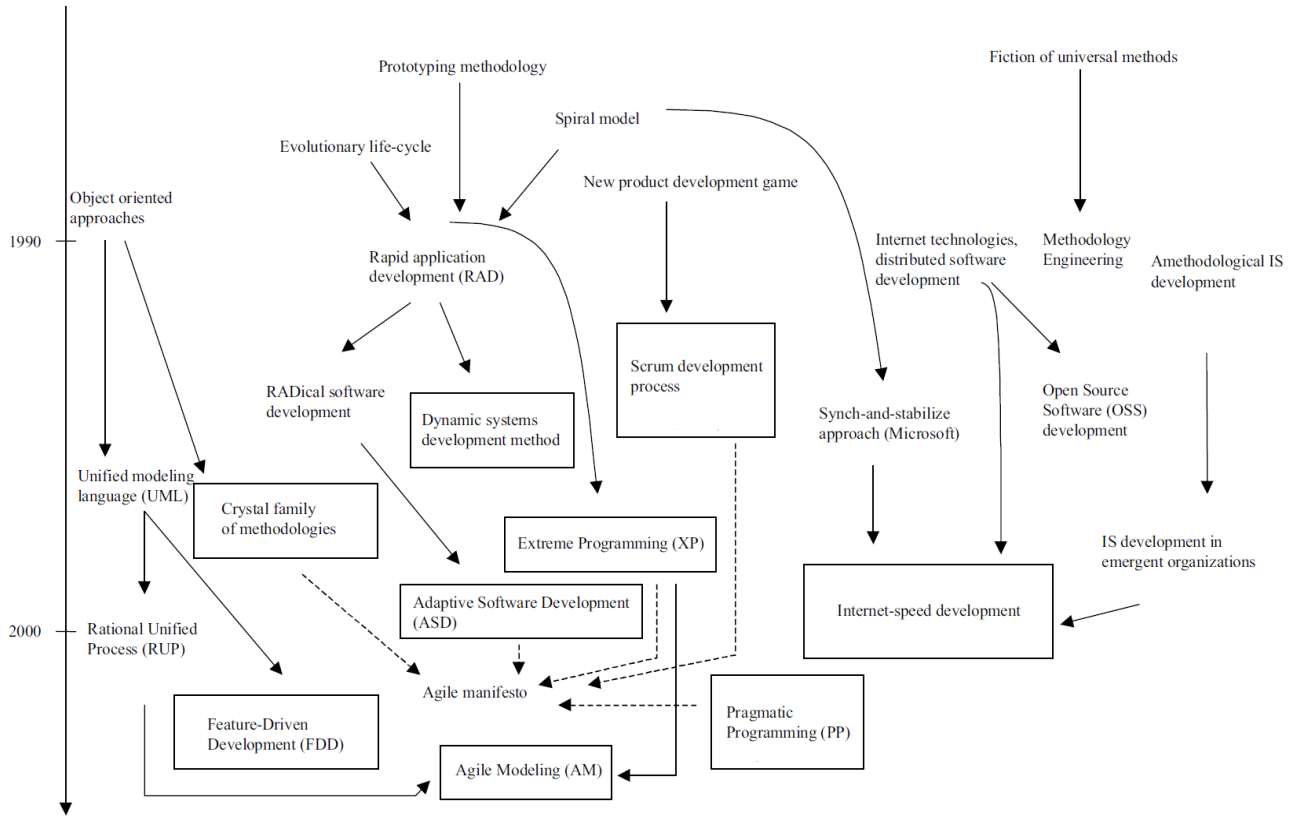

Figure 1. Evolutionary map of agile development methods (adapted from [10]).

2.1. Agile Modeling (AM)

Agile Modeling is a new approach for performing modeling activities [11]. It gives developers a guideline of how to build models-using an agile philosophy as its backbone-that resolve design problems and support documentation purposes but not "over-build" these models (**Figure 2**). The aim is to keep the amount of models and documentation.

2.2. Feature-Driven Development (FDD)

Feature-Driven Development consists of a minimalist, five-step process that focuses on building and design phases [12] each defined with entry and exit criteria, building a features list, and then planning-by-feature followed by iterative design-by-feature and build-by-feature steps. In the first phase, the overall domain model is developed by domain experts and developers. The overall model consists of class diagrams with classes, relationships, methods, and attributes. The methods express functionality and are the base for building a feature list (**Figure 3**). A feature in FDD is a client-valued function. The feature lists is prioritized by the team. The feature list is reviewed by domain members [13]. FDD proposes a weekly 30-minute meeting in which the status of the features is discussed and a report about the meeting is written.

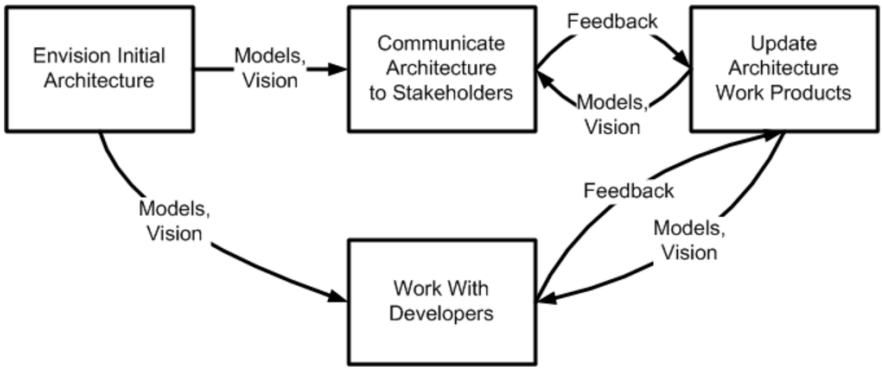

Figure 2. Agile modeling [11].

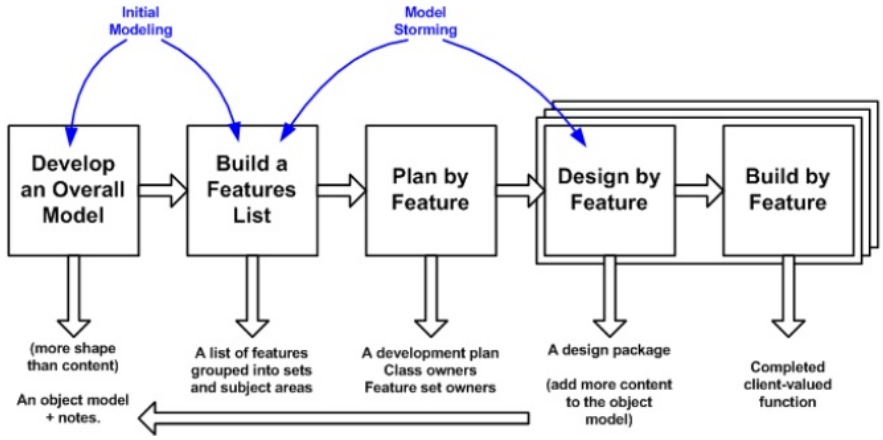

Figure 3. Feature-driven development [13].

2.3. Dynamic Systems Development Method (DSDM)

Dynamic Systems Development Method was developed in the U.K. in the mid-1990s [14]. It is an outgrowth of, and extension to, Rapid Application Development (RAD) practices [15]. The first two phases of DSDM are the feasibility study and the business study. During these two phases the base requirements are elicited (**Figure 4**). DSDM has nine principles include active user involvement, frequent delivery, team decision making, integrated testing throughout the project life cycle, and reversible changes in development.

2.4. Extreme Programming (XP)

Extreme Programming is based on values of simplicity, communication, feedback, and courage [16]. XP aims at enabling successful software development despite vague or constantly changing software requirements (**Figure 5**). XP relies on methods the individual practices are collected and lined up to function with each other. Some of the main practices of XP are short iterations with small releases and rapid feedback, close customer participation, constant communication and coordination, continuous refactoring, continuous integration and testing, and pair programming [17].

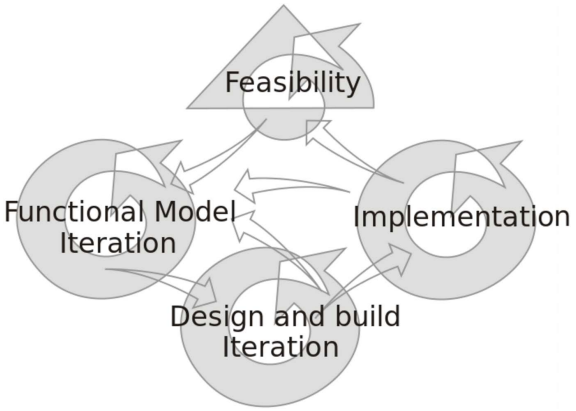

Figure 4. Dynamic systems development method [15].

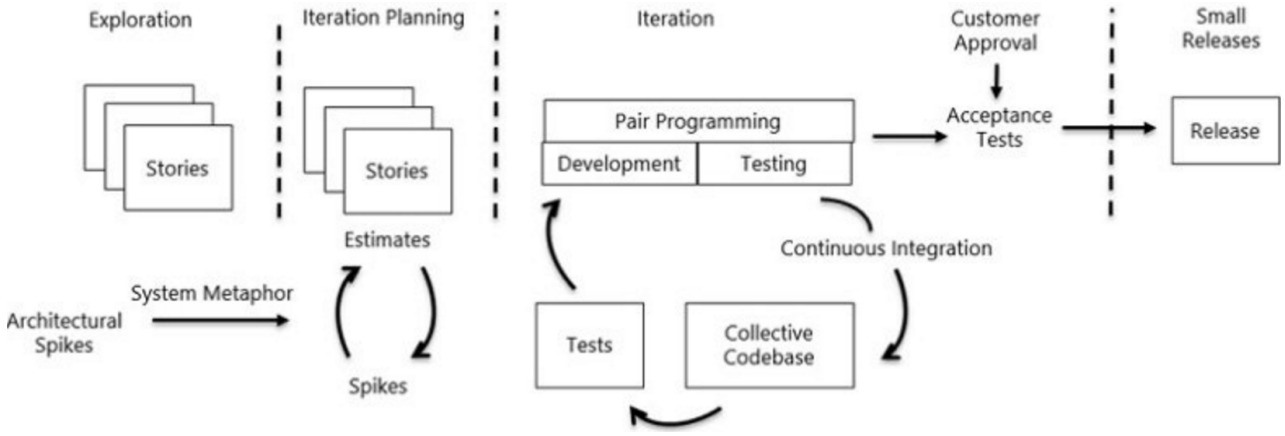

Figure 5. Extreme programming [17].

2.5. Scrum

Scrum is an empirical approach based on flexibility, adaptability and productivity [18]. Scrum allows developers to choose the specific software development techniques, methods, and practices for the implementation process. Scrum provides a project management framework that focuses development into 30-day sprint cycles in which a specified set of backlog features are delivered. The core practice in Scrum is the use of daily 15-minute team meetings for coordination and integration. Scrum has been in use for nearly ten years and has been used to successfully deliver a wide range of products. **Figure 6** details the work-flow of the Scrum agile software development.

2.6. Crystal Methodology

Crystal Methodology is a family of different approaches from which the appropriate methodologies can be chosen for each project [10]. Different members of the family can be tailored to fit varying circumstances. The members are indexed by different colors to indicate the "heaviness": Clear, Yellow, Orange, Red, Magenta, Blue, Violet [19]. Three Crystal methodologies have been used. These are Clear, Orange, and Orange Web. The difference between Orange and Orange Web is that Orange Web does not deal with a single project [10]. Crystal includes different agile methods fitting the needs of teams with different sizes (**Table 1**).

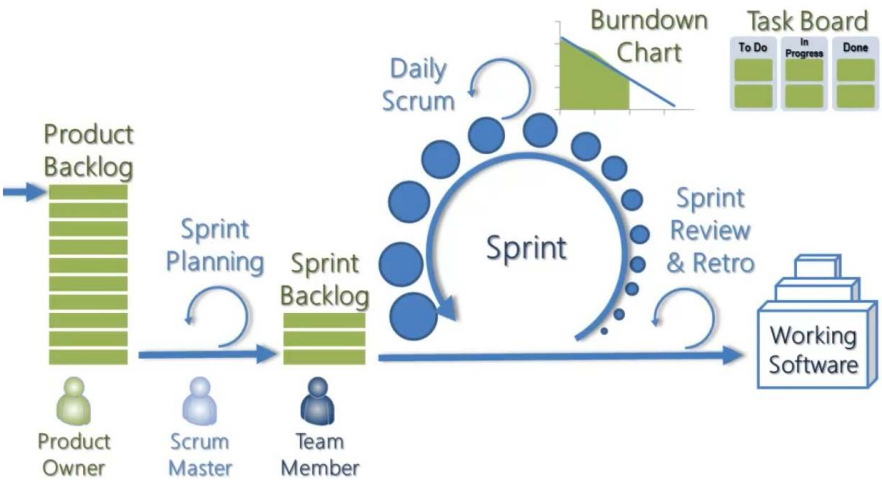

Figure 6. Scrum agile software development.

Table 1. Crystal family.

Methodology	Team (number of people)
Crystal Clear	2 - 6
Crystal Yellow	6 - 20
Crystal Orange	20 - 40
Crystal Red	40 - 80

2.7. Adaptive Software Development (ASD)

Adaptive software development attempts to bring about a new way of seeing software development in an organization, promoting an adaptive paradigm [20]. It offers solutions for the development of large and complex systems. The method encourages incremental and iterative development, with constant prototyping. One ancestor of ASD is "RADical Software Development" [21]. ASD claims to provide a framework with enough guidance to prevent projects from falling into chaos, while not suppressing emergence and creativity.

2.8. Internet Speed Development (ISD)

Internet-speed development is arguably the least known approach to agile software development. ISD refers to a situation where software needs to be released fast, thereby requiring short development cycles [22]. ISD puts forth a descriptive, management-oriented framework for addressing the problem of handling fast releases. This framework consists of time drivers, quality dependencies and process adjustments.

3. Software Architecture

3.1. Definition

Software architecture is a way of thinking about computing systems, for example, their configuration and design. By computing systems, we mean the hardware, the software and the communication components [6]. A set of components gathered together does not provide us with a problem solution [23]. We must impose a topology for interaction and communication upon them and ensure the components both integrate (physically communicate) as well as interoperate (logically communicate) [24].

3.2. Software Architecture Views

The process of software design and architecture is usually separated into four views: conceptual, module, execution, and code. This separation is based on our study of the software architectures of large systems, and on our experience designing and reviewing software architectures [25]. The different views address different engineering concerns, and separation of such concerns helps the architect make sound decisions about design trade-offs. The notion of this kind of separation is not unique: most of the work in software architecture to date either recognizes different architecture views or focuses on one particular view in order to explore its distinct characteristics and distinguish it from the others [23].

The 4 + 1 approach separates architecture into multiple views [26] [27]. The Garlen *et al.* work focuses on the conceptual view [28]. Over the years there has been a great deal of work on the module view [29]. Moreover, other works focus on the execution view, and in particular explores the dynamic aspects of a system [30]. The code view has been explored in the context of configuration management and system building.

The conceptual view describes the architecture in terms of domain elements. Here the architect designs the functional features of the system. For example, one common goal is to organize the architecture so that functional features can be added, removed, or modified. This is important for evolution, for supporting a product line, and for reuse across generations of a product.

The module view describes the decomposition of the software and its organization into layers. An important consideration here is limiting the impact of a change in external software or hardware. Another consideration is the focusing of software engineers' expertise, in order to increase implementation efficiency.

The execution view is the run-time view of the system: it is the mapping of modules to run-time images, defining the communication among them, and assigning them to physical resources. Resource usage and performance are key concerns in the execution view. Decisions such as whether to use a link library or a shared library, or whether to use threads or processes are made here, although these decisions may feed back to the module view and require changes there.

The code view captures how modules and interfaces in the module view are mapped to source files, and run-time images in the execution view are mapped to executable files. Some of the views also have a configuration, which constrains the elements by defining what roles they can play in a particular system. In the configuration, the architect may want to describe additional attributes or behavior associated with the elements, or to describe the behavior of the configuration as a whole.

3.3. Software Architecture Activities

Software architecture is comprised of a number of specific architecting activities (covering the entire architectural lifecycle) and a number of general architecting activities (supporting the specific activities). In the following sections, we provide a short overview on software architecture activities and processes. The specific software architecture activities are composed of five items:

- Architectural Analysis (AA) defines the problems an architecture must solve. The outcome of this activity is a set of architecturally significant requirements (ASRs) [31].
- Architectural Synthesis (AS) proposes candidate architecture solutions to address the ASRs collected in AA, thus this activity moves from the problem to the solution space [31].
- Architectural Evaluation (AE) ensures that the architectural design decisions made are the right ones, and the candidate architectural solutions proposed in AS are measured against the ASRs collected in AA [31].
- Architectural Implementation (AI) realizes the architecture by creating a detailed design [32].
- Architectural Maintenance and Evolution (AME) is to change an architecture for the purpose of fixing faults and architectural evolution is to respond to new requirements at the architectural level [33] [34] [35].

3.4. Software Architecture Processes

An architecture process is composed of the six specific items [31] [32]:

- Architectural Recovery (AR) is used to extract the current architecture of a system from the system's implementation [36].
- Architectural Description (ADp) is used to describe the architecture with a set of architectural elements (e.g., architecture views). This activity can help stakeholders (e.g., architects) understand the system, and improve the communication and cooperation among stakeholders [37].
- Architectural Understanding (AU) is used to comprehend the architectural elements (e.g., architectural decisions) and their relationships in an architecture design [38].
- Architectural Impact Analysis (AIA) is used to identify the architectural elements, which are affected by a change scenario [39]. The analysis results include the components in architecture that are affected directly, as well as the indirect effects of changes to the architecture [39].
- Architectural Reuse (ARu) aims at reusing existing architectural design elements, such as architecture frameworks, decisions, and patterns in the architecture of a new system [40].
- Architectural Refactoring (ARf) aims at improving the architectural structure of a system without changing its external behavior [38] [41].

4. Software Architectural Design in Agile Environments

The proposed methodology for software architectural design in agile environments is detailed in **Figure 7**.

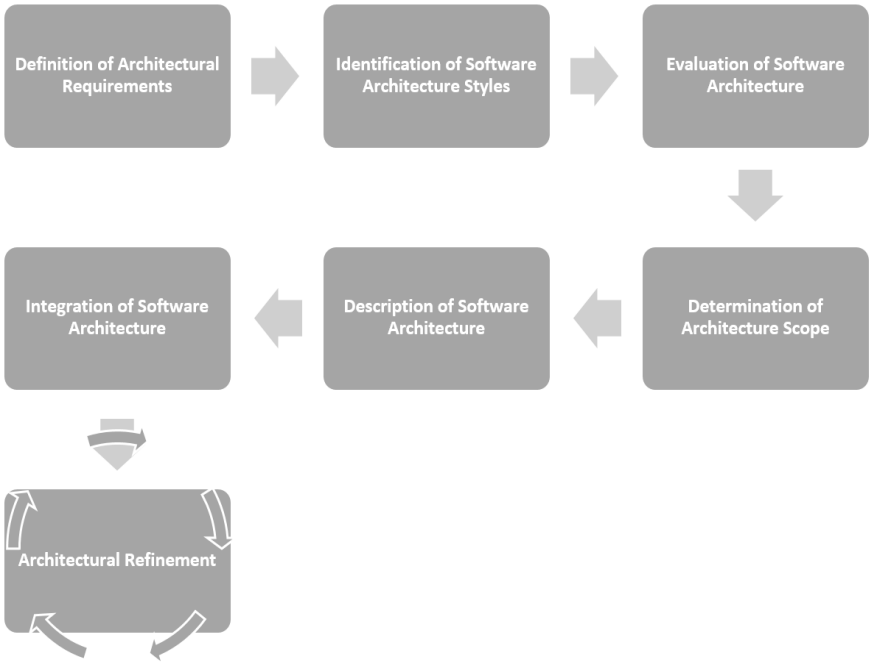

Figure 7. Software design methodology in agile environment.

4.1. Step 1: Definition of Architectural Requirements

Establishing the driving architectural requirements: Driving architectural requirements are obtained by analyzing the business drivers and system context as well as the issues deemed critical to system success by the product stakeholders. The goal is a specification for the architecture directing the architects to create a structure for the system that is sufficient to ensure success in the eyes of the stakeholders. These requirements prevent creation of an architecture that is overly complex or that strives for unnecessary elegance at the expense of critical system properties. The definition of architectural requirements aims to meet the following goals:

- Describe a necessary change to components in an architecture. This might mean adding new components, removing outdated ones, replacing or improving components, or changing the way in which they are organized and how they work together. What is going to change?
- Include the reasoning or motivations behind the change. Why does it need to change? It should explain why the existing components are inadequate, limiting or constraining. What problems, issues or concerns are caused by the current architecture?
- Outline the available options for future architectures that address all concerns. How do alternate target architectures eliminate the problems of the current architecture?
- Explain the benefits, value, risks, costs, opportunities, constraints, and future options associated with each alternative. How do we decide between one alternative and another?
- Outline any alternative routes to close the gaps and get from the current to the target architecture. How do we make the transition or transformation from what we have got now to what we need in the future?

4.2. Step 2: Identification of Software Architecture Styles

Architectural structures and coordination strategies are developed to satisfy the driving architectural requirements. Alternative architecture solutions may be proposed and analyzed to identify an optimal solution for the product or product line being developed. When product lines are involved, adaptation of the product line architecture to specific product requirements or fully develop the architecture for an individual product. The identification of software architecture styles aims to precise the associated elements, forms, and rationales:

- Elements: There are three classes of software elements, namely processing elements, data elements, and connecting elements. The processing elements are those components that take some data and apply transformations on them, and may generate updated or new data. The data elements are those that contain the information to be used, transformed and manipulated. The connecting elements bind the architectural description together by providing communication links between other components. The connecting elements

may themselves be processing or data elements, e.g., procedure calls, shared data, or messages.

- Forms: The architectural form consists of weighted properties and relationships. The definition implies that each component of the architecture would be characterized by some constraints, generally decided by the architect, and some kind of relationship with one or more other components. Properties define the constraints on the software elements to the degree desired by the architect.
- Rational: The rationale explains the different architectural decisions and choices; for example, why a particular architectural style or element or form was chosen. Rationale is tied to requirements, architectural views and stakeholders. Probably all choices are governed by what the requirement is. There are many different external components that have an interest in the system, and expect different things from the same system. We therefore have to consider the different external demands and expectations that affect and influence the architecture and its evolution.

4.3. Step 3: Evaluation of Software Architecture

Software architecture evaluation determines when and what methods of architecture evaluation are appropriate. The results of such evaluation are then analyzed and measures are determined and applied to improve the developing architecture. A formal software architecture evaluation should be a standard part of our software architecture methodology in agile environments. Software architecture evaluation is a cost-effective way of mitigating the substantial risks associated with this highly important artifact. The achievement of a software system's quality attribute depends much more on the software architecture than on code-related issues such as language choice, fine-grained design, algorithms, data structures, testing, and so forth. Most complex software systems are required to be modifiable and have good performance. They may also need to be secure, interoperable, portable, and reliable. Several software architecture evaluation methods exist in literature; Architecture Tradeoff Analysis Method (ATAM) [42], Software Architecture Analysis Method (SAAM), Active Reviews for Intermediate Designs (ARID) [43].

4.4. Step 4: Determination of Architecture Scope

Before defining an architecture, the developers determine how many of the system-design decisions should be established by the architecture of the system. This scope delimits the activities of application developers, allowing them to concentrate on what they do best. Software architecture scope is a reflection of system requirements and trade-offs that made to satisfy them. Possible scope determination factors include:

- Performance;
- Compatibility with legacy software;
- Software reuse;

- Distribution profile (current and future);
- Safety, security, fault tolerance, evolvability;
- Changes to processing algorithms or data representation;
- Modifications to the structure/functionality.

4.5. Step 5: Description of Software Architecture

An architecture must be described in sufficient detail and in an easily accessible form for developers and other stakeholders. The architecture is one of the major mechanisms that allow stakeholders to communicate about the properties of a system. Architecture documentation determines what views of software are useful for the stakeholders, the amount of detail required, and how to present the information efficiently. Agile methods agree strongly on a central point: "If information is not needed, do not document it". All documentation should have an intended use and audience in mind, and be produced in a way that serves both. One of the fundamental principles of technical documentation is "Write for the reader". Another central idea to remember is that documentation is not a monolithic activity that holds up all other progress until it is complete. With that in mind, the following is the suggested approach for describing software architecture using agile-like principles [44]:

- Create a skeleton document (document outline) for a comprehensive view-based software architecture document using the standard organization schemes;
- Decide which architectural views should be to produced, given the software architecture scope (step 4) with respect to available resources;
- Annotate each section of the outline with a list of the stakeholders who should find the information it contains of benefit;
- Prioritize the completion of the remaining sections. For example. If a section's constituency includes stakeholders for whom face-to-face conversation is impractical or impossible (e.g., maintainers in an as-yet-unidentified organization), that section will need to be filled in. If it includes only such stakeholders, its completion can be deferred until the conclusion of the software architecture and design phase.

4.6. Step 6: Integration of Software Architecture

The software architecture integration process is a set of procedures used to combine software architectural components into larger components, subsystems or final software architecture [37]. Software architecture integration enables the organization to observe all important attributes that a software will have; functionality, quality and performance. This is especially true for software systems as the integration is the first occurrence where the full result of the software development effort can be observed. Consequently, the integration activities represent a highly critical part of the software development process in agile environments. Usually, Architecture Analysis and Design Language (AADL) are used in order

to build integrated software-reliant systems [45]. The AADL is designed for the specification, analysis, automated integration and code generation of real-time performance-critical (timing, safety, fault tolerant, security, etc.) software. It allows analysis of system designs (and system of systems) prior to development and supports a model-based, model-driven development approach throughout the software development life cycle. During software architecture integration, the software architect, checks whether the models provided by the component developers, system deplorers, and domain experts as well as his or her own components assembly model are complete. If values are missing, the software architect estimates them or communicates with the responsible role. The result of this step (Integration of Software Architecture) is an overall quality-annotated model.

4.7. Step 7: Continuous Architectural Refinement

Architectural refinement aims to help provide the degree of architectural stability required to support the next iterations of development. This stability is particularly important to the successful operation of multiple parallel Scrum teams. Making architectural dependencies visible allows them to be managed and for teams to be aligned with them. The architecture refinement supports the team decoupling necessary to allow independent decision-making and reduce communication and coordination overhead. During the preparation phase, agile teams identify an architecture style of infrastructure sufficient to support the development of features in the near future. Product development using an architectural refinement most likely occurs in the preservation phase. Architectural refinement is one of the key factors to successfully scale agile. Describing and maintaining (through refinement) software architectural design enables a system infrastructure sufficient to allow incorporation of near-term high-priority features from the product backlog. The proposed software architecture methodology in agile environments allows the software architecture and design to support the features without potentially creating unanticipated rework by destabilizing refactoring. Larger software systems (and teams) need longer architectural refinements. Building and re-architecting software takes longer than a single iteration or release cycle. Delivery of planned functionality is more predictable when the architecture for the new features is already in place. This requires looking ahead in the planning process and investing in architecture by including design work in the present iteration that will support future features and customer needs. The architectural refinement is not complete. The refinement process intentionally is not complete because of an uncertain future with changing technology orientations and requirement engineering. This requires continuously extending the architectural refinement to support the development teams.

5. Discussion

Different agile methods cover different phases of the software development

life-cycle. However, none of them covers the software architectural design phase. Moreover, the rationalization of phases covered was missing. The question raised is whether an agile method is more profitable to cover more and to be more extensive, or cover less and to be more precise and specific. On one hand, some agile methods that cover too much ground, *i.e.* all organizations, phases and situations, are too general or shallow to be used. On the other hand, agile methods that cover too little (e.g., one phase) may be too restricted or lack a connection to other methods. Completeness, a notion introduced by Kumar and Welke [46], requires a method to be complete as opposed to partial. In the final analysis it was realized "completeness" is an element associated both with vertical (*i.e.*, level of detail) and horizontal (*i.e.*, life-cycle coverage) dimensions. None of the existing agile methods were either extensive or precise. Practitioners and experts are still struggling with partial solutions to problems that cover a wider area than agile methods do.

In the following subsections, we discuss the limits and perspectives of the architectural refinement process. Finally, we provide an overview on team organization in agile environment in support of software architecture and design activities and processes.

Relationship between Software Requirements and Architectural Activities in Agile Environments

The important feature of agile methods is that they do not assume that there is a sequential process, where each phase of the software development life-cycle is expected to be completed before proceeding to the next one, as for example in a classical waterfall process [47]. Thus it is expected that requirements engineering or software architecture phases are not happening just once, but they are rather continuously distributed along the development process. Once there is a first, usually incomplete, set of requirements available, an architect proceeds to the architectural design. A tighter integration of requirements engineering and software architectural activities is suggested in the twin peak process model [48]. While requirements engineering phases and architectural activities phases alternate in traditional processes, the twin peak model emphasizes that these two activities should be executed in parallel to support immediate continuous feedback from one to another (**Figure 8**). The goal of this process is that requirement analysts and software architects better understand problems by being aware of requirements and their prioritization non one hand and architecture and in particular architectural constraints on the other hand. Additionally, being able to quickly switch back and forth between the problem to solve (the requirements) and its solution (the architecture) can help to more clearly distinguish the two and to avoid mixing up problem and solution already in the requirements engineering phase.

Team Organization in its simplest instantiation, an agile development environment consists of a single collocated, cross-functional team with the skills,

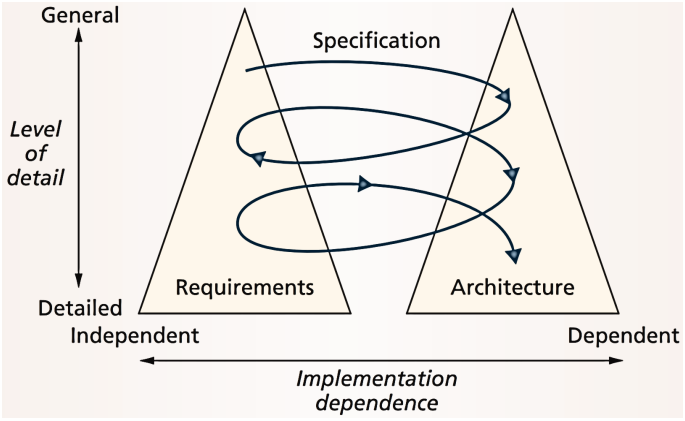

Figure 8. The twin peaks model [48] showing the interplay of requirements and architecture.

authority, and knowledge required to specify requirements and architect, design, code, and testing of the system. As software grows in size and complexity, the single-team model may no longer meet development demands.

A number of different strategies can be used to scale up the overall software development organization while maintaining an agile development approach. One approach is replication, essentially creating multiple Scrum teams with the same structure and responsibilities, sufficient to accomplish the required scope of work. Some organizations scale Scrum through a hybrid approach. The hybrid approach involves Scrum team replication but also supplements the cross-functional teams with traditional functionally oriented teams.

An example would be using an integration-and-test team to merge and validate code across multiple Scrum teams.

In general, we recognized two criteria used to organize the teams. First organizing the teams either horizontally or vertically and assigning different teams the responsibility for either components (horizontal) or features (vertical). The second is assigning the teams responsibilities according to development phases.

6. Conclusion and Future Works

In this paper, we provided an overview on software architectural design related issues in agile environments and proposed a methodology to guide and assist practitioners adopting agile software design in such environments. Our methodology relies on seven processes namely: 1) Definition of architectural requirements; 2) Identification of software architectural styles; 3) Evaluation of software architecture; 4) Determination of architecture scope; 5) Description of software architecture; 6) Integration of software architecture; and 7) Architectural refinement.

Agile software development methods have evoked a substantial amount of literature and debates. However, academic research on the subject is still scarce, as most existing publications are written by practitioners or consultants. Yet, many organizations are considering future use or have already applied practices that

are claiming successes in performing and delivering software in a more agile form.

To conclude, we observed that agile methods, without rationalization only cover certain phases of the life-cycle. A majority of them did not provide true support for software architectural design for project management. While universal solutions have strong support in the respective literature, empirical evidence on their adaptation and use in agile environments is currently very limited.

Acknowledgements

This research project has been partially funded by the professional developmental fund provided by St. Cloud State University. The author would like to thank faculty from Department of Computer Science and Information Technology for their valuable comments and continuous support.

References

[1] Buchmann, F., Nord, R.L. and Ozakaya, I. (2012) Architectural Tactics to Support Rapid and Agile Stability. Technical Report, DTIC Document.

[2] Floyd, C. (1992) Software Development as Reality Construction. In: *Software Development and Reality Construction*, Springer, 86-100. https://doi.org/10.1007/978-3-642-76817-0_10

[3] Nuseibeh, B. (2001) Weaving Together Requirements and Architectures. *Computer*, **34**, 115-119. https://doi.org/10.1109/2.910904

[4] Edeki, C. (2015) Agile Software Development Methodology. *European Journal of Mathematics and Computer Science*, **2**.

[5] Choudhary, B. and Rakesh, S.K. (2016) An Approach Using Agile Method for Software Development. 2016 *International Conference on Innovation and Challenges in Cyber Security* (*ICICCS-INBUSH*), 155-158. https://doi.org/10.1109/ICICCS.2016.7542304

[6] Sommerville, I. (1996) Software Process Models. *ACM Computing Surveys* (*CSUR*), **28**, 269-271. https://doi.org/10.1145/234313.234420

[7] Babar, M.A., Brown, A.W. and Mistrík, I. (2013) Agile Software Architecture: Aligning Agile Processes and Software Architectures. Newnes.

[8] Feiler, P. (2013) Architecture Analysis and Design Language (AADL) Annex Volume 3: Annex e: Error Model v2 Annex. Number SAE AS5506/3 (Draft) in SAE Aerospace Standard. SAE International.

[9] Magee, J. and Kramer, J. (1996) Dynamic Structure in Software Architectures. *ACM SIGSOFT Software Engineering Notes*, **21**, 3-14. https://doi.org/10.1145/250707.239104

[10] Awan, R., Muhammad, S., Fahiem, M. and Awan, S. (2016) A Hybrid Software Architecture Evaluation Method for Dynamic System Development Method. *Nucleus*, **53**, 180-187.

[11] Postma, A., America, P. and Wijnstra, J.G. (2004) Component Replacement in a Long-Living Architecture: The 3rdba Approach. *Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture*, *WICSA*, June 2004, 89-98. https://doi.org/10.1109/WICSA.2004.1310693

[12] Martini, A., Pareto, L. and Bosch, J. (2012) Enablers and Inhibitors for Speed with Reuse. *Proceedings of the* 16*th International Software Product Line Conference*, **1**, 116-125. https://doi.org/10.1145/2362536.2362554

[13] Clements, P., Ivers, J., Little, R., Nord, R. and Stafford, J. (2003) Documenting Software Architectures in an Agile World. Technical Report, DTIC Document.

[14] Babar, M.A., Zhu, L. and Jeffery, R. (2004) A Framework for Classifying and Comparing Software Architecture Evaluation Methods. *Proceedings of Australian Software Engineering Conference*, 309-318.

[15] Lata, P. (2016) Agile Software Development Methods. *International Journal of Computer* (*IJC*), **20**.

[16] Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H. and Carriere, J. (1998) The Architecture Tradeoff Analysis Method. *Fourth IEEE International Conference on Engineering of Complex Computer Systems*, *ICECCS*'98, 68-78. https://doi.org/10.1109/ICECCS.1998.706657

[17] Ambler, S.W. (2001) Agile Requirements Modeling. The Official Agile Modeling (AM) Site.

[18] Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J. and Little, R. (2002) Documenting Software Architectures: Views and Beyond. Pearson Education.

[19] Singh, S., Chaurasia, M. and Gaikwad, M.H. (2016) Importance of 4 + 1 Views Model in Soft-Ware Architecture. *Imperial Journal of Interdisciplinary Research*, **2**.

[20] Jaafar, N.H., Rahman, M.A. and Mokhtar, R. (2016) Adapting the Extreme Programming Approach in Developing E-Corrective and Preventive Actions: An Experience. *Regional Conference on Science, Technology and Social Sciences* (*RCSTSS* 2014), 801-809.

[21] Hofmeister, C., Kruchten, P., Nord, R.L., Obbink, H., Ran, A. and America, P. (2007) A General Model of Software Architecture Design Derived from Five Industrial Approaches. *Journal of Systems and Software*, **80**, 106-126. https://doi.org/10.1016/j.jss.2006.05.024

[22] Bellomo, S., Nord, R.L. and Ozkaya, I. (2013) A Study of Enabling Factors for Rapid Fielding Combined Practices to Balance Speed and Stability. 3*5th International Conference on Software Engineering* (*ICSE*), 982-991. https://doi.org/10.21236/ADA591481

[23] Bengtsson, P., Lassing, N., Bosch, J. and van Vliet, H. (2004) Architecture-Level Modifiability Analysis (ALMA). *Journal of Systems and Software*, **69**,129-147. https://doi.org/10.1016/S0164-1212(03)00080-3

[24] Kruchten, P. (1995) The 4 + 1 View Model of Architecture. *IEEE Software*, **12**, 42-50. https://doi.org/10.1109/52.469759

[25] Herzog, J. (2015) Software Architecture in Practice Third Edition Written by Len Bass, Paul Clements, Rick Kazman. *ACM SIGSOFT Software Engineering Notes*, **40**, 51-52. https://doi.org/10.1145/2693208.2693252

[26] Lange, B., Flynn, S., Proffitt, R., Chang, C.-Y., *et al.* (2015) Development of an Interactive Game-Based Rehabilitation Tool for Dynamic Balance Training. Topics in Stroke Rehabilitation.

[27] Tang, A., Avgeriou, P., Jansen, A., Capilla, R. and Babar, M.A. (2010) A Comparative Study of Architecture Knowledge Management Tools. *Journal of Systems and Software*, **83**, 352-370. https://doi.org/10.1016/j.jss.2009.08.032

[28] Dingsøyr, T. and Lassenius, C. (2016) Emerging Themes in Agile Software Development: Introduction to the Special Section on Continuous Value Delivery. *Infor-*

mation and Software Technology, **77**, 56-60.
https://doi.org/10.1016/j.infsof.2016.04.018

[29] Erickson, J., Lyytinen, K. and Siau, K. (2005) Agile Modeling, Agile Software Development, and Extreme Programming: The State of Research. *Journal of Database Management*, **16**, 88. https://doi.org/10.4018/jdm.2005100105

[30] Malavolta, I., Lago, P., Muccini, H., Pelliccione, P. and Tang, A. (2013) What Industry Needs from Architectural Languages: A Survey. *IEEE Trans. Softw. Eng.*, **39**, 869-891. https://doi.org/10.1109/TSE.2012.74

[31] Kaisler, S.H. (2005) Software Paradigms. John Wiley & Sons.
https://doi.org/10.1002/0471703567

[32] Yang, C., Liang, P. and Avgeriou, P. (2016) A Systematic Mapping Study on the Combination of Software Architecture and Agile Development. *Journal of Systems and Software*, **111**, 184. https://doi.org/10.1016/j.jss.2015.09.028

[33] IEEE Standard for Information Technology-System and Software Life Cycle Processes-Reuse Processes (2010) IEEE Std 1517-2010 (Revision of IEEE Std 1517-1999), 1-51.

[34] Abrahamsson, P., Salo, O., Ronkainen, J. and Warsta, J. (2002) Agile Software Development Methods: Review and Analysis.

[35] Ramesh, B., Cao, L. and Baskerville, R. (2010) Agile Requirements Engineering Practices and Challenges: An Empirical Study. *Information Systems Journal*, **20**, 449-480. https://doi.org/10.1111/j.1365-2575.2007.00259.x

[36] Nierstrasz, O. and Kurš, J. (2015) Parsing for Agile Modeling. *Science of Computer Programming*, **97**, 150-156. https://doi.org/10.1016/j.scico.2013.11.011

[37] Li, Z., Liang, P. and Avgeriou, P. (2013) Application of Knowledge-Based Approaches in Soft-Ware Architecture: A Systematic Mapping Study. *Information and Software Technology*, **55**, 777-794. https://doi.org/10.1016/j.infsof.2012.11.005

[38] Mahdavi-Hezave, R. and Ramsin, R. (2015) Fdmd: Feature-Driven Methodology Development. *International Conference on, Evaluation of Novel Approaches to Software Engineering* (*ENASE*), **2015**, 229-237.
https://doi.org/10.5220/0005384202290237

[39] Cej, A. (2010) Agile Software Development with Scrum.

[40] Abrahamsson, P., Warsta, J., Siponen, M.T. and Ronkainen, J. (2003) New Directions on Agile Methods: A Comparative Analysis. 25*th International Conference on Software Engineering*, 244-254.

[41] Baskerville, R., Ramesh, B., Levine, L., Pries-Heje, J. and Slaughter, S. (2003) Is Internet-Speed Software Development Different? *IEEE Software*, **20**, 70.
https://doi.org/10.1109/MS.2003.1241369

[42] Kumar, K. and Welke, R.J. (1992) Methodology Engineering R: A Proposal for Situation-Specific Methodology Construction. In: *Challenges and Strategies for Research in Systems Development*, John Wiley & Sons, Inc., 257-269.

[43] Bass, L., Clements, P. and Kazman, R. (2012) Software Architecture in Practice. 3rd Edition, Addison-Wesley Professional.

[44] Ducasse, S. and Pollet, D. (2009) Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, **35**, 573-591.
https://doi.org/10.1109/TSE.2009.19

[45] Highsmith, J. (2013) Adaptive Software Development: A Collaborative Approach to Managing Complex Systems. Addison-Wesley.

[46] Larsson, S. (2007) Key Elements of Software Product Integration Processes.

[47] Tekinerdogan, B. (2004) Asaam: Aspectual Software Architecture Analysis Method. *Fourth Working IEEE/IFIP Conference on Software Architecture*, *WICSA* 2004, 5-14. https://doi.org/10.1109/WICSA.2004.1310685

[48] Qian, K., Fu, X., Tao, L., Xu, C.-W. and Diaz-Herrera, J. (2009) Software Architecture and Design Illuminated. Jones and Bartlett Publishers, Inc., USA.