Architecture and AgileMaking SoftwareApproaches Work Together: [[1]](#footnote-1)

Foundations and

Approaches

Muhammad Ali Babar

The University of Adelaide, Adelaide, SA, Australia

CHAPTER CONTENTS

1.1 Introduction ........................................................................................................ 1

1.2 Software Architecture .......................................................................................... 3

1.2.1 Software Architecture Process and Architecture Lifecycle ......................4

CHAPTER

1.2.2 Architecturally Significant Requirements .............................................6

1.2.3 Software Architecture Design Methods .................................................8

1.2.4 Documenting Software Architecture ....................................................9

1.2.5 Software Architecture Evaluation ......................................................10

1.3 Agile Software Development and Architecture ..................................................... 11

1.3.1 Scrum ............................................................................................12

1.3.2 Extreme Programming ......................................................................13 1.4 Making Architectural and Agile Approaches Work ............................................... 14

Acknowledgments .................................................................................................... 18

1

of the Agile Manifesto,[[2]](#footnote-2) Agile practitioners have proposed several methods and approaches, such as Scrum [1], feature-driven development [2], extreme programming [3], and test-driven development. We refer to all of them as ASD methods in thischapter.Whilethere isnodoubtthatthere hasbeenmanifoldincrease intheadoptionofASDmethods byall sortsofcompanies,therehasalways been agrowingskepticism about the reliability, effectiveness, and efficiency of those ASD methods that do not pay sufficient attention to the important roles of SA-related principles, practices, and artifacts [4–6]. It has been widely recognized that SA can be an effective tool to cut development and evolution cost and time and to increase the conceptual integrity and quality of a system [7]. However, the followers of ASD methods view architecture-centric approaches as part of the plan-driven development paradigm [4]. According to them, upfront design and evaluation of SA as high ceremony activities are likely to consume a lot of time and effort without providing a system’s customers with valuable deliverables (i.e., code for features). The proponents of SA believe that sound architectural practices cannot be followed using agile approaches.

It can be asserted that this situation has arisen from two extreme views of ASD methods and SA-centric methods. The supporters of architecture-centric approaches appear to be less convinced that any software-intensive system of a significant size can be successfully built and evolved without paying sufficient attention to architectural issues, especially in domains such as automotive, telecommunication, finance, andmedicaldevices.TheadvocatesofASD methods appeartoapply Youaren’tgonna need it thinking to architecture-centric activities (e.g., design, evaluation, documentation). According to them, refactoring can help fix most of a software-intensive system’s structural problems. It has been claimed that refactoring is worthwhile as long as the high-level design is good enough to limit the need for large-scale refactoring [6,8,9]. And many experiences show that large-scale refactoring often results in significant defects, which are very costly to address later in the development lifecycle.

Mostofthe descriptionsofASDmethodspayverylittle attentionto commonarchitectural design activities [10], such as architectural analysis, architectural synthesis, architectural evaluation, and the artifact types [10] associated with these activities. Most of the ASD methods tend to assume that architectural design is high-level design without explicit structuring forces, such as quality attributes. Thapparambil [11] asserts that Refactoring is the primary method to develop architecture in the Agile world. The primary incremental design practice of the second edition of the XP book [3] claims that architecture can emerge in daily design. The emergent design means that architecture relies on looking for potentially poor architectural solutions in the implemented code and making a better architecture when needed through refactory. According to this approach, architecture emerges from code rather than some upfront structure.

It is beginning to be recognized that both disciplines (i.e., ASD methods and architecture-centric approaches) have important and complementary roles in software development and evolutionary activities. While ASD methods promise to enable companies to achieve efficiency, quality, and flexibility for accommodating changes,itiscriticallyimportant to follow solid architectural practices forlarge-scale software development projects. There is also agrowing recognitionof the importance of paying more attention to architectural aspects in agile approaches [4–6,12].

This situation has stimulated several efforts aimed at identifying the mechanics and prerequisites of integrating appropriate architecture-centric principles and practices in ASD methods [4,8]. One of the main objectives of these efforts is to help practitioners to understand the contextual factors and reasons for paying attention— or not—to the role and importance of a system’s SA when implementing ASD methods [8,13]. Researchers and practitioners have also identified the technical and organizational challenges involved in integrating Agile approaches in traditional software development methods [14,15]. However, while anecdotal evidence reveals that there are large organizations in the midst of agile transformation, and that the architectural issues are being addressed, there has been no significant effort to synthesize and present a reliable body of knowledge about the architecture-centric challenges faced by ASD projects and potential solutions to those challenges.

Thisbookprovidesauniqueandcomprehensivebodyofknowledgeaboutthechallenges and opportunities for making agile and architectural approaches coexist when developing safety, business, security, or mission-critical, software-intensive systems and services. The body of knowledge presented in this book is expected to help companies and practitioners build their architectural capabilities in the context of ASD methods or enable architectural-centric companies to make their approaches and practices agile and lean.That means companies will beable to adopt ASD methods without compromising on the architectural aspects of their software-intensive systems. The theories and practical examples in this book will enable companies of all sizes and contexts to gain the expertise, knowledge, and technological know-how to combine the strengths and benefits of architectural and ASD methods to achieve their goals. Quality, productivity, and profitability can be increased by improving the efficiency and effectiveness of companies’ software development processes.

In the following sections and subsections, we briefly describe some of the wellknown architecture-centric concepts and approaches and their origins and applicability contexts. Itcan beassertedthatthe SA-relatedconceptsand principlesdescribed inthis chapter can be tailored and integrated into ASD methods. Then we provide a brief description of two of the most popular ASD methods. And eventually, we discuss a few ways to integrate architecture-centric approaches in ASD methods.

# 1.2 SOFTWARE ARCHITECTURE

Software architecture is an important sub-discipline of software engineering. While SA’s important role in achieving the quality goals of a software-intensive system gained popularity during the 1990s, the idea of ensuring software quality through high-level design decisions emerged in the 1970s. Parnas showed how modularization and information hiding could be used as a means of improving a system’s flexibility and comprehensibility [16]. Soon after, Stevens et al. presented the idea of module coupling and cohesion as a characteristic of quality software design [17]. However, software engineers did not realize the importance of the relationship between non-functional requirements (NFRs) and SA design until the early 1990s. The practice of using design patterns and architectural styles for producing quality designs in short periods of time provided impetus for new interest in addressing quality issues at the architecture level [18–20].

Software architecture may mean different things for different people. It is difficult to claim that there is a widely accepted definition of SA in the software industry [21]. One of the first definitions of SA was provided by Perry and Wolf in their widely cited paper [22]. They define SA as follows:

SA ¼fElements;Form;Rationaleg

AccordingtothisdefinitionSAisacombinationof(1)asetofarchitecturalelements (i.e., processing, data, and connecting), (2) the form of these elements as principles guidingthe relationship between the elements and their properties, and (3)the rationale forchoosingelementsandtheirformincertainway.Thisdefinitionprovidedabasisfor initial research in the area of SA. The recent trend of describing SA as a set of design decisions and the rationale underpinning those design decisions has highlighted the importance of rationale in making and describing design decisions [23].

Bass, Clements, and Kazman [24] have defined SA in this way: The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

Structures in SA represent the partitioning and interaction decisions made to divide the responsibilities of satisfying requirements among a set of components and defining components’ relationships with each other. A structural partitioning is guided by the specific requirementsand constraintsofanapplication.Oneofthemainconsiderations duringthepartitioningdecisionsisto createa looselycoupled architecturefroma setof highly cohesive components to minimize dependencies between components. By controllingunnecessarydependencies,theeffectofchangesindifferentcomponentislocalized [7]. The structural partitioning should be driven by both functional requirements and NFRs. Architectural structures of large-scale, software intensive systems are considered critical to the satisfaction of many NFRs. Each architectural structure can help architects reason about a system’s different quality attributes. Architectural structures are documented using various architectural views [7].

## 1.2.1 Software architecture process and architecture lifecycle

It is also important to have a good understanding of the SA design process and the socalled lifecycle of SA. It is usually assumed that architecture design is a creative activity without a well-defined process. It can be considered a correct assumption for a large many systems’ architecture design. However, in a serious attempt to design and evaluate SA for a large-scale, complex system, it is important that there is a disciplined process that can support the creativity with a more controlled and reflective approach. Moreover, like any other artifact, SA also has a lifecycle that goes through different phases and activities. Each phase of the architecture lifecycle has its own prerequisites for use and applicability.

Several process models and methods have been devised and promoted for supporting the SA process. Some of the well-known ones are the attribute-driven design (ADD) method [7], business architecture process and organization [25], the Rationale Unified Process’s 4þ1 Views [26], Siemens’ 4 Views [27], and architectural separation of concerns [28]. In order to rationalize the options available to software project managers and architects, the developers of five well-known architecture design methods decided to develop a new general model of SA design by merging their respective SA design methods [29]. The original general model of architecture design consisted of three activities. This general model was extended by Tang and colleagues [30] to cover architectural materialization and evolution activities. For this chapter, we have slightly modified the names of the activities in the model. Each of the activities in the general model of architecture design has been briefly described below (see Figure 1.1):

1. Analyze problem domain: This activity consists of several sub-activities and tasks. This activity aims at defining the problems to be solved. Some of the main activities can be examining the architectural requirements (or even eliciting and clarifying architectural requirements), going through the stakeholders’ concerns and context to separate and prioritize the architecturally significant requirements (ASRs) from those that are not architecturally significant.
2. Design and describe architectural decisions: This activity aims at making key architectural design decisions based on the ASRs. An architect may consider several available design options before selecting the ones that appear to be the most appropriate and optimal. An architect is also responsible for documenting the designed architecture using appropriate documentation notations and templates.
3. Architectural evaluation: This activity intends to ensure that the architectural solutions chosen during the previous process are the right ones. Hence, the proposed architectural solutions are evaluated against the ASRs.

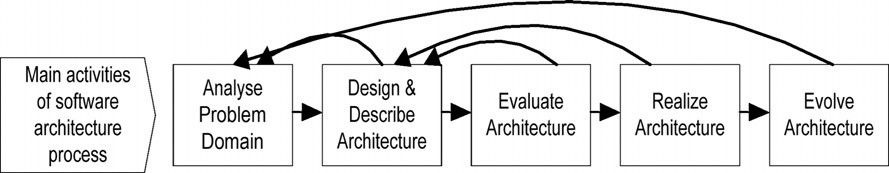


FIGURE 1.1

A model of software architecture process.

Based on [

31

].

1. Realize architecture: This is a phase wherein a designed architecture is deconstructed into a detailed design and implemented. During this phase, software developers make several dozen decisions which need to be aligned with the high-level architecture design decisions. That means software developers need to ensure that their decisions are in conformance with the architecture designed by an architect.
2. Maintenance of architecture: This involves making architectural changes as the architecture evolves because of enhancement and maintenance requirements, which place several new demands on the architecture underpinning a system. From the knowledge management perspective, prior design decisions are reassessed for the potential impact of the required changes and new decisions are made to accommodate the required changes without damaging the architectural integrity.

It should be noted that the abovementioned activities do not follow a sequential process like the waterfall model. Rather, these activities are undertaken in a quite iterative/evolutionary manner and tasks related to one particular activity can be performed and/or revisited while performing any other activity. In the following sub-sections, we briefly discuss different methods and techniques that are designed to support the SA process described in this section.

## 1.2.2 Architecturally significant requirements

Software requirements are mainly divided into functional requirements and NFRs. Functional requirements correspond to desired features of a system; while NFRs specify the required properties of a system. There are various terms used for NFRs, such as qualityattributes,constraints,goals,andnon-behavioralrequirements[32].Recently,it is being recognized that all these terms can be used for ASRs, but ASRs may include functional requirements as well. For this chapter, we specifically use the term architecturallysignificantrequirements(ASR)[33].Chenandcolleagueshaverecentlydefined ASRs as those requirements that have a measurable impact on a software system’s architecture [33]. Some obvious examples of ASRs are reliability, modifiability, performance, and usability. ASRs are usually subjective, relative, and interacting [32,33]. They are subjective, because they can be viewed, interpreted, and analyzed differently bydifferentpeopleandindifferentcontexts.ASRsarealsorelative,becausetheimportance of each ASR is often determined relative to other ASRs in a given context. ASRs are considered to interact in the sense that attempting to achieve a particular ASR may in turn positively or negatively affect other ASRs. ASRs are often specified by a system’s main stakeholders, such as end users, developers, managers, and maintainers. The ASRs are used to guide the SA design and analysis processes [7].

ASRs are less understood and managed than functional requirements [7,33–35]. This situation of ASRs not gaining sufficient attention upfront is quite common irrespective of the software development paradigm being used, whether Agile or nonAgile. Chung et al. claim that quality requirements (or ASRs as we call them) are generally stated informally during requirements analysis, are often contradictory, can be difficult to enforce during software development, and are not easy to validate when the software system is ready for delivery [32]. It is asserted that one of the main reasons for this state is the number and definitions of quality attributes that can be considered for a system. There are as many as 161 quality attributes listed in [32], which are not claimed to be an exhaustive list. Moreover, the existence of numerous classifications of quality attributes is another hurdle to fully comprehending the meaning of quality attributes. Moreover, there is no universal definition of so-called quality attributes (such as performance, availability, modifiability, and usability) that usually form the core of any set of ASRs. However, a precise specification of ASRs is important for facilitating rigorous analysis. In order to address this situation, SA researchers have proposed several approaches, such as scenarios for characterizing quality attributes [7,36], framework-based approach to characterizing ASRs [33], and architecturally savvy personas [37]. All three of these approaches are not only complementary to each other when it comes to eliciting and specifying ASRs for architecture design and evaluation but can also be easily integrated in ASD methods for treating the ASRs’ as first-class entities.

The approach based on architecturally savvy personas has been described in Chapter 4 of this book. Chen and colleagues have provided an evidence-based framework for systematically characterizing ASRs [33]. The framework is expected to cater to different stakeholders’ needs for eliciting, specifying, and understanding ASRs for designingandevaluatingarchitecturaldesigndecisions.Scenarioshavebeenusedfora long time in several areas of different disciplines (military and business strategy, decision making,). Scenarios are expected to be an effective means of specifying quality attributes for SA processes because they are normally very concrete, enabling the user to easily and precisely understand their detailed effect [38]. A scenario is a textual, system-independent specification of a quality attribute [7]. A well-structured scenario must clearly state an ASR in terms of stimulus and response. It is important that a scenario have clearly identifiable response measures to successfully analyze SAs. Bass et al. [7] provided a framework (shown in Table 1.1) to structure scenarios.

|  |  |
| --- | --- |
| Table 1.1 Six Elements Scenario Generation Framework [7] | |
| Elements | Brief Description |
| Stimulus A condition that needs to be considered when it arrives at a system  Response The activity undertaken after the arrival of the stimulus  Source of An entity (human, system, or any actuator) that generates the stimulus stimulus  Environment A system’s condition when a stimulus occurs, e.g., overloaded, running, etc.  Stimulated Some artifact that is stimulated; may be the whole system or a part of it artifact  Response The response to the stimulus should be measurable in some fashion so measure that the requirement can be tested | |

The scenario generation framework shown in Table 1.1 is considered quite effective for eliciting and structuring scenarios gathered from stakeholders. It is argued that this framework provides a relatively rigorous and systematic approach to capture and document quality-sensitive scenarios, which can be used to select an appropriate reasoning framework for analyzing SA. Scenarios can be abstract or concrete. Abstract scenarios are used to aid in the bottom-up elicitation of scenarios. The abstract scenarios are system independent and focused on ASRs. Concrete scenario is a textual specification of an ASR for a particular system.

## 1.2.3 Software architecture design methods

The software architecture community has developed several methods and techniques to support the architecture design process. One of the key differentiating aspects of the design methods developed by the SA researchers and practitioners is that they elevate ASRs from being almost totally ignored to being an important consideration during SA design. Each of architecture-centric design methods has its strengths and weaknesses. One way of leveraging their strengths and overcoming weak points is to select different approaches and techniques from different methods and apply them based on contextual requirements.

Bosch [39] proposed a method that explicitly considers ASRs during the design process. Hofmeister and colleagues proposed a framework—global analysis—to identify, accommodate, and describe ASRs early into the design phase [27]. However, these methods have their critics for considering functional requirements ahead of ASRs. The work of Chung et al. provides a framework to systematically deal with NFRs duringthe designprocess[32].TheNFR framework helps formally reasonabout the relationshipbetween adesigndecisionandsupportedorinhibitedqualityattributes. However, it does not provide support to explicitly perform trade-off analysis between competing design decisions. Researchers from the Software Engineering Institute (SEI) have developed several methods to support architecture design—for example, ADD [7] and attribute-based architecture styles [40]. Al-Naeem et al. [41] have proposed an architectural decision-making support framework for designing an architecture that is composed of design decisions already evaluated with respect to desired quality attributes and organizational constraints.

From this brief analysis of the well-known architecture-centric design methods, it is clear that an architecture design method should not only help identify suitable design solutions with respect to ASRs but must also include an activity to determine if the proposed architecture design has the potential to fulfill the required ASRs. Most of the existing design methods attempt to leverage knowledge-based approaches in terms of applying design patterns and architectural styles. However, most of the existing architecture-centric methods are considered heavyweight and ceremonial. Hence, they need to be appropriately tailored and contextualized for ASD environments. Several research efforts are geared toward providing guidance on how to tailor architecture design and evaluation methods for agile methods [42,43].

## 1.2.4 Documenting software architecture

It is well recognized that architecture is a vehicle for communication among stakeholders. Hence, it should be described unambiguously and in sufficient details, which can provide relevant information to each type of stakeholder [44]. Architectural documentation is also a vital artifact for several key activities, such as architecture decisions analysis, work breakdown, and post-deployment maintenance [7]. Architecture documentation may consume a large amount of resources that need to be justifiably allocated. That is why architecture documentation is not commonly practiced in general and in agile and lean worlds in particular. An important issue in architecture documentation is to choose a suitable means of architecture description that can serve the main goals (e.g., communication, analysis, implementation, and maintenance) of documenting SAs.

The boxes and lines notation is probably the most commonly used technique for explaining or documenting architectural decisions [21]. However, without having sufficient contextual information, such architectural description can be interpreted in several ways. Moreover, the boxes and lines notation does not capture several other types of information (such as interfaces and behavioral aspects). Hence, this notation is not considered expressive enough to communicate architectural decisions in a manner that serves the abovementioned main objectives of architecture documentation.

Recently, there has been an increasing emphasis on documenting SAs using different views suitable to different stakeholders [45]. An architectural view is a representation of a system from a related set of concerns, which are important to different stakeholders. Hence, each view addresses the concerns of one or more of a system’s stakeholders. The term “view” is used to express a system’s architecture with respect to a particular viewpoint. According to the IEEE standards for describing SA [45], architectural description is organized into various views. One of the most popular views-based approaches is called “4þ1” views [26]. The 4þ1 view model intends to describe an SA using five concurrent views. Each of them addresses a specific set of concerns.

* Logical view denotes the partitions of the functional requirements onto the logical entities in an architecture. This view illustrates a design’s object model in an object-oriented design approach.
* Process view is used to represent some types of ASRs, such as concurrency and performance. This view can be described at several levels of abstraction, each of which addresses an individual issue.
* Development view illustrates the organization of the actual software modules in the software development environment. This view also represents internal properties, such as reusability, ease of development, testability, and commonality. It is usually made up of subsystems, which are organized in a hierarchy of layers. This view also supports allocation of requirements and work division, cost assessment, planning, progress monitoring, and reasoning about reuse, portability and security.
* Physical view represents the mapping of the architectural elements captured in the logical, process, and development views onto networks of computers. This view takes into consideration the NFRs (e.g., availability, reliability (fault tolerance), performance (throughput), and scalability).
* Scenarios are used to demonstrate that the elements of other views can work together seamlessly. This fifth view is made up of a small subset of important scenarios and has two main objectives: design driver, and validation/illustration.

Clements and colleagues have proposed another approach, called Views and Beyond (V&B) [44], to documenting SA using views. Like the IEEE Std 1471, their approach is based on the philosophy that instead of prescribing a fixed set of views like Kruchten, SA should be documented using whatever views are useful for a system being designed. The V&B approach’s main contribution is to map concrete architectural styles to views and providing templates to capture relevant information. Apart from architecture documentation approaches, the SA community has proposed several ADLs (such as Rapide [46] and Unicon [47]), which are considered formal approaches to describing SA. There have been two comparative studies of the Architectural Description Languages (ADLs) reported in [48,49]. Unified Modeling Language (UML) [50] has become a de facto standard notation for documenting a software for any kinds of software development environment, agile or non-agile. Before a major upgrade in the UML 2.0, the UML had nine diagrams: class diagram, object diagram, use case diagram, sequence diagram, collaboration diagram, state chart diagram,activity diagram, component diagram, and deployment diagram. The UML 2.0 has addressed a major weakness of UML by providing new diagrams for describing the structure and behavior of a system.

## 1.2.5 Software architecture evaluation

Software architecture evaluation is an important activity in the software architecting process. The fundamental goal of architecture evaluation is to assess the potential of a proposed/chosen architecture to deliver a system capable of fulfilling required quality requirements and to identify any potential risks [51,52]. Researchers and practitioners have proposed a large number of architecture evaluation methods for which a classification and comparison framework has also been proposed [53]. Most widely used architecture evaluation methods are scenario-based. These methods are called scenario-based because scenarios are used to characterize the quality attributes required of a system. It is believed that scenario-based analysis is suitable for development-time quality attributes (such as maintainability and usability) rather than for run-time quality attributes (such as performance and scalability), which can be assessed using quantitative techniques such as simulation or mathematical models [39]. Among the well-known, scenario-based architecture evaluation methods are the SA analysis method (SAAM) [54], the architecture tradeoff analysis method (ATAM) [55], the architecture level maintainability analysis (ALMA) [56], and the performance assessment of SA (PASA) [57].

### 1.3 Agile Software Development and Architecture

SAAM is the earliest method proposed to analyze architecture using scenarios. The analysis of multiple candidate architectures requires applying SAAM to each of the proposed architectures and then comparing the results. This can be very costly in terms of time and effort if the number of architectures to be compared is large. SAAM has been further extended into a number of methods, such as SAAM for complex scenarios [58], extending SAAM by integration in the domain-centric and reusebased development process [59], and SAAM for evolution and reusability [60]. ATAM grew outofSAAM. The key advantagesofATAM areexplicitways ofunderstanding how an architecture supports multiple competing quality attributes and of performing trade-off analysis. ATAM uses both qualitative techniques, such as scenarios, and quantitative techniques for measuring the qualities of the architecture.

Benstsson and Bosch proposedseveral methods(such as SBAR [61],ALPSM [62], and ALMA) [56]. All these methods use one or a combination of various analysis techniques (i.e., scenarios, simulation, mathematical modeling, or experience-based reasoning [39]). All of these methods use scenarios to characterize quality attributes. The desired scenarios are mappedontoarchitecturalcomponents toassessthe architecture’s capability to support those scenarios or identify the changes required to handle those scenarios. PASA is an architecture analysis method that combines scenarios and quantitative techniques [57]. PASA uses scenarios to determine a system’s performance objectives and applies principles and techniques from software performance engineering (SPE) to determine whether an architecture is capable of supporting the performance scenarios. PASA includes performance-sensitive architectural styles and anti-patterns as analysis tools and formalizes the architecture analysis activity of the performance engineering process reported in [63].

# 1.3 AGILE SOFTWARE DEVELOPMENT AND ARCHITECTURE

Agile software development methods promise to support continuous feedback and accommodate changes in software requirements throughout the software development life cycle, support close collaboration between customers and developers, and enable early and frequent delivery of software features required for a system [4]. The ASD methods are based on the Agile Manifesto that was published by a group of software developers and consultants in 2001. According to the Agile Manifesto:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

* Individuals and interactions over process and tools,
* Working software over comprehensive documents, • Customer collaboration over contract negotiation,
* Responding to change over following a plan.

That is, while there is value in the items on the right, we value the items on the left more

This manifesto describes the core values underpinning the agile community’s views about different aspects of software development processes, people, practices, and artifacts. According to this manifesto, the ASD methods are designed and implemented in ways that are aligned with the core ASD values, such as individuals and interactions over process and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan [13]. Agile Alliance has also enlisted a number of common principles for agile processes, including customer satisfaction through early and continuous software delivery, co-located active customer participation, ability to handle change even late in the software development lifecycle, simplicity of software development processes, short feedback loops, mutual trust, and common code ownership [14].

Some of the well-known ASD methods are extreme programming (XP) [3], Crystal Clear [64] and Scrum [1]. There are a large number of books and research papers for describing the details of each of the well-known and widely practiced ASD methods. We can refer a reader of this chapter to two good sources [65,66] for introductory information about different ASD methods such as Scrum, feature driven development, dynamic software development method, adaptive software development, extreme programming, and crystal methodologies. Since there are a large variety of ASD and practices, it seems appropriate that we keep our views and discussions focused on integrating architectural approaches into a few well-known and widely adopted ASD methods. Hence, this chapter will briefly touch on two of the well-known ASD methods—Scrum, an agile project management method, and XP. By limiting the number of agile methods for discussion with respect to architectural principles and practices, we expect to provide a precise but more coherent and deep discussion of how to make ASD methods and architecture-centric practices work in harmony to leverage the advantages of both disciplines for developing high-quality and cost-effective software iteratively and incrementally without unnecessary project delays and risks.

## 1.3.1 Scrum

Scrum has emerged as one of the leading (if not the leading) ASD method that has been designed to manage software development projects. Scrum is a term used in the game of rugby where it means “getting an out-of-play ball back into the game” through team efforts [67]. In software development, Scrum is an iterative and incremental project management approach that provides a simple inspect and adapt framework rather than specific techniques. Scrum-based projects deliver software in increments called sprints (usually 3-4 week iterations, or even 2 weeks in some instances). Each sprint starts with planning, during which user stories are taken from backlogs based on priorities, and ends with a sprint review. The planning activity is expected to last for a few hours (e.g., 4 hours) and not too long. The sprint review meeting can also last around 4 hours. All the key stakeholders are expected to participate in the sprint planning and the sprint review meetings at the beginning and completion of each sprint.

### 1.3 Agile Software Development and Architecture

A Scrum team holds a short meeting (e.g., maximum 15 mininutes) at the beginning of each day. This meeting is called the “daily Scrum meeting,” and is aimed at enabling each team member to addresses only three questions: “What did I do yesterday, what will I do today, and what are the showstoppers in my work?” Each Scrum project is expected to have at least three artifacts: product backlogs, sprint backlogs, and burn-down charts. The software architecture community has also borrowed the term “backlogs” and proposed that the architecting process should keep an architectural backlog when architecture is being designed and evaluated iteratively. The Scrum backlogs consist of requirements that need to be implemented during the current or future sprint cycles. An iterative and incremental approach to architecting also incorporates the concept of architectural backlogs [10]. A third artifact is the daily burn-down chart that is aimed at providing a status report in terms of the cumulative work yet to be done.

## 1.3.2 Extreme programming

Extreme programming is another popular agile approach that was developed based on commonsense principles and practices taken to extreme levels. Like other ASD methods, XP also advocates short iteration and frequent releases of working code with the aim of increasing productivity but still accommodating requirements changes. XP was designed for collocated teams of eight to ten developers working with object-oriented programming language. The approach quickly became popular, among software developers who were not satisfied with the traditional software development approaches like waterfall. Following are some of the key XP practices.

* Planning game: A close interaction between customers and developers is encouraged for estimating and prioritizing requirements for the next release. The requirements are captured as users’ stories on story cards. The programmers are expected to plan and deliver only the user stories agreed upon with customers.
* Small releases: An initial version of a system is released for operation after a few iterations. New features are delivered in subsequent releases on a daily or weekly basis.
* Metaphor: The development team and customers develop a set of metaphors for modeling the system to be developed.
* Simple design: XP encourages developers to keep the design of a system as simple as possible. According to Beck say everything once and only once.
* Tests: The test-first principle means developers write acceptance tests for their code before they write the code itself. Customers write functional tests for each iteration, and at the end of each iteration, all tests are expected to run successfully.
* Refactoring: The design of a system evolved by transforming existing design of the system in a way that all the test cases run successfully.
* Pair programming: The production code is written by two developers sitting next to each other on a computer.
* Continuous integration: All new code is integrated into the system as frequently as possible. All functional tests must still pass after integration or the new code is discarded.
* Collective ownership: All developers working on a system jointly own the code. That means any developer can make changes anywhere in the code at any time it is felt necessary.
* On-site customer: A customer sits with the development team all the time. The onsite customer answers questions, performs acceptance tests, and ensures progress on the development.
* Fourty-hour weeks: If someone from the development team has to work overtime in two consecutive weeks, it is a sign of a big problem. The requirements should be selected for each iteration in a way that developers do not need to put in overtime.
* Open workspace: Developers have a common workspace set up with small cubicles around the periphery and a common development machine in the center for pair programmers.
* Justrules:Ateam’smemberssubscribetoasetofrules.Therulescanbechangedat anytimeaslongasthereisaconsensusabouthowtoassesstheeffectsofthechange.

# 1.4 MAKING ARCHITECTURAL AND AGILE APPROACHES WORK

It has been stated throughout this chapter that there is a growing recognition of the importance of paying more attention to architectural aspects in Agile approaches [4–6,14]. Hence, there are an increasing number of efforts aimed at identifying the technical and organizational challenges involved in integrating agile approaches in traditional software development methods [14,15]. These efforts have resulted in several proposals for combining the strengths of the core elements of agile and architecture-centric approaches. For example, Refs. [68,69] combine the strengths of the core elements of the risk-driven, architecture-centric rational unified process (RUP) and the XP [3] process. The combinations were enabled by the fact that RUP and XP share the cornerstones of iterative, incremental, and evolutionary development [70]. Nord and Tomayko [4] propose an integration of specific SEI architecture-centric methods into the XP framework [71]. Many others have emphasized the importance of finding a middle ground between two extreme views of architecture-centric and agile approaches [9,13,12]. Beck has also emphasized the importance of paying sufficient attention to quality attributes and the need of scaling XP based on the context. For example, he states the following in the second edition of his book, XP Explained: Embracing Change:

A system isn’t certifiably secure unless it has been built with a set of security principles in mind and has been audited by a security expert. While compatible with XP, these practices have to be incorporated into the team’s daily work.

## 1.4 Making Architectural and Agile Approaches Work

With awareness and appropriate adaptations, XP does scale. Some problems can be simplified to be easily handled by a small XP team. For others, XP must be augmented. The basic value and principles apply at all scales. The practices can be modified to suit your situation.

It can be argued that one of the important prerequisites for bridging the gap between agile and architectural approaches is to build and disseminate an evidence-based body of knowledge about the points of conflict and conciliation between agile and architectural principles, practices, and their proponents’ views. Such a body of knowledge should also include the challenges that software development teams face when they attempt to follow architecturally savvy principles in an agile development shop and the problems and risks that may have appeared in agile projects that did not incorporate architecture-centric principles and practices. We have taken an empirical approach to gain and disseminate an understanding of the challenges of and solutions for combining agile and architecture-centric approaches [42,72,73]. We have reached several conclusions about combining agile and architectural approaches, and some of those findings have been summarized in this chapter to provide a reader with an appropriate context from which to read and benefit from the rest of the chapters in this book. Table 1.2 represents our understanding of placing some of the well-known agile practices along with architectural practices to show that many of the agile practices have equivalent principles or practices in architecture disciplines, and these can easily be tailored and applied in agile settings.

One key observation from our ongoing research on agile and architecture has been reported in [73]. According to that observation, there is an increased emphasis on the vital role of and responsibilities of software architects in successfully combining agile and architecture methodologies. Software architects are expected to act as facilitators in whole software development projects and as the representatives of a system’s overall quality attributes. From our other research [72], we have identified some of the key types of tasks an architect in an agile environment is expected to perform in order to successfully combine architecturally savvy principles and practices (outlined in previous sections) and ASD methods.

* An architect should have a good understanding of agile approaches.
* An architect should know how to sell a key design decision to product owners in conflicting situations.
* Α project architect should know the overall architecture, required features, and implementation status.
* An architect should document and communicate the architecture to all the stakeholders.
* An architect should be willing to wear multiple architectural hats—solution architect, software architect, and implementation architect—or should be able convince his/her organization to have different architectural roles established depending on the nature of a project.
* An architect should spearhead an effort to institutionalize the role of architects as facilitators and service providers in projects.

|  |  |
| --- | --- |
| Table 1.2 Placing Agile and Architectural Practices with Each Other | |
| Some of the ASD  Practices | Frequency of Use |
| Sprint Iterative nature of general model of software architecture (SA) design with backlogs of architectural concerns to be addressed  Sprint planning Prioritizing architecturally significant requirements for each  iteration  Sprint review Architectural review  Daily meetings Sharing architecture rationale and knowledge in architecture group meetings  Onsite customer Involvement of key stakeholder in as many phases of architecting lifecycle as possible  Continuous Architecture-level integration and interoperability—quality integration attribute approaches  Refactoring Architecture-level refactoring using patterns and architectural  styles  Metaphor SA design and architecturally savvy personas  Simple design Pattern-based design to keep the design simple and well known  Collective code Buy-in of stakeholders on key architectural design decisions ownership  Coding standards Architectural templates and standards to support common goals and standards  Test-driven Architecture-based testing development | |

It is important to keep in mind that software architects usually design architecture, but it is developers who materialize the designed architecture. Hence, software developers should be equally responsible for treating SA as a first-class entity that provides the blueprint of the whole system. That is why we have argued that the role of software developers is equally important in successfully combining agile and architecture approaches; a development team must decide how to use various architectural artifacts and documents. However, there is little knowledge about how ASD teams perceive and use SA. This knowledge should be considered important because if an ASD team considers SA relevant to their tasks, there would not be much effort required to convince them to apply the architectural principles and practices that can be relevant to their project and context.

Falessi et al. [74] reported that agile developers had positive views of SA because Agile developers used architectural artifacts for communication among team members, provided input on subsequent design decisions, documented design assumptions, and evaluated design alternatives, to name a few. Falessi and colleagues’ findings were consistent to what we had found from the study reported in [72]. Other recently proposed solutions for combining architecture-centric and agile approaches include the Responsibility-Driven Architecture (RDA) approach

## 1.4 Making Architectural and Agile Approaches Work

presented by Blair and his colleague [75]. Their approach exploits the concepts of real option theory using a spreadsheet-based simple tool. Faber presented an approach to help architecture and software development teams focus on NFRs that are usually ignored in most ASD methods [76]. According to his approach, an architect takes ownership and responsibility for representing NFRs at all stages of the software development lifecycle. Madison [77] has described the architect as a linchpin who can tie up agile and architecture approaches, and he strongly advocates the use of agile for getting to a good architecture by suitably combining architectural functions (such as communication, quality attributes, and design patterns) and architectural skills at four points (i.e., upfront planning, storyboarding, sprint, and working software).

Several solutions for combining agile and architecture methods are detailed in this book. The first part of this book, Fundamentals of Agile Architecting, has several chapters that report approaches and techniques for combining some architectural issues and approaches with ASD methods. The problems addressed in that part include architecture level refactoring, design decision making in agile methods, and leveraging personas to elicit, specify, and validate ASRs. For example, Michael Stal has shown how to refactor at the architecture level by exploiting knowledge about architectural styles and patterns and ASRs. Stal states that a systematic architectural refactoring can help prevent architectural erosion by evaluating the existing software design before adding new artifacts or changing existing ones. Van der Ven and Bosch have proposed an approach to improving architecture design decision making in agile settings. The findings from their case study provide practical insights for agile architecture design decision making. Cleland-Huang and her colleagues present a persona-based approach to eliciting and addressing ASRs. They provide several concrete examples to show how to use ASPs for deriving architecture design and evaluation.

The second part of this book, Managing Architecture in Agile Projects, includes chapters that provide methods, approaches, and tools for addressing important problems, such as variability management, knowledge management, and architecture evaluation, in projects using ASD methods. Several of the architectural principles and practices presented in the first part of this chapter have been leveraged in the approaches presented in the second part of the book for support of agile architecting (e.g., variability management in agile projects, architectural knowledge management in Scrum projects, and incremental architecture evaluation).

The third part of this book, Agile Architecting in Specific Domains, includes solutions that combine agility and architecture for new and emerging technological solutions, such as cloud computing. Testing cloud-based applications and designing and deploying multi-tenancy applications are significantly complicated and onerous activities and there is relatively little knowledge about how to effectively and efficiently perform them in an agile and lean manner. These chapters propose agile ways to design and analyze multi-tenancy applications and to test them by leveraging architecture-centric principles and artifacts. The presented approaches are good examples of agile architecting.

Having read and understood the theoretical principles and approaches that should underpin agile architecting and having seen their applications for different systems in various domains, A reader of this book will likely enjoy reading some real-world examples of agile architecting to learn from industrial efforts in this area. The fourth part of this book includes four chapters that have been written based on several industrial projects that make architecture and agile work together in their respective environments. For example, Hopkins and Harcombe discuss the factors that need to be considered when planning the delivery of large-scale agile projects; architecture planning is a centerpiece of advice that they provide to combine agile approaches and architecture for large-scale software development projects. Eeles’s work focuses on sharing experiences of designing and evolving “change-ready” systems by leveraging agile and architectural principles. Friedrichsen brings up the role of well-defined architectural principles for supporting continuous refactoring as a means of emergent software design championed by agile followers. The last set of industrial insights comes from the tale of evolving a complex software platform combining agile practices and architectural principles. The industrial chapter provides useful information and insights about drawing a compromise between the purity of agile approaches and practical business concerns that need significant attention paid to architectural role and integrating.

It is argued that there is an important and urgent need to understand the importance, opportunities, and challenges involved in making architecture-centric and agile approaches for developing software intensive systems and services work together. One of our key goals is to build an evidence-based body of knowledge by identifying and understanding the main points of clashes when combining agile and architecture and how those clashes can be turned into advantages based on a project’s needs and context. This book offers several views and approaches aimed at helping companies and individuals learn and apply appropriate methods, strategies, and tools to make architecture and agile approaches work together for agile architecting. We hope readers (both researchers and practitioners) not only find the approaches presented in this book useful and applicable but also share their experiences of combining agile and architecture by publishing the failure and success stories in order to contribute to the growing body of knowledge on this topic that is hugely important to the software development community.

# Acknowledgments

Some of the most significant contributions to my understanding and writings on this topic were made by Professor Philippe Kruchten and Professor Pekka Abrahamsson through sharing their writings, ideas, and experiences. Some of the ideas presented in this chapter came through my collaboration with Minna Pikkarainen and Toumas Ihme of VTT, Finland. This chapter also benefited from the knowledge that I gained from the articles submitted to our call to a special issue of IEEE Software back in 2009/2010. Kieran Conboy also helped in collecting the data. Some content is based on my research in the FLEXI ITEA2 project. I also acknowledge the generosity of my co-editors of this book for allowing me to author this chapter.

## References

# References

1. [Schwaber K. Agile project management with scrum. Washington, USA: Microsoft Press; 2004.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0010)
2. [Palmer SR, Felsing JM. A practical guide to feature-driven development. USA: Prentice Hall; 2002.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0015)
3. [Beck K. Extreme programming explained: embrace change. Reading, MA: Addison Wesley Longman, Inc.; 2000.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0020)
4. [Nord RL, Tomayko JE. Software architecture-centric methods and agile development. IEEE Softw 2006;23:47–53.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0025)
5. [Parsons R. Architecture and agile methodologies—how to get along. In: WICSA; 2008.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0030)
6. [Ihme T, Abrahamsson P. Agile architecting: the use of architectural patterns in mobile java applications. Int J Agile Manufacturing 2005;8:1–16.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0035)
7. [Bass L, Clements P, Kazman R. Software architecture in practice. 2nd ed. Boston, MA: Addison-Wesley; 2003.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0040)
8. [Kruchten P. Situated agility. In: 9th International conference on agile processes and eXtreme programming in software engineering, Limerick, Ireland; 2008.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0045)
9. [Boehm B. Get ready for agile methods, with care. IEEE Computer 2002;35:64–9.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0050)
10. [Hofmeister C, Kruchten P, Nord RL, Obbink H, Ran A, America P. A general model of software architecture design derived from five industrial approaches. J Sys Softw 2007;80:106–26.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0055)
11. [Thapparambil P. Agile architecture: pattern or oxymoron? Agile Times 2005;6:43–8.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0060)
12. [Ali Babar M, Abrahamsson P. Architecture-centric methods and agile approaches. In: Proceedings of the 9th international conference on agile processes and eXtreme programming in software engineering Limerick, Ireland; 2008. p. 242–3.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0065)
13. [Kruchten P. Voyage in the agile memeplex. ACM queue 2007; July/August:38–44.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0070)
14. [Lycett M, Macredie RD, Patel C, Paul RJ. Migrating agile methods to standardized development practice. IEEE Comput 2003;36:79–85.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0075)
15. [Boehm B, Taylor R. Management challenges to implementing agile processes in traditional development organizations. IEEE Softw 2005;22:30–9.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0080)
16. [Parnas DL. On the criteria to be used in decomposing systems into modules. Commun ACM 1972;15:1053–8.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0085)
17. [Stevens WP, Myers GJ, Constantine LL. Structured design. IBM Sys J 1974;13:115–39.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0090)
18. [Gamma E, Helm R, Johnson R, Vlissides J. Design patterns-elements of reusable objectoriented software. Reading, MA: Addison-Wesley; 1995.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0095)
19. [Garlan D, Shaw M. An introduction to software architecture. Pennsylvania: SEI, Carnegie Mellon University; 1994.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0100)
20. [Kazman R, Barbacci M, Klein M, Carriere SJ. Experience with performing architecture tradoff analysis. In: Proceedings of the 21th international conference on software engineering. New York, USA: ACM Press; 1999. p. 54–63.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0105)
21. [Gorton I. Essential software architecture. Heidelberg: Springer; 2006.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0110)
22. [Perry DE, Wolf AL. Foundations for the study of software architecture. ACM SIGSOFT 1992;17:40–52.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0115)
23. [Ali Babar M, Dingsoyr T, Lago P, Van Vliet H. Software architecture knowledge management: theory and practice. Heidelberg: Springer-Verlag; 2009.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0120)
24. [Bass L, Clements P, Kazman R. Software architecture in practice. 3rd ed. Massachusetts: Addison-Wesley; 2013.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0125)
25. [America P, Rommes E, Obbink H. Multi-view variation modeling for scenario analysis. In: Proceedings of fifth international workshop on product family engineering. Siena, Italy: Springer-Verlag; 2003.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0130)
26. [Kruchten P. Architectural blueprints—the “4](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0135)þ[1” view model of software architecture. IEEE Softw 1995;12:42–50.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0135)
27. [Hofmeister C, Nord RL, Soni D. Applied software architecture. Reading, MA: AddisonWesley; 2000.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0140)
28. Ran A. ARES conceptual framework for software architecture. In: Jazayeri M, Ran A, Linden FVD, editors. Software architecture for product families: principles and practice. Boston, MA: Addison-Wesley; 2000.
29. [Hofmeister C, Kruchten P, Nord RL, Obbink H, Ran A, America P. A general model of software architecture design derived from five industrial approaches. J Sys Softw 2006;80:106–26.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0145)
30. [Tang A, Avgeriou P, Jansen A, Capilla R, Babar MA. A comparative study of architecture knowledge management tools. J Sys Softw 2010;83:352–70.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0150)
31. [Tang A, Avgeriou P, Jansen A, Capilla R, Babar MA. A comparative study of architecture knowledge management tools. J Sys Softw 2010;83(3):352–70.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0155)
32. [Chung L, Nixon BA, Yu E, Mylopoulos J. Non-functional requirements in software engineering. Boston: Kluwer Academic Publishing; 1999.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0160)
33. [Chen L, Babar MA, Nuseibeh B. Characterizing architecturally significant requirements. IEEE Softw 2013;30:38–45.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0165)
34. [Bass L, Klein M, Bachmann F. Quality attribute design primitives and the attribute driven design method. In: The 4th international workshop on software product-family engineering; 2002.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0170)
35. [Bruin HD, Vliet HV. Quality-driven software architecture composition. J Sys Softw 2003;66:269–84.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0175)
36. [Barbacci M, Klein MH, Longstaff TA, Weinstock CB. Quality attributes. Pennsylvania: Software Engineering Institute, Carnegie Mellon University; 1995.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0180)
37. [Cleland-Huang J, Czauderna A, Keenan E. A persona-based approach for exploring architecturally significant requirements in agile projects. In: 19th International working conference requirements engineering: foundation for software quality (REFSQ). Essen, Germany: Springer; 2013.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0185)
38. [Lassing N, Rijsenbrij D, Vliet HV. How well can we predict changes at architecture design time? J Sys Softw 2003;65:141–53.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0190)
39. [Bosch J. Design and use of software architectures: adopting and evolving a product-line approach. Boston, MA: Addison-Wesley; 2000.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0195)
40. [Klein M, Kazman R. Attribute-based architectural styles, Tech Report CMU/SEI-99TR-022. Software Engineering Institute, Carnegie Mellon University; 1999.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0200)
41. [Al-Naeem T, Gorton I, Ali Babar M, Rabhi F, Benatallah B. A quality-driven systematic approach for architecting distributed software applications. In: Proceedings of the 27th international conference on software engineering, ICSE, St. Louis, USA; 2005. p. 244–53.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0205)
42. [Babar MA. An exploratory study of architectural practices and challenges in using agile software development approaches. In: Joint working IEEE/IFIP conference on software architecture 2009 and european conference on software architecture (WICSA/ECSA). Cambridge, UK: IEEE Computer Society; 2009.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0210)
43. [Babar MA, Ihme T, Pikkarainen M. An industrial case of exploiting product line architectures in agile software development. In: 13th international conference on software product lines (SPLC) San Francisco, California, USA; 2009.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0215)

## References

1. [Clements P, Bachmann F, Bass L, Garlan D, Ivers j, Little R, et al. Documenting software architectures: views and beyond. USA: Addison-Wesley; 2002.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0220)
2. IEEE. IEEE recommended practices for architecture description of software-intensive systems. In: IEEE Std 1471-2000; 2000.
3. [Luckham DC, Kenney JJ, Augustin LM, Vera J, Bryan D, Mann W. Specification](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0225) and analysis of software architecture using Rapide. IEEE Trans Softw En[g](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0225)

[1995;21:336–55.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0225)

1. [Shaw M, DeLine R, Klein D, Ross T, Young D, Zelesnik G. Abstractions for software architecture and tools to support them. IEEE Trans Softw Eng 1995;21:314–35.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0230)
2. [Clements PC. A survey of architecture description languages. In: Proceedings of the 8th international workshop on software specification and design. Germany; 1996.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0235)
3. [Medvidovic N, Taylor RN. A classification and comparison framework for software architecture description languages. IEEE Trans Softw Eng 2000;26:70–93.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0240)
4. [Fowler M. UML distilled: A Brief Guide to the Standard Object Modelling Language. 3rd ed. Boston, MA: Addison-Wesley;](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0245) [2004.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0245)
5. [Lassing N, Rijsenbrij D, Hv Vliet. The goal of software architecture analysis: confidence building or risk assessment. In: Proceedings of first BeNeLux conference on software, architecture; 1999.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0250)
6. [Maranzano JF, Rozsypal SA, Zimmerman GH, Warnken GW, Wirth PE, Weiss DM. Architecture reviews: practice and experience. IEEE Softw 2005;22:34–43.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0255)
7. [Babar MA, Zhu L, Jeffery D. A framework for classifying and comparing software architecture evaluation methods. In: Australian software engineering conference (ASWEC). Melbourne, Australian; 2004.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0260)
8. [Kazman R, Bass L, Abowd G, Webb M. SAAM: a method for analyzing the properties of software architectures. In: Proceedings of the 16th international conference on software, engineering; 1994. p. 81–90.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0265)
9. [Clements P, Kazman R, Klein M. Evaluating software architectures: methods and case studies. Boston, MA: Addison-Wesley; 2002.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0270)
10. [Bengtsson P, Lassing N, Bosch J, van Vliet H. Architecture-level modifiability analysis (ALMA). J Sys Softw 2004;69:129–47.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0275)
11. [Williams LG, Smith CU. PASA: an architectural approach to fixing software performance problems. In: Proceedings of the international conference of the Computer Measurement Group, Reno, USA; 2002.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0280)
12. [Lassing N, Rijsenbrij D, Hv Vliet. On software architecture analysis of flexibility, complexity of changes: size isn’t everything. In: Second nordic software architecture workshop (NOSA ’99); 1999.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0285)
13. [Molter G. Integrating SAAM in domain-centric and reuse-based development processes. In: Second nordic workshop software architecture (NOSA ’99); 1999.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0290)
14. [Lung C, Bot S, Kalaichelvan K, Kazman R. An approach to software architecture analysis for evolution and reusability. In: CASCON ’97, Toronto, Canada; 1997.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0295)
15. [Bengtsson PO, Bosch J. Scenario-based architecture reengineering. In: Fifth international conference on software reuse (ICSR5), Victoria, Canada; 1998.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0300)
16. [Bengtsson PO, Bosch J. Architecture level prediction of software maintenance. In: Third European conference on software maintenance and reengineering, Amsterdam, Netherlands; 1999.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0305)
17. [Smith CU, Williams LG. Performance solutions: a practical guide to creating responsive, scalable software. Boston, MA: Addison-Wesley; 2002.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0310)
18. [Cockburn A. Crystal clear: a human-powered methodology for small teams. Boston, USA: Addison-Wesley; 2004.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0315)
19. [Abrahamsson P, Salo O, Ronkainen J, Warsta J. Agile software development methods: review and analysis. Oulu, Finland: VTT Publications 478; 2002.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0320)
20. [Cohen D, Lindvall M, Costa P. An introduction to agile methods. Adv Comput](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0325)

[2004;61:1–66.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0325)

1. [Schwaber K, Beedle M. Agile software development with scrum. Upper Saddle River, NJ: Prentice-Hall; 2002.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0330)
2. [Ambler SW. Agile modeling: effective practices for eXreme programming and the unified process. New York: John Wiley & Sons; 2002.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0335)
3. [IBM. RUP for extreme programming (xp) plug-ins. IBM; 2004.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0340)
4. [Larman C. Agile and iterative development: a manager’s guide. Boston, MA: Addison Wesley Professional; 2003.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0345)
5. [Beck K, Andres C. Extreme programming explained: embrace change. 2nd ed. Reading, MA.: Addison Wesley Longman; 2004.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0350)
6. [Ali Babar M, Iheme T, Pikkarainen M. An industrial case of exploiting product line architectures in agile software development. In: Proceedings of the 13th international software product line conference (SPLC), San Francisco, USA; 2009.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0355)
7. [Abrahamsson P, Babar MA, Kruchten P. Agility and architecture: can they coexist? IEEE Softw 2010;27:16–22.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0360)
8. [Falessi D, Cantone G, Sarcia`](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0365) [SA, Calavaro G, Subiaco P, D’Amore C. Peaceful coexistence: agile developer perspectives on software architecture. IEEE Softw 2010;27:23–5.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0365) [75] [Blair S, Watt R, Cull T. Responsibility-driven architecture. IEEE Softw 2010;27:26–32.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0370)
9. [Faber R. Architects as service providers. IEEE Softw 2010;27:33–40.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0375)
10. [Madison J. Agile architecture interactions. IEEE Softw 2010;27:41–8.](http://refhub.elsevier.com/B978-0-12-407772-0.00001-0/rf0380)

1. .1 INTRODUCTION

   The Agile software development (ASD) paradigm has been widely adopted by hundreds of large and small companies in an effort to reduce costs and increase their ability to handle changes in dynamic market conditions. Based on the principles [↑](#footnote-ref-1)
2. <http://agilemanifesto.org/> [↑](#footnote-ref-2)