Agile Software Architecture

Agile Software Architecture

Aligning Agile Processes and Software Architectures

Edited by

Muhammad Ali Babar

Alan W. Brown

Ivan Mistrik

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Morgan Kaufmann is an imprint of Elsevier

Acquiring Editor: *Todd Green*
Editorial Project Manager: *Lindsay Lawrence*
Project Manager: *Punithavathy Govindaradjane*
Designer: *Maria Inês Cruz*

Morgan Kaufmann is an imprint of Elsevier
225 Wyman Street, Waltham, MA 02451, USA

Notices
Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods or professional practices, may become necessary. Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information or methods described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data
Agile software architecture : aligning agile processes and software architectures / edited by Muhammad Ali Babar, Alan W. Brown, Ivan Mistrik.
pages cm
Includes bibliographical references and index.
ISBN 978-0-12-407772-0 (pbk.)
1. Agile software development. 2. Software architecture. I. Ali Babar, Muhammad. II. Brown, Alan W., 1962- III. Mistrik, Ivan.
QA76.76.D47A3844 2013
005.1'2–dc23

2013040761

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

ISBN: 978-0-12-407772-0

This book has been manufactured using Print On Demand technology. Each copy is produced to order and is limited to black ink. The online version of this book will show color figures where appropriate.

For information on all MK publications visit our website at *www.mkp.com*

Contents

PART 3 AGILE ARCHITECTING IN SPECIFIC DOMAINS

Acknowledgments

The editors would like to acknowledge the significant effort Kai Koskimies made during different phases of this book's editing phases. Judith Stafford also helped in framing the initial proposal for this book. We also sincerely thank many authors who contributed their works to this book. The international team of anonymous reviewers gave detailed feedback on early versions of chapters and helped us to improve both the presentation and accessibility of the work. Ali Babar worked on this project while based at Lancaster University UK and IT University of Copenhagen, Denmark. Finally, we would like to thank the Elsevier management and editorial teams, in particular to Todd Green and Lindsay Lawrence, for the opportunity to produce this unique collection of articles covering the wide range of areas related to aligning agile processes and software architectures.

About the Editors

MUHAMMED ALI BABAR

Dr. M. Ali Babar is a Professor of Software Engineering (Chair) at the School of Computer Science, the University of Adelaide, Australia. He also holds an Associate Professorship at IT University of Copenhagen, Denmark. Prior to this, he was a Reader in Software Engineering at Lancaster University UK. Previously, he worked as a researcher and project leader in different research centers in Ireland and Australia. He has authored/co-authored more than 140 peer-reviewed research papers in journals, conferences, and workshops. He has co-edited a book, *Software Architecture Knowledge Management: Theory and Practice*. Prof. Ali Babar has been a guest editor of several special issues/sections of *IEEE Software, JSS, ESEJ, SoSyM, IST,* and *REJ*. Apart from being on the program committees of several international conferences such as WICSA/ECSA, ESEM, SPLC, ICGSE, and ICSSP for several years, Prof. Ali Babar was the founding general chair of the Nordic-Baltic Symposium on Cloud Computing and Internet Technologies (NordiCloud) 2012. He has also been co-(chair) of the program committees of several conferences such as NordiCloud 2013, WICSA/ECSA 2012, ECSA2010, PROFES2010, and ICGSE2011. He is a member of steering committees of WICSA, ECSA, NordiCloud and ICGSE. He has presented tutorials in the areas of cloud computing, software architecture and empirical approaches at various international conferences. Prior to joining R&D field, he worked as a software engineer and an IT consultant for several years in Australia. He obtained a PhD in computer science and engineering from University of New South Wales, Australia.

ALAN W. BROWN

Alan W. Brown is Professor of Entrepreneurship and Innovation in the Surrey Business School, University of Surrey, UK. where he leads activities in the area of corporate entrepreneurship and open innovation models. In addition to teaching activities, he focuses on innovation in a number of practical research areas with regard to global enterprise software delivery, agile software supply chains, and the investigation of "open commercial" software delivery models. He has formerly held a wide range of roles in industry, including Distinguished Engineer and CTO at IBM Rational, VP of Research at Sterling Software, Research Manager at Texas Instruments Software, and Head of Business Development in a Silicon Valley startup. In these roles Alan has worked with teams around the world on software engineering strategy, process improvement, and the transition to agile delivery approaches. He has published over 50 papers and written four books. He holds a Ph.D. in Computing Science from the University of Newcastle upon Tyne, UK.

IVAN MISTRIK

Ivan Mistrik is a computer scientist who is interested in system and software engineering (SE/SWE) and in system and software architecture (SA/SWA); in particular, he is interested in life cycle system/software engineering, requirements engineering, relating software requirements and architectures, knowledge management in software development, rationale-based software development, aligning enterprise/system/software architectures, and collaborative system/software engineering. He has more than forty years' experience in the field of computer systems engineering as an information systems developer, R&D leader, SE/SA research analyst, educator in computer sciences, and ICT management consultant. In the past 40 years, he has worked primarily at various R&D institutions and has consulted on a variety of large international projects sponsored by ESA, EU, NASA, NATO, and UN. He has also taught university-level computer sciences courses in software engineering, software architecture, distributed information systems, and human-computer interaction. He is the author or co-author of more than 80 articles and papers that have been published in international journals and books and presented at international conferences and workshops; most recently, he wrote the chapter "Capture of Software Requirements and Rationale through Collaborative Software Development" in the book *Requirements Engineering for Sociotechnical Systems*, the paper "Knowledge Management in the Global Software Engineering Environment," and the paper "Architectural Knowledge Management in Global Software Development." He has also written over 90 technical reports and presented over 70 scientific/technical talks. He has served on many program committees and panels of reputable international conferences and organized a number of scientific workshops, most recently two workshops on Knowledge Engineering in Global Software Development at the International Conference on Global Software Engineering 2009 and 2010. He has been a guest editor of *IEE Proceedings Software: A Special Issue on Relating Software Requirements and Architectures,* published by IEE in 2005. He has also been lead editor of the book *Rationale Management in Software Engineering*, published in 2006; the book *Collaborative Software Engineering,* published in 2010; and the book *Relating Software Requirements and Architectures,* published in 2011. He has also co-authored the book *Rationale-Based Software Engineering*, published in May 2008. He is a lead editor of the *Expert Systems Special Issue on Knowledge Engineering in Global Software Development* to be published in 2012, and he has organized the IEEE International Workshop on the Future of Software Engineering for/in the Cloud (FoSEC) that was held in conjunction with IEEE Cloud 2011. He was a guest editor of the *Journal of Systems and Software Special Issue on the Future of Software Engineering for/in the Cloud* in 2013 and a lead editor of the book on *Aligning Enterprise, System, and Software Architectures* to be published in 2012.

List of Contributors

Sarah Al-Azzani
University of Birmingham, Birmingham, UK

Ahmad Al-Natour
University of Birmingham, Birmingham, UK

Paris Avgeriou
University of Groningen, Groningen, The Netherlands

Muhammad Ali Babar
The University of Adelaide, Adelaide, SA, Australia

Rami Bahsoon
University of Birmingham, Birmingham, UK

Kawtar Benghazi
Universidad de Granada, Granada, Spain

Jan Bosch
Chalmers University of Technology, Gothenburg, Sweden

Georg Buchgeher
Software Competence Center Hagenberg (SCCH), Hagenberg, Austria

Lawrence Chung
University of Texas at Dallas, Richardson, TX, USA

James O. Coplien
Gertrud & Cope, Espergærde, Denmark

Jane Cleland-Huang
DePaul University, Chicago, IL, USA

Adam Czauderna
DePaul University, Chicago, IL, USA

Jessica Díaz
Universidad Politécnica de Madrid (Technical U. of Madrid), Madrid, Spain

Peter Eeles
IBM, London, UK

Veli-Pekka Eloranta
Tampere University of Technology, Tampere, Finland

Uwe Friedrichsen
Codecentric AG, Solingen, Germany

Matthias Galster
University of Canterbury, Christchurch, New Zealand

Juan Garbajosa
Universidad Politécnica de Madrid (Technical U. of Madrid), Madrid, Spain

Stephen Harcombe
Northwich, Cheshire, UK

Richard Hopkins
IBM, Cleveland, UK

Ben Isotta-Riches
Aviva, Norwich, UK

Kai Koskimies
Tampere University of Technology, Tampere, Finland

José Luis Garrido
Universidad de Granada, Granada, Spain

Mehdi Mirakhorli
DePaul University, Chicago, IL, USA

Manuel Noguera
Universidad de Granada, Granada, Spain

Jennifer Pérez
Universidad Politécnica de Madrid (Technical U. of Madrid), Madrid, Spain

Janet Randell
Aviva, Norwich, UK

Trygve Reenskaug
University of Oslo, Oslo, Norway

Antonio Rico
Universidad de Granada, Granada, Spain

Jan Salvador van der Ven
Factlink, Groningen, The Netherlands

Michael Stal
Siemens AG, Corporate Research & Technology, Munich, Germany

Rainer Weinreich
Johannes Kepler University Linz, Linz, Austria

Agustín Yagüe
Universidad Politécnica de Madrid (Technical U. of Madrid), Madrid, Spain

Foreword by John Grundy

Architecture vs Agile: competition or cooperation?

Until recently, conventional wisdom has held that software architecture design and agile development methods are somehow "incompatible," or at least they generally work at cross-purposes [1]. Software architecture design has usually been seen by many in the agile community as a prime example of the major agile anti-pattern of "big design up front." On the other hand, agile methods have been seen by many of those focusing on the discipline of software architecture as lacking sufficient forethought, rigor, and far too dependent on "emergent" architectures (a suitable one of which may never actually emerge). In my view, there is both a degree of truth and a substantial amount of falsehood in these somewhat extreme viewpoints. Hence, the time seems ripe for a book exploring leading research and practice in an emerging field of "agile software architecture," and charting a path for incorporating the best of both worlds in our engineering of complex software systems.

In this foreword, I briefly sketch the background of each approach and the anti-agile, anti-software architecture viewpoints of both camps, as they seem to have become known. I deliberately do this in a provocative and all-or-nothing way, mainly to set the scene for the variety of very sensible, balanced approaches contained in this book. I hope to seed in the reader's mind both the traditional motivation of each approach and how these viewpoints of two either-or, mutually exclusive approaches to complex software systems engineering came about. I do hope that it is apparent that I myself believe in the real benefits of both approaches and that they are certainly in no way incompatible; agile software architecting—or architecting for agile, if you prefer that viewpoint—is both a viable concept and arguably the way to approach the current practice of software engineering.

SOFTWARE ARCHITECTURE—THE "TRADITIONAL" VIEW

The concept of "software architecture"—both from a theoretical viewpoint as a means of capturing key software system structural characteristics [2] and practical techniques to develop and describe [3, 4]—emerged in the early to mid-1980s in response to the growing complexity and diversity of software systems. Practitioners and researchers knew implicitly that the concept of a "software architecture" existed in all but the most trivial systems. Software architecture incorporated elements including, but not limited to, human machine interfaces, databases, servers, networks, machines, a variety of element interconnections, many diverse element properties, and a variety of further structural and behavioral subdivisions (thread

management, proxies, synchronization, concurrency, real-time support, replication, redundancy, security enforcement, etc.). Describing and reasoning about these elements of a system became increasingly important in order to engineer effective solutions, with special purpose "architecture description languages" and a wide variety of architecture modeling profiles for the Unified Modeling Language (UML). Software architecting includes defining an architecture from various perspectives and levels of abstraction, reasoning about the architecture's various properties, ensuring the architecture is realizable by a suitable implementation which will meet system requirements, and evolving and integrating complex architectures.

A number of reusable "architecture patterns" [3] have emerged, some addressing quite detailed concerns (e.g., concurrency management in complex systems), with others addressing much larger-scale organizational concerns (e.g., multitier architectures). This allowed a body of knowledge around software architecture to emerge, allowing practitioners to leverage best-practice solutions for common problems and researchers to study both the qualities of systems in use and to look for improvements in software architectures and architecture engineering processes.

The position of "software architecting" in the software development lifecycle was (and still is) somewhat more challenging to define. Architecture describes the solution space of a system and therefore traditionally is thought of as an early part of the design phase [3, 4]. Much work has gone into developing processes to support architecting complex systems, modeling architectures, and refining and linking architectural elements into detailed designs and implementations. Typically, one would identify and capture requirements, both functional and nonfunctional, and then attempt to define a software architecture that meets these requirements.

However, as all practitioners know, this is far easier said than done for many real-world systems. Different architectural solutions themselves come with many constraints—which requirements can be met and how they can be met, particularly nonfunctional requirements, are important questions. Over-constrained requirements may easily describe a system that has no suitable architectural realization. Many software applications are in fact "systems of systems" with substantive parts of the application already existent and incorporating complex, existent software architecture that must be incorporated. In addition, architectural decisions heavily influence requirements, and coevolution of requirements and architecture is becoming a common approach [5]. Hence, software architectural development as a top-down process is under considerable question.

AGILE METHODS—THE "TRADITIONAL" VIEW

The focus in the 1980s and 90s on extensive up-front design of complex systems, development of complex modeling tools and processes, and focus on large investment in architectural definition (among other software artifacts) were seen by many to have some severe disadvantages [6]. Some of the major ones identified included

over-investment in design and wasted investment in over-engineering solutions, inability to incorporate poorly defined and/or rapidly changing requirements, inability to change architectures and implementations if they proved unsuitable, and lack of a human focus (both customer and practitioner) in development processes and methods. In response, a variety of "agile methods" were developed and became highly popular in the early to mid- 2000s. One of my favorites and one that I think exemplifies the type is Kent Beck's eXtreme Programming (XP) [7].

XP is one of many agile methods that attempt to address these problems all the way from underlying philosophy to pragmatic deployed techniques. Teams comprise both customers and software practitioners. Generalist roles are favored over specialization. Frequent iterations deliver usable software to customers, ensuring rapid feedback and continuous value delivery. Requirements are sourced from focused user stories, and a backlog and planning game prioritizes requirements, tolerating rapid evolution and maximizing value of development effort. Test-driven development ensures requirements are made tangible and precise via executable tests. In each iteration, enough work is done to pass these tests but no more, avoiding over-engineering. Supporting practices, including 40-hour weeks, pair programming, and customer-on-site avoid developer burnout, support risk mitigation and shared ownership, and facilitate human-centric knowledge transfer.

A number of agile approaches to the development of a "software architecture" exist, though most treat architecture as an "emergent" characteristic of systems. Rather than the harshly criticized "big design up front" architecting approaches of other methodologies, spikes and refactoring are used to test potential solutions and continuously refine architectural elements in a more bottom-up way. Architectural spikes in particular give a mechanism for identifying architectural deficiencies and experimenting with practical solutions. Refactoring, whether small-scale or larger-scale, is incorporated into iterations to counter "bad smells,"—which include architectural-related problems including performance, reliability, maintainability, portability, and understandability. These are almost always tackled on a need-to basis, rather than explicitly as an up-front, forward-looking investment (though they of course may bring such advantages).

SOFTWARE ARCHITECTURE—STRENGTHS AND WEAKNESSES WITH REGARD TO AGILITY

Up-front software architecting of complex systems has a number of key advantages [8]. Very complex systems typically have very complex architectures, many components of which may be "fixed" as they come from third party systems incorporated into the new whole. Understanding and validating a challenging set of requirements may necessitate modeling and reasoning with a variety of architectural solutions, many of which may be infeasible due to highly constrained requirements. Some requirements may need to be traded off against others to even make the overall

system feasible. It has been found in many situations to be much better to do this in advance of a large code base and complex architectural solution to try and refactor [8]. It is much easier to scope resourcing and costing of systems when a software architecture that documents key components exists upfront. This includes costing nonsoftware components (networks, hardware), as well as necessary third party software licenses, configuration, and maintenance.

A major criticism of upfront architecting is the potential for over-engineering and thus over-investment in capacity that may never be used. In fact, a similar criticism could be leveled in that it all too often results in an under-scoped architecture and thus under-investing in required infrastructure, one of the major drivers in the move to elastic and pay-as-you-go cloud computing [9]. Another major criticism is the inability to adapt to potentially large requirements changes as customers reprioritize their requirements as they gain experience with parts of the delivered system [6]. Upfront design implies at least some broad requirements—functional and nonfunctional—that are consistent across the project lifespan. The relationship between requirements and software architecture has indeed become one of mutual influence and evolution [5].

AGILE—STRENGTHS AND WEAKNESSES WITH REGARD TO SOFTWARE ARCHITECTURE

A big plus of agile methods is their inherent tolerance—and, in fact, encouragement—of highly iterative, changeable requirements, focusing on delivering working, valuable software for customers. Almost all impediments to requirements change are removed; in fact, many agile project-planning methods explicitly encourage reconsideration of requirements and priorities at each iteration review—the mostly widely known and practiced being SCRUM [10]. Architectural characteristics of the system can be explored using spikes and parts found wanting refactored appropriately. Minimizing architectural changes by focusing on test-driven development—incorporating appropriate tests for performance, scaling, and reliability—goes a long way to avoiding redundant, poorly fitting, and costly over-engineered solutions.

While every system has a software architecture, whether designed-in or emergent, experience has shown that achieving a suitably complex software architecture for large-scale systems is challenging with agile methods. The divide-and-conquer approach used by most agile methods works reasonably well for small and some medium-sized systems with simple architectures. It is much more problematic for large-scale system architectures and for systems incorporating existent (and possibly evolving!) software architectures [8]. Test-driven development can be very challenging when software really needs to exist in order to be able to define and formulate appropriate tests for nonfunctional requirements. Spikes and refactoring support small-system agile architecting but struggle to scale to large-scale or even medium-scale architecture evolution. Some projects even find iteration sequences

become one whole refactoring exercise after another, in order to try and massively reengineer a system whose emergent architecture has become untenable.

BRINGING THE TWO TOGETHER—AGILE ARCHITECTING OR ARCHITECTING FOR AGILE?

Is there a middle ground? Can agile techniques sensibly incorporate appropriate levels of software architecture exploration, definition, and reasoning, before extensive code bases using an inappropriate architecture are developed? Can software architecture definition become more "agile," deferring some or even most work until requirements are clarified as develop unfolds? Do some systems best benefit from some form of big design up front architecting but can then adopt more agile approaches using this architecture? On the face of it, some of these seem counter-intuitive and certainly go against the concepts of most agile methods and software architecture design methods.

However, I think there is much to be gained by leveraging strengths from each approach to mitigate the discovered weaknesses in the other. Incorporating software architecture modeling, analysis, and validation in "architectural spikes" does not seem at all unreasonable. This may include fleshing out user stories that help to surface a variety of nonfunctional requirements. It may include developing a variety of tests to validate that these requirements are met. If a system incorporates substantive existing system architecture, exploring interaction with interfaces and whether the composite system meets requirements by appropriate test-driven development seems like eminently sensible early-phase, high-priority work. Incorporating software architecture-related stories as priority measures in planning games and SCRUM-based project management also seems compatible with both underlying conceptual models and practical techniques. Emerging toolsets for architecture engineering, particularly focusing on analyzing nonfunctional properties, would seem to well support and fit agile practices.

Incorporating agile principles into software architecting processes and techniques also does not seem an impossible task, whether or not the rest of a project uses agile methods. Iterative refinement of an architecture—including some form of user stories surfacing architectural requirements, defining tests based on these requirements, rapid prototyping to exercise these tests, and pair-based architecture modeling and analysis—could all draw from the demonstrated advantages of agile approaches. A similar discussion emerges when trying to identify how to leverage design patterns and agile methods, user-centered design and agile methods, and model-driven engineering and agile methods [1, 11, 12]. In each area, a number of research and practice projects are exploring how the benefits of agile methods might be brought to these more "traditional" approaches to software engineering, and how agile approaches might incorporate well-known benefits of patterns, User Centered Design (UCD), and Model Driven Engineering (MDE).

LOOKING AHEAD

Incorporating at least some rigorous software architecting techniques and tools into agile approaches appears—to me, at least—to be necessary for successfully engineering many nontrivial systems. Systems made up of architectures from diverse solutions with very stringent requirements, particularly challenging, nonfunctional ones, really need careful look-before-you-leap solutions. This is particularly so when parts of the new system or components under development may adversely impact existing systems (e.g., introduce security holes, privacy breaches, or adversely impact performance, reliability, or robustness). Applying a variety of agile techniques—and the philosophy of agile—to software architecting also seems highly worthwhile. Ultimately, the purpose of software development is to deliver high-quality, on-time, and on-budget software to customers, allowing for some sensible future enhancements. A blend of agile focus on delivery, human-centric support for customers and developers, incorporating dynamic requirements, and—where possible—avoiding over-documenting and over-engineering exercises, all seem to be of benefit to software architecture practice.

This book goes a long way toward realizing these trends of agile architecting and architecting for agile. Chapters include a focus on refactoring architectures, tailoring SCRUM to support more agile architecture practices, supporting an approach of continuous architecture analysis, and conducting architecture design within an agile process. Complementary chapters include analysis of the emergent architecture concept, driving agile practices by using architecture requirements and practices, and mitigating architecture problems found in many conventional agile practices.

Three interesting works address other topical areas of software engineering: engineering highly adaptive systems, cloud applications, and security engineering. Each of these areas has received increasing attention from the research and practice communities. In my view, all could benefit from the balanced application of software architecture engineering and agile practices described in these chapters.

I do hope that you enjoy this book as much as I enjoyed reading over the contributions. Happy agile software architecting!

John Grundy
Swinburne University of Technology,
Hawthorn, Victoria, Australia

References

[1] Nord RL, Tomayko JE. Software architecture-centric methods and agile development. IEEE Software 2006;23(2):47–53.
[2] Garlan D, Shaw M. Software architecture: perspectives on an emerging discipline. Angus & Robertson; 1996.
[3] Bass L, Clements P, Kazman R. Software architecture in practice. Angus & Robertson; 2003.

[4] Kruchten P. The 4+ 1 view model of architecture. IEEE Software 1995;12(6):42–50.
[5] Avgeriou P, Grundy J, Hall JG, Lago P, Mistrík I. Relating software requirements and architectures. Springer; 2011.
[6] Beck K, Beedle M, Bennekum van A, Cockburn A, Cunningham W, Fowler M, et al. Manifesto for agile software development, http://agilemanifesto.org/; 2001.
[7] Beck K. Embracing change with extreme programming. Computer 1999;32(10):70–7.
[8] Abrahamsson P, Babar MA, Kruchten P. Agility and architecture – can they co-exist? IEEE Software 2010;27(2):16–22.
[9] Grundy J, Kaefer G, Keong J, Liu A. Software engineering for the cloud. IEEE Software 2012;29(2):26–9.
[10] Schwaber K. Agile project management with SCRUM. O'Reily; 2009.
[11] Dybå T, Dingsøyr T. Empirical studies of agile software development: A systematic review. Inform Software Tech 2008;50(9–10):833–59.
[12] McInerney P, Maurer F. UCD in agile projects: dream team or odd couple? Interactions 2005;12(6):19–23.

Foreword by Rick Kazman

Since their first appearance over a decade ago, the various flavors of agile methods and processes have received increasing attention and adoption by the worldwide software community. So it is natural that, with this increased attention, software engineers are concerned about how agile methods fit with other engineering practices.

New software processes do not just emerge out of thin air; they evolve in response to a palpable need. In this case, the software development world was responding to a need for projects to be more responsive to their stakeholders, to be quicker to develop functionality that users care about, to show more and earlier progress in a project's lifecycle, and to be less burdened by documenting aspects of a project that would inevitably change.

Is any of this inimical to the use of architecture? I believe that the answer to this question is a clear "no." In fact, the question for a software project is not "should I do agile or architecture?" but rather questions such as "How much architecture should I do up front versus how much should I defer until the project's requirements have solidified somewhat?", "When and how should I refactor?", "How much of the architecture should I formally document, and when?", and "Should I review my architecture—and, if so, when?". I believe that there are good answers to all of these questions, and that agile methods and architecture are not just well-suited to live together but in fact critical companions for many software projects.

We often think of the early software development methods that emerged in the 1970s—such as the waterfall method—as being plan-driven and inflexible. But this inflexibility is not for nothing. Having a strong up-front plan provides for considerable predictability (as long as the requirements don't change too much) and makes it easier to coordinate large numbers of teams. Can you imagine a large construction or aerospace project without heavy up-front planning?

Agile methods and practitioners, on the other hand, often scorn planning, instead preferring teamwork, frequent face-to-face communication, flexibility, and adaptation. This enhances invention and creativity. The next Pixar hit movie will not be created by an up-front planning process—it is the result of a seemingly infinite number of tweaks and redos.

Agile processes were initially employed on small- to medium-sized projects with short time frames and enjoyed considerable early success. In the early years, agile processes were not often used for larger projects, particularly those employing distributed development. But these applications of agile methodologies are becoming increasingly common. What are we to make of this evolution? Or, to put it another way, is the agile mindset right for every project?

In my opinion, successful projects clearly need a successful blend of the two approaches. For the vast majority of nontrivial projects this is not and never should be an either/or choice. Too much up-front planning and commitment can stifle creativity and the ability to adapt to changing requirements. Too much agility can be

chaos. No one would want to fly in an aircraft with flight control software that had *not* been rigorously planned and thoroughly analyzed. Similarly, no one would want to spend 18 months planning an e-commerce web site for their latest cell phone model, or video game, or women's shoe style (all of which are guaranteed to be badly out of fashion in 18 months).

There are two activities that can add time to the project schedule: (1) Up-front design work on the architecture and up-front risk identification, planning, and resolution work, and (2) Rework due to fixing defects and addressing modification requests. Intuitively, these two activities trade off against each other.

What we all want is the sweet spot—what George Fairbanks calls "just enough architecture." This is not just a matter of doing the right amount of architecture work, but also doing it at the right time. Agile projects tend to want to evolve the architecture, as needed, in real time, whereas large software projects have traditionally favored considerable up-front analysis and planning.

And it doesn't stop at the architecture. Surely we also want "just enough" architecture documentation. So, when creating documentation, we must not simply document for the purpose of documentation. We must write with the reader in mind: If the reader doesn't need it, don't write it. But always remember that the reader may be a maintainer or other newcomer not yet on the project!

What about architecture evaluation? Does this belong in an agile project? I think so. Meeting stakeholders' important concerns is a cornerstone of agile philosophies. An architecture evaluation is a way of increasing the probability that this will actually occur. And an architecture evaluation need not be "heavyweight." These can be easily scaled down and made an organic part of development—no different than testing or code walkthroughs—that support the goals of an agile project.

So what should an architect do when creating the architecture for a large-scale agile project? Here are my thoughts:

- If you are building a large, complex system with relatively stable and well-understood requirements and/or distributed development, doing a large amount of architecture work up-front will likely pay off. On larger projects with unstable requirements, start by quickly designing a candidate architecture even if it leaves out many details.
- Be prepared to change and elaborate this architecture as circumstances dictate, as you perform your spikes and experiments, and as functional and quality attribute requirements emerge and solidify.
- On smaller projects with uncertain requirements, at least try to get agreement on the major patterns to be employed. Don't spend too much time on architecture design, documentation, or analysis up front.

Rick Kazman
University of Hawaii and SEI/CMU,
Honolulu, Hawaii, USA

Preface

Today's software-driven businesses feel increasing pressure to respond ever more quickly to their customer and broader stakeholder needs. Not only is the drive for increased business flexibility resulting in new kinds of products being brought to market, it's also accelerating evolution of existing solutions and services. Handling such rapid-paced change is a critical factor in enterprise software delivery, driven by market fluctuations, new technologies, announcements of competitive offerings, enactment of new laws, and more. But change cannot mean chaos. In software-driven businesses, all activities—and change activities in particular—must be governed by a plethora of formal and informal procedures, practices, processes, and regulations. These governance mechanisms provide an essential function in managing and controlling how software is delivered into production.

Traditionally, the pressure on enterprise software delivery organizations has been to balance their delivery capabilities across four key dimensions:

- *Productivity* of individuals and teams, typically measured in terms of lines of code or function points delivered over unit time.
- *Time-to-market* for projects to complete and deliver a meaningful result to the business. This can be measured in average time for project completion, project over-runs, or turnaround time from request for a new capability to its delivery in a product.
- *Process maturity* in the consistency, uniformity and standardization of practices. Measurements can be based on adherence to common process norms or on maturity approaches, such as capability maturity model levels.
- *Quality* in shipped code, errors handled, and turnaround of requests. Measures are typically combinations of defect density rates and errors fixed per unit time.

However, with the increasing pressure to respond more quickly, finding an appropriate balance across these enterprise software delivery success factors is increasingly difficult. For example, efforts to enhance productivity by introducing large off-shore teams have frequently resulted in a negative impact on the quality of delivered solutions. Likewise, increasing process maturity by introducing common process governance practices has typically extended time-to-market and reduced flexibility.

These challenges are moving organizations toward rapidly embracing agile software delivery techniques. Since the publication of the *Manifesto for Agile Software Development* over a decade ago, there has been widespread adoption of techniques that embody the key tenets of the "agile manifesto" to the point that a number of surveys offer evidence that agile practices are the dominant approaches in many of today's software delivery organizations. However, these approaches are not without their critics. Notably, agile delivery of software faces increasing pressure as the context for software delivery moves from smaller colocated teams toward larger

team structures involving a complex software supply chain of organizations in multiple locations. In these situations, the lighter-weight practices encouraged by agile software delivery approaches come face-to-face with the more extensive control structures inherent in any large-scale software delivery effort. How can flexibility and speed of delivery be maintained when organizational inertia and complex team dynamics threaten to overwhelm the essential nature of an agile approach?

For many people, the focus of this question revolves around the central theme of software and systems architecture. It is this architectural aspect that provides coherence to the delivered system. An architectural style guides the system's organization, the selection of key elements and their interfaces, and the system's behavior through collaboration among those elements. A software architecture encompasses the significant decisions about the organization of the software system, the selection of structural elements and interfaces by which the system is composed, and determines their behavior through collaboration among these elements and their composition into progressively larger subsystems. Hence, the software architecture provides the skeleton of a system around which all other aspects of a system revolve. Consequently, decisions concerning a system's software architecture play a critical role in enhancing or inhibiting its overall flexibility, determine the ease by which certain changes to the system can be made, and guide many organizational aspects of how a system is developed and delivered.

Over the past decade, many different opinions and viewpoints have been expressed on the term "agile software architecture." However, no clear consensus has yet emerged. Fundamental questions remain open to debate: how much effort is devoted to architecture-specific tasks in an agile project, is the architecture of an agile software system designed up front or does it emerge as a consequence of ongoing development activities, who participates in architectural design activities, are specific architectural styles more appropriate to agile software delivery methods, how are architecturally-significant changes to a system handled appropriately in agile software delivery methods, and so on.

This book provides a collection of perspectives that represent one of the first detailed discussions on the theme of agile software architecture. Through these viewpoints, we gain significant insight into the challenges of agile software architecture from experienced software architects, distinguished academics, and leading industry commentators.

The book is organized into four major sections.

- Part I: *Fundamentals of Agile Architecting* explores several of the most basic issues surrounding the task of agile software delivery and the role of architectural design and decision-making.
- Part II: *Managing Software Architecture in Agile Projects* considers how core architectural ideas impact other areas of software delivery, such as knowledge management and continuous system delivery.
- Part III: *Agile Architecting in Specific Domains* offers deeper insight into how agile software architecture issues affect specific solution domains.

- Part IV: *Industrial Viewpoints on Agile Architecting* takes a practical delivery perspective on agile software delivery to provide insights from software engineers and the lessons learned from the systems they have been responsible for delivering.

As we summarize below, each of the chapters of this book provides you with interesting and important insights into a key aspect of agile software architecture. However, more importantly, the comprehensive nature of this book provides us with the opportunity to take stock of how the emergence of agile software delivery practices change our understanding of the critical task of architecting enterprise-scale software systems.

PART I: FUNDAMENTALS OF AGILE ARCHITECTING

Over the past few years, an increasingly large number of researchers and practitioners have been emphasizing the need to integrate agile and software architecture-centric approaches to enable software development professionals to benefit from the potential advantages of using agile approaches without ignoring the important role of architecture-related issues in software development projects—a trend that can be called "*agile* architecting." This section includes four chapters that are aimed at emphasizing the importance of integrating agile and architectural approaches, as well as providing a set of practices and principles that can be leveraged to support agile architecting. As such, the key objectives of the chapters included in this section are the following:

- To provide a good understanding of the role and importance of software architecture within software development teams using agile approaches, and
- To describe and illustrate a few practices for supporting agile architecting in large-scale industrial software development.

In Chapter 2, Coplien and Reenskaug provide a detailed comparison of the evolution of thinking about architecture in the building construction and the software worlds. Such comparison is important and relevant for gaining an understanding of the importance and role of integrating architecture-focused and agile approaches due to similarities in constructing buildings and software in areas such as design patterns. The authors argue that most of the progress in architectural thinking in both fields is the result of learning in the field of design and in collective human endeavor. They present and discuss a paradigm called DCI (data, context, and interaction) that places the human experiences of design and use of programs equally at center stage. According to the authors, DCI follows a vision of having computers and people mutually supportive in Christopher Alexander's sense of great design. They explain different aspects of the DCI, its philosophical basis, and practical relevance to software and systems delivery.

In Chapter 3, Stal emphasizes the importance of systematic refactoring of software architecture to prevent architectural erosion. The author argues that like any other changes in a software intensive system, architectural modifications are also quite common. According to the author, a systematic architectural refactoring enables a software architect to prevent architectural erosion by evaluating the existing software design before adding new artifacts or changing existing ones. That means software architects proactively identify architectural problems and immediately resolve them to ensure architectural sustainability. The author has presented an agile architectural refactoring approach that consists of problem identification, application of appropriate refactoring techniques, and testing of the resulting architecture. According to the author, the architecture refactoring is often combined with code refactoring activities for the best value-add. Additionally, the refactoring patterns can offer a toolset to software engineers.

In Chapter 4, Cleland-Huang, Czauderna, and Mirakhorli present an approach aimed at addressing the challenges associated with eliciting and analyzing Architecturally Significant Requirements (ASRs) during the early phases of a project. Compared with existing heavy-weight approaches (e.g., win-win and i*) to elicit and analyze ASRs, they present a lightweight approach based on the use of personas of different stakeholders of a system. They introduce the notion of architecturally-savvy persona (ASP) for eliciting and analysing stakeholders' quality concerns and to drive and validate the architectural design. The authors present several personas from different domains and explain how personas can be used for discovering, analyzing, and managing architecturally significant requirements, and designing and evaluating architectural solutions. Through illustrated examples, the authors also show how ASPs can be used to discover quality attributes, steer architectural design, and support traceability.

In Chapter 5, van der Ven and Bosch address the important topic of improving the architecture design decisions-making process when using agile development methods. The authors present a framework of three axes that can be used to project the architectural decision process, which they evaluate in five industrial case studies. The findings from the case studies provide evidence to support the utility and usefulness of the presented Triple-A Framework for helping locate the places where the architecture process can be improved as the agility of a project changes.

PART II: MANAGING SOFTWARE ARCHITECTURE IN AGILE PROJECTS

Traditionally, various kinds of activities have been associated with the software development process and seen as important areas of software engineering. These activities include the main phases of a waterfall process, such as requirements analysis, design, coding, testing, integration, deployment, and maintenance. In addition, there are more focused subactivities that either crosscut these main phases or are part

of them. An example of the former is variability handling; an example of the latter is software architecture analysis.

The need for all these activities has been recognized during several decades of industrial software development, and there is no reason to doubt their rationale. Whatever the process model is, the concerns that are behind these activities must somehow be taken care of. The agile movement does not say that these concerns are groundless, but rather that the activities are not sequential in nature, and that for the most part these concerns can be satisfied without heavy management, relying more on the capabilities of teams and individuals. In particular, we believe that the agile movement is now mature enough to more explicitly consider how various kinds of focused subactivities can be manifested in an agile setting, without deviating from the agile path. The chapters in this part argue that by doing this, it is possible to strengthen agile projects from the viewpoint of a particular concern that can otherwise be easily overlooked.

A particularly interesting crosscutting concern is variability—that is, the ability of a software system to be adapted for a specific context. This is a central quality property of almost any software system, essential not only for maintenance and reuse but also for development time flexibility. The most systematic approaches for handling variability have been developed in the context of product lines. On the other hand, the core of agility is to embrace change. In a way, both product lines and agile methods strive for malleable software: the former tries to plan and specify the required variability beforehand and build variation points to support it, while the latter emphasizes practices that allow responding to changing requirements during development. Obviously, agile methods benefit from software solutions that support flexible changes in the system, and on the other hand the heavy-weightiness of traditional product-line engineering could be relieved by agile approaches.

In Chapter 6, Galster and Avgeriou discuss the challenges and benefits of combining variability handling and agility, and propose an approach for agile-inspired variability handling. In contrast to pure agile, their approach involves certain upfront planning, namely the identification and expression of the desired variability—the so-called variability profile—and an initial sketch of the software architecture with variation points. Initial variability profiles and architectural sketches with variation points can be regarded as the minimal amount of planning required for lightweight variability handling during the development process. These artifacts are revised iteratively in the process when new variability requirements emerge. The approach is demonstrated using a realistic running example.

Another concern which is generally overlooked in agile contexts is ensuring the quality of software architecture. Software architecture is typically not identified as a first-class artifact with explicit presentation in agile approaches. Accordingly, a central concern in agile is not the quality of software architecture, but rather the overall quality of the produced system as experienced by the customer. Still, architectural analysis offers obvious benefits independently of the process paradigm. Software architecture is a first expression of the system to be produced, and in principle it allows the identification of problems and risks before spending resources to

implement something that is not admissible. Unfortunately, most of the architectural analysis techniques have been developed with a traditional waterfall mindset, assuming comprehensive architectural descriptions and the availability of considerable time and resources. Furthermore, architectural analysis has been traditionally regarded as a one-shot activity, carried out when the architecture has been designed. This has made it hard to integrate architectural analysis as a part of agile development.

In Chapter 7, Buchgeher and Weinreich point out that in agile approaches, software architecture is incomplete and continuously evolving. Thus, any architecture analysis method applied in the agile context should be incremental, allowing continuous analysis activity that can be carried out with reasonable resources and time. An obvious approach for less resource-demanding architecture analysis is automation: if a substantial part of the work can be automated with a simple tool, the analysis can be performed in agile development without unreasonable deviation from the agile principles. This could be compared to tool-assisted testing of the implementation. On the other hand, the scope of automated architecture analysis is necessarily limited: this is a tradeoff between coverage and efficiency.

Buchgeher and Weinreich discuss the benefits and problems of different architecture analysis approaches in agile software development, and conclude that a potential technique in this context would be so-called dependency analysis. This is an analysis technique which aims to extract static dependencies from the source code and compare the actually implemented architecture with the intended one, using the dependencies as an indication of architecture-level relationships. They further present a tool-assisted approach, LISA (Language for Integrated Software Architecture), to support this kind of continuous architecture analysis in an agile project context. This approach has been studied in several projects, including a sizable industrial one.

Another general crosscutting concern of software development is the management of various kinds of knowledge produced and consumed during the development process. Regarding software architecture, the term architecture knowledge management (AKM) has been coined to refer to all the activities related to the creation, managing, and using representations of software architecture and its rationale. Traditionally, these kinds of activities are downplayed in agile development in favor of face-to-face communication. However, there can be many reasons that make more systematic approaches to AKM necessary in real life, regardless of the process paradigm. For example, in large multisite projects architectural knowledge needs to be transferred between hundreds of stakeholders and globally distributed sites, and for systems with a lifespan of decades, the architectural knowledge has to be transferred over many generations of architects.

In Chapter 8, Eloranta and Koskimies suggest that it would be possible to achieve a lightweight approach to AKM suitable for agile development by combining the use of an architectural knowledge repository with a decision-based architecture evaluation technique. This kind of evaluation technique analyzes the architecture decision by decision, in a bottom-up manner, rather than taking a top-down, holistic view of the architecture. Thus, decisions can be analyzed as soon as they are made in agile

development, without a need for an offline, heavyweight architectural evaluation. Since a decision-based analysis makes the architectural decisions and their rationale explicit, a significant portion of architectural knowledge emerges and can be recorded in a repository as a side effect of the analysis. Using existing techniques for generating specialized documents from the repository, an agile project can be augmented with virtually effortless architectural documentation services. The authors further study in detail how the proposed approach could be applied in the context of the observed architectural practices in industry.

The core activity related to software architecture is of course the actual design of the architecture. Many agile approaches are deliberately vague about this activity. Early agilists even argued that architecture need not be explicitly designed, but just emerges during the development. While this might be true in some cases, today it is generally understood that software architecture and its design play a significant role in agile development—especially in large-scale projects. However, the incremental and iterative nature of agile development poses a major challenge for software architecture design: how to build software architecture in a systematic manner piecewise, in parallel with the implementation.

In Chapter 9, Pérez, Diaz, Garbajosa, and Yagüe address this question by introducing the concept of a working architecture. This is an architecture that evolves together with the implemented product. A central element of a working architecture is a malleable, incomplete, so-called plastic partial component. A new working architecture can be expressed in each agile iteration cycle using such components. Eventually, the components become complete, constituting the final architecture associated with the delivered product. The approach supports changes in the requirements by maintaining traceability links between features and their realization in the working architecture. Given a change in the features, the involved parts of the working architecture can be automatically detected using such links. The proposed techniques are integrated with Scrum, and tried out in a large case study project.

PART III: AGILE ARCHITECTING IN SPECIFIC DOMAINS

Agile architecting in specific domains share many commonalities and many of their concerns overlap, but they also have marked differences in focus and approach. Each solves problems for different stakeholders, uses different technologies, and employs different practices. The specialization on their respective solutions has made it difficult to transfer methods and knowledge across a broad range of topics. One way to align these topics is to shift the focus from solution to problem domains. As the system evolves, verifying its security posture is indispensable for building deployable software systems. Traditional security testing lacks flexibility in (1) providing early feedback to the architect on the resilience of the software to predict security threats so that changes are made before the system is built, (2) responding to changes in user and behavior requirements that could impact the security of software, and (3) offering real design fixes that do not merely hide the symptoms of the problem

(e.g., patching). There is a need for an architecture-level test for security grounded on incremental and continuous refinements to support agile principles.

Part III contains two chapters looking at agile architecting in specific domains. The chapters in this section present practical approaches and cases. Chapter 10 focuses on architecture-centric testing for security from an agile perspective and Chapter 11 describes supporting agile software development and deployment in the cloud using a multitenancy multitarget architecture (MT^2A).

In Chapter 10, Al-Azzani, Bahsoon, and Natour suggest using architecture as an artifact for initiating the testing process for security, through subsequent and iterative refinements. They extend the use of implied scenario detection technique to reveal undesirable behavior caused by ambiguities in users' requirements and to analyze its security implications. The approach demonstrates how architecture-centric evaluation and analysis can assist in developing secure agile systems. They apply the approach to a case study to evaluate the security of identity management architectures. They reflect on the effectiveness of the approach in detecting vulnerable behaviors, and the cost-effectiveness in refining the architecture before vulnerabilities are built into the system.

Chapter 11 emphasizes the need for a systematic approach for supporting agile software development and deployment in the cloud. Rico, Noguera, Garrido, Benghazi, and Chung propose a MT^2A for managing the cloud adoption process. Multitenancy (MT) architectures (MTAs) allow for multiple customers (i.e., tenants) to be consolidated into the same operational system, reducing the overhead via amortization over multiple customers. Lately, MTAs are drawing increasing attention, since MT is regarded as an essential attribute of cloud computing. For MTAs to be adopted in practice, however, agility becomes critical; there should be a fast change to the system so as to accommodate potential tenants in as short a period of time as possible. In this chapter, they introduce a MT^2A. MT^2As are an evolution to traditional MTAs that reduce the various overhead by providing multiple services instead of a single service. In MT^2As, there are new components added to its corresponding MTAs for managing the (now possibly) multiservice. MT^2As enhance the agility of MTAs, not only in deployment but also in development, by enabling the reuse of common components of the architecture. In this chapter, they also present an implementation of the architecture through an MT2 system called Globalgest.

PART IV: INDUSTRIAL VIEWPOINTS ON AGILE ARCHITECTING

For many people involved in creating, maintaining, or evolving software-intensive systems, the reality of any approach is the extent to which it helps with the day-to-day challenges of building complex software efficiently to support the business's needs. Hence, any discussion on agile software architecture would be incomplete without considering the practical realities that face software engineers when they deliver new capabilities into production. Here, the drive for speed and adaptability offered by an agile approach must be aligned with the broader project delivery needs to supply the

capabilities required by the stakeholders to a deadline and at a cost that makes business sense to the managing organizations. This equation is yet more difficult to resolve where such projects are large in scale, take place over many months or years, and involve hundreds of people from perhaps dozens of different organizations.

Part IV contains four chapters that explore some of the practical considerations in agile software architecture from the viewpoint of practicing software architects.

Chapter 12 considers the challenges facing agile software development practices in the context of large-scale complex systems delivery situations. Hopkins and Harcombe explore what happens when rapid delivery cycles and flexible decision-making come up against large-scale systems engineering concerns typical of enterprise solutions—hundreds of people, long project lifetimes, extensive requirements planning, and a constant ebb and flow of changing priorities. A particular interest of this chapter is efficiency in large-scale software delivery, and the need to manage distributed teams in the most effective manner possible. Through a series of examples, Hopkins and Harcombe discuss critical success factors for agile software delivery, and the critical role that architectural planning and design can play in ensuring that as the project scale increases, the value of an agile approach is amplified rather than being overwhelmed.

In Chapter 13, Eeles considers the importance of supporting evolution and change to a software-intensive system, and the practical implications of creating a "change-ready" system. From his perspective, a focus on how a system changes throughout its lifetime shapes the critical choices an architect makes during design and construction of that system. Important elements of this viewpoint are highlighted by tracing the history of software development practices from waterfall phases through iterative design to agile techniques. A particular focus of the chapter is the innovation that is essential in both the delivered system, and in the environment of tools and practices that produces that system. Based on his work on several industrial projects, Eeles makes a series of important observations to guide architects in delivering solutions that are more adaptable to changing needs.

In Chapter 14, Friedrichsen addresses a question central to many discussions surrounding agile approaches to architecting software: In agile projects, is the architecture of the system designed up-front, or does it emerge over time as the result of the agile software development process? For Friedrichsen, the idea of emergent architecture is the result of constant refactoring of a system based on a well-defined set of architectural principles. The chapter considers the importance of such an emergent style of architecture, its key properties, and the kinds of situations in which this emergent approach has particular value. One of the main conclusions is that, in practice, a hybrid form combining both explicit and emergent architectural techniques is feasible and useful.

Finally, Chapter 15 describes the journey toward more agile software development practices that took place in one IT team as it took on the task of evolving a complex software platform at a large insurance company in the United Kingdom. Isotta-Riches and Randell discuss their motivations for adopting a more agile software development approach, and the challenges they faced making the changes they

needed to their practices, processes, and skills. The importance of focusing on new architectural thinking in the teams was soon considered to be central to this journey, and the chapter highlights how this need surfaced, what they did to explore its implications, and how they dealt with the challenges raised. As often occurs in practice, the result was a compromise between the purity of an agile approach as found in textbooks, and the need to address the practical business reality driving the project's timeframe, capabilities, and costs. The chapter offers sobering lessons for all those involved with creating not only elegant solutions to problems, but also systems that pay their way.

Muhammad Ali Babar
Alan W. Brown
Ivan Mistrik