Rationale-Based Software Engineering

Janet E. Burge • John M. Carroll •
Raymond McCall • Ivan Mistrík

Rationale-Based Software Engineering

Springer

Authors

Janet E. Burge
Miami University
School of Eng. & Appl. Science
Computer Science & Systems Analysis
205 Benton Hall
Oxford, OH 45056
USA
burgeje@muohio.edu

John M. Carroll
Penn State University
School of Information Sciences
and Technology
504 Rider I Building
120 S. Burrowes Street
University Park, PA 16801-3857
USA
jcarroll@ist.psu.edu

Raymond McCall
University of Colorado
College Architecture & Planning
314 UCB
Boulder, CO 80309-0314
USA
Mccall@colorado.edu

Ivan Mistrík
Independent Consultant
Werderstr. 45
69120 Heidelberg
Germany
i.j.mistrik@t-online.de

ISBN 978-3-540-77582-9 e-ISBN 978-3-540-77583-6

Library of Congress Control Number: 2008924869

ACM Computing Classification (1998): D.2, K.6

Cover design: KünkelLopka Werbeagentur, Heidelberg, Germany

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

Foreword

The Search for Meaning

At the risk of appearing to exaggerate, I will argue that the pursuit of rationale in engineering is nothing less than a *search for meaning*. On the face of it, capturing, recording, and perusing rationale in support of software engineering is a worthy software management activity, whose benefits are well documented and accepted. Indeed chapters of this book speak to this issue. However, there is a more significant reason for the pursuit of rationale: a desire to make sense of the world – to explain it and to explain its behavior, both expected and unexpected. Weick calls this *sensemaking*, and of course is right insofar as the world makes 'sense'. Wouldn't it be grand if we were able to understand why the world is structured as it is, and why artifacts in the world have been engineered to behave the way they do? Sometimes, the reasons why are straightforward: an engineer solving a problem in the world may recognize it as a *normal problem* that he has encountered before, the solution of which is well understood, tried, and tested. The rationale for his engineering solution in this case is mostly reusable – after all he is engaging in normal engineering, in *normal design* (Vincenti 1990).

But what if the problem encountered is *radical?* Well, Vincenti tells us that we need to engage in radical engineering, in *radical design*. The consequence of this is that we should expect to fail in our first attempts at a solution, but strive to learn from our failures, so that future encounters with our radical problem become more normal.

It is in this transition from radical to normal that rationale research offers attractive opportunities for advancing the state of the art in software engineering, and offers an intellectual umbrella for breaking new ground in this area. This umbrella needs to cover both *problem analysis* in pursuit of stakeholder requirements, and *engineering design* in pursuit of solutions to those requirements. Research on the relationship between requirements and design, on managing traceability and software evolution, and ultimately on assuring the quality of software engineering solutions, all sit comfortably under this umbrella.

However, there is a difficulty, observed by Jackson (Jackson 2007). As we specialize and strive to evolve the discipline of software engineering into normal engineering, we find that much normal design rationale is hidden, perhaps lost in time, when imaginable alternative solutions were considered and discarded. We then find ourselves back to where we started: trying to make sense of what we have already, trying to understand the reasons why the normal design we have before us is the way it is. Weick writes:

> Sensemaking is about such things as placement of items into frameworks, comprehending, redressing surprise, constructing meaning, interacting in pursuit of mutual understanding, and patterning. (Weick 1995, p.6)

My colleague Simon Buckingham Shum (2007) has taken Weick's message to heart, and has made sensemaking the centerpiece of his framework for constructive argumentation and explanatory rationale.

A book such as this is important because the development of software is *engineering* and not science. It is not enough simply to understand why software behaves the way it does, but rather how it can be built – rationally (Parnas and Clements 1986), or at least systematically – to behave as intended. We need the framework offered by this volume to develop such meaningful software.

Bashar Nuseibeh
Professor of Computing and Director of Research
The Open University, UK

Foreword

Design Rationale: Retrospect and Prospect

Danish philosopher Soren Kierkegaard once said that life can only be understood backwards but must be lived forward. He might easily have been talking about the software design process. In a software project, many developers work together on a system development effort, some of them only for some phases of the project, and few with an overview of the entire system. As a system emerges from this process, it has to be explained for future designers, maintenance programmers, and others. Some of the logic, the rationale for the way the system is, may have been apparent from the beginning. But in many cases, the trajectory of all the little decisions that contribute to the way a system is when it is released may only be apparent in retrospect when tied together in a deliberate activity of sensemaking and documentation.

It is not supposed to be this way. Software engineering textbooks and process improvement manuals exhort us to have well-defined requirements and a core architectural vision that drive the details forward. Such requirements and architectural features must be documented and internalized by project staff so that everyone appreciates the significance and impact of changes. But the real life of software projects is not so simple. Stakeholders change their minds. The business context of a system changes during its development. As infrastructure technology changes, new implementation opportunities become available. Architectural commitments have to be changed or diluted as their consequences become apparent. People simply forget what they are doing and develop different styles in how they work and implement software that may conflict. Kierkegaard, of course, was not a software architect. His struggle was not with customers, users, and intransigent or imperfect fellow developers, but 19th century institutions, repression, and hypocrisy. But the consequences of both struggles are the same: we cannot always make sense of what is

going on when we are in the middle of things until after the smoke has cleared. And the root causes are essentially the same too: the world is complex, and people are only human.

Design rationale research started in the 1980s from the recognition that the results of design often do not make sense to those outside the design team, but have to be made sense of to foster better understanding during the ongoing process of maintenance and feature evolution. There were two strands to this research emerging from different communities, and these strands persist to the present day. In a nutshell, the difference between them goes back to Kierkegaard's comment: if we have to make a choice between the two alternative modes of sense making, should we try to make design easier to explain in retrospect, or should we make it more transparent and reflective while it is going on? These answers led to retrospective rationale research and prospective rationale research.

In the retrospective rationale community, the concern was primarily how to document large-scale software architectures or standards. Since an architecture is a stable foundation for a continually evolving system, and standards are expected to endure over generations of many systems produced by many organizations, it is essential to document the architecture and standards and the reasons behind them. Architectures and standards are the types of thing that we are stuck with once we make a commitment to them and then come to depend on for a myriad of detailed decisions. They are therefore high-risk commitments.

The first notable example of the use of rationale in after-the-fact documentation came from the team led by David Parnas, who used the avionics software of the A-7 aircraft as a microcosm for exploring design specification and documentation techniques (Parnas and Clements 1986).

The A-7 work has had an influence in more recent efforts and methods, such as the Software Engineering Institute (SEI) architecture initiative (Clements et al. 2002). Different but similarly motivated efforts have led to architectural decision support technologies such as the Architecture Design Decision Support System (ADSS) (Capilla et al. 2007). There is a general consensus emerging that the documentation of rationale, whatever form it takes, should be tightly bound to the documentation about the architecture itself. For example, Zhu and Gorton (2007) devised a Unified Modeling Language (UML) profile for adding rationale information to standard UML design diagrams. In this way, the rationale is a first-class part of the documentation, not an addendum or collection of low-value notes.

In the standards community, it has become almost universal to document the rationale for parts of a standard, often in terms of comparisons between what the standard requires and plausible but inferior alternatives. A pioneering example of this in software engineering was the

documentation and rationale for the Ada programming language and programming support environment. The latter (code named "Stoneman") is a particularly good example of the analysis of why the environment had a layered structure (Druffel and Buxton 1980).

The explanatory role of retrospective rationale documentation means that it is not critical that the rationale be historically accurate. Designers might well have had considerations in mind that led to architecture decisions that no longer seem relevant in retrospect. Conversely, making sense of the overall sweep of the architectural design process in retrospect, it may be clear that reasons were prominent that were not completely apparent at the time. Reasons given may be a white lie, they may oversimplify or distort that convoluted process that people went through. They may even be self-serving or apologetic, designed to protect the authors from criticism. After the fact, these factors may not be important to maintenance programmers or designers of later releases. An approximate, simplified and glossed rationale, even one that is somewhat distorted, may be more suitable for supporting the concerns of these professionals than documentation that more faithfully describes the agonizing process of decision-making that the developers actually went through.

In keeping with this, creative and constructive distortion of the design decision-making process is another property of retrospective rationale: it has to be carefully crafted. Like the documentation of the architecture's form, the documentation of its rationale is expected to endure and become part of the project's knowledge base as the project goes forward. It therefore makes economic sense to invest resources and time in writing rationale documents and clearly articulating them.

In contrast, another community came to design rationale research in the 1980s. These researchers had been inspired by the pioneering work of Horst Rittel and other design theorists in their attempts to provide structure to collaborative decision-making among designers and other stakeholders, particularly in community architecture and urban planning projects, when the problems they faced were "wicked". A wicked problem is characterized by dispute about what the problem actually is and how one would recognize whether it had been solved. Thus, a wicked problem is not just hard; its very nature is contested and cries out for discussion. In this second tradition of design rationale research, therefore, an emphasis was placed on semi-structured representations of ongoing issues, positions, and arguments. The emphasis was on supporting problem formulation and decision-making as they occurred rather than seeking to justify decisions for people who came after.

Such support is support for design rationale, though, for two reasons. The first reason is definitional: the rationale for a decision consists of the

reasons why it was chosen. These reasons do not have to be documented with a future consumer, such as a maintenance programmer, in mind. Even if nobody were to read the rationale in the future, its documentation and value during the unfolding of the design would not make it any the less a record of rationale. A second reason for this type of ongoing decision-making support counting as rationale is more practical: the information may well be useful in the future by accident, even though that may not be the motivation for its capture.

In fact, the term "capture" reveals a fundamental difference between the two streams of research. Both streams emphasize care and professionalism during design. But the careful audience analysis, crafting, and writing of rationale documentation in the retrospective rationale tradition emphasizes that the recording of rationale is a significant part of a project and should be budgeted for and rewarded. In contrast, the capturing of design rationale in the prospective rationale tradition implies that rationale documentation is a fortuitously gathered by-product of another activity. That other activity, collaborative design argumentation and decision-making, may be serious, it may be planned and budgeted, and it may be highly structured in its processes. But the rationale produced is expected to be immediately valuable, and any later benefits that accrue from it should not require any further planning or writing. These benefits should come for free.

Probably the most influential prototype prospective rationale management system was gIBIS (Conklin and Begeman 1988), which although it never created a major user community, was used in NCR for the development of hotel and restaurant support systems (Conklin and Burgess-Yakemovic 1991), and the IBIS argumentation model at its core has been extremely influential as the baseline for representation of nearly all rationale.

In parallel to the use of rationale capture in software engineering, a similar argumentation model based on explicit decision criteria was influencing research into user interface design in the human–computer interaction (HCI) community (Maclean et al. 1989). Here the design decisions were typically more local in scope, such as in the choice of alternative user interface widgets or menu structures to support a user's task. The model used, Design Space Analysis, based on questions, options, and criteria, rather than the issues, positions and arguments of IBIS, emphasized the making of choices between mutually exclusive options and was based on explicit and frequently quantitative criteria. The design problems addressed were therefore constrained and clearly specified. While they may have been subtle and far reaching in their impact on usability, they were anything but wicked problems in the sense defined above.

More recent research in prospective design rationale in software engineering tends to emphasize quantitative criteria for choosing among alternatives, and the normal targets of these decisions are architectural choices such as the distribution of services across a network. Recent work in software engineering economics represents an attempt to make these design decisions rigorous in the same way that financial decisions in business can be based on rigorous projections and risk models (Boehm et al. 1995; Bose 1998). Typical of the decision-making methods and models that are incorporated into such work are Cost–Benefit Analysis (e.g., Kazman et al. 2003) and the Analytical Hierarchy Process (e.g., Lozano-Tello and Gomez-Perez 2001; Wallin et al. 2007).

In software engineering, other than the early, limited experiments with gIBIS, prospective design rationale research foundered for several years, possibly because the unique qualities of software design were largely neglected. The increasingly complex argumentation models could have applied to the design of anything. In customizing design rationale representations to software engineering, a key early insight was that software design methods are sets of heuristics for making requirements, design, and implementation decisions, not just software notations. Object-oriented methods, for example, provide guidelines for the identification of objects and their responsibilities and guidelines for refactoring when these early decisions lead to reorganization. Such methods essentially encapsulate reusable design knowledge. Before the widespread adoption of object-oriented methods, the methodology community was rather fragmented, and so extensions of design rationale representations to software methods tended to focus on illustrative methods. Among these were Potts and Bruns's (1988) adaptation of the Liskov and Guttag abstraction-based design method, the Potts (1989) treatment of Jackson's Jackson System Development (JSD), and the later incorporation of goal-based and scenario-based representations of system requirements into the Inquiry Cycle model of prospective rationale (Potts et al. 1994).

In addition to design methods, which tend to focus on generic design decisions, domain-specific issues can also arise that can be captured and represented as reusable rationale. An early example of this was Belotti's (1993) attempt to integrate theoretical and practitioner perspectives on HCI guidelines through Design Space Analysis. More recent work has included domain-specific architectural rationale for automotive software engineering (Wallin et al. 2007).

Not satisfied with the attachment of design rationale to design artifacts represented by these methodological extensions or by the introduction of explicit and often quantitative criteria in the Design Space Analysis community in HCI, some researchers sought to extend the rationale models

in such a way that some decisions could be made automatically or dependencies between decisions could be computed and maintained consistently (Ali Babar and Gorton 2007; Lee 1991; Lee and Lai 1991; Wang and Xiong 2001) by means of an elaborate and formal data model for design rationale information and its relationship to elements in the design itself. It is not clear to what extent the benefits of such computational support outweigh the burden of recording the rationale information in such a rigorous and necessarily fine-grained fashion. Nor is it clear whether the structure of such models can be easily maintained as the design and its rationale change. Such approaches do promise to be extremely valuable when coupled with a formal theory of design change and configuration management.

Recently, however, the focus has returned to reusable software engineering knowledge and rationale. The entire software patterns community (Gamma et al. 1995) can be regarded as engaged in a quest to produce a corpus of rationale documents that discuss design patterns, the issues that arise when they are used, the arguments for when they are appropriate and when they cause problems, how they interact, and illustrations of their use. Debating whether a pattern library is a generic library of design artifacts or a rationale library seems rather fruitless, since the design alternatives faced by a designer, criteria and considerations that affect the decisions, illustrative solutions, and warnings about interactions are so inextricably interwoven. The role of rationale is woven through the patterns literature, although it has a secondary role to the capturing of artifact knowledge. A more explicit role for rationale can be seen in Baniassad et al.'s (2003) Design Pattern Rationale Graph (DPRG), a tool for linking designs and implementations through rationale information.

Thus, much of the development of work in the prospective design rationale tradition can be seen to be aimed at producing information that can be used subsequently, not just as an aid to decision-making in the moment. Such subsequent uses may include specific rationale information to be referred to later in the same project, or it may even take the form of more generic lessons learned that can be applied across projects.

There have been few comprehensive reviews on design rationale, and none of monograph length. The theoretical models of design rationale, the phases of the design process during which they are useful, the domain-specific contexts in which they can be applied, and evidence of practicality have been lacking. Only one software engineering textbook, that of Bruegge and Dutoit (2004) makes a thorough attempt to integrate design rationale into the software engineering process.

As we enter the third decade of design rationale research, however, now is a good time to take stock. Everyone acknowledges that designing is

difficult, that it involves many people often over long periods of time who need explicit records of who did what, when, and for what reasons. The support for design rationale and its integration into software engineering processes has not yet reached the mainstream of software engineering writing and practice, and it is time that it did. Or as Kierkegaard also said: truth always rests with the minority.

Colin Potts
Associate Professor in the School of Interactive Computing
Georgia Institute of Technology, USA

Preface

The most distinctive thing about humans is not the thumb, of course. It is design. Unlike any other animal, we incessantly and dramatically reshape both ourselves and our environment. We design ourselves through innovating concepts, language, culture, and other practices, and we design almost everything around us. It is telling that we now speak of "natural" places on the Earth to distinguish the few places we have not (yet!) redesigned.

Among the most complex, diverse, and pervasive things that humans design are software systems. The history of software design is almost entirely a history of trying to catch up with complexity and diversity. As we look back to the 1960s the notion of what was then called the "software crisis" seems almost amusing. At that time, barely a decade after the invention of software, it was recognized that the complexity and diversity of software systems was being elaborated far more rapidly than were engineering techniques to manage software development. What is amusing is that this was (optimistically) called a *crisis*, as if it were a temporary threat that would in the course of time be rectified.

But this never happened. Instead the software crisis became chronic. It became the context for the software industry and for software engineering. And by now, as almost every system is, incorporates, or fundamentally depends upon software, as software systems have become utterly pervasive, the software crisis has really become an epoch in human history.

No one is very happy about this, and from time to time manifestations of the ongoing software crisis bubble up into dramatic mass media reports about how vital defense systems are fundamentally unverifiable, about how medical systems make it more or less inevitable than surgeons will kill their patients, about how banking systems occasionally share account information with unknown hackers, and so forth.

What are we to do? There are many answers, many approaches, but none of them is a "silver bullet" (as Fred Brooks vividly put it). The most obvious approach, and quite likely the most powerful, is to explicitly describe and justify the design, implementation, and use of software systems, and to do this routinely, iteratively, and regularly throughout the software development process. We call this "Rationale-Based Software

Engineering." It is not a new idea, though in some areas there are new tools and techniques. Rather, it is an essential idea that has been around, that we cannot afford to lose track of, and that perhaps can be pushed to greater fruition now. In this book, we try to bring together a broad discussion of rationale and focus on aspects of the very old and very weighty challenge of the software crisis.

Book Overview

This book consists of four parts. Part 1 sets the context for the work and describes why Software Engineering Rationale (SER) and Rationale-Based Software Engineering (RBSE) are essential contributors toward improving the software development process. Part 2 describes how Software Engineering Rationale can be used to support software development. Part 3 describes how RBSE can be applied throughout the software engineering lifecycle as well as supporting software reuse. Part 4 presents architectural and conceptual frameworks for RBSE as well as our vision of future directions for RBSE research.

Part 1: Introduction

So why capture rationale? Before making a case for why SER capture and use should be an essential part of software development, it is important to first define what it is. Part 1 defines rationale and sets the context for the remainder of the book.

Chapter 1, "What is Rationale and Why Does it Matter" provides an initial discussion of the scope and value of rationale in software engineering. An initial introduction of previous work on rationale is provided and we make our initial case for why rationale is useful during software engineering.

Chapter 2, "What Makes Software Different" describes some of the key differences between applying rationale to software engineering and applying rationale to other domains. This includes both opportunities for use in software engineering that are lacking when developing other artifacts as well as some of the unique challenges posed by software development. Specifically, we look at the role of the computer in software development versus physical artifact development as well as the implications of the necessity to support iteration in software development on rationale management.

Chapter 3, "Rationale and Software Engineering" introduces both Software Engineering and Software Engineering Rationale (Dutoit et al.

2006b). Rationale has a role to play in defining software processes, supporting software project management, and as a mechanism to both document and guide decision-making throughout the software process.

Chapter 4, "Learning from Rationale Research in Other Domains" describes key rationale research in other domains and its implication to software engineering. The chapter focuses on four areas: domain-oriented design environments using Procedural Hierarchy of Issues (PHI) (McCall 1991); automating design rationale capture in Computer-Aided Design, more specifically that using the Rationale Construction Framework (Myers et al. 1999); rationale support via Parameter Dependency Networks and DRIVE (de la Garza and Alcantara 1997); and how Case-Based Reasoning (CBR) systems such ARCHIE (Zimring et al. 1995) relate to rationale.

Chapter 5, "Decision-Making in Software Engineering" examines the role that human decision-making has in software engineering. The chapter describes naturalistic decision-making and Klein's recognition-primed decision model (Klein 1998), which addresses some of the problems with classical decision making by proposing a strategy more consistent with observations of human decision-makers, where the first acceptable alternative is selected. The chapter concludes with a discussion of rationale as a resource for decision-making and how rationale relates to both the classical and naturalistic views.

Part 2: Uses of Rationale

There is little or no point in capturing rationale if there are not ways in which it can be used. Part 2 describes some key uses of rationale in software development.

Chapter 6, "Presentation of Rationale" looks at rationale presentation. The two major classes of presentation formats, semiformal and informal, are described. The chapter then describes new opportunities for presentation provided by reusable rationale databases, multiscale presentation, and development tool integration.

Chapter 7, "Evaluation" describes how rationale can be used for evaluation from two angles. The first is how argumentation-based rationale can be used for decision evaluation by evaluating the consistency and completeness of the rationale as well as evaluating support for development alternatives taking into account decision criteria, input from multiple developers, and uncertainty. The second approach to evaluation describes scenario-based evaluation as supported by scenario-based design (Carroll and Rosson 1992).

Chapter 8, "Support for Collaboration" discusses rationale and collaboration from two perspectives. The first is how the highly collaborative nature of software development supports the development, codification, and use of rationale. The need for collaborators to justify their decisions to each other is a key source of rationale. The other is how rationale supports collaboration by encouraging the exchange of information and awareness of the goals of team members.

Chapter 9, "Change Analysis" identifies the important role that rationale can play in assessing the impact of changing requirements, design criteria, and assumptions on a software system. By explicitly recording the impact that those elements had on the decisions involved and relating the results of the decision-making process to the artifacts that instantiate them, the rationale can be used to detect where changes will be required if requirements, criteria, and assumptions change. In addition, rationale can also capture crucial inter-decision dependencies and alert the developer if one of those dependencies is later violated.

Part 3: Rationale and Software Engineering

In software engineering, decision-making is not restricted to only part of the process. There are critical decisions to be deliberated throughout the lifecycle of the software system. Part 3 describes how rationale supports the various stages of the software lifecycle and how rationale research relates to other software engineering research that also supports those stages.

Chapter 10, "Rationale and the Software Lifecycle" gives a brief introduction to the stages of software development and how rationale can be utilized. The topic of lifecycle modeling is then introduced and the application of rationale to sequential models, such as waterfall and the v-model is described as well as how rationale can be applied to iterative approaches. The chapter concludes with a discussion of how rationale supports process improvement initiatives.

Chapter 11, "Rationale and Requirements Engineering" describes rationale's contribution to requirements engineering. This includes how rationale can support the requirements definition process by assisting with requirements elicitation, achieving consensus on requirements, identifying requirements inconsistency, and supporting requirement prioritization. Rationale's role in requirements traceability and the relationship between rationale and nonfunctional requirements is also described. The chapter concludes with how rationale can assist in adapting to changing requirements, one of the major challenges in software engineering.

Chapter 12, "Rationale and Software Design" describes design rationale as applied to software design. The chapter begins with a description of the nature and importance of software design rationale, both that generated by the designers while designing and that generated during construction and use. Two fundamentally different types of decisions are described—design space decisions and rationale for non-design-space decisions that represent a deeper reflection on the design process. We conclude with a look at some specific approaches to rationale as applied to software design and software architecture.

Chapter 13, "Rationale and Software VV&T" defines verification and validation and then describes the issues involved in the major types of software tests—inspection, unit testing, integration testing, and system testing. The role of rationale in software testing is described by focusing on three major uses: the contribution of rationale to testability, rationale's contribution to test case prioritization, and using rationale to support component testing and selection.

Chapter 14, "Rationale and Software Maintenance" describes how rationale can be used to support software maintenance. The chapter describes four areas where rationale can support maintenance: maintenance prediction, impact assessment, program comprehension, and maintenance rationale. The chapter then concludes with a discussion of why rationale should also be captured during software maintenance and some existing research that supports the capture of maintenance rationale.

Chapter 15, "Rationale and Software Reuse" begins with a description of key software reuse concepts and categories, along with defining types of rationale that support reuse. The chapter then describes several ways that rationale has been, or can be, applied to assist with software re-use.

Part 4: Frameworks for Using Rationale in Software Engineering

In this part, we take a look ahead. In order to support Rationale-Based Software Engineering, it is necessary to have frameworks to define the key concepts and architectural needs for Rationale Management Systems. In this part, we define a conceptual framework and architectural framework to support Rationale-Based Software Engineering.

Chapter 16, "A Conceptual Framework for Rationale-Based Software Engineering" describes the goals of conceptual frameworks in general, followed by what is needed by a conceptual framework for rationale use in software engineering. To support the decision-centric approaches, we define a taxonomy of software decisions that could be answered using

SER. To support usage-centric approaches, we describe how Carroll and Rosson's (1992) Scenario Claims Analysis (SCA) rationale can be applied to software engineering. We conclude with a discussion of the implications of iteration, a summary of current challenges to rationale use, and propose some potential solutions.

Chapter 17, "An Architectural Framework for Rationale-Based Software Engineering" describes the key features needed for a Rationale Management System (RMS) to support software engineering. This includes the model management subsystem (which includes support for capture and formalization), the underlying hypermedia substrate, and the necessary integrations between RMS and external software development support systems.

Chapter 18, "Rationale-Based Software Engineering: Summary and Prospect" serves two purposes. First, it summarizes the work presented in this book and its implications for future rationale research and use. We then look at some key future challenges to software development and conclude with a discussion of both the promises of and challenges to Rationale-Based Software Engineering.

Acknowledgements

This book would not have been possible without the support of many people. First of all we would like to thank Ralf Gerstner of Springer, Germany for making this project possible and for invaluable advice in publishing matters. We appreciate his infinite patience with watching this project come to fruition. We would also like to thank Bashar Nuseibeh and Colin Potts for their excellent and inspiring forewords. At Miami University, Monica Baxter provided secretarial assistance in pulling together the various components of the book. Last, but not least, we would like to thank our friends, family, and colleagues for their support, patience, and encouragement throughout this project.

Author Biographies

Janet E. Burge is an assistant professor at Miami University Computer Science and Systems Analysis Department. Dr. Burge's major research interests are in software engineering and artificial intelligence. Her primary research area is in design rationale, with a focus on design rationale for software maintenance. Prior to her appointment at Miami University in 2005, she taught software engineering and assembly language at Worcester Polytechnic Institute (WPI) for four years. During and prior to that time, she worked for eight years at Charles River Analytics Inc. on various projects using genetic algorithms for decision support and on a knowledge elicitation workstation. Before joining Charles River Analytics, she worked for one year at Fidelity Investments developing an expert system to monitor their midrange computer systems and for 11 years at Raytheon Corporation as a software engineer. She received her PhD in Computer Science from WPI in 2005, her M.S in Computer Science from WPI in 1999, and her B.S. in Computer Science from Michigan Technological University in 1984.

John M. Carroll is the Edward M. Frymoyer Chair Professor of Information Sciences and Technology at the Pennsylvania State University. His research interests include methods and theory in human–computer interaction, particularly as applied to networking tools for collaborative learning and problem solving, and the design of interactive information systems. His books include *Making Use* (MIT Press, 2000), *HCI in the New Millennium* (Addison-Wesley, 2001), *Usability Engineering* (Morgan-Kaufmann, 2002, with M.B. Rosson) and *HCI Models, Theories, and Frameworks* (Morgan-Kaufmann, 2003). He serves on several editorial boards for journals, handbooks, and series and is Editor-in-Chief of the ACM Transactions on Computer–Human Interactions. He received the Rigo Award and the CHI Lifetime Achievement Award from the ACM, the Silver Core Award from the IFIP, and the Alfred N. Goldsmith Award from the IEEE. He is a fellow of the ACM, IEEE, and HFES.

Raymond McCall is an associate professor in the Department of Planning and Design at the University of Colorado, Denver. His major areas of

research are in design rationale methods and systems. Since 1992, most of his research has concentrated on the use of rationale to support the design of artifacts for human exploration of space. For much of this time he collaborated with NASA contractors and with employees of the Johnson Space Center in Houston. He has nearly 30 years of experience in design rationale usage in architectural design, planning, policy making and software design. He created the first hypertext systems for support of design rationale in the 1970s and 1980s and was the first to integrate support for rationale capture and delivery into 3D computer-aided design systems. Before coming to the University of Colorado, he worked for six years at the Gesellschaft fuer Information und Dokumentation in Heidelberg, Germany. He received a Ph.D. in Architecture in 1978 from the University of California, Berkeley, and an M.S. in Product Design in 1975 from the Institute of Design at the Illinois Institute of Technology.

Ivan Mistrík is an independent consultant for Software-Intensive Systems Engineering. He is a computer scientist who is interested in software engineering and software architecture, in particular: relating software requirements and architectures, knowledge management in software development, rationale-based software engineering, and collaborative software engineering. He has more than 40 years experience in the field of computer systems engineering, primarily working at renowned R&D institutions as a principal scientist and project manager; in parallel he has done consulting on a variety of large international IT projects sponsored by DFG, 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 in international journals, conferences, books and workshops and was the editor of the book *Rationale Management in Software Engineering* published by Springer-Verlag in 2006. In addition, he was the editor of the Special Issue on *Relating Software Requirements and Architectures* published by IEE Proceedings Software in 2005.

Bashar Nuseibeh is professor and Director of Research in Computing at The Open University (OU), and a visiting professor at Imperial College London and the National Institute of Informatics, Japan. Previously he was a reader at Imperial College London and head of its Software Engineering Laboratory. His research interests are in software requirements engineering and design, software process modeling and technology, and technology transfer. He has published over 100 refereed papers and consulted widely with industry, working with organizations such as the UK National Air