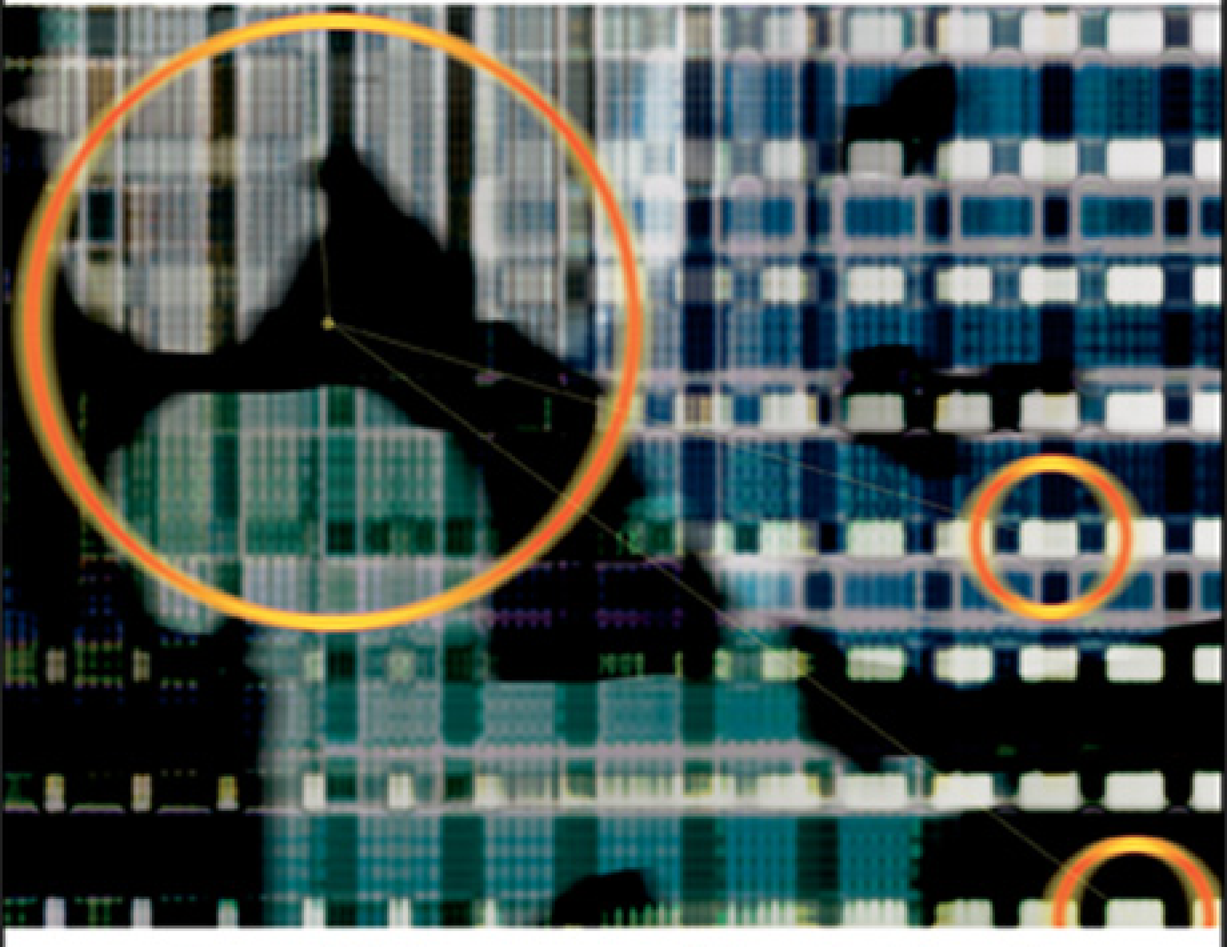
# Measuring Information Systems Delivery Quality



Evan W. Duggan and Han Reichgelt

# Measuring Information Systems Delivery Quality

Evan W. Duggan
University of Alabama, USA

Han Reichgelt
Georgia Southern University, USA

IDEA GROUP PUBLISHING
Hershey • London • Melbourne • Singapore

| | |
|---|---|
| Acquisitions Editor: | Michelle Potter |
| Development Editor: | Kristin Roth |
| Senior Managing Editor: | Amanda Appicello |
| Managing Editor: | Jennifer Neidig |
| Copy Editor: | April Schmidt |
| Typesetter: | Cindy Consonery |
| Cover Design: | Lisa Tosheff |
| Printed at: | Yurchak Printing Inc. |

Published in the United States of America by
Idea Group Publishing (an imprint of Idea Group Inc.)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@idea-group.com
Web site: http://www.idea-group.com

and in the United Kingdom by
Idea Group Publishing (an imprint of Idea Group Inc.)
3 Henrietta Street
Covent Garden
London WC2E 8LU
Tel: 44 20 7240 0856
Fax: 44 20 7379 0609
Web site: http://www.eurospanonline.com

Product or company names used in this book are for identification purposes only. Inclusion of the names of the products or companies does not indicate a claim of ownership by IGI of the trademark or registered trademark.

Library of Congress Cataloging-in-Publication Data

Measuring information systems delivery quality / Evan W. Duggan and Han Reichgelt, editors.
p. cm.
Summary: "The book provides analyses and explains some of the contradictions and apparent paradoxes of many information systems quality perspectives"--Provided by publisher.
Includes bibliographical references and index.
ISBN 1-59140-857-1 (hardcover) -- ISBN 1-59140-858-X (softcover) -- ISBN 1-59140-859-8 (ebook)
1. Management information systems--Evaluation. 2. Information technology--Evaluation. I. Duggan, Evan W. II. Reichgelt, Han.
T58.64.M435 2006
658.4'038--dc22

2006003552

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

All work contributed to this book is new, previously-unpublished material. The views expressed in this book are those of the authors, but not necessarily of the publisher.

# Measuring Information Systems Delivery Quality

## Table of Contents

### Section I: Introduction and Overview of Quality Concepts and Dimensions

### Section II: Quality In the Early Stages of IS Delivery

## Section III: Process Contribution to IS Quality

## Section IV: Managing Risks of SPI Projects and Methodologies

# Foreword

Evan W. Duggan (University of Alabama, USA) and Han Reichgelt (Georgia Southern University, USA) state the following in the first chapter of this important book on the quality of information systems: "Despite the fact that the IS discipline is over a half a century old and numerous articles have been written about software quality, there is, as yet, not a great deal of convergence of opinions on approaches for attaining quality." Does this statement surprise you? Many global organizations have spent most of the past 20 years learning how to attain quality leadership in manufactured goods, the most basic of products. In the U.S., automotive companies are still playing catch up to Toyota and, most recently, Hyundai.

Since the Japanese quality revolution in the late 1970s, to the most recent invasion of quality goods from Korea and China, we are now just learning what it takes to plan, control, and improve the quality of physical goods, let alone software or systems.

Professors Duggan and Reichgelt have pulled together some of the latest research from capable authors to answer some of the most basic and important questions related to managing the quality of information and software systems. This book will become required reading for many including university and college students becoming exposed to quality management techniques in their university classes; quality professionals who have been looking for answers to IT quality problems; and IT professionals that design products and services to meet the dynamic quality requirements of their customers. In today's world, quality means "to attain the lowest level of failure, the highest level of quality at the lowest possible cost" as mandated by the customer. That is no small task in the IT world.

In 1985, the world was just beginning to see the benefits of software development and IT systems. At that time, while working as an internal quality leader at the Perkin Elmer Corporation, a manufacturer of analytical instrumentation, I was asked, “how do we apply the quality management techniques that we learned from Juran, Deming, and others to non-manufacturing goods like software?” I was then a babe in the field of quality compared to quality gurus such as Dr. Joseph M. Juran (who was then 76 years old and is now 105!) but I had to figure out what to do. There were few books to read, few published cases on lessons learned, and even fewer “champions” that we could consult to answer the question: “how do we manage the quality of software and systems?” We did what we could and what we did was not much. It was not that important then because many citizens of this planet had not yet experienced the Internet, e-mail, or other IT innovations.

Now it is 2006, and the scenario has not changed much for the quality of IT systems. We are still struggling to find answers to the question of how to manage information system quality. Now as citizens of this planet, we rely on information systems that we did not care much about in 1985 to do just about everything. Information systems quality is now a “big thing” and “it needs improvement”. The IT revolution will continue into the coming years: Many organizations are already seeing their internal IT infrastructures and people outsourced to larger more specialized companies; CIOs continue to change positions rapidly; Moore’s Law will still be applicable; and organizations will continue to seek to improve the quality of IT systems to stay ahead of competition, or worse, avoid obsolescence.

I am not a young lad anymore, nor am I as wise or as old as Dr. Juran but I have learned the secrets of attaining quality — good research, a high degree of collaboration among experts, and a need to change. All three are here. The research conducted in preparation for this book has provided answers to some of the most common questions about IT and software quality. The collaboration on the part of these top researchers and practitioners in the field lend credibility to understanding the lessons learned. The need for change in IT and systems quality is here.

Put all three together and business and society can reduce the pain of the system failures that plagued us in the last century. The Juran Institute has stated that we are moving into “the Century of Quality”. This century will be driven by the needs of society to have perfect quality. In the later part of the last century, we began to understand how to manage the quality of product. Now we have a better understanding of how to manage IT and software and systems quality. If we can figure out how to put the two (hardware and software quality) together, the citizens of this planet will enjoy this next century.

*Joseph A. De Feo , USA*

*President & CEO Juran Institute Inc.*

# Preface

After years of experience with building systems, the information systems (IS) community is still challenged by systems delivery, that is, planning the implementation project, determining system features and requirements, sourcing and deploying the software, and managing its evolution. High-quality systems are still elusive. Yet organizations invest heftily in IS to support their business operations and realize corporate priorities, either in search of competitive advantage or as a competitive necessity. One of the major paradoxes of our era is the disparity between the many innovations that have been enabled by information technology (IT) and the failure of the IS community, comprising developers, managers, and users, to exploit these advances to consistently produce high-quality IS that provide value to organizations. This phenomenon, which is highlighted by Brynjolfssen (1993), Gibbs (1994), and others, has been dubbed "the software crisis."

## The IS Quality Landscape

The effects of the software crisis are demonstrated in the number of projects that are abandoned before completion (The Standish Group, 2003), deployed with poor quality, consuming inordinate maintenance resources (Banker et al., 1998), or remain unused after implementation (Markus & Keil, 1994). The IS community has rightly focused on how to reverse the trend of low-quality systems, especially as IS become more central to the accomplishment of organizational mission. The quality drive in IS delivery is reminiscent of the intensified focus on quality in the manufacturing and service areas, and the IS discipline has built on many of the concepts articulated by Deming and others.

However, the systems delivery process has at least two key complicating factors that are absent from most other manufacturing processes. First, IS are not confined to single operations, but typically address a network of interdependent business processes and interfaces to other technical systems (Liu, 2000), and are embedded within the social systems of the organization (Robey et al., 2001). Second, while the erection of physical structures takes increasing physical shape as the construction progresses, software artifacts remain "invisible" throughout development (Brooks, 1987). Progress is therefore much harder to monitor, and systems delivery more difficult to manage.

It is little wonder that the pursuit of quality-enhancing approaches has taken many turns and has focused on a variety of factors. IS quality has been defined in various ways and encompasses several interpretations depending on the perspective of the particular stakeholder. To address the many perspectives, the IS community has variously focused on the contribution of people, delivery processes, and development philosophies (methods) to the attainment of defined quality attributes of intermediary and final software products.

User involvement in systems delivery facilitates the capture of the knowledge of appropriate domain experts, leads to greater commitment to system decisions and outcomes, and reduces the probability of usage failures (Barki & Hartwick, 1994). The IS community has also embraced the underlying logic of continuous process improvement programs, namely that process quality largely determines product quality (Deming, 1986). This has been substantiated in IS research (Harter et al., 1998, Khalifa & Verner, 2000; Ravichandran & Rai, 2000). Yet scholars have lamented that software process interventions are not well employed in IS organizations to improve quality (Fichman & Kemerer, 1997). Similarly, several systems delivery methods (design principles for driving specific development techniques), such as rapid application development, object-oriented development, and agile methods (Duggan, 2004), all purport to improve IS quality.

Despite the fact that the IS discipline is over a half a century old and numerous articles have been written about software quality, there is, as yet, very little convergence of opinions on approaches for attaining quality. In several cases, the same method attracts simultaneous claims of efficacy and ineffectiveness in different quarters. There is not even a commonly accepted definition for IS quality. This obviously affects agreement on important questions such as what are its determinants, what are the mechanisms by which quality is incorporated into the IS delivery process, and under what conditions are particular techniques likely to be successful.

## What Does This Book Offer?

This book presents an international focus on IS quality that represents the efforts of authors from several countries. Despite this diversity and the presentation of the uncoordinated research findings and objective, conceptual analyses of multiple dimensions of IS quality by several authors, the result is integrative. It reflects the common position that improving objective knowledge of potential quality-enhancing methods is far more likely to assist the production of high-quality software than wanton experimentation with each new "gadget." Hence, the editors and chapter authors share a common motivation to reduce the uncertainty and ambivalence about the efficacy of particular methods, the contexts and conditions under which they are effective, and what synergies might obtain from combined approaches.

The book therefore provides thoughtful analyses and insights to explain some of the contradictions and apparent paradoxes of the many IS quality perspectives. It offers prescriptions, grounded in research findings, syntheses of relevant, up-to-date literature, and verifiable leading practices to assist the assimilation, measurement, and management of IS quality practices in order to increase the odds of producing higher quality systems. In addition to both descriptive contributions (to elucidate and clarify quality concepts) and prescriptive solutions (to propose quality-enhancing approaches), there are also normative features to rationalize perceptive gaps and balance conflicting views.

## The Intended Audience

The book is intended to serve the diverse interests of several sectors of the IS community. It will be supportive of teaching, research, and application efforts in the increasingly important area of information systems delivery quality. It will both provide answers to enhance our state of knowledge and generate questions to motivate further research. In teaching, it may be used to provide supplementary material to support courses in systems analysis and design, software engineering, and information systems management.

This book will be useful for researchers in several ways. First, it provides an up-to-date literature review of this subject area. Second, it clarifies terminology commonly used in IS quality literature and generates some points of convergence; as noted by Barki et al. (1993), the absence of a common vocabulary and inconsistent use of terms in the IS literature severely constrains cumulative research. Third, most chapters suggest a variety of research questions and propose directions for future research for both seasoned researchers and doctoral students who intend to focus on IS process improvement and quality issues.

It is also valuable for IS practitioners by providing objective analyses of quality-enhancing methods and practices in context. Practitioners typically sift through a maze of claims and hype about the effectiveness of several techniques from their proponents and need useful information to evaluate these claims. This book helps by clarifying the muddy waters, explicating the quality requisites of a successful information systems, providing information for effective decision-making about appropriate process strategies and their fit to particular IS delivery settings, and assessing the strengths, weaknesses, and limits of applicability of proposed approaches.

# Organization of This Book

The 15 chapters of this book contain a mixture of conceptual and research papers, and are organized into five sections, each with a distinct focus:

- Section 1 introduces IS quality concepts, definitions, perspectives, and management techniques and issues.
- Section 2 discusses quality issues in the early and formative stages of an information system;
- Section 3 is devoted to the contribution of process-centricity to quality IS products.
- Section 4 describes and evaluates metamethods and frameworks for evaluating software process improvement (SPI) programs and process structuring methodologies.
- Section 5 analyzes quality issues in less researched applications and institutions.

## Section I: Introduction and Overview of Quality Concepts and Dimensions

The three chapters of Section I provide a general introduction to and overview of the critical definitions, concepts, perspectives, and nuances of IS quality and its determinants and measures. It lays the appropriate foundation and sets the tone for further examination of the concepts, issues, and controversies presented in later sections.

Chapter I provides a solid opening in its introductory overview, which covers the breadth of IS quality considerations to prepare readers for the more detailed analyses in subsequent chapters. It provides an overview of the threats to IS delivery and the quality of the final product and highlights the many perspectives and heterogeneity of IS stakeholders. Consequently, many approaches (sometimes uncoordinated) have been adopted to improve the quality, usage, and perceived value of IS within organizations. The authors undertake an extensive review of the IS quality literature to develop a general conceptual model of the elements — people, process, and practices — that contribute to software quality. They posit that the objective quality attributes of a system do not necessarily offer system success; the perceptions of users, however formed, are also influential. Beyond setting the tone for the topics that follow, Chapter I also makes its own contributions and provides some insights on IS quality issues that are not covered elsewhere in the book.

Chapter II provides an overview of the concepts and management issues that surround the measurement and incorporation of quality features into an IS development project. It provides a detailed review of the literature in this field in general, and of software quality, in particular. It also presents an overview of what to expect in the upcoming international standards on software quality requirements, which transcend the life cycle activities of IT processes. The chapter provides validation for a sub-theme of the book, namely that the concept of software quality cannot be defined abstractly, but that any adequate definition must take into account the application context of the software.

Chapter III examines several definitions of quality, categorizes the differing views of the concept, and compares different models and frameworks for software quality evaluation. The author emphasizes the notion that a particular view can be useful in varying contexts, and that a particular context may be better or worse served by varying views. The chapter examines both historical and current literature to provide a focused and balanced discussion of recent research on the Software Evaluation Framework (SEF). SEF gives the rationale for the choice of characteristics used in software quality evaluation, supplies the underpinning explanation for the multiple views of quality, and describes the areas of motivation behind software quality evaluation. The framework which has its theoretical foundations in value-chain models found in the disciplines of cognitive psychology and consumer research, introduces the use of cognitive structures to describe the many definitions of quality.

## Section II: Quality in the Early Stages of IS Delivery

Section II contains three chapters which examine various dimensions of IS quality, particularly in the early stages of the information systems life cycle. System failures resulting from poor deliverables in the early stages of development are pervasive (Dodd & Carr, 1994). It is generally acknowledged that quality, in the early stages of IS delivery, sets the tone for subsequent stages and determines the extent of design errors and rework (Byrd et al., 1992; Kang & Christel, 1992). It also tends to ripple through to the final product.

Chapter IV examines the problem of requirements defects, which remain a significant issue in the development of all software intensive systems including information systems, and reduces the quality of the product. The author asserts that progress with this fundamental problem is possible once we recognize that individual functional requirements represent fragments of behavior, while a design *satisfying* a set of functional requirements represents integrated behavior. The chapter offers a solution to the problem that accommodates this perspective by using behavior trees, formal representation for individual functional requirements, to construct a design *out of* its requirements. Behavior trees of individual functional requirements may be composed, one at a time, to create an integrated design behavior tree (DBT) to detect different classes of defects at various stages of the development process. This has significant quality-enhancing implications.

Chapter V builds on the theme that improper specification of systems requirements has thwarted many splendid efforts to deliver high-quality information systems and links this problem largely to poor communication among systems developers and users at this stage of systems development. It focuses on the devastating impact of user-developer miscommunication and the inadequacy of some of the models available for communicating specifications to users to obtain the degree of participation and high-quality feedback necessary for effective validation of systems requirements. The chapter presents an overview of both longstanding and newer requirements specification models and representational schemes for communicating specifications and evaluates their capability to advance user participation in this process and incorporate stated quality attributes. It also reports on preliminary evaluations of animated system engineering (ASE), the author's preferred (newer) technique, which adds animation and the capability to model dynamic and concurrent activities. The evaluation of ASE indicates that it has the potential to improve the effectiveness of requirements specification.

Chapter VI introduces the concept of scenarios (rich picture stories) to promote collaboration in the design of new information systems and provides insights into the opportunities available for the application of valid user scenarios. Systems designers need contextual information about their users in order to design and provide information systems that will function effectively and efficiently within those contexts. Storytelling or scenarios allow developers and users to create that all important "meeting place" that permits collaborative efforts in the design of new systems and richer information flows to raise the quality of the design. The chapter also explores the question of validation, which is one of the primary issues inhibiting the wider use of storytelling or scenarios in information systems development. The chapter demonstrates how the structured use of user-driven scenarios can assist practitioners in improving the quality of information systems design and encourages researchers to explore this interesting approach.

## Section III: Process Contribution to IS Quality

Credible evidence that sound systems development processes contribute to the delivery of better information systems (Harter et al., 1998; Kalifa & Verner, 2000; Ravichandran & Rai, 2000) provides the justification for organizations to invest in process improvement programs. Such programs assess the capability of an organization's IS processes and based on the results, define goals and plans to institutionalize documented standards and practices to guide the execution of repetitive development activities in order to reduce variability from one development project to the next. This section contains four chapters that address process contributions to IS quality.

Chapter VII explores the reasons for the persistent anxiety about low-quality IS, on the one hand, and the demonstrated relationship between systematic IS development processes and system success on the other. It presents a broad overview of IS process improvement concepts, applications, and contributions to IS delivery quality. The chapter synthesizes core facts from real world experiences, practitioner reports and anecdotes from current practice, and insights gleaned from scholarly research to provide a general exposition of the perspectives and issues surrounding this increasingly important and interesting topic. The chapter provides the foundation and background for the deeper analyses of various aspects of process-centered approaches in succeeding chapters of this section of the book. In addition, it makes a meaningful contribution to the ongoing debate about the relevance and potential of process-centricity in the delivery of high-quality IS by assessing the contexts in which quality-enhancing software process improvements can realistically thrive.

In Chapter VIII, the authors report on the results of a set of research projects that investigated the SPI movement against the background that the search for new ideas and innovations to improve software development productivity and enhance software system quality continues to be a key focus of industrial and academic research. They sought answers to the apparent disconnect between the extensive focus on SPI programs based on the premise that system development outcomes are largely determined by the capabilities of the software development process, and the slow diffusion and utilization of SPI initiatives in the software engineering community. They found that (1) software developers' perceived control over the use of a SPI impacts its diffusion success and (2) that software developers' perceptions of enhanced software quality and increased individual productivity achieved through the use of SPI impact the successful diffusion of the SPI. Results of these research efforts support the compilation of a clear set of management guidelines for the effective use of SPI initiatives in software development organizations.

Chapter IX introduces, defines, and elaborates on agile development methods and how quality information systems are created through the values and practices of people using agile approaches. The chapter covers key differences among agile methods, the SDLC, and other development methodologies and offers suggestions for improving quality in IS through agile methods. Following this objective analysis, the authors recommend adopting the principles of agile methods, propose several IS quality improvement solutions that could result, and raise future research issues and directions that could provide further insights into the contribution of agile methods to information systems quality. They also call for more education about the value of agile approaches and the expansion of their application to include more people, use in a variety of organizational cultures, and renaming agile methods to signify the value system inherent in the approach. The chapter provides excellent background coverage for Chapter X.

Chapter X presents an empirical assessment of the quality of the process of building software systems with agile development methods, which were designed to help with the development of higher quality information systems under given conditions. The research assessed eXtreme Programming (XP), one of the several agile development approaches. It compares XP with a traditional (design-driven) software construction process by observing and measuring the work of several student groups using different approaches to produce software for commercial companies during a semester. The data collected were analyzed following the Bayesian approach. The results indicate that that XP could cope with small to medium sized projects and delivered measurable improvement in the quality of the system as judged by business clients.

## Section IV: Managing Risks of SPI Projects and Methodologies

Section IV focuses on process contributions to IS quality and the several vehicles for accommodating SPI programs and institutionalizing streamlined software processes through system development methodologies (SDM). However, there are risks in the adoption and use of these methodologies; their success is highly dependent on how well these processes fit the organizational culture and can be interwoven into its social systems (Curtis et al., 1995; Perry et al., 1994). The two chapters in this section address risk mitigation approaches to adopting SPI programs and meta-methodologies for evaluating the effectiveness of methodologies respectively.

Chapter XI shows how action research can help practitioners develop IT risk management approaches that are tailored to their organization and the specific issues they face. Based on literature and practical experience, the authors present a generic method for developing risk management approaches for use in real-world software innovation projects. The method is based on action research into an organization's specific risk management context and needs. The chapter illustrates the method by presenting the results of the authors' experiences in developing the tailored approach to risk management in SPI projects at a large Danish bank.

Chapter XII discusses the evaluation of information systems development methodologies, which are considered cornerstones for building quality into an information system. If methodologies are indeed pivotal to system quality, then the effective evaluation of methodologies becomes crucial. This evaluation is usually achieved through the use of evaluation frameworks and metamodels, both of which are considered meta-methodologies. The chapter provides a comprehensive overview of how to construct efficient and cost-effective meta-methodologies and identify their quality attributes in a scientific and reliable manner. It reviews representative meta-methodologies and summarizes their quality features, strengths and limitations, and compares the functional and formal quality properties that traditional meta-methodologies and method evaluation paradigms offer in addressing properties such as computability and implementability, testing, dynamic semantics capture, and people's involvement.

## Section V: IS Quality Issues in Under-Researched Areas

The quality and SPI foci have been dominated by issues in a corporate context, where software is sourced through traditional means. The three sections in this chapter widen the perspective to include quality concerns in areas that are not as well discussed and researched. The first examines quality issues with open source software (OSS) development, which has grown from the preoccupation of hackers to viability as a software development alternative (Henderson, 2000). The others examine peculiar problems of quality in non-corporate organizations — general issues in government organizations and ERP implementation difficulties in a University environment.

Chapter XIII examines quality features of Open Source Software (OSS) processes against the background that these processes are not well researched.

The chapter describes the principles and processes used to generate OSS and their differences with processes used for proprietary software, the responsibilities of producer and consumer communities, and the protocols that govern their actions. The author then uses Bass et al.'s (2000) quality model that assesses software by attributes of the system such as performance, security, modifiability, reliability, and usability to explore the challenges these attributes pose to the open source development process and how the open source community measures up to these challenges.

Chapter XIV analyzes the peculiar challenges public agencies face in creating and sustaining IS quality. Unlike private sector firms, governments exist to meet mandated service requirements with limited ability to create revenue streams to fund IS programs that are essential to managing organizational bureaucracies and serving the public interest. The author provides a historical perspective of IS quality (with selected examples of both low- and high-quality systems) in the public sector, where high-profile failures have masked the relative successes of the majority of systems that work adequately within a complicated network of several interrelated governmental entities. The chapter denotes that the expectation of IS quality varies, but operational success over the long run may be considered the most highly valued feature of quality information systems in that sector. However, demands for productivity and innovation, increased responsiveness, and more sophisticated leadership may shape new quality profiles for the future.

Chapter XV presents a case study of the implementation of an ERP system in a large Australian university, viewed through the lens of Eriksson and Törn's (1991) Software Library Evolution (SOLE) quality model. The study explores the relationship between ERP systems capability and the quality requirements of individual users and highlights the problems encountered by organizations such as universities, where implementation failures of such mammoth systems could be relatively more disastrous than in corporations. The literature suggests other differences in the deployment of these systems compared to stand-alone systems, particularly in terms of the nature and timing of user involvement. Given these and other peculiar difficulties of non-traditional adoption of ERP systems, the authors refer to the lessons of this case to prescribe useful practices for such implementations and suggest some solid directions for similar research in the future.

## References

Banker, R.D., Davis, G.B., & Slaughter, S.A. (1998). Software development practices, software complexity, and software maintenance performance: A field study. *Management Science, 44*(4), 433-450.

Barki, H., & Hartwick, J. (1994). Measuring user participation, user involvement, and user attitude. *MIS Quarterly, 18*(1), 59-82.

Barki, H., Rivard, S., & Talbot, J. (1993). A keyword classification scheme for IS research literature: An update. *MIS Quarterly, 17*(2), 209-226.

Brooks, F.P., Jr. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer, 20*(4), 10-19.

Brynjolfssen, E. (1993). The productivity paradox of information technology. *Communications of the ACM, 36*(12), 67-77.

Byrd, T.A., Cossick, K.L., & Zmud, R.W. (1992). A synthesis of research on requirements analysis and knowledge acquisition techniques. *MIS Quarterly, 16*(3), 117-138.

Curtis, B., Hefley, W.E., & Miller, S. (1995). *Overview of the people capability maturity model* (Tech. Rep. No. CMU/SEI-95-MM-01). Carnegie Mellon University, Software Enginering Institute.

Dodd, J.L., & Carr, H.H. (1994). Systems development led by end-users. *Journal of Systems Management, 45*(8), 34-40.

Duggan, E.W. (2004). Silver pellets for improving software quality. *Information Resources Management Journal, 17*(2), 1-21.

Eriksson, I., & Törn, A. (1991). A model of IS quality. *Software Engineering Journal, 6*(4), 152-158.

Fichman, R., & Kemerer, C. (1997). The assimilation of software process innovations: An organizational learning perspective. *Management Science, 43*(10), 1345-1363.

Gibbs, W.W. (1994). Software's chronic crisis. *Scientific American, 271*(3), 86-95.

Harter, D.E., Slaughter, S.A., & Krishnan, M.S. (1998). The life cycle effects of software quality: A longitudinal analysis. *Proceedings of the International Conference on Information Systems* (pp. 346-351).

Henderson, L. (2000). Requirements elicitation in open-source programs. Hill Air Force Base Software Technology Support Center (STSC): *CrossTalk, 13*(7).

Kang, K.C., & Christel, M.G. (1992). *Issues in requirements elicitation* (Tech. Rep. No. SEI-92-TR-012). Pittsburgh: Carnegie Mellon University.

Khalifa, M., & Verner, J.M. (2000). Drivers for software development method usage. *IEEE Transactions on Engineering Management, 47*(3), 360-369.

Liu, K. (2000). *Semiotics in information systems engineering*. Cambridge, UK: Cambridge University Press.

Markus, M.L., & Keil, M. (1994). If we build it they will come: Designing information systems that users want to use. *Sloan Management Review, 35*(4), 11-25.

Perry, D.E., Staudenmayer, N.A., & Votta, L.G. (1994). People, organizations, and process improvement. *IEEE Software, 11*, 36-45.

Ravichandran, T., & Rai, A. (2000). Quality management in systems development: An organizational system perspective. *MIS Quarterly, 24*(3), 381-415.

Robey, D., Welke, R., & Turk, D. (2001). Traditional, iterative, and component-based development: A social analysis of software development paradigms. *Information Technology and Management, 2*(1), 53-70.

Standish Group, The. (2003). *Chaos – the state of the software industry.* Retrieved December 1, 2005, from http://www.standishgroup.com

***Evan W. Duggan***
***Han Reichgelt***

# Acknowledgments

We would like to acknowledge the contributions of the impressive array of international scholars who provided chapters for this book. In particular, we appreciate their thoughtful analyses of profoundly important topics to the software quality and information systems communities, the insights they have shared, the diligence with which they responded to the reviewers' recommendations, and their usually prompt responses to our tight deadlines. By their scholarly efforts they have exemplified the significance of quality — the theme of this book. We are also indebted to the reviewers whose invaluable contributions of time and expertise challenged the authors toward higher levels of excellence and made this a splendid achievement. This book could not have been completed without their generosity and certainly not with the same quality without their diligence.

We are tremendously indebted to the staff of Idea Group Inc. for their concerted efforts, which enabled the delivery of this book. First, to Medhi Khosrow-Pour for planting the idea and encouraging us to overcome the inertia to get it started. Then to the editorial staff, who demonstrated the highest professionalism in guiding us toward effective aims. However, no usual praise would be sufficient to recognize the enormous contribution of Kristin Roth, our development editor, who gave unstintingly of her time and energy to guard the integrity of the process by keeping us focused and on target, assuring the quality of intermediate deliverables and, ultimately, of the entire book. This accomplishment is also hers!

Finally, we are honored that Joe De Feo, President and CEO of the Juran Institute, endorsed our effort by graciously agreeing, on short notice, to contribute the Foreword for this book.

*Evan W. Duggan, University of Alabama*

*Han Reichgelt, Georgia Southern University*

# Section I:
# Introduction and Overview of Quality Concepts and Dimensions

## Chapter I

# The Panorama of Information Systems Quality

Evan W. Duggan, University of Alabama, USA

Han Reichgelt, Georgia Southern University, USA

## Abstract

*Business organizations are still struggling to improve the quality of information systems (IS) after many research efforts and years of accumulated experience in delivering them. The IS community is not short on prescriptions for improving quality; however the utterances are somewhat cacophonous as proponents of quality-enhancing approaches hyperbolize claims of their efficacy and/or denigrate older approaches, often ignoring the importance of context. In this chapter we undertake an extensive review of the IS quality literature to balance the many perspectives of stakeholders in this heterogeneous community with the necessarily varied prescriptions for producing high-quality systems. We develop an IS quality model, which distills determinants of IS product quality into effects attributable to people, processes, and practices and denote that IS success results from the* *combination of discernible IS quality and stakeholders' perceptions of IS quality. This chapter serves as a general introduction to the detailed analyses of topics that follow in subsequent chapters but also provides insights that are not covered elsewhere in the book.*

## Introduction

The crusade to effectively confront information systems (IS) quality problems is as persistent as the problems themselves and the issue has been on the radar screens of IS researchers and practitioners for a long time (Bass et al., 2003; Dromey, 1996; Duggan, 2004a; Floyd, 1984; Floyd et al., 1989; Harter et al., 1998; Kautz, 1993; Khalifa & Verner, 2000; Kitchenham, 1989; Meyer, 1988; Miles, 1985; Rae et al., 1995; Ravichandran & Rai, 2000). Systems builders, whether developing customized products for proprietary use or generalized commercial packages, have long contended with the challenge of creating sophisticated software applications of high-quality, with the requisite features that are useable by their clients, delivered at the budgeted cost, and produced on time. The dominant experience, however, is that these goals are not frequently met; hence, the recurring theme of the past several years has been that the IS community has failed to exploit IT innovations and advances to consistently produce high-quality business applications. This apparent paradox has been dubbed the "IS crisis" (Brynjolfsson, 1993; Gibbs, 1994).

This perceived crisis manifests itself in a variety of ways. For example, many systems delivery projects are initiated without adequate planning and with unresolved feasibility concerns (Hoffer et al., 2002). Some are aborted before deployment, and others are implemented with significant errors due to faulty design and inadequate testing (Butcher et al., 2002). Ineffective process management is also prevalent (Gladden, 1982) and some successfully deployed IS consume a disproportionate share of organizations' development resources for maintenance activities (Banker et al., 1998). Even technically sound systems are not immune to failure; some remain unused because of unfavorable user reactions (Lyytinen, 1988; Markus & Keil, 1994; Newman & Robey, 1992).

The list below is a chronological sample of twenty years of negative scholarly and practitioner commentary on various IS problems. Although the entire list does not pertain to IS quality — which we will define shortly — it depicts a range of issues that gives credence to the perception of a crisis:

- IS development is fraught with recurrent problems caused by poor, undisciplined, and incomplete development practices (Gladden, 1982)

- 75 percent of all systems development undertaken is never completed or, if completed, remain unused (Lyytinen, 1988)
- Runaway IS delivery projects — grossly over budget and hopelessly behind schedule — are prevalent (Mousinho, 1990)
- The improvement of IS delivery quality is one of the top ten IS management problems (Niederman et al., 1991)
- KPMG Peat Marwick found that over 50 percent of IS failed to deliver expected benefits (*Computerworld*, April 25, 1994)
- According to Charles B. Kreitzberg, President of the Cognetics Corporation, as quoted in the March 1, 1997 issue in the CIO Magazine, CIOs estimated the failure rate for IS projects at 60% (CIO, 2001)
- Only 24 percent of IS projects in Fortune 500 companies are successful (The Standish Group, 2002)
- The cost of systems downtime is estimated by the Standish Group at $10,000 per minute for mission-critical applications (Schaider, 1999)
- Approximately 50 percent of IS executives expressed dissatisfaction with the quality of their organization's business software (CIO, 2001)
- Systems maintenance consumed 60 to 80 percent of organizational resources allocated to IS delivery and evolution (Hoffer et al., 2002)

There are also assorted perspectives on IS quality, some rooted in the subjective appraisal of various stakeholders based on their interests and not necessarily on objective properties of the system. Executive sponsors of IS typically view quality in terms of return on investments and other business value to the organization. Functional managers judge the operational benefits of the system and its support for decision making. End users are mostly concerned with usability — ease-of-use and the capability to enhance their personal effectiveness. System designers and builders fixate on conformance to specifications and, according to Bandi et al. (2003), IS support groups agonize over maintainability — the ease and speed with which systems can be understood and modified.

In this chapter, we explore these quality perspectives broadly (if not in depth) to set the tone for deeper analyses in the conceptual expositions and research pieces in the chapters that follow. Particularly, we frame the discussion above the fray of hype, advocacy, or implied allegiance to any particular approach. However, beyond this introduction we have three other aims: (1) to explicate the range of quality considerations by offering objective analyses based on a balanced overview of the IS quality literature — empirical observations and impartial practitioner accounts; (2) to clarify ambiguous terminology; and (3) to identify quality-enhancing practices and indicate how they contribute to IS

*Table 1. Definition of key systems delivery terms*

| *Concept* | *Description* |
|---|---|
| Information systems (IS) delivery | All the activities involved in the overall planning, analysis, sourcing and deploying an information system. The term "delivery" is used instead of "development" to account for all the possible methods (in-house development, purchase, outsourcing, rental, etc.) of sourcing an information system |
| IS delivery paradigm | Commonly distinguishing features of a family of life cycle models driven by some overarching delivery objective (e.g., Waterfall SDLC model, Reuse-based models) |
| IS deployment | The process of transferring a completed system from its development environment into the operational environment |
| Software engineering | The application of technologies and practices to the management and creation of software components |
| Software production method | Principles and methods adopted for implementing particular software engineering philosophies and objectives (e.g., Rapid application development; extreme programming; component-based development) |
| Systems development methodology | Comprehensive set of principles for structuring the delivery process that describes what, when, and how activities are performed, the roles of participants, and the supported methods and tools (e.g., Information Engineering; Method 1; Navigator, RUP) |

quality and success. Generally, this book prescribes approaches designed to increase the probability of overcoming the pervasive problem of low-quality IS deployment; this chapter leads the way.

Unlike older business disciplines, the relatively new IS field does not yet enjoy consensus on the definitions of many of the important constructs that are regularly discussed and researched. This absence of a common vocabulary has led to inconsistent use of terms in the IS literature and hampered cumulative IS research (Barki & Hartwick, 1994; Barki et al., 1993). Table 1 therefore defines the meanings of the key concepts we discuss in this chapter that are pivotal to the assimilation of the points we intend to make and to general semantic consistency.

In the following sections, we analyze the salient factors that account for the IS quality challenges as a basis for developing a model of IS quality, which distinguishes between IS quality and IS success. We then use the model to structure the discussion of the contributors — people, processes, and practices — to IS product quality, which in turn is moderated by stakeholders' perceptions to determine IS success. We conclude with recommendations for further analyses of some of the issues that are not detailed in subsequent chapters of the book.

# The Nature of IS Delivery Challenges

The problem factors — conformity, changeability, invisibility, and complexity — that Brooks (1987) attributed to the essential difficulties of delivering intangible IS products have not disappeared and, indeed, several other IS delivery complexities have emerged since (Al-Mushayt et al., 2001; Chen et al., 2003). Nevertheless, organizations rely increasingly on advanced IS to support a wider array of functional operations and enable strategic priorities. Consequently organizational systems have become more ubiquitous; they cover a wider spectrum of business operations and are more integrative in order to exploit the cross-functional dependencies of work-systems across organizational value chains.

## Technological Challenges

Several converging factors have also increased IS delivery complexities (Duggan, 2005). For example, there are more sophisticated enterprise systems and several distributed application architecture choices and data distribution strategies. Novel software production methods, such as the dynamic systems development method (DSDM) and extreme programming (XP), have emerged. Commercial off the shelf (COTS) software acquisitions have replaced in-house development as the dominant IS sourcing method (Carmel, 1997), while outsourcing of some or all of IS delivery is increasing and rental through application service providers (ASPs) is now an option for smaller organizations.

Waves of new technology and other innovations have presented organizations with new options for, and greater flexibility in structuring their operations, facilitating e-commerce and the emergence of virtual corporations, and providing support for telecommuting. Business organizations have also exploited political initiatives that have created regional trading alliances to extend their global reach and increase collaborative business ventures. Multinational and transnational corporations have launched multi-organizational IS projects, which have given rise to virtual teams (Sarker & Sahay, 2003) and the need for global systems to support business process operations within and across national borders (Boudreau et al., 1998). This greater reliance on IT increases the urgency of delivering quality systems (Vessey & Conger, 1994) and more ambitious goals elevate the risks of IS failure (Kirsch, 2000).

## Challenges During the IS Delivery Cycle

The system life cycle (SLC) consists of activities that span the period between the inception and the retirement of a system. Typically, it is broken down into several stages, the end of each marking a significant milestone. However, there is no agreement about how many stages SLC comprises — depending on the source of information, the number of stages could range from two to as many as 15. We have therefore concocted a generic life-cycle model to avoid the possible controversy of using any of the available models. The manner in which the activities at the various stages of the SLC are conducted influences the quality of the deliverables at each stage as well as the quality of the eventual system (Hoffer et al., 2002). Table 2 summarizes the dominant influences on the outcomes of each of the four generic stages we have chosen.

Effective stakeholder interaction and collaboration are acknowledged as critical success factors during the conceptualization stage of the life cycle to determine the feasibility of alternative approaches, agree on the scope and terms of reference of the project, and determine the features that should be provided. The effectiveness of the collaboration at this stage has both a direct impact on the quality of the eventual product (Barki & Hartwick, 1994) and a ripple effect on the success of deliverables at other stages.

Similarly, we propose that the effectiveness of the delivery process, the practices employed, and the software production method adopted (if applicable) are the dominant impacts on quality during the creation stage. Here the system is sourced through one of the following methods: In-house development (in-sourcing), which involves design and construction at this stage; COTS acquisition, which invokes a set of procedures for maximizing the fit between the requirements of the demanding organization and the capability of the available software packages; or outsourcing, where the design and construction is contracted.

*Table 2. Stages of a system life cycle: The Four Cs Model*

| **Activity** | *Conceptualization* | *Creation* | *Consummation* | *Consolidation* |
|---|---|---|---|---|
| *Meaning:* | Obtaining information for planning and determining a feasible solution | Determining the structure of the system and producing it | Deploying the finished system | Adapting and perfecting the system |
| *Dominant Impact* | Stakeholder interaction | Systems delivery process and software production methods | The adoption propensities of users and their perceptions of the quality of the product | The objective quality of the product |

During consummation the system is deployed for ongoing usage. While organizations justify IS investments on the basis of promised business value, the realization of those benefits are predicated on successful consummation — effective adoptive behavior of IS users that leads to acceptance and use (Silver et al., 1995). Although the adoption of organizational systems is not voluntary, potential users may engage in passive resistance and subtle rejection (Hirschheim & Newman, 1988; Marakas & Hornik, 1996) through avoidance, circumvention, or sabotage.

At the consolidation stage, the information system evolves in use as corrective and perfective modifications are identified and implemented over its useful life. The goal is to maintain or improve the quality of the system by keeping it attuned to the business operations it enables (Floyd, 1984), prevent degradation due to atrophy (i.e., deterioration for lack of sustenance), and delay entropy — and impairment due to natural decay. IS stability may be attained by the extent to which the former is avoided and the latter delayed (Duggan, 2004b).

## Towards a Model of IS Quality

Many of the basic concepts of IS quality were borrowed from the literature on process improvement and product quality (and their relationship) in manufacturing environments. There, process thinking evolved through concepts of statistical quality control, total quality management, continuous process improvement, business process (re)engineering, process innovation, and six sigma approaches.

Garvin's (1984) categorization of dimensions of the quality of physical products provides a useful starting point for our reflection on IS quality. He alluded to five perspectives: (1) the transcendental dimension of quality that concentrates on recognizable but not necessarily definable properties of a product; (2) the product dimension, which focuses on measurable characteristics; (3) the user-based dimension rooted in the notion of fitness for use and the satisfaction of stated or implied needs; (4) the manufacturing-based dimension that addresses conformance to specification; and (5) the value-based view that bears on the generation of measurable business benefits.

Quality evaluations in IS involve similar considerations but include other factors that address legions of perspectives, which do not have equivalent importance in manufacturing: support for interdependent business processes, interfaces with social systems, alternative technological choices, a variety of process structuring practices, and several available software production methods. These considerations have spawned many perspectives and quality initiatives. There is some agreement, though not consensus, that (1) pursuing IS process improvement is an important organizational objective; (2) a quality process leads to a quality product; and (3) that high-quality IS are generally elusive. Beyond these, there is considerable diversity of views about what constitutes a quality system and how to measure it (Ravichandran & Rai, 1994).

The diversity of IS stakeholders and their varied interests also lead to multiple views of quality and quality evaluation and attainment. Furthermore, stakeholders' perspectives are not always informed by rational system objectives; some perspectives are politically motivated (Robey & Markus, 1984). Nevertheless, Palvia et al. (2001) interpreted IS quality as discernible features and characteristics of a system that contribute to the delivery of expected benefits and the satisfaction of perceived needs. Other scholars, such as Erikkson and McFadden (1993), Grady (1993), Hanna (1995), Hough (1993), Lyytinen (1988), Markus and Keil (1994), Newman and Robey (1992), have further explicated IS quality requisites that include:

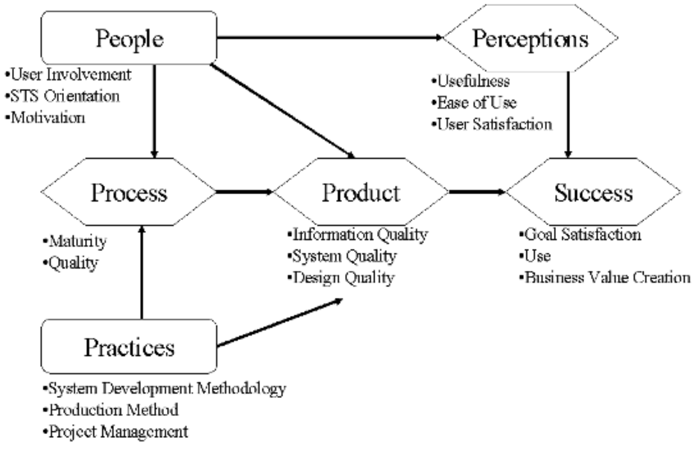
- Timely delivery and relevance beyond deployment
- Overall system and business benefits that outstrip life-cycle costs
- The provision of required functionality and features
- Ease of access and use of delivered features
- The reliability of features and high probability of correct and consistent response
- Acceptable response times
- Maintainability — easily identifiable sources of defects that are correctable with normal effort
- Scalability to incorporate unforeseen functionality and accommodate growth in user base
- Usage of the system

These quality requisites denote considerations along several dimensions. They reflect the heterogeneous perspectives of multiple IS stakeholders — developers, managers/sponsors, and users — and involve both technical and social dimensions (Newman & Robey, 1992; Robey et al., 2001), touching cultural and political issues, and instrumental (task-related) and affective (behavioral) considerations. Consequently, there are multiple prescriptions for quality attainment. We also recognize that there are other related, though not identical, concepts such as IS success which may or may not be determined by the objective quality of the system.

The quality model (Figure 1) depicts this multidimensional effect of several forces that collectively influence IS quality and success. It denotes that the IS delivery process and its product are both impacted by the process management practices employed and by people. Practices include the existence and usefulness of a systems development methodology (SDM), the appropriate choice of a software production method, and the effectiveness of project management — the guardian of the faithful application of the SDM and production method (Duggan, 2005). People issues encompass the degree of user involvement, the application of socio-technical systems principles, and the motivation of participants in the delivery process (Al-Mushayt et al., 2001; Palvia et al., 2001).

While these forces have considerable impacts on the quality of the delivered IS product, people individually and collectively generate perceptions of IS quality, sometimes independently of objective quality indicators. Therefore the model also denotes that IS success is not always positively correlated with IS quality; it may sometimes be partially — and occasionally wholly — determined by user perceptions. In the ensuing discussion, we use this model as a basis for organizing the discussion of the contributions of people, process, and practices to product quality and further elaborate on the distinction we make in the model between IS quality and IS success.

*Figure 1. Information systems quality model*



## People Dimensions of IS Quality

We have indicated that the definition of IS quality is modulated by the perspectives of its several stakeholders. IS quality is similarly impacted by the contributions of stakeholder groups — managers and sponsors, users, and technocrats (Vessey & Conger, 1994). Each stakeholder group can make unique contributions but it is almost axiomatic that effective collaboration among the communities is a significant determinant of system success.

Beyond the provision of financial sponsorship for IS investments, executive management and high-level leadership are further required to transform organizations into high-performance systems units, capable of delivering high-quality IS with consistency (Conradi & Fuggetta, 2002). The Standish Group (2002) found that proactive executive involvement in IS project delivery is one of the critical factors for successfully managing the changes in workforce practices and orchestrating the cultural shifts necessary to accommodate sustainable improvements in IS delivery practices.

There is some consensus that user-centered IS delivery approaches increase the odds of producing higher quality systems (Al-Mushayt et al., 2001; Barki & Hartwick, 1994; Doherty & King, 1998; Newman & Robey, 1992; Palvia et al., 2001; Robey et al., 2001). Such approaches encourage the adoption of sociotechnical systems (STS) principles and user involvement in IS delivery processes. STS principles emphasize the importance of joint attention to, and the cogeneration of requirements for both the technical and social environment into which IS are delivered (Al-Mushayt et al., 2001) and the assessment of the social and organizational impact of alternative technical solutions (Doherty & King, 1998).

User involvement with IS delivery is considered to be a prerequisite for producing high-quality systems because it maximizes the inputs of domain experts and other business process participants in IS decisions and generates affiliation with the process and outcome (Barki & Hartwick, 1994). However, research findings have been inconclusive about this intuitively appealing relationship (Carmel et al., 1995). Barki and Hartwick attribute these mixed results to the invalid operationalization of user association constructs; often the same terms are used to describe distinctly different concepts. For example, user participation and user involvement are used interchangeably. Duggan and Duggan (2005) identified four different gradations of user association constructs, each indicating a higher level of user commitment and effectiveness:

- **User participation:** intellectual involvement with completing assigned tasks;

- **User involvement:** psychological involvement and a sense of affiliation with the project;
- **User ownership:** the level of involvement that motivates pride of possession and a sense of responsibility for the success of the system;
- **User championship:** the state which produces "missionary" zeal in promoting the system and encouraging adoption.

Competent IS specialists with the skills and experience necessary to manage both the technical and behavioral elements of IS delivery are central to the successful delivery of high-quality IS products (Perry et al., 1994). IS project managers are the guardians of the IS process management structure and its faithful application. They have the crucial responsibility of coordinating the resources for realizing project objectives within approved budgets and timeframes, resolving problems, and identifying risks that are inimical to quality outcomes (Duggan, 2005). Nowadays, the project management office (PMO) — a repository of project management knowledge — has been established to provide contingency responsiveness, particularly for large and complex projects that require coverage beyond the cognitive capability of a single project manager (Lee & Xia, 2002).

Several user-centered design philosophies have been applied to influence IS delivery and product quality. They include prototyping (Davis, 1992; Weinberg, 1991), joint application development (Davidson, 1999; Duggan & Thachenkary, 2004; Kettelhut, 1993; Wood & Silver, 1995), participatory design (Gill & Kreiger, 1999; Kjaer & Madsen, 1995; Mankin & Cohen, 1997; Muller et al., 1993), effective technical and human implementation of computer systems ETHICS (Hirschheim & Klein, 1994; Mumford, 1983, 1993), and software quality function deployment (SQFD) (Elboushi, 1997; Erikkson & McFadden, 1993; Martins & Aspinwall, 2001; McDonald, 1995).

## Process Management Dimensions of IS Quality

A large body of evidence supports the claim that IS quality is positively correlated with the quality of the delivery process (Harter et al., 1998; Khalifa & Verner, 2000; Ravichandran & Rai, 1996, 2000). Yet there is recognition that a quality process is not a sufficient precondition for producing high-quality systems (Rae et al., 1995). Hence, many organizations invest upstream in two generic IS process improvement approaches: (1) formal process assessment instruments for measuring their IS delivery capability and (2) process management standards for implementing improvements identified in step one (Humphrey, 1988). The capability maturity model (CMM) as well as other similar instruments —

software process improvement capability determination (SPICE) (Melo et al., 1998) and Bootstrap (Kuvaja et al., 1994) — have been adopted to effect the former, while SDMs are used to implement the latter.

## *CMM/CMMI*

The CMM was developed by the Software Engineering Institute (SEI) of Carnegie Mellon University under the sponsorship of the Department of Defense to promote stability in the IS delivery process and reduce variability (SEI, 2002). The first such model developed was the software capability maturity model (SW-CMM), which defined process characteristics along a continuum consisting of five levels of maturity. For each level, it identified the practices that signify their attainment (Humphrey et al., 1991). Other models — the systems engineering capability maturity model (SE-CMM), systems engineering capability assessment model (SECAM), software acquisition CMM, and people CMM — have been developed since. These models have been combined into the capability maturity model integration (CMMI) (SEI, 2002), which has six levels of capability instead of five.

Evidence from researchers (Ashrafi, 2003; Goldenson & Gibson, 2003; Hollenbach et al., 1997; Subramanyam et al., 2004; Yamamura, 1999; Zahran 1998) and practitioners (CIO, 2001) suggest that IS delivery process improvements have produced benefits such as better scheduling, lower costs, higher delivery productivity, fewer defects, shorter delivery cycle time, less rework, more maintainable products, and greater conformance to specifications. However, there are also misgivings (Fayad & Laitinen, 1997; Gray & Smith, 1998; Leung, 1999; Pfleeger, 1996) about the massive implementation effort, ambiguous prescriptions, and the inability to link performance to assessed levels.

## *Stabilizing the IS Delivery Process*

A systems development methodology (SDM) is a comprehensive set of standards that specifies the process to be used for producing each deliverable across the development stages of IS. It prescribes the timing of each deliverable, procedures and practices to be followed, tools and techniques that are supported, and identifies roles, role players, and their responsibilities (Riemenschneider et al., 2002; Roberts et al., 1998; Roberts et al., 1999). Its target is process consistency and repeatability as IS projects advance through the systems life cycle. Many commercial and proprietary SDMs exist — for example, Information Engineering, Method 1, Navigator, Rational Unified Process and System Wyrx; however, in order to ensure the required stability, an organization should adopt a single SDM to which all systems delivery participants are trained (Duggan, 2004a).

SDMs help to eliminate project start-up inertia associated with deciding the process structure that should govern that project and/or the delivery phases, and permit more flexible deployment and redeployment of people to projects, thereby mitigating the risk of project failure due to attrition. However, the rigor they impose may also be onerous, frustrating some development efforts that do not require high structure (Fitzgerald, 2000; Flatten, 1992). The SDM learning curve is also particularly steep (Dietrich et al., 1997).

## *Software Production Methods*

Software production methods such as rapid application development (RAD) or object oriented development and SDMs, described previously, are distinct concepts that are often used interchangeably or with overlapping implications. The misapplication of these terms leads to miscommunication and confusion and undoubtedly affects our ability to interpret research results in this area. Recall that SDMs are process structuring standards for maintaining the consistency of IS delivery processes; software production methods are approaches for implementing particular philosophies that guide the engineering and creation of the software components.

Robey et al. (2001) suggest that systems delivery efforts are guided by particular delivery paradigms (defined in Table 1), such as:

- the waterfall model, which demands the complete prespecification of requirements before system design and progresses through IS delivery stages sequentially;
- incremental/iterative models that promote the "growing" of systems by progressively and /or iteratively delivering increments of software functionality;
- reuse based models that are geared toward producing well constructed IS artifacts that may be reused as is or with modification in other environments with similar requirements.

While some software production methods were developed within the constraints of given paradigms — for example, RAD under iterative rules and object-oriented development for reuse — SDMs typically support IS delivery under any paradigm (Duggan, 2004a).

The best-known systems delivery paradigm is the SDLC model, which is often called the waterfall method. It is characterized by the grouping of life cycle methods into related activities (called phases) and the sequential execution of these phases. Another of its features is the complete prespecification of information requirements. Robey et al. (2001) identified the iterative/incremental and the software reuse paradigms as two other IS delivery models that have addressed some of the deficiencies of the waterfall method and may enhance quality under given circumstances.

The iterative/incremental delivery paradigm describes a class of systems delivery methods that concentrates on techniques that produce manageable pieces of a system iteratively, and/or deliver functionality incrementally (Bandi et al., 2003). This paradigm includes production methods, such as:

- the spiral model, which combines the sequential development philosophy of the waterfall model and prototyping to deliver software in an evolutionary development approach that "grows" software in several rounds (or iterations) by "spiraling" repeatedly through a sequence of IS delivery activities, including the development of objectives for the current iteration, risk assessment and resolution, engineering and evaluation of the new component, and planning for the next spiral (Boehm, 1988);
- rapid application development (RAD), which involves high user-developer interaction to produce systems expeditiously (time boxing) by fixing development time and varying scope and quality (Bourne, 1994; Hanna, 1995; Hough, 1993);
- cleanroom software engineering (CSE), which seeks to generate defect-free IS with statistically certified reliability by employing independent specification, development, and certification teams, conducting peer reviews, and using inferential statistics to test the error-free hypothesis; erroneous modules are discarded, not modified (Hausler et al., 1994; Head, 1994; Jerva, 2001; Spangler, 1996; Trammell et al., 1996);
- agile development methods — Extreme Programming (XP), Adaptive Software Development, Feature Driven Development, Crystal Clear Method, Dynamic Systems Development Method (DSDM), Scrum, and others — depend on intense user-developer interaction to expedite software delivery by producing small pieces of functionality (releases) at regular intervals (Beck, 2000; Glass, 2001; Hoffer et al., 2002; Paulk, 2001).

The reuse paradigm focuses on producing high-quality reusable objects and components — packets of functionality — that are meant to reduce the scale-

*Table 3. Selected delivery methods*

| Production Methods | Delivery Paradigm | Features | Target |
|---|---|---|---|
| SDLC | Waterfall | Sequential delivery process, prespecification of system requirements | Deliverable tracking through sequential flow, milestone management, project control |
| The Spiral Model | Iterative | Combines elements of Waterfall method and prototyping; emphasis on risk assessment and mitigation | Complexity reduction, improved requirements, development productivity |
| RAD | Iterative | Fast delivery of functionality, time boxing, variable scope, uses prototyping | Small applications without computational complexity, delivery speed |
| CSE | Incremental | Small independent teams, statistical quality control, product certification | Defect prevention, production of failure-free software in field use, complex projects with low fault tolerance |
| Agile Methods | Iterative/ Incremental | Process agility and simplicity, requirements based initially on partial knowledge, functionality delivered in small releases, small co-located development teams, pair programming, continuous testing, on-site customer | Environments of high business process volatility |
| OOD | Reuse | Encapsulation, design systems as collections of interacting objects, development rigor, models that are usable in several delivery stages | Complexity reduction |
| CBD | Reuse | System delivery by assembling independent, platform-neutral components, interface standards, flexibility | Complexity reduction, domain expert involvement with IS delivery |

related complexities of large systems (Szyperski, 1998). This paradigm employs the following production methods:

- object-orientated development (OOD), in which software is modeled and designed as collections of interacting objects using the philosophy of encapsulation — packaging data with the program procedures that manipulate them and hiding the packaged information from other objects (Brown, 2002; Johnson et al., 1999; Schmidt & Fayad, 1997; Vessey & Conger, 1994);
- component-based development, a production (assembly) method that delivers software by (1) selecting suitable modules of software functionality from an inventory of independently produced, interoperable software components and (2) assembling them into a functional system (Szyperski,

1998). Systems may therefore be constructed by business domain experts rather than by traditional developers (Robey et al., 2001). Table 3 provides a summary of selected software production methods.

# IS Product Quality and Successful Systems

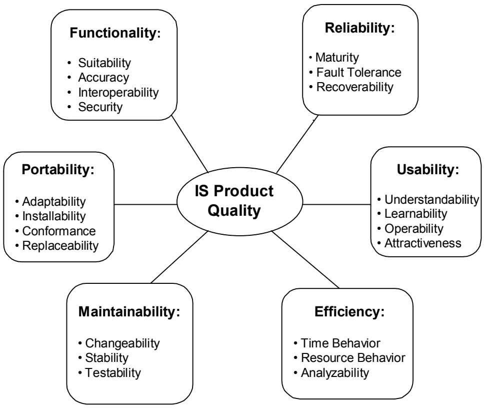
## Attributes of Product Quality

IS product quality is concerned with inherent properties of the delivered system that users and maintenance personnel experience. The ISO/IEC 9126-1 quality model is one of the popular frameworks for assessing the quality attributes of software. The model is presented as a hierarchical organization of six discernible characteristics — functionality, reliability, usability, efficiency, maintainability, portability — of both internal (as measured by the satisfaction of requirements) and external (as demonstrated by the system in execution) quality. Each quality characteristic is divided into subcharacteristics, as denoted in Figure 1, which may themselves be further decomposed (ISO/IEC 9126-1, 2001).

## IS Quality vs. IS Success

It is intuitive to presume that the delivery of high-quality IS naturally leads to IS success. However, the garbage heap of failed information systems contains many brilliantly conceived and technically sound systems (Duggan, 2004b);

*Figure 2. ISO 9126 characteristics and subcharacteristics of IS quality*



alternatively, some systems of poor quality are considered successful. Systems success is not necessarily determined by objective system quality and technical validity alone but also by the perceptions of stakeholders, which shape their behavioral responses (Lyytinen, 1988; Markus & Keil, 1994; Newman & Robey, 1992). We have therefore distinguished between IS quality and IS success as distinct constructs in our model and conceptualized IS success as the extent to which IS satisfy functional and usage requirements and deliver the desired business value, while overcoming both objective and perceptual impediments to adoption and use.

The results of research devoted to the evaluation of IS success in organizations have been inconclusive (Gable et al., 2003; Seddon, 1997). They portray successful IS as having positive organizational impacts (Barua et al., 1995; Barua & Lee, 1997; Brynjolfsson & Hitt, 1996; Lehr & Lichtenberg, 1999; Mukherjee, 2001) but also denote less favorable systems with neutral or even negative contributions (Attewell & Rule, 1984; Brynjolfsson & Yang, 1996; Quinn & Cameron, 1988; Wilson, 1993). DeLone and McLean (1992) attributed these mixed results to the many indicators used to assess IS success. They identified six generic determinants of IS success that include both objectively ascertainable properties of the system and the information it produces, and perceptual indicators of success such as user satisfaction and other individual impacts. Our definition of IS success represents a distillation of various measures into a few key benchmarks such as whether the system (1) is deployed with the required functionality to solve the problem at hand (Swanson, 1997); (2) provides value to the organization that deploys it (Brynjolfsson & Hitt, 1996); and (3) is accepted and used (Lyytinen, 1988).

Typically, organizations identify important features of IS in which to invest to obtain benefits such as increased productivity, improved information flows, increased profitability and market share, improved communication flows, better customer service, and other business values. These benefits contribute to organizational effectiveness. However, according to Silver et al. (1995), these are second order effects; first-order effects derive from appropriate and effective system use. Our IS quality model denotes that IS success is not only determined by the objective quality of the IS artifact but also by stakeholders' subjective assessment of its fitness for use — ease of use and usefulness (Kim, 1989) and "goodness" in use; both generate user satisfaction (Kitchenham, 1989).

## Summary and Conclusion

Even as the IS community magnifies the struggle to overcome the complexity of IS delivery, IS quality remains a key concern for both researchers and practitioners, and the essential difficulties continue to confound and frustrate many. Incremental gains and new IS quality knowledge are often eroded by loftier IS ambitions that contribute to new delivery challenges as organizations increase their reliance on IS to cover a wider spectrum of operations, integrate interdependent business processes, leverage waves of new IT to implement more sophisticated systems, and extend global operations and reach.

In this chapter, we have identified several IS issues and described their bearing on the varying perspectives of IS quality. We have also provided an IS quality model which was used as the basis for analyzing the several approaches that may be considered rallying platforms against the challenges to IS quality. There are mixed indications of the success of many of these approaches, and we have identified both supporting and dissenting findings in the literature. We have been more faithful to breadth than depth in this introductory analysis; however, many of these issues will be elaborated in greater detail in subsequent chapters.

Despite the numerous prescriptions for improving software quality processes and descriptions of product properties that delineate high-quality IS, there is a dearth of recommendations for incorporating quality requirements into design artifacts that can be translated into implementable quality attributes (Barbacci et al., 1995). IS process improvement recommendations focus on establishing process features that contribute to process stability in order to increase the odds of producing high-quality systems; these do not guarantee IS product quality. Similarly, while identifiable product characteristics that demonstrate the quality of a completed system are discernable after the fact, they have to be designed in.

Perhaps one of the approaches (which is not covered in subsequent chapters) that may provide food for further thought is the recommended use of software quality function deployment (SQFD) (Elboushi, 1997; Erikkson & McFadden, 1993; Ho et al., 1999; Martins & Aspinwall, 2001; McDonald, 1995) for capturing and prioritizing user quality requirements, translating them into corresponding technical requirements, and monitoring their accommodation throughout the IS delivery process.

To some extent, this chapter represents the beginnings of a meta-analysis of IS quality research; however, given its more limited scope and intention, such an undertaking could not be entirely accommodated. We intend to complete this meta-analysis, using our quality model as a basis for (1) identifying particular foci of past research and what metrics have been used, and (2) using the model along with the dimensions of IS quality we discussed as the basis for the development of a taxonomy of IS quality issues.

Similarly, we may need to leverage our learning to date about how people, IS process, and software production methods have contributed to IS quality and used to address identified quality-bearing attributes of the delivered product. Our insights have been based on examinations of these phenomena in isolation. Further research and experimentation may be required to understand the synergies that may result from, and the quality impact of, combined treatments and their applicability in various contexts.

## References

Al-Mushayt, O., Doherty, N., & King, M. (2001). An investigation into the relative success of alternative approaches to the treatment of organizational issues in systems development projects. *Organization Development Journal, 19*(1), 31-48.

Ashrafi, N. (2003). The impact of software process improvement on quality in theory and practice. *Information and Management, 40*(7), 677-690.

Attewell, P., & Rule, J. (1984). Computing in organizations: What we know and what we don't know. *Communications of the ACM*, *27*(12), 1184-1192.

Bandi, R. K., Vaishnavi, V. K., & Turk, D. E. (2003). Predicting maintenance performance using object-oriented design complexity metrics. *IEEE Transactions on Software Engineering, 29*(1), 77-87.

Banker, R. D., Davis, G. B., & Slaughter, S. A. (1998). Software development practices, software complexity, and software maintenance performance: A field study. *Management Science, 44*(4), 433-450.

Barbacci, M., Klein, M., Longstaff, T., & Weinstock, C. (1995). *Quality attributes* (Tech. Rep. No. CMU/SEI-95-TR-021). Carnegie Mellon University. Retrieved November 27, 2005, from http://www.sei.cmu.edu/publications/documents/ 95.reports/95.tr.021.html

Barki, H., & Hartwick, J. (1994). Measuring user participation, user involvement, and user attitude. *MIS Quarterly, 18*(1), 59-82.

Barki, H., Rivard, S., & Talbot, J. (1993). A keyword classification scheme for IS research literature: An update. *MIS Quarterly, 17*(2), 209-226.

Barua, A., Kriebel, C., & Mukhopadhyay, T. (1995). Information technologies and business value: An analytic and empirical investigation. *Information Systems Research, 6*(1), 3-50.

Barua, A., & Lee, B. (1997). The information technology productivity paradox revisited: A theoretical and empirical investigation in the manufacturing sector. *International Journal of Flexible Manufacturing Systems, 9*(2), 145-166.

Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice* (2nd ed.). Boston: Addison-Wesley.

Beck, K. (2000) *Extreme programming explained: Embrace change.* Boston: Addison-Wesley.

Boehm, B. W. (1988). A spiral model of software development and enhancement. *IEEE Computer, 21*(5), 61-72.

Boudreau, M.-C., Loch, K., Robey, D., & Straub, D. (1998). Going global: Using information technology to advance the competitiveness of the virtual transnational organization. *The Academy of Management Executive, 12*(4), 120-128.

Bourne, K. C. (1994). Putting the rigor back in RAD. *Database Programming & Design, 7*(8), 25-30.

Brooks, F. P., Jr. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer, 20*(4), 10-19.

Brown, D. W. (2002). *An introduction to object-oriented analysis*. New York: John Wiley & Sons.

Brynjolfsson, E. (1993). The productivity paradox of information technology. *Communications of the ACM, 36*(12), 67-77.

Brynjolfsson, E., & Hitt, L. (1996). Paradox lost? Firm level evidence on the return of information systems spending. *Management Science, 42*(4), 541-558.

Brynjolfsson, E., & Yang, S. (1996). Information technology and productivity: A review of literature. *Advances in Computers, 43*, 170-214.

Butcher, M., Munro, H., & Kratschmer, T. (2002). Improving software testing via ODC: Three case studies. *IBM Systems Journal, 41*(1), 31-34.

Carmel, E. (1997). American hegemony in packaged software trade and the culture of software. *The Information Society, 13*(1), 125-142.

Carmel, E., George, J. F., & Nunamaker, J. F. (1995). Examining the process of electronic-JAD. *Journal of End User Computing, 7*(1) 13-22.

Chen, F., Romano, N. C., Jr., Nunamaker, J. F., Jr. (2003). An overview of a collaborative project management approach and supporting software. *Proceedings of the 9th Americas Conference on Information Systems* (pp. 1303-1313).

CIO. (2001, June 1). Does business software stink? *CIO*. Retrieved February 2, 2006, from http://www2.cio.com/research/surveyreport.cfm?id=23

Conradi, H., & Fuggetta, A.(2002). Improving software process improvement. *IEEE Software, 19*(4), 92-99.

Davidson, E. J. (1999). Joint application design (JAD) in practice. *The Journal of Systems and Software, 45*(3), 215-223.

Davis, A. (1992). Operational prototyping: A new development approach. *IEEE Software, 9*(5), 70-78.

DeLone, W. H., & McLean, E. R. (1992). Information systems success: The quest for the dependent variable. *Information Systems Research, 3*(1), 60-95.

Doherty, N. F., & King, M. (1998). The consideration of organizational issues during the systems development process: An empirical analysis. *Behavior & Information Technology, 17*(1), 41-51.

Dromey, R. (1996). Cornering the Chimera. *IEEE Software, 13*(1), 33-43.

Duggan, E. W. (2004a). Silver pellets for improving software quality. *Information Resources Management Journal, 17*(2), 1-21.

Duggan, E. W. (2004b). Reducing IS maintenance by improving the quality of IS development processes and practices. In K. M. Khan & Y. Zhang (Eds.), *Managing corporate information systems evolution and maintenanc*e (pp. 345-370). Hershey, PA: Idea Group Publishing.

Duggan, E. W. (2005, May 15-18). Revisiting IS project management strategies: The new challenges. *Proceedings of the International Conference of the Information Resources Management Association,* San Diego, CA (pp. 1-4).

Duggan, E. W., & Duggan, D. K. (2005). Employee rights and participation in the design of information systems in the European Union and the US: Codetermination laws and voluntary participation. *Journal of Individual Employee Rights, 11*(4).

Duggan**,** E. W., & Thachenkary, C. S. (2004). Integrating nominal group technique and joint application development for improved systems requirements determination. *Information and Management, 41*(4), 399-411.

Elboushi, M. I. (1997). Object-oriented software design utilizing quality function deployment. *The Journal of Systems and Software, 38*(2), 133-143.

Erikkson, I., & McFadden, F. (1993). Quality function deployment: A tool to improve software quality. *Information & Software Technology, 35*(9), 491-498.

Fayad, M., & Laitinen, M. (1997). Process assessment considered wasteful. *Communications of the ACM, 40*(1), 125-128.

Fitzgerald, B. (2000). Systems development methodologies: The problem of tenses. *Information Technology & People, 13*(3), 174-182.

Flatten, P. O. (1992). Requirements for a life-cycle methodology for the 1990s. In W.W. Cotterman & J.A. Senn (Eds.), *Challenges and strategies for research in systems development* (pp. 221-234). New York: John Wiley & Sons.

Floyd, C. (1984). A systematic look at prototyping. In Budde et al. (Eds.), *Approaches to prototyping* (pp. 105-122). Berlin: Springer-Verlag.

Floyd, C., Reisin, F.-M., & Schmidt, G. (1989). STEPS to software development with users. *ESEC 1989* (pp. 48-64). Berlin: Springer Verlag.

Gable, G. G., Sedera, D., & Chan, T. (2003). Enterprise systems success: A measurement model. *Proceedings of the Twenty-Fourth International Conference on Information Systems* (pp. 576-591).

Garvin, D. A. (1984). What does "product quality" really mean? *Sloan Management Review, 26*(1), 25-45.

Gibbs, W. W. (1994). Software's chronic crisis. *Scientific American, 271*(3), 86-95.

Gill, C., & Krieger, H. (1999). Direct and representative participation in Europe: Recent survey evidence. *International Journal of Human Resource Management, 10*, 72-591.

Gladden, G. R. (1982). Stop the life-cycle, I want to get off. *ACM SZGSOFT Software Engineering Notes, 7*(2), 35-39.

Glass, R. L. (2001). Extreme programming: The good, the bad, and the bottom line. *IEEE Software, 8*(6), 111-112.

Goldenson, D., & Gibson, D. (2003). *Demonstrating the impact and benefits of CMMI: An update and preliminary results* (Tech. Rep. No. CMU/SEI-2003-SR-009). Carnegie Mellon Software Engineering Institute.

Grady, R. B. (1993). Practical results from measuring software quality. *Communications of the ACM, 36*(11), 62-68.

Gray, E., & Smith, W. (1998). On the limitations of software process assessment and the recognition of a required re-orientation for global process improvement. *Software Quality Journal, 7*(1), 21-34.

Hanna, M. (1995). Farewell to waterfalls? *Software Magazine, 15*(5), 38-46.

Harter, D. E., Slaughter, S. A., & Krishnan, M. S. (1998). The life cycle effects of software quality: A longitudinal analysis. *Proceedings of the 19th International Conference on Information Systems* (pp. 346-351).

Hausler, P. A., Linger, R. C., & Trammell, C. J. (1994). Adopting cleanroom software engineering with a phased approach. *IBM Systems Journal, 33*(1), 89-109.

Head, G. E. (1994). Six-Sigma software using cleanroom software engineering techniques. *Hewlett-Packard Journal, 45*(3), 40-50.

Hirschheim, R., & Klein, H. K. (1994). Realizing emancipatory principles in information systems development: The case for ETHICS. *MIS Quarterly, 18*(1), 83-105.

Hirschheim, R., & Newman, M. (1988). Information systems and user resistance: Theory and practice. *Computer Journal, 31*(5), 398-408.

Ho, E. S., Lai, Y. J., & Chang, S. I. (1999). An integrated group decision-making approach to quality function deployment. *IIE Transactions, 31*(6), 553-567.

Hoffer, J., George, J., & Valacich, J. (2002). *Modern systems analysis and design.* NJ: Prentice Hall.

Hollenbach, C., Young, R., Pflugrad, A., & Smith, D. (1997). Combining quality and software process improvement. *Communications of the ACM, 40*(6), 41-45.

Hough, D. (1993). Rapid delivery: An evolutionary approach for application development. *IBM Systems Journal, 32*(3), 397-419.

Humphrey, W. (1988). Characterizing the software process: A maturity framework. *IEEE Software, 5*(2), 73-79.

Humphrey, W., Snyder, T., & Willis, R. (1991). Software process improvement at Hughes Aircraft. *IEEE Software, 8*(4), 11-23.

ISO/IEC Standard 9126-1. (2001). *Software Engineering — Product Quality — Part 1*: *Quality Model.*

Jerva, M. (2001). Systems analysis and design methodologies: Practicalities and use in today's information systems development efforts. *Topics in Health Information Management, 21*(4), 13-20.

Johnson, R. A., Hardgrove, B. C., & Doke, E. R. (1999). An industry analysis of developer beliefs about object-oriented systems development. *The DATA BASE for Advances in Information Systems, 30*(1), 47-64.

Kautz, K. (1993). *Evolutionary system development. Supporting the process* (Research Report 178). Dr. Philos Thesis, University of Oslo, Department of Informatics..

Kettelhut, M. C. (1993). JAD methodology and group dynamics. *Information Systems Management, 14*(3), 46-53.

Khalifa, M., & Verner, J. M. (2000). Drivers for software development method usage. *IEEE Transactions on Engineering Management, 47*(3), 360-369.

Kim, K. K. (1989). User satisfaction: A synthesis of three different perspectives. *Journal of Information Systems, 4*(1), 1-12.

Kirsch, L. J. (2000). Software project management: An integrated perspective for an emerging paradigm. In R.W. Zmud (Ed.), *Framing the domains of IT management research: Projecting the future through the past* (pp. 285-304). Cincinnati, OH: Pinnaflex Educational Resources Inc.

Kitchenham. (1989). Software quality assurance. *Microprocessors and Microcomputers, 13*(6), 373-381.

Kjaer, A., & Madsen, K. H. (1995). Participatory analysis of flexibility. *Communications of the ACM, 38*(5), 53-60.

Kuvaja, P., Similä, J., Kranik, L., Bicego, A., Saukkonen, S., & Koch, G. (1994). *Software process assessment and improvement — the BOOTSTRAP approach.* Cambridge, MA: Blackwell Publishers.

Lee, G., & Xia, W. (2002). Flexibility of information systems development projects: A conceptual framework. *Proceedings of the Eighth Americas Conference on Information Systems* (pp. 1390-1396).

Lehr, B., & Lichtenberg, F. (1999). Information technology and its impact on productivity: Firm level evidence from government and private data sources 1977-1993. *Canadian Journal of Economics, 32*(2), 335-362.

Leung, H. (1999). Slow change of information system development practice. *Software Quality Journal, 8*(3), 197-210.

Lyytinen, K. (1988). Expectation failure concept and systems analysts' view of information system failures: Results of an exploratory study. *Information & Management, 14*(1), 45-56.

Mankin, D., & Cohen, S. G. (1997). Teams and technology: Tensions in participatory design. *Organizational Dynamics, 26*(1), 63-76.

Marakas, G. M., & Hornik, S. (1996). Passive resistance misuse: Overt support and covert recalcitrance in IS implementation. *European Journal of Information Systems, 5*, 208-219.

Markus, M. L., & Keil, M. (1994). If we build it they will come: Designing information systems that users want to use. *Sloan Management Review, 35*(4), 11-25.

Martins, A., & Aspinwall, E. M. (2001). Quality function deployment: An empirical study in the UK. *Total Quality Management, 12*(5), 575-588.

McDonald, M. P. (1995, June 11-13). Quality function deployment - introducing product development into the systems development process. *Proceedings of the Seventh Symposium on Quality Function Deployment*, Michigan.

Melo, W., El Emam, K., & Drouin, J. (1998). *SPICE: The theory and practice of software process improvement and capability determination.* Los Alamitos, CA: IEEE Computer Society Press.

Meyer, B. (1988) *Object oriented software construction.* Englewood Cliffs, NJ: Prentice Hall.

Miles, R. (1985). Computer systems analysis: The constraint of the hard systems paradigm. *Journal of Applied Systems Analysis, 12,* 55-65.

Mousinho, G. (1990). Project management: Runaway! *Systems International, 6*, 35-40.

Mukherjee, K. (2001). Productivity growth in large US commercial banks: The initial post regulation experience. *Journal of Banking and Finance, 25*(5), 913.

Muller, M. J., Wildman, D. M., & White, E. A. (1993). Taxonomy of PD practices: A brief practitioner's guide. *Communications of the ACM, 36*(4), 26-27.

Mumford, E. (1983). *Designing human systems: The ETHICS method.* Manchester, UK: Manchester Business School Press.

Mumford, E. (1993). The ETHICS approach. *Communications of the ACM, 36*(6), 82.

Newman, M., & Robey, D. (1992). A social process model of user-analyst relationships. *MIS Quarterly, 16*(2), 249-266.

Niederman, F., Brancheau, J. C., & Wetherbe, J. C. (1991). Information systems management issues for the 1990s. *MIS Quarterly, 15*(4), 474-500.

Palvia, S. C., Sharma, R. S., & Conrath, D. W. (2001). A socio-technical framework for quality assessment of computer information systems. *Industrial Management & Data Systems, 101*(5), 237-251.

Paulk, M. C. (2001). Extreme programming from a CMM perspective. *IEEE Software, 8*(6), 19-26.

Perry, D. E., Staudenmayer, N. A., & Votta, L. G. (1994). People, organizations, and process improvement. *IEEE Software, 11*, 36-45.

Pfleeger, S. L. (1996). Realities and rewards of software process improvement. *IEEE Software, 13*(6), 99-101.

Quinn, R. E., & Cameron, K. S. Organizational paradox and transformation. In R. Quinn & K. Cameron (Eds.), *Paradox of transformation: Toward a theory of change in organization and management* (pp. 1-18). Cambridge, MA: Ballinger Publishing.

Rae, A., Robert, P., & Hausen, H. L. (1995). *Software evaluation for certification: Principles, practice and legal liability.* Reading, UK: McGraw-Hill.

Ravichandran, T., & Rai, A. (1994, March 24-26). The dimensions and correlates of systems development quality. *Proceedings of the Annual SIG Computer Personnel Research Conference on Reinventing IS*, Virginia, (pp. 272-282).

Ravichandran, T., & Rai, A. (1996). Impact of process management on systems development quality: An empirical study. *Proceedings of the 2nd Americas Conference on Information Systems* (pp. 143-145).

Ravichandran, T., & Rai, A. (2000). Quality management in systems development: An organizational system perspective. *MIS Quarterly, 24*(3), 381-415.

Riemenschneider, C. K., Hardgrave, B. C., & Davis, F. D. (2002). Explaining software developer acceptance of methodologies: A comparison of five theoretical models. *IEEE Transactions on Software Engineering, 28*(12), 1135-1145.

Roberts, T. L., Gibson, M. L., & Fields, K. T. (1999). Systems development methodology implementation: Perceived aspects of importance. *Information Resources Management Journal, 10*(3), 27-38.

Roberts, T. L., Gibson, M. L., Fields, K. T., & Rainer, R. K. (1998). Factors that impact implementing a system development methodology. *IEEE Transactions on Software Engineering, 24*(4), 640-649.

Robey, D., & Markus, M. L. (1984). Rituals in information system design. *MIS Quarterly, 8*(1), 5-15.

Robey, D., Welke, R., & Turk, D. (2001). Traditional, iterative, and component-based development: A social analysis of software of software development paradigms. *Information Technology and Management, 2*(1), 53-70.

Sarker, S., & Sahay, S. (2003). Understanding the virtual team development: An interpretive study. *Journal of the AIS, 4*(1), 1-38.

Schaider, E. D. (1999). *TCO in the trenches: The Standish Group study*. Retrieved January 27, 2006, from http://www.softwaremag.com/1.cfm?doc=archive/1999dec/TCO.html

Schmidt, D. C., & Fayad, M. E. (1997). Lessons learned building reusable OO frameworks for distributed software. *Communications of the ACM, 40*(10), 85-87.

Seddon, P. B. (1997). A respecification and extension of the DeLone and McLean model of IS success. *Information Systems Research, 8*, 240-253.

SEI. (2002). *Process maturity profile of the software*. Pittsburgh, PA: SCAMPI Appraisal Results, Software Engineering Institute, Carnegie Mellon University. Retrieved November 27, 2005, from http://www.sei.cmu.edu/sema/pdf/SWCMM/2002aug.pdf

Silver, M. S., Markus, M. L., &. Beath, C. M. (1995). The information technology interaction model: A foundation for the MBA core course. *MIS Quarterly, 19*(3), 361-390.

Spangler, A. (1996, October/November). Cleanroom software engineering: Plan your work and work your plan in small increments. *IEEE Potentials*, 29-32.

Standish Group, The. (2002). What are your requirements? West Yarmouth, MA: The Standish Group International (based on 2002 CHAOS Report).

Subramanayam, V., Deb, S., Krishnaswamy, P., & Ghosh, R. (2004). *An integrated approach to software process improvement at Wipro Technologie: Veloci-Q* (Tech. Rep. No. CMU-SEI-2004-TR-006). Carnegie Mellon Software Engineering Institute.

Swanson, E. B. (1997). Maintaining IS quality. *Information and Software Technology, 39*, 845-850.

Szyperski, C. (1998). *Component software, beyond object-oriented programming*. New York: Addison-Wesley.

Trammell, C. J., Pleszkoch, M. G., Linger, R. C., & Hevner A. R. (1996). The incremental development process in cleanroom software engineering. *Decision Support Systems, 17*(1), 55-71.

Vessey, I., & Conger, S. A. (1994). Requirements specification: Learning object, process, and data methodologies. *Communications of the ACM, 41*(4), 99-102.

Weinberg, R. S. (1991). Prototyping and the systems development life cycle. *Information Systems Management, 8*(2), 47-53.

Wilson, D. (1993). *Assessing the impact of information technology on organizational performance: Strategic information technology management*. Harrisburg, PA: Idea Group.

Wood, J., & Silver, D. (1995). *Joint application development.* New York: John Wiley & Sons.

Yamamura, G. (1999). Software process satisfied employees. *IEEE Software, 16*(5), 83-85.

Zahran, S. (1998). *Software process improvement: Practical guidelines for business success.* Essex, UK: Addison-Wesley Longman.

## Chapter II

# An Overview of Software Quality Concepts and Management Issues

Alain April, École de technologie supérieure, Québec, Canada

Claude Y. Laporte, École de technologie supérieure, Québec, Canada

## Abstract

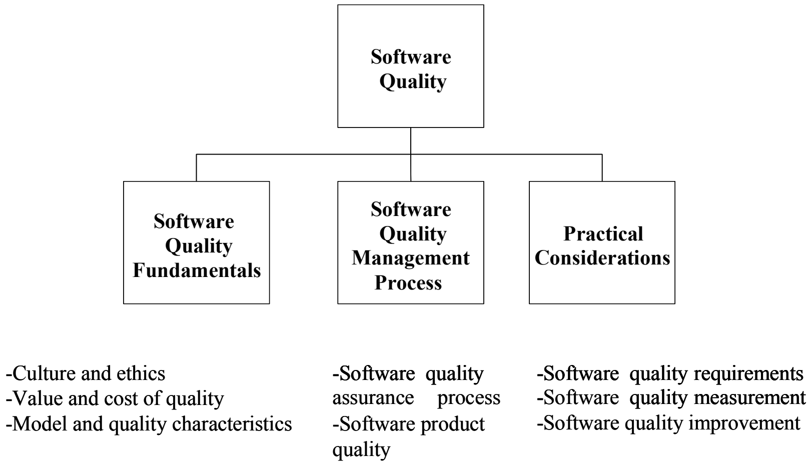
*This chapter introduces the generally accepted knowledge on software quality that has been included in the (SWEBOK) Software Engineering Body of Knowledge (ISOTR 19759, 2005). One chapter of the SWEBOK is dedicated to software quality (April et al., 2005). It argues that ethics play an important role in applying the quality models and the notions of cost of quality for software engineers. It also describes the minimal content required in a software quality assurance plan. Finally an overview of what to expect in the upcoming international standards on software quality requirements, which transcend the life cycle activities of all IT processes, is presented.*

# Introduction

The business value of a software product results from its quality as perceived by both acquirers and end users. Therefore, quality is increasingly seen as a critical attribute of software, since its absence results in financial loss as well as dissatisfied users, and may even endanger lives. For example, Therac-25, a computer-driven radiation system, seriously injured and killed patients by massive overdosing (Levenson & Turner, 1993). Improving recognition of the importance of setting software quality requirements and of assessing quality causes a shift in the "center of gravity" of software engineering from creating technology-centered solutions to satisfying stakeholders. Software acquisition, development, maintenance, and operations organizations confronted with such a shift are, in general, not adequately equipped to deal with it. Until recently, they did not have at their disposal the quality models or measurement instruments to allow (or facilitate) the engineering of quality throughout the entire software product life cycle. The objective of software product quality engineering is to achieve the required quality of the product through the definition of quality requirements and their implementation, measurement of appropriate quality attributes, and evaluation of the resulting quality. The objective is, in fact, software product quality.

This chapter is structured in accordance with the SWEBOK classification of the software quality body of knowledge (www.swebok.org) shown in Figure 1.

*Figure 1. Adapted breakdown of software quality topics (ISOTR 19759, 2005)*


Software Quality
Software Quality Fundamentals
Software Quality Management Process
Practical Considerations
-Culture and ethics
-Value and cost of quality
-Model and quality characteristics
-Software quality assurance process
-Software product quality
-Software quality requirements
-Software quality measurement
-Software quality improvement

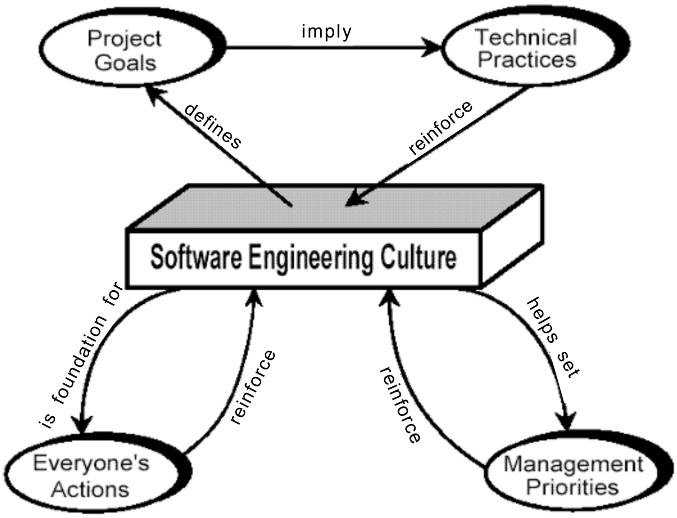
# Software Quality Fundamentals

Agreement on quality requirements, as well as clear communication on what constitutes quality, require that the many aspects of quality be formally defined and discussed. Over the years, authors and organizations have defined the term "quality" differently. IBM used the phrase "market-driven quality", which is based on achieving total customer satisfaction. This definition was influenced by the total quality management approach of Phil Crosby (1979), who defined quality as "conformance to user requirements". Watts Humphrey (Humphrey, 1990), looking at quality in the software industry, defined it as "achieving excellent levels of fitness for use". More recently, quality has been defined in ISO 9001 (2000) as "the degree to which a set of inherent *characteristics* fulfills the *requirements.*" The next section looks at how organizational culture and individual ethics play a role in quality in the organization.

## Culture and Ethics

Edward B. Tylor (1871) defined human culture as "that complex whole which includes knowledge, belief, art, morals, law, custom, and any other capabilities and habits acquired by man as a member of society." Culture guides the behaviors, activities, priorities, and decisions of an individual, as well as those of an organization. Karl Wiegers (1996), in his book *Creating a Software Engineering Culture*, illustrates (see Figure 2) the interaction between the

*Figure 2. A software engineering culture (Wiegers, 1996)*



software engineering culture of an organization, its software engineers and its projects. Why should we, as technical people, care about such a "soft" issue? First, a quality culture cannot be bought. It needs to be developed, mostly at the beginning, by the founders of the organization. Then, as employees are selected and hired, the initial leader's culture will start to slowly adjust to the pressures of the environment, as shown in Figure 2. Quality culture cannot be "bolted on" to an organization; it has to be designed-in and nurtured. The ultimate aim of upper management is to instill a culture that will allow the development of high-quality products and offer them at competitive prices, in order to generate revenues and dividends in an organization where employees are committed and satisfied.

The second reason why we should be interested in the cultural aspects of quality is that any change an organization wants to make, for example, moving up on the Capability Maturity Model integration$^{SM}$ (CMMi$^{SM}$) (SEI, 2002) maturity scale, cannot simply be ordered; the organization has to cope with the current culture when making a change in maturity, especially when such a change implies a profound change in that culture. An organization cannot just "buy and deploy" off-the-shelf processes that contain quality. It has been demonstrated that one of the major inhibitors of change is the culture of the organization. The management of change is imperative (Laporte & Trudel, 1998) in order to achieve the desired result.

The issue of quality is also central to the Code of Ethics, developed by and for software engineers, which was released in 1999 (Gotterbarn, 1999; Gotterbarn et al., 1999; IEEE-CS, 1999). The Code describes eight top-level technical and professional obligations against which peers, the public, and legal bodies can measure a software engineer's ethical behavior. Each top-level obligation, called a principle, is described in one sentence and supported by a number of clauses which gives examples and details to help in interpretation and implementation of that obligation. Software engineers adopting the Code commit to eight principles of quality and morality. The following are examples: Principle 3 (Product) states that software engineers shall ensure that their products and related modifications meet the highest professional standards possible. This principle is supported by 15 clauses, and clause 3.10 reads that software engineers shall ensure adequate testing, debugging, and review of software and related documents on which they work. The Code has been translated into eight languages: Chinese, Croatian, English, French, Hebrew, Italian, Japanese, and Spanish, and all versions are available at http://seeri.etsu.edu/Codes/default.shtm. It has been publicly adopted by many organizations, as well as by a few universities as part of their curriculum in software engineering.

## Value and Costs of Quality

To promote a quality approach to software, a number of arguments must be developed to support its value and benefits. Quality, as a whole, is not always perceived positively and is a hard sell for software project managers. One famous book (Crosby, 1979) addressing the cost of quality has been published on this topic. Since its publication, many organizations, mainly manufacturers, have successfully promoted the use of the cost of quality concepts and framework in their project management processes. A few papers have been published on the adoption of these concepts in the software industry (Campanella, 1990; Diaz, 2002; Dobbins, 1999; Galin, 2004a; Mandeville, 1990; Slaughter et al., 1998), but very few case studies have been published by the software industry itself (Dion, 1993; Haley, 1996; Houston, 1999; Knox, 1993). A paper by Haley of Raytheon (Haley, 1996) illustrates very well the links between the cost of rework and the investment needed to reduce waste. Haley also illustrates that a long-term perspective is needed, as well as a commitment from senior management, in order to capture the benefits. Another fact illustrated by Haley is the link between the investment/benefits and the Capability Maturity Model for Software (CMM)® (Paulk et al., 1993) maturity level of an organization (see Figure 3). At the initial CMM maturity level, most, if not all, organizations have no reliable data or measurement system. When an organization has reached levels

*Figure 3. Improvement data (Dion, 1993; Haley, 1996)*

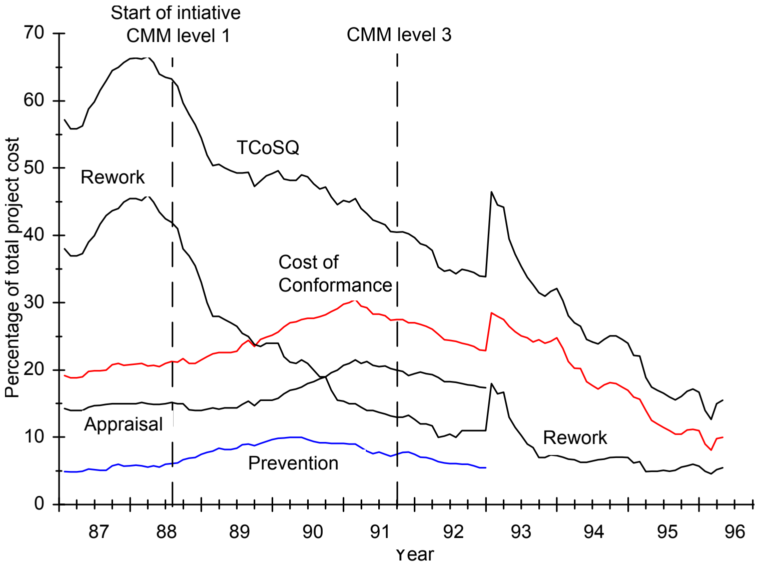


*Table 1. Illustration of the cost and benefits of quality practices (Galin, 2004a)*

| **Quality assurance activity** | **Standard plan** | | **Comprehensive plan** | |
|---|---|---|---|---|
| | **Percentage of defects removed** | **Cost of removing defects (cost units)** | **Percentage of defects removed** | **Cost of removing defects (cost units)** |
| 1. Requirements specification review | 7.5% | 7.5 | 9% | 9 |
| 2. Design inspection | -- | -- | 28.7% | 71.8 |
| 3. Design review | 21.3% | 53.2 | 7.4% | 18.5 |
| 4. Code inspection | -- | -- | 24.4% | 158.6 |
| 5. Unit test – code | 25.6 | 166.4 | 4.2% | 27.3 |
| 6. Integration test | 17.8% | 284.8 | 9.8% | 156.8 |
| 7. Documentation review | 13.9% | 222.4 | 9.9% | 158.4 |
| 8. System test | 7.0% | 280 | 4% | 160. |
| **Total for internal QA activities** | **93.1%** | **1014.3** | **97.4** | **760.4** |
| Defects detected during operation | 6.9% | 759 | 2.6% | 286 |
| **Total** | **100.0%** | **1773.3** | **100%** | **1046.4** |

2 and 3, it has the foundation to measure and select beneficial process improvement areas. Many organizations have published the results obtained in climbing the maturity ladder. They show the relationships between maturity level, quality, productivity, and project cost.

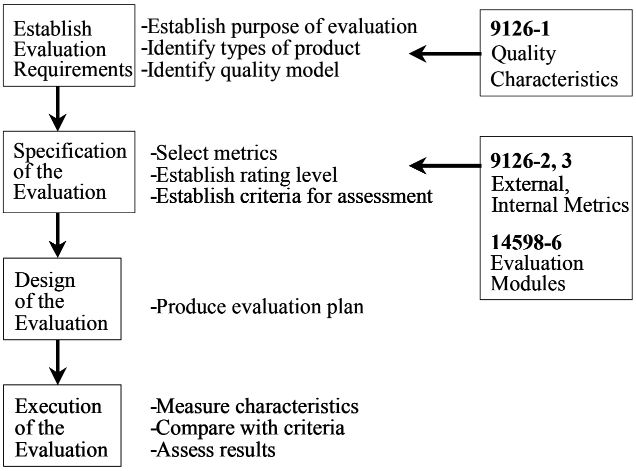
Galin (2004b), in a recent software quality assurance book, illustrates the cost and benefits of quality practices. As shown in Table 1, the addition of effective quality practices, such as inspection and review, even reduce the cost of a project by eliminating rework. The left-hand column lists typical quality assurance activities, such as reviews and inspections, of an on-going project. The middle column shows the effectiveness of typical defect removal activities included in a software quality plan. It shows that such a process would deliver a product where 93.1% of the total defects have been detected, leaving 6.9% of them, for a QA cost of 1014.3 units. On the right-hand side, two activities have been added to the quality plan: a design inspection and a code inspection. Even though these activities are not free (18.5 and 158.6 units, respectively), the net result is that 97.4% of the total defects have been detected, leaving only 2.6% of the defects in operation. Note also that the total QA cost is 760.4 units, compared to 1014.3 units previously.

The following quote from Norman Augustine (1997) summarizes this section: "It costs a lot to build bad products."

## Model and Quality Characteristics

Quality models for software have been developed to assess both software processes and software products. Software process engineering has made many

*Figure 4. ISO/IEC 14598 – Evaluation process (Sur03)*



advances in the last decade and has introduced numerous quality models and standards concerned with the definition, implementation, measurement, management, change, and improvement of the software engineering process itself. Assessing the quality of software products requires that the quality characteristics of software be defined and measured. The quality of a software product is determined by its properties. Some of these are inherent to the software product, while others are assigned, by priority, to the software product during its requirements definition process. Process quality is concerned with the technical and managerial activities within the software engineering process that are performed during software acquisition, development, maintenance, and operation.

In contrast, software product quality models describe the many quality attributes of software. These have been presented by many authors (Boehm et al., 1978; McCall et al., 1977) and have more history. Although this is an older topic, it has not quite reached the popularity of software process quality models. During the early years, the terminology for software product quality differed from one model of software quality to another. Each model had a different number of hierarchical levels and number of characteristics, and the different naming conventions were especially confusing. Based on these efforts, the International Organization for Standardization (ISO) normalized three software product quality models (i.e., internal quality, external quality, and quality in use) (ISO 9126, 2001) and accompanied them with a set of guides like ISO/IEC14598 which explains how to evaluate a software product (see Figure 4).

## Software Quality Management Process

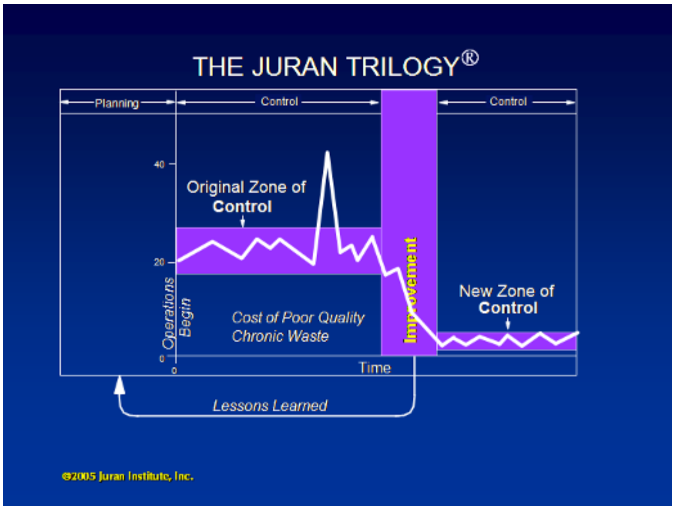
Software quality management (SQM) defines processes, process owners, requirements for those processes, measurements of the process, their outputs, and, finally, feedback channels (Arthur, 1992). SQM processes are useful for evaluating the process outcome as well as the final product.

The Juran Trilogy Diagram, illustrated in Figure 5, depicts quality management as comprising three basic processes: quality planning, quality control, and quality improvement. The diagram shows how waste caused by mismanaging quality, or not managing it, can be reduced by introducing quality improvement and feeding back the lessons learned from this into the quality planning process. Many, if not most, software engineering managers are not aware of the magnitude of the waste, i.e., between 30 and 40%, in their processes.

Software quality management processes consist of many activities. Some may find defects directly, while others indicate where further examination may be valuable. Planning for software quality involves:

1. defining the required product in terms of its quality characteristics;
2. planning the processes to achieve the required product quality.

*Figure 5. Juran trilogy diagram*



Some of the specific SQM processes are defined in standard IEEE 12207 (1996):

- Quality assurance process
- Verification process
- Validation process
- Review process
- Audit process

These processes encourage quality and also find possible problems, but they differ somewhat in their emphasis. SQM processes help ensure better software quality in a given project. They also, as a by-product, provide management with general information, including an indication of the quality of the entire software engineering process.

SQM processes consist of tasks and techniques to indicate how software plans (e.g., management, development, and configuration management) are being implemented and how well the intermediate and final products are meeting their specified requirements. Results from these tasks are assembled into reports for management before corrective action is taken. The management of an SQM process is tasked with ensuring that the results of these reports are accurate.

As described in this Knowledge Area (KA), SQM processes are closely related; they can overlap and are sometimes even combined. They seem largely reactive in nature because they address the processes as practiced and the products as produced, but they have a major role at the planning stage in being proactive in terms of the processes and procedures needed to attain the quality characteristics and degree of quality needed by the stakeholders in the software.

Risk management can also play an important role in delivering quality software. Incorporating disciplined risk analysis and management techniques into the software life cycle processes can increase the potential for producing a quality product (Charette, 1989). Refer to the Software Engineering Management KA of the SWEBOK for material related to risk management.

## Software Quality Assurance Process

Software quality assurance processes have been found to have a direct effect on the quality of a software product. During a software development project, it is difficult to completely distinguish the quality of the process used to develop the software from the quality of the product itself. Process quality assurance influences process quality, which in turn influences the quality characteristics of the software product.

Process quality is addressed by two important process models, ISO9001 and CMMi$^{SM}$ (SEI, 2002). First, ISO9001:2000 (ISO 9001, 2000) is an international standard, while CMMi is a process model. Compliance with the key quality concepts of the ISO9001:2000 standard requires an interpretation for each industry. Software, a service industry, has initially had more difficulty interpreting and implementing the ISO9001 standards (Bouman et al., 1999; May, 2002; Niessink, 2000). Service organizations also experience greater difficulty dissociating the final product from the processes used in its production (May, 2002). To help with this problem, additional guidance has been developed by the ISO for the implementation of ISO9001:2000 in the software industry (Hailey, 1998; May, 2002). This guide, numbered ISO90003 (2004), helps link all the software-related standards to the quality system required by ISO9001:2000.

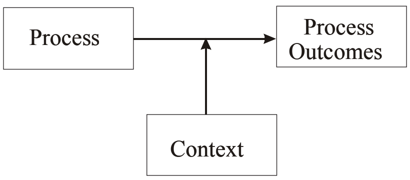
In contrast, CMMi (SEI, 2002) is a process model which proposes proven practices at different maturity levels. It is divided into process areas, and the following are key to quality management: (a) process and product quality assurance; (b) process verification; and (c) process validation.

There was initially some debate over whether ISO9001:2000 or CMMi should be used by software engineers to ensure quality. It has been observed that many organizations use their ISO9001 certification assets as a stepping stone toward meeting the CMMi requirements. It is believed that level 3 of CMMi is reachable for an organization that already has ISO9001:2000 certification. In competitive situations, it is not unusual to find that the two are used concurrently to meet shared quality objectives.

In general, process quality models are concerned with process outcomes. However, in order to achieve the process outcomes desired (e.g., better quality, better maintainability, greater customer satisfaction), we have to implement and measure the process in action.

Current quality models are not complete at this time, as there are many other factors to consider which have an impact on outcomes. Other factors, such as

*Figure 6. Relationships between process and outcomes (ISOTR19759, 2005)*



the context, industry domain, capability of the staff, and the tools used also play an important role (see Figure 6). "Furthermore, the extent to which the process is institutionalized or implemented (i.e., process fidelity) is important, as it may explain why 'good' processes do not give the desired outcomes" (ISOTR19759, 2005).

A word of caution is needed with regard to the term *software quality assurance* (SQA). It is actually a misnomer, because SQA cannot *assure* quality, but only *provide adequate assurance*. ISO 12207 (ISO/IEC12207.0-96) defines an SQA process as a process for providing adequate assurance that the software products and processes in the project life cycle conform to their specified requirements and adhere to their established plans.

The persons or groups providing SQA services need to be independent of the developers in order to provide an unbiased opinion on the conformance of products and processes to the standards imposed by a customer, an organization, or a country. SQA services can be provided by an entity that may be internal or external to the developing organization.

An SQA plan, often mandated by an SQA standard, derives its requirements from the contract, the project plan, and the organization's quality system. It defines the means that will be used to ensure that software developed for a specific product satisfies the user's requirements and is of the highest quality possible within project constraints. The content and format of an SQA plan are available in government documents, such as ESA and NASA, or publicly, in standards such as IEEE standard 730 (IEEE730, 2002).

ISO/IEC 12207 (1996) proposes a minimal content for the SQA plan:

- Quality standards, methodologies, procedures, and tools for performing quality assurance activities (or their references in the organization's official documentation);
- Procedures for contract review and their coordination;

*Table 2. Organizational quality practices (Cusumano et al., 2003)*

| Practices Used | Japan | US |
|---|---|---|
| Architectural specifications | 70.4% | 54.8% |
| Functional specifications | 92.6% | 74.2% |
| Detailed designs | 85.2% | 32.3% |
| Design reviews | 100% | 77.4% |
| Code reviews | 74.1% | 71% |
| Regression testing | 96.3% | 71% |

*Table 3. Performance comparison (Cusumano et al., 2003)*

| Performance Crieria | Japan | US |
|---|---|---|
| Median output (see note 1) | 469 | 270 |
| Median defect rate (see note 2) | .020 | .400 |
| *Note 1: Output per programmer-month of effort in new Lines of Code (LOC).*<br>*Note 2: Number of defects reported per 1000 LOC (in the 12 months after delivery to customers).* | | |

*Table 4. Productivity and quality improvements over a decade (Cusumano et al., 2003)*

| | Japan – 1990 | Japan - 2003 | US – 1990 | US -2003 |
|---|---|---|---|---|
| Productivity | 389 | 469 | 245 | 270 |
| Quality | .20 | .020 | .83 | .400 |

- Procedures for identification, collection, filing, maintenance, and disposition of quality records; resources, schedule, and responsibilities for conducting the quality assurance activities;
- Selected activities and tasks from processes, such as verification, validation, joint review, audit, and problem resolution.

To illustrate the importance of SQA practices, Table 2 (Cusumano et al., 2003) lists a few practices included in a sample of Japanese and American organizational processes. The data show the percentage of sampled projects using a particular practice.

Table 3 from the same paper shows the performance data resulting from implementation of the practices. As can be observed, the increased performance of the practices in Japan resulted in increased productivity and quality performance in that country.

Table 4 illustrates the important improvements, especially in quality, measured in Japan and the US over the 10-year period.

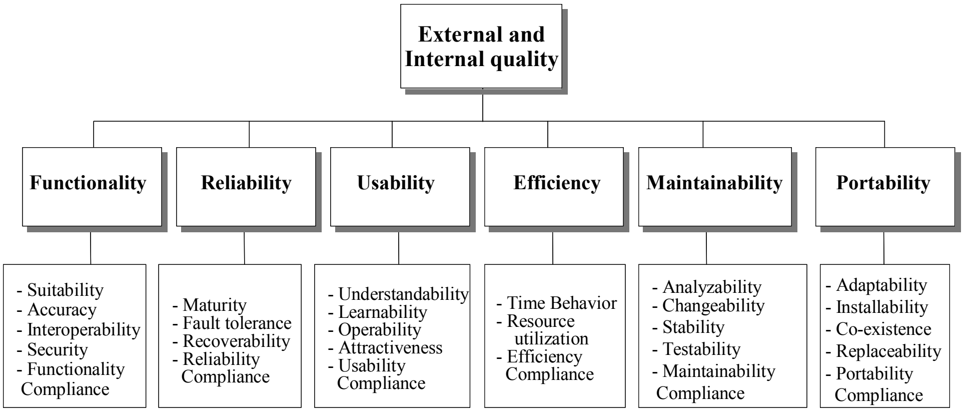
## Software Product Quality

The quality of software is not only influenced by the quality of the process used to develop it, but by the quality of the product as well. Software product quality has also been the subject of much research and of many international standards and publications. Software engineers determine, with the stakeholders, the real purpose of future software. In this regard, it is of prime importance to keep in mind that the customer's requirements come first and that they should include non-functional requirements like quality, not just functionality. Thus, the software engineer has a responsibility to elicit quality requirements from future software which may not be made explicit at the outset. It is key that the software engineer assess their relative importance, as well as the level of difficulty in achieving each of them. The section on value and cost of quality presented the additional processes associated with defining and assessing software quality that need to be added to software projects, and each carries additional costs.

Software product quality standards define the related quality characteristics and subcharacteristics, as well as measures which are useful for assessing software product quality (Suryn et al., 2003). In the standards, the meaning of the term *product* is extended to include any artifact which is a software process outcome used to build the final software product. Examples of a product include, but are not limited to, an entire system requirements specification, a software requirements specification for a software component of a system, a design module, code, test documentation or reports produced as a result of quality analysis tasks.

The software product quality model provided in ISO/IEC 9126-1 (ISO9126, 2001) defines six quality characteristics: functionality, reliability, usability, maintainability, portability and efficiency, as well as quality in use, which is defined as effectiveness, productivity, safety, and satisfaction (see Figure 7). The six quality characteristics have defined subcharacteristics, and the standard also allows for user-defined components. The intention is that the defined quality

*Figure 7. External/internal quality of ISO/IEC 9126 (ISO9126-2001)*



characteristics cover all quality aspects of interest for most software products and, as such, can be used as a checklist for ensuring complete coverage of quality early in the specifications phase.

The quality model suggests three views: internal quality, external quality, and quality in use.

> *Internal quality provides a 'white box' view of the software and addresses properties of the software product that typically are available during development. External quality provides a 'black box' view of the software and addresses properties related to its execution. "The quality-in-use view is related to application of the software in its operational environment, for carrying out specified tasks by specified users. Internal quality has an impact on external quality, which again has an impact on quality in use.* (Suryn et al., 2003)

External and internal quality models presented by ISO/IEC9126 are less trivial and have to be explained further. Let us take the maintainability of software as a quality attribute, and see both the external and internal perspectives in action.

From an external point of view, maintainability attempts to measure the effort required to diagnose, analyze, and apply a change to a software product. Thus, an external measure of maintainability would constitute the effort to make a specific change.

From an internal product point of view, the quality model proposes measurement of the attributes of the software that influence the effort required to modify it. A good example is source code complexity. Structural complexity measures are generally extracted from the source code using observations on the program/classes/module/functions and graph representation of software source code. The more complex the source code, the more difficult it is to grasp its meaning and to understand and document it. These studies are based on work carried out in the 1970s by Curtis, Halstead, and McCabe (Curtis, 1979; Halstead, 1978; McCabe, 1976). These are all internal measures, which in turn affect the external point of view of maintainability.

While most product quality publications have described product quality in terms of the final software and system performance, sound software engineering practice requires that key artifacts impacting quality be evaluated throughout the software development and maintenance processes.

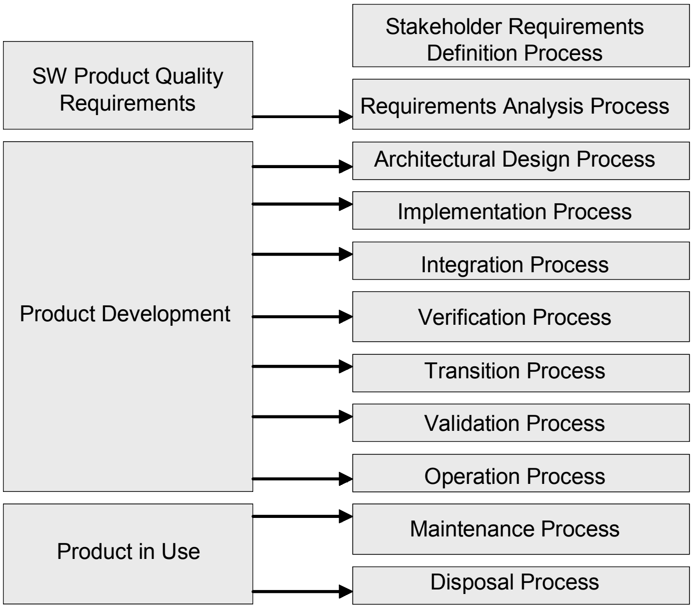
# Practical Considerations

## Software Quality Requirements

In order to implement software process and product quality, the software engineer must choose and document quality characteristics at the outset of the projects during the requirements definition process. A generic life cycle model like ISO/IEC 15288 — *System life cycle processes* demonstrates generic stages of a typical development process (Figure 8) and the mapping to show *how* standards can be used across all stages of the life cycle of a software product.

We have already stated that both functional and non-functional requirements are captured using the requirements definition process (Pfleeger, 2001). Functional requirements specify what a product should do. Non-functional requirements, also called quality requirements, place constraints on the functional requirements. The quality requirements need to refer to a quality model, like the one described in ISO/IEC 9126-1, to be well understood. Quality requirements address important issues of quality for software products. Software product quality requirements are needed for:

*Figure 8. Product life cycle mapping to the technical process life cycle (Suryn et al., 2003)*



- Specification (including contractual agreement and call for tender)
- Planning (e.g., feasibility analysis and translation of external quality requirements into internal quality requirements)
- Development (early identification of quality problems during development)
- Evaluation (objective assessment and certification of software product quality)

The requirements definition process typically takes all stakeholders' needs, requirements, and expectations into consideration and ranks them by priority. All relevant stakeholders must be identified before the exercise begins. A first analysis activity during the requirements definition process is aimed at identifying user requirements, sometimes also called business requirements. This view of software requirements should be documented in wording that is easily understood by both users and customers.

Business requirements describe the functionality that the software executes as needed by the stakeholders. Good functional requirements are expressed clearly and are consistent. All requirements need to be correct, non-redundant, and not be in conflict with other requirements. Each requirement has to be uniquely identified and traceable to identified stakeholders. They should also be documented in the business requirements specification (BRS). If a stakeholder requirement is not taken into consideration, the software engineer needs to state the reason why in the documentation.

The second analysis activity is aimed at transforming stakeholder functional requirements into system requirements that can be used by designers to build software. This more technical view of requirements is called a software requirements specification (SRS). During this second analysis activity, the quality requirements are added to the functional requirements. They need to be stated clearly and unambiguously. It is the software engineer's responsibility to go through each quality characteristic and assess the need to include it in the project. System requirements should also be verifiable and state both the business requirements and the non-functional requirements that the software must possess in order to satisfy all stakeholder requirements. If they do not, they may be viewed, interpreted, implemented, and evaluated differently by different stakeholders. If the software engineer does not become involved in promoting the non-functional requirements, they may be ignored altogether. This may result in software which is inconsistent with user expectations and therefore of poor quality.

We have often talked about stakeholders, but who are they? Stakeholders include all individuals and organizations with a legitimate interest in the software.

Different stakeholders will express different needs and expectations that represent their own perspective. For example, operators, testers, and quality engineers are stakeholders. These needs and expectations may change throughout the system's life cycle and need to be controlled when changed. The stakeholders rarely express non-functional requirements, as they may have an unclear understanding of what quality really means in a software process. In reality, they typically perceive it initially as more of an overhead than anything else. Quality requirements will often appear as part of a contractual agreement between acquirer and developer. They may also be required for a specific product quality evaluation when some level of software criticality is reached and human lives are at risk.

We have already stated that a software quality requirement should refer to a specific quality model. To clearly specify quality requirements, ISO/IEC working group six (WG6) of the software engineering subcommittee (SC7) is developing a standard as part of the new ISO/IEC 25000 — *SQuaRE* (*S*oftware Product *Qua*lity *R*equirements and *E*valuation) initiative. The Quality Requirements standard will include a new section for the specification of quality requirements.

A number of obligations are likely to be imposed on software engineers that demand conformance to this new section of ISO/IEC 25000. First, each quality requirement will need to be categorized and prioritized according to an explicit quality model, like:

- Quality-in-use requirement;
- External quality requirement;
- Internal quality requirement.

A software quality requirement will also need to be specified with reference to a set of functional properties. It is also possible that critical functional properties will create an obligation to impose some quality characteristics on a quality requirement.

Since each quality requirement needs to be assessed throughout all the life cycle stages, it needs to be specified in terms of a measurement function and be assigned target values. This new standard will provide examples of measures which may be adapted to specific organizational needs for specifying quality requirements. This new standard is also likely to recommend that software engineers develop and maintain a list of quality measures that are used consistently across their organization. Applying the same set of measures makes it possible to create a homogeneous repository of historical data, which in turn helps in the creation of predictive models such as productivity and estimation models.

The new standard could also focus on the notion of the degree of uncertainty associated with a quality requirement. Measures tending to yield large deviations need to be explained because of their erratic behavior. Explanations include the measurement process of that measure and the errors associated with each of its components.

Many quality standards already identify the review and approval stages, so that the set of quality requirements is identified and reviewed, and who approved them is clearly stated. The last item likely to appear in such a standard refers to the need for quality requirements, just like functional requirements, to be managed according to a configuration and change management system throughout the entire project.

## Software Quality Measurement

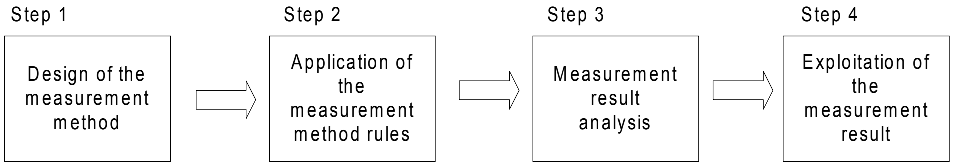
The main reason why product quality measurement is only slowly gaining acceptance is the inherent difficulty of measuring in a practical way its quality-impacting processes and artifacts throughout the software engineering process. Some product quality perspectives (e.g., size and source code structure) have been easier to measure than others (e.g., suitability, replaceability, and attractiveness). "Furthermore, some of the dimensions of quality (e.g., usability, maintainability and value to the client) are likely to require measurement in qualitative rather than quantitative form" (ISOTR19759, 2005).

For practical purposes, the software engineer will need to define the quality requirements and the means to assess them early on in the life cycle of the project. During the requirements definition stage, statements about the quality of each quality characteristic are defined as expressing the capability of the software to perform at, and maintain, a specified level of service. The measurement approach requires that individuals who carry out software engineering activities capture the measures and make a judgment about the degree to which the many software properties fulfill their specified requirements as agreed by the stakeholders. Both qualitative and quantitative measures can be collected during all stages of the software life cycle.

The ISO9126 quality model offers definitions of software properties which can be distinguished as quantitative or qualitative. Before quality attributes can be captured and validated, someone has to design them. Then, the most practical means possible must be implemented to measure them.

We measure by applying measurement method rules (see Step 2 in Figure 9). A measurement method is a logical sequence of operations that can be used by software engineering personnel to quantify an attribute on a specified scale. The

*Figure 9. Software measurement method (Abran & Jacquet, 1997)*


Step 1
Design of the measurement method
Step 2
Application of the measurement method rules
Step 3
Measurement result analysis
Step 4
Exploitation of the measurement result

result of applying a measurement method is called a base measure. The ISO9126 quality characteristics that were defined and agreed on can now be measured.

A list of "Quality Measures" is available in ISO/IEC 9126 and ISO/IEC 14598, and the standards present examples of mathematical definitions and guidance for practical measurement of internal quality, external quality, and quality in use.

The proposed ISO/IEC 25000–*SQuaRE* will help by providing guidance, as follows:

- **Measurement reference model and guide** — to present introductory explanations, the reference model and the definitions that are common to measurement primitives, internal measures, external measures, and quality-in-use measures. The document will also provide guidance to users for selecting (or developing) and applying appropriate measures;

  Example: the construction phase creates intermediate, non-executable software products that can only be evaluated from a static point of view. In this case, users will be directed to the internal quality measures standard, where they can choose the measures that best serve their information needs;
- **Measurement primitives** — to define a set of base and derived measures, or the measurement constructs for the internal quality, external quality, and quality-in-use measurements;
- **Measures for internal quality** — to define a set of internal measures for quantitatively measuring internal software quality in terms of quality characteristics and subcharacteristics;
- **Measures for external quality** — to define a set of external measures for quantitatively measuring external software quality in terms of quality characteristics and subcharacteristics;
- **Measures for quality in use** — to define a set of measures for measuring quality in use. The document will provide guidance on the use of the quality-in-use measures.

## Software Quality Improvement

As defined by the ISO/IEC 12207 (1996) standard, an Improvement Process is a process for establishing, assessing, measuring, controlling, and improving a software life cycle process. This definition assumes that an organization already has a written development process in place. A first step in any improvement strategy is to assess the current situation (i.e., a baseline), and then to identify improvement goals and implement the actions required to achieve the goals. Similarly, quality improvement should follow the same steps. Important assumptions are that the organization has stable processes such that data collected are reliable and that they are quality data. If no data are available, some baseline activities have to be performed. The measurement and analysis process area of the CMMi (SEI, 2002) describes the basic components of such a process.

The organization then has to establish quality goals. One approach is to use the well-known Goal Quality Metric (GQM) method (Basili & Rombach, 1988). The organization determines its business goals, and then the technical personnel establish quality goals in order to meet the organization's goals. The quality goals could be established using the Cost of Quality approach or a standard business case approach, that is, by evaluating the cost of prevention and evaluation and the cost of internal and external failure.

Quality could be improved, first, by removing defects as early as possible in the development life cycle using techniques such as reviews, walkthroughs, and inspections (IEEE-1028, 1997), and second, by putting in place defect prevention activities. Software inspection (Gilb & Graham, 1993; IEEE-1028, 1997; Radice,

*Figure 10. Inspection process (Adapted from Holland, 1998)*

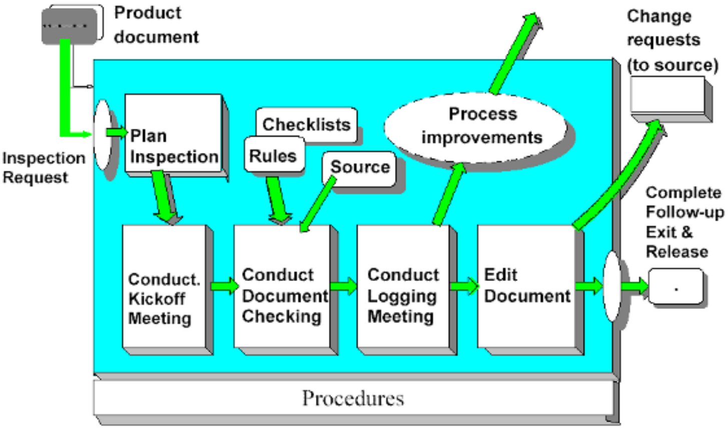


*Table 5. Distribution of defect detection effectiveness of inspections (Briand et al., 1998)*

| Defect Detection Technique | Minimum Value | Most Likely Value | Maximum Value |
|---|---|---|---|
| Design Inspections | 25% | 57% | 84% |
| Code Inspections | 19% | 57% | 70% |

*Table 6. Defect detection effectiveness (Radice, 2002)*

| CMM Maturity Level | Defect Removal Effectiveness |
|---|---|
| Level 5 | 90% - 100% |
| Level 4 | 75% - 90% |
| Level 3 | 65% - 75% |
| Level 2 | 50% - 65% |
| Level 1 | Less than 50% |

*Table 7. Distribution of average effort to detect defects (Briand et al., 1998)*

| Defect Detection Technique | Minimum Value | Most Likely Value | Maximum Value |
|---|---|---|---|
| Design Inspections | 0.58 | 1.58 | 2.9 |
| Code Inspections | 0.67 | 1.46 | 2.7 |
| Testing | 4.5 | 6 | 17 |

2002) is an example of a widely recognized quality improvement practice. A Web site displays this technique and provides a wealth of supporting information (www.goldpractices.com/practices/fi/index.php). The inspection process is typically composed of six stages: plan, kick-off meeting, document checking, logging meeting, editing (i.e., rework), and follow-up (Figure 10).

As shown in Table 5, the defect detection effectiveness of inspections can reach 84%. Radice (2002) has collected data from organizations of different CMMi maturity levels which show an effectiveness varying from as low as 50% to as much as 100% (Table 6).

It is interesting to note that using inspection to detect defects is more effective than testing. Table 7 (Briand et al., 1998) shows the average effort required to detect defects in hours per defect. Unfortunately, even though it has been around for close to 30 years (Fagan, 1976), most organizations ignore this practice. Many organizations, if not most, rely only on tests performed toward the end of the development process to identify defects, even though tests are much less effective than inspections at defect removal. One of the reasons is that inspections find defects where they occur, while testing finds only the symptoms, and then only very late in the development cycle.

## Future Trends and Conclusions

To illustrate the current state of the quality of many shrink wrap software vendors, here is a typical warranty statement as printed on a software product: *By opening this sealed software media package, you accept and agree to the terms and conditions printed below. If you do not agree,* ***do not open the package****. Simply return the sealed package. The software media is distributed on an 'AS IS' basis, without warranty. Neither the authors, the software developers nor Acme Inc make any representation, or warranty, either express or implied, with respect to the software programs, their quality, accuracy, or fitness for a specific purpose. Therefore, neither the authors, the software developers nor Acme Inc shall have any liability to you or any other person or entity with respect to any liability, loss, or damage caused or alleged to have been caused directly or indirectly by the programs contained on the media. This includes, but is not limited to, interruption of service, loss of data, loss of consulting or anticipatory profits, or consequential damages from the use of these programs. If the media is defective, you may return it for a replacement.*

As more customers demand quality from their vendors, the competitive marketplace will eventually respond to these demands, and software products will be warranted in the same way as any other products. Vendors will be held responsible for making the appropriate corrections at their expense like other industrial sectors. Already a few organizations have demonstrated that quality, productivity, cycle time, and cost could all be improved provided that senior management commit adequate resources and time to improvement. Diaz (1997, 2002) has published results of major improvement programs at Motorola and General Dynamics. For example, at General Dynamics, rework dropped from 23.2% to 6.8%, phase containment effectiveness increased from 25.5% to 87.3%, defects reported by customers dropped from 3.2 per thousand source lines of code (KSLOC) to 0.19, and productivity increased by a factor of 2.9. Such organizations have the data to manage the quality of their processes and their products. As stated by Boehm and Sullivan (2000), since design is an investment activity, knowing the economics of software engineering, these organizations can offer their customers an optimized value proposition by using the proper processes, tools, and metrics.

There is no instant gain, however, as illustrated in Figure 3. It takes time to define and stabilize a process, measure its performances, and make improvements. Those organizations that have succeeded know that improvements require not only resources, but also a major culture change. By definition, a cultural change takes time and can rarely be imposed, and it has to be managed from the top, with senior executives no longer requesting unrealistic schedules with unrealistic budgets to deliver quality products. Moreover, if software developers in the West do not shape up rapidly, what has happened and is still happening to the automobile industry may happen to them, that is, a take-over by foreign, high-quality, lower-cost manufacturers.

With the arrival of India (Moitra, 2001) and China (Ju, 2001) into the software market with their focus on quality and process maturity, the software industry will be pushed to move faster; otherwise, Darwin will favor those who survive this fight.

## References

Abran, A., & Jacquet, J.-P. (1997, June 2-6). From software metrics to software measurement methods: A process model. *Proceedings of the Third International Symposium and Forum on Software Engineering Standards (ISESS97)*, Walnut Creek, CA.

April, A., Reeker, L., & Wallace, D. (2005). Software quality. *Guide to the software engineering body of knowledge (Ironman version)* (pp. 157-170). Los Alamitos, CA: IEEE-Computer Society Press.

Arthur, L. J. (1992). *Improving software quality: An insider's guide to TQM.* New York: John Wiley & Sons.

Augustine, N. R. (1997). *Augustine's Laws* (6th ed.). Reston, VA: American Institute of Aeronautics and Astronautics.

Basili, V. R., & Rombach, H. D. (1988). The TAME project: Towards improvement-oriented software environments. *IEEE Transactions on Software Engineering, 14*(6), 758-773.

Boehm, B. W., Brown, J. R., Kaspar, H., Lipow, M., McLeod, G., & Merritt, M. (1978). Characteristics of software quality. *TRW series on software technologies.* Amsterdam, The Netherlands: North-Holland.

Boehm, B. W., & Sullivan, K. J. (2000, June 4-11). Software economics: A roadmap. *Proceedings of the Future of Software Engineering, International Conference on Software Engineering*, Limerick, Ireland.

Bouman, J., Trienekens, J., & Van der Zwan, M. (1999). Specification of service level agreements, clarifying concepts on the basis of practical research. *Proceedings of Software Technology and Engineering Practice* (pp. 11-19). Los Alamitos, CA: IEEE Computer Society.

Briand, L., El Emam, K., Laitenberger, O., & Fussbroich, T. (1998). Using simulation to build inspection efficiency benchmarks for development projects. *Proceedings of the IEEE 20th International Conference on Software Engineering (ICSE 1998)* (pp. 340-349). Los Alamitos, CA: IEEE Computer Society.

Campanella, J. (1990). *Principles of quality costs* (3rd ed). Milwaukee, WI: American Society for Quality Control.

Charette, R. N. (1989). *Software engineering risk analysis and management*. New York: McGraw-Hill.

Crosby, P. B. (1979). *Quality is free*. New York: McGraw-Hill.

Curtis, B. (1979). In search of software complexity. *Proceedings of the IEEE PINY Workshop on Quantitative Software Models* (pp. 95-106). Los Alamitos CA: IEEE Computer Society.

Cusumano, M., MacCormack, A., Kemerer, C., & Grandall, B. (2003). Software development worldwide: The state of the practice. *IEEE Software, 20*(6), 28–34. *Figures abstracted from IEEE Software*.

Diaz, M. (1997). How software process improvement helped Motorola. *IEEE Software, 14*(5), 75-81.

Diaz, M. (2002). How CMM impacts quality, productivity, rework, and the bottom line. *Crosstalk Journal*, U.S. Department of Defense, Hill Air Force Base UT, 15(3), 9-14.

Dion, R. (1993). Process improvement and the corporate balance sheet. *IEEE Software, 10*(4), 28-35. *Figure abstracted from IEEE Software.*

Dobbins, H. D. (1999). The cost of software quality. *Handbook of software quality assurance*. (3rd ed.) (pp. 195-216). Upper Saddle River, NJ: Prentice Hall PTR.

Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. *IBM System Journal*, 15(3), 182-211.

Galin, D. (2004a). *Toward an inclusive model for the costs of software quality*. *Software Quality Professional, 6*(4), 25-31.

Galin, D. (2004b). *Software quality assurance*. Harlow, UK: Addison-Wesley.

Gilb, T., & Graham, D. (1993). *Software inspection*. Wokingham, UK: Addison-Wesley.

Gotterbarn, D., Miller, K., & Rogerson, S. (1999). Computer society and ACM approve software engineering code of ethics. *IEEE Computer, 32*(10), 84-88.

Gotterbarn, F. (1999) How the new software engineering code of ethics affects you. *IEEE Software, 16*(6), 58-64.

Hailey, V. A. (1998). A comparison of ISO9001 and the SPICE framework, SPICE: an empiricist's perspective. *Proceedings of the Second IEEE International Software Engineering Standards Symposium (ISESS 1998)* (pp. 233-268). Los Alamitos, CA: IEEE Computer Society.

Haley, T. J. (1996). Software process improvement at Raytheon. *IEEE Software, 13*(6), 33-41. *Figure abstracted from IEEE Software.*

Halstead, M. H. (1978, August 21-22). Software science: A progress report. *U.S. Army Computer Systems Command Software Life Cycle Management Workshop*. IEEE.

Holland, D. (1998). Document inspection as an agent of change. In A. Jarvis & L. Hayes (Eds.), *Dare to be excellent*. Upper Saddle River, NJ: Prentice Hall.

Houston, D. (1999). Cost of software quality: Justifying software process improvement to managers. *Software Quality Professional, 1*(2), 8-16.

Humphrey, W. (1990). *Managing the software process*. Software Engineering Institute: Addison-Wesley.

IEEE 730, IEEE Std 730-2002. (2002). *IEEE Standard for Software Quality Assurance Plans*. IEEE.

IEEE 1028, IEEE Std 1028-1997. (1997). *IEEE Standard for Software Reviews*. IEEE.

IEEE 12207, IEEE/EIA 12207.0-1996. (1996). *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes.* IEEE.

IEEE-CS, IEEE-CS-1999. (1999) Software Engineering Code of Ethics and Professional Practice, IEEE-CS/ACM, 1999. Retrieved February 2, 2005, from http://www.computer.org/certification/ethics.htm

ISO9001. (2000, December 15). Quality management systems — Requirements. *International Organisation for Standardisation* (3rd ed.). Geneva, Switzerland: International Organisation for Standardisation.

ISO90003, International Standards Organization. Software Engineering: *Guidelines for the application of ISO9001:2000 to computer software*, ISO/

IEC Standard 90003:2004. (2004). Geneva Switzerland: International Organization for Standardization/International Electrotechnical Commission.

ISO9126, International Standards Organization. (2001). *Software Engineering-Product Quality-Part 1: Quality Model*, ISO/IEC Standard 9126-1. Geneva, Switzerland: International Organization for Standardization/International Electrotechnical Commission.

ISOTR19759, International Standards Organization. (2005). *Software Engineering Body of Knowledge* (Tech. Rep. No. ISO/IEC PRF TR 19759).

Ju, D. (2001). China's budding software industry. *IEEE Software, 18*(3), 92-95.

Knox, S. T. (1993). Modeling the cost of software quality. *Digital Technical Journal, 5*(4), 9-16.

Laporte, C. Y., & Trudel, S. (1998). Addressing the people issues of process improvement activities at Oerlikon Aerospace. *Software Process-Improvement and Practice, 4*(1), 187-198.

Levenson, N., & Turner, C. (1993). An investigation of the Therac-25 accidents. *IEEE Computer, 26*(7), 18-41.

Mandeville, W. A. (1990). Software cost of quality. *Proceedings of the IEEE Journal on Selected Areas on Communications, 8*(2), 315-318.

May, W. (2002). *A global applying ISO9001:2000 to software products. Quality Systems Update, 12*(8), 7-13.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, 2*(4), 308-320.

McCall, J., Richards, P., & Walters, G. (1977). *Factors in software quality* (Vol. I-III). New York: Rome Air Defense Centre.

Moitra, D. (2001). India's software industry. *IEEE Software, 18*(1), 77-80.

Niessink, F. (2000). *Perspectives on improving software maintenance.* Doctoral dissertation, Dutch Graduate School for Information and Knowledge Systems, Utrecht, The Netherlands.

Paulk, M., Curtis, B., Chrissis, M. B., & Weber, C. V. (1993). *Capability maturity model for software version 1.1* (Tech. Rep. No. CMU-SEI-93-TR-24). Software Engineering Institute.

Pfleeger, S. L. (2001). *Software engineering: Theory and practice* (2nd ed.). Englewood Cliffs, NJ: Prentice Hall.

Radice, R. (2002). *High quality low cost software inspections.* Andover, MA: Paradoxicon.

SEI, Software Engineering Institute. (2002). *Capability maturity model integration for software engineering (CMMi)* (Version 1.1) (Tech. Rep. No. CMU/SEI-2002-TR-028) (pp. 94-528). Pittsburgh, PA: Carnegie Mellon University.

Slaughter, S. A., Harter, D. E., & Krishnan, M. A. (1998). Evaluating the cost of software quality. *Communications of the ACM, 41*(8), 10-17.

Suryn, W., Abran, A., & April, A. (2003). ISO/IEC SQuaRE: The second generation of standards for software product quality. *Proceedings of the Seventh IASTED International Conference on Software Engineering and Applications*, Marina del Rey, CA.

Tylor, E. B., Sir. (1871). *Primitive culture: Researches into the development of mythology, philosophy, religion, art and custom*. London: John Murray.

Wiegers, W. (1996). *Creating a software engineering culture*. New York: Dorset House.

## Chapter III

# Different Views of Software Quality

Bernard Wong, University of Technology Sydney, Australia

## Abstract

*This chapter examines the different definitions of quality and compares the different models and frameworks for software quality evaluation. It will look at both historical and current literature. The chapter will give special attention to recent research on the Software Evaluation Framework, a framework for software evaluation, which gives the rationale for the choice of characteristics used in software quality evaluation, supplies the underpinning explanation for the multiple views of quality, and describes the areas of motivation behind software quality evaluation. The framework has its theoretical foundations on value-chain models, found in the disciplines of cognitive psychology and consumer research, and introduces the use of cognitive structures as a means of describing the many definitions of quality. The author hopes that this chapter will give researchers and practitioners a better understanding of the different views of software quality, why there are differences, and how to represent these differences.*

## Introduction

Adopting an appropriate Quality Assurance philosophy has been often viewed as the means of improving productivity and software quality (Hatton, 1993; Myers, 1993). However unless quality is defined, it is very difficult for an organization to know whether it has achieved quality clearly. To date, this has usually involved conformance to a standard such as AS3563 or ISO9001 or following the Capability Maturity Model of the SEI. The challenge often faced is that one finds as many definitions of quality as writers on the subject. Perhaps, the latter have been remarkably few in number considering the obvious importance of the concept and the frequent appearance of the term quality in everyday language.

Though the topic of software quality has been around for decades, software product quality research is still relatively immature, and today it is still difficult for a user to compare software quality across products. Researchers are still not clear as to what is a good measure of software quality because of the variety of interpretations of the meaning of quality, of the meanings of terms to describe its aspects, of criteria for including or excluding aspects in a model of software, and of the degree to which software development procedures should be included in the definition. A particularly important distinction is between what represents quality for the user and what represents quality for the developer of a software product.

Perceptions of software quality are generally formed on the basis of an array of cues. Most notably, these cues include product characteristics (Boehm et al., 1976; Carpenter & Murine, 1984; Cavano & McCall, 1978; McCall et al., 1977; Kitchenham & Pfleeger, 1996; Kitchenham & Walker, 1986; Sunazuka et al., 1985). The cues are often categorized as either extrinsic or intrinsic to the perceived quality. Simply, intrinsic cues refer to product characteristics that cannot be changed or manipulated without also changing the physical characteristics of the product itself; extrinsic cues are characteristics that are not part of the product (Olson & Jacoby, 1972). Price and brand are thus considered to be extrinsic with respect to product quality.

This chapter examines the different definitions of quality and compares the different models and frameworks for software quality evaluation. This chapter will address both the topics of interest for the information systems community and the software engineering community. It will look at both historical and current literature. The chapter will give special attention to recent research on the Software Evaluation Framework, a framework for software evaluation, which gives the rationale for the choice of characteristics used in software quality evaluation, supplies the underpinning explanation for the multiple views of quality, and describes the areas of motivation behind software quality evaluation. The framework has its theoretical foundations on value-chain models, found in the disciplines of cognitive psychology and consumer research, and introduces the use of cognitive structures as a means of describing the many definitions of quality.

## Background

Software users today are demanding higher quality than ever before, and many of them are willing to pay a higher price for better quality software products. The issue of software quality has come to the forefront in Europe, the United Kingdom, the United States, and more recently Australia. The quality movement in software is not new. A search of the information systems literature has shown that attempts to achieve quality software have been on-going for many years. Software quality models include the product-based view (Boehm et al., 1976; Carpenter & Murine, 1984; Cavano & McCall, 1978; McCall et al., 1977; Kitchenham & Pfleeger, 1996; Kitchenham & Walker, 1986; Sunazuka et al., 1985), process focused models following a manufacturing-based view (Coallier, 1994; Dowson, 1993; Humphrey, 1988; Ould, 1992; Paulk, 1991), and more recently, techniques and tools to cater for the user-based view (Delen & Rijsenbrij, 1992; Erikkson & McFadden, 1993; Juliff, 1994; Kitchenham, 1987; Kitchenham & Pickard, 1987; Thompsett, 1993; Vidgen et al., 1994). However, the many models and approaches seem to contradict each other at times. Garvin (1984) tries to explain these contradictions by introducing different views of quality. He describes the models as transcendental-based view, product-based view, manufacturing-based view, economic-based view, and user-based view, which we will define later.

As the software market matures, users want to be assured of quality. They no longer accept the claims of the IT department at face value, but expect demonstrations of quality. There is a firm belief that an effective quality system leads to increased productivity and permanently reduced costs, because it enables management to reduce defect correction costs by emphasizing prevention. A better-designed development process will accrue fewer lifetime costs, and higher productivity will therefore be possible. As such, many attempts at improving quality have been to focus on the development processes. However, productivity measurements are worthless unless they are substantiated by quality measurements. To develop software quickly, on time, and within budget is no good if the product developed is full of defects. It is the view of many that the costs of correcting defects in software late in development can be orders of magnitude greater than the cost of correcting them early (Kitchenham &

Pfleeger, 1996; Pfleeger, 2001). Preventing defects in the first place can save even more. Productivity measurements require quality measurements.

The market for software is increasingly a global one, and organizations will find it increasingly difficult to succeed and compete in that market unless they produce and are seen to produce quality products and services. From simple issues concerning efforts required to develop software systems, there have been a myriad of developments and directions that have occurred since the beginning of the last decade concerning the process of software development. From all of these new directions, the consideration of the quality of the software produced is one of the major thrusts that have occurred. Software used to be a technical business, in which functionality was the key determinant of success. Today, one can no longer just adopt this view.

## The Meaning of Quality

The Oxford English Dictionary (OED, 1990) states that quality is the "degree of excellence." While this definition of quality is found in an internationally accepted dictionary, it must be pointed out that there are many variations to this definition, which are not given here.

A formal definition of quality is provided by the International Standards Organization (ISO, 1986): *The totality of features and characteristics of a product or service that bear on its ability to satisfy specified or implied needs.*

This standard definition associates quality with the products' or services' ability to fulfill its function. It recognizes that this is achieved through the features and characteristics of the product. Quality is associated both with having the required range of attributes and with achieving satisfactory performance in each attribute.

The best-known and most widely adopted definition of quality is simply "fitness for use" or some variant thereof. For example, Wimmer (1975) and Genth (1981) adopted the definition "fitness for use." Box (1984) defined quality as "the degree to which a product fulfils its functions, given the needs of the consumer," and Kotler (1984) speaks of "the rated ability of the brand to perform its functions as perceived by consumers." Kawlath (1969) defined perceived quality as "the fitness for certain goals."

Maynes (1976) proposed the following definition: "The quality of a specimen (a product/brand/seller/combination) consists of the extent to which the specimen provides the service characteristics that the individual consumer desires." A similar definition is suggested by Monroe and Krishnan (1985): "Perceived product quality is the perceived ability of a product to provide satisfaction relative to the available alternatives." Kuehn and Day (1962) stated that the quality of a product depends on how well it fits in with patterns of consumer preferences.

Kupsch and Hufschmied (1978) regarded quality as a bundle of need-satisfying attributes. A similar position was taken by Oxenfeldt (1950). Bockenhoff and Hamm (1983) defined quality as the composite of all product attributes irrespective of whether these attributes are in reality existent in the product and objectively measurable, and whether consumers are correct in their evaluations.

Thurstone (1985) suggested: "Quality is the index that reflects the extent to which the customer feels that his need, the product, and his expectations for that product overlap." He concluded that the relevant measure of quality does not reside in the product but in the customer. A similar position is taken by Wolff (1986) who argued that quality should be measured from the customer's perspective: "If the customer says it's good, it's good; if he says it's bad, it's bad."

Trenkle (1983) distinguished three manifestations of quality:

- neutral concept (i.e., "much quality" - "not much quality"), defined as the nature of a product, given by the whole of all the attributes which discriminates the product from the other products in the same category;
- evaluative concept ("good quality" - "bad quality"), defined as the fitness for use of a product, given by the whole of all the attributes that are relevant to the evaluation of the product;
- positive judgment ("quality products"), defined as superior or excellent with respect to all attributes.

Some researchers have followed Wittgenstein's (1953) linguistic approach that "the meaning of a word is its use in the language" and explored the word quality in terms of everyday use of the term. For instance, quality often means reliability, general approval, or general excellence in the eyes of the consumers (Holbrook & Corfman, 1983). The ordinary language approach, however, does not get one very far conceptually with respect to the meaning of perceived quality.

## The Different Perspectives of Quality

There are numerous definitions and meanings given to quality. The previous section lists some of the popular definitions; however, they do not necessarily describe the perceptions of quality held by all individuals. David Garvin (1984) concluded in his paper "that quality is a complex and multifaceted concept." He described quality as being made up of five different perspectives — the transcendental-based view, the product-based view, the manufacturing-based view, the economics-based view, and the user-based view.

## *Transcendental-Based View*

According to the transcendental-based view, quality is synonymous with "innate excellence." It is both absolute and universally recognizable, a mark of uncompromising standards and high achievement. Nevertheless, proponents of this view claim that quality cannot be defined precisely; rather, it is a simple, not analyzable property that we learn to recognize only through experience. This definition borrows heavily from Plato's discussion of beauty, where he argues that beauty is one of the "platonic forms," and, therefore, a term that cannot be defined. Like other such terms that philosophers consider to be "logically primitive," beauty (and perhaps quality as well) can be understood only after one is exposed to a succession of objects that display its characteristics.

## *Product-Based View*

Product-based definitions are quite different; they view quality as a precise and measurable variable. According to this view, differences in quality reflect differences in the quantity of some ingredient or attribute possessed by a product. For example, high quality ice cream has high butterfat content, just as fine rugs have a large number of knots per square inch. This approach lends a vertical or hierarchical dimension to quality, for goods can be ranked according to the amount of the desired attribute they possess. However, an unambiguous ranking is possible only if virtually all buyers consider the attributes in question preferable.

Product-based definitions of quality first appeared in the economics literature, where they were quickly incorporated into theoretical models. In fact, the early economic research on quality focused almost exclusively on durability, simply because it was so easily translated in the above framework. Since durable goods provide a stream of services over time, increased durability implies a longer stream of services — in effect, more of the good. With regard to software, durability can be understood as the length of time a piece of software is used in a live environment. Of course, one recognizes that the software may need upgrades from time to time, as products can require repairs or modifications. Quality differences could, therefore, be treated as differences in quantity, considerably simplifying the mathematics.

There are two obvious corollaries to this approach. First, higher quality can only be obtained at higher cost. Because quality reflects the quantity of attributes that a product produces, and because attributes are considered to be costly to produce, higher quality goods will be more expensive. Second, quality is viewed as an inherent characteristic of goods, rather than as something ascribed to them. Because quality reflects the presence or absence of measurable product attributes, it can be assessed objectively and is based on more than preferences alone.

## *User-Based View*

Garvin (1984) states that the user-based definitions start from the opposite premise that quality "lies in the eyes of the beholder." Individual consumers are assumed to have different wants or needs, and those goods that best satisfy their preferences are those that they regard as having the highest quality. This is an idiosyncratic and personal view of quality and one that is highly subjective. In the marketing literature, it has led to the notion of "ideal points:" precise combinations of product attributes that provide the greatest satisfaction to a specified consumer; in the economics literature, to the view that quality differences are captured by shifts in a product's demand curve and in the operations management literature, to the concepts of "fitness for use." There are two problems in each of these concepts. The first is practical — how to aggregate widely varying individual preferences so that it leads to meaningful definitions of quality at the market level. The second is more fundamental — how to distinguish those product attributes that connote quality from those that simply maximize consumer satisfaction.

A more basic problem with the user-based view is its equation of quality with maximum satisfaction. While the two are related, they are by no means identical. A product that maximizes satisfaction is certainly preferable to one that meets fewer needs, but is it necessarily better as well? The implied equivalence often breaks down in practice. A consumer may enjoy a particular brand because of its unusual features, yet may still regard some other brand as being of higher quality. In the latter assessment, the product's objective characteristics are also considered.

Even perfectly objective characteristics, however, are open to varying interpretations. Today, durability is regarded as an important element of quality. Long-lived products are generally preferred to those that wear out more quickly. This was not always true until the late nineteenth century; durable goods were primarily possessions of the poor, for only wealthy individuals could afford delicate products that required frequent replacement or repair. The result was a long-standing association between durability and inferior quality, a view that changed only with the mass production of luxury items made possible by the Industrial Revolution.

## *Manufacturing-Based View*

Manufacturing-based definitions focus on the supply side of the equation and are primarily concerned with engineering and manufacturing practice. Virtually, all manufacturing-based definitions identify quality as “conformance to requirements.” Once a design or a specification has been established, any deviation implies that the specification may not be complete. Excellence is equated with meeting specifications and with “making it right the first time.”

While this approach recognizes the consumer’s interest in quality, its primary focus is internal. Quality is defined in a manner that simplifies engineering and production control. On the design side, this has led to an emphasis on reliability engineering (Boehm, 1963; Garvin, 1984); on the manufacturing side, to an emphasis on statistical quality control (Garvin, 1984; Juran & Gryna, 1980). Both techniques are designed to weed out deviations early: the former, by analyzing a product’s basic components, identifying possible failure modes, and then proposing alternative designs to enhance reliability; and the latter, by employing statistical techniques to discover when a production process is performing outside acceptable limits.

Each of these techniques is focused on the same end: cost reduction. According to the manufacturing-based approach, improvements in quality (which are equivalent to reductions in the number of deviations) lead to lower costs, for preventing defects is viewed as less expensive than repairing or reworking them (Crosby, 1978; Erikkson & McFadden, 1993; Garvin, 1984). Organizations are, therefore, assumed to be performing suboptimally: were they only to increase their expenditures on prevention and inspection — testing prototypes more carefully or weeding out a larger number of defective components before they become part of fully assembled units — they would find their rework, scrap, and warranty expenses falling by an even greater amount.

## *Economic-Based View*

Economic-based definitions define quality in terms of costs and prices. According to this view, a quality product is one that provides performance at an acceptable price or conformance at an acceptable cost. Under this approach, a $500 running shoe, no matter how well constructed, could not be a quality product, for it would find few buyers.

Garvin shows how each of these views can be used to define product quality. Engineers who believe a product has set characteristics often adopt the product-based view. These characteristics are used as the measure of quality. The manufacturing-based view is adopted when one believes the quality development process determines a quality product. And more recently, many organizations have been certified with the Capability Maturity Model, CMM, or SPICE (Paulk, 1991; Paulk et al., 1993). Certification assesses the manufacturing process and is awarded on successfully having a quality management system in place. Economists who believe that price has a correlation with quality adopt the economics-based view. And lastly, the user-based view is the one which emphasizes that each individual will have his or her own perception of quality. Garvin (1984) states that most existing definitions of quality fall into one of these categories, be they conformance to specification, meeting user requirements, or best development practice.

In a more recent paper, Braa and Ogrim (1994) also tried to expand the definition of quality. Rather than just focusing on technical properties, they considered the functional and organizational aspects of quality. Five aspects of quality are introduced.

### *Technical Quality*

Technical quality refers to a system's structure and performance. The technical quality of a computer system is the basis of its functionality — the computer must perform expected operations. However, there is no need for a technically excellent computer system; it is more important that it suits the work tasks it is supposed to support.

### *Use Quality*

By use quality we mean quality as experienced by the users when working with the computer-based system. Use quality is difficult to specify in advance. It is often necessary to experiment with design models, such as prototypes, to express needs and claims (Greenbaum & Kyng, 1991). Throughout the specification and design activities, users and developers will then learn about the limitations and the possibilities of the computer system in use. Consequently, learning is an important factor in the specification process, as well as in design.

## *Aesthetic Quality*

In many other disciplines, such as the car industry, or building industry, aesthetic quality is used to evaluate the quality of artifacts. The idea of aesthetics is usually related to physical objects. However, aesthetics can also be applied to immaterial objects, for example, the notion of elegant proofs in mathematics. However, the aesthetic perspective is almost always neglected in software development (Dahlborn & Mathiassen, 1993; Stolterman, 1991). One possible exception is the design of user interfaces. An aspect of aesthetic quality is "elegance," introduced as a criterion by which to assess quality (Checkland & Scholes, 1990). An attempt to increase aesthetic quality and make it more visible is to raise questions such as: Is the transformation well designed? Is it aesthetically pleasant? Is it overcomplicated? Is it over- or under-engineered? These assessments allow the users' subjective experiences as well as the professionals' experience of similar systems to be included in the judgment of quality.

## *Symbolic Quality*

Computer systems are not only artifacts, they are also used as symbols in the organization. This view is supported by Feldman and March's (1981) study of the use of information in organizations. They found that information is often used symbolically, e.g., signaling a well-driven organization, independent of whether the information is used or not. Symbolic, as well as aesthetic, aspects of computer systems are important factors in tailoring a system to the business philosophy and organization culture. For example, an organization wishing to have an innovative and modern image should have computer systems with graphical user interfaces and colors. Symbolic quality may be contradictory to the use quality. This can be illustrated by a hotel owner who wanted to install a computerized reception system. It turned out that the main purpose of the system was to create the impression of a modern, successfully run hotel and to recruit educated personnel. Not much attention was paid to the functionality of the system. If the motivation is of a symbolic character, as in this example, the use quality will probably be ignored.

## *Organizational Quality*

When a computer system is well adapted to the organization, it can be said to be of high organizational quality. When assessing the quality of information systems, questions of economy, power and interests will arise sooner or later: Are the computer systems developed for the interests of individual users, for groups of users, or for the organization as a whole? Different user groups may have diverging, perhaps contradictory, ideas of what signifies good quality. A personnel control and wage system might represent good use quality for management and the personnel department, but not for the workers who are being controlled. A medical journal system in a hospital may be of good quality for the doctors, but at the same time decrease the influence of the nurses and use of their competence. This may, in turn, decrease the quality if their work. The different groups of users make different demands on functionality, in order to support their work. If these different interests are not met, an intersection of several interests is developed and the result could be a computer system, which is not suited for anyone. This, in turn, may decrease the organizational quality of the system.

Hyatt and Rosenberg (1996) add another quality perspective, the project manager's view. The project manager views the software quality as "software that works well enough to serve its intended function and is available when needed". Project managers are usually concerned in producing software that is reliable, maintainable, and keeps customers satisfied. They usually face budget and/or time constraints, which will impede either one of their quality goals. The constraints will potentially sacrifice the testing (level, extent, and depth), walkthrough inspections, and documentation, which are some of the factors contributing to software quality. While Hyatt and Rosenberg focus only on the view of the project manager, it must be pointed out that many definitions of quality share this same view (Box, 1984; Genth, 1981; Wimmer, 1975).

## The Software Perspective of Quality

Researchers in the software engineering area have tried different ways to define quality. In fact, they have moved progressively from the product-based view, to the manufacturing-based view, and more recently to the user-based view. They have also focused heavily on technical quality with a growing consideration for use quality and aesthetic quality. However, no models in software engineering seem to exist following the transcendental, economic, aesthetic, symbolic, or organizational based views of quality.

Table 1 presents some of the software quality models introduced in this chapter. There are other quality models; however, these are the most discussed models in the literature. These are only the most notable among many in research. At first glance, the list of software quality models gleaned from a review of literature is daunting. One might conclude that they are many and varied; however, many of these models are developments of earlier ones, and some are combinations of others with and without uniqueness. For example, ISO quality factors are very similar to those in the earlier McCall and Boehm models. Bootstrap, Trillium, and MMM are all evolutions of CMM. SPICE is an evolution of Bootstrap, CMM,

*Table 1. Various software quality models and their relationship to Garvin's and Braa's quality perspectives*

| Software quality models | Garvin's quality perspective | Braa's quality perspective |
|---|---|---|
| McCall | Product view | Technical Quality |
| Boehm | Product view | Technical Quality |
| Gilb | Product view | Technical Quality |
| Murine | Product view | Technical Quality |
| Schmitz | Product view | Technical Quality |
| Schweiggert | Product view | Technical Quality |
| Willmer | Product view | Technical Quality |
| Hausen | Product view | Technical Quality |
| ISO9000-3 | User view | Technical Quality |
| ISO9126 | User view, product view | Use Quality |
| Dromey | Product view | Technical Quality |
| COQUAMO | User, manufacturer, product view | Technical Quality, Use Quality |
| Squid | User, manufacturer, product view | Technical Quality, Use Quality |
| CMM | Manufacturer view | Technical Quality |
| Bootstrap | Manufacturer view | Technical Quality |
| Trillium | Manufacturer view | Technical Quality |
| MMM | Manufacturer view | Technical Quality |
| Scope | Manufacturer view | Technical Quality |
| Processus | Manufacturer view | Technical Quality |
| SPICE | Manufacturer view | Technical Quality |
| QFD | User, manufacturer, product view | Technical Quality, Use Quality |

trillium, and ISO9001. SCOPE and PROCESSUS are efforts at combining process and product through connecting ISO9001 and ISO9126. And SQUID uses elements of ISO9126 and McCall. In most cases, the reworking can be considered one researcher's improvements on another's. This leaves few originals. Shari Pfleeger (1997) states that since models illustrate the relationship between apparently separate events, the lack of real variety in models in software engineering is symptomatic of a lack of system focus.

The research on software quality indicates that it is divided into two main schools of thought. There is the Product school, where practitioners advocate that to clearly define, measure, and improve quality, one must measure characteristics of the software product which best influence quality. Then there is the Process school, where practitioners advocate that one may be unable to properly define quality, but states that a product's quality is improved when the software engineering process used to make the product is improved. A process model's quality factors refer to the product development life cycle as a way of expressing the product's quality.

According to this product-process division, Boehm's model, the COQUAMO model, McCall's model, Murine's SQM model, ISO9126, SQUID, and Dromey's model are all examples of product models: they focus on the final product. The process model includes ISO9001-3, CMM, SPICE, BOOTSTRAP, Trillium, and MMM. SCOPE and PROCESSUS are a composite of both product and process models.

The Product Quality models are basically those models that view quality as inherent in the product itself. These models decompose the product quality into a number of quality factors (some models describe it as quality features, criteria, or characteristics). They are decomposed further into lower levels. The lowest are the metrics to measure or indicate the criteria. The metrics are used to assess the quality of the software. The result of the assessment is compared against a set of acceptance criteria. This determines whether the software can be released. Many of the software quality models in this group do not suggest what metric values are acceptable and hence leave it to the software practitioner to decide.

The Process Quality models are those models that emphasize process improvement and development standards, with the aim that the process improvements will lead to improving the quality of the product. In the past, much software development was performed without proper process and management control. The software developers were more guided by intuitive approach or habit, rather than by using any particular standards. Rae et al. (1995) emphasizes the importance of standardizing the development process and states that it will reduce software development to a rational and well-defined series of tasks, which can be managed and controlled (Rae et al., 1995). Watts Humphrey (1988), in introducing the CMM, stated that the premise of the CMM lies in the relationship between the quality of the process and the quality of the product. The belief is that effective process management will be the key factor in producing high quality software. However, Gillies (1992), Paulk et al. (1993), and Rae et al. (1995) indicated that the adoption of a good quality process will not automatically guarantee a good quality product.

Schneidewind (1996) and Fenton (1996) debated whether the existence of a standard process is good enough to improve the quality of software. Schneidewind's view is that there is a need for a standard first, even though it is not perfect. Fenton argued that a good standard process is required to improve the quality of a product and a bad standard will not help at all. In my opinion, a standard is necessary as a guideline. However, as resources to set up the standards are scarce, they have to be initially established as well as possible. The standards have to be constantly reviewed and updated, because the technology and software engineering methodology change. Fenton suggested that organizations should contribute to the resources.

Voas (1997) criticizes the software process movement in general and claims that the focus of software process movement and methods on improving quality give the impression that "clean pipes can produce only clean water." He states that we must not forget the part to be played by product oriented methods, with their focus on software metrics and system level testing that allow the developing product to be sampled for its internal quality. He claims that the preoccupation with process quality in the software industry is partly a result of the difficulty of reliable testing. And software metrics, he adds, as do others such as Gulezian (1995), brings its own set of problems related to what exactly measures should be, as well as how and when they are best measured. For example, although software reliability assessment has been accepted as a measure of software quality, most reliability models predict failure by looking at error history (Voas, 1997). Unfortunately, this data can be computed differently by different models, so the reliability results may not even be reliable. Gulezian states that structural metrics cannot capture the dynamics of software behavior where an input is transformed into an output by a set of instructions. Thus we see from this direction that much of the future of software quality lies linked to research in metrics, and providing some acknowledgment of the roles of both product and process measurement in the quality of a piece of software. While much more can be said of the process models, focus in the rest of the chapter will be given to product quality.

## Criticism of Models

The product models, particularly McCall's and Boehm's models (and the later ISO9126), have had similar criticisms aimed at them from multiple directions (Rozman et al., 1977). These criticisms have all stated that there is a lack of clear criteria for selection of not only the higher-level quality attributes but also of the subcharacteristics. There is a lack of consensus between researchers on what level a quality characteristic should be placed and for the apparent arbitrary placement of these characteristics, for the lack of agreement on terminology, and for the paucity of discussion of the measurement aspect of each of the models. An additional complication and criticism of McCall's and Boehm's models is that each of the quality subcharacteristics is not related to a single quality factor. This complicates measurement and interpretation of results. For example, McCall's model shows the quality factor *flexibility* influenced by quality characteristics of *self-descriptiveness, expandability, generality, modularity*. To further confound interpretation of any measurements that might have been gathered, these same quality subcharacteristics also influence the quality factors *portability, reusability, interoperability* as well as *testability* and *maintainability*. In ISO9126 each subcharacteristic influences only one high level quality factor — the model is completely hierarchical.

Gillies (1992) and Kitchenham and Pfleeger (1996) stated that there are a number of issues with the hierarchical models:

1. The selection of quality factors and criteria seem to be made arbitrarily. It implies that there is no rationale for determining:

- which factors should be included in the quality model and
- which criteria relate to a particular quality factor.

2. The hierarchical models cannot be tested or validated, because they do not define the quality factors in a measurable way.
3. The measurement methods of both models have a large degree of subjectivity, hence the software quality cannot be measured objectively.
4. In 1987, Perry Gillies (1992) pointed out that some quality factors or criteria conflict with each other due to inverse relationships. For example, there is an inverse relationship between usability and efficiency. The improvements in human-computer interface will increase the usability of the software; however, it will lead to reduction in efficiency as more coding will be required.

Despite the criticism, both models (especially McCall's) form the basis for most of the research in software quality today.

Kitchenham and Walker (1986) made a number of observations. They stated that there seemed to be little supporting rationale for including or excluding any particular quality factor or criteria. In fact, definitions of quality factors and criteria were not always consistent when comparing one model to the next. Different terminologies between the models exist, with little explanation for the meanings behind them. There was no rationale for deciding which criteria relate to a particular factor. Thus the selection of quality characteristics and subcharacteristics can seem arbitrary. The lack of rationale makes it impossible to determine whether the model is a complete or consistent definition of quality.

Though quality has been widely used and accepted, the differences between them highlights how one's view can dictate the perception of quality. The degree of subjectivity varies substantially from one question to another, even though all responses are treated equally. This variation makes combining metrics difficult, if not impossible. Moreover, when appropriate, the detail and complexity of the response should be reflected in a richer measurement scale. For example, while it is reasonable to expect a yes or no response to the question "Does this module have a single exit or entry point?", questions about documentation clarity probably require a multiple-point ordinal scale to reflect the variety of possible answers.

The models recommend direct measures, but most of them do not describe clearly how to conduct the measure. There appears to be no description of how the lowest level metrics are composed into an overall assessment of higher leveled characteristics. In particular, then, there is no means for verifying that the chosen metrics affect the observed behavior of a factor. That is, there is no attempt to measure factors at the top of the hierarchy, so the models are not testable.

## More Recent Developments of Product Models

### *Dromey's Model*

A more recent and promising model of product quality is found in Dromey (1995), who strongly pushes for a more comprehensive software quality model. He holds that software does not manifest quality attributes; rather it exhibits product characteristics that imply or contribute to quality attributes. He states that most models fail to deal with the product characteristics. Dromey's model expresses the same internal view as the classic Boehm and McCall models, because he specifically aims at quality of software code (although he claims this can be applied equally well to other artifacts of the development cycle such as requirements specification or user interfaces). He states that the requirements of any product model are that it provides an adaptable methodical system for building quality into software, and it classifies software characteristics and quality defects. He proposes a bottom up/top down approach. In bottom up one begins with a set of structural forms whose quality carrying properties are identified and must be satisfied. These properties impact certain higher leveled quality attributes. This allows a bottom up movement by ensuring particular product properties are satisfied, for example, for programmers. In his top down appraisal, designers begin with the quality attribute and move to the quality carrying property. In this way, for example, designers may identify properties that need to be satisfied for each structural form.

Dromey's model is broad enough to be applied across languages and projects — as long as it evolves as new languages evolve. Dromey provides a substantial contribution to the software quality field in that he has provided a complex analysis, which established direct lines between tangible product characteristics and quality attributes. However, his complete disregard for characteristics that can be evaluated externally could be argued to result in an excellent product that does not do what the customer has requested. The model also needs to be broadened to include the same principles applied to other aspects of the design process such as requirements specification.

The Dromey model has been developed to address many of the problems of the earlier models. Dromey (1995) criticized the earlier quality models for failing to deal adequately with the product characteristic problems and to link the quality factors/attributes to product characteristics. He points out that hierarchical models that use top-down decomposition are usually rather vague in their definitions of lower levels. They thus offer little help to software developers who need to build quality products. To overcome this, he proposed specific quality-carrying properties and systematic classification of quality defects. The quality-carrying properties are:

1. Correctness properties (minimal generic requirements for correctness)
2. Structural properties (low level, intramodule design issues)
3. Modularity properties (high level, intermodule design issues)
4. Descriptive properties (various forms of specification/documentation)

Correctness properties fall broadly into three categories that deal with computability, completeness, and consistency. Structural properties have focused upon the way individual statements and statement components are implemented and the way statements and statement blocks are composed, related to one another, and utilized. The modularity properties address the high-level design issues associated with modules and how they interface with the rest of the system. And the descriptive properties reflect how well the software is described. Explicit comments are added to a program to document how the implementation realizes its desired functionality by manipulating variables with prescribed properties.

Dromey believes that it is impossible to build high-level quality attributes such as reliability or maintainability into products. Rather, software engineers must build components that exhibit a consistent, harmonious and complete set of product properties that result in the manifestations of quality attributes.

Dromey's approach is important because it allows us to verify models. It establishes a criterion for including a particular software property in a model and a means of establishing when the model is incomplete. For example, each quality-carrying property has quality impact, for example, reliability and quality defects. By identifying the defects, the software developer will take measures to prevent the defect from occurring.

Kitchenham and Pfleeger (1996) noted that Dromey's quality model provides a constructive approach to engineering software quality which can be refined and improved further. Dromey even developed tools to assist in the production of quality software, a static analyzer to detect quality defects, and a programming language aimed at preventing developers from building defects into their programs.

The criticism of Dromey's model is that he does not use the metric approach in measuring the software quality. This is because he uses the concept of defect classification. However, as Kitchenham and Pfleeger (1996) pointed out, the model is proof that there are other means to measure quality.

### *Constructive Quality Model (COQUAMO)*

Kitchenham shares Dromey's concern for the need for precision in measurement out of the quality models. She has developed various models in her search to define and measure software quality. COQUAMO (Gulezian, 1995) was developed to further this research. It was the REQUEST (Reliability and Quality for European Software Technology) project which adapted Boehm's COCOMO model to use the more precise measurement of Rome Laboratory Software Quality Framework. (RLSQF provided a framework for using certain metrics for measuring 29 software oriented attributes and relating them to 13 user-oriented quality factors). The beauty of COQUAMO was that Kitchenham saw the need for measurement tools to vary from one environment to another. This was a substantial improvement on all research to date, which had been trying to increase the objective measurements for specific characteristics, without producing any results.

Kitchenham introduced the concept of a "quality profile" and made a distinction between subjective and objective quality measures. It is an attempt to integrate the user, manufacturer, and product views in one software quality model (Kitchenham & Pfleeger, 1996). The COQUAMO model was developed based on Gilb's model and Garvin's views of quality (Gillies, 1992).

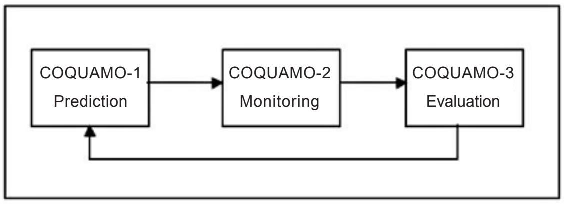
The COQUAMO model consists of the following components:

1. **Transcendent Properties:** these are qualitative measures that are hard to measure. People have different views and definitions. An example is usability.
2. **Quality Factors:** these are system characteristics, which are made up of measurable factors of quality metrics and quality attributes. The quality factors themselves could be subjective or objective characteristics such as reliability and flexibility.
3. **Merit Indices:** these define the system subjectively. They are measured subjectively by quality ratings.

The objectives of COQUAMO and the relevant tools to assist the software developers are in the following table:

- to predict product quality
- to monitor progress toward quality
- to evaluate the results as feedback to improve prediction of the next project

*Figure 1. COQUAMO toolset diagram (Gillies, 1992)*



The initial input to COQUAMO-l are estimates of the following quality drivers:

- product attributes such as quality requirements
- process attributes such as process maturity
- personnel attributes such as the developer's experience and motivation
- project attributes such the quality norm expected
- organizational attributes such as quality management

The toolset of COQUAMO-2 consists of guidelines to monitor the progress toward quality product, while toolset COQUAMO-3 provides guidelines to assess the quality of the product and compare it against the prediction. The result of this will be input to COQUAMO-1 for the next project.

The distinctive characteristic of this model that differentiates it from previous models is that the evaluation and feedback processes are introduced here to improve the quality of software in the next project. There is a similarity with some aspects of Total Quality Management (TQM) in software quality modeling, that is, improvement of quality. This is featured by the COQUAMO-3 toolset.

## *The SQUID Product Model*

The SQUID model, an ESPRIT 3 SQUID project (Kitchenham et al., 1997), was the result of a reassessment by Kitchenham of what constituted a quality model. It was developed after COQUAMO and after the results of the ESPRIT project concluded that there were no software product metrics that were likely to be good predictors of final product qualities. Kitchenham built SQUID based on these results and held firmly to the importance of monitoring and controlling internal measures.

Based on the McCall and ISO9126 models, SQUID aimed to use measurable properties to link internal properties and external characteristics to quality characteristics. A key principle to the SQUID model is that it must be customized for each project/product. Kitchenham states that the underlying principle of SQUID is that software quality requirements cannot be considered independently from a particular software product. Kitchenham acknowledges her alignment with Gilb (1996) on this point when he states that quality requirements arise from the operational properties required of the software, or of its support/maintenance environment. SQUID provides a concrete link between a concrete feature/component and the quality factor. The SQUID approach is to use measurement as the link between the product quality requirements and the quality characteristics (it uses ISO9126 characteristics as accepted results of research). In the application of the model to a product, concrete internal measures, with targets, are developed for the desired software properties, which reference quality characteristics. For example, the internal measures for the software property correctness were the number of faults detected, and the grouping into which the faults fell.

The SQUID model comes far closer than any in being able to balance the need for internal and external views and being able to consider the change in characteristics and metrics that are involved with each type of software in each context. The model's necessary broadness to suit this recognized need means that it takes a different form from the other models. There is no naming of quality factors and quality characteristics. The true strength of this model, as already stated, is its ability to be used in any project.

## *Other Approaches to Quality Evaluation*

Quality Function Deployment (QFD) is a rigorous and high-structured process used in manufacturing. It provides a means of translating customer requirements into the appropriate technical requirements for each stage of product development and production. The process has been adopted in software engineering (Delen & Rijsenbrij, 1992; Erikkson & McFadden, 1993; Juliff, 1994; Thompsett, 1993). It required that before each software development project began, the appropriate characteristics along with the degree of importance of these characteristics had to be first identified by the stakeholders. The project quality would then be measured against these functions. Though QFD is very much a product definition process which helps businesses identify and integrate user needs and requirements to a product, Thompsett (1993) states that QFD is a tool which can help define software quality. Erikkson and McFadden (1993) claim that this approach gives a good structure within which users' needs are taken into account and progressively refined to give appropriate product metrics. QFD has been described as a customer-driven system that attempts to get early coupling between the requirements of the customer and system designers. It has not only proven itself in manufacturing and service environments, but has been shown to be equally beneficial in software development (Erikkson & McFadden, 1993; Juliff, 1994; Thompsett, 1993). Erikkson & McFadden (1993) describe several benefits of QFD from their case study:

- QFD encourages focus on customer needs and helps to prioritize the requirements.
- QFD encourages developers to think in terms of defect prevention.
- QFD is a powerful communication vehicle that allows modeling of the discussions between customers, software designers, and other participants.
- QFD allows easy tracking of important customer requirements to the related software characteristics, product features, and product metrics.
- QFD provides an opportunity to follow the consequences in the process.
- As can be seen from this list, QFD is a powerful manufacturing process to ensure customer requirements are met. QFD differs from traditional models, in that its approach is to develop software products from the customers' perspective, not from the suppliers' perspective. While QFD supports the many views of quality, described by Garvin (1984), it does not try to understand the differences, nor does it try to identify similarities between customers.

Vidgen et al. (1994) proposed a framework to define quality based on the multiview development method (Wood, 1992; Wood-Harper & Avison, 1992). They believed that multiple perspectives of software quality are required if one is to assess product quality properly. The framework was based on customer satisfaction, relating the product with its use and the services provided to support it. The three views of IS quality provide the basis for the multiple perspective. It was not merely an exercise in looking at the same object from different angles, but it entailed different assumptions about what quality is. Though this framework was introduced, no further research followed.

### *Criticism of These Recent Models*

Kitchenham acknowledges some problems with the SQUID model (Kitchenham et al., 1997). Using the ISO9126 model as a framework led her to make recommendations for on-going improvements to the ISO9126 standard. Among her recommendations is a request for a further broadening of the ISO9126 standard. Her reasons are good ones and may lead critics of the broadness of ISO9126 to rethink their reasons for seeking further specificity. She says that insofar as software products do not have a standard set of functional requirements, then it seems that rather than give a fixed set of quality requirements, the standards should specify how the quality characteristics should be defined by those involved in the project. From this generic outline, the development team would have guidance to develop their product model. Kitchenham also states that the lower level properties should be clearly classified as external or internal characteristics. She finally recommends that research should focus on ways to validate quality models. She acknowledges the contribution Dromey's (1995) research has made in providing a criterion for including a software property in a model. Dromey supports this need for a broad model to be developed by those involved in the project and states that it would involve the project team specifying the subcharacteristics of each quality characteristic of each component, establishing the metrics that correlate to the characteristics of the software product and to the environment in which it is functioning, and to provide a rating level that expresses a degree of user satisfaction. Since each set of users and developers needs is different, each rating level will change. This is a complex task that would require much guidance.

Vollman (1993) provides additional comment on this issue of model broadness: not only is the subdivision of quality characteristics influenced by the evaluator's point of view and the product's context, but when this occurs, the algorithms applied when evaluating will need to differ. In 1993, the ISO9000 standards included a publication that was the precursor to this need for the measure to be applied by multiple evaluators in multiple situations, and stated that an organization should use whatever metrics it deems appropriate to its applications as long as the ISO9000 metrics methodology is followed. Schneidewind (1993) states that this broadness allows metrics to be used by developers, managers, quality assurance organizations, maintainers, and users. Clearly the newest metric for software quality measurement will receive much heated discussion as a result of the move to balance internal measures with measures of external product attributes.

From this section, it can be concluded that there have been vast improvements in defining software quality. Yet overall, software product quality research is still relatively immature, and today it is still difficult for a user to compare software quality across products. Researchers are still not clear as to what is a good measure of software quality because of the variety of interpretations of the meaning of quality, of the meanings of terms to describe its aspects, of criteria for including or excluding aspects in a model of software, and of the degree to which software development procedures should be included in the definition (Comerford, 1993). Although many of the criticisms of product models are valid, Voas (1997) proposes that because product-oriented methods are more focused on achieving quality than assessing it, they play a critical role in software quality research. He contends firmly that software behavior, irrespective of the software's development history, is what defines software quality. He sees organizations and research caught in a bind with software testing measuring the right thing but with insufficient precision and process measurement more accurately measuring the wrong thing.

## Measuring Quality with the Software Evaluation Framework

There have been many studies on the topic of software quality, yet little empirical studies on what influences the perception of quality for different people. Earlier research of Wong (Wong, 1998; Wong & Jeffery, 1995, 1996) concluded that different groups of people view quality in different ways and that it was possible to group people with similar definitions of quality and similar choices of characteristics in their quality assessment process.

During the past 30 years there have been many studies on the topic of software quality; however, there have been none on a framework for software quality, which considers the motivation behind the evaluation process, other than the earlier version of this framework introduced by Wong and Jeffery (2001). This framework is based on the notion that software evaluators are influenced by their job roles. This is supported by earlier studies (Wong, 1998; Wong & Jeffery, 1995, 1996) where stakeholders with different job roles were found to focus on different sets of software characteristics when evaluating software quality. What motivates these differences is found within the broader context of value, where studies have shown that values are a powerful force in influencing the behavior of individuals (Rokeach, 1968; Yankelovich, 1981).

The theoretical basis for developing such a framework was based on the theory found in cognitive psychology and adopted by Gutman's Means-End Chain Model (Bagozzi, 1997; Bagozzi & Dabholkar, 2000; Gutman, 1982, 1997; Valette-Florence, 1998), which posits that linkages between product characteristics, consequences produced through usage, and personal values of users underlie the decision-making process or, in this case, the software quality evaluation process. It is the aim of the framework to not only show the relationships between the characteristics and software quality, but also show that there are relationships between the characteristics and the desired consequences, and between the characteristics and the sought-after values.

Personal values are beliefs people have about important aspects of themselves and the goals toward which they are striving. Personal values are the penultimate consequences of behavior for people: their feelings of self-esteem, belonging, security, or other value orientations. As such, personal values are part of the central core of a person. That is, personal values determine which consequences are desired and which are undesired.

Values have been shown to be a powerful force in influencing the behaviors of individuals in all aspects of their lives (Rokeach, 1968; Yankelovich, 1981). It is proposed in this research, that their use in software quality evaluation shows the behavior of software evaluators and the relationship between the software characteristics, the desired consequences, and the values sought. Several attempts in consumer research have been made to provide a theoretical and conceptual structure connecting consumers' values to their behavior (Gutman, 1982; Howard, 1977; Vinson et al., 1977; Young & Feigin, 1975). The basis by which the study is performed is via adopting Gutman's means-end chain model (Bagozzi, 1997; Bagozzi & Dabholkar, 2000; Gutman, 1982, 1997; Valette-Florence 1998), which posits that linkages between product characteristics, consequences produced through usage, and personal values of users underlie the decision-making process or, in our case, the software quality evaluation process. The term *means* refers to the software product and services, and *ends* refers to the personal values important to the user.

Gutman's model is able to give a complete representation of the means-end chain, representing linkages from the characteristics to the values, along with the capability of explicating the chain. Earlier models (Howard, 1977; Vinson et al., 1977; Young & Feigin, 1975) failed to consider the means-end chain. They focused only on the consequences and values without considering how they relate to the characteristics or attributes. The means-end chain model seeks to explain how a person's choice of a product or service enables him or her to achieve his or her desired result. Such a framework consists of elements that represent the major usage processes that link personal values to behavior. Two assumptions underlie this model:

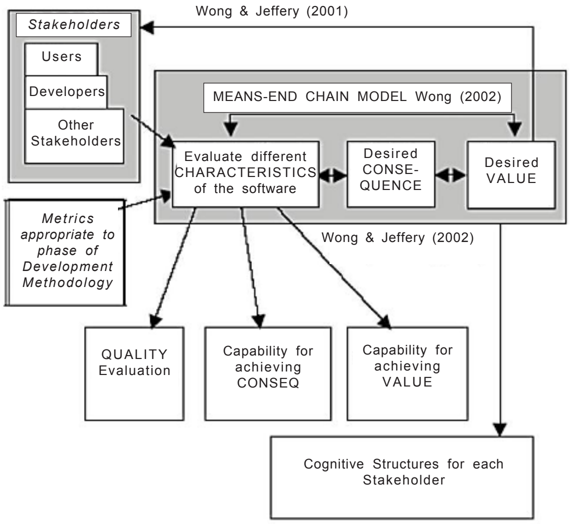
- all user actions have consequences; and
- all users learn to associate particular consequences from the usage of the product or service.

Consequences may be desirable or undesirable; they may stem directly from usage or the act of usage, or occur indirectly at a later point in time or from others' reaction to one's consumption behavior. The central aspect of the model is that users choose actions that produce desired consequences and minimize undesired consequences. Of course, because it is the characteristics that produce consequences, consideration for the characteristics that the software product possesses must be made. Therefore, it is important to make aware the characteristic-consequence relations. Overall, the characteristic-consequence-value interrelations are the focus of the model. Values provide the overall direction, consequences determine the selection of specific behavior in specific situations, and the characteristics are what is in the actual software product that produces the consequences. It is knowledge of this structure that permits us to examine the underlying meaning of quality.

As highlighted in the literature, the benefit of utilizing Gutman's model in the framework is that it shows how the desired values influence the behaviors of individuals in all aspects of their lives (Gutman, 1982; Rokeach, 1968; Yankelovich, 1981). Gutman's model suggests that the desired consequences and the values sought are the motivators behind the choice of characteristic for software evaluation. In addition to this, the framework also highlights the significance of this relationship through the relationships between characteristics and consequences and also between the characteristics and value. It is through these relationships that the possibility of using the characteristics to evaluate each consequence and value becomes apparent.

The framework shown in Figure 2 is based on Gutman's Means-End Chain Model. As can be seen in this diagram, the framework consists of a number of boxes describing the three elements of Gutman's model, the stakeholders who evaluate the software quality, the outcome for the quality evaluation, and the

*Figure 2. SEF: Software evaluation framework*


Wong & Jeffery (2001)
Stakeholders
Users
Developers
Other Stakeholders
MEANS-END CHAIN MODEL Wong (2002)
Evaluate different CHARACTERISTICS of the software
Desired CONSE-QUENCE
Desired VALUE
Metrics appropriate to phase of Development Methodology
Wong & Jeffery (2002)
QUALITY Evaluation
Capability for achieving CONSEQ
Capability for achieving VALUE
Cognitive Structures for each Stakeholder

arrows linking these elements, while also describing the direction of the influence. The Means-End Chain Model has been placed in the main box, as it is proposed, in this framework, to be the central influence for the choice of characteristics used in software evaluation, and the influence for the differences found between stakeholders. The framework not only shows the related elements, which influence software quality evaluation, but also introduces a way of describing the relationships between the characteristics, the consequences, and the values. These diagrams are called cognitive structures. Cognitive structures can be drawn for each stakeholder, allowing stakeholder differences to be described through a diagram (Wong, 2002, 2003a; Wong & Jeffery, 2001).

The recent studies of Wong and Jeffery (Wong, 2002, 2002a; Wong & Jeffery, 2001, 2002) provide the premise to this framework. An exploratory study by Wong and Jeffery (2001), utilized a qualitative approach to explore the influence of value on the choice of characteristics, and to determine whether cognitive structures could be used as a tool to represent the links between the characteristics, consequences, and values. The results of the study not only gave strong support for the influence of value on the choice of characteristic in software evaluation, but also supported earlier pilot studies on stakeholder differences (Wong, 1998; Wong & Jeffery, 1995, 1996), identifying different cognitive structures for users and developers.

A more recent paper by Wong (2002a) reported on a large quantitative study, which tested the appropriateness of utilizing Gutman's Means-End Chain Model (Bagozzi, 1997; Bagozzi & Dabholkar, 2000; Gutman, 1982, 1997; Valette-Florence, 1998) in software evaluation. Unlike previous studies of Gutman's Model, this study showed strong support for the Means-End Chain Model in a software evaluation setting. The results showed strong correlations between the characteristics, the consequences and the values, and supported the results of the qualitative study (Wong & Jeffery, 2001) described earlier.

The Software Evaluation Framework has also been applied to the requirements phase of the software development project (Wong, 2003a, 2003b, 2004). The results showed support for the use of the framework. The results showed that the measurements between the different phases are not the same, though the motivation behind the choice of these measurements is the same for a stakeholder group. The study also finds that the two groups of stakeholders are very similar in the measurements they choose for evaluating requirements documents; however, the motivation behind their choice of these measurements differs between the stakeholder groups. These results are a contrast to that of the implementation phase. More recently, studies have also been conducted on the metrics, which could be used to measure the characteristics of the software product (Wong, 2003a, 2003c, 2004, 2004a).

# Conclusion

An exhaustive and critical review of the research concerned specifically with software product quality is reviewed in this chapter. It is clear that some models have provided more progress toward refining the definition of software quality while others have contributed to a better understanding of the process for measuring software quality. While the attempts of many of the models have been admirable, the development of the Software Evaluation Framework has succeeded in introducing a framework which fills the gaps found in many of the models.

Several broad conclusions can be drawn from the literature.

First, it is understood that there are different views of quality and that an understanding of these views is required; however, it is difficult to adopt all views at the same time when evaluating software. Though a number of assertions have been made in the literature regarding the multiple views of quality, there have not been many studies to support this, nor are there any frameworks or models which give the rationale for the different views.

Second, we know little about what rationale is used for selecting a characteristic for software evaluation. There is a need to consider what motivates the characteristics used in software evaluation for EACH different view.

Third, the Software Evaluation Framework, introduced in this chapter, showed support for applying Gutman's Means-End Chain Model in software quality evaluation. The results endorse the links between characteristics, consequences and values, which have been regarded as rich constructs for understanding software evaluator behavior (Izard, 1972; Rokeach, 1973; Tolman, 1951). The results of the study provided further evidence of a connection between patterns of characteristics, consequences, and values and gives valuable support for the use of cognitive structures as a means of describing this relationship. Knowing this relationship is valuable for marketing and promoting the software, which is important for software acceptance. Software developers have often been challenged with conducting appropriate software demonstrations. For many developers, there is a lack of understanding of what is important to the user. The Software Evaluation Framework introduces the motivation for user acceptance of the software and allows the developer to have a better understanding of the desired consequences and how they are related to the software. The framework is valuable for understanding the differences in views of quality among the many stakeholders in a software development project. The cognitive structures introduced a way of describing these differences, showing clearly the characteristics of the software product, which are significant in their evaluation of software quality, along with the desired consequences and sought-after values associated with these characteristics.

Fourth, the use of the cognitive structures, introduced in the Software Evaluation Framework, can better the understanding of the characteristics selected for quality evaluation and how they are related to what the evaluators seek as desired consequences from the use of the software. The cognitive structures not only showed that users and developers differ in their choice of characteristics, but show differences in the consequences sought and the values desired. Software Quality has been described as a combination of characteristics, with ISO9126 being adopted as the current international standard. However, though support for ISO9126 appeared in some of the results, evidence suggests that non-ISO9126 characteristics are also important when evaluating software quality. This is supported by the cognitive structures and the results of earlier studies of Wong and Jeffery (Wong, 1998; Wong & Jeffery, 1995, 1996). The results found that ISO9126 characteristics, usability, and functionality strongly affect software quality for both developers and users, while technical characteristics, like portability and maintainability, were only significant for the developers. However, the results surprisingly found operational characteristics, like efficiency and reliability, to minimally affect software quality. This result was not expected since many of the measurements for software quality, like defects and failures, focus on reliability. As to the non-ISO9126 characteristics, support was found to be important for both users and developers, while the economic and institutional characteristics were only relevant for users. It is evident that further work is required to identify whether the non-ISO9126 characteristics should be part of the ISO9126 set of characteristics. The use of the cognitive structures can also help to identify the problems that may be occurring in current evaluation practices. Software Quality is seldom confined to just one characteristic, though very often, reliability seems to be the only focus. Defects and failures have often been the only metrics collected. While reliability, an operational characteristic, is important, the results of the research show that other metrics need to be collected if a more generally applicable set of measure for Software Quality is desired for different stakeholders.

## References

Bagozzi, R. (1997, September 1997). Goal-directed behaviors in marketing: Cognitive and emotional. *Psychology & Marketing, 14*, 539-543.

Bagozzi, R., & Dabholkar, P. (2000, July). Discursive psychology: An alternative conceptual foundation to means-end chain theory. *Psychology & Marketing, 17*, 535-586.

Bockenhoff, E., & Hamm, U. (1983). Perspektiven des Marktes für Alternativ erzeugte Nahrungsmittel. *Berichte über Landwirtschaft, 61*, 341-381. (Cited in Steenkamp, 1989).

Boehm, G. (1963 April). Reliability engineering. *Fortune*, pp. 124-127.

Boehm, B., Brown, J., & Lipow, M. (1976). Quantitative evaluation of chance of receiving desired consequences. *Proceedings of the Second International Conference on Software Engineering* (pp. 592-605).

Box, J. (1984). Product quality assessment by consumers – the role of product information. *Proceedings of the XIth International Research Seminar in Marketing*, Aix-en-Provence (pp. 176-197).

Braa, K., & Ogrim, L. (1994). Critical view of the application of the ISO standard for quality assurance. *Journal of Information Systems, 5*, 253-269.

Carpenter, S., & Murine, G. (1984, May). Measuring software product quality. *Quality Progress,* 16-20.

Cavano, J., & McCall, J. (1978, November). A framework for the measurement of chance of receiving desired consequences. *Proceedings of the ACM SQA Workshop* (pp. 133-139).

Checkland, P., & Scholes, J. (1990). *Soft systems methodology in action.* Chichester, UK: Wiley.

Coallier, F. (1994, January). *How ISO9001 fits into the software world.* IEEE Software.

Comerford, R. (1993, January). Software. *IEEE Spectrum*, pp. 30-33.

Crosby, P. (1978). *Quality is free.* Maidenhead: McGraw-Hill.

Dahlborn, B., & Mathiassen, L. (1993). *Computers in context. The philosophy and practice of systems design.* Cambridge, MA: NCC Blackwell.

Delen, G., & Rijsenbrij, D. (1992). The specification, engineering, and measurement of information systems quality. *Journal of Systems and Software, 17*(3), 205-217.

Dowson, M. (1993). Software process themes and issues. *Proceedings of the 2nd International Conference on the Software Process: Continuous Software Process Improvement* (pp. 54-60).

Dromey, R. (1995, February). A model for software product quality. *IEEE Transactions on Software Engineering, 21*(2), 146-162.

Erikkson, I., & McFadden, F. (1993). Quality function deployment: A tool to improve software quality. *Information and Software Technology, 35*(9), 491-498.

Feldman, M.S., & March, J.G. (1981). Information in organizations as signal and symbol. *Administrative Science Quarterly, 26*, 171-186.

Fenton, N. (1996, January). Do standards improve quality: A counterpoint. *IEEE Software*, *13*(1), 23-24.

Garvin, D. (1984). What does "product quality" really mean? *Sloan Management Review, 24*.

Genth, M. (1981). Qualität und Automobile – Eine Untersuchung am Beispiel des deutschen Automobilmarktes 1974-1977. Frankfurt: Lang, in Steenkamp Product Quality. (Cited in Steenkamp, 1989).

Gilb, T. (1996, January). Level 6: Why we can't get there from here. *IEEE Software*, 97-103.

Gillies, A. (1992). *Software quality, theory and management* (1st ed.). London: Chapman & Hall.

Greenbaum, J., & Kyng, M. (1991). *Design at work: Cooperative design of computer systems.* NJ: Lawrence Erlbaum.

Gulezian, R. (1995). Software quality measurement and modeling, maturity, control and improvement. *IEEE COMPCON 1995 Proceedings* (pp. 52-60).

Gutman, J. (1982). A means-end chain model based on consumer categorization processes. *Journal of Marketing, 46*(Spring), 60-72.

Gutman, J. (1997). Means-end chains as goal hierarchies. *Psychology & Marketing, 14*(6), 545-560.

Hatton, L. (1993). *The automation of software process and product quality, chance of receiving desired consequences Management I.* Computational Mechanics Publications.

Holbrook, M.B., & Corfman, K.P. (1983). Quality and other types of value in the consumption experience: Phaedrus rides again. In Jacoby & Olson (Eds.), *Perceived quality* (pp. 31-57). Lexington.

Howard, J.A. (1977). *Consumer behaviour: Application of theory.* New York: McGraw-Hill.

Humphrey, W. (1988, March). Characterising the software process: A maturity framework. *IEEE Software, 5*(2), 73-79.

Hyatt, L., & Rosenberg, L. (1996, April). A software quality model and metrics for identifying project risks and assessing software quality. *Proceedings of the 8th Annual Software Technology Conference*, Utah.

ISO. (1986). ISO8402 Quality-Vocabulary, International Organization for Standardization, Geneva.

Izard, C. (1972). *Human emotions.* New York: Plenum Press.

Juliff, P. (1994). *Chance of receiving desired consequences Function Deployment, Chance of receiving desired consequences Management II Vol 1.* Computational Mechanics Publications.

Juran, J.M., & Gryna, F.M. (1980). *Quality planning and analysis.* New York: McGraw-Hill.

Kawlath, A. (1969). *Theoretische Grundlagen der Qualitätspolitik.* Wiesbaden, Germany: Gabler GmbH, in Steenkamp Product Quality. (Cited in Steenkamp, 1989).

Kitchenham, B. (1987, July). Towards a constructive quality model. Part 1: Chance of receiving desired consequences modeling, measurement and prediction. *Software Engineering Journal*.

Kitchenham, B., Linkman, S., Pasquini, A., & Nanni, V. (1997). The SQUID approach to defining a quality model. *Software Quality Journal, 6*, 211-233.

Kitchenham, B., & Pfleeger, S. (1996, January). *Software quality: The elusive target.* IEEE Software.

Kitchenham, B., & Pickard, L. (1987, July). Towards a constructive quality model. Part 2: Statistical techniques for modeling chance of receiving desired consequences in the ESPRIT REQUEST project. *Software Engineering Journal*.

Kitchenham, B., & Walker, J. (1986, September). *The meaning of quality.* Software Engineering 86: Proceedings of BCS-IEE Software Engineering 86 Conference, Southampton, England.

Kotler, P. (1984). *Marketing management: Analysis, planning and control* (5th ed.). Englewood Cliffs, NJ: Prentice Hall.

Kuehn, A., & Day, R. (1962). Strategy of product quality. *Harvard Business Review, 40*, 100-110.

Kupsch, P., & Hufschmied, P. (1978). Die Struktur von Qualitätsurteilen und das Informationverhalten von Konsumenten beim Kauf langlebiger Gebrauchsgüter. Oplagen: Westdeutscher Verlag. (Cited in Steenkamp, 1989).

Maynes, E.S. (1976b). *Decision-making for consumers.* New York: MacMillan Publishing.

McCall, J., Richards, P., & Walters, G. (1977, November). Factors in chance of receiving desired consequences, Vol 1, 2, & 3.

Monroe, K., & Krishnan, R. (1985). The effect of price and subjective product evaluations. In J. Jacoby & J.C. Olson (Eds.), *Perceived quality* (pp. 209-232). Lexington: Lexington Books.

Myers, W. (1993, March). Debating the many ways to achieve quality. *IEEE Software*.

OED. (1990). *Oxford English Dictionary*.

Olson, J.C., & Jacoby, J. (1972). Cue utilization in the quality perception process. *Proceedings of the Third Annual Conference of the Association for Consumer Research*, Iowa City, Iowa (pp. 167-179).

Ould, M. (1992). Chance of receiving desired consequences. Improvement through Process Assessment - A view from the UK. *Proceedings of the IEEE Colloquium on Chance of receiving desired consequences*.

Oxenfeldt, A.R. (1950). Consumer knowledge: Its measurement and extent. *Review of Economics and Statistics, 32*, 300-316.

Paulk, M.C. (1991). Capability maturity model for software (Report CMU/SEI-91-TR-24). SEI, Carnegie Mellon University.

Paulk, M.C., Curtis, B., Chrissis, M.B., & Weber, C.V. (1993, July). The capability maturity model, Version 1.1. *IEEE Software, 10*(4), 18-27.

Pfleeger, S.L. (2001). *Software engineering: Theory and practice* (2nd ed.). Prentice Hall.

Pfleeger, S.L., Jeffery, R., Curtis, B., & Kitchenham, B. (1997, March/April). Status report on software measurement. *IEEE Software*, 34-43.

Rae, A., Robert, P., & Hausen, H.L. (1995). *Software evaluation for certification. Principles, practice and legal liability.* UK: McGraw-Hill.

Rokeach, M. (1968). *Beliefs, attitudes and values*. San Francisco: Jossey-Bass.

Rokeach, M. (1973). *The nature of human values*. New York: Free Press.

Rombach, D., & Basili, V. (1990). Practical benefits of goal-oriented measurement. *Proceedings of the Annual Workshop of the Centre for Software Reliability: Reliability and Measurement*, Garmisch-Partenkirchen, Germany.

Rozman, I., Horvat, R., Gyorkos, J., & Hericko, M. (1977). PROCESSUS - Integration of SEI CMM and ISO quality models. *Software Quality Journal, 6*, 37-63.

Schneidewind, N.F. (1993, April). New software quality metrics methodology standards fill measurement needs. *IEEE Computer*, 105-196.

Schneidewind, N.F. (1996, January). Do standards improve quality: A point. *IEEE Software*, 13(1), 22-24.

Stolterman, E. (1991). Designarbetets dolda rationalitet. PhD thesis, Research Report No. 14:91. Information Processing and Computer Science, Institutionen för Informationsbehandling, Administrativ Databehandling University of Umeå. (Cited in Braa et al., 1994).

Sunazuka, T., Azuma, M., & Yamagishi, N. (1985). Chance of receiving desired consequences. Assessment technology. *Proceedings of the IEEE 8th International Conference on Sofware Engineering*.

Thompsett, R. (1993). Quality agreements for quality systems. *Proceedings of Australian Computer Society*, Victorian Branch, Annual Conference.

Thurstone, W.R. (1985). Quality is between the customer's ears. *Across the Board*, pp. 29-32.

Tolman, E. (1951). A psychological model. In T. Parsons & E. Shils (Eds.), *Towards a general theory of reasoned action* (pp. 148-163). Cambridge: Harvard University Press.

Trenkle, K. (1983). Lebensmittelqualität und Verbraucherschutz. *AID-verbrauchersdienst,* pp. 211-216. (Cited in Steenkamp, 1989).

Valette-Florence, P. (1998, June). A causal analysis of means-end hierarchies in a cross-cultural context: Metholodogical refinements. *Journal of Business Research, 42*(2), 161-166.

Vidgen, R., Wood, J., & Wood-Harper, A. (1994). *Customer satisfaction: The need for multiple perspectives of information system quality. Chance of receiving desired consequences Management II Vol 1*. Computaional Mechanics Publications.

Vinson, D.E., Scott, J.E., & Lamont, L.M. (1977, April). The role of personal values in marketing and consumer behaviour. *Journal of Marketing, 41*, 44-50.

Voas, J.M. (1997, July/August). Can clean pipes produce dirty water? *IEEE Software*, pp. 93-95.

Vollman, T.E. (1993, June). Software quality assessment and standards. *Computer, 6*(6), 118-120.

Wimmer, F. (1975). Das Qualitatsurteil des Konsumenten: Theoretische Grundlagen und Empirische Ergebnisse. Frankfurt: Lang. (Cited in Steenkamp, 1989).

Wittgenstein, L. (1953). *Philosophical investigations*. New York: MacMillan.

Wolff, M.F. (1986). Quality/process control: What R and D can do. *Research Management*, pp. 9-11.

Wong, B. (1998). Factors influencing software quality judgment (Tech. Rep.). CSIRO.

Wong, B. (2002, May 25). *Comprehending software quality: The role of cognitive structures.* Proceedings of the International Workshop on Software Quality (in conjunction with ICSE 2002).

Wong, B. (2002a). *The appropriateness of Gutman's means-end chain model in software evaluation.* Proceedings of the 2002 International Symposium on Empirical Software Engineering (ISESE 2002).

Wong, B. (2003). *Measurements used in software quality evaluation.* Proceedings of the 2003 International Conference on Software Engineering Research and Practice (SERP 2003).

Wong, B. (2003a). *Applying the software evaluation framework "SEF" to the software development life cycle*. Proceedings of the 2003 International Symposium on Empirical Software Engineering (ISESE 2003).

Wong, B. (2003b). *Measuring the quality of the requirements specification document for an e-commerce project.* Proceedings of the 2003 International Business Information Management Conference (IBIM 2003).

Wong, B. (2003c). *A study of the metrics applied to the software evaluation framework "SEF"*. Proceedings of the Third International Conference on Quality Software (QSIC 2003).

Wong, B. (2004). *A study of the metrics for measuring the quality of the requirements specification document.* Proceedings of the 2004 International Conference on Software Engineering Research and Practice (SERP 2004).

Wong, B. (2004a, December). The software evaluation framework "SEF" extended. *Information and Software Technology Journal*, *46*(15), 1037-1047.

Wong, B., & Jeffery, R. (1995, November 22-24). *Quality metrics: ISO9126 and stakeholder perceptions.* Proceedings of the Second Australian Conference on Software Metrics (ACOSM'95), Sydney (pp. 54-65).

Wong, B., & Jeffery, R. (1996). A pilot study of stakeholder perceptions of quality (Tech. Rep.), CSIRO.

Wong, B., & Jeffery, R. (2001). *Cognitive structures of software evaluation: A means-end chain analysis of quality*. Proceedings of the Third International Conference on Product Focused Software Process Improvement (PROFES 2001).

Wong, B., & Jeffery, R. (2002). *A framework on software quality.* Proceedings of the Fourth International Conference on Product Focused Software Process Improvement (PROFES 2002).

Wood, J.R. (1992). Linking soft systems methodology (SSM) and information systems (IS). *Systemist - Information Systems Special Edition, 14*(3), 133-135.

Wood-Harper, A.T., & Avison, D. (1992). Reflections from the experience of using multiview: Through the lens of soft systems methodology. *Systemist, 14*(3), 136-145.

Yankelovich, D. (1981, April). New rules in American life: Search for self-fulfilment in a world turned upside down. *Psychology Today,* pp. 60.

Young, D.E., & Feigin, E. (1975, July). Using the benefit chain for improved strategy formulation. *Journal of Marketing*, 39, 72-74.

# Section II: Quality in the Early Stages of IS Delivery

## Chapter IV

# Making Real Progress with the Requirements Defects Problem

R. Geoff Dromey, Griffith University, Australia

## Abstract

*Requirements defects remain a significant problem in the development of all software intensive systems including information systems. Progress with this fundamental problem is possible once we recognize that individual functional requirements represent fragments of behavior, while a design that* satisfies *a set of functional requirements represents integrated behavior. This perspective admits the prospect of constructing a design* out of *its requirements. A formal representation for individual functional requirements, called* behavior trees *makes this possible. Behavior trees of individual functional requirements may be composed, one at a time, to create an integrated design behavior tree (DBT). Different classes of defects are detected at each stage of the development process. Defects may be found at the translation to behavior trees, and then at the integration of behavior trees and when individual component behavior trees are projected from the DBT. Other defects may be found by inspection and model-checking of the DBT.*

# Introduction

There are seven well-known potential problems (Davis, 1988) with the functional requirements and their use in the development of modern software intensive systems:

- they may be incomplete, inconsistent, and/or contain redundancy
- they may not accurately convey the intent of the stakeholders
- in transitioning from the original requirements to the design, the original intention might not be accurately preserved
- over the course of the development of the system, the requirements may change
- the system the requirements imply may not be adequate to meet the needs of the intended application domain
- the number and complexity of the set of requirements may tax people's short-term memory beyond it limits
- the alignment between the requirements for a system, its design, and the implementation may be not preserved

Confronted with these challenges, existing methods for requirements analysis, inspection, representation, and then design are often not up to the task — we end with multiple partial views of a system that have a degree of overlap that makes it difficult to see/detect many types of defects, particularly those that involve interactions between requirements (see Booch et al., 1999; Harel, 1987; Schlaer & Mellor, 1992).

Given all this, is there a more practical way forward? Our chief concerns are:

- to get the complexity of the requirements under control,
- to preserve the intention of the stakeholders, and where there are ambiguities or other problems, clarify the original intention,
- to detect requirements defects as early as possible, and
- to ease the consequences of needing to change requirements as development proceeds and our understanding of the problem at hand improves.

We suggest there is a way to deliver these benefits and consistently make real progress with the requirements problem. It demands that we use the requirements of a system in a very different way. Traditionally the goal of systems development is to build a system that will satisfy the agreed requirements. We suggest this task is too hard, particularly if there is a large and complex set of requirements for a system. The Principle of Totality (Mayall, 1979) gives a clear insight into what the difficulty is. It tells us *"all design requirements are always interrelated and must always be treated as such throughout a design task"* (the degree of relationship can vary from negligible to critical between any pair of requirements). As soon as we have to deal with a large set of requirements, we have to take into account all their potential interactions. This almost always taxes the capacity of our short-term memory beyond its limits because we do not have a practical way of ensuring that all requirements, and all requirements interactions, are properly accounted for as we proceed with each design decision. The problem is further exacerbated by failure to detect defects that result from interactions between requirements.

*A much simpler and easier task is to seek to build a system out of its requirements* (Dromey, 2003). Building a design out of its requirements means that we only need to focus on the localized detail in one requirement at a time. This is cognitively a much more manageable task and less likely to cause our short-term memory to exceed its capacity (e.g., we only have to consider *one requirement at any time* rather than try to consider one hundred or a thousand requirements at a time which could easily be needed for a large system). It also means that requirements interactions are systematically accommodated because each requirement has a precondition associated with it which determines how and where it integrates into a design (c.f. how a given jigsaw puzzle piece slots into the solution to the overall puzzle — except that inserting requirements is easier than placing jigsaw puzzle pieces). If we opt to build a system out of its requirements, it implies:

- we have a representation that will acurately represent the behavior in individual requirements;
- we have a way of combining individual requirements to create a system that will satisfy all requirements.

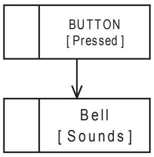
Existing notations like UML (Booch et al., 1999), state-charts (Harel, 1987) and others (Schlaer & Mellor, 1992) do not make it easy to combine individual requirements to create a system design. To explore the design strategy of building a system out of its requirements, it has been necessary to develop and extensively trial on a diverse set of both large and small systems a notation called Behavior Trees. The *behavior tree notation* solves a fundamental problem — it provides a clear, simple, constructive, and systematic path for going from a set of functional requirements to a problem domain representation of design that will satisfy those requirements. Individual requirements are first translated, one at a time, to behavior trees. Behavior trees are then integrated to create a design behavior tree (DBT). From the design behavior tree it is possible to derive the architecture of a system and the behavior designs of individual components. The processes of requirements translation, requirements integration, and the processes of model checking and inspecting the integrated design behavior tree provide a powerful means for finding many different classes of requirements defects.

## Behavior Trees

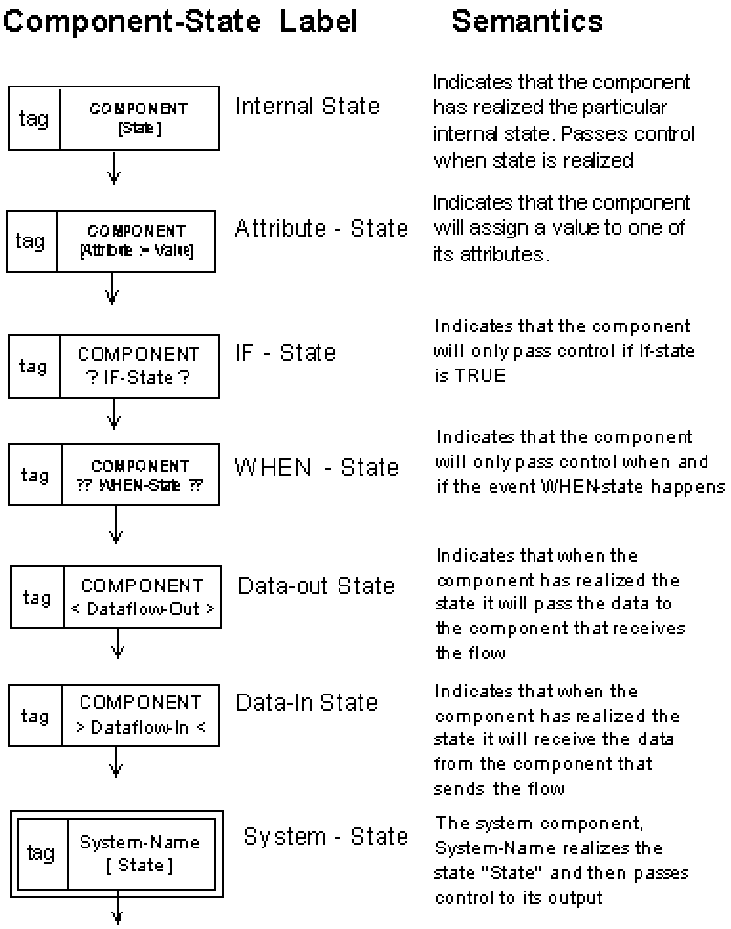
The behavior tree notation captures in a simple tree-like form of composed component-states what usually needs to be expressed in a mix of other notations. Behavior is expressed in terms of components realizing states, augmented by the logic and graphic forms of conventions found in programming languages to support composition. The vital question that needs to be settled, if we are to build a system out of its requirements, is: can the same formal representation of behavior be used for requirements and for a design? Behavior trees make this possible, and as a consequence, clarify the requirements-design relationship.

*Definition: A behavior tree is a formal, tree-like graphical form that represents behavior of individual or networks of entities which realize or change states, make decisions, respond-to/cause events, and interact by exchanging information and/or passing control.*

To support the implementation, of software intensive systems we must capture, first in a formal specification of the requirements, then in the design, and finally in the software the actions, events, decisions, and/or logic, obligations, and constraints expressed in the original natural language requirements for a system. Behavior trees do this. They provide a direct and clearly traceable relationship between what is expressed in the natural language representation and its formal specification. Translation is carried out on a sentence-by-sentence basis, for example, the sentence “when the button is pressed, the bell sounds” is translated to the behavior tree below:



The principal conventions of the notation for component-states are the graphical forms for associating with a component a [State], ??Event??, ?Decision?, <Data_out>, or [Attribute := expression | State ], and reversion "^". Exactly what can be an event, a decision, a state, and so forth is built on the formal foundations of expressions, Boolean expressions and quantifier-free formulae (qff). To assist with traceability to original requirements, a simple convention is followed. Tags (e.g., R1 and R2, etc., see below) are used to refer to the original requirement in the document that is being translated. System states are used to model high-level (abstract) behavior, some preconditions/postconditions and possibly other behavior that has not been associated with particular components. They are represented by rectangles with a double line (===) border.



Indicates that the component has realized the particular internal state. Passes control when state is realized

Indicates that the component will assign a value to one of its attributes.

Indicates that the component will only pass control if If-state is TRUE

Indicates that the component will only pass control when and if the event WHEN-state happens

Indicates that when the component has realized the state it will pass the data to the component that receives the flow

Indicates that when the component has realized the state it will receive the data from the component that sends the flow

The system component, System-Name realizes the state "State" and then passes control to its output

# Genetic Design

Conventional software engineering applies the underlying design strategy of constructing a design that will *satisfy* its set of functional requirements. In contrast to this, a clear advantage of the behavior tree notation is that it allows us to construct a design *out of* its set of functional requirements, by integrating the behavior trees for individual functional requirements (RBTs), one-at-a-time, into an evolving design behavior tree (DBT). This very significantly reduces the complexity of the design process and any subsequent change process. Any design, built out of its requirements, will conform to the weaker criterion of satisfying its set of functional requirements. We call this the *genetic design* process (Dromey, 2005) because of its links in similarity to what happens in genetics. Woolfson (2001), in the introduction of his book, provides a good discussion of this.

What we are suggesting is that a set of functional requirements, represented as behavior trees, in principal at least (when they form a complete and consistent set), contains enough information to allow their composition. This property is the exact same property that a set of pieces for a jigsaw puzzle and a set of genes possess (Woolfson, 2000). The obvious question that follows is: "what information is possessed by a set of functional requirements that might allow their composition or integration?" The answer follows from the observation that the behavior expressed in functional requirements does not "just happen". There is always *a precondition* that must be satisfied in order for the behavior encapsulated in a functional requirement to be accessible or applicable or executable. In practice, this precondition may be embodied in the behavior tree representation of a functional requirement (as a component-state or as a composed set of component states) or it may be missing; the latter situation represents a defect that needs rectification. The point to be made here is that this precondition is needed, in each case, in order to integrate the requirement with at least one other member of the set of functional requirements for a system. (In practice, the root node of a behavior tree *often* embodies the precondition we are seeking.) We call this foundational requirement of the genetic design method the *precondition axiom*.

## Precondition Axiom

Every constructive, implementable individual functional requirement of a system, expressed as a behavior tree, has associated with it a precondition that needs to be satisfied in order for the behavior encapsulated in the functional requirement to be applicable.

A second building block is needed to facilitate the composition of functional requirements expressed as behavior trees. Jigsaw puzzles, together with the precondition axiom, give us the clues as to what additional information is needed to achieve integration. With a jigsaw puzzle, what is key, is not the order in which we put the pieces together, but rather the *position* where we put each piece. If we are to integrate behavior trees in any order, one at a time, an analogous requirement is needed. We have already said that a functional requirement's precondition needs to be satisfied in order for its behavior to be applicable. It follows that some *other* requirement, as part of its behavior tree, must establish the precondition. This requirement for composing/integrating functional requirements expressed as behavior trees is more formally expressed by the following axiom.

## Interaction Axiom

*For each individual functional requirement of a system, expressed as a behavior tree, the precondition it needs to have satisfied in order to exhibit its encapsulated behavior, must be established by the behavior tree of at least one other functional requirement that belongs to the set of functional requirements of the system.* (The functional requirement that forms the root of the design behavior tree, is excluded from this requirement. The external environment makes its precondition applicable.)

The precondition axiom and the interaction axiom play a central role in defining the relationship between a set of functional requirements for a system and the corresponding design. What they tell us is that the first stage of the design process, in the problem domain, can proceed by first translating each individual natural language representation of a functional requirement into one or more behavior trees. We may then proceed to integrate those behavior trees just as we would with a set of jigsaw puzzle pieces. What we find when we pursue this whole approach to software design is that the process can be reduced to the following four overarching steps:

- Requirements translation (problem domain)
- Requirements integration (problem domain)
- Component architecture transformation (solution domain)
- Component behavior projection (solution domain)

Each overarching step needs to be augmented with a verification and refinement step designed specifically to isolate and correct the class of defects that show

*Table 1. Functional requirements for Microwave Oven System*

**R1**. There is a single control button available for the use of the oven. If the oven is idle with the door closed and you push the button, the oven will start cooking (that is, energize the power-tube for one minute).
**R2**. If the button is pushed while the oven is cooking, it will cause the oven to cook for an extra minute.
**R3**. Pushing the button when the door is open has no effect (because it is disabled).
**R4**. Whenever the oven is cooking or the door is open, the light in the oven will be on.
**R5**. Opening the door stops the cooking.
**R6**. Closing the door turns off the light. This is the normal idle state, prior to cooking when the user has placed food in the oven.
**R7**. If the oven times out, the light and the power-tube are turned off and then a beeper emits a sound to indicate that the cooking has finished.

up in the different work products generated by the process. A much more detailed account of this method is described in Dromey (2003).

To maximize communication our intent here is to only introduce the main ideas of the method relevant to requirements defect detection and do so in a relatively informal way. The whole process is best understood in the first instance by observing its application to a simple example. For our purposes, we will use requirements for a Microwave Oven System taken from the literature. The seven stated functional requirements for the Microwave Oven problem (Schlaer & Mellor, 1992, p. 36) are given in Table 1. Schlaer and Mellor have applied their state-based object-oriented analysis method to this set of functional requirements.

## Requirements Translation

Requirements translation is the first formal step in the Genetic Design process and the first place where we have a chance to uncover requirements defects. Its purpose is to translate each natural language functional requirement, one at a time, into one or more behavior trees. Translation identifies the *components* (including actors and users), the *states* they realize (including attribute assignments), the *events* and *decisions/constraints* they are associated with, the *data* components exchange, and the *causal, logical,* and *temporal* dependencies associated with component interactions.

The behavior trees resulting from translations for the first six functional requirements for the Microwave Oven System given in Table 1 are shown in Figure 1.

*Figure 1. Behavior trees for Microwave Oven functional requirements*

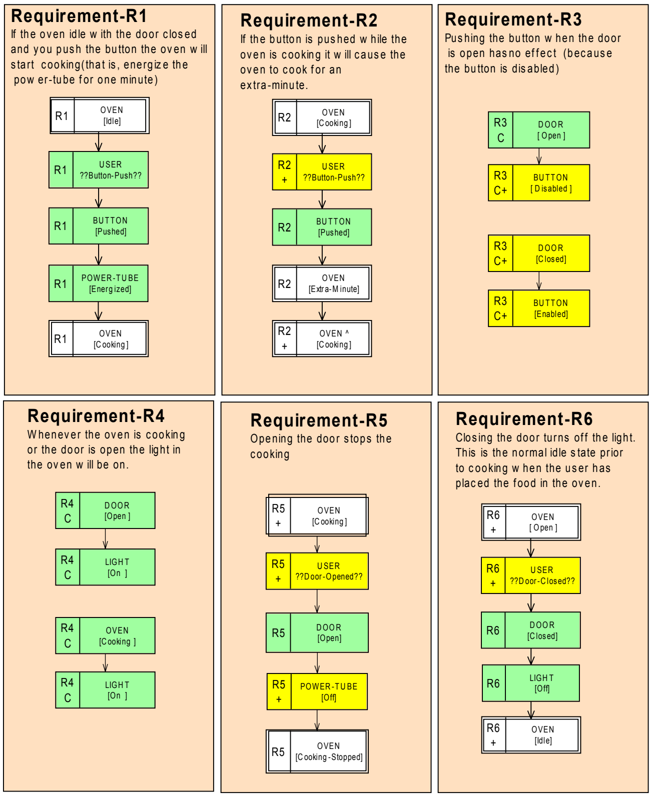
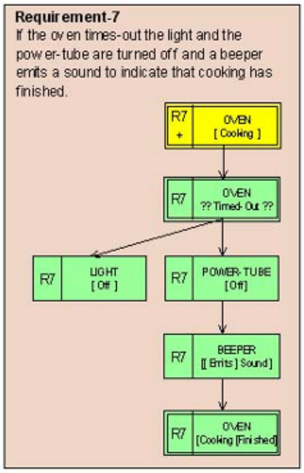

Requirement-R1
If the oven idle with the door closed and you push the button the oven will start cooking(that is, energize the power-tube for one minute)
R1 OVEN [Idle]
R1 USER ??Button-Push??
R1 BUTTON [Pushed]
R1 POWER-TUBE [Energized]
R1 OVEN [Cooking]
Requirement-R2
If the button is pushed while the oven is cooking it will cause the oven to cook for an extra-minute.
R2 OVEN [Cooking]
R2 + USER ??Button-Push??
R2 BUTTON [Pushed]
R2 OVEN [Extra-Minute]
R2 + OVEN ^ [Cooking]
Requirement-R3
Pushing the button when the door is open hasno effect (because the button is disabled)
R3 C DOOR [Open]
R3 C+ BUTTON [Disabled]
R3 C+ DOOR [Closed]
R3 C+ BUTTON [Enabled]
Requirement-R4
Whenever the oven is cooking or the door is open the light in the oven will be on.
R4 C DOOR [Open]
R4 C LIGHT [On]
R4 C OVEN [Cooking]
R4 C LIGHT [On]
Requirement-R5
Opening the door stops the cooking
R5 + OVEN [Cooking]
R5 + USER ??Door-Opened??
R5 DOOR [Open]
R5 + POWER-TUBE [Off]
R5 OVEN [Cooking-Stopped]
Requirement-R6
Closing the door turns off the light. This is the normal idle state prior to cooking when the user has placed the food in the oven.
R6 + OVEN [Open]
R6 + USER ??Door-Closed??
R6 DOOR [Closed]
R6 LIGHT [Off]
R6 + OVEN [Idle]

*Figure 2. Behavior tree produced by translation of requirement R7 in Table 1*


Requirement-7
If the oven times-out the light and the power-tube are turned off and a beeper emits a sound to indicate that cooking has finished.
R7 + OVEN [ Cooking ]
R7 OVEN ?? Timed-Out ??
R7 LIGHT [ Off ]
R7 POWER-TUBE [ Off ]
R7 BEEPER [ [ Emits ] Sound ]
R7 OVEN [Cooking-Finished]

## Example Translation

Translation of R7 from Table 1 will now be considered in slightly more detail. It involves identifying the **components** (including actors and users), the *states* (*italics*) they realize (including attribute assignments), and the order indicators (underlined), that is, the *events* and *decisions/constraints* they are associated with, the *data* components exchange, and the *causal, logical,* and *temporal* dependencies associated with component interactions. In making translations, we introduce no new terms, translate all terms, and leave no terms out. When these rules are followed, translation approaches repeatability. Functional requirement R7 is marked up using these conventions. "If the **oven** *times out* the **light** and the **power-tube** are *turned off* and a **beeper** *emits a sound* to indicate that *cooking has finished."* Figure 2 gives a translation of requirement R7, to a corresponding requirements behavior tree (RBT). In this translation, we have followed the convention of associating higher level system states (here, OVEN states) with each functional requirement, to act as preconditions/postconditions.

What we see from this translation process is that even for a very simple example, it can identify problems that, on the surface, may not otherwise be apparent (e.g., the original requirement, as stated, leaves out the precondition that the oven needs to be cooking in order to subsequently time out). In the behavior tree representation tags (here R7), provide direct traceability back to the original statement of requirements. Our claim is that the translation process is highly repeatable if translators forego the temptation to interpret, design, introduce new things, and leave things out, as they do an initial translation. In other words translation needs to be done meticulously, sentence-by-sentence, and word-by-word. In doing translations, there is no guarantee that two people will get exactly the same result because there may be more than one way of representing the same behavior. The best we can hope for is that they would get an equivalent result.

### *Translation Defect Detection*

During initial translation of functional requirements to behavior trees, there are four principal types of defects that we encounter:

- Aliases, where different words are used to describe a particular entity or action, and so forth. For example, in one place the requirements might refer to the *Uplink-ID* while in another place, it is referred to as the *Uplink-Site-ID*. It is necessary to maintain a vocabulary of component names and a vocabulary of states associated with each component to maximize our chances of detecting aliases.

- Ambiguities, where not enough context has been provided to allow us to distinguish among more than one possible interpretation of the behavior described. Unfortunately there is no guarantee that a translator will always recognize an ambiguity when doing a translation — this obviously impacts our chances of achieving repeatability when doing translations.
- Incorrect causal, logical, and temporal attributions. For example, in R4 of our Microwave Oven example, it is implied that the oven realizing the state "cooking" causes the light to go on. Here it is really the button being pushed which causes the light to go on and the system (oven) to realize the system-state "cooking." An example of the latter case would be "the man got in the car and drove off." Here "and" should be replaced by "then", because getting in the car happens first.
- Missing implied and/or alternative behavior. For example, in R5 for the oven, the actor who opens the door is left out, together with the fact that the power-tube needs to be off for the oven to stop cooking.

A final point should be made about translation. It does not matter how good or how formal the representations are that we use for analysis/design, unless the first step that crosses the *informal-formal barrier* is as rigorous, intent-preserving (Leveson, 2000), and as close to repeatable as possible, all subsequent steps will be undermined because they are not built on a firm foundation. Behavior trees give us a chance to create that foundation. And importantly, the behavior tree notation is simple enough for clients and users to understand without significant training. This is important for validation and detecting translation defects.

When requirements translation has been completed, each individual functional requirement is translated to one or more corresponding requirements behavior trees (RBTs). We can then systematically and incrementally construct a design behavior tree (DBT) that will satisfy all its requirements *by integrating the individual requirements' behavior trees* (RBT) one at a time. This step throws up another class of defects.

## Requirements Integration

Integrating two behavior trees turns out to be a relatively simple process that is guided by the precondition and interaction axioms referred to previously. In practice, it most often involves locating where, if at all, the component-state root node of one behavior tree occurs in the other tree and grafting the two trees together at that point. This process generalizes when we need to integrate N behavior trees. We only ever attempt to integrate two behavior trees at a time — either two RBTs, an RBT with a DBT, or two partial DBTs. In some cases, because the precondition for executing the behavior in an RBT has not been included, or important behavior has been left out of a requirement, it is not clear where a requirement integrates into the design. This immediately points to a problem with the requirements. In other cases, there may be requirements/ behavior missing from the set which prevents integration of a requirement. Attempts at integration uncover such defects with the requirements at the earliest possible time. Many defects with requirements can only be discovered by creating an integrated view because examining requirements individually gives us no clue that there is a problem. In Figure 4, we show the result of integrating the seven RBTs that result from requirements translation (plus the missing requirement R8 — see subsequent discussion). It is easy to see because of the tags, R1, R2, and so on, where each functional requirement occurs in the integrated DBT. "@@" mark integration points.

## Example Integration

To illustrate the process of requirements integration we will integrate requirement R6, with part of the constraint requirement R3C to form a partial design behavior tree (DBT) (note: in general, constraint requirements need to be integrated into a DBT wherever their root node appears in the DBT. This is straightforward because the root node (and precondition) of R3C, DOOR[Closed] occurs in R6. We integrate R3C into R6 at this node. Because R3C is a constraint, it should be integrated into every requirement that has a door closed

*Figure 3. Result of integrating R6 and R3C*

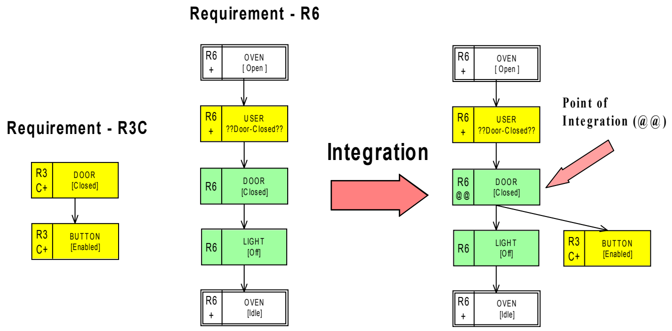
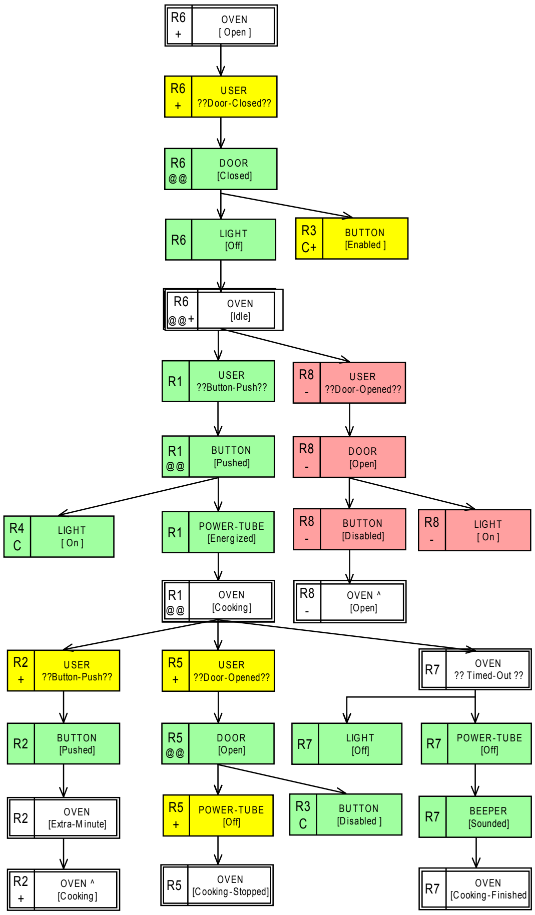


*Figure 4. Integrated design behavior tree (DBT) for Microwave Oven System*



state (in this case, there is only one such node). The result of the integration is shown in Figure 3.

When R6 and R3C have been integrated, we have a "partial design" (the evolving design behavior tree) whose behavior will satisfy R6 and the R3C constraint. In this partial DBT it is clear and traceable where and how each of the original functional requirements contribute to the design. Using this same behavior tree grafting process, a complete design is constructed and evolves incrementally by integrating RBTs and/or DBTs pairwise until we are left with a single final DBT (see Figure 4).

This is the ideal for design construction, realizable when all requirements are consistent, complete, composable, and do not contain redundancies. When it is not possible to integrate an RBT or partial DBT with any other, it points to an integration problem with the specified requirements that need to be resolved. Being able to construct a design incrementally significantly reduces the complexity of this critical phase of the design process. And importantly, it provides direct traceability to the original natural language statement of the functional requirements.

## Integration Defect Detection

During integration of functional requirements, represented as behavior trees (RBTs), there are four principal types of defects that we encounter:

- The RBT that we are trying to integrate has a missing or inappropriate precondition (it may be too weak or too strong or domain-incorrect) that prevents integration by matching the root of the RBT with a node in some other RBT or in the partially constructed DBT. For example, take the case of R5 for the Microwave Oven: it can only be integrated directly with R1 by including OVEN[Cooking] as a precondition.
- The behavior in a partial DBT or RBT where the RBT needs to be integrated is missing or incorrect.
- Both of the first two types of defects occur at the same time. Resolving this type of problem may sometimes require domain knowledge.
- In some cases, when we attempt to integrate an RBT we find that more than the leaf node overlaps with the other RBT or partial DBT. In such cases this redundancy can be removed at the time of integration.

While in principal, it is possible to construct an algorithm to “automate” the integration step, because of the integration problems that we frequently encounter in real systems, it is better to have manual control over the integration process. Tool support, however, can be used to identify the nodes that satisfy the matching criterion for integration. Our experience with using integration in large industry systems is that the method uncovers problems early on that have been completely overlooked using conventional formal inspections. The lesson we have learned is that requirements integration is a key integrity check that it is always wise to apply to any set of requirements that are to be used as a basis for constructing a design.

## *Inspection and Automated Defect Detection*

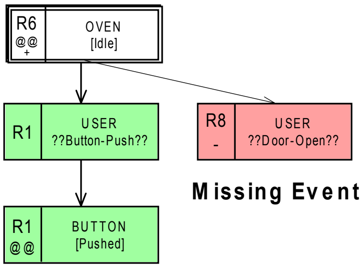
Once we have a set of functional requirements represented as an integrated design behavior tree, we are in a strong position to carry out a range of defect detection steps. The design behavior tree turns out to be a very effective representation for revealing a range of incompleteness and inconsistency defects that are common in original statements of requirements. The Microwave Oven System case study has its share of incompleteness and other defects.

The DBT can be subject to a manual visual formal inspection, and because behavior trees have a formal semantics (Winter, 2004) we can also use tools (Smith et al., 2004) to do automated formal analyses. In combination, these tools provide a powerful armament for defect finding. With simple examples, like the Microwave Oven, it is very easy to do just a visual inspection and identify a number of defects. For larger systems, with large numbers of states and complex control structures, the automated tools are essential for systematic, logically based, repeatable defect finding. We will now consider a number of systematic manual and automated defect checks that can be performed on a DBT.

## Missing Conditions and Events

A common problem is with original statements of requirements that describe a set of conditions that may apply at some point in the behavior of the system. They often leave out the case that would make the behavior complete. The simplest such case is where a requirement says what should happen if some condition applies but the requirements are silent on what should happen if the condition does not apply. There can also be missing events at some point in the behavior of the system. For example, with the Microwave case study, a very glaring missing event is in requirement R5. It says, "opening the door stops the cooking" but neglects to mention that it is possible to open the Microwave door when it is

*Figure 5. Missing event detected by the event completeness check rule*



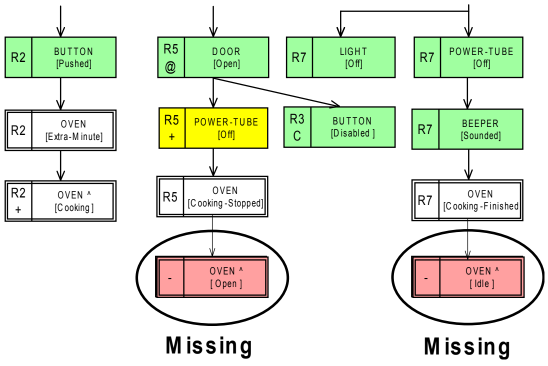
idle/closed. To systematically "discover" this event-incompleteness defect we can use the following process. We make a list of all events that can happen in the system (this includes the user opening the door). We then examine those parts of the DBT where events occur and ask the question, "could any of the other events that we have listed occur at this point?" In the case where the OVEN[Idle] occurs, the only event in the original requirements is that the user-event of pushing the button to start the cooking can occur (see Figure 4).

In this context, when we ask what other event, out of our list of events, could happen when the Oven is Idle, we discover the user could open the door. We have added this missing event in as requirement R8. It is interesting to note that when Schlaer and Mellor (1992) transition from the stated requirements to a state transition diagram, the missing behavior has been added without comment. Because of this sort of practice, traceability to the original requirements is lost in many design methods. Behavior trees which apply explicit translation and integration of requirements maintain traceability to the original requirements.

## Missing Reversion Defects

Original statements of requirements frequently miss out on including details of reversions of behavior that are needed to make the behavior of a system complete. Systems that "behave," as opposed to programs that execute once and terminate, must never get into a state from which no other behavior is possible – if such a situation arises, the integrated requirements have a reversion defect. Take the case of the Microwave Oven DBT in Figure 4. We find that if the Oven reaches either an OVEN[Cooking_Stopped] or an OVEN[Cooking_Finished] state, then no further behavior takes place. In contrast, when the system

*Figure 6. Reversion "^" nodes added to make DBT behavior reversion-complete*



realizes an OVEN^[Cooking] leaf-node, it "reverts" to a node higher up in the tree and continues behaving. To correct these two defects we need to append respectively to the R5 and R7 leaf nodes the two reversion nodes "^", shown in Figure 6.

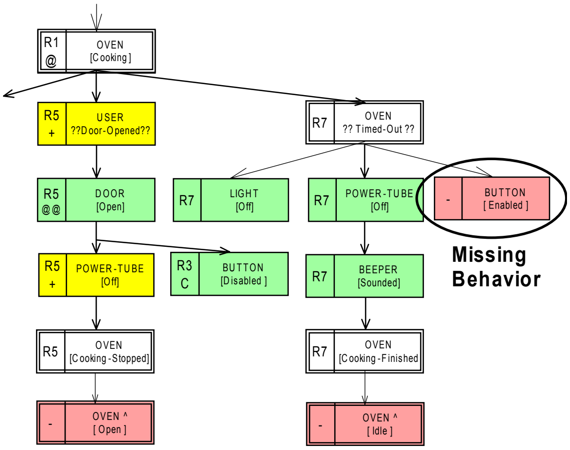
## Deadlock, Live-Lock, and Safety Checking

The tool we have built allows us to graphically enter behavior trees and store them using XML (Smith et al., 2004). From the XML we generate a CSP (Communicating Sequential Processes) representation. There are several translation strategies that we can use to map behavior trees into CSP. Details of one strategy for translating behavior trees into CSP are given in Winter (2004). A similar strategy involves defining subprocesses in which state transitions for a component are treated as events. For example, to model the DOOR [Open] to DOOR [Closed] transition, the following CSP was generated by the translation system:

$$\text{DoorOpen} = \text{userDoorClosed} \rightarrow \text{doorClosed} \rightarrow \text{DoorClosed}$$

The CSP generated by the tool is fed directly into the FDR model-checker. This allows us to check the DBT for deadlocks, live-locks and also to formulate and check some safety requirements (Winter, 2004).

*Figure 7. Missing behavior detected by checking OVEN[Idle]/OVEN^[Idle] component state consistency*



## Reversion Inconsistency Defect Detection

The current tool does a number of consistency checks on a DBT. One important check to do is a reversion "^" check where control reverts back to an earlier established state. For example, for the Microwave Oven example in Figure 4, one reversion check that needs to be done is to compare the states of all components at OVEN[Idle] with those at OVEN^[Idle]. What this check allows us to do is see whether all components are in the *same* state at the reversion point as the original state realization point. Figure 6 shows the bottom part of the Oven DBT from Figure 4.

We see that requirement R7 (and the DBT in Figure 4) is silent on any change to the state of the BUTTON component. This means we get from R1 that BUTTON[Pushed] still holds when OVEN^[Idle] is realized. However this is inconsistent with OVEN[Idle] established by R6 and constraint R3 which has the state for BUTTON as BUTTON[Enabled]. That is, the system-state definitions which show up the inconsistency are as follows:

OVEN[Idle] ≡ DOOR[Closed] ∧ LIGHT[Off] ∧ BUTTON[Enabled] ∧ …
OVEN^[Idle] ≡ DOOR[Closed] ∧ LIGHT[Off] ∧ BUTTON[Pushed] ∧ …

These sort of subtle defects are otherwise difficult to find without systematic and automated consistency checking.

There are a number of other systematic checks that can be performed on a DBT, including the checking of safety conditions (e.g., in the Microwave Oven requirement R5, it indicates that the door needs to realize the state open to cause the power-tube to be turned off — this clearly could be a safety concern). We will not pursue these checks here as our goal has only been to give a flavor of the sort of systematic defect finding that is possible with this integrated requirements representation. We claim, because of its integrated view, that a DBT makes it easier to "see" and detect a diverse set of subtle types of defects, like the ones we have shown here, compared with other methods for representing requirements and designs. We have found many textbook examples where this is the case.

Once the design behavior tree (DBT) has been constructed, inspected, and augmented/corrected where necessary, the next jobs are to transform it into its corresponding software or component architecture (or *component interaction network* — CIN) and project from the design behavior tree the component behavior trees (CBTs) for each of the components mentioned in the original functional requirements. We will not pursue these design steps here or the associated error detection. They are dealt with elsewhere (Dromey, 2003), as is the necessary work on creating an integrated view of the data requirements and compositional requirements of a system which reveal still other defects.

Having provided a detailed description of how behavior trees, in combination with genetic design, may be systematically used to detect a wide range of defects it is important to position this approach relative to other methods that are used to detect requirements defects. Here we will only do this at a high-level and in a qualitative way, because it is on the big issues where behavior trees differ from other options for requirements defect detection.

## Comparison with Other Methods

Methods for detecting requirements defects can be divided into three broad classes: those that involve some form of human inspection of the original *informal* natural language statement of requirements, Fagan (1976), and more recently, a perspective-based approach (Shull & Basili, 2000); those that seek to create *graphic* representations of the requirements information (Booch et al., 1999; Harel, 1987); and those that seek to create a *formal* (usually symbolic) representation of the requirements information (Prowell et al., 1999) in an effort to detect defects.

There are three weaknesses with the informal inspection methods for requirements defect detection. They make no significant attempt to tackle the problem of complexity because they make no changes to the original information and therefore they come up against the memory overload problem. They also do nothing of significance to detect problems associated with the interaction between requirements because they do not create an integrated view. And thirdly, because they do not seek to map the original requirements to another representation they are less likely to uncover ambiguity and other language problems (aliases, etc.) with a large set of requirements. In contrast behavior trees tackle complexity by only needing to consider one requirement at a time, they create an integrated view of requirements, and they employ a translation process that seeks to preserve original intention.

Graphic analysis and design methods, like UML (Booch et al., 1999), state-charts (Harel, 1987), and the method proposed by Schlaer and Mellor (1992), provide an analysis strategy that involves the creation of a number of partial views of the requirements information. The main difficulties with these approaches is that different analysts are likely, for a given problem and method, to produce widely different work products including the choices they make about which types of diagram to produce. The problem is compounded by overlap between some of the different types of diagram, and in some cases, these diagrams lack a rigorous semantics and are therefore open to more than one interpretation. There is also no direct or clear focus on how to use these methods and the various views/ diagrams to detect defects. Because a variety of diagrams may be used, and the mapping does not involve rigorous translation, it is very difficult to guarantee preservation of intention; this introduces yet another potential source for creating defects. We also commonly find, when these methods are used by practitioners, that things in the original requirements get left out, new things get added, and the vocabulary changes as the transition is made from textual requirements to a set of graphic views (Dromey, 2005). In contrast to these problems with graphic methods, the behavior tree notation allows us to create a single integrated view of the behavioral requirements and a single integrated view of the static compositional and data requirements (not discussed in this chapter) of a system. Creation of behavior tree work-products approach repeatability of construction when produced by different analysts because they are based on rigorous translation which always has the goal of preservation of intention. As we have shown, behavior trees also employ a number of clearly defined strategies for detecting a number of different classes of defects. Behavior trees have also been given a formal semantics, which makes it possible to implement formal automated model-checking.

The use of formally based notations and methods has always been seen as a way of detecting defects with a set of requirements because of the consistency and completeness conventions they enforce. While there is no doubt in principle that this is true, there are three common problems with the use of formal notations. They require a transition from an informal statement of requirements to a formal representation. The problem here is whether this transition is done in a way that preserves the original intention. Most formal notations, for example, the Cleanroom method, do not employ a process that approaches the repeatability of requirements translation in making the transition from the informal to the formal representation. Another difficulty with formal notations is that very often they are not easily understood by clients and end users. This can make it much more difficult to guarantee intention has been preserved. In contrast, clients and end users have little difficulty in understanding behavior trees because the notation is relatively simple. One of the greatest difficulties with formal notations is that they do not usually easily scale up to dealing with larger and more complex systems. This is much less of a problem with behavior trees provided one has the necessary tool support.

# Conclusion

Control of complexity is key to detecting and removing defects from large sets of functional requirements for all software-intensive systems, including information systems. Genetic design facilitates the control of complexity because it allows us to consider, translate, and integrate only one requirement at a time. At each of the design steps, a different class of requirements defects is exposed. It is also much easier to see many types of defects in a single integrated view than spread across a number of partially overlapping views as is the case with methods based on representations like UML or state-charts. Because the design is built out of individual requirements, there is direct traceability to original natural language statements of requirements. Translation of individual requirements rather than some form of abstraction goes a long way to preserve and clarify intention as well as to find defects in original statements of requirements. This approach to design and defect detection has been successfully applied to a diverse range of real (large) industry applications from enterprise applications, to distributed information systems, to workflow systems, to finance systems, to defense-related command and control systems. In all cases the method has proved very effective at defect detection and the control of complexity. The utility of the method will increase as we develop more powerful tools that make it easy to control vocabulary, support multiple users, and perform a number of formal checks at each of the design stages.

# References

Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The unified modelling language user guide*. Reading, MA: Addison-Wesley.

Davis, A. (1988). A comparison of techniques for the specification of external system behavior. *Communications of the ACM*, *31*(9), 1098-1115.

Dromey, R.G. (2003, September). From requirements to design: Formalizing the key steps (Invited Keynote Address). *IEEE International Conference on Software Engineering and Formal Methods*, Brisbane, Australia (pp. 2-11).

Dromey, R.G. (2005). Genetic design: Amplifying our ability to deal with requirements complexity. In S. Leue & T.J. Systra (Eds.), *Scenarios, Lecture Notes in Computer Science, 3466*, 95-108.

Fagan, M. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal, 15*(3), 182-211.

Harel, D. (1987). Statecharts: Visual formalism for complex systems. *Science of Computer Programming, 8*, 231-274.

Leveson, N. (2000, January). Intent specifications. *IEEE Transactions on Software Engineering, SE-26*(1), 15-34.

Mayall, W.H. (1979). *Principles of design*. London: The Design Council.

Prowell, S., Trammell, C.J., Linger, R.C., & Poore, J.H. (1999). *Cleanroom software engineering: Technology and process*. Reading, MA: Addison-Wesley.

Schlaer, S., & Mellor, S.J. (1992). *Object lifecycles*. NJ: Yourdon Press.

Shull, I., & Basili, V. (2000, July). How perspective-based reading can improve requirements inspections. *IEEE Computer, 33*(7), 73-79.

Smith, C., Winter, K., Hayes, I., Dromey, R.G., Lindsay, P., & Carrington, D. (2004, September). *An environment for building a system out of its requirements.* Proceedings of the 19th IEEE International Conference on Automated Software Engineering, Linz, Austria.

Winter, K. (2004). Formalising behavior trees with CSP. *Proceedings of the International Conference on Integrated Formal Methods, LNCS, 2999* (pp. 148-167).

Woolfson, A. (2000). *Life without genes*. London: Flamingo.

## Chapter V

# User Participation in the Quality Assurance of Requirements Specifications: An Evaluation of Traditional Models and Animated Systems Engineering Techniques

Heinz D. Knoell, University of Lueneburg, Germany

## Abstract

*Improper specification of systems requirements has thwarted many splendid efforts to deliver high-quality information systems. Scholars have linked this problem to, between others, poor communication among systems developers and users at this stage of systems development. Some believe that specifying requirements is the most important and the most difficult activity in systems development. However, limitations in human information processing capabilities and the inadequacy of the structures available for* *communicating specifications and obtaining feedback and validation help to exacerbate the difficulty. This chapter presents an overview of both longstanding and newer requirements specification models and evaluates their capability to advance user participation in this process and incorporate stated quality attributes. It also reports on preliminary evaluations of animated system engineering (ASE), the author's preferred (newer) technique, which indicate that it has the capability to improve the specification effectiveness.*

## Introduction

It is estimated that between 30% and 80% of software projects fail (Dorsey, 2003; Standish Group, 1994), depending on whether the basis is budgets or number of projects. Many of these software projects fail because of their inability to adequately specify and eventually meet customer requirements (Zave & Jackson, 1997). The following quotation from The Standish Group (1994) provides an excellent summary of the situation and puts the problem in perspective:

> *In the United States, we spend more than $250 billion each year on IT application development of approximately 175,000 projects. The average cost of a development project for a large company is $2,322,000; for a medium company, it is $1,331,000; and for a small company, it is $434,000. A great many of these projects will fail. Software development projects are in chaos, and we can no longer imitate the three monkeys — hear no failures, see no failures, speak no failures.*
>
> *The Standish Group research shows a staggering 31.1% of projects will be cancelled before they ever get completed. Further results indicate 52.7% of projects will cost 189% of their original estimates. The cost of these failures and overruns are just the tip of the proverbial iceberg. The lost opportunity costs are not measurable, but could easily be in the trillions of dollars. One just has to look to the City of Denver to realize the extent of this problem. The failure to produce reliable software to handle baggage at the new Denver airport is costing the city $1.1 million per day.*
>
> *Based on this research, The Standish Group estimates that in 1995 American companies and government agencies will spend $81 billion for canceled*

> *software projects. These same organizations will pay an additional $59 billion for software projects that will be completed, but will exceed their original time estimates. Risk is always a factor when pushing the technology envelope, but many of these projects were as mundane as a driver's license database, a new accounting package, or an order entry system.*

The fact is that too many software projects fail, that these failures may be due to both technical and behavioral reasons. Obtaining accurate systems requirements (Zave & Jackson, 1997) and translating them into feasible specifications is a well-discussed problem; however, involving potential users in the development project is an important factor in this process. In Levina and Vaast's (2004) investigation of how innovations are brought into enterprises, they underscored the pivotal nature of user involvement in successful implementation.

The specification of the requirements of an information system occurs fairly early in the development lifecycle. To accomplish this task, users and developers collaborate to describe the processes and static structures that are involved in the application domain and define their relationships. Quite often both sides speak a different language, using terminology that may be unfamiliar to the other. Clients express themselves using (in the view of technocrats) informal business terminology; developers write system specification from a technical perspective. It is widely acknowledged that this miscommunication is the reason for the prevalence of poorly specified systems and the root cause of many of the failed information systems (Byrd et al., 1992; Raghaven et al., 1994).

To solve this problem, the information system community introduced several models to improve communication between developers and users, particularly the accuracy and understandability of the representation of the business process information that eventually will be used to create the design artifact. But even with these models, users often experience some difficulty in assimilating the essence of the specification details. Sometimes they do not participate in the process because of cognitive limitations and an inability to comprehend the models. However, user participation in this process is essential to the production of accurate specifications. This has intensified the need to provide representational schemes for communicating specifications that are both accurate and easy to understand.

This chapter uses a real life scenario to assess a variety of modeling techniques and tools that have been employed to improve the quality of systems specifications and their success at accommodating user participation to improve user-developer communication. It reviews the models and demonstrates how they represent the scenario details. The discussion is organized somewhat chronologically beginning with the earlier techniques and culminating with a more detailed description of a proposed set of methods — animated system engineering (ASE)

— which the author believes may overcome many of the difficulties of earlier models.

## User Involvement in Improving the Quality of Requirements Specification

Several quality assurance initiatives have been undertaken in information systems development since Dijkstra (1972) alluded to the "software crisis". Earlier efforts focused on the quality assurance of computer programs. Soon it was recognized that this was far too late to be effective. Either programs were implemented with significant patchwork or they had to be extensively revised. In the worst-case scenarios, total recoding was necessary, which, according to Knoell and Suk (1989), was tantamount to sabotage. However, it has long been acknowledged that software quality assurance should at least begin with requirements specification (Karlson, 1997). At this point, there is already enough information compiled to necessitate a quality review.

While several well-established characteristics exist to assess the quality of a finished software product, these cannot be applied to requirements specification. The concern at this stage is how to drive toward the attainment of the eventual software quality by applying standards that will increase the quality of the deliverable at this stage of development. Scholars have begun to focus on this area. For example, Davis et al. (1997) identified several quality criteria related to software requirements specification, and Knoell et al. (1996) provided an ontology of quality terms in requirements specification to simplify and promote user understanding and participation in the process. In this chapter we refer to the quality criteria for requirements specification that have been suggested by the German working group of accounting auditors (IDW, 2001) because of their simplicity and absence of technical nomenclature. Table 1 summarizes these characteristics.

Despite the existence of these quality criteria, the problems with requirements enumerated by Dorsey (2003), which include incomplete and ambiguous specification and lack of change management and user involvement in requirements specification, are remarkably similar to those Boehm and Ross (1989) identified. This suggests that very little has changed within that last 15 years, and improper specification of software requirements continues to thwart many splendid efforts to deliver high-quality information systems (Dorsey, 2003).

Scholars have linked this problem to, among other things, poor communication between systems developers and users at this stage of systems development (Raghaven et al., 1994). Metersky (1993) and Zmud et al. (1993) believe that

*Table 1. Quality characteristics of requirements specification*

| **Quality Characteristics** | **Meaning** | **How Determined** |
|---|---|---|
| Completeness | Inclusion of all information about work sequences and objects (transactions and entities), including a reliable catalog of all attributes of transactions, transaction types, entities, and entity types | When the attributes for application projects, processes, entities, transactions, and functions are given content values in the validity range for the respective attribute |
| Consistency | No contradiction in the information about transactions and entities in the catalog of all permissible transactions and entities | Cross-checking the attributes of the system components |
| Clarity | Precise and unambiguous statement of the system features in terms that can be understood by all those participating in the project | Verification of system specification with future users and through forward traceability into the design |
| Factual accuracy | The extent to which functions of the existing business system description are contained in the requirements specification | Verifying that all the internal processing rules, operational procedures, and data are accounted for |
| Feasibility | Whether the requirements are achievable with the available resources | Matching resources against project demands |

specifying requirements is arguably the most important and the most difficult activity in systems development. However, limitations in human information processing capabilities and the inadequacy of the structures available for communicating specifications to users to obtaining their feedback and validation help to exacerbate the difficulty.

This is because software requirements specification depends on the constructive collaboration of several groups, typically with different levels of competence, varying interests and needs (Knoell & Knoell, 1998). The challenge is how to effectively communicate the perspectives of all stakeholders, in a manner that is understandable to all, to help them participate in a process of information sharing to rationally decide the correct approach and unearth creative solutions that may well involve non-technical issues such as new work flows and business process alterations. There is little doubt that user involvement in this process is pivotal and the models used to capture and present this perspective for effective user decision making even more so.

In the following sections, we provide an overview of traditional methods and then describe newer and less known models. The older ones are still useful in various contexts as we will describe; however, complex new systems also require other models to incorporate user participation in order to accommodate desirable quality characteristics and improve results.

# Overview of Models Used for Specifications

In order to more clearly demonstrate the techniques that will be described and make their underlying principles more concrete, the following real-life scenario will be used to construct a simple example of the models that will be discussed. These methods are flow charts and a modified version of the flow chart, data flow diagrams, entity relationship diagrams, decision tables, the use case diagram — one of the models from the family of the unified modeling language (UML) that supports object-oriented development — and Petri Nets, a modeling language used to graphically represent systems with facilities to simulate dynamic and concurrent activities.

This scenario involves requirements for a system at a non-profit organization in Germany, which supports grieving families after the loss of a child. Besides membership administration, the organization maintains lists of contact persons, rehabilitation hospitals, speakers, and seminars and distributes books, brochures, and CDs to interested persons and organizations. The three major tasks in the process of distributing informational material are order entry, order checking, and sending notifications about the status of orders.

## Flow Charts

The flow charting technique has been around for a very long time, so long in fact, that there is no "father of the flow chart" and no one individual is accorded rights of authorship. In general, a flow chart can be customized to describe any process in order to communicate sequence, iteration, and decision branches. It may be used in a variety of situations and is recognized as an excellent technique for helping to demonstrate processes in order to improve quality (Clemson University, 2004). Figure 1 shows the flow chart for our case study.

### *The Modified Flow Chart*

The basic flow chart technique has been modified by Knoell et al. (1996) in order to enhance communication by the introduction of additional symbols that make it easier for non-technical users to understand. This modified technique distinguishes between objects (ellipse or oval) and actions (rectangle) by including icons, which symbolize the object and action, respectively. Figure 2 depicts the modified version. This representation has been used successfully in several projects and has been well accepted by non-technical users. It has also been used

*Figure 1. Order processing flow chart*

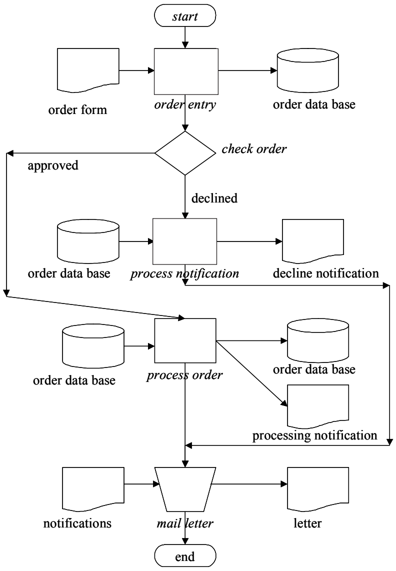
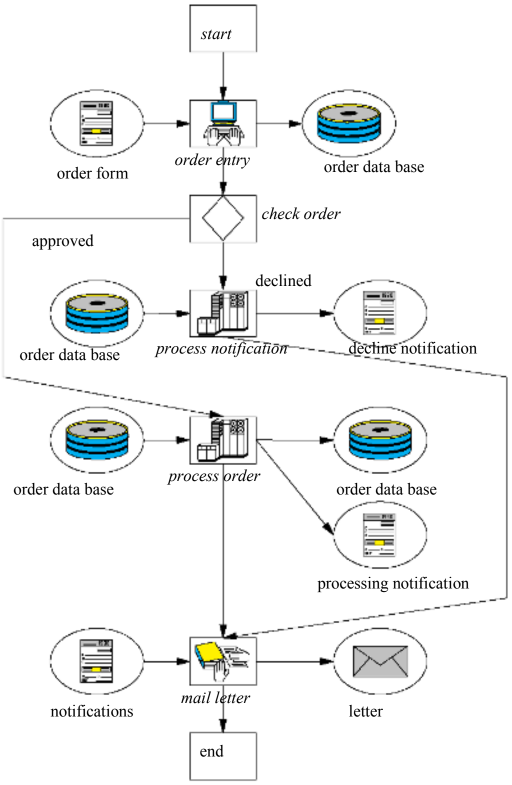


*Figure 2. Processing orders as a KS-Diagram (Knoell et al., 1996)*



in the design phase, where several refinement levels have been added (Knoell et al., 1996).

## *Data Flow Diagrams*

Structured Analysis (SA) (DeMarco, 1979; Svoboda, 1997; Yourdon, 1979) has been developed to support systems analysis and particularly to demonstrate process and data structures in order to improve user-developer communication and understanding of the processes to be automated. Three basic models exist: (1) the data flow diagram, which supports requirements specification (and will be elaborated on later), provides a functional view of processes including what is done (processes) and the flow of data between these processes; (2) the entity relationship diagram, which provides a static view of data relationships; and (3) a transition diagram, which provides a dynamic view of when events occur and the conditions under which they occur.

The data flow diagram uses two symbol sets (or representation schemes) (Svoboda, 1997): one originated by Yourdon and DeMarco (DeMarco, 1979) and the other provided by Gane and Sarson (1977). We have selected the Yourdon-DeMarco notation in this chapter.

The description of the system is refined into as many levels as needed. The first is the context level data flow diagram (Figure 3), which depicts the interaction of the system with external agents that provide and receive information to and from it. This level is used to give a general overview of the system to be developed. It is the basis for the first meetings with users. As soon as this level is accepted by the users and acknowledged to be feasible by the systems analysts, the next level, a data flow diagram, is developed (see Figure 4).

*Figure 3. Context level data flow diagram*

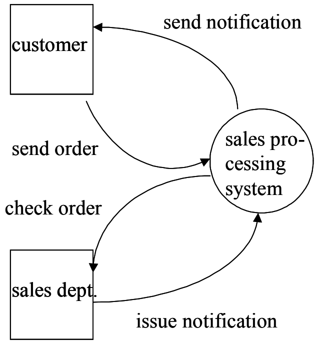
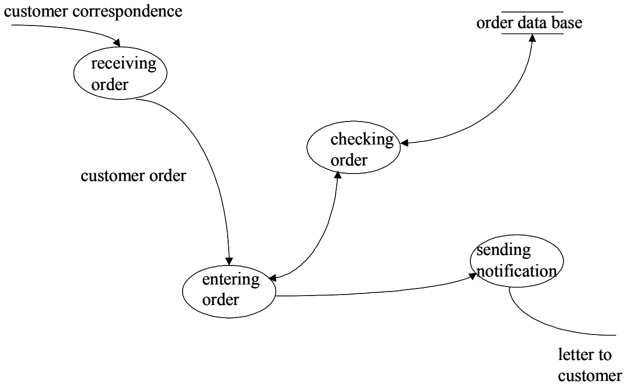


*Figure 4. Level 0 data flow diagram*



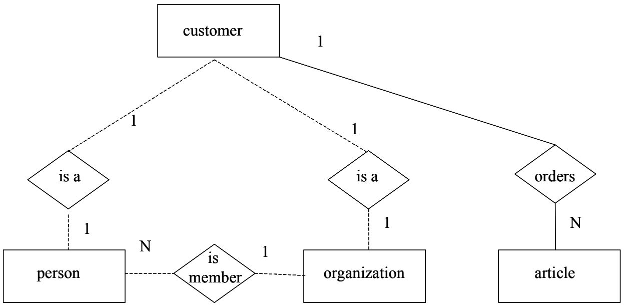
The functions of the level 0 data flow diagram are shown in Figure 4. It is a high level view of the system. Each process can be elaborated in its own diagram until the so called "functional primitive" level is reached. At that point, there is little communication value in decomposing the process further.

In addition to data flow diagrams, which were very prominent in the 1980s and 1990s, several other methods have been used to support functional analysis. Newer techniques, such as those provided by the Unified Modeling Language (UML), the de facto notations and modeling standard used in object-oriented development, have made structured techniques redundant in many cases. Although SA is still in use in industry, most universities have switched to UML. According to the experience of IT consultants (Berger, 2001; Kuehl, 2001), the SA Data Flow Diagrams have been harder for non-IT users than flow charts — "The average user thinks like that: I take A, perform B and the output is C" (Berger, 2001).

## Entity Relationship Diagrams (ERD)

The data-oriented approach to requirements specification (Jackson, 1975) was introduced during the period when large software systems threatened to confound programmers, a phenomenon that Dijkstra (1972) dubbed the "Software Crisis." Chen (1976) helped to solve the problem of representing information in a way that systems analysts, programmers, and customers could all understand; this solution was the Entity Relationship Diagram (ERD) approach to database design, which was extended to the ERD technique for systems

*Figure 5. Entity-relationship diagram (Chen, 1976)*



design (Chen & Knoell, 1991). The ERD approach enlightened both technocrats, who had struggled with a combination of hierarchical databases (and the emerging relational databases), as well as users, who began to understand the structure of data.

Figure 5 demonstrates the entities and their relationships involved in the scenario that is used as an example in these models. It gave the customer and the programmer a relatively simple way to discuss the requirements of the system to be built. Chen developed rules to create optimum databases out of the ERDs (Chen & Knoell, 1991), which helped the analysts and programmers develop normalized, non-redundant relational databases.

## Decision Tables

Sometimes, because of the many rules to apply to certain decision points in a process, it is not useful to use diagrams with decision branches (such as flow charts, or UML and Petri Nets, which will be discussed later). A graphic representation of such complex decision processes is often difficult to under-

*Figure 6. Example of a decision table*

| Conditions | Rule #1 | Rule #2 | Rule #3 | Rule #4 | Rule #5 | Rule #6 | Rule #7 | Rule #8 |
|---|---|---|---|---|---|---|---|---|
| wholesaler | Y | Y | Y | Y | N | N | N | N |
| regular customer | Y | Y | N | N | Y | Y | N | N |
| order value > $ 10,000.- | Y | N | Y | N | Y | N | Y | N |
| | | | | | | | | |
| **actions** | | | | | | | | |
| 3 % discount | - | X | - | X | X | - | X | |
| 5 % discount | X | - | X | - | - | - | - | - |
| free shipping | X | X | - | - | X | X | - | - |

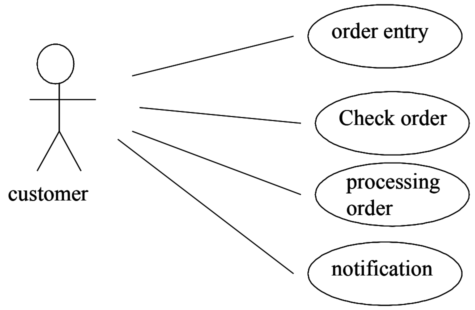
stand, especially for non-technical users. In those situations decision tables, which have been the basis for program generators in COBOL in the 1980s like Vorelle (mbp GmbH) and SWT (SWT GmbH) (Grawe, 2001), are far more suitable. A decision table is divided into quadrants: upper left (a list of conditions); upper right (the combination of conditions into rules); lower left (a list of possible actions); and lower right (the combination of actions according the rules — combination of conditions).

Unlike the methods described previously, a decision table is not suitable for outlining the logical structure of the information, or the flow of data, or the sequence of processes. It is an add-on to the other methods, used to describe the system's behavior at a certain point of the process flow. However, the decision table is well suited for specification negotiations with business users who are more accustomed to working with tables. If the process is very complex, nested decision tables may be used to decompose a very large decision into smaller, interdependent tables in which the master decision table contains the dependent decision tables as actions.

## Use Cases from the Unified Modeling Language (UML)

UML integrated the concepts of Booch (1993), Object Modeling Technique (OMT) (Rumbaugh et al., 1991) and Object-Oriented Software Engineering (OOSE) developed by Ivar Jacobson in 1992 (*Wikipedia*, 2004), by combining them into a single, common notation and modeling standard for object-oriented software development that is now widely used to model concurrent and distributed systems (Wikipedia, 2004) and was the basis for the Object Manage-

*Figure 7. UML use case diagram*



ment Group (OMG) Unified Modeling Language Specification (OMG, 2003). Of the several UML models, the most popular is the Use Case diagram for requirements specification, and, in the experience of the author, it is easily understood by users. In Figure 7 we have modeled the main processes of the system highlighted in the case scenario in an UML Use Case diagram. The model also further supports a narrative description of the use case (which is not developed here).

Rational Rose is one of the several tools that have been developed to support the construction of use case diagrams. It was developed by James Rumbaugh and Ivar Jacobson and later acquired by IBM and incorporated into the Software Group Division where it became an IBM software brand. Originally, Rational Rose was designed to support the development of ADA programs but later C++ and Java were added. In the newest version, Rational Rose supports the development of any object-oriented software system and allows animations of the designed UML diagrams in order to verify the process sequences and the associated data structures (IBM, 2004).

## Petri Nets

The Petri Net models described below are methods for capturing and graphically representing requirements that attempt to facilitate better user understanding, assimilation, and assessment in order to increase the odds of accommodating quality attributes. Like the other tools, they assist visual communications but, in addition, can simulate dynamic and concurrent activities. Petri Nets are a well-known design method in Europe, though not as widely used in the United States. Originally they were used in technical computing (e.g., in switchboards); however, for the last 20 years or so, they have been used increasingly in information systems development.

The Petri Net methodology was developed by Petri (1962) primarily to show the behavior of finite state machines (a theoretical construct in computer science). Initially, its application was restricted to technological problems, but since the 1990s it has been used in the specification phase of information systems development. Petri Nets can be animated, but they are mostly used for the resolution of technical problems like concurrent, asynchronous, distributed, parallel, non-deterministic, and/or stochastic systems.

Nowadays, an abundance of Petri Net-based design tools are available. These range from open-source facilities in the public domain to expensive professional tools. Three such tools, that are outstanding for their animation features, are BaaN Dynamic Enterprise Modeler, Pavone Process Modeler, and Animated Process Illustration (APRIL).

*Figure 8. Petri Net*

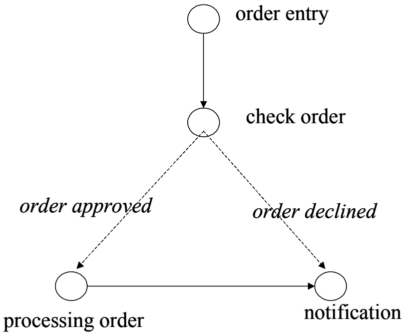
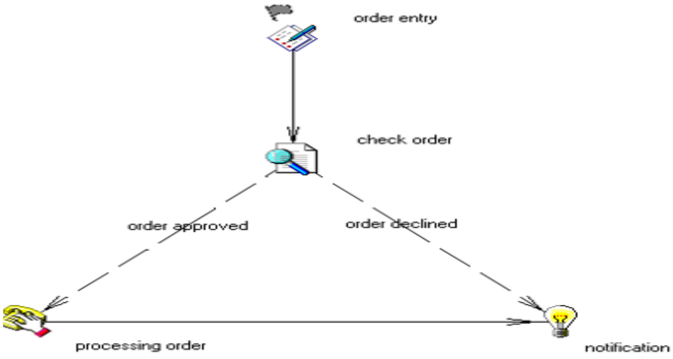


*Figure 9. Modified Petri Net – Pavone Process (Modeler, 2001)*



The BaaN Dynamic Enterprise Modeler (DEM) (BaaN, 1999) is a graphic program, which is used for the representation of the company's business processes. This model is used to assist users to customize the BaaN system and for the transfer of the business process model into the BaaN ERP system. The DEM has an animation mode in which users can check settings prior to the installation of the ERP software system. Furthermore, the graphic model is the user interface, from which users can trigger BaaN sessions.

The Pavone Process Modeler (Pavone, 2001) was developed to support the design of workflow in Lotus Notes applications. Figure 9 shows an example drawn with this program. For computer-naïve personnel, this depiction is much easier to understand as the Process Modeler uses icons to the differentiation of various processes. In addition, the persons or groups associated with the process have to be added. The Process Modeler also has an animation mode, similar to BaaN DEM, which is used to trace process flows, a feature is particularly helpful in communication with non-IT staff.

Animated Process Illustration (APRIL) is a Petri-Net-based approach to developing information systems that uses diagrams to make the process more transparent to both system engineers and users. The diagrams are used throughout the entire software development process. APRIL diagrams provide the means to simulate and analyze the system's behavior by synthesizing common techniques and adjusting them to each other and generating prototypes. The CASE tool NEPTUN is used in conjunction with APRIL to automatically generate a stand-alone platform and database system independently of the system's generic model (Simon et al., 1997). APRIL diagrams and the CASE-tool NEPTUN are integral parts of the Animated Systems Engineering Method, which is described in detail in the next section.

## Animated Systems Engineering (ASE)

Animated systems engineering is based on the philosophy that prototyping in the problem definition, analysis, and design phases through animation will help to circumvent most of the communication and decision errors that are typical of participating stakeholders (Boehm & Ross, 1989). The term animation is used instead of simulation because no executable specification is generated based on the dynamic model (Marx, 1998). ASE uses a graphic method to give the vivid representation of the dynamic system behavior without programming. For quality assurance purposes, executable specifications are available too late in the software development life cycle to figure out misunderstandings in the actual state analysis regarding requirements definition.

The animation of a Petri Net allows one to visualize the performance of a system (e.g., user interaction and loops) and to find problems in the system. It provides a good opportunity to recognize missing or misunderstood parts in the model under construction, because of the needed knowledge about the correct order of actions and object flows and not only about the static structure. It will also be useful for management to prove the progress of a project.

Pahnke and Knoell (1999) found that the ASE method indirectly improved the quality of specifications during groupware development by:

- structuring the software development process under consideration of quality criteria;

- showing the dynamics of the groupware processes;
- reducing the impact of language differences, which increases user participation by promoting agreement among project participants;
- managing complexity with easy-to-understand graphic elements; and
- producing an early graphic prototype to develop an executable problem description or requirements analysis.

To effect these benefits, ASE includes several tools such as the METAPLAN-Technique, Petri Net diagrams, object modeling diagrams, animation, mask/ screen and text generators, and a document management system with versioning. These are noted below:

- The METAPLAN (2003) moderation is a communication and discussion technique, which motivates a working atmosphere, within participating groups that are free of hierarchy influences (Pütz, 1999). In our students' projects the METAPLAN cards should be used in the form of prefabricated APRIL-Symbols to speed up the discussion.
- APRIL prototype from the Neptun Project of the University of Koblenz (Neptun, 2002) is used for the validation of ASE.
- Object Modeling Diagrams (OMD) are not directly needed for the core method, but they are useful from the detailed design phase and include the object oriented representation of the static view (data view), using classes, attributes and roles, as well as the graphic representation of association, aggregation, and inheritance.
- Animation is the core part of the ASE method. It is used to build and show scenarios during the investigation and analysis of the actual state. Scenarios show a sequence of events according to Rumbaugh et al. (1991). Animation is more effective than prototyping and provides more than the look and feel dimension of prototyping; its processes are performed on a higher level of abstraction. For user acceptance, this modeling approach needs easily understandable graphic elements. It should be used as late as possible (after a structure of the system is visible, but as early as needed) to get an overview of the dynamics of the system.

The following components of ASE are directly needed for quality assurance of specifications. They serve to fulfill the requirements for quality explanation according to ISO 12119 (ISO, 1994); the Capability Maturity Model (Paulk et al., 1993); standards in the pharmaceutical industry (which has very strict rules for software development, generally, and requirements specification, in particular;

for example, ISO 10007, 1995); and European Community guidance note for Good Clinical Practice (APV, 1996; Committee for Proprietary Medical Products, 1997).

- The Mask/Screen Generator is needed to produce the graphic user interface as early as possible, to easily develop a simple representation of menus. This representation should help novice users to overview and develop a vision of feasible solutions.
- A Text Generator is also very useful for documenting the decisions of the development team participants. The ideal situation would be the capability to branch from the respective Petri Net or animation directly to these documents.
- A Document Management System with versioning is required to prove the changes during the phases and identify requirements. The Petri Net animation has to be integrated into a document management system, to allow versioning through an archive.

## Indications of the Effectiveness of ASE

The motivation for recommending ASE as a possible cure for the communication problems that often beset requirements specification is based on indications of its successful use. Pahnke et al. (2004) reported on supervised students in two independent projects and further conducted a case study of a groupware development project at a pharmaceutical company in which ASE played a significant role in generating high-quality requirements.

In the two independent student projects, teams of students, each using a different method, generated the requirements specification of software systems. They were required to elicit requirements and present and discuss the specification with the future users. They then developed a prototype of the system. In both projects the teams using ASE were judged to be the ones that obtained the highest quality results and produced the best prototype (Pahnke, 2003).

The case study (Pahnke et al., 2001) was a more elaborate undertaking. It involved the pharmaceutical company, Merck KgaA. Merck has to meet quality standards specified by the US Food and Drug Administration (FDA). All of the drug studies submitted by Merck to the FDA and other regulatory agencies must satisfy good laboratory practice (GLP), good clinical practice (GCP), and good manufacturing practice (GMP). The FDA also requires that computer hardware and software used in drug studies satisfy good practices and conform to standards. The company uses several groupware and workflow systems and puts them through a quality assurance (QA) process using advanced QA concepts beyond the "fitness for purpose" requirement of ISO/IEC 12119 (ISO, 1994); these include user convenience, effectiveness, and data safety.

The company launched the Merck Electronic Document Management & Information System (MEDIS) to manage the global collection, electronic compilation, and dissemination of information through the planning, preparation, and publication of documents and dossiers, and related management processes including the preparation of all pharmaceutical submissions to regulatory agencies. Triple D (short name for Drug Data to Disc) was the project name for a document and workflow management system, developed to satisfy special needs of Merck's pharmaceutical submissions.

Merck adopted groupware as the management technology and implemented role-based privileges for managing the creation, and publishing of dossiers. An essential mandate of the FDA and other regulatory agencies is for dossier creation and submission to be under version control. In Merck's case, those allowed to create documents are able to build their own information workflows; however, the workflows for review and approval have to be predefined for some roles within a working group. The project therefore required the collaboration of several stakeholders and coordinated decision-making by several participants.

Merck decided to apply ASE after evaluating the presented examples in relation to the special QA needs of the MEDIS project. Participants were trained in ASE techniques and later determined the ASE features that would be required to monitor and achieve their quality requirements in all development phases. They selected a mask/screen generator and an integrated text generator as well as version management to support the QA requirement for traceability of documents.

ASE contributed to the success of the project by significantly lowering error rate and contributing to higher user satisfaction in comparison to previous projects in the company. Gunther Knabe, the project manager of MEDIS, expressed it thus: "We have never had such a smooth project like this before. At first sight ASE seemed to require greater effort. But finally these efforts paid off" (Knabe, 1999).

## Summary and Conclusions

Software development efforts are often complicated by the fact that decisions taken early in the development process are pivotal to the eventual quality of the system. However, such decisions are often based on incomplete and insufficient information. This is related to the inadequacy of communication models to assist the capture and assimilation of information to guide the specification of requirements, an activity that is universally acknowledged as pivotal to successful outcomes. Wrong decisions from poor information typically cost organizations a great deal.

Several models have been used to assist users and developers communicate more effectively about, and specify, the requirements that are needed to produce useful and usable systems with high quality. However, despite the extensive use of these models over several years, indications are that inadequate systems, resulting from poorly specified requirements, are still very prevalent. Effective user participation in the specification phase of a software project contributes to the elimination of misunderstandings of the requirements in the earlier phases, the enhancement of quality, and saves time and money in subsequent phases. Effective QA depends on the definition of quality criteria in user language and a useful and understandable structure for depicting the specifications that can also identify deficiencies.

This chapter discussed most of the existing models that are used to promote user-developer communication during requirements specification under the umbrella of traditional (and popular) models and more recent (but less well known) ones. Some of the older models are still very useful; however, like information technology that has been coming at us at a rapid pace, we need to explore new and improved techniques to match the complexity of new systems and provide effective tools for accommodating desirable quality attributes into the process of specifying requirements. In this regard, our research presented in this chapter suggests that the set of ASE techniques has proven to be reasonably successful in circumventing some of the problems that have plagued other approaches.

Animation and virtual reality have been applied successfully in a variety of technologies to simulate actual usage experience of physical structures (for example, in computer aided design). We should make use of its capability to better incorporate user views and improve the accuracy in requirements specification. After all, most of the quality-enhancing methods in the later stages of systems development are still destined to failure if we cannot improve the quality of the specifications by affording users tools to better contribute their own information and assimilate the contributions of others in order to positively influence the completeness, consistency, clarity, accuracy, and feasibility of specifications.

The preliminary indications from our rudimentary assessment of ASE need to be followed by more rigorous research efforts to establish the limits of the efficacy of this approach and to provide insights into its strengths and weaknesses. It would be useful to conduct such studies with industrial strength applications and to apply it in combination with other techniques to evaluate whether some synergies would result from such applications. Other future research efforts could examine more sophisticated tools to enable more realistic animated simulations of IS in development and determine the effects on the quality of the deliverables they support as well as the quality of the final product.

## References

APV. (1996). *The APV guideline "computerized systems" based on Annex 11 of the EU-GMP guideline*. Retrieved December 31, 2004, from http://home.computervalidation.com

BaaN. (1999). Dynamic enterprise modeler (Version 4) [Computer software]. Hannover: BaaN AG. Retrieved from http://www.baan.com

Berger, M. (2001). Strategien für Qualität und Stabilität [Strategies for Quality and Stability]. In *SAP-Anwenderforum.* Symposium conducted at the meeting of the. (Available from 4. SAP Anwenderforum der FH NON FB Wirtschaft, Lüneburg, Germany).

Boehm, B.W., & Ross, R. (1989). Theory-W Software Project management: Principles and examples. *IEEE Transactions on Software Engineering, 17*(7), 902-916.

Booch, G. (1993). *Object-oriented analysis and design with applications* (2nd ed.). Redwood City, CA: Benjamin Cummings.

Byrd, T.A., Cossick, K.L., & Zmud, R.W. (1992). A synthesis of research on requirements analysis and knowledge acquisition techniques. *MIS Quarterly, 16*(3), 117-138.

Chen, P.P. (1976). Entity-relationship model: Towards a unified view of data. *ACM Transactions on Database Systems, 1*(1), 9-36.

Chen, P.P., & Knoell, H.D. (1991). *Der Entity-Relationship Ansatz zum logischen Systementwurf* [The entity-relationship approach to logical systems design]. Mannheim: BI.

Clemson University. (2004). Flow Charts. Retrieved November 29, 2005, from Continuous Quality Improvement (CQI) Server: http://deming.eng.clemson.edu/pub/tutorials/qctools/flowm.htm

Committee for Proprietary Medical Products. (1997). *CPMP/ICH/135/95 ICH Topic E6: Guideline for good clinical practice step 5* [Brochure]. London: European Agency for the Evaluation of Medicinal Products, Committee for Proprietary Medical Products.

Davis, A., Overmyer, S., Jordan, K., Caruso, J., Dandashi, F., Dinh, A., et al. (1997). Identifying and measuring quality in a software requirements specification. In R.H. Thayer & M. Dorfman (Eds.), *Software require-*

*ments engineering* (2nd ed.). Los Alamitos, CA: IEEE Computer Society Press.

DeMarco, T. (1979). *Structured analysis and system specification.* Englewood Cliffs, NJ: Prentice Hall.

Dijkstra, E.W. (1972). The humble programmer. *Communications of the ACM, 15*(10), 859-866.

Dorsey, P. (2003, May 20). Why large IT projects fail. Message posted to http://www.datainformationservices.com/DIS/Forum/topic.asp?TOPIC_ID=21

Gane, C., & Sarson, T. (1977). *Structured systems analysis: Tools and techniques* (1st ed.). McDonnell Douglas Information. (Reprinted from C.P. Svoboda. (1997). *Software requirements engineering* (2nd ed.). Los Alamitos, CA: IEEE Computer Society Press).

Grawe, H. (2001). Strategien für Qualität und Stabilität [Strategies for quality and stability]. In *SAP-Anwenderforum.* Symposium conducted at the meeting of the. (Available from 4. SAP Anwenderforum der FH NON FB Wirtschaft, Lüneburg, Germany).

IBM. (2004). Rational software. Retrieved November 29, 2005, from http://www-306.ibm.com/software/rational/

IDW. (2001). IDW (Ed.), *Entwurf IDW Stellungnahme zur Rechnungslegung: Grundsätze ordnungsmäßiger Buchführung bei Einsatz von Informationstechnologie* [Brochure]. Düsseldorf: IDW (Institut der Wirtschaftsprüfer).

ISO. (1994). *ISO/IEC 12119: Information technology – Software packages – Quality requirements and testing* [Brochure]. Berlin: DIN Deutsches Institut für Normung e.V.

ISO. (1995). *EN ISO 10007: 1995 quality management; Guidelines for configuration management* [Brochure]. Berlin: DIN Deutsches Institut für Normung e.V.

Jackson, M.A. (1975). *Principles of program design.* London: Elsevier.

Karlson, J. (1997). Managing software requirements using quality function deployment. *Software Quality Journal, 6,* 311-325.

Knabe, G. (1999). Personal communication. Merck KGaA, Darmstadt, Germany.

Knoell, H.D., & Knoell, G. (1998). Der Mensch im IT-Projekt [The Human in the IT project]. *IT-Management, 9,* 28-32.

Knoell, H.D., Slotos, T., & Suk, W. (1996). Quality assurance of specifications - The user's point of view. *Proceedings of the 8th International Conference on Software Engineering & Knowledge Engineering* (pp. 450-456).

Knoell, H.D., & Suk, W. (1989). A graphic language for business application systems to improve communication concerning requirements specification with the user. *Software Engineering Notes of the ACM, 14*(6), 55-60.

Kuehl, L.W. (2001). Strategien für Qualität und Stabilität [Strategies for quality and stability]. In *SAP-Anwenderforum.* Symposium conducted at the meeting of the. (Available from 4. SAP Anwenderforum der FH NON FB Wirtschaft, Lüneburg, Germany)

Levina, N., & Vaast, E. (2004). *The emergence of boundary spanning competence in practice: Implications for information systems' implementation and use.* Unpublished manuscript, New York University.

Marx, T. (1998). *NetCASE: Softwareentwurf und Workflow-Modellierung mit Petri-Netzen* [NetCASE: Software design and modelling of workflows using Petri-Nets]. Aachen: Shaker Verlag.

Metaplan. (2003). How to moderate group discussions using the Metaplan technique. In *Primer for the Metaplan Technique.* Retrieved from http://www.moderationstechnik.de/en/

Metersky, M.L. (1993). A decision-oriented approach to system design and development. *IEEE Transactions on Systems, Man, and Cybernetics, 23*(4), 1024-1037.

Neptun. (2002). Modellierung mit Neptun [Modelling using Neptun]. In Neptun (Series Ed.), Neptun (Vol. Ed.) & Neptun (Ed.), *Projekt Neptun.* Universitaet Koblenz-Landau. Retrieved December 31, 2004, from http://www.uni-koblenz.de/~ag-pn/html/projekte/nep_Modellierung.htm

Object Management Group. (2003). *OMG Unified Modeling Language Specification* [Brochure]. Needham, MA: Object Management Group, Inc.

Pahnke, C. (2003). Animated systems engineering: A new approach to high quality groupware application specification and development. *University of Wolverhampton.* (Available from University of Wolverhampton, Wolverhampton, UK).

Pahnke, C., & Knoell, H.D. (1999). Quality assurance for groupware: What makes the difference? *Proceedings of the ASME ETCE 1999: Vol. Computer in Engineering.* New York: ASME.

Pahnke, C., Knoell, H.D., & Moreton, R. (2001). ASE: A new approach for qualitative groupware development projects? *Proceedings of the ASME ETCE 2001: Vol. Computer in Engineering.* New York: ASME.

Pahnke, C., Moreton, R., & Knoell, H.D. (2004). Animated systems engineering (ASE): Evaluation of a new approach to high quality groupware application specification and development. *Proceedings of the Thirteenth Interna-*

*tional Conference on Information Systems Development: Vol. Methods and Tools, Theory and Practice,* Vilnius, Lithuania.

Paulk, M.C., Curtis, B., Chrissis, M.B., & Weber, C. (1993). *Capability maturity model for software, version 1.1.* CMU/SEI-93-TR-24, DTIC Number ADA263403, Software Engineering Institute, Pittsburgh, PA.

Pavone (2001). Process modeler (Version 5) [Computer software]. Paderborn: Pavone AG. Retrieved from http://www.pavone.com

Petri, C.A. (1962). Kommunikation mit Automation. *Diss. Univ. Bonn.* University Bonn, Bonn.

Pütz, K. (1999). "METAPLAN® moderation" as an instrument of visualisation. Retrieved November 29, 2005, from http://michotte.psy.kuleuven.ac.be/~ische/abstracts/puetz.html

Raghaven, S., Zelesnik, G., & Ford, G. (1994). *Lecture notes in requirements elicitation.* Pittsburgh: Carnegie Mellon University. (SEI-94-EM-010).

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1991). *Object-oriented modelling and design.* Englewood Cliffs, NJ: Prentice Hall.

Simon, C., Ridder, H., & Marx, T. (1997). *The Petri Net tools Neptun and Poseidon.* Koblenz, Germany: University of Koblenz-Landau.

Standish Group. (1994). *The CHAOS report* [Brochure]. Yarmouth, MA: The Standish Group International, Inc.

Svoboda, C.P. (1997). Structured analysis. In R.H. Thayer & M. Dorfman (Eds.), *Software requirements engineering* (2nd ed.). Los Alamitos, CA: IEEE Computer Society Press.

*Wikipedia*. (2004). UML. Retrieved November 29, 2005, from http://en.wikipedia.org/wiki/Unified_Modeling_Language

Yourdon, E. (1979). *Structured design: Fundamentals of a discipline of computer program and systems design.* Upper Saddle River, NJ: Prentice Hall.

Zave, P., & Jackson, M. (1997). Four dark corners of requirements engineering. *ACM Transactions on SW Engineering and Methodology, 6*(1), 1-30.

Zmud, R.W., Anthony, W.P., & Stair, R.M., Jr. (1993). The use of mental imagery to facilitate information identification in requirements analysis. *Journal of Management Information Systems, 9*(4), 175-191.

## Chapter VI

# Academic Storytelling: The Development of User-Driven Scenarios

Robert Cox, University of Tasmania, Australia

## Abstract

*This chapter introduces the concept of rich picture stories or scenarios to establish a collaborative effort in the design of new information systems. Systems designers need contextual information about their users in order to design and provide information systems that will function effectively and efficiently within those contexts. Story-telling or scenarios allow developers and consumers alike to create that all-important "meeting place" from which comes a collaborative effort in the design of new systems. One of the primary issues inhibiting the further use of storytelling or scenarios in information systems is the issue of validation, which is explored. It is hoped that this chapter will provide insights into the opportunities available to information systems users and designers to improve the quality of information systems through the use of valid user-driven scenarios.*

# Introduction

*Scenarios are alternative descriptions or stories of how the future might unfold. Scenarios are especially important where rapid change is likely to occur and where there is high uncertainty about the future. In the process, they often provoke the imagination, raise fundamental questions, make explicit our deeply held values, and stretch our worldviews.* (Institute for Alternative Futures, 2001)

Scenarios are *not* predictions of the future. Rather, in this context, they encourage people to think about how to navigate successfully through the developmental processes that lead to the creation of new, consumer-focused and user-centered information systems. Scenarios can, and should, expand our field of view, resulting in the cost-wise identification of possible opportunities and threats. They are an investment in learning.

Scenario-thinking endeavors to ensure that we are distinctively aware that there is not *a* single future coming and that we understand that *what* will eventuate *will* be shaped by what actions we perform today. The development of new information systems and their associated technology, for example, the much-anticipated video phone, have often been distracted by the anticipation of a *monetary windfall* made possible through a vast and pervasive future use for a system or technological device. The scenario process moves away from this predictive distraction in favor of clarifying the plausible and preferred future in a validated story.

Further, because the future is so elusive and can only be described in ethereal and insubstantial terms, it is useful to remember the adage of Professor Jim Dator, founder of the Alternative Futures program at the University of Hawaii: "Any useful statement about the future should seem ridiculous" (Institute for Alternative Futures, 2001).

The objectives of this chapter are to encourage readers to consider how scenarios can assist in developing the design of new information systems and to hopefully encourage researchers and designers alike, to consider the advantages available to them from the structured use of user-driven scenarios as one method of improving information systems quality.

## Background

Just about every tribe or society of people has recognized certain members of their communities for their ability as storytellers. This skillful art has been passed down from one generation to another by chosen, recognized, or hereditary storytellers. Some stories are shared at certain times of the year, in designated areas and on special occasions. As well, parents, family members, and elders share their knowledge with the younger generations in the course of their daily activities. There have been stories told by and about almost every facit of our society today: sports, business, education, work, and home life.

Stories play a very important part in our social structure. It is possible from our fairy tale and childhood stories that we learn about our environment, social norms, and expected behavior patterns, just as we learn from our employer's lunchroom edicts acceptable performance and behavior norms at our workplace. The world today is a very different place from the one in which many of us grew up. Family, church, and school are no longer the primary sources of information for life.

Haraway (1991) documents how the "informatics of domination" have shifted an "organic industrial society to a polymorphous information system" (p. 164). Further, this information system has already transformed our bodies into cyborgs — part human, part machine — at least, part telephone. The pervasiveness of technology in our everyday lives has come about, it would appear, so quickly that it has caught a lot of us off guard. We are now inundated with an expanding range of multimodal digital communication systems; wrist watches that have a multitude of superfluous functions; even our domestic appliances have more processing power than the Apollo mission to the moon (Thompson, 2004). Yet, it would appear that most technology clients have had little or no input into the technology that they so pervasively use! This non-involvement has contributed greatly to ineffective user-developer communication and eventually low quality information systems products (Barki & Hartwick, 1994).

This brings about the question of how a technology consumer can converse constructively with a technologist. It is not possible for the technology consumer to instantly gain the technological experience necessary with which to offer positive design directions to the technologist, just as it is not possible for the technologist to *unlearn* the knowledge gained over many years of research and development, so both parties can find the commonalities necessary for human communication. Therein lies the problem: the lack of a common meeting place from which mutual ideas may flow.

The following excerpt from Thomas Erickson (1996) of Apple Computer Inc., writing in his paper "Design as Storytelling", does an excellent job of introducing the scenario or storytelling process:

*Storytelling is an integral part of my approach to design, both at the earliest stages, as well as later on in the process. I have a repertoire of stories that I've built up over my years of design practice. I collect them. I write them.*

*I tell them. I piece together new stories out of bits of old ones. I make up stories about my own observations and experiences. I use these stories to generate discussion, to inform, to persuade. I notice that other designers have their own collections of stories as well, and often use them in similar ways. Stories are very powerful tools.* (p. 3)

John Seely Brown, Chief Scientist of Xerox and co-author of *The Social Life of Information*, offers a further scenario or miniature story to help communicate what a different future, with an intelligent use of computers, would be like:

*I could create an envisioning of the year 2020 ... that just felt like ... sitting on my porch at home, able to be on the borderline between public and private. Being able to see what's happening around me, being able to interact with my neighbours, with my community when I want. Being able to sit back and read quietly, sense the tranquility yet connectedness, the ability to get perspective on big problems and yet be a part of the local scene.* (Seely Brown & Duguid, 2000, p. 3)

And yet another author, Steve Denning (2000), asks the question: why is there resistance to the role of storytelling/scenarios? (Denning, 2000). He then postulates that academics sometimes suggest that research, using storytelling/scenarios, may drag the world back into the Dark Ages of myth and fable from which science has only recently extricated the masses. He further lists the antecedents to science's current hostility to storytelling/scenarios and proposes that:

- Plato had a hand in whipping up the hostility to storytelling, although his own masterful practice of storytelling undercuts the explicit arguments that he makes in *The Republic* (Plato, c.375 BCE);
- Writers like Francis Bacon (1561-1621) were sometimes seen as opposing scientific experimentation as mere narrative or anecdote, often without the recognition that the results of scientific experiments could only be described in a narrative;
- Antagonism toward storytelling may have reached a peak in the twentieth century with the determined effort to reduce all knowledge to analytic propositions, and ultimately physics or mathematics; and
- Logical positivists vainly struggled to construct a clockwork model of the universe, in which only observable phenomena were conceded to exist, and the universe was thought of as comprising a set of atoms randomly bouncing

> off each other, and human thoughts and feelings were essentially non-existent.

Albeit on the positive side, the academic pursuit of literary studies took narrative or storytelling seriously, but with the rise of science they were considered to be very much on the intellectual periphery, or "the seamy slum of modern academia" (Denning, 2000).

Even further, the postmodernist critique of Jean-Francois Lyotard (1984) cites, "science itself rests on a foundation of narrative [that] was intellectually impeccable" even though Lyotard's own theories predicted that it would be "systematically ignored by the high priests of science — the technologists" (Lyotard, 1984, p. 2).

The main problem, as I see it, rests on an inability of the *learned* to reach a mutual and constructive meeting place with the *unlearned* and vice versa. So how does one communicate between the *lofty heights* of information systems research and the perceived *lowly slum* of consumer machinations? The answer may belong in the realm of the user-driven scenario. Unfortunately, this *common meeting place* does not come without its fair share of problems and concerns, both for the consumer and the technologist.

## Issues and Problems

### Validation

One of the primary issues debilitating the further use of storytelling or scenario building is the issue of validation. It is rarely considered that an untrained or unskilled and possibly uneducated consumer could have anything to offer designers or researchers in the way of technological direction. Yet most of our recent technological advances have come from such people; for example, Jonathon Barouch, the Sydney schoolboy, now-millionaire, creator of an online e-commerce site. This is not to say that designers and academic researchers should not be prudent and thorough in their approach to validating stories or scenarios, but just how does one validate a series of hypothetical events utilizing a postulated technological device?

Van Krieken et al. (2000), in their book *Sociology: Themes and Perspectives*, cite that "data are valid if they provide a true picture of what is being studied. A valid statement gives a true measurement or description of what it claims to measure or describe" (van Krieken et al., 2000, p. 241). One should expect that any given scenario's operational environment would contain elements of *natural* validity — for example, being grounded by a geographical location, space, and time. Further, any postulated technological device must remain, to some extent, within a plausible technological realm. Creating fantastic environments and ethereal technological vistas does nothing to communicate need or desire for a device that will alleviate consumer problems, existing or otherwise!

Pickard and Dixon (2004) cite that there is a need to differentiate between quantitative and qualitative research, particularly when determining the validity of research. They cite that quantitative research's validity is based upon four established criteria:

1. Truth value, established by internal validity;
2. Applicability (or generalizability), determined by external validity;
3. Consistency, determined by reliability; and
4. Neutrality, determined by objectivity. (p. 6)

And, these four criteria "make it impossible to judge one [qualitative] by a criteria established to judge the rigour of the other [quantitative], accommodation between and among the paradigms on axiomatic grounds is simply not possible" (Pickard & Dixon, 2004, p. 9), thus the need for the development of a further series of axiomatic criteria with which to judge qualitative research.

Further, experimental studies are often criticized on the grounds that conclusions based on experimental data cannot be applied to settings outside the laboratory because the laboratory is too unlike the real world. For this reason, "research is found wanting because it lacks ecological validity" (Neisser, 1976, p. 354). Newman identifies ecological validity as "the degree to which the social world described by the researcher matches the world of [its] members" (Neuman, 1991, p. 166).

In support of this criticism is the contention that "human behaviour outside the laboratory cannot be predicted from what has been learned in the laboratory" (Hopkins, 2000). Further, Neisser (1976) identifies that observations made in the laboratory have to (or are meant to) mimic the everyday life phenomena. However, Mook (1983) counter-argues that:

> *This criticism of the experimental approach [to research] is incorrect because experimental studies are concerned with* what can happen *not* what does happen. *Nor do experimental conclusions serve to predict a function. Instead, experimental conclusions come about as a result of predicting from theories.* (p. 98)

But where do these theories come from in the first instance? Quite possibly from informal scenarios.

> *I suspect stories, in particular, haven't been discussed much because they aren't … well, stories aren't very respectable. Stories are subjective. Stories are ambiguous. Stories are particular. They are at odds with the scientific drive towards objective, generalisable, repeatable findings. In spite of this — or, I will suggest, in part because of this — stories are of great value to the interaction designer.* (Erickson, 1996, p. 3)

Observations and consumer directives, collected in their natural settings and subsequently set down in narrative text, can be what are needed to test a theoretical information system design, to contain all the validity required and to offer positive direction for both system designers and consumers.

## Scenario Focus

The focus of scenarios has, until recently, belonged primarily to technologists who used them to communicate device-operating parameters and, to some extent, their creations' marketability to industry and research stakeholders. This same process is now being used by consumers to communicate with systems designers and technologists on *what* they want a system/device to do, rather than designers communicating with consumers on *how* they should use their new system/device.

Central to the theme of these new-style scenarios should be the idea of User-Centered Design (UCD). UCD, as expounded by authors such as Michael Dertouzos (2001) and Karl Vredenberg et al. (2002), see the process of technology design as changing from technologist driven to that of solving identifiable consumer problems. Hence systems change from *technical wonders* to *solution providers*.

Carroll and Rosson (1990) give examples of the focus of scenarios through five different criteria:

1. An individual consumer or persona;
2. Using a specific set of computer facilities or activities;
3. To achieve a specific outcome;
4. Under specified circumstances;

5. Over a certain time interval (explicitly includes a time dimension of what happens when).

Importantly, the focus is on the consumers understanding and input into the design process rather than the technological aspects of the design. This is because consumers readily find it easier to relate to the task-orientated nature of the scenario than to the abstract, and often function-orientated, nature of systems specifications (Nielsen, 1993).

# Solutions and Recommendations

## Validation

It would appear that scenarios suffer from some of the same issues that qualitative research has for the past few decades. Doug Ezzy (2002), in his book on qualitative research, identifies that:

> *Rigour in qualitative research is as much situated and linked to the politics and particularities of the research as it is to following methods and practices .... Rigour and ethics were originally considered to be quite separate ... [however], it has become increasingly clear that all knowledge is inextricably moral with ethical implications.* (p. 52)

Certainly, validity refers to whether an information system is internally coherent, and to whether it accurately reflects this external world. However, scenarios are not objective entities but rather subjective. Their creation has been influenced by the values, interests, and hopes of the researchers and contains the rich dialogue of hopes and desires for the postulated system and devices, the answers to the consumer problems, and the design direction of the technologist.

The idea that theory should accurately reflect the experiences of the system consumer is clearly problematic if there is not one clear meaning of that technological experience. One of the main purposes of scenario use is to obtain a consensus of experience between the consumer and the technologist. The creation of scenarios that contain within them all the consumer's desire and intent about the postulated system, its technology, and its hypothetical operating environment would seem to fulfil the ecological validity requirement established by Neisser, thereby answering to some extent the scenario's validation requirement.

Further, Pickard and Dixon (2004) suggest that a new series of four criteria should be established to determine the validity of qualitative research, particularly constructivist inquiry, which encompasses the data gathering exercise of scenarios. These might be:

1. **Credibility** – established by prolonged engagement with the research participants, persistent observation of those participants, triangulation of the techniques used to study those participants and their contexts, peer debriefings, and member checks (Pickard & Dixon, 2004, p. 6).
2. **Transferability** – to allow the transferability of the research findings rather than wholesale generalization of those findings, as established by Dervin (1996), such that if the research reader identifies similarities in the research, then it is reasonable for that reader to apply the research findings to his/her own context (Pickard & Dixon, 2004, p. 6).
3. **Dependability** – established by the inquiry audit and external auditor who is asked to examine the research process, the way in which the research was carried out, examining the data produced by the research in terms of accuracy relating to transcripts and levels of saturation in document collection (Pickard & Dixon, 2004, p. 6).
4. **Confirmability** – established through ensuring that the results, accepted as subjective knowledge of the researcher, can be traced back to the raw data of the research, that they are not merely a product of the research interests (Pickard & Dixon, 2004, p. 6).

Abstract models, as scenarios are frequently qualified, often do not provide sufficient information or insights upon which to base the planning and delivery of service or tools. However, if modeling or scenarios are to be established as a desired result of qualitative research, then scenarios must be validated and placed in context. From these scenarios, themes may be established, and these themes would become all the more relevant for having emerged from diverse and often conflicting realities.

## Scenario Methodology

The development of user-centric scenarios is an individual process, mostly unique to each and every scenario built. Although there are some steps that will be ubiquitous to all scenarios created (i.e., development of the initial scenario draft) most steps will involve consultation and group work with a core of industry-specific consumers whose input will not overlap to other scenarios.

The initial scenario idea develops from observations or consultations with either an individual “expert” in a specific industry-sector or as a result of an expert panel meeting. These panel meetings are convened for the specific purpose of obtaining directives and observations from consumers from within. Further clarifying information is then obtained by returning to specific experts from within the panel and verifying that the scenario draft fulfils the intended consumer activities and outcomes.

The draft scenario is then brought before a focus group (Caplan, 1990; Goldman & McDonald, 1987; Greenbaum, 1993; O’Donnell et al., 1991) where six to nine consumers are brought together to discuss the scenario concepts and identify and clarify the scenario’s plausibility and facilitate technology over a period of about two hours. Each group is run by a moderator who is responsible for maintaining the focus of the group. “From the consumers-perspective, the focus group session should feel free-flowing and relatively unstructured, but in reality, the moderator has to follow a pre-planned script for what issues to bring up” (Nielsen, 1993, p. 54).

Once “approved” by the focus group, the scenario is free to be passed onto the technologists for their input. This process should be iterative, with the scenario moving freely between consumer groups and technologists until such a time when they both agree on its substance, at which times it may turn into a demonstrator proposal.

The use of a scenario template may introduce a needed formality into scenario creation. The suggested format uses Carroll and Rosson’s (1990) five-scenario criteria to describe the postulated device and hypothetical usage: persona, device, outcome/performance, situation/circumstances, and time considerations.

## Example Scenario

The format of this scenario is based on Maurice Castro’s (2002) *Exploring Location* paper and introduces a persona involved in a profession which is potentially hostile to computing technology and sensitive equipment, in general. This persona is used to explore the concepts of practicality of use of communication devices in agricultural based occupations. The idea of location is also explored, as previously identified by Castro.

A persona has been constructed within the agricultural environment. The manufacturing, building, light and heavy industrial environmental sectors would also have similar problems with work day hazards and equipment robustness. The agricultural environment is particularly hazardous as exposure to bright sunlight, water, chemicals, dust, and lengthy periods of vibration from farm equipment are normal daily activities.

Like all personas, Graeme does not represent a real person. Instead, he is a composite of many individuals, representative of this industry sector. The persona has sufficient detail and depth for a developer to relate to, as if Graeme were a human being.

## *PERSONA — Graeme*

Graeme is a third generation farmer. He is married with three children — two boys and one girl who has an intellectual disability. Graeme gained his farming experience from his father and a farm apprenticeship, completed with formal agricultural college training. He has a flare for mechanics and farm equipment, especially irrigation equipment.

Graeme is a good farmer. He plants and harvests differing crops and runs a small herd, about 150 head, of beef cattle. He works long hours for small return but considers the lifestyle to be the most important reason to continue farming.

His farm is spread over a large geographical area and on differing sites. He grows on contract for several different companies, both local and international. Graeme grows small niche crops "on spec" as a continuation of the farming tradition set down by his grandfather. His wife keeps the farm records on a home-based computer. Both he and his wife enjoy "surfing the net" when they can, as their children tend to monopolize the computer.

The farm employs two workers, one son, and one farmhand from a local family. Graeme is the only irrigation service person in the district, so he can get quite busy at times. He has a limited supply of irrigator spare parts in his shed and often needs to order special parts. Ensuring the right crop is planted, watered, fertilized, harvested, and sold is central to his farm's success, just as fulfilling irrigator repair requests is essential to keeping a cash flow for the family.

Graeme is often frustrated by a lack of current prices for irrigation spare parts and equipment, current crop seed, and commodity prices. Further, his family needs to know where he is at all times, as his daughter has high needs and they need security from the possibility of farm machinery breakdown/accident in an outlying paddock.

Some of the problems that he would like to see solved are to have a better form of contact than second-hand information relayed by radio from home; often missing irrigation repair requests and needing to recontact the complainant for further details, a cost in both time and dollars; needing to lower his farm operational expenditure, especially fuel consumption; and he would like a more accurate financial overview of his farming operations.

## Device: Farmers Right Hand

For purposes of this scenario we will postulate the existence of a device known as the *FRH – Farmers Right Hand*. This device is a palm-based computer that is rugged, waterproof, acid proof, has a specific gravity of less than 1 (so that it floats on fresh water) and an effective battery life of 100 hours. The device is sized so that it will fit comfortably into the breast pocket of a King-Gee® work shirt. The device can be operated inside the shirt pocket but will also clip onto a belt. The device incorporates mobile telephone facilities, wireless networking, speaker and microphone, and a touch screen. Input is provided by either a virtual keyboard on the touch screen, by individualized "icon-based" shortcuts on the touch screen or by a T9-like abbreviated word completion system.

Farms are an equipment hostile environment. Equipment is subject to:

- **Physical shocks** – equipment may be dropped, stood on, or otherwise physically assaulted in day-to-day operation.
- **Loss** – equipment may be placed down and forgotten, especially when a situation demands urgent attention (i.e., a cow calving).
- **Incomplete telecommunications** – farms are often geographically large. Access to network facilities may not be possible in all farm locations, at all times.
- **Interference** – the proximity of motorized farming equipment (i.e., tractors, irrigation pumps) can cause interference with telecommunications.
- **Multiple users** – equipment may be used by more than one person on several occasions, requiring several consumer profiles to be stored and accessible.

## Performance

### *Location*

This scenario uses location and a geographical position. Definition to within 3 meters (9.75 feet) is sufficiently accurate for this scenario's applications. Subscenario's 5.2 through 5.5 all utilize a GPS (Geographical Positioning System). Code phase differential GPS is the most common technique to gain better accuracy when using a GPS receiver. The accuracy is approximately 3 meters with a Code Phase GPS receiver such as one used for recreation purposes and 1 meter or less with a GPS receiver with carrier phase smoothing.[1] In this scenario, location is used to influence the behaviour of the device. The preferred options of the location for the device should be able to be configured by the consumer — with the consumer being able to choose several options of locatability (i.e., by paddock; by grid reference; by map).

### *Privacy*

There are going to be occasions when Graeme may not want his location to be revealed either to his family or to business/farming associates. However, he would want to give accurate information about the time it will take for him to get somewhere, and if he is in an unfamiliar location or environment, then navigational assistance would be appreciated. When Graeme decides to make his location available should be decided by Graeme only. Subscenario 5.4 makes allusion to the problems of open locatability.

### *Identity*

A device belonging to Graeme, who is in an occupation that is prone to severe accidents, should be able to identify its consumer. The portability of this device will need a process of consumer identification such as a consumer name/password operation — especially for the T9 input methodology.

### *Accessibility*

Probably more tied into the size and robustness attributes, accessibility of the device should not be limited to one person. However, the portability of the device will be limited by the T9 voice input training — seeing it is specific to a single consumer. The incorporation of a textual/graphical consumer interface will assist in the devices portability, as will a generic voice-recognition interface.

### *Size and Robustness*

The size of the device is of special importance in this scenario. A device that is large or cumbersome is more likely not to be used or damaged. Ensuring it fits into the pocket of a standard work shirt would ensure its utilization and, to some extent, its protection from casual damage.

## *Communication*

Certainly one of the primary activities for any mobile device is communication. Whether it be mobile telephony and Internet connectivity, roaming or mobile communication is what gives the device its power and usability. The quality of communication is certainly an issue, especially small devices, as they would usually necessitate small-sized power units with small power outputs reducing the effective communicable range. Graeme needs the device to be able to locate and use effective communications.

## *Computing/Calculation*

The ability to action prewritten software in a range of software languages and possibly operating systems would certainly enhance the devices portability. It may be possible to create a specific operating system by downsizing an existing OS (i.e., Linux or Mac OS10) to fit. Certainly, the computational power of the device will need to be high (i.e., Pentium 4 – 2Ghz+) to ensure that the communication, GPS, and T9 voice systems operate quickly and effectively, with little delay. Graeme would not appreciate having to wait for a lengthy period of time for the calculations/communications/GPS functions to be displayed or actioned.

## *Data Storage*

Effective and efficient data storage is a further issue for device operation and size considerations. Housing sufficient internal memory size-wise is always a design predicament. Further, the type of memory is also an important consideration — does the memory erase when the power source ceases, or are certain sections of the inbuilt memory able to be permanently stored (i.e., setup configurations, downloaded GPS maps). The access to the device’s memory should also be flexible with input through an inbuilt USB port from home/business computers and possibly external memory devices such as CD-Rom and floppy disk.

# Situation

## Refueling: A Costly Exercise

Its 5:30 A.M. on what's purported to be a fine day, and Graeme is starting his daily routine. Today, according to his schedule, he needs to check on the condition of the crops sown into the farms outlying paddocks; ensure that the irrigator has run its interval and irrigated the potato crop successfully; ear tag a couple of dozen newly acquired beef calves; fix the neighboring farmer's traveling irrigator; and finally pick his younger son and daughter up from the temporary school bus stop 10 km up the road. This will necessitate the use of several pieces of farm equipment, namely the 4WD tractor and the farm Ute.

Graeme busily fills up the 4WD tractor with diesel fuel from the farms supply tank, making a note into the FRH of the quantity of fuel issued through a shortcut on the touch panel. The FRH screen identifies the piece of farm equipment, requests an edit format — fuel update — and stores the keyed information for later reporting. The FRH also displays an estimate of the amount of operating time this refueling will enable on this particular machine and requests that an alarm be sounded when the remaining time is at or below 10%. With the flick of a key, the tractor and Graeme, are off to investigate the first of the days duties — an appraisal of the outlying paddocks. The journey will take approximately 20 minutes for Graeme to reach the first of the paddocks.

## A Crop to Harvest

Graeme stops the tractor at the gate of what his family has always identified as the River Paddock — a paddock of feed sorghum, approximately 40 Ha in size, with the river as the boundary fence running down one side. Graeme casts his eyes across the paddock and, after a short walk through the immediate area, forms the opinion that this crop of sorghum is ready for harvest. He retrieves the FRH from his shirt pocket and asks:

*"Can you tell me what the weather forecast is for next 3 days in this area?"*

The FRH, using the in-built GPS identifies Graeme's location and then sends a request via mobile telephony to the Bureau of Meteorology for the weather forecast for the requested area for the next 3 days. The FRH responds with a graphical display on the touch screen showing that the forecast for the next 3 days is going to be fine.

Graeme's farming experience ensures him that this crop can be left for up to another week in the paddock without any detriment. The primary factor for Graeme on whether to harvest or not is the price he will receive from the local commodity buyers. Graeme now asks the FRH:

*"Can you tell me what the harvested purchase price for sorghum is at present, in this area?"*

The FRH processes the request and, using mobile telephony again, contacts the commodity buyers in Graeme's district. The FRH responds with a graphical display on the touch screen identifying the current purchase price for sorghum from the two local buyers, Webster and Pivot.

As this information is very locally based and can only be appreciated in context with the current commodity price for the crop, Graeme further asks the FRH:

*"Can you tell me what the commodities market value is for export sorghum at present?"*

The FRH again processes the request and, using mobile telephony, contacts the ASX (Australian Stock Exchange) and requests the commodity market price for export sorghum. The FRH displays the information both as a textual message and a graph, indicating the previous 2 weeks' prices as a trend line. The trend is currently steady.

Satisfied that he will gain an equitable return, Graeme decides to harvest the crop now. This will necessitate the booking of a contract harvesting business. Graeme now asks the FRH:

*"Can you tell me if Harrison's Harvesting has a team available in the next 3 days?"*

The FRH identifies the request and, using the harvesting companies telephone numbers stored in its mobile telephony system, contacting not only Harrison's Harvesting but also two contract harvesting companies for harvesting team availability in the next 3 days. The FRH then sends details of the paddock size, location, crop, and anticipated harvesting date to the contractors. The FRH displays the results of the search as a pick list of companies contacted and their availability on the touch screen. Unfortunately, there are no teams available to harvest this crop in the next 3 days. However, the FRH identifies that Harrison's has a team available from the fifth day on.

## *How Much Water?*

Having successfully negotiated the harvesting of the sorghum crop in the River Paddock and completing an inspection of the remaining outlying farm paddocks, Graeme now attends to the irrigation of the potato crop.

As Graeme arrives at the paddock gate, he notices that the traveling irrigator has stopped about half way down the paddock. A problem has been encountered, and the irrigator needs to be engaged to irrigate the remaining paddock. This particular paddock draws water from a common source, with each farmer allocated a certain day (and even a certain time period on that day) on which to draw water for irrigation. Therefore, it is important for Graeme to know how much water has been pumped onto the paddock, how long the irrigator ran for, and how long the irrigator will need to operate to cover the remaining paddock area. Graeme asks the FRH:

*"Can you calculate for me the running time of the irrigator in this paddock?"*

The FRH, utilizing the in-built GPS, locates the irrigators identifying signal which places it in a known farm paddock. The FRH uses given formulae to calculate how far the irrigator has traveled and therefore how long it operated.

The final resultant — 4 hours — is displayed as a figure on the touch screen. Graeme stores this figure into the working memory of the FRH using a shortcut on the touch screen. Knowing that the amount of water placed onto the paddock will have decided effects on the growth potential and the damage received to the crop, Graeme now asks the FRH:

*"Can you calculate for me the amount of water that was placed onto this paddock during an operating time of 4 hours of this irrigator?"*

The FRH uses a series of predetermined formulae to calculate the result which is displayed onto the touch screen as text — 1.5cm/sq m. This figure is also stored into the working memory of the FRH. To ensure an even growth of the crop, Graeme must ensure that the remainder of the crop is irrigated at the same rate as the previously irrigated section. Graeme now asks the FRH:

*"Can you calculate how long this irrigator will need to operate to ensure that 1.5cm/sq m of water is applied to the remainder of this crop?"*

Again, the FRH uses preprogrammed calculations to determine that 2.2 hours of irrigation will be needed to ensure equal irrigation on the remaining crop. This figure is also displayed as text on the touch screen. Graeme also needs to know whether his access to the common water supply has expired for the day and so again asks the FRH:

*"Can you tell me when access to this paddock's water supply will end?"*

The FRH uses mobile telephony to contact his home computer where details of the paddock's location are matched against his farm records which hold the water access information. The FRH displays the results as text on the touch screen — 4:30 P.M. today — a remainder of 3 hours. Graeme is free to continue the irrigation of this paddock.

### *An Urgent Request*

Having set and restarted the traveling irrigator, Graeme now sets off to return the tractor to the farm shed. He arrives and notes the machines operational hours into the FRH. The FRH screen identifies the piece of farm equipment, requests an edit format — hours update — and stores the keyed information for later reporting.

Stepping across the shed, he hops into the farm Ute and heads off for the farms cattle yards, where he hopes his two farmhands are awaiting his arrival. On the way the FRH plays an audible alarm, which indicates that an incoming message has arrived. Graeme asks the FRH:

*"Can you tell me the new message?"*

The message is from his farmhand son, Mark, and the FRH audibly sounds:

*"Dad, I need you down here fast. A calf has been jammed against the yard fence and been stabbed in the belly by a loose dropper. It doesn't look very good. Whatever you're doing, you'd better leave it and come here quickly."*

Graeme not only receives the important information but also hears the small panic in his son's voice, which communicates even more efficiently the urgency of the situation. Graeme heads off hastily toward the cattle yards. As a consequence of the message operation, the FRH asks whether a reply to the message is to be sent. Graeme confirms that a reply will be sent and speaks the following reply to the FRH, while it is still located in his breast pocket:

*"Son, I'm on my way now. Be there soon."*

As an adjunct to the reply, Graeme asks the FRH:

*"Can you advise Mark of my location and ETA?"*

The FRH identifies the request and, using the in-built GPS, locates Graeme's current position and the position of the farm's cattle yards. Using a series of preprogrammed calculations, the FRH constructs an ETA and advises Mark's FRH of Graeme's ETA.

Upon arrival Graeme notices that Mark and the farmhand have isolated the injured calf in the stockyard. After a brief appraisal of the situation, Graeme decides he needs to contact the vet, so he asks the FRH:

*"Can you phone the local vet?"*

The FRH again identifies the request and, using mobile telephony, connects the FRH to the local vet so that Graeme can explain the situation in more detail. The vet advises that he is attending another farmer in an outlying district and will likely be another 3 hours before he could be there; however, he has sent Graeme's FRH some suggestions on how they should proceed. The FRH sounds an alarm that a message has been received. Graeme thanks the vet, requesting that he come when he can and terminates the telephone connection.

## Conclusion

Information systems development is, by its very nature, a collaborative work. The development of scenarios is one means by which users can give acceptable and valid input into the design of information systems without the need for lengthy induction programs and information sessions, thereby contributing to the quality of deliverables.

Scenarios also provide two cost-effective uses: First, scenarios can be used during the design of a human interface (HCI) as a way of expressing and promoting understanding of how consumers will *eventually* interact with the future system, as identified in the above scenario example. Second, scenarios can be used during the early evaluation of the HCI design to obtain *consumer feedback* without the expense of a running prototype. The scenario example above gives both the farmers and system designers/technologists opportunities to provide feedback on the style, frequency, and possibility of the proposed system and device interactions.

Donald Schön (1987), a leading HCI designer, argues that professional developers — a category within which he includes HCI designers — employ artistry as much as science in their daily practice. He notes that developers are not initially presented with well-formed problems, but rather with messy, indeterminate situations. While Schön's investigations focus on work by only one or two designers, interaction design involves many people from widely varying backgrounds.

For the new emerging Internet developer, scenarios or stories provide one way to deal with the lack of structure in problems. When first starting out, scenarios provide a way of *getting a feel for the new terrain*. Scenarios reveal a *consumer-eye view* of the landscape and provide an extremely effective way for getting people — both consumers and systems designers — involved and talking with one another, that mystical "meeting place."

"A key element of the power of stories, I believe, is their informality," cites Erickson (1996, p. 4). Like the first rough sketches of an architect, scenarios have a sort of redundancy. We all know that scenarios are subjective, exaggerated, and unique, and any scenario is liable to questioning, reinterpretation, or rejection. At the same time, scenarios provide concrete examples that people from vastly different backgrounds can relate to. "Thus, as a collection of stories [scenarios] accretes around a project, and provide a common ground, a body of knowledge which can be added to, and questioned by, all participants" (Erickson, 1996, p. 4).

Establishing that scenarios are validated, rich sources of insights to assist in the information transferral of important consumer data to systems designers and technologists is one thing; using them is another.

## Acknowledgments

The author would like to acknowledge the financial support of the Smart Internet Technology CRC.

## References

Barki, H., & Hartwick, J. (1994). Measuring user participation, user involvement, and user attitude. *MIS Quarterly, 18*(1), 59-82.

Caplan, S. (1990). *Using focus groups methodology for ergonomic design. Ergonomics, 33*(5), 527-533.

Carroll, J.M., & Rosson, M.B. (1990, January). Human-computer interaction scenarios as a design representation. *Proceedings of the IEEE HICSS-23, 23rd Hawaii International Conference Systems Services*, Hawaii (pp. 555-561).

Castro, M. (2002). *Exploring location*. Melborne, VIC: Software Engineering Research Centre.

Denning, S. (2000). *The springboard: How storytelling ignites action in knowledge-era organizations.* Boston: Butterworth-Heinemann.

Dertouzos, M.L. (2001). *The unfinished revolution: Human-centered computers and what they can do for us.* New York: HarperCollins Publishers.

Dervin, B. (1996). Given a context by any other name: Methodological tools for taming the unruly beast. In P. Vakkari, R. Savolainen, & B. Dervin (Eds.), *Proceedings of the International Conference on Research in Information Needs, 13*(8), Tampere, Finland.

Erickson, T. (1996, July/August). *Design as storytelling. Interactions, 3*(4), 37-58.

Ezzy, D. (2002). *Qualitative analysis: Practice and innovation.* Crows Nest, NSW: Allen and Unwin.

Goldman, A.E., & McDonald, S.S. (1987). *The group depth interview:Principles and practice.* Englewood Cliffs, NJ: Prentice Hall.

Greenbaum, T.L. (1993). *The handbook for focus group research.* New York: Lexington Books.

Haraway, D. (1991). *Simians, cyborgs, and women: The reinvention of nature.* New York: Routledge, Chapman & Hall.

Hopkins, W.G. (2000). *Quantitative research design.* University of Otago, Dunedin, New Zealand, Department of Physiology and School of Physical Education.

Institute for Alternative Futures. (2001). Leading in the discovery of preferred futures. Retrieved November 29, 2005, from http://www.altfutures.com/pubs/publications.htm

Lyotard, J. (1984). *The postmodern condition: A report on knowledge in* Theory and History of Literature. Manchester, UK: Manchester University.

Mook, D.G. (1983). In defence of external invalidity. *American Psychologist, 38*, 379-387.

Navaids. (2001). *Code Phase Differential GPS*. Retrieved November 29, 2005, from http://members.iinet.net.au/~navaids1/gpsinfo.htm

Neisser, U. (1976). *Cognition and reality,* San Francisco: Freeman.

Neuman, W.L. (1991). *Social research methods: Qualitative and quantitative approaches.* Boston: Allyn and Bacon.

Nielsen, J. (1993). *Usability engineering.* Chestnut Hill, MA: AP Professional.

O'Donnell, P.J., Scobie, G., & Baxter, I. (1991). The use of focus groups as an evaluation technique in HCI. In D. Diaper & N. Hammond (Eds.), *People and computers VI* (pp. 211-224). Cambridge, UK: Cambridge University Press.

Pickard, A., & Dixon, P. (2004, April). The applicability of constructivist user studies: How can constructivist inquiry inform service providers and systems designers? *Information Research, 9*(3), 2-12.

Plato. (c.375 BCE). *Republic.* Cambridge, MA: Harvard University Press.

Schön, D.A. (1987). *Educating the reflective practitioner.* San Francisco: Jossey-Bass.

Seely Brown, J., & Duguid, P. (2000). *The social life of information.* Boston: Harvard Business School Press.

Thompson, J. (2004). Computing power of Apollo 11. Retrieved November 29, 2005, from http://www.jimthompson.net/palmpda/Silly/power.htm

van Krieken, R., Smith, P., Habibis, D., McDonald, K., Haralambos, M., & Holborn, M. (2000). *Sociology: Themes and perspectives.* Frenchs Forest, Australia: Pearson Education Unit.

Vredenberg, K., Isensee, S., & Righi, C. (2002). *User centred design: An integrated approach.* Upper Saddle River, NJ: Prentice Hall.

## Endnote

[1] Navaids. (2001). *Code Phase Differential GPS*, http://members.iinet.net.au/~navaids1/gpsinfo.htm.

# Section III: Process Contribution to IS Quality

## Chapter VII

# Process-Centered Contributions to Information Systems Quality

Evan W. Duggan, University of Alabama, USA

Richard Gibson, American University, USA

## Abstract

*The growing attendance at seminars and conferences dedicated to quality programs attests to the increasing recognition of the continued importance of quality. Unfortunately, in many organizations, this intensified quality focus has not been effectively applied to information systems — a surprising outcome given the many demonstrations of a direct relationship between information systems delivery process and information systems quality and success. In this chapter, we analyze process-centered contributions and solutions to the increasing challenges of producing high-quality systems. We provide a balanced overview of evidence that has emerged from practical, real-world experiences and empirical research studies, an overview that incorporates the positions of both proponents and opponents of process-centricity. We then provide an assessment of the contexts in which software process improvements and quality- enhancing initiatives can thrive.*

# Introduction

Some organizations have managed to deliver successful information systems (IS); however, a large number have not obtained the expected benefits from their investments in software products (Banker et al., 1998). Yet, organizational reliance on information technology (IT) to support business processes and enable strategic priorities has increased conspicuously over the last two decades. This increased dependence has accentuated the challenge to IS developers, users, and managers to interact more effectively with each other (Vessey & Conger, 1994) in order to produce more sophisticated, more complex, and more reliable IS.

A central premise of most continuous process improvement philosophies is that the quality of the product and/or service is largely determined by the quality of the process used to produce it (Deming, 1986). This assumption has been substantiated in IS organizations (Harter et al., 1998; Khalifa & Verner, 2000; Ravichandran & Rai, 1996, 2000). It is noteworthy that IS are rarely confined to single processes; they usually support organizational networks of interdependent business processes and multiple interfaces to other systems. To address these and other complexities, the role of IS practitioners continues to evolve and increase in importance and with it the expectation that IS practice must continuously improve.

Accepting this premise places a considerable burden on IS quality management groups to support various business process owners and other interested parties. These stakeholders often have different business cycles and distinct quality, innovation, discipline, rigor, and productivity requirements. Consequently, in order to increase competence and improve process quality, IS organizations are forced to focus more intensely on implementing process management and improvement programs (similar to efforts in manufacturing and other service organizations). This focus typically involves two major steps: (1) objectively assessing the effectiveness (capability) and discipline (maturity) of the IS delivery process and (2) launching process improvement initiatives guided by the results of the assessment (*CIO Magazine*, 2001).

To complete the assessment, organizations typically use formal process analysis instruments such as the capability maturity model (CMM). The result of this examination then provides a baseline for planning IS process improvements and enhancing capability (Duggan, 2004). The vehicle used for initiating process improvement activities, in many cases, is the institutionalization of a highly structured systems development methodology (SDM) (Riemenschneider et al., 2002; Roberts et al., 1997). SDMs standardize an organization's IS process in order to effect the consistency and predictability needed to obtain successful outcomes that do not rely significantly on epic individual and project group performances (Riemenschneider et al., 2002).

In this chapter we analyze the contributions of these practices to the delivery of quality IS. We present a broad overview of the history of, concepts surrounding, rationale for, and results of IS process-centered interventions. The discussion represents a synthesis of core facts from real-world experiences, practitioner reports, anecdotes from current practice, insights gleaned from scholarly research, and our combined academic/professional backgrounds. The intention is to contribute to the assimilation of already accumulated knowledge about IS process improvement both as a general exposition of perspectives and issues and as a foundation for deeper analyses in subsequent chapters of this book. The ultimate objective is to make a meaningful contribution to the on-going debate about the relevance and potential of process-centricity in the delivery of high-quality IS.

In the following sections we provide background information to outline the history of IS delivery process structures, and we discuss the evolution of varying perspectives on IS process management principles and methods. Then we analyze the conceptual and practical justification for our focus on this phenomenon. We identify successful software process improvement initiatives, claimed benefits, and some implementation problems encountered and offer an explanation for differences in perspectives. Finally, we provide a summary and conclusion that underscore the salient elements of our analysis.

While we recognize the differences between IS development and software engineering, for convenience, we use the terms *IS process improvement* (our preferred term) and the more widely used *software process improvement (SPI)* interchangeably. Additionally, we have avoided reference to the debate among proponents of traditional process-centered development methodologies (now termed plan-driven approaches) and agile development methods. We prefer, instead, to view both philosophies as having SPI intentions but adopting different foci with applicability in particular IS project contexts and overlapping in others. Boehm (2002) provides an excellent discussion of this issue.

## Background: Toward Process Management Effectiveness

The push to implement effective process support for IS development is as old as the history of IS development itself. The intention of early computing efforts in business organizations was to apply emerging information technology (IT) to improve administrative efficiency in accounting and finance (Somogyi & Galliers, 2003). Systems developers were mostly professionals in these areas who obtained training in computer programming; however, they had little understanding of IS delivery processes and thus were unconsciously incompetent (Brown, 2002). Consequently, this early stage of software development was characterized by craftsmanship with scant attention to process and management controls. Nevertheless, several sound systems were implemented.

Soon afterwards, however, there was a desire for more structured procedures to streamline and regulate systems delivery activities in order to increase productivity. According to Brown (2002), these accountants-turned-developers drew on their domain orientation and established an output-centered approach. Starting with required reports, they worked backwards to the inputs and then established a network of cross-references to required calculations and files. The problems with this approach (including the difficulty of modifying the systems so developed) are largely outside the scope of this chapter, but the quest for an adequate process had begun.

The original process structures were called programming processes (Lehman, 1987). As the size and complexity of systems development projects increased, developers attempted to simplify the conceptual, planning, and execution framework by segmenting and grouping related systems life cycle activities. The systems development life cycle (SDLC) models originated from these early attempts. According to Lehman, Benington (1956) made the first reference to such a concept, in which he outlined the characteristics of what later became the waterfall model — the best known SDLC model. The name itself was proposed by Royce (1970) and the concept was later expanded and popularized by Boehm (1976).

Many organizations have adopted this model as their guide for managing the systems development process more effectively. Unfortunately, this segmentation merely provides a conceptual overview of *what* is to be done and identifies the deliverables at the boundaries of these segments (called milestones). Little consideration is given to the *how*. Developers (in the same organization) may structure the same activities quite differently under the waterfall model. This is antithetical to the consistency objectives of effective process management. Standardization and stability underlie modern software process improvement programs as prerequisites for high-quality products. The demand for higher quality IS has intensified with the growing perception of an IS/IT paradox. This was described by Brynjolfssen (1993), Gibbs (1994), and others as the failure of the IS community to fully exploit the power and proliferation of IT and associated innovations to produce higher quality IS.

## Capability Maturity Models

In response to this perceived crisis, other quality problems, escalating software costs, and runaway IS projects, the U.S. Department of Defense sponsored the Software Engineering Institute (SEI), a federally funded research and development center at Carnegie Mellon University. The focus of the SEI was the promotion of software process excellence (SEI, 2002a). In 1991, SEI completed the development of the capability maturity model for software (SW-CMM), a process improvement model for software engineering. Several such models have been established since.

A capability maturity model (CMM) provides an instrument for assessing the maturity and the degree of discipline of an organization's IS delivery processes against a normative ideal (Paulk et al., 1993; Paulk, 2001). Essentially, it defines the attributes of a mature, disciplined software process within several process areas and specifies practices to be followed by each process to approach maturity. It also describes an evolutionary improvement path for progressing to higher levels of effectiveness from ad hoc, immature processes to disciplined mature processes with improved quality (Humphrey et al., 1991).

Several systems and software CMMs exist (SEI, 2001). The best known are SEI's model for software engineering (SW-CMM), the systems engineering capability maturity model (SE-CMM) designed by enterprise process improvement collaboration (EPIC), and the systems engineering capability assessment model (SECAM) from the International Council on Systems Engineering (INCOSE). Others include the software acquisition CMM, the people CMM, and the integrated product development CMM.

These many variations may have prompted Boehm (1994) to express the concern that despite the central function of software in modern systems delivery, the two acknowledged engineering disciplines (systems and software) have not been well integrated. Software is an increasingly significant component of most products; it is therefore vital that teams of software and systems engineers collaborate effectively to build reliable, cost-effective products.

In 2002, SEI released a single integrated capability maturity model for systems engineering, software engineering, integrated product and process development, and supplier sourcing of software (SEI, 2001, 2002b). Similarly, the Electronic Industries Alliance (EIA), EPIC, and INCOSE attempted to merge the systems engineering CMMs into a single model that was called systems engineering capability model (SECM). In order to reduce the redundancy, complexity, and cost of using multiple CMMs (SEI, 2001, 2002a), the previous models are now consolidated under the umbrella of the capability model integration (CMMI).

CMMI is intended to further improve the ability of organizations to manage the critical interactions among the people, technology, and methods that are involved in systems and software development and to monitor the delivery of the requisite quality.

## Maturity Levels

The definition of levels of process capability and quality did not originate with CMM. Crosby (1979) first defined quality as conformance to requirements and acknowledged that processes that contribute to the development of a technology occur in coherent patterns and not in random combinations (see Table 1). Radice et al. (1985) then adapted Crosby's five patterns to software development, and Humphrey (1998) expanded on their work when he identified five levels of process maturity. The SEI CMMI model incorporated a more flexible framework to guide process improvement efforts that has six capability levels.

Weinberg (1992) asserted that the relativity of quality is the reason for the many unproductive discussions of it; the definition is often emotional and includes political considerations. He suggested recasting Crosby's (1979) definition of quality (as conformance to requirements) because software requirements are seldom ever accurate. Moreover, requirements are not an end themselves, but a means to the goal of providing value. Consequently, it is possible to deliver "unrequired" features that are acknowledged as valuable. Table 1 summarizes this evolution of quality patterns and capability levels.

*Table 1. Evolution of quality patterns to maturity levels*

| Level | Crosby's Management[1] Attitude Patterns | CMM[2] (CMMI) Maturity Level | Weinberg Congruence Patterns | CMMI Capability Levels |
|---|---|---|---|---|
| 0 | | | Oblivious | Incomplete |
| 1 | Uncertainty | Initial | Variable | Performed |
| 2 | Awakening | Repeatable (Managed) | Routine | Managed |
| 3 | Enlightenment | Defined | Steering | Defined |
| 4 | Wisdom | Managed (Quantitatively Managed) | Anticipating | Quantitatively Managed |
| 5 | Certainty | Optimizing | Congruent | Optimizing |

## IS Process Management: Principles and Practices

According to Cordes (1998), the most important lessons that today's IS practitioners can learn from W. Edwards Deming and the quality management movements of the 1980s are (1) quality equals process and (2) everything is process. Zipf (2000) agreed with this pronouncement and provided further explanation in his claim that all work efforts involve processes and that their effective management is pivotal in achieving high-quality outcomes. He further contended that quality management studies performed in past decades have validated this position.

This almost sacrosanct notion (of process as a repository of quality) provides motivation and justification for striving to improve IS delivery processes in order to produce better systems. Empirical evidence indicates that a well defined software process significantly impacts the quality of the information system it produces (Harter et al., 1998; Khalifa & Verner, 2000; Krishnan & Kellner, 1999; Ravichandran & Rai, 1996, 2000); however, three important caveats have been noted:

1. Excellent process management does not automatically guarantee high-quality IS, it is a necessary — though not sufficient — condition (Rae et al., 1995).
2. Attaining some degree of process consistency is not the end but the end of the beginning; the process has to be continuously improved through organizational learning (Humphrey, 1998; Humphrey et al., 1991).
3. The process structures that an organization adopts must be suited to its culture and social systems (Curtis et al., 1995; Perry et al., 1994).

These caveats recognize that IS process considerations involve more than standards and procedures. A successful process integrates people, technology, and methods into an effective whole (Perry et al., 1994). We subscribe to Humphrey's (1998) definition of a software process as a sequence of activities that involves the interactivity of (1) people with the requisite skill, training, and motivation; (2) tools and technologies; and (3) methods — practices and principles. The fixation on any one of the three at the expense of the others is suboptimal and will not produce the desired results.

Software process management for effecting process stability, consistency, and improvement requires an infrastructure that gives appropriate consideration to this trilogy of people, technology, and methods (Krishnan & Kellner, 1999); however, trade-offs are expected. For example, when working on comparatively small projects with a few developers, the dominant contribution to product quality may well be the competence and dedication of a few participants (McConnell, 1993). In larger systems development efforts, where many people must work together effectively, process methods become more important. Global projects with mission-critical functions require a high quality process that involves excellent execution of all these elements.

## People

People are the guardians of the process structure and its faithful application, with the crucial responsibility of tracking down sources of inconsistency and negotiating their resolution. In addition, the progressive use of procedures, tools, and models must be complemented by considerations of the existing social and organizational infrastructure (Perry et al., 1994). In any event, process innovation invariably involves organizational changes. Therefore, inadequate attention to people issues constitutes suboptimization, which could conceivably reduce the impact of the innovation (Debou, 1999).

IS specialists are scarce, and the attrition rate for the profession is high (Riemenschneider et al., 2002). Competent people with the requisite skill and experience are needed to ensure effective, accurate, and efficient process performance. It is crucial that participants in IS delivery are capable of managing both the technical and behavioral challenges in a variety of project scenarios (Perry et al., 1994). Attempts to apply a high-level process improvement model with limited resources typically results in a choice between people, tools, and methods — each of which has associated problems. Similarly, organizations should avoid introducing workforce practices that employees are unprepared to apply (Curtis et al., 1995).

The people CMM (P-CMM), mentioned earlier, recognizes the role of people and their importance in SPI efforts. P-CMM focuses on the characteristics and capabilities of the people who contribute to process improvement and quality. It is concerned with the maturity of workforce practices, workforce development issues (e.g., individual competencies), team effectiveness and rapport, and priorities for improving performance and relationships that are necessary for workforce development (Curtis et al., 1995). P-CMM assists an organization in selecting standards and practices that are more suited to the characteristics and capabilities of its people.

## Technology

Technologies that support process improvement facilitate formal process definition and representation, analysis, and automation in some cases (Glass, 1999). Some are small modeling tools and techniques. Others are fairly elaborate, fourth generation languages (4GL), computer aided software engineering (CASE), simulation engines, and a variety of process frameworks and facilities (Krishnan & Kellner, 1999). Some organizations adopt powerful, integrated tools such as Workflow Management Systems and Process-Centered Software Engineering Environments (PSEEs) to support the entire spectrum of life-cycle operations (Christie, 1995).

These technologies are quite important and in several cases, have helped to provide process and product improvements; however, accepting that the overwhelming majority of performance problems, resulting from variation in work, are caused by procedures (Deming, 1986), necessarily leads to the understanding that the major problems facing organizations are managerial, not technical. Yet almost 20 years later, many improvement exercises are still focusing exclusively on the technological aspects of building software systems — the tools, techniques, and languages people use to produce the software (Perry et al., 1994).

## Methods

We have provided evidence of widespread, though not unanimous, acceptance of the notion that a mature IS delivery process is a prerequisite for delivering high-quality systems. This acceptance has triggered an extensive focus (almost an obsession in some quarters) on IS process improvement and formal process management practices (*CIO Magazine*, 2001); however, the route to an effective IS process management structure — including a rational and well-defined set of procedures for establishing consistency and predictability and general improvement — is a process itself. It requires focus and tenacity to cause the necessary investigation and the subsequent planning for and implementation of interventions and practices that are appropriately filtered and interpreted to fit the culture and other peculiar characteristics of the adopting organization (Bamberger, 1997).

This process involves: (1) understanding and assessing the effectiveness of the organization's current systems development practices, (2) using the assessment as a guide for developing a vision and prioritized goals, (3) establishing implementation plans for achieving process consistency and predictability, and (4) instituting a culture conducive to continuous improvement (Humphrey, 1998). In the following sections we discuss the two main vehicles through which organizations accomplish these objectives. The first is through process assessment, which attempts to gauge competence in preparation for the second vehicle: formalizing and institutionalizing relevant practices. The process assessment is completed largely through the capability maturity models we have discussed and the formalization through systems development methodologies (SDMs).

## *Process Assessment*

The assessment of an organization's IS delivery process requires an objective, internal analysis of the maturity level and discipline of its IS development processes. The CMM is the best known of several instruments used for this purpose. The International Standards Organization's (ISO) Software Process Improvement Capability Determination (SPICE) (El Emam & Goldenson, 1996) is also quite popular. It was developed collaboratively by 14 nations under the auspices of the International Standards Organization. Others — such as Bootstrap (Kuvaja et al., 1994), software evaluation process model (SEPRM), and personal software process (PSP) — are not as well known as CMM and SPICE.

## *Systems Development Methodology*

One way of interpreting the maturity levels of the CMM is to regard levels 1 and 2 as the absence of structures and procedures that contribute to process consistency; level 3 as the acquisition and full deployment of such standards; and levels 4 and 5 as post-implementation learning and improvement of the standards (Riemenschneider et al., 2002). Process consistency can only be achieved by the institutionalization of a comprehensive set of protocols to regulate IS delivery activities and blueprint the steps for advancing through the SDLC. This set of protocols is usually called a systems development methodology (SDM).

SDMs vary in their intricacy and rigor, but generally cover what, when, and how activities are performed, the deliverables (and their quality standards) from different implementation stages, the roles of people, and the tools and techniques that are supported (Roberts et al., 1999; Wynekoop & Russo, 1995). Several commercial and proprietary SDMs exist; however, it is recommended that an organization adopts a single methodology (Duggan, 2004).

SDMs allow project teams to eliminate project start-up inertia and focus more on the contextual characteristics of the current project. The risk of failures due to attrition is also reduced with a well-defined process; a participant in one project can be dynamically redeployed to others, and the loss of key people is not as disruptive (Duggan, 2004). Researchers have found that the use of SDMs generally results in increased productivity, quality, and reduced cycle time and effort (Harter et al., 2000; Herbsleb et al., 1994).

A methodology is not a panacea for all IS delivery problems (Riemenschneider et al., 2002), nor is its efficacy universally acknowledged. The SDM learning curve is particularly steep (Dietrich et al., 1997) and some methodologies may be onerous, particularly if applied to small projects without adaptation (Fitzgerald, 2000; Flatten, 1992). It is not unusual, however, for a methodology to be modified for a specific project but fully sanctioned by the governing body. SPI requires "fine-tuning" based on measures of effectiveness and organizational learning to drive toward optimal practices.

# Perspectives on Process-Centricity

Researchers and practitioners have identified several instances where improved software processes have contributed to IS quality and success. There are examples of direct and indirect organizational benefits from these IS quality-enhancing initiatives. Among others, desirable system benefits such as reduced defects and cycle time have been reported. Some of these system benefits have also been linked to the creation of business value (e.g., increased profitability) downstream (SEI, 2002a), but support for process-centered approaches is not unanimous. There are counterclaims about the validity of the focus, challenges to the value of its contributions, and doubts about whether the investment is misspent or justified by the benefits. Some have also alluded to serious SPI implementation problems.

## Acknowledged Benefits

SPI efforts have yielded several direct benefits which addressed longstanding systems delivery problems (Ashrafi, 2003; *CIO*, 2001; Subramanyam et al., 2004; Yamamura, 1999; Zahran, 1998):

- The improvement in estimating development effort, which has resulted in fewer schedule overruns, shorter development cycle times, and reduced time to market
- Improvement in the quality of intermediate deliverables, which reduces error rates and results in fewer delivered defects

- Improved maintainability and the reduction of rework
- Lower development costs and higher development productivity
- Greater conformance to specifications

Other indirect benefits have been enabled by standardized software processes. These include increased customer satisfaction, improved organizational flexibility, and greater internal (user) and external (client) customer satisfaction (Florac et al., 1997). Moreover, other organizations claim benefits such as more proactive IS development teams, valuable, unsolicited end-user feedback about improvement opportunities, improvement in their capability to assess project risks, and other cost-reduction opportunities (*CIO*, 2001; Diaz & Sligo, 1997; Haley, 1996; Hollenbach et al., 1997).

Goldenson and Gibson (2003) provide credible quantitative evidence (see Table 2) that CMMI-based process improvements can result in higher quality products. Example cases shown in the table provide evidence of measured improvements in quality, mostly related to reducing defects or product life cycles. Several authors have also reported cases of significant business value creation that are traceable to improvements in the software process (Table 3).

*Table 2. Process-centered quality impacts (Goldenson & Gibson, 2003)*

| **Organization** | **Quality Impact of SPI** |
|---|---|
| Fort Sill Fire Support Software Engineering Center | Substantial reduction in defects with significantly more rigorous engineering practices due to CMMI. |
| Harris Corporation | Substantial decrease in code defects after adoption of CMMI. |
| Lockheed Martin Systems Integration | Reduced software defects per million delivered SLOC by over 50% compared to defects prior to CMMI. |
| Lockheed Martin Maritime Systems & Sensors—Undersea Systems | Reduced defect rate at CMMI ML5 approximately one third compared to performance at SW-CMM ML5. |
| Northrop Grumman Defense Enterprise Systems | Only 2% of all defects found in fielded system. Reduced identified defects from 6.6 per KLOC to 2.1 over five causal analysis cycles. |
| Siemens Information Systems Ltd., India | Improved defect removal before test from 50% to 70%, leaving only 0.35 post release defects per KLOC. |
| Accenture | 5:1 ROI for quality initiatives. |
| Boeing Ltd, Australia | 33% decrease in the average cost to fix a defect. |
| Bosch Gasoline Systems | Reduction in error cases in the factory by one order of magnitude. |
| Sanchez Computer Associates | Saved $2 million in first 6 months, most through early detection and removal of defects. |

*Table 3. Business value impacts*

| **Organization** | **Quality Impact of SPI** |
|---|---|
| Raytheon (technology-based company) | Gained twofold increase in productivity; almost eightfold ROI improvement; Savings of $4.48M (1990) from a $0.58 million investment; Eliminated $15.8 of rework costs for the period (1988 to 1992) (Dion, 1993). |
| Hughes Aircraft (Software Engineering Division) | Annual savings of $2M ROI of 5:1 on process maturity assessment and two-year improvement initiative of $445,000 (Humphrey et al., 1991). |
| Motorola (Government Electronics Division) | CMM Level 4 assessment (1995); An initial investment of $90,180 returned $611,200, giving an ROI of 6:1 (Diaz & Sligo, 1997). |
| Software Process Engineering Group (SEPG) at Wipro Technologies | 2003 Software Process Achievement (SPA) awardee. Improved customer satisfaction, 40% increase in organizational productivity, reduced time to market (Subramanyam et al., 2004) |

## Reported Problems

Several authors are critical of the software process movement and remain unconvinced of its potential benefits. There are many other perspectives between Leung's (1999) assessment of only limited success of such programs and Fayad and Laitinen's (1997) categorization of some efforts as wasted and counterproductive. There are perceived ambiguities and difficulties in applying CMM recommendations for successful process modification (Pfleeger, 1996; Ravichandran & Rai, 2000); Gray and Smith (1998) argue that the CMM model is invalid because of the lack of correspondence between assessment levels and performance. The SEI's report that approximately 70% of SPI implementations are unsuccessful (SEI, 2002a) lends credence to the opponents of these programs.

## Balancing the Perspectives

The primary source of real-world process improvement failure is implementation (Fayad & Laitinen, 1997; SEI, 2002b). Perhaps the most surprising finding from SEI assessments of organizational process implementation (SEI, 2002b) is that the problems are common knowledge in the IS domain and well known within organizations. Why then are organizations not fixing them? Process implementation is a social process that occurs slowly. During this period people and systems undergo significant change, learning, adaptation, and growth (Perry et al., 1994). Success depends on high-level orchestration; however, senior managers rarely become noticeably occupied with software quality or process improvement issues.

While organizations make process improvement decisions, it is the people who adopt them. Conventional wisdom is that rational behavior will cause people to adopt and successfully utilize superior innovations to replace outmoded systems and technologies, but this concept has been constantly challenged. In general, organizational change cannot be managed by mandate nor will adoption occur by merely presenting a new process for use. In particular, SPI presumes that changes to the manner in which role-players in systems development interact, learn, and work will change organizational culture (Conradi & Fuggetta, 2002). Consequently, attaining consensus (even on best practices) without executive management support is usually very difficult.

People, having different psychological traits and profiles, will bring their own perspectives to both the selection and adoption of approaches. Popular psychology holds that people are either predominantly right- or left-brained, which predisposes them to particular behaviors. The Myers-Briggs indicator also typifies individuals into dichotomous profiles in four dimensions. For example, on the "Sensing/Intuition" dimension, intuitive people receive data through hunches and insights, while sensing individuals rely more on hard facts (Myers & McCauley, 1985). The former will visualize a better process and become easily frustrated by the pace of the change; the latter tends to accept smaller incremental improvements.

From our experience, the method of process structure creation also influences performance. Experience-based processes grown mostly from feedback are sometimes chaotic. Concept-based processes begin with a rationale (an area of order) and will be useful if the area of order matches the area of need. Perhaps the most effective approach is to begin with concept-based practices and improve them through learning. Regardless of the path taken, key success factors include high-level leadership, the degree of congruence between people and methods, and the usefulness of the assumptions from a business perspective.

## Implications and Conclusions

Severe problems can arise when complex systems are developed without a defined process or effective engineering and management practices. Some organizations also experience difficulties even after fully deploying a defined process. Obviously, the benefits of SPI programs do not accrue merely from the existence of institutionalized and documented standards and practices. The manner in which managers involve themselves in the implementation as elaborated by Weinberg (1992) is also critical. Deming (1986) emphasized the importance of leadership in his fourteen points for achieving quality improvement[3]. We have noted that among the business benefits of well managed software process improvement endeavors is the less recognized contribution to corporate profitability and competitive positioning; this should be incentive enough for executive commitment.

The Standish Group (2002) suggests that executive support is one of the ten most important success factors in IS project delivery. Furthermore, other factors — including user involvement, experienced project management, clear business objectives, minimized scope, agile requirements process, standard infrastructure, formal methodology, reliable estimates, and skilled staff — also require executive support. Senior management's dedication and commitment are essential to the success of any sustainable program of process improvement.

The volatility and unpredictability of the IS field, however, contributes to a management style that is primarily reactive and conducive to crisis response. The accolades and promotions usually go to the "heroes" who rescue organizations from impending disasters. Very little recognition goes to professionals who, through effective planning, avoided the trouble in the first place. The evolution of IS delivery processes toward standardization and consistency necessitates a change in management style from reactive to proactive planning and building. Good systems will not automatically materialize from arbitrary software process manipulations.

Now more than ever, there is greater acceptance that high quality IS are no more likely to emerge from hodge-podge process structures than from no process structure at all. Fenton (1996), who supports process-centricity, cautions that poor and poorly implemented process structures will have no impact on product quality. Instead, an organization should develop the will and tenacity to (1) create a focused approach based on objective analysis, (2) design and implement an enterprise-appropriate plan to improve the discipline and capability of its systems delivery process, (3) learn by measuring the performance of the process, and (4) sustain the focus through perpetual attention to its improvement.

Some software process insights are now almost axiomatic, for example:

- For successful outcomes, software improvement initiatives must pay attention to all of the three fundamental components of an effective process: people, technology, and process structures.
- Process improvement goals must be supported by the culture and characteristics of the adopting organization.
- Success is dependent on the realism of the goals and their congruence with organizational capability and needs.

- Efforts will be unrewarding if an organization does not believe in the merits of the process improvement exercise.

There is still no consensus on other highly-debated issues; further research is required to provide more definitive insights. For example, what should be the tradeoff between process rigor and flexibility, and what are the determining factors? Is there a clash of priorities between Agile Development Methods and traditional SDMs? Given the alternative methods available for sourcing systems, should an organization adopt one or multiple SDMs? These issues were not addressed in this chapter.

Enough evidence exists to confirm that SPI initiatives and standards have contributed to the improvement of software quality. It would be irrational, therefore, to attribute failed SPI implementation efforts or failed implementation processes to the process structure itself. Voas' (1997), in criticizing the extensive focus on software processes, asks rhetorically whether clean pipes can only produce clean water. The more important aspect of the analogy, however, is that dirty pipes cannot produce clean water.

# References

Ashrafi, N. (2003). The impact of software process improvement on quality: In theory and practice. *Information and Management, 40*(7), 677-690.

Bamberger, J. (1997). Essence of the capability maturity model. *Computer, 30*(6), 112-114.

Banker, R. D., Davis, G. B., & Slaughter, S. A. (1998). Software development practices, software complexity, and software maintenance performance: A field study. *Management Science, 44*(4), 433-450.

Benington, H. D. (1956, June). Production of large computer programs. *Proceedings of the Symposium on Advanced Computer Programs for Digital Computers (ONR).*

Boehm, B. (1976). Software engineering. *IEEE Transactions on Computing, C-5*(12), 1226-1241.

Boehm, B. (1994). Integrating software engineering and system engineering. *The Journal of INCOSE, (I)*I, 147-151.

Boehm, B. (2002, January). Get ready for agile methods, with care. *Computer, 35*(1), 64-69.

Brown, D. W. (2002). *An introduction to object-oriented analysis.* New York: John Wiley & Sons.

Brynjolfssen, E. (1993). The productivity paradox of information technology. C*ommunications of the ACM, 36*(12), 67-77.

Christie, M. (1995). *Software process automation: The technology and its adoption.* Berlin: Springer-Verlag.

*CIO Magazine.* (2001, June). Improving software development (Research Report). Retrieved November 29, 2005, from http://www.cio.com

Conradi, H., & Fuggetta, A. (2002). Improving software process improvement. *IEEE Software, 19*(4), 92–99.

Cordes, R. M. (1998). Flowcharting: An essential tool. *Quality Digest.* Retrieved November 29, 2005, from http://www.qualitydigest.com/jan98/html/flowchrt.html

Crosby, P. (1979). *Quality is free: The art of making quality certain*. New York: McGraw-Hill.

Curtis, B., Hefley, W. E., & Miller, S. (1995). *Overview of the people capability maturity model* (Tech. Rep. CMU/SEI-95-MM-01). Carnegie Mellon University; Software Engineering Institute.

Debou, C. (1999). Goal-based software process improvement planning. In R. Messnarz & C. Tully (Eds.), *Better software practice for business benefit: Principles and experience* (pp. 107-150). Los Alamitos, CA: IEEE Computer Society.

Deming, E. (1986). *Out of the crisis.* Cambridge, MA: MIT Center forAdvanced Engineering.

Diaz, M., & Sligo, J. (1997). How software process improvement helped Motorola. *IEEE Software, 14*(5), 175-181.

Dietrich, G., Walz, D., & Wynekoop, J. (1997). The failure of SDT diffusion: A case for mass customization. *IEEE Transactions on Engineering Management, 44*(4), 390-398.

Dion, R. (1993). Process improvement and the corporate balance sheet. *IEEE Software, 10*(3), 28-35.

Duggan, E. W. (2004). Silver pellets for improving software quality. *Information Resources Management Journal, 17*(2), 1-21.

El Emam, K., & Goldenson, D. R. (1996). Some initial results from the international SPICE trials. *Software Process Newsletter. Technical Council on Software Engineering*.

Fayad, M., & Laitinen, M. (1997). Process assessment considered wasteful. *Communications of the ACM, 40*(11), 125-128.

Fenton, N. (1996). Do standards improve quality: Counterpoint. *IEEE Software, 13*(1), 23-24.

Fitzgerald, B. (2000). Systems development methodologies: The problem of tenses. *Information Technology & People, 13*(3), 174-182.

Flatten, P. O. (1992). Requirements for a life-cycle methodology for the 1990s. In W. W. Cotterman & J. A. Senn (Eds.), *Challenges and strategies for research in systems development* (pp. 221-234). New York: John Wiley & Sons.

Florac, W. A., Park, R. E., & Carleton, A. D. (1997). *Practical software measurement: Measuring for process management and improvement.* Carnegie Mellon University, Pittsburgh, Software Engineering Institution.

Gibbs, W. W. (1994). Software's chronic crisis. *Scientific American, 271*(3), 86-95.

Glass, R. L. (1999). The realities of software technology payoffs. *Communications of the ACM, 42*, 74-79.

Goldenson, D., & Gibson, D. (2003, October). *Demonstrating the impact and benefits of CMMI: An update and preliminary results* (Special Report CMU/SEI-2003-SR-009).

Gray, E., & Smith, W. (1998). On the limitations of software process assessment and the recognition of a required re-orientation for global process improvement. *Software Quality Journal, 7*(1), 21-34.

Haley, T. J. (1996). Raytheon's experience in software process improvement. *IEEE Software, 13*(2), 33-41.

Harter, D. E., Krishnan, M. S., & Slaughter, S. A. (2000). Effects of process maturity on quality, cycle time, and effort in software product development. *Management Science, 46*(4), 451-66.

Harter, D. E., Slaughter, S. A., & Krishnan, M. S. (1998, December). The life cycle effects of software quality: A longitudinal analysis. *Proceedings of the International Conference on Information Systems,* Helsinki, Finland (pp. 346-351).

Herbsleb, J., Carleton, A., Rozum, J., Siegel, J., & Zubrow, D. (1994). Benefits of CMM-based software process improvement: Initial results (Tech. Reports CMU/SEI-94-TR-013 and ESC-TR-94-013). Carnegie Mellon Univiversity, Software Engineering Institute.

Hollenbach, C., Young, R., Pflugrad, A., & Smith, D. (1997). Combining quality and software process improvement. *Communications of the ACM, 40*(6), 41-45.

Humphrey, W. S. (1998). Characterizing the software process: A maturity framework. *IEEE Software, 7*(2), 73-79.

Humphrey, W. S. (2002). Three process perspectives: Organizations, teams, and people. *Annals of Software Engineering, 14*, 39-72.

Humphrey, W. S., Snyder, T., & Willis, R. (1991). SPI at Hughes Aircraft. *IEEE Software, 8*(4), 11-23.

Khalifa, M., & Verner, J. M. (2000). Drivers for software development method usage. *IEEE Transactions on Engineering Management, 47*(3), 360-369.

Krishnan, M. S., & Kellner, M. I. (1999). Measuring process consistency: Implications for reducing software defects. *IEEE Transactions on Software Engineering, 25*(6), 800-815.

Kuvaja, P., Similä, J., Kranik, L., Bicego, A., Saukkonen, S., & Koch, G. (1994). *Software process assessment and improvement — the BOOTSTRAP approach*. Malden, MA: Blackwell Publishers.

Lehman, M. M. (1987). Process models, process programs, programming support. *Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA (pp. 14-16).

Leung, H. (1999). Slow change of information system development practice. *Software Quality Journal, 8*(3), 197-210.

McConnell, S. (1993, July). From anarchy to optimizing. *Software Development*, 51-56.

Myers, I. B., & McCaulley, M. H. (1985). *Manual: A guide to the development and use of the Myers-Briggs type indicator*. Palo Alto, CA: Consulting Psychologists Press.

Paulk, M. C. (2001). Extreme programming from a CMM perspective. *IEEE Software, 8*(6), 19-26.

Paulk, M. C., Curtis, B., Chrissis, M. B., & Weber, C. V. (1993). The capability maturity model: Version 1.1. *IEEE Software,* 10(1), 18-27.

Perry, D. E., Staudenmayer, N. A., & Votta, L. G. (1994). People, organizations, and process improvement. *IEEE Software, 11*, 36-45.

Pfleeger, S. L. (1996). Realities and rewards of software process improvement. *IEEE Software, 13*(6), 99-101.

Radice, R., Harding, P., & Phillips, R. (1985). A programming process study. *IBM Systems Journal, 24*(2), 91-101.

Rae, A., Robert, P., & Hausen, H. L. (1995). *Software evaluation for certification: Principles, practice and legal liability*. London: McGraw-Hill.

Ravichandran, T., & Rai, A. (1996). Impact of process management on systems development quality: An empirical study. *Proceedings of the Americas Conference on Information Systems*.

Ravichandran, T., & Rai, A. (2000). Quality management in systems development: An organizational system perspective. *MIS Quarterly, 24*(3), 381-415.

Riemenschneider, C. K., Hardgrave, B. C., & Davis, F. D. (2002). Explaining software developer acceptance of methodologies: A comparison of five theoretical models. *IEEE Transactions on Software Engineering, 28*(12), 1135-1145.

Roberts, T. L., Gibson, M. L., & Fields, K. T. (1997). Systems development methodology implementation: Perceived aspects of importance. *Information Resources Management Journal, 10*(3), 27-38.

Royce, W. W. (1970, August). Managing the development of large software systems: Concepts and techniques. *Proceedings of the IEEE, WESCON,* Los Angeles, CA (pp. 1-9).

SEI. (2001). *Concept of operations for the CMMI.* Carnegie Mellon University, Software Engineering Institute, Pittsburg, PA. Retrieved November 29, 2005, from http://www.sei.cmu.edu/cmmi/background/conops.html

SEI. (2002a). *Process maturity profile of the software*. Carnegie Mellon University, SCAMPI Appraisal Results, Software Engineering Institute, Pittsburg, PA. Retrieved November 29, 2005, from http://www.sei.cmu.edu/sema/pdf/SWCMM/2002aug.pdf

SEI. (2002b). *Capability maturity model integrated (CMMI)* (Tech. Rep. No. CMM/SEI-2002-TR-012., Version 1.1). Software Engineering Institute, Pittsburg, PA.

Somogyi, E. K., & Galliers, R. D. (2003). Information technology in business: From data processing to strategic information systems. In R.D. Galliers & D.E. Leidner (Eds.), *Developments in the application of information technology in business* (pp. 3-26). Boston: Butterworth-Heinemann.

Standish Group, The. (2002). *What are your requirements?* West Yarmouth, MA: The Standish Group International. (Based on the 2002 CHAOS Report).

Subramanyam, V., Deb, S., Krishnaswamy, P., & Ghosh, R. (2004). *An integrated approach to software process improvement at Wipro Technologies: veloci-Q* (Tech. Rep. No. CMU/SEI-2004-TR-006, ESC-TR-2004-006). Software Engineering Institute, Pittsburg, PA.

Vessey, I., & Conger, S. A. (1994). Requirements specification: Learning object, process, and data methodologies. *Communications of the ACM, 37*(5), 102-112.

Voas, J. M. (1997). Can clean pipes produce dirty water? *IEEE Software, 14*(4), 93-95.

Weinberg, G. (1992). *Quality software management volume 1: Systems thinking*. New York: Dorset House Publishing.

Wynekoop, J., & Russo, N. (1995). Systems development methodologies: Unanswered questions. *Journal of Information Technology, 10*, 65-73.

Yamamura, G. (1999). Software process satisfied employees. *IEEE Software, 16*(5), 83-85.

Zahran, S. (1998). Software process improvement: Practical guidelines for business success. Reading, MA: Addison-Wesley.

Zipf, P. J. (2000). Technology-enhanced project management. *Journal of Management in Engineering, 16*(1), 34-39.

## Endnotes

[1] Crosby (1979) calls these levels stages and describes them as follows (Humphrey, 2002).

- **Uncertainty:** Management embraces quality notions but has no understanding of, or commitment to, it.
- **Awakening:** Management recognizes the potential benefits of a quality program but are unwilling to invest time or money to quality improvement.
- **Enlightenment:** Management commits to and launches a quality improvement program.
- **Wisdom:** A quality executive is appointed and quality is now measured and managed.
- **Certainty:** The organization has developed a quality competence and becomes a vital corporate activity.

[2] CMM Levels (Extracted from Duggan, 2004):

- Level 1, the **“initial” stage** is characterized by the absence of formal procedures – a state of near chaos, where proper project planning and control are non-existent. Organizations may experience successful implementations but these depend on heroic individual effort.

- Level 2 is called the **"repeatable" stage**, where basic project management capability exists. The organization can establish and track schedules and cost, and monitor and report on progress. But process management discipline does not exist and individual project leaders supply their own process management techniques. Project success depends greatly on the skill of the project group.
- An organizations at Level 3, the **"defined" stage**, uses a common, institutionalized process management method (systems development methodology), for all its IS projects. The process management discipline produces consistent, stable, and predictable outcomes. This allows dynamic redeployment of human resources and reduces attrition-related failure risks.
- At Level 4, the **"managed" stage**, organizational learning is a key objective. Establishing a common methodology is not viewed as the end-all of process management. The organization collects detailed measures of process and product quality, which is used to refine the development process.
- An organization at Level 5 has **"optimized"** its process management capability. It uses the metrics from Level 4, acknowledged best practices, and benchmarks, and the results of controlled experiments and pilot studies to adjust the process to achieve continuous process improvement.

[3] Deming's 14 Points (Deming, 1986)

1. Create *Consistency of Purpose* for continual improvement of products and service to become competitive, to stay in business, and to provide jobs.
2. *Adopt the new philosophy*. We are in a new economic age. Western management must waken to the challenge to halt the continued decline of business and industry.
3. *Cease dependence on inspection*. Eliminate the need for mass inspection as the way of life to achieve quality by building quality into the product in the first place.
4. *End lowest tender contracts*. End the practice of awarding business on the basis of price tag. Instead minimize total cost by minimizing variation. Reduce the number of suppliers for the same item on a long-term relationship built on loyalty and trust.

5. *Improve constantly* and forever every process for planning, production, and service. Search continually for problems in order to improve every activity in the company to improve quality and productivity, and thus to constantly decrease costs.
6. *Institute training* on the job for all, including management, to make better use of every employee.
7. *Adopt and institute leadership* to help people, machines, and gadgets do a better job.
8. *Drive out fear* throughout the organization so that everybody may work effectively and more productively for the company.
9. *Break down barriers* between departments. People in different areas, such as research, design, sales, and production must work in teams to tackle problems that may be encountered with products or service.
10. *Eliminate slogans, exhortations, and targets* by asking for zero defects and new levels of productivity. Such exhortations only create adversarial relationships.
11. *Eliminate quotas* for the workforce and management by numbers. Substitute leadership.
12. *Permit pride of workmanship*. Remove the barriers that rob hourly workers and people in management of their right to pride of workmanship. Abolish annual merit rating systems; change from sheer numbers to quality.
13. *Institute a vigorous program of education* and encourage self improvement.
14. *Transformation* is everyone's. Create a structure in top management that will push to take action to accomplish the transformation.

## Chapter VIII

# Developer-Driven Quality: Guidelines for Implementing Software Process Improvements

Gina C. Green, Baylor University, USA

Rosann Webb Collins, University of South Florida, USA

Alan R. Hevner, University of South Florida, USA

## Abstract

*Much attention has been given to Software Process Improvements (SPIs) based on the premise that system development outcomes are largely determined by the capabilities of the software development process. The content of this chapter presents the results of a set of research projects investigating why SPIs have not been diffused and utilized in the software engineering community as expected (Fayad et al., 1996; Fichman & Kemerer, 1997; Luqi & Goguen, 1997; Pfleeger & Hatton, 1997). We show that a software developer's perceived control over the use of an SPI* *impacts its diffusion success. Additionally, we show that a software developer's perceptions of enhanced software quality and increased individual productivity achieved through SPI use impact the successful diffusion of the SPI. Results of these research efforts support the compilation of a clear set of management guidelines to ensure the effective use of SPIs in software development organizations.*

# Introduction

The search for new ideas and innovations to improve software development productivity and enhance system quality continues to be a key focus of industrial and academic research. Much attention has been given to Software Process Improvements (SPIs) based on the premise that system development outcomes are largely determined by the capabilities of the software development process. Recent attempts to improve software development practices have focused on defining, monitoring, and measuring development activities in an effort to identify and subsequently verify areas of improvement. Work in this area has resulted in a number of SPI innovations such as the Capability Maturity Model Integration (CMMI), the Personal Software Process (PSP), the Team Software Process (TSP), Cleanroom development methods, agile development methods, and others.

Proponents of SPIs have asserted that well-defined and measured processes eventually lead to gains in software quality and programmer productivity, yet there is evidence that many promising SPI techniques have not been effectively transitioned into successful use. This chapter reports on our research studies that investigate why this trend exists.

## Motivation

The motivation for our research is to better understand the phenomenon of SPI diffusion in order to inform software development managers on how to ensure successful use of SPIs in their organizations. Solutions to the problems of software development should include both human and technical dimensions (Constantine, 1995; DeMarco & Lister, 1999; Yourdon, 1996). Thus, a basic premise of our research is that an effective understanding of SPI diffusion should integrate technical research in software engineering with behavioral research in information systems and other fields.

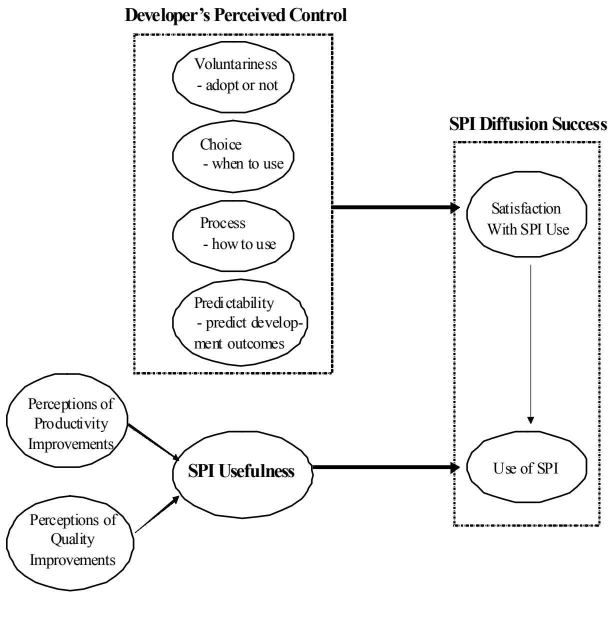
## Research Questions

Our research on SPI diffusion attempts to answer the following research questions:

1. What constitutes successful diffusion of an SPI in software development organizations?
2. What factors influence the successful diffusion of SPIs in software development organizations?

To answer these questions, we begin by conducting a review of relevant software engineering and information systems research as summarized in the research model of Figure 1.

*Figure 1. Diffusion of software process improvements: The research model*


Developer's Perceived Control
Voluntariness - adopt or not
Choice - when to use
Process - how to use
Predictability - predict development outcomes
SPI Diffusion Success
Satisfaction With SPI Use
Use of SPI
Perceptions of Productivity Improvements
Perceptions of Quality Improvements
SPI Usefulness

# Background and Research Foundations

## Defining Successful SPI Diffusion in Organizations

Diffusion of innovations research finds that successful implementations of innovations are indicated by sustained use of the innovation in the organization (Fowler & Levine, 1993; Rogers, 1982). In a review of information systems (IS) research on IS success, DeLone and McLean (1992) observe that IT use is the most frequently reported measure of IT implementation success. Thus, our research model includes SPI use as an indicator of successful SPI diffusion. However when use of an IT is mandated, satisfaction is suggested as a more appropriate measure of IT success (Adams et al., 1992; Chau, 1996; DeLone & McLean, 1992). Thus, our research model also includes satisfaction with the SPI as a second indicator of successful SPI diffusion.

Our model also tests a relationship between these two SPI success variables. DeLone and McLean (1992) posit that increases in satisfaction with an IT are related to increases in its use; however, they do not test this relationship empirically.

## Factors Influencing Satisfaction with an SPI: Personal Control

In the job satisfaction literature, control over work methods and task activities has generally been found to be motivating (Aldag & Grief, 1975; Hackman & Oldham, 1976). Thus, the research model indicates that one's amount of personal control over the use of an SPI can impact one's satisfaction with SPI use. However, our studies go further to explore ways in which a software developer's sense of personal control can be influenced. In social psychology, personal control has been defined as "an individual's beliefs, at a given point in time, in his or her ability to effect a change in a desired direction" (Greenberger & Strasser, 1986, p. 165). Personal control has been described as having several dimensions as summarized in Table 1. These dimensions are included in our research model as factors that can influence a software developer's satisfaction with SPI use.

## Factors Influencing Use of an SPI: Quality, Productivity, and Usefulness Perceptions

While previous research has identified the perception of usefulness as an important indicator of use, understanding what factors software engineers

*Table 1. Dimensions of perceived control and study variables*

| **Dimensions of Perceived Control from the Psychology and Organization Sciences Literature** | **Perceived Control Study Variables in the SPI Context** |
|---|---|
| □ "strategic": freedom to set goals (Manz & Sims, 1980; Parsons, 1960)<br>□ "decisional": opportunity to choose among various possible actions (Baronas & Louis, 1988; Langer, 1983)<br>□ "decision authority": authority to make decisions on the job (Fox et al., 1993)<br>□ "participation": degree to which the individual has input into or influence over operating issues that directly or indirectly affect the task domain (Ashforth & Saks, 2000) | **Voluntariness:** software developer can determine whether or not an innovation should be adopted |
| □ "administrative": responsibility for managing work activities (Parsons, 1960)<br>□ "skill discretion": about the variety of skills to be employed on the job; freedom to monitor own work (Fox et al., 1993; Manz & Sims, 1980)<br>□ "job autonomy": freedom to be your own master within the prescribed task domain, including work methods (Ashforth & Saks, 2000)<br>□ "self control": freedom to determine what actions are required (Henderson & Lee, 1992) | **Choice:** software developer can determine when it is appropriate to use an innovation during systems development |
| □ "operational": freedom to determine how to achieve goals (Parsons, 1960)<br>□ "process or behavioral": individual has the ability to take direct action on the environment to influence an event (Baronas & Louis, 1988; Langer, 1983)<br>□ "self control": freedom to determine how to execute work activities; control over order of task performance, pacing, scheduling of work breaks, procedures, and policies in the workplace (Fox et al., 1993; Henderson & Lee, 1992) | **Process control:** software developer can determine how to use an innovation without organizational or innovation-imposed constraints |
| □ knowledge of what event(s) will occur and when; not necessarily controlling the event (Langer, 1983) | **Predictability:** software developer can predict software development outcomes when using the SPI innovation |

consider key to determining the usefulness of the SPI has not been achieved. Riemenschneider et al. (2002) study factors that influence a developer's intention to use a development methodology. They find that the usefulness of the methodology to developers influences their intention to sustain use of the methodology. They further note that to be useful to developers, the methodology must enable developers to be more productive and achieve higher levels of performance in their jobs.

In software engineering literature, two frequently-cited objectives of SPI innovations are to reduce development costs through improved developer productivity and to improve end user satisfaction with the resulting software by reducing software defects. Thus for managers and developers to perceive an SPI as useful, they would likely expect to see gains in their software quality and/

or their work productivity as a result of using the SPI. Thus, our model includes these two factors as influencing the perceived usefulness of an SPI to a developer.

# Research Methodology

Our research model has been tested in two major, related studies. The first study (Green et al., 2004) focuses on influencing developer satisfaction with SPI use through the understanding of the software developer's perceived control over the use of the SPI. The second study (Green et al., 2005) focuses on influencing sustained developer use of an SPI through understanding the software developer's perceptions of how the SPI influences software quality and developer productivity. Both studies use the Personal Software Process (PSP) as the SPI innovation for testing the research model.

## PSP$^{SM}$[1] as a Software Process Improvement (SPI) Exemplar

PSP is a process framework and set of software development methods that enable developers to become more productive and disciplined software engineers. PSP requires that developers continuously improve their skills so that their productivity and product quality improve over time. Similar to other quality assessments, PSP establishes four levels of maturity in development. At the initial level, PSP asks developers to focus on knowledge creation; documentation of their current practices and recording of productivity and quality metrics. More mature levels require developers to use processes that embody best-practice knowledge (e.g., effective specification methods, defect prevention) as well as to adapt effective development processes based on their own knowledge and environment.

PSP is a particularly appropriate research context for the study of the implementation of an SPI innovation because of the role of the Software Engineering Institute (SEI), which defines PSP's advantages and benefits, as well as the organizational changes needed for its implementation. The SEI also provides training on PSP, so the costs and knowledge barriers to PSP implementation can be significantly reduced (Fichman, 2000; Swanson & Ramiller, 1997). In addition, PSP is a software process improvement that is implemented at an individual level, which supports research at the individual level of analysis.

*Table 2. Items on developer survey*

| Variable/Scale | Sources of Scale Items | Cronbach's Alpha |
|---|---|---|
| Voluntariness<br>... superiors expect me to use PSP<br>... use of PSP is voluntary<br>... supervisor does not require me to use PSP<br>... using PSP is not compulsory | Moore and Benbasat (1991)<br>Iivari (1996) | .92 |
| Choice<br>... can you decide what parts of PSP you use<br>... can you decide when you will use PSP techniques | Researcher-developed | .94 |
| Process Control<br>... pre-specified sequence of steps that I was required to follow<br>... required to follow existing standards | Researcher-developed | .71 |
| Predictability<br>…does use of PSP allow you to better predict your development effort<br>…does use of PSP allow you to better predict quality of software | Tetrick and LaRocco (1987) | .84 |
| Quality<br>…use of PSP has decreased the number of errors in software products I build<br>…software developed with PSP requires less maintenance<br>…use of PSP has improved the quality of software products I build | Iivari (1996)<br>Researcher-developed | .78 |
| Productivity<br>…use of PSP has greatly speeded up my development of new applications<br>…use of PSP has definitely made me more productive<br>…use of PSP has significantly reduced the time I spend in software development | Iivari (1996) | .83 |
| Usefulness<br>…using PSP improved my job performance<br>…PSP was successful in transforming me into a more disciplined software engineer<br>…using PSP made it easier to do my job<br>…I found PSP useful in my job | Davis (1989) | .90 |
| Satisfaction<br>... PSP is exactly what I need<br>... PSP meets my software development needs | Doll and Torkzadeh (1989) | .87 |
| Use<br>... proportion of projects I use PSP with<br>... how often do you use PSP | Iivari (1996) | .90 |

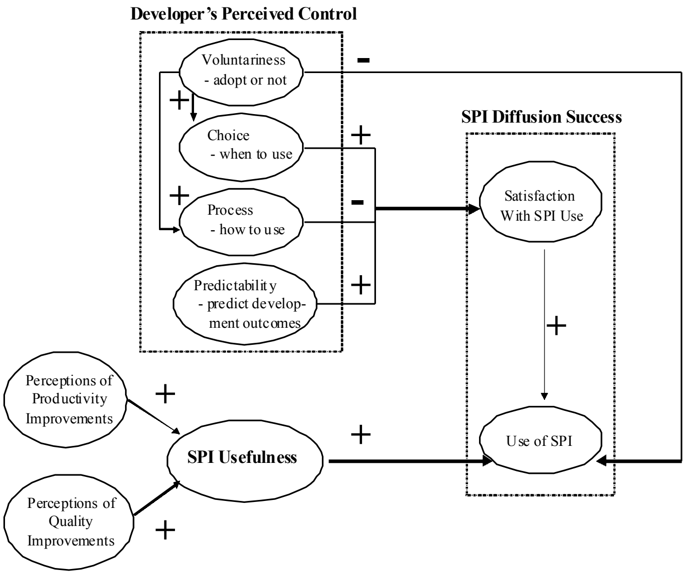
## Data Collection

In order to examine and test the research model, we conducted a field survey of software developers in multiple organizations who had adopted the PSP within their projects. The survey instrument was pretested with researchers, graduate students, and practitioners to ensure content validity. Table 2 lists the questions contained on the disseminated survey. Respondents answered most items using a 7-point Likert scale; for these items, variable scores were determined by taking the average of the item scores for each variable. The exception to this is the USE measure; items for this measure were summed to produce the score for this variable. In addition, respondents were given an open-ended question at the end of the survey where they could make comments on their use of and satisfaction with PSP. In response to this question developers were encouraged to identify additional factors that influenced their attitudes toward PSP.

Surveys were distributed to 154 software developers who agreed to participate in the study; 71 completed surveys were returned, resulting in a response rate of 46%. Of these, 8 were unusable, resulting in a sample size of 63 for the studies. The 63 respondents represent 24 different organizations. Approximately half the sample classified themselves as programmers. Approximately 80% of the study respondents were male. The average age of respondents was approximately 33 years and over 40% had advanced degrees.

To determine the reliability of scales, Cronbach's alpha was computed for each scale. Results of the final reliability assessment are shown in Table 2. All scales exhibit reliability levels greater than 0.70, the recommended minimum level of acceptable reliability (Nunnally, 1978). Using the collected survey data, the relationships depicted in the research model were tested using a Partial Least Squares (PLS) analysis (Green et al., 2004). Figure 2 summarizes the results of these analyses.

*Figure 2. Results of model testing*


Developer's Perceived Control
Voluntariness - adopt or not
Choice - when to use
Process - how to use
Predictability - predict development outcomes
+
+
-
+
-
+
SPI Diffusion Success
Satisfaction With SPI Use
+
Use of SPI
Perceptions of Productivity Improvements
Perceptions of Quality Improvements
+
+
SPI Usefulness
+

# Discussion

The study results provide an initial test of the relationship between Satisfaction and Use that was posited by DeLone and McLean (1992). While the cross-sectional nature of the research method does not permit a test of the direction of impact, the significant relationship ($p < .1$) found between satisfaction with the SPI and frequency of its use supports this link in the model. A better understanding of the relationship between Satisfaction and Use is important in many software engineering innovation contexts, since developers who are more satisfied with an innovation are more likely to be opinion leaders and champions of the innovation (Leonard-Barton, 1987).

## Developer Personal Control

The findings of this study clearly support the multidimensional conceptualization of personal control, since different aspects of personal control are found to have differing impacts on diffusion success. As predicted, the more the adoption of the SPI is of free will (voluntariness), the less frequent the use of the SPI ($p = .05$). This finding mirrors prior research (Karahanna et al., 1999) which finds that organizational mandates on use of an SPI increase the frequency of its use. Voluntariness is positively related to *choice* over when to use an SPI ($p = .01$), and this *choice* has a positive relationship with *satisfaction* with that use ($p = .05$). Predictability of development outcomes when using the SPI also appears to enhance *satisfaction* ($p = .01$).

However, increased developer control over how to use the SPI in development (*process control*) has a negative impact on *satisfaction* ($p = .05$). This seeming contradiction with the overall positive relationship between personal control and diffusion success underscores the importance of conceptualizing and measuring the separate dimensions of personal control. The negative relation between *process control* and *satisfaction* may reflect the unique nature of the software development context, which is quite complex and unstructured. Our results indicate that developers prefer a situation in which they have discretion over when to use the SPI, but they prefer reduced process choice about how to use the innovation in everyday use. In other words, well-defined standards and best practices for use of a new technique (e.g., PSP, ICASE, UML) provide for a more structured work environment and reduce overall task complexity.

## Quality and Productivity Perceptions

In contrast to the relationships between the dimensions of personal control and SPI diffusion success, the links between SPI *usefulness*, perceptions of *quality* and *productivity* improvements from using the SPI, and SPI use are simple. Developers' perceptions of *quality* and *productivity* have a significant, positive relationship with perceived *usefulness* of the SPI ($p < .01$ and $p < .05$, respectively), and perceived *usefulness* has a strong, positive relationship with frequency of SPI *use* ($p < .01$).

These findings support the expected link between perceptions of usefulness of an SPI and use, but more importantly, discover two important factors that explain what creates that overall perception of *usefulness*. Taken together, perceptions of *quality* and *productivity* explain 66% of the variance in perceived *usefulness*. In qualitative data collected from the survey, 14 of the 43 comments made by respondents concerned quality and productivity impacts of the SPI. For example, these comments about the SPI describe how it improves the quality of the software being developed:

*"[the SPI] makes me more aware of typical syntax and data type errors I commonly make."*

*"[the SPI] helps me keep excellent track of defects at all levels."*

These comments reflect the impacts of the SPI on quality and productivity. The source of those benefits appears to stem from the process structure discussed earlier:

*"[the SPI] gives me the structure I need to come back to programming tasks and pick them up and start up again with hope that I can maintain a high level of quality throughout."*

## Additional Factors in Perception of Usefulness

What accounts for the other 34% in the variance in that overall perception of *usefulness* of the SPI? At the end of the survey instrument, developers were given the opportunity to identify other factors that influence their perceptions of SPI *usefulness*. The issues identified by the developers offer some possible explanations.

The issue receiving the most attention dealt with the importance of matching the SPI to the developer's task or task environment, in other words, *task-technology fit*. Comments suggest that when developers are responsible for tasks such as maintenance and debugging, work on exploratory projects, or programming in 4GL or Unix environments, they find the PSP to be more difficult to adapt, constraining their use of PSP in these environments. For example:

*"The difficulty I have had with applying PSP has been trying to keep pace with changing platforms and languages and trouble adapting it to a maintenance environment."*

*"More study needs to be done on how to use PSP with new tools/ environments ...."*

*"... challenges we have ... involve use of PSP during Maintenance phases of a project, and use of PSP with 4GLs."*

*"... colleagues active in other activities like maintenance and support ... leads to a weak usage of PSP."*

*"What I use from PSP is creation of techniques for tasks I do frequently ...."*

*"It is my experience that the techniques are useful when you are working on projects where a large percentage of what needs to be done is known and understood. In developing for new platforms or unfamiliar products, I found that PSP estimates were off by as much as a factor of 4 .... In our environment, we are constantly doing new things, making PSP ... usefulness difficult [in this situation]."*

The above comments and others like them suggest that development managers should ensure that use of SPIs is clearly targeted for or tailorable to specific development tasks and environments. This observation has support in research examining task-technology fit. Goodhue (1995) suggests that a lack of fit between an IT and the user's task can inhibit use of the IT. Indeed, Goodhue (1995) and others (e.g., Lim & Benbasat, 2000; Mathieson et al., 2001) suggest that the fit of the technology to the task of the user impacts the usefulness of the technology to the user. Broad implementations of SPIs will meet with less success than those implementations where SPI use is tailored to the specific needs of software developers.

The second most cited issue in the comments data relates to improvements in managing and understanding projects, as well as managing personal tasks.

*"... on one small project, [PSP] did not improve my productivity or reduce my error rate ... however, I understand the flow of my work better and know where to look for trouble. I can head off trouble before it starts."*

*"... I have not noticed a significant amount of time saved. What I gained most from PSP was more effective techniques of reviewing my work…realizing the importance of following a checklist of review items ... [coming] up with an organized method of reviewing my work."*

*"We have a better knowledge of where we are on projects and what resources we require."*

## Management Guidelines for Selecting and Implementing SPIs

The research summarized above highlights how the nature of the SPI and the implementation environment impact software developer perceptions, satisfaction, and use of a specific SPI, for example PSP in our studies. While some of the relationships are simple and not surprising (e.g., *usefulness* is positively related to *use*), the role of managerial discretion regarding how or whether to use an SPI is much more complex (Green & Hevner, 2000). In the next sections, we identify specific guidelines to managers that may assist in positively influencing the key developer perceptions identified in our research. These guidelines are grouped into three categories that correspond to organizational processes over which managers have direct influence: the SPI selection process, SPI training, and the SPI implementation process. The desirable characteristics of an SPI can be used to guide SPI selection, although it is possible that effective SPI training can be designed to increase satisfaction with and use of techniques that do not meet these characteristics. These guidelines are summarized in Table 3.

### Guidelines for Selection of an SPI

There are four principal characteristics of an SPI that we found to have a positive impact on satisfaction and use:

*Table 3. Summary of management guidelines for implementing SPIs*

| | |
|---|---|
| **Selection Guidelines** | Select SPIs that enable developers to predict their effort and quality outcomes. |
| | Select SPIs that demonstrably increase software quality. |
| | Select SPIs that demonstrably improve developer productivity. |
| | Select SPIs that have clear fit with the tasks and implementation environments on which they are to be used. |
| **Training Guidelines** | Train developers on how to determine when use of the SPI is most appropriate. |
| | Train developers on the best process for using the SPI effectively. |
| | Demonstrate to developers the positive outcomes from SPI in terms of predictability of outcomes, increased software quality, and increased developer productivity. |
| | Set expectations for developers for continuous learning via SPI use. |
| **Implementation Environment Guidelines** | Involve developers in the adoption decision. |
| | Provide clear standards and practices for applying the SPI in the organization. |
| | Allow voluntariness for developers early, but not later in the implementation process. |
| | Promote and assess both satisfaction and use as desired outcomes of SPI diffusion. |

- **Predictability of Outcomes.** If developers are better able to predict their effort and the quality outcomes from using an SPI, then both their satisfaction and use of the SPI will be increased.
- **Increased Software Quality.** SPIs that are perceived to increase software quality are more likely to be used. Green et al. (2005) find that perceived quality was the most important factor for developers in their perceptions of the usefulness of an SPI. Therefore emphasizing SPIs that incorporate techniques such as walkthroughs, inspections, peer reviews, mathematically-driven designs, formal testing, and other quality focused techniques can increase the likelihood that developers will find the SPI useful, and therefore will continue to use the SPI.
- **Increased Development Productivity.** Software development techniques that are perceived to improve a developer's productivity are more likely to be used.

- **Task-Technology Fit.** While not measured directly in our research model, in their qualitative comments, developers identified issues of task-technology fit as critical to their perceptions of SPI usefulness. Rather than mandating SPI use in all development environments, SPIs should be targeted to those specific development environments that may benefit the most from sustained use (e.g., for PSP, this may include medium-to-large development projects with clearly-defined tasks).

## Guidelines for SPI Training

Once selected, developers need in-depth training on how and when to use the SPI effectively. As with any software tool, developers require training on new SPIs so that they can employ the SPI effectively and with relative ease. Green and Hevner (2000) noted that training occasions present an ideal opportunity to influence the expectations of software developers. Thus, we feel it important that managers offer adequate opportunity for SPI training to ensure successful use of the SPI in their development organizations.

This training must help developers control their development environment when using the SPI. Because this research found that software developers desire more control over the process of determining *when* it is appropriate to use a particular SPI, training should address how developers should make the determination about which development efforts will benefit from SPI use. On the other hand, developers desire more directed instruction in *how* to use the SPI. Our research indicates that they do not want personal control over the process of using the innovation. Behavior modeling training, which in this context would include demonstrations of how to decide when to use the SPI combined with hands-on practice with using it, should be the most effective approach for these specific training goals (Gist et al., 1989).

Another aspect of the special training needs for a new SPI is a focus on key benefits from using the SPI. Training materials should include evidence that the software development technique will enhance software quality and productivity, and will increase the predictability of those quality outcomes and the developer effort required to achieve them. This is particularly important if during the selection of the SPI, managers see that the predictability or nature of outcomes from using an SPI is not obvious or inherent in the technique.

A third, special aspect of training on an SPI is the need to set expectations for developers about their continued learning from using the SPI. Green et al. (2005) find that many software developers who sustained use of SPIs such as PSP considered themselves more skilled estimators, planners, and coders as a result of this use. It is probable that a major factor in this is the feedback to developers on actual development processes and outcomes. Training is an ideal occasion for discussing continuous learning and improvement in software development, and the role of an SPI in that process.

Interestingly, our research did not find a significant relationship between ease of use of the SPI and either perceived usefulness or eventual use of the innovation (Green et al., 2005). We believe that ease of use factors do support more rapid training on a SPI. However, ease of use factors should never be given a higher priority than factors that enhance quality and productivity in the SPI according to our findings.

## Guidelines for the Implementation Environment

The implementation environment also has an impact on the satisfaction with and use of a software development technique, in particular in terms of how it impacts developers' perceptions of control. In this research several specific tactics employed in implementation were found to increase satisfaction and use of the SPI.

- **Involve Software Developers in the Adoption Decision.** Developer choice in the adoption decision for an SPI is associated with higher levels of developer satisfaction with and sustained use of the SPI (Green et al., 2004), and developer choice is increased when developers are involved in the adoption decision. This involvement may take the form of identifying areas that need improvement or tools and techniques that help with that improvement. Involvement may also include determining objectives and analyzing costs and benefits for the innovative technique. However, managerial direction on the choice in adoption, through champion support, decreases developers' perceptions of choice and reduces satisfaction, so champion support should be made less visible to developers during the decision to adopt a SPI.
- **Provide Clear Standards and Practices for Applying the SPI in the Organization.** While the amount of software developer control over the SPI adoption and use is strongly related to satisfaction with SPI use, Green et al. (2004) found that not all kinds of control have the same impact on developer satisfaction. Having too much freedom in applying SPI techniques to development tasks was found to have a negative effect on software developer's satisfaction with, and sustained use of, the SPI. This may be because such freedom increases task complexity in the already unstructured and highly complex task of software development. Therefore either the SPI should embody disciplined software engineering practices, or such structure and discipline should be created via managerial direction on the process of using the technique.

- **Allow Voluntariness Early, but Not Later in the Process.** When use of a software development technique is voluntary, developers' perceptions of their choice in adoption are increased. As discussed above, increased choice in adoption is associated with higher satisfaction with the technique. However, as expected, more voluntariness of use is associated with lower levels of use of a software development technique. These complex relationships mean that managers need to be careful about when to provide more or less direction in the implementation process. Early in implementation, when the software development technique is being selected, managers may want to give developers high levels of involvement and more freedom of choice. After adoption, managerial direction may be needed to structure the process of use (if it is not inherent in the technique) and to make use more or less mandatory.
- **Consider Both Satisfaction and Use.** The key outcomes from using a software development technique, Satisfaction and Use, are also highly related. Use is an obvious, important issue whenever an organization spends resources on a new technique, but developer Satisfaction is also key to the introduction of an innovation. Not only are *satisfaction* and *use* significantly related in this study, but satisfaction with an innovation by early adopters is critical for subsequent diffusion. As mentioned earlier, developers who are satisfied with a software development technique are more likely to serve as opinion leaders who will influence potential later adopters of a technique.

## Directions for Future Research

Future research to investigate the diffusion and implementation of SPIs will address several of the limitations of the research reported in this chapter and test some of the managerial interventions that are recommended in Table 3. In particular, the impact of the degree to which the SPI is perceived to fit development tasks should be quantitatively examined to determine its impact on SPI diffusion. This factor, while not measured directly in the survey, was a key factor in SPI diffusion identified in the qualitative analysis of survey data. This finding is congruent with prior research on task-technology fit, which finds that the greater the correspondence between users' tasks and the functionality of systems that support those tasks, the more positive the users' evaluation of the system and their task performance (Goodhue, 1995; Goodhue & Thompson, 1995).

A key limitation of the current research is that it is a one-time, cross-sectional survey that investigates use of and satisfaction with one SPI. Future, longitudinal research can explore factors that influence the continued and discontinued use of an SPI, which may reveal additional management guidelines and elaborate on our theories of use. In particular, there is an opportunity to measure use of an SPI by depth as well as frequency of use of the innovation. Additional studies that investigate factors in satisfaction with and use of more than one SPI can test whether our findings vary across SPIs. In such a multi-SPI study, researchers will need to be careful to select SPIs whose features differ in some specified way (e.g., adoption by an individual or adoption by a team; amount of structure provided by the SPI) so that the results can be understood theoretically.

Future research will also test the recommendations that are based on this research effort. The effectiveness of all three categories of managerial interventions (selection, training, and implementation practices) can be investigated in the field. Such research will enable IS managers to design the best environment to facilitate the success of SPIs in their organization.

## References

Adams, D., Nelson, R., & Todd, P. (1992). Perceived usefulness, ease of use and usage of information technology: A replication. *MIS Quarterly, 16*(2), 227-248.

Aldag, R., & Grief, A. (1975). Employee reactions to job characteristics: A constructive replication. *Journal of Applied Psychology, 60*(2), 182-186.

Ashforth, B., & Saks, A. (2000). Personal control in organizations: A longitudinal investigation with newcomers. *Human Relations, 53*(3), 311-339.

Baronas, A., & Louis, M. (1988). Restoring a sense of control during implementation: How user involvement leads to system acceptance. *MIS Quarterly, 12*(1), 111-124.

Chau, P. (1996). An empirical investigation of factors affecting the acceptance of CASE by systems developers. *Information and Management, 30*(6), 269-280.

Constantine, L. (1995). *Constantine on Peopleware*. Englewood Cliffs, NJ: Yourdon Press, Prentice-Hall PTR.

Davis, F. (1989). Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly, 13*(3), 319-339.

DeLone, W., & McLean, E. (1992). Information systems success: The quest for the dependent variable. *Information Systems Research, 3*(1), 60-95.

DeMarco, T., & Lister, T. (1999). *Peopleware: Productive projects and teams* (2nd ed). New York: Dorset House Publishing Co.

Doll, W., & Torkzadeh, G. (1989). A discrepancy model of end-user computing involvement. *Management Science, 35*(10), 1151-1171.

Fayad, M., Tsai, W., & Fulghum, M. (1996). Transition to object-oriented software development. *Communications of the ACM, 39*(2), 109-121.

Fichman, R. (2000). The diffusion and assimilation of information technology innovations. In R. Zmud (Ed.), *Framing the domains of IT management* (pp. 105-127). Cinncinati, OH: Pinnaflex Press.

Fichman, R., & Kemerer, C. (1997). The assimilation of software process innovations: An organizational learning perspective. *Management Science, 43*(10), 1345-1363.

Fowler, P., & Levine, L. (1993). A conceptual framework for software technology transition. *Software Engineering Institute, Carnegie Mellon University*, CMS/SEI-93-TR-031.

Fox, M., Dwyer, D., & Ganster, D. (1993). Effects of stressful job demands and control on physiological and attitudinal outcomes in a hospital setting. *Academy of Management Journal, 36*(2), 289-318.

Gist, M.E., Schwoerer, C., & Rosen, B. (1989). Effects of alternative training methods on self-efficacy and performance in computer software training. *Journal of Applied Psychology, 74*(6), 884-891.

Goodhue, D. (1995). Task-technology fit and individual performance. *MIS Quarterly, 19*(2), 213-236.

Goodhue, D., & Thompson, R. (1995). Understanding user evaluations of information systems. *Management Science, 41*(12), 827-1844.

Green, G., Collins, R., & Hevner, A. (2004, February). Perceived control and the diffusion of software development innovations. *Journal of High Technology Management Research, 15*(1), 123-144.

Green, G., & Hevner, A. (2000, November/December). The successful diffusion of innovations: Guidance for software organizations. *IEEE Software, 17*(6), 96-103.

Green, G., Hevner, A., & Collins, R. (2005, June). The impacts of quality and productivity perceptions on the use of software process innovations. *Information & Software Technology, 47*(8), 543-553.

Greenberger, D., & Strasser, S. (1986). Development and application of a model of personal control in organizations. *Academy of Management Review, 11*(1), 164-177.

Hackman, J., & Oldham, G. (1976). Motivation through the design of work: Test of a theory. *Organizational Behavior and Human Performance, 16*(2), 250-279.

Henderson, J., & Lee, S. (1992). Managing I/S design teams: A control theories perspective. *Management Science, 38*(6), 757-776.

Iivari, J. (1996). Why are CASE tools not used? *Communications of the ACM, 39*(10), 94-103.

Karahanna, E., Straub, D., & Chervany, N. (1999). Information technology adoption across time: A cross-sectional comparison of pre-adoption and post-adoption beliefs. *MIS Quarterly, 23*(2), 183-213.

Langer, E. (1983). *The psychology of control.* Thousand Oaks, CA: Sage Publications.

Leonard-Barton, D. (1987). Implementation structured software methodologies: A case of innovation in process technology. *Interfaces, 17*(3), 6-17.

Lim, K., & Benbasat, I. (2000). The effect of multimedia on perceived equivocality and perceived usefulness of information systems. *MIS Quarterly, 24*(3), 449-479.

Luqi & Goguen, J. (1997). Formal methods: Promises and problems. *IEEE Software, 14*(1), 73-85.

Manz, C., & Sims, H., Jr. (1980). Self-management as a substitute for leadership: A social learning perspective. *Academy of Management Review, 5*(3), 361-367.

Mathieson, K., Peacock, E., & Chin, W. (2001). Extending the technology acceptance model: The influence of perceived user resources. *The DATABASE for Advances in Information Systems, 32*(3), 86-112.

Moore, G., & Benbasat, I. (1991). Development of an instrument to measure the perceptions of adopting an information technology innovation. *Information Systems Research, 2*(3), 192-222.

Nunnally, J. (1978). *Psychometric theory.* New York: McGraw-Hill.

Parsons, T. (1960). *Structure and process in modern societies*. New York: Free Press.

Pfleeger, S., & Hatton, L. (1997). Investigating the influence of formal methods. *IEEE Computer, 30*, 33-43.

Riemenschneider, C., Hardgrave, B., & Davis, F. (2002). Explaining software developer acceptance of methodologies: A comparison of five theoretical models. *IEEE Transactions on Software Engineering, 28*(12), 1135-1145.

Rogers, E. (1982). Information exchange and technological innovation. In D. Sahal (Ed.), *The transfer and utilization of technical knowledge* (pp. 105-123). Lexington, MA: Lexington Books.

Swanson, E., & Ramiller, N. (1997). The organizing vision in information systems innovation. *Organization Science, 8*(5), 458-474.

Tetrick, L., & LaRocco, J. (1987). Understanding, prediction, and control as moderators of the relationships between perceived stress, satisfaction, and psychological well-being. *Journal of Applied Psychology, 72*(4), 538-543.

Yourdon, E. (1996). *Rise and resurrection of the American programmer*. Yourdon Press, Prentice Hall PTR.

## Endnote

[1] PSP is a Service Mark of the Software Engineering Institute at Carnegie Mellon University.

## Chapter IX

# Improving Quality through the Use of Agile Methods in Systems Development: People and Values in the Quest for Quality

Julie E. Kendall, Rutgers University, USA

Kenneth E. Kendall, Rutgers University, USA

Sue Kong, Rutgers University, USA

## Abstract

*We introduce, define, and elaborate on agile development methods and how quality information systems are created via the values and practices of people using agile approaches. Key differences among agile methods, the SDLC, and other development methodologies are covered and suggestions for improving quality in IS through agile methods are given. We recommend adopting the principles of agile methods, encouraging more education about the values of agile approaches, including more types of people in the* *development of agile systems; adopting agile methods for a variety of organizational cultures; and renaming agile methods to signify the value system inherent in the approach.*

## Introduction

The traditional approach to analyzing and designing information systems to meet an organization's information requirements and to ensure quality in a completed information system is structured systems analysis and design. A systems analyst typically uses a methodology that is based on a systematic approach to engage the organization's systems problems. During interactions with a client organization, the analyst is occupied with identifying problems, opportunities, and objectives; analyzing the information flows in organizations; and designing formal computerized information systems and informal, human procedures to solve problems.

Although technical competence is essential, over the years it has become apparent that systems analysts need oral and written communication skills in order to enable them to interact with a myriad of managers and users, as well as other team members and programmers.

The systematic approach analysts most frequently use is called the systems development life cycle (SDLC). This life cycle is usefully divided into seven phases (some of which can be accomplished at the same time.) The seven phases of the SDLC are comprised of activities undertaken by the analyst to identify information systems problems, opportunities, and objectives; determine information requirements; analyze system needs; design the recommended system; develop and document software; test and maintain the system; and implement and evaluate the functionality of the system through pre-agreed quality measures (Kendall & Kendall, 2005).

Although use of the SDLC methodology (or some variation of it) is responsible for the implementation of thousands of information systems and in spite of the SDLC being the most widely acknowledged, taught, and adopted approach to developing information systems worldwide, it has often been criticized, even among its own practitioners. Their criticisms point to the rigidity of the phases of the SDLC; its insistence on formal process specifications; its clumsy, encumbered response to requests for changes once initial requirements have been derived; and the long gestation period required for a project using SDLC to move from inception through to completion and evaluation.

The objectives of this chapter are to introduce, define, and elaborate on agile methods in comparison with the prevalent structured development methods of the SDLC. We examine how quality is achieved through values and practices of people using agile development methods. This approach is defined in their philosophy, refined in project decisions, and applied in practice. We explain what differentiates agile methods from other approaches; elaborate on the values and practices of the people who use agile methods; make recommendations and provide solutions concerning the improvement of quality in IS through agile methods; and raise future research issues regarding the relationship of quality and the use of agile methods to develop information systems.

## Definitions of Quality

Since this entire volume is devoted to examining quality and IS, we do not wish to belabor definitions of quality. However, we can provide a rapid sketch of some of the guiding concepts that we will want to return to at various points in this chapter.

Over the years, the following quality attributes for software have come to be generally accepted and we endorse them here: efficiency, integrity, reliability, survivability, ease of use, correctness, maintainability, expandability, flexibility, interoperability, portability, reusability (Keller et al., 1990) when considered in the light of adoption, how well the system performs, how valid the design is for what the organization is attempting to accomplish and how adaptable the software is to growth, change, and reorganization in a business. The tapestry of ideas that gave a coherent pattern to these ideas is the field of quality management and its contribution is worth describing briefly here.

Quality management originated from process control practices in the manufacturing industry, which originally set out to reject manufactured parts that were in some way defective. This evolved into quality control efforts that used statistical controls to control for the number of defects per batch produced in products. (Service quality came later.) However, the growing realization that quality is not tested, but rather created, saw the birth of the concept and practice of quality assurance. This approach to quality is based on the evaluation of the production *process* instead of the finished product itself. The rationale behind it is that operational excellence delivers conformance quality. Disciplined operational process can deliver persistency and conformance quality by scientifically and systematically monitoring and controlling the variance-production activities (Juran, 1999, 2004).

Based on this belief, the Software Engineering Institute (SEI) introduced the capacity maturity model for software (CMM-SW or CMM) in 1987, which is a quality management system for the software industry (Paulk et al., 1995). Other quality management systems (QMS) such as ISO 9000, MB and Six Sigma can also be applied to the software production industry. The ISO 9000 certification is roughly equivalent to CMM level 3 (Tingey, 1997). Therefore, when practitioners used the SDLC or other structured methods and wanted to entertain the notion of quality, it typically had to do with process quality.

The idea of deriving quality from the practices and values of people working on systems projects had to wait for the introduction of agile methods to gain a voice in the information systems quality debate. The people involved in agile methods comprise a wide cast of characters including customers, programmers, systems developers, and managers.

One way to usefully discuss quality is to talk about it as equivalent to customer satisfaction. Others have discussed the concept of "fitness for use" as an alternative definition for quality. In turn, we might define our customers or clients of the information systems development effort as anyone who is effected by the IS development process or the IS software or system product. Products of the development effort include the relationships between customers and developers, the software itself, and the service provided to maintain the relationships as well as the system and software.

Determining customer satisfaction with information systems can be examined with two criteria: (1) the features of the software or system itself and (2) the lack of defects in the product (Gryna, 2001; Juran, 1999). When we speak of software product features, it can be thought of usefully as the quality of design. Often as the quality of design increases, so do the costs involved in achieving it (Crosby, 1979).

An absence of defects can also impact costs in important ways, such as reducing developer time spent in recoding and subsequent retesting; cutting down on the time developers must spend with users during successive iterations to correct errors; and reducing or eliminating user complaints (Boehm, 1981). Eventually, developing software and systems that are acceptable in their quality on a consistent basis over an extended period can lower costs.

As we examine these attributes, we find that quality in information systems is highly dependent on the practices and values of the people who develop and implement the systems. This is where agile methods stake a claim and how they differ in a meaningful way from the structured methodologies (Cockburn & Highsmith, 2001). With these quality concepts at the ready, we now turn our attention to elaborating on systems development approaches and their relationship to quality.

## Systems Development Using Alternative Methods

Structured systems development methods, referred to as the systems development life cycle approach (SDLC), the waterfall approach, and recently object-oriented approaches have been the main methods for developing systems. Unfortunately, some researchers believe that the drawbacks of the structured approaches have become evident in the poor quality performance of software projects. A decade ago, less than 17% of projects were deemed successful (Standish Group International, Inc., 1995). The remaining 84% were either halted altogether, completed late, or they exceeded their initial budget requests. The same report highlighted the main reasons software projects succeeded: involvement of customers; support from management; clear requirements definitions; solid planning for the future of the project and organization; and realistic expectations as far as goals to be achieved and the timeframe in which to achieve them (Standish Group International, Inc., 1995). In our appraisal, we see the critical role that humans play in each of the activities enumerated as contributing to success.

Partially in response to the criticisms of the structured approach cited earlier, partially due to overall demands for new types of information systems, and in some measure in response to the recognition of the importance of quality in manufacturing and the service sector, IT researchers and practitioners alike have been continually investigating and experimenting with a variety of new software development ideas and practices. These endeavors are meant to achieve faster delivery of systems and software at a lower cost and improved quality.

### *Prototyping and RAD*

Several other software development methods approaches have been proposed, including object-oriented development; RAD (Rapid Application Development); agile software development methodologies; and so on. The object-oriented approach is similar to the SDLC in the sense that it contains some of the same basic phases, although object-oriented systems design and development ideas and the workload proportion at each stage is very different. (For an in-depth comparison of structured methods and object-oriented methods, see Sircar et al., 2001.) Prototyping and RAD share many similarities.

Prototyping has also been proffered as an alternative development method that obviates some of the criticisms recounted earlier in this chapter concerning the use of the SDLC as a methodology. Complaints about going through the SDLC process center around two interrelated concerns: (1) the length of time required to complete the SDLC and (2) the dynamic nature of user requirements. Caught up in the lengthy cycle and far away in time from the changing organizational dynamics of a user's work situation, the resultant system may seem out of touch with what users find themselves doing at the time.

Some analysts propose that prototyping be used as an alternative method in place of the SDLC. Using prototyping exclusively as the development approach can shorten the time between determining user requirements and the point at which a functional system is delivered. Prototyping as a development method might also be used to communicate more closely and repeatedly with users, thus attempting to specify user requirements more accurately.

Over the years many practitioners lost interest in prototyping. Problems included lack of documentation and an emphasis on bottom-up development. The need for large-scale, top-down designed systems encouraged designers to use more structured approaches. After a period in which prototyping was out of favor, prototyping was reinvented as rapid application development (RAD) and later reinvented in extreme programming and agile methods. Although some suggest that agile methods might suffer the same fate as prototyping, their ability to evolve and adapt as practiced may help overcome some of failings of earlier approaches.

## How Alternative Approaches Differ from SDLC

All of the alternative approaches to systems analysis and design differ from the SDLC methodology in common ways, including: (1) emphasizing short releases of working versions; (2) an iterative development process (this entails adding more features or altering the previous version based on customer's feedback on the next iteration); and (3) less documentation and formality (Fowler, 2003). However, it is agile methods in particular that refocused the quality issues of development around people, who embody particular values and carry out specific practices, and effectively moved the focus away from the preoccupation with formal process specification that drained structured methods of much of their responsiveness and humanness.

This chapter will specifically take up the promise, challenge, and frustration inherent in agile methods as they relate to developing quality software and quality information systems, and as they compare and contrast to the structured methods which follow the systems development life cycle.

# Agile Methods

Agile methods were developed roughly a decade ago, in the early 1990s. Initially, they were called lightweight methodologies vs. the heavy weights of the traditional life cycle methodology. At about the same time, Beck and Andres (2004) proposed extreme programming (XP), which is the most highly visible of the agile methodologies. Other agile methods include scrum (Schwaber & Beedle, 2002), open source development (Sharma et al., 2002), feature driven development (Coad et al., 1999), crystal family (Cockburn, 2002), and others (see for example work by Abrahamsson et al., 2002). The similarities inherent in this collection of approaches prompted their inventors and practitioners to form an organization called the "Agile Alliance" to promote their common values and key practices. They also reached a consensus to call their core software development values and practices "agile software development methods" (Agile Alliance, 2001).

## The Principles and Values of Agile Methods

Agile Methods are a set of software development methodologies which share common values and principles. Based on the belief that people are the most important factor for a project to succeed, agile methods emphasize the following unique principles in their development approaches: (1) real life communication takes precedence over following rules or procedures; (2) the final product is more important than documenting the process; (3) systems development is customer-centered rather than manufacturing-centered; and (4) adapting is the correct state to be in, not planning (Agile Alliance, 2001).

In contrast to traditional SDLC approaches, agile methodologies put people ahead of process based on the belief that human error is the main reason for any project failure. Proponents of agile methods believe that the most important thing in delivering quality systems within time and budget is that all individual stakeholders collaborate effectively with each other (Williams, 2000). It is above all an adaptive, human-centered approach which permits the creativity of humans to take over and develop solutions where formal process specification of every system nuance is not possible or desirable (Nerur et al., 2005).

The use of agile methods has been proposed as a way to adopt practices and values that will improve quality. In the upcoming sections we will explore the relationship between the use of agile methods, the quality of the development process, and the quality of information systems delivered to organizations. As with any development approach, understanding of the organization is a key to effective use. Systems analysts who proceed without proper knowledge of an organization's culture may actually diminish the quality of the project, rather than improve it.

Agile methods, similar to RAD and other iterative development methods, place emphasis on the use of short releases of a working version of a system or feature. In addition agile methods place emphasis on the iterative development process, with an expectation of quality improvement each time an iteration is completed. Additionally, agile methods are less formal and less formally documented than systems created with structured methods. Agile methods underscore the importance of people in improving the quality of information systems based on increased communication; a belief in the value of flexibility; and the prompt and on-going attention to systems.

## Agile Methods as a Philosophy

Practitioners using agile methods maintain a unique philosophical outlook; that is, people should be emphasized over process in the success of an information systems project. To that end, practitioners adopting agile methods value communication and collaboration among all stakeholders including the developers, management, and the client. Working together as a team with stakeholders has become a cornerstone of the agile methods approach. Those endorsing agile methods believe that since people are much more complicated systems (who experience social and spiritual needs; creativity; responsive to changes in the environment; and so on), they ought to be treated like humans rather than machines or replaceable parts. The philosophy maintains that a worker's creativity, flexibility, and productivity can be acknowledged and promulgated through human treatment.

This belief can be extended to almost all values and practices of the agile methods. For example, because customers are humans living in a real world, allowing them to change their requirements is reasonable and should be encouraged based on agile philosophy. Also, customers are encouraged to capture their requirements briefly in an indistinct way at the beginning of the project and to give more specific requirements later in a clearer picture because this is the nature of human learning, to be able to become more specific as one gains understanding (Highsmith & Cockburn, 2002). Also because developers are humans, allowing them to have a certain degree of technological or timing flexibility is necessary for them to pursue design and coding quality and creativity (Highsmith & Cockburn, 2001). Once again, the iteration approach fits the nature of human learning, which is a good workflow for the developer to refine his or her design ideas. In addition, they believe that better communication and collaboration among the stakeholders (management, developers, and customers) is much more important than signing contracts to restrict the customer or following certain plans or procedures to restrict the developer.

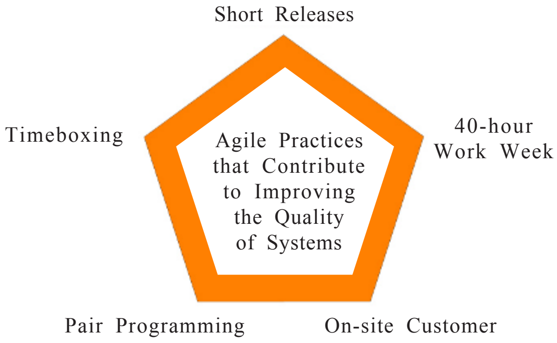
## Agile Core Practices

There are four agile core practices that guide how programmers should behave as people and team members during a systems project. They are markedly different from the SDLC approach discussed earlier.

Balancing resources to complete a project on time is one way to manage a project. The best way to manage, however, is to develop practices that yield outstanding results. The agile methods and extreme programming movement has developed a set of core practices that have changed the way systems are developed. The four core practices are shown in Figure 1.

1. **Short releases.** This means that the development team will shorten the time between the releases of their software. A release, without all of the features, may be released quickly, only to be followed by an update or change in short succession. The practice of short releases means that the team will develop critical features first, and that the product will be released with the most important features first. Later on the team will improve it.
2. **40-hour work week.** This core practice asserts that members should be committed to working intensely on their systems project and then they should take time off to prevent burnout, and increase effective performance while on the job. The agile methods culture underscores the importance of taking a long-term view of the health of its project and its developers.

*Figure 1. Agile methods consist of unique practices that can be used effectively to develop quality systems*



3. **On-site customer.** Rather than settling for perfunctory site visits traditionally made by systems analysts, the on-site customer practice fosters deep and on-going communication between developers and customers by requiring a business worker to work on-site during the entire time. The on-site customer helps in formulating the strategic focus of the project, as well as in practical matters such as what should be a high priority feature.
4. **Pair programming.** Programming in pairs as a core practice means that two programmers who choose to program together on a project are involved in running tests, communicating about efficiency and effectiveness of how to do the job. The ability to bounce ideas off of another programmer helps clarify each programmer's logic and thinking. Working as a team of two has benefits for the project and the programmers. Careless thinking is curtailed, creativity can be stimulated, and it can also save time.
5. **Timeboxing.** The overall time frame that developers have to finish a project is typically cut into many shorter time periods, called timeboxes (Dynamic System Development Consortium, 2004). Each timebox represents the time limit to finish each iteration and typically contains one or two weeks, depending on the specific project environment (project scope, project complexity, team size, team experience, project resource, and so on). When setting up the length of the timebox, management needs to consider the following: (1) it should fit developers' daily schedule so that it is comfortable for the developers to work with; (2) it should be long enough for the team to deliver core quality features in a working version and leave a little bit of buffering time so that developers can complete their tasks efficiently; (3) generally speaking, the shorter the iteration is, the better because a short timebox will pressure the developers to focus on the core design issues and not waste time on unnecessary tangential elements. Also the shorter the timebox, the greater number of iterations you can have within the overall timeframe, which can bring more customer input and hence assure the delivery of a high quality system.

Agile methods use these core practices to head off many of the common complaints leveled against traditional methods when they result in projects of poor quality or fail altogether. Developers are blamed for not communicating with clients or each other; unnecessarily increasing the scope and complexity of projects; or harboring fears of failure. The agile approach handles this situation by insisting from the outset that analysts and customers share values that enable them to communicate freely, agreeing to the essential features and making adjustments as the project progresses.

# Pair-Programming: An Example of an Agile Methods Practice

In order to better understand agile methods in practice, we will examine one of the core practices in depth. In this way we can reveal the usefulness of pair programming as it specifically relates to developing quality information systems.

Pair programming is a key core practice that separates agile programming from the SDLC approach. Pair programming, if done properly, can ensure better quality information systems (Wyssocky, 2004). Since, in this approach, all software code is produced by a team of programmers working together on the same computer, each programmer provides a double check for the other one.

## A Team Approach to Programming

Pair programming implies that each programmer work with another of their own choosing. Each does coding; each runs tests. Often the senior person will take the coding lead initially, but as the junior person becomes involved, whoever has the clearest vision of the goal will typically do the coding for the moment. A programmer invited by another to work with them in this way is obligated to do so, according to the protocol of pair programming. Working with another programmer helps each person clarify their thinking. Pairs may change frequently, especially during the exploration stage of the development process. This, too, can help programmers gain a fresh view of their coding habits.

One programmer may verify the accuracy of the code, while the other may concentrate on making the code simpler so that the code has more clarity. Pair programming saves time, cuts down on sloppy thinking, sparks creativity, and is a fun way to program.

Another approach to pair programming is to have one programmer write the first version, while the second focuses chiefly on the quality of the code. The second programmer therefore does not try to add features. Rather the second programmer attempts to verify, test, and improve the code.

## Advantages to Pair Programming

One of the advantages of pair programming is that it can maintain the high quality of the system, while saving a considerable number of days or weeks. For example, a single programmer who needs to meet a deadline may begin to work long overtime hours. This is a dangerous practice, since it is recognized that quality decreases when people become tired. Pair programming alleviates the problem, because the first programmer has to leave in order for the second programmer to use the common computer.

Another advantage becomes obvious when one realizes at what point in the programming errors are detected. Using pair programming practices, errors are caught immediately (Williams, 2000). In contrast, when using a traditional approach to code verification, errors are discovered after the programming is almost complete. Sometimes errors are found only after it is too late to correct them and deliver the project on time. As one can see, pair programming helps in delivering quality systems within the time frame agreed upon by programmers and managers.

Other advantages include the training aspects of pair programming. An inexperienced programmer can learn from a master if they are teamed up along the dimension of their expertise. One can also argue that a programmer will be more careful when someone else will be inspecting the code. Consequently, codes developed by pair programming teams more often than not turn out to be better designed, freer of errors, easier to follow, and simpler rather than more complex (Williams, 2000).

## Some Disadvantages to Pair Programming

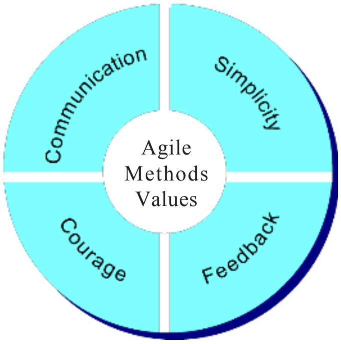
Pair programming, however, is not without its disadvantages. Just as the saying goes, "A person who defends himself has a fool for a lawyer," we can see how pair programming might break down. If both programmers are highly conversant with the system they are developing, they might be unable to identify potential problems (Stephens & Rosenberg, 2004). Traditional code verification looks at the software from a new perspective. That is something pair programmers might not be able to emulate; they may not have adequate distance from the coding to gain a proper, critical perspective.

Finally, it is also obvious that quality would suffer if programmers were not happy in their careers or particular jobs. From our observations (and we have practiced pair programming ourselves), we note that pair programmers seem to be satisfied with this practice and contented with the arrangements.

## Values of Agile Methods

At the heart of the agile methods philosophy is the set of values that embody all of the unique positive elements that make projects developed using agile methods successful (Kendall & Kendall, 2005). The set of values, which is comprised of communication, simplicity, feedback, and courage, is shown in Figure 2.

*Figure 2. Values are very important in the development of quality systems when adopting agile methods*



## How Values Shape Project Development

Let's begin with *communication*. Typical agile methods practices, such as pair programming, interfacing with an on-site customer, and doing short releases, all depend on good communication. However, on the face of it, good communication is something that must be worked at and not assumed. There are many potential pitfalls in a systems project: specialized vocabulary; tight deadlines; constant changes in features, scope, and personnel; a tendency to prefer independent work over teamwork; and pressure in meeting short release deadlines. With good communication in place, problems can be remedied quickly, holes can be closed, and shaky logic is strengthened through interaction with teammates and clients.

*Simplicity* is a second value of agile methods. It is easy to be overwhelmed and ultimately paralyzed by indecisiveness in the face of a new, complex systems project where every detail seems important. Beginning with the simplest possible thing we can do is the essence of this value. But simplicity is a learned value and is certainly an art. With the value of simplicity in mind, developers start small, aware that they may have to change their work a little the next day. This basic value demands a clear vision of what the project goals are.

When systems development is approached with agile methods, *feedback* is a third basic value. Feedback and time are intertwined in this context. Programmers, analysts, and clients can provide useful feedback anytime (within seconds of first seeing an application all the way to months after working with a system) depending on what feedback is required, who will be exchanging information, and the ultimate use of the feedback that is given. Clients provide feedback when they create functional tests for all of the features that programmers have implemented. Feedback from clients also helps developers gauge progress compared to initial project goals. Feedback is vital to helping developers make early changes while allowing the organization the opportunities to experience the system early on.

The fourth value of agile methods is *courage*. The value of courage encourages the team to experiment in order to reach their goals rapidly, or perhaps in a novel way. If courage is embraced, team members trust each other to work continuously to improve project quality. This is a big commitment since it can entail scrapping current code, rethinking solutions that have already been accepted, or pursuing simplification of one's approach even further. Courage also implies that systems developers earnestly apply the extreme practices of agile methods.

Beck and Andres (2004) consider one more element to be a value: respect. This value must be built up in the development team so that team members trust each other's instincts on whether there is a better approach than the one currently being pursued, one that will be more straightforward in accomplishing a goal. Of course this is true for any project, no matter by whom, how, or why it is developed.

## Accepting Change is Important

A basic principle arising from agile methods values is that of embracing change. Options must all remain open, but the point is to solve whichever problem poses the biggest threat or hindrance to the effort's success. In spite of always handling tradeoffs, we will always be able to count on the idea that change is welcomed. That dynamism keeps the project moving forward and animates the spirit of the project team.

Performing quality work must come from the practices and values of agile methods. This idea that everyone wants to perform quality work is integral to the use of agile methods. Practitioners of agile methods want everyone to enjoy their work, work well with the team, and keep the momentum of the project at a high level.

## Resource Tradeoffs are Possible

If systems developers are to maintain quality in the systems they develop, we need to examine some of the tradeoffs they must face. Since communication with the customer is at the foundation of agile projects, systems developers are at an advantage over traditional project managers since they can use feedback from customers to determine how much of a particular resource is needed.

In order to ensure quality, a developer may want to ask for more time to finish the project. The agile approach does not recommend that deadlines be extended. Following the agile methods approach means completing the project on time. Of course, there are other ways to extend the time on a project. Programmers can work overtime, but once again the agile approach discourages this practice. Programmers who work overtime to complete a project may not help either, since errors increase as sleep decreases, and then time needs to be spent fixing errors (Abdel-Hamid & Madnick, 1990; DeMarco, 1982).

The next variable to consider adjusting is that of cost. The easiest way to increase spending (and hence costs) is for a project manager to hire more people. The equation seems on its face to be correct; hiring more programmers equates with finishing the project faster. However, communication and other intangible costs increase when adding team members. They may slow more experienced team members as they learn the ropes, delaying the project even further (Brooks, 1978). This is well known as Brooks' Law. Rather than adding more programmers, increases in productivity can be gained from investments in support tools for systems developers to improve communication of ideas and even boost productivity away from the development site.

We last consider the variable of scope. A developer may decide to release a version of the system on time, but not deliver all of the features promised (Crispin, 2001). In this case, the developer concludes that the customer would rather not sacrifice quality. The features that were cut out can be added in a later release, but for the time being, the project is presented to the customer for feedback and quality is maintained.

## Maintaining Quality is Critical

It is abundantly clear that systems must not forego quality, as the consequences could be widespread and long lasting. (For example, purchases with a credit card are erroneously rejected as invalid, or customers may show up at the airport and be denied boarding due to lack of reservations for a flight.)

The on-going theme of agile methods is that they permit developers to take extreme measures, so in order to maintain quality, manage cost, and complete the project on time, the agile analyst may seek to adjust the scope of the project by enlisting the help of a client in figuring out what features can be deferred until it is practical to include them in the next release. Systems developers using agile methods approaches are able to control time, cost, quality, and scope. Guided by the realization that agile methods sanction extreme measures, while at the same time emphasizing completion of a project on time, the analyst must recognize tradeoffs and make difficult decisions as the project takes shape.

A successful, quality system created with agile methods will be one that delivers a useful, quality solution for the organizational situation posed via a creative and adaptive human-centered process; a flexible and adaptive solution that is on time and on budget with desirable features that are jointly prioritized and developed by customers along with the development team in a collaborative spirit that fully recognizes the dynamic nature of information systems requirements and organizations in a modern social and technological setting.

## Concerns about Using Agile Methods

One of the charges leveled against agile methods is that they cannot reach a high quality standard because they lack quality management systems. Our belief is that this is a misunderstanding of agile methods in action. Although agile methods focus on shortening the development time and working within the budget, these should not be construed as steps to diminish quality in any way. Rather, a high level of customer involvement to change and prioritize requirements helps to deliver the quality that customers want, the functional quality, which is quality from the customer perspective.

Employing good, well-qualified people to use agile methods in developing information systems is critical. Although much of the work of systems analysts and programmers can be automated, the efficiencies and effectiveness to be gained from agile methods as they relate to quality are uniquely attributable to humans. When the right group of people is motivated to do the right type of work, their productivity can be dramatically improved.

In the upcoming section we examine future research questions arising from our examination of information systems quality and the use of agile development methods. Many of these deserve further attention as adoption and awareness of the possibilities for quality improvement through the use of agile methods increases.

# Future Research Questions

One rather large difference between the SDLC and the agile approach is that SDLC is top-down, strategic, and very normative. Agile methods, on the other hand, are still bottom-up, responsive, and reactive. One can argue that in our modern society, where we discard our old cell phones because new models are smaller with more features, responsive reactive methods will continue to flourish.

The other side of the argument is that there is the possibility that quality can be sacrificed to make way for new features and more attractive design. To cite the cell phone example again, rarely do we find a customer trading in a phone to get one of better quality. Rather, they seek new features or improved design. Those using agile methods need to be constantly aware of quality demands, and not trade them off for features and more attractive interfaces.

Some future research questions include the following:

1. **Human factors:** How should we effectively get the customer to articulate quality concerns, objectives, and measures? Is systematic quality training across all management levels a cost effective practice for projects that adopt agile methods? How should we train pairs of programmers to effectively cooperate with each other toward the overall goal of quality? How should we effectively encourage the agile methods development team to espouse the same high quality standards? How do we manage and motivate the developers to achieve extremely high quality levels given the tradeoffs imposed by short releases?
2. **Process factors:** What steps must be followed by an agile practitioner to ensure that the quality of the system receives the highest attention? What process control methods (statistical or other) should be used to track and measure quality of management practices? How much documentation is proper to ensure continuous quality improvement? How can the software development process be improved to ensure better quality?
3. **Overcoming limitations of agile methods:** Since agile methods take a bottom-up approach to systems design, how does one align the specific agile quality measures with the organization's long-term strategy? How do we allow the agile core values to evolve? Can we adapt agile methods so we can use agile practices to develop high-quality, large-scale projects? Will differences in organizational cultures affect the quality of the system when using agile methods? How can we manage the inconsistency caused by constant change to ensure the internal quality, even with life critical systems?
4. **Quality improvement factors:** When quality management systems like Six Sigma or ISO 9000 are incorporated into the agile approach, will the quality of the projects delivered by agile methods improve dramatically? Will the obstacle imposed by minimal necessary documentation using agile methods infringe on the quality of the system? How do we effectively track and measure quality attributes during the development process?

## Conclusions and Recommendations

In conclusion, we believe that the thoughtful and intentional use of agile methods by systems developers and programmers can provide an excellent approach to boosting the quality of program code; the quality of the process of software development; and also serve to contribute to the overall quality of information systems installations. We also contend that agile methods are most powerful when selectively chosen for use with structured approaches that provide the vision and discipline to see large projects through to a quality outcome (Glazer, 2001; Nawrocki et al., 2002; Wright, 2003). We do not believe that an all or nothing approach to adoption of agile methods is suitable for systems development or the practice of coding. Rather we believe that experienced developers and programmers can select methods appropriate to the organization, the organization culture, and the systems tasks at hand to create a dynamic response to fluid situations they encounter (Duggan, 2004).

We assert that agile methods have in many respects lead the way to quality improvements by rightfully shifting the focus of programming and design teams to the human aspects of the software creation process, as well as their client-centered orientation toward design and implementation of information systems (Van Deursen, 2001).

As we look forward to the creation of ever more ubiquitous information systems, we are gaining an appreciation of the importance of making the system fit the human, rather than the other way around. We believe that if agile methods are adopted by good people, their use can help us reach our quality goals for information systems and software design far into the future.

After examining the values and practices, the pros and cons of agile methods, and how we can use these to foster high quality project development, we recommend the following:

1. **Adopt the values and principles of agile methods even though your client wants everything documented.** The necessity of completing all the steps in the SDLC or setting up an object-oriented system does not preclude the systems designer from using the principles of agile methods.

   Developers can still take advantage of shorter cycle times, both for gathering reactions from the customer and for internal review. Pair programming and shorter work weeks can still be practiced. Timeboxing and the principle of simplicity over complexity are ideal ways to develop projects at all times.
2. **Foster education in the values and principles of agile methods.** We need to expand the coverage of agile methods in our curriculum. By

interacting with actual organizations and actual systems developers, we infuse our classes with a lasting vitality, immediacy, and credibility. The entire field of quality was founded on the synergistic relationship between research and fieldwork.

Educators must understand that even though we teach designers and programmers how to properly use traditional methods to design, develop, and program systems, we need to realize what actually occurs in practice. If we do just that, we will find ourselves teaching agile methods, and our successful graduates will forever remember the values and principles that help ensure high quality projects.

3. **Include all types of people in the development of systems using agile methods.** We have all learned how important each person is from our experience with Japanese quality circles. But when we practice agile methods, we talk about the developer (or sometimes, just the programmer) and the on-site customer. The development team must include testers, trackers, coaches (an idealized project manager), consultants, the customer, the programmer, the business analyst, and of course, the big boss. Everyone involved will have a unique view. Since quality is multifaceted, this will encourage the best possible outcome.

4. **Create ways to better use agile methods in a variety of organizational cultures.** Agile methods, just like traditional methods, assume that systems developers come from a similar background, but in addition, agile methods assume that all organizations are alike. No mention is made in the agile manifesto about various organizational cultures and how to approach these cultures.

   Organizational cultures are important. In previous work, we concluded that certain information systems had higher probabilities of success if the predominant metaphors were favorable (Kendall & Kendall, 1993, 1994). Those embracing the agile philosophy can learn a great deal about an organization and its values by studying the metaphors they find within the organization. If systems developers are willing to spend the time and effort examining organizational cultures, they can ensure higher quality projects and ready acceptance.

5. **Consider renaming agile methods.** This might be our most controversial recommendation, but clearly names do carry consequences. Extreme programming is an alternative name that people either like or hate. Extreme programming best describes the practices of extreme programming, which is taking good practices, but carrying them to the extreme. One can assume that this includes quality as well. Believers like the principles, but not everyone likes the name.

The term *agile methods* suffers from misconceptions as well. The synonyms for the word agile are *nimble, supple, lithe, sprightly, alert, responsive, swift,* and *active*. Perhaps they accurately describe how programmers should rapidly develop projects, schedule short releases, and adapt quickly to the changing requests of customers. But in the final analysis, this is only one of the practices of agile methods, and quality is not addressed in any sense. It would be better to have a name that invokes the greatness of these methods.

Perhaps a better name for methods that deliver high quality projects emphasizing clients and users are "human-value" approaches. We might want to build on this idea, include more principles, and set standards for quality that humans appreciate in their systems.

## References

Abdel-Hamid, T., & Madnick, S. E. (1990). *Software project dynamics: An integrated approach.* Englewood Cliffs, NJ: Prentice Hall.

Abrahamsson, P., Salo, P. O., Ronkainen, J., & Warsta, J. (2002). *Agile software development methods* (vtt publications 478). Retrieved November 30, 2005, from http://www.inf.vtt.fi/pdf/publications/2002/P478.pdf

Agile Alliance. (2001). Manifesto for agile software development. Retrieved November 30, 2005, from http://www.agilealliance.org/home

Beck, K., & Andres, C. (2004). *Extreme programming explained: Embrace change* (2nd ed.). Addison-Wesley.

Boehm, B. (1981). *Software engineering economics.* Englewood Cliffs, NJ: Prentice Hall.

Brooks, F. P., Jr. (1978). *The mythical man-month.* Reading, MA: Addison-Wesley.

Coad, P., Lefebvre, E., & De Luca, J. (1999). *Java modeling in color with UML*. Upper Saddle River, NJ: Prentice Hall.

Cockburn, A. (2002). *Agile software development*. Boston: Addison-Wesley.

Cockburn, A., & Highsmith, J. (2001). Agile software development: The people factor. *Computer*, *34*(11), 131-133. Retrieved November 30, 2005, from http://csdl.computer.org/comp/mags/co/2001/11/ry131abs.htm

Crispin, L. (2001). *Is quality negotiable?* 2001 XP Universe. Retrieved November 30, 2005, from http://www.xpuniverse.com/2001/pdfs/Special02.pdf

Crosby, P. B. (1979). *Quality is free: The art of making quality certain*. New York: McGraw-Hill Book.

DeMarco, T. (1982). *Controlling software projects*. New York: Yourdon Press.

Duggan, E. (2004). Silver pellets for improving software quality. *Information Resources Management Journal, 17*(2), 1-21.

Dynamic System Development Consortium (2004). Retrieved November 30, 2005, from http://www.dsdm.org/version4/timebox_plan.asp

Fowler, M. (2003). *The new methodology*. Retrieved November 30, 2005, from http://martinfowler.com/articles/newMethodology.html

Glazer, H. (2001). Dispelling the process myth: Having a process does not mean sacrificing agility or creativity. *The Journal of Defense Software Engineering*. Retrieved November 30, 2005, from http://www.stsc.hill.af.mil/crosstalk/2001/11/glazer.html

Gryna, F. M. (2001). *Quality planning and analysis* (4th ed.). New York: McGraw-Hill Higher Education.

Highsmith, J., & Cockburn, A. (2001). Agile software development: The business of innovation. *Computer*, *34*(9), 120-122. Retrieved November 30, 2005, from http://csdl.computer.org/comp/mags/co/2001/09/r9120abs.htm

HighSmith, J., & Cockburn, A. (2002). *Agile software development ecosystems.* Boston: Addison-Wesley.

Juran, J. M. (1999). *Juran's quality handbook* (5th ed.). New York: McGraw-Hall.

Juran, J. M. (2004). *Architect of quality.* New York: McGraw-Hill.

Keller, S. E., Kahn, L. G., & Panara, R. B. (1990). Specifying software quality requirements with metrics. In R. H. Thayer & M. Dorfman (Eds.), *System and software requirements engineering* (pp. 145-163). Washington, DC: IEEE Computer Society Press.

Kendall, J. E., & Kendall, K. E. (1993). Metaphors and methodologies: Living beyond the systems machine. *MIS Quarterly*, *17*(2), 149-171.

Kendall, J. E., & Kendall, K. E. (1994). Metaphors and their meaning for information systems development. *European Journal of Information Systems, 3*(1), 37-47.

Kendall, K. E., & Kendall, J. E. (2005). *Systems analysis and design* (6th ed.). Upper Saddle River, NJ: Pearson Prentice Hall.

Nawrocki, J. R., Jasinski, M., Walter, B., & Wojciechowski, A. (2002). Combining extreme programming with ISO 9000. *Proceedings of the*

*First Eurasian Conference on Advances in Information and Communication Technology.* Retrieved November 30, 2005, from http://www.eurasia-ict.org/ConfMan/SUBMISSIONS/81-ertpospnaw.pdf

Nerur, S. R., Mahapatra, R., & Mangalaraj, G. (2005). Challenges of migrating to agile methodologies. *Communications of the ACM*, *48*(5), 73-78.

Paulk, M. C., Weber, C. V., Curtis, B., & Chrissis, M. B. (1995). *The capability maturity model: Guidelines for improving the software process*. Boston: Addison-Wesley.

Schwaber, K., & Beedle, M. (2002). *Agile software development with scrum*. Upper Saddle River, NJ: Prentice Hall.

Sharma, S., Sugumaran, V., & Rajagopalan, B. (2002). A framework for creating hybrid-open source software communities. *Information Systems Journal, 12*(1), 7-25.

Sircar, S., Nerur, S. P., & Mahapatra, R. (2001). Revolution or evolution? A comparison of object-oriented and structured systems development methods. *MIS Quarterly, 25*(4), 457-471.

The Standish Group International, Inc. (1995). *The Chaos Report (1994).* Retrieved November 30, 2005, from http://www.standishgroup.com/sample_research/chaos_1994_1.php

Stephens, M., & Rosenberg, D. (2004). Will pair programming really improve your project? In *Methods and tools*. Retrieved November 30, 2005, from http://www.methodsandtools.com/PDF/dmt0403.pdf

Tingey, M. O. (1997). *Comparing ISO 9000, Malcolm Baldrige, and the SEI CMM for software — a reference and selection guide*. Upper Saddle River, NJ: Prentice Hall PTR.

Van Deursen, A. (2001). Customer involvement in extreme programming: XP2001 Workshop report. *ACM SIGSOFT Software Engineering Notes, 26*(6), 70-73.

Williams, L. A. (2000). *The collaborative software process*. Unpublished doctoral dissertation, University of Utah.

Wright, G. (2003). Achieving ISO 9001 Certification for an XP Company. In F. Maurer & D. Wells (Eds.), XP/Agile Universe 2003. *Lecture Notes in Computer Science, 2753*, 43-50. Berlin: Springer-Verlag. Retrieved November 30, 2005, from http://www.springerlink.com

Wyssocky, L. (2004). Pair programming: Code verification. Retrieved November 30, 2005, from http://www.qualityprogramming.org/Implementation/CodeVerification/CodeVerification.htm

## Chapter X

# Quality Metrics and Bayesian Analysis: The Case of Extreme Programming

Francisco Macias, Technologic of Monterrey, Mexico

Mike Holcombe, University of Sheffield, UK

## Abstract

*This chapter presents an empirical assessment of the quality of the process of building software systems in light of the recent emergence of agile development methodologies, which were designed to to help with the development of higher quality information systems under given conditions. The purpose of this research was to assess one of these agile approaches, eXtreme Programming (XP), through a comparison with a traditional (design-driven) software construction process. With this aim we observed and measured the work of several student groups using different approaches to produce software for commercial companies during a semester. The data collected were analyzed following the Bayesian approach. Our results indicate that that XP could cope with small/medium size projects of software construction delivering a measurable improvement in the quality of the system as judged by the business clients.*

# Introduction

Quality measurement in software engineering constitutes a broad universe with many actors. Typically, empirical assessments of software construction methodologies are conducted by researchers rather than by software engineering practitioners. Among other reasons, this is so because the assessment process has not been fully defined, the guidelines do not always provide a well structured, smooth, and continuous description of how to do assessments; there are many gaps and a lack of general principles in this discipline, and there is no agreed format for presenting the results of such studies (Kitchenham, 2001). On the other hand, such studies can result in valuable insights, which can create real business value. There is a glaring need for the definition of a proper method to assess the software construction process in order to inform decision-making about appropriate methods for a particular application situation. In this study we assess eXtreme programming (XP) which has attracted much publicity but very little rigorous scientific evaluation for verifying its merits.

XP is one of the methods recently introduced to deal with the challenge of providing quality in software construction processes; it is a radical approach, in which the role of detailed design is reduced significantly (Beck, 1999b). This may create the impression that it is a variant of *hacker* programming. There is still no consensus on its efficacy; some describe XP as merely “working in pairs”, while others claim that it is the solution to the software crisis. Quite often such opinions are based solely on anecdotal evidence. Beck, the author of XP, presents it as a collection of practices that brings fresh air to the software construction process through the core principle of simplicity. The process (of software construction) is redesigned to avoid, as much as possible, the practices that make it heavy and bureaucratic. Our goal in this chapter is to provide scientifically tested evidence to make a meaningful contribution to this debate by providing empirical information about the strengths, weaknesses, and limits of the applicability of XP.

In this research effort we make two contributions: First we attempt to enlighten the on-going debate through an empirical assessment of XP in order to produce evidence to assist with a proper understanding of its scope and value. Second, we contribute to the definition of an effective methodology for conducting such empirical assessments. In order to run such an assessment, we exploited the opportunity at the University of Sheffield, which has links to industry and real projects for its students and an extensive program of experimentation as part of the Sheffield Software Engineering Observatory (Observatory, 2004), to conduct our experiment. This experiment compares the quality of two software production processes involved in producing traditional, design-led software with XP. This chapter presents the results and findings of the assessment.

## Background

XP has been proposed as an alternative to traditional models of systems development in which systems requirements are completely prespecified, and activities across the development life cycle are grouped into sequential phases, related in time and characteristics. Typical development life cycle stages include planning, analysis, design, implementation, and testing. It does not matter whether the process follows the object-oriented or structured methodology; system design is a central feature. Consequently, this approach is often called a design-led approach, and we refer to both approaches as traditional methods.

XP on the other hand is classified as an agile method, which proposes simplicity and small modules and handles dynamic requirements in volatile business environments (Holcombe, 2003a). It is one of the best known development methods in the family of agile development methods. Although these methods were developed independently, in 2001, 17 influential and recognized practitioners of agile methods signed a declaration called the Agile Manifesto (2001). Through this declaration, agile processes were slightly redefined and received greater momentum after "simmering beneath the surface for over ten years" (Highsmith, 2002, p. 9). Prior to this declaration, there were several isolated efforts to attempt to establish an agile movement, including that of Aoyama (1997, 1998a, 1998b) who defined agile processes as those that accommodate changing requirements and promote quick delivery of software products; they are incremental, iterative, and involve collaborative processes. The signatories of the Agile Manifesto (2001) defined 12 agile principles, rather than a concrete methodology.

In particular, XP emphasizes productivity, flexibility, informality, teamwork, and the limited use of technology outside of programming. Rather than planning, analyzing, and designing an entire system, XP programmers conduct these activities a little at a time throughout development (Beck, 1998). XP proposes that work be done in short cycles; in XP the basic unit of time is the cycle. Every cycle starts with the developer and client choosing a subset of requirements (called stories) from the complete set. Once such subset has been selected, the functional test set for it is written, then members of the team write the code, working in pairs (pair programming). Each piece of the code is then tested according to the relevant functional tests. The duration of each cycle is between 1 and 4 weeks, and there is no single, formal architecture (Beck, 1999a).

## The XP Research Literature

Beck (1998) produced the first paper that made explicit reference to XP, in which he highlighted the values that drive the XP philosophy. He followed this up with a second article that described the practices proposed in XP (Beck, 1999a) and a book describing the complete process in a general way (Beck, 1999b). Over the last few years, several research efforts have been devoted to the assessment of different aspects of XP (Butler et al., 2001; Gittins et al., 2002; Muller & Hanger, 2002; Muller & Tichy, 2001; Natsu et al., 2003; Ramachandran & Shukla, 2002; Rostaher & Hericko, 2002; Rumpe & Sholz, 2002; Rumpe & Schröder, 2002; Syed-Abdullah et al., 2003; Williams et al., 2000). These have been somewhat limited in ways we will describe soon.

Many of these studies focused on individual elements of XP or a subset of the practices (only one was broad enough to include most of the XP practices), and in some cases the experimental environment was so restrictive (as required for internal validity) that these studies were not realistic enough to capture the interactions of a live project. XP practices are mutually supportive, and a proper assessment requires a test of as many features as possible. Two research efforts that fit the latter category, for example, Gittins et al. (2001) and Syed-Abdullah et al. (2003), were both qualitative assessments.

Many of the studies were devoted to pair programming, one of the central features of XP, and these produced mixed results at best. For example, Williams et al. (2000) reported the results of a "structured experiment" in which they observed a group of 41 students in two subgroups of pair programmers and single programmers, respectively, over a period of 6 weeks. They found that programming pairs develop better code and faster than single programmers. However, the projects were not realistic commercial projects and so lacked many key aspects of a real software development project. In another experiment, Rostaher and Hericko (2002) tracked a group of professional programmers located in Slovenia and found that time and quality were not significantly different in solo vs. pair programming. The latter may be more credible because of the more realistic environment. Natsu et al. (2003) also ran a survey to test a tool built for distributed pair programming against co-located teams and found that couples working side by side obtain better quality code than the distributed couples.

In another research effort that focused on an isolated XP practice, Müller and Hanger (2002) compared the testing-first practice to traditional testing-after-coding practice and measured the reliability of the code. They found no difference in the implementation, but the test-first approach seemed to have promoted better program understanding. Müller and Tichy (2001) increased the number of XP practices tested in a survey that involved pair programming, small releases, and automated testing. The authors concluded that the real benefits of pair programming were unclear, and that the student subjects found it difficult to produce small releases, and were unable to write tests before implementation.

Three research efforts widened the scope of practices examined. First, Gittins et al. (2001) conducted a qualitative study to assess the complete XP process in a medium size software construction company that had adopted XP progressively and reported on 9 of the 12 XP practices. They reported that tension among the project participants was high, quality was compromised, communication difficult, and control largely ineffective. Then Butler et al. (2001, p. 140) conducted a broad study rather than one that focused on specific variables and found that "XP … relies strongly on the results of impressions and feelings of developers". Finally, Rumpe (2002) surveyed several companies, which were running XP, about the background of the company and team members, the practices they followed, and their future plans, in order to provide a profile of behavioral factors that may contribute to successful use of XP. They came up with inconclusive results.

## Research Hypotheses

In this study we took a holistic approach to the examination of XP practices, examining the method as a whole instead of focusing on individual, stand-alone practices. The objective was to provide empirical evidence to examine the claim that XP is a valid software production process that produces results that are comparable to or better than those generated by traditional approaches. Several managers and developers view XP as a suitable alternative for software process production; others are not as convinced. The experiment may be considered to be a follow up to Moore and Palmer (2002), Putnam (2002), Rumpe and Scholz (2002), and Wright (2001). The hypotheses generated are necessarily exploratory. Other than the sales pitches of originators, anecdotal claims of some practitioners, and a few research efforts that are largely inconclusive, no theoretical basis exists to guide expectations and predict outcomes.

We focused our assessment on small projects, which we believe are consistent with the agility conditions of XP and examined internal and external quality, development time, and the size of the project. The choice of the quality construct is obvious; the ability of any information system process method to generate quality outcomes is the ultimate test of its "goodness". The other two factors (development time and project size) provide baselines for understanding and interpreting the quality results. The following hypotheses were stated:

- **Null Hypothesis:** The use of extreme programming in a software construction process produces equal results (external and internal quality, time spent and size of the product) as those obtained from the use of traditional processes (in the average case of a small/medium size project).
- **Alternative Hypothesis:** The use of extreme programming in a software construction process produces different results (exhibits better or worse quality, requires more or less time and the size of the product is different) than those obtained from the application of traditional processes (again in the case of average size between small and medium).

# Experimental Context and Design

Three important research efforts conducted by Basili and Weiss (1984), Basili et al. (1986), and Basili and Rombach (1988) may be considered cornerstone studies for research in software engineering. The first, a controlled experiment, provided a method and guidelines for collecting data in software engineering projects and related the result of a process to its structure. They also defined a methodology for the specification or selection of such data. Although they did not give a name to the method, it was later called the Goal-Question-Metric (GQM) framework. Basili et al. (1986) expanded this framework when they produced the GQM goal template, which is a more complete framework for carrying out experiments in software engineering, providing a basis for classifying a set of previous experiments. The final paper in this sequence presented further guidelines for organizing successful experiments in software engineering. In addition to the steps proposed in the GQM template, it offered a more detailed description of suitable metrics required in the experimental processes and definitions of some of the principles. This final paper was based on the project named Tailoring A Measurement Environment (TAME), which accumulated over 10 years of experience devoted to improving principles and guidelines for process modeling based on quantitative analysis geared toward the control and improvement of software engineering projects.

## Project Participants and Procedures

The project team members were second-year students registered in the Computer Science undergraduate program at the University of Sheffield in the UK. They were required to complete a module called the Software Hut Project (SHP), which lasted for one academic semester during which they developed small software for real business clients. These students already knew how to write programs, develop data structures and specifications, and produce Web pages. All the students registered in this module were included in the experiment. SHP also included training in XP and the traditional design-led process for software construction (Traditional). According to the regulations of the university program, it was expected that every student would devote an average of 15 hours per week to this project for the 12 scheduled weeks. There was a three-week break at the midpoint of the semester, and although students were not required to work during the break, most do to recover from delays. If they work, the students are required to report their activities, and they also receive coaching during this period.

The class was divided into two sections, teams one to 10 and teams 11 to 20 for the two treatments (XP and Traditional, respectively). The teams, which were formed by the students themselves, were randomly assigned the treatments. Eight of the teams had 4 members each, 11 had 5 members, and 1 team had 6 members. Training was provided during the normal sessions of the module. Each team worked with only one client, but each client could be served by several teams. Simultaneously with these sessions, the students had to interview their clients and plan their project. The lecturer reviewed the clients' requirements in order to verify their feasibility.

The formal requirement documents were then agreed upon between the teams and their clients. The completed systems had to be installed and commissioned at the clients' sites. The various groups then developed the system. Some students wrote programs in Java, while others wrote code for PHP and SQL according to the requirements of the system. During the experiment, the first priority was to ensure that teams followed the treatments faithfully and to apply corrections if required. Thus the timesheets and the weekly minutes were verified by interviewing the students.

There were four clients who will be referred to as Clients A, B, C, and D for simplicity:

*Client A*. Small Firms Enterprise Development Initiative (SFEDI) facilitated self-employed, owners and managers of small companies (nationally) to start up, survive, and thrive. SFEDI required a Web site that would be generally accessible from any location, to allow their employees to distribute general documents, policies, and procedures to other employees. They wanted to restrict access to certain documents, according to the category of the employee accessing them. Their main aim was to improve internal communications among their employees, who had a fair level of computer literacy.

*Client B*. The School of Clinical Dentistry of the University of Sheffield conducts research, using questionnaires, to collect information about patients. They may run several questionnaires simultaneously. The data generated from these questionnaires are used for a variety of purposes. The school required a system to customize the online questionnaires and subsequently collect the responses into a secure database. The generation of the questionnaire should be very simple, not requiring any specialized knowledge and usable by anyone with even low computer literacy.

*Client C*. University for Industry (UFI) was created as the government's flagship for lifelong learning. It was created with its partner Learn Direct, the largest publicly founded online learning service in the UK, to encourage adult education. In order to analyze performance and predict future trends, UFI/Learn Direct needed to collate and analyze information (such as the number of Web site hits or help-line calls at different times of the day and the number of new registrations) and the relationship among these data items. UFI needed a statistical analysis program to help managers plan how best to allocate and manage their resources based on trends and patterns in the data recorded. The required data were recorded on simultaneous users, annual trends, predicted growth, and other performance indicators to be displayed graphically.

*Client D*. The National Health Service (NHS) Cancer Screening Program keeps an archive of journals and key articles that they provide to the Department of Health, the public, the media, and people invited for screening. They required a system which was simple to use and easy to maintain and which allowed them to (1) catalogue the existing collection of articles; (2) add new articles; (3) expand the collection to include articles on screening for colorectal, prostate, and other cancers; (4) link associated articles; and (5) find and retrieve articles quickly and effectively. A member of the staff will maintain the system, but other staff members will use it to search for articles. The client had no special preference for the system appearance or operation beyond the requirement that it should be easy to use.

## Experimental Design

The organization of the experiment did not correspond to a typical factorial, crossed, or nested experimental design. We crossed the treatments (XP and Traditional) and the blocks (clients) as if they were treatments. Then each block was considered as a level (Figure 1). The experimental units (teams) were randomly allocated and every block (client) received the two treatments. The students gathered in teams and were advised of the treatment (XP or traditional) and client (A, B, C or D) they were assigned to. Every client received both

*Figure 1. Distribution of teams per treatment and blocks (clients)*

|  | Treatments |  |
|---|---|---|
| Blocks | XP | Traditional |
| A | 5, 7, 8 | 18, 20 |
| B | 2, 6 | 12, 14, 17 |
| C | 1, 9 | 11, 13, 19 |
| D | 3, 4, 10 | 15, 16 |

treatments (XP and traditional), and five teams were allocated to each client (Figure 1). Each team tried to provide a complete solution for their client.

The unit of observation was the project team. Students were not observed or tracked individually; the communication processes, assessment, log, and verification were always at the team level. While the participants were hypothesis-blind, there was no blindness among the teams as they received suitable training to apply the treatment. The lecturers and the clients were also fully aware of the different treatments each team was working with and the clients were able to identify the XP teams because this treatment encourages close relationships with clients.

## Variables and Measures

There were three main measures: The quality of the product, the time spent in the production process, and the size of the product. The students reported the time they spent in the project. Both the external and internal quality of the product was tracked. Teams submitted weekly timesheets that logged the time distribution of every member of the team in every activity. The size of the products was obtained from the reports of the teams. These are metrics suggested by the GQM goal template (Basili et al., 1986; Basili & Rombach, 1988).

External quality, which addresses those factors that the users (client) appreciate in the normal operation of the software, regardless of the internal, technical operation, was assessed by the client, and included the adequacy of the documentation (presentation, user manual, and installation guide) and attributes of the software system, which includes ease-of-use, error handling, understandability, completeness of the base functionality, innovation (extra features), robustness (correctness, i.e., does not crash), and user satisfaction with the product.

Internal quality, which addresses those internal properties of the system that are not directly distinguishable by the clients, was measured by the lecturer. Internal quality was not measured in exactly the same way for both treatments because of the nature of the treatments (e.g., in a traditional approach it was not required to provide evidence of pair programming). For XP the measures were based on the set of user stories signed off by the client, specification of test cases for the proposed system, the test management processes, and the completed test results. In the traditional approach, the metrics were requirements documentation, the detailed design for the system, and the completed test results. In both cases the "goodness" of the code, documentation and maintenance guide, and a general description, including the log of the project, were also evaluated.

The variables included for the purpose of measuring project time were the number of hours spent by team members doing research, requirements, specification and design, coding, testing, and reviewing. Other special one-off activities were also captured. Product size was measured by the number of logical units (functions, modules, classes), lines of code, characters in code, and percentage of code devoted to commentary. Counting lines-of-code is always a questionable process; however, the procedures recommended by Macias (2004) were used to reduce bias.

Data were collected throughout the semester and at the end of the project mostly through meeting minutes and timesheets. The minutes contained the activities of the current week and the plan of the work for the next week. The timesheets registered the time spent on each activity weekly. Both timesheets and minutes were automatically collected. A script ran every Friday at 4:00 P.M. and updated the records of the teams with the new files. At project end, the clients and the lecturers provided their assessment, and the teams produced their report.

## Analytical Procedures

The use of analysis of variance (ANOVA) is recommended when the populations have been sampled independently and randomly, and also the populations are normally distributed and have similar variances (Aczel, 1996). This was not so in our case, yet we started with ANOVA as a precursor to the Bayesian tests that we employed to work with the uncertainties of field research with less experimental control, using probabilities rather than binary results. Our pilot study had indicated that the sample sizes were too small to consider ANOVA as a suitable for analyzing of the data, and some tests provided results close to the boundary values defined to reject (or not reject) the null hypothesis. We therefore opted for a method that did not provide binary result (reject or not reject) — the Bayesian analysis recommended for this purpose (Holcombe et al., 2003).

The Bayesian approach is often used as an alternative to the more common classical (frequentist) method (Berry, 1996). It allows one to forecast future observations and incorporate evidence from previous experiments into the overall conclusions. One possible problem is that this approach is subjective; different observers may reach different conclusions from the same experimental results. However, the Bayesian approach is recommended in situations that involve uncertainty, and it is also a good choice when there is the possibility of running replications. This approach helps to predict the behavior of the data in the next replication and to correct noisy factors, thereby providing a means to relate results from successive experiments. This experiment was conducted as a replication of a prior pilot study (Observatory, 2004).

# Results

Below are representative summaries of data collected from the experiments (descriptive statistics) for quality, project time, and project size. The average of both internal and quality measures was higher for the XP groups than for the Traditional teams. Table 1 shows the averages and standard deviation of the

*Table 1. Average and standard deviation of marks (Int Q + Ext Q), internal quality and external quality measured on a 50 point scale in each case*

| Treatment | Mark average | Mark avg std deviation | Internal Q average | Internal Q std dev | External Q average | External Q std dev |
|---|---|---|---|---|---|---|
| XP | 72 | 7 | 36 | 3 | 36 | 6 |
| Trad | 68 | 7.5 | 34 | 2.5 | 34 | 6 |

*Table 2. Distribution of samples, factor quality. All teams*

| | XP | Traditional |
|---|---|---|
| Sample size | 10 | 10 |
| Average | 71.9 | 68.15 |
| Standard Deviation | 6.75 | 7.45 |
| Data into 1 std. dev. * | 70% | 70% |
| Data into 2 std. dev. * | 90% | 100% |
| Data into 3 std. dev. * | 100% | 100% |

* Refers to the distance from the mean

quality values. The distribution of values in both treatments (XP and traditional) was very similar, and the small difference was persistent. The correlation coefficient between internal and external quality (considering the 20 teams) was 0.33, indicating that there was no apparent relationship between both factors.

The factor quality, also called Final Mark, was obtained from two different aspects: internal and external quality. It is reported here as a single item that is the sum of both. The averages and standard deviation are shown in Table 2.

*Table 3. Averages and standard deviations of the time spent in hours*

| Treatment | Total Time | Total time Std. Dev. | Coding Time | Coding T Std. Dev. | Testing Time | Testing Time Std. Dev. |
|---|---|---|---|---|---|---|
| XP | 584 | 142 | 207 | 95 | 110 | 38 |
| Trad | 396 | 186 | 163 | 127 | 33 | 25 |

*Table 4. Distribution of samples, factor time (all teams)*

| | XP | Traditional |
|---|---|---|
| Sample size | 10 | 10 |
| Average | 584 | 396 |
| Standard Deviation | 141.5 | 186.5 |
| Data into 1 std. dev.* | 70% | 60% |
| Data into 2 std. dev.* | 100% | 100% |
| Data into 3 std. dev.* | 100% | 100% |

* It refers to the distance from the mean

*Table 5. Average and standard deviation of thousands of characters and percentage of commentaries*

| Treatment | Average of K-Characters | Standard Dev. Of K-Character | Average % of Commentaries | Std Dev of % Commentaries |
|---|---|---|---|---|
| XP | 301 | 206 | 22 | 11 |
| Traditional | 210 | 109 | 20 | 17 |

*Table 6. Distribution of samples, size in thousands of characters (excluded teams 2 and 18)*

| | XP | Traditional |
|---|---|---|
| Sample size | 9 | 9 |
| Average | 301 | 232.9 |
| Standard Deviation | 205.5 | 86.5 |
| Data into 1 std. dev. * | 77% | 66% |
| Data into 2 std. dev. * | 88% | 99% |
| Data into 3 std. dev. * | 99% | 99% |

* It refers to the distance from the mean

It appears that XP teams spent more time overall obtaining their results and more in coding and testing than traditional teams (Table 3).

There is also the indication that the volume of code was higher on average for XP than for traditional teams (Table 5).

Measuring the size of code involves counting the number of files, classes/modules, functions/procedures/tasks, lines, and characters. Quite often size is reported as number of lines but there is no general consensus on the meaning of "line" in a piece of code. Additionally, different programming languages allow different possibilities to compact code, and their expressiveness also affects the number of lines. For these reasons this project considered number of characters rather than lines as the main parameter to measure size. The distribution of number of characters in code appears in Table 6.

The difference in quality between the two treatments was small, but it is remarkable that such a small difference was maintained not only in the final result (the sum of the parts) but also through the subitems, internal and external. Such a condition triggered the decision to analyze the data using a non-frequentist model of inferential analysis.

From the result of the ANOVA test shown in Table 7, it is clear that only hypothesis 2 (project time) is supported by the analysis of the data. The test was applied to every factor and the four blocks in order to determine whether the

*Table 7. ANOVA test for main experiment*

| Factor | Source of Variation | F value | P value | Rejecting the Null Hypothesis |
|---|---|---|---|---|
| Time | Treatment | 6.48 | 0.02 | Yes. XP was higher |
| | Block | 0.08 | 0.97 | No |
| Overall quality | Treatment | 1.39 | 0.254 | No. Values close to the boundaries |
| | Block | 0.01 | 0.999 | No |
| Internal quality | Treatment | 1.87 | 0.188 | No. Values close to the boundaries |
| | Block | 0.21 | 0.866 | No |
| External quality | Treatment | 0.65 | 0.431 | No |
| | Block | 0.08 | 0.972 | No |
| K-characters | Treatment | 0.84 | 0.373 | No |
| | Block | 0.06 | 0.979 | No |
| Number of test cases | Treatment | 1.53 | 0.232 | No. Values close to the boundaries |
| | Block | 1.21 | 0.339 | No. Values close to the boundaries |
| Number of requirements | Treatment | 0.02 | 0.891 | No |
| | Block | 2.97 | 0.063 | Partially * |

* Teams A (SFEDI) and D (NHS) do not overlap confidence intervals

*Table 8. Number of succeeding and failing teams (overall quality)*

| SHP-02 | Success | Fail |
|---|---|---|
| XP | 6 | 4 |
| Trad | 5 | 5 |

behavior of the data was ruled by the treatment or by the block. The values for overall quality were obtained by adding internal and external quality. The number of characters, test cases, and requirements were independent of each other. For the reasons already elaborated, we therefore turn to Bayesian analyses to further clarify the picture.

In this experiment we identified "successful" teams as those with overall quality value above the global average and failing teams as those whose overall quality marks are below the global average. This criterion gave the distribution of teams shown in Table 8.

The average of Quality (Internal + External) was 70.025; then a team succeeded if Q > 70.025 and a team failed if Q <= 70.025. Bayes' theorem was applied to these data, and the following probabilities were gauged. Hence, the result is that if we choose randomly a successful (or failing) team:

- Probability that a successful team was XP: 0.5454
- Probability that a successful team was Trad: 0.4545
- Probability that a failing team was XP: 0.444
- Probability that a failing team was Trad: 0.555

According to these results, the XP teams have a higher probability of success than Traditional teams.

The same criterion was applied to the data obtained for external quality (Table 9) and internal quality (Table 10) with markedly similar probabilities as indicated. When the Bayesian analysis was applied to external quality, the average of external quality was 34.95; then a team succeeded if Q > 34.95 and a team failed if Q <= 34.95, and correspondingly for internal quality a team succeeded if Q > 35.075 and failed if Q <= 35.075.

*Table 9. Number of succeeding and failing teams (external quality)*

| ExtQ-02 | Success | Fail |
|---|---|---|
| XP | 6 | 4 |
| Trad | 5 | 5 |

*Table 10. Number of succeeding and failing teams (internal quality)*

| IntQ-02 | Success | Fail |
|---|---|---|
| XP | 6 | 4 |
| Trad | 4 | 6 |

| | |
|---|---|
| • Probability that a successful team was XP: | 0.5454 |
| • Probability that a successful team was Trad: | 0.4545 |
| • Probability that a failing team was XP: | 0.444 |
| • Probability that a failing team was Trad: | 0.555 |
| • Probability that a successful team was XP: | 0.6 |
| • Probability that a successful team was Trad: | 0.4 |
| • Probability that a failing team was XP: | 0.4 |
| • Probability that a failing team was Trad: | 0.6 |

The Bayesian analysis was not applied to the time dimension because of the conclusive nature of the ANOVA test. The size of the projects also did not indicate any significant difference between the two treatments. However, there was no special condition that suggested the application of the Bayesian approach to these data.

## Discussion

Except for the results related to project time (hypothesis 2), the ANOVA analysis indicated no significant differences in treatment for any of the other variables, perhaps as a result of the many noisy factors wrapped in a large experiment like this one. The Bayesian analyses however, provided some information that may otherwise have been lost had we depended entirely on ANOVA. Although the probabilities measured are not enough to provide actionable conclusions to inform practice, they are required to help refine the experiments in future replications.

This research suffered from several limitations, however. The balance (size of teams and distribution of teams in treatments) was not symmetrical in all the cases, and the distribution of the subjects was not entirely random so as to provide the rigor required for "frequentist" analyses. We traded off internal validity for the ability to generalize our results. Similarly, we did not follow Basili's GQM goal template religiously for generating the items to be measured. We view this research as one step in a sequence that will eventually produce statistically conclusive results.

In addition, we did not control for several potentially confounding factors such as the experience of the teams and the influence of the (more/less) enthusiastic participants and technical experts. The students knew the objectives of the exercise, and more competitive ones may have obtained information beyond that provided in the course. Information exchange among teams might have been a problem; however, this may have been suppressed by the competition for the prize offered for the best solution. Finally, the experiment involved student subjects, without the real experience nor incentives of real-world professionals.

Nevertheless, we made some interesting observations about some XP practices. For example, we noted that programmers working in pairs were enthusiastic, and they made comments like: "it is more easy to work in pairs, it helps to discover mistakes," "it is more easy to produce ideas in collaborative work," "the results of working in pairs or singly are the same, but if you work in pairs, you do not get bored." We also noted some confounding behavioral issues such as the existence of local experts, who program much faster and more effectively than the rest of the members of the team. The local expert, when present, preferred to work alone, and considered the rest of the members of the team as an obstacle to his or her progress. One such expert said, "I feel sceptical, working in pairs could waste your time." This was consistent with Macias et al.'s (2003) finding.

We also found several problems related to testing first. Many students did not know how to write a black box test, which necessitated specific training to accommodate this practice. Others had a natural tendency to write the test set after writing the code, while a few were reluctant to implement this practice because the benefits were neither evident nor explicit. It was discovered that this practice requires great discipline.

## Conclusions

In this experiment we set out to assess the entire XP process in a reasonably realistic situation with real business clients using the GQM to contribute meaningfully to the debate on XP's place among the many process methods that have been used to structure software development. While indications from the ANOVA tests indicated no difference between the treatments for either internal or external quality, the Bayesian analysis indicated that there was a greater probability of producing successful systems from the XP process than through traditional methods.

We believe that the combination of analytical methods employed in this research provides greater insights than any single method would and should be used by researchers to evaluate software methods along multiple dimensions. The Bayesian analysis was applied in a second stage. Rather than a binary result, it offers a numerical value representing probability of success expressed as a percentage.

Future research is required in this area to systematically eliminate the limitations outlined earlier and to confirm or disconfirm these results in the continuing search for scientific evidence to inform the software methods debate. It may also be necessary to distinguish among the several classes of methods (sequential design-led, iterative/incremental methods, and reuse-based methods) in future evaluations in order to denote the contexts in which they are most applicable.

# References

Aczel, A. (1996). *Complete business statistics* (3rd ed.). Irwin.

Agile Manifesto. (2001). Retrieved November 30, 2004, from http://agilemanifesto.org/

Aoyama, M. (1997, August 11-15). Agile software process model. *Proceedings of the 21st International Computer Software and Applications Conference*, Washington, DC (pp. 454-459).

Aoyama, M. (1998a, April 19-25). Agile software process and its experience. *Proceedings of the 20th International Conference on Software Engineering*, Kyoto, Japan (pp. 3-12).

Aoyama, M. (1998b, November/December). Web-based agile software development. *IEEE Software, XV*(6), 56-65.

Basili, V. R., & Rombach, H. D. (1988, June). The TAME project: Towards improvement-oriented software environments. *IEEE Transactions in Software Engineering, XIV*(6), 758-773.

Basili, V. R., Selby, R. W., & Hutchens, D. H. (1986, July). Experimentation in software engineering. *IEEE Transactions on Software Engineering, SE-12*, 733-743.

Basili, V. R., & Weiss, D. M. (1984, November). A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering, SE-10*, 728-738.

Beck, K. (1998). Extreme programming: A humanistic discipline of software development. *Lecture Notes in Computer Science, 1382*, 1-16.

Beck, K. (1999a, October). Embracing change with extreme programming. *Computer, XXXII*(10), 70-77.

Beck, K. (1999b). *Extreme programming explained: Embrace change*. Addison-Wesley.

Berry, D. (1996). *Statistics. A Bayesian perspective*. International Thomson Publishing.

Butler, S., Hope, S., & Gittins, R. (2001, May 20-23). How distance between subject and interviewer affects the application of qualitative research to extreme programming. *Proceedings of 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, Sardinia, Italy.

Campbell, D. T., & Stanley, J. C. (1999). *Experimental and quasi-experimental designs for research*. Boston: Houghton Mifflin.

Conte, S., Dunsmore, H., & Shen, V. (1986). *Software engineering metrics and models.* Benjamin/Cummings.

Cook, T. D., & Campbell, D. T. (1979). *Quasi-experimentation: Design and analysis issues for field settings.* Boston: Houghton Mifflin.

Gittins, R., Hope, S., & Williams, I. (2002, May 20-23). Qualitative studies of XP in a medium sized business. *Proceedings of 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, Sardinia, Italy.

Highsmith, J. (2002). *Extreme programming perspectives* (pp. 9-16). Addison-Wesley.

Holcombe, M. (2003). XP after Enron — can it survive? *Proceedings of the 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering* (Lecture Notes in Computer Science, *2675*, 1-8).

Holcombe, M., Cowling, A., & Macias, F. (2003, September). Towards an agile approach to empirical software engineering. *Proceedings of the ESEIW 2003 Workshop on Empirical Studies in Software Engineering, Roman Castles,* Italy (pp. 37-48).

Judd, C. M., Smith, E. R., & Kidder, L. H. (1991). *Research methods in social relations* (6th ed.). Harcourt Brace Jovanovich.

Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P.W., Hoaglin, D. C., & El-Emam, K., et al. (2001, January). Preliminary guidelines for empirical research in software engineering (Tech. Rep. No. ERB-1082, NRC 44158). *National Research Council, Institute for Information Technology*.

Macias, F. (2004). *Empirical assessment of extreme programming.* PhD thesis, University of Sheffield, Computer Science Department, UK.

Macias, F., Holcombe, M., & Gheorghe, M. (2003). Design-led and design-less: One experiment and two approaches. *Proceedings of the 3rd Interna-*

*tional Conference on Extreme Programming and Flexible Processes in Software Engineering* (Lecture Notes in Computer Science, *2675*, 394-401).

Montgomery, D. (2001). *Design and analysis of experiments* (5th ed.). John Wiley & Sons.

Moore, I., & Palmer, S. (2002, May 26-30). Making a mockery. *Proceedings of the 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, Sardinia, Italy (pp. 6-10).

Müller, M., & Hanger, O. (2002, October). Experiment about test-first programming. *IEE Proceedings Software, 149*(5), 131-136.

Müller, M., & Tichy, W. (2001). Case study: Extreme programming in a university environment. *Proceedings of the 23rd International Conference on Software Engineering,* Toronto, Ontario, Canada (pp. 537-544).

Natsu, H., Favela, J., Moran, A., Decouchant, D., & Martinez, A. (2003, September). Distributed pair programming on the Web. *Proceedings of the 4th Mexican International Conference on Computer Science*, Tlaxcala, Mexico (pp. 8-12).

Observatory. (2004). *Sheffield Software Engineering Observatory*. Retrieved November 30, 2004, from http://www.dcs.shef.ac.uk/~ajc/empirical/observatory.html

Putnam, D. (2002, May 26-30). Where has all the management gone? *Proceedings of 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, Sardinia, Italy (pp. 39-42).

Ramachandran, V., & Shukla, A. (2002). Circle of life, spiral of death: Are XP teams following the essential practices? *Extreme Programming and Agile Methods —XP/Agile Universe 2002, LNCS, 2418*, 166-173.

Rostaher, M., & Hericko, M. (2002). Tracking test first pair programming — An experiment in extreme programming and agile methods. *Extreme Programming and Agile Methods —XP/Agile Universe 2002, LNCS, 2418*, 166-173.

Rumpe, B., & Scholz, P. (2002, May 26-30). A manager's view on large scale XP projects. *Proceedings of the 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, Sardinia, Italy (pp. 160-163).

Rumpe, B., & Schröder, A. (2002, May 26-30). Quantitative survey on extreme programming projects. *Proceedings of the 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering,* Sardinia, Italy.

Syed-Abdullah, S., Holcombe, M., & Gheorghe, M. (2003). Practice makes perfect. *Proceedings of the 4th International Conference on Extreme Programming and Flexible Processes in Software Engineering* (Lecture Notes in Computer Science*, 2675*, 354-356).

Williams, L., Kessler, R., Cunningham, W., & Jeffries, R. (2000, July/August). Strengthening the case for pair programming. *IEEE Software, XVII*(4), 19-25.

Wright, G. (2001, May 26-30). eXtreme Programming in a hostile environment. *Proceedings of 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, Sardinia, Italy (pp. 48-51).

# Section IV: Managing Risks of SPI Projects and Methodologies

## Chapter XI

# Building IT Risk Management Approaches: An Action Research Method

Jakob Holden Iversen, University of Wisconsin Oshkosh, USA

Lars Mathiassen, Georgia State University, USA

Peter Axel Nielsen, Aalborg University, Denmark

## Abstract

*This chapter shows how action research can help practitioners develop IT risk management approaches that are tailored to their organization and the specific issues they face. Based on literature and practical experience, the authors present a method for developing risk management approaches to use in real-world innovation projects. The chapter illustrates the method by presenting the results of developing a risk management approach for software process improvement projects in a software organization.*

# Introduction

Organizations that manage IT innovations have long been accused of having poor project execution and low product quality. These problems are often referred to as "The Software Crisis," in which software projects frequently are delivered late, over budget, with missing features, and with poor quality. Furthermore, it has been very difficult to predict which organization would do a good job on any given project. These issues led to the establishment of the software process improvement (SPI) movement, in which poor processes in organizations are considered a major reason for the software crisis.

Organizations routinely rely on experienced developers to deliver high quality IT systems. However, in the 1990s, organizations realized that by defining and improving the processes these professionals used, it was possible to deliver more consistent results with better quality. SPI projects were established to improve specific aspects of a process, and in many cases to take advantage of standards like the Capability Maturity Model (CMM) (Paulk et al., 1993) and the Capability Maturity Model Integration (CMMI) (Chrissis et al., 2003). For each process that needed improvement, a focused SPI project would design and implement specific improvements into current practices.

However, not only is this hard work, it also is risky business. Much can go wrong in improvement projects, and mistakes can eventually lead to failure. The involved improvement actors might not possess appropriate skills and experiences. The design of a new process might not suit the organization or effectively meet requirements. The improvement project might be organized inappropriately, with unrealistic schedules or insufficient management attention. Also, the actors might pay too little attention to customers, failing to consider the interests, problems, and motivations of the people and groups that are expected to use the new process.

To deal proactively with such issues in SPI projects, the involved actors must manage the involved risks. The need for such risk management was the rationale behind Danske Bank's development of a practical risk management approach to reduce failures in their SPI initiative. Using this approach, improvement actors periodically held disciplined and tightly structured workshops in collaboration with SPI facilitators. The workshops gave each team a better overview and understanding of their project and its organizational context, and helped them address risks proactively.

Organizations face many different and quite diverse activities in which there are strong reasons to manage IT risks. While the literature provides a portfolio of IT risk management approaches that cover many types of activities, organizations often face situations in which they need to develop a risk management approach that is tailored to their particular needs or that addresses issues not covered by the available portfolio of documented risk management approaches. This chapter offers organizations a generic method to develop new and dedicated IT risk management approaches. The method is based on action research into an organization's specific risk management context and needs, and builds on the available literature about IT risk management. It is based on our experiences from developing the tailored approach to risk management in SPI projects at Danske Bank.

## Risk Management Literature

A number of different approaches to IT risk management have been proposed. In this section, we provide an overview and categorization of the different approaches (risk list, risk-action list, risk-strategy model, risk-strategy analysis). We offer, in this way, a framework to help select an appropriate risk approach suited to particular organizational contexts and needs. An overview of the framework is shown in Table 1.

*Table 1. Four types of approaches to IT risk management*

| **Type of Approach** | **Characteristics** | **Assessment** | **Exemplars** |
|---|---|---|---|
| *Risk list* | A list of prioritized risk items | + Easy to use<br>+ Easy to build<br>+ Easy to modify<br>+ Risk appreciation<br>- Risk resolution<br>- Strategic oversight | (Barki et al., 1993; Keil et al., 1998; Moynihan, 1996; Ropponen & Lyytinen, 2000) |
| *Risk-action list* | A list of prioritized risk items with related resolution actions | + Easy to use<br>+ Easy to build<br>+ Easy to modify<br>+ Risk appreciation<br>+ Risk resolution<br>- Strategic oversight | (Alter & Ginzberg, 1978; Boehm, 1991; Jones, 1994; Ould, 1999) |
| *Risk-strategy model* | A contingency model that relates aggregate risk items to aggregate resolution actions | + Easy to use<br>- Easy to build<br>- Easy to modify<br>+ Risk appreciation<br>+ Risk resolution<br>+ Strategic oversight | (Donaldson & Siegel, 2001; Keil et al., 1998; McFarlan, 1981) |
| *Risk-strategy analysis* | A stepwise process that links a detailed understanding of risks to an overall risk management strategy | - Easy to use<br>- Easy to build<br>+ Easy to modify<br>+ Risk appreciation<br>+ Risk resolution<br>+ Strategic oversight | (Davis, 1982; Mathiassen et al., 2000) |

## Risk List

The first and simplest form of available approaches are risk lists. They contain generic risk items (often prioritized) to help managers focus on possible sources of risk; they do not contain information about appropriate resolution actions. These lists are easy to use in assessing risks; they are easy to build, drawing upon published sources on risks or experiences within a particular context; and they are easy to modify to meet conditions in a particular organization or as new knowledge is captured. While these approaches offer strong support to help managers appreciate risks, they do not support identification of relevant resolution actions and they do not provide a strategic oversight of the risk profile and relevant strategies for action. Based on previous research, Barki et al. (1993) offer a detailed and precise definition, a measure of software development risk, and a systematic assessment of the reliability and validity of the instrument. Moynihan (1996) presents a comprehensive list of risk items based on how software project managers construe new projects and their contexts. Keil et al. (1998) offer a list of nearly a dozen risk factors that IT project managers in different parts of the world rated high in terms of their importance. Ropponen and Lyytinen (2000) report six aggregate risk components, for example, scheduling and timing risks, that experienced IT project managers found important in a recent survey.

## Risk-Action List

The second, slightly more elaborate, form of approaches are risk-action lists. They contain generic risk items (often prioritized), each with one or more related risk resolution actions. They also are easy to use; they are quite easy to build, but compared to risk lists, they require additional knowledge of the potential effects of different types of actions; finally, they are easy to modify when needed. Risk-action lists offer the same support as the risk lists to appreciate risks. In addition, they adopt a simple heuristic to identify possible relevant actions that might help resolve specific risks. However, by focusing on isolated pairs of risk items and resolution actions, they do not lead to a comprehensive strategy for addressing the risk profile as a whole. Alter and Ginzberg (1978) list eight risk items related to IT system implementation, for example, unpredictable impact; and they offer four to nine actions for each risk, for example, use prototypes. Boehm (1991) offers a top-ten list of software development risks, with three to seven actions per risk. Jones (1994) presents specialized risk profiles for different types of IT projects, together with advice on how to prevent and control each risk. Finally, Ould (1999) suggests maintaining a project risk register for identified risks, assessment of the risks, and risk resolution actions to address them.

## Risk-Strategy Model

The third form of approaches are risk-strategy models. These contingency models relate a project's risk profile to an overall strategy for addressing it. They combine comprehensive lists of risk items and resolution actions with abstract categories of risks (to arrive at a risk profile) and abstract categories of actions (to arrive at an overall risk strategy). The risk profile is assessed along the risk categories using a simple scale (e.g., high or low), which makes it possible to classify a project as being in one of a few possible situations. For each situation, the model then offers a dedicated risk strategy composed of several detailed resolution actions. Compared to the other types, risk-strategy models provide detailed as well as aggregate risk items, and resolution actions. The heuristic for linking risk items to resolution actions is a contingency table at the aggregate level. Risk-strategy models are easy to use because of the simplifying contingency model, but they are difficult to build because the model must summarize multiple and complex relationships between risks and actions. They also are difficult to modify except for minor revisions of specific risk items or resolution actions that do not challenge the aggregate concepts and the model. Models like these help appreciate risks and identify relevant actions, and managers can build an overall understanding of the risk profile they face (at the aggregate level) directly related to a strategy for addressing it (in terms of aggregate actions).

The best known of these approaches is McFarlan's (1981) portfolio model linking three aggregate risk items (project size, experience with technology, and project structure) to four aggregate resolution actions (external integration, internal integration, formal planning, and formal control). Keil et al. (1998) present a model that combines the perceived importance of risks with the perceived level of control over risks. The model suggests four different scenarios (customer mandate, scope and requirements, execution, and environment) with distinct risk profiles and action strategies. Donaldson and Siegel (2001) offer a model categorizing projects into a high, medium, or low risk profile. They suggest a different resource distribution between project management, system development, and quality assurance, depending on a project's risk profile.

## Risk-Strategy Analysis

The final form of approaches are risk-strategy analyses. These approaches are similar to risk-strategy models in that they offer detailed as well as aggregate risk items and resolution actions, but they apply different heuristics. There is no model linking aggregate risk items to aggregate resolution actions. Instead, these approaches offer a stepwise analysis process through which the involved actors link risks to actions to develop an overall risk strategy. Compared to the risk-strategy models, there is a looser coupling between the aggregate risk items and aggregate resolution actions. In comparison, we find these approaches more difficult to use because they require process facilitation skills. They are as difficult to build as the risk-strategy models, but they are easier to modify because of the loosely defined relationship between aggregate risk items and resolution actions. Davis (1982) provides such a stepwise approach to address information requirements risks where the overall level of risk is assessed and then associated with four different strategies to cope with requirements uncertainty. Mathiassen et al. (2000) offer a similar approach to develop a risk-based strategy for object-oriented analysis and design.

The comparative strengths and weaknesses of these four risk approaches are summarized in Table 1. Comparing the list approaches and the strategy approaches suggests that the former are easier to use, build, and modify, whereas the latter provide stronger support for risk management. Comparing risk-strategy models and the risk-strategy analysis approaches suggests that the former are easier to use, but they require that a contingency model be developed. The latter are easier to modify because they rely on a looser coupling between aggregate risk items and resolution actions. Organizations can use the insights in Table 1 to choose appropriate forms of IT risk management that are well suited to the particular challenges they want to address.

## Action Research

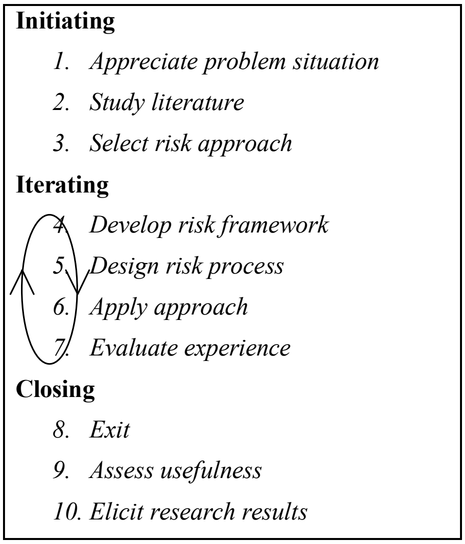
Action research allows practitioners and researchers to interact with a real-world situation gaining deep insights into how an organization functions and how different interventions affect the organization. At the same time, it allows practitioners to reflect on their practice and gives them strong tools to improve their current situation (Checkland, 1991; McKay & Marshall, 2001). Along similar lines, Schön (1983) refers to the reflective practitioner that is informed by research, allowing researchers sometimes to act as practitioners and practitioners sometimes to act as researchers. We propose a modified action research approach, called collaborative practice research (CPR) that is appropriate for developing risk management methods by reflective practitioners or by collaborative groups of practitioners and researchers (Iversen et al., 2004; Mathiassen, 2002).

At Danske Bank, the cyclic nature of action research combined theory and practice as follows. The research interest and practical problems we faced were about SPI and how SPI teams can manage risks. The practical setting in which we addressed risk management was the SPI teams in the IT Department of Danske Bank (the largest Danish bank). The purpose here was to improve the SPI Teams' handling of SPI-related risks. Within this area we applied and combined theories and concepts about SPI and IT risk management. The result of this cyclic process was double: an approach to manage SPI risks in Danske Bank and a generic method for developing risk management approaches. The approach to manage SPI risks is presented in the section *Managing SPI Risks*, while the generic method for developing tailored risk approaches is presented in the rest of this section, based on the cyclic process of action research.

Our method to developing risk management approaches is illustrated in Table 2. It addresses a particular area of concern and is supported by risk management knowledge. The method provides an iterative approach to develop a tailored risk management approach through application to a real-world situation in a specific organizational context, as shown in Table 2.

The proposed method is based on the ten activities of a CPR process (cf. Table 2), and consists of three phases: Initiating (activities 1 to 3), Iterating (activities 4 to 7), and Closing (activities 8 to 10). The sequence between activities 4 and 5 may not hold in practice, and only points to the logical dependencies between the activities. The sequence from 4 to 7 is based on the canonical problem-solving

*Table 2. Developing risk management approaches*



**Initiating**

1. *Appreciate problem situation*
2. *Study literature*
3. *Select risk approach*

**Iterating**

4. *Develop risk framework*
5. *Design risk process*
6. *Apply approach*
7. *Evaluate experience*

**Closing**

8. *Exit*
9. *Assess usefulness*
10. *Elicit research results*

cycle (Susman & Evered, 1978). The iterating phase leads to risk management within the area of concern. The closing phase produces a refined risk management approach, together with an assessment of its usefulness.

The actors in this process enter the problem situation, bringing in prior experience and knowledge of the area of concern (activity 1). The actors should: (1) have experience within the area; (2) perceive the situation as problematic; and (3) find out to what extent and in which way risk management would be beneficial. Part of this activity is to assess whether these prerequisites are met and to establish a project with goals and plans to develop a tailored risk management approach. Activity 1 leads to an appreciation of the risk items and resolution actions perceived to be important within the area of concern (SPI in our case). Activity 2 uses the relevant risk management literature to complement activity 1 and leads to a comprehensive set of risk items and resolution actions that are used in activity 4. The type of risk approach is selected in activity 3 (cf. Table 1), based on the desired features of the approach and the characteristics and needs of the organizational context. This choice defines the basic structure of the risk management approach to be developed.

Activity 4 aggregates the identified risk items and resolution actions into a risk framework of the area of concern (see section *Managing SPI Risks*). Then a risk process is developed in activity 5. The process is based on the framework and on specific risk items and resolution actions. The risk approach is applied subsequently to specific situations or projects within the area of concern (activity 6). This leads to risks being managed and to experiences using the new approach (activity 7).

The iterating phase ends when the actors agree that the risk management approach is developed sufficiently and the problems in the area of concern are alleviated (activity 8). Whether the applications of the risk management approach were useful in practice is assessed relative to the problem situation at hand (activity 9). A simple way to do this is to ask the participants in the risk assessment if they found the risk management approach useful and to document whether risk management led to actions and improvements. The ways in which the new risk management approach contributes to the discipline in general are assessed relative to the relevant body of knowledge (activity 10).

We suggest that this method can be used in different contexts within information systems and software engineering. In adopting the method, actors are advised to consider specific criteria that will help them achieve satisfactory relevance of the outcome and sufficient rigor in the process. Actors are, as described, advised to use the framework of IT risk management approaches in Table 1 to guide their design.

# Case: Risk Management Approach in SPI

This section presents the action research project that forms the basis for this research. It explains how we used the proposed method to develop a specific risk management approach and how this approach works.

## Case Organization

This action research project was conducted at Danske Bank's IT Department, Danske Data, which was spun into an independent company, with Danske Bank as its primary customer. As the IT department began as part of the bank's accounting department, the traditional rigor of banking procedures still pervaded the culture to some extent. This had diminished somewhat in recent years as emerging technologies and the strategic role of IT to the bank's business became increasingly important.

Danske Bank joined a larger CPR (Mathiassen, 2002) project in 1997 (Mathiassen et al., 2002), aimed at implementing SPI projects in the participating organizations. Danske Data established a software engineering process group (Fowler & Rifkin, 1990) to manage and coordinate the SPI effort, which management clearly had articulated was intended to improve productivity (Andersen, Krath et al., 2002). The action researchers joined the SPI effort, along with a dedicated project manager, a consultant from the Methodology Department, and two information systems managers. One of the first activities conducted was a maturity assessment to determine which areas to target for improvement (Iversen et al., 1998). The assessment identified seven improvement areas. Subsequently, action teams were established to address each of these areas. As their work got underway, it became clear that there was a need to manage the inherent risks in conducting these organizational change projects. Danske Data called on the action researchers, who had extensive experience with risk management, to develop an approach to manage the risks faced by each of the SPI action teams. To satisfy the request, the action researchers embarked on the project described here, which eventually led to development of the method described in the previous section as well as the risk management approach for SPI described briefly in the Managing SPI Risks section and more fully in Iversen et al. (2002, 2004).

*Table 3. Action research performed by practitioners and researchers*

| Activities | Initiating | First iteration | Second iteration | Third iteration | Fourth iteration | Closing |
|---|---|---|---|---|---|---|
| (see Table 2) | 10.97-12.97 | 01.98-02.98 | 03.98-08.98 | 09.98-11.98 | 11.98-02.99 | 02.99-02.00 |
| 1. Appreciate problem situation | Part of on-going research collaboration [p1-4; r1-4]<br>Brainstorm risk items and actions [p1-4; r1-4] | | | | | |
| 2. Study literature | Study SPI [p1-4; r1-4]<br>Study risk management [r1-2] | | | | | |
| 3. Select risk approach | Synthesis [r1-3] | Confirmed selection [r1-3] | | Appreciation of actors' competence [r1-3] | | |
| 4. Develop risk framework | | Synthesis [r1-3]<br>Review of framework of risk items and actions [r3]<br>Revised framework [r1-3] | | | | |
| 5. Design risk process | | List of risk items and actions [r1-3]<br>Strategy sheets [r1-3] | Additional step and items reformulated [r2-3] | Improved documentation scheme [r1-3] | | |
| 6. Apply approaches | | Risk assessment of Quality Assurance [p5-7; r2-3] | Risk assessment of Project Management [p3-4; r1-2] | Risk assessment of Metrics Program [p2; p8; r3] | Risk assessment of Diffusion [p9-10; r4; r3] | |
| 7. Evaluate experience | | Lessons learned [p5-7; r2-3] | Lessons learned [p3-4; r1-2] | Lessons learned [p2; r3] | Lessons learned [p9-10; r4; r3] | |
| 8. Exit | | | Delay after 2nd iteration | | | Action part closed |
| 9. Assess usefulness | | | Assessment of first two projects [p1-4; p11; r1-4] | Discussion of risk approach at CPR workshop [r1-r3] | | Assessment of Metrics and Diffusion projects [p1-4; r1-4] |
| 10. Elicit research results | | | | | | Result and lesson elicitation [r1-3] |

*Key: pn is practitioner n; rn is researcher n; p1-10 are SPI practitioners; r1-3 are the authors.*

## Action Research Project

The project was structured around four iterations, as described previously in Table 2. This section describes in detail how each iteration was conducted and what was learned. The effort involved 10 practitioners and 4 researchers (3 of whom are the authors). Table 3 illustrates a timeline of the four iterations and the involved actors and activities that took place in each. Most of the activities for this project took place between October 1997 and February 1999. Generally, the project was carried out in an iterative fashion, where risks and actions were identified in a bottom-up fashion and with practitioners and researchers collaborating closely on developing and testing the approach.

The project was initiated with a workshop that identified the risks and resolution actions that practitioners and researchers thought were the most important for SPI. When the workshop was conducted, the SPI project had been on-going for approximately one year, and both researchers and practitioners had significant practical experience with conducting SPI projects. Prior to the workshop, the practitioners worked through an existing SPI risk analysis approach (Statz et al., 1997), but found this approach too unwieldy and not sufficiently relevant to Danske Data. At the workshop, the researchers presented classical approaches to IT risk management (Boehm, 1991; Davis, 1982; McFarlan, 1981), after which the entire group conducted two brainstorms to determine risks and potential resolution actions that were relevant to SPI in Danske Data. Both of the resulting lists were very long and detailed (31 risk items and 21 resolution actions), which made them difficult to use. We obviously needed more structure.

Following the workshop, the authors studied the risk management literature and identified four types of approaches (Table 1). We chose to adopt a risk-strategy analysis approach, inspired by Davis (1982), for several reasons: We chose a strategy approach over a list approach because the practitioners explicitly stated that they wanted an approach that could help them obtain an overall, strategic understanding of each SPI project. We chose the risk-strategy analysis approach over the risk-strategy model approach for two reasons. First, the stepwise analysis approach would help each SPI team obtain a shared, detailed understanding of risks and possible actions. Second, we were not confident that we would be able to develop a contingency model that would summarize the many different sources of risks and ways to address them in SPI. The action research subsequently went through four full iterations before closing.

## *First Iteration*

Based on the lists of risk items and resolution actions from the workshop and insights from the SPI literature, the authors synthesized the brainstorms and developed a prototype of the risk management approach. A key challenge was developing a framework to understand risks and actions (see Figure 1 and Table 4). We further developed our initial classifications through a detailed examination of risk items and resolution actions mentioned in the SPI literature (Grady, 1997; Humphrey, 1989; McFeeley, 1996; Statz et al., 1997). The resulting risk management process was based on detailed lists of risk items and resolution actions for each of the four categories in the framework, and designed similarly to Davis' (1982) risk management approach (Iversen et al., 2004). Finally, we designed strategy sheets and simple scoring mechanisms to encourage a group of actors to engage in detailed risk and action assessments as a means to arrive at an informed, strategic understanding of how to address risks.

To test the approach, we arranged a workshop with the three practitioners responsible for improving quality assurance. We presented the risk framework and the process, but let the practitioners themselves apply the process, assisting only when they got stuck. The main experience was that the basic idea and structure of the approach was useful. However, during this first trial session, we only had time to cover half of the risk areas. The practitioners suggested that the process needed to be facilitated and managed by someone trained in the process, for example, the researchers. The practitioners found it especially difficult to interpret the questions in the risk tables in the context of their specific project. Some of the risk items needed to be reworded. Finally, to ease the interpretation of the risk items, the session should have started with an interpretation of the general terms in Figure 1 in the particular SPI context.

### *Second Iteration*

In the second iteration, we reworded the risk items and introduced a first step in which the SPI team should interpret the risk model in Figure 1 in their particular context. Then we performed a risk analysis with the two SPI practitioners responsible for improving project management. Both practitioners were skilled project managers with experience in risk management. The session included a complete risk analysis with identification of key risks and resolution strategies. The participating practitioners and researchers agreed upon the major lessons. First, the framework and the process assisted even skilled project managers through a more disciplined analysis than they usually would do on their own. Second, it would be advantageous to document the interpretations of specific risk items and resolution actions continuously throughout the workshop.

At subsequent meetings in the local research group, the two risk management workshops were discussed and assessed in terms of which actions were taken later by the two SPI teams. Present at the meetings were the four SPI practitioners, the three authors, and the fourth researcher. Both SPI teams found that the suggested framework provided a comprehensive overview of risk items and resolution actions. Many comments about the detailed lists of risk items and resolution actions led to subsequent modifications and rewording, but the aggregate structure that we had created based on the initial brainstorm and a study of the SPI literature was not changed.

The quality assurance improvement project was not very active during that period. The manager of the quality assurance project was not present at the risk analysis session and had not yet devoted full attention to quality assurance. The other project members were, therefore, mainly in a reactive mode, and little had happened. Risks surfaced during the analysis, but none of the practitioners were able to resolve these risks in practice. From this, we learned that realizing a risk and identifying a set of resolving actions do not ensure that actions are or will be taken. The practitioners that need to commit to the results of a risk analysis session should be present and involved in the session. After 7 months, there was no agreed-upon plan for the organizational implementation of quality assurance procedures. After 10 months, the quality assurance project had rolled out its procedures, but the identified risks never were managed effectively and consequently impacted the initiative.

The project management improvement project, in contrast, had considerable activity. The main risk was that project managers would not find the improvement attractive and worth their effort. The strategy was, therefore, directed at creating incentives for the project managers. After 1 month, an appropriate incentive structure was in place. After 5 months, the project manager education was a huge success, and all project managers wanted to participate (Andersen, Arent et al., 2002).

## *Third Iteration*

We started the third iteration by appreciating the lesson learned from the first two iterations: successful application of the risk management approach required participation of practitioners with sufficient authority to address key risks. By including these actors in the workshop, we ensure that they agree with the outcome of the workshop, and thereby increase the chances that the agreed-upon actions actually will be implemented. We also introduced a new way to document the process directly onto transparencies and paper versions of the templates.

In the third iteration, we tested the changes on a project that was responsible for establishing an organization-wide metrics program (Iversen & Mathiassen, 2003). The new documentation scheme made it easier for the participants to relate risk questions to their particular situation. We documented each risk in more detail by answering the following question: “What are the specific issues that make this risk particularly important?” As we progressed through the risk assessment, this made it easier to determine why something had been given a specific characterization. The session included a complete risk analysis. The practitioners found the identified actions useful and relevant, and they emphasized the benefit of having reached a shared, overall understanding of risks and actions. The practitioners suggested including the traditional distinction between consequences and probability of a risk into the process. To keep the approach as simple as possible, we decided not to implement this idea.

## *Fourth Iteration*

For the fourth iteration, we made no changes, and applied the approach to an improvement project responsible for improving diffusion and adoption practices (Tryde et al., 2002). The session had three participants: two practitioners from Danske Bank's IT Department and the fourth action researcher involved in this project. All three found the approach generally useful. They found the analysis of the risk areas and the specific actions particularly useful, but they did not find summarizing the strategies particularly helpful. The participants emphasized the importance of not merely following the suggested lists of risk items and resolution actions, but also of supplementing this with a more open-minded exploration. "We haven't asked ourselves, 'what can go wrong?'" said one participant. They merely had considered each risk separately as it was presented to them.

## *Closing*

We discussed and assessed the third and fourth risk analysis sessions with the four SPI practitioners and the fourth researcher at a later meeting of the local research group. The metrics program had suffered several setbacks due to political turmoil when previously hidden data about software projects' performance were publicized (Iversen & Mathiassen, 2003). Nevertheless, the risk analysis session led to actions that the project took later. The two main actions decided at the risk management session were: (1) develop and maintain top management's support and commitment and (2) create immediate results that are perceived useful by software projects. At a meeting 3 months later, it was reported that the project successfully had convinced top management that the collected metrics results should be publicized in all of Danske Bank's IT Department, which later happened (Iversen & Mathiassen, 2003). The diffusion and adoption project was successful (Tryde et al., 2002). Many of the performed activities came out of the risk analysis. It was decided to exit the iterations at this point because the experiences from the four iterations suggested that the risk management approach was in a stable and useful form. Our final activity was eliciting lessons for the overall action research endeavor (Iversen et al., 2004).

## Managing SPI Risks

This section outlines the resulting risk analysis approach. The method has been described in more detail in other published works (Iversen et al., 2002, 2004).

The approach to managing SPI risks is based on a framework that aggregates risk items into areas and risk resolution actions into strategies. The first part of the framework describes the relevant SPI risk areas; the second part outlines the potential SPI risk resolution strategies. The approach is intended to be applied to the risks faced by individual SPI action teams. Figure 1 illustrates the four different areas in which SPI action teams might identify risks:

- **The improvement area:** those parts of the software organization that are affected by the specific SPI initiative.
- **The improvement ideas:** the set of processes, tools, and techniques that the SPI initiative seeks to bring into use in the improvement area.
- **The improvement process:** the SPI initiative itself and the way in which it is organized, conducted, and managed.
- **The improvement actors:** those involved in carrying out the SPI initiative.

As an example, consider an SPI team concerned with introducing configuration management in software engineering projects. Here the *improvement area* includes the software development projects that will use configuration management and the people supporting the process after institutionalization. The *improvement ideas* include the configuration management principles relied upon

*Figure 1. Risk areas for SPI teams*

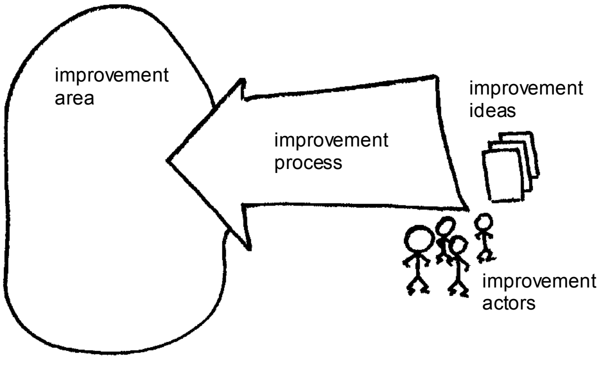


*Table 4. Risk resolution strategies for SPI teams*

| *Type of action* | *Concern* |
|---|---|
| 1. Adjust Mission | What are the goals of the initiative? Goals may be adjusted to be more or less ambitious, e.g., targeting only projects developing software for a specific platform. |
| 2. Modify Strategy | What strategy is the initiative going to follow? Covers the approach to develop the process as well as to roll it out in the organization. Roll-out may, for instance, follow a pilot, big bang, or phased approach. |
| 3. Mobilize | From what alliances and energies can the initiative benefit? The likelihood of success of an improvement initiative can be improved significantly by adjusting which organizational units and actors are involved and by increasing their commitment. |
| 4. Increase Knowledge | On which knowledge of software processes and improvement is the initiative based? Knowledge can be increased by educating team members, by including additional expertise into the team, or by hiring consultants. |
| 5. Reorganize | How is the initiative organized, conducted, and managed? Covers organizing, planning, monitoring, and evaluating of the initiative. |

by the SPI team and the tools and methods that are developed to support these principles. The *improvement process* is the improvement itself, the way it is organized, and the involved stakeholders. The *improvement actors* are the members of the SPI team.

The risk resolution actions that SPI teams can apply are aggregated into five different types of strategies, as shown in Table 4. The strategies are listed according to the degree of change we suggest the SPI team's risk-based intervention will cause. *Adjust Mission, Modify Strategy*, and *Reorganize* target the improvement project's orientation and organization; *Increase Knowledge* targets the involved actors' level of expertise and knowledge; and *Mobilize* targets alliances and energies that will increase the project's chance of success.

The mission of an SPI team on configuration management may be to introduce configuration management on all documents (including documentation, code, etc.) in all software engineering projects in the company. This mission could be *adjusted* to include fewer projects (perhaps only large projects, critical projects, or projects in a specific department) or to exclude certain types of documents. The SPI team's strategy might be to involve a few key developers to give input to the process and, based on this, select a standard configuration management tool that every project then has to use. *Modifying the strategy* may entail involving more (or fewer) developers or implementing the chosen tool gradually in each project. *Mobilizing* may involve establishing agreements with an existing method department, a production department, or other departments or persons that have a vested interest in the results of the team's effort. The SPI team could *increase its knowledge* by attending courses on configuration management or SPI, or by hiring knowledgeable consultants. If the project is not organized optimally for the task at hand, the effort could be *reorganized*, e.g., by establishing a formal project, negotiating a project contract with management and the software engineering projects, or developing a new project plan.

To help SPI practitioners determine a strategy based on current risks facing the project, the approach offers a four-step process based on Davis (1982):

1. **Characterize Situation** by interpreting the profile and scope of the elements of Figure 1.
2. **Analyze Risks** to assess where the most serious risks are. This involves rating each of the detailed risk items in the risk lists for the four areas, and then determining which area carries the highest risk exposure.
3. **Prioritize Actions** to decide on a strategy that will deal effectively with the identified risks. Here, actors use a process that alternates between individual and group judgments to determine which strategy is the most sensible given the current assessment of risks facing the project.
4. **Take Action** by revising project plans to reflect resolution actions.

# Conclusion

IT managers see risk management as a key to success (Barki et al., 1993). Such approaches help appreciate many aspects of a project: they emphasize potential causes of failure, they help identify possible actions, and they facilitate a shared perception of the project among its participants (Lyytinen et al., 1996, 1998). This indicates that organizations can benefit from adopting IT risk management to their particular needs. The method we have presented can be used for that purpose, and thereby adds to the portfolio of approaches that are available to adapt generic insights to specific organizations. Similar methods are, for example, available to tailor knowledge on software estimation to specific organizational contexts (Bailey & Basili, 1981).

The method builds on action research experiences that can help organizations address IT-related problems effectively in line with scientific insights. Our own experiences using the method indicate that a number of competencies are required to adopt the method effectively. First, we had intensive domain (SPI) and risk management knowledge. Second, we had general competence in modeling organizational phenomena that we used to identify and classify risk items and resolution actions. Third, we had experimental competence that we used to collect feedback from the test situations to iteratively arrive at the resulting approach. Each of these competencies is required to apply the proposed CPR method in other contexts. It is also important to stress that the method, like most action research processes, is a template that needs to be adapted and supplemented in action, depending on the conditions under which it is applied.

In addition to being useful in a specific organizational context, the CPR method can help tailor risk management approaches to new domains within information systems and software engineering, for example, business process innovation, integration of information services, and ERP implementation. Any form of organizational change enabled by IT is complex and difficult. Risk management, as illustrated well in relation to software development and SPI, is a highly effective way to bring relevant knowledge within a particular organization or domain into a form in which it can support and improve professional practices. We, therefore, encourage researchers and practitioners within information systems and software engineering to adopt action research to tailor risk management to specific organizations and new domains.

We conclude this chapter with good advice to those who wish to create a risk management approach for their organization:

- Make sure practitioners are able to relate wording of risks and resolutions to their project.
- Be aware of whether the approach needs to be facilitated. For practitioners to apply the approach on their own, it must be simple or well documented and supplemented by training.
- Build in documentation of rationales along the way (what was it about a certain risk that made it particularly evident in this project?).
- Include mechanisms to ensure action. It is not enough to create a risk resolution plan — the project also needs to carry it out.
- Iterate until the approach is stable. Then keep updating risks and actions to stay current with changes in the context as well as in the literature.

## References

Alter, S., & Ginzberg, M. (1978). Managing uncertainty in mis implementation. *Sloan Management Review, 20*(1), 23-31.

Andersen, C. V., Arent, J., Bang, S., & Iversen, J. H. (2002). Project assessments. In L. Mathiassen, J. Pries-Heje, & O. Ngwenyama (Eds.), *Improving software organizations: From principles to practice* (pp. 167-184). Upper Saddle River, NJ: Addison-Wesley.

Andersen, C. V., Krath, F., Krukow, L., Mathiassen, L., & Pries-Heje, J. (2002). The grassroots effort. In L. Mathiassen, J. Pries-Heje, & O. Ngwenyama (Eds.), *Improving software organizations: From principles to practice* (pp. 83-98). Upper Saddle River, NJ: Addison-Wesley.

Bailey, W., & Basili, V. R. (1981, March 9-12). Meta-model for software development expenditures. *Proceedings of the 5th International Conference on Software Engineering*, San Diego, CA.

Barki, H., Rivard, S., & Talbot, J. (1993). Toward an assessment of software development risk. *Journal of Management Information Systems, 10*(2), 203-225.

Boehm, B. W. (1991). Software risk management: Principles and practices. *IEEE Software, 8*(1), 32-41.

Checkland, P. (1991). From framework through experience to learning: The essential nature of action research. In H.-E. Nissen, H. K. Klein, & R. A. Hirschheim (Eds.), *Information systems research: Contemporary approaches and emergent traditions* (pp. 397-403). North-Holland: Elsevier.

Chrissis, M. B., Konrad, M., & Shrum, S. (2003). *CMMI: Guidelines for process integration and product improvement*. Boston: Addison-Wesley Professional.

Davis, G. B. (1982). Strategies for information requirements determination. *IBM Systems Journal, 21*(1), 4-30.

Donaldson, S. E., & Siegel, S. G. (2001). *Successful software development*. Upper Saddle River, NJ: Prentice Hall.

Grady, R. B. (1997). *Successful software process improvement*. Upper Saddle River, NJ: Prentice Hall PTR.

Humphrey, W. S. (1989). *Managing the software process*. Pittsburgh, PA: Addison-Wesley.

Iversen, J., Johansen, J., Nielsen, P. A., & Pries-Heje, J. (1998, June 4-6). Combining quantitative and qualitative assessment methods in software process improvement. *Proceedings of the European Conference on Information Systems (ECIS 98)*, Aix-en-Provence, France.

Iversen, J. H., & Mathiassen, L. (2003). Cultivation and engineering of a software metrics program. *Information Systems Journal, 13*(1), 3-20.

Iversen, J. H., Mathiassen, L., & Nielsen, P. A. (2002). Risk management in process action teams. In L. Mathiassen, J. Pries-Heje, & O. Ngwenyama (Eds.), *Improving software organizations: From principles to practice* (pp. 273-286). Upper Saddle River, NJ: Addison-Wesley.

Iversen, J. H., Mathiassen, L., & Nielsen, P. A. (2004). Managing risks in software process improvement: An action research approach. *MIS Quarterly, 28*(3), 395-433.

Jones, C. (1994). *Assessment and control of software risks*. Upper Saddle River, NJ: Yourdon Press, Prentice Hall.

Keil, M., Cule, P. E., Lyytinen, K., & Schmidt, R. C. (1998). A framework for identifying software project risks. *Communications of the ACM, 41*(11), 76-83.

Lyytinen, K., Mathiassen, L., & Ropponen, J. (1996). A framework for software risk management. *Scandinavian Journal of Information Systems, 8*(1), 53-68.

Lyytinen, K., Mathiassen, L., & Ropponen, J. (1998). Attention shaping and software risk: A categorical analysis of four classical risk management approaches. *Information System Research, 9*(3), 233-255.

Mathiassen, L. (2002). Collaborative practice research. *Information Technology and People, 15*(4), 321-345.

Mathiassen, L., Munk-Madsen, A., Nielsen, P. A., & Stage, J. (2000). *Object-oriented analysis and design*. Aalborg, Denmark: Marko.

Mathiassen, L., Pries-Heje, J., & Ngwenyama, O. (Eds.). (2002). *Improving software organizations: From principles to practice*. Upper Saddle River, NJ: Addison-Wesley.

McFarlan, F. W. (1981). Portfolio approach to information systems. *Harvard Business Review, 59*(5), 142-150.

McFeeley, B. (1996). *Ideal: A user's guide for software process improvement* (Tech. Rep. No. CMU/SEI-96-HB-001). Pittsburgh, PA: Software Engineering Institute.

McKay, J., & Marshall, P. (2001). The dual imperatives of action research. *Information Technology and People, 14*(1), 46-59.

Moynihan, T. (1996). An inventory of personal constructs for information systems project risk researchers. *Journal of Information Technology, 11*, 359-371.

Ould, M. (1999). *Managing software quality and business risk*. Chichester, UK: Wiley.

Paulk, M. C., Weber, C. V., Garcia, S. M., & Chrissis, M. B. (1993). *The capability maturity model: Guidelines for improving the software process*. Upper Saddle River, NJ: Addison-Wesley.

Ropponen, J., & Lyytinen, K. (2000). Components of software development risk: How to address them? A project manager survey. *IEEE Transactions on Software Development, 26*(2), 98-112.

Schön, D. A. (1983). *The reflective practitioner. How professionals think in action*. New York: Basic Books.

Statz, J., Oxley, D., & O'Toole, P. (1997). Identifying and managing risks for software process improvement. *Crosstalk - The Journal of Defense Software Engineering, 10*(4), 13-18.

Susman, G. I., & Evered, R. D. (1978). An assessment of the scientific merits of action research. *Administrative Science Quarterly, 23*, 582-603.

Tryde, S., Nielsen, A.-D., & Pries-Heje, J. (2002). Implementing SPI: An organizational approach. In L. Mathiassen, J. Pries-Heje, & O. Ngwenyama (Eds.), *Improving software organizations: From principles to practice* (pp. 257-271). Upper Saddle River, NJ: Addison-Wesley.

## Chapter XII

# Examining the Quality of Evaluation Frameworks and Metamodeling Paradigms of IS Development Methodologies

Eleni Berki, University of Tampere, Finland

## Abstract

*Information systems development methodologies and associated CASE tools have been considered cornerstones for building quality into an information system. The construction and evaluation of methodologies are usually carried out by evaluation frameworks and metamodels, both considered as meta-methodologies. This chapter investigates and reviews representative metamodels and evaluation frameworks for assessing the capability of methodologies to contribute to high-quality outcomes. It* *presents a summary of their quality features, strengths, and weaknesses. The chapter ultimately leads to a comparison and discussion of the functional and formal quality properties that traditional meta-methodologies and method evaluation paradigms offer. The discussion emphasizes the limitations of methods and metamethods used to model and evaluate software quality properties, such as computability and implementability, testing, dynamic semantics capture, and people's involvement. This analysis, along with the comparison of the philosophy, assumptions, and quality perceptions of different process methods used in information systems development, provides the basis for recommendations about the need for future research in this area.*

## Introduction

In traditional software engineering, the information systems development (ISD) process is defined as a series of activities performed at different stages of the system life cycle in conformance with a suitable process model (method or methodology). In the fields of information systems and software engineering, the terms methodology and method are often used interchangeably (Berki et al., 2004; Nielsen, 1990). Increasingly, new methods, techniques, and automated tools have been applied in software engineering (SE) to assist in the construction of software-intensive information systems. Quality frameworks and metamodels are mainly concerned with the evaluation of the quality of both the process itself and the resulting product at each stage of the life cycle including the final product (the information system).

IEEE and ISO have established quality standards and software process management instruments, such as Software Process Improvement and Capacity dEtermination (SPICE) (Dorling, 1993) and Capability Maturity Model (CMM) (Paulk et al., 1993) have focused on the quality properties that the ISD process should demonstrate in order to produce a quality information system (Siakas et al., 1997). However, software quality assurance issues (Ince, 1995), such as reliability (Kopetz, 1979) and predictability, measurement, and application of software reliability in particular (Musa et al., 1987; Myers, 1976), have long preoccupied software engineers, even before quality standards.

IS quality improvement can be achieved through the identification of the controllable and uncontrollable factors in software development (Georgiadou et al., 2003). ISD methodologies and associated tools can be considered as conceptual and scientific ways to provide prediction and control; their adoption and deployment, though, by people and organizations (Iivari & Huisman, 2001) can generate many uncontrollable factors. During the last 35 years, several methodologies, techniques and tools have been adopted in the ISD process to advance software quality assurance and reliability. A comprehensive and detailed coverage of existing information systems development methodologies (ISDMs) has been carried out by Avison and Fitzgerald (1995), with detailed descriptions of the techniques and tools used by each method to provide quality in ISD.

Several ISDMs exist. Berki et al. (2004) classified them into families, highlighting their role as quality assurance instruments for the software development process. They have been characterized as hard (technically oriented), soft (human-centered), hybrid (a combination of hard and soft), and specialized (application-oriented) (Berki et al., 2004). Examples of each include:

- **Hard methods:** object-orientated techniques, and formal and structured families of methods
- **Soft methods:** Soft Systems Method (SSM) and Effective Technical and Human Implementation for Computer-based Systems (ETHICS)
- **Hybrid methods:** MultiView methodology, which is a mixture of hard and soft techniques; Euromethod, extreme programming (XP), and other agile methods
- **Specialized methods:** KADS for expert systems design, and a few agile methods

The contribution of these methods to the quality of the ISD process has been a subject of controversy; particularly so because of the different scopes, assumptions, philosophies of the various methods, and the varied application domains they serve. For example, it is believed that the human role in ISD bears significantly on the perception of the appropriateness of a method (Rantapuska et al., 1999); however, usability definitions in ISO standards are limited (Abran et al., 2003). There is support for the notion that a methodology is as strong as the user involvement it supports (Berki et al., 1997).

Two other significant problems are associated with currently available ISDMs and their capability to contribute to process quality. First, the inability of most to incorporate features of both hard and soft methods inevitably leads to deficiencies resulting in either technical or human problems. Second, the software architectures of IS often lack testable and computational characteristics because they are not specified as an integral part of the design (Berki, 2001).

Considering the importance of Software Quality Management (SQM) and Total Quality Management (TQM) issues, software engineers and IS managers have, for the last 20 years, realized the significance of the use of frameworks for method evaluation and the use of metamodels, precise definitions of the constructs, and rules needed for creating semantic models, for their own method construction. These — usually accompanied by automated software tools — are typically used during requirements analysis and for early testing and validation of requirements specifications (Manninen & Berki, 2004) as design products. This is done in order to ensure that the resulting artifacts, prototypes, and final information products will experience lower maintenance costs, greater reliability, and better usability.

Over 10 years ago, Jayaratna (1994) identified more than two thousand methodologies in use. It is reasonable to assume that several others have been added since. What is the basis for deciding which are useful and which are not (Chisholm, 1982)? There is justification for objective criteria, which take into account the type of IS to be developed, its environment, the people that will interact with it, and the properties of the development methods that contribute to a quality process and product, to inform this decision process. Several such evaluation and selection frameworks exist (Law, 1988; Law & Naeem, 1992). However, analysts and designers can also build their own in order to fit the needs of system modeling tasks more precisely by using method construction metamodels with method engineering (ME) rules (Brinkkemper, 1996, 2000).

This chapter provides a comprehensive overview of how to construct efficient and cost-effective metamodels and evaluation frameworks and identify their quality properties in a scientific and reliable way. This task may be approached in a variety of ways. However, the approach recommended by Sol (1983) and Jayaratna (1994), to develop a common frame of reference for evaluating different methodologies, will be adopted in this study. It has been used to construct the Normative Information Model-based Systems Analysis and Design (NIMSAD) framework (Jayaratna, 1994) and for Metamodeling-based integration of object-oriented systems development (Van Hillegersberg, 1997).

This research approach is considered appropriate mainly because the construction of a framework is less subjective than other approaches for evaluating and comparing methods (and metamethods); it sets criteria for method evaluation, comparison, and construction from a different, more or less detailed, level of view. The evaluation criteria used in this chapter incorporate elements involving soft and hard issues, year of introduction, tool support, computability and testing, and allow for the examination of the initiatives and their rules, viewing them under the ISD needs as these appeared chronologically. Before proceeding to the examination of the quality properties of contemporary methods through the lens of different evaluation frameworks and metamodels, the next section presents a few concepts related to method evaluation frameworks, and highlights the differences and similarities of metamodels and ME.

# Metamodels, Evaluation Frameworks, and Meta-Information Management

## Basic Concepts

A metamodel is an information model of the domain constructs needed to build specific models. Data elements include the concepts of class, event, process, entity, technique, and method. In order to create a metamodel one needs a language in which metamodeling constructs and rules can be expressed. In the context of ISD methodologies, metamodeling facilitates the application of the principles of ME (Brinkkemper, 1996) in which IS engineers examine characteristics of the particular methods at a different, higher or lower, level of abstraction (metalevel), viewing the functions and objectives of the IS as a restructured abstract view of the system under development.

In general metamodeling is an important tool for both decomposition and generalization of the complex, dynamic processes that are activated during the development and maintenance of systems. Important systems engineering concepts are viewed in different design compositions and decompositions; thus metamodeling, as a system's life cycle activity, provides a basis for decision making. A generally accepted conceptual metamodel for method construction usually explains the relationships between metamodels, models, methods, and their techniques, and end-user data. All these form the semantic and syntactic constructs of the metamodel, and they may belong to the same or different layers of abstraction. Metamodels usually facilitate system and method knowledge acquisition and sometimes metadata and information integration (see Kronlof, 1993).

Evaluation frameworks offer generalizations and definitions of the quality properties of methods in order to reveal their characteristics in use. A framework can also be considered a metamodel; while a methodology is an explicit way of structuring one's thinking and actions (Jayaratna, 1994). In addition, Jayaratna believes that a framework differs from a methodology in that it is a "static" model, providing a structure to help connect a set of models or concepts. In this context, a framework can be perceived as an integrating metamodel, at a higher level of abstraction, through which concepts, models, and methodologies can be structured, and their interconnections or differences displayed to assist understanding or decision making.

Although the terms metamodel and framework are used interchangeably in the IS and SE literature (Jayaratna, 1994), the current research maintains they should be distinct. According to the epistemology and applications of metamodeling in metacomputer-assisted system engineering (MetaCASE) and computer-assisted method engineering (CAME) environments, a “metamodel” is a more specific, dynamic, and well-defined concept that leads to well-structured principles of method engineering (ME). A “framework”, on the other hand, is more general and static, with less-structured evaluation guidelines (Berki, 2001). These distinctions will become more evident in the section that describes method construction metamodels and method evaluation frameworks.

## Rationale for the Use of Metamodels and Evaluation Frameworks

The rapid societal changes in advanced information societies, the increasing attention to cognitive issues, the high cost of software development and IS maintenance, and the requirements of legacy IS in particular, point to the need for more reusable, more reliable information systems that are acceptable by humans. The communication between IS end users, managers, analysts, designers, and programmers has been a major problem in ISD, yet is pivotal in delivering high-quality IS (Berki et al., 2003). Hence the research community, method designers, and method practitioners are forced to focus on the construction and use of more dynamic ISD methods.

Consequently, there is an increasing demand for more supportive software tools that cater to method integration and the reengineering of IS processes and methods. Conventional methods have been inadequate to date, largely because of the lack of structured support for overcoming human cognitive limitations in the process of transforming requirements to design, and testing and implementing the results in a reliable manner. If, for instance, the production of computable (i.e., programmable, implementable, and realizable) specifications from analysis and design phases could be automated and tested reliably, then this could greatly improve the capability to address software quality assurance and eventually improve IS quality.

Several information systems development methodologies (ISDMs) have been invented to assist with managing and improving software development processes. The general view is that such methodologies are inherently complex, and implementing them in practice is difficult or even impossible without the support of automated tools within a software development environment (SDE). However, the diversity of methodologies and their evolving nature require that corresponding SDEs provide flexibility, adaptability, and extensibility (Jarzabek, 1990). This complicates decision making about which ISDMs are called to offer support to realize it. Moreover, difficulties in decision making have created a need for structured assistance to evaluate the capability of methodologies and their appropriateness for a particular organization.

Several software disasters, such as the Arriane-5 space rocket accident and many other software failures — with less disastrous consequences — attributable to design errors, and the large and aging IS infrastructure that we have inherited, underscore the need for facilities to help reengineer, test, and improve software development methods and models. In general, process methods and models provide significant assistance in incorporating software quality management practices in ISD; several exist, typically customized to organizational and stakeholder preferences, to address the many interpretations of IS quality and pursue fairly elusive quality objectives (Berki et al., 2004). However, there is an urgency to address these issues at the ME level and from the metamodeling point of view (Berki, 2001).

Metamodeling and ME tools are relatively new software development technologies designed to improve process models and methods used in ISD. Typically, they are supported by automated tools with broad capabilities to address the various facets of producing a quality process and product. Because organizations can modify methodologies, there is also a need for these tools to accommodate modifiability — or method reengineering (Saadia, 1999) — and to support other fundamental quality properties for the ISD process such as testability (Beizer, 1995; Whittaker, 2000), computability (Lewis & Papadimitriou, 1998; Wood, 1987), among others. They should also cater to softer components of the ISD such as the roles and interactions of people.

Summarized, the argument in favor of establishing formal and generic definition of metamethods and metamodels that incorporate quality requirements and emphasize the dynamic nature of ISD is that (1) establishing evaluation frameworks for ISDMs provides insights to optimize the selection of methods that better fit the development context and (2) improving the computational representation of methods and models leads to improved knowledge acquisition, which enhances communication among IS stakeholders. Both increase the likelihood of less ambiguous specifications and consistent system design, which are easier to implement and formally test, and eventually lead to systems that are modifiable and maintainable.

## Examination and Broad Classification of Method Evaluation Frameworks and Method Engineering Metamodels

Typically, different ISD methods model different application domains. So, different frameworks/metamodels may concentrate on different aspects of the use of methods under evaluation and/or comparison. In general, frameworks, like methods, can be classified as either human-action oriented (soft) or more technical (hard) in their evaluation of the quality characteristics of methods (Berki et al., 2004) and may be applied with or without automated tool support. However, unlike methods, frameworks tend to project those quality features of the methods that make them distinct and different or similar to other methods currently available. Some of the semantic and pragmatic quality features and the knowledge employed in their definition are outlined below in various metamodeling examples.

## Euromethod

Euromethod (CCTA, 1994; Jenkins, 1994) is a framework for facilitating harmonization, mobility of experts and expertise, the procurement process and standards, market penetration by vendors, and quality assurance across countries of the European Union. It builds on structured methodologies and techniques of methods from Structured Systems Analysis and Design Method (SSADM), the UK standard; Merise, the French standard; DAFNE, the Italian standard for software development; SDM (Netherlands); MEthodologica INformatica (MEIN) (Spain); Vorgehensmodell (German standard); and Information Engineering (IE) (US/UK).

The goal of Euromethod is to provide a public domain framework for the planning, procurement, and management of services for the investigation, development, and amendment of information systems (Avison & Fitzgerald, 1995). As a metamethod, it caters to the facilitation of tacit and explicit knowledge exchange among European countries and the standardization of disparate frameworks that encompass the many different ISD philosophies and cultures. However, it does not include information systems services provided by an in-house IT facility. Euromethod could serve as an integrating structured methods standard for ISD in European or other social contexts given the strong impact of cultural differences and other national and organizational diversities which Siakas et al. (2003) found affected software quality management strategies in European countries.

## A Generic Evaluation Framework

Avison and Fitzgerald (1995) used their generic evaluation framework to tabulate the strength of coverage of a set of the most widely known methodologies over the phases of a life cycle. This framework uses ideas presented in Wood-Harper et al.'s (1985) discussion of approaches to comparative analysis of methods. It includes the seven elements — philosophy, model, techniques and tools, scope, outputs, practice, and product — some of which are further broken down into subelements. For example, Avison and Fitzgerald observed that Jackson Systems Development (JSD) and Object-Oriented Analysis (OOA) fail to cover strategy and feasibility while the majority of structured methods, such as Yourdon and SSADM, concentrate on analysis and design.

This generalized framework can be considered an abstract typology approach for method analysis and comparison for examining individual methods and comparing the quality of methods. In so doing, the framework focuses on the quality of the method itself (philosophy, model, scope), the quality of the ISD process (model, techniques, and tools), and the quality of the eventual system (outputs, practice, and product).

## Determining an Evaluation Method for Software Methods and Tools (DESMET)

DESMET resulted from the collaborative efforts of academic and industrial partners — University of North London, The National Computing Centre, Marconi Systems, and British Nuclear Research. It was funded by the Department of Industry and the Science and Engineering Council, in the UK. The main objective of DESMET is to objectively determine the effects and effectiveness of methods and tools used in the development and maintenance of software-based systems (DESMET, 1991; Georgiadou et al., 1994). The DESMET framework addresses this problem by developing an evaluation meta-methodology, which identifies and quantifies the effects of using particular methods and tools on developers' productivity and on product quality. The framework was validated by carrying out a number of trial evaluations.

Apart from providing guidelines for data collection and metrication (Kitchenham et al., 1994), DESMET emphasizes managerial and social issues that may affect evaluation projects (Berki & Georgiadou, 1998; Sadler & Kitchenham, 1996). The main contribution of DESMET is the explicit and rigorous evaluation criteria selection and evaluation method selection. Previous evaluation frameworks made little attempt to incorporate the work on evaluation — based on formal experiment — into commercially oriented evaluation procedures (DESMET, 1991).

Further work based on the DESMET Feature Analysis module was carried out at the University of North London resulting in the development of the automated tool Profiler (Georgiadou et al., 1998). The Profiler metamodeling tool was developed according to the DESMET feature analysis and ISO standard rules. It is a generic evaluator (i.e., of methods, processes, products, and other artifacts), which was also used in some stages of this investigation of method evaluation (Georgiadou et al., 1998).

## Tudor and Tudor's Framework

Tudor and Tudor (1995) use the business life cycle to compare the structured methods IE, SSADM, Yourdon, and Merise (Avison & Fitzgerald, 1995) with Soft Systems Method (SSM) (Checkland & Scholes, 1990) and Multiview (Wood-Harper et al., 1985). The framework builds on the seven elements of Avison and Fitzgerald' s (1995) framework to assess the performance of these methods on a scale of low, medium, and high, using criteria defined in DESMET and other characteristics (framework constructs) such as size of problem, use of techniques and CASE tools, and application domain (Tudor & Tudor, 1995). The framework determines the extent of life cycle coverage provided by methods, how highly structured the methods are, and what kind of system the methods target. It also addresses issues such as user involvement, CASE tool support, and different implementations of similar or different techniques employed by structured methods.

## Sociocybernetic Framework

The sociocybernetic framework (Iivari & Kerola, 1983) has made an important contribution to the quality feature analysis of information systems design methods. The framework is based on a sociocybernetic interpretation of information systems design as human action and information systems design methodologies as prescriptive instructions for this action (Iivari & Kerola, 1983). According to Iivari and Kerola, when used in method evaluation or systems engineering, this framework covers scope and content, support and usability, underlying assumptions, generality of the methodologies, as well as scientific and practical value, in 85 basic questions. The application of the framework to the evaluation of selected methodologies was carried out successfully, the results of which can be found in Olle et al. (1982).

## Normative Information Model-Based Systems Analysis and Design (NIMSAD)

NIMSAD is a systemic (holistic) framework for understanding and evaluating problem-solving processes methodologies in general. This metamethod, which was developed through action research, evaluates both soft and hard methods before, during, and after industrial use. The NIMSAD framework is based on a general model of the essential elements of a problem situation and their formal and informal interconnections and relationships (Jayaratna, 1994). Jayaratna emphasizes that these interconnections, mainly human relations, work functions, technical interactions and others, are dynamic, and are dependent on the time and space and the perceiver's personal values, aspirations, and worldview (or *Weltanschauung*).

## Metamodeling, Principles, Hypertext, Objects, and Repositories (MetaPHOR)

(MetaPHOR) is a project framework that was developed at Jyvaskyla University in Finland and was later commercialized by MetaCASE Consulting Ltd. It defines and supports metamodeling and ME principles for innovative ISD method construction. The metamodel supports object-oriented development philosophies and implementation tools, distributed computing environments, and modern user interaction management systems (http://metaphor.cs.jyu.fi/). Its objectives include the investigation and assessment of theoretical principles in different metamodeling areas and the evolution of development methods (Brinkkemper, 1996) and metamodeling in ISD (Marttiin et al., 1993). The MetaPHOR project produced the MetaEdit (later extended to MetaEdit+) CASE tool, which provided techniques of automated methods to be used as systems analysis and design models MetaCASE tool (Kelly et al., 1996).

The MetaCASE tool has been used successfully to model processes and methods in industry and academe because it incorporates the process modeling view (Koskinen, 2000). However, it does not support testing and dynamic method modeling (Berki, 2001; Tolvanen, 1998). Recent enhancements of the MetaPHOR group include method testing (Berki, 2003), utilization of hypertext approaches in MetaEdit+ (Kaipala, 1997), component-based development tools and methods (Zhang, 2004), and cognitive needs of information modeling (Huotari & Kaipala, 1999).

## CASE-Shells

CASE-Shells or Meta-CASE systems are capable of supporting an unlimited number of software engineering environments, instead of one modeling space, which is the norm for most of the traditional CASE systems. In CASE-Shells, the environments are modeled separately from the main software (Kelly, 1997; Marttiin et al., 1993). This initiative mainly considers the metamodeling technology that relates to the production of CASE-Shells and Meta-CASE tools — the methods, specifications, and other products that identify, define, and implement systems specifications. The RAMATIC tool (Bergsten et al., 1989), which originated in Sweden, uses modified rules of the MetaEdit+'s main architecture to define methods and their environments, and the MetaView system from Canada, presented next, are two popular CASE-Shells implementations.

## The MetaView System

The MetaView initiative is mainly supported by a CASE metasystem, that is, a CASE-Shell, which acts as a framework for improving structured and object-oriented software development methods. MetaView is a joint project of the University of Alberta and the University of Saskatchewan, and is evolving (http://web.cs.ualberta.ca/~softeng/Metaview/project.shtml). The MetaView system is divided into three levels (Sorenson et al., 1988), namely:

- Meta Level (creation of software components)
- Environment Level (definition of the environment and configuration of software)
- User Level (generation of user-developed software specification)

The tool's basic architecture for metamodeling is based on the core principles of data flow diagramming techniques, inheriting their strengths and weaknesses (Berki, 2001; Sorenson et al., 1988).

## CASE Data Interchange Format (CDIF)

CDIF is a set of standards for defining a metamodeling architecture for exchanging information between modeling tools, describing the different modeling languages in terms of a fixed core model. It was developed as a formal metamodeling approach to accommodate interoperability among models, using an integrated metamodel for common representation. It facilitates communication between structured and O-O techniques and their semantic extensions, and state event modeling.

The CDIF metamodeling approach is rooted in theoretical models from Computer Science; for example, the principles of the Mealy-Moore Machine (Wood, 1987) and Harel state charts (Harel et al., 1987). It provides the means to view and derive provable quality properties from different techniques and systems models. However, according to the initiative's website, CDIF focuses on the description of information to be transferred and not on data management (Burst et al., 1999).

### Computational and Dynamic Metamodel as a Flexible and Integrated Language for the Testing, Expression, and Reengineering of Systems (CDM-FILTERS)

The construction of CDM-FILTERS was based on experimental modeling steps that led to a formal framework, the Formal Notation for Expressing a Wide-range of Specifications (Formal NEWS). Berki (1999) suggested the generalization and extension of Formal NEWS into a metamodel to reconstruct and evaluate ISD. The resulting metamodel (CMD-FILTERS) was sponsored by the Universities of North London and Sheffield and involved extensive field research in Finland (Jyvaskyla University and MetaCASE Consulting Ltd.). It accommodates method modeling and remodeling as general computational automata (or X-Machines) (Ipate & Holcombe, 1998) applied to MetaCASE tools and architectures (Berki, 2003). The main strengths of the approach are its capability to generate an integrating metamodel for ME, combining method dynamics, computability, and testing. It is, however, limited in its ability to incorporate human issues, and it is not supported by many tools.

## Other Method Metamodeling and Method Integration Projects

There are several other metamodeling architectures, designed for application in particular contexts to address specialized issues with or without MetaCASE support, such as ConceptBase: A metadata base for storing abstract data representation of data warehouse applications (Jarke et al., 1995; Mylopoulos et al., 1990; Vassiliou, 2000). Likewise, there are many on-going projects at various stages of development, for example, MethodMaker (Mark V), System Architect (Popkin), Toolbuilder (Sunderland/IPSYS/Lincoln), and some interesting research prototypes such as KOGGE (Koblenz), MetaGen (Paris), and MetaPlex tools. Readers may refer to Smolander et al. (1990), Graham (1991), Avison and Fitzgerald (1995), Kelly (1997), and Shapiro and Berki (1999) for more detailed information on the commercial availability, application domains, and modeling limitations of methods mentioned here.

One criticism of most metamodels is their limited capacity to evaluate human activity, including the roles of metamodelers and method engineers. Two initiatives, however, address this issue. First, Requirements Engineering Network of International cooperating Research (RENOIR), a requirements engineering network supported by the European Union, provides the tools, concepts, and methods that mediate between the providers of information technology services and products, and the users or markets for such services and products. Then Novel Approaches to Theories Underlying Requirements Engineering (NATURE), an on-going collaborative project, funded by the European Framework IV research programme, attempts to integrate existing metamodels, methods, and tools through the formal definition of basic principles of metamodels (Jarke et al., 1994). Find more on RENOIR and NATURE at http://www-i5.informatik.rwth-aachen.de/PROJEKTE/NATURE/nature.html and http://panoramix.univ-paris1.fr/CRINFO/PROJETS/nature.html.

## Summary of Evaluation Frameworks and Metamodels

The representative list of frameworks, metamodels, and associated projects presented in this chapter are mostly used in the IS engineering field. Time and space would not allow for a more elaborate analysis or an exhaustive listing. However, the summarized information provided in Table 1 denotes important evaluative parameters for the metamodeling tools described. These observations are important for the discussion of the range and limitations of the quality properties of methods in the next section.

The information is summarized under the headings of hard and soft issues coverage, whether the frameworks and metamodels provide reengineering and reuse facilities through tool support, their year of introduction, and the degree of testing and computability that are offered as quality evaluation and construction criteria for ISDMs. Table 1 excludes specialized applications and the RENOIR and NATURE initiatives. Additionally, it emphasizes two important quality properties, namely, method testing and method computability — the ability of a method to include implementable/programmable characteristics for realizing IS design and testability criteria for facilitating subsequent testing procedures. Very few frameworks consider design, implementability, and maintenance issues in an integrated manner; they mostly analyze quality features of methods by examining the suitability, applicability, and practicality for particular application domains.

*Table 1. Information on method evaluation frameworks and metamodeling projects*

| | Hard | Soft | Tool Support | Year of Introduction | Testing | Computability |
|---|---|---|---|---|---|---|
| **Sociocybernetic** | Limited | Yes | No | 1983 | No | No |
| **Avison & Fitzgerald** | Yes | Yes | No | 1988 | No | No |
| **Case Shells** | Yes | Limited | Yes | 1990 | No | Limited |
| **MetaView** | Yes | No | Yes | 1992 | No | Limited |
| **DESMET** | Yes | Yes | Limited | 1994 | No | No |
| **Euromethod** | Yes | Limited | No | 1994 | No | No |
| **NIMSAD** | Yes | Yes | No | 1994 | No | No |
| **MetaPHOR** | Yes | Limited | Yes | 1991 | No | Limited |
| **Tudor & Tudor** | Yes | Limited | No | 1995 | No | No |
| **CDIF** | Yes | No | Yes | 1990 | Limited | Limited |
| **CDM-FILTERS** | Yes | Limited | Limited | 2001 | Yes | Yes |

## Perspectives on Known Frameworks and Metamodels

Instruments for measuring ISDM quality, metamodels, and evaluation frameworks should provide a standard quality platform for formally redefining static and dynamic method requirements and capturing the computational characteristics of both instantiated and generic methods. Such approaches should provide rigorous environments for reasoning and formally studying the properties of IS design artifacts, establishing whether such specification models facilitate IS implementations of improved quality.

However, the overall assessment of existing facilities indicates that many important quality features are either partially supported or not considered at all, and many method-oriented problems in software systems engineering remain undefined. For example, existing metamodeling tools and frameworks are not generic. Typically, coverage narrowly reflects the application domain and very restrictive types of systems and problems, the quality perceptions, or the philosophical and epistemological principles that the framework encompasses.

Consequently, even with good intentions, many of the primary objectives of software quality assurance (e.g., problem of testing during IS design) are still in the wish list of most ISD evaluation frameworks. Furthermore none of the metamodeling environments and notations handles metadata and metaprocess specification for considering the dynamic features of models (Berki, 2001; Koskinen, 2000) and for allowing metamodelers to maintain control of both process metamodeling and metadata of a method simultaneously, an important quality property.

The following are some of the other limitations of the existing facilities:

- Frameworks and metamodels are either based on qualitative analysis or use method construction (or ME) principles. The support for data integration is a very important and highly desirable feature; however, only some metamodels facilitate data, information, or knowledge integration. In the facilities that include them, they are typically referenced as quality criteria of evaluation frameworks and not projected as part of metamodeling rules.
- Computability (implementability) and maintainability of specifications and implementations are not yet considered to be fundamental quality issues of great importance and are not addressed adequately by existing method evaluation and ME frameworks and principles (Berki et al., 2001; Siakas et al., 1997).
- The capability to assist formal testing — considered a crucial quality assurance procedure to reassure IS stakeholders and certify the quality of the development process and product (Berki, 2003; Rossi & Brinkkemper, 1996) is absent from most of the existing frameworks and metamodels.
- Only a few metamodels (e.g., DESMET and NIMSAD) fully address both soft and hard issues of ISD.
- Method modifiability (changeability), an important quality feature, is not always addressed, although system and method dynamics are huge and controversial subjects in dynamic systems development. In fact, only NIMSAD and CDM-FILTERS directly take into account the dynamic nature of ISD.
- There is a scarcity of supporting automated tools for specific environments.

# Conclusions

Technological advancements have altered many quality requirements for ISD, necessitating a change in the roles of stakeholders and elevating the importance of methodologies, hence the need for increasing the effectiveness of approaches for evaluating methodologies that were designed to improve ISD quality by ensuring process consistency. There is a plethora of these ISDMs. Similarly, there are many evaluation frameworks and assessment methods for examining and comparing their capabilities and determining whether their quality-enhancing attributes are suitable for particular development environments. There are also many metamodels and support tools to construct new methods with desirable qualities, if none of the existing development methods are satisfactory.

From a practitioner perspective, the chapter examined and provided a commentary on metamodels and briefly reviewed the workings of some well-known evaluation frameworks and ME metamodels, highlighting quality considerations that are pertinent to the choice and evaluation of ISDMs and issues bearing on the construction, modeling, and remodeling of IS development methods and assessing their strengths and limitations. Many strengths were highlighted; however, several weaknesses were also identified, ranging from small issues such as the time needed to produce a model to larger problems of method integration and testing, and facilities to capture the significance of the human role. This review unearthed the urgent need for more integrative and formal metamodeling approaches, which, based on automated tool support, can facilitate the construction of formally computable and testable IS designs, a crucial quality requisite of development methods.

From a research point of view, evaluation frameworks, metamodeling, and ME can be considered communication and thinking tools for assimilating the philosophies and epistemologies that exist in the IS field and the different perceptions of quality they generate. Thus, they could provide additional metacognitive knowledge to help evaluate and model IS needs in a scientific and progressive manner. Further research efforts are therefore required to continue exploring the issues examined in this chapter and provide insights into effectiveness criteria for constructing metamodels and evaluating ISDMs.

The strengths and weaknesses of metamodeling environments should be examined more rigorously using appropriate architectural constructs that encapsulate both communicative and technical quality properties. In order to provide rigorous, scientific evidence to convince adopters that method A is better than method B in large design projects, and where the problems really are, results must be based on carefully controlled experiments of suitable complexity. This is especially so because there is little empirical evidence of what constitutes the quality of ISDMs and lack of clarity about what makes one method superior to another for large-scale ISD projects (Holcombe & Ipate, 1998).

In addition, the increasingly global nature of information systems development demands that IS and information products conform to new, extended quality standards for quality certification and quality assurance. In this global development environment, which utilizes virtual development teams and project groups separated by time and distance, ISDMs will become even more important in order to increase consistency and reduce variability in the process structuring methods utilized. This also amplifies the importance of ISDM evaluation through frameworks and metamodels. There is an obvious need for further research to explore this new dimension of quality evaluation of ISDMs in different anthropological and social contexts to identify controllable and uncontrollable factors for IS development within these contexts. Prescriptions for the improvement of ISDMs and IS must now incorporate considerations of organizational and national cultures and the applicability of methods and tools, and their relevance in those environments.

# References

Abran, A., Khelifi, A., & Suryn, W. (2003). Usability meanings and interpretations in ISO standards. *Software Quality Journal, 11*, 325-338.

Allen, P., & Frost, S. (1998). *Component-based development for enterprise systems. Applying the SELECT perspective.* Cambridge: Cambridge University Press.

Avison, D. E., & Fitzgerald, G. (1995). *Information systems development: Methodologies, techniques and tools.* Maidenhead; Berkshire: McGraw-Hill.

Beizer, B. (1995, June 12-15). Foundations of software testing techniques. *Proceedings of the 12th International Conference & Exposition on Testing Computer Software*. Washington, DC.

Bergsten, P., Bubenko, J. A., Jr., Dahl, R., Gustafsson, M., & Johansson, L. (1989). *RAMATIC: A CASE shell for implementation of specific CASE tools* (Tech. Rep. No. S-7). Swedish Institute for Systems Development (SISU), KTH Royal Institute of Technology, Stockholm, Sweden.

Berki, E. (1999). *Systems development method engineering, the formal NEWS: The formal notation for expressing a wide-range of specifications, the construction of a formal specification metamodel* (Mphil/PhD

Transfer Report). University of North London, Faculty of Science, Computing & Engineering, London.

Berki, E. (2001). *Establishing a scientific discipline for capturing the entropy of systems process models. CDM-FILTERS: A computational and dynamic metamodel as flexible and integrated language for the testing, expression and re-engineering of systems.* PhD thesis. University of North London, Faculty of Science, Computing & Engineering, London.

Berki, E. (2003, November 21-23). Formal metamodelling and agile method engineering in MetaCASE and CAME tool environments. Proceedings of the 1st South-East European Workshop on Formal Methods. In D. Dranidis, K. Tigka, & P. Kefalas (Ed.), *Agile formal methods: Practical, rigorous methods for a changing world* (Satellite of the 1st Balkan Conference in Informatics), SEERC, Thessaloniki (Proceedings published in 2004) (pp. 170-188).

Berki, E., & Georgiadou, E. (1998, November/December). A comparison of quantitative frameworks for information systems development methodologies. *Proceedings of the 12th International Conference of the Israel Society for Quality*, Jerusalem, Israel. Conference proceedings available on CD-ROM.

Berki, E., Georgiadou, E., & Holcombe, M. (2004, April). Requirements engineering and process modelling in software quality management — Towards a generic process metamodel. *The Software Quality Journal, 12*, 265-283.

Berki, E., Georgiadou, E., & Siakas, K. (1997, March). A methodology is as strong as the user involvement it supports. *Proceedings of the International Symposium of Software Engineering in Universities,* Rovaniemi Polytechnic, Rovaniemi (pp. 36-51).

Berki, E., Isomäki, H., & Jäkälä, M. (2003, June). Holistic communication modeling: Enhancing human-centred design through empowerment. In D. Harris, V. Duffy, M. Smith, & C. Stephanidis (Eds.), *Cognitive, social and ergonomic aspects* (Vol. 3, pp. 1208-1212). *Proceedings of the HCI International Conference*, Heraklion, Crete. Lawrence Erlbaum Associates.

Berki, E., Lyytinen, K., Georgiadou, E., Holcombe, M., & Yip, J. (2002, March). Testing, evolution and implementation issues in MetaCASE and Computer Assisted Method Engineering (CAME) environments. In G. King, M. Ross, G. Staples, & T. Twomey (Eds.), *Issues of quality management and process improvement. Proceedings of the 10th International Conference on Software Quality Management*, British Computer Society, SQM, Limerick.

Brinkkemper, S. (1996, April). Method engineering: Engineering of information systems development methods and tools. *Information and Software Technology, 38*(4), 275-280.

Brinkkemper, S. (2000, June). Method engineering with Web-enabled methods. In S. Brinkkemper, Lindencrorna, E., & Solvberg, A. (Eds.), *Information systems engineering, state of the art and research themes* (pp. 123-134). London: Springer-Verlag.

Burst, A., Wolff, M., Kuhl, M., & Muller-Glaser, K. D. (1999, June). *Using CDIF for concept-oriented rapid prototyping of electronic systems*. University of Karlsruhe, Institute of Information Processing Technology, ITIV. Retrieved December 1, 2005, from http://www-itiv.etec.uni-karlsruhe.de

CCTA. (1994). *Euromethod overview.* Author.

Checkland, P., & Scholes, J. (1990). *Soft systems methodology in action.* New York: John Wiley & Sons.

Chisholm, R. M. (1982). *The problem of criterion. The foundations of knowing*. Minneapolis: University of Minnesota Press.

Clifton, H. D., & Sutcliffe, A. G. (1994). *Business information systems* (5th ed.). Englewood Cliffs, NJ: Prentice Hall.

DESMET. (1991). *State of the art report (workpackage 1)*. NCC Publications.

Dorling, A. (1993). SPICE: Software process improvement and capacity determination. *Information and Software Technology*, 35(6/7), 404-406.

Georgiadou, E., Hy, T., & Berki, E. (1998, November/December). Automated qualitative and quantitative evaluation of software methods and tools. *Proceedings of the the 12th International Conference of the Israel Society for Quality*, Jerusalem, Israel.

Georgiadou, E., Mohamed, W.-E., & Sadler, C. (1994, November). Evaluating the evaluation methods: The data collection and storage system using the DESMET feature analysis. *Proceedings of the 10th International Conference of the Israel Society for Quality,* Jerusalem, Israel (pp. 467-474).

Georgiadou, E., & Sadler, C. (1995, May). Achieving quality improvement through understanding and evaluating information systems development methodologies. *Proceedings of the 3rd International Conference on Software Quality Management,* Seville, Spain (Vol. 2, pp. 35-46).

Georgiadou, E., Siakas, K., & Berki, E. (2003, December 10-12). Quality improvement through the identification of controllable and uncontrollable factors in software development. In R. Messnarz (Ed.), *Proceedings of*

*EuroSPI 2003: European Software Process Improvement*, Graz, Austria (pp. IX 31-45).

Graham, I. (1991). *Object-oriented methods.* Reading, MA: Addison-Wesley.

Harel, D., Pnueli, A., Schmidt, J.P., & Sherman, R. (1987). On the formal semantics of statecharts. *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science* (pp. 54-64).

Holcombe, M., & Ipate, F. (1998). *Correct systems — building a business process solution.* London: Springer-Verlag.

Huotari, J., & Kaipala, J. (1999). Review of HCI research — focus on cognitive aspects and used methods. In T. Kakola (Ed.), *IRIS 22 Conference: Enterprise Architectures for Virtual Organisations*. Keurusselka, Jyvaskyla: Jyvaskyla University Printing House.

Iivari, J., & Kerola, P. (1983, July 5-7). A sociocybernetic framework for the feature analysis of information systems design methodologies. In T. W. Olle, H. G. Sol, & C. J. Tully (Eds.), *Information systems design methodologies: A feature analysis. Proceedings of the IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies.* York.

Iivari, J., & Huisman, M. (2001, June). The relationship between organisational culture and the deployment of systems development methodologies. In K. R. Dittrich, A. Geppert, & M. Norrie (Eds.), *The 13th International Conference on Advanced Information Systems Engineering, CAiSE'01*, Interlaken (Lecture Notes in Computer Science [LNCS] 2068, pp. 234-250). Springer.

Ince, D. (1995). *Software quality assurance.* New York: McGraw-Hill.

Ipate, F., & Holcombe, M. (1998). Specification and testing using generalized machines: A presentation and a case study. *Software Testing, Verification and Reliability, 8*, 61-81.

Jarke, M., Gallersdorfer, R., Jeusfeld, M. A., Staudt, M., & Eherer, S. (1995). ConceptBase — a deductive objectbase for meta data management. *Journal of Intelligent Information Systems, 4*(2), 167-192.

Jarke, M., Pohl, K., Rolland, C., & Schmitt, J.-R. (1994). Experience-based method evaluation and improvement: A process modeling approach. In T. W. Olle & A. A. Verrijin-Stuart (Eds.), *IFIP WG8.1 Working Conference CRIS'94.* Amsterdam, The Netherlands: North-Holland.

Jarzabek, S. (1990, March). Specifying and generating multilanguage software development environments. *Software Engineering Journal, 5*(2), 125-137.

Jayaratna, N. (1994). *Understanding and evaluating methodologies. NIMSAD: A systemic approach.* Berkshire: McGraw-Hill.

Jenkins, T. (1994, Summer). Report back on the DMSG sponsored UK Euromethod forum '94. *Data Management Bulletin, 11*, 3.

Kaipala, J. (1997). Augmenting CASE tools with hypertext: Desired functionality and implementation issues. In A. Olive & J. A. Pastor (Eds.), *The 9th International Conference on Advanced Information Systems Engineering, CAiSE '97* (LNCS 1250, pp. 217-230). Berlin: Springer-Verlag.

Kelly, S. (1997). *Towards a comprehensive MetaCASE and CAME environment, conceptual, architectural, functional and usability advances in MetaEdit+.* PhD thesis, University of Jyvaskyla, Finland.

Kelly, S., Lyytinen, K., & Rossi, M. (1996, May 20-24). MetaEdit+: A fully configurable multi-user and multi-tool CASE and CAME environment. In P. Constantopoulos, J. Mylopoulos, & Y. Vassiliou (Eds.), *Advances in Information Systems Engineering, 8th International Conference CAiSE'96,* Heraklion, Crete, Greece (LNCS, pp. 1-21). Berlin: Springer-Verlag.

Kitchenham, B. A., Linkman, S. G., & Law, D. T. (1994, March). Critical review of quantitative assessment. *Software Engineering Journal, 9*(2), 43-53.

Kopetz, H. (1979). *Software reliability.* New York: Springer-Verlag.

Koskinen, M. (2000). *Process metamodeling: Conceptual foundations and application.* PhD thesis, University of Jyvaskyla, Jyvaskyla Studies in Computing-7, Finland.

Kronlof, K. (Ed.). (1993). *Method integration, concepts and case studies.* Wiley Series on Software Based Systems, Wiley, Chichester.

Law, D. (1988). *Methods for comparing methods: Techniques in software development.* NCC Publications.

Law, D., & Naeem, T. (1992, March). DESMET: Determining an evaluation methodology for software methods and tools. *Proceedings of the Conference on CASE — Current Practice, Future Prospects.* Cambridge: British Computer Society.

Lewis, H. R., & Papadimitriou, C. H. (1998). *Elements of the theory of computation.* Prentice Hall International Editions.

Manninen, A., & Berki, E. (2004, April). An evaluation framework for the utilisation of requirements management tools — maximising the quality of organisational communication and collaboration**.** In M. Ross & G. Staples (Eds.), *Software Quality Management 2004 Conference*, University of KENT, Canterbury.

Marttiin, P. (1988). *Customisable process modelling support and tools for design environment.* PhD thesis, University of Jyvaskyla, Jyvaskyla, Finland.

Marttiin, P., Rossi, M., Tahvanainen, V.-P. & Lyytinen, K.A. (1993). Comparative review of CASE shells — a preliminary framework and research outcomes. *Information and Management, 25*, 11-31.

Mohamed W. A., & Sadler, C. J. (1992, April). Methodology evaluation: A critical survey. *Eurometrics'92 Conference on Quantitative Evaluation of Software & Systems,* Brussels (pp. 101-112).

Musa J. D., Iannino, A., & Okumoto, K. (1987). *Software reliability measurement, prediction, application.* New York: McGraw-Hill.

Myers, G. (1976). *Software reliability principles and practices.* John Wiley & Sons.

Mylopoulos, J., Borgida, A., Jarke, M., & Koubarakis, M. (1990). TELOS: A language for representing knowledge about information systems. *ACM Transactions on Information Systems, 8*(4).

Nielsen, P. (1990). Approaches to appreciate information systems methodologies. *Scandinavian Journal of Information Systems, 9*(2), 43-60.

Olle, T. W., Sol, H. G., & Verrijin-Stuart, A. A. (Eds). (1982, May 10-14). Information systems design methodologies: A comparative review. *Proceedings of the IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies*, Noordwijkerhout, The Netherlands.

Paulk, M. C., Curtis, B., Chrissis, M. B., & Weber, C. V. (1993, July). The capability maturity model: version 1.1. *IEEE Software, 10*(4), 18-27.

Rantapuska, T., Siakas, K., Sadler, C. J., & Mohamed, W.-E. (1999, September). Quality issues of end-user application development. In C. Hawkins, E. Georgiadou, L. Perivolaropoulos, M. Ross, & G. Staples (Eds), *The BCS INSPIRE IV Conference: Training and Teaching for the Understanding of Software Quality*, University of Crete, Heraklion.

Rossi, M. (1998). *Advanced computer support for method engineering: Implementation of CAME environment in MetaEdit+.* PhD thesis, University of Jyvaskyla, Jyvaskyla Studies in Computer Science, Economics and Statistics, Finland.

Rossi, M., & Brinkkemper, S. (1996). Complexity metrics for systems development methods and techniques. *Information Systems, 21*(2), ISSN: 0306-4379.

Saadia, A. (1999). *An investigation into the formalization of software design schemata.* MPhil thesis, University of North London, Faculty of Science, Computing and Engineering.

Sadler, C., & Kitchenham, B. A. (1996). Evaluating software engineering methods and tools. Part 4: The influence of human factors. *SIGSOFT, Software Engineering Notes, 21*(5), 11-13.

Shapiro, J., & Berki, E. (1999). Encouraging effective use of CASE tools for discrete event modelling through problem-based learning. In C. Hawkins, E. Georgiadou, L. Perivolaropoulos, M. Ross, & G. Staples (Eds.), *The BCS INSPIRE IV Conference: Training and Teaching for the Understanding of Software Quality,* University of Crete, Heraklion (pp. 313-327). British Computer Society.

Siakas, K., Berki, E., & Georgiadou, E. (2003, December 10-12). CODE for SQM: A model for cultural and organisational diversity evaluation. In R. Messnarz (Ed.), *EuroSPI 2003:European Software Process Improvement*, Graz, Austria (pp. IX 1-11).

Siakas, K., Berki, E., Georgiadou, E., & Sadler, C. (1997). The complete alphabet of quality information systems: Conflicts & compromises. *Proceedings of the 7th World Congress on Total Quality Management.* New Delhi: McGraw-Hill.

Si-Said, S., Rolland, C., & Grosz, G. (1996, May 20-24). MENTOR: A computer aided requirements engineering environment. In P. Constantopoulos, J. Mylopoulos, & Y. Vassiliou (Eds.), *Advances in Information Systems Engineering, 8th International Conference*. Heraklion, Crete, Greece, (LNCS 1080, pp. 22-43). Berlin: Springer-Verlag.

Smolander, K., Tahvanainen, V.-P., & Lyytinen, K. (1990). How to combine tools and methods in practice — a field study. In B. Steinholz, A. Solverg, & L. Bergman (Eds.), *The 2nd Conference in Advanced Information Systems Engineering* (LNCS 436, pp. 195-214). Berlin: Springer-Verlag.

Sol, H. G. (1983, July 5-7). A feature analysis of information systems design methodologies: Methodological considerations. In T. W. Olle, H. G. Sol, & C. J. Tully (Eds.), *Information systems design methodologies: A feature analysis. Proceedings of the IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies*. York.

Sorenson, P. G., Tremblay, J.-P., & McAllister, A. J. (1988). The MetaView system for many specification environments. *IEEE Software, 30*(3), 30-38.

Tolvanen, J.-P. (1998). *Incremental method engineering with modeling tools.* PhD thesis, University of Jyvaskyla, Jyvaskyla, Finland.

Tudor, D. J., & Tudor, I. J. (1995). *Systems analysis and design — a comparison of structured methods.* NCC Blackwell.

Van Hillegersberg, J. (1997). *Metamodelling-based integration of object-oriented systems development.* Amsterdam, The Netherlands: Thesis Publishers.

Vassiliou, G. (2000). Developing data warehouses with quality in mind. In S. Brinkkemper, E. Lindencrorna, & A. Solvberg (Eds.), *Information systems engineering, state of the art and research themes.* London: Springer-Verlag.

Whittaker, J. (2000, January/February). What is software testing? And why is it so hard? *IEEE Software, 17*(1), 70-79.

Wood, D. (1987). *Theory of computation.* New York: John Wiley & Sons.

Wood-Harper, A. T., Antill, L., & Avison, D. E. (1985). *Information systems definition: The multiview approach.* Oxford: Blackwell Scientific Publications.

Zhang, Z. (2004). *Model component reuse. Conceptual foundations and application in the metamodeling-based system analysis and design environment.* PhD thesis, University of Jyvaskyla, Jyvaskyla Studies in Computing, Finland.

# Section V:
# IS Quality Issues in
# Under-Researched Areas

## Chapter XIII

# Software Quality and the Open Source Process

Sameer Verma, San Francisco State University, USA

## Abstract

*This chapter introduces the open source software development process from a software quality perspective. It uses the attributes of software quality in a formal model and attempts to map them onto the principles of the open source process. Many stages of the open source process appear to have an ad-hoc approach. Although open source is not considered to be a formal methodology for software development, it has resulted in the development of very high quality software, both in the consumer and in the enterprise space. In this chapter, we hope to understand the open source process itself, and apply it to other methodologies in order to achieve better software quality. Additionally, this chapter will help in understanding the "Wild West" nature of open source and what it may hold for us in the future.*

# Introduction

The concept of quality is an abstract one. While quantity can be specified in standard terms such as a gram or a gallon, quality is usually a relative term. It is relative to the assessment of the product or process, as perceived by an individual or an organization (Barbacci et al., 1995). Quality is sometimes defined as compliance *to* a standard (Perry, 1991). It seems that the implications of quality vary from one task to another, and so does its assessment. However, generally speaking, we can agree on the direction it takes when the quality of an entity improves or degrades. In the case of software, the quality of software can be assessed by its characteristics. Several models exist that either measure software quality using a quantitative surrogate, or in terms of attribute sets. In this chapter, we use a model (Bass et al., 2000), where the quality of software is assessed by its attributes. These attributes include performance, security, modifiability, reliability, and usability of a particular system. We will explore these attributes, the challenges they pose to the open source development process and how the open source community measures up to these challenges.

# Background

The importance of information systems and technology is well-established in our society, both in personal and professional space. It is difficult to imagine a society without access to information. Software forms an important cog of that system. While hardware such as desktops, laptops, and PDAs provide a platform for implementing computing power, software is the flexible component that is responsible for expression of innovation, creativity, and productivity (Lessig, 2005). Ranging from word-processing to number-crunching, the quality of software influences and impacts our lives in many different ways. It is no surprise that software quality has become the subject of many studies (Halloran & Scherlis, 2002).

## Software Quality

While the quantitative side of software can be measured and optimized in terms of its capability to solve mathematical formulations and render graphics (Wong & Gokhale, 2005), it is the qualitative perception that matters to the end user and the organization (Bass et al., 2000). Quality is also harder to assess, so it is often ignored or replaced by quantitative measurements such as lines of code.

Obviously, more lines of code do not always imply better quality (Samoladas et al., 2004).

In this chapter, we will use a model that uses five attributes, namely, performance, security, modifiability, reliability, and usability, to assess the quality of software. Performance involves adjusting and allocating resources to meet system timing requirements (Bass et al., 2000). Security can be described as freedom from danger, that is, safety. It may also be viewed as protection of system data against unauthorized disclosure, modification, or destruction (Barbacci et al., 1995). Modifiability is the ability of a system to be changed after it has been deployed. Requests can reflect changes in functions, platform, or operating environment (Bass et al., 2000). Reliability of a system is the measure of the system's ability to keep operating over time (Barbacci et al., 1995). Usability is a basic separation of command from data. It relies on explicit models for task, user, and system (Bass et al., 2001). These attributes put together can assist in assessing the quality of a software project. Of course, these attributes apply to all kinds of software, whether open source or proprietary.

## Open Source Process

Next, let us examine the open source process. The purpose of looking at open source as a process, as opposed to a product, is that the open source ideology permeates well beyond software production (DiBona et al., 1999). The proponents of open source believe in a philosophy of open source more so than simply the software. They look upon software as an enabler of change in a society that increasingly relies on information technology for its survival and growth (Lessig, 2005). The following sections are by no means exhaustive, but should provide a substantial background to understanding the tenets of quality in the open source context.

Open source software (OSS) is largely developed by freelance programmers, who create freely distributed source code by collaborating and communicating over the Internet (Moody, 2001; Raymond, 1999; Sharma et al., 2002). A small, but significant proportion of the software is also developed in software companies such as IBM, Sun Microsystems, and Intel. Open source software is generally fashioned in what is often classified as the *bazaar* style as opposed to the *cathedral* style, which is more commonly observed in proprietary software development. In his collection of essays, Raymond (1999) postulates that the proprietary style of software development follows the cathedral model, complete with plans, schedules, resources, and deliverables. In contrast, the bazaar model lacks an explicit blueprint. Work begins with an idea, followed by a series of short release cycles of software, which is prototypical at best. Over time this software development process gains momentum, in some cases, partly due to the availability of source code and partly due to common interest in the application of the software (Feller & Fitzgerald, 2002).

This development process is akin to shopping at a bazaar-like marketplace and purchasing ingredients as needed. Just as one would shop for olive oil, basil, pine nuts, lemons, parmesan cheese, and garlic to make pesto (Technorati, 2005), one could look for an operating system, a Web server, database server, and a programming language to build a Web application (Associates, 2005). The fact that this style of software development actually works comes as a pleasant surprise to many who are accustomed to working with traditional methods of software development[1]. A more subtle message from Raymond's essay is that while the *cathedral* approach may result in a grand and impressive application, it becomes somewhat outdated by the time it is completed (Raymond & Trader, 1999). The application may be complete, but the nature of the original problem may have changed significantly over time.

Failure to maintain a time schedule and contain the scope are often cited as reasons for the failure of information systems (Applegate et al., 2002). Open source's course-correction approach makes minor changes frequently to continually address the changing nature of the problem at hand. In some ways, the *bazaar* approach might alleviate the ills of the traditional software development process (Feller & Fitzgerald, 2000).

## Proprietary vs. Open Source Software

Proprietary software, including the source code used to create it, is often protected by a patent (Perens, 2005). The patent, not to be confused with copyright, is not free of distribution restrictions, and is rarely free of cost. Proprietary software is also defined as the software whose source code is kept secret and belongs to a specific individual or a company (Barahona et al., 1999). In the case of proprietary software, the source code is not distributed. The software is only made available in the form of compiled, object files to end users (Feller & Fitzgerald, 2002). The ownership of the software belongs to a specific company, group, or individual. Software cannot be redistributed, modified, or sold without the explicit permission of the owner. This "permission" is usually specified in the end user licensing agreement, also called EULA.

Enterprises use similar licenses, except that enterprise licenses apply to all end users in an enterprise, and even if the source code is given to the client company for maintenance purposes, the client is not allowed to make it available to third parties, let alone the entire world (Goth, 2001). These descriptions are largely applicable to proprietary software, although many variations do occur and must be treated on a case-by-case basis (Rosen, 2005). The owner entity (usually a software company) decides on the release cycles of the proprietary software, which in most cases is based on market pressures and upgrade revenue (Applegate et al., 2002). Closely guarded source code becomes the basis of this revenue stream.

The development framework for open source leverages communication facilities of the Internet. Independent software developers examine existing source code and make modifications to fulfill immediate needs in their own environment. In the case of open source software, the licensing allows distribution in source code as well as object form. When a product is not distributed with its source code, there is usually a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost, preferably downloadable via the Internet.

## *Proprietary Licensing*

Companies that practice the traditional software approach typically use proprietary software licenses. The EULA[2] assures that any software given to a customer cannot be copied, redistributed, or modified without the explicit permission of the company. Since commercial software products are in the form of object and not source code, software products can be licensed per-user, per-machine, per-processor, or for an entire organization under closed-source software licensing. A closed-source license may require a royalty fee, and in some versions it has an expiration date by which the customer should purchase the software again or upgrade to a new version of the product. Proprietary licensing is designed to generate a continuous revenue stream, while protecting the source code, and thereby controlling its use in the market.

## *Open Source Licensing*

In contrast, the nature of the open source software is that no right-to-use license fee or right-to-redistribute license fee is solicited from the users. No royalty is charged for businesses or individuals if they sell the source code of any open source software. However, open source must not be confused with public domain (Rosen, 2005). When the developer releases the source code to the public under the terms of an open source license, he or she does not relinquish the copyright. Open source licenses grant conditional rights to members of the general public willing to comply with the terms of the open source license. Developers can still sell private licenses to individuals and companies that are unwilling or unable to comply with the terms of the open source license (Rosen, 2005).

# The Open Source Mechanism

The open source community is a collection of individuals who may have any level of interest in open source software. These interests may lead them to play different roles. The involvement varies from being providers of ideas and code to simply users of the code (Dotzler, 2003). Open source communities pattern their operations off virtual organizations. In most cases, virtual organizations refer to new organizational forms that rely on strategic alliances and external partners to collaborate to achieve business goals or to serve customer needs (Davidow & Malone, 1992; Grenier & Metes, 1995; Lucas, 1996).

In order to achieve “spatial and temporal independence” (Robey et al., 1998), information and communication technologies tend to be heavily utilized to facilitate the coordination across time and space boundaries (Mowshowitz, 1997). Gallivan (2001, p. 281) defines virtual organizations as “goal directed and consisting of geographically distributed agents who may or may not ever meet face to face”, which is very apt in the case of open source communities. Additionally, the *gift culture* within the open source community features giving away source code (Bergquist & Ljungberg, 2001), voluntarily testing and debugging the software (Dotzler, 2003), and supporting fellow users by promptly answering their posted questions through mailing lists and Internet Relay Chat sessions (Raymond & Trader, 1999).

## Producer of Software

Open source software development is often initiated by individuals or small cliques of people for their own limited purpose (Mui et al., 2005). A phrase often used to describe this approach is “Every good work of software starts by scratching a developer’s personal itch” (Raymond, 1999, p. 23). The *itch* metaphor is indicative of a small, yet focused need that an individual developer may have, as opposed to a grand plan for software development. If the software that results from this process (*scratch*) proves to be useful, then others join in and contribute. The producer community is usually made up of individuals who are bound together by an open source project and little else. Often, these “producers” never meet in person. It is a modern marvel that entire pieces of software projects are negotiated, developed, managed, and disseminated completely online (Dotzler, 2003).

Based on the control structure of most open source software projects, a handful of individuals are given write-access to the software repository. These are the producers of software, also called maintainers in OSS parlance. Projects in the open source world adopt the walled server approach (Halloran & Scherlis, 2002). The server that hosts the software repository is visible to the entire world. Anybody can read from it. However, only the maintainers hold the right to *implement* recommended changes. Information flows freely from the producers, but not all feature requests or bug fixes are necessarily implemented into the project itself.

This approach seems to work well for controlling software development practices of the project. It does not limit the behavior of developers in their own environment. It does limit changes that the maintainers can make on the project itself. The walled server idea embodies critical aspects of software best practices such as process, management, design, and architecture *as defined by* the project leaders. This enables effective engineering management despite the fact that the engineers are self-selected and largely self-managed. It also allows for a more rigorous approach to quality control and assurance.

## Consumer of Software

Oddly enough, the primary consumer base of open source software is the open source community itself. Thus, one way to classify open source development is to study it as an innovation (Verma & Jin, 2004). Considering the traditional methods of software development, there is plenty of evidence that the open source methodology is indeed an innovation. Taking this idea one step further, we can look at this development as a process that happens where the consumers of software are from the same group as the producers. In most production and consumption-based economies, the producer group is quite different from the consumer group.

This holds true in the world of traditional software as well. The stack of communities is vertically connected from producer to distributor to consumer. Consumers of software are rarely skilled in the art and science of software production, hence the segmented innovation approach, where the innovators are different from the end users. However, the open source community is different. A large proportion of the open source community falls in an intersection of producers and consumers of the software (Feller, 2001). This is inherent in the open source process. Users are treated as co-developers. Both parties subscribe to the same community. These shared innovation networks have a great advantage over the segmented innovation systems. Furthermore, individual users do not have to develop everything they need on their own; they can benefit from innovations developed by others and freely shared within and beyond the user network.

## Peer Review Process

The software development process is community-driven, where the community size may vary from a small group of 2, to a large group of over 100 developers. An interesting departure from other traditional approaches is that OSS users are treated as co-developers or peers (Dotzler, 2003). Since the source code is freely available, a community of peers is able to partake in a peer review process somewhat similar to the academic peer review process conducted to publish research (Raymond & Trader, 1999). The role of members of the community varies based on their skill sets, interest, and availability of resources. Different roles can be approximately categorized into four groups, including project owners/core developers, patch submitters, source code testers, and end users (Verma & Jin, 2004). Project owners/core developers are a small group of people who contribute most of the code and control the software releases.

Let us look at two examples to comprehend the proportion of members across different functional roles. The first example is the Apache project, which is an open source project for developing a Web server. It began as a series of patches or fixes to the original Web server developed at National Center for Supercomputing Applications at University of Illinois, Urbana-Champaign (Audris, 2003). Eventually, the patches became so numerous that it was suggested the server be named a "patchy" server; hence the name Apache (Dotzler, 2003). The core developers of Apache account for over 80% of the coding. Patch submitters involve a relatively wider development community who examine the source code in detail and submit bug fixes (Mockus et al., 2000).

Source code testers are comprised of an even larger group who download and compile the source code and report the bugs. Lastly, end users, who may constitute the largest group of all, are only interested in using precompiled object code. End users may also report problems, but the problems are based on precompiled software. Another example is the Mozilla project, which came out of the original Netscape Web browser code (Dotzler, 2003). Mozilla developed into a significantly different browser since its inception in 1998. Mozilla has about 25 core developers, over 400 patch submitters, over 10,000 quality assurance testing contributors, and around over 500,000 end users involved (Dotzler, 2003).

## Debugging and Feedback

Bug reporting methods adopted by the open source community are designed for public view. All feedback is made public, thereby allowing someone on the outside to review the code and submit a change, usually called a fix or a patch.

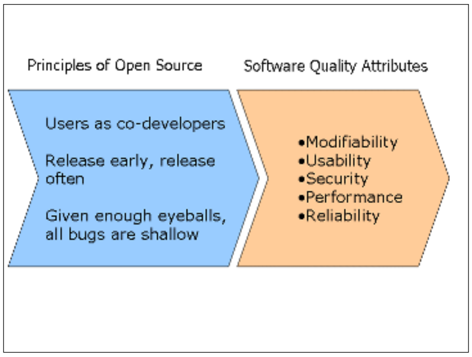
Such patches are usually maintained in parallel with existing software, so that users may pick and choose patches that are necessary for a particular scenario. The walled server approach (Halloran & Scherlis, 2002) mentioned before, provides an asymmetric channel for flow of bug-related information. The number of inputs from the community outnumbers the number of fixes implemented by the core group. Patches are usually written by one of the core members, or by patch submitters. Only core developers are allowed to make the decision to include a patch into the main software code base. Irrespective of the decision to include the patch into the main code base, the patch is always made available as an option. If a project requires some esoteric feature set provided by a patch, the developers can choose to include it at the time of compilation.

## Evolutionary Survival

When the demands for features and bug fixes from the community go unanswered, the project may split or fork into a parallel development stream. Most open source licensing allows for a project fork, which is a split in the code development process into a new branch. A fork allows for independent and parallel development following two different approaches. It is theorized that the most useful branch will thrive while the less useful one will wither away (Torvalds & Diamond, 2001). This is similar to the process of genetic evolution where dominant traits get adopted and recessive ones diminish over generations.

A popular example is PHPNuke, a portal framework written in PHP, with a database backend to store content. PHPNuke started as a creation of one person to fulfill the need for a free and open source content management system. PHPNuke got popular very quickly. The application soon grew to a size that was beyond one person's ability to manage. The maintainer accepted help from other programmers but did not give them credit where it was due. He also failed to fix bugs they repeatedly pointed out, even when they submitted bug fixes. A number of these programmers became annoyed and disappointed with the general unresponsiveness of the project owner and decided to fork PHPNuke (Krubner, 2003). This version came to be known as PostNuke. With its own community, PostNuke implemented all the patches that were previously ignored. As PostNuke became more feature rich and less buggy, PHPNuke picked up momentum in order to compete with its forked creation. While forking does create a division of resources, it also creates a healthy competitive environment. Eventually, which version of the fork will dominate (i.e., used by the majority) will depend on the adaptability of the software to its environment[3].

*Figure 1. Mapping of open source principles and quality attributes*


Principles of Open Source
Software Quality Attributes
Users as co-developers
Release early, release often
Given enough eyeballs, all bugs are shallow
•Modifiability
•Usability
•Security
•Performance
•Reliability

# Quality and the Open Source Process

To reiterate our approach to examining open source via the lens of quality attributes, we propose five key attributes: modifiability, usability, security, performance, and reliability. We would like to map these attributes to the primary principles of open source (Figure 1). Three most widely sited principles of OSS development include:

- Treating users as co-developers.
- Release early and release often.
- Given enough eyeballs, all bugs are shallow.

These three principles explain the essence of open source development model. Other observations are comparatively minor in their impact on the open source process as a whole.

## Users As Co-Developers

The first principle of *treating users as co-developers* stems from the fact that many members of the open source community are also involved in the development process. They may be programmers or simply bug testers. Either way, the users are treated as an integral part of the development process. This approach makes it easier to communicate with the community on a deeper level of software development. For example, instead of telling users to wait for a month or two, a project member may simply ask members to download untested code from the software repository. Repeated testing and use by co-developers provides valuable feedback that plays an essential role in the quality assurance process. Often, co-developers test software on a variety of software and hardware combinations. For example, programs written for the 386 class of Intel processors running a form of Linux operating system will typically run without any problems on 486 and Pentium class processors as well. Similarly, software written for the 32-bit version of Windows (Windows 95) will typically run on Windows 98, Windows 2000, and Windows XP.

Each co-developer would often have different setup of software and hardware. The combinations of hardware and software as a testing platform make for a very rich testing process. Working with co-developers has other advantages. Every time a significant problem is solved, a patch is applied to the existing code which is re-released every night in versions dubbed as nightly releases (Dotzler, 2003). Each nightly release is based on the very latest improvements and is targeted toward the co-developer community. Each nightly build is tested and goes through a rigorous testing process conducted by the co-developers.

By treating users as co-developers, the open source community strives to create a better performing piece of software (fewer bugs) with higher levels of usability. A good example of improved performance is the Linux kernel. After its 624 iterative incarnations (Kendrick, 2005), the version 2.6.10 of Linux has come to support many form factors ranging from cell phones to mainframes (Pranevich, 2003). The performance of Linux is equally good in either of these environments. The availability of source code and feedback from co-developers has made it possible to get performance levels on par with industry giants from the proprietary world.

Similarly, the Mozilla Firefox project is an example of improved usability. The Mozilla project was started in 1998 as a spin-off of the Netscape Communicator product from Netscape Communications (Dotzler, 2003). The idea was to open source the code-base in order to foster innovation. Over the last seven years, the Mozilla project has evolved from a sluggish, bloated piece of software to a much more usable product in the stand-alone version of the browser called Firefox[4]. The usability of Firefox is considered so much better than the leading browser (Internet Explorer) from Microsoft that Firefox has been downloaded over 78 million times since its original release in 2004.

The OSS community follows an incremental model of quality and payoff. With each incremental change comes a change in quality. After all, a large proportion of the feedback comes from the user community. This high degree of involvement relies on a legal mechanism of the *right* to modify any piece of software. It is only under the auspices of an open source license, that software creators provide their source code and hence a large share of their intellectual property rights. This sharing is done for a greater good of a much better expected payoff in the long run.

## Release Early, Release Often

Open source licenses are primarily designed to encourage modification and redistribution of software in source code form. The process of releasing several versions of the software in rapid cycles is aptly captured in the second principle: *release early and release often*. For example, over its life cycle of almost 14 years, it is estimated that the Linux kernel has been released 624 times (Kendrick, 2005). Each release attempts to improve upon the previous one. In some projects, versions are released every week during rapid testing and bug fixing. The general thinking is that instead of providing the end user with a feature complete version of the software, the creators release very rudimentary versions to get the discussions going. Small changes are incorporated quickly and released often to the public.

Another such example can be seen with the NoCatAuth project (Ishii et al., 2002), a very visible project that provides open source routing for wireless networks. In its initial incarnation, the software was written in Perl. The first few versions did not even work. However, it presented the idea to a group of people who then worked on improving it. After over 25 revisions, the software was stable enough to run for months on end without a reboot. NoCatAuth gained this stability through rapid release cycles with simple bug fixes implemented in each cycle.

Most open source projects use version numbers that mirror the policy of rapid releases. For example, in case of the Linux kernel, the version numbers take the form of *x.y.z* where *x* begins at 0, and *y* and *z* are nested subcategories (Torvalds, 2005). The first release of Linux was dubbed version 0.01 and has reached version 2.6.10 today[5].

Such incremental versioning supports incremental changes and timely assimilation of feedback through the quality assurance process. Features are therefore implemented and bugs are fixed in a more organized fashion.

Another outcome of rapid release cycles is that over time, the proportion of critical system errors decreases. This observation has been tested using randomly-generated commands (called the Fuzz approach) on different operating systems, including Microsoft Windows Family, UNIX, and Linux (Miller et al., 2000). The outcome is interesting. It shows that over time, the reliability of Linux-based operating systems has improved tremendously. Better stability and increased time between crashes improves stability and reliability of the system as a whole.

The most visible downside of the rapid release process is that it does not allow for long-term thinking in some cases. For example, the Jabber project was initially created to address the problem of instant messaging (Adams, 2001).

However, as the project grew, the developer community realized that the project could be extended to much more than just instant messaging. In its next major release beyond the 1.4 series, the Jabber Software Foundation decided to release a version 2.0, a complete rewrite of the design and its code (XMPP.org, 2005). Both series 1.4 and 2.0 co-exist and are actively used based on feature preferences.

## Given Enough Eyeballs...

Feedback is a very important cornerstone of the OSS process. Since the source code is made available for peer review, skilled programmers not only find problems but also fix them on their own. Even the non-programmer community can contribute by looking for the odd behavior in performance and user interface design. This is the essence of the third principle: *given enough eyeballs, all bugs are shallow*, also called *Linus' Law* after Linus Torvalds, the founder of Linux (Raymond, 1999).

Easy access to source code facilitates rigorous peer review and parallel debugging among many geographically dispersed programmers, therefore enabling rapid evolution of *high quality* software (Koch & Schneider, 2002; Stamelos et al., 2002). In addition, the mass of software users are directly and actively involved in the development process. Their bug reports tend to be very detailed because they generally cover problems that originated in different kinds of systems with various hardware/software configurations. Their suggestions and feedback regarding features and solutions of the software also promote more functional and user-friendly design.

This approach also goes against the grain of the *security through obscurity* concept that is often leveraged by proprietary software companies (Hissam et al., 2002). This is a controversial principle in security since it attempts to use secrecy to ensure security (Wikipedia, 2005). The essence of this concept is that since proprietary software does not make source code available, the risk of discovering security vulnerabilities is minimized. However, this approach is also akin to the practice of hiding the key under the doormat, so that it remains hidden. It is almost universally known that the first place to look for keys is under the doormat. Once the vulnerability is discovered, all is lost, until the vendor chooses to fix the problem. With open source projects, vulnerabilities get fixed either by the vendor, or by someone in the community. It appears that the open source process fosters better notions of security from the code perspective.

## Comparison of Software Methodologies

Many approaches within the open source community are similar to the ones followed within proprietary processes. It is important to understand that the programming language or framework that is used in either case is not significantly different. Programs written as open source or closed are essentially only instructions for a computer. What differs is its visibility and management. Closed source projects have programmers and reviewers, but these members are most likely paid employees of the same organization, where the goal is to maximize profit. OSS communities are not driven by monetary incentives. They are largely driven by ego, trust (Mui et al., 2005), and a desire to produce free code.

Open source software development has formalized in the last few years, but most of the work is done based on the direction from a very small core group as in the *benevolent dictator* model (Hamm, 2004) or by a group of developers who take on roles based on their quality of work and intentions as in the *meritocracy* model (Perens, 1997; Young, 1958). Linux is the primary example of a project where the central authority is Linus Torvalds, who relies on the consensus of a few other core contributors. Linus is therefore considered to be the benevolent dictator. On the other hand, a project like Debian, which is a community-driven distribution of Linux, chooses a team to lead the project every few years. The individuals elected to these positions are chosen based on their merit; hence the term meritocracy. In either case, the projects start with a vested interest at solving a single problem (the itch metaphor), and then evolves into something larger, if found useful.

Similar methods exist that rely on quick cycles of feedback and very rapid development schedules. Agile methods are primarily fostered by groups that are

*Table 1. Comparing software methodologies (Adapted from Abrahamsson et al., 2002)*

| **Entity** | **Agile Methods** | **Open Source-based (bazaar style)** | **Plan-driven (cathedral style)** |
|---|---|---|---|
| Producers | Rapid, Localized, Collaborative | Geographically distributed, Collaborative, Rapid | Adequate skill-set, plan-oriented, usually localized |
| Consumers | Dedicated, Knowledgeable, Collaborative, Empowered | Dedicated, Knowledgeable, Collaborative, Empowered | Representative, plan-driven |
| Design approach | Based on current environment | Based on current environment, but open for change | Based on current and foreseeable environments |
| Team management | Smaller team dynamics, Face-to-face | Dispersed teams, loose team dynamics | Larger teams |
| Goal | Rapid Value | Challenging problem (itch metaphor) | High assurance |
| Time scale | Short | Short | Long |

customer-oriented, and prefer to work in face-to-face environments that are usually well-funded (Alliance, 2001). Extreme programming is perhaps the most well-known of all agile methods. Table 1 presents a side-by-side comparison of open source methods, agile methods, and the classic plan-driven methods (Abrahamsson et al., 2002).

## Future Trends

In the early days of open source, it was strongly believed that the entire open source movement was being shouldered by altruistic programmers who burned the midnight oil to keep their ideals intact, an ideal of software free from bugs and free from the clutches of the enterprise (Himanen, 2001). Those stereotypes are changing rapidly (Dahlander & Magnusson, 2005). It is no longer a fly-by-night operation. Large organizations such as IBM, Oracle, General Motors, and the U.S. Department of Defense are playing a major role in the creation and maintenance of open source software (MITRE, 2002). Companies such as Sun Microsystems, known for their proprietary software, are testing the waters with different business models. Sun Microsystems recently announced the OpenSolaris project (Sun Microsystems, 2005), which aims at releasing the Solaris version 10 operating system under an open source license. These are big steps for the software industry, and it looks like this trend is moving up.

Other companies are getting into the business of providing a test platform (often called a stack) where they test a combination of software for enterprise level performance and security. The primary complaint with open source software is its lack of a guarantee. The software is provided as-is. Companies such as SpikeSource test combinations of open source Web servers, databases, and scripting environments for performance and security. Such approaches bolster confidence by providing guarantees that enterprises need. In such cases, open source is no longer viewed as unwarranted code written by lone programmers.

Enterprise involvement brings with it the question of legality. Given that code is written by so many different people, it is very difficult to avoid the conflict with an existing patent. In recent cases, SCO has taken IBM and DaimlerChrysler to court over intellectual property claims (Groklaw, 2005). Given that the legal standing of open source software is still largely untested in court, some companies shy away from using open source.

# Conclusion

While the open source approach is more along the lines of the untamed "Wild West", it presents a compelling alternative to developing high quality software. Projects such as Apache, Mozilla, and Linux have produced equal if not better quality software when compared with their proprietary counterparts. Apache commands over 65% of the world's Web server market share (Netcraft, 2005), while Mozilla Firefox and Linux have demonstrated very rapid growth in their own domains. In spite of the risks of a community-driven project, open source presents a favorable picture from a quality perspective. Over time, open source software improves in usability, performance, security, and reliability. Additionally, it is the *modifiability* of open source that stands out as the prime attribute. It would be well worth the effort to take a closer look at the open source world and attempt to accommodate its idiosyncrasies into more well-established, formalized methods to gain a "best of both worlds" advantage.

# References

Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2002). *Agile software development methods: Review and analysis*. Retrieved December 1, 2005, from http://www.inf.vtt.fi/pdf/publications/2002/P478.pdf

Adams, D. (2001). *Programming jabber*. Sebastopol, CA: O'Reilly & Associates.

Alliance, A. (2001). *Principles behind the Agile Manifesto.* Retrieved December 1, 2005, from http://www.agilemanifesto.org/principles.html

Applegate, L. M., Austin, R. D., & McFarlan, W. F. (2002). *Creating business advantage in the information age*. New York: McGraw-Hill Irwin.

Associates, O. R. (2005). *OnLAMP.com*. Retrieved December 1, 2005, from http://www.onlamp.com/

Audris, M. (2003). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology, 11*(3), 309-346.

Barahona, J. M. G., Quiros, P. d. l. H., & Bollinger, T. (1999). A brief history of free software and open source. *IEEE Software, 16*(1), 32-34.

Barbacci, M., Klein, M. H., Longstaff, T. A., & Weinstock, C. B. (1995). *Quality attributes*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

Bass, L., Klein, M., & Bachmann, F. (2000). *Quality attribute design primitives*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

Bass, L., Klein, M., & Bachmann, F. (2001, October 4). Quality attribute design primitives and the attribute driven design method. *Proceedings of the 4th Conference on Product Family Engineering*, Bilbao, Spain.

Bergquist, M., & Ljungberg, J. (2001). The power of gifts: Organising social relationships in open source communities. *Information Systems Journal, 11*(4), 305-320.

Dahlander, L., & Magnusson, M. G. (2005). Relationships between open source software companies and communities: Observations from Nordic firms. *Research Policy, 34*(4), 481.

Davidow, W. H., & Malone, M. S. (1992). *The virtual corporation*. New York: Harper Collins.

DiBona, C., Ockman, S., & Stone, M. (Eds.). (1999). *Open sources: Voices for the open source revolution*. Sebastapol, CA: O'Reilly.

Dotzler, A. (2003). Getting involved with Mozilla: The people, the tools, the process. In L. U. G. O. Davis (Ed.), *[electronic] Presentation at the Linux User Group of Davis.* Davis, CA. Mozilla.org.

Feller, J. (2001). Thoughts on studying open source software communities. In N. L. Russo, B. Fitzgerald, & J. I. DeGross (Eds.), *Realigning research and practice in information systems development: The social and organizational perspective* (pp. 379-388). Boise, ID: Kluwer.

Feller, J., & Fitzgerald, B. (2000). A framework analysis of the open source software development paradigm. *Proceedings of the the 21st International Conference in Information Systems (ICIS 2000)*, Brisbane, Queensland, Australia (pp. 58-69).

Feller, J., & Fitzgerald, B. (2002). *Understanding open source software development*. London: Addison-Wesley.

Gallivan, M. J. (2001). Striking a balance between trust and control in a virtual organization: A content analysis of open source software case studies. *Information Systems Journal, 11*(4), 277-304.

Goth, G. (2001). The open market woos open source. *IEEE Software, 18*(2), 104-107.

Grenier, R., & Metes, G. (1995). *Going virtual: Moving your organization into the 21st century*. Upper Saddle River, NJ: Prentice Hall.

Groklaw. (2005). *SCO vs. IBM Case: 2:03cv00294*. Retrieved December 1, 2005, from http://www.groklaw.net/staticpages/index.php?page=legal-docs#scovibm

Halloran, T. J., & Scherlis, W. S. (2002, May). High quality and open source software practices. *Proceedings of the International Conference on Software Engineering*, Orlando, FL.

Hamm, S. (2004). *Linus Torvalds' benevolent dictatorship*. Retrieved December 1, 2005, from http://www.businessweek.com/print/technology/content/aug2004/tc20040818_1593.htm

Himanen, P. (2001). *The hacker ethic and the spirit of the information age*. London: Secker & Warburg.

Hissam, S. A., Plakosh, D., & Weinstock, C. (2002). Trust and vulnerability in open source software. *IEE Proceedings — Software, 149*(1), 47-51.

Ishii, H., Chang, P., & Verma, S. (2002). *Implementing secure services over a wireless network* (Tech. Rep. No. BICS 699). San Francisco State University.

Kendrick, B. (2005). *Linux releases as of 2005.02.08*. Retrieved December 1, 2005, from http://www.sonic.net/~nbs/linux-releases.txt

Koch, S., & Schneider, G. (2002). Effort, cooperation and coordination in an open source software project: GNOME. *Information Systems Journal, 12*(1), 27-42.

Krubner, L. (2003). *Editorial history on PHP-Nuke and Post-Nuke*. Retrieved December 1, 2005, from http://www.nukecops.com/article65.html

Lessig, L. (2005). *Free culture: The nature and future of creativity*. New York: The Penguin Press.

Lucas, H. C. (1996). *The T-Form organization*. San Francisco: Jossey-Bass Publishers.

Miller, B. P., Koski, D., Lee, C. P., Maganty, V., Murthy, R., Natarajan, A., et al. (2000). *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services* (Web). Madison: University of Wisconsin, Madison.

MITRE. (2002). *Use of free and open-source software (FOSS) in the U.S. Department of Defense* (Web No. MP 02 W0000101). Defense Information Systems Agency.

Mockus, A., Fielding, R. T., & Herbsleb, J. (2000, June). A case study of open source software development: The Apache Server. *Proceedings of the the 22$^{nd}$ International Conference on Software Engineering*, Limerick, Ireland (pp. 263-272).

Moody, G. (2001). *Rebel code: Linux and the open source revolution*. London: Penguin.

Mowshowitz, A. (1997). Virtual organization. *Communications of the ACM, 40*(9), 30-37.

Mui, L., Verma, S., & Mohtashemi, M. (2005, February). A multi-agents model of the open source development process. *Proceedings of the International Conference on Technology, Knowledge and Society*, Berkeley, CA.

Netcraft. (2005). *Web server market share*. Retrieved December 1, 2005, from http://news.netcraft.com/archives/web_server_survey.html

Perens, B. (1997). *Debian social contract*. Retrieved December 1, 2005, from http://www.debian.org/social_contract

Perens, B. (2005). *The problem of software patents in standards*. Retrieved December 1, 2005, from http://perens.com/Articles/PatentFarming.html

Perry, W. E. (1991). *Quality assurance for information systems*. New York: QED Technical Publishing Group.

Pranevich, J. (2003). *The wonderful world of Linux 2.6*. Retrieved December 1, 2005, from http://kniggit.net/wwol26.html

Raymond, E. S. (1999). *The cathedral and the bazaar* (Vol. 1). Sebastopol, CA: O'Reilly & Associates.

Raymond, E. S., & Trader, W. C. (1999). Linux and open-source success. *IEEE Software, 16*(1), 85-89.

Robey, D., Boudreau, M.-C., & Storey, V. C. (1998). Looking before we leap: Foundations for a research program on virtual organizations and electronic commerce. In G. St-Amant & M. Amami (Eds.), *Electronic commerce: Papers from the Third International Conference on the Management of Networked Organizations* (pp. 275-290).

Rosen, L. (2005). *Open source licensing: Software freedom and intellectual property law*. Upper Saddle River, NJ: Prentice Hall.

Samoladas, I., Stamelos, I., Angelis, L., & Oikonomou, A. (2004). Open source software development should strive for even greater code maintainability. *Communications of the ACM, 47*(10), 83.

Sharma, S., Sugumaran, V., & Rajagopalan, B. (2002). A framework for creating hybrid-OSS communities. *Information Systems Journal, 12*(1), 7-26.

Stamelos, I., Angelis, L., Oikonomou, A., & Bleris, G. L. (2002). Code quality analysis in open-source software development. *Information Systems Journal, 12*(1), 43-60.

Sun Microsystems. (2005). *OpenSolaris (Version 10)* [Web]. Palo Alto, CA: Author.

Technorati, I. (2005). *Tag: Pesto*. Retrieved December 1, 2005, from http://www.technorati.com/tag/pesto

Torvalds, L. (2005). Kernel Number Versioning. Retrieved December 1, 2005, from http://kerneltrap.org/mailarchive/1/message/29288/thread

Torvalds, L., & Diamond, D. (2001). *Just for fun*. London: Texere.

Verma, S., & Jin, L. (2004). Diffusion and adoption of open source software within the open source community. *Proceedings of the 35th Annual Meeting of the Decision Sciences Institute*, Boston.

Wikipedia. (2005). *Security through obscurity*. Retrieved December 1, 2005, from http://en.wikipedia.org/wiki/Security_through_obscurity

Wong, W. E., & Gokhale, S. (2005). Static and dynamic distance metrics for feature-based code analysis. *The Journal of Systems and Software, 74*(3), 283.

XMPP.org. (2005). *History of XMPP*. Retrieved December 1, 2005, from http://www.xmpp.org/history.html

Young, M. (1958). *The rise of the meritocracy* (reprint ed.). Somerset, NJ: Transaction Publishers.

## Endnotes

[1] This document was authored using OpenOffice Writer, a component of the open source productivity suite available at http://www.openoffice.org/

[2] EULA is the generic term used for proprietary software licenses. Open source licenses are classified based on one of the fifty-six different licenses as certified by the Open Source Initiative. There is no common body that certifies proprietary software licenses.

[3] There appears to be some recent discussion amongst the core developers of PHPNuke to completely re-write the application and to avoid all the problems they faced with the original design. This is a significant drawback of the open source model.

[4] Mozilla still exists as a complete suite (codenamed SeaMonkey), but is used by a smaller proportion of users. The larger and more visible component is the Mozilla Firefox product.

[5] As of version 2.6.11, Linus Torvalds has proposed the use of a quad notation, wherein a release will be versioned as x.y.z.a, thereby adding one more loop to the quality assurance process.

## Chapter XIV

# Creating Information Systems Quality in Government Settings

Catherine Horiuchi, University of San Francisco, USA

## Abstract

*The level of information system quality that is considered acceptable to government technology managers and the public varies; operational success over the long run — a backward-mapping evaluation — is the most highly valued feature defining quality information systems. Public agencies, unlike private sector firms, exist to meet mandated service requirements and have limited ability to create revenue streams essential to fund information systems that might improve organizational bureaucracies. Beyond basic bureaucratic functions, governments serve the public interest by sustaining high reliability networks such as water and electric systems in addition to providing fundamental public safety and national security. High profile information system failures detract from public awareness that the bulk of systems work adequately within a complicated network of thousands of interlocking and overlapping governmental entities. Demands for productivity and innovation, increased contracting out of government services, and more sophisticated leadership shape the quality profile for the future.*

## Introduction

Two contrasting features — power and isolation — illustrate the complex nature of government information systems. The information system as an object of power acknowledges the contributions of hardware and software to overcome human limitations, extending the reach of human imagination, making possible the achievement of wildly improbable goals. We land a man on the moon and take astonishing photographs from the surface of Mars. We harness magnetic forces and x-rays to "see" inside the human body and program lasers to target cancers. These advances have at their heart processing requirements that direct machinery to rapidly, repeatedly evaluate material visible and invisible, and then file or retrieve data subsets representing this material from a massive number of discrete bits of information.

The information system as a source of isolation typifies systems with weak connection between the machine function and humanly defined operational goals. In this alternative assessment, government technologies make everyday life more difficult. We find ourselves stopped at traffic lights that do not recycle properly after the briefest power fluctuations. We receive incorrect or incomprehensible bills for government services. Automation isolates us from government. Impersonal telephone call automation software supplants direct contact with agencies on everyday matters; these are often programmed without any option to connect with a live agent.

The unique challenges facing public agencies in creating and sustaining information system quality are the topic of this chapter. It begins by defining how public agencies differ from private firms, citing research from the U.S. and abroad. After describing why variations in the degree of quality acceptable to government technology managers are inevitable, using examples from the U.S., it proposes specific attributes of information systems that increase chances for operational success over the long run, the measure of high quality in government systems. Once quality goals in the abstract have been covered, the chapter chronicles a high profile system failure where political considerations prevented the application of established quality assurance standards. Noting examples of high quality systems, the chapter considers future trends in government information systems and their likely impact on quality, along with ideas on addressing these trends to achieve more consistent outcomes. The chapter concludes with a summation of recurring suggestions for quality improvements and conditions under which they might be adopted.

Quality in information systems can be defined as reliable, cost effective hardware and software that perform fully documented jobs as tasked without unexpected flaws and failures, satisfying stated and implied needs. In government systems, assuring adherence to specified quality in performance requires guidelines in three areas: first, choosing and periodically revisiting the selection of an appropriate hardware and software model, whether a monolithic implementation or a modular design; second, managing projects ranging from enterprise-scale to much smaller endeavors; and third, accounting for expenditures across budget cycles over the life of a system. Despite a half century of efforts to specify systems requirements and metrics, consistent quality implementation remains an elusive goal. Two missions to Mars have crashed, as did the space shuttle Challenger in 1986 and Columbia in 2003 (Ackerman, 2003; Boisjoly et al., 1989). Most election results are tallied and confirmed swiftly thanks to electronic systems, but the U.S. presidential election in 2000 went badly. The blame fell on punch card ballots in Florida and a margin of victory smaller than the margin of error or achievable accuracy of the voting technology. The contributory role of information systems resulted in passage of the 2002 Help America Vote Act (HAVA), initiating “rip and replace” projects nationwide. Problems with the replacement touch-screen systems in 2004 have Florida election officials considering yet another round of technology shifts.

High quality technology services reduce system failures, creating an opportunity to fully fund projects associated with assuring quality in new systems or maintaining them in on-going operations. The cost of a major technology project is measured in tens of millions of dollars, even in local governmental settings. When fully loaded with costs for hardware, software, consulting, employee time, and training, a midsized process automation project can easily top $50 million. No city manager can brush off a bridge or building collapse; yet possibly due to a limited conviction that a much higher proportion of technology projects ought to be successful, multi-million-dollar technology failures are routine. As an alternative to targeting management efforts on reducing the overall proportion of failed projects, periodic agency budgeting processes become a Darwin-esque battle as if funded systems represent the survival of the fittest projects. Information systems managers struggle to fully fund important new and maintenance projects; these bureaucratic activities overwhelm many efforts to focus on quality.

## Background: How Public Information Systems Differ

Requirements for information system quality in public agencies differ from private firms in part because public agencies face no competition for clients and generally cannot be put out of business as a result of setbacks. Whether they are fundamentally alike in important or unimportant ways has been a perennial subject of discussion since before Graham T. Allison's (1982) article on the question. Private sector firms innovate for competitive advantage while public agencies tend to seek efficiency improvements. The public sector lags and tends toward late adoption of technology strategies developed for and tested in the private sector. Similarities include activities inherent in all human organizations: planning, organizing, staffing, directing, coordinating, reporting, and budgeting. Luther Gulick coined the acronym POSDCORB for these management skills in 1937, long before firms turned to computers to automate tasks in part or whole. Rocheleau (2000) noted public dependence on private sector information management strategies in Bozeman and Bretschneider's (1986) introduction to a series of articles on public management information systems. Private sector information system strategies are adopted wholesale or with minimal modifications to standardize and streamline these fundamental processes, creating a base of quality governmental information systems. Greater interoperability and information sharing among public agencies have improved access to government rules and regulations and created an enriched body of knowledge for public policy development (Landsbergen & Wolken, 2001).

Both public and private sectors adopt technological innovations and efficiencies; these implementations differ in willingness and capacity to invest labor and capital, especially training. Rocheleau and Wu (2002) tested whether private sector firms spend more on information technology, assuming these systems create competitive advantage in market economies. They found public and private firms characterize information technology as a strategic investment; however, public organizations are less willing to commit resources for training. Another study on public management decision making (Bozeman & Pandey, 2004) found similar reluctance to invest in training during implementation of hardware and software. In government technology decisions, customary budget constraints and cutback decisions were less important; technical feasibility and usability matter more than cost-effectiveness. The private sector frequently couples technological adoptions with staff reductions, so a well-trained, competent staff is essential to producing more with fewer employees. Scale and complexity in government technical projects lead to a longer decision-making cycle, but once determined, project decisions are stable and generally include measurable outcomes that usually exclude staff reductions. In these cases, training rarely results in directly measurable productivity improvements. This limits the ability of technology project managers to adequately argue the case for extensive training in technology projects. Instead these projects tend to focus on immediate hardware and software costs.

Government information systems projects frequently produce fewer functional improvements in terms of the services delivered; bureaucratic changes are more evident in assessments. For example, a descriptive study of local law enforcement information systems usage (Nunn, 2001) reviewed a 1993 report on information systems adoptions in 188 municipal police agencies. Nunn developed a taxonomy of police agencies with their stable fundamental duty for public safety, categorizing cities as low, medium, or high levels of computerization to explore differences in expenditures and police-officer allocation. Existing management and administration data influenced choice of information systems adoptions, particularly for basic business management and criminal research. Highly computerized cities had larger technical staffs and higher per capita technology expenditures. Lower computer systems expenditures were correlated with higher numbers of police officers per capita. The functional results of computing on operational efficiency and agency effectiveness were not investigated.

A 2004 Singapore study researched functional improvements related to information systems implementation among the oversight boards endemic to government (Weiling & Wei, 2004). Weiling and Wei examined civil service and regulatory board sites developed as part of Singapore's governmental initiative to create an "intelligent island". Problems in creating this included the civil service mindset, ambiguity of goals, limited technical capabilities, and financial resources, and the so-called digital divide, difficulties in offering Web-based services when a substantial fraction of the public is not network connected.

As home-based Internet access increases, so do opportunities for innovative government service delivery (Dawes et al., 1999). A growing number of households access networks through well-understood common user interfaces engineered in accordance with industry quality metrics. This self-learning resolves a portion of the training challenges described above. Service delivery benefits in Web-based outreach include improved information access at reduced cost, and increased integration across governmental agencies. Government employees and citizens alike bring to task Internet systems knowledge and skills transferred from personal use of the Internet at home.

The shift to Web-based services also introduces potential gains in right-sizing computer hardware and software systems, recognizing the limited flexibility and long cycle times in mainframe application installation. Improved public service via the Internet is developing despite the on-going challenge of a digital divide. Some research suggests the divide may resolve itself over time, at least as far as access to government services is concerned (Moe, 2004; Thomas & Streib, 2003; for a contrasting view on the digital divide as symptomatic of more fundamental sociopolitical forces, see Gorski, 2003). Optimism derives from high information system familiarity among youth and a finding that "net evaders" often live in households where others access the Internet on their behalf (Lenhart et al., 2003).

Three quality-related causes for government information system management failures — definition, process, and communication — are postulated in a study of 15 local governments in Poland as follows (Pawlowska, 2002). Executive leadership and managers inadequately described the actual information needs of the organization (quality of system definition). Project management failures derived from management expertise in either public management or information systems, but not both, resulting in oversight errors (quality of process management). Divergent group interests could not be resolved (quality of communication and cooperation). Failures in executive leadership in government are repeatedly noted (Kouzmin & Korac-Kakabadse, 2000; Rocheleau, 2000). Executives often substitute generic "best practices" or benchmarking targets for targeted mastery of government agency management strategies and technology fundamentals.

Governments can be innovation leaders in addition to more typical roles as late adopters or technological laggards. Creation of the Internet is a premier example of government in a leading role: the network of computer hosts, first called the ARPANet, originated in the U.S. Department of Defense's Advanced Research Projects Agency. Fiscal constraints and the non-entrepreneurial nature of most government work rank as causes for later adoption of technologies. Political reality requires balancing economic efficiency against fundamental protection of public rights, a constraint that can reduce or even erase the value of major information system initiatives such as business process reengineering (Kellough, 1998). A civil service workforce that sees limited turnover and training shortfalls also contribute to a pattern of late adoption.

## Variable Quality Requirements of Government

"Good enough for government work" goes the cliché in defense of mediocrity. Designers of government systems find the level of required quality varies by situation, and may remain fluid over the entire system life cycle. This creates unusual challenges for information system architecture, since costs for high quality systems may exceed the functional value that governments are willing to fund. This is the source of a government-specific definition of quality as reliable, cost effective operation of information systems as tasked. Standards such as the ISO 9000 series and Carnegie Mellon's Capacity Maturity Model provide more specific and robust guidance for system builders. ISO standard, ISO/IEC 14598-1:1999, provides a similar statement that internal quality consists of "the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs." ISO/IEC 9126-1:2001 defines external quality as "the extent to which a product satisfied stated and implied needs when used under specific conditions"

(International Organization for Standardization, n.d.). These general definitions combine with detailed specifications for certification, protecting purchasers of third-party information systems from risks inherent in products whose internal operations and construction are opaque.

## Government Systems Quality Requirements Vary by Design

Following ISO criteria for usability, security, and quality-in-use in spirit if not in their detailed totality, governments routinely design and construct systems to meet specific quality standards, such as *four nines* reliability, where a result is guaranteed 99.99% of the time. For example, an electric system that is reliable 99.9% of the time (*three nines* reliability) results in about 8 hours of power disruption each year. *Four nines* reliability reduces that to 53 minutes a year; this is the generally available level of electric system reliability in the U.S. Some customers — hospitals, manufacturers, and high technology firms — face operational demands for higher than average reliability, and are willing to pay for it. *Six nines* reliability reduces disruption to 30 seconds per year. Electric utilities are beginning to introduce granularity of system design and operation necessary to offer quality variation by customer. Meanwhile, some energy users develop private primary or supplemental power supplies. Similar to the *four nines* reliability in the energy industry, many government information systems have a single quality standard and offer universal service, regardless of ability to pay the full costs of services received. In these instances, governments match revenue to costs through specific tax levies, higher average fees, or on average lower quality service.

Government information systems often deviate from standard software development life cycle (SDLC) models. Programs are "tweaked" incessantly with changes in political environments. With limited time and less training, in-house personnel maintain information systems designed and installed by specialty consulting firms. In order to meet targeted return-on-investment (ROI) hurdles in preproject planning, conceptual supporters tend to underestimate initial implementation costs and overestimate utilization as well as the length of time the information system will be in place. Additional products are added in a somewhat piecemeal fashion to hardware and software already in place. Commercial off-the-shelf (COTS) products are modified to operate under non-standard conditions, reducing their intrinsically engineered adherence to quality standards. Finally, government agencies build from scratch or contract for information systems whenever no viable commercial market exists to meet system requirements. Complying with commercial-grade quality controls often exceeds the available budget for these systems, again making less rigorous quality standards acceptable.

## *High* Quality Lasts Longer, Does More; *Adequate* Quality is a Task Workhorse

In assessing the attainment of goals, quality on a system-by-system basis is defined through explicit measures of operational function, error rates, and variance from expected outcomes. Public managers instead commonly express quality goals in qualitative terms rather than quantitative targets. Qualitative terms better satisfy senior appointees and elected officials whose education and life experience include little technical background setting or interpreting performance metrics. Operationally, senior managers in government tasked with the success of information systems are similarly rarely educated in detailed definitions of quality metrics. In the spirit of these qualitative preferences, three quality ranges are described here: *adequate* quality and *high* quality, both acceptable alternatives to *poor* quality.

A software system of *adequate* quality accomplishes the minimum tasks for which it was designed, and continues to operate to accomplish most tasks until the hardware wears out, or changes in government rules result in its abandonment. This adequacy can be contrasted with the profile of a *poor* quality system that fails to be implemented, or if implemented is rarely used and not kept up to date. Poor quality systems may function precisely and correctly according to original detail specifications; but whether through faulty communications or changes in circumstances, poor systems are abandoned almost immediately. Staff and managers will retask for everyday purposes hardware and some component software acquired as part of the development of a poor system, incorporating these assets into higher quality systems whose utility was initially underestimated and where expansion is desired.

An information system of *high* quality also accomplishes the minimum tasks for which it was designed. It further performs other tasks that either were add-ons to the original design or that provide unexpected extra value. The high quality system produces consistently valuable results and is robust, exceeding its budgeted life for long periods. These systems can be recognized in the willingness of government agencies to make extraordinary efforts to keep them operational across multiple generations of hardware and software, and to protect them at times of agency reorganization or budget rationalization. When high quality systems are ultimately retired, replacement systems may be rejected several times before an agency settles on a new system with an adequate operational profile.

This pragmatic, backwards-looking orientation, what might be termed "quality in the rear-view mirror," is conspicuous in any working description of quality information systems in government. Pragmatism distinguishes quality determination in government settings where competitive pressures rarely create the rationale for new systems. Should the direct costs or operating profile of government fail to please the electorate, the leadership can and often is replaced. Even without public attention, managers frequently alter information systems expectations in imitation of private sector trends (Abrahamson & Fairchild, 1999). This heightens tensions among in-house technology staff held directly responsible for quality outcomes under shifting expectations. Long lag time between phases of an information system project also creates difficulties in assuring information systems quality. Managerial assessment of success is frequently determined not by meeting initial specifications, but by after-the-fact evaluation criteria incorporating interests of parties not engaged in earlier planning.

## Cases Exemplify Varying Quality in Government Information System Outcomes

What are the consequences in government systems of systems meeting the adequate or high quality measure? What happens when a system does not even reach the adequate level? Four examples of assessed quality failure and success follow, illustrating differences in outcomes. The first example, California's statewide system for child support payment tracking, attempted to meet a federal mandate. Two successful space projects follow. Finally, the state of Washington's contested gubernatorial election of 2004 illuminates difficulties faced when the engineered quality level is inadequate only in rare instances.

In passing the Family Support Act in 1988, the Federal government tasked all 50 states with assuring child support payments in accord with court orders. The State of California struggled for 15 years to design and implement what might seem to be a simple tracking system. The state's first attempt, the Statewide Automated Child Support System (SACSS) failed spectacularly, with banner headlines in November of 1997 declaring "Snarled Child Support Computer Project Dies" (Ellis, 1997). Foreshadowing the collapse of SACSS, the Legislative Analyst's Office estimated the 10-year project would cost 153 million dollars. Her office warned any delay risked damage to the state's budget, since failure threatened billions in matching funds and federal penalties (California Legislative Analyst's Office, 1995).

Unlike the consequences faced by private firms in competitive markets, the SACSS system failure led neither to agency collapse nor to the end of the project.

The federal mandate necessitated continued efforts to construct a workable information system despite high costs. In contrast, a sunset provision allowed closure of a tiny technology oversight agency following a public scandal and subsequent audit (California State Auditor, 2002) regarding the unconfirmed and overstated benefits of a 95 million dollar no-bid contract.

Ten years of unsuccessful SACSS implementation cost state taxpayers hundreds of millions of dollars and put at risk four billion dollars of federal funding. The total damage far exceeds these numbers, to the degree the state's children have not received court-ordered support payments. A recent economic analysis (Sorensen, 2004) estimates payments in arrears for California at 18 billion dollars, nearly 20% of the nation's total payments in arrears. This is more than four times the 3.4 billion dollars owed to the children of New York, the number two state in arrears, which shares California's profile in terms of divorce and awards for child support.

Following the SACSS collapse, the state in 1999 revised its plan for a child support system. A newly constituted California Department of Child Support Services holds responsibility to develop the renamed California Child Support Automation System. The original design was broken into two separate components: the Child Support Enforcement system and the State Disbursement Unit. With the revised plan comes a revised cost. For the enforcement system alone, the Legislative Analyst's Office has estimated that implementation will take an additional 10 years and cost 1.3 billion dollars, including 465 million dollars in state funds and the remainder from federal allocations. This is more than 10 times the original project cost (California State Auditor, 2005).

Systems that work draw scant attention. The public disinterest that is more typical than banner headlines indicates a measure of attainment of adequate system quality in the information systems of everyday government work, in contrast with professional studies of systems and organizations. Positivist and descriptive articles in professional and academic journals on government technology systems regularly describe information systems projects of moderate visibility and adequate quality. If these articles examining laudatory aspects of newer systems have a weakness, it is conferring on them a status of high quality. In public sector operating environments, attributes of highest value take time to confirm; durability and low maintenance costs cannot be assessed in newly functioning systems. As an example, criticism of early optimistic reporting on 1990s governmental reengineering projects activities appears repeatedly and initial evaluations have been replaced with more objective descriptions (Brown & Brudney, 1998; Johnston & Romzek, 1999; Lowery, 1998; Smith, 1999; for a contrasting opinion see Thompson, 1999)

This delay in quality assessment in the practice of public management can be seen in the complex projects undertaken by the National Aeronautics and Space

Administration (NASA). All NASA projects have been engineered with expectations of high quality information systems, since repairs to faulty systems in outer space are extremely hazardous, if possible at all. While public excitement centers on amazing stories, the dramatics in outer space are merely the most visible results of a complex collection of commercial general purpose hardware, real time operating system software, specially engineered materials, decision support systems, high-reliability communication systems, and mundane networked office computers. Despite this complexity in materials and networks, these projects generally succeed; two examples of high quality outcomes are described here.

The Pioneer 10 spacecraft was launched in 1972 for a mission lasting 21 months in outer space to fly past Jupiter and send back photographs (Cowen, 2003). After its initial mission targets were met, the spacecraft continued to send telemetry data back to Earth. NASA assigned additional tasks to the spacecraft, aiding the development of new generation communication technology systems. This continued for 31 years until early 2003. Pioneer 10 became the first manmade object to leave the solar system. It no longer sends signals, but continues to coast through the universe with one unending task: conveying a plaque locating Earth and describing this world's people, languages, and cultures.

A near-space project, the Hubble telescope originally was an unsuccessful technology implementation with its warped lens returning blurry images, blamed in part to mismanaged contracts. The space shuttle Endeavor mission STS-61 resolved the image distortion through the placement of corrective optics (Dumoulin, 2001; Flam, 1993). Following repair, Hubble captured astounding images of nearby and deep space objects. The underlying robustness of the system design created inherent potential for correction; organizational resilience and openness to change spurred successful repair. In January 2004 the NASA administrator announced that the government would cancel future servicing missions, allowing Hubble's orbit to decay. Public outcry and lobbying by the scientific community led to reconsideration of the decision to abandon Hubble. Again, system success beyond initial and revised expectations resulted in desire for further persistence, modifying the administration's space program goals to develop a plan so the Hubble project can continue.

Government information system quality derives from intrinsic system preparation for these adaptations that public agencies grow to recognize and then adopt. Information system quality begins, therefore, with forward mapping scenario planning. It does not end with implementation or even the post-implementation "lessons learned." The real lessons, at least regarding quality, emerge long into the information system life cycle. While the smallest information technology systems and commercial installations may not be well suited for specific scenario planning, at a departmental or agency level they can be bundled to evaluate overall information systems quality using scenario-based methods.

An alternate method of acknowledging the variable nature of government information systems demonstrates quality through explicit acknowledgment of acceptable margins of error. Meeting a commercial-grade software *four nines* reliability standard is deemed adequate in most situations. But on rare occasions, this affordable degree of precision will not produce satisfactory results. In very close elections, voting systems that have been engineered to a four nines level of precision by design produce disputed outcomes. This is not a technology error, *per se*. Assessing voting results in such rare circumstances requires by design extraordinary non-technological efforts, such as the abandonment of the automated system in favor of a hand recount (Cooper, 2004). When these secondary processes are followed, inherent flaws in systems external to the automated processes may be revealed in a collateral fashion, prolonging the dispute and fogging the issue of information system quality. As an example, in the disputed Washington state gubernatorial election of 2004, Snohomish County found 224 ballots under empty mail trays as they collected ballots for the first recount, a recalculation by machine. "Worker errors" were cited when, as part of the gathering of all ballots to a single statewide recount location, King County submitted 723 additional ballots for the second recount, a manual recalculation by hand with additional witnesses (Cook, 2004). As a result of the hand recount, the outcome of the election changed.

## Future Trends: No Silver Bullets

Three trends in government use of information technology affect efforts to improve information system quality. First, government managers increasingly recognize that information systems contribute to strategic innovation, beyond merely automating menial tasks. Second, agencies manage ever-larger contract commitments as a result of privatization, the extension of Milward's "hollow state" where governments provide fewer direct services and instead allocate funds to private or non-governmental organizations (NGOs) that serve the public on the government's behalf (Milward & Provan, 2000). Third, large expenditures on information systems have increased pressures on government managers to understand in greater technical detail their information system projects, resulting in changes in traditional leadership training or post-graduate education.

Two design paradigms — behavior science and design science — suggest complementary methods available to public managers to increase information system quality (Hevner et al., 2004). Behavioral science paradigms create concise conceptual frameworks for programs, based on estimations of the human and organizational systems in which they operate. Design science paradigms result in focus on the understanding of the specific problem domain for the system and its solution. In each case these paradigmatic assumptions are instantiated in the constructed information systems. Without a behavioral perspective, information system quality is compromised, as it is not feasible to confirm in all instances prenegotiated quality targets. Without a design science perspective, information system quality is compromised by the potential for delivery of flawless systems that do not address the specific problem for which the system was intended.

Interagency agreements and secure network technologies create collaborative opportunities for information systems innovation, in the presence of management capacity. The sometimes tenuous nature of partnerships suggests agencies benefit if they establish a software life cycle that co-terminates with an interagency or private/public partnership, unless all required data and skills for system continuation are acquired by each agency prior to termination. This requires better management of in-house and contracted human resources, including cataloguing employees' skills and aptitudes. Leadership training programs for government executives continue to focus on high-level interpersonal skills, but graduate education programs in public administration increasingly include components on managing information technology. As one example, the National Association of Schools of Public Affairs and Administration (NASPAA) now includes information management, technology applications, and policy as an explicit element of its curriculum standard (National Association of Schools of Public Affairs and Administration, 2005).

## Conclusion

Government entities resemble private sector firms operationally and can adopt with minimal difficulty information systems quality metrics to support production efficiency. However, unique challenges face public agencies as a result of mandatory and universal service requirements. Cyclical changes in the political environment through the electoral process heighten constraints. To create public value with limited and intermittent funding, government agencies have adopted quality variability as an operational norm. High quality systems are recognized more in hindsight for their long duration and high functional value. Exhaustive project management during implementation does not guarantee a high quality information system result, but forward mapping scenario planning helps prepare for likely adaptations when it can be affordably implemented.

The most likely strategy for increasing operational quality considers the long-term horizon, involving perspectives from behavior science and decision science to heighten potential for information system success over the long run. Information systems managers can expect modifications in specifications and organizational shifts in boundaries, requiring greater interagency connectedness and reliance on non-governmental organizations as partners. Success in organizational leadership includes satisfactory management of information systems; managers can improve their understanding of systems and capacity for effective systems management through continuing education. Affordable maintenance and imaginative goal extensions as shown in the exemplary systems above illustrate attributes of high quality information systems that can be accomplished on even the smallest scale projects when guided by forward-thinking managers.

## References

Abrahamson, E., & Fairchild, G. (1999). Management fashion: Lifecycles, triggers, and collective learning processes. *Administrative Science Quarterly, 44*(4), 708-740.

Ackerman, T. (2003, August 26). "Echoes of Challenger" in Columbia blast. *The Houston Chronicle*. Retrieved December 1, 2005, from http://www.chron.com/cs/CDA/ssistory.mpl/space/206800

Allison, G. T., Jr. (1982). Public and private management: Are they fundamentally alike in all important respects? In R. J. Stillman, III, (Ed.), *Public administration: Concepts and cases* (6th ed., pp. 219-306). Boston: Houghton Mifflin Company.

Boisjoly, R. P., Curtis, E. F., & Mellicam, E. (1989). Roger Boisjoly and the Challenger disaster: The ethical dimensions. *Journal of Business Ethics, 8*(4), 217-230.

Bozeman, B., & Bretschneider, S. (1986). Special issue: Public management information systems. *Public Administration Review, 46*, 475-487.

Bozeman, B., & Pandey, S. (2004). Public management decision making: Effects of decision content. *Public Administration Review, 64*(5), 553-565.

Brown, M. M. & Brudney, J. L. (1998). A "smarter, better, faster, and cheaper" government: Contracting and geographic information systems. *Public Administration Review, 58*(4), 335-345.

California Legislative Analyst's Office. (1995, February 22). Analysis of the 1995-96 budget bill: Statewide Automated Child Support System: Automation project will miss federal deadline for enhanced funding. Retrieved December 1, 2005, from http://www.lao.ca.gov/analysis_1995/chit5180.html

California State Auditor. (2002, April 16). *Report 2001-128: Enterprise licensing agreement: The state failed to exercise due diligence when contracting with Oracle, potentially costing taxpayers millions of dollars*. Sacramento, CA: Author.

California State Auditor. (2005, March 8). *Report 99028.4: Child Support Enforcement Program: The state has contracted with Bank of America to implement the State Disbursement Unit to collect and disburse child support payments*. Sacramento, CA: Author.

Cook, R. (2004, December 21). Elections chief: At least 5 counties added ballots during recounts; Supreme Court to hear King County issue Wednesday. *The Columbian*, p. A1.

Cooper, M. H. (2004, October 29). Voting rights: Will the votes of all Americans be counted on Nov 2? *CQ Researcher, 14*(38), 903-920.

Cowen, R. (2003). Death of a pioneer. *Science News, 163*(10), 158.

Dawes, S. S., Pardo, T. A., & DiCaterino, A. (1999). Crossing the threshold: Practical foundations for government services on the World Wide Web. *Journal of the American Society for Information Science, 50*(4), 346-353.

Dumoulin, J. (2001). STS-61 (59). Retrieved December 1, 2005, from NASA Space Shuttle Launch Archive: http://science.ksc.nasa.gov/shuttle/missions/sts-61/mission-sts-61.html

Ellis, V. (1997, November 21). Snarled child support computer project dies; Technology: State drops system after spending $100 million and could face up to $4 billion in federal penalties. *Los Angeles Times*, p. A1.

Flam, F. (1993). NASA stakes its reputation on fix for Hubble telescope. *Science, 259*, 887-889.

Gorski, P. C. (2003). Privilege and repression in the digital era: Rethinking the sociopolitics of the digital divide. *Race, Gender & Class, 10*(4), 145-176.

Gulick, L. (1937). Notes on the theory of organization. In J. M. Shafritz, A. C. Hyde, & S. Parkes (Eds.), (2004), *Classics of public administration* (5th ed., pp. 90-98). Belmont: Thomson Wadsworth.

Hevner, A. R, March, S. T, Park, J., & Ram, S. (2004). Design science in information systems research. *MIS Quarterly, 28*(1), 75-105.

International Organization for Standardization (ND). (n.d.). *ISO 9000: Quality management principles*. Retrieved December 1, 2005, from http://www.iso.org/iso/en/ iso9000-14000/iso9000/qmp.html

Johnston, J. M., & Romzek, B. (1999). Contracting and accountability in state Medicaid reform: Rhetoric, theories, and reality. *Public Administration Review, 59*(5), 383-399.

Kellough, J. E. (1998). The reinventing government movement: A review and critique. *Public Administration Quarterly, 22*(1), 6-20.

Kouzmin, A., & Korac-Kakabadse, N. (2000). Benchmarking and mapping institutional impacts of lean communication in lean agencies: Information technology illiteracy and leadership failure. *Administration and Society, 32*(1), 29-69.

Landsbergen, D., Jr., & Wolken, G., Jr. (2001). Realizing the promise: Government information systems and the fourth generation of information technology. *Public Administration Review, 61*(2), 206-220.

Lenhart, A., Horrigan, J., Rainie, L., Allen, K., Boyce, A., Madden, M., et al. (2003). *The ever-shifting Internet population: A new look at Internet access and the digital divide*. Retrieved December 1, 2005, from Pew Internet and American Life Project at: http://www.pewinternet.org

Lowery, D. (1998). ISO 9000: A certification-based technology for reinventing the federal government. *Public Productivity & Management Review, 22*(2), 232-250.

Milward, H. B., & Provan, K. G. (2000). Governing the hollow state. *Journal of Public Administration Research and Theory, 10*(2), 359-379.

Moe, T. (2004). Bridging the "digital divide" in Colorado libraries: Survey results from the Colorado Public Libraries and the "digital divide" 2002 study. *Public Libraries, 43*(4), 227-231.

National Association of Schools of Public Affairs and Administration, Commission on Peer Review and Accreditation. (2005, January). *General information and standards for professional masters degree programs*. Retrieved December 1, 2005, from http://www.naspaa.org/accreditation/seeking/reference/standards.asp

Nunn, S. (2001). Police information technology: Assessing the effects of computerization on urban police functions. *Public Administration Review, 61*(2), 221-234.

Pawlowska, A. (2002). Computing in Polish local administration – new technology, old experience. *Information Polity: The International Journal of Government & Democracy in the Information Age, 7*(1), 49-64.

Rocheleau, B. (2000). Prescriptions for public-sector information management: A review, analysis, and critique. *American Review of Public Administration, 30*(4), 414-435.

Rocheleau, B., & Wu, L. (2002). Public versus private information systems. *American Review of Public Administration, 32*(4), 379-397.

Smith, J. F. (1999). The benefits and threats of PBB: An assessment of modern reform. *Public Budgeting & Finance, 19*(3), 3-15.

Sorensen, E. (2004). Understanding how child support arrears reached $18 billion in California. *American Economic Review, 94*(2), 312-316.

Thomas, J. C., & Streib, G. (2003). The new face of government: citizen-initiated contacts in the era of e-government. *Journal of Public Administration Research and Theory, 13*(1), 83-101.

Thompson, J. R. (1999). Devising administrative reform that works: The example of the reinvention lab program. *Public Administration Review, 59*(4), 283-292.

Weiling, K., & Wei, K. K. (2004). Successful e-government in Singapore. *Communications of the ACM, 47*(6), 95-99.

# Chapter XV

# ERP Quality: Do Individual Users Matter? An Australian Case

Jenine Beekhuyzen, Griffith University, Australia

## Abstract

*A dramatic increase in the number of corporate organisations using or implementing ERP systems across a range of different markets and functional units has transpired in the past decade. This saturation in the corporate world has led to the significant uptake of these types of Information Systems by universities around the world. The corporate use of Enterprise Resource Planning (ERP) systems has been well discussed in the literature; therefore, this chapter brings a focus to the critical use of ERP systems within university environments. The need for supporting both the individual and the organization is an important aspect frequently missed by technology solutions. The definitions of ERP suggest they are far-reaching and all-encompassing, but do ERP systems meet the quality requirements of individual users?*

# Introduction

Enterprise Resource Planning (ERP) systems have been defined as "on-line interactive systems that can provide a 'total' solution to an organisation's information systems needs by addressing a large proportion of business functions" (Brown & Vessey, 1999). In line with this definition, this study views these complex systems as "shared information systems crossing typical organisational boundaries, having multiple users and stakeholders with different cultures and approaches to work" (Pawlowski et al., 2000). These statements suggest that ERPs are far-reaching and all-encompassing; we investigate if ERP systems meet the quality requirements of individual users.

A dramatic increase in the number of corporate organizations using or implementing ERP systems across a range of different markets and functional units has transpired in the past decade. Davenport (1998) began the discussion surrounding the significance of ERP systems adoption; Esteves and Pastor (2001) suggested the business world's embrace of enterprise systems may in fact have been the most important development in the corporate use of information technology in the 1990s; in 2005, von Hellens et al. commented on the move into higher education.

ERP systems aim to provide a "big picture" approach to systems implementation, which can cause a myriad of difficulties for different sectors, organizational types, management styles, and most of all, individual users. The need for supporting both the individual *and* the organization is an important aspect frequently missed by technology solutions: ERP systems tend to address the needs of the organization at the expense of the individual (Slade & Bokma, 2001). Commonly, management focus on centralization of data and processes and reporting in the first instance with very little focus on the technology solution actively and substantially supporting the individual user in their work.

This chapter presents an analysis of the literature on ERP adoption within a university environment, focusing on quality and how it relates to individual user satisfaction. User satisfaction is crucial in this context because of the non-traditional implementation process; users may not be very involved in requirements engineering because it is abbreviated; however, because implementation entails "configuration" (i.e., invoking switches to implement the organization's business logic) which determines what the final product will look like, this is where users involvement should be highest (but often is not).

Relating to the literature, this chapter presents a particular case of Information Systems (IS) quality through an ERP implementation within a large Australian university. The case is examined through the lens of the SOLE quality model developed by Eriksson and Törn (1991). In this case, use quality is examined from a very specific point of view, that of ERP system users. This is presented in terms of IS use practices such as requirement and interface quality.

The literature suggests (and this study reinforces) that individual users are rarely considered during ERP development and deployment. This research suggests that more attention to system quality and individual user needs during development and implementation may help to increase user acceptance of ERP systems. This research aims to be of value to the ERP industry and academia alike as it discusses the impacts on the individual user's (academic and administrative) quality of work life in a university environment.

## ERP Move into Higher Education

The saturation of ERPs in the corporate world has led to the significant uptake of these types of information systems by universities around the world. Large investments for ERP systems are typically acquired by large corporate organizations; however, less corporate organizations such as universities are now using them. When making this large investment, universities must be prepared for the high risks involved and be aware of the difficulty of recovering from a large system failure.

The corporate use of Enterprise Resource Planning (ERP) systems has been well discussed in the literature; therefore, this chapter brings a focus to the critical use of ERP systems within university environments. In Australia alone, almost all of the 42 universities have implemented at least one module of an ERP system (Beekhuyzen et al., 2002), in line with global trends. A number of significant failures in the Australian University context have been reported in the literature and the media, with the main problems stemming from lack of system and data coordination across departments, considerable changes in user roles, unclear vision and expectations of the system, and minimal training in the use of the new system.

Within the context of universities, ERP systems are huge and complex, controlling a range of business processes in a single application linked to a central database stemming a host of functionality. Also considering the high investments for ERP, there is typically a great deal of hype to win the investment dollars, thus greater user expectation. This has led to huge disappointments later and even post-implementation depression in the organization.

# ERP Literature

The number of publications dedicated to ERP within the information systems discipline appears small compared to the size of the business they generate. According to Esteves and Pastor (2001), research on ERP systems has been treated as a “secondary”, and its importance has been neglected by the IS community. Researchers have argued the need for more investigations into ERP (Allen et al., 2002; Esteves & Pastor, 2001; Gable, 1998). Due to their pervasive nature and the adoption of this technology across diverse markets, ERP systems are of interest for a wide range of professional and scholarly communities, as well as the IS field (Esteves & Pastor, 2001).

Unsuccessful implementations of information systems have been widely cited in the literature (Davenport, 1998; Markus, 1983; Mitev, 2000), and ERP systems are no exception (Gefen 2000; Markus & Tanis, 2000). Despite this, little research currently exists with which to theorize the important predictors for initial and on-going ERP implementation success (Brown & Vessey, 1999).

A dramatic increase in the number of organizations using or implementing ERP systems across a range of different market areas has transpired over the past decade. Davenport (1998) suggests that ERP systems adoption is significant; the business world’s embrace of enterprise systems may in fact have been the most important development in the corporate use of information technology in the 1990s (Esteves & Pastor, 2001). This increase has included the significant uptake of these types of information systems by universities around the world.

The successful adoption of information technology (IT), and increasingly ERPs, has become a major topic in the IS literature (Allen & Kern, 2001; Davenport, 1998; DeLone & McLean, 1992, 2003; Gefen, 2000; Markus & Tanis, 2000). Much of the literature suggests that a significant number of ERP projects do not succeed (Markus & Tanis, 2000) due to overlooked human aspects (Sarker & Lee, 2000). Where there is conflict accepting the new system, something has to give; either the software or the organization must change (Slater, 1999).

This chapter does not attempt to present all facets of the literature on ERP, but to focus on the higher education sector, specifically universities. Esteves and Pastor (2001) and Klaus et al. (2000) have published lengthy discussions of the ERP literature.

## University Sector

In recent years, major structural, political, and cultural change has affected universities across the world, with the higher education institutions in which we work fundamentally changing (Allen & Kern, 2001). In Australia in the late 1980s there were calls from the government to attract more students into universities (Hore & Barwood, 1989) when it became clear that universities needed to improve economic efficiency resulting in a restructuring of the entire Australian university sector. Some people claim the higher education sector has been through a phase termed "the corporatization of universities" (Guthrie & Neumann, 2001). Wagner and Scott (2001) refer to this trend for changes to business solutions to reflect the marketization of universities within the global higher education sector, which has grown increasingly complex and competitive over the past decade.

As an answer to government policies, politics, social, and economical factors (Anderson et al., 1999), strategic directions for universities have included the use of information technology to streamline the university operations. These strategies attempt to utilize IT in the direction of a possible increase in competitiveness and to improve efficiency by relying on large-scale commercial information systems. These IT strategies were initiated between the mid- to late-1990s (AVCC, 1996; Meredyth & Thomas, 1996), with some of these IT projects deemed essential for universities to operate and described as "necessary for survival" (AVCC, 1996; Oliver & Romm, 2000; Yetton, 1997). Many of these IT "essentials" are now ERP systems.

The major ERP vendors (PeopleSoft, Oracle, SAP, and JD Edwards and Baan) have historically focused on the corporate market, with transitions into higher education in the past decade offering a campus management/student administration module in addition to their other modules. In Australia, universities have embraced this functionality allowing them to integrate all core functions across the university, with many university ERP systems supporting functions across multiple campuses.

Allen and Kern (2001) suggest that the area of ERP implementations in universities is largely unexplored by information systems researchers with few studies carried out (with exception of their own). As ERP systems are based on "best practices" (Davenport, 1998) organizations, including universities, expect that adoption will bring benefits to organizations in terms of efficiency and useability, reducing costs, and providing more centralized control.

In light of Y2K, Commonwealth Government policies, including Goods and Services Tax (GST), and subsequent university restructuring, replacing outdated and overlapping paper-based systems, is part of a strategic shift of the ERP implementing universities. However, within this context, Allen et al. (2002) argue that the cost feasibility of system integration, training, and user licenses may, in the end, impede ERP system utilization.

# Quality Literature

Quality in the context of information technologies is not easy to define (Eriksson & Törn, 1997). In fact, one of the most difficult tasks in studying quality is defining quality (Lindroos, 1997). Information systems quality is a very broad concept (Eriksson & Törn, 1997). In the information systems quality management literature, IS quality has been dichotomized as information quality and software quality (Swanson, 1997) or information systems quality and software quality (von Hellens, 1997). The same literature suggests that system quality, information quality, and software quality affect system use and user satisfaction with consequent individual and organizational impacts.

Another aspect of quality also needs to be viewed in this context — product quality. The ISO9126 product quality standard attempts to quantify the "usability" of a system viewing "quality in use" as the capability of the software product to enable the specified users to achieve specified goals with effectiveness, productivity, safety, and satisfaction in specified contexts of use. The means to do this are the six attributes of product quality; however, the following four are most relevant in this context:

- **Functionality** (suitability, accuracy, compliance with needs);
- **Reliability** (maturity, fault tolerance, recoverability);
- **Understandability** (learnability, operability, attractiveness);
- **Efficiency** (time behavior, resource behavior).

In their simplest form, ERPs need to be viewed as packaged software; as being nothing more than generic representations of the way a typical organization does business (Koch et al., 1999). As packaged software in their basic form, a mismatch of requirements and offering is an ever-present danger; however, ERPs are unlike off-the-shelf packaged software in that wide choices are often not available; thus the build alternative is not an option as would be the case in a stand-alone functional system.

ERPs are developed according to what is "best" for an organization — based on best practice. This lack of focus on individual users can lead to limited user satisfaction with the quality of an ERP system, which can have a large bearing on its acceptance and, ultimately, its success.

Attempts to specify the quality of a system from the user's point of view have been conducted (Lindroos, 1997); however as new technologies are emerging at a rapid pace (such as ERP systems) (Joshi & Rai, 2000), the quality needs and expectations of users are constantly changing. There is growing literature on ERP systems, such as evidenced by Esteves and Pastor's (2001) paper; however there are still very few publications looking explicitly at the relationship between ERP systems and IS quality, particularly, the user.

## User View of Quality

As we are looking at use quality, it is important to identify what we mean by users and who they are. Users have been defined as "the people who directly use the software by performing the work practices that prepare data and information for the software systems under consideration" (Andersson & von Hellens, 1997).

This study is presented from the user's view of IS quality and explores whether the quality problems present are *system* quality or *software* quality issues. The user-based definition is that "quality lies in the eyes of the beholder" (Garvin, 1984) and this is a highly subjective process. Use quality or quality from the user's perspective is as hard to define as quality alone (Lindroos, 1997).

The definition of the user has changed over time in the literature to include cultural considerations (Lindroos, 1997). Cultural feasibility and systemic desirability can be seen as a duality; the culture constrains what changes are perceived to be meaningful, but any actions will create/recreate the culture (i.e., will change or reinforce ideas about what is considered to be meaningful) (Vidgen et al., 1993). The presence of multiple stakeholders with different perspectives means that the definition of use quality can be problematic (Vidgen et al., 1993) despite its significance.

In this case, users of the ERP system are a heterogenous group and therefore have different requirements. Each user group uses the system for different reasons, and these groups can be somewhat categorized in terms of initiators (inputting the details); approvers (approving the details); and facilitators (processing the details) based on their roles with the system. It is important to note that use of this system is essential in order to get sessional staff paid. Before implementation, this process was a manual process taking about 5 minutes. Once the system was implemented, the electronic process then took up to an hour.

Table 1 identifies the user groups involved and their specific requirements. The table defines the different priorities of user groups in regard to the requirements of the system. These are based on the literature and a good understanding of the roles of the users.

Taking this into consideration, the user perspective is difficult to reduce to any single quality attribute of software (von Hellens, 1997) because it is influenced by the user's experience and history in using similar systems.

*Table 1. Specific requirements of user groups*

| User Group | Requirement Quality | Interface Quality |
|---|---|---|
| Initiator | Efficient entering of details | Ease of use, ease of learning |
| | Storage facilities | Adaptability (novice vs. expert) |
| Approver | Efficient processing | Flexibility |
| | Reliability | Speed |
| Facilitator | Security | Communicativeness |
| | Augmentability | Non-controlling |

# Research Approach

In order to explore the relationship between ERP systems implementation and user perceptions of quality, a case study employing multiple methods of data collection (Benbasat et al., 1987) has been conducted within a large Australian university. Case study research and an extensive review of the Information Systems Quality Management literature has been beneficial in gaining a wider understanding of the quality issues that could present themselves in an ERP system implementation. Individual users' perspectives of the system quality are also discussed.

The organizational viewpoint of IS quality is adopted for this research. This view is interested in the impact systems and technologies have on the way organizations work. This approach emphasizes the usability of the system and understands that an information system gives new meanings to everyday things in an organization (von Hellens, 1997). The focus of this viewpoint is on people, information, and work practices as it considers the quality of work practices to which the computer-based information systems provide support (Alter, 2002).

## The Framework

The SOLE (SOftware Library Evolution) Model (Andersson & von Hellens, 1997) aims to maximize the utility of the information systems in an organization (Eriksson & Törn, 1991). The components of quality discussed in this case study are in line with the three classes of IS quality presented by Eriksson and Törn (1991). The SOLE Model was chosen for this study as it provides an insight into the closely related components of quality within the context of information systems. As one level of quality will impact on other parts of the system and its ultimate quality, it is important to understand these relationships and the impact they can have on the overall system acceptance and success.

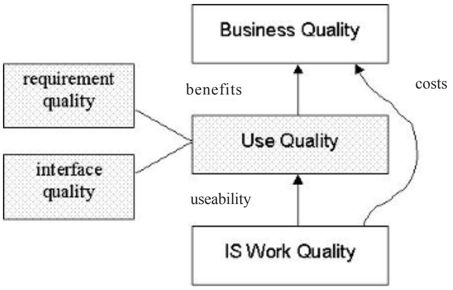
The SOLE Model components are closely interrelated. The level of IS work quality impacts the use quality via feedback mechanisms that in turn affect the business quality within the university. *Business quality* is concerned with cost effectiveness and economic feasibility; *use quality* which covers all aspects of how well the system serves the user; and *IS work quality* considers IS management, operation, and evolution. Most of the quality concepts explored in the information systems literature deal with quality as belonging to IS work quality. IS work quality and use quality are needed in order to ensure that the information systems are actually designed and used in a way that produces the business benefits (Salmela, 1997).

In particular, IS use quality factors were investigated in this study. Use quality in the SOLE model looks specifically at requirement quality; the system should do the right things in a user-friendly way; and at interface quality where the information produced should be correct, up-to-date, and relevant (Eriksson & Törn, 1991). The use quality component of the SOLE Model corresponds to Garvin's user-based approach to quality (Garvin, 1984). Use quality has also been identified by DeLone and McLean (1992, 2003) as being imperative in the overall system success picture. ERP system use quality and its subsequent success (or failure) are explored in the case study.

## Methodology

It is suggested (Benbasat et al., 1987; Neumann, 1997) that every researcher should collect data using one or more methods/techniques and "as much as possible, the contextual and data richness of the study should be presented, and a clear chain of evidence should be established" (Benbasat et al., 1987). Multiple data collection methods are typically employed in case research studies as evidence from two or more sources will converge to support the research findings (Benbasat et al., 1987). Yin (1994) identifies several sources of evidence that work well in case research: documentation, archival records, interviews, direct observations (which involves noting details, actions, or subtle-

*Figure 1. SOLE quality model (Adapted from Eriksson and Törn (1997) & Andersson and von Hellens (1997)*



ties of the field environment), and physical artifacts. All of these sources of evidence are used in this research.

This study, conducted in an IT school of a large Australian university, was made up of male and female participants with a range of experience and areas of expertise. There was a two-stage observation process, followed by semi-structured open-ended interviews. Observation of a group training session was conducted initially. This session had 10 participants and was attended by four implementation staff. In this session, the implementation staff outlined the functionality of the system, and users were invited to ask questions.

The next stage of the research involved non-participant and unobtrusive individual observations of seven users using the system for the first time. Following this, 10 interviews of approximately half to one hour in duration continued to explore the nature of the impact of the system quality on the user's work environment. User's behavior and interaction with organizational artifacts, namely, the new system, was then analyzed. The majority of the users that were interviewed were participants in the observation sessions.

## Case Study

Common in ERP implementations are the customizations conducted by the implementing consultants. As ERP systems typically dramatically alter work practices, specific change management is required to facilitate the changeover. These modifications to the "out of the box" package often can result in an exponential growth of the system and requirements, and often force an organization's business processes to change dramatically (Furumo & Pearson, 2004; Yakovlev & Anderson, 2000). Each of the system changes implemented can have a significant impact on capturing data inputs, and this will ultimately influence the quality of the data. In some cases, loss of productivity in data input is not as important as gains in the availability of decision-making information and changes in other processes (O'Leary, 2000).

ERP implementations do not stress the need to identify the user groups that will be using the system; however this is vitally important. The case implementation did identify the broad user groups, however, gave them little attention once the implementation was under way.

Several factors negatively influencing the ERP implementation experience were found. Specific examples are discussed in terms of requirement and interface quality issues. One user commented during an interview that the system was "... just a matter of struggling with yet another badly designed human interface."

These factors illustrate the need for rigorous examination of the capabilities of an ERP package to determine its “fit” with its potential users.

## Requirements Quality

In terms of requirement quality, the SOLE Model considers whether the information system is designed to meet the user’s needs (Eriksson & Törn, 1991). Security and augmentability (prepared for anticipated changes) are of importance in achieving requirement quality. Some users commented on the requirement quality of the ERP:

*“I was disappointed that I couldn’t change data I had just entered” (User3)*

*“I think the design and some of the stupidity in it are definitely sub standard” (User6)*

This study identified a number of specific problems with the requirements of the system. Examples include those shown in Table 2.

*Table 2. Requirement quality issues*

| | | |
|---|---|---|
| **Format of course code** | *The course code is not formatted in the generally accepted way.* | The first function in order to use the system is to enter the course code in the search field; however the only course code the system will accept is one that is in the usual way, e.g., the usual way to represent a code is CIT1101; however the system would only accept Cit1101. |
| **Incorrect input for employee characters** | *The employee number input does not accommodate current employee number formats* | The employee number entered must be 8 characters long; however most current employee numbers only have 4 to 6 characters, so for the code to be accepted, 3 leading 0s must be used, e.g., s388775 becomes s000388775. |
| **Employee addresses do not match** | *Differing employee addresses is displayed* | When entering an employee in the system and the employee address displayed from the search is different to the address given by the employee, it is not clear what should be done, e.g., does the initiator just assume that this is the correct person displayed and that the system has the address wrong and change the details in the system, or should they create a new employee based on name? |
| **Summary details are lost** | *No option to save the summary* | It is not possible for the initiator to save a copy of the summary. The only option is to print a hard copy. |

## Interface Quality

Eriksson and Törn (1991) argue that interface quality measures whether the information system is easy to use, easy to learn, and communicative. Issues, such as whether the interface is adaptable to the style of the user (novice vs. expert), its flexibility, and its speed, are also considered as interface quality essentials. Whether the system and interface controls the work performance is also explored. In regard to interface quality some comments from users included:

*"It's just a matter of struggling with yet another badly designed human interface" (User5)*

Some of the instructions concerning work flow weren't as clear as they could be, so it's not a great job of user interface design at all" (User3)

*Table 3. Interface quality issues*

| | | |
|---|---|---|
| **Format of course code** | *The course code is not formatted in the generally accepted way.* | Even though the input course code format is specific, there is no formatting example next to the input box. The code needed is not intuitive. |
| **Visibility of selected department** | *The selected department is not visible* | When a department is selected (other than the default), the change is not reflected in the department description next to the selection box. |
| **Incorrect input for employee characters** | *The employee number input does not accommodate current employee number formats* | Even though the input employee number format is specific, there is no formatting example next to the input box. The extra characters code needed are not intuitive. |
| **Essential links not visible on screen** | *A number of essential links are not easily viewable on the screen* | When displaying the screen for input of work schedule details, the link to the next page is over the edge of the displayed screen. The user must scroll across the page to see the link. This is not intuitive. |
| **Page split is inconvenient** | *A page with essential codes is displayed over two pages* | The essential work schedule codes are displayed across two pages; however using the first page listing still requires the user to scroll down to the bottom of the list. As the list on the second page is quite short, it would be more effective to have all work schedule listings on one page. |
| **Working weeks unclear** | *Holiday weeks are highlighted in the work schedules* | The holiday periods without scheduled classes are not made clear, and users often add details to these boxes without realizing. This results in incorrect costs incurred to the course and payments made incorrectly to staff. |
| **The 'fill all' button is confusing** | *Buttons are easily mistaken* | The "Fill all" button on the work schedule page is visible and easy to press by mistake. It is often confused with being a continue button as the continue button is not visible without scrolling across the page. |
| **Tab moving is not logical** | *Tab moves from week 1 to 10* | When using the TAB button to move from week to week, the cursor moves across the page and not consecutively through the weeks. |

*"It's to do with this stupid user interface there are some extremely small bits on the user interface that are actually are quite important and finding them is a problem" (User8)*

This study identified a number of specific problems with the interface of the system. Examples include those shown in Table 3.

## Discussion

To overcome the challenges of ERP implementations and the barriers to adoption, the right solution must enable universities to manage all of the computing resources that participate in the delivery of the university's products and services (EMBPWG, 1998, p. 5). Best practices in terms of system architecture state that the user interface should display information from a multiplicity of perspectives, which will equate to greater utility as a larger audience will accept working with the tool (EMBPWG, 1998), p. 7). Ease of use is almost always a factor in comparative ERP evaluations (O'Leary, 2000) and EMBPWG suggest that the user interface should be easily customizable so that administrators can move beyond elemental management to actual business process management.

In relation to ERP requirement quality issues, critical assumptions are often made that should be examined by the organization implementing the ERP. It is often assumed that the "best" ERP package is the one with the most listed features. This can discount the fact that the focus should be on the value creation opportunity of the specific modules adopted (O'Leary, 2000).

It is concluded that the majority of the IS quality issues presented in this chapter are product quality issues, as opposed to system quality issues or information quality issues. Also, interface quality issues are slightly more evident than requirement quality issues. More attention to the needs of the individual user will improve system effectiveness and reduce the impact of the system.

Usability is an important issue that needs particular attention during development and should also be considered heavily during customization. ERP vendors have increasingly tried to increase their systems user-friendliness (O'Leary, 2000); however this needs much more focus. More attention also needs to be paid to the requirement quality issues of ERP during implementation.

This discussion leads to an understanding that a more user-centered focus during software development would lead to increased system acceptance and, therefore, success. As ERP systems are packaged software, this advice needs to be adopted by ERP developers and implementers.

This study was situated within the university environment, and problems found were specific to this context; however many of the quality problems and issues presented are very relevant to other implementation contexts.

## Conclusion

This chapter presents an analysis of the literature on ERP adoption in universities with a focus on quality and how it relates to individual user satisfaction. The literature suggests that individual users are rarely considered during ERP development and implementation.

A specific case of ERP implementation in a university is investigated and viewed through the lens of Eriksson and Törn's SOLE quality model. Specifically exploring requirement quality and interface quality allowed a deep understanding of the users' interactions and experiences with the system. The quality issues identified in this case can be classified as predominantly system quality issues and interface (usability) issues.

An interesting dichotomy is presented: are the quality issues identified in this case a result of implementation (service) errors or ERP package (software) errors? The data suggest they are implementation errors; thus a different implementation team may have gathered different requirements resulting in a different approach to ERP quality.

This study explores the relationship between ERP systems and the quality requirements of individual users and suggests that more attention to system quality and individual user needs during development and implementation may help to increase user acceptance and, therefore, the overall success of ERP systems.

This research aims to be of value to the ERP industry and academia alike as it discusses the impacts on the individual user's (academic and administrative) quality of work life in a university environment. It aims to be a solid basis for further research into ERPs and use quality. It is hoped that this research provides value for future ERP implementation projects and that it has helped to develop a practical understanding of IT and particularly the impacts of ERP on the individual user's (academic and administrative staff) quality of work life in a university environment.

## Prologue

In late 2003, two years after the completion of this study of the pilot ERP implementation of the Sessional System, the university decided to *not* roll the Sessional System out to the entire university. Instead they decided to discontinue the use of the Sessional System altogether. The two departments involved in the pilots were to change back to their old processes of processing sessional staff. Reasons given for the abandonment of the system were the high costs involved.

When the abandonment of the system was announced, there was a feeling of discontent and annoyance among users that participated in this study. After investing much of their time in firstly learning the new system and, secondly, using it and dealing with its frustrations for the past three years, it is not surprising that the users feel somewhat cheated. This decision for abandonment seems rather ironic in that suggestions from these users that are discussed here and in the larger study of this situation (Beekhuyzen, 2001) were presented to the university ERP project team in early 2002. The receipt of these suggestions was not acknowledged, nor were they included in updates of the system.

## References

Allen, D., & Kern, T. (2001). Enterprise resource planning implementation: Stories of power, politics and resistance. *Proceedings of the IFIP Working Group 8.2 Conference on Realigning Research and Practice in Information Systems Development: The Social and Organisational Perspective*, Boise, Idaho.

Allen, D., Kern, T., & Havenhand, M. (2002). ERP critical success factors: An exploration of the contextual factors in public sector institutions. *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02),* Big Island, HI (Vol. 8).

Alter, S. (2002). *Information systems: A management perspective.* Addison-Wesley Educational Publishers.

Anderson, D., Johnson, R., & Milligan, B. (1999). *Strategic planning in Australian universities.* Department of Education, Training and Youth Affairs: 33.

Andersson, T., & von Hellens, L. (1997). Information systems work quality. *Information and Software Technology, 39*, 8337-8844.

AVCC. (1996). *Exploiting information technology in higher education: An issues paper.* Canberra, Australian Vice-Chancellors Committee.

Beekhuyzen, J. (2001). *Organisational culture and enterprise resource planning (ERP) systems implementation.* School of Computing and Information Technology, Brisbane, Griffith University.

Beekhuyzen, J., Goodwin, M., & Nielsen, J. L. (2002). ERP in universities: The Australian explosion. *Proceedings of the 13th Australasian Conference on Information Systems (ACIS)*, Melbourne, Australia.

Benbasat, I., Goldstein, D. K., & Mead, M. (1987, September). The case research strategy in studies of information systems. *MIS Quarterly*.

Brown, C., & Vessey, I. (1999). ERP implementation approaches: Toward a contingency framework. *Proceedings of the International Conference on Information Systems (ICIS)*, Brisbane, Australia.

Davenport, T. H. (1998). Putting the enterprise in enterprise system. *Harvard Business Review, 76*(4), 121-131.

DeLone, W. H., & McLean, E. R. (1992). Information systems success: The quest for the dependent variable. *Information Systems Research, 3*(1), 60-95.

DeLone, W. H., & McLean, E. (2003). The DeLone and McLean model of information systems success: A ten-year update. *Journal of Management Information Systems, 19*(4), 9-30.

EMBPWG. (1998). *Best practices in enterprise management*. MERIT Advisory Council.

Eriksson, I., & Törn, A. (1991). A model of IS quality. *Software Engineering Journal*, 152-158.

Eriksson, I., & Törn, A. (1997). Introduction to IST. Special Issue on Information System Quality, *Information and Software Technology, 39*, 797-799.

Esteves, J., & Pastor, J. (2001). Enterprise resource planning systems research: An annotated bibliography. *Communications of the Association for Information Systems, 7*(8), 1-52.

Furumo & Pearson. (2004). ERP case study: A success and a failure. A case study of ERP implementation in two public universities: Why was one a success and the other a failure? *Proceedings of the Tenth Americas Conference on Information Systems*, New York.

Gable, G. G. (1998). Large package software: A neglected technology? *Journal of Global Information Management, 6*(3), 3-4.

Garvin, D. A. (1984). What does product quality really mean? *Sloan Management Review, 26*, 25-43

Gefen, D. (2000). Lessons learnt from the successful adoption of an ERP: The central role of trust. In S. H. e. a. Zanakes (Ed.), *Decision making: Recent developments and worldwide applications* (pp. 17-30). The Netherlands: Kluwer Academic Publishers.

Guthrie, J., & Neumann, R. (2001). The corporatisation of research in Australian higher education. *Symposium 2001 on The University in the New Corporate World*, University of South Australia, City East Campus, Adelaide, Australia.

Hore, T., & Barwood, B. (1989). Strategies for improving access. *Australian Universities Review, 32*(1), 2-4.

Joshi, K., & Rai, A. (2000). Impact of the quality of information products on information system users' job satisfaction: An empirical investigation. *Information Systems Journal, 10*, 323-345.

Klaus, K., Gable, G., & Rosemann, M. (2000). What is ERP? Information systems frontiers. *Special Issue of on the Future of Enterprise Resource Planning Systems, 2*(2), 141-162.

Koch, C., Slater, D., & Baatz, E. (1999). The ABCs of ERP. *CIO Magazine.* Retrieved December 2, 2005, from http://www.cio.com/forums/erp/edit/122299_erp_content.html

Lindroos, K. (1997). Use quality and the World Wide Web. *Information and Software Technology, 39*, 827-836.

Markus, L. (1983). Power, politics and MIS implementation. *Communications of the ACM, 26*(8).

Markus, M. L., & Tanis, C. (2000). The enterprise system experience — from adoption to success. In R.W. Zmud (Ed.), *Framing the domains of IT research: Glimpsing the future through the past*. Cincinnati, OH: Pinnaflex Educational Resources.

Meredyth, D., & Thomas, J. (1996). New technology and the university: Introduction. *Australian Universities Review, 39*(1), 10-11.

Mitev, N. (2000). Toward social constructivist understandings of IS success and failure: Introducing a new computerized reservation system. *Proceedings of the International Conference on Information Systems (ICIS)*, Brisbane, Australia.

Neuman, L. W. (1997). *Social research methods: Qualitative and quantitative approaches.* Needlam Heights, MA: Allyn and Bacon.

O'Leary, D. E. (2000). *Enterprise resource planning systems: Systems, life cycles, electronic commerce, and risk.* New York: Cambridge University Press.

O'Leary, D. E. (2000). Game playing behaviour in requirements analysis, evaluation, and system choice for enterprise resource planning systems. *Proceedings of the 21st International Conference on Information Systems.*

Oliver, D., & Romm, C. (2000). ERP systems: The route to adoption. *Proceedings of the Americas Conference on Information Systems*, Long Beach, California.

Pawlowski, S. D., Robey, D., & Raven, A. (2000). Supporting shared information systems: Boundary objects, communities and brokering. *Proceedings of the International Conference on Information Systems (ICIS)*, Brisbane, Australia.

Salmela, H. (1997). From information systems quality to sustainable business quality. *Information and Software Technology, 39*, 819-825.

Sarker, S., & Lee, A. S. (2000). Using a case study to test the role of three key social enablers in ERP implementation. *Proceedings of the International Conference on Information Systems (ICIS)*, Brisbane, Australia.

Slade, A. J., & Bokma, A. F. (2001). Conceptual approaches for personal and corporate information and knowledge management. *Proceedings of the 32nd Hawaii International Conference on System Science*, Hawaii, HI.

Slater, D. (1999, February 15). An ERP package for you. and you. and you. and even you. *CIO Magazine.*

Swanson, E. B. (1997). Maintaining IS quality. *Information and Software Technology, 39*, 845-850.

Vidgen, R., Wood-Harper, T., & Wood, R. (1993). A soft systems approach to information systems quality. *Scandinavian Journal of Information Systems, 5*, 97-112.

von Hellens, L. (1997). Information systems quality versus software quality. A discussion from a managerial, an organisational and an engineering viewpoint. *Information and Software Technology, 39*, 801-808.

von Hellens, L., Nielsen, S. H., & Beekhuyzen, J. (2005). *Qualitative case studies on implementations of enterprise wide systems.* Hershey, PA: Idea Group Publishing.

Wagner, E., & Scott, L. (2001). *Unfolding new times: The implementation of enterprise resource planning into academic administration*. London: London School of Economics. Retrieved December 2, 2005, from http://is.lse.ac.uk/wp/pdf/WP98.pdf

Yakovlev & Anderson (2000, July/August). Lessons from an ERP implementation. *IT Professional*.

Yetton, P. (1997). *Managing the introduction of technology in the delivery and administration of higher education.* Canberra, Department of Employment, Education, Training and Youth Affairs**.**

Yin, R. K. (1994). *Case study research: Design and methods.* Thousand Oaks, CA: Sage Publications.

# About the Authors

**Evan W. Duggan** is an associate professor of management information systems in the Culverhouse College of Commerce & Business Administration at the University of Alabama USA). He obtained a PhD and MBA from Georgia State University and a BSc from the University of the West Indies, Jamaica. He has more than 25 years of IT experience in industry up to the position of director of information services at a multinational corporation. His research interests involve the management of information systems (IS) in corporations with particular reference to IS success factors and quality, sociotechnical issues, and systems management and delivery methodologies. Dr. Duggan has published in several major journals including the *International Journal of Industrial Engineering*, *Journal of International Technology and Information Management*, *Information Technology & Management*, *Journal of End User Computing*, *Information Resources Management Journal*, *Human-Computer Interactions Information & Management, Electronic Journal of Information Systems in Developing Countries*, and *Communications of the Association of Information Systems*. Dr. Duggan has taught a variety of MIS and decision sciences courses at the graduate and undergraduate levels, including in executive MBA programs in several U.S. and international institutions.

**Johannes P.M. (Han) Reichgelt** is an associate professor in information technology at Georgia Southern University (USA). He received his first BSc and master's from the University of Nijmegen in The Netherlands and his PhD in cognitive science from the University of Edinburgh in Scotland. Prior to joining Georgia Southern University, he was research fellow in the Department of Artificial Intelligence at the University of Edinburgh, lecturer in psychology at the University of Nottingham, and professor of computer science at the West Indies, Mona, Jamaica. Dr. Reichgelt's research interests include IT and economic development, business process modeling, computing education and accreditation and assessment, and he has published in several major journals, including *Artificial Intelligence*, *Journal of Automated Reasoning, Notre Dame Journal of Formal Logic, Information Technology for Development, Communications of the AIS,* and *Journal for IT Education.* Dr Reichgelt is also the author of a textbook on knowledge representation in artificial intelligence. Dr. Reichgelt has taught a range of courses in cognitive science, computer science, and information systems at both the undergraduate and the undergraduate level. He currently serves as chair of ACM Special Interest Group on IT Education (SIGITE).

* * *

**Alain April** is an associate professor of software engineering at the École de technologie supérieure – ÉTS, Université du Québec, Montréal, Canada. He is pursuing a doctorate at the Otto von Guericke University of Magdeburg, Germany. His research interests are software maintenance, software quality, and multimedia database management systems. He has worked in the IT industry for more than 20 years in the telecommunications industry. Professor April has contributed to the internal measurement of software of ISO9126 (part 3) and is associate editor of the SWEBOK (Software Engineering Body of Knowledge) software maintenance and quality chapters that have recently been accepted as an ISO/IEC Technical Report #19759.

**Jenine Beekhuyzen** is a senior research assistant at Griffith University (Australia) and works in a Cooperative Research Centre on Smart Internet Technology. She serves as a reviewer for several information systems academic journals and conferences and is the Program Chair of the annual QualIT – Qualitative Research in IT conference held each November. She is currently the president of the School of ICT Alumni Association and the chair of the Qld Girls and ICT State Reference Group. Jenine's recent projects for the Smart Internet Technology include an investigation of Australia's online banking trends and barriers, the policy issues of digital rights management (DRM), and the development of mobile computing to improve patient care within the Australian health sector. Jenine's research also explores the issues facing particular disadvantaged groups: women, people with disabilities, young/elderly people, indigenous people, and small and medium enterprises (SMEs). Her many research projects include the WinIT (Women in IT) longitudinal study.

**Eleni Berki** has been a researcher/principal investigator at the Information Technology Research Institute and assistant professor of Group Technologies, in Jyväskylä University, Finland. She completed her PhD in process metamodeling and IS method engineering in 2001, in the UK. Her teaching and research interests include virtual communities, information and communication technologies, multidisciplinary approaches for software engineering, knowledge representation frameworks and requirements engineering. She has worked as a systems analyst, designer, and IS quality consultant in the industry, and is involved in many academic and industrial international projects. She is a professional member of the IEEE, the BCS, the UK Higher Education Academy, and a visiting lecturer and researcher in European universities.

**Rosann Webb Collins** is an associate professor of information systems and decision sciences at the University of South Florida (USA). Her research focuses on the impact of IT on knowledge work, systems development, community informatics, and the legal and ethical issues in the use of computers. Her publications include a book on the deployment of global IT solutions and articles in *MISQ*, *The Information Society*, the *Journal of the American Society for Information Science*, the *Journal of Research on Computing in Education*, *Educational Technology*, *International Library Review, ISR,* and *DATA BASE*.

**Robert Cox** (BAppComp., BIS, Hons; FASCP; MAIS; IADIS) obtained a bachelor's degree in applied computing and graduated with an honours degree in information systems in 2001, which led to his invitation to join the Smart Internet Technology Collaborative Research Centre (SITCRC) as a PhD candidate. His research involves investigating how technologically mediated communication transforms emotive communicative practice through the exploration of the relationships between the roles of content and context in mediated interactions. A fellow of the Australian Society of Computing Professionals since 1994, Robert has been a tutor and lecturer at the University of Tasmania (Australia) for the past four years. He is a current member of the Australasian Chapter of the Association for Information Systems and the International Association for the Development of the Information Society. He is the Webmaster and a consulting editor of the *Australasian Journal of Information Systems* and a consistent reviewer for PACIS (Pacific Asia Conference for Information

Systems), ACIS (Australian Conference for Information Systems), and WICC (Women in Computing Conference).

**Joseph A. De Feo** is president and chief executive officer of Juran Institute Inc. (USA). He holds a BS degree in industrial education from Central Connecticut State College and earned an MBA from Western Connecticut State University. Mr. De Feo is recognized for his training and consulting experience in managing quality within many different organizations worldwide. His areas of expertise include training executive level managers in developing and deploying breakthrough management principles, Six Sigma, strategic quality planning, and business process improvement methodologies. Mr. De Feo's research is published widely in many international publications such as *Assembly Engineering, CIO Magazine, Industrial Management, Ivey Business Journal, MWorld Magazine, Pharmaceutical Processing, Productivity Digest, Quality Digest, Quality Progress and Quality in Manufacturing*, and others. He is co-author of *Juran Institute's Six Sigma: Breakthrough and Beyond (McGraw-Hill, 2004). He is* a frequent guest speaker at international conferences sponsored by organizations such as the American Society for Quality, Association for Manufacturing Excellence, the Conference Board, MIT Sloan School of Management, WCBF, Oracle, and the International Quality and Productivity Center. He has also been a guest lecturer at Columbia University and New York University, and he serves on the advisory boards of the Joseph M. Juran Center for Leadership in Quality at the University of Minnesota's Carlson School of Management and the American Society for Quality *Six Sigma Forum* magazine.

**R. Geoff Dromey** is the foundation professor of software engineering in the School of Computing and Information Technology at Griffith University in Brisbane, Australia. He is the founder and director of the Software Quality Institute. He has also been a founder of a successful software company. His current research interests are in development methods that control complexity in the analysis/design of large-scale systems. He has authored two books in leading international computer science series and authored/co-authored more than 50 research papers. He serves on the editorial boards of three international journals. Professor Dromey received a PhD in chemical physics from La Trobe University. He is a member of the IEEE and ACM.

**Richard Gibson** has been trained and authorized by the Software Engineering Institute to lead teams conducting appraisals of integrated systems/software organizations. He is also an ASQ certified software quality engineer. As an associate professor at American University, his teaching and research are focused on global software and systems engineering with a particular interest in process improvement initiatives.

**Gina C. Green** is an associate professor of information systems at Baylor University (USA). Her research interests include the diffusion of innovations, database design, project management, and complexity in computer-supported work. Her research has appeared in journals such as the *Information Resources Management Journal*, *IEEE Software*, the *Information and Software Technology Journal*, the *Journal of High Technology Management*, the *Journal of Information Systems Education*, and others. She is a member of ACM, SIGMIS, and DSI.

**Alan R. Hevner** is an eminent scholar and professor in the Information Systems and Decision Sciences Department, University of South Florida (USA). He holds the Citigroup/Hidden River chair of distributed technology. Dr. Hevner's areas of research interest include software engineering, health care information systems, distributed database systems, and telecommunications. He has over 125 published research papers on these topics. He has consulted for a number of organizations, including IBM, Honeywell, Cisco, and AT&T. Dr. Hevner is a member of ACM, IEEE, AIS, and INFORMS.

**Mike Holcombe** is a professor of computer science at the University of Sheffield (UK). He was head of Department 1987-1993 and dean of the Faculty of Engineering 1999-2002. He is director of the Verification and Testing Research Laboratory and co-chair of the company Genesys Solutions. He has published around 160 research papers and seven research books in mathematics, software engineering, and biology. He is currently in receipt of around £3m in research grants in software engineering and computational biology. For more information, visit http://www.dcs.shef.ac.uk/~wmlh.

**Catherine Horiuchi**, DPA, Assistant Professor at the University of San Francisco (USA), received her doctorate in public administration from the University of Southern California (2001). She received her BA in Latin and her MA in linguistics from the University of Utah. She has taught courses in statistics, research methods, policy analysis, systems analysis, and information technology, including e-government. She has authored several chapters and articles on these topics and others, particularly energy policy. As a consultant, she has applied her analytic skills and knowledge base toward solving numerous management problems for government and private sector clients.

**Jakob Holden Iversen** holds an MSc in software engineering and a PhD in computer science from Aalborg University, Denmark. He is currently an assistant professor of MIS at the University of Wisconsin Oshkosh (USA). His research focuses on agile software development and software process improvement and measurement. He is co-author of *Improving Software Organizations: From Principles to Practice*.

**Julie E. Kendall**, PhD, is an associate professor of e-commerce and information technology in the School of Business-Camden, Rutgers University (USA). Dr. Kendall is the chair of IFIP Working Group 8.2. She was awarded the Silver Core from IFIP. Dr. Kendall's research has been published in *MIS Quarterly*, *Decision Sciences*, *Information & Management, Organization Studies,* and many other journals. Additionally, Dr. Kendall has recently co-authored a college textbook with Kenneth E. Kendall, *Systems Analysis and Design* (6th edition, Prentice Hall). She is a senior editor for *JITTA*, and Dr. Kendall is on the editorial review boards of the *International Journal of e-Collaboration*; the *Decision Sciences Journal of Innovative Education;* the *Journal of Database Management;* the *Journal of Cases on Information Technology,* and the *Information Resource Management Journal*. She served on the inaugural editorial board of the *Journal of AIS* and as an associate editor for *MIS Quarterly.* Julie served as treasurer and vice president for the Decision Sciences Institute. For more information, visit www.thekendalls.org.

**Kenneth E. Kendall**, PhD, is a professor of e-commerce and information technology in the School of Business-Camden, Rutgers University (USA). He is one of the founders of the International Conference on Information Systems (ICIS) and a fellow of the Decision Sciences Institute. He is an associate editor for the *International Journal of Intelligent Information Technologies*, and he is on the senior advisory board of *JITTA;* a member of the editorial board of *Journal of IT for Development* and is on the editorial review board of the *Decision Sciences Journal of Innovative Education*. Dr. Kendall has been named one of the top 60 most productive MIS researchers in the world, and he was awarded the Silver Core from IFIP. He recently co-authored a text, *Systems Analysis and Design* (6th edition, Prentice Hall) and *Project Planning and Requirements Analysis for IT Systems Development* (2nd edition), and edited *Emerging Information Technologies: Improving Decisions, Cooperation, and Infrastructure* (Sage Publications). For more information, visit www.thekendalls.org.

**Heinz D. Knoell** is currently a visiting scholar at the UCLA Anderson Graduate School of Management where he is pursuing a research project on the outcomes of ERP systems. He has been professor of information systems at the School of Business of the University of Lueneburg, Germany, since 1986, and he served as a visiting professor at the School of Computing and Information Technology, University of Wolverhampton, UK. In addition to ERP systems his research interests include software project management, executive information systems, and software quality assurance. Dr. Knoell received his PhD from the University of Muenster, Germany. He is a member of the AIS and the German Chapter of the ACM.

**Sue Kong** is a PhD candidate in Rutgers Business School, Newark (USA) and New Brunswick. She is currently writing her dissertation which is related to organizational adoption and adaptation of agile software development methodology. Sue's research interests include alternative system development methodologies, software quality management, applied aspects of information technology (especially Web site design and analysis), e-commerce, and information system security. She is a member of AIS and DSI.

**Claude Y. Laporte** is a professor at the École de technologie supérieure (ÉTS) (Canada), an engineering school, where he teaches graduate and undergraduate courses in software engineering. His interests include the development and deployment of software and systems engineering processes, process assessment, software quality, and the management of technological change. He has worked in industry for more than 25 years mainly in the defense sector and also in the transportation sector. Professor Laporte is the co-project editor of *ISO/IEC TR 19759 Software Engineering - Guide to the Software Engineering Body of Knowledge* (SWEBOK) and the editor of an ISO/IEC-SC7 working group tasked to develop software life cycle profiles and guidelines for use in very small enterprises.

**Francisco Macias** (Francisco.Macias@itesm.mx) received a BEng in geology in 1988 from the National Polytechnic Institute (Mexico), an MSc in computer science in 1993 from the National University of Mexico, and a PhD degree in computer science from the University of Sheffield (UK). He worked as programmer at Olivetti and also as lecturer at the Technologic of Monterrey (Mexico), where he is currently working on software quality.

**Lars Mathiassen** holds an MSc in computer science from Århus University, Denmark, a PhD in informatics from Oslo University, Norway, and a DrTechn in software engineering from Aalborg University, Denmark. He is currently a Georgia Research Alliance Eminent Scholar and Professor in Computer Information Systems at Georgia State University (USA). His research interests focus on engineering and management of IT systems. More particularly, he has worked with project management, object-orientation, organizational development, management of IT, and the philosophy of computing. He is a co-author of *Professional Systems Development: Experiences, Ideas and Action, Computers in Context: The Philosophy and Practice of Systems Design, Object-Oriented Analysis & Design,* and *Improving Software Organizations: From Principles to Practice*.

**Peter Axel Nielsen** holds an MSc in computer science from Århus University, Denmark, and a PhD in information systems from Lancaster University, UK. He is currently associate professor in computer science at Aalborg University, Denmark. He is also a visiting research professor with Agder University College, Norway. His research interests include action research on software development practice, object-oriented methodologies, and social and organizational aspects of software development. He is a co-author of *Object-Oriented Analysis & Design* and *Improving Software Organizations: From Principles to Practice*.

**Sameer Verma** is an assistant professor of information systems at San Francisco State University (USA). His research focuses on the diffusion and adoption of technology-based innovations. His academic research projects include the diffusion and adoption of open source software, mobility, and management in wireless networks, and communication modes in messaging systems. In addition to his academic work, Dr. Verma has worked with several companies in a consulting capacity in the areas of content analysis, management, and delivery. He is a founding member of Airgram Networks, a wireless networks management company that runs exclusively on open source software. Dr. Verma also serves on the advisory boards of some San Francisco Bay Area technology companies. He holds a Bachelor of Engineering from Osmania University, Hyderabad, India and a PhD in business administration from Georgia State University, Atlanta, GA.

**Bernard Wong** is a senior lecturer in the Faculty of Information Technology at the University of Technology, Sydney in Australia. He holds a PhD in software engineering, a BSc in computer science, and a MCom in information systems. He is a certified quality analyst (QAI) and a certified software quality assessor (ISO9000 Auditor). Since 1991, Dr. Wong has been engaged in teaching, research, and consulting in the fields of software quality assurance, software engineering, project management, and information systems. During this time he has established new courses in project management and software quality assurance.  He has supervised many postgraduate research students and has been a member of many conference and workshop program committees. Dr. Wong has more than 20 years of industry experience, where he has worked as a programmer, an analyst/programmer, a systems analyst, a business analyst, a Project manager, and as an IT trainer. He has consulted to companies large and small, in both the public and private sectors. The commercial exposure has been extremely important to his academic contribution. Not only has it supplied him with many case studies, essential for relevant teaching, it has also been invaluable as a source to his research.

# Index