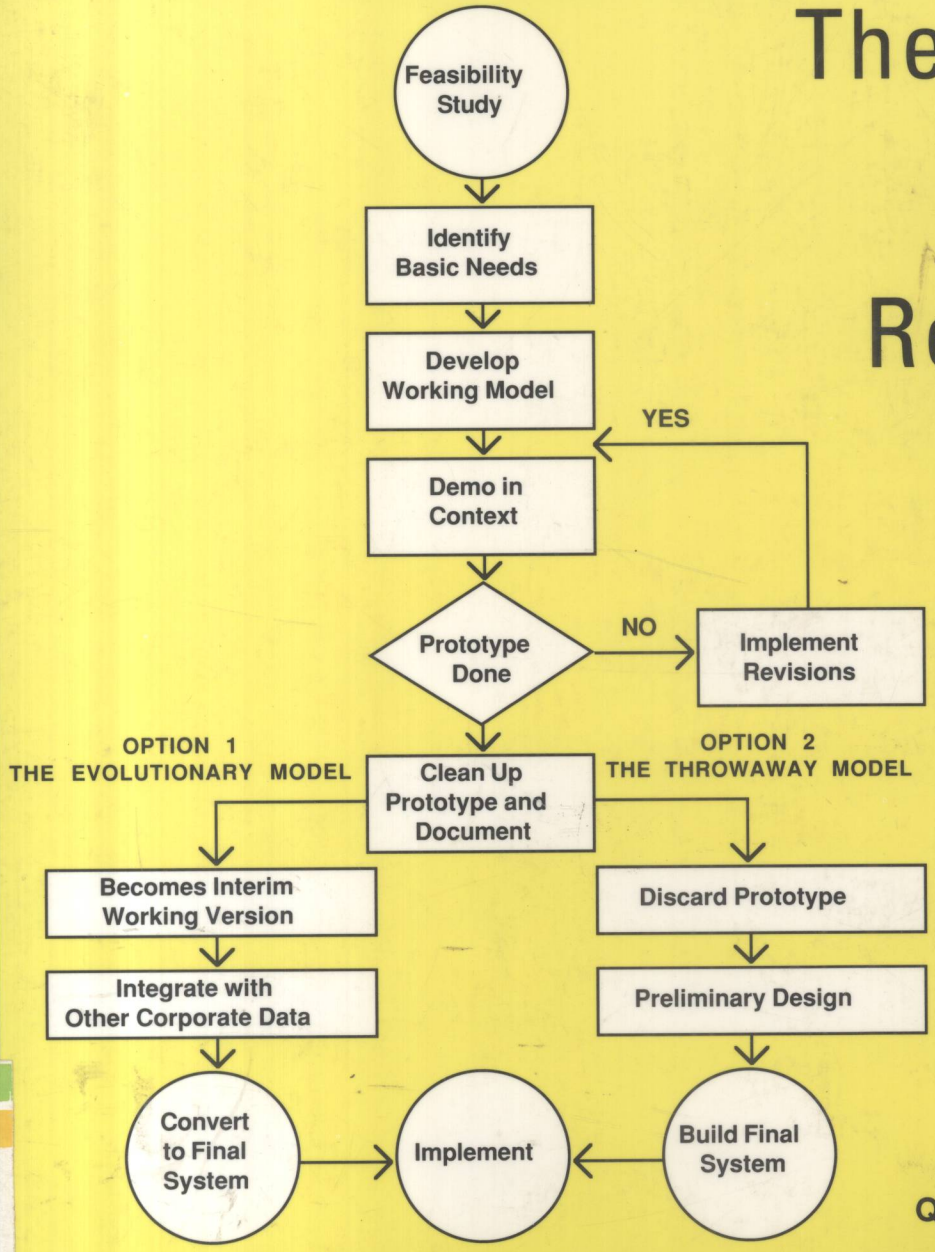
Second Edition

RAPID APPLICATION PROTOTYPING

The Storyboard Approach to User Requirements Analysis



Stephen J. Andriole, Ph.D.

QED Technical Publishing Group

RAPID APPLICATION PROTOTYPING

The Storyboard Approach to User Requirements Analysis

STEPHEN J. ANDRIOLE

Second Edition

QED Technical Publishing Group
Boston • Toronto • London

P.O. Box 82-181
Wellesley, MA 02181

QED Technical Publishing Group is a division of
QED Information Sciences, Inc.

Library of Congress Catalog Number: 91-18242
International Standard Book Number: 0-89435-403-5

Printed in the United States of America
92 91 90 10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data

Andriole, Stephen J.
Rapid application prototyping : the storyboarding approach to user requirements analysis.
p. cm.
Bibliography: p.
Includes index.
ISBN 0-89435-403-5
1. Computer software—Development. 2. System design. I. Title.
QA76.76.D47A43 1991
005.1—dc19

RAPID APPLICATION PROTOTYPING

The Storyboard Approach to User Requirements Analysis

Books from QED

Database

- Migrating to DB2
- DB2: The Complete Guide to Implementation and Use
- DB2 Design Review Guidelines
- DB2: Maximizing Performance of Online Production Systems
- Embedded SQL for DB2: Application Design and Programming
- SQL for DB2 and SQL/DS Application Developers
- Using DB2 to Build Decision Support Systems
- The Data Dictionary: Concepts and Uses
- Logical Data Base Design
- Entity-Relationship Approach to Logical Data Base Design
- Database Management Systems: Understanding and Applying Database Technology
- Database Machines and Decision Support Systems: Third Wave Processing
- IMS Design and Implementation Techniques
- Repository Manager/MVS: Concepts, Facilities and Capabilities
- How to Use ORACLE SQL*PLUS
- ORACLE: Building High Performance of Online Systems
- ORACLE Design Review Guidelines
- Using ORACLE to Build Decision Support Systems
- Understanding Data Pattern Processing: The Key to Competitive Advantage
- Developing Client/Server Aplications in an Architected Environment

Systems Engineering

- Quality Assurance for Information Systems: Methods, Tools, and Techniques
- Handbook of Screen Format Design
- Managing Software Projects: Selecting and Using PC-Based Project Management Systems
- The Complete Guide to Software Testing
- A User's Guide for Defining Software Requirements
- A Structured Approach to Systems Testing
- Storyboard Prototyping: A New Approach to User Requirements Analysis
- The Software Factory: Managing Software Development and Maintenance
- Data Architecture: The Information Paradigm
- Advanced Topics in Information Engineering
- Software Engineering with Formal Metrics

Management

- Introduction to Data Security and Controls
- CASE: The Potential and the Pitfalls

Management (cont'd)

- Strategic and Operational Planning for Information Services
- Information Systems Planning for Competitive Advantage
- How to Automate Your Computer Center: Achieving Unattended Operations
- Ethical Conflicts in Information and Computer Science, Technology, and Business
- Mind Your Business: Managing the Impact of End-User Computing
- Controlling the Future: Managing Technology-Driven Change
- The UNIX Industry: Evolution, Concepts, Architecture, Applications, and Standards

Data Communications

- Designing and Implementing Ethernet Networks
- Network Concepts and Architectures
- Open Systems: The Guide to OSI and its Implementation

IBM Mainframe Series

- CSP: Mastering Cross System Product
- CICS/VS: A Guide to Application Debugging
- MVS COBOL II Power Programmer's Desk Reference
- VSE COBOL II Power Programmer's Desk Reference
- CICS Application and System Programming: Tools and Techniques
- QMF: How to Use Query Management Facility with DB2 and SQL/DS
- DOS/VSE: Introduction to the Operating System
- DOS/VSE: CICS Systems Programming
- DOS/VSE/SP Guide for Systems Programming: Concepts, Programs, Macros, Subroutines
- Advanced VSE System Programming Techniques
- Systems Programmer's Problem Solver
- VSAM: Guide to Optimization and Design
- MVS/JCL: Mastering Job Control Language
- MVS/TSO: Mastering CLISTS
- MVS/TSO: Mastering Native Mode and ISPF
- REXX in the TSO Environment

Programming

- C Language for Programmers
- VAX/VMS: Mastering DCL Commands and Utilities
- The PC Data Handbook: Specifications for Maintenance, Repair, and Upgrade of the IBM/PC, PS/2 and Compatibles
- UNIX C Shell Desk Reference

QED books are available at special quantity discounts for educational uses, premiums, and sales promotions. Special books, book excerpts, and instructive materials can be created to meet specific needs.

This is Only a Partial Listing. For Additional Information or a Free Catalog contact

QED Information Sciences, Inc. • P. O. Box 82-181 • Wellesley, MA 02181

Telephone: 800-343-4848 or 617-237-5656 or fax 617-235-0826

For Emily Brett, the youngest one . . .

Acknowledgments

This second edition of *Storyboard Prototyping* (now called *Rapid Application Prototyping*) is the product of many years of thinking about how to identify and validate exactly what it is systems are supposed to do for their users. We began building screen displays of functional capabilities in the early 1970s. It was much harder then than now because of the absence of special-purpose and off-the-shelf tools. Today, of course, things are very different. There are many tools available for storyboard prototyping. The first acknowledgment thus goes to the community of software developers and vendors who have provided designers with a set of extremely powerful and versatile tools. I would also like to acknowledge the colleagues—too numerous to mention individually—with whom I have worked over the years designing, developing, and demonstrating interactive storyboards. I would like to thank my students at George Mason University and, more recently, at Drexel University for their insights into storyboarding.

On a more personal level, I would like to thank my wife of over twenty years, Denise, for her legendary understanding. Her love and support make it possible for me to pursue my vocations and avocations.

Finally, the kids deserve special mention. Katherine and Emily have grown accustomed to seeing me on the Macintosh. While they have no idea what I really do on the computer, they know that for me it is fun. They don't see it as competition for them but rather as Daddy's toy, and they have—even at their tender ages—shown the kind of understanding and patience that not every adult I've met has mastered. Thanks, kids, for not making me feel guilty—at least not that often.

All errors, omissions, and other technical crimes are entirely mine.

SJA
Bryn Mawr,
Pennsylvania

Introduction

Software engineering—if the term can even be used to describe the process by which we convert user requirements into executable code—is not cost-effective. Studies suggest that we spend more on modifications and maintenance than we do on the initial design of even simple software systems. A variety of explanations for this sad state of affairs come to mind: computer science is not really a science but much more of an art . . . users have a hard time articulating their needs . . . inappropriate hardware configurations are often foisted upon systems analysts . . . we don't pay enough . . .

The fact remains that a large percentage of our software systems simply don't work very well. They often don't do what they were intended to do, more often than not cost far more than planned, and almost always take longer to design and develop than even the most cynical pessimist anticipates. Users, by and large, are unhappy with their computing environments. They are usually forced to learn elaborate routines just to initialize simple programs. They are expected to be adaptive and creative. They are also expected to be endlessly patient for proverbial "enhancements." Documentation is often poor or utterly incomprehensible.

What is going on? It is safe to say that the origins of modern computing cannot be traced to "user friendliness." When it all began years ago, computer software, much like an about-to-retire information systems analyst, was cranky. It was written by scientists for scientists. Complicated command languages, inexplicable menu structures, and slow response time were the rule rather than the exception, but no one really minded because computers were specialized tools intended for use by a select few—a group honored to have the opportunity to work with the new electronic calculators. Then, of course, technology provided us with time-sharing, distributed databases, and personal computers. The world of computing would never be the same.

It is possible that our failure to appreciate non-technical users is anchored in the origins of modern computing. The software industry itself is largely unprepared to respond to complicated user requirements, having instead concentrated for years on the design and development of embedded software or software that had only limited contact with human operators. The current push in computer-aided software engineering—or CASE—is representative of this emphasis. We have nearly perfected the implementation of structured methods for software specification, but have spent relatively little on the acquisition of system-level and user-functional requirements.

Relatedly, many software engineers who began their careers in the 1960s do not fully appreciate the need for flexible, friendly interfaces or polite dialogue. Their expertise lies in the design and development of algorithms for handling databases or in managing scientific computations, not in the design of elegant interfaces.

When computers were used by experts for esoteric purposes, and when requirements were invented along the way, the need for tools and techniques for requirements validation was small. Today, nearly all of the leverage lies in front-end analyses, in accurate requirements definitions. This book—the second edition—seeks such leverage.

In the 1970s, when we failed to capture user requirements the first time through, we simply modified the program (and modified the program, and modified the program, and . . .) until we got it right. Users waited while systems analysts reviewed their requirements data, and managers cringed when programming teams were put back to work again and again (and again and again . . .). By the 1980s, we recognized the recalcitrance of user requirements and ushered in the era of "rapid prototyping."

Prototyping is a euphemism for failure. Prototyping legitimizes failure. It also presumes that requirements will evolve over time and that the programming of executable code should lag behind the validation of requirements. But the key to successful prototyping is cost-effectiveness. If your first and second (and third and fourth . . .) prototypes are expensive, then there will be little left for the "real" system. If your early prototypes consume months or years, then your users—your *clients*—will forget your commitment to their problem. Prototypes must be fast and cheap. This book suggests how to accomplish these goals.

This is the second edition of *Storyboard Prototyping*. Over the past few years, we have continued to apply the storyboard prototyping process successfully. We have—as the new appendices suggest—begun to use the technique to validate the requirements of some unconventional systems. We have improved the process by which system concepts—represented in storyboard prototypes—are "sized," or scaled up to full-fledged working systems. We have also identified a much larger set of off-the-shelf tools that support storyboard prototyping.

The book argues faithfully that before *software* requirements (expressed in data flow diagrams, entity-relationship diagrams, and state transition charts, among other forms) can be specified, *user* requirements must be defined. Most articles and books on requirements analysis ignore this critical distinction and concentrate exclusively on user requirements.

Motivation for both editions of the book came from several sources. First and foremost, its origins can be traced to a number of system failures we experienced in the 1970s. Many of our information and decision support systems simply went unused. When we conducted system post-mortems, we discovered that although we developed "formal" requirements definitions, they were frequently validated not by users but by programmers—a fatal flaw in any systems design process. We also tried to protect as much time and money from the requirements analysis process as possible, so that project resources would be adequate for programming. (Later we discovered that the more you spend on requirements the less you need to spend on programming.) We also had a weak inventory of requirements analysis tools, especially software tools.

Motivation can also be traced to our desire to find a better way. Over the years, we have experimented with several approaches, tools and, techniques. This book reports on one—storyboarding—that has paid huge dividends.

The book is intended to be practical. Substantial parts have been designed to communicate with practitioners. Other parts offer some food for applied thought. There is very little in the book that is purely theoretical or abstract. At the same time, theoretical debts are easy to find, especially as they pertain to the use of generic task and user taxonomies. In short, the book tries to explain an approach to requirements analysis and system sizing that has worked in the trenches.

The book is organized into five chapters and several important appendices. Chapter 1 describes some modern systems analysis challenges, noting how requirements have evolved from static and data-intensive to dynamic and analytical. Chapter 2 reviews the conventional systems design process and replaces it with one that relies strongly upon prototyping. Chapter 3 describes the prototyping strategy in detail, concentrating on requirements analysis methods and on how requirements can be modeled. Chapter 4 details the use of the storyboarding technique for rapid prototyping. It also describes the approach to "system sizing" that we have adopted. Chapter 5 gets a little philosophical about storyboarding: what are its strengths and weaknesses? where is the technique heading?

The case studies in the appendices are drawn from varied experiences in systems design. A cross-section of applications is described to suggest just how

flexible the technique is, that even in the design of physical interfaces—such as for the operation of electronic equipment—the technique can be powerful. The first case study deals with a resource allocation problem, the second with an information retrieval problem. The third case—by Peggy Brouse—turns to knowledge-based health care claims processing, while the fourth—by Hal Gumbert, Brian Magee, and Greg Senior—looks at how storyboard prototypes can be used to design a dental information system and how such concepts can evolve from functional hierarchies into data flow and entity-relationship diagrams through storyboarding. The fifth case study—by Peter Aiken and Kim Madsen—examines the design for an interface to a piece of electronic equipment. It is a relatively unconventional application of the technique but one that suggests the versatility of storyboard prototyping.

This second edition of *Storyboard Prototyping* refines the approach via the presentation of some new storyboards and—especially—the identification of a new set of off-the-shelf tools for rapid storyboard prototyping. These two major changes will, we hope, make storyboarding even more useful.

Contents

Figures

Tables

CHAPTER 1 Modern Systems Design and Development in Perspective

Where did we go wrong? Why do so many of our interactive computer-based systems fail to satisfy user requirements? The short answer is that things have changed. No longer are applications limited to well-bounded inventory control problems. No longer are we preoccupied with the design and development of systems to maintain small databases. Today, users expect all kinds of decision support, all kinds of interactive options. Users today want systems to augment their problem-solving capabilities in real time. They want *analytical* support.

Analytical requirements are difficult to identify, define, and validate. Users find it difficult to express analytical requirements. Systems analysts find it difficult to organize analytical requirements for subsequent conversion to software.

The future is not necessarily bright. The analytical challenges to software systems designers are growing in number and complexity, but our methods for capturing analytical requirements are not evolving proportionately. Expectations about real-time software support for strategic battle management in space, corporate crisis management, and personal financial planning are growing as our ability to conduct complex requirements analyses has atrophied. Clearly, there is a need for some new ways to think about requirements analysis, requirements modeling, and the design strategies and tactics that ultimately determine the success of all software projects.

The New Era of Analytical Computing

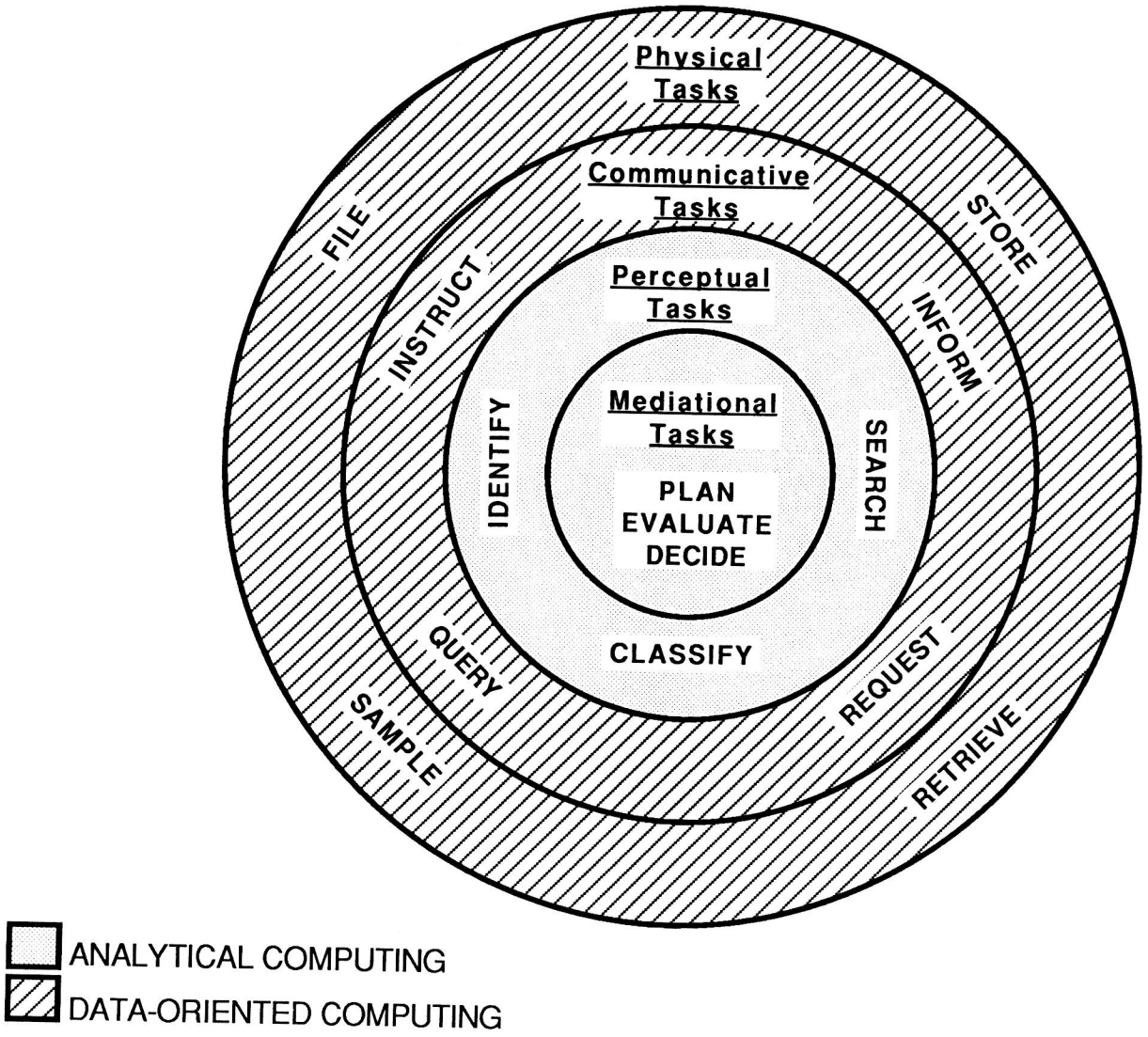
Not so many years ago, computers were used mostly by scientists and engineers. As the field matured, computing was distributed to a larger subset of professionals—accountants, budgeteers, and some managers. The personal computer (PC) altered forever the way we think about computing. Initially, the appeal of desktop power was mitigated by cost, but as soon as PCs became affordable, the revolution in personal computing began.

Years ago, computers were used to perform calculations that were prohibitively expensive via any other means. Early interactive systems were barely so, and engineers had to hack at them until they behaved. When general-purpose mainframes emerged, large organizations with huge databases expressed the most interest. It is safe to say that most early applications of general-purpose mainframe computers were database oriented.

Today there are interactive "decision support systems" that profess to augment the decision-making power of human information processors. There are systems that help users generate options, evaluate options, and interpret the feedback received after options are implemented. There are systems that help users plan strategies, create scenarios, and diagnose diseases.

Figure 1.1 suggests where database-oriented and

Figure 1.1 "Analytical" vs. data-oriented computing.



analytical computing begin and end. The differences are clear. Analytical problem solving assumes some degree of cognitive information processing. While all cognitive processing is anchored in "data" and "knowledge" that must be stored and manipulated, there are unique properties of cognitive information processing that call for unique requirements definitions. The difference between the collection and interpretation of diagnostic data illustrates database-oriented versus analytical problem solving (and, by implication, data base-oriented versus analytical computing).

As computers become cheaper, smaller, and faster, and as expectations about how they can be used rise, more and more instances of "analytical computing" will become necessary and—eventually—commonplace . But we cannot necessarily get there from here. The leverage lies in our ability to identify, define, and validate complex requirements.

Elusive Requirements

Anyone who has conducted a requirements analysis with real users who have complex analytical problems knows how frustrating the requirements analysis pro-

cess can be. Analytical requirements are inherently recalcitrant. A simple example might illustrate some of the important difficulties. Imagine a computing challenge that calls for the design and development of a system capable of assisting managers in their selection of display terminals for the office. The requirements for such a system are clearly analytical. Users of the system would ideally be able to evaluate terminals vis-à-vis a set of weighted criteria; perform "what-if" analyses by varying the importance of criteria; add, delete and combine criteria; and rank-order candidate terminals according to how well (or badly) they scored on the criteria. They might also want to permit judgments about the candidates to be "qualitative" and "quantitative." Any number of requirements might emerge from the requirements analysis.

What else? Over time the list would grow longer and longer. The job of the requirements analyst would, among other tasks, involve defining and distilling these requirements to their most diagnostic properties. Without question, this process would be iterative. Why? Because analytical requirements are nearly impossible to identify and define accurately in a single take, because users are notoriously inarticulate, and because analytical problems are usually ill-structured, and as such, defy definitions that are conveniently well-bounded.

The real world of design constraints also exerts an impact upon the analytical requirements analysis process. Given that few design projects have unlimited funds and flexible schedules, requirements must be prioritized. Users are inevitably confronted with trade-off decisions: "If we give you these capabilities, we cannot give you these. . . . Which are more important?" Judgments about what is possible given time and money constraints are usually made qualitatively. Users can express preferences, but system architects synthesize judgments with information to arrive at a pragmatic system development blueprint. This process of "synthesis," while abstract, defines the essence of the requirements dilemma. When systems analysts are confronted with (a) analytical requirements, (b) stated user preferences, (c) user-identified priorities, and (d) empirical constraints, they often produce constraint-dominated design concepts. These "conservative" designs emerge because constraints are often the most empirical variables in the equation and because it is far easier to approach design via the identification of what cannot be done rather than the integration and synthesis of what can and should be done.

Analytical requirements are elusive because they are complex. It is foolish to assume that they can be easily captured the first time, the second time, or perhaps even the third. It is also important to remember that requirements cannot be captured in a vacuum. Users remain integral parts of the requirements analysis process.

We need some new tools, techniques, and methods to elicit, define, represent, and validate complex analytical requirements. When a planner calls for decision support, when a manager needs to allocate resources optimally, and when a forecaster must predict the future, systems analysts must call upon all the science and art at their disposal. Unfortunately, there is perhaps as much art in requirements analysis as there is science. Really good systems analysts are orders of magnitude better than competent ones, but they often have great difficulty introspecting upon the processes by which they cull requirements from inarticulate users. Much too much of the process—when it goes right—is not reproducible. We have precious little data on the essence of successful requirements definitions.

The Need for New Design Concepts, Methods, and Tools

If projections about the need for analytical computing are accurate, requirements challenges will grow dramatically over the next few years and well into the foreseeable future. We are currently unprepared to deal with the onslaught of requirements problems—problems that will grow in number and complexity as hardware becomes cheaper and user expectations higher.

The most important change must occur in the problem-solving perspective that we bring to the systems design and development process. Conventional models of the design process have proven inadequate to the development of quality software. We are at a crossroads today: one direction is slowly and painfully evolutionary, oriented toward incremental changes in the design process; another direction is more radical, suggesting that the design process should be front-loaded with time, money, and talent. The working assumption must be that there is leverage in the front end of the design process, leverage that can protect a software project from disaster as it matures.