THE ART OF SYSTEMS ARCHITECTING

SECOND EDITION

THE ART OF SYSTEMS ARCHITECTING

SECOND EDITION

Mark W. Maier
Aerospace Corporation
Chantilly, Virginia

Eberhardt Rechtin
University of Southern California
Los Angeles, California

CRC Press
Boca Raton London New York Washington, D.C.

Library of Congress Cataloging-in-Publication Data

The art of systems architecting / edited by Mark W. Maier, Eberhardt Rechtin.—2nd ed.
p. cm.
Pre. ed. entered under Rechtin
Includes bibliographical references and index.
ISBN 0-8493-0440-7 (alk. paper)
1. Systems Engineering. This book is no longer in a series. I Maier, Mark.
Rechtin, Eberhardt. III. Rechtin, Eberhardt. Art of systems architecting.
TA168 .R368 2000
620′.001′171—dc21 00-03948
CIP

No claim to original U.S. Government works
International Standard Book Number 0-8493-0440-7
Library of Congress Card Number 00-03948
Printed in the United States of America 1 2 3 4 5 6 7 8 9 0
Printed on acid-free paper

Dedications

To Leigh, Stuart, and Charlie, my anchors and inspiration.

Mark Maier

To Deedee, for whom life itself is an art.

Eberhardt Rechtin

Preface

The continuing development of systems architecting

> Architecting, the planning and building of structures, is as old as human societies — and as modern as the exploration of the solar system.

So began this book's 1991 predecessor.* The earlier work was based on the premise that architectural methods, similar to those formulated centuries before in civil works, were being used, albeit unknowingly, to create and build complex aerospace, electronic, software, command, control, and manufacturing systems. If so, then other civil works architectural tools and ideas — such as qualitative reasoning and the relationships between client, architect and builder — should be found even more valuable in today's more recent engineering fields. Five years later, at the time of the first edition of this book, judging from several hundred retrospective studies at the University of Southern California of dozens of post-World War II systems, the original premise has been validated. With another five years of perspective the observations only hold more strongly. Today's systems architecting is indeed driven by, and serves, much the same purposes as civil architecture — to create and build systems too complex to be treated by engineering analysis alone.

Of great importance for the future, the new fields have been creating architectural concepts and tools of their own and at an accelerating rate. This book includes a number of the more broadly applicable ones, among them heuristic tools, progressive design, intersecting waterfalls, feedback architectures, spiral-to-circle software acquisition, technological innovation, and the rules of the political process as they affect system design.

Arguably, these developments could, even should, have occurred sooner in this modern world of systems. Why now?

* Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991. Hereafter called Rechtin, 1991.

Architecting in the systems world

A strong motivation for expanding the architecting process into new fields has been the retrospective observation that success or failure of today's widely publicized systems often seems preordained; that is, traceable to their beginnings. It is not a new realization. It was just as apparent to the ancient Egyptians, Greeks, and Romans who originated classical architecting in response to it. The difference between their times and now is in the extraordinary complexity and technological capability of what could then and now be built.

Today's architecting must handle systems of types unknown until very recently; for example, systems that are very high quality, real-time, closed-loop, reconfigurable, interactive, software-intensive, and, for all practical purposes, autonomous. New domains like personal computers, intersatellite networks, health services, and joint service command and control are calling for new architectures — and for architects specializing in those domains. Their needs and lessons learned are in turn leading to new architecting concepts and tools and to the acknowledgment of a new formalism, and evolving profession, called systems architecting; a combination of the principles and concepts of both systems and of architecting.

The reasons behind the general acknowledgment of architecting in the new systems world are traceable to that remarkable period immediately after end of the Cold War in the mid-1980s. Abruptly, by historical standards, a 50-year period of continuity ended. During the same period, there was a dramatic upsurge in the use of smart, real-time systems, both civilian and military, which required much more than straightforward refinements of established system forms. Long-range management strategies and design rules, based on years of continuity, came under challenge. It is now apparent that the new era of global transportation, global communications, global competition, and global turmoil is not only different in type and direction, it is unique technologically and politically. It is a time of restructuring and invention, of architecting new products and processes, and of new ways of thinking about how systems are created and built.

Long-standing assumptions and methods are under challenge. For example, for many engineers, architectures were a given; automobiles, airplanes, and even spacecraft had had the same architectural forms for decades. What need was there for architecting? Global competition soon provided an answer. Architecturally different systems were capturing markets. Consumer product lines and defense systems are well-reported examples. Other questions include: how can software architectures be created that evolve as fast as their supporting technologies? How deeply should a systems architect go into the details of all the system's subsystems? What is the distinction between architecting and engineering?

Distinguishing between architecting and engineering

Because it is the most asked by engineers in the new fields, the first issue to address is the distinction between architecting and engineering in general; that is, regardless of engineering discipline. Although civil engineers and civil architects, even after centuries of debate, have not answered that question in the abstract, they have in practice. *Generally speaking*, engineering deals almost entirely with measurables using analytic tools derived from mathematics and the hard sciences; that is, engineering is a <u>de</u>ductive process. Architecting deals largely with unmeasurables using nonquantitative tools and guidelines based on practical lessons learned; that is, architecting is an <u>in</u>ductive process. At a more detailed level, engineering is concerned with quantifiable costs, architecting with qualitative worth. Engineering aims for technical optimization, architecting for client satisfaction. Engineering is more of a science, architecting more of an art.

In brief, the practical distinction between engineering and architecting is in the problems faced and the tools used to tackle them. This same distinction appears to apply whether the branch involved is civil, mechanical, chemical, electrical, electronic, aerospace, software, or systems.* Both architecting and engineering can be found in every one. Architecting and engineering are roles which are distinguished by their characteristics. They represent two edges of a continuum of systems practice. Individual engineers often fill roles across the continuum at various points in their careers or on different systems. The characteristics of the roles, and a suggestion for an intermediate role, are shown in Table 1.1.

As the table indicates, architecting is characterized by dealing with ill-structured situations, situations where neither goals nor means are known with much certainty. In systems engineering terms, the requirements for the system have not been stated more than vaguely, and the architect cannot appeal to the client for a resolution as the client has engaged the architect precisely to assist and advise in such a resolution. The architect engages in a joint exploration of requirements and design, in contrast to the classic engineering approach of seeking an optimal design solution to a clearly defined set of objectives.

Because the situation is ill-structured, the goal cannot be optimization. The architect seeks satisfactory and feasible problem-solution pairs. Good architecture and good engineering are both the products of art and science, and a mixture of analysis and heuristics. However, the weight will fall on heuristics and "art" during architecting.

One way to clearly see the distinction is in the approach to interfaces and system integrity. When a complex system is built (say one involving 10,000 person-years of effort) only absolute consistency and completeness of interface descriptions and disciplined methodology and process can suffice. When a system is physically assembled it matters little whether an

* The systems branch, possibly new to some readers, is described in Rechtin, 1991, and in Chapter 1 of this book.

Table 1.1 The Architecting-Engineering Continuum

Characteristic	Architecting	A & E	Engineering
Situation/goals	Ill-structured	Constrained	Understood
	Satisfaction	Compliance	Optimization
Methods	Heuristics	$\leftrightarrow$	Equations
	Synthesis	$\leftrightarrow$	Analysis
	Art and science	Art **and** Science	**Science** and Art
Interfaces	Focus on "mis-fits"	Critical	Completeness
System integrity maintained through	"Single mind"	Clear objectives	Disciplined methodology and process
Management issues	Working for Client	Working with Client	Working for Builder
	Conceptualization and certification	Whole waterfall	Meeting project requirements
	Confidentiality	Conflict of interest	Profit vs. cost

interface is high- or low-tech, if it is not exactly correct the system does not work. In contrast, during architecting it is necessary only to identify the interfaces that cannot work, the misfits. Misfits must be eliminated during architecting, and then interfaces should be resolved in order of criticality and risk as development proceeds into engineering.

One important point is that the table represents management in the classical paradigm of how architecting is done, not necessarily how it actually is done. Classically, architecting is performed by a third party working for the client. In practice, the situation is more complex as the architecture might be done by the builder before a client is found, might be mixed into a competitive procurement, or might be done by the client. These variations are taken up in chapters to come.

Systems architecting is the subject of this book, and the art of it in particular, because, being the most interdisciplinary, its tools can be most easily applied in the other branches. From the author's experience, systems architecting quickly and naturally abstracts and generalizes lessons learned elsewhere, not only for itself but also for transfer and specialization in still other branches. A good example is the system guideline (or heuristic), abstracted from examples in all branches, of **Simplify. Simplify. Simplify.** It will appear several times in this text.

It is important in understanding the subject of this book to clarify certain expressions. The word "architecture" in the context of civil works can mean a structure, a process, or a profession; in this text it refers only to the structure. The word "architecting" refers only to the process. Architecting is an invented word to describe how architectures are created much as engineering describes how "engines" and other artifacts are created. In another, subtler, distinction from conventional usage, an "architect" is meant here to be an individual engaged in the process of achitecting, regardless of domain, job title, or employer; by definition and practice both. From time to time an architect may perform engineering and an engineer may perform architecting — whatever it takes to get the job done.

Clearly, both processes can and do involve elements of the other. Architecting can and does require top-level quantitative analysis to determine feasibility, and quantitative measures to certify readiness for use. Engineering can and occasionally does require the creation of architecturally different alternatives to resolve otherwise intractable design problems. For complex systems, both processes are essential.* In practice, it is rarely necessary to draw a sharp line between them.

Criteria for mature and effective systems architecting

An increasingly important need of project managers and clients is for criteria to judge the maturity and effectiveness of systems architecting in their projects — criteria analogous to those developed for software development by Carnegie Mellon's Software Engineering Institute. Based upon experience to date, criteria for systems architecting appear to be, in rough order of attainment:

- A recognition by clients and others of the need to architect complex systems
- An accepted discipline to perform that function; in particular, the existence of architectural methods, standards, and organizations
- A recognized separation of value judgments and technical decisions between client, architect, and builder
- A recognition that architecture is an art as well as a science; in particular, the development and use of nonanalytic as well as analytic techniques
- The effective utilization of an educated professional cadre; that is, of masters-level, if not doctorate-level, individuals and teams engaged in the process of systems-level architecting.

By those criteria, systems architecting is in its adolescence, a time of challenge, opportunity, and controversy. History and the needs of global competition would seem to indicate adulthood is close at hand.

* For further elaboration on the related questions of the role of the architect, see Rechtin, 1991, pp. 11–14; on the architect's tools, Parts One and Three of this book; and on architecting as a profession, Part Four of this book and *Systems Engineering*, the Journal of the International Council on Systems Engineering.

The architecture of this book

The first priority of this book has been to restate and extend into the future the retrospective architecting paradigm of Rechtin, 1991.* An essential part of both retrospective and extended paradigms is the recognition that systems architecting is part art and part science. Part One of this book further develops the art and extends the central role of heuristics. Part Two introduces five important domains that both need and contribute to the understanding of that art. Part Three helps bridge the space between the science and the art of architecting. In particular, it develops the core architecting process of modeling and representation by and for software. Part Four concentrates on architecting as a profession: the political process and its part in system design, and the professionalization of the field through education, research, and peer-reviewed journals.

The architecture of Part Two deserves an explanation. Without one, the reader may inadvertently skip some of the domains — builder-architected systems, manufacturing systems, social systems, software systems, and collaborative systems — because they are outside the reader's immediate field of interest. These chapters, instead, recognize the diverse origins of heuristics, illustrating and exploiting them. Heuristics often first surface in a specialized domain where they address an especially prominent problem. Then, by abstraction or analogy, they are carried over to others and become generic. Such is certainly the case in the selected domains. In these chapters the usual format of stating a heuristic and then illustrating it in several domains is reversed. Instead, it is stated, but in generic terms, in the domain where it is most apparent. Readers are encouraged to scan all the chapters of Part Two. The chapters may even suggest domains, other than the reader's, where the reader's experience can be valuable in these times of vocational change. References are provided for further exploration. For professionals already in one of the domains, the description of each is from an architectural perspective, looking for those essentials that yield generic heuristics and providing, in return, other generic ones that might help better understand those essentials. In any case, the chapters most emphatically are not intended to advise specialists about their specialties.

Architecting is inherently a multidimensional subject, difficult to describe in the linear, word-follows-word format of a book. Consequently, it is occasionally necessary to repeat the same concept in several places, internally and between books. A good example is the concept of systems. Architecting

* This second book is an extension of Rechtin, 1991, not a replacement for it. However, this book reviews enough of the fundamentals that it can stand on its own. If some subjects, such as examples of specific heuristics, seem inadequately treated, the reader can probe further in the earlier work. There are also a number of areas covered there that are not covered here, including the challenges of ultraquality, purposeful opposition, economics and public policy; biological architectures and intelligent behavior; and assessing architecting and architectures. A third book, "*Systems Architecting of Organizations, Why Eagles Can't Swim*," E. Rechtin, CRC Press, Boca Raton, FL, 1999, introduces a part of systems architecting related to, but different from, the first two.

can also be organized around several different themes or threads. Rechtin, 1991, was organized around the well-known waterfall model of system procurement. As such, its applicability to software development was limited. This book, more general, is organized by fundamentals, tools, tasks, domains, models, and vocation. Readers are encouraged to choose their own personal theme as they go along. It will help tie systems architecting to their own needs.

Exercises are interspersed in the text, designed for self-test of understanding and critiquing the material just presented. If the reader disagrees, then the disagreement should be countered with examples and lessons learned — the basic way that mathematically unprovable statements are accepted or denied. Most of the exercises are thought problems, with no correct answers. Read them, and if the response is intuitively obvious, charge straight ahead. Otherwise, pause and reflect a bit. A useful insight may have been missed. Other exercises are intended to provide opportunities for long-term study and further exploration of the subject. That is, they are roughly the equivalent of a masters thesis.

Notes and references are organized by chapter. Heuristics, by tradition, are bold faced when they appear alone, with an appended list of them completing the text.

Changes since the first edition

Since the publication of the first edition it has become evident that some sections of the book could be clearer, and new subjects have come to be important in the discussion of architecting. The authors have benefited from extensive feedback from working systems architects through seminars and other contacts. Where appropriate, that feedback has been incorporated into the book in the form of clearer explanations, better examples, and corrections to misunderstandings.

In several areas we have added new material. Two new chapters cover collaborative systems (a.k.a. systems-of-systems) and architecture description frameworks. The tremendous growth of the Internet and World Wide Web have led to greater appreciation for truly collaborative systems. A collaborative system is one in which the components voluntarily operate jointly to form a greater whole. The degree of independence of operation and management can subclassify systems of this overall type, and they appear to have methods and heuristic applications specific to this class.

The second new chapter discusses architecture description frameworks. As the importance of architectures has become more broadly accepted, standards work has begun to codify good practices. A major area for such standards work is architecture description, the equivalent of blueprint standards. Most of the emerging standards are roughly similar in intellectual approach, but they use distinctly different terminology and make quite different statements about what features are important.

In addition, the chapters on software and modeling have been substantially reworked, and the chapter on professionalization has been updated. The pace of work in software architecture has been very fast, and the chapter updated to reflect this. Also since the publication of this book, new university educational programs have begun that incorporate system architecture as an important element.

The intended readership for this book

This book is written for present and future systems architects, for experienced engineers interested in expanding their expertise beyond a single field, and for thoughtful individuals concerned with creating, building, or using complex systems.

From experience with its predecessor, the book can be used as a reference work for graduate studies, for senior capstone courses in engineering and architecture, for executive training programs, and for the further education of consultants, systems acquisition and integration specialists, and as background for legislative staffs. It is a basic text for a Masters of Science degree in Systems Architecture and Engineering at the University of Southern California.

Acknowledgments

Special acknowledgment for contributions to the second edition is due from the authors to three members of the USC faculty, Associate Dean Richard Miller, now President of Olin College; Associate Dean Elliot Axelband, who originally requested this book and now directs the USC Master's Program in Systems Engineering and Architecture; and to Professor Charles Weber of Electrical Engineering Systems. And to two members of the School of Engineering staff, Margery Berti and Mary Froehlig, who architected the Master of Science in Systems Architecture and Engineering from an experimental course and a flexible array of multidisciplinary courses at USC. Perhaps only other faculty members can appreciate how much such support means to a new field and its early authors.

To learn, teach — especially to industry-experienced, graduate-level students. Much indeed has been learned by the authors of this book in a decade of actively teaching the concepts and processes of systems architecting to over a thousand of these students. At least a dozen of them deserve special recognition for their unique insights and penetrating commentary: Michael Asato, Kenneth Cureton, Susan Dawes, Norman P. Geis, Douglas R. King, Kathrin Kjos, Jonathan Losk, Ray Madachy, Archie W. Mills, Jerry Olivieri, Tom Pieronek, and Marilee Wheaton. The quick understanding and extension of the architecting process by all the students has been a joy to behold and is a privilege to acknowledge.

Following encouragement from Professor Weber in 1988 to take the first class offered in systems architecting as part of his Ph.D. in Electrical Engineering Systems, Mark Maier, its finest student according to its originator, went on to be an associate professor at the University of Alabama in Huntsville, a senior staff member at the Aerospace Corporation, and the first author of this edition. He, in turn, thanks his colleagues at Hughes Aircraft who, years earlier, introduced him to system methods, especially Dr. Jock Rader, Dr. Kapriel Krikorian, and Robert Rosen; and his current colleagues at the Aerospace Corporation who have worked with him on the Aerospace Systems Architecting Program: Dr. Mal Depont, Dr. William Hiatt, Dr. Bruce Gardner, Dr. Kevin Kreitman, and Ms. Andrea Amram.

The authors owe much to the work of Drs. Larry Druffel, David Garlan, and Mary Shaw* of Carnegie Mellon University's Software Engineering Institute, Professor Barry Boehm of USC's Center for Software Engineering, and Dr. Robert Balzer of USC's Information Sciences Institute, software architects all, whose seminal ideas have migrated from their fields both to engineering systems in general and its architecting in particular. Dr. Brenda Forman, then of USC, now retired from the Lockheed Martin Corporation and the author of Chapter 12, accepted the challenge of creating a unique course on the "facts of life" in the national political process and how they affect — indeed often determine — architecting and engineering design.

Manuscripts may be written by authors, but publishing them is a profession and contribution unto itself requiring judgment, encouragement, tact, and a considerable willingness to take risk. For all of these we thank Norm Stanton, a senior editor of CRC Press, Inc., editor of the first edition of this book, who has understood and supported the field beginning with the publication of Frederick Brooks' classic architecting book, *The Mythical Man-Month*, more than two decades ago.

And a concluding acknowledgment to those who inspire and sustain us, our wives and families: without you this book was inconceivable. With you, it came to life.

Eberhardt Rechtin Mark Maier
March, 2000
Southern California The Aerospace Corporation Northern Virginia

* Authors of the watershed book, *Software Architecture, Perspectives on an Emerging Discipline,* Prentice-Hall, Englewood Cliffs, NJ, 1996.

The Authors

Mark W. Maier, Ph.D., is a senior engineering specialist at The Aerospace Corporation, Reconnaissance Systems Division, an architect-engineering firm specializing in space systems for the United States government. His specialty is systems architecture. He teaches and consults in that subject for The Aerospace Corporation, its clients, and corporations throughout the United States and Europe. He has done pioneering work in the field, especially in collaborative systems, socio-technical systems, modeling, and application to distributed systems-of-systems. He is senior member of the IEEE, a member of the author team for IEEE P1471 *Recommended Practice for Architecture Description*, and co-chair of the systems architecture working group of the International Council on Systems Engineering.

Dr. Maier received his B.S. degree in engineering and applied science and an M.S. in electrical engineering from Caltech in 1983 and 1984, respectively. He joined the Hughes Aircraft Company in El Segundo, CA, on graduation from Caltech in 1983. At Hughes he was a section head responsible for signal processing algorithm design and systems engineering. While at Hughes he was awarded a Howard Hughes Doctoral Fellowship, on which he completed a Ph.D. in electrical engineering, specializing in radar signal processing, at USC. At USC he began his collaboration with Dr. Eberhardt Rechtin, who was beginning the first systems architecture academic program. In 1992 he joined the faculty of the University of Alabama in Huntsville in the Electrical and Computer Engineering Department. As an Associate Professor at UAH he carried out research in system architecture, stereo image compression, and radar signal processing. He has published several dozen journal and conference papers in these fields. He was also the faculty advisor to the SEDSAT-1 student satellite project which was launched in 1998.

Eberhardt Rechtin, Ph.D., recently retired from the University of Southern California (USC) as a professor with joint appointments in the departments of Electrical Engineering/Systems, Industrial and Systems Engineering, and Aerospace Engineering. His technical specialty is systems architecture in those fields. He had previously retired (1987) as President-emeritus of The Aerospace Corporation, an architect–engineering firm specializing in space systems for the U.S. Government.

Rechtin received his B.S. and Ph.D. degrees from Caltech in 1946 and 1950, respectively. He joined Caltech's Jet Propulsion Laboratory in 1948 as an engineer, leaving in 1967. At JPL he was the chief architect and director of the NASA/JPL Deep Space Network. In 1967 he joined the Office of the Secretary of Defense as the Director of the Advance Research Projects Agency and later as Assistant Secretary of Defense for Telecommunications. He left the Defense Department in 1973 to become chief engineer for Hewlett-Packard. He was elected as the president and CEO of Aerospace in 1977, retiring in 1987, and joining USC to establish its graduate program in systems architecture.

Rechtin is a member of the National Academy of Engineering and a Fellow of the Institute of Electrical and Electronic Engineers, the American Institute of Aeronautics and Astronautics, and the American Association for Advancement of Science, and is an academician of the International Academy of Astronautics. He has been further honored by the IEEE with its Alexander Graham Bell Award, by the Department of Defense with its Distinguished Public Service Award, NASA with its Medal for Exceptional Scientific Achievement, the AIAA with its Robert H. Goddard Award, Caltech with its Distinguished Alumni Award, The International Council on Systems Engineering with its Pioneer Award, and by the (Japanese) NEC with its C&C Prize. Rechtin is also the author of *Systems Architecting, Creating and Building Complex Systems*, Prentice-Hall, 1991, and *The Art of Systems Architecting*, CRC Press LLC, 1997.

Contents

Part one

Introduction — the art of architecting

A brief review of classical architecting methods

Architecting: the art and science of designing and building systems.[1]

The four most important methodologies in the process of architecting are characterized as normative, rational, participative, and heuristic (Table 1.1).[2] As might be expected, like architecting itself, they contain both science and art. The science is largely contained in the first two, normative and rational, and the art in the last two, participative and heuristic.

Table 1.1 Four Architecting Methodologies

Normative (solution-based)
Examples: building codes and communications standards
Rational (method-based)
Examples: systems analysis and engineering
Participative (stakeholder-based)
Examples: concurrent engineering and brainstorming
Heuristic (lessons learned)
Examples: **Simplify. Simplify. Simplify.** and **SCOPE.**

The normative technique is solution-based; it prescribes architecture as it "should be;" that is, as given in handbooks, civil codes, and in pronouncements by acknowledged masters. Follow it and the result will be successful by definition.

Limitations of the normative method — such as responding to major changes in needs, preferences, or circumstances — lead to the rational method: scientific, and mathematical principles to be followed in arriving at a solution to a stated problem. It is method- or rule-based. Both the normative and rational methods are analytic, deductive, experiment-based, easily certified, well understood, and widely taught in academia and industry. Moreover, the best normative rules are discovered through engineering science (think of modern building codes), truly a formidable set of positives.

However, because science-based methods are absolutely necessary parts of architecting, they are not the focus of this book. They are already well treated in a number of architectural and engineering texts. Equally necessary, and the focus of this part of the book, is the art, or practice, needed to complement the science for highly complex systems.

In contrast with science-based methodologies, the art or practice of architecting — like the practice of medicine, law, and business — is nonanalytic, inductive, difficult to certify, less understood, and, at least until recently, is seldom taught formally in either academia or industry. It is a process of insights, vision, intuitions, judgment calls, and even "taste."[3] It is key to creating truly new types of systems for new and often unprecedented applications. Here are some of the reasons why it is key.

For unprecedented systems, past data is of limited use. For others, analysis can be overwhelmed by too many unknowns, too many stakeholders, too many possibilities, and too little time for data gathering and analysis to be practical. To cap it off, many of the most important factors are not measurable. Perceptions of worth, safety, affordability, political acceptance, environmental impact, public health, and even national security provide no realistic basis for numerical analyses — even if they weren't highly variable and uncertain. Yet, if the system is to be successful, these perceptions must be accommodated from the first, top-level, conceptual model down through its derivatives.

The art of architecting, therefore, complements its science where science is weakest: in dealing with immeasurables, in reducing past experience and wisdom to practice, in conceptualization, in inspirationally putting disparate things together, in providing "sanity checks," and in warning of likely but unprovable trouble ahead. Terms like reasonable assumptions, guidelines, indicators, elegant design, and beautiful performance are not out of place in this art; nor are lemon, disaster, snafu, or loser. These are hardly quantifiable, but as real in impact as any science.

The participative methodology recognizes the complexities created by multiple stakeholders. Its objective is consensus. As a notable example, designers and manufacturers need to agree on a multiplicity of details if an end product is to be manufactured easily, quickly, and profitably. In simple but common cases, only the client, architect, and contractor have to be in agreement; but as systems become more complex, new and different participants have to agree as well.

Concurrent engineering, a recurrently popular acquisition method, was developed to help achieve consensus among many participants. Its greatest value, and its greatest contentions, are for systems in which widespread cooperation is essential for acceptance and success; for example, systems which directly impact on the survival of individuals or institutions. Its well-known weaknesses are undisciplined design by committee, diversionary brainstorming, the closed minds of "group think," and members without power to make decisions but with the unbridled right to second guess.

Arguably, the greatest mistake that can be made in concurrent engineering is to attempt to quantify it. It is not a science. It is a very human art.

The heuristics methodology is based on "common sense," that is, on what is sensible in a given context. Contextual sense comes from collective experience stated in as simple and concise a manner as possible. These statements are called heuristics, the subject of Chapter 2, and are of special importance to architecting because they provide guides through the rocks and shoals of intractable, "wicked" system problems. **Simplify** is the first and probably most important of them. They exist in the hundreds if not thousands in architecting and engineering, yet they are some of the most practical and pragmatic tools in the architect's kit of tools.

Different methods for different phases of architecting

The nature of classical architecting changes as the project moves from phase to phase. In the earliest stages of a project it is a structuring of an unstructured mix of dreams, hopes, needs, and technical possibilities when what is most needed has been called an inspired synthesizing of feasible technologies. It is a time for the art of architecting. Later on, architecting becomes an integration of, and mediation among, competing subsystems and interests — a time for rational and normative methodology. And, finally, there comes certification to all that the system is suitable for use, when it may take all the art and science to that point to declare the system as built is complete and ready for use.

Not surprisingly, architecting is often individualistic, and the end results reflect it. As Frederick P. Brooks put it in 1982[4] and Robert Spinrad stated in 1987,[5] the greatest architectures are the product of a single mind — or of a very small, carefully structured team. To which should be added, in all fairness: a responsible and patient client, a dedicated builder, and talented designers and engineers.

Notes and references

1. *Webster's II New Riverside University Dictionary,* Riverside Publishing, Boston, 1984. As adapted for systems by substitution of "building systems" for "erecting buildings."
2. For a full discussion of these methods, see Lang, J., *Creating Architectural Theory, The Role of the Behavioral Sciences in Environmental Design,* Van Nostrand Reinhold Company, New York, 1987; and Rowe, P. G., *Design Thinking,* MIT Press, Cambridge, MA, 1987. They are adapted for systems architecting in Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991, 14.
3. Spinrad, R. J., In a lecture to the University of Southern California, 1988.
4. Brooks, F. P., *The Mythical Man-Month, Essays on Software Engineering,* Addison-Wesley, Reading, MA, 1983.
5. Spinrad, R. J., at a Systems Architecting lecture at the University of Southern California, fall 1987.

chapter one

Extending the architecture paradigm

Introduction: the classical architecting paradigm

The recorded history of classical architecting, the process of creating architectures, began in Egypt more than 4000 years ago with the pyramids, the complexity of which had been overwhelming designers and builders alike. This complexity had at its roots the phenomenon that as systems became increasingly more ambitious, the number of interrelationships among the elements increased far faster than the number of elements themselves. Pyramids were no longer simple burial sites; they had to be demonstrations of political and religious power, secure repositories of god-like rulers and their wealth, and impressive engineering accomplishments. Each demand, of itself, would require major resources. When taken together, they generated new levels of technical, financial, political, and social complications. Complex interrelationships among the combined elements were well beyond what the engineers' and builders' tools could handle.

From that lack of tools for civil works came classical civil works architecture. Millennia later, technological advances in shipbuilding created the new and complementary fields of marine engineering and naval architecture. In this century, rapid advances in aerodynamics, chemistry, materials, electrical energy, communications, surveillance, information processing, and software have resulted in systems whose complexity is again overwhelming past methods and paradigms. One of those is the classical architecting paradigm.

Responding to complexity

Complex: composed of interconnected or interwoven parts.[1]

System: a set of different elements so connected or related as to perform a unique function not performable by the elements alone.[2]

It is generally agreed that increasing complexity* is at the heart of the most difficult problems facing today's systems architecting and engineering. When architects and builders are asked to explain cost overruns and schedule delays, by far the most common, and quite valid, explanation is that the system is much more complex than originally thought. The greater the complexity, the greater the difficulty. It is important, therefore, to understand what is meant by system complexity if architectural progress is to be made in dealing with it.

The definitions of complexity and systems given at the start of this section are remarkably alike. Both speak to interrelationships (interconnections, interfaces, etc.) among parts or elements. As might be expected, the more elements and interconnections, the more complex the architecture and the more difficult the system-level problems.

Less apparent is that qualitatively different problem-solving techniques are required at high levels of complexity than at low ones. Purely analytical techniques, powerful for the lower levels, can be overwhelmed at the higher ones. At higher levels, architecting methods, experience-based heuristics, abstraction, and integrated modeling must be called into play.[3] The basic idea behind all of these techniques is to simplify problem solving by concentrating on its essentials. Consolidate and simplify the objectives. Stay within guidelines. Put to one side minor issues likely to be resolved by the resolution of major ones. Discard the nonessentials. Model (abstract) the system at as high a level as possible, then progressively reduce the level of abstraction. In short, **Simplify!**

It is important in reading about responses to complexity to understand that they apply throughout system development, not just to the conceptual phase. The concept that a complex system can be progressively partitioned into smaller and simpler units — and hence into simpler problems — omits an inherent characteristic of complexity, the interrelationships among the units. As a point of fact, poor aggregation and partitioning during development can *increase* complexity, a phenomenon all too apparent in the organization of work breakdown structures.

This primacy of complexity in system design helps explain why a single "optimum" seldom if ever exists for such systems. There are just too many variables. There are too many stakeholders and too many conflicting interests. No practical way may exist for obtaining information critical in making a "best" choice among quite different alternatives.

* A system need not be large or costly to be complex. The manufacture of a single mechanical part can require over 100 interrelated steps. A $10 microchip can contain thousands, even millions, of interconnected active elements.

The high rate of advances in the computer and information sciences

Unprecedented rates of advance in the computer and information sciences have further exacerbated an already complex picture. The advent of smart, software-intensive systems is producing a true paradigm shift in system design. Software, long treated as the glue that tied hardware elements together, is becoming the center of system design and operation. We see it in personal computers. The precipitous drop in computer hardware costs has generated a major design shift, from "keep the computer busy" to "keep the user busy." We see it in automobiles, where microchips increasingly determine the performance, quality, cost, and feel of cars and trucks.

We see the paradigm shift in the design of spacecraft and personal computers where complete character changes can be made in minutes. In effect, such software-intensive systems "change their minds" on demand. It is no longer a matter of debate whether machines have "intelligence;" the only real questions are of what kinds of intelligence and how best to use each one. And, because its software largely determines what and how the user perceives the system as a whole, its design will soon control and precede hardware design much as hardware design controls software today. This shift from "hardware first" to "software first" will force major changes on when and how system elements are designed, and who, with what expertise, will design the system as a whole. The impact on the value of systems to the user has been and will continue to be enormous.

One measure of this phenomenon is the proportion of development effort devoted to hardware and software for various classes of product. Anecdotal reports from a variety of firms in telecommunications and consumer electronics commonly show a reversal of the proportion from 70% hardware and 30% software to 30% hardware and 70% software. This shift has created major challenges and destroyed some previously successful companies. When the cost of software development dominates, development systems should be organized to simplify software development. But good software architectures and good hardware architectures are often quite different. Good architectures for complex software usually emphasize layered structures that cross many physically distinct hardware entities. Good software architectures also emphasize information hiding and close parallels between implementation constructs and domain concepts at the upper layers. These are in contrast to the emphasis on hierarchical decomposition, physical locality of communication, and interface transparency in good hardware architectures. Organizations find trouble when their workload moves from hardware- to software-dominated but their management and development skills no longer "fit" the systems they should support.

Particularly susceptible to these changes are systems that depend upon electronics, and information systems and that do not enjoy the formal partnership with architecting that structural engineering has long enjoyed. This book is an effort to remedy that lack by showing how the historical principles of classical architecting can be extended to modern systems architecting.

The foundations of modern systems architecting

Although the day-to-day practice may differ significantly,[4] the foundations of modern systems architecting are much the same across many technical disciplines. Generally speaking, they are a systems approach, a purpose orientation, a modeling methodology, ultraquality, certification, and insight.[5] Each will be described in turn.

A systems approach

A systems approach is one that focuses on the system as a whole, particularly when making value judgments (what is required) and design decisions (what is feasible). At the most fundamental level, *systems are collections of different things which together produce results unachievable by the elements alone.* For example, only when all elements are connected and working together do automobiles produce transportation, human organs produce life, and spacecrafts produce information. These system-produced results, or so-called "system functions," derive almost solely from the interrelationships among the elements, a fact that largely determines the technical role and principal responsibilities of the systems architect.

Systems are interesting because they achieve results, and achieving those results requires different things to interact. Based on experience with systems over the last decade, it is difficult to underestimate the importance of this specific definition of systems to what follows, literally, on a word-by-word basis. Taking a systems approach means paying close attention to results or the reasons we build a system. Architecture *must* be grounded in the client's/user's/customer's purpose. Architecture is not just about the structure of components.

It is the responsibility of the architect to know and concentrate on the critical few details and interfaces that really matter and not to become overloaded with the rest. It is a responsibility that is important not only for the architect personally, but for effective relationships with the client and builder. To the extent that the architect must be concerned with component design and construction, it is with those specific details that critically affect the system as a whole.

For example, a loaded question often posed by builders, project managers, and architecting students is, "How deeply should the architect delve into each discipline and each subsystem?" A graphic answer to that question is shown in Figure 1.1, exactly as sketched by Bob Spinrad in a 1987 lecture at the University of Southern California. The vertical axis is a relative measure of how deep into a discipline or subsystem an architect must delve to understand its consequences to the system as a whole. The horizontal axis lists the disciplines, such as electronics or stress mechanics, and/or the subsystems, such as computers or propulsion systems. Depending upon the specific system under consideration, a great deal of , or a very little understanding may be necessary.

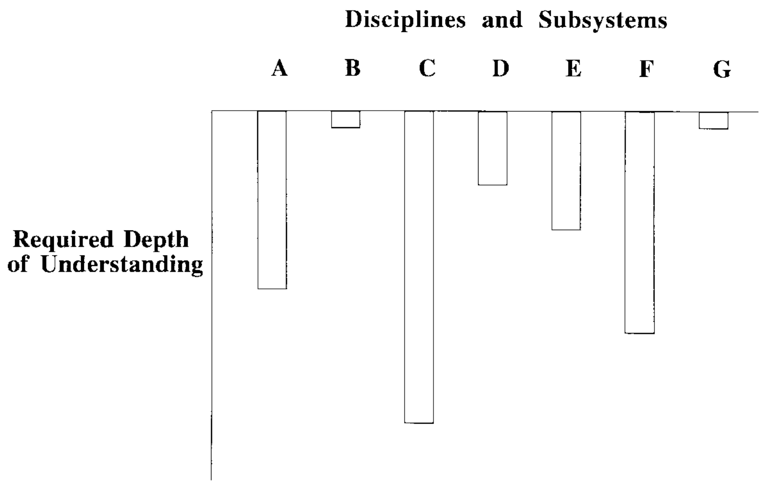


Figure 1.1

This leads to the question: "How can the architect possibly know before there is a detailed system design, much less before a system test, what details of what subsystem are critical?" A quick answer is: only through experience, through encouraging open dialog with subsystem specialists, and by being a quick, selective, tactful, and effective student of the system and its needs. Consequently, and perhaps more than any other specialization, architecting is a continuing, day-to-day learning process. No two systems are exactly alike. Some will be unprecedented, never built before.

> Exercise: Put yourself in the position of an architect asked to help a client build a system of a new type whose general nature you understand (a house, a spacecraft, a nuclear power plant, or a system in your own field) but which must considerably outperform an earlier version by a competitor. What do you expect to be the critical elements and details and in what disciplines or subsystems? What elements do you think you can safely leave to others? What do you need to learn the most about? Reminder: You will still be expected to be responsible for all aspects of the system design.

Critical details aside, the architect's greatest concerns and leverage are still, and should be, with the systems' connections and interfaces because: (1) they distinguish a system from its components; (2) their addition produces unique system-level functions, a primary interest of the systems architect; (3) subsystem specialists are likely to concentrate most on the core and least on the periphery of their subsystems, viewing the latter as (generally welcomed) external constraints on their internal design. Their concern for the system as a whole is understandably less than that of the systems architect — if not managed well, the system functions can be in jeopardy.

A purpose orientation

Systems architecting is a process driven by a client's purpose or purposes. A president wants to meet an international challenge by safely sending astronauts to the moon and back. Military services need nearly undetectable strike aircraft. Cities call for pollutant-free transportation.

Clearly, if a system is to succeed, it must satisfy a useful purpose at an affordable cost for an acceptable period of time. Note the explicit value judgments in these criteria: a *useful* purpose, an *affordable* cost, and an *acceptable* period of time. Each and every one is the client's prerogative and responsibility, emphasizing the criticality of client participation in all phases of system acquisition. Of the three criteria, satisfying a *useful* purpose is predominant. Without it being satisfied, all others are irrelevant. Architecting therefore begins with, and is responsible for maintaining, the integrity of the system's utility or purpose.

For example, the Apollo manned mission to the moon and back had a clear purpose, an agreed cost, and a no-later-than date. It delivered on all three. Those requirements, kept up front in every design decision, determined the mission profile of using an orbiter around the moon and not an earth-orbiting space station, and on developing electronics for a lunar orbit rendezvous instead of developing an outsize propulsion system for a direct approach to the lunar surface.

As another example, NASA headquarters, on request, gave the NASA/JPL Deep Space Network's huge ground antennas a clear set of priorities: first performance, then cost, then schedule, even though the primary missions they supported were locked into the absolute timing of planetary arrivals. As a result, the first planetary communication systems were designed with an alternate mode of operation in case the antennas were not yet ready. As it turned out, and as a direct result of the NASA risk-taking decision, the antennas were carefully designed, not rushed, and not only satisfied all criteria for the first launch, but for all launches for the next 40 years or so.

The Douglas Aircraft DC-3, though originally thought by the airline (now TWA) to require three engines, was rethought by the client and the designers in terms of its underlying purpose — to make a profit on providing affordable long-distance air travel over the Rocky and Sierra Nevada mountains for paying commercial passengers. The result was the two-engine DC-3, the plane that introduced global air travel to the world.

In contrast, the promises of the shuttle as an economically viable transporter to low earth orbit were far from met, although in fact the shuttle did turn out to be a remarkably reliable and unique launch vehicle for selected missions.

As of this date, the presumed purposes of the space station do not appear to be affordable and a change of purpose may be needed. The unacceptable cost/benefit ratios of the supersonic transport, the space-based ballistic missile defense system, and the superconducting supercollider terminated all these projects before their completion.

Curiously, the end use of a system is not always what was originally proposed as its purpose. The F-16 fighter aircraft was designed for visual air-to-air combat, but in practice it has been most used for ground support. The ARPANET-INTERNET communication network originated as a government-furnished computer-to-computer linkage in support of university research; it is now most used, and paid for, by individuals for e-mail and information accessing. Both are judged as successful. Why? Because, as circumstances changed, providers and users redefined the meaning of useful, affordable, and acceptable. A useful heuristic comes to mind: **Design the structure with "good bones."** It comes from the architecting of buildings, bridges, and ships where it refers to structures that are resilient to a wide range of stresses and changes in purpose. It could just as well come from physiology and the remarkably adaptable spinal column and appendages of all vertebrates — fishes, amphibians, reptiles, birds, and mammals.

> Exercise: Identify a system whose purpose is clear and unmistakable. Identify, contact, and, if possible, visit its architect. Compare notes and document what you learned.

Technology-driven systems, in notable contrast to purpose-driven systems, tell a still different story. They are the subject of Chapter 3.

A modeling methodology

Modeling is the creation of abstractions or representations of the system to predict and analyze performance, costs, schedules, and risks, and to provide guidelines for systems research, development, design, manufacture, and management. Modeling is the centerpiece of systems architecting — a mechanism of communication to clients and builders, of design management with engineers and designers, of maintaining system integrity with project management, and of learning for the architect, personally.

> Examples: The balsa wood and paper scale models of a residence, the full-scale mockup of a lunar lander, the rapid prototype of a software application, the computer model of a communication network, or the mental model of a user.

Modeling is of such importance to architecting that it is the sole subject of Part Three. At this point it is important to clear up a misconception. There has been a common perception that modeling is solely a *conceptual* process; that is, its purpose is limited to representing or simulating only the general features of an eventual system. If so, a logical question can be raised: when to stop modeling and start building? (It could almost have been asked as: when to stop architecting and begin engineering?)

Indeed, just that question was posed by Mark Maier in the first USC graduate class. Thinking had just begun on the so-called "stop question" when it was realized that modeling did *not* stop at the end of conceptualization in any case. Rather, it progressed, evolved, and solved problems from the beginning of a system's acquisition to its final retirement. There are, of course, conceptual models, but there are also engineering models and subsystem models; models for simulation, prototypes, and system test; demonstration models, operational models, and mental models by the user of how the system behaves.

Models are, in fact, created by many participants, not just by architects. These models must somehow be made consistent with overall system imperatives. It is particularly important that they be consistent with the architect's system model, a model that evolves, becoming more and more concrete and specific as the system is built. It provides a standard against which consistency can be maintained, and is a powerful tool in maintaining the larger objective of system integrity. And, finally, when the system is operational and a deficiency or failure appears, a model — or full-scale simulator if one exists — is brought into play to help determine the causes and cures of the problem. The more complete the model, the more accurately possible failure mechanisms can be duplicated until the one and only cause is identified.

In brief, modeling is a multipurpose, progressive activity, evolving and becoming less abstract and more concrete as the system is built and used.

Ultraquality implementation

Ultraquality is defined as a level of quality so demanding that it is impractical to measure defects, much less certify the system prior to use.[6] It is a limiting case of quality driven to an extreme, a state beyond acceptable quality limits (AQL) and statistical quality control. It requires a zero-defect approach not only to manufacturing, but to design, engineering, assembly, test, operation, maintenance, adaptation, and retirement — in effect, the complete life cycle.

Some examples are a new-technology spacecraft with a design lifetime of at least 10 years, a nuclear power plant that will not fail catastrophically within the foreseeable future, and a communication network of millions of nodes, each requiring almost 100% availability. Ultraquality is a recognition that the more components there are in a system, the more reliable each component must be to a point where, at the element level, defects become impractical to measure within the time and resources available. Yet, the reliability goal of the system as a whole must still be met. In effect, it reflects the not unreasonable demand that a system, regardless of size or complexity, should not fail to perform more than about 1% or less of the time. An ICBM shouldn't. A shuttle, at least 100 times more complex, shouldn't. An automobile shouldn't. A passenger airliner, at least 100 times more complex, shouldn't; as a matter of fact, we expect the airliner to fail far, far less than the family car.

> Exercise: Trace the histories of commercial aircraft and passenger buses over the last 50 years in terms of the number of trips that a passenger would expect to make without an accident. What does that mean to vehicle reliability as trips lengthen and become more frequent, as vehicles get larger, faster, and more complex? How were today's results achieved? What trends do you expect in the future? Did more software help or hinder vehicle safety?

The subject would be moot if it were not for the implications of this "limit state" of zero defects to design. Zero defects, in fact, originated as long ago as World War II, largely driven by patriotism. As a motivator, the zero defects principle was a prime reason for the success of the Apollo mission to the moon.

To show the implications of ultraquality processes, if a manufacturing line operated with zero defects there would be no need, indeed it would be worthless, to build elaborate instrumentation and information processing support systems. This would reduce costs and time, instead, by 30%. If an automobile had virtually no design or production defects, then sales outlets would have much less need for large service shops with their high capital and labor costs. Indeed, the service departments of the finest automobile manufacturers are seldom fully booked, resembling something like the famous Maytag commercial. Very little repair or service, except for routine maintenance, is required for 50 to 100,000 miles. Not coincidentally, these shops invariably are spotlessly clean, evidence of both the professional pride and discipline required for sustaining an ultraquality operation. Conversely, a dirty shop floor is one of the first and best indicators to a visitor or inspector of low productivity, careless workmanship, reduced plant yield, and poor product performance. The rocket, ammunition, solid state component, and automotive domains all bear witness to that fact.

As another example, microprocessor design and development has maintained the same per-chip defect rate even as the number and complexity of operations increased by factors of thousands. The corresponding failure rate per individual operation is now so low as to be almost unmeasurable. Indeed, for personal computer applications, a microprocessor hardware failure more than once a year borders on the unacceptable.

Demonstrating this limit state in high quality is not a simple extension of existing quality measures, though the latter may be necessary in order to get within range of it. In the latter there is an heuristic: (measurable) **acceptance tests must be both complete and passable.** How, then, can an inherently unmeasurable ultraquality be demanded or certified? Most of the answer seems to be in surrogate procedures, such as a zero defects program, and less in measurements, not because finer measurements would not be useful but because system complexity has outpaced instrument accuracy.

In that respect, a powerful addition to pre-1990 ultraquality techniques was the concept, introduced in the last few years, that each participant in a system acquisition sequence is both a buyer and a supplier. The original application, apparently a Japanese idea, was that each worker on a production line was a buyer from the preceding worker in the production line as well as a supplier to the next. Each role required a demand for high quality; that is, a refusal to buy a defective item and a concern not to deliver a defective one likely to be refused.[7] In effect, the supplier-buyer concept generates a self-enforcing quality program with built-in inspection. There would seem to be no reason why the same concept should not apply throughout system acquisition — from architect to engineer to designer to producer to seller to end user. As with all obvious ideas, the wonder is why it wasn't self-evident earlier.

When discussing ultraquality it may seem odd to be discussing heuristics. After all, isn't something as technologically demanding as quality beyond measure, the performance of things like heavy space boosters, not the domain of rigorous, mathematical engineering? In part, of course, it is. But experience has shown that rigorous engineering is not enough to achieve ultraquality systems. Ultraquality is achieved by a mixture of analytical and heuristic methods. The analytical side is represented by detailed failure analysis and even the employment of proof techniques in system design. In some cases these very rigorous techniques have been essential in allowing certain types of ultraquality systems to be architected.

Flight computers are a good example of the mixture of analytical and heuristic considerations in ultraquality systems. Flight control computers for statically unstable aircraft are typically required to have a mean time between failures (where a failure is one which produces incorrect flight control commands) on the order of 10 billion hours. This is clearly an ultraquality requirement since the entire production run of a given type of flight computer will not collectively run for 10 billion hours during their operational lifetimes. The requirement certainly cannot be proved by measurement and analysis. Nevertheless, aircraft administration authorities require that such a reliability requirement be certified.

Achieving the required reliability would seem to require a redundant computer design as individual parts cannot reach that reliability level. The problem with redundant designs is that introducing redundancy also introduces new parts and functions, specifically the mechanisms that manage the redundancy and must lock out the signals from redundant sections that have failed. For example, in a triple redundant system the redundant components must be voted to take the majority position (locking out a presumptive single failure). The redundancy management components are themselves subject to failure, and it possible that a redundant system is actually more likely to fail than one without redundancy. Further, “fault tolerance” depends upon the fault to be tolerated. Tolerating mechanical failure is of limited value if the fault is human error.

Creating redundant computers has been greatly helped by better analysis techniques. There are proof techniques that allow pruning of the unworkable failure trees by assuming so called “Byzantine” failure* models. These techniques allow strong statements to be made about the redundancy properties of designs. The heuristic part is trying to verify the absence of “common-mode-failures,” or failures in which several redundant and supposedly independent components fail at the same time for the same reason.

The Ariane 5 space launch vehicle was destroyed on its initial flight in a classic common mode failure. The software on the primary flight control computer caused the computer to crash shortly after launch. The dual redundant system then switched to the backup flight control computer, which had failed as well moments before for exactly the same reason that the primary computer failed. Ironically, the software failure was due to code leftover from the Ariane 4 and not actually necessary for the phase of flight in which it was operating. Arguably, in the case of the Ariane 5, more rigorous proof-based techniques of the mixed software and systems design might have found and eliminated the primary failure.

The analytical side is not enough, however. The best analysis of failure probabilities and redundancy can only verify that the system as built agrees with the model analyzed, and that the model possesses desired properties. It cannot verify that the model corresponds to reality. Well-designed ultraquality systems fail, but they typically fail for reasons not anticipated in the reliability model.

Certification

Certification is a formal statement by the architect to the client or user that the system, as built, meets the criteria both for client acceptance and for builder receipt of payment, i.e., it is ready for use (to fulfill its purposes). Certification is the grade on the “final exams" of system test and evaluation. To be accepted it must be well-supported, objective, and fair to client and builder alike.

> Exercise: Pick a system for which the purposes are reasonably clear. What tests would you, as a client, demand be passed for you to accept and pay for the system? What tests would you, as a builder, contract to pass in order to be paid? Whose word would each

* A Byzantine failure is one in which the failed component does the worst possible thing to the system. It is as if the component were possessed by a malign intelligence. The power of the technique is that it lends itself to certification, at least within the confines of well-defined models.

of you accept that the tests had or had not been passed? When should such questions be posed? (In first concept.)

Clearly, if certification is to be unchallenged, then there must be no perception of conflict of interest of the architect. This imperative has led to three widely accepted, professionally understood, constraints[8] on the role of the architect:

1. A *disciplined avoidance of value judgments*, that is, of intruding in questions of worth to the client; questions of what is satisfactory, what is acceptable, affordable, maintainable, reliable, etc. Those judgments are the imperatives, rights, and responsibilities of the client. As a matter of principle, the client should judge on desirability and the architect should decide (only) on feasibility. To a client's question of "What would you do in my position?" the experienced architect responds only with further questions until the client can answer the original one. To do otherwise makes the architect an advocate and, in some sense, the "owner" of the end system, preempting the rights and responsibilities of the client. It may make the architect famous, but the client will feel used. Residences, satellites, and personal computers have all suffered from such preemption (Frank Lloyd Wright houses, low earth orbiting satellite constellations, and the Lisa computer, respectively).
2. *A clear avoidance of perceived conflict of interest* through participation in research and development, including ownership or participation in organizations that can be, or are, building the system. The most evident conflict here is the architect recommending a system element from which the architect will supply and profit. This constraint is particularly important in builder-architected systems (Chapter 3).*
3. *An arms-length relationship with project management*, that is, with the management of human and financial resources other than of the architect's own staff. The primary reason for this arrangement is the overload and distraction of the architect created by the time-consuming responsibilities of project management. A second conflict, similar to that of participating in research and development, is created whenever architects give project work to themselves. If clients, for reasons of their own, nonetheless ask the architect to provide project management, it should be considered as a separate contract for a different task requiring different resources.

* Precisely this constraint led Congress to mandate the formation in 1960 of a non-profit engineering company, The Aerospace Corporation, out of the for-profit TRW Corporation, a builder in the aerospace business.

Insights and heuristics

A picture is worth a thousand words.
Chinese Proverb. 1000 BC
One insight is worth a thousand analyses.
Charles Sooter, April 1993

Insight, or the ability to structure a complex situation in a way that greatly increases understanding of it, is strongly guided by lessons learned from one's own or others' experiences and observations. Given enough lessons, their meaning can be codified into succinct expressions called "heuristics," a Greek term for guide. Heuristics are an essential complement to analytics, particularly in situations where analysis alone cannot provide either insights or guidelines.[9] In many ways they resemble what are called principles in other arts; for example, the importance of balance and proportion in a painting, a musical composition, or the ensemble of a string quartet. Whether as heuristics or principles, they encapsulate the insights that have to be achieved and practiced before a masterwork can be achieved.

Both architecting and the fine arts clearly require insight and inspiration as well as extraordinary skill to reach the highest levels of achievement. Seen from this perspective, the best systems architects are indeed artists in what they do. Some are even artists in their own right. Renaissance architects like Michaelangelo and Leonardo da Vinci were also consummate artists. They not only designed cathedrals, they executed the magnificent paintings in them. The finest engineers and architects, past and present, are often musicians; Simon Ramo and Ivan Getting, famous in the missile and space field, and, respectively, a violinist and pianist, are modern day examples.

The wisdom that distinguishes the great architect from the rest is that insight and inspiration, combined with well-chosen methods and guidelines and fortunate circumstances, creates masterworks. Unfortunately, wisdom does not come easily. As one conundrum puts it:

Success comes from wisdom.
Wisdom comes from experience.
Experience comes from mistakes.

Therefore, because success comes only after many mistakes, something few clients would willingly support, it presumably is either unlikely or must follow a series of disasters.

This reasoning might well apply to an individual. But applied to the profession as a whole, it clearly does not. The required mistakes and experience and wisdom gained from them can be those of one's predecessors, not necessarily one's own.

And from that understanding comes the role of education. It is the place of education to research, document, organize, codify, and teach those lessons so that the mistakes need not be repeated as a prerequisite for future successes. Chapter 2, Heuristics as Tools, is a start in that direction for the art of systems architecting.

The architecture paradigm summarized

This book uses the terms architect, architecture, and architecting with full consciousness of the "baggage" that comes with their use. Civil architecture is a well-established profession with its own professional societies, training programs, licensure, and legal status. Systems architecting borrows from it its basic attributes.

1. The architect is principally an agent of the client, not the builder. Whatever organization the architect is employed by, the architect must act in the best interests of the client for whom the system is being developed.
2. The architect works jointly with the client and builder on problem and solution definition. System requirements are an output of architecting, not really an input. Of course, the client will provide the architect some requirements, but the architect is expected to jointly help the client determine the ultimate requirements. An architect who needs complete and consistent requirements to begin work, though perhaps a brilliant builder, is not an architect.
3. The architect's product, or "deliverable," is an architecture representation, a set of abstracted designs of the system. The designs are not (usually) ready to use to build something. They have to be refined, just as the civil architect's floorplans, elevations, and other drawings must be refined into construction drawings.
4. The architect's product is not just physical representations. As an example, the civil architect's client certainly expects a "ballpark" cost estimate as part of any architecture feasibility question. So, too, in systems architecting where an adequate system architecture description must cover whatever aspects of physical structure, behavior, cost, performance, human organization, or other elements are needed to clarify the clients' priorities.
5. An initial architecture is a vision. An architecture description is a set of specific models. The architecture of a building is more than the blueprints, floorplans, elevations, and cost estimates. It includes elements of ulterior motives, beliefs, and unstated assumptions. This distinction is especially important in creating standards. Standards for architecture, like community architectural standards, are different from blueprint standards promoted by agencies or trade associations.

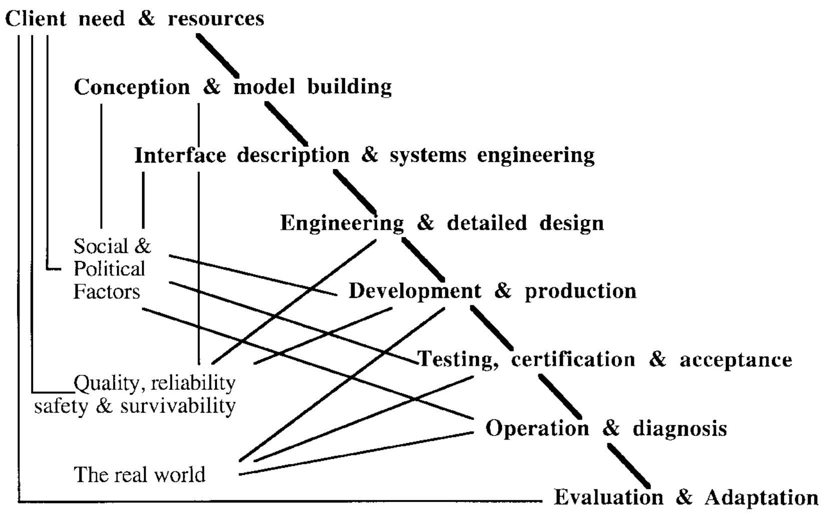
Architecting takes place within the context of an acquisition process. The traditional way of viewing hardware acquisitions is known as the waterfall model. The waterfall model captures many important elements of architecting practice, but it is also important in understanding other acquisition models, particularly the spiral for software, incremental development for evolutionary designing, and collaborative assembly for networks.

The waterfall model of systems acquisition

As with products and their architectures, no process exists by itself. All processes are part of still larger ones, and all processes have subprocesses. As with the product of architecture, so also is the process of architecting a part of a still larger activity, the acquisition of useful things.

Hardware acquisition is a sequential process that includes design, engineering, manufacturing, testing, and operation. This larger process can be depicted as an expanded waterfall, Figure 1.2.[10] The architect's functional relationship with this larger process is sketched in Figure 1.3. Managerially, the architect could be a member of the client's or the builder's organization, or of an independent architecting partnership in which perceptions of conflict of interest are to be avoided at all costs. In any case, and wherever the architect is physically or managerially located, the relationships to the client and the acquisition process are essentially as shown. The strongest (thickest line) decision ties are with client need and resources, conception and model building, and with testing, certification, and acceptance. Less prominent are the monitoring ties with engineering and manufacturing. There are also important, if indirect, ties with social and political factors, the "illities" and the "real world."

This waterfall model of systems acquisition has served hardware systems acquisition well for centuries. However, as new technologies create new, larger scale, more complex systems of all types, others have been



Client need & resources
Conception & model building
Interface description & systems engineering
Engineering & detailed design
Social & Political Factors
Development & production
Testing, certification & acceptance
Quality, reliability safety & survivability
Operation & diagnosis
The real world
Evaluation & Adaptation

Figure 1.2 The expanded waterfall.

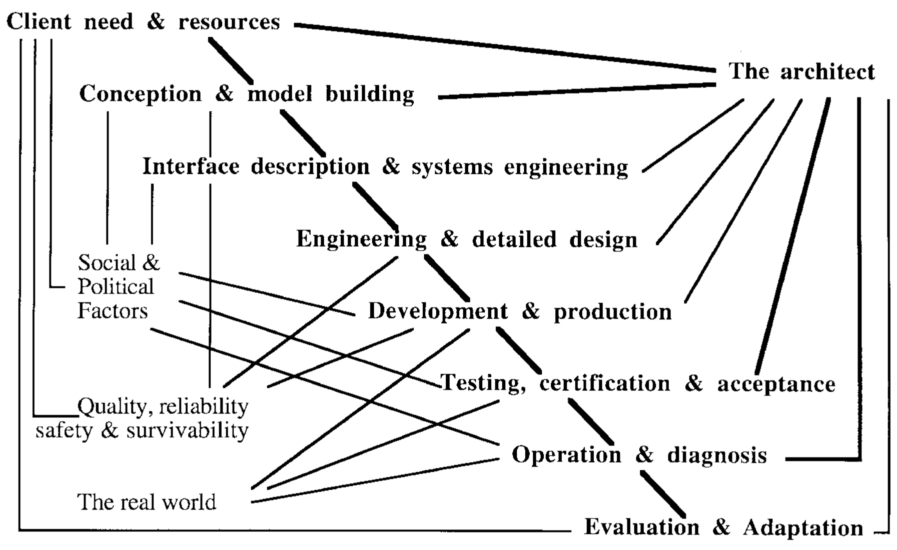

Client need & resources
The architect
Conception & model building
Interface description & systems engineering
Engineering & detailed design
Social & Political Factors
Development & production
Testing, certification & acceptance
Quality, reliability safety & survivability
Operation & diagnosis
The real world
Evaluation & Adaptation

Figure 1.3 The architect and the expanded waterfall. (Adapted from Rechtin 1991. With permission from Prentice-Hall.)

needed and developed. The most recent ones are due to the needs of software-intensive systems, as will be seen in Chapters 4 and 6 and in Part Three. While these models change the roles and methods of the architecting process, the basic functional relationships shown in Figure 1.3 remain much the same.

In any case, the relationships in Figure 1.3 are more complex than simple lines might suggest. As well as indicating channels for two-way communication and upward reporting, they infer the tensions to be expected between the connected elements, tensions caused by different imperatives, needs, and perceptions.

Some of competing technical factors are shown in Figure 1.4.[11] This figure was drawn such that directly opposing factors pull in exactly opposite directions on the chart. For example, continuous evolution pulls against product stability; a typical balance is that of an architecturally stable, evolving product line. Low-level decisions pull against strict process control, which can often be relieved by systems architectural partitioning, aggregation, and monitoring. Most of these tradeoffs can be expressed in analytic terms, which certainly helps, but some cannot, as will become apparent in the social systems world of Chapter 5.

> Exercise: Give examples from a specific system of what information, decisions, recommendations, tasks, and tensions might be expected across the lines of Figure 1.4.

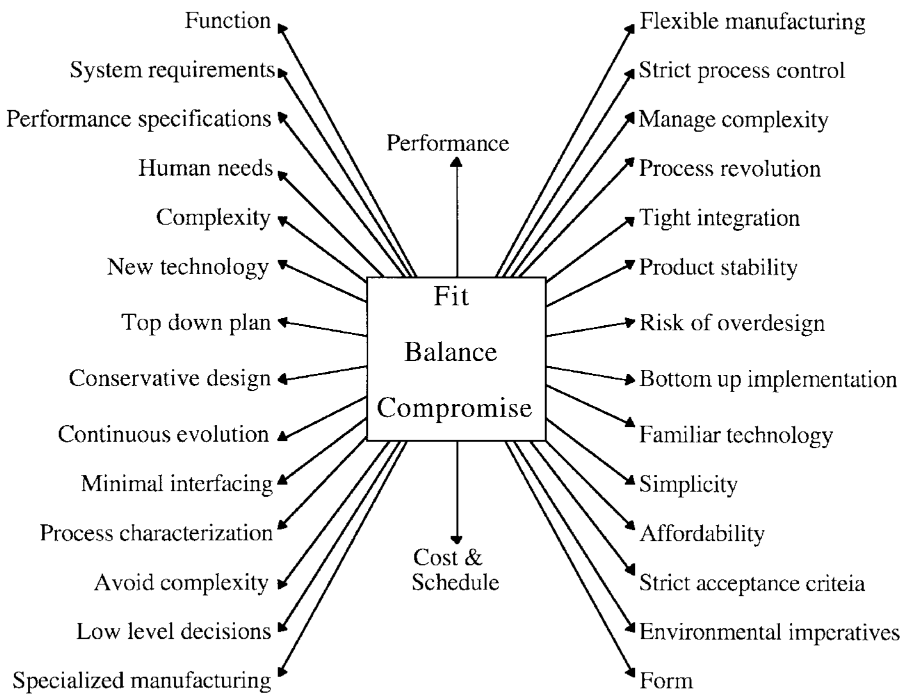


Figure 1.4 Tensions in systems architecting. (From Rechtin 1991. With permission from Prentice-Hall.)

Spirals, increments, and collaborative assembly

Software developers have long understood that most software-intensive projects are not well suited to a sequential process but to a highly iterative one such as the spiral. There is a strong incentive to iteratively modify software in response to user experience. As the market, or operational environment, reveals new desires, those desires are fed back into the product. One of the first formalizations of iterative development is due to Boehm and his famous spiral model. The spiral model envisions iterative development as a repeating sequence of steps. Instead of traversing a sequence of analysis, modeling, development, integration, and test just once, software may return over and over to each. The results of each are used as inputs to the next. This is depicted in Figure 1.5. This model is extended in Chapter 4 to the "Spiral-and-Circle" model depicted in Figure 4.3.

The original spiral model is intended to deliver one, hopefully stable, version of the product, the final of which is delivered at the end of the last spiral cycle. Multiple cycles are used for risk control. The nominal approach is to set a target number of cycles at the beginning of development, and partition the whole time available over the target number of cycles. The objective of each cycle is to resolve the most risky thing remaining. For example, if user acceptance was judged the most risky at the beginning of

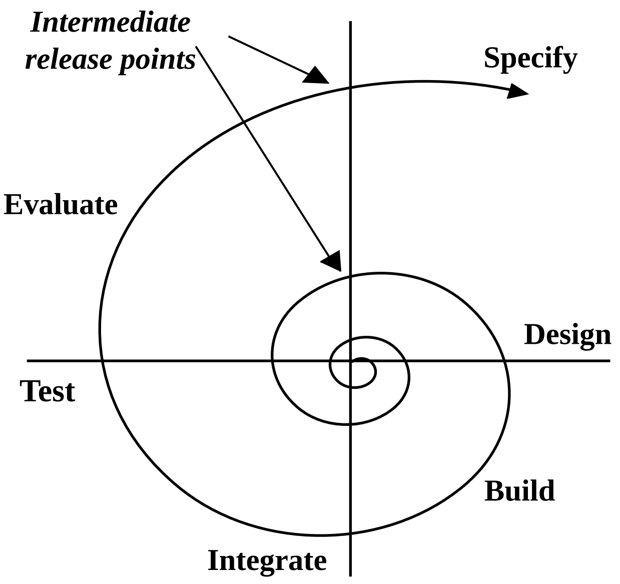


Figure 1.5

the project, the first spiral would concentrate on those parts of the system that produce the greatest elements of user experience. Even the first cycle tests would focus on increasing user acceptance. Similarly, if the most risky element was judged to be some internal technical performance issue, the product of the initial cycle would focus on technical feasibility.

Many software products, or the continuing software portion of many product lines, are delivered over and over again. A user may buy the hardware once and expect to be offered a steady series of software upgrades that improve system functionality and performance. This alters a spiral development process (which has a definite end) to an incremental process, which has no definite end. The model is now more like a spiral spiraling out to circles which represent the stable products to be delivered. After one circle is reached, an increment is delivered and the process continues. Actually, the notion of incremental delivery appears in the original spiral model where the idea is that the product of spirals before the last can be an interim product release if, for example, the final product is delayed.

Finally, there are a number of systems in use today that are essentially continuously assembled, and the assembly process is not directly controlled. The canonical example is the Internet, where the pieces evolve with only loose coupling to the other pieces. Control over development and deployment is fundamentally collaborative. Organizations, from major corporations to individual users, choose which product versions to use and when. No governing body exists (at least not yet) that can control the evolution of the elements. The closest thing at present to a governing body, the Internet Society, and its engineering arm, the Internet Engineering Task Force (IETF), can affect other behavior only through persuasion. If member organizations do not choose to support IETF standards the IETF has no authority to compel compliance or block noncomplying implementations.

We call systems like this "collaborative systems." The development process is collaborative assembly. Whether or not such an uncontrolled process can continue for systems like the Internet as they become central to daily life is unknown, but the logic and heuristics of such systems now is the subject of Chapter 7.

Summary and conclusions

A system is a collection of different things that together produce results unachievable by themselves alone. The value added by systems is in the interrelationships of their elements.

Architecting is creating and building structures, i.e., "structuring." *Systems* architecting is creating and building *systems*. It strives for fit, balance, and compromise among the tensions of client needs and resources, technology, and multiple stakeholder interests.

Architecting is both an art and a science — synthesis and analysis, induction and deduction, and conceptualization and certification — using guidelines from its art and methods from its science. As a process, it is distinguished from systems engineering in its greater use of heuristic reasoning, lesser use of analytics, closer ties to the client, and particular concern with certification of readiness for use

The foundations of systems architecting are a systems approach, a purpose orientation, a modeling methodology, ultraquality, certification, and insight. To avoid perceptions of conflict of interest, architects must avoid value judgments, avoid perceived conflicts of interest, and keep an arms-length relationship with project management.

A great architect must be as skilled as an engineer and as creative as an artist or the work will be incomplete. Gaining the necessary skills and insights depends heavily on lessons learned by others, a task of education to research and teach.

The role of systems architecting in the systems acquisition process depends upon the phase of that process. It is strongest during conceptualization and certification, but never absent. Omitting it at any point, as with any part of the acquisition process, leads to predictable errors of omission at that point to those connected with it.

Notes and references

1. *Webster II, New Riverside University Dictionary,* Riverside Publishing, Boston, 1984, p. 291.
2. Rechtin, 1991, p. 7.

3. Another factor in overruns and delays is uncertainty, the so-called unknowns and unknown unknowns. Uncertainty is highest during conceptualization, less in design, still less in redesign, and least in an upgrade. As with complexity, the higher the level, the more important become experience-based architecting methods. See Carpenter, R. G., *System Architects' Job Characteristics and Approach to the Conceptualization of Complex Systems*, Doctoral dissertation in Industrial and Systems Engineering, University of Southern California, Los Angeles, CA, 1995.
4. Cuff, D., *Architecture: The Story of Practice,* MIT Press, Cambridge, MA, 1991.
5. Rechtin, E., Foundations of systems architecting, *Sys. Eng. J. Nat. Counc. Sys. Eng.,* 1(1), 35, July/September 1994.
6. Discussed extensively in Juran, J. M., *Juran on Planning for Quality,* Free Press, New York, 1988; Phadke, M. S., *Quality Engineering Using Robust Design,* Prentice-Hall, Englewood Cliffs, NJ, 1989; and Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991.
7. See also Chapter 4, Manufacturing Systems.
8. From Cantry, D., What architects do and how to pay them, *Arch. Forum*, 119, 92, September 1963; and from discussions with Roland Russell, AIA.
9. See King, D. R., The Role of Informality in System Development or, A Critique of Pure Formality, University of Southern California 1992 (unpublished but available through USC).
10. Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991, 4.
11. Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991, 156.

chapter two

Heuristics as tools

Introduction: a metaphor

Mathematicians are still smiling over a gentle self-introduction by one of their famed members. "There are *three* kinds of mathematicians," he said, "those that know how to count and those that don't." The audience waited in vain for the third kind until, with laughter and appreciation, they caught on. Either the member couldn't count to three — ridiculous — or he was someone who believed that there was more to mathematics than numbers, important as they were. The number theorists appreciated his acknowledgment of them. The "those that don'ts" quickly recognized him as one of their own, the likes of a Gödel who, using thought processes alone, showed that no set of theorems can ever be complete.

Modifying the self-introduction only slightly to the context of this chapter, there are three kinds of people in our business: those who know how to count and those who don't, including the authors.

Those who know how to count (most engineers) approach their design problems using analysis and optimization, powerful and precise tools derived from the scientific method, and calculus. Those who don't (most architects) approach their qualitative problems using guidelines, abstractions, and pragmatics generated by lessons learned from experience; that is, heuristics. As might be expected, the tools each use are different because the kinds of problems they solve are different. We routinely and accurately describe an individual as "thinking like an engineer" or architect, or scientist, or artist. Indeed, by their tools and works ye shall know them.

This chapter, metaphorically, is about architects' heuristic tools. As with the tools of carpenters, painters, and sculptors, there are literally hundreds of them — but only a few are needed at any one time and for a specific job at hand. To continue the metaphor, although a few tool users make their own, the best source is usually a tool supply store, whether it be for hardware, artists' supplies, software, or heuristics. Appendix A, Heuristics for Systems-Level Architecting, is a heuristics store, organized by task, just like any good hardware store. Customers first browse, then select a kit of tools based on the job, personal skill, and a knowledge of the origin and intended use of each tool.

Heuristic has a Greek origin, *heuriskein*, a word meaning "to find a way" or "to guide" in the sense of piloting a boat through treacherous shoals. Architecting is a form of piloting. Its rocks and shoals are the risks and changes of technology, construction, and operational environment that characterize complex systems. Its safe harbors are client acceptance and safe, dependable, long life. Heuristics are guides along the way — channel markings, direction signs, alerts, warnings, and anchorages — tools in the larger sense. But they must be used with judgment. No two harbors are alike. The guides may not guarantee safe passage, but to ignore them may be fatal. The stakes in architecting are just as high — reputations, resources, vital services, and, yes, lives. Consonant with their origin, the heuristics in this book are intended to be trusted, time-tested guidelines for serious problem solving.

Heuristics as thus defined are narrower in scope, subject to more critical test and selection, and intended for more serious use than other guidelines; for example, conventional wisdom, aphorisms, maxims, and rules of thumb and the like. Also, a pair of mutually contradictory statements like *look before you leap* and *he who hesitates is lost* are hardly useful guides when encountering a cliff while running for your life. In this book, neither of these examples would be a valid heuristic because they offer contradictory advice for the same problem.

The purpose of this chapter is, therefore, to help the reader — whether architect, engineer, or manager — find or develop heuristics that can be trusted, organize them according to need, and use them in practice. The first step is to understand that heuristics are abstractions of experience.

Heuristics as abstractions of experience

One of the most remarkable characteristics of the human race is its ability not only to learn, but to pass on to future generations sophisticated abstractions of lessons learned from experience. Each generation knows more, learns more, plans more, tries more, and succeeds more than the previous one because it needn't repeat the time-consuming process of reliving prior experiences. Think of how extraordinarily efficient are such quantifiable abstractions as $F = ma$, $E = mc^2$ and $x = F(y,z,t)$; of algorithms, charts, and graphs; and of the basic principles of economics. This kind of efficiency is essential if large, lengthy, complex systems and long-lived product lines are to succeed. Few architects ever work on more than two or three complex systems in a lifetime. They have neither the time nor the opportunity to gain the experience needed to create first-rate architectures from scratch. By much the same process, qualitative heuristics, condensed and codified practical experience, came into being to complement the equations and algorithms of science and engineering in the solving of complex problems. Passed from architect to architect, from system to system, they worked and helped satisfy a real need.

In contrast to the symbols of physics and mathematics, the format of heuristics is words expressed in the natural languages. Unavoidably, they reflect the cultures of engineering, business, exploration, and human relations in which they arose. The birth of a heuristic begins with anecdotes and stories, hundreds of them, in many fields which become parables, fables, and myths,[1] easily remembered for the lessons they teach. Their impact, even at this early stage, can be remarkable not only on politics, religion, and business, but on the design of technical systems and services. The lessons that have endured are those that have been found to apply beyond the original context, extended there by analogy, comparison, conjecture, and testing.* At their strongest they are seen as self-evident truths requiring no proof.

There is an interesting human test for a good heuristic. An experienced listener, on first hearing one, will know within seconds that it fits that individual's model of the world. Without having said a word to the speaker, the listener almost invariably affirms its validity by an unconscious nod of the head, and then proceeds to recount a personal experience that strengthens it. Such is the power of the human mind.

Selecting a personal kit of heuristic tools

The art in architecting lies not in the wisdom of the heuristics, but in the wisdom of knowing which heuristics apply, a priori, *to the current project.*[2]

All professions and their practitioners have their own kits of tools, physical and heuristic, selected from their own and others' experiences to match their needs and talents. But, in the case of architecting and engineering prior to the late 1980s, selections were limited and, at best, difficult to acquire. An effort was therefore made in the USC graduate course in systems architecting to create a much wider selection by gathering together lessons learned throughout the West Coast aerospace, electronics, and software industries and expressing them in heuristic form for use by architects, educators, researchers, and students.

An initial collection[3] of about 100 heuristics was soon surpassed by contributions from over 200 students, reaching nearly 1000 heuristics within 6 years.[4] Many, of course, were variations on single, central ideas — just as there are many variations of hammers, saws, and screwdrivers — repeated

* This process is one of inductive reasoning, "a process of truth estimation in the face of incomplete knowledge which blends information known from experience with plausible conjecture." Klir, G. J., *Architecture of Systems Problem Solving*, Plenum Press, New York, 1985, 275. More simply, it is an extension or generalization from specific examples. It contrasts with deductive reasoning, which derives solutions for specific cases from general principles.

time and time again in different contexts. The four most widely applicable of these heuristics were, in decreasing order of popularity:

- **Don't assume that the original statement of the problem is necessarily the best, or even the right one.**

  Example: The original statement of the problem for the F-16 fighter aircraft asked for a high supersonic capability, which was difficult and expensive to produce. Discussions with the architect, Harry Hillaker, brought out that the reason for this statement was to provide a quick exit from combat, something far better provided by a high thrust-to-weight, low supersonic design. In short, the original high speed statement was replaced by a high acceleration one, with the added advantage of exceptional maneuverability.

- **In partitioning, choose the elements so that they are as independent as possible; that is, elements with low external complexity and high internal complexity.**

  Example: One of the difficult problems in the design of microchips is the efficient use of their surface area. Much of that area is consumed by connections between components; that is, by communications rather than by processing. Professor Carver Mead of Caltech has now demonstrated that a design based on minimum communications between process-intensive nodes results in much more efficient use of space, with the interesting further result that the chip "looks elegant" — a sure sign of a fine architecture and another confirmation of the heuristic, **the eye is a fine architect. Believe it.**

- **Simplify. Simplify. Simplify.**

  Example: One of the best techniques for increasing reliability while decreasing cost and time is to reduce the piece part count of a device. Automotive engineers, particularly recently, have produced remarkable results by substituting single castings for multiple assemblies and by reducing the number of fasteners and their associated assembly difficulties by better placement. A comparable result in computer software is the use of reduced instruction set computers (RISC) which increase computing speed for much the same hardware.

- **Build in and maintain options as long as possible in the design and implementation of complex systems. You will need them.**

  Example: In the aircraft business they are called "scars." In the software business they are called "hooks." Both are planned breaks or entry points into a system which can extend the functions the system can provide. For aircraft, they are used for lengthening the fuselage to carry more passengers or freight. For software, they are used for inserting further routines.

Though these four heuristics do not make for a complete tool kit, they do provide good examples for building one. All are aimed at reducing complexity, a prime objective of systems architecting. All have been trusted in one form or another in more than one domain. All have stood the test of time for decades if not centuries.

The first step in creating a larger kit of heuristics is to determine the criteria for selection. The following were established to eliminate unsubstantiated assertions, personal opinions, corporate dogma, anecdotal speculation, mutually contradictory statements, etc. As it turned out, they also helped generalize domain-specific heuristics into more broadly applicable statements. The strongest heuristics passed all the screens easily. The criteria were

- The heuristic must make sense in its original domain or context. To be accepted, a strong correlation, if not a direct cause and effect, must be apparent between the heuristic and the successes or failures of specific systems, products, or processes. Academically speaking, both the rationale for the heuristic and the report that provided it were subject to peer and expert review. As might be expected, a valid heuristic seldom came from a poor report.
- The general sense, if not the specific words, of the heuristic should apply beyond the original context. That is, the heuristic should be useful in solving or explaining more than the original problem from which it arose. An example is the foregoing **don't assume** heuristic. Another is **before the flight it's opinion; after the flight it's obvious.** In the latter, the word "flight" can be sensibly replaced by test, experiment, fight, election, proof, or trial. In any case, the heuristic should not be wrong or contradictory in other domains where it could lead to serious misunderstanding and error. This heuristic applies in general to ultraquality systems. When they fail, and they usually fail after all the tests are done and they are in actual use, we wonder how we missed such an obvious failure of our assumptions.

- The heuristic should be easily rationalized in a few minutes or on less than a page. As one of the heuristics itself states: "**If you can't explain it in five minutes, either you don't understand it or it doesn't work** (Darcy McGinn 1992, from David Jones). With that in mind, the more obvious the heuristic is on its face, and the fewer the limitations on its use, the better. Example: **A model is not reality.**
- The opposite statement of the heuristic should be foolish, clearly not "common sense." For example: The opposite of **Murphy's Law — "If it can fail, it will"** — would be "If it can fail, it won't," which is patent nonsense.
- The heuristic's lesson, though not necessarily its most recent formulation, should have stood the test of time and earned a broad consensus. Originally this criterion was that the heuristic *itself* had stood the test of time, a criterion that would have rejected recently formulated heuristics based on retrospective understanding of older or lengthy projects. Example: **The beginning is the most important part of the work** (Plato 4th Century B. C.), reformulated more recently as **All the serious mistakes are made in the first day** (Robert Spinrad, 1988).

It is probably true that heuristics can be even more useful if they can be used in a set, like wrenches and screwdrivers, hammers and anvils, or files and vises. The taxonomy grouping to follow achieves that possibility in part.

It is also probably true that a proposed action or decision is stronger if it is consistent with several heuristics rather than only one. A set of heuristics applicable to acceptance procedures substantiates that proposition.

And it would certainly seem desirable that a heuristic, taken in a sufficiently restricted context, could be specialized into a design rule; a quantifiable, rational evaluation; or a decision algorithm. If so, heuristics of this type would be useful bridges between architecting, engineering, and design.

Using heuristics

Virtually everybody, after brief introspection, sees that heuristics play an important role in their design and development activities. However, even if we accept that everyone uses heuristics, it is not obvious that those heuristics can be communicated to and used by others. This book takes the approach that heuristics can be effectively communicated to others. One lesson from student use of Rechtin, 1991, and the first edition of this book, is that heuristics do transfer from one person to another, but not always in simple ways. It is useful to document heuristics and teach from them, but learning styles differ.

People typically use heuristics in three ways. First, they can be used as evocative guides. They work as guides if they evoke new thoughts in the reader. Some readers have reported that they use the catalog of heuristics in the appendices at random when faced with a difficult design problem. If one of the heuristics seems suggestive, they follow up by considering how that heuristic could describe the present situation, what solutions it might suggest, or what new questions it suggests.

The second usage is as codifications of experience. In this usage the heuristic is like an outline heading, a guide to the detailed discussion that follows. In this case the stories behind the heuristics can be more important than the basic statement. The heuristic is a pedagogical tool, a way of teaching lessons not well captured in other engineering teaching methods.

The third usage, of special interest in this second edition, is the most structured. It is when heuristics are integrated into development processes. A good example is in software. A number of software development methods have a sequence of models, from relatively abstract to code in a programming language. Object-oriented methods, for example, usually begin with a set of textual requirements: build a model of classes and objects, and then refine the class/object model into code in the target programming environment. There are often intermediate steps in which the problem domain derived objects are augmented with objects and characteristics from the target environment. A problem in all such methods is knowing how to construct the models at each step. The transformation from a set of textual requirements to classes and objects is not unique; it involves extensive judgment by the practitioner. Some methods provide assistance to the practitioner by giving explicit, prescriptive heuristics for each step.

Heuristics on heuristics

A phenomenon observed as heuristics was discovered by the USC graduate students when the discoverers themselves began thinking heuristically. They found themselves creating heuristics directly from observation and discussion, then trying them out on professional architects and engineers, some of whose experiences had suggested them. (Most interviewees were surprised and pleased at the results.) The resultant provisional heuristics were then submitted for academic review as parts of class assignments.

Kenneth L. Cureton, carrying the process one step further, generated a set of heuristics on how to generate and apply heuristics,[5] from which the following were chosen.

Generating useful heuristics

- Humor (and careful choice of words) in a heuristic provide an emotional bite that enhances the mnemonic effect (Karklins).
- Use words that transmit the "thrill of insight" into the mind of the beholder.
- For maximum effect, try embedding both descriptive and prescriptive messages in a heuristic.
- Many heuristics can be applied to heuristics themselves; e.g., **Simplify. Scope.**

- Don't make a heuristic so elegant that it only has meaning to its creator, thus losing general usefulness.
- Rather than adding a conditional statement to a heuristic, consider creating a separate but associated heuristic that focuses on the insight of dealing with that conditional situation.

Applying heuristics

- If it works, then it's useful.
- Knowing when and how to use a heuristic is as important as knowing the what and why.
- Heuristics work best when applied early to reduce the solution space.
- Strive for balance — too much of a good thing or complete elimination of a bad thing may make things worse, not better!
- Practice, practice, practice.
- Heuristics aren't reality, either.

A taxonomy of heuristics

The second step after finding or creating individual heuristics is to organize them for easy access so that the appropriate ones are at hand for the immediate task. The collection mentioned earlier in this chapter was accordingly refined and organized by architecting task.* In some ways, the resultant list — presented in Appendix A — was self-organizing. Heuristics tended to cluster around what became recognized as basic architecting tasks. For example, although certifying is shown last and is one of the last formal phases in a waterfall, it actually occurs at many milestones as "sanity checks" are made along the way and subsystems are assembled. The tasks, elaborated in Chapter 9, pages 177–186, are

- Scoping and planning
- Modeling
- Prioritizing
- Aggregating
- Partitioning
- Integrating
- Certifying
- Assessing
- Evolving and rearchitecting

The list is further refined by distinguishing between two forms of heuristic. One form is <u>de</u>scriptive; that is, it describes a situation but does not

* The original 100 of Rechtin, 1991, were organized by the phases of a waterfall. The list in Appendix A of this book recognizes that many heuristics apply to several phases, that the spiral model of system development would in any case call for a different categorization, and that many of the tasks described here occur over and over again during systems development.

indicate directly what to do about it. Another is prescriptive; that is, it prescribes what might be done about the situation. An effort has been made in the appendix to group prescriptions under appropriate descriptions with some, but not complete, success. Even so, there are more than enough generally applicable heuristics for the reader to get started.

Then there are sets of heuristics that are domain-specific to aircraft, spacecraft, software, manufacturing, social systems, etc. Some of these can be deduced or specialized from more general ones given here; or, they can be induced or generalized from multiple examples in specialized subdomains. Still more fields are explored in Part Three, adding further heuristics to the general list.

The readers are encouraged to discover still more, general and specialized, in much the same way the more general ones here were — by spotting them in technical journals, books,[6] project reports, management treatises, and conversations.

The Appendix A taxonomy is not the only possible organizing scheme any more than all tool stores are organized in the same way. In Appendix A, one heuristic follows another, one dimensionally, as in any list; but some are connected to others in different categories, or could just as easily be placed there. Some are "close" to others and some are further away. Dr. Ray Madachy, then a graduate student, using hypertext linking, converted the list into a two-dimensional, interconnected "map" in which the main nodes were architecting themes: conception and design; the systems approach; quality and safety; integration, test, and certification; and disciplines.[7] To these were linked each of the 100 heuristics in the first systems architecting text,[8] which in turn were linked to each other. The ratio of heuristic-to-heuristic links to total links was about 0.2; that is, about 20% of the heuristics overlapped into other nodes.

The Madachy taxonomy, however, shared a limitation common to all hypertext methods — the lack of upward scalability into hundreds of objects — and, consequently, was not used for Appendix A. Nonetheless, it could be useful for organizing a modest-sized personal tool kit or for treating problems already posed in object-oriented form; for example, computer-aided design of spacecraft.[9]

New directions

Heuristics are a popular topic in systems and software engineering, though they don't often go by that name. A notable example is the pattern language. The idea of patterns and pattern languages comes from Christopher Alexander and has been adapted to other disciplines by other writers. Most of the applications are to software engineering.

A pattern is a specific form of prescriptive heuristic. A number of forms have been used in the literature, but all are similar. The basic form is a pattern name, a statement of a problem, and a recommended form of solution (to that problem). For example, a pattern in civil architecture has the title Masters and Apprentices, the problem statement describes the need for junior workers to learn from senior master workers while working and the recommended solution consists of suitable arrangements of work spaces.

When a number of patterns in the same domain are collected together they can form a pattern language. The idea of a pattern language is that it can be used as a tool for synthesizing complete solutions. The architect and client use the collected problem statements to choose a set that is well-matched to the client's concerns. The resulting collection of recommended solutions is a collection of fragments of a complete solution. It is the job of the architect to harmoniously combine the fragments into a whole.

In general, domain-specific, prescriptive heuristics are the easiest for apprentices to explain and use. So, patterns on coding in programming are relatively easy to teach and to learn to use. This is borne out by the observed utility of coding pattern books in university programming courses. Similarly, an easy entry to the use of heuristics in when they are attached as step-by-step guides in a structured development process. At the opposite end, descriptive heuristics on general systems architecting are the hardest to explain and use. They typically require the most experience and knowledge to apply successfully. The catalog of heuristics in Appendix A has heuristics across the spectrum.

Summary

Heuristics, as abstractions of experience, are trusted, nonanalytic guidelines for treating complex, inherently unbounded, ill-structured problems. They are used as aids in decision making, value judgments, and assessments. They are found throughout systems architecting, from earliest conceptualization through diagnosis and operation. They provide bridges between client and builder, concept and implementation, synthesis and analysis, and system and subsystem. They provide the successive transitions from qualitative, provisional needs to descriptive and prescriptive guidelines and, hence, to rational approaches and methods.

This chapter has introduced the concept of heuristics as tools — how to find, create, organize, and use them for treating the qualitative problems of systems architecting. Appendix A provides a ready source of them organized by architecting task; in effect, a tool store of systems architecting heuristic tools.

Notes and references

1. For more of the philosophical basis of heuristics, see Asato, M., The Power of the Heuristic, USC, 1988; and Rowe, A. J., The meta logic of cognitively based heuristics, in *Expert Systems in Business and Finance: Issues and Applications*, Watkins, P. R. and Eliot, L. B., Eds., John Wiley & Sons, New York, 1988.
2. Williams, P. L., 1992 Systems Architecting Report, University of Southern California, unpublished.

3. Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991, 312.
4. Rechtin, E.,Ed., Collection of Student Heuristics in Systems Architecting, 1988-93, University of Southern California, unpublished, 1994.
5. Cureton, K. L., Metaheuristics, USC Graduate Report, December 9, 1991.
6. Many of the ones in Appendix A come from a similar appendix in Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991, Appendix.
7. Madachy, R. Thread Map of Architecting Heuristics, USC 4/21/91; and Formulating Systems Architecting Heuristics for Hypertext, USC, April 29, 1991.
8. Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991, Appendix A.
9. As an example, see Asato, M., Final Report. Spacecraft Design and Cost Model, USC, 1989.

Part two

New domains, new insights

Part Two explores, from an architectural point of view, five domains beyond those of aerospace and electronics, the sources of most examples and writings to date. The chapters can be read for several purposes. For a reader familiar with a domain, there are broadly applicable heuristics for more effective architecting of its products. For ones unfamiliar with domains, there are insights to be gained from understanding problems differing in the degree but not in kind from one's own. To coin a metaphor, if the domains can be seen as planets, then this part of the book corresponds to comparative planetology — the exploration of other worlds to benefit one's own. The chapters can be read for still another purpose: as a template for exploring other equally instructive domains. An exercise for that purpose can be found at the end of Chapter 7, Collaborative Systems.

From an educational point of view, Part Two is a recognition that one of the best ways of learning is by example, even if the example is in a different field or domain. One of the best ways of understanding another discipline is to be given examples of problems it solves; and one of the best ways of learning architecting is to recognize that there are architects in every domain and at every level from which others can learn and with whom all can work. At the most fundamental level, all speak the same language and carry out the same process: systems architecting. Only the examples are different.

Chapter 3 explores systems for which form is predetermined by a builder's perceptions of need. Such systems differ from those that are driven by client purposes by finding their end purpose only if they succeed in the marketplace. The uncertainty of end purpose has risks and consequences which are the responsibility of architects to help reduce or exploit. Central to doing so are the protection of critical system parameters and the formation of innovative architecting teams. These systems can be either evolutionary or revolutionary. Not surprisingly, there are important differences in the architectural approach.

Chapter 4 highlights the fact that manufacturing has its own waterfall, quasi-independent of the more widely discussed product waterfall, and that these two waterfalls must intersect properly at the time of production. A spiral-to-circle model is suggested to help understand the integration of hardware and software. Ultraquality and feedback are shown to be the keys to both lean manufacturing and flexible manufacturing, with the latter needing a new information flow architecture in addition.

Chapter 5, Social Systems, introduces a number of new insights to those of the more technical domains. Economic questions and value judgments play a much stronger role here, even to the point of an outright veto of otherwise worthwhile systems. A new tension comes to center stage, one central to social systems but too often downplayed in others until too late — the tension between facts and perceptions. It is so powerful in defining success that it can virtually mandate system design and performance, solely because of how that architecture is perceived.

Chapter 6, Software Systems, serves to introduce this rapidly expanding domain as it increasingly becomes the center of almost all modern systems designs. Consequently, whether stand-alone or as part of a larger system, software systems must accommodate to continually changing technologies and product usage. In very few other domains is annual, much less monthly, wholesale replacement of a deployed system economically feasible or even considered. In point of fact, it is considered normal in software systems, precisely because of software's unique ability to continuously and rapidly evolve in response to changes in technology and user demands. Software has another special property; it can be as hard or as soft as needed. It can be hard-wired if certification must be precise and unchanging, or it can be as soft as a virtual environment molded at the will of a user. For these and other reasons, software practice is heavily dependent on heuristic guidelines and organized, layered modeling. It is a domain in which architecting development is very active, particularly in progressive modeling and rapid prototyping.

Chapter 7 introduces an old but newly significant class of systems, collaborative systems. Collaborative systems exist only because the participants actively and continuously work to keep it in existence. A collaborative system is a dynamic assemblage of independently owned and operated components, each one of which exists and fulfills its owner's purposes whether or not it is part of the assemblage. These systems have been around for centuries in programs of public works. Today we find wholly new forms in communications (the Internet and World Wide Web), transportation (intelligent transportation systems), military (multinational reconnaissance-strike and defensive systems), and software (open source software). The architecting paradigm begins to shift in collaborative systems because the architect no longer has a single client who can make and execute decisions. The architect must now deal with more complex relationships, and he must find architectures in less familiar structures, such as architecture through communication or command protocol specification.

The nature of modern software and information-centric systems, and their central role in new complex systems, makes a natural lead-in to Part Three, Models and Representations.

chapter three

Builder-architected systems

No system can survive that doesn't serve a useful purpose.
Harry Hillaker*

Introduction: the form-first paradigm

The classical architecting paradigm is not the only way to create and build large complex systems, nor is it the only regime in which architects and architecting is important. A different architectural approach, the "form first," begins with a builder-conceived architecture in mind, rather than with a set of client-accepted purposes. Its architects are generally members of the technical staff of the company. Their client is the company itself; although the intention is to reach a customer base in the market.

Incremental development for an existing customer

Most builder-initiated architectures are variations of existing ones; as examples consider jet aircraft, personal computers, smart automobiles, and follow-on versions of existing software applications. The original architectures having proved by use to be sound, variations and extensions should be of low risk. Extensive reuse of existing modules should be expected because design assumptions, system functions, and interfaces are largely unchanged.

The architect's responsibilities remain much the same as under the classical paradigm, but with an important addition: the identification of proprietary architectural features deemed critical to maintaining competitive advantage in the marketplace. Lacking this identification, the question "who owns what?" can become so contentious for both builder and customer that product introduction can be delayed for years.

Far more important than these relatively low risks is the paradigm shift from function-to-form (purpose driven) to one of form-to-function (form driven). Unlike the classical paradigm, in form-first architecting, one's cus-

* Chief architect, General Dynamics F-16 Fighter. As stated in a USC Systems Architecting lecture, November, 1989.

tomers judge the value of the product after rather than before the product has been developed and produced, much less "test-driven." The resultant risk has spawned several risk-reduction strategies. The simplest is an early prototype demonstration to present customers, with its associated risks of premature rejection. The most recent strategy is probably the Open Source method for designing software — an "anyone can play," unbelievably open process in which anyone even marginally interested can participate, comment, submit ideas, develop software, and use the system, all at no cost to the participant. With the project being tied together by the Internet (and some unique social conventions), everyone — and particularly the builder and potential clients — knows and can judge its utility. The risk of rejection is sharply reduced at the possible cost of control of design. The open source community is a principal example of collaborative system assembly. We discuss that topic specifically in Chapter 7.

New markets for existing products

The next level of architecting intensity is reached when the builder's motivation is to reach uncertain or "latent" markets in which the unknown customer must acquire the product before judging its value. Almost certainly, the product will have to be at least partially rearchitected in cost, performance, availability, quantities produced, etc. To succeed in the new venture, architecting must be particularly alert, making suggestions or proposing options without seriously violating the constraints of an existing product line. Hewlett-Packard in the 1980s developed this architecting technique in a novel way. Within a given product line, say that of a "smart" analytic instrument, a small set of feasible "reference" architectures are created, each of which is intended to appeal to a different kind of customer. Small changes in that architecture then enable tailoring to customer-expressed priorities. Latent markets discovered in the process can then be quickly exploited by expansion of the product line.

The original product line architecture can be maintained with few modifications or risks until a completed system is offered to the market. Because buyers determine the real value of a system, end purpose and survival are determined last, not first.

New products, new markets

Of greatest risk are those form-first, technology-driven systems that create major qualitative changes in system-level behavior, changes in kind rather than of degree. Systems of this type almost invariably require across-the-board new starts in design, development, and use. They most often arise when radically new technologies become available such as jet engines, new materials, microprocessors, lasers, software architectures, and intelligent machines. Although new technologies are infamous for creating unpleasant technological and even sociological surprises, by far the greatest single risk in these systems is one of timing. Even if the form is feasible, introducing a new product either too early or too late can be punishing. Douglas Aircraft Company was too late into jet aircraft, losing out for years to the Boeing Company. Innumerable small companies have been too early, unable to sustain themselves while waiting for the technologies to evolve into engineered products. High tech defense systems, most often due to a premature commitment to a critical new technology, have suffered serious cost overruns and delays.

Technological substitutions within existing systems

The second greatest single risk is in not recognizing that before they are completed, technology-driven architectures will require much more than just replacing components of an older technology one at a time. Painful experience shows that without widespread changes in the system and its management, technology-driven initiatives seldom meet expectations and too often cost more for less value. As examples, direct replacements of factory workers with machines,[1] of vacuum tubes with transistors, of large inventories with Just-In-Time deliveries, and of experienced analysts with computerized management information systems, all collapsed when attempted by themselves in systems that were otherwise unchanged. They succeeded only when incorporated in concert with other matched and planned changes. It is not much of an exaggeration to say that the latter successes were well architected, the former failures were not.

In automobiles, the most recent and continuing change is the insertion of ultraquality electronics and software between the driver and the mechanical subsystems of the car. This remarkably rapid evolution removes the driver almost completely from contact with, or direct physical control of, those subsystems. It considerably changes such overall system characteristics as fuel consumption, aerodynamic styling, driving performance, safety, and servicing and repair, as well as the design of such possibly unexpected elements as engines, transmissions, tires, dashboards, seats, passenger restraints, and freeway exits. As a point of fact, the automotive industry expects that by the turn of the century more than 93% of all automotive equipment will be computer-controlled,[2] a trend evidently welcomed and used by the general public. A telling indicator of the public's perception of automotive performance and safety is the recent, virtually undisputed increase in national speed limits. Safe, long-distance highway travel at 70 mph (117 km/h) was rare, even dangerous, a decade ago. Even if the highways were designed for it, conventional cars and trucks were not. It is now common, safe, and legal. Perhaps the most remarkable fact about this rapid evolution is that most customers were never aware of it. This result came from a commitment to quality so high that a much more complex system could be offered that, contrary to the usual experience, worked far better than its simpler predecessor.

In aircraft, an equivalent, equally rapid, technology-driven evolution is "fly by wire," a change that, among other things, is forcing a social revolution in the role of the pilot and in methods of air traffic control. More is involved than the form-fit-function replacement of mechanical devices with a combination of electrical, hydraulic, and pneumatic units. Aerodynamically stable aircraft, which maintain steady flight with nearly all controls inoperative, are steadily being replaced with ones which are less stable, more maneuverable, and computer-controlled in all but emergency conditions. The gain is more efficient, potentially safer flight; but the transition has been as difficult as that between visual- and instrument-controlled flight.

In inventory control, a remarkable innovation has been the very profitable combination in one system of point-of-sale terminals, of a shift of inventory to central warehouses, and of Just-In-Time deliveries to the buyer. Note the word *combination*. None of the components has been particularly successful by itself. The risk here is greater susceptibility to interruption of supply or transportation during crises.

In communications, satellites, packet switching, high-speed fiber optic lines, e-mail, the World Wide Web, and electronic commerce have combined for easier access to a global community, but with increasing concerns about privacy and security. The innovations now driving the communications revolution were not, individually, sufficient to create this revolution. It has been the interaction of the innovations, and the changes in business processes and personal habits connected to them, that have made the revolution.

In all of these examples, far more is affected than product internals. Affected also are such externals as manufacturing management, equity financing, government regulations, and the minimization of environmental impact — to name but a few. These externals alone could explain the growing interest by innovative builders in the tools and techniques of systems architecting. How else can a well-balanced, well-integrated, financially successful, and socially acceptable total system be created?

Consequences of uncertainty of end purpose

Uncertainty of end purpose, no matter what the reason, can have serious consequences. The most serious is the likelihood of serious error in decisions affecting system design, development, and production. Builder-architected systems are often solutions looking for a problem and, hence, are particularly vulnerable to the infamous "error of the third kind": working on the wrong problem.

Uncertainty in system purposes also weakens them as criteria for design management. Unless a well-understood basis for configuration control exists and can be enforced, system architectures can be forced off course by accommodations to crises of the moment. Some of the most expensive cases of record have been in attempts to computerize management information systems. Lacking clear statements of business purposes and market priorities, irreversible *ad hoc* decisions were made which so affected their performance, cost, and schedule that the systems were scrapped. Arguably, the best prevention against "system drift" is to decide on provisional or baseline purposes and stick to them. But what if those baseline purposes prove to be wrong in the marketplace?

Reducing the risks of uncertainty of end purpose

A powerful architecting guide to protect against the risk of uncertain purposes is to **build in and maintain options.** With options available, early decisions can be modified or changed later. One possibility is to build in options to stop at known points to guarantee at least partial satisfaction of user purposes without serious losses in time and money; for example, in databases for accounting and personnel administration. Another possibility is to create architectural options that permit later additions, a favorite strategy for automobiles and trucks. Provisions for doing so are hooks in software to add applications and peripherals, scars in aircraft to add range and seats, shunts in electrical systems to isolate troubled sections, contingency plans in tours to accommodate cancellations, and forgiving exits from highways to minimize accidents.

In software, a general strategy is **use open architectures. You will need them once the market starts to respond.** As will be seen, a further refinement of this domain-specific heuristic will be needed, but this simpler version makes the point for now.

And then there is the always welcome heuristic: every once in a while, **pause and reflect.** Reexamine the cost effectiveness of system features such as high-precision pointing for weather satellites or crosstalk levels for tactical communication satellites.* Review why interfaces were placed where they were. Check for unstated assumptions such as the cold war continuing indefinitely** or the Sixties Generation turning conservative as it grew older.

Risk management by intermediate goals

Another strategy to reduce risk in the development of system-critical technologies is by scheduling a series of intermediate goals to be reached by precursor or partial configurations. For example, build simulators or prototypes to tie together and synchronize otherwise disparate research efforts.[3] Building partial systems, demonstrators, or models to help assess the sensitivity of customer acceptance to the builder's or architect's value judgments[4] is a widely used market research technique. As will be seen in Chapter 4, if these goals result in stable intermediate forms, they can be powerful tools for integrating hardware and software.

* In real life, both features proved to be unnecessary but could not be eliminated by the time that truth was discovered.

** A half-joking question in defense planning circles in the early 1980s used to be, "What if peace broke out?" Five years later, it did.

Clearly, precursor systems have to be almost as well architected as the final product. If not, their failure in front of a prospective customer can play havoc with future acceptance and ruin any market research program. As one heuristic derived from military programs warns **"the probability of an untimely failure increases with the weight of brass in the vicinity."** If precursors and demonstrators are to work well "in public," they had better be well designed and well built.

Even if a demonstration of a precursor succeeds, it can generate excessive confidence, particularly if an untested requirement is critical. In one case, a USAF satellite control system successfully and very publicly demonstrated the ability to manage one satellite at a time; the critical task, however, was to control multiple, different satellites, a test which it subsequently flunked. Massive changes in the system as a whole were required. In a similar case, a small launch vehicle, arguably successful as a high-altitude demonstrator of single-stage-to-orbit, could not be scaled up to full size or full capability for embarrassingly basic mechanical and materials reasons.

These kinds of experiences led to the admonition: **do the hard parts first.** This is an extraordinarily difficult heuristic to satisfy if the hard part is a unique function of the system as a whole. Such has been the case for a near impenetrable missile defense system, stealthy aircraft, a general aviation air traffic control system, a computer operating system, and a national tax reporting system. The only credible precursor, to demonstrate the hard parts, had to be almost as complete as the final product.

In risk management terms, if the hard parts are, perhaps necessarily, left to last, then the risk level remains high and uncertain to the very end. The justification for the system therefore must be very high and the support for it very strong or its completion will be unlikely. For private businesses this means high-risk venture capital. For governments it means support by the political process, a factor in system acquisition for which few architects, engineers, and technical managers are prepared. Chapter 12, The Political Process and Systems Architecting, is a primer on the subject.

The "what next?" quandary

One of the most serious long-term risks faced by a builder of a successful, technology-driven system is the lack of, or failure to win a competition for, a successor or follow-on to the original success.

The first situation is well exemplified by a start-up company's lack of a successor to its first product. Lacking the resources in its early, profitless years to support more than one research and development effort, it could only watch helplessly as competitors caught up and passed it by. Ironically, the more successful the initial product, the more competition it will attract from established and well-funded producers anxious to profit from a sure thing. Soon the company's first product will be a "commodity," something which many companies can produce at a rapidly decreasing cost and risk. Unable to repeat the first success, the start-up enterprise fails or is bought up at fire-sale prices when the innovator can no longer meet payroll. Common. Sad. Avoidable? Possibly.

The second situation is the all-too-frequent inability of a well-established company which had been successfully supplying a market-valued system to win contracts for its follow-on. In this instance, the very strength of the successful system, a fine architecture matched with an efficient organization to build it, can be its weakness in a time of changing technologies and shifting market needs. The assumptions and constraints of the present architecture can become so ingrained in the thinking of participants that options simply don't surface. In both situations the problem is largely architectural, as is its alleviation.

For the innovative company, it is a matter of control of critical architectural features. For the successful first producer, it is a matter of knowing, well ahead of time, when purposes have changed enough that major re-architecting may be required. Each situation will be considered in turn.

Controlling the critical features of the architecture

The critical part of the answer to the start-up company's "what next" quandary is control of the architecture of its product through proprietary ownership of its basic features.[5] Examples of such features are computer operating systems, interface characteristics, communication protocols, microchip configurations, proprietary materials, and unique and expensive manufacturing capabilities. Good products, while certainly necessary, are not sufficient. They must also arrive on the market as a steadily improving product line, one that establishes, de facto, an architectural standard.

Surprisingly, one way to achieve that objective is to use the competition instead of fighting it. Because success invites competition, it may well be better for a start-up to make its competition dependent, through licensing, upon a company-proprietary architecture rather than to have it incentivized to seek architectural alternatives. Finding architectural alternatives takes time; but licensing encourages the competition to find new applications, add peripherals, and develop markets, further strengthening the architectural base, adding to the source company's profits and its own development base.[6] Heuristically:

Successful architectures are proprietary, but open.*

This strategy was well exemplified by Microsoft in opening and licensing its personal computer (PC) operating system while Apple refused to do so for its Macintosh. The resultant widespread cloning of the PC expanded not only the market as a whole, but Microsoft's share of it. The Apple share dropped. The dangers of operating in this kind of open environment, however, are also illustrated in the case of PC hardware. The PC standard proved

* "Open" here means adaptable, friendly to add-ons, and selectively expandable in capability.

much more open than IBM intended. Where it was assumed they could maintain a price advantage through the economies of scale, the advantage disappeared. The commiditization of the PC also drove down profit margins until even a large share proved substantially unprofitable, at least for a company structured as IBM. IBM struggled for years (unsuccessfully) to move the PC market in a direction that would allow it to retain some degree of proprietary control and return profits. In contrast, Microsoft and Intel have struck a tremendously profitable balance between proprietary protection and openness. The Intel instruction set architecture has been copied, but no other company has been able to achieve a market share close to Intel's. Microsoft has grown both through proprietary and open competition, the former in operating systems and the latter in application programs.

A different kind of architectural control is exemplified by the Bell telephone system with its technology generated by the Bell Laboratories, its equipment produced largely by Western Electric, and its architectural standards maintained by usage and regulation. Others include Xerox in copiers, Kodak in cameras, Hewlett-Packard in instruments, etc. All of these product line companies began small, controlled the basic features, and prospered.

Thus, for the innovator the essentials for continued success are not only a good product, but also the generation, recognition, and control of its basic architectural features. Without these essentials there may never be a successor product. With them, many product architectures, as architecturally controlled product lines, have lasted for years following the initial success, which adds even more meaning to **there's nothing like being the first success.**[7]

Abandonment of an obsolete architecture

A different risk reduction strategy is needed for the company which has established and successfully controlled a product line architecture[8] and its market, but is losing out to a successor architecture that is proving to be better in performance, cost, and/or schedule. There are many ways that this can happen. Perhaps the purposes that original architecture has satisfied can be done better in other ways. Typewriters have largely been replaced by personal computers. Perhaps the conceptual assumptions of the original architecture no longer hold. Energy may no longer be cheap. Perhaps competitors found a way of bypassing the original architectural controls with a different architecture. Personal computers destroyed the market for Wang word processors. The desktop metaphor for personal computers revolutionized their user friendliness and their market. And, as a final example, cost risk considerations precluded building larger and larger spacecraft for the exploration of the solar system.

To avoid being superceded architecturally requires a strategy, worked out well ahead of time, to set to one side or cannibalize that first architecture, *including the organization matched with it,* and to take preemptive action to create a new one. The key move is the well-timed establishment of an innovative architecting team, unhindered by past success and capable of creating a successful replacement. Just such a strategy was undertaken by Xerox in a remake of the corporation as it saw its copier architecture start to fade. It thereby redefined itself as "The Document Company."[9]

Creating innovative teams

Clearly the personalities of members of any team, particularly an innovative architecting team, must be compatible. A series of USC Research Reports[10] by Jonathan Losk, Tom Pieronek, Kenneth Cureton, and Norman P. Geis, based on the Myers-Briggs Type Indicator (MBTI), strongly suggests that the preferred personality type for architecting team membership is NT.[11] That is, members should tend toward systematic and strategic analysis in solving problems. As Cureton summarizes, "Systems architects are made and not born, but some people are more equal than others in terms of natural ability for the systems architecting process, and MBTI seems to be an effective measure of such natural ability. No single personality type appears to be the 'perfect' systems architect, but the INTP personality type often possesses many of the necessary skills."

Their work also shows the need for later including an ENTP, a "field marshal" or deputy project manager, not only to add some practicality to the philosophical bent of the INTPs, but to help the architecting team work smoothly with the teams responsible for building the system itself.

Creating *innovative* teams is not easy, even if the members work well together. The start-up company, having little choice, depends on good fortune in its recruiting of charter members. The established company, to put it bluntly, has to be willing to change how it is organized and staffed from the top, down based almost solely on the conclusions of a presumably innovative team of "outsiders," albeit individuals chartered to be such. The charter is a critical element, not so much in defining new directions as in defining freedoms, rights of access, constraints, responsibilities, and prerogatives for the team. For example, can the team go outside the company for ideas, membership, and such options as corporate acquisition? To whom does the team had respond and report, and to whom does it not? Obviously, the architecting team had better be well designed and managed. Remember, if the team does not succeed in presenting a new and accepted architecture, the company may well fail.

One of the more arguable statements about architecting is the one by Frederick P. Brooks, Jr. and Robert Spinrad that **"the best architectures are the product of a single mind."** For modest-sized projects that statement is reasonable enough, but not for larger ones. The complexity and work load of creating large, multidisciplinary, technology-driven architectures would overwhelm any individual. The observation of a single mind is most easily accommodated by a simple but subtle change from "a single mind" to "a team of a single mind." Some would say "of a single vision" composed of ideas, purposes, concepts, presumptions, and priorities.

In the simplest case, the single vision would be that of the chief architect and the team would work to it. For practical as well as team cohesiveness reasons, the single vision needs to be a shared one. In no system is that more important than in the entreprenurially motivated one. There will always be a tension between the more thoughtful architect and the more action-oriented entrepreneur. Fortunately, achieving balance and compromise of their natural inclinations works in the system's favor.

An important corollary of the shared vision is that the architecting team itself, and not just the chief architect, must be seen as creative, communicative, respected, and of a single mind about the system-to-be. Only then can the team be credible in fulfilling its responsibilities to the entrepreneur, the builder, the system, and its many stakeholders. Internal power struggles, basic disagreements on system purpose and values, and advocacies of special interests can only be damaging to that credibility.

As Ben Bauermeister, Harry Hillaker, Archie Mills, Bob Spinrad,[12] and other friends have stressed in conversations with the authors, innovative teams need to be cultural in form, diverse in nature, and almost obsessive in dedication.

Cultural is meant as a team characterized by informal creativity, easy interpersonal relationships, trust, and respect — all characteristics necessary for team efficiency, exchange of ideas, and personal identification with a shared vision. To identify with a vision, they must deeply believe in it and in their chief. The members either acknowledge and follow the lead of their chief or the team disintegrates.

Diversity in specialization is to be expected; it is one of the reasons for forming a team. Equally important, a balanced diversity of style and programmatic experience is necessary to assure open-mindedness, to spark creative thinking in others, and to enliven personal interrelationships. It is necessary, too, to avoid the "groupthink" of nearly identical members with the same background, interests, personal style, and devotion to past architectures and programs. Indeed, team diversity is one of the better protections against the second-product risks mentioned earlier.

Consequently, an increasingly accepted guideline is that, to be truly innovative and competitive in today's world, **the team that created and built a presently successful product is often the best one for its evolution — but seldom for creating its replacement.**

A major challenge for the architect, whether as an individual or as the leader of a small architecting team, is to maintain dedication and momentum not only within the team but also within the managerial structure essential for its support. The vision will need to be continually restated as new participants and stakeholders arrive on the scene such as engineers, managers active and displaced, producers, users, even new clients. Even more difficult, it will have to be transformed as the system proceeds from a dream to a concrete entity, to a profit maker, and finally to a quality production. Cultural collegiality will have to give way to the primacy of the bottom line and, finally, to the necessarily bureaucratic discipline of production. Yet the integrity of the vision must never be lost or the system will die.

The role of organizations in architectures, and the architecture of organizations, is taken up at much greater length by one of the present authors.[13]

Architecting "revolutionary" systems

A key distinction to be made at this point is between architecting in precedented, or evolutionary, environments, and architecting unprecedented systems. One of the most notable features of the first book by one of the present authors[14] was an examination of the architectural history of clearly successful and unprecedented systems. A central observation is that all such systems have a clearly identifiable architect or small architect team. They were not conceived by the consensus of a committee. Their basic choices reflect a unified and coherent vision of one individual or a very small group. Further reflection, and study by students, has only reinforced this basic conclusion, while also filling in some of the more subtle details.

Unprecedented systems have been both purpose- and technology-driven. In the purpose-driven case the architect has sometimes been part of the developer's organization and sometimes not. In the technology-driven case the architect is almost always in the developer's organization. This should be expected as technology-driven systems typically come from intimate knowledge of emerging technology, and someone's vision of where it can be applied to advantage.[15] This person is typically not a current user, but is rather a technology developer. It is this case that is the concern of this section.

The architect has a lead technical role, but this role cannot be properly expressed in the absence of good project management. Thus the pattern of a strong duo, project manager and system architect, is also characteristic of successful systems. In systems of significant complexity it is very difficult to combine the two roles. A project manager is typically besieged by short-term problems. The median due date of things on the project manager's desk is probably yesterday. In this environment of immediate problems it is unlikely that a person will be able to devote the serious time to longer-term thinking and broad communicating that are essential to good architecting.

The most important lesson in revolutionary systems, at least those which are not inextricably tied to a single mission, is that success is commonly not found where the original concept thought it would be. The Macintosh computer was a success because of desktop publishing, not the market assumed in its original rollout (which was as a personal information appliance). Indeed, desktop publishing did not exist as a significant market when the Macintosh was introduced. This pattern of new systems becoming successful because of new applications has been common enough in the computer industry to have acquired a nickname, the "killer app(lication)." Taken narrowly, a killer app is an application so valuable that it drives the sales of a particular computer platform. Taken more broadly, a killer app is any new system usage so valuable that, by itself, it drives the dissemination of the system.

One approach to unprecedented systems is to seek the killer application that can drive the success of a system. A recent noncomputer example that illustrates the need, and the difficulty, is the search for a killer application for reusable space launch vehicles. Proponents believe that there is a stable economic equilibrium with launch costs an order of magnitude lower, and flight rates around an order of magnitude higher, than currently used. But, if flight rates increase and space payload costs remain the same, then total spending on space systems will be far higher. For there to be a justification for high flight rate launch there has to be an application that will realistically exploit it.

Various proposals have been floated, including large constellations of communication satellites, space power generation, and space tourism. If the cost of robotic payloads was reduced at the same time, their flight rate might increase without total spending going up so much. The only clear way of doing that is to move to much larger-scale serial production of space hardware to take advantage of learning curve cost reductions.[16] This clearly indicates a radical change to the architecture not only of launch, but to satellite design, satellite operations, and probably to space manufacturing companies as well; and all of these changes need to take place synchronously for the happy consequence of lowered cost to result. Thus far, this line of reasoning has not produced success. Launches remain expensive, and the most efficient course appears to be greater reliability and greater functionality per pound of payload.

Sometimes such synchronized changes do occur. The semiconductor industry has experienced decades of 40% annual growth because such synchronized changes have become ingrained in the structure of the computer industry. The successful architect exploits what the market demonstrates as the killer application. The successful innovator exploits the first-to-market position to take advantage of the market's demonstration of what it really wants faster than does the second-to-market player. The successful follower beats the first-to-market by being able to exploit the market's demonstration more quickly.

Systems architecting and basic research

One other relationship should be established, that between architects and those engaged in basic research and technology development. Each group can further the interests of the other. The architect can learn without conflict of interest. The researcher is more likely to become aware of potential sponsors and users.

New technologies enable new architectures, though not singly nor by themselves. Consider solid state electronics, fiber optics, software languages, and molecular resonance imaging, for starters. Also, innovative architectures provide the rationale for underwriting research, often at a very basic level.

Although both innovative architecting and basic research explore the unknown and unprecedented, there seems to be little early contact between their respective architects and researchers. The architectures of intelligent machines, the chaotic aerodynamics of active surfaces, the sociology of intelligent transportation systems, and the resolution of conflict in multimedia networks are examples of presumably common interests. Universities might well provide a natural meeting place for seminars, consulting, and the creation and exchange of tools and techniques.

New architectures, driven by perceived purposes, sponsor more basic research and technology development than is generally acknowledged. Indeed, support for targeted basic research undoubtedly exceeds that motivated by scientific inquiry. Examples abound in communications systems which sponsor coding theory, weapons systems which sponsor materials science and electromagnetics, aircraft which sponsor fluid mechanics, and space systems which sponsor the fields of knowledge acquisition and understanding.

It is therefore very much in the mutual interest of professionals in R&D and systems architecting to know each other well. Architects gain new options while researchers gain well-motivated support. Enough said.

Heuristics for architecting technology-driven systems

General

- An insight is worth a thousand market surveys.
- Success is defined by the customer, not by the architect.
- In architecting a new program, all the serious mistakes are made in the first day.
- The most dangerous assumptions are the unstated ones.
- The choice between products may well depend upon which set of drawbacks the users can handle best.
- As time to delivery decreases, the threat to user utility increases.
- If you think your design is perfect, it's only because you haven't shown it to someone else.
- If you don't understand the existing system, you can't be sure you are building a better one.
- Do the hard parts first.
- Watch out for domain-specific systems. They may become traps instead of useful system niches, especially in an era of rapidly developing technology.
- The team that created and built a presently successful product is often the best one for its evolution, but seldom the best one for creating its replacement. (It may be locked into unstated assumptions that no longer hold.)

Specialized

From Morris and Ferguson, 1993:

- Good products are not enough. (Their features need to be owned.)
- Implementations matter. (They help establish architectural control.)
- Successful architectures are proprietary, but open. (Maintain control over the key standards, protocols, etc., that characterize them, but make them available to others who can expand the market to everyone's gain.)

From Chapters 2 and 3:

- Use open architectures. You will need them once the market starts to respond.

Summary

Technology-driven, builder-architected systems, with their greater uncertainty of customer acceptance, encounter greater architectural risks than those that are purpose-driven. Risks can be reduced by the careful inclusion of options, the structuring of innovative teams, and the application of heuristics found useful elsewhere. At the same time, they have lessons to teach in the control of critical system features and the response to competition enabled by new technologies.

Exercises

1. The architect can have one of three relationships to the builder and client. The architect can be a third party, can be the builder, or can be the client. What are the advantages and disadvantages of each relationship? For what type of system is one of the three relationships necessary?
2. In a system familiar to you, discuss how the architecture can allow for options to respond to changes in client demands. Discuss the pros and cons of product vs. product-line architecture as strategies in responding to the need for options. Find examples among systems familiar to you.
3. Architects must be employed by builders in commercially marketed systems because many customers are unwilling to sponsor long-term development; they purchase systems after evaluating the finished product according to their then perceived needs. But placing the architect in the builder's organization will tend to dilute the independence needed by the architect. What organizational approaches can

help to maintain independence while also meeting the needs of the builder organization?
4. The most difficult type of technology-driven system is one that does not address any existing market. Examine the history of both successful and failed systems of this type. What lessons can be extracted from them?

Notes and references

1. Majchrzak, A., *The Human Side of Automation,* Jossey-Bass Publishers, San Francisco, 1988, pp. 95-102. The challenge: 50-75% of the attempts at introducing advanced manufacturing technology into U.S. factories are unsuccessful, primarily due to lack of human resource planning and management.
2. Automobile Club of Southern California speaker, Los Angeles, August 1995.
3. The Goldstone California planetary radar system of the early 1960s tied together antenna, transmitter, receiver, signal coding, navigation, and communications research and development programs. All had to be completed and integrated into a working radar at the fixed dates when target planets were closest to the Earth, and well in advance of commitment for system support of communication and navigation for an as-yet-not-designed spacecraft to be exploring the solar system. Similar research coordination can be found in NASA aircraft, applications software, and other rapid prototyping efforts.
4. Hewlett-Packard marketers have long tested customer acceptance of proposed instrument functions by presenting just the front panel and asking for comment. Both parties understood that what was behind the panel had yet to be developed, but could be. A more recent adaptation of that technique is the presentation of "reference architectures."
5. Morris, C. R. and Ferguson, C. H., How architecture wins technology wars, *Harvard Business Review,* March–April 1993; and its notably supportive follow-up review, Will architecture win the technology wars?, *Harvard Business Review,* May-June 1993. A seminal article on the subject.
6. Morris, C. R. and Ferguson, C. H.
7. Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991, 301.
8. It is important to distinguish between a product line architecture and a product in that line. It is the first of these that is the subject here. Hewlett-Packard is justly famous for the continuing creation of innovative products to the point where half or less of their products on the market were there five years earlier. Their product *lines* and their architectures, however, are much longer lived. Even so, these, too, were replaced in due course, the most famous of which was the replacement of "dumb" instruments with smart ones.
9. Described by Robert Spinrad in a 1992 lecture to a USC systems architecting class. Along with Hewlett-Packard, Xerox demonstrates that it is possible to make major architectural, or strategic, change without completely dismantling the organization. But the timing and nature of the change is critical. Again, **a product architecture and the organization that produces it must match, not only as they are created but as they are in decline.** For either of the two to be too far ahead or behind the other can be fatal.

10. Geis, N. P., Profiles of Systems Rearchitecting Teams, USC Research Report, June 1993, extends and summarizes the earlier work of Losk, Pieronek, and Cureton.
11. It is beyond the scope of this book to show how to select individuals for teams, only to mention that widely used tools are available for doing so. For further information on MBTI and the meaning of its terms see Myers, I., Briggs, K., and McCaulley, *Manual: A Guide to the Development and Use of the Myers-Briggs Type Indicator,* Consulting Psychologists Press, Palo Alto, CA, 1989.
12. Benjamin Bauermeister is an entrepreneur and architect in his own software company. He sees three successive motivations as product development proceeds: product, profit, and production corresponding roughly to cultural, bottom line, and bureaucratic as successive organization forms. Harry Hillaker is the retired architect of the Air Force F-16 fighter and a regular lecturer in systems architecting. Archie W. Mills is a middle-level manager at Rockwell International and the author of an unpublished study of the practices of several Japanese and American companies over the first decades of this century in aircraft and electronics systems. Bob Spinrad is a Xerox executive and former director of Xerox Palo Alto Research Center (PARC).
13. Rechtin, E., *Systems Architecting of Organizations, Why Eagles Can't Swim,* CRC Press, Boca Raton, FL, 1999.
14. Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991.
15. In contrast, and for reasons of demonstrated avoidance of conflict of interest, independent architects rarely suggest the use of undeveloped technology. Independent architects, interested in solving the client's purposes, are inherently conservative.
16. This isn't the first time these arguments have happened. See Rechtin, E., A Short History of Shuttle Economics, NRC Paper, April, 1983; which demonstrated this case in detail.

chapter four

Manufacturing systems

Introduction: the manufacturing domain

Although manufacturing is often treated as if it were but one step in the development of a product, it is also a major system in itself. It has its own architecture.[1] It has a system function that its elements cannot perform by themselves: making other things with machines. And it has an acquisition waterfall for its construction quite comparable to those of its products.

From an architectural point of view, manufacturing has long been a quiet field. Such changes as were required were largely a matter of continual, measurable, incremental improvement — a step at a time on a stable architectural base. Though companies came and went, it took decades to see a major change in its members. The percentage of sales devoted to research and advanced development for manufacturing, per se, was small. The need was to make the classical manufacturing architecture more effective; that is, to evolve and engineer it.

A decade or so ago the world that manufacturing had supported for almost a century changed, and at a global scale. Driven by new technologies in global communications, transportation, sources, markets, and finance, global manufacturing became practical and then, shortly thereafter, dominant. It quickly became clear that qualitative changes was required in manufacturing architectures if global competition was to be met. In the order of conception, the architectural innovations were ultraquality,[2] dynamic manufacturing,[3] lean production,[4] and "flexible manufacturing."* The results to date, demonstrated first by the Japanese, have greatly increased profits and market share, and sharply decreased inventory and time-to-market. Each of these innovations will be presented in turn.

Even so, rapid change is still underway. As seen on the manufacturing floor, manufacturing research as such has yet to have a widespread effect. Real-time software is still a rarity. Trend instrumentation, self-diagnosis, and self-correction, particularly for ultraquality systems, are far from common-

* Producing different products on demand on the same manufacturing line.

place. Thus far, the most sensitive tool for ultraquality is the failure of the product being manufactured.

Architectural innovations in manufacturing

Ultraquality systems

At the risk of oversimplification, a common perception of quality is that quality costs money; that is, that quality is a tradeoff against cost and/or profit. Not coincidentally, there is an accounting category called "cost of quality." A telling illustration of this perception is the "DeSoto story." As the story goes, a young engineer at a DeSoto automobile manufacturing plant went to his boss with a bright idea on how to make the DeSoto a more reliable automobile. The boss's reply: "Forget it, kid. If it were more reliable it would last more years and we would sell fewer of them. It's called planned obsolescence." DeSoto is no longer in business, but the perception remains in the minds of many manufacturers of many products.

The difficulty with this perception is partly traceable to the two different aspects of quality. The first is quality associated with features like leather seats and air conditioning. Yes, those features cost money, but the buyer perceives them as value added and the seller almost always *makes* money on them. The other aspect of quality is absence of defects. As it has now been shown, absence of defects *also* makes money, and for both seller and buyer, through reductions in inventory, warranty costs, repairs, documentation, testing, and time-to-market *provided that* the level of product quality is high enough* and that the whole development and production process is architected at that high level.

To understand why absence of defects makes money, imagine a faultless process that produces a product with defects so rare that it is impractical to measure them; that is, none are anticipated within the lifetime of the product. Testing can be reduced to the minimum required to certify system-level performance of the first unit. Delays and their costs can be limited to those encountered during development; if and when later defects occur, they can be promptly diagnosed and permanently eliminated. "Spares" inventory, detailed parts histories, and statistical quality control can be almost nonexistent. First-pass manufacturing yield can be almost 100% instead of today's highly disruptive 20-70%. Service in the field is little more than replacing failed units, free.

The only practical measurement of ultraquality would then be an end system-level test of the product itself. Attempting to measure defects at any subsystem level would be a waste of time and money — defects would have to be too rare to determine with high confidence. Redundancy and fault-tolerant designs would be unnecessary. Indeed, they would be impractical

* Roughly less than 1% per year rate of failure at the system level regardless of system size. The failure rate for subsystems or elements clearly must be much less.

because, without an expectation of a specific failure (which then should be fixed), protection against rare and unspecified defects is not cost effective.

To some readers, this ultraquality level may appear to be hopelessly unrealistic. Suffice to say that it has been approached for all practical purposes. For decades, no properly built unmanned satellite or spacecraft failed because of a defect known before launch (any defect would have been fixed beforehand). Microprocessors with millions of active elements, sold in the millions, now outlast the computers for which they were built. Like the satellites, they become technically and financially obsolete long before they fail. Television sets are produced with a production line yield of over 99%, far better than the 50% yield of a decade ago, with a major improvement in cost, productivity, and profit.

Today's challenge, then, is to achieve and maintain such ultraquality levels even as systems become more complex. Techniques have been developed which certainly help.* More recently, two more techniques have been demonstrated that are particularly applicable to manufacturing.

1. **Everyone in the production line is both a customer and a supplier,** a customer for the preceding worker and a supplier for the next. Its effect is to place quality assurance where it is most needed, at the source.
2. **The Five Whys,** a diagnostic procedure for finding the basic cause of a defect or discrepancy. Why did this occur? They why did that, in turn, occur. Then, why *that*?, and so on until the offending causes are discovered and eliminated.

To these techniques can be added a relatively new understanding: **some of the worst failures are system failures**; that is, they come from the interaction of subsystem deficiencies which of themselves do not produce an end system failure, but together can and do. Four catastrophic civil space system failures were of this kind: Apollo 1, Apollo 13, Challenger, and the Hubble Telescope. For Tom Clancy buffs, just such a failure almost caused World War III in his *Debt of Honor*. In all these cases, had any one of the deficiencies not occurred, the near catastrophic end result could not have occurred. That is, though each deficiency was necessary, none were sufficient for end failure. As an admonition to future failure review boards, until a diagnosis is made that indicates that the set of presumed causes are both necessary and suffi-

* Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991, Chapter 8, 160-187. One technique mentioned there — fault tolerance through redundancy — has proved to be less desirable than proponents had hoped. Because fault-tolerant designs "hide" single faults by working around them, they accumulate until an overall system failure occurs. Diagnostics then become very much more difficult. Symptoms are intertwined. Certification of complete repair cannot be guaranteed because successful-but-partial operation again hides undetected (tolerated) faults. The problem is most evident in servicing modern, microprocessor-rich automobile controls. The heuristic still holds, **fault avoidance is preferable to fault tolerance in system design.**

cient — and that no other such set exists — the discovery-and-fix process is incomplete and ultraquality is not assured.

Successful ultraquality has indeed been achieved, but there is a price that must be paid. Should ultraquality *not* be produced at any point in the whole production process, the process may collapse. Therefore, when something does go wrong, it must be fixed immediately; there are no cushions of inventory, built-in holds, full-time expertise, or planned workarounds. Because strikes and boycotts can have instantaneous effects, employee, customer, and management understanding and satisfaction is essential. Pride in work and dedication to a common cause can be of special advantage, as has been seen in the accomplishments of the zero defect programs of World War II, the American Apollo lunar landing program, and the Japanese drive to world-class products.

In a sense, ultraquality-built systems are fine-tuned to near perfection with all the risks thereof. Just how much of a cushion or insurance policy is needed for a particular system is an important value judgment that the architect must obtain from the client, the earlier the better. That judgment has strong consequences in the architecture of the manufacturing system. Clearly, then, ultraquality architectures are very different from the statistical quality assurance architectures of only a few years ago.*

Most important for what follows, it is unlikely that either lean production or flexible manufacturing can be made competitive at much less than ultraquality levels.

Dynamic manufacturing systems

The second architectural change in manufacturing systems is from comparatively static configurations to dynamic, virtual real-time, ones. Two basic architectural concepts now become much more important. The first concerns intersecting waterfalls and the second, feedback systems.

Intersecting waterfalls

The development of a manufacturing system can be represented as a separate waterfall, distinct from, and intersecting with, that of the product it makes. Figure 4.1 depicts the two waterfalls, the process (manufacturing) diagonally and the product vertically intersecting at the time and point of production. The manufacturing one is typically longer in time, often decades, and contains more steps than the relatively shorter product sequence (months to years) and may end quite differently (the plant is shut down and demolished). Sketching the product development and the manufacturing process

* One of the authors, a veteran of the space business, recently visited two different manufacturing plants and correctly predicted the plant yields (acceptance vs. start rates) by simply looking at the floors and at the titles (not content) and locations of a few performance charts on the walls. In the ultraquality case the floors were painted white, the charts featured days-since-last-defect instead of running average defect rates, and the charts were placed at the exits of each work area. Details, but indicative.

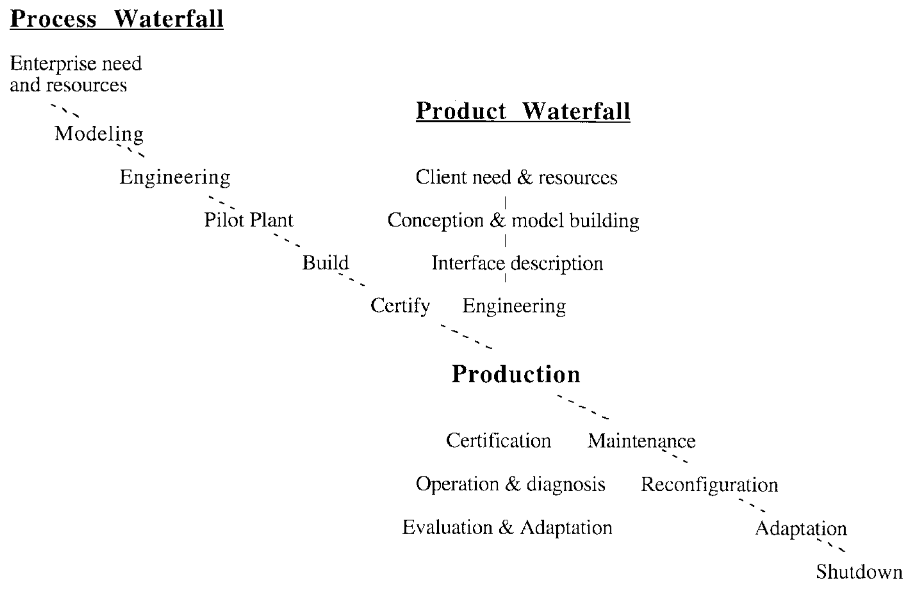


Figure 4.1

as two intersecting waterfalls helps emphasize the fact that manufacturing has its own steps, time scales, needs, and priorities distinct from those of the product waterfall. It also implies the problems its systems architect will face in maintaining system integrity, in committing well ahead to manufacture products not yet designed, and in adjusting to comparatively abrupt changes in product mix and type. A notably serious problem is managing the introduction of new technologies, safely and profitably, into an inherently high inertia operation.

There are other differences. Manufacturing certification must begin well before product certification or the former will preclude the latter; in any case, the two must interact. The product equivalent of plant demolition, not shown in the figure, is recycling, both now matters of national law in Europe. Like certification, demolition is important to plan early, given its collateral human costs in the manufacturing sector. The effects of environmental regulations, labor contracts, redistribution of useable resources, retraining, right-sizing of management, and continuing support of the customers are only a few of the manufacturing issues to be resolved — well before the profits are exhausted.

Theoretically if not explicitly, these intersecting waterfalls have existed since the beginning of mass production; but not until recently have they been perceived as having equal status, particularly in the U.S. Belatedly, that perception is changing, driven in large part by the establishment of global manufacturing — clearly not the same system as a wholly owned shop in one's own backyard. The change is magnified by the widespread use of sophisticated software in manufacturing, a boon in managing inventory but a costly burden in reactive process control.[5] Predictably, software for manufacturing process and control, more than any element of manufacturing, will determine the practicality of flexible manufacturing. As a point in proof, Hayes, Wheelright, and Clark[6] point out that a change in the architecture of (even) a mass production plant, particularly in the software for process control, can make dramatic changes in the capabilities of the plant without changing either the machinery or layout.

The development of manufacturing system software adds yet another production track. The natural development process for software generally follows a spiral,* certainly not a conventional waterfall, cycling over and over through functions, form, code (building), and test. The Figure 4.2 software spiral is typical. It is partially shaded to indicate that one cycle has been completed with at least one more to go before final test and use. One reason for such a departure from the conventional waterfall is that software, as such, requires almost no manufacturing, making the waterfall model of little use as a descriptor.** The new challenge is to synchronize the stepped waterfalls and the repeating spiral processes of software-intensive systems. One of the most efficient techniques is through the use of stable intermediate forms,[7] combining software and hardware into partial but working precursors to the final system. Their important feature is that they are *stable* configurations; that is, they are reproducible, well-documented, progressively refined baselines. In other words, they are identifiable architectural waypoints and must be treated as such. They can also act as operationally useful configurations,*** built-in "holds" allowing lagging elements to catch up, or parts of strategies for risk reduction as suggested in Chapter 3.

The spiral-to-circle model

Visualizing the synchronization technique for the intersecting waterfalls and spirals of Figure 4.2 can be made simpler by modifying the spiral so that it remains from time to time in stable concentric circles on the four-quadrant process diagram. Figure 4.3 shows a typical development from a starting point in the function quadrant cycling through all quadrants three times — typical of the conceptualization phase — to the first intermediate form. There the development may stay for a while, changing only slightly, until new functions call for the second form, say an operational prototype. In Air Force procurement, that might be a "fly-before-buy" form. In space systems, it might be a budget-constrained "operational prototype," which is actually

* See also Chapter 2.

** Efforts to represent the manufacturing and product processes as spirals have been comparably unsuccessful. Given the need to order long lead time items, to "cut metal" at some point, and to write off the cost of multiple rapid prototypes, the waterfall is the depiction of choice.

*** Many commanders of the Air Force Space and Missiles Division have insisted that all prototypes and interim configurations have at least some operational utility, if only to help increase the acceptance in the field once the final configuration is delivered. In practice, field tests of interim configurations almost always clarify if not reorder prior value judgments of what is most important to the end user in what the system can offer.

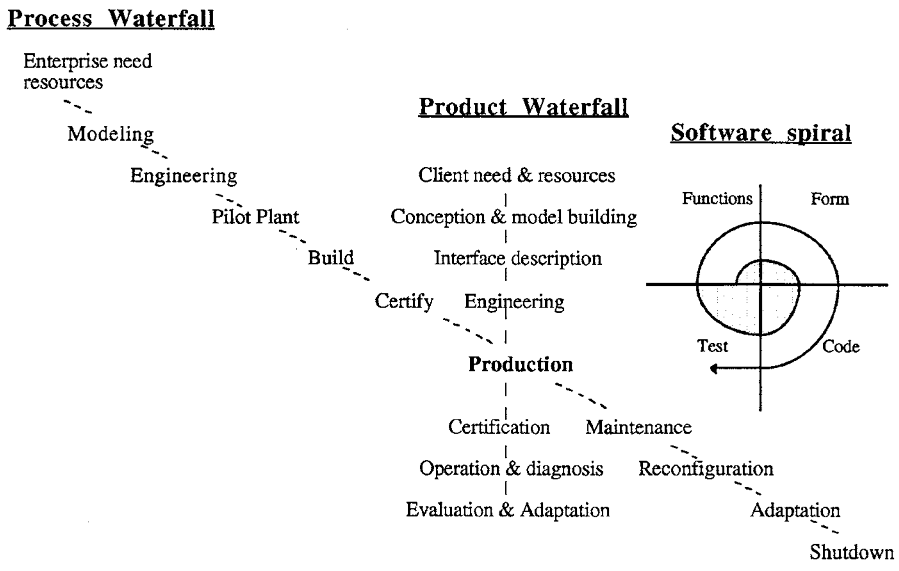


Figure 4.2

flown. In one program, it turned out to be the *only* system flown. But, to continue, the final form is gained, in this illustration, by way of a complete four-quadrant cycle.

The spiral-to-circle model can show other histories; for example, a failure to spiral to the next form, with a retreat to the preceding one, possibly with less ambitious goals, a transition to a still greater circle in a continuing evolution, or an abandonment of these particular forms with a restart near the origin.

Synchronization can also be helped by recognizing that cycling also goes on in the multistep waterfall model, except that it is depicted as feedback from one phase to one or more preceding ones. It would be quite equivalent to software quadrant spiraling if all waterfall feedback returned to the beginning of the waterfall; that is, to the system's initial purposes and functions, and from there down the waterfall again. If truly major changes are called for, the impact can be costly, of course, in the short run. The impact in the long run might be cost effective, but few hardware managers are willing to invest.

The circle-to-spiral model for software-intensive systems in effect contains both the expanding-function concept of software and the step-wise character of the waterfall. It also helps to understand what and when hardware and software functions are needed in order to satisfy requirements by the *other* part of the system. For example, a stable intermediate form of software should arrive when a stable, working form of hardware arrives that needs that software, and vice versa.

It is important to recognize that this model, with its cross-project synchronizations requirement, is notably different from models of procurements

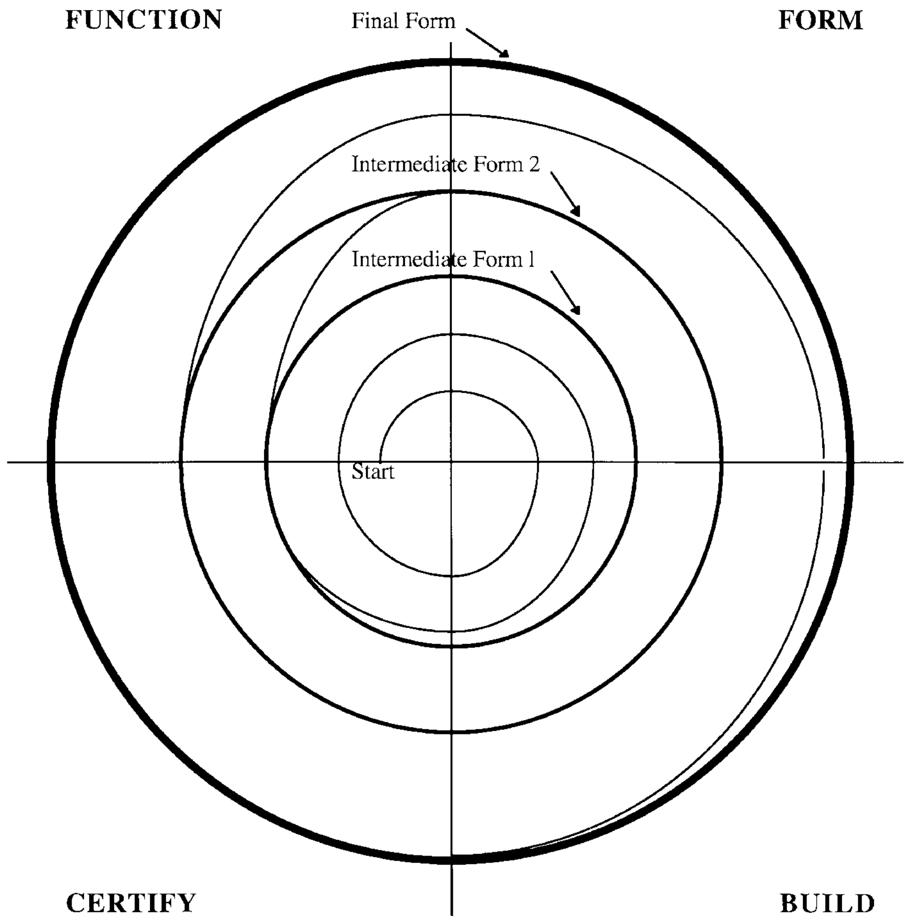


Figure 4.3

in which hardware and software developments can be relatively independent of each other. In the spiral-to-circle model, the intermediate forms, both software and hardware, must be relatively unchanging and bug-free. A software routine that never quite settles down or that depends upon the user to find its flaws is a threat, not a help, in software-intensive systems procurement. A hardware element that is intended to be replaced with a "better and faster" one later is hardly better. Too many subsequent decisions may unknowingly rest on what may turn out to be anomalous or misunderstood behavior of such elements in system test.

To close this section, although this model may be relatively new, the process that it describes is not. Stable intermediate forms, blocks (I, II, etc.), or "test articles," as they are called, are built into many system contracts and perform as intended. Yet, there remains a serious gap between most hardware and software developers in their understanding of each other and their joint venture. As the expression goes: "These guys just don't talk to each other." The modified spiral model should help both partners bridge that gap, to accept the reasons both for cycling and for steps, and to recognize that neither acquisition process can succeed without the success of the other.

There should be no illusion that the new challenge will be easy to meet. Intermediate software forms will have to enable hardware phases at specified milestones, not just satisfy separate software engineering needs. The forms must be stable, capable of holding at that point indefinitely, and practical as a stopping point in the acquisition process if necessary. Intermediate hardware architectures must have sufficient flexibility to accommodate changes in the software as well as in the hardware. And, finally, the architects and managers will have a continuing challenge in resynchronizing the several processes so that they neither fall behind nor get too far ahead. Well-architected intermediate stable forms and milestones will be essential.

Concurrent engineering

The intersecting waterfall model in Figure 4.2 also helps identify the source of some of the inherent problems in concurrent (simultaneous, parallel) engineering in which product designers and manufacturing engineers work together to create a well-built product. Concurrent engineering in practice, however, has proven to be more than modifying designs for manufacturability. However defined, it is confronted with a fundamental problem, evident from Figure 4.2; namely, coordinating the two intersecting waterfalls and the spirals, each with different time scales, priorities, hardware, software, and profit-and-loss criteria. Because each track is relatively independent of the others, the incentives for each are to optimize locally even if doing so results in an impact on another track or on the end product. After all, it is a human and organizational objective to solve one's own problems, to have authority reasonably commensurate with responsibilities, and to be successful in one's own right. Unfortunately, this objective forces even minor technical disagreements to higher, enterprise management where considerations other than just system performance come into play.

> A typical example: A communications spacecraft design was proceeding concurrently in engineering and manufacturing until the question came up of the spacecraft antenna size. The communications engineering department believed that a 14-ft diameter was needed while the manufacturing department insisted that 10 feet was the practical limit. The difference in system performance was a factor of two in communications capability and revenue. The reason for the limit, it turned out, was that the manufacturing department had a first-rate subcontractor with all the equipment needed to build an excellent antenna, but no larger than 10 feet. To go larger would cause a measurable

> manufacturing cost overrun. The manufacturing manager was adamant about staying within his budget, having taken severe criticism for an overrun in the previous project. In any case, the added communications revenue gain was far larger than the cost of re-equipping the subcontractor. Lacking a systems architect, the departments had little choice but to escalate the argument to a higher level where the larger antenna was eventually chosen and the manufacturing budget increased slightly. The design proceeded normally until software engineers wanted to add more memory well after manufacturing had invested heavily in the original computer hardware design. The argument escalated, valuable time was lost, department prerogatives were again at stake … and so it went.

This example is not uncommon. A useful management improvement would have been to set up a trusted, architect-led team to keep balancing the system as a whole within broad top-management guidelines of cost, performance, risk, and schedule.

If so established, the architectural team's membership should include a corporate-level (or "enterprise") architect, the product architect, the manufacturing architect, and a few specialists in system-critical elements and no more.[8] Such a structure does exist implicitly in some major companies, though seldom with the formal charter, role, and responsibilities of systems architecting.

Feedback systems

Manufacturers have long used feedback to better respond to change. Feedback from the customer has been, and is, used directly to maintain manufacturing quality, and indirectly to accommodate changes in design. Comparably important are paths from sales to manufacturing and from manufacturing to engineering.

What is new is that the pace has changed. Multiyear is now yearly, yearly is now monthly, monthly is now daily, and daily, especially for ultraquality systems, has become hourly if not sooner. What was a temporary slowdown is now a serious delay. What used to affect only adjacent sectors can now affect the whole. What used to be the province of planners is now a matter of real-time operations.

Consequently, accustomed delays in making design changes, correcting supply problems, responding to customer complaints, introducing new products, and reacting to competitors' actions and the like were no longer acceptable. The partner to ultraquality in achieving global competitiveness was to counter the delays by anticipating them. In other words, to use anticipatory feedback in as close to real-time as practical. The most dramatic industrial example to date has been in lean production[9] in which feedback to suppliers, coupled with ultraquality techniques, cut the costs of inventory in half and resulted in across-the-board competitive advantage in virtually all business parameters. More recently, criteria for certification, or those of its predecessor, quality assurance, is routinely fed back to conceptual design and engineering — one more recognition that **quality must be designed in, not tested in.**

A key factor in the design of any real-time feedback system is loop delay, the time it takes for a change to affect the system "loop" as a whole. In a feedback system, delay is countered by anticipation based on anticipated events (like a failure) or on a trend derived from the integration of past information. The design of the anticipation, or "correction," mechanism, usually the feedback paths, is crucial. The system as a whole can go sluggish on the one hand or oscillatory on the other. Symptoms are inventory chaos, unscheduled overtime, share price volatility, exasperated sales forces, frustrated suppliers, and, worst of all, departing long-time customers. Design questions are: What is measured? How is it processed? Where is it sent? And, of course, to what purpose?

Properly designed, feedback control systems determine transient and steady-state performance, reduce delays and resonances, alleviate nonlinearities in the production process, help control product quality, minimize inventory, and alleviate nonlinearities in the production process. By way of explanation, in nonlinear systems, two otherwise independent input changes interact with each other to produce effects different from the sum of the effects of each separately. Understandably, the end results can be confusing if not catastrophic. An example is a negotiated reduction in wages followed immediately by an increase in executive wages. The combination results in a strike; either alone would not.

A second key parameter, the resonance time constant, is a measure of the frequency at which the system or several of its elements tries to oscillate or cycle. Resonances are created in almost every feedback system. The more feedback paths, the more resonances. The business cycle, related to inventory cycling, is one such resonance. Resonances, internal and external, can interact to the point of violent, nonlinear oscillation and destruction, particularly if they have the same or related resonant frequencies. Consequently, a warning: **avoid creating the same resonance time constant in more than one location in a (production) system.**

Delay and resonance times, separately and together, are subject to design. In manufacturing systems, the factors that determine these parameters include inventory size, inventory replacement capacity, information acquisition and processing times, fabrication times, supplier capacity, and management response times. All can have a strong individual and collective influence on such overall system time responses as time to market, material and information flow rates, inventory replacement rate, model change rate, and employee turnover rate. Few, if any, of these factors can be chosen or designed independently of the rest, especially in complex feedback systems.

Fortunately, there are powerful tools for feedback system design. They include linear transform theory, transient analysis, discrete event mathematics, fuzzy thinking, and some selected nonlinear and time-variant design methods. The properties of at least simple linear systems designed by these techniques can be simulated and adjusted easily. A certain intuition can be developed based upon long experience with them. For example:

- *Behavior with feedback can be very different from behavior without it.*
  Positive example: Provide inventory managers with timely sales information and drastically reduce inventory costs. Negative example: Ignore customer feedback and drown in unused inventory.
- *Feedback works. However, the design of the feedback path is critical. Indeed, in the case of strong feedback, its design can be more important than that of the forward path.*
  Positive example: Customer feedback needs to be supplemented by anticipatory projections of economic trends and of competitors' responses to one's own strategies and actions to avoid delays and surprises. Negative examples: If feedback signals are "out of step" or of the wrong polarity, the system will oscillate, if not go completely out of control. Response that is too little, too late is often worse than no response at all.
- *Strong feedback can compensate for many known vagaries, even nonlinearities in the forward path, but it does so "at the margin."*
  Example: Production lines can be very precisely controlled around their operating points; that is, once a desired operating point is reached, tight control will maintain it, but off that point or on the way to or from it (e.g., start up, synchronization,[10] and shut down) off-optimum behavior is likely. Example: Just in Time response works well for steady flow and constant volume. It can oscillate if flow is intermittent and volume is small.
- *Feedback systems will inherently resist unplanned or unanticipated change, whether internal or external.* Satisfactory responses to anticipated changes, however, can usually be assured. *In any case, the response will last at least one time constant (cycle time) of the system.* These properties provide stability against disruption. On the other hand, abrupt mandates, however necessary, will be resisted and the end results may be considerably different in magnitude and timing from what advocates of the change anticipated.
  Example: Social systems, incentive programs, and political systems notoriously "readjust" to their own advantage when change is mandated. The resultant system behavior is usually less than, later than, and modified from that anticipated.
- *To make a major change in performance of a presently stable system is likely to require a number of changes in the overall system design.*
  Example: The change from mass production to lean production to flexible production[11] and the use of robots and high technology.

Not all systems are linear, however. As a warning against overdependence on linear-derived intuition, typical characteristics of nonlinear systems are:

- *In general, no two systems of different nonlinearity behave in exactly the same way.*
- *Introducing changes into a nonlinear system will produce different (and probably unexpected) results if they are introduced separately than if they are introduced together.*
- *Introducing even very small changes in input magnitude can produce very different consequences even though all components and processes are deterministic.*
  Example: Chaotic behavior (noiselike but with underlying structure) close to system limits is such a phenomenon. Example: When the phone system is saturated with calls and goes chaotic, the planned strategy is to cut off all calls to a particular sector (e.g., California after an earthquake) or revert back to the simplest mode possible (first come, first serve). Sophisticated routing is simply abandoned as it is part of the problem. Example: When a computer abruptly becomes erratic as it runs out of memory, the simplest and usually successful technique is to turn it off and start it up again (reboot), hoping that not too much material has been lost.
- *Noise and extraneous signals irreversibly intermix with and alter normal, intended ones, generally with deleterious results.*
  Example: Modification of feedback and control signals is equivalent to modifying system behavior; that is, changing its transient and steady-state behavior. Nonlinear systems are therefore particularly vulnerable to purposeful opposition (jamming, disinformation, overloading).
- *Creating nonlinear systems is of higher risk than creating well-understood, linear ones.* The risk is less that the nonlinear systems will fail under carefully framed conditions than that they will behave strangely otherwise.
  Example: In the ultraquality spacecraft business there is a heuristic: **If you can't analyze it, don't build it** — an admonition against unnecessarily nonlinear feedback systems.

The two most common approaches to nonlinearity are first, when nonlinearities are both unavoidable and undesirable, minimize their effect on end system behavior through feedback and tight control of operating parameters over a limited operating region. Second, when nonlinearity can improve performance as in discrete and fuzzy control systems, be sure to model and simulate performance *outside* the normal operating range to avoid "non-intuitive" behavior.

The architectural and analytic difficulties faced by modern manufacturing feedback systems are that they are neither simple nor necessarily linear.

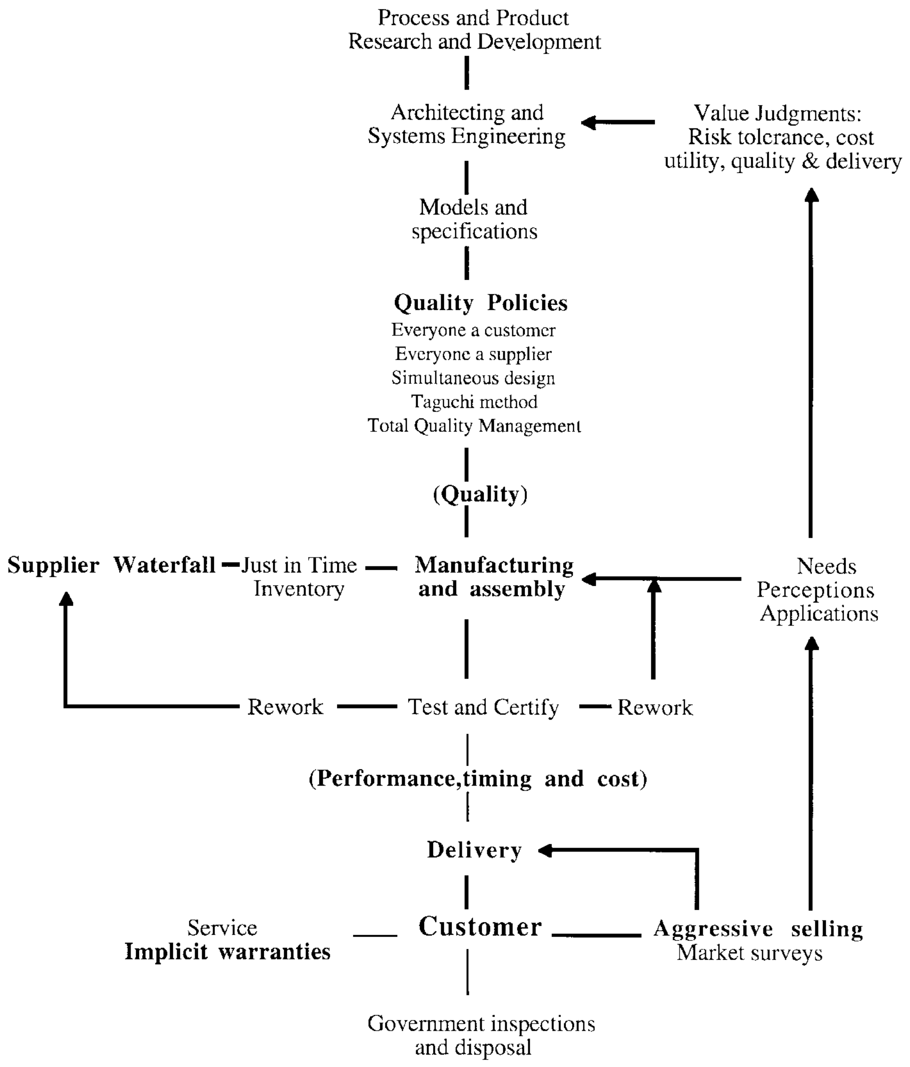
They are unavoidably complex, probably contain some nonlinearities (limiters and fixed time delays), are multiply-interconnected, and are subject to sometimes drastic external disturbances, not the least of which are sudden design changes and shifts in public perception. Their architectures must therefore be robust and yet flexible. Though inherently complex, they must be simple enough to understand and modify at the system level without too many unexpected consequences. In short, they are likely to be prime candidates for the heuristic and modeling techniques of systems architecting.

Lean production

One of the most influential books on manufacturing in recent years is the 1990 best seller, *The Machine that Changed the World, The Story of Lean Production*.[12] Although the focus of this extensive survey and study was on automobile production in Japan, the U.S., and Europe, its conclusions are applicable to manufacturing systems in general; particularly the concepts of quality and feedback. A 1994 follow-up book, *Comeback, The Fall & Rise of the American Automobile Industry*,[13] tells the story of the American response and the lessons learned from it, though calling it a "comeback" may have been premature. The story of lean production systems is by no means neat and orderly. Although the principles can be traced back to 1960, effective implementation took decades. Lean production certainly did not emerge full-blown. Ideas and developments came from many sources, some prematurely. Credits were sometimes misattributed. Many contributors were very old by the time their ideas were widely understood and applied. Quality was sometimes seen as an end result instead of as a prerequisite for any change. The remarkable fact that virtually *every* critical parameter improved by at least 20%, if not 50%,[14] does not seem to have been anticipated. Then, within a few years, everything worked. But when others attempted to adopt the process, they often failed. Why?

One way to answer such questions is to diagram the lean production process from an architectural perspective. Figure 4.4 is an architect's sketch of the lean production waterfall derived from the texts of the just-mentioned books, highlighting (bold facing) its nonclassical features and strengthening its classical ones.* The most apparent feature is the number and strength of its feedback paths. Two are especially characteristic of lean production: the supplier waterfall loop and the customer-sales-delivery loop. Next evident is the Quality Policies box, crucial not only to high quality but to the profitable and proper operation of later steps, Just in Time inventory, rework, and implicit warranties. Quality policies are active elements in the sequence of steps, are a step through which all subsequent orders and specifications

* Strictly speaking, though the authors of the lean production books did not mention it, an architect's perspective should also include the intersecting product waterfalls and software spirals. Interestingly, because it seems to be true for all successful systems, it is possible to find where and by whom systems architecting was performed. Two of the more famous automotive production architects were Taiichi Ohno, the pioneer of the Toyota Motor Company Production System, and Yoshiki Yamasaki, head of automobile production at Mazda.



Process and Product Research and Development
Architecting and Systems Engineering
Value Judgments: Risk tolerance, cost utility, quality & delivery
Models and specifications
Quality Policies
Everyone a customer
Everyone a supplier
Simultaneous design
Taguchi method
Total Quality Management
(Quality)
Supplier Waterfall
Just in Time Inventory
Manufacturing and assembly
Needs Perceptions Applications
Rework
Test and Certify
Rework
(Performance,timing and cost)
Delivery
Service
Implicit warranties
Customer
Aggressive selling
Market surveys
Government inspections and disposal

Figure 4.4

must pass, and are as affected by its feedback input as any other step. That is, policies must change with time, circumstance, technology, product change, and process imperatives.

Research and development (R&D) is not "in the loop," but instead is treated as one supplier of possibilities, among many, including one's competitors' R&D. As described in the 1990 study, R&D is not a driver, though it wouldn't be surprising if its future role were different. Strong customer feedback, in contrast, is very much within the loop, making the loop responsive to customer needs at several key points. Manufacturing feedback to suppliers is also a short path, in contrast with the standoff posture of much U.S. procurement.

The success of lean production has induced mass producers to copy many of its features, not always successfully. Several reasons for lack of success are apparent from the figure. If the policy box does not implement ultraquality, little can be expected from changes further downstream, regardless of how much they ought to be able to contribute. Just in Time inventory is an example. Low-quality supply mandates a cushion of inventory roughly proportional to the defect rate; shifting that inventory from the manufacturer back to the supplier without a simultaneous quality upgrade simply increases transportation and financing costs. To decrease inventory, decrease the defect rate, then apply the coordination principles of Just in Time, not before.

Another reason for limited success in converting piecemeal to lean production is that any well-operated feedback system, including those used in classical mass production, will resist changes in the forward (waterfall) path. The “loop” will attempt to self-correct, and it will take at least one loop time constant before all the effects can be seen or even known. To illustrate, if supply inventory is reduced, what is the effect on sales and service inventory? If customer feedback to the manufacturing line is aggressively sought, as it is in Japan, what is the effect on time-to-market for new product designs?

A serious question raised in both books is how to convert mass production systems into lean production systems. It is not, as the name “lean" might imply, a mass production system with the “fat" of inventory, middle management, screening, and documentation, taken out. It is to recognize lean production as a different architecture based on different priorities and interrelationships. How, then, to begin the conversion? What is both necessary and sufficient? What can be retained?

The one and only place to begin conversion, given the nature of feedback systems, is in the Quality Policies step. In lean production, quality is not a production result determined post-production and post-test, it is a prerequisite policy imperative. Indeed, Japanese innovators experienced years of frustration when TQM, JIT, and the Taguchi methods at first seemed to do very little. The level of quality essential for these methods to work had not yet been reached. When it was, the whole system virtually snapped into place with results that became famous. Even more important for other companies, unless their quality levels are high enough, even though all the foregoing methods are in place, the famous results will not — and cannot — happen.

Conversely, at an only slightly lower level of quality, lean systems sporadically face at least temporary collapse.[15] As a speculation, there appears to be a direct correlation between how close to the cliff of collapse the system operates and the competitive advantage it enjoys. Backing off from the cliff would seem to decrease its competitive edge, yet getting too close risks imminent collapse: line shutdown, transportation jam-up, short-fuse customer anger, and collateral damage to suppliers and customers for whom the product is an element of a still larger system production.

To summarize, lean production is an ultraquality, dynamic feedback system inherently susceptible to any reduction in quality. It depends upon well-designed, multiple feedback. Given ultraquality standards, lean production arguably is less complex, simpler, and more efficient than mass production; and, by its very nature, it is extraordinarily, fiercely, competitive.

Flexible manufacturing

Flexible manufacturing is the capability of sequentially making more than one product on the same production line. In its most common present-day form, it customizes products for individual customers, more or less on demand, by assembling different modules (options) on a common base (platform). Individually tailored automobiles, for example, have been coming down production lines for years. But with one out of three or even one out of ten units having to be sent back or taken out of a production stream, flexible manufacturing in the past has been costly in inventory, complex in operation, and high-priced per option compared to all-of-a-kind production.

What has changed is customer demands and expectations, especially in consumer products. Largely because of technological innovation, more capability for less cost now controls purchase rate rather than wearout and increasing defect rate. An interesting epilog to the DeSoto story.* One consequence of the change is more models per year with fewer units per model, the higher costs per unit being offset by use of techniques such as preplanned product improvement, standardization of interfaces and protocols, and lean production methods.

A natural architecture for the flexible manufacturing of complex products would be an extension of lean production with its imperatives — additional feedback paths and ultraquality-produced simplicity — and an imperative all its own, human-like information command and control.

At its core, flexible manufacturing involves the real-time interaction of a production waterfall with multiple product waterfalls. Lacking an architectural change from lean production, however, the resultant multiple information flows could overwhelm conventional control systems. The problem is less that of gathering data than of condensing it. That suggests that flexible manufacturing will need judgmental, multiple-path control analogous to that of an air traffic controller in the new "free flight" regime. Whether the resultant architecture will be fuzzy, associative, neural, heuristic, or largely human is arguable.

To simplify the flexible manufacturing problem to something more manageable, most companies today would limit the flexibility to a product *line* that is forward- and backward-compatible, uses similar if not identical mod-

* Parenthetically, the Japanese countered the automobile obsolescence problem by quadrennial government inspections so rigorous that it was often less expensive to turn a car in and purchase a new one than to bring the old one up to government standards. Womack, et al., 1990, 62.

ules, keeps to the same manufacturing standards, and is planned to be in business long enough to write off the cost of the facility out of product line profits. In brief, production would be limited to products having a single basic architecture; for example, producing either Macintosh computers, Hitachi TV sets, or Motorola cellular telephones, but not all three on demand on the same production line.

Even that configuration is complex architecturally. To illustrate: a central issue in product line design is where in the family of products, from low- to high-end, to optimize. Too high in the series and the low end is needlessly costly; too low and the high end adds too little value. A related issue arises in the companion manufacturing system. Too much capability and its overhead is too high; too little, and it sacrifices profit to specialty suppliers.

Another extension from lean production to flexible manufacturing is much closer coordination between the design and development of the product line and the design and development of its manufacturing system. Failure to achieve this coordination, as illustrated by the problems of introducing robots into manufacturing, can be warehouses of unopened crates of robots and in-work products that can't be completed as designed. Clearly, **the product and its manufacturing system must match.** More specifically, their time constants, transient responses, and product-to-machine interfaces must match, recognizing that any manufacturing constraint means a product constraint, and vice versa.

As suggested earlier, the key technological innovation is likely to be the architecture of the joint information system.[16] In that connection, one of the greatest determinants of the information system's size, speed, and resolution is the quality of the end product and the yield of its manufacturing process; that is, their defect rates. The higher these defect rates, the greater the size, cost, complexity, and precision of the information system that will be needed to find and eliminate them quickly.

Another strong determinant of information system capacity is piece-part count, another factor dependent on the match of product and manufacturing techniques. Mechanical engineers have known for years that whenever possible, replace a complicated assembly of parts with a single, specialized piece. Nowhere is the advantage of piece-part reduction as evident as in the continuing substitution of more and more high-capacity microprocessors for their lower-capacity predecessors. Remarkably, this substitution, for approximately the same cost, also decreases the defect rate per computational operation.

Also, especially for product lines, the fewer different parts from model to model, the better, as long as that commonality doesn't decrease system capability unduly. Once again, there is a close correlation between reduced defect rate, reduced information processing, reduced inventory, and reduced complexity — all by design.

Looking further into the future, an extension of a lean production architecture is not the only possibility for flexible manufacturing. It is possible that flexible manufacturing could take quite a different architectural turn based on a different set of priorities. Is ultraquality production really necessary for simple, low cost, limited warranty products made largely from commercial off-the-shelf (COTS) units; e.g., microprocessors and flat screens? Or is the manufacturing equivalent of parallel processors (pipelines) the answer? Should some flexible manufacturing hark back to the principles of special, handcrafted products; one-of-a-kind planetary spacecraft? The answers should be known in less than a decade, considering the profit to be made in finding them.

Heuristics for architecting manufacturing systems

- The product and its manufacturing system must match (in many ways).
- Keep it simple (ultraquality helps).
- Partition for near autonomy (a tradeoff with feedback).
- In partitioning a manufacturing system, choose the elements so that they minimize the complexity of material and information flow (Savagian, Peter J., 1990, USC).
- Watch out for critical misfits (between intersecting waterfalls).
- In making a change in the manufacturing process, how you make it is often more important than the change itself (management policy).
- When implementing a change, keep some elements constant to provide an anchor point for people to cling to (Schmidt, Jeffrey H., 1993, USC; a tradeoff when a new architecture is needed).
- Install a machine that even an idiot can use, and pretty soon everyone working for you is an idiot (Olivieri, J. M., 1991, USC; an unexpected consequence of mass production Taylorism). See next heuristic.
- Everyone a customer, everyone a supplier.
- To reduce unwanted nonlinear behavior, linearize.
- If you can't analyze it, don't build it.
- Avoid creating the same resonance time constant in more than one location in a (production) system.
- The five whys (a technique for finding basic causes, and one used by every inquisitive child to learn about the world at large).

In conclusion

Modern manufacturing can be portrayed as an ultraquality, dynamic feedback system intersecting with that of the product waterfall. The manufacturing systems architect's added tasks, beyond those of all systems architects, include: (1) maintaining connections to the product waterfall and the software spiral necessary for coordinated developments; (2) assuring quality levels high enough to avoid manufacturing system collapse or oscillation; (3) determining and helping control the system parameters for stable and timely performance and, last but not least, (4) looking farther into the future than do most product-line architects.

Exercises

1. Manufacturing systems are themselves complex systems which need to be architected. If the manufacturing line is software-intensive, and repeated software upgrades are planned, how can certification of software changes be managed?
2. The feedback lags or resonances of a manufacturing system of a commercial product interact with the dynamics of market demand. Give examples of problems arising from this interaction and possible methods for alleviating them.
3. Examine the following hypothesis: increasing quality levels in manufacturing enable architectural changes in the manufacturing system which greatly increase productivity, but may make the system increasingly sensitive to external disruption. For a research exercise, use case studies or a simplified quantitative model.
4. Does manufacturing systems architecture differ in mass production systems (thousands of units) and very low-quantity production systems (fewer than ten produced systems)? If so, how and why?
5. Current flexible manufacturing systems usually build very small lot sizes from a single product line in response to real-time customer demand; for example, an automobile production line that builds each car to order. Consider two alternative architectures for organizing such a system, one centralized and one decentralized. The first would use close centralized control, centralized production scheduling, and resource planning. The other would use fully distributed control based on disseminating customer/supplier relationships to each work cell. That is, each job and each work cell interact individually through an auction for services. What would be advantages and disadvantages of both approaches? How would the architecture of the supporting information systems (extending to sales and customer support) have to differ in the two cases?

Notes and references

1. Hayes, R. H., Wheelwright, S. C., and Clark, K. B., *Dynamic Manufacturing, Creating the Learning Organization,* Free Press, New York, 1988, chap. 7, "The Architecture of Manufacturing: Material and Information Flows," p. 185, defines a manufacturing architecture as including its hardware, material, and information flows, their coordination, and managerial philosophy. This textbook is highly recommended, especially Chapters 7–9.

2. Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991, chap. 8, 160-187. Ultraquality: Quality (absence of defects) so high as to be impractical to prove by measure with confidence.
3. Hayes, R. H., Wheelwright, S. C., and Clark, K. B., *Dynamic Manufacturing, Creating the Learning Organization,* Free Press, New York, 1988.
4. Womack, J. P., Jones, D. T., and Roos, D., *The Machine that Changed the World, The Story of Lean Production,* Harper Collins Publishers, New York, 1990, paperback, p. 4.
5. See also Hayes et al., 1988, Chap. 8, "Controlling and Improving the Manufacturing Process."
6. Hayes et al., 1988, Chap. 7, p. 185.
7. See Brooks, F. P., Jr., No silver bullet, essence and accidents of software *e*ngineering, *Computer,* April 1987, pp. 10-19, especially p. 18; and Simon, H., *The Sciences of the Artificial,* 2nd ed., MIT Press, Cambridge, MA, 1987, pp. 200-209.
8. For the equivalent in a surgical team, see Brooks' *Mythical Man-Month*.
9. See Womack et al., *The Machine that Changed the World, The Story of Lean Production*.
10. See Lindsey, W. C., *Synchronization Systems in Communication and Control,* Prentice-Hall, Englewood Cliffs, NJ, 1972.
11. Hayes et al., 1988, p. 204.
12. Womack et al., 1990.
13. For an update, see Ingrassia, P. and White, J. B., *Comeback, The Fall & Rise of the American Automobile Industry,* Simon & Schuster, New York, 1994.
14. Womack et al., 1994, Title page.
15. Womack et al., 1990, p. 62.
16. Hayes et al., 1988, Chap. 7, p. 185.

chapter five

Social systems

Introduction: defining sociotechnical systems

Social: concerning groups of people or the general public

Technical: based on physical sciences and their application

Sociotechnical systems: technical works involving significant social participation, interests, and concerns.

Sociotechnical systems are technical works involving the participation of groups of people in ways that significantly affect the architectures and design of those works. Historically, the most conspicuous have been the large civil works — monuments, cathedrals, and urban developments, dams and roadways among them. Lessons learned from their construction provide the basis for much of civil (and systems) architecting and its theory.[1]

More recently, others, physically quite different, have become much more sociotechnical in nature than might have been contemplated at their inception — ballistic missile defense, air travel, *e-mail*, information networks, welfare and health delivery, for example. Few can even be conceived, much less built, without major social participation, planned or not. Experiences with them have generated a number of strategies and heuristics of importance to architects in general. Several are presented here. Among them are three heuristics: the four whos, economic value, and the tension between perceptions and facts. In the interests of informed use, as with all heuristics, it is important to understand the context within which they evolved, the sociotechnical domain. At the end of this chapter are a number of general heuristics of applicability to sociotechnical systems in particular.

Public participation

At the highest level of social participation, members of the public directly use — and may own a part of — the system's facilities. At an intermediate level they are provided a personal service, usually by a public or private utility. Most important, individuals and not the utilities — the architect's clients — are the end users. Examples are highways, communication and information circuits, general aviation traffic control, and public power. Public cooperation and personal responsibility are required for effective operation. That is, users are expected to follow established rules with a minimum of policing or control. Drivers and pilots follow the rules of the road. Communicators respect the twin rights of free access and privacy.

At the highest level of participation, participating individuals choose and collectively own a major fraction of the system structure such as cars, trucks, aircraft, computers, telephones, electrical and plumbing hardware, etc. In a sense, the public "rents" the rest of the facilities through access charges, fees for use, and taxes. Reaction to a facility failure or a price change tends to be local in scope, quick, and focused. The public's voice is heard through specialized groups such as automobile clubs for highways, retired persons associations for health delivery, professional societies for power, and communications services, etc.

Market forces are used to considerable advantage through adverse publicity in the media, boycotts, and resistance to stock and bond issues on the one hand, and through enthusiastic acceptance on the other. Recent examples of major architectural changes strongly supported by the public are superhighways, satellite communications, entertainment cable networks, jet aircraft transportation, health maintenance organizations, and a slow shift from polluting fossil fuels to alternative sources of energy.

Systems of this sort are best described as collaborative systems, systems which operate only through the partially voluntary initiative of their components. This collaboration is an important subject in its own right since the Internet, the World Wide Web, and open source software are collaborative assemblages. We address this topic in detail in Chapter 7.

At the other extreme of social participation are social systems used solely by the government, directly or through sponsorship, for broad social purposes delegated to it by the public; for example, NASA and DoD systems for exploration and defense, Social Security and Medicare management systems for public health and welfare, research capabilities for national well-being, and police systems for public safety. The public pays for these services and systems only indirectly, through general taxation. The architect's client and end user is the government itself. The public's connection with the design, construction, and operation of these systems is sharply limited. Its value judgments are made almost solely through the *political* process, the subject of Chapter 10. They might best be characterized as "politicotechnical."

The foundations of sociotechnical systems architecting

The foundations of sociotechnical systems architecting are much the same as for all systems: a systems approach, purpose orientation, modeling, certification, and insight. Social system quality, however, is less a foundation than a case-by-case tradeoff; that is, the quality desired depends on the system to be provided. In nuclear power generation, modern manufacturing, and manned space flight, ultraquality is an imperative. But in public health, pollution control, and safety, the level of acceptable quality is only one of many economic, social, political, and technical factors to be accommodated.

But if sociotechnical systems architecting loses one foundation, ultraquality, it gains another — a direct and immediate response to the public's needs and perceptions. Responding to public perceptions is particularly difficult, even for an experienced architect. The public's interests are unavoidably diverse and often incompatible. The groups with the strongest interests change with time, sometimes reversing themselves based on a single incident. Three Mile Island was such an incident for nuclear power utilities. Pictures of the earth from a lunar-bound Apollo spacecraft suddenly focused public attention and support for global environmental management. An election of a senator from Pennsylvania avalanched into widespread public concern over health insurance and Medicare systems.

The separation of client and user

In most sociotechnical systems, the client, the buyer of the architect's services, is not the user. This fact can present a serious ethical, as well as technical, problem to the architect: how to treat conflicts between the preferences, if not the imperatives, of the utility agency and those of the public (as perceived by the architect) when preferences strongly affect system design.

It is not a new dilemma. State governments have partly resolved the problem by licensing architects and setting standards for systems that affect the health and safety of the public. Buildings, bridges, and public power systems come to mind. Information systems are already on the horizon. The issuing or denial of licenses is one way of making sure that public interest comes first in the architect's mind. The setting of standards provides the architect some arguments against overreaching demands by the client. But these policies do not treat such conflicts as the degree of traffic control desired by a manager of an Intelligent Transportation System (ITS) as compared with that of the individual user/driver, nor the degree of governmental regulation of the Internet to assure a balance of access, privacy, security, profitmaking, and system efficiency. One of the ways of alleviating some of these tensions is through economics.

Socioeconomic insights

Social economists bring two special insights to sociotechnical systems. The first insight, which might be called **the four whos**, asks four questions that need to be answered *as a self-consistent set* if the system is to succeed economically; namely, **who benefits? who pays? who provides? and, as appropriate, who loses?**

Example: The answers to these questions of the Bell Telephone System were: (1) the beneficiaries were the callers and those who received the calls; (2) the callers paid based on usage because they initiated the calls and could be billed for them; (3) the provider was a monopoly whose services and charges were regulated by public agencies for public purposes; and (4) the losers were those who wished to use the telephone facilities for services not offered or to sell equipment not authorized for connection to those facilities. The telephone monopoly was incentivized to carry out widely available basic research because it generated more and better service at less cost, a result the regulatory agencies desired. International and national security agreements were facilitated by having a single point of contact, the Bell System, for all such matters. Standards were maintained and the financial strategy was long-term, typically 30 years. The system was dismantled when the losers evoked antitrust laws, creating a new set of losers, complex billing, standards problems, and a loss of research. Arguably, it enabled the Internet sooner than otherwise. Subsequently, separate satellite and cable services were established, further dismantling what had been a single service system.

Example: The answers to the four whos for the privatized Landsat System, a satellite-based, optical-infrared surveillance service, were: (1) the beneficiaries were those individuals and organizations who could intermittently use high-altitude photographs of the earth; (2) because the *value* to the user of an individual photograph was unrelated to its cost (just as is the case with weather satellite TV), the individual users could not be billed effectively; (3) the provider was a private, profit-making organization which understandably demanded a cost-plus-fixed-fee contract from the government as a surrogate customer; and (4) when the government balked at this result of privatization, the Landsat system faced collapse. Research had been crippled, a French government-sponsored service (SPOT) had acquired appreciable market share, and legitimate

> customers faced loss of service. Privatization was reversed and the government again became the provider.
>
> Example: Serious debates over the nature of their public health systems are underway in many countries, triggered in large part by the technological advances of the last few decades. These advances have made it possible for humanity to live longer and in better health, but the investments in those gains are considerable. The answers to the four whos are at the crux of the debate. Who benefits — everyone equally at all levels of health? Who pays — regardless of personal health or based on need and ability to pay? Who provides — and determines cost to the user? Who loses — anyone out of work or above some risk level, and who determines who loses?

Regardless of how the reader feels about any of these systems, there is no argument that the answers are matters of great social interest and concern. At some point, if there are to be public services at all, the questions must be answered and decisions made. But who makes them and on what basis? Who and where is the architect in each case? How and where is the architecture created? How is the public interest expressed and furthered?

The second economics insight is comparably powerful: **in any resource-limited situation, the true value of a given service or product is determined by what a buyer is willing to give up to obtain it.** Notice that the subject here is *value*, not price or cost.

> Example: The public telephone network provides a good example of the difference between cost and value. The cost of a telephone call can be accurately calculated as a function of time, distance, routing (satellite, cable, cellular, landline, etc.), location (urban or rural), bandwidth, facility depreciation, etc. But the value depends on content, urgency, priority, personal circumstance, and message type, among other things. As an exercise, try varying these parameters and then estimating what a caller might be willing to pay (give up in basic needs or luxuries). What is then a fair allocation of costs among all users? Should a sick, remote, poor caller have to pay the full cost of remote TV health service, for example? Should a business which can pass costs on to its customers receive volume discounts for long-distance calling via satellite? Should home TV be pay-per-view for everyone? Who should decide on the answers?

These two socioeconomic heuristics, used together, can alleviate the inherent tensions among the stakeholders by providing the basis for compromise and consensus among them. The complainants are likely to be those whose payments are perceived as disproportionate to the benefits they receive. The advocates, to secure approval of the system as a whole, must give up or pay for something of sufficient value to the complainants that they agree to compromise. Both need to be made to walk in the others' shoes for a while. Therein can be the basis of an economically viable solution.

The interaction between the public and private sectors

A third factor in sociotechnical systems architecting is the strong interplay between the public and private sectors, particularly in the advanced democracies where the two sectors are comparable in size, capability, and influence, but differ markedly in how the general public expresses its preferences.

By the middle of the 1990s, the historic boundaries between public and private sectors in communications, health delivery, welfare services, electric power distribution, and environmental control were in a state of flux. This chapter is not the place to debate the pros and cons. Suffice to say the imperatives, interests, and answers to the economists' questions are sharply different in the two sectors.[2] The architect is well advised to understand the imperatives of both sectors prior to suggesting architectures that must accommodate them. For example, the private sector must make a profit to survive; the public sector does not and treats profits as necessary evils. The public sector must follow the rules; the private sector sees rules and regulations as constraints and deterrents to efficiency. Generally speaking, the private sector does best in providing well-specified things at specified times. The public sector does best at providing services within the resources provided.

Because of these differences, one of the better tools for relieving the natural tension between the sectors is to change the boundaries between them in such negotiable areas as taxation, regulation, services provided, subsidies, billing, and employment. Because perceived value in each of these areas is different in the two sectors and under different circumstances, possibilities can exist where each sector perceives a net gain. The architect's role is to help discover the possibilities, achieve balance through compromise on preferences, and assure a good fit across boundaries.

Architecting a system, such as a public health system, which involves both the public and private sectors can be extraordinarily difficult, particularly if agreement does not exist on a mutually-trusted architect, on the answers to the economist's questions, or on the social value of the system relative to that of other socially desirable projects.

Facts vs. perceptions: an added tension

Of all the distinguishing characteristics of social systems, the one which most sharply contrasts them with other systems is the tension between facts and perceptions about system behavior. To illustrate the impact on design, architects are well familiar with the tradeoffs between performance, schedule, cost, and risk. These competing factors might be thought of as pulling architecting four different directions, as sketched in Figure 5.1. Figure 5.2 can be thought of as the next echelon or ring — the different sources or components of performance, schedule, cost, and risk. Notice that performance has an aesthetic component as well as technical and sociopolitical sources. Automobiles are a clear example. Automobile styling often is more important than aerodynamics or environmental concerns in their architectural design. Costs also have several components, of which the increased costs to people of cost reduction in money and time are especially apparent during times of technological transition.

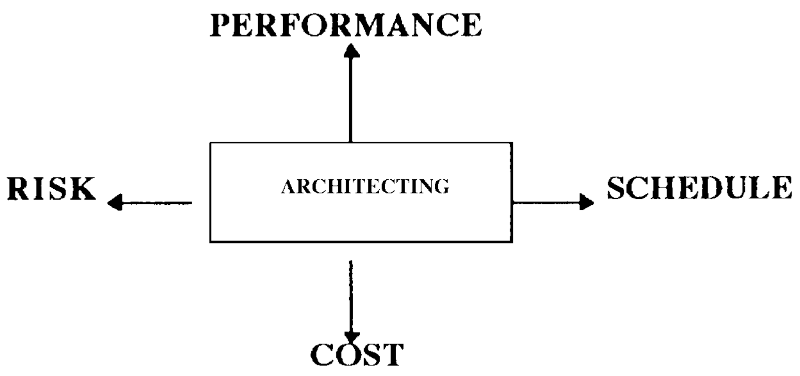


Figure 5.1

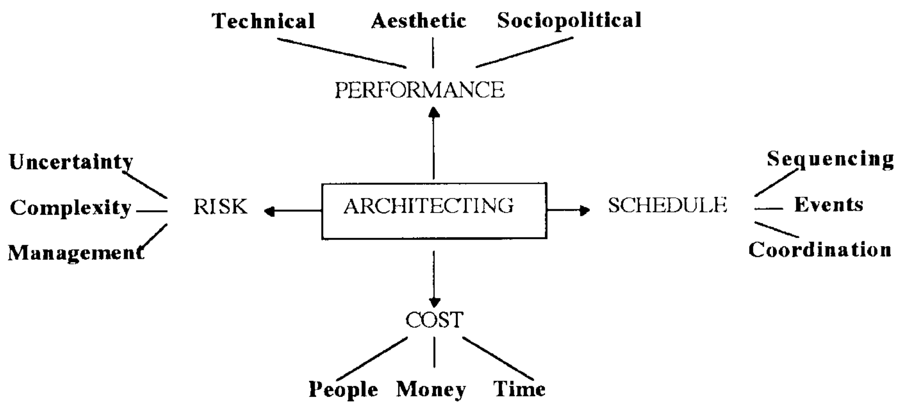


Figure 5.2

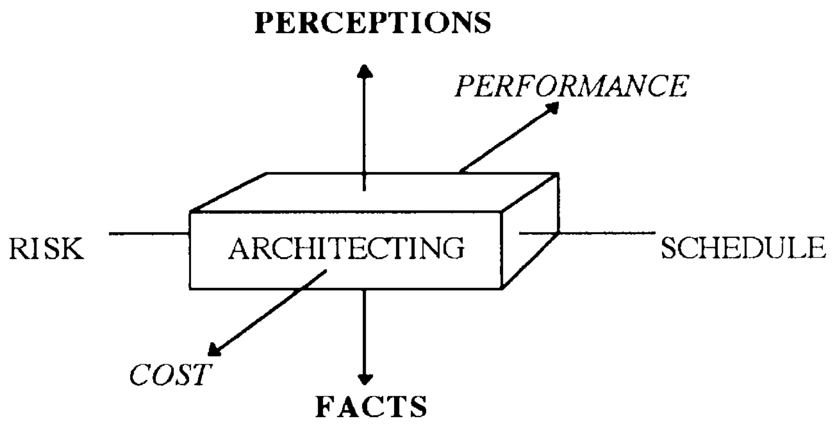


Figure 5.3

To these well-known tensions must be added another, one which social systems exemplify but which exist to some degree in all complex systems; namely, the tension between perceptions and facts, shown in Figure 5.3. This added tension may be dismaying to technically trained architects, but it is all too real to those who deal with public issues. Social systems have generated a painful design heuristic: **it's not the facts, it's the perceptions that count.** Some real-world examples:

- It makes little difference what facts nuclear engineers present about the safety of nuclear power plants, their neighbors' perception is that someday their local plant will blow up. Remember Three Mile Island and Chernobyl? A.M. Weinberg of Oak Ridge Associated Universities suggested perhaps the only antidote: "The engineering task is to design reactors whose safety is so transparent that the skeptical elite is convinced, and through them the general public."[3]
- Airline travel has been made so safe that the most dangerous part of travel can be driving to and from the airport. Yet, every airliner crash is headline news. A serious design concern, therefore, is how many passengers an airliner should carry — 200? 400? 800? — because even though the average accident rate per departure would probably remain the same,[4] more passengers would die at once in the larger planes, and a public perception might develop that larger airliners are less safe, facts notwithstanding.
- One of the reasons that health insurance is so expensive is that health care is perceived by employees as nearly "free" because almost all its costs are paid for either by the employee's company or the government. The facts are that the costs are either passed on to the consumer, subtracted from wages and salaries, taken as a business deduction against taxes, paid for by the general taxpayer, or all of the above. There is no free lunch.

- One of the most profound and unanticipated results of the Apollo flights to the moon was a picture of the Earth from afar, a beautiful blue, white, brown, and green globe in the blackness of space. We certainly had understood that the Earth was round, but that distant perspective changed our perception of the vulnerability of our home forever, and, with it, our actions to preserve and sustain it. Just how valuable was Apollo, then and in our future? Is there an equivalent value today?

Like it or not, the architect must understand that perceptions can be just as real as facts, just as important in defining the system architecture, and just as critical in determining success. As one heuristic states**: "the phrase, "I hate it," is direction.**[5] There have even been times when, in retrospect, perceptions were "truer" than facts which changed with observer, circumstance, technology, and better understanding. Some of the most ironic statements begin with, "It can't be done, because the facts are that..."

Alleviating the tension between facts and perceptions can be highly individualistic. Some individuals can be convinced — in *either* direction — by education, some by prototyping or anecdotes, some by A. M. Greenberg's antidote given earlier, some by better packaging or presentation, and some only by the realities of politics. Some individuals will never be convinced, but they might be accepting. In the end, it is a matter of achieving a balance of perceived values. The architect's task is to search out that area of common agreement that can result in a desirable, feasible system.

Heuristics for social systems

- Success is in the eyes of the beholder (not the architect).
- Don't assume that the original statement of the problem is necessarily the best, or even the right one. (Most customers would agree.)
- In conceptualizing a social system, be sure there are mutually consistent answers to the Four Whos: Who benefits? Who Pays? Who supplies (provides)? And, as appropriate, Who loses?
- In any resource-limited situation, the true value of a given service or product is determined by what one is willing to give up to obtain it.
- The choice between the architectures may well depend upon which set of drawbacks the stakeholders can handle best (not on which advantages are the most appealing).
- Particularly for social systems, it's not the facts, it's the perceptions that count (try making a survey of public opinion).
- The phrase, "I hate it." is direction (Or weren't you listening?)
- In social systems, *how* you do something may be more important than *what* you do. (A sometimes bitter lesson for technologists to learn.)

- When implementing a change, keep some elements constant as an anchor point for people to cling to (at least until there are some new anchors).
- It's easier to change the technical elements of a social system than the human ones (enough said).

In conclusion

Social systems, in general, place social concerns ahead of technical ones. They exemplify the tension between perception and fact. More than most systems, they require consistent responses to questions of who benefits?; who pays?; who supplies (provides, builds, etc.); and, sociologically at least, who loses?

Perhaps more than other complex systems, the design and development of social ones should be amenable to insights and heuristics. Social factors, after all, are notoriously difficult to measure, much less predict. But, like heuristics, they come from experience, from failures as well as successes, and from lessons learned.

Exercises

1. Public utilities are examples of sociotechnical systems. How are the heuristics discussed in this chapter reflected in the regulation, design, and operation of a local utility system?
2. Apply the **four whos** to a sociotechnical system familiar to you. Examples: Internet, air travel, communication satellites, a social service.
3. Many efforts are underway to build and deploy intelligent transport systems using modern information technologies to improve existing networks and services. Investigate some of the current proposals and apply the **four whos** to the proposal.
4. Pollution and pollution control are examples of a whole class of sociotechnical systems where disjunctions in the **four whos** are common. Discuss how governmental regulatory efforts, both through mandated standards and pollution license auctions, attempt to reconcile the **four whos**. To what extent have they been successful? How did you judge success?
5. Among the most fundamental problems in architecting a system with many stakeholders is conflicts in purposes and interests. What architectural options might be used to reconcile them?
6. Give an example of the application of the heuristic: **in introducing technological change, *how* you do it is often more important than *what* you do.**

Notes and references

1. Lang, J., *Creating Architectural Theory, The Role of the Behavioral Sciences in Environmental Design,* Van Nostrand Reinhold, New York, 1987.
2. See Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991, pp. 270-274; and "Why Not More Straight Commercial Buying," *Government Executive,* October 1976, pp. 46-8.
3. Weinberg, A. M., Engineering in an age of anxiety: the search for inherent safety, in *Engineering and Human Welfare NAE 25, Proc. 25th Annual Meeting,* National Academy of Engineering, Washington, D.C.
4. U.S. airline safety, scheduled commercial carriers, in *The World Almanac® and Book of Facts 1994,* Funk and Wagnalls Corporation © 1993. According to the National Transportation Safety Board source, the fatal accidents per 100,000 miles have remained at or less than 0.l00 from 1977 through 1992 despite a 40% increase in departures, major technological change, and the addition of increasingly larger aircraft to the airline fleets.
5. Gradous, L. I., USC, October 18, 1993.

chapter six

Software and information technology systems

Today I am more convinced than ever. Conceptual integrity is central to product quality. Having a system architect is the most important step toward conceptual integrity.
Frederick P. Brooks, Jr., *The Mythical Man-Month* After Twenty Years

Introduction: the status of software architecting

Software is rapidly becoming the centerpiece of complex system design, in the sense that an increasing fraction of system performance and complexity is captured in software. Software is increasingly the portion of the system that enables the unique behavioral characteristics of the system. Competitive developers of end-user system products find themselves increasingly developing software, even though the system itself combines both hardware and software. The reasons stem from software's ability to create intelligent behavior and to quickly accommodate technical-economic trends in hardware development.

Although detailed quantitative data is hard to come by, anecdotal stories tell a consistent story. A wide variety of companies in different industries (e.g., telecommunications, consumer electronics, industrial controls) have reported a dramatic shift in the relative engineering efforts devoted to hardware and software.* Where 10 to 15 years ago the ratio was typically 70% hardware and 30% software, it is now typically reversed, 30% hardware and 70% software, and the software fraction is continuing to grow. This should not be surprising given how the semiconductor industry has changed. Where product developers used to build from relatively simple parts (groups of logic gates), they now use highly integrated microprocessors with most peripheral devices on the chip. The economies of scale in semiconductor

* The numbers are anecdotal, but reflect private communications to the one of the present authors from a wide variety of sources.

design and production have pushed the industry toward integrated solutions where the product developer primarily differentiates through software. Moreover, microcontrollers have become so inexpensive and have such low power consumption that they can be placed in nearly any product, even throwaway products. The microprocessor-based products acquire their functionality by the software that executes on them. The product developer is transformed from a hardware designer to a hardware integrator and software developer. As software development libraries become larger, more capable, and accepted, many of the software developers will be converted to software integrators.

The largest market for software today is usually termed "information technology," which is a term encompassing the larger domain of computers and communications applied to business and public enterprises. We consider both here as software architecture, which as a field is becoming a distinct specialty. What is usually called software architecture, at least in the research community, is usually focused on developing original software rather than building information-processing systems through integration of large software and hardware components. Information technology practice is less and less concerned with developing complete original applications and more and more concerned with building systems through integration.

The focus of this chapter is less on the architecting of software (though that is discussed here and in Part Three) than it is on the impact of software on system architecting. Software possesses two key attributes that affect architecting. First, well-architected software can very rapidly evolve. Evolution of deployed software is much more rapid than that of deployed hardware, because an installed base of software can be regularly replaced — annual and even quarterly replacement is common. Annual or more frequent field software upgrades are normal for operating systems, databases, end-user business systems, large-scale engineering tools, and communication and manufacturing systems. This puts a demanding premium on software architectures because they must be explicitly designed to accommodate future changes and to allow repeated certification with those changes.

Second, software is an exceptionally flexible medium. Software can easily be built which embodies many logically complex concepts such as layered languages, rule-driven execution, data relationships, and many others. This flexibility of expression makes software an ideal medium with which to implement system "intelligence." In both the national security and commercial worlds, intelligent systems are far more valuable to the user and far more profitable for the supplier than their simpler predecessors.

In addition, a combination of technical and economic trends favor building systems from standardized computer hardware and system-unique software, especially when computing must be an important element of the system. Building digital hardware at very high integration levels yields enormous benefits in cost per gate, but it requires comparably large capital investments in design and fabrication systems. These costs are fixed, giving a strong competitive advantage to high production volumes. Achieving high production volumes requires that the parts be general purpose. For a system to reap the benefits of very high integration levels, its developers must either use the standard parts (available to all other developers as well) or be able to justify the very large fixed expense of a custom development. If standard hardware parts are selected, the remaining means to provide unique system functionality is software.

Logically, the same situation applies to writing software. Software production costs are completely dominated by design and test. Actual production is nearly irrelevant. Thus, there is likewise an incentive to make use of large programming libraries or components and amortize the development costs over many products. In fact, this is already taking place. While much of the software engineering community is frustrated with the pace of software reuse, there are many successful examples. One obvious one is operating systems. Very few groups who use operating systems write one from scratch anymore. Either they use an off-the-shelf product from one of the remaining vendors, or they use an open source distribution and customize it for their application. Databases, scripting languages, and Web applications are all examples of successful reuse of large software infrastructures.

A consequence of software's growing complexity and central role is a recognition of the importance of software architecture and its role in system design. An appreciation of sound architectures and skilled architects is taking hold. The soundness of the software architecture will strongly influence the quality of the delivered system and the ability of the developers to further evolve the system. When a system is expected to undergo extensive evolution after deployment, it is usually more important that the system be easily evolvable than that it be exactly correct at first deployment.

Architects, architectures, and architecting are terms increasingly seen in the technical literature, particularly in the software field.[1] Software architecture is frequently, though inconsistently, discussed. Many groups have formally defined the term software architecture. Although those definitions all differ, a distillation of commonly used ideas is that the architecture is the overall structure of a software system in terms of components and interfaces. This definition would include the major software components, their interfaces with each other and the outside world, and the logic of their execution (single-threaded, interrupted, multithreaded, combination). To this is often added principles defining the system's design and evolution, an interesting combination of heuristics with structure to define architecture. A software architectural "style" is seen as a generic framework of components or interfaces that defines a class of software structures. The view taken in this book, and in some of the literature,[2] is somewhat more expansive and also includes other high-level views of the system: behavior, constraints, and applications.

High-level advisory bodies to the Department of Defense are calling for architects of ballistic missile defense, CI^4 (command, control, communications, computers, and intelligence), global surveillance, defense communications, internetted weapons systems, and other "systems-of-systems." Formal standards are in process defining the role, milestones, and deliverables of system architecting.* Many of the ideas and terms of those standards come directly from the software domain, e.g., object-oriented, spiral process model, and rapid prototyping. The carryover should not be a surprise; the systems for which architecting is particularly important are behaviorally complex, data-intensive, and software-rich. Examples of software-centered systems of similar scope are appearing in the civilian world, such as the Information Superhighway, Internet, global cellular telephony, healthcare, manned space flight, and flexible manufacturing operations.

The consequences of this accelerating trend to smarter systems to software design are now becoming apparent. For the same reason that guidance and control specialists became the core of systems leadership in the past, software specialists will become the core in the future. In the systems world, software will change from having a support role (usually after the hardware design is fixed) to becoming the centerpiece of complex systems design and operation. As more and more of the behavioral complexity of systems is embodied in software, software will become the driver of system configuration. Hardware will be selected for its ability to support software instead of the reverse. This is now common in business information systems and other applications where compatibility with a software legacy is important.

If software is becoming the centerpiece of system development, it is particularly important to reconcile the demands of system and software development. Even if 90% of the system-specific engineering effort is put into software, the system is still the end product. It is the system, not the software inside, the client wishes to acquire. The two worlds share many common roots, but their differing demands have led them in distinctly different directions. Part of the role of systems architecting is to bring them together in an integrated way.

Software engineering is a rich source for integrated models; models that combine, link, and integrate multiple views of a system. Many of the formalisms now used in systems engineering had their roots in software engineering. This chapter discusses the differences between system architecting and software architecting, current directions in software architecting and architecture, and heuristics and guidelines for software. Chapter 10 provides further detail on four integrated software modeling methods, each aimed at the software component of a different type of system.

Software as a system component

How does the architecture and architecting of software interact with that of the system as a whole? Software has unique properties that influence overall system structure.

* The IEEE has recently upgraded a planning group to a working group to develop architecting standards. U. S. Military Standards work is also still in progress; see Chapter 11.

1. Software provides a palette of abstractions for creating system behavior. Software is extensible through layered programming to provide abstracted user interfaces and development environments. Through the layering of software it is possible to directly implement concepts such as relational data, natural language interaction, and logic programming that are far removed from their computational implementation. Software does not have a natural hierarchical structure, at least not one that mirrors the system-subsystem-component hierarchy of hardware.
2. It is economically and technically feasible to use evolutionary delivery for software. If architected to allow it, the software component of a deployed system can be completely replaced on a regular schedule.
3. Software cannot operate independently. Software must always be resident on some hardware system and, hence, must be integrated with some hardware system. The interactions between, and integration with, this underlying hardware system become key elements in software-centered system design.

For the moment there are no emerging technologies that are likely to take software's place in implementing behaviorally complex systems. Perhaps some form of biological or nano-agent technology will eventually acquire similar capabilities. In these technologies, behavior is expressed through the emergent properties of chaotically interacting organisms. But the design of such a system can be viewed as a form of logic programming in which the "program" is the set of component construction and interface rules. Then the system, the behavior that emerges from component interaction, is the expression of an implicit program, a highly abstracted form of software.

System architecting adapts to software issues through its models and processes. To take advantage of the rich functionality there must be models that capture the layered and abstracted nature of complex software. If evolutionary delivery is to be successful, and even just to facilitate successful hardware/software integration, the architecture must reconcile continuously changing software with much less frequently changing hardware.

Software for modern systems

Software plays disparate roles in modern systems. Mass-market application software, one-of-a-kind business systems, real-time analysis and control software, and human interactive assistants are all software-centered systems, but each is distinct from the other. The software attributes of rich functionality and amenability to evolution match the characteristics of modern systems. These characteristics include:

1. Storage of, and semiautonomous and intelligent interpretation of, large volumes of information.

2. Provision of responsive human interfaces that mask the underlying machines and present their operation in metaphor.
3. Semiautonomous adaptation to the behavior of the environment and individual users.
4. Real-time control of hardware at rates beyond human capability with complex functionality.
5. Constructed from mass-produced computing components and system-unique software, with the capability to be customized to individual customers.
6. Coevolution of systems with customers as experience with system technology changes perceptions of what is possible.

The marriage of high-level language compilers with general-purpose computers allows behaviorally complex, evolutionary systems to be developed at reasonable cost. While the engineering costs of a large software system are considerable, they are much less than the costs of developing a pure hardware system of comparable behavioral complexity. Such a pure hardware system could not be evolved without incurring large manufacturing costs once again. Hardware-centered systems do evolve, but at a slower pace. They tend to be produced in similar groups for several years, then make a major jump to new architectures and capabilities. The time of the jump is associated with the availability of new capabilities and the programmatic capability of replacing an existing infrastructure.

Layering of software as a mechanism for developing greater behavioral complexity is exemplified in the continuous emergence of new software languages and in Internet and Web applications being built on top of distributed infrastructures. The trend in programming languages is to move closer and closer to application domains. The progression of language is from machine level (machine and assembly languages) to general purpose computing (FORTRAN, Pascal, C, C++, Ada) to domain-specific (MatLab, Visual Basic for Applications, dBase, SQL, PERL, and other scripting languages). At each level the models are closer to the application and the language components provide more specific abstractions. By using higher- and higher-level languages, developers are effectively reusing the coding efforts that went into the language's development. Moreover, the new languages provide new computational abstractions or models not immediately apparent in the architecture of the hardware on which the software executes. Consider a logic programming language like PROLOG. A program in PROLOG is more in the nature of hypothesis and theorem proof than arithmetic and logical calculation. But it executes on a general-purpose computer as invisibly as does a FORTRAN program.

Systems, software, and process models

An architectural challenge is to reconcile the integration needs of software and hardware to produce an integrated system. This is both a problem of representation or modeling and of process. Modeling aspects are taken up subsequently in this chapter and in Part Three. On the process side, hardware is best developed with as little iteration as possible, while software can (and often should) evolve through much iteration. Hardware should follow a well-planned design and production cycle to minimize cost, with large scale production deferred to as close to final delivery as possible (consistent with adequate time for quality assurance). But software cannot be reliably developed without access to the targeted hardware platform for much of its development cycle. Production takes place nearly continuously, with release cycles often daily in many advanced development organizations.

Software distribution costs are comparatively so low that repeated complete replacement of the installed base is normal practice. When software firms ship their yearly (or more frequent) upgrades they ship a complete product. Firms commonly "ship" patches and limited updates on the Internet, eliminating even the cost of media distribution. The cycle of planned replacement is so ingrained that some products (for example, software development tools) are distributed as subscriptions; a quarterly CD-ROM with a new version of the product, application notes, documentation, and prerelease components for early review.

In contrast, the costs of hardware are often dominated by the physical production of the hardware. If the system is mass-produced this will clearly be the case. Even when production volumes are very low, as in unique customized systems, the production cost is often comparable to or higher than the development cost. As a result, it is uneconomic, and hence impractical, to extensively replace a deployed hardware system with a relatively minor modification. Any minor replacement must compete against a full replacement, a replacement with an entirely new system designed to fulfill new or modified purposes.

One important exception to the rule of low deployment costs for software is where the certification costs of new releases are high. For example, one does not casually replace the flight control software of the Space Shuttle any more than one casually replaces an engine. Extensive test and certification procedures are required before a new software release can be used. Certification costs are analogous to manufacturing costs in that they are costs required to distribute each release but do not contribute to product development.

Waterfalls for software?

For hardware systems, the process model of choice is a waterfall (in one of its pure or more refined incarnations). The waterfall model tries to keep iterations local, that is, between adjacent tasks such as requirements and design. Upon reaching production there is no assumption of iteration, except the large-scale iteration of system assessment and eventual retirement and/or replacement. This model fits well within the traditional architecting paradigm as described in Chapter 1.

Software can, and sometimes does, use a waterfall model of development. The literature on software development has long embraced the sequential paradigm of requirements, design, coding, test, and delivery. But dissatisfaction with the waterfall model for software led to the spiral model and variants. Essentially all successful software systems are iteratively delivered. Application software iterations are expected as a matter of course. Weapon system and manufacturing software is also regularly updated with refined functionality, new capabilities, and fixes to problems. One reason for software iterations is to fix problems discovered in the field. A waterfall model tries to eliminate such problems by doing a very high-quality job of the requirements. Indeed, the success of a waterfall development is strongly dependent on the quality of the requirements. But in some systems the evolvability of software can be exploited to reach the market faster and avoid costly, and possibly fruitless, requirements searches.

> Example: Data communication systems have an effective requirement of interoperating with whatever happens to be present in the installed base. Deployed systems from a global range of companies may not fully comply with published standards, even if the standards are complete and precise (which they often are not). Hence, determining the "real" requirements to interoperate is quite difficult. The most economical way to do so may be to deploy to the field and compile real experience. But that, in turn, requires that the systems support the ability to determine the cause of interoperation problems and be economically modifiable once deployed to exploit the knowledge gained.

But, in contrast, a casual attitude toward evolution in systems with safety or mission-critical requirements can be tragic.

> Example: The Therac 25 was a software-controlled radiation treatment machine in which software and system failures resulted in six deaths.[3] It was an evolutionary development from a predecessor machine. The evidence suggests that the safety requirements were well understood, but that the system and software architectures both failed to maintain the properties. The system architecture was flawed in that all hardware safety interlocks (which had been present in the predecessor model) were removed, leaving software checks as the sole safeguard. The software architecture was flawed because it did not guarantee the integrity of treatment commands entered and checked by the system operator.

One of the most extensively described software development problems is customized business systems. These are corporate systems for accounting, management, and enterprise-specific operations. They are of considerable economic importance, are built in fairly large numbers (though no two are exactly alike), and are developed in an environment relatively free of government restrictions. Popular and widely published development methods have strongly emphasized detailed requirements development, followed by semi-automated conversion of the requirements to program code — an application-specific waterfall.

While this waterfall is better than *ad hoc* development, results have been disappointing. In spite of years of experience in developing such business systems, large development projects regularly fail. As Tom DeMarco has noted,[4] "somewhere, today, an accounts payable system development is failing" in spite of the thousands of such systems that have been developed in the past. Part of the reason is the relatively poor state of software engineering compared to other fields. Another reason is failure to make effective use of methods known to be effective. An important reason is the lack of an architectural perspective and the benefits it brings.[5]

The architect's perspective is to explicitly consider implementation, requirements, and long-term client needs in parallel. A requirements-centered approach assumes that a complete capture of documentable requirements can be transformed into a satisfactory design. But existing requirements modeling methods generally fail to capture performance requirements and ill-structured requirements like modifiability, flexibility, and availability. Even where these nonbehavioral requirements are captured, they cannot be transformed into an implementation in any even semiautomated way. And it is the nature of serious technological change that the impact will be unpredictable. As technology changes and experience is gained, what is demanded from systems will change as well.

The spiral model as originally described did not embrace evolution. Its spirals were strictly risk based and designed to lead to a fixed system delivery. Rapid prototyping envisions evolution, but only on a limited scale. Newer spiral model concepts do embrace evolution.[6] Software processes, as implemented, spiral through the waterfall phases, but do so in a sequential approach to moving release levels. This modified model was introduced in Chapter 4, in the context of integrating software with manufacturing systems, and it will be further explored below.

Spirals for hardware?

To use a spiral model for hardware acquisition is equivalent to repeated prototyping. A one-of-a-kind, hardware-intensive system cannot be prototyped in the usual sense. A complete "prototype" is, in fact, a complete system. If it performs inadequately it is a waste of the complete manufacturing cost of the final system. Each one, from the first article, needs to be produced as though it were the only one. A prototype for such a system must be a limited version or component intended to answer specific developmental questions. The development process needs to place strong emphasis on requirements development and attention to detailed purpose throughout the design cycle. Mass-produced systems have greater latitude in prototyping because of the prototype-to-production cost ratio, but still less than in software. However, the initial "prototype" units still need to be produced. If they are to be close to the final articles they need to be produced on a similar manufacturing line. But setting up a complete manufacturing line when the system is only in prototype stage is very expensive. Setting up the manufacturing facilities may be more expensive than developing the system. As a hardware-intensive system itself, the manufacturing line cannot be easily modified, and leaving it idle while modifying the product to be produced represents a large cost.

Integration: spirals and circles

What process model matches the nature of evolutionary, mixed technology, behaviorally complex systems? As was suggested earlier, a spiral and circle framework seems to capture the issues. The system should possess stable configurations (represented as circles) and development should iteratively approach those circles. The stable configurations can be software release levels, architectural frames, or hardware configurations.

This process model matches the accepted software practice of moving through defined release levels, with each release produced in cycles of requirements-design-code-test. Each release level is a stable form that is used while the next release is developed. Three types of evolution can be identified. A software product, like an operating system or shrink-wrapped application, has major increments in behavior indicated by changes in the release number, and more minor increments by changes in the number after the "point." Hence a release 7.2 product would be major version seven, second update. The major releases can be envisioned as circles, with the minor releases cycling into them. On the third level are those changes that result in new systems or rearchitected old systems. These are conceptually similar to the major releases, but represent even bigger changes. The process with software annotations is illustrated in Figure 6.1. By using a side view one can envision the major releases as vertical jumps. The evolutionary spiral process moves out to the stable major configurations, then jumps up to the next major change. In practice, evolution on one release level may proceed concurrently with development of a major change.

> Example: The Internet and World Wide Web provide numerous examples of stable intermediate forms promoting evolution. The architecture of the Internet, in the sense of an organizing or unifying structure, is clearly the Internet Protocol (IP), the basic packet switching definition. IP defines how packets are structured and addressed, and how the routing network interacts with the packets. It determines the kinds of

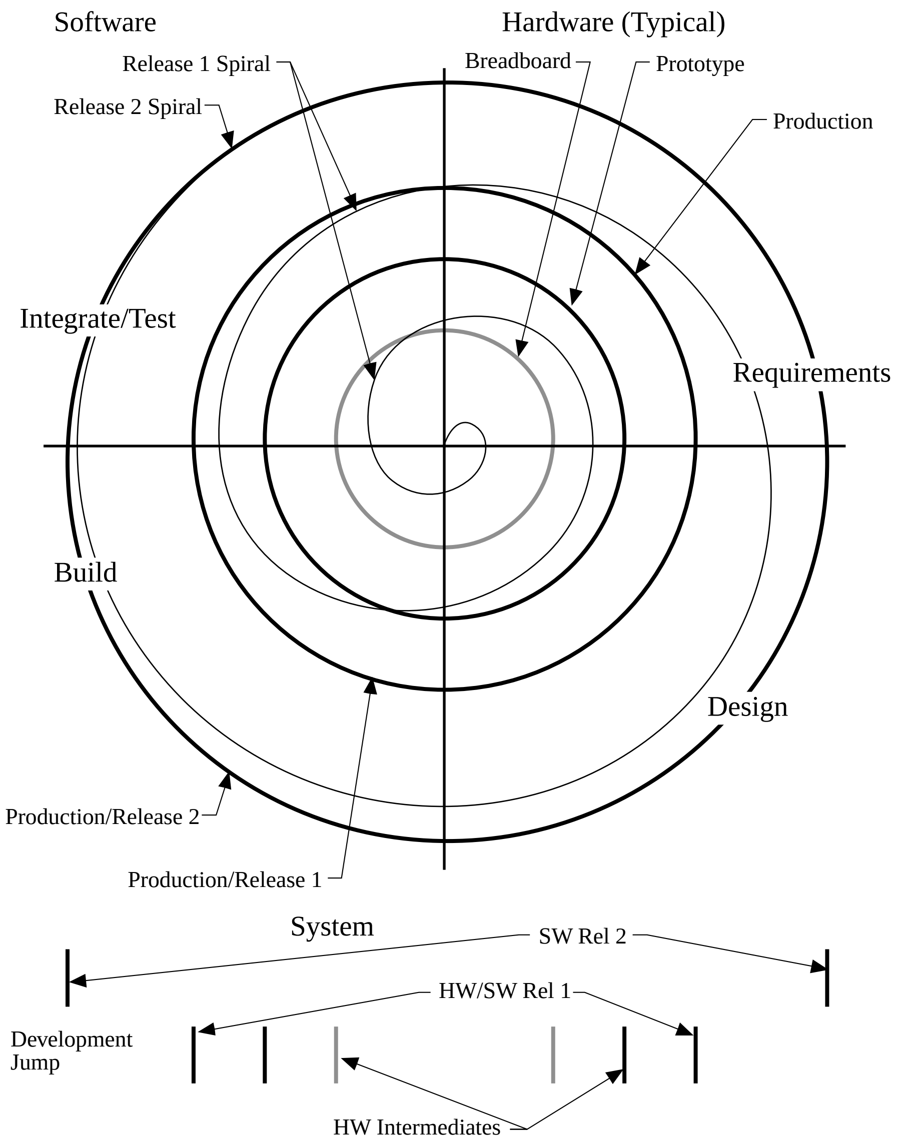


Figure 6.1

services that can be offered on the Internet, and in so doing constrains application construction. As the Internet has undergone unprecedented growth in users, applications, and physical infrastructure, IP has remained stable. Only now is there a slow process of evolving IP, and its success is considered problematic given how entrenched the current version of IP is. The

World Wide Web has similarly undergone tremendous growth and evolution on top of a simple set of elements, the hypertext transfer protocol (http) and the hypertext markup language (html). Both the Internet and the World Wide Web are classic examples of systems with nonphysical architecture, a topic that becomes central in the discussion of collaborative systems in Chapter 7.

Hardware-software integration adds to the picture. The hardware configurations must also be stable forms, but should appear at different points than the software intermediates on the development timeline. Some stable hardware should be available during software development to facilitate that development. A typical development cycle for an integrated hardware-software system illustrates parallel progressions in hardware and software with each reaching different intermediate stable forms. The hardware progression might be wire-wrap breadboard, production prototype, production, then (possibly) field upgrade. The software moves through a development spiral aiming at a release 1.0 for the production hardware. The number of software iterations may be many more than for the hardware. In late development stages, new software versions may be built weekly.[7] Before that, there will normally be partial releases that run on the intermediate hardware forms (the wire-wrap breadboards and the production prototypes). Hardware/software co-design research is working toward environments in which developing hardware can be represented faithfully enough so that physical prototypes are unnecessary for early integration. Such tools may become available, but iteration through intermediate hardware development levels is still the norm in practice.

A related problem in designing a process for integration is the proper use of the heuristic **do the hard part first.** Because software is evolved or iterated, this heuristic implies that the early iterations should address the most difficult challenges. Unfortunately, honoring the heuristic is often difficult. In practice, the first iterations are often good-looking interface demonstrations or constructs of limited functionality. If interface construction is difficult, or user acceptance of the interface is risky or difficult, this may be a good choice. But if operation-to-time constraints under loaded conditions is the key problem, some other early development strategy should be pursued. In that case the heuristic suggests gathering realistic experimental data on loading and timing conditions for the key processes of the system. That data can then be used to set realistic requirements for components of the system in its production configuration.

Example: Call Distribution Systems manage large numbers of phone personnel and incoming lines, as in technical support or phone sales operation. By tying the system into sales databases it is possible to develop

> sophisticated support systems that ensure that full customer information is available in real-time to the phone personnel. To be effective, the integration of data sources and information handling must be customized to each installation and evolve as understanding develops of what information is needed and available. But, because the system is real-time and critical to customer contact, it must provide its principal functionality reliably and immediately upon installation.

Thus, an architectural response to the problems of hardware-software integration is to architect both the process and the product. The process is manipulated to allow different segments of development to match themselves to the demands of the implementation technology. The product is designed with interfaces that allow separation of development efforts where the efforts need to proceed on very different paths. How software architecture becomes an element of system architecture, and more details on how this is to be accomplished, are the subjects to come.

The problem of hierarchy

A central tenet of systems engineering is that all systems can be viewed in hierarchies. A system is composed of subsystems, which are themselves composed of small units. A system is also embedded in higher-level systems in which it acts as a component. One person's system is another person's component. A basic strategy is to decompose any system into subsystems, decompose the requirements until they can be allocated to subsystems, carefully specify and control the interfaces among the subsystems, and repeat the process on every subsystem until you reach components you can buy or are the products of disciplinary engineering. Decomposition in design is followed by integration in reverse. First, the lowest-level components are integrated into the next level subsystems, those subsystems are integrated into larger subsystems, and so on until the entire system is assembled.

Because this logic of decomposition and integration is so central to classical systems engineering, it is difficult for many systems engineers to understand why it often does not match software development very well. To be sure, some software systems are very effectively developed this way. The same logic of decomposition and integration matches applications built in procedural languages (like C or Pascal*) and where the development effort writes all of the application's code. In these software systems the code begins with a top-level routine, which calls first-level routines, which call second-level routines, and so forth, to primitive routines that do not call others. In a strictly procedural language the lower-level routines are contained within

* Strictly speaking, C is not a procedural language and some of what follows does not precisely apply to it. Those knowledgeable in comparative programming languages can consider the details of the procedural vs. object-oriented paradigms in the examples to come.

or encapsulated in the higher-level routines that use them. If the developer organization writes all the code, or uses only relatively low-level programming libraries, the decomposition chain terminates in components much like the hardware decomposition chain terminates. Like in the classical systems engineering paradigm, we can integrate and test the software system in much the same way, testing and integrating from the bottom, up, until we reach the topmost module.

As long as the world looks like this, on both the hardware and software side, we can think of system decompositions as looking like Figure 6.2. This figure illustrates the world, and the position of software, as classical systems engineers would portray it. Software units are contained within the processor units that execute them. Software is properly viewed as a subsystem of the processor unit.

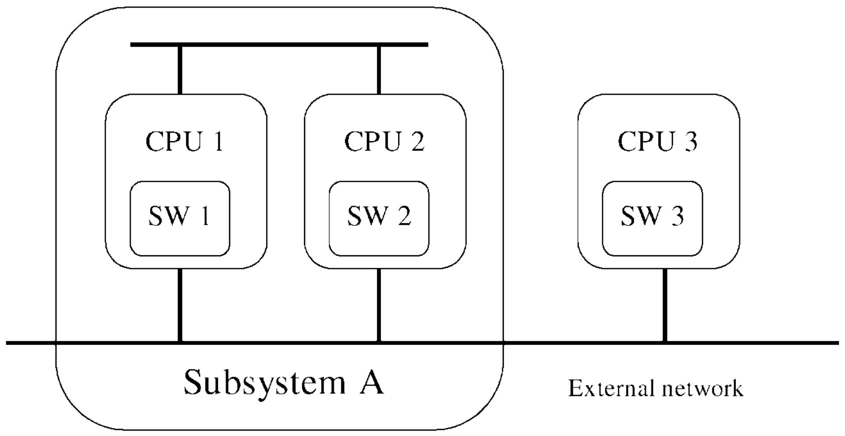


Figure 6.2

However, if we instead went to the software engineering laboratory of an organization building a modern distributed system and asked the software engineers to describe the system hierarchy, we might get a very different story. Much modern software is written using object-oriented abstractions, is built in layers, and makes extensive use of very large software infrastructure objects (like operating systems or databases) that don't look very much like simple components or the calls to a programming library. Each of these issues creates a software environment that does not look like a hierarchical decomposition of encapsulated parts; and, to the extent that a hierarchy exists, it is often quite different from the systems/hardware hierarchy. We consider each of these issues in turn.

Object-orientation

The software community engages in many debates about exactly what "object-oriented" should mean, but only the fundamental concepts are important for *systems* architecting. An object is a collection of functions (often called methods) and data. Some of the functions are public, that is, they are exposed to other software objects and can be called by them. Depending on the specific language and run-time environment, calling a function may be a literal function call, or it may simply mean sending a message to the target object, which interprets it and takes action. Objects can be "active," that is, they can run concurrently with other objects. In some software environments concurrent objects can freely migrate from computer to computer over an intervening network. Often, the software developer does not know, and does not want to know, on which machine a particular object is running, and does not want to directly control its migration. Concurrent execution of the objects is passed to a distributed operating system, which may control object execution through separately defined policies.

In object-oriented systems the number of objects existing when the software executes can be indeterminate. An object has a defining "template" (although the word template means something slightly different in many object-oriented languages) known as a "class." A class is analogous to a type in procedural programming. Thus, just as one can declare many variables of type "float," so one can declare many objects corresponding to a given class. In most object-oriented languages the creation of objects from classes happens at run-time, when the software is executing. If objects are not created until run-time the number of them can be controlled by external events.

This is a very powerful method of composing a software system. Each object is really a computational machine. It has its own data (potentially a very large amount) and as much of its own program code as the class author decides. This sort of dynamic, object-oriented software can essentially manufacture logical machines, in arbitrary numbers, and set them to work on a network in response to events that happen during program execution. To compare this to classical notions of decomposition, it is as though one could create subsystems on the fly during system operation.

Layered design

The objects are typically composed in a layered design. Layers are a form of hierarchy, with a critical difference. In a layered system the lower-level elements (those making up a lower layer) are not contained in the upper layer elements. The elements of a layer interact to produce a set of services, which are made available to the next higher layer (in a strictly layered system). Objects in the next higher layer can use the services offered by the next lower layer, but cannot otherwise access the lower layer objects. Within a layer the objects normally treat each other as peers. That is, no object is contained within another object. However, object-orientation does have the notion of encapsulation. An object has internals, and the internals (functions and data) belong to that object alone, although they can be duplicated in other objects with the same class.

A modern distributed application may be built as a set of interacting, concurrent objects. The objects themselves interact with a lower layer, often called "middleware services." The middleware services are provided by externally supplied software units. Some of the services are part of commercial operating systems, others are individual commercial products. Those middleware components ride on lower layers of network software, supplied as part of the operating system services. In a strong distributed environment, the application programmers, who are writing the objects at the top level, do not know what the network configuration is on which their objects ride. Of course, if there are complex performance requirements, it may be necessary to know and control the network configuration and to program with awareness of its structure. But in many applications no such knowledge is needed, and the knowledge of the application programmers about what code is actually running ceases when the thread of execution leaves the application and enters the middleware and operating systems.

The hierarchy problem is that, at this point, the software hierarchy and the hardware hierarchy have become disconnected. To the software architect the natural structure of the system is layers of concurrent objects as illustrated in Figure 6.3. This means the systems and software architects may clash in their partitioning of the system, and inappropriate constraints may be placed on one or the other. Before investigating the issue of reconciliation, we must complete the discussion with the nature of software components.

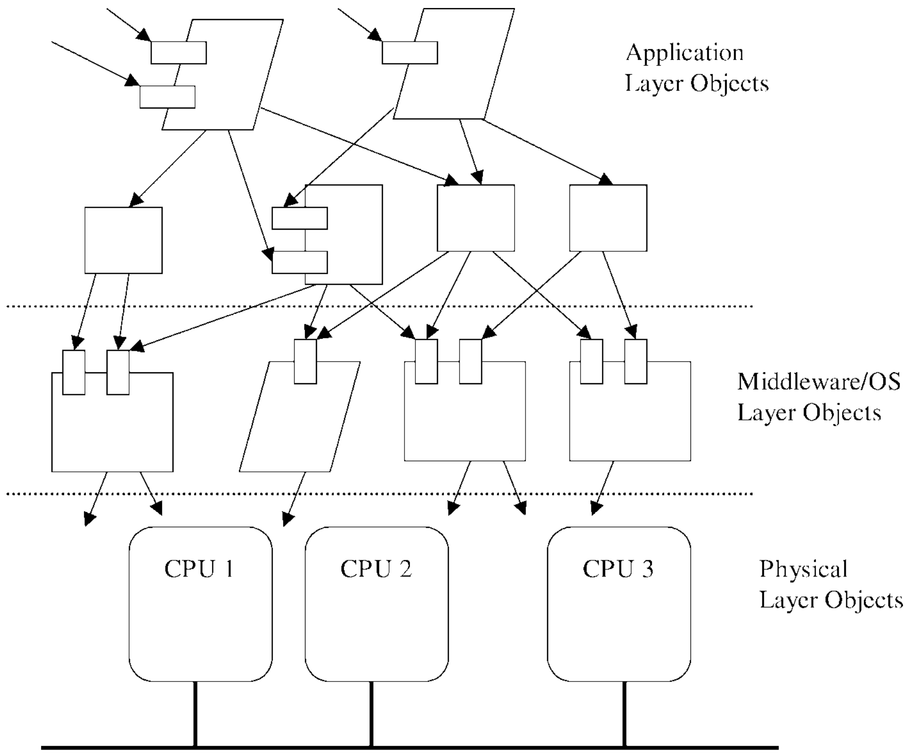


Figure 6.3

Large, autonomous components

When taking a decompositional approach to design, the designer decomposes until he or she reaches components that can be bought or easily built. In both hardware and software some of the components are very large. In software, in particular, the design decomposition are often very large software units, such as operating systems and databases. Both of these are now often millions of lines of programming language code and possess rich functionality. More significantly, they act semiautonomously when used in a system. An operating system is not a collection of functions to be passively called by an application. To be sure, that is one of the services offered by modern operating systems; but modern operating systems also manage program memory, schedule program units on processors, and synchronize concurrent objects across multiple processors. An advanced operating system may present unified services that span many individual computers, possibly geographically widespread.

These large and autonomous components change architecting because the architect is forced to adapt to the components. In principle, of course, the architect and client need not adapt. They can choose to sponsor a from-scratch development instead. But the cost of attempting to replicate the enormous software infrastructure that applications now commonly reuse is prohibitive. For example, the market dominance and complexity of very large databases forces us to use commercial products in these applications. The commercial products support particular kinds of data models, but do not support others. The architecture must take account of the kinds of data models supported, even when those are not a natural choice for the problem.

Reconciling the hierarchies

Our challenge is to reconcile the systems and software worlds. Because software is becoming the dominant element, in terms of its cost pacing what can be developed, one might argue for simply adopting software's models and abandoning the classic systems view. This is inappropriate for several reasons. First, the migration of software to object-oriented, layered structures is only partial. Much software is procedurally structured, and is likely to remain so for many years to come. The infrastructure for supporting distributed, concurrent, object-oriented applications is not mature. While leading-edge applications take this path, many others with strong reliability or just predictability requirements will use more traditional structures.

Second, both approaches are fundamentally valid. Both Figures 6.2 and 6.3 are correct views of the system; they just represent different aspects. No single view can claim primacy. As we move into complex, information-centric systems we will have to accept the existence of many views, each representing different concerns, and each targeted at a different stakeholder audience. The architect, and eventually systems engineers, will have to be sure the multiple views are consistent and complete with respect to the stakeholders' concerns.

Third, every partitioning has its advantages and drawbacks. Building a system in which each computational unit has its own software confined within it has distinct advantages. In that case, each unit will normally have much greater autonomy (because it has its own software and doesn't depend on others). That means each unit can be much more easily outsourced or independently developed. Also, the system does not become dependent of the presence of some piece of software infrastructure. Software infrastructure elements (operating systems and middleware) have a poor record for on-schedule delivery and feature completeness. Anybody depending on an advanced feature of an operating system to be delivered more than a year in the future runs a high risk of being left with nothing when the scheduled delivery date comes.

Nevertheless, the modern approaches have tremendous advantages in many situations. Consider the situation when the units in Figure 6.2 share a great deal of functionality. If separate development teams are assigned to each, the functionality is likely to be independently developed as many times as there are units. Redundant development is likely to be the least of the problems, however, because those independent units probably interact with each other; the test burden has the potential for rising as the square of the number of units. Appropriate code sharing, that is, the use of layered architectures for software, can alleviate both problems.

The role of architecture in software-centered systems

In software as in systems, the architect's basic role is the reconciliation of a physical form with the client's needs for function, cost, certification, and technical feasibility. The mindset is the same as described for system architecting in general, though the areas of concentration are different. System architecting heuristics are generally good software heuristics, though they may be refined and specialized. Several examples are given in Chapter 9. In addition, there are heuristics that apply particularly to software. Some of these are mentioned at the end of this chapter.

The architect develops the architecture. Following Brooks' term,[8] the architect is the user's advocate. As envisioned in this book, the architect's responsibility goes beyond the conceptual integrity of the systems as seen by the user, to the conceptual integrity of the system as seen by the builder and other stakeholders. The architect is responsible for both what the system does as well as how the system does it. But that responsibility extends, on both counts, only as far as is needed to develop a satisfactory and feasible system concept. After all, the sum of both is nearly the whole system, and the architect's role must be limited if an individual or small team is to carry it out. The latter role, of defining the overall implementation structure of the system, is closer to some of the notions of software architecture in recent literature.

The architect's realm is where views and models combine. Where models that integrate disparate views are lacking, the architect can supply the insight. When disparate requirements must interact if satisfaction is to be achieved, the architect's insight can insure that the right characteristics are considered foremost; and, moreover, to develop an architecture that can reconcile the disparate requirements. The perspective required is predominantly a system perspective. It is the perspective of looking at the software and its underlying hardware platforms as an integrated whole that delivers value to the client. Its performance as a whole, behavioral and otherwise, is what gives it its value.

Architecting for evolution is also an example of the **greatest leverage is at the interfaces** heuristic. Make a system evolvable by paying attention to the interfaces. In software, interfaces are very diverse. With a hardware emphasis, it is common to think of communication interfaces at the bit, byte, or message level. But in software communication, interfaces can be much richer and capture extensively structured data, flow of control, and application-specific notions. Current work in distributed computing is a good example. The trend in middleware is to find abstractions well above the network socket that allow flexible composition. Network portable languages like Java allow each machine to express a common interface for mobile code (the Java virtual machine).

Programming languages, models, and expressions

Models are languages. A programming language is a model of a computing machine. Like all languages they have the power to influence, guide, and restrict our thoughts. Programmers with experience in multiple languages understand that some problems will decompose easily in one language, but only with difficulty in another, an example of fitting the architecture of the solution to that of a prescriptive solution heuristic. The development of programming languages has been the story of moving successively higher in abstraction from computing hardware.

The layering of languages is essential to complex software development because a high-level language is a form of software reuse. Assembly languages masked machine instructions while procedural languages modeled computer instructions in a language more like prose. Modern languages containing object and strong structuring concepts continue the pattern by providing a richer palette of representation tools for implementing computing constructs. Each statement in FORTRAN, Pascal, or C reuses the compiler writer's machine-level implementation of that construct. Even more important examples are the application of specific languages like mathematical languages or databases. A statement in a mathematical language like MatLab or Mathematica may invoke a complex algorithm requiring long-term development and deep expertise. A database query language encapsulates complex data storage and indexing codes.

One way of understanding this move up the ladder of abstraction is a famous software productivity heuristic on programmer productivity. A purely programming oriented statement of the heuristic is:

Programmers deliver the same number of lines of code per day regardless of the language they are writing in.

Hence, to achieve high software productivity, programmers must work in languages that require few lines of code.[9] This heuristic can be used to examine various issues in language and software reuse. The nature of a programming language, and the available tools and libraries, will determine the amount of code needed for a particular application. Obviously, writing machine code from scratch will require the most code. Moving to high-level languages like C or Ada will reduce the amount of original code needed, unless the application is fundamentally one that interacts with the computing hardware at a very low level. Still less original code will be required if the language directly embodies application domain concepts, or, equivalently, application-specific code libraries are available.

Application-specific languages imitate domain language already in use and make it suitable for computing. One of the first and most popular is spreadsheets. The spreadsheet combines a visual abstraction and a computational language suited to a range of modeling tasks in business offices, engineering, and science. An extremely important category is database query languages. Today it would be quite unusual to undertake an application requiring sophisticated database functionality and not use an existing database product and its associated query language. Another more recent category is mathematical languages. These languages, such as Mathematica, MacSyma, and MatLab, use well-understood mathematical syntax and then process those languages into computer-processable form. They allow the mathematically literate user to describe solutions in a language much closer to the problem than in a general-purpose programming language.

Application-specific programming languages are likely to play an increasingly important role in all systems built in reasonably large numbers. The only impediment in using these abstractions in all systems is the investment required to develop the language and its associated application generator and tools. One-of-a-kind systems will not usually be able to carry the burden of developing a new language along with a new system unless they fit into a class of system for which a "meta-language" exists. Some work along these lines has been done, for example, in command and control systems.[10]

Architectures, "unifying" models, and visions

Architectures in software can be definitions in terms of tasks and modules, language or model constructs, or, at the highest abstraction level, metaphors. Because software is the most flexible and ethereal of media, its architecture, in the sense of a defining structure, can be equally flexible and ethereal.

The most famous example is the Macintosh desktop metaphor, a true architecture. To a considerable degree, when the overall human interface guidelines are added, this metaphor defines the nature of the system. It defines the types of information that will be handled, and it defines much of the logic or processing. The guidelines force operation to be human-centered; that is, the system continuously parses user actions in terms of the effects on objects in the environment. As a result, Macintosh, and now Microsoft Windows, programs are dominated by a main event loop. The foremost structure the programmer must define is the event loop, a loop in which system-defined events are sequentially stripped from a queue, mapped to objects in the environment, and their consequences evaluated and executed.

The power of the metaphor as architecture is twofold. First, the metaphor suggests much that will follow. If the metaphor is a desktop, its components should operate similarly to their familiar physical counterparts. This results in fast and retentive learning "by association" to the underlying metaphor. Secondly, it provides an easily communicable model for the system that all can use to evaluate system integrity. System integrity is being maintained when the implementation to metaphor is clear.

Directions in software architecting

Software architecture and architecting has received considerable recent attention. There have been several special issues of *IEEE Software Magazine* devoted to software architecture. Starting with Shaw and Garlan's book,[11] a whole series has appeared. Much of the current work in software architecture focuses on architectural structures and their analyses. Much as the term "architectural style" has definite meaning in civil architecture, usage is attached to style in current software work. In the terminology of this book, work on software architecture styles is attempting to find and classify the high-level forms of software and their application to particular software problems.

Focusing on architecture is a natural progression of software and programming research that has steadily ascended the ladder of abstraction. Work on structured programming led to structured design and to the multitasking and object-oriented models to be described in Chapter 10. The next stage of the progression is to further classify the large-scale structures that appear as software systems become progressively larger and more complex.

Current work in software architecture primarily addresses the product of architecting (the structure or architecture) rather than the process of generating it. The published studies cover topics such as classifying architectures, mapping architectural styles to particularly appropriate applications, and the use of software frameworks to assemble multiple related software systems. However, newer books are addressing process and the work on software architecture patterns is effectively work on process in that it provides directive guidance in forming a software architecture. This book presents some common threads of the architectural process that underlie the generation of architectures in many domains. Once a particular domain is entered, such as software, the architect should make full use of the understood styles, frameworks, or patterns in that domain.

The flavor of current work in software architecture is best captured by reviewing some of its key ideas. These include the classification of architectural styles, patterns and pattern languages in software, and software frameworks.

Architectural styles

At the most general level, a style is defined by its components, connectors, and constraints. The components are the things from which the software system is composed. The connectors are the interfaces by which the components interact. A style sets the types of components and connectors that will make up the system. The constraints are the requirements that define system behavior. In the current usage, the architecture is the definition in terms of form, which does not explicitly incorporate the constraints. To understand the constraints one must look to additional views.

As a simple example, consider the structured design models described previously. A pure structured style would have only one component type, the routine, and only one connector type, invocation with explicit data passing. A software system composed only using these components and connectors could be said to be in the structured style; but the notion of style can be extended to include considerations of its application and deviations.

David Garlan and Mary Shaw give this discussion of what constitutes an architectural style*:

> "An architectural style then defines a family of such systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style. Additionally, a style might define topological constraints on architectural descriptions (e.g., no cycles). Other constraints — say, having to do with execution semantics — might also be part of the style definition.
>
> "Given this framework, we can understand what a style is by answering the following questions: What is the structural pattern — the components, connectors, and topologies? What is the underlying computational model? What are the essential invariants of the style — its 'load bearing walls?' What are some common examples of its use? What are the advantages and dis-

* Garlan, D., and Shaw, M., An Introduction to Software Architecture, Technical Report, School of Computer Science, Carnegie Mellon University, p. 6.

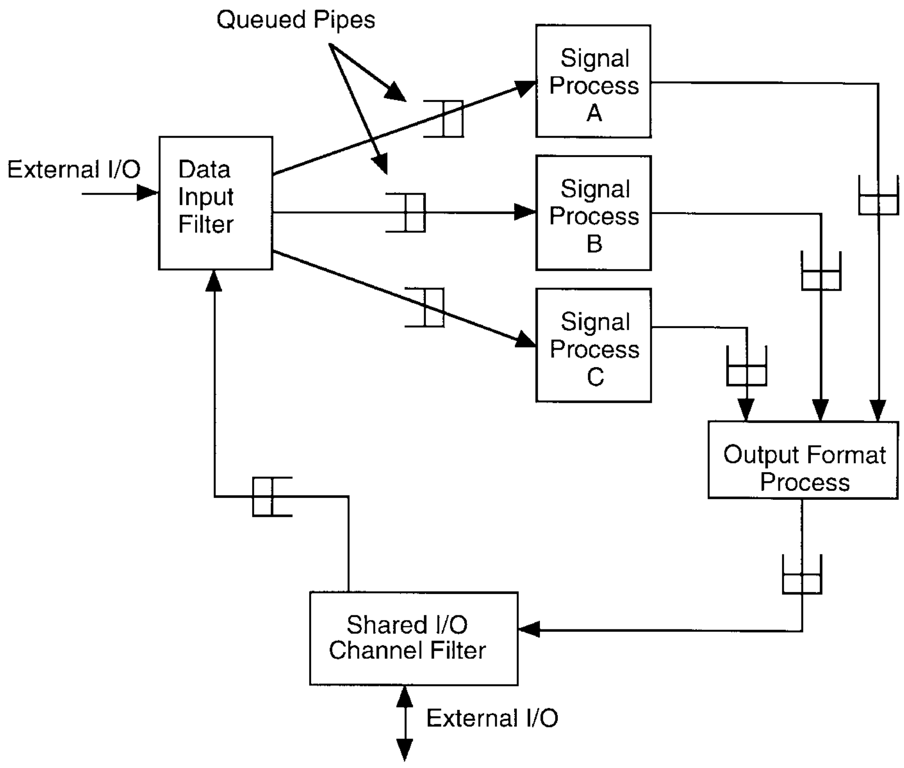


Figure 6.4

> advantages of using that style? What are some of the common specializations?"

Garlan and Shaw have gone on to propose several root styles. As an example, their first style is called "pipe and filter." The pipe and filter style contains one type of component, the filter, and one type of connector, the pipe. Each component inputs and outputs streams of data. All filters can potentially operate incrementally and concurrently. The streams flow through the pipes. Likewise, all stream flows are potentially concurrent. Because each component acts to produce one or more streams from one or more streams, it can be thought of as an abstract sort of filter. A pipe and filter system is schematically illustrated in Figure 6.4.

UNIX shell programs and some signal processing systems are common pipe and filter systems. The UNIX shell provides direct pipe and filter abstractions with the filters concurrent with UNIX processes, and the pipes interprocess communication streams. The pipe and filter abstraction is a natural representation for block-structured signal processing systems in which concurrent entities perform real-time processing on incoming sampled data streams.

Some other styles proposed include object-oriented, event-based, layered, and blackboard. An object-oriented architecture is built from components that encapsulate both data and function and exchange messages. An event-based architecture has as its fundamental structure a loop which receives events (from external interfaces or generated internally), interprets the events in the context of system state, and takes actions based on the combination of event and state. Layered architectures emphasize horizontal partitioning of the system with explicit message passing and function calling between layers. Each layer is responsible for providing a well-defined interface to the layer above. A blackboard architecture is built from a set of concurrent components which interact by reading and writing asynchronously to a common area.

Each style carries its advantages and weaknesses. Each of these styles are descriptions of implementations from an implementer's point of view, and specifically from the software implementer's point of view. They are not descriptions from the user's point of view, or even from the point of view of a hardware implementer on the system. A coherent style, at least of the type currently described, gives a conceptual integrity that assists the builder, but may be no help to the user. Having a coherent implementation style may help in construction, but it is not likely to yield dramatic improvements in productivity or quality because it does not promise to dramatically cut the size of what must be implemented.

This is reflective of a large fraction of the current software architecture literature. The primary focus is on the structure of the software, not on the structure of the problem that the software is to solve. The architecture description languages being studied are primarily higher-level or more abstracted descriptions of programming language constructs. Where user concerns enter the current discussion is typically through analysis. For example, an architecture description language developer may be concerned with how to analyze the security properties of a system description written in the language. This approach might be termed "structuralist." It places the structure of the software first in modeling, and attempts to derive all other views from it. There is an intellectual attraction to this approach since the structural model becomes the root. If the notation for structure can be made consistent, then the other views derived from it should retain that consistency. There is no problem of testing consistency across many views written in different modeling languages. The weakness of the approach is that it forces the stakeholders other than the software developers to use an unfamiliar language and trust unfamiliar analyses. In the security example, instead of using standard methods from the security community, those concerned with security must trust the security analysis performed in the architectural language. This approach may grow to be accepted by broad communities of stakeholders, but it is likely to be a difficult sell.

In contrast to the perspective that places structure first in architecture, this book has repeatedly emphasized that only the client's purpose can be first. The architect should not be removed from the purpose or requirements, the architect should be immersed in them. This is a distinction between architecting as described here and as is often taught in software engineering. We do not assume that requirements precede architecture. The development of requirements is part of architecting, not part of its preconditions.

The ideal style is one that unifies both the user's and builder's views. The mathematical languages mentioned earlier are examples. They structure the system from both a user's and an implementer's point of view. Of course, the internals of the implementation of such a complex software system will contain many layers of abstraction. Almost certainly, new styles and abstractions specific to the demands of implementation in real computers will have to be created internally. When ideal styles are not available, it is still reasonable to seek models or architectural views which unify some set of considerations larger than just the software implementer. For implementation of complex systems it would be a useful topic of research to find models or styles that encompass a joint hardware-software view.

Architecture through composition

Patterns, styles, and layered abstraction are inherent parts of software practice. Except for the rare machine-level program, all software is built from layered abstractions. High-level programming languages impose an intellectual model on the computational machine. The nature of that model inevitably influences what kinds of programs (systems) are built on the machine.

The modern trend is to build systems from components at higher and higher levels of abstraction. This is necessary as no other means is available to build very large and complex systems within acceptable time and effort limits. Each high-level library of components imposes its own style and lends itself to certain patterns. The patterns that match the available libraries are encouraged, and it may be very difficult to implement architectures that are not allowed for in the libraries.

> Example: Graphical Macintosh and Windows programs are almost always centrally organized around an event loop and handlers, a type of event-driven style. This structure is efficient because the operating systems provide a built-in event loop to capture user actions such as mouse clicks and keypresses. However, since neither has multithreading abstractions (at least before 1995), a concurrent, interacting object architecture is difficult to construct. Many applications would benefit from a concurrent interaction object architecture, but these architectures are very difficult to implement within the constraints of existing libraries.

The logical extension is to higher and higher level languages, and from libraries to application-specific languages that directly match the nature of the problem they were meant to solve. The modern mathematical software packages are, in effect, very high-level software languages designed to mimic the problem they are meant to solve. The object of the packages is to do technical mathematics. So, rather than provide a language into which the scientist or engineer must translate mathematics, the package does the mathematics. This is similar for computer-aided design packages and, indeed, most of the shrink-wrap software industry. These packages surround the computer with a layered abstraction that closely matches the way users are already accustomed to work.

Actually, the relationship between application-specific programming language, software package, and user is more symbiotic. Programmers adapt their programs to the abstractions familiar to the users, but users eventually adapt their abstractions to what is available and relatively easy to implement. The best example is the spreadsheet. The spreadsheet as an abstraction partially existed in paper form as the general ledger. The computer-based abstraction has proven so logical that users have adapted their thinking processes to match the structure of spreadsheets. It should probably be assumed that this type of interactive relationship will accelerate when the first generation of children to grow up with computers reaches adulthood.

Heuristics and guidelines in software

The software literature is a rich source for heuristics. Most of those heuristics are specific to the software domain and are often specific to restricted classes of software-intensive systems. The published sets of software heuristics are quite large. The newer edition of Brooks's *The Mythical Man-Month: Essays in Software Engineering*,[12] includes a new chapter, The Propositions of the Mythical Man-Month: True or False?, which lists the heuristics proposed in the original work. The new chapters reinforce some of central heuristics and reject a few others as incorrect.

The heuristics given in "Man-Month" are broad-ranging, covering management, design, organization, testing, and other topics. Several other sources give specific design heuristics. The best sources are detailed design methodologies that combine models and heuristics into a complete approach to developing software in a particular category or style. Chapter 10 discusses three of the best documented: ADARTS,* structured design,** and object-

* The published reference on ADARTS, which is quite thorough, is available through the Software Productivity Consortium, *ADARTS Guidebook*, Version 2.00.13, Vols. 1-2, September, 1991. ADARTS is an Ada language specific method, though its ideas generalize well to other languages. In fact, this has been done, although the resulting heuristics and examples are available only to Software Productivity Consortium members.

** Structured design is covered in many books. The original reference is Yourdon, E. and Constantine, L L., *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Yourdon Press, 1979.

oriented.* Even more specific guidelines are available for the actual writing of code. A book by McConnell[13] contains guidelines for all phases and a detailed bibliography.

From this large source-set, however, there are a few heuristics that particularly stand out as broadly applicable and as basic drivers for software architecting.

- Choose components so that each can be implemented independently of the internal implementation of all others.
- Programmer productivity in lines of code per day is largely independent of language. For high productivity, use languages as close to the application domain as possible.
- The number of defects remaining undiscovered after a test is proportional to the number of defects found in the test. The constant of proportionality depends on the thoroughness of the test, but is rarely less than 0.5.
- Very low rates of delivered defects can be achieved only by very low rates of defect insertion *throughout* software development, and by layered defect discovery — reviews, unit test, system test.
- Software should be grown or evolved, not built.
- The cost of removing a defect from a software system rises exponentially with the number of development phases since the defect was inserted.
- The cost of discovering a defect does not rise. It may be cheaper to discover a requirements defect in customer testing than in any other way, hence the importance of prototyping.
- Personnel skill dominates all other factors in productivity and quality.
- Don't fix bugs later, fix them now.

As has been discussed, the evolvability of software is one of its most unique attributes. A related heuristic is: **a system will develop and evolve much more rapidly if there are stable intermediate forms than if there are not.** In an environment where wholesale replacement is the norm, what constitutes a stable form? The previous discussion has already talked about releases as stable forms and intermediate hardware configurations. From a different perspective, the stable intermediate forms are the unchanging components of the system architecture. These elements that do not change provide the framework within which the system can evolve. If they are well chosen, that is, if they are conducive to evolution, they will be stable and facilitate further development. A sure sign the architecture has been badly chosen is the need to change it on every major release. The architectural elements involved could be the use of specific data or control structures,

* Again, there are many books on object-oriented design, and many controversies about its precise definition and the best heuristics or design rules. *Object-Oriented Modeling and Design,* by Rumbaugh, discussed in Chapter 10, is a good introduction, as is the UML documentation and associated books.

internal programming interfaces, or hardware-software interface definitions. Some examples illustrate the impact of architecture on evolution.

Example: The Point-to-Point Protocol (PPP) is a publicly defined protocol for computer networking over serial connections (such as modems). Its goal is to facilitate broad multivendor interoperability and to require as little manual configuration as possible. The heart of the protocol is the need to negotiate the operating parameters of a changing array of layered protocols (for example, physical link parameters, authentication, IP control, AppleTalk control, compression, and many others). The list of protocols is continuously growing in response to user needs and vendor business perceptions. PPP implements negotiation through a basic state machine that is reused in all protocols, coupled with a framework for structuring packets. In a good implementation, a single implementation of the state machine can be "cloned" to handle each protocol, requiring only a modest amount of work to add each new protocol. Moreover, the common format of negotiations facilitates troubleshooting during test and operation. During the protocols development the state machine and packet structure have been mapped to a wide variety of physical links and a continuously growing list of network and communication support protocols.

Example: In the original Apple Macintosh operating system the architects decided not to use the feature of their hardware to separate "supervisor" and "user" programs. They also decided to implement a variety of application programming interfaces through access to global variables. These choices were beneficial to the early versions because they improved performance. But these same choices (because of backward compatibility demands) have greatly complicated efforts to implement advanced operating system features such as protected memory and preemptive multitasking. Another architectural choice was to define the hardware-software interface through the Macintosh Toolbox and the published Apple programming guidelines. The combination has proven to be both flexible and stable. It has allowed a long series of dramatic hardware improvements, and now a transfer to a new hardware architecture, with few gaps in backward compatibility (at least for those developers who obeyed the guidelines).

Example: The Internet Protocol, combined with the related Transport Control Protocol (TCP/IP), has become the software backbone of the global Internet. Its partitioning of data handling, routing decisions, and flow control has proven to be robust and amenable to evolutionary development. The combination has been able to operate across extremely heterogeneous networks with equipment built by countless vendors. While there are identifiable architects of the protocol suite, control of protocol development is quite distributed with little central authority. In contrast, the proprietary networking protocols developed and controlled by several major vendors have frequently done relatively poorly at scaling to diverse networks. One limitation in the current IP protocol suite that has become clear is the inadequacy of its 32-bit address space. However, the suite was designed from the beginning with the capability to mix protocol versions on a network. As a result, the deployed protocol version has been upgraded several times (and will be again to IPv6) without interfering with the ongoing operation of the Internet.

Exercises

1. Consult one or more of the references for software heuristics. Extract several heuristics and use them to evaluate a software-intensive system.
2. Requirements defects that are delivered to customers are the most costly because of the likelihood they will require extensive rework. But discovering such defects anytime before customer delivery is likewise very costly because only the customers' reaction may make the nature of the defect apparent. One approach to this problem is prototyping to get early feedback. How can software be designed to allow early prototyping and feedback of the information gained without incurring the large costs associated with extensive rework?
3. Pick three software-intensive systems of widely varying scope. For example, a pen computer-based data entry system for warehouses, an internetwork communication server, and the flight control software for a manned space vehicle. What are the key determinants of success and failure for each system? As an architect, how would these deter-

minants change your approach to concept formulation and certification?

4. Examine some notably successful or unsuccessful software-intensive systems. To what extent was success or failure due to architectural (conceptual integrity, feasibility of concept, certification) issues and to what extent was it due to other software process issues.
5. Are their styles analogous to those proposed for software that jointly represent hardware and software?

Notes and references

1. *IEEE Software Magazine,* Special issue on the The Artistry of Software Architecture, November 1995. This issue contains a variety of papers on software architecture, including the role of architecture in reuse, comparisons of styles, decompositions by view, and building-block approaches.
2. Kruchten, P. B., A 4+1 view model of software architecture, *IEEE Software Magazine,* p. 42, November 1995.
3. Leveson, N. G. and Turner, C. S., An investigation of the Therac 25 accidents, *Computer,* p. 18, July 1993.
4. DeMarco and Lister, *Peopleware: Productive Projects and Teams,* Dorset House, New York, 1987.
5. Lambert, B., Beyond Information Engineering: The Art of Information Systems Architecture, Technical Report, Broughton Systems, Richmond, VA, 1994.
6. Software Productivity Consortium, Herndon, VA, *Evolutionary Spiral Process,* Vol. 1-3, 1994.
7. Maguire, S., *Debugging the Development Process,* Microsoft Press, Redmond, WA, 1994.
8. Brooks, F., *The Mythical Man Month,* Addison-Wesley, Reading, MA, 1975. This classic book has recently been republished with additional commentary by Brooks on how his observations have held up over time; p. 256 in the new edition.
9. Data associated with this heuristic can be found in several books. Two sources are *Programming Productivity* and *Applied Software Measurement,* both by Capers Jones, McGraw Hill, New York.
10. Domain-Specific Software Architectures, USC Information Sciences Institute, Technical Report, 1996.
11. Shaw, M. and Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline,* Prentice-Hall, Englewood Cliffs, NJ, 1996.
12. Brooks, F., *The Mythical Man Month,* Addison-Wesley, Reading, MA, 1975.
13. McConnell, S., *Code Complete: A Practical Handbook of Software Construction,* Microsoft Press, Redmond, WA, 1993.

chapter seven

Collaborative systems

Introduction: collaboration as a category

Most of the systems discussed thus far have been the products of deliberate and centrally controlled development efforts. There was an identifiable client or customer (singular or plural), clearly identifiable builders, and users. Client, in the traditional sense, means the person or organization who sponsors the architect and who has the resources and authority to construct the system of interest. The role of the architect existed, even if it was hard to trace to a particular individual or organization. The system was the result of deliberate value judgment by the client and existed under the control of the client. However, many systems are not under central control, either in their conception, their development, or their operation. The Internet is the canonical example, but many others exist, including electrical power systems, multinational defense systems, joint military operations, and intelligent transportation systems. These systems are all collaborative in the sense that they are assembled and operate through the voluntary choices of the participants, not through the dictates of an individual client. These systems are built and operated only through a collaborative process.

A problem in this area is the lack of standard terminology for categories of system. Any system is an assemblage of elements that possesses capabilities not possessed by an element. This is just saying that a system possesses emergent properties, indeed that possessing emergent properties is the defining characteristic of a system. A microwave oven, a laptop computer, and the Internet are all systems, but each can have radically different problems in design and development.

This chapter discusses systems distinguished by the voluntary nature of the systems assembly and operation. Examples of systems in this category include most intelligent transport systems,[1] military C4I and Integrated Battlespace,[2] and partially autonomous flexible manufacturing systems.[3] The arguments here apply to most of what are often referred to as systems-of-systems, a term some readers may prefer. One of the authors (Maier) has discussed the contrast between the concepts elsewhere.[4]

What exactly is a collaborative system? In this chapter a system is a "collaborative system" when its components:

1. Fulfill valid purposes in their own right, and continue to operate to fulfill those purposes if disassembled from the overall system
2. Are managed (at least in part) for their own purposes rather than the purposes of the whole; the component systems are separately acquired and integrated but maintain a continuing operational existence independent of the collaborative system

Misclassification as a "conventional" system vs. a collaborative system (or vice versa) leads to serious problems. Especially important is a failure to architect for robust collaboration when direct control is impossible or inadvisable. This can arise when the developers believe they have greater control over the evolution of a collaborative system than they actually do. In believing this, they may fail to ensure that critical properties or elements will be incorporated by failing to provide a mechanism matched to the problem.

As with other domains, collaborative systems have their own heuristics, and familiar heuristics may have new applications. To find them for collaborative systems we look first at important examples, and then we generalize to find the heuristics. A key point is the heightened importance of interfaces and the need to see interfaces at many layers. The explosion of the Internet and the World Wide Web is greatly facilitating collaborative system construction, but we find that the "bricks and mortar" of Internet-based collaborative systems are not at all physical. The building blocks are communication protocols, often at higher layers in the communications stack that is familiar from past systems.

Collaborative system examples

Systems built and operated voluntarily are not unusual, even if they seem very different from classical systems engineering practice. Most of the readers of this book will be living in capitalist democracies where social order through distributed decisions is the philosophical core of government and society. Nations differ in the degree to which they choose to centralize vs. decentralize decision making, but the fundamental principle of organization is voluntary collaboration. This book is concerned with technological systems, albeit sometimes systems with heavy social or political overtones. So, we take as our examples systems whose building blocks are primarily technical. The initial examples are the Internet, intelligent transportation systems (for road traffic), and joint air defense systems.

The Internet

When we say "the Internet" we are not referring to the collection of applications that have become so popular (e-mail, World Wide Web, chats, etc.).

Instead, we are referring to the underlying communications infrastructure on which the distributed applications run. A picture of the Internet that tried to show all of the physical communications links active at one time would be a sea of lines with little or no apparent order; but, properly viewed, the Internet has a clear structure. The structure is a set of protocols called TCP/IP for Transmission Control Protocol/Internet Protocol. Their relationship to other protocols commonly encountered in the Internet is shown in Figure 7.1.[5] The TCP/IP suite includes the IP, TCP, and UDP protocols in the figure. Note in Figure 7.1 that all of the applications shown ultimately depend on IP. Applications can use only communications services supported by IP. IP, in turn, runs on many link and physical layer protocols. IP is "link friendly" in that it can be made to work on nearly any communications channel. This has made it easy to distribute widely, but prevents much exploitation of the unique features of any particular communication channel.

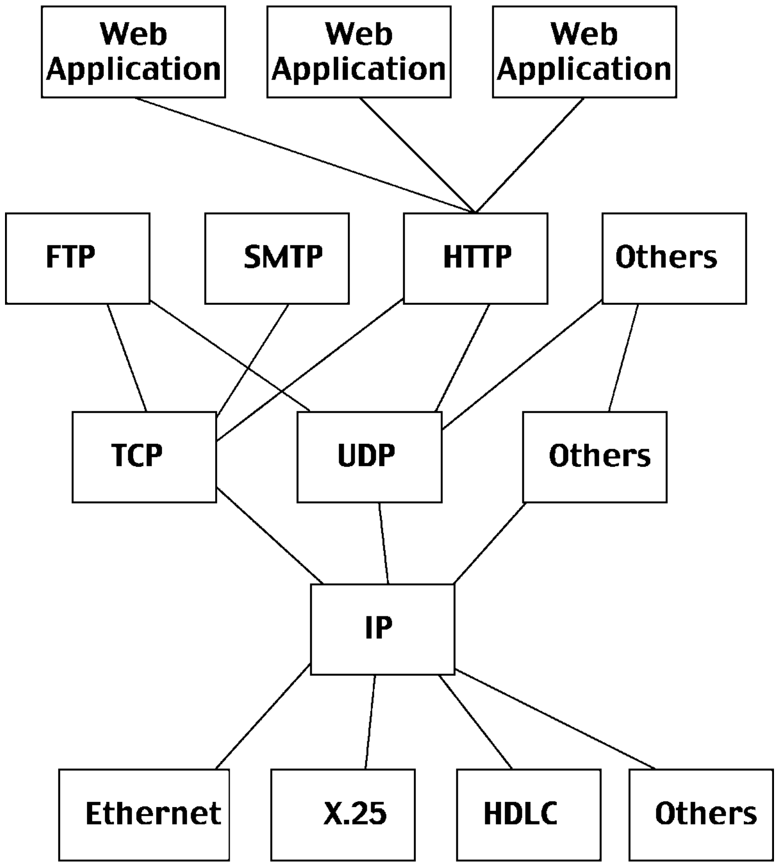


Figure 7.1

The TCP/IP family protocols are based on distributed operation and management. All data is encapsulated in packets, which are independently forwarded through the Internet. Routing decisions are made locally at each routing node. Each routing node develops its own estimate of the connection state of the system through the exchange of routing messages (also encapsulated as IP packets). The distributed estimates of connection state are not, and need not be, entirely consistent or complete. Packet forwarding works in the presence of *some* errors in the routing tables (although introduction of bad information can also lead to collapse).

The distributed nature of routing information, and the memoryless forwarding, allows the Internet to operate without central control or direction. A decentralized development community matches this decentralized architecture. There is no central body with coercive power to issue or enforce standards. There is a central body which issues standards, the Internet Engineering Task Force (IETF), but its practices are unlike nearly any other standards body. The IETF approach to standards is, fundamentally, to issue only those which have already been developed and deployed. Almost anybody can go to the IETF and try and form a working group to build standards in a given area. The organization accepts nearly any working group that has the backing of a significant subset of participants. The working group can issue "internet-drafts" with minimal overhead. For a draft to advance to the Internet equivalent of a published standard it must be implemented and deployed by two or more independent organizations. All Internet standards are available for free, and very strong efforts are made to keep them unencumbered by intellectual property. Proprietary elements are usually accepted only as optional extensions to an open standard.

Distributed operation, distributed development, and distributed management are linked. The Internet can be developed in a collaborative way largely because its operation is collaborative. Because the Internet uses best-effort forwarding and distributed routing, it can easily offer new services without changing the underlying protocols. Those new services can be implemented and deployed by groups that have no involvement in developing or operating the underlying protocols; but only so long as those new services do not require any new underlying services. For example, groups were able to develop and deploy IP-Phone (a voice over the Internet application) without any cooperation from TCP/IP developers or even Internet service providers. However, the IP-Phone application cannot offer any quality of service guarantees because the protocols it is built on do not offer simultaneous delay and error rate bounding.

In contrast, networks using more centralized control can offer richer building block network services, including quality of service guarantees. However, they are much less able to allow distributed operation. Also, the collaborative environments that have produced telecommunications standards have been much slower moving than the Internet standards bodies. They have not adopted some of the practices of the Internet bodies that have enabled them to move quickly and rapidly capture market share. Of course, some of those practices would threaten the basic structure of the existing standards organizations.

In principle, a decentralized system like the Internet should be less vulnerable to destructive collective phenomena and be able to locally adapt around problems. In practice, both the Internet with its distributed control model and the telephone system with its greater centralization have proven vulnerable to collective phenomena. It turns out that distributed control protocols like TCP/IP are very prone to collective phenomena in both transmission and routing (Bertsekas, 1992, Chapter 6). Careful design and selection of parameters has been necessary to avoid network collapse phenomena. One reason is that the Internet uses a "good intentions" model for distributed control which is vulnerable to nodes that misbehave either accidentally or deliberately. There are algorithms known which are robust against bad intentions faults, but they have not been incorporated into network designs. The decentralized nature of the system has made it especially difficult to defend against coordinated distributed attacks (e.g., distributed denial of service attacks). Centralized protocols often deal more easily with these attacks since they have strong knowledge of where connections originate, and can perform aggressive load-shedding policies under stress.

Wide area telephone blackouts have attracted media attention and shown that the more centralized model is also vulnerable. The argument about decentralized vs. centralized fault tolerance has a long history in the electric power industry, and even today it has not reached full resolution.

Intelligent transportation systems

The goal of most initiatives in intelligent transportation is to improve road traffic conditions through the application of information technology. This subject is large and cannot be addressed in detail here.[6] We picked out one issue to illustrate how a collaborative system may operate, and the architectural challenges in making it happen.

One intelligent transportation concept is called "fully coupled routing and control." In this concept a large fraction of vehicles are equipped with devices that determine their position and periodically report it to a traffic monitoring center. The device also allows the driver to enter his or her destination when beginning a trip. The traffic center uses the traffic conditions report to maintain a detailed estimate of conditions over a large metropolitan area. When the center gets a destination message it responds with a recommended route to that destination, given the vehicle's current position. The route could be updated during travel if warranted. The concept is referred to as fully coupled because the route recommendations can be coupled with traditional traffic controls (e.g., traffic lights, on-ramp lights, reversible lanes, etc.).

Obviously, the concept brings up a wide array of sociotechnical issues. Many people may object to the lack of privacy inherent in their vehicle periodically reporting its position. Many people may object to entering their destination and having it reported to a traffic control center. Although there are many such issues, we narrow down, once again, to just one concept that best illustrates collaborative system principles. The concept only works if:

1. A large fraction of vehicles have, and use, the position reporting device.
2. A large fraction of drivers enter their (actual) destination when beginning a trip.
3. A large fraction of drivers follow the route recommendations they are given.

Under current conditions, vehicles on the roads are mostly privately owned and operated for the benefit of their owners. With respect to the collaborative system conditions, the concept meets it if using the routing system is voluntary. The vehicles continue to work whether or not they report their position and destination; and vehicles are still operated for their owners' benefit, not for the benefit of some "collective" of road users. So, if we are architecting a collaborative traffic control system, we have to explicitly consider how the three conditions above needed to gain the emergent capabilities are ensured.

One way to ensure them is to not make the system collaborative. Under some social conditions we can ensure all of the above conditions by making them legally mandatory and providing enforcement. It is a matter of judgment whether or not such a mandatory regime could be imposed.

If one judges that a mandatory regime is impossible, then the system must be collaborative. Given that it is collaborative, there are many architectural choices that can enhance the cooperation of the participants. For example, we can break apart the functions of traffic prediction, routing advice, and traditional controls and allocate some to private markets. Imagine an urban area with several "Traffic Information Provider" services. These services are private and subscription-based, receive the position and destination messages, and disseminate the routing advice. Each driver voluntarily chooses a service, or none at all. If the service provides accurate predictions and efficient routes, it should thrive. If it cannot provide good service, it will lose subscribers and die.

Such a distributed, market-based system may not be able to implement all of the traffic management policies that a centralized system could. However, it can facilitate social cooperation in ways the centralized system cannot. A distributed, market-based system also introduces technical complexities into the architecture that a centralized system does not. In a private system it must be possible for service providers to disseminate their information securely to paying subscribers. In a public, centralized system, information on conditions can be transmitted openly.

Joint air defense systems

A military system may seem like an odd choice for a collaborative system. After all, military systems work by command, not voluntary collaboration. Leaving aside the social issue that militaries must always induce loyalty, which is a social process, the degree to which there is a unified command on military systems or operations is variable. A system acquired and operated on a single service can count on central direction. A system that comes together only in the context of a multiservice, multinational, joint military operation cannot count on central control.

All joint military systems and operations have a collaborative element, but here we consider just air defense. An air defense system must fuse a complex array of sensors (ground radars, airborne radars, beacon systems, human observers, and other intelligence systems) into a logical picture of the airspace and then allocate weapon systems to engage selected targets. If the system includes elements from several services or nations, conflicts will arise. Nations, and services, may want to preferentially protect their own assets. Their command channels and procedures may affect greater self-protection, even when ostensibly operating solely for the goals of the collective.

Taking a group of air defense systems from different nations and different services and creating an effective integrated system from them is the challenge. The obvious path might be to try and convert the collection into something resembling a single service air defense system. This would entail unifying the command, control, and communications infrastructure. It would mean removing the element of independent management that characterizes collaborative systems. If this could be done, it is reasonable to expect that the resulting integrating system would be closer to a kind of point optimum. But, the difficulties of making the unification are likely to be insurmountable.

If, instead, we accept the independence, then we can try and forge an effective collaborative system. The technical underpinnings are clearly important. If the parts are going to collaborate to create integrated capabilities greater than the sum of the parts, they are going to have to communicate. So, even if command channels are not fully unified, communications must be highly interoperable. In this example, as in other sociotechnical examples, the social side should not be ignored. It is possible that the most important unifying elements in this example will be social. These might include shared training or educational background, shared responsibility, or shared social or cultural background.

Analogies for architecting collaborative systems

One analogy that may apply is the urban planner. The urban planner, like the architect, develops overall structures. While the architect structures buildings for effective use by the client, the urban planner structures effective communities. The client of an urban planner is usually a community government or one of its agencies. The urban planner's client and the architect's client differ in important respects. The architect's client is making value judgments for him or herself, and presumably has the resources to put into action whatever plan is agreed to with the architect. When the architect's plan is received, the client will hire a builder. The urban planner's client does not actually build the city. The plan is to constrain and guide many other developers and architects who will come later, and hopefully guide their efforts into a whole greater than if there had been no overall plan. The urban planner and client are making value judgments for other people, the people who will one day inhabit the community being planned. The urban planner's client usually lacks the resources to build the plan, but can certainly stop something from being built if it is not in the plan. To be successful, the urban planner and client have to look outward and sell their vision. They cannot bring it about without others' aid, and they normally lack the resources and authority to do it themselves.

Urban planning also resembles architecting in a spiral or evolutionary development process more than in the waterfall. An urban plan must be continuously adapted as actual conditions change. Road capacity that was adequate at one time may be inadequate at another. The mix of businesses that the community can support may change radically. As actual events unfold, the plan must adapt and be resold to those who participate in it, or it will be irrelevant.

Another analogy for collaborative systems is in business relationships. A corporation with semi-independent division is a collaborative system if the divisions have separate business lines, individual profit and loss responsibilities, and also collaborate to make a greater whole. Now consider the problem of a post-merger company. Before the merger the components (the companies that are merging) were probably centrally run. After the merger the components may retain significant independence. Now if they are to jointly create something greater they must do it through a collaborative system instead of their traditional arrangement. If the executives do not recognize this and adapt, it is likely to fail. A franchise that grants its franchisees significant independence is also like a collaborative system. It is made up of independently owned and operated elements which combine to be something greater than they would achieve individually.

Collaborative system heuristics

As with builder-architecting, manufacturing, sociotechnical, and software-intensive systems, collaborative systems have their own heuristics. The heuristics discussed here have all been given previously, either in this book or its predecessor. What is different is their application. Looking at how heuristics are applied to different domains gives a greater appreciation for their use and applicability in all domains.

Stable intermediate forms

The heuristic on stable intermediate forms is given (Rechtin, E., 1991) as:

> Complex systems will develop and evolve within an overall architecture much more rapidly if there are stable intermediate forms than if there are not.

It is good practice in building a home or bridge to have a structure that is self-supporting during construction. So, in other systems it is important to design them to be self-supporting before they reach the final configuration. For better or worse, in collaborative systems it cannot be assumed that all participants will continue to collaborate. The system will evolve based on continuous self-assessments of the desirability for collaboration by the participants.

Stability means that intermediate forms should be technically, economically, and politically self-supporting. Technical stability means that the system operates to fulfill useful purposes. Economic stability means that the system generates and captures revenue streams adequate to maintain its operation. Moreover, it should be in the economic interests of each participant to continue to operate rather than disengage. Political stability can be stated as the system has a politically decisive constituency supporting its continued operation, a subject we return to in Chapter 12.

- Integrated air defense systems are subject to unexpected and violent "reconfiguration" in normal use. As a result, they are designed with numerous fallback modes, down to the anti-aircraft gunner working on his own with a pair of binoculars. Air defense systems built from weapon systems with no organic sensing and targeting capability have frequently failed in combat when the network within which they operate has come under attack.
- The Internet allows components nodes to attach and detach at will. Routing protocols adapt their paths as links appear and disappear. The protocol encapsulation mechanisms of IP allow an undetermined number of application layer protocols to simultaneously coexist.

Policy triage

This heuristic gives guidance in selecting components, and in setting priorities and allocating resources in development. It is given (Rechtin, E., 1991)as:

> The triage: Let the dying die. Ignore those who will recover on their own. And treat only those who would die without help.

Triage can apply to any systems, but especially applies to collaborative systems. Part of the scope of a collaborative system is deciding what not to control. Attempting to overcontrol will fail for lack of authority. Undercontrol will eliminate the system nature of the integrated whole. A good choice enhances the desired collaboration.

- Motion Picture Experts Group (MPEG) chose to only standardize the information needed to decompress a digital video stream.[7] The standard defines the format of the data stream and the operations required to reconstruct the stream of moving picture frames. However, the compression process is deliberately left undefined. By standardizing decompression, the usefulness of the standard for interoperability was assured. By not standardizing compression, the standard leaves open a broad area for the firms collaborating on the standard to continue to compete. Interoperability increases the size of the market, a benefit to the whole collaborative group, while retaining a space for competition eliminates a reason to not collaborate with the group. Broad collaboration was essential both to ensure a large market, and to ensure that the requisite intellectual property would be offered for license by the participants.

Leverage at the interfaces

Two heuristics, here combined, discuss the power of the interfaces. The greatest leverage in system architecting is at the interfaces. The greatest dangers are also at the interfaces.

When the components of a system are highly independent, operationally and managerially, the architecture of the system *is* the interfaces. The architect is trying to create emergent capability. The emergent capability is the whole point of the system; but, the architect may only be able to influence the interfaces among the nearly independent parts. The components are outside the scope and control of an architect of the whole.

- The Internet oversight bodies concern themselves almost exclusively with interface standards. Neither physical interconnections nor applications above the network protocol layers are standardized. Actually, both areas are the subject of standards, but not the standards process of the IETF.

One consequence is attention to different elements than in a conventional system development. For example, in a collaborative system, issues like life cycle cost are of low importance. The components are developed collaboratively by the participants, who make choices to do so independently of any central oversight body. The design team for the whole cannot choose to minimize life cycle cost, nor should they, because the decisions that determine costs are outside their scope. The central design team can choose interface standards, and can choose them to maximize the opportunities for participants to find individually beneficial investment strategies.

Ensuring cooperation

If a system requires voluntary collaboration, the mechanism and incentives for that collaboration must be designed in.

In a collaborative, the components actively choose to participate or not. Like a market, the resulting system is the web of individual decisions by the participants. Thus, the economists' argument that the costs and benefits of collaboration should be superior to the costs and benefits of independence for each participant individually should apply. As an example, the Internet maintains this condition because the cost of collaboration is relatively low (using compliant equipment and following addressing rules) and the benefits are high (access to the backbone networks). Similarly in MPEG video standards, compliance costs can be made low if intellectual property is pooled, and the benefits are high if the targeted market is larger than the participants could achieve with proprietary products. Without the ability to retain a competitive space in the market (through differentiation on compression in the case of MPEG), the balance might have been different. Alternatively, the cost of noncompliance can be made high, though this method is used less often.

An alternative means of ensuring collaboration is to produce a situation in which each participant's well-being is partially dependent on the well-being on the other participants. This joint utility approach is known, theoretically, to produce consistent behavior in groups. A number of social mechanisms can be thought of as using this principle. For example, strong social indoctrination in military training ties the individual to the group and serves as a coordinating operational mechanism in integrated air defense.

Another way of looking at this heuristic is through the metaphor of the franchise. The heuristic could be rewritten for collaborative systems as:

> Consider a collaborative system a franchise. Always ask why the franchisees choose to join, and choose to remain.

Variations on the collaborative theme

The two criteria provide a sharp definition of a collaborative system, but they still leave open many variations. Some collaborative systems are really centrally controlled, but the central authority has decided to devolve authority in the service of system goals. In some collaborative systems a central authority exists, but power is expressed only through collective action. The participants have to mutually decide and act to take the system in a new direction. And, finally, some collaborative systems lack any central authority. They are entirely emergent phenomena.

A collaborative system where central authority exists and can act is called a closed collaborative system. Closed collaborative systems are those in which the integrated system is built and managed to fulfill specific purposes. It is centrally managed during long-term operation to continue to fulfill those purposes and any new ones the system owners may wish to address. The component systems maintain an ability to operate independently, but their normal operational mode is subordinated to the central managed purpose. For example, most single-service air defense networks are centrally managed to defend a region against enemy systems, although the component systems retain the ability to operate independently, and do so when needed under the stress of combat.

"Open" collaborative systems are distinct from the closed variety in that the central management organization does not have coercive power to run the system. The component systems must, more or less, voluntarily collaborate to fulfill the agreed-upon central purposes. The Internet is an open collaborative system. The IETF works out standards, but has no power to enforce them. IETF standards work because the participants choose to implement them without proprietary variations, at least for the most part.

As the Internet becomes more important in daily life, in effect as it becomes a new utility like electricity or the telephone, it is natural to wonder whether or not the current arrangement can last. Services on which public safety and welfare depend are regulated. Public safety and welfare, at least in industrial countries, is likely to depend on Internet operation in the near future, if it does not already. So, will the Internet and its open processes eventually come under regulation? It may, although the most recent trends have actually been away from centralization. For example, authority over domain naming, which is central to Internet management, has been broken up among competing companies at the insistence of the U.S. government in international negotiations.

Virtual collaborative systems lack both a central management authority and centrally agreed-upon purposes. Large-scale behavior emerges, and may be desirable, but the overall system must rely upon relatively invisible mechanisms to maintain it.

A virtual system may be deliberate or accidental. Some examples are the current form of the World Wide Web and national economies. Both "systems" are distributed physically and managerially. The World Wide Web is even more distributed than the Internet in that no agency ever exerted direct central control, except at the earliest stages. Control has been exerted only through the publication of standards for resource naming, navigation, and document structure. Although essentially just by social agreement, major decisions about Web architecture filter through very few people. Web sites choose to obey the standards or not, at their own discretion. The system is controlled by the forces that make cooperation and compliance to the core standards desirable. The standards do not evolve in a controlled way, rather they emerge from the market success of various innovators. Moreover, the purposes the system fulfills are dynamic and change at the whim of the users.

National economies can be thought of as virtual systems. There are conscious attempts to architect these systems through politics, but the long-term nature is determined by highly distributed, partially invisible mechanisms. The purposes expressed by the system emerge only through the collective actions of the system's participants.

Misclassification

Two general types of misclassification are possible. One is to incorrectly regard a collaborative system as a conventional system, or the reverse. Another is to misclassify a collaborative system as directed, voluntary, or virtual.

In the first case, system vs. collaborative system, consider open source software. Open source software is often thought of as synonymous with Linux, a particular open source operating system. Actually, there is a great deal of open source, "free" software. Software is usually considered open source if anybody can obtain the source code, use it, modify it, and redistribute it for free. Because Linux has been spectacularly successful in market share growth, and (not incidentally) in creating initial public offering riches, many others have tried to emulate the open source model. The open source model is built on a few basic principles,[8] perhaps heuristics. These include:

1. Designs, and initial implementations, should be carried out by gifted individuals or very small teams.
2. Software products should be released to the maximum possible audienc, as quickly as possible.
3. Users should be encouraged to become testers, and even co-developers by providing source code.
4. Code review and debugging can be arbitrarily parallalized, at least if you distribute source code to your reviewers and testers.
5. Incremental delivery of small increments, with a very large user/tester population, leads to very rapid development of high-quality software.*

Of course, a side effect of this model is losing the ability to make any significant amount of money distributing software you have written. The open source movement advocates a counterpoint that effective business models may still be built on service and customization, but some participants in the process are accustomed to the profit margins normally gained from manufacturing software. A number of companies and groups outside of the Linux community have tried to exploit the success of the Linux model for other classes of products, with very mixed results.

* The speed and quality of Linux releases can be measured and it is clearly excellent. Groups of loosely coordinated programmers achieve quality levels equivalent to those of well-controlled development processes in corporations. This point is even admitted in the Microsoft "Halloween" memos on Linux published at http://www.opensource.org/.

Some of this can be understood by realizing that open source software development is a collaborative system. Companies or groups that have open-sourced their software without success typically run into one of two problems that limit collaboration. First, many of the corporate open source efforts are not fully open. For example, both Apple and Sun Microsystems have open-sourced large pieces of strategic software; but both have released them under licenses that significantly restrict usage compared to the licenses in the Linux community. They (Apple and Sun) have argued that their license structure is necessary to their corporate survival and can lead to a more practical market for all involved. Their approach is more of a cross between traditional proprietary development and true open source development. However, successful open source development is a social phenomenon, and even the perception that it is less attractive or unfair may be sufficient to destroy the desired collaboration.

Second, the hypothesis that the quality of open source software is due to the breadth of its review may simply be wrong. The real reason for the quality may be that Darwinian natural selection is eliminating poor-quality packages, the disappointed companies among them. In a corporation a manager can usually justify putting maintenance money into a piece of software the company is selling, even when the piece is known to be of very low quality. It will usually seem easier, and cheaper, to pay for "one more fix" than to start over and rewrite the bad software from scratch, this time correctly. But in the open source community there are no managers who can insist that a programmer maintain a particular piece of code. If the code is badly structured, hard to read, prone to failure, or otherwise unattractive, it will not attract the volunteer labor needed to keep it in the major distributions and will effectively disappear. If nobody works on the code it doesn't get distributed and natural selection has culled it.

For the second case, classification within the types of collaborative system, consider a multiservice integrated battle management system. Military C4I systems are normally thought of as closed collaborative systems. As the levels of integration cross higher and higher administrative boundaries, the ability to centrally control the acquisition and operation of the system lessens. In a multiservice battle management system there is likely to be much weaker central control across service boundaries then within those boundaries. A mechanism that ensures components will collaborate within a single service's system-of-systems, say a set of command operational procedures, may be insufficient across services.

In general, if a collaborative system is misclassified as closed, the builders and operators will have less control over purpose and operation than they may believe. They may use inappropriate mechanisms for insuring collaboration, and may assume cooperative operations across administrative boundaries that will not reliably occur in practice. The designer of a directed system-of-systems can require that an element behave in a fashion not to its own advantage (at least to an extent). In a collaborative system it is unlikely that a component will be induced to behave to its own detriment, but more likely to the detriment of the system as a whole.

A virtual collaborative system misclassified as open may show very unexpected emergent behaviors. In a virtual system neither the purpose nor structure are under direct control, even of a collaborative body. Hence, new purposes and corresponding behaviors may arise at any time. The large-scale distributed applications on the Internet, for example, USENET and the World Wide Web, exhibit this. Both were originally intended for exchange of research information in a collaborative environment, but are now used for diverse purposes, some undesired and even illegal.

Standards and collaborative systems

The development of multicompany standards is a laboratory for collaborative systems. A standard is a framework for establishing some collaborative systems. The standard (e.g., a communication protocol or programming language standard) creates the environment within which independent implementations can coexist and compete.

> Example: Telephone standards allow equipment produced by many companies in many countries to operate together in the global telephone network. A call placed in one country can traverse switches from different manufacturers and media in different countries with nearly the same capabilities as if the call were within a single country on one company's equipment.

> Example: Application programming interface (API) standards allow different implementations of both software infrastructure and applications to coexist. So, operating systems from different vendors can support the same API and allow compliant applications to run on any systems from any of the vendors.

Historically, there has been a well-established process for setting standards. There are recognized national and international bodies with the responsibility to set standards, such as the International Standards Organization (ISO), the American National Standards Institute (ANSI), etc. These bodies have detailed processes that have to be followed. The process defines how standards efforts are approved, how working groups operate, how voting is carried out, and how standards are approved. Most of these processes are rigorously democratic (if significantly bureaucratic). The intention is that a standard should reflect the honest consensus of the concerned community and is thus likely to be adopted.

Since 1985 this established process has been run over, at least within the computer field, by Internet, Web, and open source processes. The IETF, which never votes on a standard, has completely eclipsed the laboriously constructed Open Systems Interconnect (OSI) networking standard. The official standards for operating systems interface have trivial market share compared to the proprietary standards (from Microsoft, Apple, and others) and the open source standards (Linux).

Since the landscape is still evolving, it may be premature to conclude what the new rules are. It may be that we are in a time of transition, and that after the computing market settles down we will return to more traditional methods. It may be that when the computer and network infrastructure is recognized as a central part of the public infrastructure (like electricity and telephones), it will be subjected to similar regulation and will respond with similar bureaucratic actions. Or, it may be that the traditional standards bodies will recognize the principles that have made the Internet efforts so successful and will adapt. Some fusion may prove to be the most valuable yet. In that spirit, we consider what heuristics may be extracted from the Internet experience. These heuristics are important not only to standards efforts, but to collaborative systems as a whole as standards are a special case of a collaborative system.

Economists call something a "network good" if it increases in value the wider it is consumed. For example, telephones are network goods. A telephone that doesn't connect to anybody is not valuable. Two cellular telephone networks that can't interoperate are much less valuable than if they can interoperate. The central observation is that:

Standards are network goods, and must be treated as such.

Standards are network goods because they are useful only to the extent that other people use them. One company's networking standard is of little interest unless other companies support it (unless, perhaps, that company is a monopoly). What this tells standards groups is that achieving large market penetration is critically important. Various practices flow from this realization. The IETF, in contrast to most standards groups, gives its standards away for free. A price of zero encourages wide dissemination. Also, the IETF typically gives away reference implementations with its standards. That is, a proposal rarely becomes a standard unless it has been accompanied by the release of free source code that implements the standard. The free source code may not be the most efficient, it may not be fully featured, it probably does not have all the extras in interface that a commercial product should have, but it is free and it does provide a reference case against which everybody else can work. The IETF culture is that proponents of an approach are rarely given much credibility unless they are distributing implementations.

The traditional standards organizations protest that they can't give standards away because the revenue from standard sales is necessary to support their development efforts. But the IETF has little trouble supporting its efforts. Its working conferences are filled to overflowing and new proposals and working groups are appearing constantly. Standards bodies don't need to make a profit, indeed, they should not. If they can support effective standards development they are successful, though removing the income of standards sales might require substantial organizational change.

Returning to collaborative systems in general, the example of standards shows the importance of focusing on real collaboration, not the image of it. Commitment to real participation in a collaboration is not indicated by voting; it is indicated by taking action that costs. Free distribution of standards and reference implementations lowers entrance costs. The existence of reference implementations provides clear conformance criteria that can be explicitly tested.

Conclusions

Collaborative systems are those which exist only through the positive choices of component operators and managers. These systems have long existed as part of the civil infrastructure of industrial societies, but have come into greater prominence as high-technology communication systems have adopted similar models, centralized systems have been decentralized through deregulation or divestiture, and formerly independent systems have been loosely integrated into larger wholes. What sets these systems apart is their need for voluntary actions on the part of the participants to create and maintain the whole. This requires that the architect revisit known heuristics for greater emphasis and additional elaboration. Among the heuristics that are particularly important are:

1. Stable intermediate forms: A collaborative system designer must pay closer attention to the intermediate steps in a planned evolution. The collaborative system will take on intermediate forms dynamically and without direction as part of its nature.
2. Policy triage: The collaborative system designer will not have coercive control over the system's configuration and evolution. This makes choosing the points at which to influence the design more important.
3. Leverage at the interfaces: A collaborative system is defined by its emergent capabilities, but its architects have influence on its interfaces. The interfaces, whether thought of as the actual physical interconnections or as higher-level service abstractions, are the primary points at which the architect can exert control.
4. Ensuring cooperation: A collaborative system exists because the partially independent elements decide to collaborate. The designer must consider why they will choose to collaborate and foster those reasons in the design.
5. A collaboration is a network good; the more of it there is the better. Minimize entrance costs and provide clear conformance criteria.

Exercises

1. The Internet, multimedia video standards (MPEG), and the GSM digital cellular telephone standard are all collaborative systems. All of them also have identifiable architects, a small group of individuals who carried great responsibility for the basic technical structures. Investigate the history of one of these cases and consider how the practices of the collaborative system architect differ from architects of conventional systems.
2. In a collaborative system the components can all operate on their own whether or not they participate in the overall system. Does this represent a cost penalty to the overall system? Does it matter? Discuss from the perspective of some of the examples.
3. Collaborative systems in computing and communication usually evolve much more rapidly than those controlled by traditional regulatory bodies, and often more rapidly than those controlled by single companies. Is this necessary? Could regulatory bodies and companies adopt different practices that would make their systems as evolvable as collaborative (e.g., Internet or Linux) while retaining the advantages of the traditional patterns of control?

Notes and references

1. IVHS America, Strategic Plan for Intelligent Vehicle-Highway Systems in the United States, IVHS America, Report No. IVHS-AMER-92-3, Intelligent Vehicle-Highway Society of America, Washington, D.C., 1992; and USDOT, National Program Plan for ITS, 1995.
2. Butler, S., Diskin, D., Howes, N., and Jordan, K., The architectural design of the common operating environment for the global command and control system, *IEEE Software*, p. 57, November 1996.
3. Hayes, R. H., Wheelwright, S. C., and Clark, K. B., *Dynamic Manufacturing*, Free Press, a Division of Macmillan, New York, 1988.
4. Maier, M. W., Architecting principles for systems-of-systems, *Systems Engineering*, 2:1, p. 1, 1999.
5. Modeled after Peterson, L., and Davie, B., *Computer Networks: A Systems Approach*, Morgan-Kaufmann, New York, 1996.
6. See the IVHS America and USDOT references above. Also, Maier, M. W., On architecting and intelligent transport systems, Joint Issue *IEEE Transactions on Aerospace and Electronic Systems/System Engineering*, AES33:2, p. 610, April, 1997 by one of the present authors discusses the architectural issues specifically.
7. Chiariglione, L., Impact of MPEG Standards on Multimedia Industry, *IEEE Proc.*, 86:6, p. 1222, June 1998.
8. The Cathedral and the Bazaar by Eric Raymond has become the most famous essay on principles. It is available at http://www.tuxedo.org/~esr/writings/cathedral-bazaar.

Exercise to close part two

Explore another domain much as builder-architected, sociotechnical, manufacturing, software, and collaborative systems are explored in this part. What are the domain's special characteristics? What more broadly applicable lessons can be learned from it? What general heuristics apply to it? Below are some suggested heuristic domains to explore.

1. Telecommunications in its several forms: point-to-point telephone network systems, broadcast systems (terrestrial and space), and packet-switched data (the Internet)
2. Electric power, which is widely distributed with collaborative control, is subject to complex loading phenomena (with a social component), and is regulated (Hill, D. J., Special Issue on Nonlinear Phenomena in Power Systems: Theory and Practical Implications, *Proc. IEEE,* 83, 11, November 1995)
3. Transportation, in both its current form and in the form of proposed intelligent transportation systems
4. Financial systems, including global trading mechanisms, and the operation of regulated economics as a system
5. Space systems, with their special characteristics of remote operation, high initial capital investment, vulnerability to interference and attack, and their effects on the design and operation of existing earthborne systems performing similar functions
6. Existing and emerging media systems, including the collection of competing television systems of private broadcast, public broadcast, cable, satellite, and video recording

Part three

Models and modeling

Introduction to part three

What is the product of an architect? While it is tempting to regard the building or system as the architect's product, the relationship is necessarily indirect. The system is actually built by the developer. The architect acts to translate between the problem domain concepts of the client and the solution domain concepts of the builder. Great architects go beyond the role of intermediary to make a visionary combination of technology and purpose that exceeds the expectation of builder or client. But the system cannot be built as envisioned unless the architect has a mechanism to communicate the vision and track construction against it. The concrete, deliverable products of the architect, therefore, are models of the system.

Individual models alone are point-in-time representations of a system. Architects need to see and treat each as a member of one of several progressions. The architect's first models define the system concept. As the concept is found satisfactory and feasible, the models progress to the detailed, technology-specific models of design engineers. The architect's original models come into play again when the system must be certified.

A civil architecture analogy

Civil architecture provides a familiar example of modeling and progression. An architect is retained to ensure that the building is pleasing to the client in all senses (aesthetically, functionally, and financially). The product of the architect is intangible; it is the conceptual vision which the physical building embodies and which satisfies the client. But the intangible product is worthless without a series of increasingly detailed tangible products — all models of some aspect of the building. Table 1 lists some of the models and their purposes.

The progression of models during the design life cycle can be visualized as a steady reduction of abstraction. Early models may be quite abstract. They may convey only the basic floor plan, associated order-of-magnitude budgets, and renderings encompassing only major aesthetic elements. Early

Table 1 Models and Purposes in Civil Architecture

Model	Purpose
Physical scale model	Convey look and site placement of building to architect, client, and builder
Floor plans	Work with client to ensure building can perform basic functions desired
External renderings	Convey look of building to architect, client, and builder
Budgets, schedules	Ensure building meets client's financial performance objectives, manage builder relationship
Construction blueprints	Communicate design requirements to builder, provide construction acceptance criteria

models may cover many disparate designs representing optional building structures and styles. As decisions are made, the range of options narrows and the models become more specific. Eventually, the models evolve into construction drawings and itemized budgets and pass into the hands of the builders. As the builders work, models are used to control the construction process and to ensure the integrity of the architectural concept. Even when the building is finished some of the models will be retained to assist in future project developments, and to act as an as-built record for building alterations.

Guide to part three

While the form of the models differs greatly from civil architecture to aerospace, computer, or software architectures, their purposes and relationships remain the same. Part Three discusses the concrete elements of architectural practice, the models of systems, and their development. The discussion is from two perspectives broken into four chapters. First, models are treated as the concrete representations of the various views that define a system. This perspective is treated in general in Chapter 8, and through domain-specific examples in Chapter 10. Second, the evolution and development of models is treated as the core of the architecting process. Chapter 9 develops the idea of progressive design as an organizing principle for the architecting process. Community efforts at standardizing architecture representation models, called architecture description frameworks, is the subject of Chapter 11.

Chapter 8, Representation Models and System Architecting, covers the types of models used to represent systems and their roles in architecting. Because architecting is multidimensional and multidisciplinary, an architecture may require many partially independent views. The chapter proposes a set of six views, and reviews major categories of models for each view. It also introduces a viewpoint as an organizing abstraction for writing architecture description standards. Because a coherent and integrated product is the ultimate goal, the models chosen must also be designed to integrate with each other. That is, they must define and resolve their interdependencies and form a complete definition of the system to be constructed.

Chapter 9, Design Progression in Systems Architecting, looks for principles to organize the eclectic architecting process. A particularly useful principle is that of progression — the idea that models, heuristics, evaluation criteria, and many other aspects of the system evolve on parallel tracks from the abstract to the specific and concrete. Progression also helps tie architecting into the more traditional engineering design disciplines. While this book largely treats system architecting as a general process, independent of domain, in practice it necessarily is strongly tied to individual systems and domains. Nevertheless, each domain contains a core of problems not amenable to rational, mechanistic solutions which are closely associated with reconciling customer or client need and with technical capability. This core is the province of architecting. Architects are not generalists; they are system specialists, and their models must refine into the technology-specific models of the domains in which their systems are to be realized.

Chapter 10 returns to models, now tying the previous two chapters together by looking at specific modeling methods. This chapter examines a series of integrating methodologies that illustrate the attributes discussed in the previous chapters: multiple views, integration across views, and progression from abstract to concrete implementation. Examples of integrated models and methods are given for computer-based systems, performance-oriented systems, software-intensive systems, manufacturing systems, and sociotechnical systems. The first part of Chapter 10 describes two general-purpose integrated modeling methods: Hatley/Pirbhai and Quantitative Quality Function Deployment. The former specializes in combining behavioral and physical implementation models, the latter in integrating quantitative performance requirements with behavioral and implementation models. Subsequent sections describe integrated models for software, manufacturing systems, and sociotechnical systems.

Chapter 11 looks outward to the community interested in architecture to review recent work in standardizing architecture descriptions. Standards for architecture description are usually referred to as architecture description frameworks. The chapter reviews three of the leading ones, with some mention of others. They are the U. S. Department of Defense Command Control Communications Computing Intelligence Surveillance and Reconnaissance architecture framework, the ISO Reference Model for Open Distributed Processing, and the IEEE's P1471 Recommended Practice for Architectural Description.

chapter eight

Representation models and system architecting

By relieving the mind of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the [human] race.
Alfred North Whitehead

Introduction: roles, views, and models

Models are the primary means of communication with clients, builders, and users; models are the language of the architect. Models enable, guide, and help assess the construction of systems as they are progressively developed and refined. After the system is built, models, from simulators to operating manuals, help describe and diagnose its operation.

To be able to express system imperatives and objectives and manage system design, the architect should be fluent, or at least conversant, with all the languages spoken in the long process of system development. These languages are those of system specifiers, designers, manufacturers, certifiers, distributors, and users.

The most important models are those which define the critical acceptance requirements of the client and the overall structure of the system. The former are a subset of the entirety of the requirements while the latter are a subset of the complete, detailed system design. Because the architect is responsible for total system feasibility, the critical portions may include highly detailed models of components on which success depends, and abstract, top-level models of other components.

Models can be classified by their roles or by their content. Role is important in relating models to the tasks and responsibilities not only of architects, but of many others in the development process. Of special importance to architects are modeling methods that tie otherwise separate models into a consistent whole.

Roles of models

Models fill many roles in system architecting, including:

1. Communication with client, users, and builders
2. Maintenance of system integrity through coordination of design activities
3. Assisting design by providing templates, and organizing and recording decisions
4. Exploration and manipulation of solution parameters and characteristics; guiding and recording aggregation and decomposition of system functions, components, and objects
5. Performance prediction and identification of critical system elements
6. Providing acceptance criteria for certification for use

These roles are not independent; each relates to the other. But the foremost role is to communicate. The architect discusses the system with the client, the users (if different), the builders, and possibly many other interest groups. Models of the system are the medium of all such communication. After all, the system itself won't come into being for some time to come. The models used for communication become documentation of decisions and designs and, thus, vehicles for maintaining design integrity. Powerful, well-chosen models will assist in decision making by providing an evocative picture of the system in development. They will also allow relevant parameters and characteristics to be manipulated and the results seen in terms relevant to client, user, or builder.

Communication with the client has two goals. First, the architect must determine the client's objectives and constraints. Second, the architect must insure that the system to be built reflects the value judgments of the client where perfect fulfillment of all objectives is impossible. The first goal requires eliciting information on objectives and constraints and casting it into forms useful for system design. The second requires that the client perceive how the system will operate (objectives and constraints) and that the client can have confidence in the progress of design and construction. In both cases models must be clear and understandable to the client, expressible in the client's own terminology. It is desirable that the models also be expressive in the builder's terms, but because client expressiveness must take priority, proper restatement from client to builder language usually falls to the architect.

User communication is similar to client communication. It requires the elicitation of needs and the comparison of possible systems to meet those needs. When the client is the user this process is simplified. When the client and the users are different (as was discussed in Chapter 5 on sociotechnical systems) their needs and constraints may conflict. The architect is in the position to attempt to reconcile these conflicts.

In two-way communication with the builder, the architect seeks to insure that the system will be built as conceived and that system integrity is maintained. In addition, the architect must learn from the builder those technical constraints and opportunities that are crucial in insuring a feasible and satisfactory design. Models that connect the client and the builder are particularly helpful in closing the iterations from builder technical capability to client objectives.

One influence of the choice of a model set is the nature of its associated "language" for describing systems. Given a particular model set and language, it will be easy to describe some types of systems and awkward to describe others, just as natural languages are not equally expressive of all human concepts. The most serious risk in the choice is that of being blind to important alternate perspectives due to long familiarity (and often success) with models, languages, and systems of a particular type.

Models, viewpoints, and views

Chapters 8-10 discuss this book's approach to modeling in systems architecting. Chapter 11 looks outward to the community to review other important approaches and draw contrasts. Unfortunately, there is a lot of variation in the usage of important terms. There are three terms that are important in setting up a modeling framework: model, view, and viewpoint. We use the definitions of model, view, and viewpoint taken from IEEE standards. A *model* is an approximation, representation, or idealization of selected aspects of the structure, behavior, operation, or other characteristics of a real-world process, concept, or system (IEEE 610.12-1990). A *view* is a representation of a system from the perspective of related concerns or issues (IEEE 1471-2000). A *viewpoint* is a template, pattern, or specification for constructing a view (IEEE 1471-2000).

As discussed above, a model is just a representation of something, in our case some aspect of the architecture of a system. The modeling languages of interest have a vocabulary and a grammar. The words are the parts of a model, and the grammar defines how the words can be linked. Beyond that, a modeling language has to have a method of interpretation, and the models produced have to mean something, typically within some domain. For example, in a block diagramming method, the words are the kinds of blocks and lines and the grammar is the allowed patterns by which they can be connected. The method also has to define some correspondence between the blocks, lines, and connections to things in the world. A physical method will have a correspondence to physically identifiable things. A functional diagramming technique has a correspondence to more abstract entities — the functions that the system carries out.

A view is just a collection of models that share the property that they are relevant to the same concerns of a system stakeholder. For example, a functional view collects the models that represent a system function. An objectives view collects the models which define the objectives to be met by building the system. The idea of view is needed because complex systems tend to have complex models and require a higher-level organizing element.

View is inspired by the familiar idea of architectural views. An architect produces elevations, floor plans, and other representations that show the system from a particular perspective. The idea of view here generalizes this when physical structure is no longer primary.

Viewpoint is an abstraction of view across many systems. It is important only in defining standards for architecture description, so we defer its use until later.

Classification of models by view

A view describes a system with respect to some set of attributes or concerns. The set of views chosen to describe a system is variable. A good set of views should be complete (cover all concerns of the architect's stakeholders) and mostly independent (capture different pieces of information). Table 8.1 lists the set of views chosen here as most important to architecting. A system can be "projected" into any view, possibly in several ways. The projection into views, and the collection of models by views, is shown schematically in Figure 8.1. Each system has some behavior (abstracted from implementation), has a physical form, retains data, etc. Views are composed of models. Not all views are equally important to system developmental success, nor will the set be constant over time. For example, a system might be behaviorally complex but have relatively simple form. Views that are critical to the architect may play only a supporting role in full development.

Table 8.1 Major System or Architectural Views

Perspective or view	Description
Purpose/objective	What the client wants
Form	What the system is
Behavioral or functional	What the system does
Performance objectives or requirements	How effectively the system does it
Data	The information retained in the system and its interrelationships
Managerial	The process by which the system is constructed and managed

Although any system can be described in each view, the complexity and value of each view's description can differ considerably. Each class of systems emphasizes particular views and has favored modeling methods, or methods of representation within each view. The architect must determine which views are most important to the system and its environment and be expert in the relevant models. While the views are chosen to be reasonably independent, there is extensive linkage among views. For example, the behavioral aspects of the system are not independent of the system's form. The system can produce the desired behavior only if the system's form

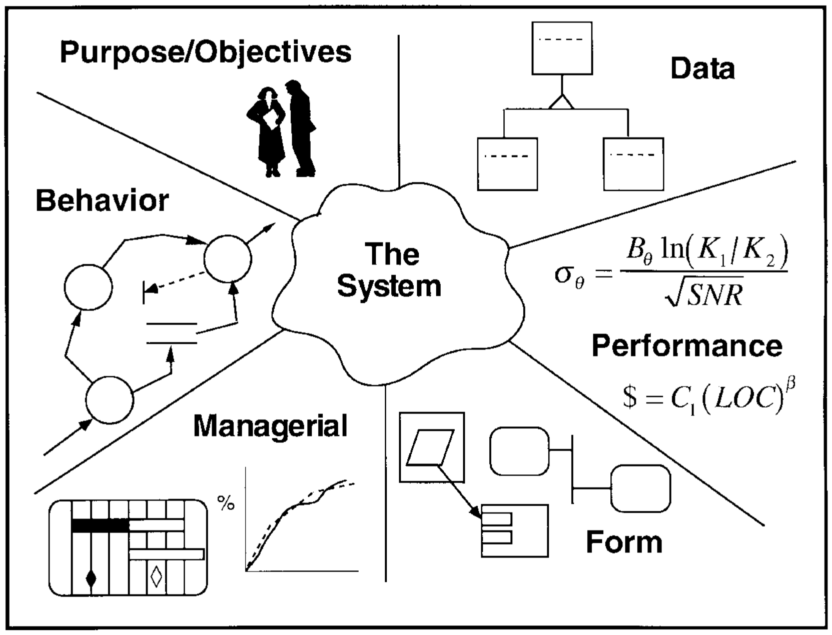


Figure 8.1

supports it. This linkage is conceptually similar to a front and side view being linked (both show vertical height), even though they are observations from orthogonal directions.

The following sections describe models used for representing a system in each of the views of Table 8.1. The section for each view defines what information is captured by the view, describes the modeling issues within that view, and lists some important modeling methods. Part of the architect's role is to determine which views are most critical to system success, build models for those views, and then integrate as necessary to maintain system integrity. The integration across views is a special concern of the architect.

Note to the reader

The sections to follow, which describe models for each view, are difficult to understand without examples meaningful to each reader. Rather than trying to present detailed examples of each for each of several system domains (a task which might require its own book), we suggest the reader do so on his or her own. The examples given in this chapter are not detailed and are chosen to be understandable to the widest possible audience. Chapter 10 describes, in detail, specific modeling methods that span and integrate multiple views. The methods of Chapter 10 are what the architect should strive for, an integrated picture of all important aspects of a system.

As stated in the introduction, Part Three can be read several ways. The chapters can be read in order, which captures the intellectual thread of model concepts, modeling processes and heuristics, specific modeling methods, and organizing frameworks. In this case it is useful to read ahead to Exercises 1 and 2 at the end of this chapter and work them while reading each section to follow. The remaining exercises are intended for after the chapter is read, although some may be approached as each section is completed. An alternative is to read Chapters 8 and 10 in parallel, reading the specific examples of models in Chapter 10 as the views are covered in Chapter 8. Because the approach of Chapter 10 is to look at integrated models, models that span views, a one-for-one correspondence is impossible. The linear approach is probably best for those without extensive background in modeling methods. Those with a good background in integrated modeling methods can use either.

Objectives and purpose models

The first modeling view is that of objectives and purposes. Systems are built for useful purposes; that is for what the client *wants*. Without them the system cannot survive. The architect's first and most basic role is to match the desirability of the purposes with the practical feasibility of a system to fulfill those purposes. Given a clearly identifiable client, the architect's first step is to work with that client to identify the system's objectives and priorities. Some objectives can be stated and measured very precisely. Others will be quite abstract and impossible to express quantitatively. A civil architect is not surprised to hear a client's objective is for the building to "be beautiful" or to "be in harmony with the natural state of the site." The client will be very unhappy if the architect tells the client to come back with unambiguous and testable requirements. The architect must prepare models to help the client to clarify abstract objectives. Abstract objectives require provisional and exploratory models, models which may fall by the wayside later as the demands and the resulting system become well understood. Ideally, all iterations and explorations become part of the systems document set. However, to avoid drowning in a sea of paper, it may be necessary to focus on a limited set. If refinement and tradeoff decisions (the creation of concrete objectives from abstract ones) are architectural drivers, they must be maintained as it is likely the key decisions will be repeatedly revisited.

Modeling therefore begins by restating and iterating those initial unconstrained objectives from the client's language until a modeling language and methodology emerges, the first major step closer to engineering development. Behavioral objectives are restated in a behavioral modeling language. Performance requirements are formulated as measurable satisfaction models. Some objectives may translate directly into physical form, others into patterns of form that should be exhibited by the system. Complex objectives almost invariably require several steps of refinement and, indeed, may evolve into characteristics or behaviors quite different from their original statement.

A low technology example (though only by modern standards) is the European cathedrals of the Middle Ages. A cathedral architect considered a broad range of objectives. First, a cathedral must fulfill well-defined community needs. It must accommodate celebration day crowds, serve as a suitable seat for the bishop, and operate as a community centerpiece. But, in addition, cathedral clients of that era emphasized that the cathedral "communicate the glory of God and reinforce the faithful through its very presence."

Accommodation of holiday celebrants is a matter of size and floor layout. It is an objective that can be implemented directly and requires no further significant refinement. The clients — the church and community leaders — because of their personal familiarity with the functioning of a cathedral, could determine for themselves the compliance of the cathedral by examining the floor plan. But what of defining a building that "glorifies God?" This is obviously a property only of the structure as a whole — its scale, mass, space, light, and integration of decoration and detail. Only a person with an exceptional visual imagination is able to accurately envision the aesthetic and religious impact of a large structure documented only through drawings and renderings. Especially in those times, when architectural styles were new and people traveled little, an innovative style would be an unprecedented experience for perhaps all but the architect.

While refinement of objectives through models is central to architecting, it is also a source of difficulty. A design that proceeds divorced from direct client relevance tends to introduce unnecessary requirements that complicate its implementation. Experience has shown that retaining the client's language throughout the acquisition process can lead to highly efficient, domain-specific architectures; for example, in communication systems.

> Example: domain-specific software architectures[1] are software application generation frameworks in which domain concepts are embedded in the architectural components. The framework is used to generate a product line of related applications in which the client language can be used nearly directly in creating the product. For a set of message handler applications within command and control systems the specification complexity was reduced 50:1.

One measure of the power of a design and implementation method is its ability to retain the original language. But this poses a dilemma. Retention implies the availability of proven, domain-specific methods and engineering tools. But unprecedented systems by definition are likely to be working in new domains, or near the technical frontiers of existing domains. By the very nature of unprecedented system development, such methods and tools are unlikely to be available. Consequently, models and methodologies must be developed and pass through many stages of abstraction, during which the original relevance can easily be lost. The architect must therefore search out domain-specific languages and methods that can somehow maintain the chain of relevance throughout.

An especially powerful, but challenging, form of modeling converts the client/user's objectives into a meta-model or metaphor that can be directly implemented. A famous example is the desktop metaphor adopted for Macintosh computers. The user's objective is to use a computer for daily, office-oriented task automation. The solution is to capture the user's objectives directly by presenting a simulation of a desktop on the computer display. Integrity with user needs is automatically maintained by maintaining the fidelity of a desktop and file system familiar to the user.

Models of form

Models of form represent physically identifiable elements of, and interfaces to, what will be constructed and integrated to meet client objectives. Models of form are closely tied to particular construction technologies, whether the concrete and steel of civil architecture or the less tangible codes and manuals of software systems. Even less tangible physical forms are possible, such as communication protocol standards, a body of laws, or a consistent set of policies.

Models of form vary widely in their degree of abstraction and role. For example, an abstract model may convey no more than the aesthetic feel of the system to the client. A dimensionally accurate but hollow model can assure proper interfacing of mechanical parts. Other models of form may be tightly coupled to performance modeling, as in the scale model of an airplane subjected to wind tunnel testing. The two categories of models of form most useful in architecting are scale models and block diagrams.

Scale models

The most literal models of form are scale models. Scale models are widely used for client and builder communication, and may function as part of behavioral or performance modeling as well. Some major examples are shown below:

1. Civil architects build literal models of buildings, often producing renderings of considerable artistic quality. These models can be abstracted to convey the feel and style of a building, or can be precisely detailed to assist in construction planning.
2. Automobile makers mock-up cars in body-only or full running trim. These models make the auto show circuit to gauge market interest, or are used in engineering evaluations.
3. Naval architects model racing yachts to assist in performance evaluation. Scale models are drag tested in water tanks to evaluate drag and handling characteristics. Reduced or full-scale models of the deck layout are used to run simulated sail handling drills.

4. Spacecraft manufacturers use dimensionally accurate models in fit compatibility tests and in crew extra-vehicular activity rehearsals. Even more important are ground simulators for on-orbit diagnostics and recovery planning.
5. Software developers use prototypes that demonstrate limited characteristics of a product that are equivalent to scale models. For example, user interface prototypes that look like the planned system but do not possess full functionality, non-real-time simulations that carry extensive functionality but do not run in real-time, or just a set of screen shots with scenarios for application use.

Physical scale models are gradually being augmented or replaced by virtual reality systems. These "scale" models exist only in a computer and in the viewer's mind. They may, however, carry an even stronger impression of reality than a physical scale model because of the sensory immersion achievable.

Block diagrams

A scale model of a circuit board or a silicon chip is unlikely to be of much interest alone, except for expanded scale plots used to check for layout errors. Nonetheless, physical block diagrams are ubiquitous in the electronics industry. To be a model of form, as distinct from a behavioral model, the elements of the block diagram must correspond to physically identifiable elements of the system. Some common types of block diagrams include:

1. System interconnect diagrams that show specific physical elements (modules) connected by physically identifiable channels. On a high-level diagram a module might be an entire computer complex, and a channel might be a complex internetwork. On a low level the modules could be silicon chips with specific part numbers and the channels' pin-assigned wires.
2. System flow diagrams that show modules in the same fashion as interconnect diagrams but illustrate the flow of information among modules. The abstraction level of information flow defined might be high (complex messaging protocols) or low (bits and bytes). The two types of diagrams (interconnect and flow) are contrasted in Figure 10.3.
3. Structure charts,[2] task diagrams,[3] and class and object diagrams[4] that structurally define software systems and map directly to implementation. A software system may have several such logically independent diagrams, each showing a different aspect of the physical structure; for example, diagrams that show the invocation tree, the inheritance hierarchy, or the "withing" relationships in an Ada program. Examples of several levels of physical software diagram are given in Figures 10.5 and 10.6.

5. Manufacturing process diagrams are drawn with a standardized set of symbols. These represent manufacturing systems at an intermediate level of abstraction, showing specific classes of operation but not defining the machine or the operational details.

Several authors have investigated formalizing block diagrams over a flexible range of architectural levels. The most complete, with widely published examples, is that of Hatley and Pirbhai.[5] Their method is discussed in more depth in Chapter 10 as an example of a method for integrating a multiplicity of architectural views across models. A number of other methods and tools that add formalized physical modeling to behavioral modeling are appearing. Many of these are commercial tools, so the situation is fluid and their methodologies are often not fully defined outside of the tools documentation. Some other examples are the system engineering extensions to ADARTS (described later in the context of software), RDD-100,[6] and StateMate.[7]

An attribute missing in most block diagram methods is the logic of data flow. The diagram may show that a data item flow from module A to module B, but does not show who controls the flow. Control can be of many types. A partial enumeration includes:

Soft push: The sender sends and the item is lost if the receiver is not waiting to receive it

Hard push: The sender sends and the act of sending interrupts the receiver who must take the data

Blocking pull: The receiver requests the data and waits until the sender responds

Non-blocking pull: The receiver requests the data and continues on without it if the sender does not send

Hard pull: When the receiver requests the data the sender is interrupted and must send

Queuing channel: The sender can push data onto the channel without interrupting the receiver and with data being stored in the channel; the receiver can pull data from the channel's store

Of course, there are many other combinations as well. The significance of the control attribute is primarily in interfacing to disciplinary engineers, especially software engineers. In systems whose development cost is dominated by software, which is now virtually all complex systems, it is essential that systems activities provide the information needed to enable software architecting as quickly as possible. One of the elements of a software architecture is the concurrency and synchronization model. The constraints on software concurrency and synchronization are determined by the data flow control logic around the software-hardware boundary. So, it is just the kind of information on data flow control that is needed to better match systems activities to software architecture.

Behavioral (functional) models

Functional or behavioral models describe specific patterns of behavior by the system. These are models of what the system *does* (how it behaves) as opposed to what the system *is* (which are models of form). Architects increasingly need behavioral models as systems become more intelligent and their behavior becomes less obvious from the systems form. Unlike a building, a client cannot look at a scale model of a software system and infer how the system behaves. Only by explicitly modeling the behavior can it be understood by the client and builder.

Determining the level of detail or rigor in behavioral specification needed during architecting is an important choice. Too little detail or rigor will mean the client may not understand the behavior being provided (and possibly be unsatisfied) or the builder may misunderstand the behavior actually required. Too much detail or rigor may render the specification incomprehensible — leading to similar problems — or unnecessarily delay development. Eventually, when the system is built, its behavior is precisely specified (if only by the actual behavior of the built system).

From the perspective of architecting, what level of behavioral refinement is needed? The best guidance is to focus on the system acceptance requirements; to ensure the acceptance requirements are passable but complete. Ask what behavioral attributes of the system the client will demand be certified before acceptance and determine through what tests those behavioral attributes can be certified. The certifiable behavior is the behavior the client will get, no more and no less.

> Example: In software systems with extensive user interface components, it has been found by experience that only a prototype of the interface adequately conveys to users how the system will work. Hence, to ensure not just client acceptance, but user satisfaction, an interface prototype should be developed very early in the process. Major office application developers have videotaped office workers as they use prototype applications. The tapes are then examined and scored to determine how effective various layouts were at encouraging users to make use of new features, how rapidly they were able to work, etc.
>
> Example: Hardware and software upgrades to military avionics almost always must remain backward compatible with other existing avionics systems and maintain support for existing weapon systems. The architecture of the upgrade must reflect the behavioral requirements of existing system interface. Some may imply very simple behavioral requirements, like pro-

> viding particular types of information on a communication bus. Others may demand complex behaviors, such as target handover to a weapon requiring target acquisition, queuing of the weapon sensor, real-time synchronization of the local and weapon sensor feeds, and complex launch codes. The required behavior needs to be captured at the level required for client acceptance, and at the level needed to extract architectural constraints.

Behavioral tools of particular importance are threads or scenarios, data and event flow networks, mathematical systems theory, autonomous system theory, and public choice and behavior models.

Threads and scenarios

A thread or scenario is a sequence of system operations. It is an ordered list of events and actions which represents an important behavior. It normally does not contain branches; that is, it is a single serial scenario of operation, a stimulus/response thread. Branches are represented by additional threads. Behavioral requirements can be of two types. The first type is to require that the system *must* produce a given thread; that is, to require a particular system behavior. The alternative is to require that a particular thread not occur. For example, that a hazardous command never be issued without a positive confirmation having occurred first. The former is more common, but the latter is just as important.

Threads are useful for client communication. Building the threads can be a framework for an interactive dialog with the client. For each input, pose the question, "When this input arrives what should happen?" Trace the response until an output is produced. In a similar fashion, trace outputs backward until inputs are reached. The list of threads generated in this way becomes part of the behavioral requirements.

Threads are also useful for builder communication. Even if not complete, they directly convey desired system behavior. They also provide useful tools during design reviews and for test planning. Reviewers can ask that designers walk through their design as it would operate in each of a set of selected threads. This provides a way for reviewers to survey a design using criteria very close to the client's own language. Threads can be used similarly as templates for system tests, ensuring that the tests are directly tied to the client's original dialog.

Another name for behavioral specification by threads and scenarios is use-cases. Use-case has become the popular term for behavioral specification by example. The term originally comes from the object-oriented software community, but it has been applied much more widely. The normal form of a use-case is the listing of an example dialog between the system and an actor. An actor is a human user of the system. The use-case consists of the sequence of messages passed between the system and actor, augmented by additional explanation in ways specific to each method. Use-cases are intended to be narrative. That is, they are specifically intended to be written in the language of users and to be understandable by them. When a system is specified by many use-cases, and the use-cases interact, there are a number of diagrams which can be used to specify the connections. Chapter 10 briefly discusses these within the section on UML.

Data and event flow networks

A complex system can possess an enormous (perhaps infinite) set of threads. A comprehensive list may be impossible, yet without it the behavioral specification is incomplete. Data and event flow networks allow threads to be collapsed into more compact but complete models. Data flow models define the behavior of a system by a network of functions or processes that exchange data objects. The process network is usually defined in a graphical hierarchy, and most modern versions add some component of finite state machine description. Current data flow notations are descendants either of DeMarco's data flow diagram (DFD) notation[8] or Functional Flow Block Diagrams (FFBD).[9] Chapter 10 gives several examples of a data flow models and their relationships with other model types. Figures 10.1 and 10.2 show examples of data flow diagrams for an imaging system. Both the DFD and FFBD methods are based on a set of root principles.

1. The systems functions are decomposed hierarchically. Each function is composed of a network of subfunctions until a "simple" description can be written in text.
2. The decomposition hierarchy is defined graphically.
3. Data elements are decomposed hierarchically and are separately defined in an associated "data dictionary."
4. Functions are assumed to be data-triggered. A process is assumed to execute any time its input data elements are available. Finer control is defined by a finite state control model (DFD formalism) or in the graphical structure of the decomposition (FFBD formalism).
5. The model structure avoids redundant definition. Coupled with graphical structuring, this makes the model much easier to modify.

Mathematical systems theory

The traditional meaning of system theory is the behavioral theory of multidimensional feedback systems. Linear control theory is an example of system theory on a limited, well-defined scale. Models of macroeconomic systems and operations research are also system theoretic models, but on a much larger scale.

System theoretic formalisms are built from two components:

1. A definition of the system boundary in terms of observable quantities, some of which may be subject to user or designer control

2. A mathematical machinery that describes the time evolution (the behavior) of the boundary quantities given some initial or boundary conditions and control strategies

There are three main mathematical system formalisms. They are distinguished by how they treat time and data values.

1. Continuous systems: These are the systems classically modeled by differential equations, linear and nonlinear. Values are continuous quantities and are computable for all times.
2. Temporally discrete (sampled data) systems: These are systems with continuously valued elements measured at discrete time points. Their behavior is described by difference equations. Sampled data systems are increasingly important since they are the basis of most computer simulations and nearly all real-time digital signal processing.
3. Discrete event systems: These are systems in which some or all of the quantities take on discrete values at arbitrary points in time. Queuing networks are the classical example. Asynchronous digital logic is a pure example of a discrete event system. The quantities of interest (say, data packets in a communication network) move around the network in discrete units, but they may arrive or leave a node at an arbitrary, continuous time.

Continuous systems have a large and powerful body of theory. Linear systems have comprehensive analytical and numerical solution methods, and an extensive theory of estimation and control. Nonlinear systems are still incompletely understood, but many numerical techniques are available, some analytical stability methods are known, and practical control approaches are available. Similar results are available for sampled data systems. Computational frameworks exist for discrete event systems (based on state machines and Petri Nets), but are less complete than those for differential or difference equation systems in their ability to determine stability and synthesize control laws. A variety of simulation tools is available for all three types of systems. Some tools attempt to integrate all three types into a single framework, though this is difficult.

Many modern systems are a mixture of all three types. For example, consider a computer-based temperature controller for a chemical process. The complete system may include continuous plant dynamics, a sampled data system for control under normal conditions, and discrete event controller behavior associated with threshold crossings and mode changes. A comprehensive and practical modern system theory should answer the classic questions about such a mixed system — stability, closed loop dynamics, and control law synthesis. No such comprehensive theory exists, but constructing one is an objective of current research. Manufacturing systems are a special example of large-scale mixed systems for which qualitative system understanding can yield architectural guidance.

Autonomous agent, chaotic systems

System-level behavior, as defined in Chapter 1, is behavior not contained in any system component, but which emerges only from the interaction of all the components. A class of system of recent interest is that in which a few types of multiply-replicated, individually relatively simple, components interact to create essentially new (emergent) behaviors. Ant colonies, for example, exhibit complex and highly organized behaviors that emerge from the interaction of behaviorally simple, nearly identical, sets of components (the ants). The behavioral programming of each individual ant, and its chaotic local interactions with other ants and the environment, is sufficient for complex high-level behaviors to emerge from the colony as a whole. There is considerable interest in using this truly distributed architecture, but traditional top-down, decomposition-oriented models and their bottom-up, integration-oriented complements do not describe it. Some attempts have been made to build theories of such systems from chaos methods. Attempts have also been made to find rules or heuristics for the local intelligence and interfaces necessary for high-level behaviors to emerge.

> Example: In some prototype flexible manufacturing plants, instead of trying to solve the very complex work scheduling problem, autonomous controllers schedule through distributed interaction. Each work cell independently "bids" for jobs on its input. Each job moving down the line tries to "buy" the production and transfer services it needs to be completed.[10] Instead of central scheduling, the equilibrium of the pseudo-economic bid system distributes jobs and fills work cells. Experiments have shown that rules can be designed that result in stable operation, near optimality of assignment, and very strong robustness to job changes and work cell failure. But the lack of central direction makes is difficult to assure particular operational aspects; for example, to assure that "oddball" jobs won't be ignored for the apparent good of the mean.

Public choice and behavior models

Some systems depend on the behavior of human society as part of the system. In such cases the methods of public choice and consumer analysis may need to be invoked to understand the human system. These methods are often *ad hoc*, but many have been widely used in marketing analysis by consumer product companies.

> Example: One concept in intelligent transportation systems proposals is the use of centralized routing. In a

central routing system each driver would inform the center (via some data network) of his beginning location and his planned destination for each trip. The center would use that information to compute a route for each vehicle and communicate the selected route back to the driver. The route might be dynamically updated in response to accidents or other incidents. In principle, the routing center could adjust routes to optimize the performance of the network as a whole. But would drivers accept centrally-selected routes, especially if they thought the route benefited the network but not them? Would they even bother to send in route information?

A variety of methods could be used to address such questions. At the simplest level are consumer surveys and focus groups. A more involved approach is to organize multiperson driving simulations with the performance of the network determined from individual driver decisions. Over the course of many simulations, as drivers evaluate their own strategies, stable configurations may emerge.

Performance models

A performance model describes or predicts how effectively an architecture satisfies some function. Performance models are usually quantitative, and the most interesting performance models are those of system-level functions; that is, properties possessed by the system as a whole but by no subsystem. Performance models describe properties like overall sensitivity, accuracy, latency, adaptation time, weight, cost, reliability, and many others. Performance requirements are often called "nonfunctional" requirements because they do not define a functional thread of operation, at least not explicitly. Cost, for example, is not a system *behavior*, but it is an important property of the system. Detection sensitivity to a particular signal, however, does carry with it implied functionality. Obviously, a signal cannot be detected unless the processing is in place to produce a detection. It will also usually be impossible to formulate a quantitative performance model without constraining the systems behavior and form.

Performance models come from the full range of engineering and management disciplines. But the internal structure of performance models generally falls into one of three categories:

1. Analytical: These models are the products of the engineering sciences. A performance model in this category is a set of lower-level system parameters and a mathematical rule of combination that predicts the performance parameter of interest from lower-level values. The model is normally accompanied by a "performance budget" or a set of nom-

inal values for the lower-level parameters to meet a required performance target.

2. Simulation: When the lower-level parameters can be identified, but an easily computable performance prediction cannot, a simulation can take the place of the mathematical rule of combination. In essence, a simulation of a system is an analytical model of the system's behavior and performance in terms of the simulation parameters. The connection is just more complex and difficult to explicitly identify. A wide variety of continuous, discrete time and discrete event simulators are available, many with rich sets of constructs for particular domains.
3. Judgmental: Where analysis and simulation are inadequate or infeasible, human judgment may still yield reliable performance indicators. In particular, human judgment, using explicit or implicit design heuristics, can often rate one architecture as better than another, even where a detailed analytical justification is impossible.

Formal methods

The software engineering community has taken a specialized approach to performance modeling known as formal methods. Formal methods seek to develop systems that provably produce formally defined functional and nonfunctional properties. In formal development the team defines system behavior as sets of allowed and disallowed sequences of operation, and may add further constraints, such as timing, to those sequences. They then develop the system in a manner that guarantees compliance to the behavioral and performance definition. Roughly speaking, the formal methods approach is:

1. Identify the inputs and outputs of the system. Identify a set of mathematical and logical relations that must exist between the input and output sequences when the system is operating as desired.
2. Decompose the system into components, identifying the inputs and outputs of each component. Determine mathematical relations on each component such that their composition is equivalent to the original set of relations one level up.
3. Continue the process iteratively to the level of primitive implementation elements. In software this would be programming language statements. In digital logic this might be low-level combinational or sequential logic elements.
4. Compose the implementation backward, up the chain of inference from primitive elements in a way that conserves the decomposed correctness relations. The resulting implementation is then equivalent to the original specification.

From the point of view of the architect, the most important applications of formal methods are in the conceptual phases and in the certification of high assurance and ultraquality systems. Formal methods require explicit determination of allowed and disallowed input/output sequences. Trying to make that determination can be valuable in eliciting client information, even if the resulting information is not captured in precise mathematical terms. Formal methods also hold out the promise of being able to certify system characteristics that can never be tested. No set of tests can certify with certainty that certain event chains cannot occur, but theorems to that effect are provable within a formal model.

Various formal and semiformal versions of the process are in limited use in software and digital system engineering.[11] While a fully formal version of this process is apparently impractical for large systems at the present time (and is definitely controversial), semiformal versions of the process have been successfully applied to commercial products.

A fundamental problem with the formal methods approach is that the system can never be more "correct" than the original specification. Because the specification must be written in highly mathematical terms, it is particularly difficult to use in communication with the typical client.

Data models

The next dimension of system complexity is retained data. What data does the system retain and what relationships among the data does it develop and maintain? Many large corporate and governmental information systems have most of their complexity in their data and its internal relationships. The most common data models have their origins in software development, especially large database developments. Methods for modeling complex data relationships were developed in response to the need to automate data-intensive, paper-based systems. While data-intensive systems are most often thought of as large, automated database systems, many working examples are actually paper-based. Automating legacy paper-based systems requires capturing the complex interrelationships among large amounts of retained data.

Data models are of increasing importance because of the greater intelligence being embedded in virtually all systems, and the continuing automation of legacy systems. In data-intensive systems, generating intelligent behavior is primarily a matter of finding relationships and imposing persistent structure on the records. This implies that the need to find structure and relationships in large collections of data will be determinants of system architecture.

> Example: Manufacturing software systems are no longer responsible just for control of work cells. They are part of integrated enterprise information networks in which real-time data from the manufacturing floor, sales, customer operations, and other parts of the enterprise are stored and studied. Substantial competitive advantages accrue to those who can make intelligent judgments from these enormous data sets.

> Example: Intelligent transport systems are a complex combination of distributed control systems, sensor networks, and data fusion. Early deployment stages will emphasize only simple behavioral adaptation, as in local intelligent light and on-ramp controllers. Full deployment will fuse data sources across metropolitan areas to generate intelligent prediction and control strategies. These later stages will be driven by problems of extracting and using complex relationships in very large databases.

The basis for modern data models are the Entity-Relationship diagrams developed for relational databases. These diagrams have been generalized into a family of object-oriented modeling techniques. An object is a set of "attributes" or data elements and a set of "methods" or functions which act upon the attributes (and possibly other data or objects as well). Objects are instances of classes that can be thought of as templates for specific objects. Objects and classes can have relationships of several types. Major relationship types include aggregation (or composition); generalization, specialization, or inheritance; and association (which may be two-way or M-way). Object-oriented modeling methods combine data and behavioral modeling into a single hierarchy organized along and driven by data concerns. Behavioral methods like those described earlier also include data definitions, but the hierarchy is driven by functional decomposition.

One might think of object-oriented models as turning functional decomposition models inside out. Functional decomposition models like data flow diagramming describe the system as a hierarchy of functions, and hang a data model onto the functional skeleton. The only data relationship supported is aggregation. An object-oriented model starts with a decomposition of the data and hangs a functional model on it. It allows all types of data relationships. Some problems decompose cleanly with functional methods and only with difficulty in object-oriented methods, and some other problems are the opposite.

An example of a well-developed, object-oriented data modeling technique (OMT) is given in Chapter 10. Figure 10.7 shows a typical example of the type of diagram used in that method, which combines conventional entity relationship diagram and object-oriented abstraction. OMT has further evolved into unified modeling language (UML), which is also discussed in Chapter 10.

Data-oriented decompositions share the general heuristics of system architecture. The behavioral and physical structuring characteristics have direct analogs — composing or aggregation, decomposition, minimal communications, etc. There are also similar problems of scale. Very large data models must be highly structured with limited patterns of relationship (analogous to limited interfaces) to be implementable.

Managerial models

To both the client and architect, a project may be as much a matter of planning milestones, budgets, and schedules as it is a technical exercise. In sociotechnical systems, planning the system deployment may be more difficult than assembling its hardware. The managerial or implementation view describes the process of building the physical system. It also tracks construction events as they occur.

Most of the models of this view are the familiar tools of project management. In addition, management-related metrics that can be calculated from other models are invaluable in efforts to create an integrated set of models. Some examples include:

1. The waterfall and spiral system development meta-models; they are the templates on which project-specific plans are built
2. PERT/CPM and related task and scheduling dependency charts
3. Cost and progress accounting methods
4. Predictive cost and schedule metrics calculable from physical and behavioral models
5. Design or specification time quality metrics — defect counts, post-simulation design changes, rate of design changes after each review.

The architect has two primary interests in managerial models. First, the client usually cannot decide to go ahead with system construction without solid cost and schedule estimates. Usually, producing such estimates requires a significant effort in management models. Second, the architect may be called upon to monitor the system as it is developed to ensure its conceptual integrity. In this monitoring process managerial models will be very important.

Examples of integrated models

As noted earlier, models which integrate multiple views are the special concern of the architect. These integrating models provide the synthesized view central to the architect's concerns. An integrated modeling method is a system of representation that links multiple views. The method consists of a set of models for a subset of views, and a set of rules or additional models to link the core views. Most integrated modeling methods apply to a particular domain. Table 8.2 lists some representative methods. These models are described in greater detail, with examples, in Chapter 10. The references are given there as well.

These methods use different models and cover different views. Their components and dimensions are summarized in Table 8.3.

Table 8.2 Integrated Modeling Methods and Their Domains

Method	Ref.	Domain
Hatley/Pirbhai (H/P)	Hatley, 1988	Computer-based reactive or event-driven systems
Quantitative quality function deployment (Q^2FD)	Maier, 1995	Systems with extensive quantitative performance objectives and understood performance models
Object modeling technique (OMT)	Rumbaugh, 1991	Large-scale, data-intensive software systems, especially those implemented in modern object languages
ADARTS	SPC, 1991	Large-scale, real-time software systems
Manufacturing system analysis (MSA)	Baudin, 1990	Intelligent manufacturing systems

Table 8.3 Comparison of Popular Modeling Methods

View	H/P	OMT	ADARTS	Q^2FD	MSA
Objectives	Text	Text	Text	Numbers	Text
Behavior	Data/control flow	Class diagrams, data flow, state charts	Data/event flow	Links only	Data flow
Performance	Text (timing only)	Text	Text	Satisfaction models, QFD matrices	Text, links to standard scheduling models
Data	Dictionary	Class/object diagrams	Dictionary	N/A	Entity-relationship diagrams
Form	Formalized block diagrams	Object diagrams	Task-object-structure charts (multilevel)	Links by allocation	ASME process flow diagrams
Managerial	N/A (link via metrics)	N/A	N/A (link via metrics)	N/A	Funds flow model, scheduling behavior

Summary

An architect's work revolves around models. Since the architect does not build the system directly, its integrity during construction must be maintained through models acting as surrogates. Models will represent and control the specification of the system, its design, and its production plan. Even after the system is delivered, modeling will be the mechanism for assessing system behavior and planning its evolution. Because the architect's concerns are broad, architecting models must encompass all views of the system. The architect's knowledge of models, like an individual's knowledge of language, will tend to channel the directions in which the system develops and evolves.

Modeling for architects is driven by three key characteristics:

1. Models are the principal language of the architect. Their foremost role is to facilitate communication with client and builder. By facilitating communication they carry out their other roles of maintaining design integrity and assisting synthesis.
2. Architects require a multiplicity of views and models. The basic ones are objective, form, behavior, performance, data, and management. Architects need to be aware of the range of models that are used to describe each of these views within their domain of expertise, and the content of other views that may become important in the future.
3. Multidisciplinary, integrated modeling methods tie together the various views. They allow the design of a system to be refined in steps from conceptually abstract to the precisely detailed necessary for construction.

The next chapter reconsiders the use of modeling in architecture by placing modeling in a larger set of parallel progressions from abstract to concrete. There, the field of view will expand to the whole architectural design process and its parallel progressions in heuristics, modeling, evaluation, and management.

Exercises

1. Choose a system familiar to you. Formulate a model of your system in each of the views discussed in the chapter. How effectively does each model capture the system in that view. How effectively do the models define the system for the needs of initial concept definition and communication with clients and builders? Are the models integrated? That is, can you trace information across the models and views?
2. Repeat Exercise 1, but with a system unfamiliar to you, and preferably embodying different driving issues. Investigate models used for the views most unfamiliar to you. In retrospect, does your system in Exercise 1 contain substantial complexity in the views you are unfamiliar with?
3. Investigate one or more popular computer-aided system or software engineering (CASE) tools. To what extent do they support each of the views? To what extent do they allow integration across views?
4. A major distinction in behavioral modeling methods and tools is the extent to which they support or demand executability in their models. Executability demands a restricted syntax and up-front decision about data and execution semantics. Do these restrictions and demands help or hinder initial concept formulation and communication with builders and clients? If the answer is variable with the system, is there a way to combine the best aspects of both approaches?

5. Models of form must be technology-specific because they represent actual systems. Investigate modeling formalisms for domains not covered in the chapter. For example, telecommunication systems, business information networks, space systems, integrated weapon systems, chemical processing systems, or financial systems.

Notes and references

1. Balzer, B. and Wile, D., Domain-Specific Software Architectures, Technical Reports, Information Sciences Institute, University of Southern California.
2. Yourdon, E. and Constantine, L L., *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design,* Yourdon Press, Englewood Cliffs, NJ, 1979.
3. *ADARTS Guidebook*, SPC-94040-CMC, Version 2.00.13, Vols. 1-2, September, 1991. Available through the Software Productivity Consortium, Herndon, VA.
4. Rumbaugh, J. et. al., *Object-Oriented Modeling and Design,* Prentice-Hall, Englewood Cliffs, NJ, 1991.
5. Hatley, D. J. and Pirbhai, I., *Strategies for Real-Time System Specification,* Dorset House, New York, 1988.
6. A comprehensive system modeling tool marketed by Ascent Logic, Inc.
7. A tool with both discrete event behavioral modeling and physical block diagrams marketed by i-Logix.
8. DeMarco, T., *Structured Analysis and System Specification,* Yourdon Press, Englewood Cliffs, NJ, 1979.
9. Functional Flow Diagrams, AFSCP 375-5 MIL-STD-499, USAF, DI-S-3604/S-126-1, Form DD 1664, June 1968. Much more modern implementations exist, for example, the RDD-100 modeling and simulation tool developed and marketed by Ascent Logic Corp.
10. Morley, R. E., The chicken brain approach to agile manufacturing, *Proc. Manufacturing, Engineering, Design, Automation Workshop,* Stanford, Palo Alto, p. 19, 1992.
11. Two references can be noted, for theory: Hoare, C. A. R., *Communicating Sequential Processes,* Prentice-Hall, Englewood Cliffs, NJ, 1985. For application in software: Mills, H. D., Stepwise refinement and verification in box-structured systems, *IEEE Computer,* p. 23, June 1988.

chapter nine

Design progression in system architecting

Introduction: architecting process components

Having outlined the products of architecting (models) in Chapter 8, this chapter turns to its process. The goal is not a formal process definition. Systems are too diverse to allow a fixed or dogmatic approach to architecting. Instead of trying for a formal process definition, this chapter develops a set of meta-process concepts for architecting activities and their relationships. Architectural design processes are inherently eclectic and wide-ranging, going abruptly from the intensely creative and individualistic to the more prescribed and routine. While the processes may be eclectic, they can be organized. Of the various organizing concepts, one of the most useful is stepwise progression or "refinement."

First, a brief review of the architecting process itself. The architect develops system models that span the range of system concerns, from objectives to implementation. The architectural approach is, from beginning to end, concerned with the feasibility as well as the desirability of the system implementation. It strives for fit, balance, and compromise between client preferences and builder capabilities. Compromise can only be assured by an interplay of activities, including both high-level structuring and such detailed design as is critical to overall success.

This chapter presents a three-part approach to the process of system architecting.

1. One approach is a conceptual model connecting the unstructured processes of architecture to the rigorous engineering processes of the specialty domains or disciplines. This model is based on stepwise reduction of abstraction (or progression) in models, evaluation criteria, heuristics, and purposes from initial architecting to formal systems engineering.

2. There is an introduction to and review of the general concepts of design, including theories of design, the elements of design, and the processes of creating a design. These frame the activities that make up the progressions.
3. There is a guide to the place of architecting and its methods with the specialized design domains and the evolutionary development of domain-specific methods. Architecting is recursive within a system as it is defined in terms of its implementation domains. A split between architecting and engineering is an irreducible characteristic of every domain, though the boundaries of that split cannot be clear until the scientific basis for the methods in a domain are known.

The progressions of architecting are inextricably bound with the progressions of all system development. Architecting is not only iterative, it can be recursive. As a system progresses, architecting may recur on subsystems. The goal here is to understand the intellectual nature of its conduct, whether it happens at a very high level or within a subsystem.

Design progression

Progressive refinement of design is one of the most basic patterns of engineering practice. It permeates the process of architecting from models to heuristics, information acquisition, and management. Its real power, especially in systems architecting, is that it provides a way to organize the progressive transition from the ill-structured, chaotic, and heuristic processes needed at the beginning to the rigorous engineering and certification processes needed later. All can be envisioned as a stepwise reduction of abstraction from mental concept to delivered physical system.

In software the process is known as "stepwise refinement." Stepwise refinement is a specific strategy for top-down program development. The same notion applies to architecting, but is applied to complex multidisciplinary system development. Stepwise refinement is the progressive removal of abstraction in models, evaluation criteria, and goals. It is accompanied by an increase in the specificity and volume of information recorded about the system, and a flow of work from general to specialized design disciplines. Within the design disciplines the pattern repeats as disciplinary objectives and requirements are converted into the models of form of that discipline. In practice, the process is neither so smooth nor continuous. It is better characterized as episodic, with episodes of abstraction reduction alternating with episodes of reflection and purpose expansion.

Stepwise refinement can be thought of as a meta-process model, much as the waterfall and spiral. It is not an enactable process for a specific project, but it is a model for building a project-specific process. Systems are too diverse to follow a fixed process or dogmatic formula for architecting.

Introduction by examples

Before treating the conceptually difficult process of general systems architecting, look to the roots. When a civil architect develops a building, does he or she go directly from client words to construction drawings? Obviously not. There are many intermediate steps. The first drawings are rough floor plans showing the spatial relationships of rooms and sizes, or external renderings showing the style and feel of the building. Following these are intermediate drawings giving specific dimensions and layouts. Eventually come the construction drawings with full details for the builder. The architect's role does not have a universally applicable stopping point, but the most common case is based on the needs of the client. When the designs are sufficiently refined (in enough views) for the client to make a decision to proceed with construction, the architect's conceptual development job is complete. The architect may be busy with the project for some time to come in shepherding the conceptual design through detailed design, overseeing construction, and advising the client on certification; but the initial concept role is complete when the client can make the construction decision.

In a different domain, the beginning computer programmer is taught a similar practice. Stepwise refinement in programming means to write the central controlling routine first. Anywhere high complexity occurs, ignore it by giving it a descriptive name and making it a subroutine or function. Each subroutine or function is "stubbed," that is, given a dummy body as a placeholder. When the controlling routine is complete, it is compiled and executed as a test. Of course, it doesn't do anything useful since its subroutines are stubbed. The process is repeated recursively on each subroutine until routines can be easily coded in primitive statements in the programming language. At each intermediate step an abstracted version of the whole program exists that has the final program's structure but lacks internal details.

Both examples show progression of system representation or modeling. Progression also occurs along other dimensions. For example, both the civil architect and the programmer may create distinct alternative designs in their early stages. How are these partial designs evaluated to choose the superior approach? In the earliest stages both the programmer and the civil architect use heuristic reasoning. The civil architect can measure rough size (to estimate cost), judge the client's reaction, and ask the aesthetic opinion of others. The programmer can judge code size, heuristically evaluate the coupling and cohesion of the resulting subroutines and modules, review applicable patterns from catalogs, and review functionality with the client. As their work progresses, both will be able to make increasing use of rational and quantitative evaluation criteria. The civil architect will have enough details for proven cost models, and the programmer can measure execution speed, compiled size, behavioral compliance, and invoke quantitative software quality metrics.

Design as the evolution of models

All architects, indeed all designers, manipulate models of the system. These models become successively less abstract as design progresses. The integrated models discussed in Chapter 10 exhibit stepwise reduction of abstraction in representation and in their design heuristics.

In Hatley/Pirbhai the reduction of abstraction is from behavioral model to technology-specific behavioral model to architecture model. There is also hierarchical decomposition within each component. The technology of modules is indeterminate at the top level, and becomes technology-specific as the hierarchy develops. The Q^2FD performance modeling technique shows stepwise refinement of customer objectives into engineering parameters. As the QFD chain continues, the engineering parameters get closer to implementation until, ultimately, they may represent machine settings on the factory floor. Likewise, the structure of integrated models in software and manufacturing systems follows the same logic or progression.

Evaluation criteria and heuristic refinement

The criteria for evaluating a design progresses or evolves in the same manner as design models. In evaluation, the desirable progression is from general to system-specific to quantitative. For heuristics, the desirable progression is from descriptive and prescriptive qualitatives to domain-specific quantitatives and rational metrics. This progression is best illustrated by following the progression of a widely recognized heuristic into quantitative metrics within a particular discipline. Start with the partitioning heuristic:

> **In partitioning, choose the elements so that they are as independent as possible, that is, elements with low external complexity and high internal complexity.**

This heuristic is largely independent of domain. It serves as an evaluation criterion and partitioning guide whether the system is digital hardware, software, human-driven, or otherwise. But, the guidance is nonspecific; neither independence nor complexity is defined. By moving to a more restricted domain, computer-based systems in this example, this heuristic refines into more prescriptive and specific guidelines. The literature on structured design for software (or, more generally, computer-based systems) includes several heuristics directly related to the partitioning heuristic.[1] The structure of the progression is illustrated in Figure 9.1.

1. Module fan-in should be maximized. Module fan-out should generally not exceed 7 ± 2.
2. The coupling between modules should be — in order of preference — data, data structure, control, common, and content.

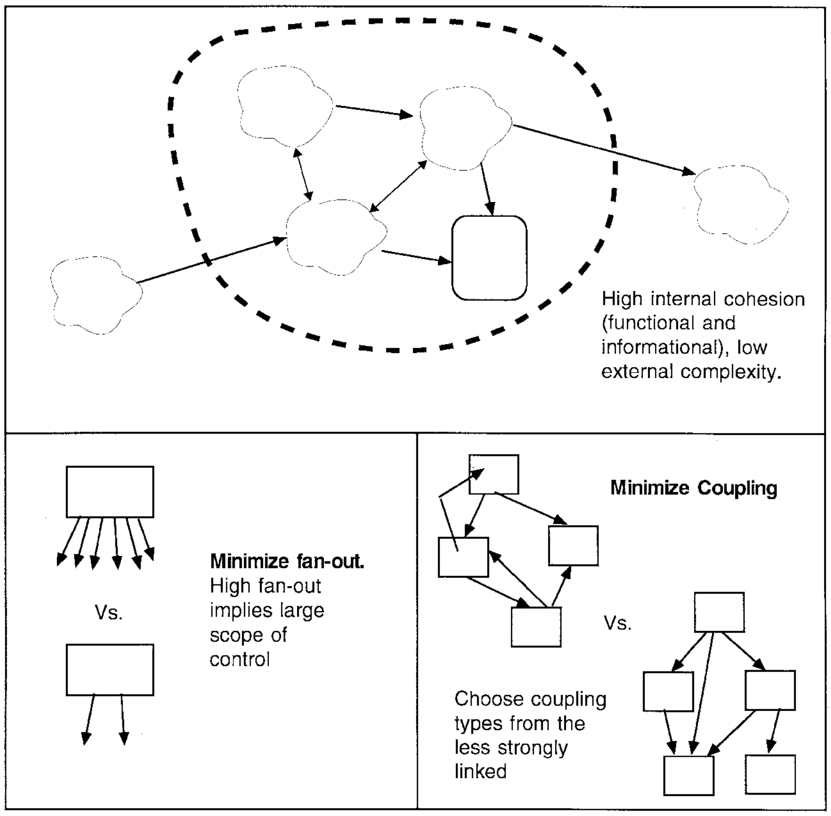

High internal cohesion (functional and informational), low external complexity.
Minimize fan-out. High fan-out implies large scope of control
Vs.
Minimize Coupling
Vs.
Choose coupling types from the less strongly linked

Figure 9.1 Software refinement of coupling and cohesion heuristic. The general heuristic is refined into a domain-specific set of heuristics.

3. The cohesion of the functions allocated to a particular module should be — in order of preference — functional/control, sequential, communicational, temporal, periodic, procedural, logical, and coincidental.

These heuristics give complexity and independence more specific form. As the domain restricts even farther, the next step is to refine into quantitative design quality metrics. This level of refinement requires a specific domain and detailed research, and is the concern of specialists in each domain. But, to finish the example, the heuristic can be formulated into a quantitative software complexity metric. A very simple example is:

Compute a complexity score by summing: one point for each line of code, 2 points for each decision point, 5 points for each external routine call, 2 points for

each write to a module variable, 10 points for each write to a global variable.*

Early evaluation criteria or heuristics must be as unbounded as the system choices. As the system becomes constrained, so do the evaluation criteria. What was a general heuristic judgment becomes a domain-specific guideline, and, finally, a quantitative design metric.

Progression of emphasis

On a more abstract level, the social or political meaning of a system to its developers also progresses. A system goes from being a product (something new) to a source of profit or something of value to a policy (something of permanence). In the earliest stages of a system's life it is most likely viewed as a product. It is something new, an engineering challenge. As it becomes established and its development program progresses, it becomes an object of value to the organization. Once the system exists it acquires an assumption of permanence. The system, its capabilities, and its actions become part of the organization's nature. To have and operate the system becomes a policy that defines the organization.

With commercial systems the progression is from product innovation to business profit to corporate process.[2] Groups innovate something new, successful systems become businesses, and established corporations perpetuate a supersystem that encompasses the system, its ongoing development, and its support. Public systems follow a similar progression. At their inception they are new, at their development they acquire a constituency, and they eventually become a bureaucratic producer of a commodity.

Concurrent progressions

Other concurrent progressions include risk management, cost estimating, and perceptions of success. Risk management progresses in specificity and goals. Early risk management is primarily heuristic with a mix of rational methods. As prototypes are developed and experiments conducted, risk management mixes with interpretation. Solid information begins to replace engineering estimates. After system construction, risk management shifts to postincident diagnostics. System failures must be diagnosed, a process that should end in rational analysis but may have to be guided by heuristic reasoning.

Cost estimating goes through an evolution similar to other evaluation criteria. Unlike other evaluation criteria, cost is a continually evolving characteristic from the systems inception. At the very beginning the need for an estimate is highest and the information available is lowest. Little information

* Much more sophisticated complexity metrics have been published in the software engineering literature. One of the most popular is the McCabe metric, for which there is a large automated toolset.

is available because the design is incomplete and no uncertainties have been resolved. As development proceeds, more information is available, both because the design and plans become more complete and because actual costs are incurred. Incurred costs are no longer estimates. When all costs are in (if such an event can actually be identified) there is no longer a need for an estimate. Cost estimating goes through a progression of declining need but of continuously increasing information.

All of the "ilities" are part of their own parallel progressions. These system characteristics are known precisely only when the system is deployed. Reliability, for example, is known exactly when measured in the field. During development, reliability must be estimated from models of the system. Early in the process the customer's desires for reliability may be well known, but the reliability performance is quite uncertain. As the design progresses to lower levels, the information needed to refine reliability estimates become known, information like parts counts, temperatures, and redundancy.

Perceptions of success evolve from architect to client and back to architect. The architect's initial perception is based on system objectives determined through client interaction. The basic measure of success for the architect becomes successful certification. But once the system is delivered, the client will perceive success on his or her own terms. The project may produce a system that is successfully certified but that nonetheless becomes a disappointment. Success is affected by all other conditions affecting the client at delivery and operation, whether or not anticipated during design.

Episodic nature

The emphasis on progression appears to define a monotonic process. Architecting begins in judgment, rough models, and heuristics. The heuristics are refined along with the models as the system becomes bounded until rational, disciplinary engineering is reached. In practice, the process is more cyclic or episodic, with alternating periods of synthesis, rational analysis, and heuristic problem solving. These episodes occur during system architecting, and may appear again in later, domain-specific stages.

The occurrence of the episodes is integral to the architect's process. An architect's design role is not restricted solely to "high-level" considerations. Architects dig down into specific subsystem and domain details where necessary to establish feasibility and determine client-significant performance (see Chapter 1, Figure 1.1, and the associated discussion). The overall process is one of high-level structuring and synthesis (based on heuristic insight) followed by rational analysis of selected details. Facts learned from those analyses may cause reconsideration of high-level synthesis decisions and spark another episode of synthesis and analysis. Eventually, there should be convergence to an architectural configuration, and the driving role passes to subsystem engineers.

Design concepts for systems architecture

While systems design is an inherently complicated and irregular practice, it has well-established and identifiable characteristics and can be organized into a logical process. As discussed in the Preface, the activities of architecting can be distinguished from other engineering activities, even if not crisply. Architecting is characterized by the following:

1. Architecting is, predominantly, an eclectic mix of rational and heuristic engineering. Other elements, such as normative rules and group processes, enter in lesser roles.
2. Architecting revolves around models, but is composed of the basic processes of scoping, aggregation, partitioning, integration, and certification. Few complete rational methods exist for these processes, and the principal guidelines are heuristic.
3. Uncertainty is inherent in complex system design. Heuristics are specialized tools to reduce or control, but not eliminate, uncertainty.
4. Continuous progression on many fronts is an organizing principle of architecting, architecture models, and supporting activities.

Civil engineering and architecture are perhaps the most mature of all engineering disciplines. Mankind has more experience with engineering civil structures than any other field. If any area could have the knowledge necessary to make it a fully rational and scientific endeavor it should be civil engineering. But it is in civil practice that the distinction between architecture and engineering is best established. Both architects and engineers have their roles, often codified in law, and their professional training programs emphasize different skills. Architects deal particularly with those problems that cannot be entirely rationalized by scientific inquiry. The architect's approach does not ignore science; it combines it with art. Civil engineers must likewise deal with unrationalizable problems, but the focus of their concerns is with well-understood, rational design and specification problems. By analogy, this suggests that all design domains contain an irreducible kernel of problems that are best addressed through creative and heuristic approaches that combine art and science. This kernel of problems, it might be called the architectonic kernel, is resistant to being subsumed into engineering science because it inherently binds together social processes (client interaction) with engineering and science. The social side is how we determine and understand people's needs. The engineering and science side is determining the feasibility of a system concept. The bridge is the creative process of imagining system concepts in response to expressions of client need.

Historical approaches to architecting

As indicated in the introduction to Part One, civil architects recognize four basic theories of design: the normative or pronouncement, the rational, the argumentative or participative, and the heuristic. While all have their roots in the civil architecture practice, they are recognizable in modern complex systems as well. They have been discussed before, in particular in Rechtin, 1991,[3] and in the Introduction to Part One. The purpose in returning to them here is to indicate their relationship to progressive modeling and to bring in their relevance to software-oriented development.* To review, normative theory is built from pronouncements (statements of what should be, a set of hard rules), most often given as restrictions on the content of particular views (usually form). A pronouncement demands that a particular type of form be used, essentially unchanged, throughout. Alternatively, one may pronounce the reverse and demand that certain types of form *not* be used. In either case, success is *defined* by accurate implementation of the normative pronouncements, not by measures of fitness.

Rational system design is tightly integrated with modeling since it seeks to derive optimal solutions, and optimality can only be defined within a structured and mathematical framework. To be effective, rational methods require modeling methods which are broad enough to capture all evaluation aspects of a problem, deep enough to capture the characteristics of possible solutions, and mathematically tractable enough to be solved for problems of useful size. Given these, rational methods "mechanically" synthesize a design from a series of modeled problem statements in progressively more detailed subsystems.

Consensual or participative system design uses models primarily as a means of communicating alternative designs for discussion and negotiation among participants. From the standpoint of modeling, consensuality is one of several techniques for obtaining understanding and approval of stakeholders, rather than of itself a structured process of design.

General heuristics are guides to — and are sometimes obtained from — models, but they are not models themselves. Heuristics are employed at all levels of design, from the most general to domain-specific. Heuristics are needed whenever the complexity of the problem, solutions, and issues overwhelms attempts at rational modeling. This occurs as often in software or digital logic design as in general system design. Within a specific domain the heuristic approach can be increasingly formalized, generating increasingly prescriptive guidance. This formalization is a reflection of the progression of all aspects of design — form, evaluation, and emphasis — from abstract to concrete.

The power of the Chapter 2 heuristics comes by cataloging heuristics which apply in many domains, giving them generality in those domains, likely extensibility in others, and a system-level credibility. Applied consistently through the several levels of system architecture, they help insure system integrity. For example, "quality cannot be tested in, it must be designed in" is equally applicable from the top-level architectural sketch to

* Stepwise refinement is a term borrowed from software that describes program development by sequential construction of programs, each complete unto itself, but containing increasing fractions of the total desired system functionality.

the smallest detail. However, the general heuristic relies on an experienced system-level architect to select the ones appropriate for the system at hand, interpret their application-specific meaning, and to promulgate them throughout its implementation. A catalog of general heuristics is of much less use to the novice; indeed, an uninformed selection among them could be dangerous. For example, "if it ain't broke, don't fix it," questionable at best, can mislead one from making the small incremental changes that often characterize successful continuous improvement programs, and can block one from recognizing the qualitative factors, like ultraquality, that redefine the product line completely.

Specialized and formalized heuristics

While there are many very useful general heuristics, there really is not a general heuristic method as such.* Heuristics most often are formalized as part of more formalized methods within specific domains. A formalized heuristic method gives up generality for more direct guidance in its domain. Popular design methods often contain formalized heuristics as guidelines for design synthesis. A good example is the ADARTS** software engineering methodology. ADARTS provides an extensive set of heuristics to transform a data-flow-oriented behavioral model into a multitasking, modular software implementation. Some examples of formalized ADARTS prescriptive heuristics include:

> **Map a process to an active I/O process if that transformation interfaces to an active I/O device**[4]
>
> **Group processes that read or update the same data store or data from the same I/O device into a single process**[5]

As the ADARTS method makes clear, these are recommended guidelines and not the success-*defining* pronouncements of the normative approach. These heuristics do not produce an optimal, certifiable, or even unique result, much less success-by-definition. There is ambiguity in their application. Different heuristics may produce conflicting software structuring. The software engineer must select from the heuristics list and interpret the design consequences of a given heuristic with awareness of the specific demands of the problem and its implementation environment.

Conceptually, general and domain-specific, formalized heuristics might be arranged in a hierarchy. In this hierarchy, domain-specific heuristics are specializations of general heuristics, and the general are abstractions of the

* On the other hand, knowledge of a codified set of heuristics can lead to new ways of thinking about problems. This could be described as heuristic thinking or qualitative reasoning.
** This method is described in the *ADARTS Guidebook*, SPC-94040-CMC, Version 2.00.13, Volume 1, September 1991, available from the Software Productivity Consortium, Herndon, VA.

specific. Architecting in general and architecting in specific domains may be linked through the progressive refinement and specialization of heuristics. To date, this hierarchy can be clearly identified only for a limited set of heuristics. In any case, the pattern of refining from abstract to specific is a broadly persistent pattern, and it is essential for understanding life cycle design progression.

Scoping, aggregation, partitioning, and certification

A development can be envisioned as the creation and transformation of a series of models. For example, to develop the systems requirements is to develop a model of what the system should do and how effectively it should do it. To develop a system design is to develop a model of what the system is. In a pure waterfall development there is rough alignment between waterfall steps and the views defined in Chapter 8. Requirements development develops models for objectives and performance, functional analysis develops models of behavior, and so on down the waterfall chain. Architects develop models for all views, though not at equal levels of detail. In uncritical views or uncritical portions of the system the models will be rough. In some areas the models may need to be quite detailed from the beginning.

Models are best understood by view because the views reflect their content. While architecting is consistent with waterfall or spiral development, it does not traverse the steps in the conventional manner. Architects make use of all views and traverse all development steps, but at varying levels of detail and completeness. Because architects tend to follow complex paths through design activities, some alternative characterization of design activities independent of view is useful. The principal activities of the architect are scoping, partitioning, aggregation, and certification. Figure 9.2 lists typical activities in each category and suggests some relationships.

Scoping

Scoping procedures are methods for selecting and rejecting problem statements, for defining constraints, and for deciding on what is “inside” or “outside” the system. Scoping implies the ability to rank alternative statements and priorities on the basis of overall desirability or feasibility. Scoping should not design system internals, though some choices may implicitly do so for lack of design alternatives. Desirably, scoping limits what needs to be considered and why.

Scoping might alternatively be referred to as purpose analysis. Purpose analysis is an inquiry into why someone wants the system. Purpose precedes requirements. Requirements are determined by understanding how having a system is valuable to the client, and what combination of fulfilled purposes and systems costs represents a satisfactory and feasible solution.*

* Dr. Kevin Kreitman has pointed out the extensive literature in soft systems theory that applies to purpose analysis.

Scoping	
Purpose Expansion/Contraction	
Behavioral Definition/Analysis	
Large Scale Alternative Consideration	
Client Satisfaction-Builder Feasibility	
Aggregation	**Partitioning**
Functional Aggregation (abstract)	Behavioral-Functional Decomposition
Functional Aggregation (to physical units)	Physical Decomposition (to lower level design)
Physical components to subsystems	Performance Model Construction
Interface Definition/Analysis	Interface Definition/Analysis
Assembly on timelines or behavioral chains	Decomposition to cyclic processes
Collection into Decoupled Threads	Decomposition into Threads
Certification	
Operational Walkthroughs	
Test and Evaluation	
Verification	
Formal Methods Verification	
Failure Assessment	

Figure 9.2 Typical activities within scoping, aggregation, partitioning, and certification.

Scoping is the heart of front-end architecture. A well-scoped system is one that is both desirable and feasible, the essential definition of success in system architecting. As the project begins, the scope forms, at least implicitly. All participants will form mental models of the system and its characteristics; in doing so, the system's scope is being defined. If incompatible models appear, scoping has failed through inconsistency. Heuristics suggest that scoping is among the most important of all system design activities. One of the most popular heuristics in the previous book has been: **all the really important mistakes are made the first day.** Its popularity certainly suggests that badly chosen system scope is a common source of system disasters.

Of course, it is as impossible to prevent mistakes on the first day as it is on any other day. What the heuristic indicates is that mistakes of initial conception will have the worst long-term impact on the project. Therefore, one must be particularly careful that a mistake of scope is discovered and corrected as soon as possible.* One way of doing this is to defer absolute decisions on scope by retaining expansive and restrictive options as long as possible, a course of action recommended by other heuristics (the options heuristics of Appendix A).

In principle, scope can be determined rationally through decision theory. Decision theory applies to any selection problem. In this case the things being selected are problem statements, constraints, and system contexts. In practice, the limits of decision theory apply especially strongly to scoping decisions. These limits, discussed in greater detail in a subsequent section,

* A formalized heuristic with a similar idea comes from software engineering. It says: the cost of removing a defect rises exponentially with the time (in project phases) between its insertion and discovery. Hence, mistakes of scope (the very earliest) are potentially dominant in defect costs.

include the problems of utility for multiple system stakeholders, problem scale, and uncertainty. The judgments of experienced architects, at least as expressed through heuristics (see Appendix A for a detailed list), is that the most useful techniques to establish system scope are qualitative.

Scoping heuristics and decision theory share an emphasis on careful consideration of who will use the system and who will judge success. Decision theory requires a utility function, a mathematical representation of system value as a function of its attributes. A utility function can be determined only by knowing whose judgments of system value will have priority and what the evaluation criteria are. Compare the precision of the utility method to related heuristics:

Success is defined by the beholder, not by the architect.

The most important single element of success is to listen closely to what the customer perceives as his requirements and to have the will and ability to be responsive (Steiner, J. E., 1978).

Ask early about how you will evaluate the success of your efforts (Hayes-Roth et al., 1983).

Scoping heuristics suggest approaches to setting scope that are outside the usual compromise procedures of focused engineering. One way to resolve intractable problems of scope is to expand. The heuristic is:

Moving to a larger purpose widens the range of solutions (Nadler, G., 1990).

The goal of scoping is to form a concept of what the system will do, how effectively it will do it, and how it will interact with the outside world. The level of detail required is the level required to gain customer acceptance, first of continued development and ultimately of the built system. Thus the scope of the architect's activities are governed not by the ultimate needs of system development, but by the requirements of the architect's further role. The natural conclusion to scoping is certification, where the architect determines that the system is fit for use. Put another way, the certification is that the system is appropriate for its scope.

Scoping is not solely a requirements-related activity. For scope to be successfully determined, the resulting system must be both satisfactory and feasible. The feasibility part requires some development of the system design. The primary activities in design by architects are aggregation and partitioning, the basic structuring of the system into components.

Aggregation and partitioning

Aggregation and partitioning are the grouping and separation of related solutions and problems. They are two sides of the same coin. Both are the processes by which the system is defined as components. While one can argue about which precedes the other, they are, in fact, used so iteratively and repeatedly that neither can be usefully said to precede the other. Conventionally, the components are arranged into hierarchies with a modest number of components at each level of the hierarchy (the famous 7 ± 2 structuring heuristic). The most important aggregation and partitioning heuristics are to minimize external coupling and maximize internal cohesion, usually worded as[6]:

In partitioning, choose the elements so that they are as independent as possible, that is, elements with low external complexity and high internal cohesion.

Group elements that are strongly related each other, separate elements that are unrelated.

These two heuristics are especially interesting because they are part of the clearest hierarchy in heuristic refinement. Design is usefully viewed as progressive or stepwise refinement. Models, evaluation criteria, heuristics, and other factors are all refined as design progresses from abstract to concrete and specific. Ideally, heuristics exist in hierarchies that connect general design guidance, such as the two preceding heuristics, to domain-specific design guidelines. The downward direction of refinement is the deduction of domain-specific guidelines from general heuristics. The upward abstraction is the induction of general heuristics from similar guidelines across domains.

The deductive direction asks, taking the coupling heuristic as an example, how can coupling be measured? Or, for the cohesion heuristic, given alternative designs, which is the most cohesive? Within a specific domain the questions should have more specific answers. For example, within the software domain these questions are answered with greater refinement, though still heuristically. Studies have demonstrated quantitative impact on system properties as the two heuristics are more and less embodied in a systems design. A generally accepted software measure of partitioning is based on interface characterization and has five ranked levels. A related metric for aggregation quality (or cohesion) has seven ranked levels of cohesion.* Studies of large software systems show a strong correlation between coupling and cohesion levels, defect rates, and maintenance costs. A major study[7] found that routines with the worst coupling-to-cohesion ratios (inter-

* The cohesion and coupling levels are carefully discussed in Yourdon, E. and Constantine, L. L., *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Yourdon Press, Englewood Cliffs, 1979. They were introduced earlier by the same authors and others in several papers.

face complexity to internal coherence) had 7 times more errors and 20 times higher maintenance costs than the routines with the best ratios.

Aggregation and partitioning with controlled fan-out and limited communication is a tested approach to building systems in comprehensible hierarchies. Numerous studies in specific domains have shown that choosing loosely coupled and highly cohesive elements leads to systems with low maintenance cost and low defect rates. However, nature suggests that much flatter hierarchies can yield systems of equal or greater robustness in certain circumstances.

Chapter 8 introduced an ant colony as an example of a flat hierarchy system that exhibits complex and adaptive behavior. The components of an ant colony, a few classes of ants, interact in a very flat system hierarchy. Communication is loose and hierarchically unstructured. There is no intelligent central direction. Nevertheless, the colony as a whole produces complex system-level behavior. The patterns of local, nonhierarchical interaction produce very complex operations. The colony is also very robust in that large numbers of its components (except the queen) can be removed catastrophically and the system will smoothly adapt. A related technological example, perhaps, is the Internet. Again, the system as a whole has a relatively flat hierarchy and no strong central direction; however, the patterns of local communication and resulting collaboration are able to produce complex, stable, and robust system-level behavior.

The observations that controlled and limited fan-out and interaction (the 7 ± 2 heuristic and coupling and cohesion studies) and that extreme fan-out and high distributed communication and control (ant colonies and the Internet) can both lead to high-quality systems is not contradictory. Rather, they are complementary observations of the refinement of aggregation and partitioning into specific domains. In both cases a happy choice of aggregations and partitions yield good systems. But the specific indicators of what constitutes good aggregation and partitioning vary with the domain. The general heuristic stands for all, but prescriptive or formalized guidance must be adapted for the domain.

Certification

To certify a system is to give an assurance to the paying client that the system is fit for use. Certifications can be elaborate, formal, and very complex, or the opposite. The complexity and thoroughness are dependent on the system. A house can be certified by visual inspection. A computer flight control system might require highly detailed testing, extensive product and process inspections, and even formal mathematical proofs of design elements. Certification presents two distinct problems. The first is determining that the functionality desired by the client and created by the builder is acceptable. The second is the assessment of defects revealed during testing and inspection, and the evaluation of those failures with respect to client demands.

Whether or not a system possesses a desired property can be stated as a mathematically precise proposition. Formal methods develop and track and verify such propositions throughout development, ideally leading to a formal proof that the system as designed possesses the desired properties. However, architecting practice has been to treat such questions heuristically, relying on judgment and experience to formulate tests and acceptance procedures. The heuristics on certification criteria do not address what such criteria should be, but the process for developing the criteria. Essentially, certification should not be treated separately from scoping or design. Certifiability must be inherent in the design. Two summarizing heuristics, actually on scoping and planning, are:

For a system to meet its acceptance criteria to the satisfaction of all parties, it must be architected, designed, and built to do so — no more and no less.

Define how an acceptance criterion is to be certified at the same time the criterion is established.

The first part of certification is intimately connected to the conceptual phase. The system can be certified as possessing desired criteria only to the extent it is designed to support such certification. The second element of certification, dealing with failure, carries its own heuristics. These heuristics emphasize a highly organized and rigorous approach to defect analysis and removal. Once a defect is discovered it should not be considered resolved until it has been traced to its original source, corrected, the correction tested at least as thoroughly as was needed to find the defect originally, and the process recorded. Deming's famous heuristic summarizes:

Tally the defects, analyze them, trace them to the source, make corrections, keep a record of what happens afterwards, and keep repeating it.

A complex problem in ultraquality systems is the need to certify levels of performance that cannot be directly observed. Suppose a missile system is required to have a 99% success rate with 95% confidence. Suppose further that only 50 missiles are fired in acceptance tests (perhaps because of cost constraints). Even if no failures are experienced during testing, the requirement cannot be quantitatively certified. Even worse, suppose a few failures occurred early in the 50 tests but were followed by flawless performance after repair of some design defects. How can the architect certify the system? It is quite possible that the system meets the requirement, but it cannot be proven.

Certification of ultraquality might be deemed a problem of requirements. Many would argue that no requirement should be levied that cannot be quantitatively shown. But the problem will not go away. The only acceptable failure levels in one-of-a-kind systems and those with large public safety impacts will be immeasurable. No such systems can be certified if certification in the absence of quantitatively provable data is not possible.

Some heuristics address this problem. The Deming approach, given as a heuristic above, seeks to achieve any quality level by continuous incremental improvement. Interestingly, there is a somewhat contradictory heuristic in the software domain. When a software system is tested, the number of defects discovered should level off as testing continues. The amount of additional test time to find each additional defect should increase, and the total number of discovered defects will level out. The leveling out of the number of defects discovered gives an illusion that the system is now defect free. In practice, testing or reviews at any level rarely consistently find more than 60% of the defects present in a system. But if testing at a given level finds only a fixed percentage of defects, it likewise leaves a fixed percentage undiscovered. The size of that undiscovered set will be roughly proportional to the number found in that same level of test or review. The heuristic can be given in two forms:

The number of defects remaining in a system after a given level of test or review (design review, unit test, system test, etc.) is proportional to the number found during that test or review

Testing can indicate the absence of defects in a system only when: (1) the test intensity is known from other systems to find a high percentage of defects, and (2) few or no defects are discovered in the system under test.

So the discovery and removal of defects is not necessarily an indication of a high-quality system. A variation of the "zero-defects" philosophy is that ultraquality requires ultraquality throughout all development processes. That is, certify a lack of defects in the final product by insisting on a lack of defects anywhere in the development process. The ultraquality problem is a particular example of the interplay of uncertainty, heuristics, and rational methods in making architectural choices. That interplay needs to be examined directly to understand how heuristic and rational methods interact in the progression of system design.

Certainty, rationality, and choice

All of the design processes — scoping, partitioning, aggregation, and certification — require decisions. They require decisions on which problem statement to accept, what components to organize the system into, or when the system has reached an acceptable level of development. A by-product of a heuristic-based approach is continuous uncertainty. Looking back on their projects, most architects interviewed for the USC program concluded that the key choices they made were rarely obvious decisions at the time. Although, in retrospect, it may be obvious that a decision was either a very good or a very bad one, at the time the decision was actually made it was not clear at all. The heuristic summarizing is **before the flight it was opinion, after the flight it was obvious.** The members of the teams were constantly arguing and decision was reached only through the authority of the leading architect for the project.*

A very considerable effort has been made to develop rational decision-making methods. The goal of a fully rational or scientific approach is to make decisions optimally with respect to rigorously determined criteria. Again, the model of architecting practice presented here is a pragmatic mixture of heuristic and rigor. Decision theory works well when the problem can be parameterized with a modest number of values, uncertainty is limited and estimates are reliable, and the client or users possess consistent utility functions with tractable mathematical expression. The absence of any of these conditions weakens or precludes the approach. Unfortunately, some or all of the conditions are usually absent in architecting problems (and even in more restricted disciplinary design problems). To understand why, one must understand the elements of the decision theoretic approach. The decision theoretic framework is:

1. Identify the attributes contributing to client satisfaction and an algorithm for estimating the value of sets of attributes. More formally, this is determining the set over which the client will express preference.
2. Determine a utility function, a function that combines all the attributes and represents overall client satisfaction. Weighted, additive utility functions are commonly used, but not required. The utility function converts preferences into a mathematically useful objective function.
3. Include uncertainty by determining probabilities, calculating the utility probability distribution, and determining the client's risk aversion curve. The risk aversion curve is a utility theory quantity that measures the client's willingness to trade risk for return.
4. Select the decision with the highest-weighted expected utility.

The first problem in applying this framework to architecting problems is scale. To choose an optimum, the decision theory user must be able to maximize the utility functions over the decision set. If the set is very large, the problem is computationally unfeasible. If the relationship between the parameters and utility is nonlinear, only relatively small problems are solvable. Unfortunately, both conditions commonly apply to the architecting and creation of complex systems.

The second problem is to workably and rationally include the effects of uncertainty or risk. In principle, uncertainty and unreliability in estimates can be folded into the decision theoretic framework through probability and

* Comments by Harry Hillaker at USC on his experience as YF-16 architect.

assessment of the client's risk aversion curve. The risk aversion curve measures the client's willingness to trade risk and return. A risk-neutral client wants the strategy that maximizes expected return. A risk-averse client prefers a strategy with certainty over opportunity for greater return. A risk-disposed client prefers the opposite, wanting the opportunity for greater return even if the expectation of the return is less.

In practice, however, the process of including uncertainty is heavily subjective. For example, how can one estimate probabilities for unprecedented events? If the probabilities are inaccurate, the whole framework loses its claim to optimality. Estimation of risk aversion curves is likewise subjective, at least in practice. When so much subjective judgment has been introduced, it is unclear if maintaining the analytical framework leads to much benefit or if it is simply a gloss.

One clear benefit of the decision theory framework is that it makes the decision criteria explicit and, thus, subject to direct criticism and discussion. This beneficial explicitness can be obtained without the full framework. This approach is to drop the analytic gloss, make decisions based on heuristics and architectural judgment, but (and this is more honored in the breach) require the basis be explicitly given and recorded.

A third problem with attempting to fully rationalize architectural decisions is that for many of them there will be multiple clients who have some claim to express a preference. Single clients can be assumed to have consistent preferences and, hence, consistent utility functions. However, consistent utility functions do not generally exist when the client or user is a group as in sociotechnical systems.* Even with single clients, value judgments may change, especially after the system is delivered and the client acquires direct experience.**

Rational and analytical methods produce a gloss of certainty, but often hide highly subjective choices. No hard-and-fast guideline exists for choosing between analytical choice and heuristic choice when unquantified uncertainties exist. Certainly, when the situation is well understood and uncertainties can be statistically measured, the decision theoretic framework is appropriate. When even the right questions are in doubt it adds little to the process to quantify them. Intermediate conditions call for intermediate criteria and methods. For example, a system might have as client objectives "be flexible" and "leave in options." Obviously, these criteria are open to interpretation. The refinement approach is to derive or specialize increasingly specific criteria from very general criteria. This process creates a con-

* This problem with multiple clients and decision theory has been extensively studied in literature on public choice and political philosophy. A tutorial reference is Mueller, D. C., *Public Choice,* Cambridge University Press, 1979.

** Non-utility theory based decision methods, such as the *Analytic Hierarchy Process*, have many of the same problems. Most writers have discussed that the primary role of decision theoretic methods should be to elucidate the underlying preferences. See Saaty, T., *The Analytic Network Process,* RWS Publications, 1996, Preface and Chapter 1.

tinuous progression of evaluation criteria from general to specific, and eventually measurable.

> Example: The U.S. Department of Transportation has financed an Intelligent Transport System (ITS) architecture development effort. Among their evaluation criteria was "system flexibility," obviously a very loose criterion.[8] An initial refinement of the loose criterion could be:
>
> 1. Architecture components should fit in many alternative architectures.
> 2. Architecture components should support multiple services.
> 3. Architecture components should expand with linear or sublinear cost to address greater load.
> 4. Components should support non-ITS services.

These refined heuristic evaluation criteria can be applied directly to candidate architectures, or they can be further refined into quantitative and measurable criteria. The intermediate refinement on the way to quantitative and measurable criteria creates a progression that threads through the whole development process. Instead of thinking of design as beginning and stopping, it continuously progresses. Sophisticated mixtures of the heuristics and rational methods are part of architecting practice in some domains. This progression is the topic of the next section.

Stopping or progressing?

When does architecting and modeling stop? The short answer is that given earlier: they never stop, they progress. The architecting process (along with many other parallel tracks) continuously progresses from the abstract to the concrete in a steady reduction of abstraction. In a narrow sense, there are recognizable points at which some aspects of architecting and modeling must stop. To physically fabricate a component of a system, its design must be frozen. It may not stop until the lathe stops turning or the final line of code is typed in, but the physical object is the realization of *some* design. In the broader sense, even physical fabrication does not stop architecting. Operations can be interpreted only through recourse to models, though those models may be quite precise when driven by real data. In some systems, such as distant space probes, even operational modeling is still somewhat remote.

The significant progressions in architecting are promoted by the role of the architect. The architect's role makes two decisions foremost, the selection of a system concept and the certification of the built system. The former decision is almost certain to be driven by heuristic criteria, the latter is more open, depending on how precisely the criteria of fitness for use can be defined. A system concept is suitable when it is both satisfactory and feasible. Only the client can judge the system "satisfactory," though the client will have to rely on information provided by the architect. In builder-architected systems the architect must often make the judgment for the client (who will hopefully appear after the system reaches the market). Feasible means the system can be developed and deployed with acceptable risk. Certification requires that the system as built adequately fulfills the client's purposes, including cost, as well as the contract with the builder.

Risk, a principal element in judging feasibility, is almost certain to be judged heuristically. The rational means of handling risk is through probability, but a probabilistic risk assessment requires some set of precedents to estimate over. That is, a series of developments of similar nature for which the performance and cost history is known. By definition, such a history cannot be available for unprecedented systems, so the architect is left to estimate risk by other means. In well-defined domains, past history should be able to provide a useful guide; it certainly does for civil architects. Civil architects are expected to control cost and schedule risk for new structures, and they can do so because construction cost estimation methods are reasonably well developed. The desired approach is to use judgment, and perhaps a catalog of domain-specific heuristics, to size the development effort against past systems, and use documented development data from those past systems to estimate risk. For example, in software systems, cost models based on code size estimates are known, and they are often calibrated against past development projects in builder organizations. If the architect can deduce code size and possible variation in code size reliably, a traceable estimate of cost risk is possible.

The judgment of how satisfactory a concept is, and the certification process, both depend on how well customer purposes can be specified. Here there is great latitude for both heuristic and rational means. If customer purposes can be precisely specified, it may be possible to precisely judge how well a system fulfills them, either in prospect or retrospect. In prospect it depends on having behavior and performance models that are firmly attached to customer purposes. With good models with certain connection between the models and implementation technology, the architect can confidently predict how well the planned system will fulfill the desired purposes. The retrospective problem is that of certification, of determining how well the built system fulfills the customer purposes. Again, well-founded, scientific models and mature implementation technologies make system assessment relatively certain.

More heuristic problems arise when the same factors do not apply. Mainly, this occurs when it is hard to formulate precise customer purpose models, when it is hard to determine whether or not a built system fulfills a precisely stated purpose, or when there is uncertainty about the connection between model and implemented system. The second two are related since they both concern retrospective assessment of architecture models against a built system in the presence of uncertainty.

The first case applies when customer purposes are vague or likely to change in response to actual experience with the system. When the customer is relatively inexperienced with systems of the type, his or her perception of the system's value and requirements is likely to change, perhaps radically, with experience. Vague customer purposes can be addressed through the architecture; for example, an emphasis on options in the architecture and a development plan that includes early user prototypes with the ability to feed prototype experience back into the architecture. This is nicely captured in two heuristics:

Firm commitments are best made after the prototype works.[9]

Hang on to the agony of decision as long as possible.[10]

The second case, problems in determining whether or not a built system fulfills a given purpose, is mainly a problem when requirements are fundamentally immeasurable or when performance is demanded in an environment that cannot be provided for test. For example, a space system may require a failure rate so low it will never occur during any practical test (the ultraquality problem); or, a weapon system may be required to operate in the presence of hostile countermeasures that will not exist outside a real combat environment. Neither of these requirements can be certified by test or analysis. To certify a system with requirements like these it is necessary to either substitute surrogate requirements agreed to by the client, or to find alternative certification criteria.

To architect-in certifiable criteria essentially means to substitute a refined set of measurable criteria for the client's immeasurable criteria. This requires the architect to be able to convince the client of the validity of a model for connecting the refined criteria to the original criteria. One advantage of a third-party architect is the independent architect's greater credibility in making just such arguments, which may be critical to developing a certifiable system. A builder-architect, with an apparent conflict of interest, may not have the same credibility. The model that connects the surrogate criteria to the real, immeasurable criteria may be a detailed mathematical model or may be quite heuristic. An example of the former category is failure tree analysis that tries to certify untestable reliability levels from testable subsystem reliability levels. A more heuristic model may be more appropriate for certifying performance in uncertain combat environments. While the performance required is uncertain, criteria like flexibility, reprogrammability, performance reserve, fallback modes, and ability to withstand damage can be specified and measured.

Rational and heuristic methods can be combined to develop ultraquality systems. A good example is a paper by Jaynarayan.[11] This paper discusses the architectural principles for developing flight control computers with failure rates as low as 10^{-10}/h. Certification of such systems is a major problem. The authors discuss a two-pronged approach. First, instead of using brute-force failure modes and effects analysis with its enormous fault trees, they design for "Byzantine failure." Byzantine failure means failure in which the failed element actively, intelligently, and malevolently attempts to cause system failure. They go on to describe formal methods for designing systems resistant to a given number of Byzantine faults, thus replacing the need to trace fault trees for each type of failure. The analysis of failure trees is then brought down to tractable size. The approach is based on designs that do not allow information or energy from a possibly failed element to propagate outside an error confinement region. The second prong is a collection of guidelines for minimizing common mode failures. In a common mode failure, several nominally independent redundant units fail simultaneously for the same reason. These are the system failures due to design errors rather than component failures. Because one cannot design-in resistance to design failure, other means are necessary. The guidelines, partially a set of heuristics, provide guidance in this otherwise nonmeasurable area.

The third and last case is uncertainty about the connection between the model and the actual system. This is an additional case where informed judgment and heuristics are needed. To reduce the uncertainty in modeling requires tests and prototypes. The best guidance on architecting prototypes is to realize that all prototypes should be purpose-driven. Even when the purposes of the system are less than clear, the purposes of the prototype should be quite clear. Thus, the architecting of the prototype can be approached as architecting a system, with the architect as the client.

Architecture and design disciplines

Not very many years ago the design of a system of the complexity of several tens of thousands of logic gates was a major undertaking. It was an architectural task in the sense it was probably motivated by an explicit purpose and required the coordination of a multidisciplinary design effort. Today, components of similar and much higher complexity are the everyday building blocks of the specialized digital designer. No architectural effort is required to use such a component, or even to design a new one. In principle, art has been largely removed from the design process because the discipline has a firm scientific basis. In other words, the design discipline or domain is well worked out, and the practitioners are recognized specialists. Today it is common to discuss digital logic synthesis directly from fairly level specifications, even if automated synthesis is not yet common practice. So there should be no surprise if systems that today tax our abilities and require architectural efforts one day become routine with recognized design methodologies taught in undergraduate courses.

The discussion of progression leads to further understanding of the distinctions between architecture and engineering. The basic distinction was reviewed in the Preface, the types of problems addressed and the tools used to address them. A refinement of the distinction was discussed in Chapter 1, the extent to which the practitioner is primarily concerned with scoping, conceptualizing, and certification. By looking at the spectrum of architecture and engineering across systems disciplines, these distinctions become clearer and can be further refined. First, the methods most associated with architecting (heuristics) work best one step beyond where rational design disciplines have been worked out. This may or may not be at the forefront of component technology. Large-scale systems, by their nature, push the limits of scientific engineering at whatever level of technology development is current. But, as design and manufacturing technology change the level of integration that is considered a component, the relative working position of the architect inevitably changes. Where the science does not exist the designer must be guided by art. With familiarity and repetition, much that was done heuristically can now be done scientifically or procedurally.

However, this does not imply that where technology is mature architecting does not exist. If it did, there would be no need for civil architects. Only systems that are relatively unique need to be architected. Development issues for unique systems contain a kernel of architectural concerns that transcend whatever technology or scientific level is current. This kernel concerns the bridge between human needs (which must be determined through social interaction and are not the domain of science) and technological systems. In low-technology systems, like buildings, only the nonroutine building needs to be architected. But dealing with the nonroutine, the unique, the client/user-customized, is different from other engineering practices. It contains an irreducible component of art. A series of related unprecedented systems establishes a precedent. The precedent establishes recognized patterns, sometimes called architectures, of recognized worth. Further, systems in the field will commonly use those established architectures, with variations more on style than in core structure.

Current development in software engineering provides an example of evolution to a design discipline. Until relatively recently the notion of software engineering hardly existed; there was only programming. Programming is the process of assembling software from programming language statements. Programming language statements do not provide a very rich language for expressing system behaviors. They are constrained to basic arithmetic, logical, and assignment operations. To build complex system behaviors, programs are structured into higher-level components that begin to express system domain concepts; but in traditional programming each of the components must be handcrafted from the raw material of programming languages.

The progression in software is through the construction and standardization of components embodying behaviors closer and closer to problem domains. Instead of programming in what was considered a "high-level language," the engineer can now build a system from components close to the problem domain. The programming language is still used, but it may be used primarily to knit together prebuilt components. Programming libraries have been in common use for many years. The libraries shipped with commercial software development environments are often very large and contain extensive class or object libraries. In certain domains the gap has grown very small.

> Example: The popular mathematics package MatLab allows direct manipulation of matrices and vectors. It also provides a rich library of functions targeted at control engineers, image processing specialists, and other fields. One can dispense with the matrices and vectors altogether by "programming" with a graphical block diagram interface that hides the computational details and provides hundreds of prebuilt blocks. Further extensions allow the block diagram to be compiled into executable programs that run on remote machines. In fact, direct connection to operating real-time systems for real-time control is possible.

Wherever a family of related systems is built, a set of accepted models and abstractions appears and forms the basis for a specialized design discipline. If the family becomes important enough, the design discipline will attract enough research attention to build scientific foundations. It will truly become a design discipline when universities form departments devoted to it. At the same time, a set of common design abstractions will be recognized as "architectures" for the family. Mary Shaw, observing the software field, finds components and patterns constrained by component and connector vocabulary, topology, and semantic constraints. These patterns can be termed "styles" of architecture in the field, as discussed in Chapter 6, for software.

Architecture and patterns

The progression from "inspired" architecture to formal design method is through long experience. Long experience in the discipline by its practitioners eventually yields tested patterns of function and form. Patterns, pattern languages, and styles are a formalization of this progression. Architecting in a domain matures as architects identify reusable components and repeating styles of connection. Put another way, they recognize recurring patterns of form and their relationships to patterns in problems. In a mature domain, patterns in both the problem and solution domain develop rigorous expression. In digital logic (a relatively mature design domain) problems are stated in formal logic and solutions in equally mathematically well-founded components. In a less mature domain the patterns are more abstract or heuristic.

A formalization of patterns in architecture is due to Christopher Alexander.[12] Working within civil architecture and urban design, Alexander developed an approach to synthesis based on the composition of formalized patterns. A pattern is a recurring structure within a design domain. They consist of both a problem or functional objective for a system and a solution. Patterns may be quite concrete (such as "a sunny corner") or relatively abstract (such as "masters and apprentices"). A template for defining a pattern is

1. A brief name that describes what the pattern accomplishes
2. A concise problem statement
3. A description of the problem including the motivation for the pattern and the issues in resolving the problem
4. A solution, preferably stated in the form of an instruction
5. A discussion of how the pattern relates to other patterns in the language

A pattern language is set of patterns complete enough for design within a domain. It is a method for composing patterns to synthesis solutions to diverse objectives. In the Alexandrian method the architect consults sets of patterns and chooses from them those patterns which evoke the elements desired in a project. The patterns become the building blocks for synthesis, or suggest important elements that should be present in the building. The patterns each suggest instructions for solution structure, or contain a solution fragment. The fragments and instructions are merged to yield a system design.

Because the definition of a pattern and a pattern language are quite general, they can be applied to other forms of architecture. The ideas of patterns and pattern languages are now a subject of active interest in software engineering.* Software architects often use the term style to refer to recurring patterns in high-level software design. Various authors have suggested patterns in software using a pattern template similar to that of Alexander. An example of a software pattern is "callbacks and handlers," a commonly used style of organizing system-dependent bindings of code to fixed behavioral requirements.

The concept of a style is related to Alexandrian patterns since each style can be described using the pattern template. Patterns are also a special class of heuristic. A pattern is a prescriptive heuristic describing particular choices of form and their relationship to particular problems. Unlike patterns, heuristics are not tied to a particular domain.

Although the boundaries are not sharp, heuristics, patterns, styles, and integrated design methods can be thought to form a progression. Heuristics are the most general, spanning domains and categories of guidance. How-

* A brief summary with some further references is Bercuzk, C., Hot topics, finding solutions through pattern languages, *IEEE Computer,* pp. 75-76, 1995.

ever, they are also the least precise and give the least guidance to the novice. Patterns are specially documented, prescriptive heuristics of form. They prescribe (perhaps suggest) particular solutions to particular problems within a domain. A style is still more precisely defined guidance, this time in the form of domain-specific structure. Still farther along the maturity curve are fully integrated design methods. These have domain-specific models and specific procedures for building the models, transforming models of one type into another type, and implementing a system from the models.

Thus, the largest scale progression is from architecting to a rigorous and disciplined design method; one that is essential to the normative theory of design. Along the way the domain acquires heuristics, patterns, and styles of proven worth. As the heuristics, patterns, and styles become more specific, precise, and prescriptive, they give the most guidance to the novice and come closest to the normative (what *should* be) theory of design. As design methods become more precise and rigorous they also become more amenable to scientific study and improvement. Thus the progression carries from a period requiring (and encouraging) highly creative and innovative architecting to one defined by quantifiable and provable science.

Civil architecture experience suggests that at the end of the road there will still be a segment of practice that is best addressed through a fusion of art and science. This segment will be primarily concerned with the clients of a system, and will seek to reconcile client satisfaction and technical feasibility. The choice of method will depend on the question. If you want to know how a building will fare in a hurricane, you know to ask a structural engineer. If you want the building to express your desires, and do so in a way beyond a rote calculation of floor space and room types, you know to ask an architect.

Conclusions

A fundamental challenge in defining a system architecting method or a system architecting tool kit is its unstructured and eclectic nature. Architecting is synthesis-oriented and operates in domains and with concerns that preclude rote synthesis. Successful architects proceed through a mixture of heuristic and rational or scientific methods. One meta-method that helps organize the architecting process is that of progression.

Architecting proceeds from the abstract and general to the domain-specific. The transition from the unstructured and broad concerns of architecting to the structured and narrow concerns of developed design domains is not sharp. It is progressive as abstract models are gradually given form through transformation to increasingly domain-specific models. At the same time, all other aspects of the system undergo concurrent progressions from general to specific.

While the emphasis has been on the heuristic and unstructured components of the process, that is not to undervalue the quantitative and scientific elements required. The rational and scientific elements are tied to the specific domains where systems are sufficiently constrained to allow scientific study. The broad outlines of architecting are best seen apart from any specific domain. A few examples of the intermediate steps in progression were given in this chapter. The next chapter brings these threads together by showing specific examples of models and their association with heuristic progression. In part, this is done for the domains of Part Two, and in part for other recognized large domains not specifically discussed in Part Two.

Exercises

1. Find an additional heuristic progression by working from the specific to the general. Find one or more related design heuristics in a technology-specific domain. Generalize those heuristics to one or more heuristics that apply across several domains.
2. Find an additional heuristic progression by working from the general to the specific. Choose one or more heuristics from the Appendix. Find or deduce domain-specific heuristic design guidelines in a technology domain familiar to you.
3. Examine the hypothesis that there is an identifiable set of "architectural" concerns in a domain familiar to you. What issues in the domain are unlikely to be reducible to normative rules or rational synthesis?
4. Trace the progression of behavioral modeling throughout the development cycle of a system familiar to you.
5. Trace the progression of physical modeling throughout the development cycle of a system familiar to you.
6. Trace the progression of performance modeling throughout the development cycle of a system familiar to you.
7. Trace the progression of cost estimation throughout the development cycle of a system familiar to you.

Notes and references

1. Yourdon, E. and Constantine, L. L., *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Yourdon Press, Englewood Cliffs, NJ, 1979.
2. Ben Baumeister, personal communication.
3. Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991, 14.
4. *ADARTS Guidebook*, p. 8-4.
5. *ADARTS Guidebook*, p. 8.
6. Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991, 312.
7. Selby, R. W. and Basili, V. R., "Analyzing error-prone system structure," *IEEE Trans. Software Eng.,* SE-17, (2), 141, February 1991.

8. Discussed further by one author in Maier, M. W., On Architecting and Intelligent Transport Systems, *Joint Issue IEEE Transactions on Aerospace and Electronic Systems/System Engineering*, AES33:2, p. 610, April 1997.
9. Comments to the authors on Rechtin, 1991 by L. Bernstein in the context of large network software systems.
10. Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991, 315.
11. Jaynarayan, H. and Harper, R., Architectural principles for safety-critical real-time *a*pplications, *Proc. IEEE,* 82:1, 25, January 1994.
12. Alexander, C., *The Timeless Way of Building,* Oxford University Press, New York; and *A Pattern Language: Towns, Buildings, Construction,* Oxford University Press, New York, 1977.

chapter ten

Integrated modeling methodologies

Introduction

The previous two chapters explored the concepts of model views, model integration, and progression along parallel paths. This chapter brings together these threads by presenting examples of integrated modeling methodologies. Part Three concludes in the next chapter where we review the architecture community's standards for architecture description. The methodologies are divided by domain-specificity, with the first models more nearly domain-independent and later models more domain-specific.

Architecting clearly is domain dependent. A good architect of avionics systems, for example, may not be able to effectively architect social systems. Hence, there is no attempt to introduce a single set of models suitable for architecting everything. The models of greatest interest are those tied to the domain of interest, although they must support the level of abstraction needed in architecting. The integrated models chosen for this chapter include two principally intended for real-time, computer-based, mixed hardware/software systems (H/P and Q^2FD), three methods for software-based systems, one method for manufacturing systems, and, conceptually at least, some methods for including human behavior in sociotechnical system descriptions.

The examples for each method were chosen from relatively simple systems. They are intended as illustrations of the methods and their relevance to architectural modeling, and to fit within the scope of the book. They are not intended as case studies in system architecting.

General integrated models

The two most general of the integrated modeling methods are Hatley-Pirbhai (H/P) and Q^2FD. The Unified Modeling Language (UML) is also quite general, although in practice it is used mostly in software systems and is dealt with there. As the UML evolves, it is likely that it will become of general applicability in systems engineering.

Hatley/Pirbhai — computer-based reactive systems

A computer-based reactive system is a system that senses and reacts to events in the physical world, and in which much of the implementation complexity is in programmable computers. Multifunction automobile engine controllers, programmable manufacturing robots, and military avionics systems (among many others) all fall into this category. They are distinguished by mixing continuous, discrete, and discrete-event logics, and being implemented largely through modern computer technology. The integrated models used to describe these systems emphasize detailed behavior descriptions, and form descriptions matched to software and computer hardware technologies and some performance modeling. Recent efforts at defining an Engineering of computer-based Systems discipline[1] are directed at systems of this type.

Several different investigators have worked to build integrated models for computer-based reactive systems. The most complete example of such integration is the Hatley-Pirbhai (H/P) methodology.* Other methods, notably the FFBD method used in computer tools by Ascent Logic and the StateMate formalism, are close in level of completeness. The UML is also close to this level of completeness, and may surpass it, but is not yet widely used in systems applications. This section concentrates on the structure of H/P. With the concepts of H/P in mind, it is straightforward to make a comparative assessment of other tools and methods.

H/P defines a system through three primary models: two behavioral models — the Requirements Model (RM) and the Enhanced Requirements Model (ERM) — and a model of form called the Architecture Model (AM). The two behavioral models are linked through an embedding process. Static allocation tables link the behavioral and form models. The performance view is linked statically through timing allocation tables. More complex performance models have been integrated with H/P, but descriptions have only recently been published. A dictionary defines the data view. This dictionary provides a hierarchical data element decomposition, but does not provide a syntax for defining dynamic data relationships. No managerial view is provided, although managerial metrics have been defined for models of the H/P type.

Both behavioral models are based on DeMarco-style data flow diagrams. The data flow diagrams are extended to include finite state and event processing through what is called the control model. The control model uses

* Wood, D. P. and Wood, W. G., Comparative Evaluation of Four Specification Methods for Real-Time Systems, *Software Engineering Institute Technical Report*, CMU/SEI-89-TR-36, 1989. This study compared four popular system modeling methods. Their conclusion was that the Hatley-Pirbhai method was the most complete of the four, though similarities were more important than the differences. In the intervening time, many of the popular methods have been extended and additional tools reflecting multiview integration have begun to appear.

data flow diagram syntax with discrete events and finite state machine processing specifications. The behavioral modeling syntax is deliberately nonrigorous and is not designed for automated execution. This "lack" of rigor is deliberate; it is intended to encourage flexibility in client/user communication. The method believes the flexibility rather than rigor at this stage enhances communication with clients and users. The method also believes, through its choice of data flow diagrams, that functional decomposition communicates to stakeholders better than specification by example methods, such as use-cases. The ERM is a superset of the requirements model. It surrounds the core behavioral model and provides a behavioral specification of the processing necessary to resolve the physical interfaces into problem domain logical interfaces. The ERM defines implementation-dependent behaviors, such as user interface and physical I/O.

Example: microsatellite imaging system

Some portions of an H/P model formulated for the imaging (camera) subsystem of a microsatellite provide an illustration of the H/P concepts. This example is to present the flavor of the H/P idiom for architects, not to fully define the imaging system. The level chosen is representative of that of a subsystem architecture (not all architecting has to be done on systems of enormous scale). Figure 10.1 shows the top-level behavioral model of the imaging system, defined as a data flow diagram (DFD). Each circle on the diagram represents a data-triggered function or process. For example, process Number 2, Evaluate Image, is triggered by the presence of a Raw Image data element. Also from the diagram, process Number 2 produces a data element of the same type (the outgoing arrow labeled Raw Image) and another data element called Image Evals.

Each process in the behavior model is defined either by its own data flow diagram or by a textual specification. During early development, processes may be defined with brief and nonrigorous textual specifications. Later, as processes are allocated to physical modules, the specifications are expanded in greater detail until implementation of appropriate rigor is reached. Complex processes may have more detailed specifications even early in the process. For example, in Figure 10.2 process Number 1, Form Image, is expanded into its own diagram.

Figure 10.2 also introduces control flow. The dotted arrows indicate flow of control elements, and the solid line into which they flow is a control specification. The control specification is shown as part of the same figure. Control flows may be interpreted either as continuous time, discrete valued data items, or discrete events. The latter interpretation is more widely used, although it is not preferred in the published H/P examples. The control specification is a finite state machine, here shown as a state transition diagram, although other forms are also possible. The actions produced by the state machine are to activate or deactivate processes on the associated data flow diagram.

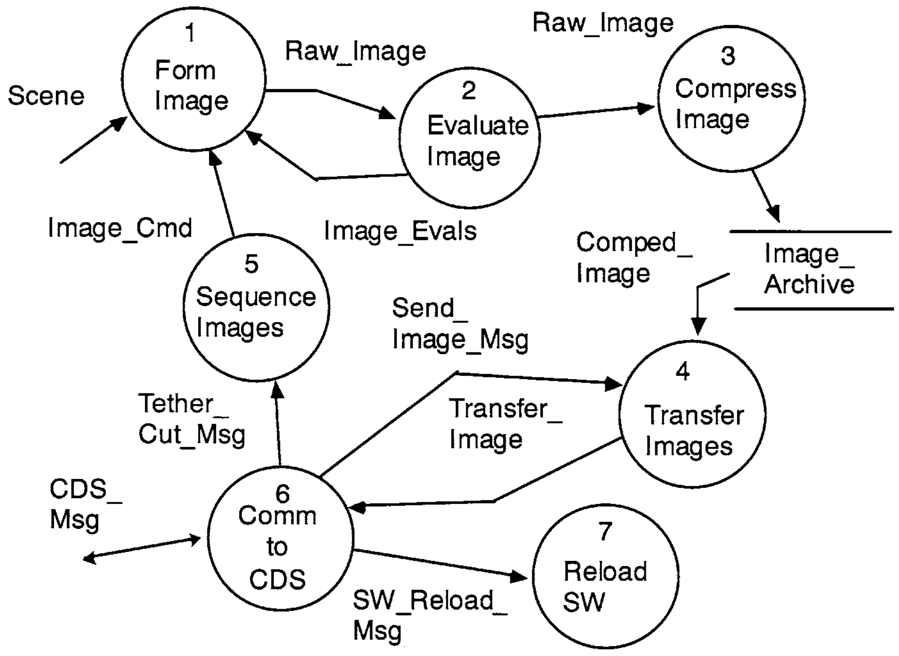


Figure 10.1

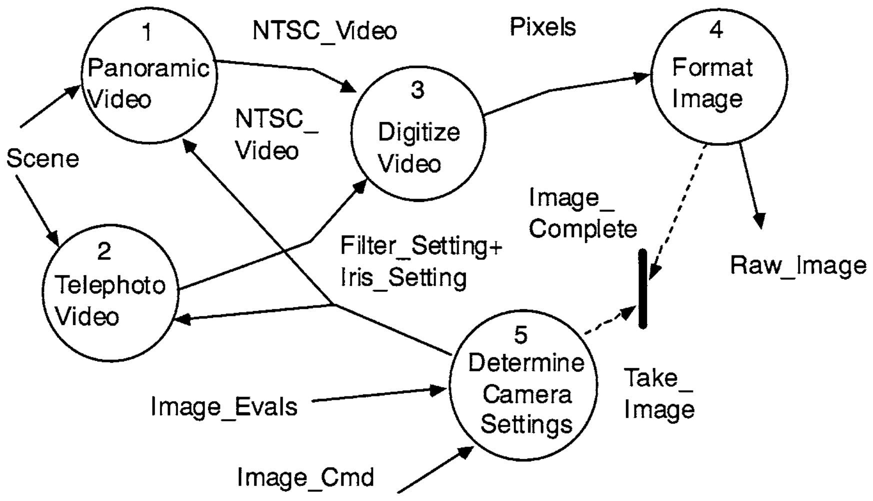


Figure 10.2

All data elements appearing on a diagram are defined in the data dictionary. Each may be defined in terms of lower-level data elements. For example, the flow Raw Image appearing in Figure 10.2 appears in the data dictionary as:

Raw Image = 768{484{Pixel}}

Indicating, in this case, that Raw Image is composed of 768×484 repetitions of the element Pixel. At early stages, Pixel is defined qualitatively as a range of luminance values. In later design stages the definition will be augmented, though not replaced, by a definition in terms of implementation-specific data elements.

In addition to the two behavior models, the H/P method contains an AM. The AM is the model of form that defines the physical implementation. It is hierarchical and it allows sequential definition in greater detail by expansion of modules. Figure 10.3 shows the paired architecture flow and interconnect models for the microsatellite imaging system.

The H/P block diagram syntax partitions a system into modules, which are defined as physically identifiable implementation elements. The flow diagram shows the exchange of data elements among the modules. Which data elements are exchanged among the modules is defined by the allocation of behavioral model processes to the modules.

The interconnection model defines the physical channels through which the data elements flow. Each interconnect is further defined in a separate specification. For example, the interconnect T-Puter Channel 1 connects the processor module and the camera control module. Allocation requires camera commands to flow over the channel. Augmentations to the data dictionary define a mapping between the logical camera commands and the line codes of the channel. If the channel requires message framing, protocol processing, or the like, it is defined in the interconnection specification. Again, the level of detail provided can vary during design based on the interface's impact on risk and feasibility.

Quantitative QFD (Q^2FD) — performance-driven systems

Many systems are driven by quantitatively stated performance objectives. These systems may also contain complex behavior or other attributes, but their performance objectives are of most importance to the client. For these systems it is common practice to take a performance-centered approach to system specification, decomposition, and synthesis. A particularly attractive way of organizing decomposition is through extended Quality Function Deployment (QFD) matrices.[2]

QFD is a Japanese-originated method for visually organizing the decomposition of customer objectives.[3] It builds a graphical hierarchy of how customer objectives are addressed throughout a system design, and it carries the relevance of customer objectives throughout design. A Q^2FD-based approach requires that the architect:

1. Identify a set of performance objectives of interest to the customer. Determine appropriate values or ranges for meeting these objectives through competitive analysis.

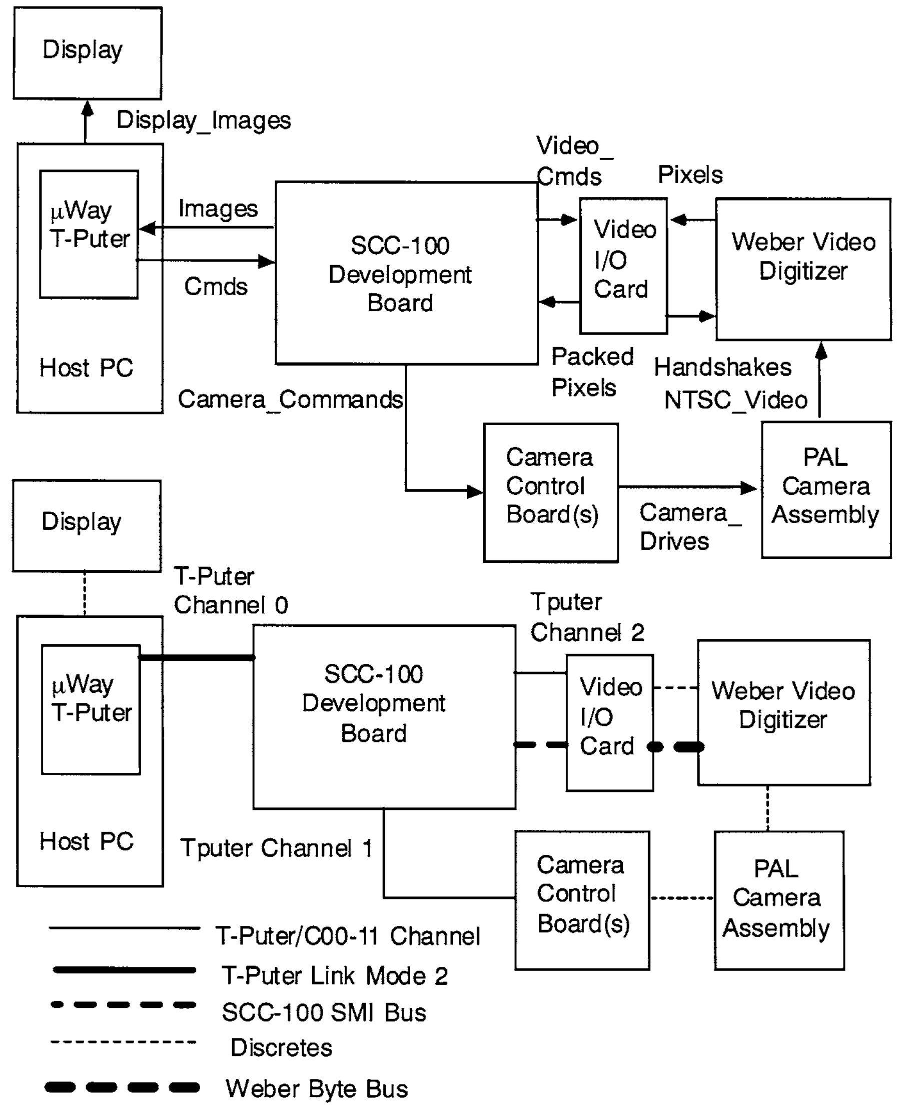


Figure 10.3

2. Identify the set of system-level design parameters that determine the performance for each objective. Determine suitable satisfaction models that relate the parameters and objectives.
3. Determine the relationships of the parameters and objectives, and the interrelationships among the parameters. Which affect which?
4. Set one or more values for each parameter. Multiple values may be set, for example, minimum, nominal, and target. Additional slots provide tracking from detailed design activities.

5. Repeat the process iteratively using the system design parameters as objectives. At each stage the parameters at the next level up become the objectives at the next level down.
6. Continue the process of decomposition on as many levels as desired. As detailed designs are developed, their parameter values can flow up the hierarchy to track estimated performance for customer objectives. This structure is illustrated in Figure 10.4.

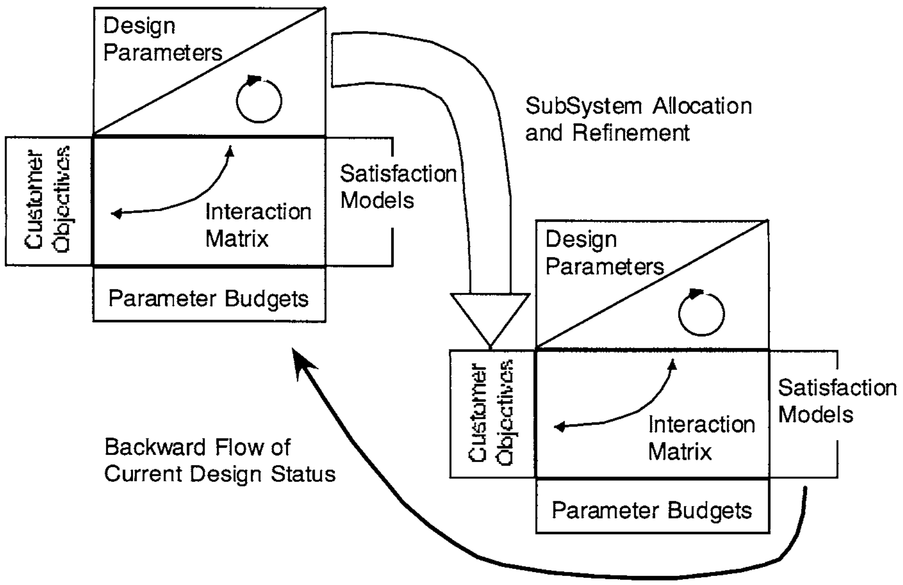


Figure 10.4

Unfortunately, QFD models for real problems tend to produce quite large matrices. Because they map directly to computer spreadsheets, this causes no difficulty in modern work environments, but it does cause a problem in presenting an example. Also, the graphic of the matrix shows the result, but hides the satisfaction models. The satisfaction models are equations, simulations, or assessment processes necessary to determine the performance measure value. The original reference on QFD by Hauser contains a qualitative example of using QFD for objective decomposition, as do other books on QFD. Two papers by one of the present authors[4] contain detailed, quantitative examples of QFD performance decomposition using analytical engineering models.

Integrated modeling and software

Chapters 8 and 9 introduced the ideas of model views and stepwise refinement in the large. Both of these ideas have featured prominently in the software engineering literature. Software methods have been the principal sources for detailed methods for expressing multiple views and development through refinement. Software engineers have developed several integrated modeling and development methodologies that integrate across views and employ explicit heuristics. Three of those methods are described in detail: structured analysis and design, ADARTS, and OMT. We also take up the current direction in an integrated language for software-centric systems, the UML.

The three methods are targeted at different kinds of software systems. Structured analysis and design was developed in the late 1970s and early 1980s and is intended for single-threaded software systems written in structured procedural languages. ADARTS is intended for large, real-time, multithreaded systems written in Ada. Its methods do generalize to other environments. OMT is intended for database-intensive systems, especially those written in object-oriented programming languages. The UML is a merger of object-oriented concepts from OMT and other sources.

Structured analysis and design

The first of the integrated models for software was the combination of structured analysis with structured design.[5] The software modeling and design paradigms established in that book have continued to the present as one of the fundamental approaches to software development. Structured analysis and design models two system views, uses a variety of heuristics to form each view, and connects to the management view through measurable characteristics of the analysis and design models (metrics).

The method prescribes development in three basic steps. Each step is quite complex and is composed of many internal steps of refinement. The first step is to prepare a data flow decomposition of the system to be built. The second step is to transform that data flow decomposition into a function and module hierarchy that fully defines the structure of the software in subroutines and their interaction. The design hierarchy is then coded in the programming language of choice. The design hierarchy can be mechanically converted to software code (several tools do automatic forward and backward conversion of structured design diagrams and code). The internals of each routine are coded from the included process specifications, though this requires human effort.

The first step, known as structured analysis, is to prepare a data flow decomposition of the system to be built. A data flow decomposition is a tree hierarchy of data flow diagrams, textual specifications for the leaf nodes of the hierarchy, and an associated data dictionary. This method was first popularized by DeMarco,[6] though the ideas had appeared previously, and it has since been extensively modified and re-presented. Figures 10.1 and 10.2, discussed in the previous example, show data flow diagrams. Behavioral analysis by data flow diagram originated in software and has since been applied to more general systems. The basic tenets of structured analysis are:

1. Show the structure of the problem graphically, engaging the mind's ability to perceive structure and relationships in graphics.
2. Limit the scope of information presented in any diagram to five to nine processes and their associated data flows.
3. Use short (<1 page) free-form and textual specifications at the leaf nodes to express detailed processing requirements.
4. Structure the models so each piece of information is defined in one and only one place. This eases maintenance.
5. Build models in which the processes are loosely coupled, strongly cohesive, and obey a defined syntax for balance and correctness.

Structured design follows structured analysis and transforms a structured analysis model into the framework for a software implementation. The basic structured design model is the structure chart. A structure chart, one is illustrated in Figure 10.5, shows a tree hierarchy of software routines. The arrows connecting boxes indicate the invocation of one routine or subroutine by another. The circles, arrows, and names show the exchange of variables and are known as data couples. Additional symbols are available for pathological connection among routines, such as unconditional jumps. Each box on the structure chart is linked to a textual specification of the requirements for that routine. The data couples are linked to a data dictionary.

Structure charts are closely aligned with the ideas and methods of structured programming, which was a major innovation at the time structured design was introduced. Structure charts can be mechanically converted to nested subroutines in languages that support the structured programming concepts. In combination, the chart structure, the interfaces shown on the chart, and the linked module specifications define a compilable shell for the program and an extended set of code comments. If the module specifications are written formally, they can be the module's program design language, or they can be compiled as module pre- and postcondition assertions.

The structured analysis and design method goes farther in providing detailed heuristics for transformation of an analysis model into a structure chart, and for evaluation of alternative designs. The heuristics are strongly prescriptive in the sense that they are stated procedurally. However, they are still heuristics because their guidance is provisional and subject to interpretation in the overall context of the problem. The transformation is a type of refinement or reduction of abstraction. The data flow network of the analysis phase defines data exchange, but it does not define execution order beyond that implied by the data flow. Hence the structure chart removes the abstraction of flow of control by fixing the invocation hierarchy. The heuristics provided are of two types. One type gives guidelines for transforming a data flow model fragment into a module hierarchy. The other type measures comparative design quality to assist in selection among alternative designs. Examples of the first type include:

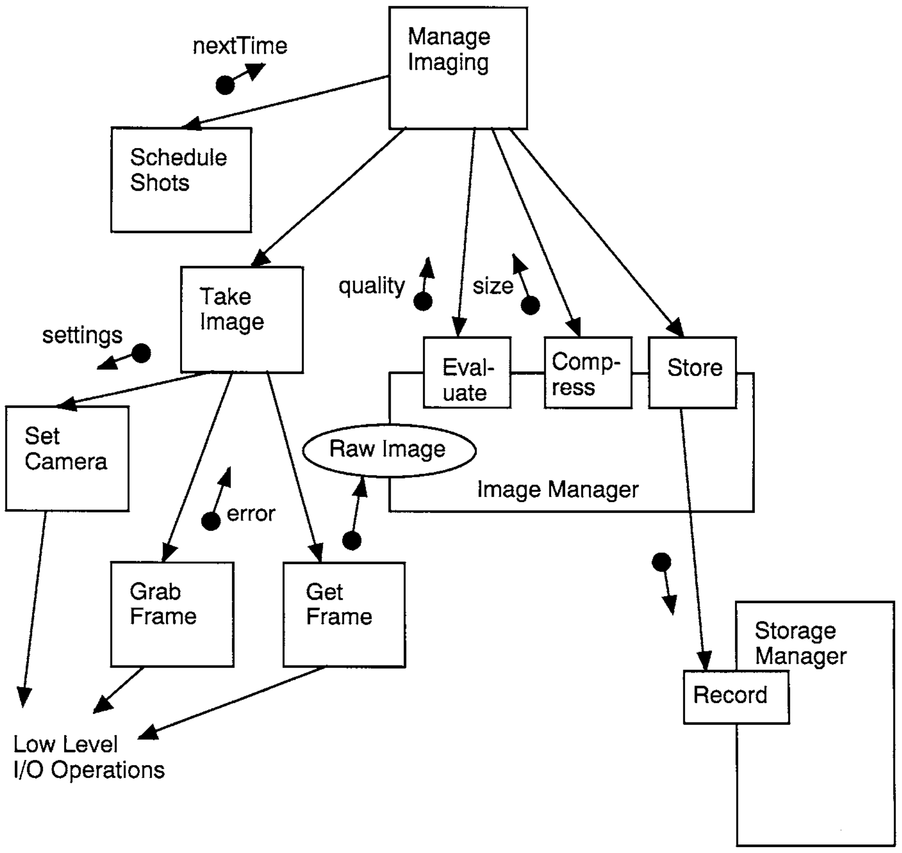


Figure 10.5

Step one: Classify each data flow diagram as "transform-oriented" or "transaction-oriented" (these terms are further defined in the method).
Step two: In each case, find either the "transform center" or the "transaction center" of the diagram and begin factoring the modules from there.

Further heuristics follow for structuring transform-centered and transaction-centered processes. In the second category are several quite famous design heuristics:

- **Choose designs which are loosely coupled. Coupling, from loosest to tightest, is measured as: data, data structure, control, global, and content.**

- **Choose designs in which the modules are strongly cohesive. Cohesion is rated as (from strongest to weakest): functional, sequential, communicational, procedural, temporal, logical, and coincidental.**
- **Choose modules with high fan-in and low fan-out.**

As discussed in Chapter 9, very general and domain-specific heuristics may be related by chains of refinement. In structured analysis and design, the software designer transforms rough ideas into data flow diagrams, data flow diagrams into structure charts, and structure charts into code. At the same time, heuristic guidelines like "strive for loose coupling" are given measurable form as the design is refined into specific programming constructs.

Various efforts have also been made to tie structured analysis and design to managerial models by predicting cost, effort, and quality from measurable attributes of data flow diagrams or structure charts. This is done both directly and indirectly. A direct approach computes a system complexity metric from the data flow diagrams or the structure charts. That complexity metric then must be correlated to effort, cost, schedule, or other quantities of management interest. A later work by DeMarco[7] describes a detailed approach on these lines, but the suggested metrics have not become popular nor have they been widely validated on significant projects. Other metrics, such as function or feature points, that are more loosely related to structured analysis decompositions have found some popularity. Software metrics is an ongoing research area, and there is a growing body of literature on measurements that appear to correlate well with project performance.

An alternative linkage is indirect by using the analysis and design models to guide estimates of the most widely accepted metrics, the constructive cost model (COCOMO) and effective lines of code (ELOC). COCOMO is Barry Boehm's famous effort estimation formula. The model predicts development effort from a formula involving the total lines of code, an exponent dependent on the project type, and various weighting factors. One problem with the original COCOMO model is that it does not differentiate between newly written lines of code and reused code. One method (there are others) of extending the COCOMO model is to use ELOC in place of total lines of code. ELOC measures the size of a software project, giving allowance for modified and reused code. A new line of code counts for one ELOC, modified and unmodified reused code packages count for somewhat less. The weight factors given to each are typically determined organization-by-organization based on past measurements. The counts by subtype are summed with their weights and the total treated as new lines in the COCOMO model.

The alternative approach is to use the models to guide ELOC estimation. Early in the process, when no code has been written, the main source of error in COCOMO is likely to be errors in the ELOC estimate. With a data flow model in hand, engineers and managers can go through it process-by-process and compare the requirements to past efforts by the organization.

This, at least, structures the estimation problem into identifiable pieces. Similarly, the structured design model can be used in the same way, with estimates of the ELOC for each module flowing upward into a system-level estimate. As code is written the estimates become facts and, hopefully, the estimated and actual efforts will converge. Of course, if the organization is incapable of producing a given ELOC level predictably, then any linkage of analysis and design models to managerial models is moot.

The architect needs to be cognizant of these issues insofar as they affect judgments of feasibility. As the architect develops models of the system, they should be used jointly by client and builder. The primary importance of cost models is in the effect they have on the client's willingness to go forward with a project. A client's resources are always limited, and an intelligent decision on system construction can be made only with knowledge of the resources it will consume. Of course, there will be risk, and in immature fields like software the use of risk mitigation techniques (such as spiral development) may partially replace accurate early estimates. Both the client's value judgments and the builder's estimates should be made in the context of the models. If builder organizations have a lot of variance in what effort is required to deliver a fixed complexity system, then that variance is a risk to the client.

ADARTS

Ada-based Design Approach for Real-Time Systems (ADARTS) is an extensively documented example of a more advanced integrated modeling method for software. The original work on data flow techniques was directly tied to the advanced implementation paradigms of the day. In a similar way, the discrete event, system-oriented specification methods like H/P can be closely tied to implementation models. In the case of real-time, event-driven software, one of the most extensive methods is the ADARTS[8] methodology of the Software Productivity Consortium. The ADARTS method combines a discrete event-based behavioral model with a detailed, stepwise refined, physical design model. The behavioral model is based on data flow diagrams extended with the discrete event formalisms of Ward and Mellor[9] (which are similar to those of H/P). The physical model includes evolving abstractions for software tasks or threads, objects, routines, and interfaces. It also includes provisions for software distributed across separate machines and their communication. ADARTS includes a catalog of heuristics for choosing and refining the physical structure through several levels of abstraction.

ADARTS links the behavioral and physical models through allocation tables. Performance decomposition and modeling is considered specifically, but only in the context of timing. There are links to sophisticated scheduling formalisms and SPC-developed simulation methodologies as part of this performance link. Again, managerial views are supported through metrics where they can be calculated from the models. Software domain-specific methods can more easily perform the management metric integration since a variety of cost and quality metrics that can be (at least roughly) calculated from software design models are known.

The example shown is a simplified version of the first two design refinements required by ADARTS applied to the microsatellite imaging system originally discussed in the Hatley/Pirbhai example. The resulting diagrams are shown in Figure 10.6. The ADARTS process takes the functional hierarchy of the behavioral model and breaks it into undifferentiated components. Each component is shown on the diagram by a cloud-shaped symbol, indicating its specific implementation structure has not yet been decided. The clouds exchange data elements dependent on the behavior allocated to each cloud. Various heuristics and engineering judgment guide the choice of clouds.

The next refinement specializes the clouds to tasks, modules or objects, and routines. ADARTS actually uses several discrete steps for this, but they are combined into one for the simple example given here. Again, the designer uses ADARTS-provided heuristics and individual judgment in making the refinements. In the example, the two tasks result from the need to provide asynchronous external communications and overall system control. The clouds which hide the physical and logical interfaces to hardware are multientry modules. The entries are chosen from the principal user functions addressed by the interface. For example, the Camera I/O module has entries that correspond to its controls (camera shutter speed, camera gain, filter wheel position, etc.). The single-thread sequence of taking an image is implemented as a simple routine-calling tree.

To avoid diagram clutter the diagram is not fully annotated with the data elements and their flow directions. In complex systems diagram clutter is a serious problem, and one not well addressed by existing tools. The architect needs to suppress some detail to process the larger picture. But correct software ultimately depends on getting each detail right. In the second part of the figure the arrowed lines indicate direction of control, not direction of data flow. Additional enhancements specify flow. The next step in the ADARTS process, not shown here, is to refine the task and module definitions once again into language- and system-specific software units. ADARTS as published assumes the use of the Ada language for implementation. When implementing in the Ada language, tasks become Ada tasks and multientry modules become packages. The public/private interface structure of the modules is implemented directly using constructs of the Ada language. Other languages can be accommodated in the same framework by working out language- and operation-specific constructs equivalent to tasks, modules, and routines. For example, in the C language there is no language construct for tasks or multientry modules. But multientry modules can be implemented in a nearly standard way using separately compilable files on the development system, the static declaration, and suitable header files. Similarly, many implementation environments support multitasking, and some development environments supply task abstractions for the programmer's use.

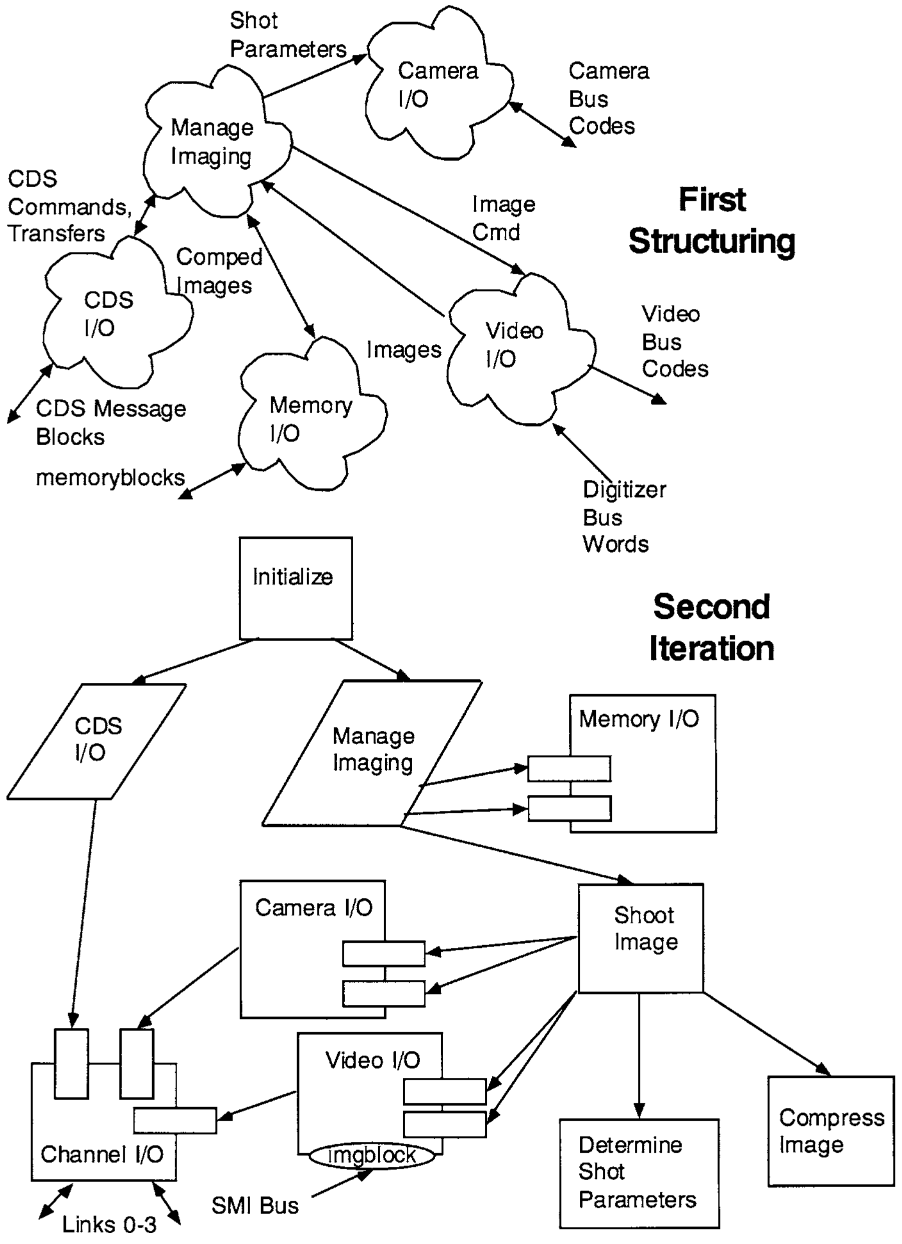


Figure 10.6

Once again, the pattern of stepwise reduction of abstraction is evident. Design is conducted through steps, and at each step a model of the client needs is refined in an implementation environment-dependent way. In environments well matched to the problem modeling method, the number of steps is small; client-relevant models can be nearly directly implemented. In less well-suited environments layers of implementation abstraction become necessary.

OMT

The Hatley/Pirbhai method and its cousins are derived from structured functional decomposition, structured software design, and hardware system engineering practice. The object-oriented methods, of which OMT[10] is a leading member, derive from data-oriented and relational database software design practice. Relational modeling methods focus solely on data structure and content and are largely restricted to database design (where they are very powerful). Object-oriented methods package data and functional decomposition together. Where structured methods build a functional decomposition backbone on which they attempt to integrate a data decomposition, the object-oriented methods emphasize a data decomposition on which the functional decomposition is arranged. Some problems naturally decompose nicely in one method and not in the other. Complex systems can be decomposed with either, but either approach will yield subsections where the dominant decomposition paradigm is awkward.

OMT combines the data (relational), behavioral, and physical views. The physical view is well captured for software-only systems, but specific abstractions are not given for hardware components. While, in principle, OMT and other object-oriented methods can be extended to mixed hardware/software systems, and even more general systems, there is a lack of real examples to demonstrate feasibility. Broad, real experience has been obtained only for predominantly software-based systems.

Neither the OMT nor other object-oriented methods substantially integrate the performance view. Again, managerial views can be integrated to the extent that useful management metrics can be derived from the object models. Because of the software orientation of object-oriented methods, there have been some efforts to integrate formal methods into object models.

As an example of the key ideas of object-oriented methods we present part of an object model. Object modeling starts by identifying classes. Classes can be thought of (for those unfamiliar with object concepts) as templates for objects or types for abstract data types. They define the object in terms of associated simple data items and functions associated with the object. Classes can specialize into subclasses which share the behavior and data of their parent while adding new attributes and behavior. Objects may be composed of complex elements or relate to other objects. Both composition or aggregation and association are part of a class system definition. The microsatellite imager described in the preceding section will produce images of various types. Consider an image database for storing the data produced by the imager. A basic class diagram is shown in Figure 10.7 to illustrate specific instances of some of the concepts.

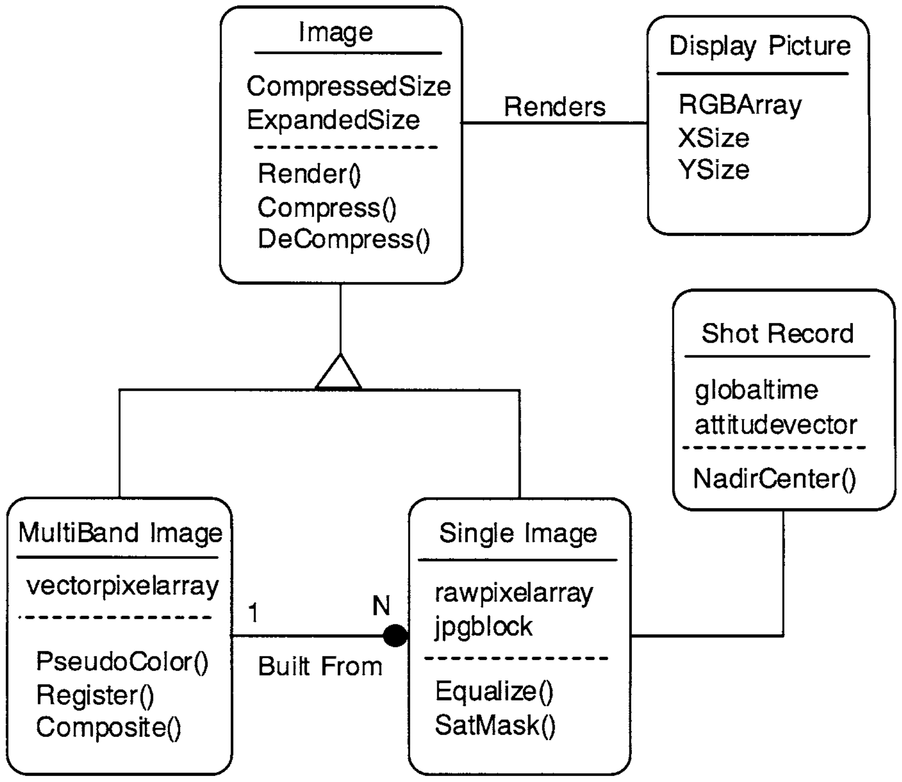


Figure 10.7

A core assumption, which the model must capture, is that images are of several distinct but related types. The actual images captured by the cameras are single grayscale images. Varying sets of grayscale images captured through different filters are combined into composite multiband images, with a particular grayscale image possibly part of several composite images. In addition, images will be displayed on multiple platforms, so we demand a common "rendered" image form. Each of these considerations is illustrated in Figure 10.7.

The top box labeled Image indicates there is a data class Image. That class contains two data attributes — CompressedSize and ExpandedSize — and three operations or "methods" (the functions Render(), Compress(), and Expand()). The triangle boxed lines down to the class boxes Multi-Band Image and Single Image defines those two classes as subclasses of Image. As subclasses they are different than their parent class, but inherit the parent class's data attributes and associated methods.

The class Single Image is the basic image data object descriptor. It contains two data arrays, one to hold the raw image and the other to hold the compressed form. It also has associated basic image processing methods. A multiband image is actually made up of many single images suitably processed and composited. This is defined on the diagram by the round-headed line connecting the two class boxes. The labeling defines a 1 to N way association named Built From. The additional methods associated with Multi-Band Image build the image from its associated simple images.

The two additional associations define other record keeping and display. The associated line between Single Image and Shot Record associates an image with a potentially complicated data record of when it was taken and the conditions at that moment. The association line to Display Picture shows the association of an image with a common display data structure. Both associations, in these cases, are one-to-one.

Figure 10.7 is considerably simplified on several points. A complete definition in OMT would require various enhancements to show actual types associated with data attributes and operations. In addition, several enhancements are required to distinguish abstract methods and derived attributes. A brief explanation of the former is in order. Consider the method Compress in the class Image. The implementation of image compression may be quite different for a single grayscale image and for a composited multiband image. A method that is reimplemented in subclasses is called either virtual or abstracted and may be noted by a diagrammatic enhancement.

The logic of object-oriented methods is to decompose the system in a data-first fashion, with functions and data tightly bound together in classes. Instead of a functional decomposition hierarchy we have a class hierarchy. Functional definition is deferred to the detailed definition of the classes. The object-oriented logic works well where data, and especially data relation complexity, dominates the system.

Object-oriented methods also follow a stepwise reduction of abstraction approach to design. From the basic class model, we next add implementation-specific considerations. These will determine whether or not additional model refinements or enhancements are required. If the implementation environment is strongly object-oriented there will be direct implementations for all of the model constructs. For example, in an object-oriented database system one can declare a class with attributes and methods directly and have long-term storage (or "persistence") automatically managed. In non-object environments it may be necessary to manually flatten class hierarchies and add manual implementations of the model features. Manual adjustments can be captured in an intermediate model of similar type. The steps of abstraction reduction depend on the environment. In a favorable implementation environment the model nearest to the client's domain can be implemented almost directly. In unfavorable environments we have no choice but to add additional layers of refinement.

UML

As object-oriented methods became popular in the 1990s, there emerged several distinctive styles of notation. These notations differed enough to make tools incompatible and automated translation difficult; but the notations did not capture fundamentally different concepts. The basic concepts of class, object, and relationship were present in all of them, with only slight notational differences. The differences were more in the additional views and how the parts were integrated. They also differed somewhat more fundamentally in their approach to the design process and which portions they chose to emphasize. For example, some of the object-oriented methods emphasized front-end problem analysis through use-cases. Others were more design oriented and focused on building information models after there was a well-understood problem statement.

Because the profusion of notations was not helpful to the community, there was some pressure to settle on a collective standard. This was done partially through several of the leading "gurus" of the different methods, all moving to work for one company (the Rational Corporation). The product of their collaboration, and a large standards effort, is the Unified Modeling Language[11] (UML). Because UML has successfully incorporated most of the best features of its roots, and has gained a fairly broad industry consensus, it is increasingly popular. Probably the most significant complaint about the UML is its complexity. It is certainly true that if you tried to model a system using all the parts of the UML the resulting model would be quite complex. But the content of the UML should not be confused with a process. A designer is no more compelled to use all the parts of the UML than is a writer compelled to use all the words in the English language. Of course, it isn't simple to figure out which parts should be used in any given situation, and it can take fairly deep knowledge of the UML to know how to ignore features.

The primary importance of UML is that it may lead to more broadly accepted standardization of software and systems engineering notations. The notations are fundamentally software-centric, but as the software fraction (measured as percentage of development effort) makes up the majority of a development effort, this will seem appropriate. The two viewpoints within UML, use-cases and class-object models, most commonly discussed are the two that are the most software-centric. There are several other views that are more clearly systems oriented.

The use-case view within UML has two parts, the textual use-cases and diagrams that show the relationships among use-cases and actors. The textual form of a use-case is not strictly defined. In general it is a narrative listing of messages that pass between an "actor," a system stakeholder, and the system. Thus a use-case, in its pure form, follows the definition of the systems boundary. The use-case diagram shows the relationships between actors and use-cases, including linkages among use-cases.

A simple form for a textual use-case has four required parts and a group of optional parts.* They are

* There are many different formats for use-cases in use. The forms described here are inspired by various UML documents, and Dr. Kevin Kreitman in private communication.

1. Title (preferably evocative)
2. Actors — A list
3. Purpose — What the actors accomplish through this use-case, why the actors use the system
4. Dialog — A step-by-step sequence of messages exchanged across the actor-system boundary. The use-case gives the normal sequence, alternative sequences (from errors or other choices) can be integrated into the use-case, given as different use-cases, or organized into the optional section
5. Optional material — Some useful adjuncts include type (such as essential, optional, phase X, etc.), an overview for a very complex use-case, and alternative paths

UML uses class-object models very similar to those described in the OMT section. The differences are primarily details of notation, such as the graphic element used to indicate a particular type of relationship. There is also a fairly complex set of textual notations for showing the components of the classes (data and methods). For example, there are textual indications for public, private, and virtual elements. The discussion of class-object notations in the OMT section gives the flavor of how a model of the same sort would work if written in UML.

UML does introduce some modeling elements not discussed to this point and of high interest to system architects. On the behavioral side, the UML defines sequence diagrams. A sequence diagram depicts both the pattern of message-passing among the system's objects, and the timing relationships. The sequence diagram is useful both for specification and for diagnosis. When the client has a complex legacy system with which the new system must interface, or when the client's problems are primarily expressed in terms of deficiencies in a current system, the sequence diagram is a method for visually presenting time relationships. This is often quite important in real-time software-intensive systems.

Another avenue for standardization in which UML might assist is in physical block diagrams. UML defines "deployment diagrams" to show how software objects relate to physical computing nodes. The diagram style for the nodes and their links could be enhanced suitably (as discussed with models of form) to handle systems-level concerns. This is a possible refinement in the next major revision to the UML, designated UML 2.0.

Performance integration: scheduling

One area of nonfunctional performance that is very important to software, and for which there is large body of science, is timing and scheduling. Real-time systems must perform their behaviors within a specified timeline. Absolute deadlines produce "hard real-time systems." More flexible deadlines produce "soft real-time systems." The question of whether or not a given software design will meet a set of deadlines has been extensively studied.* To integrate these timing considerations with the design requires integration of scheduling and scheduling analysis.

In spite of the extensive study, scheduling design is still at least partly art. Theoretical results yield scheduling and performance bounds and associated scheduling rules, but they can do so only for relatively simple systems. When system functions execute interchangeably on parallel processors, when run times are random, and when events requiring reaction occur randomly, there are no deducible, provably optimal solutions. Some measure of insight and heuristic guidance is needed to make the system both efficient and robust.

Integrated models for manufacturing systems

The domain of manufacturing systems contains nice examples of integrated models. The modeling method of Baudin[12] integrates four modeling components (data flow, data structure, physical manufacturing flow, and cash flow) into an interconnected model of the manufacturing process. Baudin further shows how this model can then be used to analyze production scheduling under different algorithms. The four parts of the core model are:

1. A data flow model using the notations of DeMarco and state transition models
2. A data model based on entity-relationship diagrams
3. A material flow model of the actual production process, the model of physical form, using ASME and Japanese notations
4. A funds flow model

These parts, which mostly use the same component models familiar from the previous discussion, form an integrated architect's tool kit for the manufacturing domain. They are shown in Figure 10.8. The data flow models are in the same fashion as the requirements model of Hatley/Pirbhai. The data model is more complex and uses basic object-oriented concepts. In the material flow model, the progression of removal of abstraction is taken to a logical conclusion. Because the physical architecture of manufacturing systems is restricted, the architecture model components are similarly restricted. Baudin incorporates, in fact exploits, the restricted physical structure of manufacturing systems by using a standardized notation for the physical or form model.

Baudin further integrates domain-specific performance and system models by considering the relationship to production planning in its several forms (MRP-II, OPT, JIT). As he shows, these formalisms can be usefully

* Stankovic, J. A., Spuri, M., Di Natale, M., and Buttazzo, G. C., Implications of classical scheduling results for real-time systems, *IEEE Computer*, p. 16, June 1995. This provides a good tutorial introduction to the basic results and a guide to the literature.

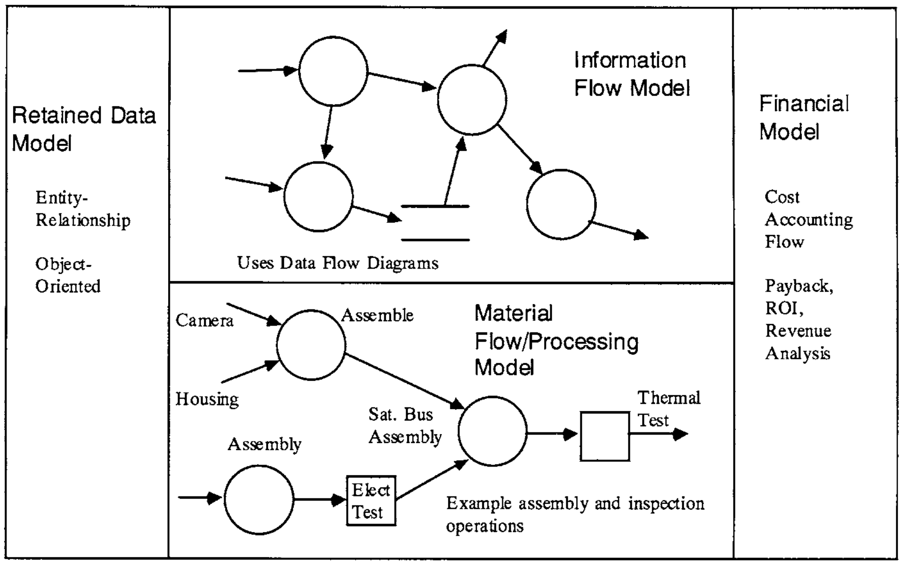


Figure 10.8

placed into context on the integrated models. In the terms used in Chapter 8, this is a form of performance model integration.

Integrated models for sociotechnical systems

On the surface, the modeling of sociotechnical systems is not greatly different from other systems, but the deeper reality is quite different. The physical structure of sociotechnical systems is the same as of other systems, though it spans a considerable range of abstraction, from the concrete and steel of transportation networks to the pure laws and policy of communication standards. But people and their behavior are inextricably part of sociotechnical systems. Sociotechnical system models must deal with the wide diversity of views and the tension between facts and perceptions as surely as they must deal with the physics of the physical systems.

Physical system representation is the same as in other domains. A civil transport system is modeled with transportation tools. A communications network is modeled with communications tools. If part of the system is an abstract set of laws or policies it can be modeled as proposed laws and policies. The fact that part of the system is abstract does not prevent its representation, but it does make understanding the interaction between the representation and the surrounding environment difficult. In general, modeling the physical component of sociotechnical systems does not present any insurmountable intellectual challenges. The unique complexity is in the interface to the humans who are components of the system, and in their joint behavior.

In purely technical systems the environment and the system interact, but it is uncommon to ascribe intelligent, much less purposively hostile, behavior to their environments.* But human systems constantly adapt. If an intelligent transport system unclogs highways, people may move farther away from work and re-clog the highways until a new equilibrium is reached. A complete model of sociotechnical system behavior must include the joint modeling of system and user behavior, including adaptive behavior on the part of the users.

This joint behavioral modeling is one area where modeling tools are lacking. The tools that are available fall into a few categories: econometrics, experimental microeconomics and equilibrium theory, law and economics, and general system dynamics. Other social science fields also provide guidance, but not generally descriptive and predictive behavior.

Econometrics provides models of large-scale economic activity as derived from past behavior. It is statistically based and usually operates by trying to discover models in these data rather than imposing models on them. In contrast, general system dynamics** builds dynamic models of social behavior by analysis of what linkages should be present and then tests their aggregated models against history. System dynamics attempts to find large-scale behavioral patterns that are robust to the quantitative details of the model internals. Econometrics tries to make better quantitative predictions without having an avenue to abstract larger-scale structural behavior.

Experimental economics and equilibrium theory try to discover and manipulate a population's behavior in markets through use of microeconomic theory. As a real example, groups have applied these methods to pricing strategies for pollution licenses. Instead of setting pollution regulations, economists have argued that licenses to pollute should be auctioned. This would provide control over the allowed pollution level (by the number of licenses issued) and be economically efficient. This strategy has been implemented in some markets and the strategies for conducting the auctions were tested by experimental groups beforehand. The object is to produce an auction system that results in stable equilibrium price for the licenses.

Law and economics is a branch of legal studies that applies micro- and macroeconomic principles to the analysis of legal and policy issues. It endeavors to assure economic efficiency in policies, and to find least cost strategies to fulfill political goals. Although the concepts have gained fairly wide acceptance, they are inherently limited to those policy areas where market distribution is considered politically acceptable.

* See Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991, chap. 9.

** An introductory reference on system dynamics is Wolstenholme, E. F., *System Enquiry: A System Dynamics Approach,* Wiley, Chichester, 1990, which explains the rationale, gives examples of application, and references the more detailed writings.

Conclusions

A variety of powerful integrated modeling methods already exist in large domains. These methods exhibit, more or less explicitly, the progressions of refinement and evaluation noted as the organizing principle of architecting. In some domains, such as software, the models are very well organized, cover a wide range of development projects, and include a full set of views. However, even in these domains the models are not in very wide use and have less than complete support from computer tools. In some domains, such as sociotechnical systems, the models are much more abstract and uncertain. But in these domains the abstraction of the models matches the relative abstraction of the problems (purposes) and the systems built to fulfill the purposes.

Exercises

1. For a system familiar to you, investigate the models commonly used to architecturally define such systems. Do these models cover all important views? How are the models integrated? Is it possible to trace the interaction of issues from one model to another?
2. Build an integrated model of a system familiar to you, covering at least three views. If the models in any view seem unsatisfactory, or integration is lacking, investigate other models for those views to see if they could be usefully applied.
3. Choose an implementation technology extensively used in a system familiar to you (software, board-level digital electronics, microwaves, or any other). What models are used to specify a system to be built? That is, what are the equivalents of buildable blueprints in this technology? What issues would be involved in scaling those models up one level of abstraction so they could be used to specify the system before implementation design?
4. What models are used to specify systems (again, familiar to you) to implementation designers? What transformations must be made on those models to specify an implementation? How can the two levels be better integrated?

Notes and references

1. White, S. et. al., Systems engineering of computer-based systems, *IEEE Computer,* p. 54, November 1993.
2. Maier, M. W., Quantitative engineering analysis with QFD, *Quality Eng.,* 7(4), 733, 1995.
3. Hauser, J. R. and Clausing, D., The house of quality, *Harvard Bus. Rev.,* 66(3), 63, May-June 1988.
4. The abovementioned 1995, as well as Maier, M. W., Integrated modeling: a unified approach to system engineering, *J. Syst. Software,* February 1996.

5. Yourdon, E. and Constantine, L. L., *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design,* Yourdon Press, Englewood Cliffs, NJ, 1979.
6. DeMarco, T., *Structured Analysis and System Specification,* Yourdon Press, Englewood Cliffs, NJ, 1979.
7. DeMarco, T., *Controlling Software Projects,* Yourdon Press, Englewood Cliffs, NJ, 1982.
8. *ADARTS Guidebook*, SPC-94040-CMC, Version 2.00.13, Vols. 1-2, September, 1991. Available through the Software Productivity Consortium, Herndon, VA.
9. Ward, P. T. and Mellor, S. J., *Structured Development for Real-Time Systems, Vol. 1-3,* Yourdon Press, Englewood Cliffs, NJ, 1985.
10. Rumbaugh, J. et. al., *Object-Oriented Modeling and Design,* Prentice-Hall, Englewood Cliffs, NJ, 1991.
11. There is a great deal of published material on UML. The Rational Corporation Web site (http://www.rational.com/) has online copies of the basic language guides, including the language reference and user guides.
12. Baudin, M., *Manufacturing Systems Analysis,* Yourdon Press Computing Series, Englewood Cliffs, NJ, 1990.

chapter eleven

Architecture frameworks

Introduction

Architecture has become a popular term in systems and software circles. There are now many "Architecture Frameworks," "Architecture Process Models," and the like to be found in government and corporate documents. The purpose of this book is not to provide a comprehensive tour of all of these approaches. Any such tour would be out of date the day after it was written, and most of it would be of little interest to most readers. Nevertheless, some review of community approaches is essential, especially those which have advanced to official standards. The practices advocated by these standards, for good or ill, have to be recognized. And we, the authors of this book, need to compare and contrast the approach here with what is in the standards.

The standards that concern architecture, and that have advanced to official status, are primarily architecture description standards. These are often referred to as Architecture Frameworks. Architecture frameworks are standards for the description of architectures. Architecture frameworks are analogous to blueprint standards or, more loosely, building codes. The analogy to building codes is very loose as much of what building codes define, specific physical forms, is not what most architecture frameworks define. A framework defines what products the architect must deliver (to the client or to some other agency with authority) and how those products must be constructed. The framework generally does not constrain the contents of any of those products, although such constraints could be incorporated.

Architecture frameworks serve much the same purposes as blueprint standards, although their developers have had additional purposes in mind as well. It is hoped these standards will improve the quality of architecting efforts by institutionalizing best practices and fostering communication about architectures through standardizing languages. Standardized architectural description languages may also facilitate architecture evaluation by standardizing the elements that must be considered in the evaluation.

Defining an architecture framework

To evaluate an architecture description framework we need to understand its goals, its definition of "architectural level" (as opposed to other design levels), and its organizing concepts. We treat each of these in turn, although the description concepts (viewpoints and views) have largely been discussed previously in Chapter 8, and are discussed in more detail in Appendix C.

Goals of the framework

Each group developing and promulgating a standard has asserted different goals, but they generally fall into a few, common categories.

1. Codify best practices for architectural description and, by so doing, improve the state of the practice
2. Ensure that the sponsors of the framework receive architectural information in the format they desire
3. Facilitate comparative evaluation of architectures through standardization of their means of description
4. Improve the productivity of development teams by presenting basic designs in a standard way
5. Improve interoperability of information systems by requiring that interoperation critical elements be described, and be described in a common way

The fairest way of evaluating different frameworks is against their own goals. Of course, the goals of any given framework may or may not match the goals of a project or adopting organization. To the extent they don't match it is poor choice, perhaps a poor "architectural" choice, on the part of the adopters or the framework developers.

Understanding "architectural level"

An architecture framework specifies information about architectures, as opposed to about detailed design, program management, or some other set of concerns. Thus, a framework needs to distinguish what information is "architectural" as opposed to something else. In this book the separation is connected to purpose. Information is architectural if it is needed to resolve the purposes of the client. The distinction is pragmatic, not theoretical. Moreover, we recognize that architecting and engineering are on a continuum of practice and sharp distinctions cannot be drawn. Other frameworks take different positions.

Organization of an architecture description framework

The architecture frameworks described here use a few basic concepts, though they use them differently and sometimes with different terminology. All of them organize architecture descriptions into collections of related models. We call a collection of logically related models a "view," as do several of the frameworks. Which views they choose and which modeling languages they require for specifying a view are different.

All of the frameworks have views that capture the high-level design of the system. High-level design means the physical implementation components (whether hardware or software) at the upper parts of a system/subsystem hierarchy.

All of the frameworks make some effort to put the client/user/customer in the architecture description, although they do so to different degrees. Some of the frameworks also explicitly pull the sponsor of the architecture (often not the user or customer) into the description. The more architecture description languages for software are academically oriented, usually the more strongly they are biased toward the design of the implementation.

The frameworks differ in how strongly they address consistency among the views. Some are informal and make no explicit provision, while others have built consistency checks into their choice of language.

Current architecture frameworks

Several standards explicitly labeled architecture frameworks have emerged in the 1990s. These frameworks are targeted at computer-based and information technology systems rather than more general systems. The three standards we consider here are the U. S. Department of Defense (DoD) C4ISR architecture framework, the International Standards Organization's RM-ODP standard, and the IEEE's P1471 Recommended Practice for Architectural Description. All three use the basic concepts given above, but take quite different approaches to the formalization and rigor required. We also discuss, very briefly, commercial information technology frameworks.

U.S. DoD C4ISR

In the early 1990s the DoD undertook the development of an architecture framework for Command, Control, Communications, Computing, Intelligence, Surveillance, and Reconnaissance (C4ISR) systems. The stated goal for this project was to improve interoperability across commands, services, and agencies by standardizing how architectures of C4ISR systems are represented. It also became a response to U. S. Congressional requirements for reform in how information technology systems are acquired.

The Architecture Working Group (AWG) published a version 1.0 of the framework in June 1996. This was followed by a version 2.0 document in December 1997. The version 2.0 document was published and is available through the U.S. DoD World Wide Web sites. The 2.0 framework requires that the architecture description be organized into summary information and three "architecture views." The three views are called the Operational Architecture View, the System Architecture View, and the Technical Architecture View. These are often contracted in discussion to the Operational Architecture, the System Architecture, and the Technical Architecture. Speaking of the "operational view of the architecture" would really be more in line with the notion of view and more consistent with the C4ISR Framework Technology. The version 2.0 document is commonly referred to as the C4ISR Architecture Framework (CAF).

Each view within the framework is composed of required (called "essential") and optional elements. A conformant description is obliged to provide all of the essential elements, and may pick from the optional elements as seen fit by the architects. Each of the essential and optional elements is a modeling method.

The CAF is a blueprint standard in that it defines how to represent a system's architecture, but it does not restrict the contents of the architecture. In practice, however, content restrictions have appeared with the CAF. The U.S. DoD has also sponsored the development of the Joint Technical Architecture (JTA). The JTA is a particular instance of a CAF technical architecture view. The JTA chooses particular standards. It has been current policy that all DoD systems should be compliant with the JTA, hence their technical architecture view is constrained to be that of the JTA. However, the JTA has also been extended with various annexes that cover specific military domains. There is also, at the time of this writing, an effort to develop a Joint Operational Architecture (JOA) that would further constrain the contents of a CAF conformant description.

Summary information

The required summary information is denoted AV-1 Overview and Summary Information and AV-2 Integrated Dictionary. Both are simple, textual objects. The first is information on scope, purpose, intended users, findings, etc. The second is definitions of all terms used in the description.

Operational view

The operational view shows how military operations are carried out through the exchange of information. It is defined as a description of task and activities, operation elements, and information flows integrated to accomplish support military operations. It has three required elements and six optional elements. Each of the elements is a model written in a particular modeling language. None of the languages is defined very formally. Some are entirely informal, as in the required High Level Concept Graphic (OV-1), while others (such as the Logical Data Model, OV-7) suggest the use of more formalized notations, though they do not require it. The essential products are

High-Level Operational Concept Graphic (OV-1): A relatively unstructured graphical description of all aspects of the systems operation, including organizations, missions, geographic configuration, and connectivity. The rules for composing this are loose with no real requirements.

Operational Node Connectivity Description (OV-2): Defines the operational nodes, and activities at each node, and the information flows between nodes. The rules for composing this are more structured than for OV-1, but are still loose.

Operational Information Exchange Matrix (OV-3): A matrix description of the information flows among nodes. This is normally done as an augmented form of data dictionary table.

The optional products are

Command Relationships Model (OV-4): A modestly structured model of command relationships.

Activity Model (OV-5): Very similar to a data flow diagram for operational activities.

Operational Rules Model (OV-6a): Defines the sequencing and timing of activities and information exchange through textual rules.

Operational State Transition Model (OV-6b): Defines the sequencing and timing of activities and information exchange through a state transition model, which is usually quite formal.

Operational Event/Trace Description (OV-6a): Defines the sequencing and timing of activities and information exchange through scenarios or use-cases. This is behavioral specification by example, as discussed in Chapter 8.

Logical Data Model (OV-7): Usually a class-object model or other type of relational data model. No specific notation is required, but most of the popular notations used are fairly formal. Defines the data requirements and relationships.

System view

The system view is defined as a description, including graphics, of a system and interconnections providing for, and supporting, warfighting functions. The system view is described with one essential product and twelve optional products. As with the operational view, these modeling methods span a wide range of methods and concerns. The one required element is

System Interface Description (SV-1): This model identifies the system's physical nodes and their interconnections. It is similar to an architecture interconnection diagram in the Hatley / Pirbhai sense as described in Chapters 8 and 10. A graphic representation method is called out in the CAF, but is not formally defined.

The twelve optional supporting products are mostly concerned with more detailed descriptions of system-level data interchange or operation. However, some of the supporting products wander very far afield from these concerns. For example, the System Technology Forecast (SV-9) is a tabular compilation of the technologies expected to be available, broken out by time frame, for the system.

Technical view

The technical view is defined as a minimal set of rules governing the arrangement, interaction, and interdependence of system parts or elements, whose purpose is to ensure that a conformant system satisfies a specified set of requirements. It has two elements, one required and one optional or supporting.

The required element is

Technical Architecture Profile (TV-1): A listing of the standards mandatory for the system being described. In most applications of the CAF the technical architecture comes from the JTA, a standards profile intended for all U. S. DoD systems.

The supporting element is the Standards Technology Forecast (TV-2), a projection of how standards and products compliant with standards will emerge during the time the system is developed and operated.

Evaluation of the C4ISR framework

The CAF has only been published for a few years and is only beginning to generate a body of experience. One clear issue is that it is often being used for purposes for which it was not intended. Recall that the purpose was primarily to facilitate interoperability through commonality of description. The goals, at least as discussed in the CAF documentation, did not include defining a standard that was complete with respect to an acquirer's concerns. For example, there is no place in the views for performance models, cost models, or other management models. Yet all of those are clearly necessary when the client is an acquirer and must make acquisition decisions.

Likewise, the CAF does not contain the elements necessary to cover the concerns of a builder of, for example, software-intensive systems. We have a fairly good understanding of best practices in software architecture description, and those practices are not mirrored in the CAF; but the purpose of the CAF is not to provide complete architecture "blueprints" for building.

Just as an architecture must be fit for purpose, so must an architecture description framework. If the CAF is misused, the fault is much more in the misuser than in the framework. Nevertheless, it can be cited as a weakness of the CAF that its parts are very loosely related. Very disparate concerns and models are lumped together into the views. Neither intra-view nor inter-view consistency is addressed at all. The individual models are so loosely defined, especially in some of the required elements, that ostensibly compliant descriptions can be produced that will not come close to guaranteeing interoperability. Because the CAF adopts such a neutral stance to methodology, it cannot enforce stronger consistency and completeness checks. It is probably not possible to strengthen consistency and completeness properties without adopting much more formal modeling methods, which would negatively impact ongoing programs on which the CAF might be mandated.

ISO RM-ODP

The International Standards Organization (ISO) has also developed an architecture description framework known as Reference Model for Open Distributed Processing (RM-ODP).[1] As the name implies, RM-ODP is computation and software-centric. It addresses open distributed processing that is multivendor, multiorganization, heterogeneous computing systems whose processing nodes are spatially distributed. As defined in RM-ODP, a distributed system is generally characterized by one which is spatially distributed, does not have a global state or clock, may have individual node failures, and operates concurrently.

The scope of RM-ODP is larger than just architectural description. RM-ODP makes extensive normative statements about how systems should be described, but also goes on to specify functions they should provide, and structuring rules to provide those functions. The architecture concerns of RM-ODP include both description (through viewpoints) and the provision of what are considered critical functions, called "transparencies," in the RM-ODP model.

The RM-ODP defines*:

1. A division of an ODP system specification into viewpoints in order to simplify the description of complex systems
2. A set of general concepts for the expression of those viewpoint specifications
3. A model for an infrastructure supporting, through the provision of distribution transparencies, the general concepts it offers as specification tools
4. Principles for assessing conformance for ODP systems

This is certainly larger than just description of architectures. Points 1 and 2 of RM-ODP are our concern here. RM-ODP is much more strongly normative than the other architecture frameworks discussed in this chapter. It takes a more normative approach both because of the inclinations of the authors (and their beliefs about best practices) and because the domain of application is narrower. RM-ODP applies to a particular class of computing

* ISO/IEC 10746-1: 1995 DIS (E), p. 8.

systems (albeit a broad class) and it seeks to be both a consistent and complete approach to describing such systems.

The heart of RM-ODP regarding descriptions is its five normative viewpoints. RM-ODP uses viewpoint to mean essentially what view means here, although it also carries the meaning of a generic specification method to be applied to any system. The RM-ODP notion of viewpoint is really a mixture of the language specification, the concerns covered, and the actual model instances for a particular system. The five ODP viewpoints are enterprise, information, computational, engineering, and technology. ODP adopts the notion that each viewpoint is a "projection" of the system's whole specification onto some set of concerns (using a specific language). The five viewpoints are chosen to be complete with respect to the concerns assumed to be relevant for an open distributed processing system. The definitions of the five viewpoints are*

1. An **enterprise** specification of an ODP system is a model of the system and the environment with which the system interacts. It covers the role of the system in the business, and the human user roles and business policies related to the system. The enterprise viewpoint is a viewpoint on the system and its environment that focuses on the purpose, scope, and policies of the system.
2. The **information** specification of an ODP system is a model of the information that it holds, and of the information processing that it carries out. The information model is extracted from the individual components and provides a consistent common view which can be referenced by the specifications of information sources and sinks, and the information flows between them. The information viewpoint on the system and its environment focuses on the semantics of the information and information processing performed.
3. The **computational** specification of an ODP system is a model of the system in terms of the individual, logical components which are sources and sinks of information. Using the computational language, computational specifications can express the requirements of the full range of distributed systems, providing the maximum potential for portability and interworking, and enabling the definition of constraints on distribution while not specifying the detailed mechanisms involved. The computational viewpoint is a viewpoint on the system and its environment that enables distribution through functional decomposition of the system into objects which interact at interfaces.
4. The **engineering** specification of an ODP system defines a networked computing infrastructure that supports the system structure defined in the computational specification and provides the distribution transparencies that it identifies. It describes mechanisms corresponding to the elements of the programming model, effectively defining an ab-

* ISO/IEC 10746-1: 1995 DIS (E), pp. 8-9, 16.

stract machine that can carry out the computational actions and the provision of the various transparencies needed to support distribution. The engineering viewpoint is a viewpoint on the system and its environment that focuses on the mechanisms and functions required to support distributed interaction between objects in the system.

5. The **technology** specification defines how a system is structured in terms of hardware and software components. The technology viewpoint is a viewpoint on the system and its environment that focuses on the choice of technology in that system.

Each viewpoint has a language associated with it and defined in the RM-ODP standard. The language specification in the standard is less specific than a typical programming language. The language specification consists of the definitions of the terms used to compose the language and constraints on constructing statements. All terms and constructions are in-built on object modeling concepts. The RM-ODP standard uses OMT conventions, although they could as easily be transferred to UML. Because RM-ODP is a component of the Object Management Group (OMG) of standards (UML is also a prominent component), such a transfer is already under way. ISO/IEC 10746-4 has mappings between the viewpoint language concepts and mathematically based formal languages from computer science.

RM-ODP recognizes the problem of inter-view and inter-view consistency. A conformant description must perform a number of cross-view checks for consistency. These checks are not a true guarantee, and the models involved don't have a precise notion of consistency built in, but the checks serve as an explicit attempt to look for inconsistencies.

Proprietary and semi-open information technology standards

Architecture has been widely addressed through proprietary and semi-open standards in information technology. Many firms have architectural standards, and many have developed their own description standards, typically as part of a development process standardization activity. The architectural description standards are tied to making specific go-ahead decisions about system development. Standardization of description products helps make those go-ahead decisions more consistent and facilitates process standardization.

One of the more widely known architecture description frameworks in information technology is usually called the Zachman framework, after the name of the author. The Zachman framework is not fixed as it has evolved with Zachman's writings. There are a number of similarities between the various Zachman frameworks and the RM-ODP standard as some of Zachman's early work popularized some notions of viewpoints and viewpoint languages. More recent published works by Zachman have added many more views than five, and have particularly emphasized the enterprise and business management aspects of choosing information technology architectures.

IEEE 1471

In April 1995 the IEEE Software Engineering Standards Committee (SESC) convened an Architecture Planning Group (APG) to study the development of an architecture standard for software-intensive systems. After publication of their report, the APG upgraded to the Architecture Working Group and was charged with the development of a Recommended Practice for architectural description — a particular type of standard. A Recommended Practice is one form of standard commonly used for relatively immature fields as it provides more general guidance rather than normative requirements. After extended debate and community consensus-building, a Recommended Practice for architecture description was published.*

The 1471 project was intended to codify the areas of community consensus on architecture description. Originally, it was envisioned that the standard would codify the notion of view and prescribe the use of particular views. In the end, consensus only developed around a framework of views and viewpoints and an organizing structure for architecture descriptions, but there was no prescription of any particular views. As a recommended practice it is assumed that community experience will eventually lead to greater detail within the standard.

P1471 concepts

Because the ontology of P1471 is independent of a specific framework of views and viewpoints, its ideas have been threaded into the discussion of this book. P1471 codifies the structure of an architecture description and the definitions of its parts. The terminology of P1471 is shown in Figure 11.1. The diagram is written in UML, but it can be easily interpreted, even without knowledge of UML. In the P1471 ontology every system has one architecture. That architecture can have several architecture descriptions. This expresses the idea that an architecture is a conceptual property of a thing, while an architecture description is a representation of the conceptual object. Several mutually consistent representations of a thing can exist so we need not specify that there be a single representation. P1471 does not make a distinction between types of system, so the relationship of architecture and architecture description could hold for an individual system, a family of systems, a system of systems, or a subsystem.

Returning to Figure 11.1, an architecture description is composed of stakeholders, concerns, viewpoints, views, and models. Stakeholders have concerns. Viewpoints cover stakeholders and concerns by their choice of language with which to represent the system. Views are groups of models

* IEEE P1471 Recommended Practice for Architecture Description, 2000. P1471 is part of the Computer Society's software engineering standards set.

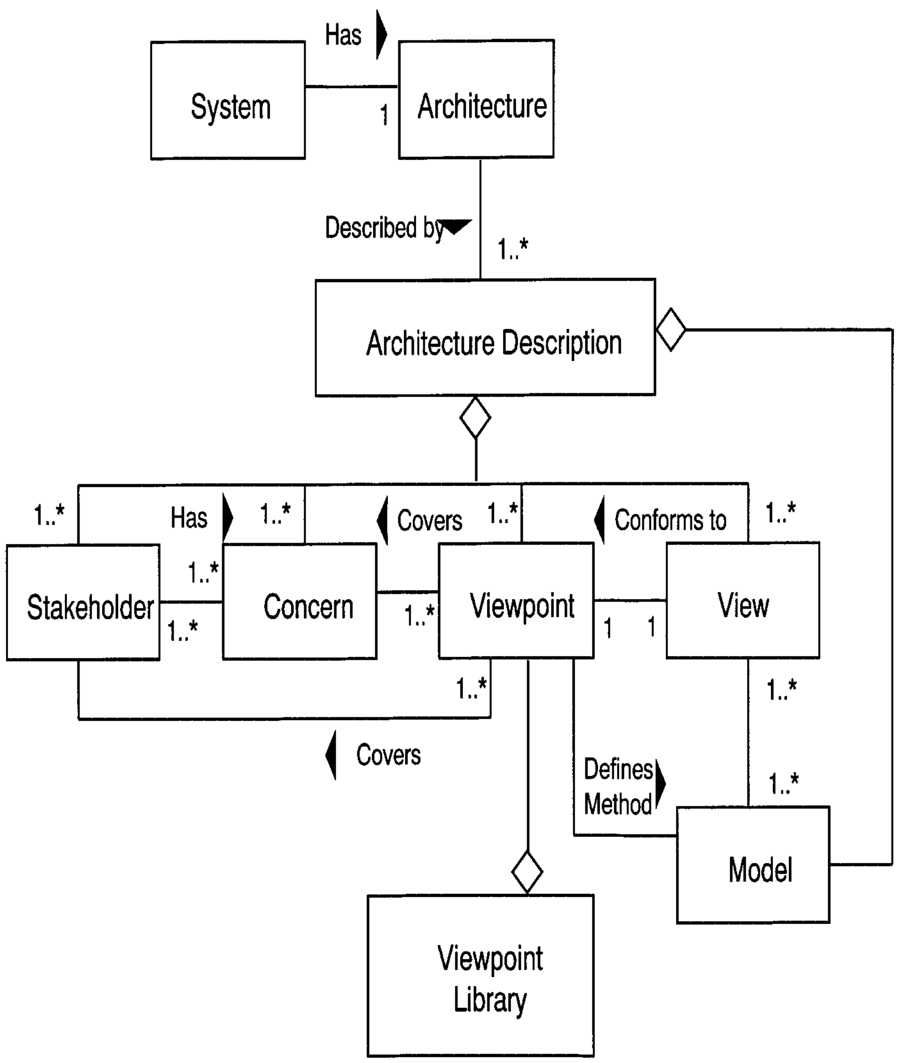


Figure 11.1

which must conform to exactly one viewpoint by using its language and rules.

Viewpoints may be drawn from a viewpoint library. A viewpoint library is not required by P1471, but it is expected in organizations that frequently develop architectural descriptions.

P1471 makes an explicit distinction between the concepts of viewpoint and view, although they are combined in the CAF and RM-ODP. If our goal is simply to write an architecture description, or to form a single standard, it is not necessary to separate the concepts. It is necessary to separate the concepts in P1471 because P1471 may be used to form many standards.

Viewpoints are the vehicle for forming a standard. Indeed, viewpoints may be placed into a library to be drawn from at the discretion of the architect and the specific stakeholders involved in a particular project. Thus, these organizing elements must be separately named and defined to allow them to be separately assembled for the needs of different sets of clients.

The viewpoints of RM-ODP are examples of P1471 compliant viewpoints. The only distinction between the viewpoint concept in RM-ODP and P1471 is that the RM-ODP version combines the abstraction (the P1471 viewpoint) and the actual instance of a representation of a particular system (the P1471 view).

P1471 normative requirements

The normative requirements of P1471 are limited, particularly compared to RM-ODP, and even the CAF. An architecture description conformant to P1471[2] must meet the following main requirements:

1. The stakeholders identified must include users, acquirers, developers, and maintainers of the system.
2. The architecture description must define its viewpoints, with some specific elements required.
3. The system's architecture must be documented in a set of views in one-to-one correspondence with the selected viewpoints, and each view must be conformant to the requirements of its associated viewpoint.
4. The architecture description document must include any known inter-view inconsistencies and a rationale for the selection of the described architecture.

There are a variety of other relatively minor normative requirements, along with various recommendations. Many of these are to make P1471 consistent with other IEEE software engineering standards, notably the overarching software engineering standard 12207.

Research directions

The current state of the art and practice in architecture description leaves much work undone. As the RM-ODP example shows, the basic architectural concepts of viewpoint, view, stakeholder, and concern can be extensively refined and tied to modeling formalisms if the domain of application is narrowed. A cost is the intellectual complexity of the resulting methods. RM-ODP is a complex standard. Its conceptual makeup is not complex in comparison to other areas in computer science, but it is quite complex compared to common practice in information technology. There may be strong benefits in mastering the complexity, but it acts as a barrier to the adoption of this technology. To make it more widely used we need better tools and better explanation and training mechanisms to pass these ideas on to the professional community.

As we move to more general systems, the range of formalized models drops off dramatically. It seems very unlikely that we can develop a really general architecture framework that will simultaneously be formalizable. It seems more likely that we must continue to work up from the engineering disciplines to create more general notations. One problem will be dealing with the disjunction between models common in the hardware (and some of the systems engineering) world and those coming from computer science and software. The hardware models are typically performance-centric and physics based. They work from physical objects. The computer science models are now commonly based on object-orientation and encapsulation of functionality within data models. It is not obvious how these will be reconciled, or to what extent it will be necessary. It may be a better approach to leave the modeling techniques as they are — taking the modeling techniques as they have been validated within the engineering disciplines. The higher-level challenge will then be to develop inter-view consistency checking techniques that don't require the disciplinary modeling methods to be altered, instead of working with them as they are.

Conclusions

The problem of blueprint standards for complex systems architectures has yielded a number of prototypical architecture frameworks. None of them is an ideal solution, but all contain important ideas. The architect faced with a normative requirement to use one of the frameworks must keep in mind their limitations and the architect's core role. The architect's core role is to assist the system's client in making the key technical decisions, particularly with which system concept to go on construction. This places a premium on models and methods that communicate effectively with the client, regardless of their correspondence (or lack thereof) to engineering models. Only as the architect's role evolves to transitioning the system to development and maintaining conceptual integrity during development does that correspondence to engineering methods become foremost.

Notes and references

1. ISO/IEC JTC1/SC21/WG7 Reference Model for Open Distributed Processing officially titled ITU-T X.901 ISO/IEC 10746 Reference Model, Parts 1-4.
2. The complete details are in the standard itself, IEEE P1471 Recommended Practice for Architectural Description, issued in 2000.

Part four

The systems architecting profession

The first three parts of this book have been about systems architecting as an activity or as a role in systems development. This part is about systems architecting as a profession; that is, as a recognized vocation in a specialized field. Two factors are addressed here. The first, the political process,* is important because it interacts strongly with the architecting process, directly affecting the missions and designs of large-scale, complex systems. The second, the professionalization of systems architecting, is important because it affects how the government, industry, and academia perceive systems architects as a group.

Chapter 12 is based on a course originated and taught at the University of Southern California by Dr. Brenda Forman of USC and the Lockheed Martin Corporation. The chapter describes the political process of the American government and the heuristics that characterize it. The federal process, instead of company politics or executive decision making,** was chosen for the course and for this book on architecting for three reasons.

First, federal governments are major sponsors *and clients* of complex systems and their development. Second, the American federal political process is a well-documented, readily available open source for case studies to support the heuristics given here. And third, the process is assumed by far too many technologists to be uninformed, unprofessional, and self-serving. Nothing could be worse, less true, nor more damaging to a publicly supported system and its creators than acting under such assumptions. In actuality, the political process is the properly constituted and legal mechanism by which the general public expresses its judgments on the value to it of the goods and services that it needs. The fact that the process is time-consuming,

* By "political process," it is meant the art and science of government, especially the process by which it acquires large-scale, complex systems.

** Company politics were felt to be too company-specific, too little documented, and too arguable for credible heuristics. Executive decisions are the province of decision theory and are best considered in that context.

messy, litigious, not always fair to all, and certainly not always logical in a technical sense, is far more a consequence of inherent conflicts of interests and values of the general public than of base motives or intrigue of its representatives in government.

The point that has been made many times in this book is that value judgments must be made by the client, the individual or authority who pays the bills, and not by the architect. For public services in representative democratic countries, that client is represented by the legislative, and occasionally the judicial, branch of the government.* Chapter 12 states a number of heuristics, the "facts of life," if you will, describing how that client operates. In the political domain, those rules are as strong as any in the engineering world. The architect should treat them with at least as much respect as any engineering principle or heuristic. For example, one of the facts of life states:

The best engineering solutions are not necessarily the best political solutions.

Ignoring such a fact is as great a risk as ignoring a principle of mathematics or physics — one can make the wrong moves and get the wrong answers.

Chapter 13 addresses the challenge in the Preface to this book to *professionalize* the field; that is, to establish it as a profession** recognized by its peers and clients. In university terms, this means at least a graduate-level degree, specialized education, successful graduates, peer-reviewed publications, and university-level research. In industry terms, it means the existence of acknowledged experts and specialized techniques. Dr. Elliott Axelband*** reports on progress toward such professionalization by tracing the evolution of systems standards toward architectural guidelines, by describing architecture-related programs in the universities, and by indicating professional societies and publications in the field. Axelband concludes the book with an assessment of the profession and its likely future.

* In the U.S., the executive branch *implements* the value judgments made by the Congress unless the Congress expressly delegates certain ones to the executive branch.

** "Any occupation or vocation requiring training in the liberal arts or the sciences and advanced study in a specialized field." *Webster's II New Riverside University Dictionary*, Riverside Publishing Company, Boston, MA, 1984, p. 939.

*** Associate Dean, School of Engineering, University of Southern California and the director of the Systems Architecting and Engineering program. Dr. Axelband previously was an executive of the Hughes Aircraft Company until his retirement in early 1994.

chapter twelve

The political process and systems architecting*

Brenda Forman, Ph.D.

Introduction: the political challenge

The process of systems architecting requires two things above all others: value judgments by the client and technical choices by the architect. The political process is the way that the general public, when it is the end client, expresses its value judgments. High-tech, high-budget, high-visibility, publicly supported programs are therefore far more than engineering challenges; they are *political* challenges of the first magnitude. A program may have the technological potential of producing the most revolutionary weapon system since gunpowder, elegantly engineered and technologically superb, but if it is to have any real life expectancy or even birth, its managers must take its political element as seriously as any other element. It is not only possible but likely that the political process will not only drive such design factors as safety, security, produceability, quantity, and reliability, but may even influence the choice of technologies to be employed. The bottom line is:

If the politics don't fly, the system never will.

* This chapter is based on a course originated and taught at the University of Southern California by Dr. Forman, now with Lockheed Martin Corporation in Washington, D.C. As indicated in the Introduction to Part Four, the political process of the American federal government was chosen for the course for three reasons: it is the process by which the general public expresses its value judgments as a customer, it is well-documented and publicized, and it is seriously misunderstood by the engineering community, to the detriment of effective architecting.

Politics as a design factor

Politics is a determining design factor in today's high-tech engineering. Its rules and variables must be understood as clearly as stress analysis, electronics, or support requirements. However, its rules differ profoundly from those of Aristotelian logic. Its many variables can be bewildering in their complexity and often downright orneriness.

In addition to the formal political institutions of Congress and the White House, a program must deal with a political process that includes interagency rivalries, intra-agency tensions, dozens of lobbying groups, influential external technical review groups, powerful individuals both inside and outside of government, and, always and everywhere, the media.

These groups, organizations, institutions, and individuals interact in a process of great complexity. This confusing and at times chaotic activity, however, determines the budgetary funding levels that either enable the engineering design process to go forward or threatens outright concellation. More often of late, it directly affects the design in the form of detailed budget allocations, assignments of work, environmental impact statements, and the reporting risks or threats.

Understanding the political process and dealing successfully with it is therefore crucial to program success.

> Example: Perhaps no major program has seen as many cuts, stretchouts, reviews, mandated designs, and risk of cancellation as the planetary exploration program of the 1970s and 1980s. Much of the cause was the need to fund the much larger Shuttle program. For more than a decade there were no planetary launches and virtually no new starts. From year to year, changes were mandated in spacecraft design, the launch vehicles to be used, and even the planets and asteroids to be explored. The collateral damage to the planetary program of the Shuttle Challenger loss was enormous in delayed opportunities, redesigns, and wasted energy. Yet, the program was so engineered that it still produced a long series of dramatic successes, widely publicized and applauded, using spacecraft designed and launched many years before.

Begin by understanding that *power is very widely distributed in Washington.* There is no single, clear-cut locus of authority to which to turn for support for long-term, expensive programs. Instead, support must be continuously and repeatedly generated from widely varying groups, each of which may perceive the program's expected benefits in quite different ways, and many of whose interests may diverge rather sharply when the pressure is on.

> Example: The nation's space program is confronted with extraordinary tensions, none of which are resolvable by any single authority, agency, branch, or individual. There are tensions between civilian and military, between science and application, between manned and unmanned flight, between complete openness and the tightest security, between the military services, between NASA centers, and between the commercial and government sectors, to name a few. Typical of the contested issues are launch vehicle development, acquisition, and use; allocations of work to different sections of the country and the rest of the world; and the future direction of every program. No one, anywhere, has sufficient authority to resolve any of these tensions and issues, much less to resolve them all simultaneously.

This broad dispersion of power repeatedly confuses anyone expecting that somebody will really be in charge. Rather, the opposite is true — anything that happens in Washington is the result of dozens of political vectors, all pulling in different directions. Everything is the product of maneuver and compromise. When those fail, the result is policy paralysis, and, all too possibly, program cancellation by default or failure to act.

There are no clear-cut chains of command in the government. It is nothing like the military, or even a corporation. The process gets even more complicated because *power does not stay put* in Washington. Power relationships are constantly changing, sometimes quietly and gradually, at other times suddenly, under the impact of a major election or a domestic or international crisis. These shifts can alter the policy agenda, and therefore funding priorities, abruptly and with little advance warning. A prime example is the ever-changing contest over future defense spending levels in the wake of the welcomed end of the Cold War.

The entire process is far better understood in dynamic than in static terms. There is a continuous ebb and flow of power and influence between the Congress and the White House, among and within the rival agencies, and among ambitious individuals. And through it all, everyone is playing to the media, particularly to television, in efforts to change public perceptions, value judgments, and support.

The first skill to master

To deal effectively with this process, *the first skill to master is the ability to think in political terms.* And that requires understanding that the political process functions in terms of an entirely different logic system than the one in which scientists, engineers, and military officers are trained. Washington functions in terms of the logic of politics. It is a system every bit as rigorous in its way as any other, but its premises and rules are profoundly different. *It will therefore repeatedly arrive at conclusions quite different from those of engineering logic, but based on the same data.*

Scientists and engineers are trained to marshal their facts and proceed from them to proof. For them, proof is a matter of firm assumptions, accurate data, and logical deduction. Political thinking is structured entirely differently. It depends not on logical proof but on past experiences, negotiation, compromise, and perceptions. Proof is a matter of "having the votes." If a majority of votes can be mustered in Congress to pass a program budget, then, by definition, the program has been judged to be worthy, useful, and beneficial to the nation. If the program cannot, then no matter what its technological merits, the program will lose out to other programs which can.

Mustering the votes depends only in part on engineering or technological merit. These are always important, but getting the votes frequently depends as much or even more on a quite different value judgment — the program's benefits in terms of jobs and revenues among the Congressional districts.

> Example: After the Lockheed Corporation won NASA's Advanced Solid Rocket Motor (ASRM) program, the program found strong support in the Congress because Lockheed located its plant in the Mississippi district of the then Chairman of the House Appropriations Committee. Lockheed's factory was only partially built when the Chairman suffered a crippling stroke and was forced to retire from his Congressional duties. Shortly thereafter, the Congress, no longer obliged to the Chairman, reevaluated and then canceled the program.

In addition to the highest engineering skills, therefore, the successful architect-engineer must have at least a basic understanding of this political process. The alternative is to be repeatedly blindsided by political events and, worse yet, not even to comprehend why.

Heuristics in the political process: "the facts of life"

Following are some basic concepts for navigating these rocky rapids: The Facts of Life. They are often unpleasant for the dedicated engineer, but they are perilous to ignore. Understanding them, on the other hand, will go far in helping anticipate problems and cope more effectively with them. They are as follows and will be discussed in turn.

- Politics, not technology, sets the limits of what technology is allowed to achieve.
- Cost rules.
- A strong, coherent constituency is essential.
- Technical problems become political problems.
- The best engineering solutions are not necessarily the best political solutions.

Fact of life # 1

Politics, not technology, sets the limits of what technology is allowed to achieve.

If funding is unavailable for it, any program will die, and getting the funding — not to mention keeping it over time — is a political undertaking. Furthermore, funding — or rather, the lack of it — sets limits that are considerably narrower than what our technological and engineering capabilities could accomplish in a world without budgetary constraints. *Our technological reach increasingly exceeds our budgetary grasp.* This can be intensely frustrating to the creative engineer working on a good and promising program.

> Example: The space station program can trace its origins to the mid-1950s. By the early 1960s it was a preferred way station for traveling to and from the moon. But when, for reasons of launch vehicle size and schedule, the Apollo program chose a flight profile that bypassed any space station and elected instead a direct flight to lunar orbit, the space station concept went into limbo until the Apollo had successfully accomplished its mission. The question then was, what next in manned spaceflight? A favored concept was a manned space station as a waypoint to the moon and planets, built and supported by a shuttle vehicle to and from orbit. Technologically, the concept was feasible; some argued that it was easier than the lunar mission. Congress balked. The President was otherwise occupied. Finally, in 1972, the Shuttle was born as an overpromised, underbudgeted fleet, without a space station to serve. Architecturally speaking, major commitments and decisions were made before feasibility and desirability had been brought together in a consistent whole.

Fact of life #2

Cost rules.

High technology gets more expensive by the year. As a result, the only pockets deep enough to afford it are increasingly the government's.* The fundamental equation to remember is **Money = Politics.** While reviews and hearings will spend much time on presumably technical issues, the fundamental and absolutely determining consideration is always affordability; and **affordability is decided by whichever side has the most votes.**

Funding won in one year, moreover, does not stay won. Instead, it must be fought for afresh every year. With exceedingly few exceptions, no program in the entire federal budget is funded for more than one year at a time. Each and every year is therefore a new struggle to head off attackers who want the program's money spent somewhere else, to rally constituents, to persuade the waverers, and, if possible, to add new supporters.

This is an intense, continuous, and demanding process requiring huge amounts of time and energy. And after one year's budget is finally passed, the process starts all over again. There is always next year. Keeping a program "sold," in short, is a continuous political exercise, and like the heroine in the old movie serial, *The Perils of Pauline*, some programs at the ragged edge will have to be rescued from sudden death on a regular basis. Rescue, if and when, may be only partial — not every feature can or will be sustained. If one of the lost features is a *system* function, the end may be near.

> Example: After the Shuttle had become operational, the question again was, what next in manned spaceflight? Although a modestly capable space station had been successfully launched by a Saturn launch vehicle, the space station program had otherwise been shelved once the Shuttle began its resource-consuming development. With developmental skill again available, the space station concept was again brought forward. However, order-of-magnitude life-cycle cost estimates of the proposed program placed the cost at approximately that of the Apollo, which in 1990-decade dollars would have been about $100 billion — clearly too much for the size of the constituency it could command. The result has been an almost interminable series of designs and redesigns, all unaffordable as judged by Congressional votes. Even more serious, the cost requirement has resulted in a spiraling loss of

* The economic expansion through the end of the 1990s sets an interesting counterpoint. The government is less and less able to influence technology in certain areas, for example, computing, simply because the commercial market has become so large relative to the federal market. Similarly, some of the most ambitious space and launch ventures in the 1990s have been privately funded.

system functions, users, and supporters. Microgravity experiments, drug testing, on-board repair of on-orbit satellites, zero-g manufacturing, optical telescopes, animal experiments, military research and development — one after another had to be reduced to the point of lack of interest by potential users. A clearly implied initial purpose of the space station, to build one because the Soviet Union had one, was finally put to rest with the U.S. government's decision to bring Russia into a joint program with the U.S., Japan, Canada, and the European Space Agency. Another space station redesign (and probably a new launch program design) is likely; however, the uncertainty level in the program remains high, complicating that redesign. One apparent certainty: the U.S. Congress has made the value judgment that a yearly cap of U.S. $2.1 billion is all that a space station program is worth. The design must comply or risk cancellation. Cost rules.

Fact of life #3

A strong, coherent constituency is essential.

No program ever gets funded solely, or even primarily, on the basis of its technological merit or its engineering elegance. By and large, the Congress is not concerned with its technological or engineering content (unless, of course, those run into problems — see Fact of Life #4). Instead, program funding depends directly on the strength and staying power of its supporters, i.e., its *constituency.*

Constituents support programs for any number of reasons, from the concrete to the idealistic. At times, the reasons given by different supporters will even seem contradictory. From the direct experience of one of the authors, some advocates may support defense research programs because they are building capability; others because research in promising better systems in the future permits reduction, if not cancellation, of present production programs.

Example: The astonishing success of the V-22 tilt-rotor Osprey aircraft program in surviving four years of hostility during the 1988-1992 period is directly attributable to the strength of its constituency, one that embraced not merely its original Marine Corps constituency but other Armed Services as well — plus groups that see it as benefiting the environment (by diminishing airport congestion), as improving the balance of trade (by tapping a large export market), and as maintaining U.S. technological leadership in the aerospace arena.

Assembling the right constituency can be a delicate challenge because a constituency broad enough to win the necessary votes in Congress can also easily fall prey to internal divisions and conflicts. Such was the case for the Shuttle and is the case for the Space Station. The scientific community proved to be a poor constituency for major programs; the more fields that were brought in, the less the ability to agree on mission priorities. On the other hand, a tight, homogeneous constituency is probably too small to win the necessary votes. The superconducting supercollider proved to be such. The art of politics is to knit these diverse motivations together firmly enough to survive successive budget battles and keep the selected program funded. Generally speaking, satellites for national security purposes have succeeded in this political art. It can require the patience of a saint coupled with the wiliness of a Metternich, but such are the survival skills of politics.

Fact of life #4

Technical problems become political problems.

In a high-budget, high-technology, high-visibility program, **there is no such thing as a purely technical problem.** Program opponents will be constantly on the lookout for ammunition with which to attack the program, and technical problems are tailor-made to that end.

The problems will normally be reported in a timely fashion. As many programs have learned, **mistakes are understandable; failing to report them is inexcusable.** In any case, reviews are mandated by the Congress as a natural part of the program's funding legislation. Any program that is stretching the technological envelope will inevitably encounter technical difficulties at one stage or another. The political result is that "technical" reports inevitably become political documents as opponents berate and advocates defend the program for its real or perceived shortcomings.

Judicious damage prevention and control, therefore, are constantly required. Reports from prestigious scientific groups such as the NRC or DSB will routinely precipitate Congressional hearings in which hostile and friendly Congressmen will pit their respective expert witnesses against one another; the program's fate may then depend not only on the expertise, but on the political agility and articulateness of the supporting witnesses. Furthermore, while such hearings will spend much time on ostensibly technical issues, the fundamental and absolutely determining consideration is always affordability; and affordability is decided by whichever side has the most votes.

Examples: Decades-long developments are particularly prone to have their technical problems become political. Large investments have to be made each and every year before any useful systems appear. The widely reported technical difficulties of the B-1 and B-2 bombers, the C-17 cargo carrier, the Hubble telescope, and the Galileo Jupiter spacecraft became matters of public as well as legislative concern. The future of long-distance air cargo transport, of space exploration, and even of NASA are all brought up for debate and reconsideration every year. Architects, engineers, and program managers have good reason to be concerned.

Fact of life #5

The best engineering solutions are not necessarily the best political solutions.

Remember that we are dealing with two radically different logic systems here. *The requirements of political logic repeatedly run counter of those of engineering logic.* Take construction schedules, for example. In engineering terms, an optimum construction schedule is one that makes the best and most economical use of resources and time and yields the lowest unit cost. In political terms, the optimum construction schedule is the one that the political process decides is affordable in the current fiscal year. These two definitions routinely collide; the political definition always wins.

Example: NASA and other agencies often refer to what is called the program cost curve. It plots total cost of development and manufacture as a function of its duration as follows (Figure 12.1):

The foregoing example leads to another provisional heuristic:

With few exceptions, schedule delays are accepted grudgingly; cost overruns are not, and for good reason.

The reason is basic. A cost overrun, that is, an increase over budget in a given year, will force the client to take the excess from some other program and that is not only difficult to do, it is hard to explain to the blameless loser and to that program's supporters. Schedule delays mean postponing benefits at some future cost, neither of which affect anyone today.

Example: Shuttle cost overruns cost the unmanned space program and its scientific constituency two decades of unpostponeable opportunities, timely mission analyses, and individual careers based on presidentially supported, wide-consensus planning.

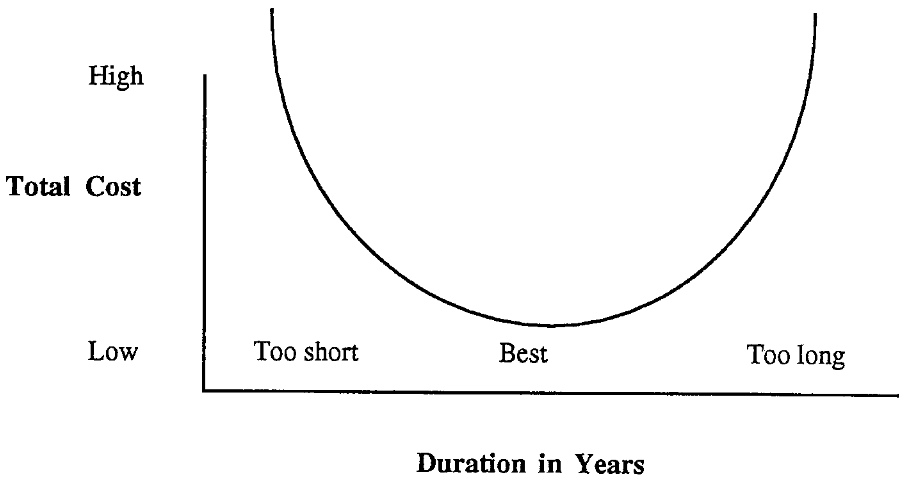


Figure 12.1 The curve is logical and almost always true. But it is irrelevant because the government functions on a cash-flow basis. Long-term savings will almost always be foregone in favor of minimizing immediate outlays. Overall life-cycle economies of scale will repeatedly be sacrificed in favor of slower acquisitions and program stretchouts because these require lower yearly appropriations, even if they cause higher unit costs and greater overall program expense. There is also the contradictory perception that if a given program is held to a tight, short, schedule it will cost less — facts not withstanding. (See Chapter 5, Social Systems, Facts vs. Perceptions.)

By the same token, a well-run program that sticks to budget can encounter very difficult technical problems and survive.

> Examples: Communication and surveillance satellite programs.

All in all, it can be a bewildering and intimidating process to the uninitiated; but it need not be so. Because, in addition to being confusing and chaotic, this is a profoundly interesting and engrossing process, every bit as challenging as the knottiest engineering problem. Indeed, it *is* an engineering challenge because it molds the context in which systems architecting and engineering must function.

The reader may well find the craziness of the political process distasteful, *but it will not go away.* The politically naive architect may experience more than a fair share of disillusion, bitterness, and failure. The politically astute program manager, on the other hand, will understand the political process, will have a strategy to accommodate it, and will counsel the architect accordingly. Some suggestions for the architect: it helps to document accurately when and why less-than-ideal technical decisions were made — and how to mitigate them later, if necessary. It helps to budget for contingencies and reasonably foreseeable risks. It helps to have stable and operationally useful interim configurations and fallback positions. It helps to acknowledge the client's right to have a change of mind or to make difficult choices without complaint from the supplier. Above all, it helps to acknowledge that living in the client's world can be painful, too. And finally, select a kit of prescriptions for the pain such as the following from Appendix A:

- The Triage: when there is only so much that can be done, let the dying die. Ignore those who will recover on their own. And treat only those who would die without help.
- The most important single element of success is to listen closely to what the customer (in this case, the Congress) perceives as his requirements and to have the will and ability to be responsive (J.E. Steiner, The Boeing Company, 1978)
- Look out for hidden agendas.
- And so on.

That doesn't mean that architects and engineers have to become expert lobbyists; but it does mean having an understanding of the political context within which programs must function, the budget battle's rules of engagement, and of those factors that are conducive to success or failure. The political process is not outside, it is *an essential element of,* the process of creating and building systems.

A few more skills to master

The following are a few more basic coping skills for the successful systems architect. First and foremost, *understand that the Congress and the political process are the owners of your project. They are the ultimate clients.* It is absolutely essential to deal with them accordingly by making sure they understand what you are trying to do, why it is important, and why it makes political sense for them to support you.

Be informed. This is your life, so be active. Learn the political process for yourself and keep track of what's going on. Figure out what information the political system needs in order to understand what the program needs and arrange to supply it to them. A chief engineer has utterly different information requirements than a Congressional oversight committee. Learn what sort of information furthers your program's fortunes in Washington and then get it to your program managers so they can get it to the political decisionmakers who determine your program's funding. Maybe your program has a great job-multiplier effect in some crucial lawmaker's district. Maybe its technology has some great potential commercial applications in areas where the U.S. is losing a competitive battle with another country.

The point is that *the political process bases its decisions on very different information than the engineering process does.* Learn to satisfy both those sets of requirements *by plan*.

Conclusion

The political process is a necessary element of the process of creating and building systems. It is not incomprehensible; it is different. Only when they are not understood do the political Facts of Life instill cynicism or a sense of powerlessness. Once understood, they become tools like any others in the hands of an astute architect. It is a compliment to the client to use them well.

chapter thirteen

The professionalization of systems architecting

Elliott Axelband, Ph.D.

Profession: Any occupation or vocation requiring training in the liberal arts or the sciences and advanced study in a specialized field.[1]

Introduction

To readers who have progressed this far, the existence of systems architecting as a *process*, regardless of who performs it, can be taken for granted. Functions and forms have to be matched, system integrity has to be maintained throughout development, and systems have to be certified for use. Visions have to be created, realized, and demonstrated.

This chapter, in contrast, covers the evolution of the systems architecting *profession*. An appropriate place to begin is with the history of the closely related profession of systems engineering, the field from which systems architecting evolved.

The profession of systems engineering

Systems engineering as a process began in the early 1900s in the communication and aircraft industries. It evolved rapidly during and after World War II as an important contributor to the development of large, innovative, and increasingly complex systems. By the early 1950s systems engineering had reached the status of a recognized, valued profession. Communication networks, ballistic missiles, radars, computers, and satellites were all recognized as systems. The "systems approach" entered into everyday language in many fields, social as well as technical. Government regulations and standards explicitly addressed systems issues and techniques. Thousands of engineers called systems engineering their vocation. Professional societies formed sections with journals devoted to systems and their development.[2] Universities established systems engineering departments or systems-oriented programs.* Books addressing the process, or aspects of it, started to appear.[3] Most recently, the profession became formally represented with the establishment of the International Council on Systems Engineering (INCOSE).[4]

The core of the systems approach from its beginnings has been the definition and management of system interfaces and tradeoffs, of which there can be hundreds in any one system. Systems analysis, systems integration, systems test, and computer-aided system design were progressively developed as powerful and successful problem-solving techniques. Some have become self-standing professions of their own under the rubric of systems engineering. Their academic, industrial, and governmental credentials are now well established.

All are science based; that is, based on measurables and a set of assumptions. In brief, these are that requirements and risks can be quantified, solutions can be optimized, and compliance specified. But these same assumptions are also constraints on the kinds of problems that can be solved. In particular, science-based systems engineering does not do well in problems that are abstract, data-deficient, perceptual, or for which the criteria are immeasurable.

For example, the meanings of such words as safe, survivable, affordable, reliable, acceptable, good, and bad are either outside the scope of systems engineering — "ask the client for some numbers" — or are force-fitted to it by subjective estimates. Yet these words are the language of the clients. Quantifying them can distort their inherent vagueness into an unintended constraint.

There is no group of professionals that better understands these difficulties than systems engineers and executives; nor who wish more to convert immeasurable factors to quantitatively stateable problems by whatever techniques can help. The first step they made was to recognize the nature of the problems. The second was to realize that almost all of them occur at the front (and back) ends of the engineering cycle. Consider the following descriptive heuristics, developed long ago from systems engineering experience:

- All the serious mistakes are made in the first day.
- Regardless of what has gone before, the acceptance criteria determine what is actually built.
- Reliability has to be designed in, not tested in.

* Among the best known are the University of Arizona at Tucson, Boston University, Carnegie Tech, Case Western Reserve, the University of Florida, Georgia Tech, the University of Maryland, George Mason University, Ohio State, MIT (Aerospace), New York Polytech, the University of Tel Aviv, the University of Southern California, Virginia Polytechnic Institute, and the University of Washington.

It is no coincidence that many systems engineers, logically enough, now consider systems architecting to be "the front end of systems engineering" and that architectures are "established structures." More precisely, systems architecting can be seen as setting up the necessary conditions for systems engineering and certifying its results. In short, systems architecting provides concepts for analysis and criteria for success. In evolving systems the functions of systems architecting, systems engineering, and disciplinary engineering are all more episodic. Concepts for analysis and criteria for success are established in early phases, but are revised with each new spiral through the development process. Systems engineers must control interfaces through many cycles of design, development, and integration, not just through one. In addition to conducting classical architecting episodically, the systems architect must also consider the issue of stable forms. The evolving system should not change everything on each cycle; it needs to retain stable substructures to evolve effectively. The definition of these substructures is part of the architect's role.

The immediate incentive for making architecting an explicit process, the necessary precursor to establishing it as a self-standing profession complementary to systems engineering, was the recognition in the late 1980s by systems executives that "something was missing" in systems development and acquisition, and that the omission was causing serious trouble: system rejection by users, loss of competitive bids to innovators, products stuck in unprofitable niches, military failures in joint operations, system overruns in cost and schedule, etc. — all traceable to their beginnings. Yet, there was a strong clue to the answer. Retrospective examinations of earlier, highly successful systems showed the existence in each case of an overarching vision, created and maintained by a very small group, which characterized the program from start to finish.[5]

Software engineers and their clients were among the first to recognize that the source of many of their software system problems were structural; that is, architectural.* Research in software architecture followed in such universities as Carnegie Mellon; the University of North Carolina, Chapel Hill; the University of California at Irvine; and the University of Southern California. Practitioners began identifying themselves as software architects and forming architectural teams. Communication, electronics, and aerospace systems architects followed shortly thereafter.

Societies established architecture working groups, notably the INCOSE Systems Architecting Working Group[6] and the IEEE Software Engineering Standards Committee's Architecture Working Group,[7] to formulate standard

* One of the earliest and most famous books on systems architecting is *The Mythical Man-Month, Essays on Software Engineering* by Frederick P. Brooks, Jr. (Addison-Wesley Publishing Company, 1974), which not only recognized the structural problems in software but, explicitly, on page 37, calls for a system architect, a select team, conceptual integrity, and for the architect to "confine himself scrupulously to architecture" and to stay clear of implementation. Brooks's analogy for the architectural team was a surgical team. He credits a 1971 Harlan Mills proposal as the source of these precepts.

definitions of terms and descriptions for systems and software architectures. These activities are essential to the development of a common internal language for systems architecting, and the integration of software architecture models and overall systems architectures in complex, software-intensive systems.

At the scale of the profession of engineering, the recognition that something was missing led to identifying it, by direct analogy with the processes of the classical architectural profession, as systems architecting.[8] Not surprisingly, the evolution of systems architecting tools was found to be already underway in model building, discussed in Part Three, and systems standards,* discussed in the next section.

Systems architecting and systems standards

Earlier chapters have pointed out that the abstract problems of the conceptual and certification phases require different tools from the analytic ones of system development, production, and test. One of the most important sets of tools is that of systems standards. Chapter 11 discussed one type of architecture standard: standards for architecture description. Here, we discuss a different category of standards, those that define development processes. For historical reasons, *architectural* process standards were not developed as a separate set. Instead, general systems process standards were developed which included systems architecting elements and principles understood at the time, most of them induced from lessons learned in individual programs. As will be seen, some key elements appeared as early as the 1950s.

Driven by much the same needs, the recognition of systems architecting in the late 1980s was paralleled, independently, by a recognition that existing systems standards needed to be modified or supplemented to respond to long-standing, systems-level structuring problems. Bringing the two tracks, architecting and standards, together should soon help both. Architecting can improve systems standards. Systems standards can provide valuable tools for the systems architecting profession.

Some of the earliest systems standards in which elements of systems architecting appeared were those of system specification, interface description, and interface management. They proliferated rapidly. A system specification can beget ten subsystem specifications, each of which is supported by ten lower-level subsystem specifications, etc. All of these had to be knitted together by a system of 100 or so interface specifications.

Even though modern computer tools (Computer-Assisted System Engineering or CASE tools) have been developed to help keep track of the

* Systems standards, for the purposes of this book, are those engineering standards having impact on the system as a whole, whether explicitly identified as such in their titles or not. They are a relatively small part of the totality of engineering standards. Many, if not most, are interface and test standards.

systems engineering process, extraordinarily disciplined efforts are still required to maintain the systems integrity.[9]

As systems complexity increased, systems engineers were faced with increasingly difficult tasks of assuring that the evolving form of the system met client needs, guaranteeing that tradeoffs maintained system intent in the face of complications arising during development, and finally assuring that the system was properly tested and certified for use. In due course, the proliferation of detailed specifications led to a need for overarching guidelines, an overview mechanism for "structuring" the complexity that had begun to obscure system intent and integrity.

Before continuing it should be pointed out that overarching guidelines are not, and cannot be, a *replacement* for quantitative system standards and specifications. The latter represent decades of corporate memory, measurable acceptance criteria, and certified practices. Guidelines — performance specifications, tailorable limits, heuristics, and the like — have a fundamental limitation. They cannot be certified by measurables. They are too "soft" and too prone to subjective perceptions to determine to the nearest few percentage points whether a system performs, or costs, what it should. At some point the system has to be measured if it is to be judged objectively.

From the standpoint of an architecting profession, the most important fact about system standards is that they are changing. To understand the trend, their development will be reviewed in some detail, recognizing that some of them are continuing to be updated and revised.

The origins of systems standards

The ballistic missile program of the 1950s

Urgent needs induce change and, eventually, improvement. The U.S./Soviet ballistic missile race begun in the mid-to-late 1950s brought about significant change as it led to the development and fielding of innovative and complex systems in an environment where national survival was threatened. To its credit, the U.S. Air Force recognized the urgent need to develop and manage the process of complex system evolution, and did so.* The response in the area of standards was the "375" System Standard, subsequently applied to the development of all new complex Air Force equipment and systems.

"375" required several things that are now commonplace in systems architecting and engineering. Timelines depicting the time-sequenced flow of system operation were to be used as a first step in system analysis.** From

* Those responsible for this development, Dr. Simon Ramo and General Bernard Schriever in particular, from time to time referred to their respective organizations as architects as well as system integrators. "Architecture," as a formalism, was largely bypassed in the urgency to build ballistic missiles as credible deterrents. Nonetheless, the essential "architectural" step of certification of readiness for launch was incorporated from the beginning and executed by all successor organizations. It became a centerpiece for the space launch programs of the 1960s and thereafter.

** This is not necessarily appropriate for all systems, but it was well suited to the missile, airplane, and weapon systems the Air Force had in mind at the time.

these, system functional block diagrams and functional requirements were to be derived as a basis for subsequent functional analysis and decomposition. The functional decomposition process in turn generated the subsystems that, with their connections and constraints, comprised the system and allowed the generation of subsystem requirements via tradeoff processes.

"375" was displaced in 1969 by a MILSTD 499 (Military Standard – 499)* which was applied throughout the Department of Defense (DoD). MILSTD 499A, an upgrade, was released in 1974 and was in effect for 20 years. MILSTD 499B, a later upgrade, was unofficially released in 1994, and it was almost immediately replaced by EIA/IS 632 Interim Systems Engineering Standard.[10]

The beginning of a new era of standards

The era of MILSTDs 499/499A/499B was an era in which military standards became increasingly detailed. It was not only these documents which governed system architecting and engineering, but they in turn referenced numerous other DoD MILSTDs, which addressed aspects of system engineering, and which were imposed on the military system engineering process as a consequence. To cite a few: MILSTD 490, Specification Practices, 1972; MILSTD 481A, Configuration Control – Engineering Changes, Deviations and Waivers (Short Form), 1972; MILSTD 1519, Test Requirements Document, 1977; and MILSTD 1543, Reliability Program Requirements for Space and Missile Systems, 1977. See Eisner, H., 1994[11] for additional examples.

This mindset changed with the end of the cold war in the late 1980s. Cost became an increasingly important decisive factor in competitions for military programs, supplanting performance, which had been the dominant factor in the prior era. Lowest cost, it was argued, could only be achieved if the restrictions of the military standards were muted. The detailed process ("how to") standards of the past, which specified how to conduct systems engineering and other program operations, needed to be replaced by standards that only provided guidelines,** leaving the engineering specifics to the proposing companies which would select these to be able to offer a low-cost product.[12] Further supporting this reasoning was the reality that the most sophisticated components and systems in fields such as electronic computer chips and computers were now available at low cost from commercial sources, whereas in the past the state of the art was only available from MILSTD-qualified sources. It was in this environment that EIA/IS 632 was born.

* The official form is "Mil. Std. – 499," but for ease of reading in a text an alternate form, "MILSTD 499," will be used here.

** As noted earlier, *replacement* is a questionable motivation for guidelines. Nonetheless, the establishment of a high-level guideline document — a key architecting technique — was a milestone.

EIA/IS 632, an architectural perspective

EIA/IS 632 is short by comparison with other military standards. Its main body is 36 pages. Including appendices, its total length is 60 pages, and these include several which have been left intentionally blank. And, most significantly, no other standards are referenced.

The scope and intent of the document is best conveyed by the following quotes from its contents:

- "The scope …. of systems engineering (activities) are defined in terms of what should be done, not how to do ... (them).." p. i
- "... (EIA/IS - 632) identifies and defines the systems engineering tasks that are generally applicable throughout the system life cycle for any program ...." p. 7

From a systems architecting perspective it is clear that the scope of the life cycle perspective includes the modern understanding of systems architecting.

One of the major activities of the systems architect, that of giving form to function, is addressed in pages 9–11. These pages summarize, in their own words and style, the client/architect relationship, the establishment of the defining system functions, the development of the system's architecture, and the process of allocating system functions to architectural elements via tradeoffs. By implication, the tradeoffs continue, with varying degrees of concentration, throughout the life cycle.

Curiously, test and validation are deferred to a later section entitled "4.0 Detailed Requirements." This is consistent with the historical organization of the preceding military standards, wherein Section 4 was dedicated to product assurance, a term which included a system test. It is, however, a significant departure from the systems architecting point of view. A basic tenet of systems architecting is that certification for use is one of its most important functions, and that this should be developed in parallel with, and as a part of, the development of a system's architecture. Consider, for example, some of the architecting heuristics that could apply:

- To be tested, a system must be designed to be tested.
- Regardless of what has gone before, the acceptance (and test) criteria determine what is actually built.

There are other sound and basic architecting principles which, suitably explained, could and should have been included as historically validated guiding principles in EIA/IS 632 which, in its own words, "provides guidance for the conduct of a systems engineering effort." Some applicable heuristics would include:

- Simplify. Simplify. Simplify.
- The greatest leverage in systems architecting is at the interfaces.

- Except for good and sufficient reasons, functional and physical structuring should match.
- In partitioning a system into subsystems, choose a configuration with minimal communications between subsystems.
- It is easier to match a system to the human one that supports it than the reverse.

Beyond these, the need for an unbiased agent, the system architect, to represent the client and technically guide the process is absent and is a serious omission.

Commercial standards

While EIA/IS 632 applies only to military systems engineering, that was not its original intent. The objective was to develop a universal standard for systems engineering, which would apply to both the military and commercial worlds and be ratified by all of industry. However, there was an urgency to publish a new military standard, and in the four-month schedule that was assigned only it could be developed. This led to two consequences. First, IEEE P1220,[13] a commercial systems engineering standard, was separately published. Second, the merging of EIA/IS 632 with IEEE P1220 to create the first universal standard for system engineering was planned for publication in 1997. The development of this universal system engineering standard involves personnel from several organizations including the American National Standards Institute (ANSI) and EIA, and is expected to be named EIA/ANSI 632.[14] Beyond that, the International Standards Organization (ISO) plans to issue a standard in 2000+ that will merge EIA/ANSI 632 with ISO 12207. This latter will be published soon and will provide a universal software engineering standard. In passing, software is included in the definition of a system as used in the prior mentioned systems engineering standards, but not with the same level of intensity as can be found in separate existing software engineering standards.

IEEE P 1220, an architectural perspective

The IEEE working group which generated P 1220 was sponsored by the IEEE Computer Society and included representatives from INCOSE, the EIA, and the IEEE AES Society. It is the first commercial standard to formally address systems engineering. P 1220's similarity with EIA/IS 632 derives from a fair degree of common authorship plus a deliberate effort to: (1) coordinate efforts in order to present a common view of systems engineering, and (2) anticipate the eventual merger of the two documents. While the similarities are therefore not surprising, there are significant differences that are worthy of mention.

To begin with similarities, both standards are guides and not "how to" instruction manuals. Both address the entire life cycle of a product. Both share a common architecture, addressing in similar ways things that are becoming similar: the processes of system engineering in the military and commercial environments. This extends to a fair degree of common vocabulary, although mercifully P 1220 is freer of acronyms.

Compared to EIA/IS 632, P 1220 is more complex and longer (58 pages in the body of the report vs. 36, and 66 pages overall vs. 60). It is much more rigorous in its definitions and use of system hierarchical structures. It has several significant differences that tend to favor the recognition and processes of systems architecting.

- The subsystems which comprise a system are understood and treated as systems. pp. 2, 4, 13
- The customer (client) is explicitly identified along with a need to determine and quantify his expectations. p. 35
- External constraints including public and political constraints are recognized as part of the process. p. 35
- The role of system boundaries and constraints in system evolution is considered. p. 36
- The need to evolve test plans with product evolution is expressed. pp. 18, 19
- The explicit need to generate functional and physical architectures is recognized, unfortunately (from a system architect's view) in the same section of the document which through usage defines system architecture as the sum of the product and its defining data package. p. A-3.

In summary, P 1220 better recognizes the systems architecting process than does EIA/IS 632. It does, however, have significant systems architecting shortfalls and would better serve as a systems engineering guide if the role of the systems architect were included, and if the architecting heuristics given in this chapter were added.

A continuing problem in all of these systems standards, highlighted in Chapter 6, is the difference in system/subsystem hierarchies across hardware and software. Both can be thought of hierarchically, but the hierarchical model for software is often changed to become layered, and the hierarchy of software units in a distributed system often does not match the associated hardware. This often leads to significant problems in development, and contributes to poorly structured software in systems where software development cost dominates total development cost. Standards for distributed system development, such as RM-ODP and UML, recognize the disjunction and allow the software and hardware elements to be represented in their own hierarchies. This frees the software architects from an imposed, and often damaging, hardware-based hierarchy, but introduces new problems in reconciling the two models to assure consistency. Engineering process standards have only begun to address this issue.

Company standards

Each company has its own set of standards and practices that incorporate unique core competencies, practices, and policies. These need to evolve for a company to improve its performance and competitive posture. Company standards serve two other functions: instructing its initiates and relating to its customers. The latter function is stimulated whenever customers change their standards, and it is from this perspective that the systems engineering standards of several companies were examined. This was not an easy task since systems architecting and engineering are viewed by those companies engaged in them as an enabler of efficient product generation, and, as such, applicable practices providing a competitive advantage are considered trade secrets.

Several generalizations are possible. Today's competitive pressures have caused self-examination and particularly reengineering to become a regular way of life. This has also been encouraged by popularized business literature.[15] Process, as opposed to product, is the focus of such institutionalized activity. In reviewing the process of product generation and support, systems architecting and, in some cases, systems architects are gaining recognition, although not always in a way clearly separated from systems engineering.

The Harris Corporation Information Systems Division recently culminated four years of activity by publishing their revised *Systems Engineering Guide Book*. A generalized description is provided in Honour, 1993.[16] The 128-page book is company proprietary. Discussions with its author, Eric Honour, indicated that while systems architecting is not delineated per se, the *processes* that constitute systems architecting account for approximately 25% of its pages.

Sarah Sheard and Elliot Margolis reported on the evolving systems engineering process within The Loral Federal Systems Organization.[17] Their conclusion is that there is an important relationship between the nature of a product and the team developing it, and that, as such, there is no one best organization for product development. However, their recent experience indicates success with a team structure that includes a distinct architecture team with a clearly identified chief architect working in conjunction with a management team and both software and hardware development teams. Their use of the terms architect, architectures, and architecting are consistent with those of this book.

Hughes Aircraft published its 3-inch thick *Systems Engineering Handbook* in 1994.[18] Its objective, stated on page P-1, is to ".... improve both the quality and efficiency of systems engineering at Hughes." The handbook describes the then applicable MILSTD 499B and the Hughes systems engineering processes for both DoD and non-DoD programs, including Hughes' organizational and other resources available to implement these. It is a very comprehensive, user-friendly book, clearly adapted from MILSTD 499B, and includes the activities of the systems architect — who is never identified by that name — within the framework of systems engineering. The systems engineering function and organization is identified as the technical lead organization for product development and is provided a unique identity in all forms of organization discussed: functional, projectized, and matrix. In that the handbook is patterned after MILSTD 499B which has a strong resemblance to EIA/IS 632, the comments made earlier with respect to EIA/IS 632 apply.

A summary of standards developments, 1950-1995

For a variety of reasons and by a number of routes, system standards and specifications are evolving consistent with the principles and techniques of systems architecting. The next step is the use of system architects to help improve systems standards, particularly in system conception, test, and certification. At the same time, improved systems standards can provide powerful tools for the systems architecting profession.

A cautionary note: the recent and understandable enthusiasm of the DoD to streamline standards and eliminate all references to prior MILSTDs could make systems architecting considerably more difficult. Useful as guidelines are, they are no substitute for quantitative standards for bidding purposes, for certifying a system for use, or for establishing responsibility and liability.* MILSTDs in many instances incorporate specific philosophical and quantitative requirements based on lessons dearly learned in the real world. They reduce uncertainty in areas that should not or need not be uncertain. To ignore these by omission is to run the risk of learning them all over again at great cost. To the extent that the lessons relearned are architectural, the risks can be enormous. As the heuristic states, all the serious mistakes are made on the first day.

Systems architecting graduate education

Systems engineering universities and systems architecting

Graduate education, advanced study, and research give a profession its character. They distinguish it from routine work by making it a vocation, a calling of the particularly qualified.

The first university to offer masters and doctorate degrees in systems engineering was the University of Arizona, beginning in 1961. The program began as a graduate program; an undergraduate program, and the addition of industrial engineering to the department title, came later. Still in existence, the graduate department has well over 1000 alumni.

The next to offer advanced degrees was the Virginia Institute of Technology in 1971, but not until after 1984 did additional universities join the systems engineering ranks. They included Boston University, George Mason University, the Massachusetts Institute of Technology (MIT), the University

* In this connection, the DoD has explicitly retained interface and certification standards as essential, not to be considered as candidates for elimination.

of Maryland, the University of Southern California (USC), the University of Tel Aviv, and the University of Washington. It is worth noting that all are located at major centers of industry or government, the principal clients and users of systems engineering.

To the best knowledge of the authors of this book, USC is the first to offer a graduate degree in systems architecting and engineering with the focus on systems architecting. However, of the universities offering graduate degrees in systems engineering, some half dozen now include systems achitecting within their curricula. Notable among them is the MIT Systems Design and Management (SDM) program. This program, which is intended as a new kind of graduate education program for technical professionals, is built on three core subjects: systems engineering, systems architecture, and project management. While the degree is not focused on systems architecting, that subject forms a major part of the curriculum's core. The MIT SDM curriculum is becoming more of a national model as it is spread through the Product Development in the 21st century (PD21) program. PD21 is creating programs that are very similar to MIT's SDM program in universities across the country. The current universities involved are the Rochester Institute of Technology, the University of Detroit-Mercy, and the Naval Postgraduate School.

Architecting is also becoming a strong interest in universities offering advanced degrees in computer science with specializations in software and computer architectures, notably, Carnegie Mellon University, the Universities of California at Berkeley and Irvine, and USC. At USC, the systems architecture and engineering degree began with an experimental course in 1989; it formally became a masters degree program in 1993 following its strong acceptance by students and industry.

In the last 10 years there has been growing recognition of the value of interdisciplinary programs, which of itself would favor systems architecting and engineering. These have been soul-searching years for industry, and the value of system architecting and engineering has become appreciated as a factor in achieving a competitive advantage. Also, the restructuring of industry has caused a rethinking of the university as a place to provide industry-specific education. These trends, augmented by the success of the system architecting and engineering education programs, have caused university architect-engineering programs to prosper.

The success of these programs can be measured in several ways. First, the direction is one of growth as 7 out of the 8 existing masters programs were started in the last 15 years; and the Universities of Maryland, Tel Aviv, and Southern California are all considering expanding their programs to include a Ph.D. Second, systems architecting and systems architecting education are making a positive difference in industry, as supported by industry surveys. In point of fact, company-sponsored systems architecting enrollments increased even when there was industrial contraction.

Curriculum design

It is not enough in establishing a profession to show that universities are interested in the subject. The practical question is what is actually taught; that is, the curriculum. Because USC is apparently the first university to offer an advanced degree specifically in systems architecture and engineering, its curriculum is described. It should be pointed out that this curriculum is at the graduate level. To date, no undergraduate degree is offered or planned.

The USC masters program admits students satisfying the School of Engineering's academic requirements and having a minimum of three years applicable industrial experience. Students propose a 10-course curriculum which is reviewed, modified if required, and accepted as part of their admission. The curriculum requires graduate-level courses as follows:

- An anchor systems architecting course
- An advanced engineering economics course
- One of several specified engineering design courses
- Two elective courses in technical management from a list of eleven which are offered
- One of eight general technical area elective courses
- Four courses from one of eleven identified technical specialization areas, each of which has six or more courses offered

The structure of this MS in Systems Architecture and Engineering curriculum has been designed based on both industrial and academic advice. Systems architecture is better taught in context. It is too much to generally expect a student to appreciate the subtleties of the subject without some experience; and the material is best understood through a familiar specialty area in which the student already practices. The three-year minimum experience requirement and the requirement of four courses in a technical specialty area derive from this reasoning.

The need for an anchor course is self-evident. Systems architecture derives from inductive and heuristic reasoning, unlike the deductive reasoning used in most other engineering courses. To fully appreciate this difference, the anchor course is taken early, if not first, in the sequence. The course contains no exams as such, but requires two professional quality reports so that the student can best experience the challenges of systems architecting and architecture by applying his knowledge in a dedicated and concentrated way.

Experience has shown that a design experience course, the advanced economics course, and courses in technical management are valuable to the system architect and, therefore, they are curriculum requirements. The additional course in a general technical area allows the student to select a course that most rounds out the student's academic experience. Possibilities include a systems architecting seminar, a course on decision support systems, and a course on the political process in systems architecture design.

To date within the USC program there have been 60 masters graduates. Over 300 have taken the anchor course, and 40 students are enrolled masters candidates. A library of over 150 research papers has been produced from the best of the student papers, and is available as a research library for those in the program. Several students are proceeding toward a Ph.D. in Systems Engineering with a specialty in Systems Architecting.

Advanced study in systems architecting

A major component of advanced study in any profession is graduate-level research and refereed publications at major universities. In systems architecting, advanced study can be divided into two relatively distinct parts: that of its science, closely related to that of systems engineering, and of its art. The universities committed to systems engineering education were given earlier. Advanced study in its art, though often illustrated by engineering examples, has many facets, including research in:

- Complexity, by Flood and Carson[19] at City University, London, England
- Problem solving, by Klir[20] at the State University of New York at Binghamton, and by Rubinstein[21] at the University of California at Los Angeles
- Systems and their modeling, by Churchman[22] at the University of California at Berkeley, and Nadler[23] and Rechtin[24] at the University of Southern California
- The behavioral theory of architecting, by Lang[25] at the University of Pennsylvania, Rowe[26] at Harvard, and Losk, Carpenter, Cureton, Geis, and Carpenter[27] at USC
- The practice of architecture by Alexander[28] and Kostof[29] at University of California at Berkeley
- Machine (artificial) intelligence and computer science by Genesereth and Nilsson[30] at Stanford, Newell[31] and Simon[32] at Carnegie Mellon University, and Brooks[33] at the University of North Carolina at Chapel Hill
- Software architecting by Garlan and Shaw[34] at Carnegie Mellon University, and Barry Boehm at USC

All have contributed basic architectural ideas to the field. Many are standard references for an increasing number of professional articles by a growing number of authors. Most deal explicitly with systems, architectures, and architects, although the practical art of systems architecting was seldom the primary motivation for the work. That situation predictably will change rapidly as both industry and government face international competition in a new era.

Professional societies and publications

Existing journals and societies were the initial professional media for the new fields of systems architecting and engineering. Since much early work was done in aerospace and defense, it is understandable that the IEEE Society on Systems Man and Cybernetics, the IEEE Aerospace Electronics Society, and the American Institute of Aeronautics and Astronautics and their journals, along with others, became the professional outlets for these fields. One excellent sample paper from this period (Boonton and Ramo, 1984[35]) explained the contributions that systems engineering had made to the U.S. ballistic missile program.

The situation changed in 1990 when the first National Council on System Engineering conference was held and attracted 100 engineers. INCOSE became the first professional society dedicated to systems engineering and soon established a Systems Architecting Working Group.

The society, with a current membership of 3500 (one third of which are outside the U.S.), publishes a quarterly newsletter and a journal. The journal first appeared in 1994 and it published jointly with The IEEE AES Society in 1996. Since then it has become a stand-alone, quarterly publication.

Conclusion: an assessment of the profession

The profession of systems architecting has come a long way, and its journey has just begun. Its present body of professionals in industry and academia — beginning most often in electronics, control and software systems, soon broadening into systems engineering — formed the core of small design teams and now consider themselves as architects. The profession has been nurtured within the framework of systems engineering, and no doubt will maintain a tight relationship with it. A masters-level university curricula now exists, and the material and ideas are suffusing into many other systems-oriented programs. Applicable research is underway in universities. Applicable standards and tools are being developed at the national level. It has an acknowledged home within INCOSE as well as other professional societies that, together with their publications, provide a medium for professional expression and development.

It is interesting to speculate on where the profession might be going and how it might get there. The cornerstone thought is that the future of a profession of systems architecting will be largely determined by the perceptions of its utility by its clients. If a profession is useful, it will be sponsored by them and prosper. To date, all indicators are positive.

Judging by the events that have led to its status today, and by comparable developments in the history of classical architecture, systems architecting could well evolve as a separate business entity. The future could hold more systems architecting firms that bid for the business of acting as the technical representative or agent of clients with their builders. There are related precedents today in Federally Funded Research and Development Centers (FFRDCs) and Systems Engineering and Test Assistance Contractors (SETACs), which are independent entities selected by the DoD to represent it with defense contractors that build end products. Similar precedents exist in NASA and the Department of Energy.

The role of graduate education is likely to grow and spread. Today's products and processes are more netted and interrelated than those of ten years ago, and tomorrow's will be even more so. System thinking is proving to be fundamental to commercial success, and systems architecting will increasingly become a crucial part of new product development. It is incumbent upon universities to capture the intellectual content of this phenomenon and embody it in their curricula. This will require a tight coupling with industry to be aware of important real-world problems, a dedication to research to provide some of the solutions, and an education program that trains students in relevant architectural thinking.

Publishe, peer-reviewed research has stood the test of time, providing the best medium for the rapid dissemination of state-of-the-art thinking. Today INCOSE's Systems Architecting Working Group provides one such outlet. Still others will be needed for further growth.

In summary, all the indicators point to a future of high promise and value to all stakeholders.

Notes and references

1. *Webster's II New Riverside University Dictionary,* Riverside Publishing, Boston, MA, 1984, p. 939.
2. *The Institute of Electrical and Electronic Engineers (IEEE) Transactions on Systems, Man, and Cybernetics,* Institute of Electrical and Electronic Engineers, New York; IEEE Transactions on Aerospace and Electronic Systems, Institute of Electrical and Electronic Engineers, New York; *J. Amer. Inst. Aeronautics Astronautics,* American Institute of Aeronautics and Astronautics, Washington, D.C.
3. Machol, R.E., *Systems Engineering Handbook,* McGraw-Hill, New York, 1965; Chestnut, H., *System Engineering Methods,* John Wiley & Sons, New York, 1967; Blanchard, B. S. and Fabrycky, *Systems Engineering and Analysis,* Prentice-Hall, Englewood Cliffs, NJ, 1981.
4. *Proc. First Annual Symp. Nat. Systems Engineering,* Seattle, WA, 1990; *J. Nat. Counc. Syst. Eng.,* Inaugural Issue, National Council on Systems Engineering, Seattle, WA, 1994; *Tools for System Engineering,* A Brochure. Ascent Logic Corporation, San Jose, CA, 1995.
5. Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991, 299.
6. See the Inaugural Issue of *Systems Engineering, J. Nat. Counc. Syst. Eng.,* 1(1), July/September 1994.
7. See "Toward a Recommended Practice for Architectural Description" of the IEEE SESC Architecture Planning Group, April 9, 1996.
8. Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991.

9. *Doors,* a brochure, Zycad Corporation, Freemont, CA, 1995; SEDA — System Engineering and Design Automation, a brochure, Nu Thena Systems, McLean, VA, 1995.
10. *EIA Standard IS - 632,* Electronic Industries Association Publication, Washington, D.C., 1994.
11. Eisner, H., *Computer-Aided Systems Engineering,* Prentice-Hall, Englewood Cliffs, NJ, 1988.
12. Perry, W. J., *Specifications and Standards — A New Way of Doing Business,* DODI 5000.2, Part 6, Section I, Department of Defense, Washington, D.C., 1995.
13. *IEEE P1220 Trail Use Standard for Application and Management of the Systems Engineering Process,* IEEE Standards Department, New York, 1994.
14. Martin, J. N., American national standard on systems engineering, *NCOSE INSIGHT, Issue 9,* International Council on Systems Engineering, Seattle, WA, September 1995.
15. Hammer, M. and Champy, J., *Reengineering the Corporation,* Harper, New York, 1994; Hamel, G. and Prahalad, C. K., *Competing for the Future,* Harvard Business School Press, Boston, MA, 1994.
16. Honour, E. C., TQM development of a systems engineering process, *Proc. Third Annual Symp. National Council Systems Engineering,* National Council on Systems Engineering, Sunnyvale, CA, 1993.
17. Sheard, S. A. and Margolis, E. M., Team structures for systems engineering in an IPT environment, *Proc. Fifth Annual Symp. National Council on Systems Engineering,* National Council on Systems Engineering, Sunnyvale, CA, 1995.
18. *Systems Engineering Handbook,* Hughes Aircraft Company, Culver City, CA, 1994.
19. Flood, R. L. and Carson, E. R., *Dealing with Complexity, An Introduction to the Theory and Application of Systems Science,* Plenum Press, New York, 1988.
20. Klir, G. J., *Architecture of Problem Solving,* Plenum Press, New York, 1988.
21. Rubinstein, M. F., *Patterns of Problem Solving,* Prentice-Hall, Englewood Cliffs, NJ, 1975.
22. Churchman, C. W., *The Design of Inquring Systems,* Basic Books, New York, 1971.
23. Nadler, G., *The Planning and Design Approach,* John Wiley & Sons, New York, 1981.
24. Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991.
25. Lang, J., *Creating Architectural Theory, The Role of the Behavioral Sciences in Environmental Design,* Van Nostrand Reinhold, New York, 1987.
26. Rowe, P. G., *Design Thinking,* MIT Press, Cambridge, MA, 1987.
27. Losk, Pieronek, Cureton, Geis, and Carpenter graduate reports are unpublished but available through the USC School of Engineering, Los Angeles, CA.
28. Alexander, C., *Notes on the Synthesis of Form,* Harvard University Press, Cambridge, MA, 1964. The first in a series of books on the subject
29. Kostof, S., *The Architect, Chapters in the History of the Profession,* Oxford University Press, New York, 1977.
30. Genesereth, M. R. and Nilsson, N. J., *Logical Foundations of Artificial Intelligence,* Morgan Kaufmann Publishers, Los Altos, CA.
31. Newell, A., *Unified Theories of Cognition,* Harvard University Press, Cambridge, MA, 1990.
32. Simon, H. A., *The Sciences of the Artificial,* MIT Press, Cambridge, MA, 1988.

33. Brooks, F. P., Jr., *The Mythical Man-Month,* Addison-Wesley Publishing, Reading, MA, 1982.
34. Garlan, D. and Shaw, M., *An Introduction to Software Architecture,* Carnegie Mellon University, Pittsburgh, PA, 1993.
35. Booton, R. C., Jr. and Ramo, S., The development of systems engineering, *IEEE Transac. Aerospace Electronic Syst.,* AES - 20, 4, 306, July 1984.

Appendix A

Heuristics for systems-level architecting

Experience is the hardest kind of teacher.
It gives you the test first and the lesson afterward.
(Susan Ruth 1993)

Introduction: organizing the list

The heuristics to follow were selected from Rechtin, 1991, the *Collection of Student Heuristics in Systems Architecting, 1988-93,*[1] and from subsequent studies in accordance with the selection criteria of Chapter 2. The list is intended as a tool store for top-level systems architecting. Heuristics continue to be developed and refined not only for this level, but for domain-specific applications as well, often migrating from domain-specific to system level, and vice versa.*

For easy search and use, the heuristics are grouped by architectural task and categorized by being either descriptive or prescriptive; that is, by whether they describe an encountered situation or prescribe an architectural approach to it, respectively.

There are over 180 heuristics in the listing to follow, far too many to study at any one time; nor were they intended to be.The listing is intended to be scanned as one would scan software tools on software store shelves, looking for ones that can be useful immediately but remembering that others are also there. Although some are variations of other heuristics, the vast majority stand on their own, related primarily to others in the near vicinity on the list. Odds are that the reader will find the most interesting heuristics in clusters, the location of which will depend on the reader's interests at the time. The section headings are by architecting task. A "D" signifies a descriptive heuristic; a "P"signifies a prescriptive one. When readily apparent, pre-

* The manufacturing, social, communication, software, management, business, and economics fields are particularly active in proposing and generating heuristics — though they usually are called principles, laws, rules, or axioms.

scriptions are grouped by insetting under appropriate descriptions or alternate prescriptions; otherwise, not. In the interest of brevity, an individual heuristic is listed in the task where it is most likely to be used most often. As noted in Chapter 2, some 20% can be tied to related ones in other tasks.

A major difference between a heuristic and an unsupported assertion is the credibility of the source. To the extent possible, the heuristics are credited to the individuals who, to the authors' knowledge, first suggested them. To further aid the reader in judging credibility or in finding the sources, the heuristics to follow are given symbols. These symbols indicate the following:

[ ] *An informal discussion* with the individual indicated, unpublished.
( ) A formal, dated source, with examples, located in the USC MS-SAE program archive, especially in the *Collection of Student Heuristics in Systems Architecting, 1988-93.* For further information, contact the Master of Science Program in Systems Architecture and Engineering, USC School of Engineering, University Park, Los Angeles, CA 90089-1450.
*Rechtin, 1991, where it is sourced more formally. By permission of Prentice-Hall, Englewood Cliffs, NJ.
Bold Key words useful for quick search. Otherwise, heuristics to follow are in plain type to make page reading easier. Real-world examples of each can be found in the references indicated.

The authors apologize in advance for any miscrediting of sources. Corrections are welcome. The readers are reminded that not all heuristics apply to all circumstances, just most to most.

Heuristic tool list

Multitask heuristics

D **Performance, cost, and schedule** cannot be specified independently. At least one of the three must depend on the others.*
D With few exceptions, schedule **delays** will be **accepted** grudgingly; cost **overruns** will not, and for good reason.
D The **time to completion** is proportional to the ratio of the time spent to the time planned to date. The greater the ratio, the longer the time to go.
D **Relationships** among the elements are what give systems their added value.*
D **Efficiency** is inversely proportional to **universality**. (Douglas R. King, 1992)

* As indicated in the introduction to this appendix, an asterisk indicates that this heuristic is taken from Rechtin, E., *Systems Architecting, Creating & Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991. With permission of Prentice-Hall, Englewood Cliffs, NJ 07632.

D **Murphy's Law,** "If anything can go wrong, it will."*

- P **Simplify**. Simplify. Simplify.*
- P The first line of defense against complexity is simplicity of design.
- P Simplify, combine, and eliminate. (Suzaki, 1987)
- P Simplify with smarter elements. (N. P. Geiss, 1991)
- P The most **reliable part** on an airplane is the one that isn't there — because it isn't needed. [DC-9 Chief Engineer, 1989]

D One person's **architecture** is another person's **detail**. One person's system is another's component. [Robert Spinrad, 1989]*

- **P** In order to **understand anything**, you must not try to understand everything. (Aristotle, 4th cent. B.C.)

P Don't confuse the functioning of the parts for the functioning of the system. (Jerry Olivieri, 1992)

D In general, each **system level** provides a context for the level(s) below. (G. G. Lendaris, 1986)

- **P** Leave the **specialties** to the specialist. The level of detail required by the architect is only to the depth of an element or component critical to the system as a whole. (Robert Spinrad, 1990) But the architect must have **access** to that level and know, or be informed, about its criticality and status. (Rechtin, 1990)
- P Complex systems will develop and **evolve** within an overall architecture much more rapidly if there are **stable intermediate** forms than if there are not. (Simon, 1969)*

D Particularly for social systems, it's the **perceptions**, not the facts, that count.

D In introducing technological and **social** change, **how** you do it is often more important than **what** you do.*

- P If social cooperation is required, the **way** in which a system is **implemented** and introduced must be an **integral part** of its architecture.*

D If the **politics** don't fly, the hardware never will. (Brenda Forman, 1990)

- D Politics, not technology, sets the limits of what technology is allowed to achieve.
- D Cost rules.
- D A strong, coherent constituency is essential.
- D Technical problems become political problems.
- D There is no such thing as a purely technical problem.
- D The best engineering solutions are not necessarily the best political solutions.

D **Good products** are not enough. Implementations matter. (Morris and Ferguson, 1993)

- P To remain competitive, determine and **control the keys** to the architecture from the very beginning.

Scoping and planning

The beginning is the most important part of the work.
(**Plato**, 4th cent. B.C.)
Scope! Scope! Scope! (William C. Burkett, 1992)

D **Success** is defined by the **beholder**, not by the architect.*

P The most important single element of success is to **listen** closely to what the customer perceives as his requirements and to have the will and ability to be responsive. (J. E. Steiner, 1978)*

P **Ask early** about how you will **evaluate** the success of your efforts. (Hayes-Roth et al., 1983)

P For a system to meet its **acceptance criteria** to the satisfaction of all parties, it must be architected, designed, and built to do so — no more and no less.*

P Define how an **acceptance** criterion is to be certified at the same time the criterion is established.*

D Given a **successful** organization or system with valid criteria for success, there are some things it **cannot do** — or at least not do well. Don't force it!

P The **strengths** of an organization or system in one context can be its **weaknesses** in another. Know when and where.*

D There's nothing like being the **first success**.*

P If at first you don't succeed, but the architecture is sound, try, try again. Success sometimes is where you find it. Sometimes it finds you.*

D A system is successful when the **natural intersection** of technology, politics, and economics is found. (A. D. Wheelon, 1986)*

D Four questions, **the Four Whos**, need to be answered as a self-consistent set if a system is to succeed economically; namely, who benefits?, who pays? and, as appropriate, who loses?

D **Risk** is (also) defined by the **beholder**, not the architect.

P If being absolute is impossible in estimating system risks, then be relative.*

D **No** complex system can be **optimum** to all parties concerned, nor all functions optimized.*

P Look out for hidden agendas.*

P It is sometimes more important to know **who** the customer is than to know what the customer **wants**. (Whankuk Je**,** 1993)

D The phrase, **"I hate it,"** is direction. (Lori I. Gradous, 1993)

P Sometimes, but not always, the best way to solve a difficult problem is to **expand** the problem, itself.*

P Moving to a **larger purpose** widens the range of solutions. (Gerald Nadler, 1990)

P Sometimes it is necessary to **expand the *concept*** in order to simplify the problem. (Michael Forte, 1993)

P [If in difficulty,] **reformulate** the problem and re-allocate the system functions. (Norman P. Geis, 1991)

P Use **open** architectures.You will need them once the market starts to respond.

P Plan to **throw one away**. You will anyway. (F. P. Brooks, Jr., 1982)

P You can't avoid **redesign**. It's a natural part of design.*

P Don't make an architecture **too smart** for its own good.*

D Amid a **wash of paper**, a small number of documents become critical pivots around which every project's management revolves. (F. P. Brooks, Jr., 1982)*

P Just because it's written, doesn't make it so. (Susan Ruth, 1993)

D In architecting a new [software] program all the **serious mistakes** are made in the **first day**. [Spinrad, 1988]

P The most **dangerous** assumptions are the **unstated** ones. (Douglas R. King, 1991)

D Some of the **worst** failures are **systems** failures.

D In architecting a new [aerospace] system, by the time of the first **design review**, performance, cost, and schedule have been **predetermined**. One might not know what they are yet, but to first order all the critical assumptions and choices have been made which will determine those key parameters.*

P **Don't assume** that the original statement of the problem is necessarily the best, or even the right, one.*

P **Extreme** requirements, expectations, and predictions should remain under challenge throughout system design, implementation, and operation.

P Any extreme requirement must be intrinsic to the system's design philosophy and must validate its selection. "Everything must pay its way on to the airplane." [Harry Hillaker, 1993]

P **Don't assume** that previous **studies** are necessarily complete, current or even correct. (James Kaplan, 1992)

P Challenge the process and solution, for surely someone else will do so. (Kenneth L. Cureton, 1991)

P Just because it worked in the past there's no guarantee that it will work now or in the future. (Kenneth L. Cureton, 1991)

P Explore the situation from more than one point of view. A seemingly impossible situation might suddenly become transparently simple. (Christopher Abts, 1988)

P **Work forward and backward**. (A set of heuristics from **Rubinstein**, 1975.)*

Generalize or specialize.
Explore multiple directions based on partial evidence.
Form stable substructures.
Use analogies and metaphors.
Follow your emotions.

P Try to hit a solution that, at worst, won't put you **out of business**. (Bill Butterworth as reported by Laura Noel, 1991)

P The **order** in which decisions are made can change the architecture as much as the decisions themselves. (Rechtin, 1975, IEEE SPECTRUM)

P Build in and maintain **options** as long as possible in the design and build of complex systems. You will need them. OR...Hang on to the agony of decision as long as possible. [Robert Spinrad, 1988]*

  P Successful architectures are **proprietary, but open**. [Morrison and Ferguson, 1993]

D Once the architecture begins to take shape, the sooner contextual constraints and **sanity checks** are made on assumptions and requirements, the better.*

D Concept **formulation** is complete when the **builder** thinks the system can be built to the **client's** satisfaction.*

D The **realities** at the end of the conceptual phase are not the models but the **acceptance criteria**.*

P Do the **hard parts** first.

  P Firm **commitments** are best made after the **prototype works**.

Modeling (see also Chapters 3 and 4)

P If you can't analyze it, don't build it.

D Modeling is a **craft** and at times an art. (William C. Burkett, 1994)

D A **vision** is an **imaginary architecture**…no better, no worse than the rest of the models. (M. B. Renton, Spring, 1995)

D From psychology: if the concepts in the mind of one person are very different from those in the mind of the other, there is no **common model** of the topic and no communication. [Taylor, 1975] OR... From telecommunications: The **best receiver** is one that contains an internal model of the transmitter and the channel. [Robert Parks and Frank Lehan, 1954]*

D A model is not **reality**.*

  D The **map** is not the territory. (Douglas R. King, 1991)*

  P Build **reality checks** into model-driven development. [Larry Dumas, 1989]*

  P Don't believe **nth order consequences** of a first order [cost] model. [R. W. Jensen, *circa* 1989]

D Constants aren't and variables don't. (William C. Burkett, 1992)

D One **insight** is worth a thousand **analyses**. (Charles W. Sooter, 1993)

  P Any war game, systems analysis, or study whose results can't easily be explained on the back of an envelope is not just worthless, it is probably dangerous. [Brookner-Fowler, *circa* 1988]

D Users develop **mental models** of systems based [primarily] upon the user-to-system interface. (Jeffrey H. Schmidt)

D If you can't explain it in **five minutes**, either you don't understand it or it doesn't work. (Darcy McGinn, 1992 from David Jones)

P The **eye** is a fine **architect**. Believe it. [Wernher von Braun, 1950]

D A good solution somehow **looks nice**. (Robert Spinrad, 1991)

  P **Taste:** an aesthetic feeling which will accept a solution as right only when no more direct or simple approach can be envisaged. [Robert Spinrad, 1994]

  P Regarding **intuition**, trust but **verify**. (Jonathan Losk, 1989)

Prioritizing (trades, options, and choices)

D In any resource-llimited situation, the **true value** of a given service or product is determined by what one is willling to **give up** to obtain it.

P When choices must be made with unavoidably inadequate information, **choose** the best available and then **watch** to see whether future solutions appear faster than future problems. If so, the choice was at least adequate. If not, go back and **choose** again.*

P When a decision makes sense through several different **frames**, it's probably a good decision. (J. E. Russo, 1989)

D The **choice** between architectures may well depend upon which set of **drawbacks** the client can handle best.*

P If trade results are inconclusive, then the wrong selection criteria were used. Find out [again] what the customer wants and why they want it, then repeat the trade using those factors as the [new] selection criteria. (Kenneth Cureton, 1991)

P The triage: Let the dying die. Ignore those who will recover on their own. And treat only those who would die without help.*

P Every once in a while you have to go back and see what the real world is telling you. [Harry Hillaker, 1993]

Aggregating ("chunking")

P Group elements that are strongly **related** to each other, **separate** elements that are **unrelated**.

D Many of the **requirements** can be **brought together** to complement each other in the total design solution. Obviously the more the design is put together in this manner, the more probable the overall success. (J. E. Steiner, 1978)

P Subsystem interfaces should be drawn so that each subsystem can be implemented independently of the specific implementation of the subsystems to which it interfaces. (Mark Maier, 1988)

P Choose a configuration with **minimal communications** between the subsystems. (computer networks)*

  P Choose the elements so that they are as independent as possible; that is, elements with low external complexity (low coupling) and high internal complexity (high cohesion). (Christopher Alexander, 1964 modified by Jeff Gold, 1991)*

P Choose a configuration in which local activity is high speed and global activity is slow change. (Courtois**,** 1985) *

P Poor aggregation results in **gray** boundaries and **red** performance. (M. B. Renton, Spring, 1995)

P **Never aggregate** systems that have a **conflict** of interest; partition them to ensure checks and balances. (Aubrey Bout, 1993)

P **Aggregate** around **"testable"** subunits of the product; **partition** around logical **subassemblies**. (Ray Cavola**,** 1993)

P Iterate the partition/aggregation procedure until a model consisting of **7 ± 2 chunks** emerge. (Moshe F. Rubinstein, 1975)

P The **optimum number** of architectural elements is the amount that leads to distinct **action**, not general planning. (M. B. Renton, Spring, 1995)

P System structure should resemble functional structure.*

P Except for good and sufficient reasons, **functional and physical** structuring should match.*

P The architecture of a **support** element must **fit** that of the system which it supports. It is easier to match a support system to the human it supports than the reverse.*

P **Unbounded limits** on element behavior may be a **trap** in unexpected scenarios. [Bernard Kuchta, 1989]*

Partitioning (decompositioning)

P Do not **slice** through regions where **high rates** of information exchange are required. (computer design)*

D The greatest **leverage** in architecting is at the **interfaces**.*

P **Guidelines** for a good quality interface specification: they must be simple, unambiguous, complete, concise, and focus on substance. Working documents should be the same as customer deliverables; that is, always use the customer's language, not engineering jargon. [Harry Hillaker, 1993]

P The **efficient architect**, using contextual sense, continually looks for likely **misfits** and redesigns the architecture so as to eliminate or minimize them. (Christopher Alexander, 1964)* It is inadequate to architect up to the boundaries or **interfaces** of a system; one must architect **across** them. (Robert Spinrad, as reported by Susan Ruth, 1993)

P Since boundaries are inherently limiting, look for solutions outside the boundaries. (Steven Wolf, 1992)

P Be prepared for **reality** to add a few interfaces of its own.*

P Design the structure with **good "bones."***

P Organize personnel tasks to **minimize** the **time** individuals spend interfacing. (Tausworthe, 1988)*

Integrating

D **Relationships** among the elements are what give systems their **added value**.*

P The greatest leverage in system architecting is at the interfaces.*

P The greatest **dangers** are also at the interfaces. [Raymond, 1988]

P Be sure to ask the question: "What is the worst thing that other elements could do to you across the interface? [Kuchta, 1989]

D Just as a piece and its template must match, so must a system and the resources which make, test, and operate it; or, more briefly, the **product and process** must match. Or, by extension, a system architecture cannot be considered complete lacking a suitable match with the process architecture.*

P When confronted with a particularly difficult interface, try changing its **characterization**.*

P Contain **excess energy** as close to the source as possible.*

P Place barriers in the paths between energy sources and the elements the energy can damage. (Kjos, 1988)*

Certifying (system integrity, quality, and vision)

D As **time to delivery** decreases, the **threat** to functionality increases. (Steven Wolf, 1992)

P If it is a good design, **insure** that it stays **sold**. (Dianna Sammons, 1991)

D Regardless of what has gone before, the **acceptance criteria** determine what is actually built.*

D The number of **defects remaining** in a (software) system after a given level of test or review (design review, unit test, system test, etc.) is proportional to the **number found** during that test or review.

P **Tally** the defects, analyze them, **trace** them to the source, make corrections, keep a **record** of what happens afterward, and keep **repeating** it. [Deming]

P **Discipline.** Discipline. Discipline. (Douglas R. King, 1991)

P The principles of **minimum communications** and proper partitioning are key to system testability and **fault isolation**. (Daniel Ley, 1991)*

P The **five whys** of Toyota's lean manufacturing. (To find the basic cause of a defect, keep asking "why" from effect to cause to cause five times.)

D The test **setup** for a system is itself a system.*

P The test system should always allow a part to pass or fail on its own merit. [James Liston, 1991]*

P To be tested, a system must be designed to be tested.*

D An element **"good enough"** in a small system is unlikely to be good enough in a more complex one.*

D Within the same class of products and processes, the **failure rate** of a product is linearly proportional to its **cost**.*

D The cost to find and **fix** an inadequate or failed part increases by an **order of magnitude** as it is successively incorporated into higher levels in the system.

> **P** The least expensive and most effective place to find and fix a problem is at its source.

D Knowing a **failure has occurred** is more important than the actual failure. (Kjos, 1988)

D **Mistakes** are **understandable**, failing to report them is **inexcusable**.

D Recovery from failure or flaw is not complete until a specific mechanism, **and no other**, has been shown to be the cause.*

D Reducing **failure rate** by each **factor of two** takes as much effort as the original development.*

D **Quality** can't be tested in, it has to be **built in**.*

> D You can't achieve quality...unless you specify it. (Deutsch, 1988)
>
> P Verify the quality close to the source. (Jim Burruss, 1993)
>
> P The five whys of Japan's lean manufacturing. (Hayes, et al., 1988)[2]
>
> D High **quality**, reliable systems are produced by high quality architecting, engineering, design, and manufacture, **not by inspection**, test, and rework.*
>
> P Everyone in the development and production line is both a customer and a supplier.

D Next to interfaces, the greatest **leverage** in architecting is in aiding the recovery from, or exploitation of, **deviations** in system performance, cost, or schedule.*

Assessing performance, cost, schedule, and risk

D A good design has benefits in more than one area. (Trudy Benjamin, 1993)

D System quality is defined in terms of customer satisfaction, not requirements satisfaction. (Jeffrey Schmidt, 1993)

D If you think your **design** is perfect, it's only because you haven't shown it to **someone else**. [Harry Hillaker, 1993]

> P Before proceeding too far, **pause and reflect**. Cool off periodically and seek an independent review. (Douglas R. King, 1991)

D Qualification and **acceptance** tests must be both definitive and passable.*

> P High **confidence**, not test completion, is the **goal** of successful qualification. (Daniel Gaudet, 1991)

P Before ordering a **test** decide what you will do if it is: (l) **positive** or (2) it is negative. If both answers are the same, **don't do** the test. (R. Matz, M. D., 1977)

D **"Proven"** and **"state of the art"** are mutually **exclusive** qualities. (Lori I. Gradous, 1993)

D The **bitterness** of **poor performance** remains long after the sweetness of low prices and prompt delivery are forgotten. (Jerry Lim, 1994)

D The **reverse of diagnostic** techniques are good architectures. (M.B. Renton, 1995)

D Unless everyone who **needs to know** does know, somebody, somewhere will foul up.

P Because there's no such thing as immaculate communication, don't ever stop **talking** about the system. (Losk, 1989)*

D Before it's tried, it's **opinion**. After it's tried, it's **obvious**. (William C. Burkett, 1992)

D Before the war it's opinion. After the war, it's too late! (Anthony Cerveny, 1991)

D The first **quick look** analyses are often **wrong**.*

D In correcting system deviations and failures it is important that all the participants know not only **what** happened and how it happened, but **why** as well.*

P Failure reporting without a **close out** system is meaningless. (April Gillam**,** 1989)

P Common , if undesirable, responses to **indeterminate outcomes** or failures:*

If it **ain't broke**, don't fix it.
Let's **wait and see** if it goes away or happens again.
It was just a **random** failure. One of those things.
Just treat the **symptom**. Worry about the cause later.
Fix everything that might have caused the problem.
Your **guess** is as good as mine.

D Chances for recovery from a **single failure** or flaw, even with complex consequences, are fairly good. Recovery from **two or more** independent failures is unlikely in real-time and uncertain in any case.*

Rearchitecting, evolving, modifying, and adapting

The test of a good architecture is that it will last.
The sound architecture is an enduring pattern.
[Robert Spinrad, 1988]

P The team that created and built a presently successful product is often the best one for its evolution — but seldom for creating its replacement.

D If you **don't understand** the existing system, you can't be sure you're re-architecting a **better** one. (Susan Ruth, 1993)

P When implementing a change, keep some elements constant to provide an **anchor** point for people to cling to. (Jeffrey H. Schmidt, 1993)

  P In large, mature systems, **evolution** should be a process of **ingress** and **egress**. (*IEEE*, 1992; Jeffrey Schmidt, 1992)

  P Before the change, it is your opinion. After the change it is your problem. (Jeffrey Schmidt, 1992)

D Unless constrained, **re-architecting** has a natural tendency to proceed unchecked until it results in a substantial transformation of the system. (Charles W. Sooter, 1993)

D Given a change, if the anticipated actions don't occur, then there is probably an invisible barrier to be identified and overcome. (Susan Ruth, 1993)

Exercises

Exercise: What favorite heuristics, rules of thumb, facts of life, or just plain common sense do you apply to your own day-to-day living — at work, at home, at play? What heuristics, etc., have you heard on TV or the radio; for example, on talk radio, action TV, children's programs? Which ones would you trust?

Exercise: Choose a system, product, or process with which you are familiar and assess it using the appropriate foregoing heuristics. What was the result? Which heuristics are or were particularly applicable? What further heuristics were suggested by the system chosen? Were any of the heuristics clearly incorrect for this system? If so, why?

Exercise: Try to spot heuristics and insights in the technical literature. Some are easy; they are often listed as principles or rules. The more difficult ones are buried in the text but contain the essence of the article or state something of far broader application than the subject of the piece.

Exercise: Try to create a heuristic of your own — a guide to action, decision making, or to instruction of others.

Notes and references

1. Rechtin, E., Editor, University of Southern California, Los Angeles, CA, March 15, 1994 (unpublished but available to students and researchers on request).
2. Hayes, R. H., Wheelwright, S. C., and Clark, K. B., *Dynamic Manufacturing, Creating the Learning Organization*, Free Press, New York, 1988.

Appendix B

Reference texts suggested for institutional libraries

The following list of texts is offered as a brief guide to books that would be particularly appropriate to an architecting library.

Architecting background

Alexander, C., *A Pattern Language: Towns, Buildings, Construction,* Oxford University Press, New York, 1977.

Alexander, C., *The Timeless Way of Building,* Oxford University Press, New York, 1979.

Alexander, C., *Notes on the Synthesis of Form,* Harvard University Press, Cambridge, MA, 1964.

Kostoff, Spiro, *The Architect,* Oxford University Press, 1977, paperback.

Lang, Jon, *Creating Architectural Theory,* van Nostrand Reinhold Company, 1987.

Rowe, P.G., *Design Theory,* The MIT Press, Cambridge, MA, 1987.

Vitruvius, *The Ten Books on Architecture,* translated by Morris Hicky Morgan, Dover Publications, 1960, paperback.

Management

Augustine, N.R., *Augustine's Laws,* AIAA, Inc., 1982.

Deal, Terrence E. and Kennedy, A. A., *Corporate Cultures, The Rites and Rituals of Corporate Life,* Addison-Wesley, Reading, MA, 1988.

DeMarco, T. and Lister, T., *Peopleware: Productive Projects and Teams,* Dorset House, New York, 1987.

Juran, J.M., *Juran on Planning for Quality,* The Free Press, A Division of Macmillan, Inc., New York, 1988.

Modeling

Eisner, H., *Computer Aided Systems Engineering,* Prentice-Hall, Englewood Cliffs, NJ, 1988.

Hatley, D. J. and Pirbhai, I., *Strategies for Real-Time System Specification,* Dorset House, New York, 1988.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., *Object-Oriented Modeling and Design,* Prentice-Hall, Englewood Cliffs, NJ, 1991.

Ward, P.T. and Mellor, S. J., *Structured Development for Real-Time Systems, Volume 1: Introduction and Tools,* Yourdon Press (Prentice-Hall), Englewood Cliffs, NJ, 1985.

Specialty areas

Baudin, M., *Manufacturing Systems Analysis,* Yourdon Press Computing Series, (Prentice-Hall), Englewood Cliffs, NJ, 1990.

Hayes, R. H., Wheelwright, S. C., and Clark, K. B., *Dynamic Manufacturing,* The Free Press A Division of Macmillan, Inc., New York, 1988.

Miller, J.G., *Living Systems,* McGraw Hill, New York, 1978.

Simon, H.A., *Sciences of the Artificial,* The MIT Press, Cambridge, MA, 1981.

Thome, B., editor, *Systems Engineering: Principles and Practice of Computer-Based Systems Engineering,* John Wiley, Baffins Lane, Chichester, Wiley Series on Software Based Systems, 1993.

Software

Boehm, B., *Software Engineering Economics,* Prentice-Hall, Englewood Cliffs, NJ, 1981.

Brooks, F. P., Jr., *The Mythical Man-month, Essays on Software Engineering, 20th Anniversary Edition,* Addison-Wesley, Reading, MA, 1995.

Gajski, D. D., Milutinovic´, V. M., Siegel, H. J., and Furht, B. P., *Computer Architecture,* The Computer Society of the IEEE, 1987 (tutorial).

Gamma, E. et. al., *Design Patterns: Elements of Reusable Object-Oriented Software Architecture,* Addison Wesley, Reading, MA, 1994.

Deutsch, M. S. and Willis, R. R., *Software Quality Engineering,* Prentice-Hall, Englewood Cliffs, NJ, 1988.

Software Productivity Consortium, ADARTS Guidebook, SPC-94040-CMC, Version 2.00.13, Vols. 1-2, September, 1991.

Shaw, M. and Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline,* Prentice-Hall, Englewood Cliffs, NJ, 1996.

Yourdon, E. E. and Constantine, L. L., *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design,* Yourdon Press, (Prentice-Hall) Englewood Cliffs, NJ, 1979.

System sciences

Flood, R. L. and Carson, E. R., *Dealing with Complexity, an Introduction to the Theory and Application of System Sciences,* Plenum Press, New York, 1988.

Genesereth, M. S. and Nilsson, N. J., *Logical Foundations of Artificial Intelligence,* Morgan Kaufmann, San Francisco, CA, 1987.

Gerstein, D. R. et al., Editors, *The Behavioral and Social Sciences, Achievements and Opportunities,* National Academy Press, Washington, D.C., 1988.

Klir, G. J., *Architecture of Systems Problem Solving,* Plenum Press, New York, 1985.

System thinking

Arbib, M. A., Brains, *Machines, and Mathematics,* Second ed., Springer-Verlag, New York, 1987.

Beam, W. R., *Systems Enginering, Architecture and Design,* McGraw-Hill, New York, 1990.

Boorstin, D. J., *The Discoverers,* Vintage Books, New York, 1985.

Boyes, J. L., Editor, *Principles of Command and Control,* AFCEA International Press, Washington, D.C., 1987.

Davis, S. M., *Future Perfect,* Addison Wesley, Reading, MA, 1987.

Gause, D. C. and Weinberg, G. M., *Exploring Requirements, Quality Before Design,* Dorset House Publishing, New York, 1989.7

Hofstadter, D. R., *Gödel, Escher, Bach: An Eternal Golden Braid,* Vintage Books, New York, 1980, paperback.

Norman, D. A., *The Psychology of Everyday Things,* Basic Books, New York, 1988.

Pearl, J., *Heuristics,* Addison-Wesley, Reading, MA, 1984.

Rechtin, E., *Systems Architecting: Creating and Building Complex Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1991.

Rubinstein, M. F., *Patterns of Problem Solving,* Prentice-Hall, Englewood Cliffs, NJ, 1975.

Weinberg, G. M., *Rethinking Systems Analysis and Design,* Dorset House, New York, 1988.

Appendix C

On defining architecture and other terms

This appendix is for those who need to come to a consensus in a group on a definition for architecture or other major terms used in this book. There are many who might have such a need, and for those who have a need this appendix might be very useful. Deciding on formal definitions is commonly part of setting up an official corporate training course, or documenting a standard (public or corporate). In these situations an inordinate amount of time can be spent arguing about fine details of definitions. It may be hard to pick and choose among the definitions offered by different standards since they usually do not record the reasoning that brought them to a decision. This appendix is a record of some of the definition related discussions one of the authors (Maier) has been involved in over several years. It is offered to help others who need to arrive at a group consensus on definitions with a ready-made set of choices and reasoning.

Defining "architecture"

One might think that, with 5000 years of history, the notion of architecture in buildings would be clearly and crisply defined. Presumably then the definition could be extended to give a clear and crisp definition to architecture in other fields. However, this is not the case. A formal definition of architecture is elusive even in the case of buildings. And if the definition is elusive in its original domain, is it surprising that a wholly satisfactory definition is elusive in more general domains?

The communities involved in architecture in systems, software, hardware, and other domains have struggled with finding a formal definition. Each group who has set out a formal definition has usually made a unique choice. The choices are often similar, but reflect significantly different ideas. The sections to follow review some of the more distinctive choices. Of course, there are many small variations on each one.

To make sense of the different definitions it is important to review them with some criteria in mind. In reviewing these definitions try to answer the following questions with respect to each definition:

1. How does the definition establish what is the concern of the architect and what is not?
2. What is the purpose of the definition? Some purposes might be defining an element of design, education, organizational survival or politics, setting legal boundaries, or even humor.
3. Choose a building with which you are familiar. What is its architecture, according to the definition? How well does the definitions implied architecture match what you would expect to be the building architect's scope of work?
4. Choose a system with which you are familiar. What is its architecture, according to the definition? What things are uniquely determined about the system from the application of that definition?
5. What is the architecture of the Internet, according to the definition?

Webster's dictionary definition

We begin with the dictionary's definition.[1]

> Architecture: 1. The art or science of building; specifically, the art or practice of designing and building structures and esp. habitable ones. 2a. Formation or construction as or as if the result of conscious act <the ~ of the garden> b. a unifying or coherent form or structure <the novel lacks ~> 3. Architectural product or work 4. A method or style of building 5. The manner in which the components of a computer or computer system are organized and integrated.

The interesting part of this definition, for our purposes, is part 2. The first definition uses architecture in the sense of the profession, not what we are looking for here. This definition say to speak of the architecture of a thing is to speak of its "unifying or coherent form." Unfortunately, it is not obvious what aspect of form is "unifying or coherent." It is something that can be judged, but is hard to define crisply. The civil building example suggests several other ideas about architecture:

1. Architecture is tied to the structure of components, but if a novel can have an architecture the notion of components is relatively abstract. Components may need to be interpreted broadly in some contexts. No one would confuse the structure of a novel with its organization into chapters — which is the "packaging" of that structure, and is

analogous to confusing the architecture of a system with its module structure.

2. The distinction between an architectural level of description and some other level of design description is not crisp. Architectural description is concerned with unifying characteristics or style, while an engineering description is concerned with construction or acquisition.
3. In common use architecture can mean a conceptual thing, the work of architects, and architectural products. Other definitions make sharper distinctions.

This book

The definition of architecture given in the glossary of this book is

> Architecture: The structure (in terms of components, connections, and constraints) of a product, process, or element

This definition is specific, it is talking about structure (although that term is itself open to some interpretation). Components, connections, and constraints are the descriptive terms for architecture; and we can talk about the architecture of a wide variety of things. This book is primarily about architecting, rather than architecture. The reason is that the most important constraints come from the process of doing the architect's role. The most important things come from working with clients to understand purpose and limitations. Architecture should, by the tenets of this book, proceed from the clients needs rather than a presupposed notion of what constitutes an architectural level definition of a system.

IEEE architecture working group (AWG)

After extended discussion in 1995-1996 in association with developing P1471 Recommended Practice for Architectural Description the IEEE Working Group chose:

> An Architecture is the highest-level concept of a system in its environment.

"System" in this definition refers back to the official IEEE definition, "a collection of components organized to accomplish a specific function or set of functions." This definition of architecture was intended to capture several ideas.

1. An architecture is a property of a thing or a concept, not a structure. The term structure is avoided specifically to avoid any connotation

that architecture was solely a matter of physical structure. Concept, which is obviously much more generic, is used instead.

2. The term highest-level is used to indicate that architecture is an abstraction, and that it is fundamental abstraction. A major defect of this definition is that highest level carries a connotation of levels of hierarchy, and in particular a single hierarchy, which is exactly one of the connotations to be avoided. Also, "highest-level concept" leaves a great deal of room for interpretation.
3. The definition says that architecture is not a property of the system alone, but that the system's environment must be included in a definition of the system's architecture. This has often been referred to as "architecture in context" as opposed to "architecture as structure." It was there to capture the idea that architecture has to encompass purpose and the relationship of the system to its stakeholders. The reader must judge whether or not that interpretation is clear.

This definition was used in several drafts of the P1471 standard, but was replaced in the final balloted version. The definition in the final balloted version was

> Architecture: the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.

This definition is a refinement of definitions from the software engineering community, as discussed below. Those who don't like it might be more inclined to say it was a compromise between conflicting points of view that suffers from the usual problems of a committee decision. The definition starts with the software communities definitions (discussed shortly) and then adds back some of the ideas of the original P1471 definition. The primary refinement is the de-emphasis on physical structure and to say that architecture is "embodied" in components, relationships, and principles. Put another way, the definition tries to recognize that, for most systems, most of the time, the architecture is in the arrangement of physical components and their relationships; but, sometimes, the fundamental organization is on a more abstract level.

INCOSE SAWG

The International Council on Systems Engineering (INCOSE) Systems Architecture Working Group (SAWG) adopted a definition for system architecture. It could as well be read as a definition for "Architecture, of a system." It reads:

> Systems architecture: The fundamental and unifying system structure defined in terms of system elements, interfaces, processes, constraints, and behaviors

This definition borrows the core of the dictionary definition that architecture represents fundamental, unifying, or essential structure. Exactly what constitutes fundamental, unifying, or essential is not easily defined. It is presumed that recognizing it is partially art and up to the participants. In this definition the role of multiple aspects making up the architecture is made explicit through the listing of elements, interfaces, processes, constraints, and behaviors. This definition makes, or facilitates making, a sharper separation between an architecture as a conceptual object, an architecture description as concrete object, and the process or act of creating architectures (architecting).

MIL-STD-498

MIL-STD-498, now canceled, had a definition of architecture that specifically pertained to a designated development task.

> Architecture: The organizational structure of a system or CSCI, identifying its components, their interfaces, and a concept of execution among them

Here architecture is described specifically in three parts, components, interfaces, and a concept of execution. In this sense it supports the idea of architecture as inherently multi-view, although it specifically defines the views where others leave them open. The meaning of "organizational structure" as opposed to some other structure (conceptual, implementation, detailed, etc.) is not made clear, although the idea is congruent to the common usage of architecture. It also uses "concept" within the definition, but only in referring to execution. Like most definitions, it doesn't clearly make a distinction between architectural and design concerns.

This definition is also "structuralist" in the sense that it emphasizes the structure of the system rather than its purposes or other relationships. One could interpret the definition to mean that the architect was not concerned with the systems purpose, that architecture came after requirements were fully defined. In fact, that is exactly the interpretation it should be given, at least in the way the associated standards envisioned the systems engineering process executing.

The original IEEE definition (in IEEE 610.12-1990) is a shorter version of this. It reads:

> The organizational structure of a system or component

Perry-Garlan

A widely used definition in the software community is due to Perry and Garlan, although the exact place it first appeared is somewhat obscure.

> Architecture: The structure of the components of a system, their interrelationships, and principles and guidelines governing their design and evolution over time.

An almost identical definition is used as the definition of architecture in the U.S. DoD C4ISR Architecture Framework, where it is incorrectly credited to the IEEE 610.12 standard for terminology. This definition is another three-part specification — components, interrelationships, and principles-guidelines. As this definition is commonly used, components and interrelationships usually refer to physically identifiable elements. This definition is mostly used in the software architecture community, and there it is common to see components identified as code units, classes, packages, tasks, and other code abstractions. The interrelationships would be calls or lines of inheritance.

The two basic objections to this definition are that it implies (if primarily through use rather than the words) that architecture is the same as physical structure, and that it makes no distinction in level of abstraction. The common usage of architecture is in reference to abstracted properties of things, not to the details. The Perry-Garlan definition can presumably apply to the structure of components at any level of abstraction. While applicability to multiple levels is, in part, desirable, it is also desirable to distinguish between what constitutes an architectural level description (whether of a whole system or of a component) from descriptions at lower levels of abstraction.

Maier's tongue-in-cheek rule of thumb

A slightly flip, but illustrative way of defining architecture is to go back to what architects are supposed to do.

> An architecture is the set of information that defines a systems value, cost, and risk sufficiently for the purposes of the systems sponsor.

Obviously, this definition reflects the issue back to architecting, when the definition of architecture reflects back to architecture. The point of this definition is that architecture is what architects produce, and that what architects do is help clients make decisions about building systems. When the client makes acquisition decisions, architecture has been done (perhaps unconsciously, and perhaps very badly, but done).

Internet discussion

One of the questions given at the beginning was "What is the architecture of the Internet?" The point of the question is that no reasonable notion of unifying, organizing, or coherent form will produce a physical description of the Internet. The specific pattern of physical links is continuously changing and of little interest. However, there is a very clear unifying structure, but it is a structure in protocols. It is not even a structure in software components, as exactly what software components implement the protocols is not known even to the participating elements. The point about protocols being the organizing structure of the Internet, and in particular the IP, was made in Chapter 7 and Figure 7.1.

Summary

Those who must choose definitions have a lot to work with, probably more than they would want. The precise form of the definition is less important than the background of what architecture should be about. What architecting should be was discussed at length in Chapter 1. The specifics of what architects will produce, that is what an architecture actually looks like, will differ from domain to domain. Ideally, the definition for a given organization should come from that knowledge — the knowledge of what is needed to successfully define a system concept and take it through development. If the organization has that knowledge it should be able to choose a formal definition that encapsulates it. If the organization does not have that knowledge then no formal definition will produce it.

Models, viewpoints, and views

The terms model, view, and viewpoint are important in setting architecture description standards, or architecture frameworks using the community terminology of Chapter 11. The meaning of these terms changes from standard to standard. The discussion below is intended to capture an argument for a distinction between the two meanings. The distinction can be useful in writing standards, though it is not important in writing architecture descriptions, nor is it extensively used in this book.

Why do we need some organizing abstraction beyond just models? Experience teaches that particular collections of models are logically related by the kinds of issues or concerns they address. The idea of a view comes from architectural drawings. In a drawing we talk about the top view or the side view of an object in referring to its physical representation as seen from a point. A view is the representation of a system from a particular point or perspective. A view is a representation of the whole system with respect to a set of related concerns. A viewpoint is the abstraction of many related views, it is the idea of viewing something from "the front," for example.

A view need not correspond to physical appearance. A functional view is a representation of a system abstracting away all non-functional or non-behavioral details. A cutaway view shows some mixture of internal and external physical features in a mixture defined by the illustrator.

A view can be thought of both projectively and constructively. In the projective sense a view is formed by taking the system and abstracting away all the details unnecessary to the view. It is analogous to taking a multidimensional object and projecting it onto a lower dimensional space (like a viewing plane). For example, a behavioral view is the system pared down to only its behaviors, its set of input to output traces.

In the constructive sense we build a complete model of the system by building a series of views. Each represents the system from one perspective, with enough the system should be "completely" defined. It is like sketching a front view, a side view, a top view, and then inferring the structure of the whole object. In more general systems, we might build a functional view, then a physical view, then a data view, then return to the functional view, etc. until a complete model is formed from the joint set of views.

In practice it usually takes several models to represent that whole system relative to typical concerns, at least for high technology systems. So, a view is usually a collection of models. For example, physical representation seems simple enough, but how many different models are needed to represent the components of an information intensive system? A complete physical view might need conventional block diagrams of information flow, block diagrams of communication interconnection, facilities layouts, and software component diagrams.

Viewpoints are motivated noticing that we build similar views, using similar methods, for many systems. By analogy, we will want to draw a top view of most systems we build. The civil architect always draws a set of elevations, and elevation drawings share common rules and structures. And an information system architect will build information models using standard methods for each system. This similarity is because related systems will typically have similar stakeholders, and these stakeholders find their concerns consistently addressed by particular types of models and analysis methods. Hence, a viewpoint can be thought of as a set of modeling or analysis methods together with the concerns those methods address and the stakeholders possessing those concerns.

Working definitions

These are summarizing definitions, augmented with the notions of consistency and completeness. The concepts here refer to the P1471 information model in Figure 11.1.

Model: An approximation, representation or idealization of selected aspects of the structure, behavior, operation or other characteristics of a real-world process, concept, or system (IEEE 610.12-1990).

Viewpoint: A template, pattern, or specification for constructing a view (IEEE 1471-2000).

View: A representation of a system from the perspective of related concerns or issues (IEEE 1471-2000).

Consistency, of views: Two or more views and consistent if at least one can exist that possesses the given views.

Completeness, of view: A set of views is complete if they satisfy (or "cover") all of the concerns of all stakeholders of interest.

Consistency and completeness

Given multiple views (like top, front, and side) of a physical object the ideas of consistency and completeness are clear. A set of views is consistent if they are abstractions of the same object. A little more generally, they are consistent if at least one real object exists which has the given views. Consistency for physical object and views can be checked through solid geometry. Figure C.1 illustrates the point. The views are consistent if the geometrical object produces them when project onto the appropriate subspace. Even without the actual object we can perform geometric checks on the different views.

We can't (yet) treat consistency in the same rigorous manner if the views are functional and physical and of a complex system. As we describe more complex views in the sections to come for systems it is useful to return to the heuristic notion of consistency. Given a few models of a system being architected, we say they are consistent if at least one implementation exists which has the models as abstractions of itself.

Completeness can also be heuristically understood through the geometric analogy. Suppose we have set of visual representations of a material object. What does it mean to claim that the set of representations (views) is "complete?" Logically, it means that the views completely define the object. But, any set of external visual representations can only define the external shape of the object, it can't define the internal structure, if any. This trivial observation is actually extremely important for understanding architecture. No set of representations is *ever* truly complete. A set of representations can only be complete with respect to something, say with respect to some set of concerns. If the concerns are an external shape, then some set of external visual representations can be complete. If the concerns are extended to include internal structures, or strength properties, or weight, or any number of other things then the set of views must likewise be extended to be "complete."

Notes and references

1. *Merriam Webster's Collegiate Dictionary,* Tenth Edition, p. 61.

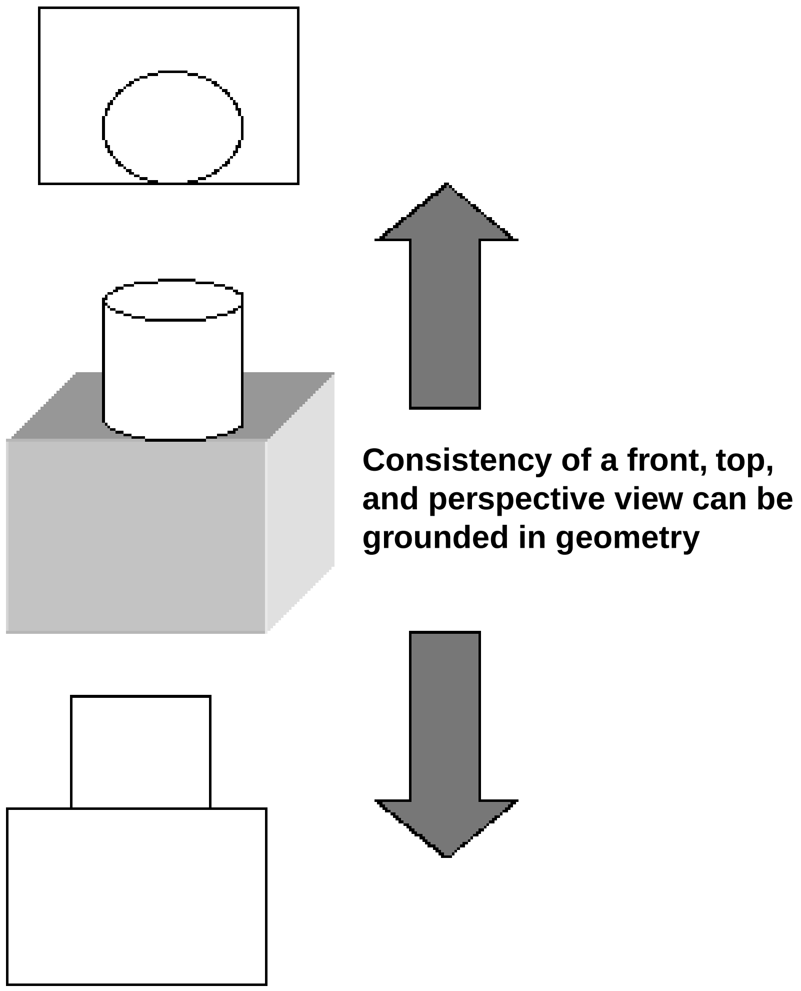


Figure C.1 For physical objects and views consistency is a matter of geometry. For systems and more complex views it does not (yet) have the same rigorous grounding.

Glossary

The fields of system engineering and systems architecting are sufficiently new that many terms have not yet been standardized. Common usage is often different among different groups and in different contexts. However, *for the purposes of this book*, we have provided the meanings of the following terms:

Abstraction A representation in terms of presumed essentials, with a corresponding suppression of the non-essential.

ADARTS Ada-based Design Approach for Real Time Systems. A software development method (including models, processes, and heuristics) developed and promoted by the Software Productivity Consortium.

Aggregation The gathering together of closely related elements, purposes, or functions.

Architecting The processing of creating and building architectures. Depending on one's perspective, architecting may or may not be seen as a separable part of engineering. Those aspects of system development most concerned with conceptualization, objective definition, and certification for use.

Architecture The structure — in terms of components, connections, and constraints — of a product, process, or element.

Architecture, open An architecture designed to facilitate addition, extension, or adaptation for use.

Architecture, (communication, software, hardware, etc.) The architecture of the particular designated aspect of a large system.

Architectural style A form or pattern of design with a shared vocabulary of design idioms and rules for using them (see Shaw and Garlan, 1996, p. 19).

ARPANET/INTERNET The global computer internetwork, principally based on the TCP/IP packet communications protocol. The ARPANET was the original prototype of the current INTERNET.

Certification A formal, but not necessarily mathematical, statement that defined system properties or requirements have been met.

Client The individual or organization that pays the bills. May or may not be the user.

Complexity A measure of the numbers and types of interrelationships among system elements. Generally speaking, the more complex a system, the more difficult it is to design, build, and use.

Deductive reasoning Proceeding from an established principle to its application.

Design The detailed formulation of the plans or instructions for making a defined system element; a follow-on step to systems architecting and engineering.

Domain A recognized field of activity and expertise, or of specialized theory and application.

Engineering Creating cost-effective solutions to practical problems by applying scientific knowledge to building things in the service of mankind (Shaw and Garlan, 1996, p. 6). May or may not include the art of architecting.

Engineering, concurrent Narrowly defined (here) as the process by which product designers and manufacturing process engineers work together to create a manufacturable product.

Heuristic A guideline for architecting, engineering, or design. Lessons learned expressed as a guideline. A natural language abstraction of experience which passes the tests of Chapter 2.

Heuristic, descriptive A heuristic that describes a situation.

Heuristic, prescriptive A heuristic that prescribes a course of action.

IEEE P 1220 An Institute of Electrical and Electronic Engineers standard for system engineering

Inductive reasoning Extrapolating the results of examples to a more general principle

Manufacturing, flexible Creating different products on demand using the same manufacturing line. In practice, all products on that line come from the same family.

Manufacturing, lean An efficient and cost-effective manufacturing or production system based on ultraquality and feedback. See Womack et al., 1990.

MBTI Meyer-Briggs Type Indicator. A psychological test for indicating the temperaments associated with selected classes of problem solving. See Meyers, Briggs, and McCaulley, 1989.

Metaphor A description of an object or system using the terminology and properties of another. For example, the desktop metaphor for computerized document processing.

MIL-STD Standards for defense system acquisition and development.

Model An abstracted representation of some aspect of a system.

Model, satisfaction A model which predicts the performance of a system in language relevant to the client.

Modeling Creating and using abstracted representations of actual systems, devices, attributes, processes, or software.

Models, integrated A set of models, representing different views, forming a consistent representation of the whole system

Normative method A design or architectural method based on "what should be," that is, on a predetermined definition of success.

OMT Object Modeling Technique. An object-oriented software development method. See Rumbaugh et al., 1991

Objectives Client needs and goals, however stated.

Paradigm A scheme of things, a defining set of principles, a way of looking at an activity, for example, classical architecting.

Participative method A design method based on wide participation of interested parties. Designing through a group process.

Partitioning The dividing up of a system into subsystems.

Progressive design The concept of a continuing succession of design activities throughout product or process development. The succession progressively reduces the abstraction of the system through models until physical implementation is reached and the system used.

Purpose A reason for building a system.

Rational method A design method based on deduction from the principles of mathematics and science.

Requirement An objective regarded by the client as an absolute; that is, either passed or not.

Scoping Sizing; defining the boundaries and/or defining the constraints of a process, product, or project.

Spiral A model of system development which repeatedly cycles from function to form, build, test, and back to function. Originally proposed as a risk-driven process, particularly applicable to software development with multiple release cycles.

System A collection of things or elements which, working together, produce a result not achievable by the things alone.

Systems, builder-architected Systems architected by their builders, generally without a committed client.

Systems, feedback Systems which are strongly affected by feedback of the output to the input.

Systems, form first Systems which begin development with a defined form (or architecture) instead of a defined purpose. Typical of builder-architected systems.

Systems, politicotechnical Technological systems the development and use of which is strongly influenced by the political processes of government.

Systems, sociotechnical Technological systems the development and use of which is strongly affected by diverse social groups. Systems in which social considerations equal or exceed technical ones.

Systems engineering A multidisciplinary engineering discipline in which decisions and designs are based on their effect on the system as a whole.

Systems architecting The art and science of creating and building complex systems. That part of systems development most concerned with scoping, structuring, and certification.

Systems architecting, the art of That part of systems architecting based on qualitative heuristic principles and techniques; that is, on lessons learned, value judgments, and unmeasurables.

Systems architecting, the science of That part of systems architecting based on quantitative analytic techniques; that is on mathematics and science and measureables.

Technical decisions Architectural decisions based on engineering feasibility.

Ultraquality Quality so high that measuring it in time and at reasonable cost is impractical. (See Rechtin, 1991, Chap. 8)

Waterfall A development model based a single sequence of steps; typically applied to the making of major hardware elements.

Value judgments Conclusions based on worth (to the client and other stakeholders).

View A perspective on a system describing some related set of attributes. A view is represented by one or more models.

Zero defects A production technique based on an objective of making everything perfectly. Related to the "everyone a supplier, everyone a customer" technique for eliminating defects at the source. Contrasts with acceptable quality limits in which defects are accepted providing they do not exceed specified limits in number or performance.