



Calhoun: The NPS Institutional Archive
DSpace Repository

Theses and Dissertations

1. Thesis and Dissertation Collection, all items

2008-09

An engineering context for software engineering

Riehle, Richard D.

Monterey, California. Naval Postgraduate School

<http://hdl.handle.net/10945/10379>

Downloaded from NPS Archive: Calhoun



Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

<http://www.nps.edu/library>



**NAVAL
POSTGRADUATE
SCHOOL**

MONTEREY, CALIFORNIA

DISSERTATION

**AN ENGINEERING CONTEXT FOR SOFTWARE
ENGINEERING**

by

Richard D. Riehle

September 2008

Dissertation Supervisor:

J. Bret Michael

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE September 2008	3. REPORT TYPE AND DATES COVERED Dissertation	
4. TITLE AND SUBTITLE: An Engineering Context for Software Engineering		5. FUNDING NUMBERS	
6. AUTHOR(S) Richard D. Riehle		8. PERFORMING ORGANIZATION REPORT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.	
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) <p>New engineering disciplines are emerging in the late Twentieth and early Twenty-first Century. One such emerging discipline is software engineering. The engineering community at large has long harbored a sense of skepticism about the validity of the term <i>software engineering</i>. During most of the fifty-plus years of software practice, that skepticism was probably justified. Professional education of software developers often fell short of the standard expected for conventional engineers; software practice seemed to be a "hit or miss" approach; and the available knowledge, tools, and language designs were not sufficiently mature to support an engineering model for software practice.</p> <p>Much progress has occurred in recent years, due to improved tools and languages along with a better ways of reasoning about and designing software products. This progress has contributed to the increase in success in the way software is developed and managed. However, even with a growing number of software successes, there are still enough horror-stories to reinforce the skepticism of the larger engineering community. Those skeptics continue to ask the reasonable question, "Where is the engineering in software engineering?"</p> <p>The <i>primary contribution of this dissertation</i> is to establish a foundation for answering the question at the end of the previous paragraph. Another contribution is a foundation for answering that same question for other emerging engineering disciplines. We call this foundation a <i>context</i>. The context is derived from: a study of conventional engineering, a review of contemporary software practices, recent advances in software engineering and computer science, and analysis of the relationships between those four concerns.</p> <p>This engineering context for software engineering includes two chapters on the topic of engineering. It opens the door to a dialogue about both the philosophical and practical concerns of emerging engineering disciplines. It also includes chapters mapping the engineering context to both current and expected trends in software engineering practices.</p>			
14. SUBJECT TERMS Software Engineering, Programming, Computer Software, Engineering, Software Process, Pre-Conditions, Post-Conditions, Invariants, Ada, Risk Management, Predictable Outcome, Linguistic Continuity, Software Physics, Design Metrics, Design to Tolerances			15. NUMBER OF PAGES 139
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AN ENGINEERING CONTEXT FOR SOFTWARE ENGINEERING

Richard D. Riehle
Visiting Professor, Computer Science Department
B.A., Brigham Young University, 1965
M.S., National University, 1990

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN SOFTWARE ENGINEERING

from the

**NAVAL POSTGRADUATE SCHOOL
September 2008**

Author:

Richard D. Riehle

Approved by:

Dr. J. Bret Michael
Professor of Computer Science
Dissertation Supervisor

Dr. Peter J. Denning
Chair, Computer Science Dept.
Dissertation Committee Chair

Dr. Dan Boger
Interim Vice Provost and
and Dean of Research

Dr. Mikhail Auguston
Associate Professor
of Computer Science

Dr. Mantak Shing
Associate Professor
of Computer Science

Dr. Qiaoyun "Liz" Li
SkySurfer Systems & Motorola

Approved by:

Dr. Peter J. Denning, Chair, Department of Computer Science

Approved by:

Douglas Moses, Associate Provost for Academic Affairs

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

New engineering disciplines are emerging in the late Twentieth and early Twenty-first Century. One such emerging discipline is software engineering. The engineering community at large has long harbored a sense of skepticism about the validity of the term *software engineering*. During most of the fifty-plus years of software practice, that skepticism was probably justified. Professional education of software developers often fell short of the standard expected for conventional engineers; software practice seemed to be a “hit or miss” approach; and the available knowledge, tools, and language designs were not sufficiently mature to support an engineering model for software practice.

Much progress has occurred in recent years, due to improved tools and languages along with a better ways of reasoning about and designing software products. This progress has contributed to the increase in success in the way software is developed and managed. However, even with a growing number of software successes, there are still enough horror-stories to reinforce the skepticism of the larger engineering community. Those skeptics continue to ask the reasonable question, “Where is the engineering in software engineering?”

The *primary contribution of this dissertation* is to establish a foundation for answering the question at the end of the previous paragraph. Another contribution is a foundation for answering that same question for other emerging engineering disciplines. We call this foundation a *context*. The context is derived from: a study of conventional engineering, a review of contemporary software practices, recent advances in software engineering and computer science, and analysis of the relationships between those four concerns.

This engineering context for software engineering includes two chapters on the topic of engineering. It opens the door to a dialogue about both the philosophical and practical concerns of emerging engineering disciplines. It also includes chapters mapping the engineering context to both current and expected trends in software engineering practices.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION AND OVERVIEW	1
A.	THESIS ORGANIZATION	1
B.	WHAT PROBLEM ARE WE TRYING TO SOLVE?	2
1.	Problem Sources	2
2.	Bounding the Problem	3
C.	CONTRIBUTION SUMMARY	5
1.	The Dual Path	5
2.	The Engineering Path	5
3.	The Software Science Path	6
4.	The Converging Path	6
5.	The Contribution Summary	8
D.	METHODOLOGICAL APPROACH	9
1.	Assumptions and Constraints	9
a.	<i>The Early Pioneer’s Vision</i>	9
b.	<i>Software Process</i>	9
2.	Research Approach	10
a.	<i>Literature Review</i>	10
b.	<i>Survey of Engineers</i>	10
c.	<i>Interviews</i>	12
d.	<i>Reading</i>	12
e.	<i>Developed Examples</i>	13
E.	EVALUATION ISSUES	13
II.	SURVEY OF PREVIOUS WORK	15
A.	INTRODUCTION TO SURVEY	15
1.	The Larger Literature	15
2.	Process Literature	15
3.	Observations from the Literature Review	16
a.	<i>Observation One</i>	16
b.	<i>Observation Two</i>	16
c.	<i>Observation Three</i>	17
B.	SOFTWARE ENGINEERING DEFINITIONS	17
1.	Representative Software Engineering Definitions	17
C.	SOFTWARE ENGINEERING – YESTERDAY AND TODAY	22
1.	Origins	22
2.	Software Engineering - Education and Industry	23
3.	Pro and Con	24
D.	SOFTWARE ENGINEERING CHALLENGES	27
E.	PERSONAL OBSERVATIONS	28
III.	THE ENGINEERING CONTEXT	31
A.	PURPOSE OF THIS CHAPTER	31
1.	Establishing Context	31

	2.	Terminology.....	31
B.		CHAPTER THESIS	31
C.		ENGINEERING GOALS.....	32
	1.	Absence of Ambiguity.....	32
	2.	Predictability	33
	3.	Absence of Failure.....	33
	4.	Serviceability	34
	5.	Economic Feasibility	35
	6.	Summary of Goals.....	35
D.		ENGINEERING DEFINITIONS	35
	1.	Current Engineering Definitions	36
		a. ABET.....	36
		b. Shaw and Garlan	37
		c. Rogers.....	38
		d. Florman.....	38
		e. Wright.....	38
E.		SUMMARY OF CURRENT DEFINITIONS	39
F.		TRADITIONAL VIEW IS TOO NARROW	40
G.		NEEDED: AN UPDATED DEFINITION	40
	1.	Improving the Definition.....	41
		a. Current Definitions	41
		b. The Engineer.....	42
		c. The Contemporary World of Engineering	43
		d. A Revised Definition of Engineering	43
	2.	Summarizing the Updated Definition	44
IV.		ELEMENTS OF THE ENGINEERING CONTEXT?.....	47
	A.	INTRODUCTION.....	47
	B.	DESIGN	47
	C.	THE ROLE OF KNOWLEDGE	48
		1. Settled/Dependable Knowledge	48
		2. Knowledge from Science	49
		3. Knowledge from Engineering Experience.....	50
	D.	FORCE(S).....	51
		1. The Concept of Force.....	51
		2. Non-Physical Forces.....	52
		3. Forces of Nature.....	52
		4. Conflicting Forces	54
	E.	RISK MANAGEMENT AND CONTROLS	55
	F.	PREDICTABILITY (PREDICTABLE OUTCOME).....	55
	G.	CONSTRAINTS AND TOLERANCES	58
	H.	ECONOMICS	58
	I.	STATE TRANSITIONS AND DISPLACEMENT	59
	J.	OTHER ASPECTS OF ENGINEERING.....	59
	K.	CHAPTER SUMMARY.....	60
V.		SOFTWARE ENGINEERING AS ENGINEERING.....	63

A.	INTRODUCTION.....	63
B.	PURPOSE OF THIS CHAPTER	63
C.	CHAPTER THESIS STATEMENT	64
D.	SOFTWARE CONTEXT	64
	1. The Algorithm Issue	64
	2. Change	66
	3. The Management of State Changes.....	67
E.	SOFTWARE IN PRACTICE	68
	1. Software Architecture	68
	2. Software Engineering	69
	3. Programming.....	69
	4. Supporting Activities	70
F.	THE ENGINEERING CONTEXT – A REVIEW	70
	1. Nature (Natural Forces)	71
	2. Intersection of Software and Physical Engineering.....	73
G.	MAPPING SOFTWARE ENGINEERING TO ENGINEERING	73
	1. Forces	74
	a. <i>Conflicting Forces</i>	74
	b. <i>Conflicting Forces: An Example</i>	76
	2. Dependable (Settled) Knowledge.....	77
	a. <i>Software Engineering Knowledge</i>	78
	b. <i>Measurement and Metrics</i>	79
	c. <i>Software Engineering Knowledge (SWEBOK)</i>	84
	3. Design to Tolerances.....	85
	4. Controls for Failure Prevention	85
H.	WHAT WE CANNOT YET ENGINEER IN SOFTWARE	86
	1. Computer Programming.....	87
	2. Risk, Testing, and Quality Management	87
	3. Human Factors.....	88
	4. The Ideal Process	88
VI.	DESIGN METRICS: DESIGNING TO TOLERANCES?	89
A.	INTRODUCTION.....	89
B.	CHAPTER THESIS STATEMENT	89
C.	TOLERANCES, CONSTRAINTS, AND CONTROLS	90
	1. Design Issues.....	90
	2. Nature of Design Tolerances	91
	a. <i>Software Tolerance Properties</i>	91
	3. Categories of Software Tolerance.....	92
	4. Software Engineering Enabling Mechanisms	94
	5. Design Tolerances in Practice	99
	6. Fault-Tolerant Design.....	99
	7. Tolerances in Software	100
	8. Modeling for Tolerances.....	101
D.	ADDITIONAL SOFTWARE TOLERANCES	101
	1. Historical Perspective	101

2.	Assertions.....	101
3.	Constraints.....	102
	<i>a. Data Constraints</i>	103
	<i>b. Behavioral Constraints</i>	104
	<i>c. Contract Constraints</i>	104
4.	Language and Tool Support	104
E.	UNLIKELY SOFTWARE DESIGN TOLERANCES	105
F.	CHAPTER SUMMARY.....	106
VII.	SUMMARY AND FUTURE WORK.....	107
	A. DISSERTATION SUMMARY	107
	B. FUTURE WORK.....	109
	BIBLIOGRAPHY.....	111
	INITIAL DISTRIBUTION LIST	121

LIST OF FIGURES

Figure 1.	Specific Contribution Summary	8
Figure 2.	Engineering Process.....	19
Figure 3.	Revised Definition of Engineering	44
Figure 4.	Intersection of Non-Engineering and Engineering	53
Figure 5.	Engineering , Software Engineering and Nature	53
Figure 6.	Engineering Definition.....	71
Figure 7.	Restricted Integer	76
Figure 8.	Opaque Restricted Integer.....	76
Figure 9.	ADT Stack Object Package Example	95
Figure 10.	ADT Stack Type Example	96
Figure 11.	ADT Own_Integer Example	97
Figure 12.	Generic Integer ADT Example	97
Figure 13.	Floating-point ADT Example	98

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1.	Survey of Practicing Engineers.....	11
Table 2.	Tentative Matrix for Evaluating Maturity of Emerging Engineering Discipline	61

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

The many people who contributed to this work include my dissertation committee, my students, many colleagues from a variety of disciplines.

My committee Chair, Dr. Peter Denning, set the guidelines and constraints for this work, and gave me the benefit of his insight and experience. Dr. Bret Michael, my dissertation supervisor, understood early the importance of what I was attempting to accomplish. Dr. Mikhail Auguston continually provided me with articles from scholarly journals, which enhanced my research and my thinking. Dr. Man-tak Shing provided frequent guidance, intellectual support and encouragement. Dr. Qiaoyun Li gave me well-reasoned feedback whenever I needed her advice. Without Dr. Dan Boger, I would not have had the opportunity to pursue this project at all. He is the person who took the risk of inviting me to be a member of the NPS faculty. I am also thankful to Dr. Luqi for her enduring faith in me. I must also thank Dr. Valdis Berzins for his sometimes annoying habit of finding some flaw in my reasoning just about the time I had some good idea nailed-down. I also need to thank my NPS colleague, Professor Charles Calvano.

I have many students from my software engineering classes to thank. In the foundry of the classroom, I was able to test some of my ideas and measure student reactions to them. This work would not have been possible without the contributions of many engineering and software professionals who have written copiously about topics in both engineering and software. They are too numerous to name here. Many appear in my cited bibliography and others appear in my “works consulted” section.

Finally, my family has been supportive throughout this entire process. Without the encouragement of my wife, Sera Hirasuna, as well as her superb editing skills, this work would not have been possible.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION AND OVERVIEW

Samuel Florman writes, “We can examine engineering indefinitely without coming close to exhausting its possibilities, its meanings and the way people feel about it” [Flo96, p. 117]. This document is limited to some of what we regard as valuable about engineering to satisfy the intent of the thesis, not as a comprehensive exposition of everything there is to know about engineering. Robert Glass tells us that much of what has been written about software engineering is more about software than about engineering. He counsels us that there is a need for an engineering reference discipline for software engineering [GVR02]. This suggestion should hold true for all emerging engineering disciplines. This thesis is written with the engineering reference discipline as its foundation. Fritz Bauer informs us that, “Software engineering is that part of computer science that is too difficult for computer scientists” [Bau73, p. 553].

A. THESIS ORGANIZATION

There are seven chapters in this work. The first chapter introduces the topic and the problem being solved. It also includes some of the conclusions and discussion of the methodology used to arrive at those conclusions. Chapter II is a review of relevant literature. The third and fourth chapters are important because they set the engineering context and provide details about that context. Chapter V builds on the foundation from Chapters III and IV to evaluate software practices as engineering practices. Chapter VI exemplifies the principles from Chapter V with the notion of design metrics along with an emphasis on “designing to tolerances,” This example illustrates on-going challenge in software engineering. Finally, the last chapter (Chapter VII) discusses some of the future work that will derive from this work.

B. WHAT PROBLEM ARE WE TRYING TO SOLVE?

1. Problem Sources

In a conversation with Peter Denning, he noted his long-standing effort to overcome the objections in the traditional science community for the claim of computer science as a legitimate branch of science. The same concern manifests itself in the engineering community in any discussion of software engineering as a legitimate branch of engineering.

The research leading to the present form of this work has required reading the viewpoints of the leading thinkers in the world of software along with many books, papers, and journals on other engineering disciplines. However, opinion is not a sufficient foundation to satisfy the vision or the goals of my own work. Therefore, it has been necessary to pursue a formulation of a view of software engineering that conforms to the more rigorous set of concepts of classical engineering. Those concepts include the principles, methods, and practices that characterize conventional engineering in the physical world. From this foundation, we can develop an inquiry into whether emerging disciplines such as software engineering deserve to be regarded as legitimate engineering disciplines.

We will see in this discussion that many traditional engineers acknowledge that it would be nice if software could be engineered, but argue that there is little indication of engineering in what they see of current software practice. To some extent, this is because many practicing engineers are unfamiliar with the progress being made in the application of engineering practices in the improvement of software practice. However, those engineers are correct in their observation that the vast majority of software development and management falls short of what they would accept as engineering. In fact, some software developers also reject the notion of software being engineered. There is even a trend away from engineering that has alarmed some in the software engineering community. This trend is described in *Balancing Agility with Discipline* [Boehm2003]. Although much of software engineering practice has failed to live up to its promise, that failure does not give us license to abandon the pursuit of an engineering foundation for its

future success. On the other hand, prominent members of the software engineering community such as Phillippe Kruchten and others have suggested that the attempt to find a correspondence between software engineering and conventional engineering may be a futile pursuit due to the unique properties of software [Kru04], [LEW00], [Bry00].

The observations from the previous paragraph, lead to several questions that we attempt to address in this dissertation:

1. Is it necessary or appropriate to regard any aspect of software practice as an engineering activity?
2. What aspects of software practice qualify for the application of engineering principles, methods, and practices?
3. Can engineering principles, methods, and practices be applied where there is a seeming absence of the constraints of natural forces?
4. Which engineering principles, methods, and practices inform and support the claim that some aspects of software practice can be engineered?

2. Bounding the Problem

It is important to note that our approach to software engineering is not a restatement of the many process-oriented models that have characterized so much of the engineering literature and practice of the last thirty-plus years. Rather, we are concerned with an engineering model for software practice that is independent of process models. While process models are important for successful engineering, they are only part of the story.

One concern in this work is to discover how engineering is demarcated from non-engineering. Once this demarcation is established, it will be possible to investigate the possibilities and the limitations of engineering software solutions to real problems. In this respect, this dissertation is unique and original. We will investigate how this demarcation can be framed in Chapter III and Chapter IV of this work.

Even as we strive to present a more definitive context for evaluating software engineering as an engineering discipline, it has become clear that our conclusions will leave more questions unanswered than answered.

We focus on the engineering problems in software, not programming problems. The engineering problems involve the many relationships between the components of a software environment. The engineering problems for software are not physical engineering problems unless one thinks of time as a problem in physical engineering.

A good historical example of a software engineering problem that was solved as a software engineering problem was Dr. Denning's work on thrashing [Den68]. While there was a physical component to the problem, it was also a software engineering problem. Similar engineering solutions are represented with the Rate Monotonic Model (RMA) developed at the Software Engineering Institute and Carnegie-Mellon University [Liu73]. RMA, along with other scheduling algorithms for real-time, concurrent software systems, has been the topic of considerable research. Schedulability for real-time software promises to become one of the most important topics for future computing as the configuration of the hardware architectures moves more and more toward multi-processing models. The problem of schedulability is one place where operations research, systems engineering, and software engineering intersect as a multi-discipline engineering problem that has little to do with so-called natural forces.

As software becomes more intrusive in the creation of devices that society requires for both commerce and entertainment, the engineering problems associated with software become more pervasive and more difficult. To suggest that these software problems do not represent engineering is to ignore a pressing reality. Even so, much of the software being produced is not engineered in the usual sense of engineering. The burden of the engineering details is often deferred to the lowest level of the development process: computer programming. While we have been able to tolerate this downward shift of responsibility in earlier software systems, it is essential that the software community reassert the need for more dependable approaches as society becomes increasingly dependent on software, not merely for commerce and entertainment, but for its very survival. An engineering imperative for software engineering is not a nicety. It is an essential part of the future of software practice.

C. CONTRIBUTION SUMMARY

1. The Dual Path

The original contribution of this dissertation follows a dual path from both traditional engineering and software science in an attempt to merge those two paths into a common discipline where the principles and practices of traditional engineering drive the development, management, maintenance, evolution, and utilization of software engineering products. We are developing a model that invigorates the notion of a software engineering culture [Wei96] as well as a model for software engineering practice.

In the previous paragraph, we emphasize sufficiency of the intersection. We must also ask whether there are properties of engineering not present in software practice that prevent it from being considered to be an engineering practice. Is software practice a craft rather than an engineering practice? Those who regard software practice as purely a matter of computer programming are probably correct in their “craftsmanship” view [McB02]. On the other hand, experienced software practitioners understand that programming is only a small part of software practice. As noted earlier, Kruchten suggests that attempting to overlay classical engineering on software is the wrong approach [Kru04]. In his view, “...we tried to impose techniques from other engineering disciplines onto software development models without understanding the real nature of software.” We agree with Kruchten up to a point, but accept the proposition that some aspects of software practice are also an engineering practice. That is the underlying vision of this dissertation, and the proposition we will attempt to support.

2. The Engineering Path

Modern engineering started in the middle of the Nineteenth Century and rapidly matured in the early years of the 20th century with the application of more rigorous mathematics and science along with new discoveries in the world of natural phenomena

[B182], [Di80], [Hol08], [RaW63], [Wri89]. Toward the middle of the 20th century, engineering diversified into a large number of sub-disciplines, and that diversification continues in modern times.

The rigor and discipline required of engineering makes entrance into the engineering community difficult to achieve. This is as it should be since the responsibilities of the professional engineer often involve the safety and security of the larger public – a public that uses the products and services developed by engineers. Engineering is largely about developing solutions characterized by predictability -- predictable operation, predictable life-cycle, predictable maintainability, and the absence of surprise [Rie05].

3. The Software Science Path

Software science is a subset of computer science. The hardware side of computer science is heavily dependent on physical engineering, especially electrical engineering. This corresponds to and supports the widespread belief by many contemporary engineers that engineering, to be credible, must involve the world of natural phenomena and natural forces. Few engineers would argue against computer hardware design as engineering. Many of those same advocates of computer hardware as an engineering discipline would argue with some ferocity that software design is not engineering.

The software science path, originally represented by computer programming, is a different story. Software science includes a wide range of disciplines such as automata theory, formal methods, design-by-contract, design metrics, large-scale systems design, software architecture, parallel computing, computer programming, compiler design, applications development, operating systems design, and more.

4. The Converging Path

This dissertation seeks a converging path where engineering can merge with software practice to evolve toward a more rigorous engineering model for software. That converging path will legitimize the convergence of software engineering and standard engineering and inform efforts to map emerging engineering disciplines to a model that

helps them come into their own as engineering practices. Such a convergence needs to be derived from the best of both of the two paths: the accepted concepts of engineering combined with the best contributions of software science. It also learns from the failures, misunderstandings, user frustrations, and other experiences that, when studied as lessons learned, contribute to the progress of all aspects of software practice.

To create a new path, it is necessary to examine the principles and practices that represent both traditional engineering and software engineering and find commonality between them. To many, in both software and traditional engineering, the software engineering path is properly a subset of software science (ergo, a sub-sub-set of computer science). In this dissertation, we will try to present an alternative perspective. Software engineering has many threads of its own including risk management, scheduling algorithms, software reuse, economic analysis, control theory, software quality assurance, project and process management, and software modeling. Not everyone agrees that these are the ideal topics for software engineering [Gla03, pp. 46-47] suggests that reuse is too difficult a problem to be of general use in software engineering.

We do not argue that all of software practice is engineering. Instead we identify a few areas that clearly involve engineering, while noting that some (perhaps many) software practices fall short of engineering, in their present form and practice. We present the following four questions and attempt to address them in this work.


1. How is engineering different from non-engineering? This includes guidelines to *demarcate* engineering from non-engineering.
2. What is required of software practice if it hopes to be accepted in the contemporary engineering community as an engineering practice?
3. What current software practices that do qualify as engineering and how do we justify those which, while required in software practice, fail to conform to engineering practices?
4. How can we develop a model/framework for software engineering through convergence of multiple disciplinary paths – engineering, computer science, software science, and mathematical science – into a single path that represents the kind of engineering discipline necessary for the future of software practice in modern society?

These concerns lead to two important scenarios:

1. Software engineering becomes accepted by the main engineering community as a respected engineering discipline, and
2. Well-engineered software combines the best of mainstream engineering and software practices, so a society that depends more and more on software is provided the best products and services possible. In other engineering practices, this concern is the province of the Professional Engineer (PE). However, the PE model is not yet part of software engineering practice.

5. The Contribution Summary

The following chart provides a summary view of the principle contributions:



Specific Contribution Summary

Reframing the argument; not settling the argument

Contribution	Importance to DoD	Importance to Software Engineering
1. Demarcating definition for engineering in 21st Century	Assists in future decisions by program managers dealing with DoD contractors	Opens a dialogue based on a more comprehensive concept of engineering
2. Conforming definition for software engineering	Leads to uniformity of decision-making in future contract negotiations	Demonstrates how software engineering can become more of an engineering discipline
3. Map software engineering to engineering	Helps defuse arguments about engineering versus software engineering	Contribute to understanding of SWE as an engineering discipline
4. Tying back individual responsibilities to the definition	Clarifies responsibilities and reinforces the need for better engineering practice relative to software	Helps refine understanding of the relationship of these three responsibilities
5. Demonstrate how engineering, as defined above, has already contributed to software engineering	Ensures that other contributions are grounded in real-life examples to parry argument about SWE	The first of many that need to be identified as actual engineering tasks instead of programming tasks.

Unclassified *September 1, 2008* 6

Figure 1. Specific Contribution Summary

D. METHODOLOGICAL APPROACH

1. Assumptions and Constraints

a. The Early Pioneer's Vision

The original report from the 1969 NATO working meeting where the term *software engineering* was officially coined, described software engineering as, “the application of engineering to the creation of software” [NaR69]. At that time, this was a vision, not a fact. That vision was in reaction to a perceived *software crisis* and the urgency to do something about that crisis. Ever since the NATO meeting, people in the software community have been striving to satisfy that vision. It is still more of an ideal vision than an accomplished reality. Many believe that the closer we can get to realizing that original vision, the more effective software practitioners will be at producing software that is as dependable as the products and services provided through traditional engineering disciplines.

b. Software Process

The focus of this dissertation is different from earlier works regarding software engineering in that we go beyond the traditional topics of software engineering as listed in the recent SWEBOK documents [Moo06]. Those documents tend to reinforce the *process* view of software engineering. The process approach to software engineering is what Agresti, Bauer and others have characterized as the Industrial Engineering approach to software engineering [Agr81]. While we affirm the progress that has been made with the process approach, we do not want to repeat, nor even evaluate the process literature. Instead, we seek an engineering foundation more closely aligned with the properties of classical engineering disciplines [Tek04].

2. Research Approach

a. Literature Review

We examined many books, journals, scholarly papers and treatises related to software engineering. We also conducted a small survey of engineers from outside the software field. We also looked at a large number of definitions of software engineering, and found all of them lacking.

This led to the one of the central questions of this dissertation, “What is engineering?” Without a clear and unambiguous answer to that question, we concluded that it is difficult to get a clear idea of what we mean by software engineering. Therefore, one of the first tasks in developing an engineering context for software engineering was to develop a definition for engineering.

This question can also be stated as, “What demarcates engineering from other creational activities?” It seems odd that, just as no one has a sufficiently rigorous definition of software engineering, none of the published definitions for engineering were sufficiently crisp to demarcate engineering from other practices. Engineers seemed to be caught up in the “I’ll know it when I see it,” mentality, and even the ABET definition fell short of what was required.

b. Survey of Engineers

In addition to a literature search, I sent out survey forms to engineers from many disciplines to gather a representative sample of opinions and viewpoints.

The main questions, using a Likert Scale, on the survey are shown below. The questions were augmented with demographic questions such as what kind of relationship the respondent has with software practice or engineering.

1.	Real engineering depends on or requires natural forces (e.g., physics, chemistry, etc.).
	Strongly Agree Agree No Opinion Disagree Strongly Disagree
2.	Natural forces, for engineering purposes, includes solutions to timing problems.
	Strongly Agree Agree No Opinion Disagree Strongly Disagree
3.	Software engineering falls short of the requirement for natural forces.
	Strongly Agree Agree No Opinion Disagree Strongly Disagree
4.	Software development is too complex to be engineered.
	Strongly Agree Agree No Opinion Disagree Strongly Disagree
5.	Software is more concerned with computer programming than with engineering.
	Strongly Agree Agree No Opinion Disagree Strongly Disagree
6.	Software engineering is a legitimate branch of engineering.
	Strongly Agree Agree No Opinion Disagree Strongly Disagree
7.	Software people are not as well prepared for engineering as traditional engineers.
	Strongly Agree Agree No Opinion Disagree Strongly Disagree
8.	Software engineers should be licensed just as other engineers are licensed.
	Strongly Agree Agree No Opinion Disagree Strongly Disagree
9.	For embedded systems (radar, avionics, etc.), software is a part of the engineering solution.
	Strongly Agree Agree No Opinion Disagree Strongly Disagree
10.	Future software developers must find a way to apply more engineering to their practice.
	Strongly Agree Agree No Opinion Disagree Strongly Disagree

Table 1. Survey of Practicing Engineers

From the questionnaire (above) most engineers in the classical engineering disciplines responded to the question (1) about natural forces with Strongly Agree or Agree. This corresponded with my one-on-one interviews with other engineers as well as with the ABET viewpoint. It is clear that those involved in conventional engineering place a high value on the presence of and ability to predict outcomes based on the knowledge of natural phenomena.

For question Six, where we ask whether software engineering is a legitimate branch of engineering, most engineers responded with Disagree. This was no surprise given the responses to question number One. The other significant response was to question number Ten where we ask about the need for a future model of software

practice that does conform to engineering. On this question, the Agree and Strongly Agree were overwhelming even from those who disagree that software engineering, as currently practiced, is anywhere close to being an engineering discipline.

c. Interviews

In some cases, the research was conducted as interviews on Internet Usenet sites. Other interviews were conducted in-person. Most engineers interviewed were agreed in their view that 1) engineering requires physical forces, 2) software engineering is not really engineering, but 3) it would be good if software could be engineered with the same rigor expected of other kinds of engineering.

I also interviewed software practitioners. During those interviews, I found that, the more experienced the software developer, the more pessimistic they are about any chance of software practice being a real engineering discipline. There was near universal agreement that software professionals are not as well prepared for engineering as traditional engineers. This reflects the reality that few computer science graduates have any education or training in engineering during their undergraduate or graduate programs. An increasing number of software developers seem to reject the notion of an engineering model for software, opting instead for the so-called “agile” approaches [McB02], [Coc06], [Bec00].

d. Reading

In pursuit of providing an engineering context for software engineering, I read a large number of books and papers on general engineering, some of which will be cited or quoted throughout the dissertation. I also examined the curricula of several engineering programs such as that at San Jose State University. The reading part of the research included papers both praising and disparaging the concept of software engineering.

The paper written by Robert Glass [GVR02] was mentioned earlier. The key point of that paper was the need for an engineering *reference discipline* for software engineering. If there was any paper that inspired this document, that was it. The concept of a *reference discipline* is a kind of *touchstone*, throughout this work.

e. Developed Examples

Some software engineering concepts are best demonstrated with examples. Unfortunately, examples sometimes have to be in the form of computer programs. We include some programming examples to illustrate a few of the ideas, coded primarily in either Ada or SPARK. Most of the programs are my own creation. Other examples will cite real projects where source code is only incidental to the actual software engineering.

The fact that coded examples are included in the dissertation should not be construed to mean that software engineering is about programming. That is only one aspect of software engineering practice, often not the most important. However, without the existence of source code there would be no software products. An engineering design must ultimately have a realization or it is little value to anyone.

E. EVALUATION ISSUES

Many practicing engineers believe the current engineering models and definitions are adequate. This is yet another place where this dissertation makes a contribution. Engineering, from its earliest beginnings, has constantly evolved to include new practices, new views of itself, and entirely new kinds of engineering. It has also diversified into large variety of sub-specialties. One can see examples of this diversification where civil engineering involves the specialty of structural engineering, or chemical engineering propagates to problems in combustion engineering. In some cases, diversification, rather than separating disciplines, actually consolidates existing practices into an umbrella of engineering disciplines such as systems engineering.

The universally accepted concepts of traditional engineering can also provide a *reference discipline* for software engineering. Specific kinds of engineering give us a more fine-grained view of the relationship of software engineering to general

engineering. Some of the important reference disciplines for software engineering include industrial engineering, systems engineering, process engineering, control engineering, and electrical engineering.

We recognize that engineering is overkill for many software projects, but insist that some aspects of engineering are essential, even demanded, for development of large-scale, complex, safety-critical, and mission-critical projects.

This dissertation will require us to present examples from other branches of engineering to show which software engineering practices are really engineering. To make the case, we will also find it necessary to reframe and broaden the definition of engineering, even as we narrow the definition of software engineering. It is important that those aspects of software practice that conform to engineering be separated from those software practices that fall short of real engineering discipline (e.g., those that are more like craftsmanship than engineering).

II. SURVEY OF PREVIOUS WORK

A. INTRODUCTION TO SURVEY

1. The Larger Literature

The body of literature about software engineering is large. Much of that literature is an assessment of what is wrong with software and how it can be made better [Bro95], [Boe84], [JeT79], [Jon96], [McB02], [MaR05], [Pfl9191], [GJ96]. The principle focus in most of these works is on processes, methods (e.g., waterfall, spiral, agile, etc.), tools, schedules, and the management of people. While these are important considerations, they are not sufficient to establish software engineering as a credible engineering discipline. The published work is full of advice about how to improve some aspect of software engineering practice such as processes, project management, estimation, risk management, better use of humans, improvement in tools, approaches to automated methods, and after-the-fact metrics, but very little about engineering.

2. Process Literature

Much of what has been published about software engineering is process or project-oriented. For example, the highly influential book by Frederick Brooks [Bro95] emphasizes the importance of people and process in the management of software. Humphrey provides several works that advance the notion of process in software engineering. In his first book, *Managing the Software Process* [Hum89], identified five levels of software process and launched a new area of practice referred to as *process improvement*. In subsequent works, Humphreys refined the scope of his process contribution to focus on the responsibilities of individual software engineers [Hum95], and software engineering teams [Hum06]. Current trends in software practice emphasize processes.

3. Observations from the Literature Review

a. Observation One

Some of the observations from Glass [GVR02], et al., include:

Regarding reference disciplines, [Software Engineering] research seldom relies on other disciplines as the basis for its work. Although there have been discussions, over the years, of the relationship between SE research and such fields as cognitive psychology, quality, engineering, and manufacturing, at this point in time there is little evidence that SE seeks to assimilate learning from other fields.

and

For the most part SE research eschews reliance on other fields for its fundamental theories and/or concepts. Ninety-eight percent of the papers examined had no reference discipline.

also, when discussing software engineering in the same paper,

It is interesting that there was no reliance on such fields as Mathematics, Engineering, or any of the Sciences... it is clear that SE research tends to be quite self-contained, not relying on any other disciplines for its thinking.

b. Observation Two

When the question, “Is software engineering really engineering?” is asked, it is often answered with, “No!,” or “Not Yet.” Software professionals such as McConnell, quoted earlier in this dissertation [McC04, p. 30], answers, “No” to the question, but qualifies his answer with, “Professional software development should be engineering. Is it? No. But should it be? Unquestionably, yes.” Software engineering educator, Mary Shaw notes that software practice could someday be engineering, but it is not there yet [Sha90].

Other software professionals, those of my personal acquaintance, call themselves software engineers, and write about software engineering, but concede that we are on shaky ground, and are almost apologetic, when comparing software engineering to generally accepted engineering disciplines.

c. Observation Three

In “Finding a History for Software Engineering,” Michael Mahoney [Mah04] provides a comprehensive treatment of the many stages of software engineering practice. From Mahoney’s work, it becomes clear that much of what has passed for software engineering has been heavily influenced by industrial engineering.

B. SOFTWARE ENGINEERING DEFINITIONS

A survey of the literature and previous work would not be complete without recognizing some of definitions that have been proposed during the nearly four decades since the term was coined. So many have been proposed that only a representative sample is possible in this chapter. The author of each software engineering textbook consulted for this research includes a definition, usually in an early chapter. Other authors also include definitions of software engineering as part of articles, general books on software, and books on object technology. For this section, we select a few representative definitions.

1. Representative Software Engineering Definitions

As noted earlier, there is no shortage of definitions for software engineering. There is a shortage of definitions that conform to standard definitions of engineering. As mentioned in the first chapter, the original vision of software engineering from Naur, et al., sought an engineering approach for a definition. Although he does not define what he means by “sound engineering principles,” he offers the following:

Software engineering is the establishment of sound engineering principles in order to obtain economical software that is reliable and works efficiently on real machines [NaR69].

Shaw [Sha90] laments that this is more of a wish for an engineering model of software than an actual description of it.

As far back as 1975, Ross, et al., [RGI80] wrote a paper where they tried to define software engineering by describing the goals and principles of software engineering. They listed the goals as: reliability, modifiability, efficiency, understandability. We have chosen to restate and refine those goals in Chapter III of this dissertation as: 1) absence of ambiguity, 2) predictability, 3) absence of failure, 4) serviceability, and 5) economic feasibility. Those authors in that same article go on to specify the principles of software engineering as: abstraction, information hiding, modularity, localization, uniformity, completeness, confirmability. An entire treatise could be written on each of these principles, but that is beyond the scope or intent of this dissertation. For a good discussion of the work of Ross, et al., we recommend reading the detailed commentary by Grady Booch [Boo94, pp. 17-25].

The goals and principles in the Ross, et al., article are compatible with the goals and principles of any good engineering practice, not just software engineering, especially when goals such as measurability and economics are included. The problem with the article is that it does not provide a rigorous definition within an engineering *reference discipline*.

McConnell does reference classical engineering in his recent work on software practice with,

Some ... object that commercial software is too dependent on changing market conditions to permit careful, time-consuming engineering ... The ... objections are based on narrow and mistaken ideas of engineering. Engineering is the application of scientific principles toward practical ends [McC04, p. 33].

While McConnell implicitly acknowledges the *reference disciplines*, his notion of engineering is not quite rigorous enough to stand against the scrutiny of engineers from other disciplines. This is an on-going problem with much of what is written about software engineering. In a later section, McConnell lays out the problem in a different way, one that is more compatible with the goals of this dissertation. He asks whether the

practice should be called software *engineering* or *software engineering*, where his italics imply a difference of emphasis [McC04, p. 182]. In one case, he says we have engineers who are software apprentices. In the other, we have software practitioners (programmers, etc.) who are engineering apprentices. The software engineering program at McMaster's University in Canada, formerly under the direction of David Parnas, teaches engineering first, then software with goal of eliminating the apprentice model with education in both disciplines [Par99].

In their textbook, Peters and Pedrycz state that, software engineering is:

... a practical, orderly, and measured development of software. The principle aim is to produce satisfactory systems on time and within budget. ... The engineering approach is practical because it is based on proven methods and practices in software development [PeP00], Chapter 1.3].

The Institute of Electrical and Electronics Engineers (IEEE90) defines software engineering in its publication numbered, IEEE Std. 610.12, as:

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software, that is, the application of engineering to software. (2) The study of approaches as in (1)

Jim Moore, in a book published by IEEE, elaborates on the concept of engineering with, “engineering can be viewed as a closed feedback loop ... An engineering process consists of related activities performed in response to a statement of needs and consuming resources to produce a product” [Moo06, p. 4], In support of his view, Moore includes the following compelling diagram.

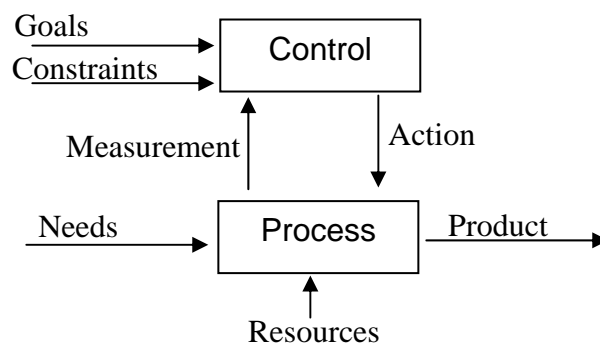


Figure 2. Engineering Process

This important diagram illustrates the role of control in engineering. Control is one of the most important facets of engineering, including software engineering. A large part of any design involves ensuring that the *construction* [Roy89, p. 79] of that design will satisfy the goals of engineering discussed earlier, and control is an essential element of every engineering design, including correctly designed software. Moore's work makes it clear that attention to developing appropriate controls must not be overlooked in the haste to produce a completed software product.

Moore recognizes that the techniques, including processes, for engineering software can be viewed, in part, as specialization of the general disciplines, such as project management, systems engineering, and quality management. He notes, "These contextual disciplines are important because ... software engineering standards must often be applied in conjunction with the standards from other disciplines." The inclusion of constraints in this diagram is essential for building a complete model of software engineering [Moo06, p. 5].

An added perspective on this same theme from Boehm suggests,

Software engineering is the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation to develop, operate, and maintain them. It is also known as Software Development or Software Production [Boe84].

One problem with Dr. Boehm's definition is that it fails to mention the relationship of software engineering to other engineering practices. Further, it implies that software engineering is more like computer programming instead of promoting it to the larger ideas that contribute to software engineering as an engineering practice. This is not representative of Dr. Boehm's actual understanding of software engineering since his professional contributions extend far beyond computer programming.

In one of its early documents, the IEEE defines software engineering as, "the systematic approach to the development, operation, maintenance, and retirement of software" [IEE90].

As with the other definitions, this one fails to satisfy us in our quest for an engineering vision of software practice. It could just as easily describe a COBOL programmer creating a printed report from a database. Richard Fairley presents one of the definitions reminiscent of industrial engineering practice,

... the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates [Fair85, p. 2].

Fairley's contribution represents what is sometimes called Taylorism [Tay11]. Frederick Winslow Taylor was a founder of what he called “scientific management.” Scientific management eventually evolved into modern Industrial Engineering. The recent reaction against Taylorism among software professionals has motivated the growing interest in so-called Agile software development processes [Coc02], [StP03], [Sch02], [Bec00]. The criticism from Beck, Schwaber, Cockburn, et al., has focused on abandoning the application of Taylor’s principles of industrial engineering. In fact, others could reasonably make the argument that most of software engineering best practices are simply a subset of modern industrial engineering rather than a subset of computer science [Agr81]. In that case, computer science would be the underlying science, but industrial engineering would be the engineering practice. One problem with this is that the list of accepted engineering practices, for Professional Engineering certification, often excludes industrial engineering. A refreshing disclaimer is found in Roger Pressman’s discussion of a definition for software engineering,

... almost every reader will be tempted to add to this definition. It says little about the technical aspects of software quality; it does not directly address the need for customer satisfaction or timely product delivery; it omits mention of the importance of measurement and metrics; it does not state the importance of a mature process. And yet Bauer’s definition provides us with a baseline. What are the “sound engineering principles” that can be applied to computer software development? How to “economically” build software so that it is “reliable”? What is required to create computer programs that work 'efficiently' on not one but many different 'real' machines:? These are the questions that continue to challenge software engineers [Pre05].

The following definition, not framed in terms of an engineering *reference discipline*, from a textbook on software engineering by Pfleeger,

Designing and developing high-quality software. Application of computer science techniques to a variety of problems. We are problem-solvers rather than theoreticians [Pfl91].

It would be possible to fill many pages with definitions of software engineering. This representative sample of attempted software engineering definitions should be sufficient to make clear that many definitions fall short of what is required to actually make the case for, or even establish the context for, software engineering as an engineering discipline, especially when seeking a definition framed in the context of an engineering *reference discipline*.

Consequently, for a more engineering-oriented definition of software engineering, it is necessary to examine engineering itself and determine whether a definition for software engineering can be developed in terms of an engineering *reference discipline*. Is there a definition for engineering that encompasses mechanical, chemical, electrical, civil and also includes emerging engineering disciplines such as software engineering? Can that definition be useful in support of a model of software engineering that can be agreed upon in both the engineering community and software engineering community?

C. SOFTWARE ENGINEERING – YESTERDAY AND TODAY

1. Origins

The term software engineering originated in a 1967 initiative that resulted in a NATO sponsored conference in Garmisch, Germany. The theme of the conference was described as follows:

In late 1967 the Study Group recommended the holding of a working conference on Software Engineering. The phrase ‘software engineering’ was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering [NaR69].

Naur's use of the word "manufacture" does not match what many software practitioners would use. Engineering is not the same as manufacturing. Rather, it is more focused on design. The above summary from Naur has been restated as: "... the application of engineering to software ..." [IEEE Std 610.12].

2. Software Engineering - Education and Industry

The results of a Google search shows that software engineering is now a part of the graduate curriculum in a growing number of universities. Some of the undergraduate programs are ABET accredited. Often, software engineering is in the engineering school instead of the computer science department [Fre98]. Even so, a review of the curricula on university web sites indicates that students majoring in undergraduate software engineering are more often educated in computer science than in engineering. This is somewhat analogous to a scenario in which chemical engineers would be educated in chemistry and have to learn the engineering as a self-study exercise.

Graduate programs in software engineering typically reside in the computer science departments rather than the schools of engineering. Parnas suggests that software engineering is an "unconsummated marriage" where the science of programming and the disciplines of engineering have never been able to find compatibility [Par99]. After exploring the explicitly stated question "What is Engineering," Maibaum writes:

If we take the above characteristics of engineering and engineering knowledge and apply it to software engineering curricula, we quickly come to realize that the latter do not generally transmit knowledge and skills which prepare our students to be professional engineers [Mai97].

Maibaum goes on to observe:

Most of the material presented in such courses either suffers from confusing craftsmanship and engineering or does not take cognizance of the difference between mathematics/science and engineering [Mai97].

3. Pro and Con

Many articles have been published that purport to make the case for software engineering. Others, however, have denounced the notion of software engineering almost characterizing it as the equivalent of “snake oil.” There is no evidence that anyone has successfully made the case for software engineering with definitive, irrefutable examples. It is difficult to find a clear mapping of those aspects of software practice that correspond to “real” engineering and those which we are obliged to exclude.

The following statement originates with Mary Shaw, of Carnegie-Mellon University.

Software engineering is a label applied to a set of current practices for software development. Using the word *engineering* to describe this activity takes considerable liberty with the common use of that term [Sha96].

In an earlier work from the IEEE publication, *Software*, Shaw writes:

Although software engineering is not yet a true engineering discipline, it has the potential to become one. Older engineering fields are examined to ascertain the character that software engineering might have. The current state of software technology is discussed, covering information processing as an economic force, the growing role of software in critical applications, the maturity of development techniques, and the scientific basis for software engineering practice. Five basic steps that the software engineering profession must take to become a true engineering discipline are described. They are: understanding the nature of expertise, recognizing different ways to get information, encouraging routine practice, expecting professional specializations, and improving the coupling between science and commercial practice [Sha90].

Commenting on a conference in 1996 devoted to the history of software engineering, Cerruzi writes,

A 1996 conference on the history of software engineering ... came to the unintended conclusion that the attempt to establish software engineering on the whole had failed [Cer98, p. 105].

In a paper on the History of Software Engineering by Mahoney, we have the further damning conclusion.

Software engineering... practitioners disagree on what software engineering is, although most of them freely confess that, whatever it is, it is not (yet) an engineering discipline [Mah02], [Mah04].

Also, from the same article,

Software engineering began as a search for an engineering discipline on which to model the design and production of software. That the search continues after 35 years suggests that software may be fundamentally different from any of the artifacts or processes that have been the object of traditional branches of engineering [Mah02].

Mahoney goes on to cite an alternative to software engineering,

... at the 1969 NATO conference, I.P. Sharp came at the issue from an entirely different angle, arguing that one ought to think in terms of 'software architecture' (design), which would be the meeting ground for theory (computer science) and practice (software engineering) [Mah02].

This is a cogent and, at first, appropriate argument. However, it oversimplifies the difference between engineering and architecture. I will address the difference between software engineering and software architecture later in this dissertation.

There is widespread disagreement about the role of software engineering and its place in among academic disciplines. Some prefer that it be in the computer science department; others vote for inclusion in one of the engineering departments as part of the engineering curriculum. For example, from a conference keynote speech by David Parnas,

It is essential that those in *Software Engineering* learn more about classical engineering and that those in classical engineering recognize Software Engineering as a new branch of their profession [Par99].

In that same conference keynote, Parnas also states that:

A ... member of the software engineering profession should know that subset of computer science that is relevant to software design, but they must also have the knowledge of mathematics, and other sciences that are traditionally known by engineers... It is time that another such specialty, software engineering, be identified and defined [Par99].

A review of existing software engineering textbooks reveals that many of them correspond to the view expressed above.

This section would be incomplete if the recently published critique about software engineering were excluded. It is representative of many such viewpoints, which denounce the very notion of the application of engineering in the development and management of software. In an article in the *Communications of the ACM*, Wei-Lung Wang writes, “Beware the Engineering Metaphor.”

Fundamentally, engineering operates within the framework of the immutable laws of nature. These laws dictate the realm of engineering possibility, and engineers work by designing and constructing within the bounds of these laws. Their permanence and universality allow engineering principles, which signal the boundary of what is safely possible, to be established. For example, engineers who violate known electrical circuit guidelines are potentially breaching the physical limits imposed by the forces of electricity. These guidelines can be established simply because their applicability is universal. Competent engineers observe well-known limits that cannot be breached, while negligent engineers who fail to observe these limits are potentially negligent.

Software engineering, on the other hand, has no fixed framework in which to operate. At its heart, software is the embodiment of a Turing program [Wang02].

Wang’s view of engineering not only puts software engineering on the defensive as an engineering discipline, it also eliminates much of contemporary industrial engineering and some of systems engineering from its scope. Wang’s challenge requires us to develop a model of software engineering that satisfies his critique. Later, we describe some of the elements necessary to accomplish this task. Engineering, according to this point-of-view, must be based on the constraints of natural forces. Our position, in this dissertation, is that natural forces are not essential to engineering practice. His suggestion that software is nothing more than the “embodiment of a Turing program” suggests that he has not examined the full range of software engineering practices very thoroughly. Even so, it is statements such as this in the computing literature that feed the misperception that software engineering is not really engineering.

Natural forces are simply those forces that we directly or indirectly observe using the currently available measuring and observation tools. Engineering must also be concerned with logical forces. Logical forces are characterized by the state transitions that occur in a software entity. Abrial calls the discrete model of software transition systems [Abr04]. State transitions are a kind of action that is equivalent to the displacement actions in physical systems. One can make the case that the influence of logical forces (state transitions) in software engineering practice are as powerful as natural forces in mechanical engineering, although those logical forces may not be as immediate or easy to detect by a casual observer.

D. SOFTWARE ENGINEERING CHALLENGES

Software engineering's traditional emphasis on process, noted earlier, is one more motivation for seeking an alternative approach. We need a new focus for software engineering that is grounded in the reference discipline model recommended by Glass, et al. It is certainly true that every kind of engineering requires a disciplined process. However, even the best process, by itself, is not enough to justify calling a practice engineering. Consequently, this dissertation is not concerned with process as a central theme.

An engineering model for software engineering must be independent of whatever process is chosen. That is, Waterfall, Spiral, V-Model, Unified Process, and eXtreme Programming are all candidate processes that can be used for software engineering [Pre05], [PeP00], [Roy98]. As noted earlier, notwithstanding the progress in the domain of software engineering process, this thesis is not about process. It is only mentioned occasionally to reinforce the fact that our research did include this aspect of software engineering.

It is also important to admit that software engineering will not be entirely conformant to every aspect of every other kind of engineering. Software is unique in many ways. One of the most important of these is the heavy use of *conditional constructs* in a software product. The variety of alternative paths through a computer program increases as the number of conditional statements increases and it is impossible to trace

every path of a large program with many conditional statements. In this respect, software's most important capability, the source of its power, is arguably, also the greatest source of its errors. One of the most important responsibilities of the software engineer is to discover, assess, and either prevent or mitigate the risks – the potential for failure -- that devolve from the ability to develop these conditional constructs. This is the vital risk management and control responsibility that characterizes much of what we expect in other kinds of engineering [RieN07], [Flo87, p. 151], [Gla99].

As noted above, a fundamental idea in all of engineering is control. The risks alluded to in the previous paragraph demand that the software engineer understand the methods and principles of *controls* as a facet of engineering. One challenge for anyone in creating software is to understand how to create appropriate levels of control within a design abstraction. Recall that Moore's diagram also puts great emphasis on the importance of control [Moo06, p. 5].

Another aspect of software engineering that needs more emphasis is design metrics. Design metrics are common in other branches of engineering. Engineering has always had a quantitative/mathematical viewpoint toward its artifacts. Although this is sometimes seen as irrelevant by software practitioners, design metrics are as critical a part of software engineering as it is in other kinds of engineering [RieJ07].

E. PERSONAL OBSERVATIONS

Any written work will reflect the primary interests of its author. This dissertation is not different in that respect, particularly since this author has many years of experience with software and the software industry. The title of this dissertation and much of its content is influenced by my professional experience collaborating with engineers in other disciplines where software was an essential component of the finished product. As noted earlier, traditional engineers not directly involved in design and development of the software, frequently ridicule software engineering as not being “real” engineering. However, even the most severe critics of the term software engineering fully understand the importance of “getting it right” when their own engineering designs depend on correct software.

While there is no shortage of software horror stories, there is seldom mention of the thousands of software success stories that benefit from an engineering approach in their development. In my own experience, these successes include a large number of military software systems programmed in Ada such as the Hellfire missile, the Navy's Aegis system, BSY-2 submarine combat system, and others. Those successes also include non-military software systems programmed in Ada such as Intelsat-VII [Rie94] (and many other communications satellites), the Boeing 777 (and more recently, Boeing 787), and railway management systems in the U.S. and Europe. The above named systems are successful, in large part, because of the emphasis on engineering rather than programming. The engineering is software-intensive, and the behavior of the software is as dependent on software control mechanisms as it is on the mechanical and electronic systems. Without sound software engineering, the above named systems would never have become operational.

THIS PAGE INTENTIONALLY LEFT BLANK

III. THE ENGINEERING CONTEXT

A. PURPOSE OF THIS CHAPTER

1. Establishing Context

This chapter is intended to provide an engineering context for emerging engineering disciplines, including software engineering. It will provide a foundation for an engineering *reference discipline* suggested (and previously referenced) by Robert Glass. The focus of this chapter is engineering, not software engineering. A later chapter will focus on what aspects of software engineering conform to the context described here. One purpose of this chapter is to demarcate engineering from non-engineering. This will allow us to later determine whether any part of software engineering qualifies as an engineering discipline.

2. Terminology

In this chapter, currently accepted engineering practice will be called *classical engineering*, or *traditional engineering* as contrasted with non-traditional, non-classical engineering. A key problem throughout discussions of engineering, especially software engineering is the inconsistency of the linguistic model [Rie06].

B. CHAPTER THESIS

The thesis of this chapter is stated in the four points listed below. These are the assumptions that provide the foundation for the rest of this dissertation.

1. Engineering is a unique and rigorous discipline with goals, principles, rules, and practices for solutions to complex “real-world” problems that are different from those in non-engineering disciplines.
2. A rigorous definition for engineering will conform to engineering goals, principles, rules and practices, and would serve to *demarcate* engineering from other, non-engineering, disciplines.
3. An engineering definition needs to be relevant to both classical/traditional engineering as well as to newer, emerging engineering practices.

4. An emerging engineering discipline, to become a legitimate branch of engineering, must fall within the demarcation zone that conforms to classical engineering.

C. ENGINEERING GOALS

Engineering is concerned with the practical issues necessary for creating products and services that solve defined problems [Flo87], [Wri89]. According to published descriptions on engineering, an engineered object/product, when used for its intended purpose, will have utility. A review of the engineering literature shows that engineering practice converges on the following goals.

1. Absence of Ambiguity

Prevention of and absence of ambiguity is an implied goal for an engineered product or service. Therefore, one would exclude poetry engineering, psychotherapy engineering, or child-rearing engineering. Although complete absence of ambiguity is the goal, the presence of conflicting forces in a design may conspire to require some ambiguity on some engineering projects. The inability to achieve total disambiguation in a design often results in engineering trade-offs where even the ambiguities are prioritized.

Absence of ambiguity is just as essential for emerging engineering disciplines as it is for traditional engineering. In traditional engineering, disambiguation has often been possible because of the presence of natural forces and constraints. Natural forces such as gravity, coefficient of friction, and speed of light are less ambiguous than literary or social forces. Gerald Weinberg illustrates this with his “Mary had a little lamb,” example where he discusses the ambiguity of the meanings of “had,” “lamb,” and even “Mary.” Did Mary eat some lamb, or cheat a lamb? Is lamb a synonym for baby? Is Mary a Biblical figure or just some woman who gave birth to a child [Wei89, pp. 94-97]?

Engineering does not involve supernatural, occult, or other such forces. Rather, some forces originate in well-established intellectual disciplines such as mathematics, computer science, or process management. Where natural forces are absent or supplanted by non-natural forces (e.g., mathematical rigor), an engineering design must consider

other forces and constraints to prevent or mitigate ambiguity. Sometimes, prevention of ambiguity requires the introduction of non-natural constraints (e.g., logical properties such as the Hoare Triple) [Ghezzi2002, p. 323].

2. Predictability

Predictability is a complement of disambiguation. This goal is sometimes known as the “principle of least surprise” [Rie05]. Engineered products and services are intended to be predictable during their use. An ethical trial lawyer, regardless of how carefully s/he prepares a case, and notwithstanding a body of evidence that seems irrefutable, cannot engineer a *predictable outcome* from a jury (short of bribery). A sporting event, even when the opposing sides are unevenly matched, is never predictable at a level of confidence required of engineering. Engineered products and processes are often designed to statistically significant tolerances to ensure a high level of confidence in predicting that each instance of the engineered product or service will dependably and unambiguously carry out its intended purpose. Often pre-established standards dictate the composition of an engineered product or service, and predictability is an outcome of the constraints in those standards. Admittedly, engineered products and services sometimes fall short of being fully predictable, but predictable outcome is an engineering goal, even for emerging engineering disciplines.

3. Absence of Failure

In the early days of computer programming, the acronym, GIGO (garbage-in, garbage-out) was a commonly accepted viewpoint. If the users entered bad data, it was not considered the fault of the software designer. It was so-called, “user error.” The program performs as intended if the user does the right thing. Under the GIGO doctrine, once the program works, the user must use it correctly. Even today, many programming languages and applications are designed with inherent opportunities for error, and the programmer is expected to know how to use those languages correctly. This is somewhat analogous to the strike-anywhere matches. If you use the match safely, there will be no problem. If you allow the box of matches to linger on a shelf in your barn where the mice

can gnaw on them and start a fire, its your own fault. In farm country, these matches are often nicknamed, “barn burners.” The matches work exactly as intended, but also include hazards.

It is not enough to ensure that something works as expected; it is equally important to anticipate the potential for aberrations in the completed product. Prevention of aberrant behavior is a central goal in every branch of engineering [RieS07]. Most engineers focus part of their analysis and design process on reliability engineering. Failure analysis requires evaluation of potential risks in a product, elimination of those defects, and controls to deal with unknown defects should they be detected during operations. Failure prevention is a byproduct of risk-management. Risk is concerned with uncertainty. Each risk is associated with a potential for failure. When a risk is identified, and the level of risk exposure (RE) computed, an engineering design must include accommodation for that risk. When a failure can occur, the engineer must understand the causes of failure, predict when and where they might occur in a design, and include design features that anticipate and tolerate those occurrences. It is a fundamental principle that engineers try to design for unidentified risks and include controls to respond to events than could not be predicted [Rie96].

“An airplane exhausted its fuel supply after the pilot incorrectly set the [software-based] inertial navigation system to 270 instead of 027. Twelve people died when the plane was forced to land in the Brazilian jungle” [VOL, p. 147]. GIGO is no longer an acceptable excuse in engineering, including software engineering. Every engineer is responsible for the absence of failure, including the software engineer.

4. Serviceability

Once a product or service is deployed, it should perform as expected in the environment for which it was designed. Serviceability can be augmented by sub-goals such as maintainability, modifiability, efficiency, and ease of use. The required serviceability level will vary from one design to another, often depending on a trade-off between economic constraints and dependability expectations.

5. Economic Feasibility

One of the most important aspects of modern engineering is the need for economic feasibility. This is a relatively recent goal in engineering. However, it has become a guiding principle in most engineering projects of the past 100 years. If this goal is not met, a proposed engineering project is almost sure to be cancelled.

Economics is not only about money. It is about scarcity. If nothing were scarce, there would be no economic problems. The scarcity could be related to fuel capacity (as in the airplane accident cited above), limitations on weight, or available memory for a satellite computer system. Sometimes scarcity will manifest itself in terms of human capital. That is, the kind of human skills needed for successful completion of a project may be limited, not by pecuniary concerns, but the simple unavailability of people who have those skills. Many other examples of scarcity in engineering could be given. The main point is that the fundamental concern of economics is present in every engineering design, and the more complex the design, the more intrusive this concern becomes [WRI, p. 109], [VOL, p. 416]. Thuesen, et al., criticize those engineers who “restrict themselves to consideration of physical factors and leave the economic and humanistic aspects of engineering to others” [Thu89, p. 5].

6. Summary of Goals

There is no expectation that these goals will be exactly met in every engineering project. Any engineering project involves trade-offs between conflicting goals along with conflicting properties of the design elements. The problem of trade-offs between conflicting goals is common to every engineering effort. Therefore, the goals are never absolute.

D. ENGINEERING DEFINITIONS

In an earlier section, we reviewed definitions for software engineering. From those definitions, we concluded that one of the best software engineering definitions was from the original NATO conference: *the application of engineering practices to the*

development and management of software. This chapter examines a context for software engineering by reviewing definitions for classical engineering. Then after critiquing those definitions, we offer a more suitable definition for engineering that will encompass many *emerging engineering disciplines*, including software engineering while still demarcating engineering from other kinds of disciplines.

1. Current Engineering Definitions

A discussion of some common definitions of engineering is appropriate when presenting the engineering context for software engineering. This section summarizes a few of those definitions from articles, textbooks, professional organizations, and engineering folklore. We choose a few representative definitions, although there are dozens more throughout the engineering literature.

a. ABET

The official Accrediting Board for Engineering Technologies (ABET) adapts a definition of engineering from a very old source [ECPD58]:

... the profession in which knowledge of the mathematical and natural sciences gained by study, experience, and practice is applied with judgment to develop ways to utilize, economically, the materials and forces of nature for the benefit of mankind [Die83].

We will have more to say about this definition in a later part of this document. The principle things to note in this definition are the emphasis on “forces of nature” and the word “economically” Holtzapple [Hol08, p. 97] describes this as a “painfully contrived definition and notes that there are missing elements such as creativity and problem-solving ability. The ABET definition does not really describe the uniqueness of engineering. That is, it does not precisely demarcate engineering from non-engineering.

b. Shaw and Garlan

The ABET definition is not rigorous enough to demarcate engineering from other disciplines. It fails to uniquely differentiate engineering from other design activities. Shaw and Garlan offer a better definition.

...common usage refers to the disciplined application of scientific knowledge to resolve conflicting constraints and requirements for problems of immediate, practical significance [ShG96].

Some of the key ideas necessary for describing engineering are present in that description. In the engineering section of their book, Shaw and Garlan expand on the ABET definition with a description of five properties of engineering practice:

- **cost-effective solutions.** Engineering is not just about solving problems; it is about solving problems with economical use of all resources.
- **practical problems ...** Engineering of practical problems with solutions matter outside the engineering domain: the customers.
- **scientific knowledge** Engineering solves problems in a particular way; by applying science, mathematics
- **building things** Engineering emphasizes solutions, which are usually tangible artifacts
- **conflicting constraints** A central problem of all engineering

Shaw and Garlan continue:

Engineering relies on coding scientific knowledge about a technological problem domain in a form that is directly useful to the practitioner, thereby providing answers for questions that commonly occur in practice ... Engineering practice allows ordinary practitioners to create sophisticated systems that work -- unspectacularly, perhaps, but reliably.

... engineering emerges from commercial exploitation to supplant craft; modern engineering relies critically on adding scientific foundations to craft and commercialization ... science and engineering support each other ... engineering generates good problems for science ... good scientific problems often follow from an understanding of the problems [of] engineering ...

c. Rogers

The ABET definition does explicitly acknowledge the importance of engineering experience. Vincenti [Vin90] in a book about aeronautical engineering places great emphasis on engineering experience. Throughout the book, he repeats the importance of experience, even for the well-educated engineer. Vincenti also quotes British engineer G.F.C. Rogers with the following definition,

Engineering refers to the practice of organizing the design and construction of any artifice which transforms the physical world around us to meet some recognized need.

When Rogers crafted that definition, computer software was not regarded as a part of the engineering problem. Consequently, the engineering of his time was strictly concerned with the physical world. Vincenti adds the need for “operation” in the definition, and then goes on to explain that *organize* is meant to convey a sense of “bring into being,” “get together,” or “arrange.” He then examines the notion of “design” in some depth making the activity of design the central idea derived from Rogers’ definition an additional theme of his book on engineering.

d. Florman

Samuel Florman [Flo87] provides this interesting definition:

Business, government, academic, or individual efforts in which knowledge or mathematical and/or natural sciences is employed in research, development, design, manufacturing, systems engineering, or technical operations with the objective of creating and/or delivering systems, products, processes, and/or services of a technical nature and content for use.

e. Wright

Paul Wright [Wri89] says this:

Engineering is a profession in which the knowledge of mathematics and the natural sciences is applied with discretion and judgment in order to use economically the materials and forces of nature for the benefit of people

It differs from other learned professions in a number of ways: in the type of service provided, in the training requirements for its practitioners, in the diversity of its leadership, and in the lack of uniformity and rigidity in its registration laws.

Engineers are concerned with the creation of structures, devices, and systems for human use. In contrast to other professionals, engineers tend to create machines, structures, processes, and the like for the use of groups of people rather than for an individual. They seldom deal with the users of their works or beneficiaries of their services, while other professionals (e.g., attorneys, physicians, psychologists, and dentists) commonly do.

Wright's definition is important because it emphasizes the professional nature of engineering. The implication is one of adherence to high-standards, specialized education, and recognition by a peer group that one has acquired the required credentials to practice the given profession. An important distinction in this definition is the acknowledgement of "processes" as a part of the domain of creation.

In this definition, the engineer is distant from the users or implementers of a design. A surgeon may have a well-designed routine for a procedure, but because of the immediacy of the surgeon in the carrying out of the design, it is not an engineering activity, if we accept the definition given by Wright.

E. SUMMARY OF CURRENT DEFINITIONS

With the exception of the definition from Florman and the inclusion of processes by Wright, the definitions rely on the concept of natural forces. Combining elements of the representative definitions given, along with many others gleaned from the engineering literature, primarily [Vol04], [Hol08], [Wri89], [Flo87], [Pet85], and [Pet96], leads to a partial list of concepts and practices that characterize engineering;

1. organization
2. management (including risk management)
3. conflicting forces (including physical and non-physical forces)
4. practicality
5. creational/design process (slightly different from creative0
6. previous engineering experience

7. accepted knowledge (usually settled knowledge)
8. tools of science
9. tools of mathematics
10. tools of logic (a specialized branch of mathematics)
11. leveraging technology
12. constraints (physical and non-physical)
13. design to tolerances
14. predictability
15. economic constraints and trade-offs
16. continuous control and feedback process

Taken together, these sixteen elements help us with our goal of demarcating engineering from other disciplines. Taken independently, they do not. Even natural forces are evident in many kinds of non-engineering crafts such as pottery and woodcarving.

F. TRADITIONAL VIEW IS TOO NARROW

Many published definitions, including the one from ABET, have placed emphasis on the narrow viewpoint where “forces of nature” is an essential part of the definition of engineering. With the emergence of new kinds of engineering, such a view is too narrow. Contemporary society requires dependable products and services. This need demands that those charged with developing those products and services understand the engineering model. They must meet the challenges of applying engineering goals, principles, methods, and practices in this kind of development, including physical entities and non-physical products, processes, and services.

G. NEEDED: AN UPDATED DEFINITION

Modern engineering is increasingly concerned with the design of processes, some of which are still governed by or include an element dependent on natural forces. However, natural forces are not always the controlling or even the constraining element of an engineering design. Increasingly, human factors, software controls, and process flow are emerging as essential elements of an engineered design. It is human factors that

guide the decision to engineer a product in a certain way. For example, it might be known that, under a given set of circumstances, a knob is more conveniently dialed clockwise instead of counter-clockwise and the engineering design must account for that factor. Or, the economics of a proposed design may suggest that the engineer consider a different set of materials from those which might be ideal to enable lower manufacturing costs, while being careful not to compromise the safety and security of those who will use the product. All of the above discussion points to a model of engineering that is much more complex and comprehensive than the oft-espoused purely mechanical view of engineering [Volland2004]. The quote from astronaut, Alan Shepard regarding the economics of space systems design is instructive. Just prior to launch and noted, Shepard glanced around the cabin and was struck with the thought, “all of this was built by the lowest bidder” [Shepard].

The updated engineering definition must accommodate emerging engineering practices, not just the traditional engineering disciplines with their exclusionary emphasis on “forces of nature.” At the same time, once an engineering definition is accepted that demarcates engineering from non-engineering, any aspirant to the engineering club must conform to that definition. This includes software engineering.

1. Improving the Definition

a. Current Definitions

Although the concept of “natural forces” and “physical world” have been important in traditional engineering, modern engineering requires the development of new practices across a larger range of problems. These new engineering practices require that we *raise* the level of abstraction for engineering, and *broaden* its domain in subtle, complex, and increasingly sophisticated ways. The associated problems and constraints of those practices are equal in importance to the natural forces model that has served traditional engineering. As new members of modern engineering community, there is the same demand for discipline, knowledge, and attention to technical detail as has been required in mechanical, electrical, civil, and chemical engineering. Insistence on an

engineering governed primarily by natural forces serves only to discourage and delay the benefits that can derive from the application of engineering practices necessary in an increasingly technological society.

An examination of the many contemporary definitions for engineering (and software engineering) for our review of the literature made clear that the currently published definitions:

- are not inclusive enough to make a clear demarcation between engineering and other activities,
- fail to account for the emerging engineering disciplines that have become essential for modern society,
- place unnecessary restrictions on what engineering is, and
- fall short of what engineering could be.

It is appropriate for engineering definitions to be conservative with regard to what can be classified as engineering. Even so, a comprehensive definition must also be inclusive enough to include emerging engineering disciplines. The goals for engineering stated in Chapter III are essential for engineering. Those goals are also conservative, but achieving them does not require “forces of nature,” as that term is usually understood.

b. The Engineer

Ultimately, engineering is performed by human beings [Pet86]. A definition of engineering must also account for the professional qualities of the practicing engineer. Those qualities are summed up in the word, “responsibility” [Flo87], [Wri89]. The conservative character – the very personality -- of the professional engineers is driven by the word, responsibility. The need to be responsible is one reason why engineers are reluctant to abandon the idea that natural forces are so central to engineering. It is an old-saw in engineering that, “Gravity is cheap and dependable.” The responsible engineer favors practices that are known to be dependable within known economic constraints. That is, engineering practices strive to support the goals described

earlier: unambiguous, dependable, serviceable, and economical. Engineers are also taught to watch for what might go wrong and develop safeguards and controls to prevent mistakes from occurring in the operational version of the product they have designed.

c. The Contemporary World of Engineering

Modern engineering requires a definition, for emerging engineering disciplines, that supports the goals, principles, and practices of engineering as well as the character of the engineer. The widespread introduction of new technologies (including software) that control everything from medical devices to transportation equipment indicates that such a definition must also include the many new and non-traditional emerging engineering practices. Any practice that falls short of our criteria (as epitomized in the forthcoming definition) also falls short of being a full member of the engineering community. A credible definition must sustain the rigor that affirms the values of traditional engineering while opening the door to new kinds of engineering.

To insist that we cannot engineer anything in the absence of natural forces is to abandon that very trait of responsibility cited earlier. The fact that something is difficult to engineer is all the more reason to seek an engineering model that works for it. This is especially true when we are building systems where risks to human safety or economic disaster can result. Giving up in exasperation because we are unable to find immediate engineering solutions to seemingly intractable problems contradicts one of the goals of engineering stated earlier: an overarching dedication to responsibility.

Expansion of the range for engineering solution-space is not simply a nice idea. In modern society, it is an essential paradigm that cannot be ignored simply because some classical engineers fail to see it as a real engineering discipline. We cannot give up on finding engineering models for emerging engineering disciplines just because it is hard.

d. A Revised Definition of Engineering

Given the discussion above, we provide, below, a composite definition that demarcates engineering from other activities. This definition serves as the core

argument for the rest of this dissertation, especially in our chapter on software engineering. Therefore, we also describe, in some depth, what each of the components means.

Engineering is the organization, application, and management of settled (dependable) knowledge using the tools of science, mathematics, and logic, along with knowledge, experience, and artifacts derived from previous engineering efforts, for reconciling conflicting forces/constraints, controlled within defined tolerances, to effect an economical, risk-averse, maintainable, fault-tolerant design toward the goal of a predictable outcome.

Figure 3. Revised Definition of Engineering

Each component of that definition is important to the overall understanding of what makes engineering different from other *creational* practices. Taken together the elements of the definition converge on the important engineering attitude of “responsibility” cited earlier.

Note that this definition contains no value judgments such as ABET’s ‘benefit to mankind.’ There is no mention of ‘natural forces.’ Rather, we are concerned, in modern engineering, with many kinds of forces, not all of which are natural forces. A key item in the definition is the word *controlled*. Engineers incorporate many kinds of control in their designs. Control is used for managing risk, errors, and feedback, as well as for ensuring that a design satisfies its own goals, objectives, and obligations. The notion of control is of fundamental importance in our later discussion of software engineering. Where natural forces are absent, other kinds of controls are imperative.

2. Summarizing the Updated Definition

The elements of the above definition of engineering support the goals, principles, rules, and practices of engineering. It preserves the requisite discipline expected by the professional engineer. It allows for the development of new kinds of tools in the support of modern engineering. It opens the opportunities to develop new fields of engineering. Finally, it gives us deeper insight into the nature of engineering practice.

This summary is derived from the definition and expanded upon in the next chapter.

1. Dependable/Settled Knowledge
2. Force(s)
3. Control
4. Predictable outcome
5. Tolerances and Constraints
6. Economics
7. Resource management
8. Design but not implementation of design

Other aspects of engineering, many of which involve processes, policies, statutory regulations, and specialized tools could also be included, but are not in the scope of this present work. They do not demarcate engineering from other practices. Even the ethical practices and licensing issues, while essential to professional engineering practice, are not essential to a demarcating definition.

This chapter laid out the boundaries or demarcation line for what is or is not engineering. The demarcation led to a more comprehensive definition for engineering, one that satisfies the continual emergence of new kinds of engineering. Subsequent chapters will explore specific topics related to these boundaries in greater depth.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. ELEMENTS OF THE ENGINEERING CONTEXT?

A. INTRODUCTION

The previous chapter enumerated the elements for demarcating engineering from other disciplines. This chapter looks at some of those elements, point-by-point. We derived much of the information for this and the preceding chapter from books on engineering, especially the book titled “Introduction to Engineering” by Simon Wright, supplemented by the many insights from Robert Florman and Henry Petroski [Wri89], [Flo87], [Pet85], along with many other textbooks on engineering. The focus is on engineering, not on software engineering. A later chapter on software engineering will use the elements described in this chapter. We choose not to footnote every statement since most engineering textbooks are in agreement with regard to our less controversial observations.

B. DESIGN

One of the things that sets engineering apart from many other *creational* disciplines (e.g., craftsmanship or skilled trade) is the emphasis on design separate from implementation. Architecture is also about design and we will deal with the differences between architecture and engineering elsewhere in this document. Engineering is largely different from craftsmanship in two ways: 1) immediacy of the design and 2) realization of a design, 3) concern for failure. The engineer, and there are exceptions, is often removed from the site where the realization of the design will be deployed. Further, the design may include a long lead-time of reviews and approvals before it is given authorization for construction.

In craftsmanship and other non-engineering activities (e.g., computer programming), the craftsperson responsible for the design is also responsible for its construction. This immediacy between design and construction is often regarded as a good thing in the software domain, and new so-called “agile” processes are touting this as

the way to create software that is on-time, within budget, works as intended, and is low risk. Choosing an agile process does not absolve the software developers from producing an engineering design when engineering is required.

C. THE ROLE OF KNOWLEDGE

1. Settled/Dependable Knowledge

We use the word “settled” to describe engineering knowledge. Settled knowledge implies the agreed-upon scientific and engineering knowledge fundamental to most kinds of engineering. Above all, it must be dependable knowledge. In fact, the notion of dependable knowledge might be the most important starting point for any kind of engineering. The notion that knowledge must originate in “nature” overlooks the many kinds of dependable knowledge that are derived from experience and experimentation rather than from the pure sciences. Engineers often depend on a range of acceptable values or statistical significance for engineering to “best fit.” This knowledge is often used for designing to predetermined tolerances.

Vincenti has this to say about engineering knowledge:

Engineers, unlike physicists, are after useful artifacts and must predict the performance of the objects they design ... and this fact is essential to their analysis [Vin90, p. 130].

Vincenti goes on to list some of the kinds of knowledge necessary to produce predictable results. These can be summarized as follows:

- Knowledge from previous engineering experience
- Artifacts from previous engineering projects
- Standards and published data sheets and tables of data
- Information from stakeholder requirements development
- Continual discoveries in many kinds of science

Knowledge evolves with each new cycle of engineering; that is, a new collection of knowledge from previous engineering projects, leads to an increase in the knowledge that can be used in future engineering projects. In this way, engineering continues to

reinforce its own professional standing. There are many accounts of Thomas Edison searching for a filament for his light-bulb. Most of these say he tried 8,000 different kinds of material. When he was finally successful, he noted that he now knew 8,000 things do not work along with one material that does. This is how engineering knowledge advances: both failure and success leading to an increase in knowledge that can be used in successive engineering projects.

The concept of *settled knowledge* is somewhat analogous to the concept of *settled law*. Every engineering discipline, including software engineering, uses settled knowledge. The world of *settled knowledge* includes settled science, settled mathematics and knowledge gained from previous engineering practice (i.e., experience). Engineering is rarely about pure research even though some engineers make their contribution through research. The engineer is designing solutions to real problems. Use of settled knowledge, including data in published tables and statutory codes, along with prior experience is critical to engineering success. To reduce the amount of surprise in the resulting design, the engineer often depends on that published knowledge.

2. Knowledge from Science

Knowledge might be thought of as an organized composite of information. Science is often the process that results in that organization. Engineering, while often concerned with deriving new knowledge from existing information, is also focused on using that information in its already organized form: settled knowledge.

There is no disagreement among engineers that settled science is a fundamental requirement for engineering. The engineer needs to know what kind of settled knowledge from science is most appropriate for the kind of problems presented. This is true of new realms of engineering as well as well-established classical engineering fields.

To illustrate the application of settled knowledge from science we can use the following example from Shaw and Garlan. Chemistry is a science. Chemical engineering depends on the science of chemistry [Sha96]. A science is mainly concerned with the discovery and recording of knowledge. An engineering design is concerned with the application of that knowledge, along with knowledge from experience with the intent of a

predictable outcome. A chemical experiment is designed to discover some new phenomena; and surprise is often greeted with joy. An engineering experiment is disappointing if it does not produce the expected result or solve the intended problem. Predictable outcome is the concern in engineering [Rie05].

3. Knowledge from Engineering Experience

Vincenti makes the point that engineering knowledge is often derived from what we learn in previous engineering experience [Vin90]. In his book on engineering practice, Vincenti describes many engineering decisions in the discipline of aircraft design that could not have been made simply by relying on natural science and a collection of equations. Sometimes the engineering experience is as simple as knowing that a pilot turning a knob clockwise in a cockpit is more effective than turning it counter-clockwise, or that a flat toggle-switch on a gun-turret is better than a rounded one.

The product of engineering is expected to work as intended, based on some set of stated requirements. Surprises are unwelcome [Rie05]. While some theoretical elements may be incorporated into prototypes and experimental designs, the engineered product relies mainly on what is known, not on guesswork, theory, or ‘hit or miss’ tactics. Large engineering designs that work are often built from the knowledge gained through smaller designs that already behave in predictable ways. The smaller designs, sometimes referred to in software practice as modules, are expected to be well-understood, have predictable behavior, published metrics, and standard interfaces. Knowledge in standard interfaces helps the engineer organize the smaller designs according to a commonly understood, and known-to-be-dependable, set of protocols.

As noted earlier, settled knowledge must sometimes be revised to account for new scientific discoveries. This means that settled knowledge is sometimes only settled until science or experience unsettles it. Often, the engineer is required to accept “good enough” knowledge, settled for now, but subject to future alteration. Although engineers may infer new information through mathematical reasoning, those inferences originate in settled knowledge.

Settled knowledge does not guarantee the success of an engineering project. That knowledge must be used correctly in a given design. The engineer must bring the tools of reasoning for a design that involves new configurations of what is known. For example, we may know the coefficient of friction for two entities, but we must also reason about what happens when they are combined into a design. Does one set of coefficients potentially cancel the effect of the other? Software engineers often define a set of constraints for a module. Does the complexity of a constraint sometimes become self-contradictory? Engineers often need to refine their understanding of the relationships between design artifacts and the corresponding constraints derived from settled knowledge. This requires experimentation, prototyping, testing, and refactoring to ensure the validity of that knowledge in its new context

Even reusable components, the ultimate in using settled knowledge, are subject to the need for confirmability when incorporated into a new design. The maiden-flight of the Ariane V rocket demonstrated how a perfectly good component might behave as expected and still give the wrong result in a new design [SOM-web]. The Ariane V reused software components from Ariane IV that were known to work properly on that rocket, but failed to consider the changes in the design of Ariane V. This is analogous to a physician giving a patient a perfectly good medicine that works for almost everyone without considering medical history of the person for who it is being prescribed. Engineers are required to evaluate settled knowledge in the new context where it is expected to be used. They must also ask the question, “How could it fail?” and design for the answer to that question.

D. FORCE(S)

1. The Concept of Force

The concept of force [Jam99], and reconciliation of conflicting forces, is a fundamental concern in every facet of engineering. Engineers, in the physical world, usually think of force as something that “causes or changes motion or displacement” [Ste05]. Each displacement can also be thought of as a change-of-state.

2. Non-Physical Forces

In modern engineering, the forces extend beyond those imposed by Mother Nature. These include economic, statutory, social, political, and ethical forces [Hol08]. We devote a separate section of this chapter to economic forces, but leave most of the discussion of other non-physical forces to people such as Samuel Florman [Flo87].

Statutory forces have become more and more important in engineering. Governmental standards and codes often restrict what kind of design an engineer may present for implementation. One might say that building codes reflect a reasoned body of knowledge regarding physical forces (e.g., design to withstand earthquakes), but that is only part of the story. Building codes also derive from community standards regarding esthetics, bigotry, and economics. The chemical engineer may be concerned with human factors such as a flavor or scent, which are not so easy to engineer, notwithstanding their natural properties. A colleague of mine at Cornell is a chemical engineer specializing in developing natural flavors for artificial foods. Predictability in flavor-engineering is an elusive goal, and depends heavily on a research component, even as existing knowledge contributes to its development. Also, some components of an otherwise successful food design violate statutory constraints.

3. Forces of Nature

The ABET definition puts emphasis on designing within constraints imposed by the *forces of nature*. There is an underlying epistemological question that could be argued regarding the theory of nature. Nature can be framed in terms of physics. Physics, in turn, can be examined using mathematics. Mathematics can lead to abstractions that predict abstract concepts of physics that are not directly observable in classical physics. A kind of “chicken or egg” ontological argument can be formulated for nature that inquires about the precedence of mathematics over physics or physics over mathematics. Epistemology and ontology, while of theoretical interest, are beyond the practical concern of this present inquiry.

It is enough, at this point, to agree that managing nature, using mathematics, has been one of the principle concerns of traditional engineering over the centuries. Nature, alone, is not enough. Designing within the constraints of nature is also a common thread in activities such as baking a pie, performing an appendectomy, or changing a diaper.

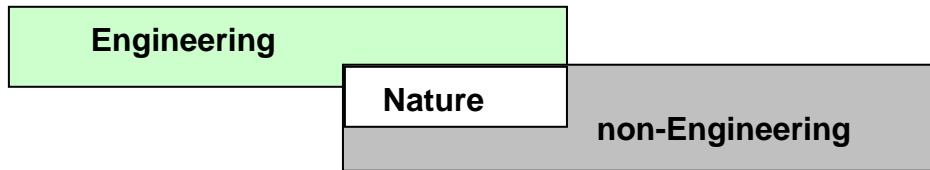


Figure 4. Intersection of Non-Engineering and Engineering

For example, in the abstract Venn Diagram (above), we might see that a chemical engineer designing a pipe to carry a caustic liquid from one part of a production facility to another. S/he is concerned with natural forces, and has standard data about the materials involved to calculate the effectiveness of a design. At the same time, the economic considerations and safety issues are essential in planning an effective design. On the other hand, a surgeon doing well-planned open-heart surgery is beyond predictable design. As the surgeon's hands slosh around in the chest cavity of a patient, s/he could be required to make sudden adjustments in the procedure if a vessel begins to bleed or the heart stops in mid-suturing.

There is an immediacy issue in these scenarios. Both the chemical engineer and the surgeon are dealing with natural forces. They are both using their knowledge to do the job. However, there is urgency in the work of the surgeon (or surgical team) that would be unusual in most kinds of engineering. One would be hard-pressed to classify the efforts of a surgeon, regardless of skill level, as engineering.

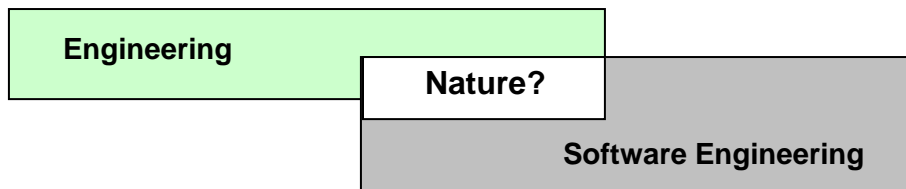


Figure 5. Engineering , Software Engineering and Nature

In the above diagram, one of the central issues is highlighted: the question of natural forces in the design, development, and realization of software. Some in the software engineering community would insist that every construct that can be described mathematically is a natural force. Others will note that, in real-time concurrent software systems, time is a natural force and the one force managed almost exclusively by the software design [Shaw2001, pp. 91-149]. Further, a careful study of the complexity of interactions between the elements of concurrent communicating processes in a real-time system involves serious engineering problems, not programming problems [Burns01] and also [Shaw2001].

Most contemporary engineers will agree that engineering practice involves more elements than the constraints imposed by nature. However, some will also insist that, without the constraints of nature, “real” engineering is absent. This restrictively-framed, natural forces viewpoint, is a common, exclusionary, theme in classical engineering. As noted earlier, there is some comfort the engineer can take from knowing that the immutable laws of nature are in play for a final design. However, to ignore the dynamics of the non-natural (e.g., social and economic forces) world for modern engineering is like wearing a pair of horse-blinders.

4. Conflicting Forces

The engineering definition provided in this dissertation includes the phrase, “reconciliation of conflicting forces.” Sometimes this is stated as simply as “resolving trade-offs.” All engineering involves trade-offs. Conflicting forces, in modern engineering, is an idea that goes far beyond natural forces.

If there were no conflicting forces, there would be no requirement for engineering. Everything would simply fit together as if designed that way by nature. Conflicting forces include economic constraints, statutory forces, risk management forces, human factors conflicts, and many more. Note that conflict, in this sense, is not like the kind of dialectic conflict that characterizes Hegelian philosophy or even the kind of conflict resolution that one finds in literature, competitive games, and social interactions.

Reconciliation of conflicting forces usually requires the engineer to reason with well-defined tools from science, mathematics, and logic. This means that the engineer can select from an array of tools for the disciplined creation of a design. Not every set of conflicting forces suggests the same kind of science, the same kind of mathematics, or the same kind of reasoning skills. Sometimes an engineer needs to invent some new tools, often leveraging the settled knowledge of the existing tools.

E. RISK MANAGEMENT AND CONTROLS

Risk management for an engineering design must be based on realistic knowledge, not guesswork [Hall97], [Jon94], [Jon96], [HeK92], [Gel06]. Risk is fundamentally about uncertainty, and the engineer must design with that reality always in mind. The awareness of risks, anticipated and unexpected, leads to the need for engineering designs to include *controls* to prevent failure.

For complex engineering projects, it is not enough to simply design a product and turn it loose to do its work. The design of *controls* is another fundamental aspect of engineering that demarcates it from other disciplines. A part engineering design is to understand how to account for and plan for both the expected and unexpected deviations for the deployed design. An entire subset of process engineering called control engineering deals with this category of concerns [Ogata97]. In a Systems Engineering curriculum, control engineering is a required course of study. Every engineering effort involves some consideration for controls in the product or process, including design for the unexpected events that may occur [Rie98]. Practitioners in emerging engineering disciplines, including software engineers, must subscribe to this notion if they are to be credible as engineers.

F. PREDICTABILITY (PREDICTABLE OUTCOME)

The earlier quote from Vincenti mentions *predictability*. In engineering, the “*principle of least surprise*” remains important. This principle, spoken or unspoken, is a common thread throughout all serious engineering. A courtroom battle, even when the case in question seems to be “open and shut,” cannot guarantee a predictable result. Two

boxers may be unequal in skill, one vastly superior, but neither boxer can be guaranteed a predictable outcome. The “lucky punch,” while rare, does occur. Engineers work cooperatively to resolve conflicting forces to achieve a predictable outcome.

When engineers work together, they are in a cooperative, not a competitive enterprise. They are participating, as a team, in the reconciliation of conflicting forces and designing controls to ensure the design will be stable and predictable under all anticipated (and sometimes unanticipated) circumstances. Even as they disagree on approaches to the design, becoming themselves, conflicting forces, they are ultimately committed to solving the conflicting forces for a predictable outcome, not simply for winning a game.

The concept of predictable outcome is not the same as perfect, exact, or absolute outcome. Rather, predictable, in the engineering sense, includes being within the range of acceptable tolerances over a defined set of constraints. Consequently, engineers apply, whenever possible, a set of measurable constraints (often from standard/published data sheets) over a design that has the potential for unexpected failure. One can think of these constraints as analogous to fuses or circuit-breakers [RIE98]. As with a simple fuse, when one of the design constraints is exceeded, detection of that violation can safely shut down a device/system, or in some circumstances, restores the engineered product to a stable state, a state with predictable attributes and behavior.

An important property of predictable outcome is the prevention of failure. This theme runs throughout the work of Petroski in his discussions of the design of a mechanical pencil, the engineering of an aluminum can, and his description of the earlier failures in commercial jet aircraft designs [Pet89]. It is also a central theme in engineering textbooks, regardless of the branch of engineering being studied.

As engineers strive for a predictable outcome, they reason about the potential for failure in their designs, and exercise multiple tests against both the designs and the prototypes. As Petroski notes:

An idea that unifies all of engineering is the concept of failure. Virtually every calculation an engineer performs ..., is a failure calculation... to ... provide the limits that cannot be exceeded ...

As an engineer calculates the forces and deflections of a trial design, each resulting numerical calculation takes on meaning and becomes acceptable only in comparison to failure criteria, which may have been determined by careful laboratory experiments on the materials and components in question.

What distinguishes an engineer from a technician is largely the ability to formulate and carry out the detailed calculations of forces, deflections, concentrations, and flows, voltages and currents, which are required to test a proposed design on paper with regard to failure criteria...

The ability to calculate is the ability to predict the performance of a design before it is built and tested ... Calculations that indicate failure conditions in a design enable the engineer to modify and re-modify the design until it is ready to be realized [Pet85].

Engineering failures do occur in all engineering domains. One famous example, the Tacoma Narrows Bridge, sometimes known as “Galloping Gertie,” has become a case study in most books about engineering [Vol04]. Voland makes the point,

... the collapse ... should remind engineers that they must be familiar with historical design failures if they want to avoid repeating the mistakes of the past.

P.G. Neumann of SRI maintains a column in Association Computing Machinery (ACM) Software Engineering Notes (SEN) in which he regularly reports on failures related to technology. Neumann’s book [Neu95] on risks should be a motivating jeremiad for software practitioners. The lesson in Neumann’s book is that, even with our best engineering efforts, it is impossible to always predict failure. Often the failures are traceable to inadequate requirements analysis, unpredictable environmental factors, or simply failing to ask all the questions necessary to prevent operational aberrations. Also, engineers, especially software engineers, are frequently stretching the limits of what they are trying to accomplish to beyond the limits of what they actually know. That is, they are presenting designs at the boundary where research stops and engineering begins, even as they strive for predictable outcome. This is one of the major risks in the effort to apply engineering practices to the creation of dependable technology-based products and services.

G. CONSTRAINTS AND TOLERANCES

For the traditional engineer constraints and tolerances, along with risks, are taken as a given concern of every problem domain. The constraints of the natural world are always present in physical engineering. These are augmented by additional constraints from the non-physical world. They are part of the settled knowledge and require the trade-off analysis that leads to reconciliation of conflicting constraints and forces.

The phrase, “design to constraints/tolerances” poses a particularly difficult problem for software practice. Chapter VIII examines this concept in terms of software engineering.

H. ECONOMICS

Economics, as noted earlier, is about scarcity. Scarcity is concerned with many kinds of resources. Financial scarcity is one, but only one of them. Others could include limited space for deployment of some design; limited fuel capacity for launching an artifact of a given weight in a communication satellite; too low a velocity of some moving object; insufficient time for a particular process to complete before its results are required. Scarcity might govern the speed of a cruise missile that needs sufficient computation power to evaluate “event horizons.”

The economics of an engineering problem are central to every solution. Without an economic consideration, there is no engineering. An engineering design that ignores the economic factors of that design falls short of being a real engineering solution. In a history of engineering, we find the following passage.

When it became obvious in the early nineteenth century that ... a structure or mechanical device designed to carry maximum contemplated loads or perform a specific function and no more was more economical than one designed on the basis of ‘experience,’ engineering science began to develop rapidly [Kwdk90].

Even the ABET definition, quoted earlier, notes the importance of economics in engineering. When Jack Kilby was inspired to design a new approach to integrated circuits, he was largely influenced by his father's (another engineer) admonition that engineering was as much an exercise in economics (balancing resources) as it was the leveraging of science.

If nothing were scarce, there would be no need for any kind of economics.

Example (from personal experience on a project):

Consider the case of a communications satellite with on-board computers [Rie94]. The ideal computer would have been a micro-miniaturized computer for this application. Ambient radiation demanded a radiation-hardened computer instead. This, in turn, led to the choice of a computer that was not a miniaturized as hoped for. Weight considerations, along with limited on-board real-estate dictated a MIL-STD 1750A. These decisions, based on scarcity (economics) led to some exceptional engineering problems in the design of the software environment. In this case, scarcity required software engineering solutions commensurate with the economics of the environment.

I. STATE TRANSITIONS AND DISPLACEMENT

Physical displacement is often measured as then change of one object relative to another object. This change is sometimes called the *context of motion*. The *context of motion* is also a *change of state*. By raising the level of abstraction from *context of motion* to *change of state*, we broaden the meaning of *change*. Engineering is about state change, whether physical or non-physical.

Research in physics along with engineering experience provides volumes of formulae (settled knowledge) that describe state/displacement transitions. Engineers benefit daily from those published results and use that knowledge in the design of new products and services, including software, new medicines, and new kinds of food products.

J. OTHER ASPECTS OF ENGINEERING

The concerns of engineering as described so far are not restricted to those enumerated in the definition. We summarize a few of those concerns below for the

purpose of presenting a more complete view. However, the following concerns are present in many other creational and project-oriented disciplines, and they are not unique to engineering.

While there are many other topics that could be covered in this discussion, few qualify as specific to engineering. These include human factors, development processes, project planning, and many more. A longer, extended version of this dissertation, in the form of a complete book, will present an expanded discussion of these issues as part of the total picture. For this dissertation, we have chosen to bound the discussion to those issues inherent in any kind of engineering, and to focus on those that, when combined as a set of topics, define engineering as a unique discipline. This combination of topics, when considered in the context of the definition given at the beginning of this chapter, sets the tone for the final chapter in which we explore the notion of whether any part of software engineering practice conforms to the definition provided in this chapter.

K. CHAPTER SUMMARY

An effective engineering model for some emerging engineering disciplines can be both elusive and incomplete. It is necessary to re-frame the entire concept of engineering so it is inclusive of, not only software, but also the many other engineering candidates (e.g., genetic engineering).

Even as we reframe and/or redefine the notion of engineering, we must remain faithful to the underlying goals, principles, and demands of engineering. The discipline must be our guide, and the rigor must be our master. When the discipline falls short of what is required under the new definition given in this chapter, that discipline must be excluded from the engineering club. It might be a rigorous model of development, but it might also fall short of qualifying as a fully-developed engineering practice.

No one can disagree with the fact that software now permeates the design of nearly every artifact of modern life. This includes fly-by-wire commercial aircraft, digital cameras, high-definition television sets, portable telephones, software-based automotive control systems, children's toys, and a long list of other examples. One consequence of this fact is that software safety has become an important area of research and study. By

safety, we mean to include physical safety, safety of such things as financial records, personal information (e.g., identity theft), government security systems, and hazard information management. All of these domains, and others, are now controlled by software. Anyone who asserts that an engineering model is unnecessary to manage risks inherent in such systems is simply not paying attention. Therefore, both classical engineers (especially systems engineers) and software engineers must collaborate, without bickering about what is or is not engineering to address these concerns. Below is a tentative matrix for evaluating the maturity of an emerging engineering discipline. This is not a decisive model and it might vary greatly by discipline, but it is offered as a starting point for future evaluations.

Emerging Engineering	Level Zero	Level One	Level Two	Level Three	Level Four	Level Five
Consistent Process	No	Yes	Yes	Yes	Yes	Yes
Design Metrics	No	No	Yes	Some	Some	Yes
Controls-based Design	No	No	No	No	Some	Yes
Natural Forces Govern	None	None	None	None	Some	Yes
Professional Education Required	No	No	No	Some	Yes	Yes
Design to Tolerances	No	No	No	Some	Some	Yes
Constrained Design	No	No	Yes	Yes	Yes	Yes
Low or No Ambiguity	No	No	No	Yes	Yes	Yes
Reconcile Conflicting Forces	No	Yes	Yes	Yes	Yes	Yes
Predictable Outcome	No	No	No	Some	Yes	Yes
Can Design for Failure	No	No	No	Some	Yes	Yes
Risk Management in Place	No	No	Some	Yes	Yes	Yes
Linguistic Continuity	No	No	Yes	Yes	Yes	Yes

Table 2. Tentative Matrix for Evaluating Maturity of Emerging Engineering Discipline

Each level represents a maturity index. Readers will recognize this as reminiscent of the work of Watts Humphrey’s Capability Maturity Model [Hum89]. Only at the highest level do we required natural forces. At Level Zero, we assume that everything is ad hoc or dependent on the star designer, the superhero, or the exceptional individual. Level One is a little better organized. At Level Two, we assume there is an exceptional manager supervising the designers. At Level Three, there are some standards in place.

Levels Three, Four, and Five begin to converge on good engineering practices. For emerging engineering disciplines, they need to get to a minimum of Level Four to be regarded as real engineering practices.

V. SOFTWARE ENGINEERING AS ENGINEERING

A. INTRODUCTION

Previous chapters examined the fundamental concepts of engineering. Of special importance were the chapters titled, “The Engineering Context,” (Chapter III) and “Elements of the Engineering Context” (Chapter IV) where we presented a set of essential properties for any discipline that aspires to be an accepted member of the engineering community. This was preparation for the present chapter which examines the legitimacy of software engineering as engineering. If the term software engineering is to have any credibility, the principles and practices described for engineering must also apply. As noted earlier, Glass, et al., call this a *reference discipline* and suggest that the lack of a reference discipline is one of the missing elements in finding a discipline of software engineering.

We acknowledge that the number of available examples of engineered software is much smaller than we would like. However, many of the tools, ideas, and methods for a professional practice of software engineering that conforms more closely to engineering are already in place.

B. PURPOSE OF THIS CHAPTER

This chapter is focused on the specific *reference engineering* concerns that map software engineering with traditional engineering. The topic of designing to tolerances, a design metrics concept fundamental to most of standard engineering, is deferred to later chapter.

The question, “Does software engineering conform to the level of rigor demanded of other branches of engineering,” is addressed here. It will be clear that not all software development activities can be accorded the legitimacy of engineering. As evidenced by the survey mentioned earlier in another chapter, many people in the software field, as well as those in other engineering fields agree that many computer programming

practices fall far short of engineering. Even so, the attempt at mapping some existing and potential software practices to what is expected of standard/traditional engineering may open the doorway into future research and provide a context that can guide that future research. This may eventually lead to a stronger model for software engineering as a legitimate engineering discipline. The chapter attempts to establish the engineering *bona fides* for at least some aspects of software practice.

C. CHAPTER THESIS STATEMENT

1. Software engineering is an emerging engineering discipline with the promise of eventually becoming a fully eligible member of the engineering community. It needs to be examined in the context of an engineering reference discipline.
2. Some current software practices do correspond to what would be expected of engineering. A point-by-point mapping of traditional engineering model reveals there is some conformity between practices of traditional engineering and software engineering.
3. Many contemporary software practices fall short of what would be expected from the rigor of other contemporary engineering disciplines. Computer programming, as currently practiced, is not an engineering discipline. Instead, as Pete McBreen [MCB] emphasizes in his book on this subject, most computer programming is best characterized as *craftsmanship*, not engineering.
4. Software practice must continue to evolve as an authentic engineering discipline, not for its own sake, but for the society that is increasingly vulnerable to defects and hazards in engineered products that depend on software as an intrinsic part of their total design.

D. SOFTWARE CONTEXT

1. The Algorithm Issue

Computer programs have been described in many ways, both formally and informally. Some descriptions use mathematical notation. Other descriptions rely on natural language. Almost every attempt at a brief description is incomplete and oversimplified. There are so many ways to design computer programs, and so many language models for doing so, that a common description of programming is “nailing jello to a wall.” From an engineering perspective this is not satisfactory.

One of the most often quoted statements about computer software is the cogent and laconic description of software from Niklaus Wirth [Wir1976] in his famous definition of a computer program:

$$\textit{Program} = \textit{Algorithms} + \textit{Data}$$

In its most elementary form, software is simply a computer program. However, modern software engineering is characterized by more than algorithms and data as evidenced in examples from a software literature that goes beyond algorithmic reasoning [Boe84], [Bro85], [Mah02], [PeSv96], [Pro98], [Ber92], [BC85], [BW77], [BCK07], [BR99], [ER03]. It is true that algorithms lurk beneath the surface and are an essential part of a software construction. It is also obvious that incorrect algorithms are the source of many software problems.

However, large-scale contemporary software systems are more appropriately seen as levels of abstraction, not simply as a collection of algorithms. These abstractions are organized, modeled, and constructed from many other software components, building-blocks. The above artifacts include types/classes, methods, class attributes, simple variables, packages, modules, patterns, and many kinds of reusable components. These facets of the software engineering problem are, in turn, tied together in a variety of relationships and configurations that include: dependencies, generalizations, specializations, associations, compositions, aggregations, collaborations, and temporal interactions. As engineering is applied to the design and construction of software, internal controls are as important as algorithms.

The recognition and acknowledgement of components and relationships simplifies software reasoning and also complicates it. In my software engineering classes, I tell students that, for large-scale software products, the concern involves components, connections, configuration, and controls. Even so, every design eventually comes to the point where someone must develop computer programs. Sometimes this requires inventing entirely new algorithms or revisions of existing algorithms.

2. Change

The first thing to note about a computer program is that it is concerned (in most cases) with change. A software entity (in most circumstances):

1. effects change,
2. detects change,
3. or both.

In this respect, a computer program corresponds to the engineering issues relative to forces described in Chapter IV. If change is neither required nor present, it is unlikely there will be a need for a software solution. The software analogue of *displacement* is a change of *state*. An engineering question is what is required to ensure the change of state is appropriate to the existing state of the context in which the change of state is effected? This is primarily an engineering problem, not a programming problem. Its solution is often left to the programmer, but that is simply an abdication of responsibility on the part of the engineer. On one project where I was a minor player, a communications satellite project being programmed in Ada, every line of code in the programs had to be justified in terms of the engineering specification. This project required almost line-by-line annotation of the code. Programmers were not encouraged to be creative. Controls were designed for each change of state by the software engineers to ensure that changes of state were continually checked for accuracy and correctness. Programmer source code was expected to carry out the state changes as prescribed. The software engineer can include assertion controls in software specifications that bound the inventiveness of an exuberantly creative programmer.

Associated with, and in collusion with state changes, software has many kinds of behavioral and informational properties. These include computations, decision-making, modification of data, transfer of control, and recording of actions. Böhm and Jacopini summarize these as: sequence, selection, and iteration in their famous paper on elementary control structures [Boh66]. The variety of mechanisms to effect these changes is too numerous to enumerate here. However, anyone with even a meager knowledge of software will appreciate the complexity as well as the quantity of both data representation and actions that can be applied to the data.

It is the complexity of software, and the corresponding potential for both success and failure that is at the heart of this inquiry into the potential or, even the necessity for, the application of engineering principles and methods for software creation, management, and control.

3. The Management of State Changes

In the world of physical design, where change is also a factor, most changes can be measured in terms of some kind of displacement (included in definition of “work.”). In software, change can be identified, but not always measured, as change of state(s). The software may detect a change of state in the external environment and record/report that change of state. Or, the software may actuate a change of state in the external environment and record its own action on some media.

The change of an integer in a computer program from one value to another is a change of state resulting from the force of the logical constructs in the design of that program. The implications of this change are no less grave than the same kind of change in a physical environment. Further, if the new value of that integer is critical to correct operation of the program in a safety-critical environment, the physical and software changes of state are equally important.

Software changes, even in computer programs can sometimes be measured, even when they are discrete events. However, discrete events cannot give us information such as sensitivity analysis. That is, the classical $y = f(x)$ is not applicable to discrete phenomena. Some changes of state, at a level of abstraction beyond discrete data can be measured, and potentially predicted within the discipline of sensitivity analysis. Consider a function,

```
function Compute (Data : in Float) return Float;
```

In the above example, the function is dealing with data that represents a non-discrete value. Even though the underlying machine representation might be (not required to be) composed of discrete binary digits (bits), The software abstraction is well beyond the limitations of that representation. A software engineering artifact is much larger and much more complex than simply a set of discrete states.

A strict boolean view of the computer where software is deployed leads to the error of reasoning that software is also confined to a true-false model. Since the underlying computer is bi-statal, in which bits can be turned on and off, there is the temptation to think that reasoning about software is restricted to boolean entities. Modern software engineering tools are in place to help us transcend that level of abstraction. When we were programming in Octal by entering data through the front-panel switches, we had to think at this bit-oriented level. When displaying a message-box in MS-Windows, we need to perform a logical OR on the bit-field for a low-level function to effect a particular kind of message-box. In higher level languages such as Python, Ada, or Java, using a class-oriented programming model, we can display that message-box without knowing anything about the underlying bit-mapping of the parameter list. In early computer programming, we once needed to understand the bit-mapping of the word-oriented machine for floating-point arithmetic. It is should be unusual for anyone, except a compiler designer, to be concerned about such low-level details.

E. SOFTWARE IN PRACTICE

Prior to the detailed discussion of software engineering, in an engineering context, it is appropriate to summarize the differences between three distinct, but closely interconnected aspects of software practice: software architecture, software engineering, and the programming activity.

1. Software Architecture

An architecture is concerned mainly with a high-level design that describes the artifacts most likely to satisfy the requirements of stakeholders. Patterns, standards, esthetics, and consistency are essential. The architecture, once established, should remain

a stable model throughout the entire process. The architecture is made-up of components, connectors, and a configuration. Architecture is a foundation onto which the stakeholder requirements can be represented in more detailed during engineering and programming [BCK07], [ShG96], [Gor06], [Pau02]. For example, the software architecture may describe the artifacts of the design in terms of a stakeholder view, but may defer consideration of the controls needed to enforce that design stability to the engineering phase.

2. Software Engineering

Engineering is also a design activity [Pet85], [Wri89]. Ideally, the engineering will not alter the architecture, but there are sometimes unpredictable issues that require revisiting the architecture [Gor06]. An engineering process, including a software engineering process, must specify the details for implementation of the design. Those details include derived requirements, quality assurance, risk management, economic trade-offs, constraints, and controls, along with the other engineering concerns [Pre05]. Of particular concern is how the engineer extends and refines the model to maximize the absence of the kind of errors that typically occur during the programming process. Components specified by the engineer must include assertions, constraint management, levels of abstraction, well-factored modules, components and classes, and error management mechanisms (controls for error detection/prevention/correction) that will ensure maximum integrity of the final product that comes from the construction process carried out by the programmers. Engineering is focused on (with apologies to Gerald Weinberg for the pun) GIGOless programming [Wei98].

3. Programming

This is the construction process. While it is far more complex than driving nails into the studs of a building under construction, it continues to be more craftsmanship than engineering [Mcb02], [ER03, Chapter 4]. The programmer is similar to an engineering technician, but requires a more sophisticated skill-set than is usually required of a technician. The skill requirements for the programmer are even more important in the

absence of well-designed architectures and poorly developed engineering specifications. Under the best of circumstances, the specifications for a given computer program are completely specified by the software engineer. However, this is rarely the case in real software projects. Consequently, completion of software depends heavily on the skill and knowledge of the programmer [Hum95], [Bec00], [Mcb02]. This reality runs counter to sound engineering practice, but will be the prevailing practice until there is a raising of the level of competence among the engineering designers.

4. Supporting Activities

Requirements development is a specialized activity that requires more than design and architectural skills. It requires a combination of a research, technology, and human-factors that people with a software technology focus often do badly. In some cases, they are weak in the technology of the application domain. In other cases, they represent extraordinary technological power, but fall short in their human interaction capabilities. I have written an entire separate book on requirements development, including case histories. The topic of requirements is beyond the scope of this present work.

Software engineering includes numerous other specialty areas such as risk management, programming languages, operating systems, automata, and algorithm analysis that software engineers need to understand, but often don't. While these supporting activities are important – even essential, they are not the focus of the engineering model software engineering in this work.

F. THE ENGINEERING CONTEXT – A REVIEW

Chapter III laid down the engineering context for software engineering. It also presented a definition of engineering and promised to examine, within the bounds of that definition, whether some practices in software development and management conformed well enough to that definition to legitimize the term “software engineering.” To review, our definition of engineering was,

Engineering is the organization, application, and management of settled(dependable) knowledge using the tools of science, mathematics, and logic, along with knowledge, experience, and artifacts derived from previous engineering efforts, for reconciling conflicting forces/constraints, controlled within defined tolerances, to effect an economical, risk averse, maintainable, failure-tolerant design with a predictable result.

This definition is summarized in the following Venn diagram.

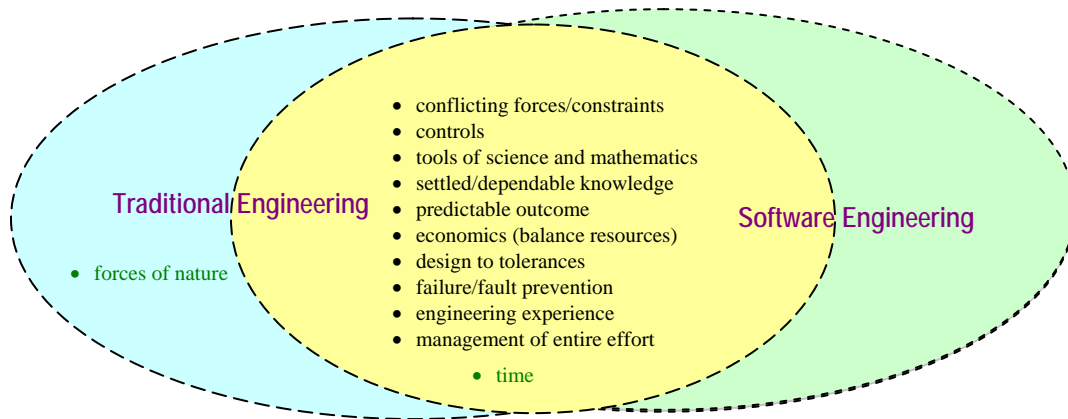


Figure 6. Engineering Definition

As shown in the Venn Diagram, many of the properties of traditional engineering are (or should be) present in software engineering. The element often outside the intersection is called “forces of nature.” One *natural force* common to both software engineering and standard engineering is *time*. While the issue of *natural forces* is a concern in other branches of engineering, other factors loom larger in software engineering. Even so, it is important to deal with the issue of *natural forces* early before going on to the other aspects of engineering.

1. Nature (Natural Forces)

Earlier, we acknowledged the absence of natural forces (i.e., physical displacement) in software. This is sometimes regarded as an obstacle to a credible model of engineering for software practice. In software, every action (force) involves some change of state, logical and/or physical. The presence of state transitions (i.e., virtual

displacement) is fundamental to software. In fact, contemporary software modeling tools such as UML include State and change-of-state as fundamental features of the model [RuJB05, pp. 597-614]. The mapping of state changes in real-world objects to their corresponding state changes in software is common in contemporary software design practice. This is due, in large part to the increased awareness and use of object technology [ScM92]. Object technology is a significant departure from the earlier style of software design where the emphasis was on the procedures and algorithms.

In a discrete domain such as software a relationship is either *true* or *false*. The result is that one instance of *false* can cause an entire design to fail. In the continuous domain, minute fractional differences can occur, and those differences can foretell a potential problem before a failure occurs, but that foretelling can signal the need for preventive action. In a discrete environment such as a computer system, failure of a single-bit is likely to be unpredictable. This requires the software engineer to design in a completely different way from the engineer in the natural world. Modern software engineering tools and practices can support a model of failure-tolerant design even though that majority of software is not designed that way. This is analogous to the physical engineer that anticipates failure in the design rather than letting a system fail during critical operation.

Chapter III also presented an engineering perspective that demonstrated how the essence of engineering can be present even in the absence of natural forces. Further, the entire notion of natural forces is not exclusive to those tangible elements of nature. Rather, natural forces are those, which inherently constrain an engineering design, whether they are logical, mathematical, physical, or economical. As noted in that chapter, the recognition of the constraints of economics was a major factor in the development of modern engineering. Further, we found that the notion of engineering as simply *applied science* in the sense of physical sciences is a naive view in today's world.

Some important work in natural forces for software included pursuit of a concept of software physics [Kol85]. While this was interesting work, it failed to materialize in to a definitive model of natural forces, and left software engineering exactly where it had been before this bold research was published. The absence of natural forces in software

does not absolve the software engineer from the responsibility to apply best engineering practices to a software design. Rather, the very absence of natural forces requires, even demands, that software used in mission-critical and safety-critical applications be subjected to the best possible engineering practices available. These include process practices, but process practices are not sufficient. We also require that engineered software be correct and defect-free by design. This means software engineers must go beyond the programming practices that have characterized software during the past fifty-plus years and insist on applying best engineering practices, as described in the mapping to the *reference discipline* described in earlier chapters.

2. Intersection of Software and Physical Engineering

The intersection of software engineering and traditional engineering (i.e., engineering based on natural forces) is derived from the Venn Diagram. This includes:

1. conflicting forces/constraints (trade-offs),
2. continuous control with feedback,
3. rigorous design tools based on science, mathematics, and logic,
4. settled/dependable knowledge,
5. predictable outcome,
6. economics (balance resources)
7. design to tolerances and defined constraints
8. failure/fault prevention (including risk management and software safety),
9. engineering experience
10. management of entire engineering effort.

Note that there are ten items in the current list where there are eight in the list from Chapter III. This is because we have separated the two topics, constraints and design-to-tolerances into two separate topics.

G. MAPPING SOFTWARE ENGINEERING TO ENGINEERING

The following discussion is a mapping of software engineering to the engineering definition given in Chapter III, titled “The Engineering Context.” The reader will recall

that Chapter III concludes with a discussion of those facets of software practice that continue to elude even the most earnest attempts to apply engineering to the design and construction of software products. It must be noted that the contribution of John V. Guttag [Gut77] is essential to the success of this mapping, along with the work of Bertrand Meyer [Mey00]. Also, we use the Ada programming language to illustrate the mapping when source code is appropriate. All three of these resources enable us to map the work from the chapter on “The Engineering Context” to engineering examples as actual software artifacts

Since this paper must be bounded to a reasonable size, there is an emphasis on four of the items from the above list:

1. forces, especially conflicting forces,
2. settled (dependable) knowledge,
3. design to tolerances; this is a separate chapter
4. failure prevention.

1. Forces

Having determined in Chapter III the fundamental importance of reconciling conflicting forces, it is time to examine this idea in the context of software. Also, earlier in this chapter we acknowledged the absence of the kind of conflicting physical forces that characterize more traditional engineering. Even so, conflicting forces are present in software design just as they would be in any kind of engineering effort.

a. Conflicting Forces

In software, the trade-off might be whether an algorithm is better designed with a loop or with recursion. Another trade-off might be whether the data structure for a design is better as a depth-search or a breadth-search tree. Of course, these trade-offs will trace directly back to the physical environment in terms of the storage-space versus compute-time, but they are still software trade-offs, and present a set of engineering challenges. Many of the data structure problems in software introduce engineering trade-off problems. These include the concerns about such things as data base normalization,

whether to use a sequential structure or a random structure for the storing of data, and the potential for corruption of data when multiple processes require read/write access to it.

For each of the problems just described, the software engineer must calculate the offsetting benefits of the conflicting design options. In many of these situations, the data becomes a kind of “force of nature” simply because of its magnitude, required availability, and potential for corruption when managed in error.

The very structuring of the data closely conforms to an engineering problem in the physical world, and that structuring is in the domain of the software professional. For example, when there is a need for some kind of repository in a software design, the engineer must examine the relative merits of known kinds of data structures. Questions include whether the data should be persistent (reside in secondary memory) or volatile (reside in primary memory); does the data require some kind of indexing scheme; will it be transported through the system; will it be transformed or augmented during the life of the software that processes it; what controls are required to keep the data stable as it passes from one stage of processing to another; are there security issues? These questions, along with many more, are required to effect an engineering solution to the management of that data.

A key *correctness-by-design* [Bar03] concern of every software engineer is the continual validation of the data being managed in a software product. This concern often requires rigorous design with data-types and formal constructs such as pre-conditions, post-conditions, and invariant statements [MiM02]. The design of constraints for typed-data is a common practice in software engineering. While the engineer for the physical domain might specify a range of values for a solution, the software engineer will refine that range as a set of well-formed types to prevent conflicts between or corruption of the specified data.

b. Conflicting Forces: An Example

An example of the above is the Ada protected-type where data that needs to be accessed by multiple software processes (tasks) can be insulated from unwarranted references. The software engineer designs this kind of protection into the product without any input from the engineer in the physical domain.

```
package Restricted_Integer_Definition is
  type Restricted_Integer is private;
  protected Restricted_Integer_Management is
    procedure Update_Restricted_Integer (Input : in Restricted_Integer);
    function Read return Restricted_Integer;
  private
    Data : Restricted_Integer := 0;
  end Restricted_Integer_Management;
private
  type Restricted_Integer is range -127..127;
  for Restricted_Integer'Size use 8;
end Restricted_Integer_Definition;
```

Figure 7. Restricted Integer

In this example, the tolerances for the type `Restricted_Integer` are tightly bound to a predetermined range and a fixed number of bits (`Size = 8` bits). This specification is inviolable. The programmer cannot make a mistake with this. The management of the `Restricted_Integer` is guaranteed to be updated in mutual exclusion even in an environment where multiple concurrent threads are trying to use it. It is guaranteed never to exceed its own bounds (-127 through 127). Any programmer that tries to misuse values of this type will be thwarted by the compiler.

```
package Restricted_Real_Definition is
  type Restricted_Real is private;
  function "+" (L, R : Restricted_Real) return Restricted_Real;
  function "-" (L, R : Restricted_Real) return Restricted_Real;
  function "*" (L, R : Restricted_Real) return Restricted_Real;
  function "/" (L, R : Restricted_Real) return Restricted_Real;
  function "=" (L, R : Restricted_Real) return Boolean;
  function ">" (L, R : Restricted_Real) return Boolean;
  function "<" (L, R : Restricted_Real) return Boolean;
private
  type RReal;
  type Restricted_Real is access RReal;
```

Figure 8. Opaque Restricted Integer

In this opaque-type example, anyone that needs to use `Restricted_Real` will have no access to any information about its structure. Everything is deferred to a package body (the implementation) where tightly controlled, designed to tolerances, methods will be designed to prevent any of the typical problems associated with floating-point values. For this discussion, it is not necessary to examine the implementation. We could apply pre-conditions and post-conditions to each of the numerical methods, but the details of the invariant type, `Restricted_Real` are hidden in the private part, thereby allowing rigorous low-level implementations of the corresponding functions.

We alluded earlier to the potential for engineering conflicts in the design of algorithms. The metrics for the chosen set of algorithmic options, when understood by the well-trained software engineer, will be taken into consideration in the choice the algorithm. Even the best set of choices will still required several levels of evaluation to ensure that those choices are conflicting with each other.

Creation of reusable components is an especially powerful engineering idea when leveraged in the environment of a good *software reuse library*. One of the most useful reuse libraries for military software is CAMP, the Common Ada Missiles Programs (a classified set of library routines), that are used in the development of a wide-range of military software systems. Another set of reusable software routines for engineered software are the well-known Booch components, used in a large number of Ada software systems. The availability of libraries helps the software engineer to make decisions about the trade-offs represented by the relative merits of the algorithms. There is still a great deal of work to be done to bring software reuse to the point where software engineering practitioners are satisfied with them.

2. Dependable (Settled) Knowledge

Chapter III stated that *settled knowledge* is that knowledge agreed-upon by practitioners in some engineering field. The need for a larger knowledge base is an on-going problem for software engineers. The concept of *settled knowledge* is as important in software practice as well as in standard engineering practice. Further, engineering

knowledge continues to expand for software practice just as it does for other kinds of engineering. Software engineering does not yet have the extensive reusable knowledge base available to classical engineering practitioners.

Many software practitioners are either ignorant of, or cavalier about the available settled knowledge from computer science and software experience. Many others do benefit from that knowledge. One example of this kind of knowledge is the so-called “Big O” performance notations used in software design metrics. Another is the growing library of pre-coded software routines that populate the specialized software reuse libraries.

As noted in Chapter III, engineering knowledge is characterized by its dependability, regardless of its origins. Dependable knowledge can originate in the non-physical world or the physical world. The important thing is validate its authenticity, its relevance, and its value to the engineering project. If the knowledge cannot be verified as authentic, it is of little value. This chapter includes an assessment of this assertion in the realm of software, particularly for what we would like to call software engineering.

a. Software Engineering Knowledge

As noted above, knowledge must be dependable. Fundamental is the principle of least surprise [Rie05]. This is particularly true when following a *correctness by design* approach to engineering the software. The question for software engineers is whether a given item of knowledge satisfies the dependability rule. Further, knowledge must support the goal of predictable outcomes. A predictable outcome, for our purposes implies three predictable properties. The first two are due to Boehm as described in Pressman [Pre05] and called *validation* versus *verification*.

1. solving the right problem (i.e., valid result)
2. solving the problem correctly (i.e., solution is correct)
3. time to solve problem is bounded [Burns01], [BR99], [PeSV96]

There is an implication in this list that the resulting product will be defect free, do exactly what it was supposed to do, and do it within a predetermined time constraint. If these properties can be satisfied, it is probably safe to say that an engineering approach was, at least a contributing factor to the success.

Satisfying this is one of the central challenges of software engineering practice. Instead of applying engineering discipline to the design and creation of software, a lot of deployed software has been driven by unrealistic time and budget constraints that emphasize the notion of “good enough” instead of well-engineered. This approach is based on the dictum, “Just get it done. We’ll worry about the niceties later.” The attempt to apply engineering practices, the “niceties,” had to be subordinated to getting operational software deployed and executing.

The third property of measurement, “time ...” is especially elusive. A review of the literature about software metrics, including algorithm metrics and deployed program metrics reveals that there are so many variables in software that absolute predictability remains an unsolved problem. Some progress has been made in this area, but the ability to apply predictable design metrics remains a difficult problem [PeSV96].

b. Measurement and Metrics

Engineering knowledge [Vin90] is characterized by some unique properties. One of the most important is its measurability. Traditional engineering, in the natural world, has benefitted from the ease with which observable physical phenomena can be measured. Such measurements, often called metrics, are often collected, classified, and published in standard tables. The torque of a standard bolt composed of a specified substance makes the job of a mechanical engineer much less of a guesswork problem than the performance of a *for ...loop* translated from a high-level language into machine-code for a targeted computer. The mechanical engineer can calculate the relationship of that bolt’s torque relative to where it will be used and specify what setting should be used on a torque-wrench used in a manufacturing (or repair) process. The software engineer is often unable to make such exact computations due to the variations in the targeted environment.

(1) Dependability and Reliability. It is often noted that software is concerned with discrete phenomena. This information is used to suggest that, because a single bit in a machine can corrupt the entire operational environment, the kind of design metrics that prove so dependable in physical engineering are not as available to software engineering. Further, those same naysayers observe that, since every algorithm depends on the evaluation of discrete conditions, the instability of a design at higher levels of abstraction suffers from the same vulnerabilities. While this may have some truth when considered in the context of conventional programming practice, it is not an inevitability. In particular, it is not inevitable when the knowledge accumulated from good software engineering practices are applied to the design of software.

The fact that most of contemporary software practice does not subscribe to the concept of collection, classification, and utilization of relevant knowledge is not a reason to dismiss its importance. For example, it is often noted that probability is not applicable to software because of its discrete nature. Yet, the accumulation of knowledge for *risk management* can lead to the use of Bayesian analysis for the prediction of errors of a particular variety in a given domain. A software development organization that designs and publishes software for inventory systems can know, from good record-keeping, that a particular kind of error(s) typically appears at a known stage of the development process, they can take action that will foreclose on and prevent that kind of error more easily. Settled knowledge, in software engineering requires documentation of the history of error-creation process. Such knowledge allows the engineer to apply risk prevention and mitigation strategies. This is the essence of process improvement, but it is seldom seen as important in software organizations. Instead, most organizations where I have worked, and most that I know of, are concentrating on getting the job done as soon as possible, not as accurately and error-free as possible. This leads to the old-saw, “Why is there always time to do it over, but never time to do it right?”

While there is a need for more information to be collected and published for the desired level of dependability engineering in software practice, the absence of such information is more a lack of engineering attitudes among software

professionals, not the inability to collect it. Information about errors in the software process needs to be recorded and used as part of future engineering efforts. The knowledge of errors, defects, and difficulties can be passed from one project to another. In this case, every new project is begun using institutional or standard engineering knowledge relevant to each new project. This is an engineering attitude which will have a major impact on the dependability of future software. Instead of relying on individual knowledge, or folklore, accumulated engineering knowledge, from past software engineering projects, will feed into the reliability of future software efforts [RieN07].

This is not to say that knowledge is not accumulated or never passed on. Much of the work done in the fertile fields of software reuse is devoted to the accumulation, classification, and standardization of software knowledge. In fact, the still emerging field of software reuse, and the continued work on designing and creating software components, at both the source code and executable levels, represents one of the best engineering practices in software engineering. Unfortunately, there is a widespread “not invented here” (NIH) attitude among programmers, along with an incomplete taxonomy scheme to make reuse as widespread as it should be. Also, the proliferation of software components, developed in a multiplicity of languages with a plethora of inter-operability problems, sometimes presents more problems than it solves.

Even so, software reuse is more widespread than many software engineers realize. The executable frameworks for “windows” programming and database management are largely built over reusable components. Scientific programmers routinely reuse Fortran code provided by both their peers and from public sources. Libraries of algorithms and data structures are widely available. No programmer should ever write another stack, linked-list, square root, or fast-Fourier transform routine. It is absurd for a professional programmer to code a routine to convert floating-point numbers to text. Huge libraries of reusable software have been placed in accessible repositories. Practicing programmers often do not know about these repositories, or the difficulty in finding the library routine s/he wants is greater than simply writing it from scratch. I sometimes give the following exercise to a programming class:

Write a routine that will take two items in a parameter list and return them as a swapped pair.

This routine is already in any library of pre-coded routines, and should never have to be written, in any language. However, the exercise has the benefit of demonstrating the vulnerability of even the simplest of routines. It is amusing to see how many experienced programmers get it wrong on the first try. When such a simple algorithm can be in error, how much more likely is it for a complex algorithm to be in error?

As noted in the preceding paragraphs, a lot of software engineering knowledge is readily available in the form of pre-published algorithms, classes, subroutines, and library modules. These libraries are analogous to the kind of published knowledge the mechanical engineer uses when choosing a given bolt of specified torque and threads-per-inch. As long as the knowledge from those software libraries is dependable, it is as real as any knowledge from the natural world. It is engineering knowledge – software engineering knowledge – but valuable in the engineering of complex software systems where dependability and predictability are the primary goals.

(2) Performance Knowledge. There is a lot of metrics data available already for software performance engineering. One of the most overlooked metrics is the so-called “Big O” metric. While Big O is not an absolute metric, it does provide relative performance metrics between algorithms. The “Big O” metric can help the engineer make design choices from among a collection of alternatives for which such relative metrics have been developed and published.

At a lower level, the engineer can use a technique analogous to theblig analysis in computing the efficiency of a set of instructions as they are linked together at the lowest level of the machine [Mog80]. This is rarely done for large-scale software, but it is not an unusual practice in response-time intensive embedded software.

Peter Denning demonstrated the importance of performance metrics in his contribution on thrashing. In this work, Denning designed a software engineering solution to what was, in part, a physical engineering problem called

“thrashing.” This problem originated with how a specific operating system managed the swapping of information between a secondary memory device and the primary memory of a large mainframe computer. The solution was a clear example of how engineering principles could be applied in software to control, not only the thrashing problem itself, but also to accommodate the additional demands of the control software within the operating system [Den68].

Another important kind of performance analysis is that used in the design of software that involves concurrent, communicating processes [Burns01]. It is here that one of the most difficult natural forces comes into play in the design of software: time. In the real-time domain, multiple processes are frequently interacting with each other. The ability to dependably schedule these processes is a serious engineering problem. This problem is compounded when those processes are required to communicate. For example, a signal is received, as an interrupt, in some part of the CPU’s memory. That signal is immediately stored in a software buffer where it can be retrieved by yet another program. Depending on whether the design is synchronous or asynchronous, other programs come into play to ensure the signal is handled properly. There might be ten different programs (tasks/threads/processes) involved in this process, each of which must communicate with another, but without writing over the data before it is relayed to another process. There is no more difficult engineering effort than the design of a system based on concurrent-communicating processes, and that difficulty is greater as the number of processes (tasks) becomes larger. The engineers who specialize in this kind of design are focused almost entirely on the software, not on the hardware. The mathematical challenges of schedulability and mutual exclusion are real engineering problems.

Another problematic engineering challenge is software stability. By stability, we mean the software is not subject to whims or opinions that can be exploited according to each situation in which it occurs. Gravity is always gravity. Gravity is cheap. It is also dependable. It can be measured. No one can express an

opinion about its validity. Though its effect may vary with the square of the distance between objects, that is an effect that can be measured, and conforms to the settled knowledge that we have about gravity.

What is true about gravity is true of most phenomena in the *natural* world. Conformity with the laws of nature is one of the foundation ideas of every accepted engineering practice. The ability to exploit those laws, the settled knowledge about those laws, is what has made engineering so successful in so many ways.

The challenge for any other discipline that aspires to become a member of the engineering club is to find a similar knowledge set that can provide the same level of measurability, predictability, and consistency as the knowledge that supports respectable branches of engineering. Without that kind of knowledge, membership in the club will continue to be denied.

c. Software Engineering Knowledge (SWEBOK)

There is a recently published work from the IEEE referred to throughout the Software Engineering community as SWEBOK [1]. SWEBOK represents a large collection of concepts and ideas, and standards based on software engineering research, analysis, and experience.

The categories of knowledge covered in SWEBOK are:

- Software Requirements
- Software Design
- Software Construction
- Software Testing
- Software Maintenance
- Software Configuration Management
- Software Engineering Management
- Software Engineering Process
- Software Engineering Tools and Methods
- Software Quality

Taken as a whole, this is an impressive list of requirements. However, it does not establish software practice as an engineering discipline beyond what one would expect of any other discipline. For example, a similar list could be constructed for the practice of neurosurgery, funeral home director, or curriculum development specialist. The concern is not that this list is unacceptable. Rather, the list is not complete with respect to understanding software engineering as an engineering discipline.

3. Design to Tolerances

This is covered in the following chapter.

4. Controls for Failure Prevention

There are a large number of opportunities for the application of controls in software. Many software designs overlook those opportunities in the haste to produce a solution to a problem quickly so the next problem can be solved. A lot of software is design according to the washing-machine example. The set of computer programs moves from one cycle to another blindly.

This is where the software engineer is different from the computer programmer and more like a conventional engineer [Ayu03], [Hal97]. While the experienced and conscientious programmer is dedicated to producing a program that works under all circumstances, the software engineer digs deeper into the design to ask questions about the risks, the unexpected, and the unknown. Then that engineer specifies a set of controls for the software that deals with those risks. Controls will often be in the form of assertions in the specification part of the source code (e.g., Eiffel contracts, Ada range constraints, Java IContract specifications) [MiM02]. Where the programmer may resist the inclusion of such controls (“They slow down the execution time”), the engineer understands the essential nature of them.

In contemporary software development, controls design is an engineering responsibility. In the absence of a software physics that impose controls based on natural forces, the software engineering must invent the physics of the software environment. This involves specifying the upper and lower bounds for numeric values, the set of

allowable states for instances of an object class, and the rules for relationships between the many entities of the software being developed, where the concept of state is fundamental in modern software engineering practice [ScM92].

The design of those controls is as important as the design of the corresponding algorithms. This is one of the most vital responsibilities for the modern software engineer. It requires experience, knowledge from past failures, a body of documented history from similar projects and a substantial amount of intuition. The controls emanate from the fundamental question every engineering, whether in software or traditional engineering must ask, “Where could my design fail?”

If the software engineer is not designing for failure, the design is not complete. If the designer does not anticipate what can go wrong as well as what is required to produce the intended solution to a problem, the design is wrong. This ability to design for unexpected failures as well as for routine success is what makes an engineer, including a software engineer, a professional rather than a skilled tradesperson, a craftsperson, or a day-to-day programmer.

This is not to suggest that conscientious programmers do not design for success and try to anticipate failures. They do just that. Some programmers are also good engineers. However, the programmer is often under the stress of just getting the job done so s/he can move on to the next project. Just as standard accounting benefits from a separation of responsibilities, in the domain of controls design for software, the professional software engineer must be in a position to take responsibility for failures in a design specification, including those that should have anticipated failure.

H. WHAT WE CANNOT YET ENGINEER IN SOFTWARE

This section reflects the opinions of the author, but it is based on years of software experience rather than intensive research. Even with an engineering model built over the original vision of this dissertation, we much recognize that there is a long way to go before we have a comprehensive model for software engineering that satisfies the kind of rigor we identified in the foregoing pages.

1. Computer Programming

As noted several times, software engineering is a discipline that helps to control and stabilize the software construction process. As reusable components and patterns are added to the repertoire of options in software reuse repositories, more and more programming will be able to make use of those components. However, it is the nature of software, and of humans who conceive of problems that can be solved with software, that new kinds of software solutions will be required for as long as mankind is able to be creative and imaginative. Therefore, people who can write working computer programs will be as necessary in the future as they are today.

That being said, the continual advances in the engineering models will inevitably produce new tools and languages that will make future programmers view present-day programming with the same admiration we have for those who could carve gigantic pillars from solid rock using nothing but a chisel and a mallet. As long as we expect programmers to code algorithms over and over in the languages currently in use, they will be the equivalent of those ancient stone carvers.

2. Risk, Testing, and Quality Management

The craft of testing does present many opportunities for engineering. However, testing is largely about anticipating and managing the risks associated with the construction of engineered artifacts. While there will be a growing number of regimens for accomplishing this task, and those regimens will be continue to be grounded in engineering, the ability to anticipate risks is a skill that depends on human experience. That experience enables a creative approach to the anticipation of risk.

Risk management is one of the most important aspects of all kinds of engineering. Barry Boehm [BOE] can be credited for forcefully bringing it to the attention of the software engineering world, and this may someday be credited to him as one of his most important contributions to software engineering.

3. Human Factors

Industrial engineering, mentioned several times in this paper, was once noted for how it treated human beings as if they were interchangeable parts [Mog80], [Tay11]. That early model of industrial engineering has largely vanished. However, a similar phenomenon intruded into the management styles of software organizations and resulted in decades of dependence on counter-productive approaches to software management such as Royce's Waterfall, and its many variations [Roy70], [Roy98]. The recent revolution in software practice, largely software process that emphasizes people over process has changed how humans are managed throughout much of the software industry. That trend is likely to continue [Bec00], [Mcb02].

Even as human beings are treated with greater respect, we must keep in mind the need for sound engineering principles and practices in the creation and management of future software projects.

4. The Ideal Process

Little needs to be said about the ideal process or the ideal engineering model, software or physical. Neither exists. That being the reality, it does not absolve software professionals from striving toward an ideal. Software engineering, when practiced as an engineering discipline, move the practice a little closer to the ideal. There is the obligation to do exactly that.

VI. DESIGN METRICS: DESIGNING TO TOLERANCES?

A. INTRODUCTION

In the chapter on the engineering context, we noted the importance of designing to tolerances. In my published paper on this topic [RieJ07], I call this “snugness of fit.” It is a concept not commonly included in the design of software since the notion of tolerances is seldom applied in ordinary computer programming. Tolerances are often used in craftsmanship. For example, I have watched Japanese joinery craftsmen snugly fit a complex arrangement of wooden pegs into beams and trusses to build nail-free furniture and temple structures. In the case of the joinery process, the design is in the mind of the craftsman along with the actual building of the artifact.

What do we mean by tolerances? In this discussion, engineering tolerances include the relationships between the artifacts of a design along with the constraints and controls required to ensure the stability and predictability of that design.

In traditional engineering, tolerances are part of the overall design, but the designer (engineer) is rarely the person who does the actual building of the design. This means the designer/engineer must produce a design that can be realized by a separate builder. Therefore, the software engineering challenge is to offer design artifacts where the tolerances are pre-set so the craftsperson (e.g., programmer) cannot introduce common errors or to ensure that such errors, whether at run-time or during construction, can be detected. Once detected, they must be mitigated in some way. This is an engineering responsibility, not a programming task. It is also a risk management task.

B. CHAPTER THESIS STATEMENT

1. Engineering is more about design than implementation of a design. Therefore, the software engineer must focus on doing the best possible design so the programmers can have a clear concept of what is required.
2. There is a model of software engineering that can be managed with design constraints and design tolerances that can limit the programmer from doing ad hoc actions that violate those constraints and tolerances.

3. There exist programming languages and software tools to support the implementation of design to tolerances and constraints.
4. When software is designed and implemented according to a constraint/tolerance model, that software fulfills an important part of our definition of engineering. We can say that such software has been engineered.

C. TOLERANCES, CONSTRAINTS, AND CONTROLS

1. Design Issues

When a software design includes a set of tolerances, those tolerances become the constraints and controls that govern the behavior of the resulting programs. In turn, those constraints and controls govern what the programmer can do during the implementation of the engineering design. In software engineering, those constraints and controls substitute for the “natural forces” that restrict the traditional engineer from trying to do things that are impossible in the physical world.

The goals of engineering (ergo, also software engineering) as described in Chapters III and IV requires us to include capabilities that will support those goals. Physical engineering disciplines automatically support an ability to design to tolerances. This same concept is a challenge for software engineering. The purpose of design tolerances is to enhance the dependability of the engineered product. This is no less important in software engineering than in any other kind of engineering. The goal is “correctness by design” a term often used in the development of software using the SPARK Examiner [Bar03], [Mey00], [MiM02].

The software engineer that understands how to design to tolerances will help the programmer create more dependable working programs. That is the responsibility of the engineer – the software engineer – not simply the province of the craftsperson (computer programmer).

2. Nature of Design Tolerances

We examine tolerances in terms of their inherent properties along with the levels of tolerance in a given design.

a. Software Tolerance Properties

For our purposes, tolerances include the following properties:

- a. constraints
- b. behavior
- c. availability (possibly visibility)
- d. metrics
- e. rules of interaction

Tolerances are not simply the numeric values of an entity. They also include the constraints and controls on each entity of the design relative to other entities in that design. In traditional engineering, the constraints include those imposed by nature. For example, gravity and friction are constraints engineers must acknowledge in a mechanical design. These constraints are based on the physical nature of the non-software world, but also, in modern engineering, include the constraints of the social, economic, and political world.

The second property, *behavior* is about what that entity is allowed to do and the context in which it is allowed to do it. Again, in the natural world we can predict the behaviors of the artifacts of nature, but not always the relationships between those artifacts.

The last three properties (availability, metrics, and rules of interaction) are important in software engineering design; they also exist in various forms for engineering in the natural world.

3. Categories of Software Tolerance

For this discussion, tolerances can be organized into five general categories:

- absolute
- range (tolerance limits)
- conditional
- relative
- probabilistic/statistical

The notion of absolute tolerances for software may sound oxymoronic, especially when viewed in the context of tolerances in general engineering. However, absolute tolerances are like those in general engineering because they are stated in actual values. Those actual values can be a range constraint, an enumeration of specific values, or an invariant bounds (even as a percentage) for the elements of a component or for the component itself. An example of an enumerated absolute tolerance is the Ada enumerated type where each value of the type is stated, by name, independently of its machine representation. Sometimes we might want to specify the values of a type along with its machine representation. For example (using Ada),

```
type Status is (Low, Medium, High);  
for Status'Size use 8;  
for Status use (Low => 16#A4#, Medium => 16#A7#,  
                High => 16#A9#);
```

Where the values are explicit as Low, Medium, High (none other are possible), the number of bits for each value is eight, and the machine representation is in hexadecimal. These are absolute and will not vary from one implementation to another, nor from one machine platform to another. When the engineer defines these tolerances, the programmer is only concerned with the values (Low, Medium, High), not with the underlying constraints on those values.

Range tolerances are closely related to absolute tolerances and many of the constraint capabilities allowed for absolute tolerances, such as size in bits, machine representation, etc., can be applied to range tolerances. Range tolerances will include a

tolerance interval as well as tolerance limits. Each software element can include an upper and lower bound of acceptable values. This can provide compatibility management such as ensuring that a metric scale is not used where a non-metric scale should be used.

A conditional tolerance is one that is dependent on the state of an element within the code. Sometimes, the condition will be tested in conjunction with or according to the state of some other element(s). Programmers commonly do his kind of thing with in-line conditional testing (e.g., if statements, case statements, etc.). Where programming language support is available (e.g., SPARK, Eiffel, Ada, etc.), these tolerances are often stated as assertions, usually invariants. Another approach to assertions is the property-table, where the properties of each element/component are enumerated, bounded, and described, along with source code constructs for deviations from those properties.

A relative tolerance is a bit like a conditional tolerance, but it is not tested within the software. Instead, it is used as a design metric. That is, a relative tolerance does not raise an exception or violate some design feature of the program. An example of a relative tolerance is the well-known Big-O notation. For Big O, the performance tolerances are specified in terms such as $O(n)$, $O(\log n)$, etc., but the absolute tolerance is a function of the platform on which the software will execute. Relative tolerances are useful for making engineering decisions regarding which component (and its underlying algorithm) will provide the best throughput in a design. The engineer can evaluate competing designs using these relative performance metrics. Actual performance will require field testing to be sure there are no hidden platform-related issues at deployment (e.g., optimization, cache layer management, etc.).

Probabilistic and statistical tolerances are often seen in the domain of general engineering. For example, in physical systems, Morrison's technique of *variance synthesis* for refinement of fine-grained errors can be applied to a design to correct deviations from expected results [Mor01]. Software engineers are at a disadvantage in this respect. That disadvantage is not because statistical tolerances are impossible, but because software engineers rarely have the collected knowledge from previous designs

with similar components or behavioral properties to inform their new designs. Without quantified knowledge from prior projects, each new project is nearly *greenfield*. Any application of probability is likely to be guesswork.

Each of the above examples of design tolerances is further challenged by the concept of *sensitivity analysis*. In *sensitivity analysis*, a small change can be measured and evaluated for its potential effect on the larger design. Meyer [Mey00] suggests the term “continuity” for this in software. He shows one approach for how a software entity can be designed to prevent signal-to-noise ratio problems, thereby preventing some sensitivity problems. Even so, sensitivity analysis continues to be difficult enough that much more research will be required, and new software tools and methods developed, before it can be managed as deftly as it is in physical engineering systems.

4. Software Engineering Enabling Mechanisms

As noted earlier, constraints in standard engineering as imposed by the natural world. The software environment needs a different set of mechanisms for its constraints. The enabling mechanisms for designing to tolerances are already in place in many existing software tools and languages. These include:

1. Data abstraction (including object technology)
2. Assertions
3. Performance metrics (e.g., Big O notation)
4. Cohesion and Coupling metrics [Pre05]
5. Management of Scope and Visibility (where scope is a separate concern from visibility)
6. Defect Prediction and Prevention Strategies
7. Process Improvement Strategies (e.g., CMM, CMMI, etc.)
8. Concurrency and Resource Management Metrics and Strategies

From the above list, data abstraction is one of the most important. However, the techniques of data abstraction are not well understood by the larger community of software professionals. Data abstraction, as represented by the Abstract Data Type

[Gut77] is seldom used as effectively as it could be. This is, in part because many programming languages fall short of what is possible, and also because of the failure to educate software developers to think beyond the concept of an algorithmic reasoning.

Guttag, in his PhD dissertation on the topic of abstract data types (ADT), clarifies the potential for the design of well-formed software artifacts that are rigorously designed to behave according to a strict set of rules. Bertrand Meyer takes this even further with his Eiffel Language [Mey02]. And the SPARK Ada design represents one of the few tool-sets that bridging the gap between formal methods and software implementation [Bar03].

In an Abstract Data Type, all of the operations on an instance of that type will be completely defined as part of the type. It might be helpful, at this point, to enumerate the fundamental properties of a software *type*. In a non-formal sense, a *type* has:

1. an identifier (the name of the type)
2. a set of possible states (e.g., range of allowable values)
3. a set of operations, methods, operators
4. a set of rules governing interactions between instances of its own type and instances of other types.

An illustration of these rules might be helpful. The following ADT is in Ada.

```
generic
  type Item is private;
package Stack is
  procedure Push(Data : in Item);
  procedure Pop (Data : out Item);
  function Is_Full return Boolean;
  function Is_Empty return Boolean;
  function Item_Count return Natural;
  Overflow : exception;
  Underflow : exception;
end Stack;
```

Figure 9. ADT Stack Object Package Example

In this example, the only operations allowed on a Stack are those enumerated in the package specification. In this case, there can be only one instance of a stack for the data type, Item. Also, the generic formal parameter, type Item, can be any other data type. Many alternative designs are possible.

Here is an example where the Stack is an abstract data type.

```
generic
  type Item is private;
package Stacker is
  type Stack is limited private;
  procedure Push(Data : in Item;
    Onto : in out Stack);
  procedure Pop (Data : out Item;
    From : in out Stack);
  function Is_Full (S : Stack) return Boolean;
  function Is_Empty (S : Stack) return Boolean;
  function Item_Count(S : Stack) return Natural;
  Overflow : exception;
  Underflow : exception;
private
  -- expanded definition of type Stack
  -- can be expanded many different ways
end Stacker;
```

Figure 10. ADT Stack Type Example

In the above example, the ADT is designed so there are no operations (not even assignment or test for equality) provided. The set of behaviors is restricted to only those methods shown in the public part of the package.

We can take this kind of example ever further, designing our own numeric types. In the example that follows, we see that the type Int has been designed so there are no multiplication or division operations available. Also, the range of values has been restricted to values between a lower-bound of -500 and an upper-bound of 500. This is an example where the tolerances include both the range constraint and the restricted set of operations.

```

package My_Number is                                -- 1
  type Int is private;                             -- 2
  function "+" (L, R : Int) return Int;           -- 3
  function "-" (L, R : Int) return Int;           -- 4
  function "=" (L, R : Int) return Boolean;        -- 5
  function ">" (L, R : Int) return Boolean;        -- 6
  function ">=" (L, R : Int) b Boolean;          -- 7
  function "<" (L, R : Int) return Boolean;        -- 8
  function "<=" (L, R : Int) b Boolean;          -- 9
  Upper_Bound_Violation : exception;              -- 10
  Lower_Bound_Violation : exception;              -- 11
private                                           -- 12
  type Local_Integer is range -500..500;          -- 13
  type Int is record                               -- 14
    Value : Local_Integer := 0;                   -- 15
  end record;                                     -- 16
end My_Number;                                    -- 17

```

Figure 11. ADT Own_Integer Example

An even more generalized version of this package could be designed as a generic reusable component,

```

generic                                           -- 1
  type Item is range <>;                          -- 2
package My_Number is                             -- 3
  type Int is private;                             -- 4
  function "+" (L, R : Int) return Int;           -- 5
  function "-" (L, R : Int) return Int;           -- 6
  function "=" (L, R : Int) return Boolean;        -- 7
  function ">" (L, R : Int) return Boolean;        -- 8
  function ">=" (L, R : Int) b Boolean;          -- 7
  function "<" (L, R : Int) return Boolean;        -- 9
  function "<=" (L, R : Int) return Boolean;        -- 10
  Upper_Bound_Violation : exception;              -- 11
  Lower_Bound_Violation : exception;              -- 12
private                                           -- 13
  type Local_Integer is range Item'First..Item'Last; -- 14
  type Int is record                               -- 15
    Value : Local_Integer := 0;                   -- 16
  end record;                                     -- 17
end My_Number;                                    -- 18

```

Figure 12. Generic Integer ADT Example

where the upper and lower bounds are set using a constrained integer type defined elsewhere, but further constrained in the operations provided in the package where it will be used. This example can be extended to floating-point types as well as to many other

types. While most programmers do not use this approach in their own practice, it is a tool available to software engineers that can ensure greater dependability and reduce errors during software development. When used by an experienced software engineer, the ADT can be designed with even more fine-grained constraints. For example, in the numeric example, above, we might want to eliminate the option of doing any kind of greater-than test. We simply do not provide such a method for the ADT, and it becomes impossible for the programmer to do it. We can even provide ADT's for floating-point types, records, etc. that includes the exact range and precision required as well as the restricted set of methods for that type. For example,

```

package My_Float_Number is                                -- 1
  type Real is private;                                  -- 2
  function "+" (L, R : Real) return Real;               -- 3
  function "-" (L, R : Real) return Real;               -- 4
  function "*" (L, R : Real) return Real;               -- 5
  procedure Divide (Dividend : in Real;                 -- 6
    By_Divisor : in Real;                               -- 7
    Giving : out Real);                                 -- 8
  function "=" (L, R : Real) return Boolean;            -- 9
  function ">" (L, R : Real) return Boolean;             -- 10
  function ">=" (L, R : Real) return Boolean;           -- 11
  function "<" (L, R : Real) return Boolean;            -- 12
  function "<=" (L, R : Real) return Boolean;          -- 13
  Upper_Bound_Violation : exception;                   -- 14
  Lower_Bound_Violation : exception;                   -- 15
private                                                 -- 16
  type Local_Real is digits 7                           -- 17
    range -20_000.0 .. 50_000_000.0                    -- 18
  type Int is record                                     -- 19
    Value : Local_Real := 0;                             -- 20
  end record;                                           -- 21
end My_Float_Number;                                    -- 22

```

Figure 13. Floating-point ADT Example

In the above example, we have eliminated the division operator in favor of a division procedure where the components of division are explicitly stated. If one were doing a problem in matrix algebra or quaternions where the associative, distributive, and commutative laws are not applicable, this design approach would be practical.

5. Design Tolerances in Practice

Tolerances suggest the idea that components fit together at the level of “snugness” where the wobble-interaction between them is neither too tight nor too loose. The tolerances for a set of bearings on a camshaft must allow the cams to move while also reducing the amount of “wobble” over the shaft. An X-Ray therapy machine must give the correct radiation dose to a patient while not allowing too much radiation to be emitted. Further the operations allowed for that machine can be designed as an ADT where the behavioral options are restricted even as the range of values is constrained [Lev95, pp. 515-553].

Another example from the mechanical world is that of a bolt and its corresponding nut. A bolt with ten threads per centimeter will not work with a nut that has ten threads per inch. That bolt would also have other physical properties constraining torque, etc. The design of an ADT for a bolt can accommodate all of these factors so there is no chance for mismatch between a [software] bolt and its corresponding [software] nut.

In the physical world, where modest increments in measurement are possible, tolerances are easy to define and often easy to enforce. In software, where so much of the solution space is discrete it is often difficult to implement such subtle opportunities.

6. Fault-Tolerant Design

Design to tolerances is not quite the same as fault-tolerant design. However, the notion of fault-tolerance is embodied in the overall concept. In fault-tolerance we see the notion of “...systems whose failures can be tolerated” [IsZ03]. This idea is a little different from the “correctness by design” approach mentioned in the previous section.

One might design a system to be fault-tolerant using design metrics. Such designs will include fault prevention, fault removal, fault forecasting, and finally, fault tolerance. The last of these, fault-tolerance is usually characterized by run-time techniques involving error-detection and recovery or error-detection and compensation. That is, we can detect an error, return the system to an error-free state, and continue processing; or

we can include code that allows for errors, but continues to produce the correct outcome anyway. An example of the latter is ignoring an attempt to divide by zero in an algorithm over a long series of numbers when such an error has no effect on the final result of a calculation. The other three items,

1. forecasting,
2. removal,
3. prevention.

are in the domain of design tolerance metrics.

7. Tolerances in Software

Designing to tolerances implies that we can achieve a correct program by design. This further implies that we have tools to evaluate the tolerances in the design before the software is released for execution. Since software does not have physical properties that can be measured in the same way one measures something such as the threshold temperature of a liquid, or the torsion of a steel bar under stress, we need some other way to specify those tolerances.

In addition, it is not appropriate to wait for testing or failure during execution to determine that a design violates its own tolerances. Rather, deviations should be observable or predictable even as the design is effected. While stress-testing and other approaches at exercising the final software product are valuable, the *ideal* tolerances are those where out-of-tolerance can be noted using methods and tools that allow the evaluation of that design even before it is actually executed.

Even when engineering depends on the constraints of the natural world, the ideal is not always a reality. Stress testing a physical artifact to determine that a design actually behaves as predicted continues to be an important part of every kind of engineering. Consequently, we cannot expect to achieve an ideal in software that is seldom possible in traditional engineering.

8. Modeling for Tolerances

Tolerances must be part of any engineering design. According to Fowler,

A model is a representation of a process. ... The purpose of a model is to formulate a description, in mathematical terms, and the analysis of the resulting model leads to a result that can be tested against observations [Fow98].

D. ADDITIONAL SOFTWARE TOLERANCES

1. Historical Perspective

The previous section mentioned C.A.R. Hoare. In the article quoted, Hoare says, “...the programmer should make a number of assertions which can be checked individually, and from which the correctness of the whole program easily follows.” In the preceding sentence, Hoare was speaking as a programmer. However, what he was describing was a practice that would eventually become one of the more important practices in software engineering – a practice still ignored by most programmers, and still unavailable in most programming language designs. More recently, the Laser Summer School on Software Engineering in 2004, sponsored by ETH in Zurich, Switzerland, included papers relative to methods for modeling discrete systems. One of the significant contributions at that conference was from J.R. Abrial [Abr04]. In his paper, Abrial presented a formal approach to discrete systems modeling (he calls them transitional systems). While his is not the only research in this area, it is exemplary of much of the research that is being done to develop a approach to managing the complexity of tolerances in software systems.

2. Assertions

Hoare was the inventor of an assertion construct sometimes called the Hoare Triplet. Briefly, it is stated as:

pre-condition {process} post-condition.

This is a simple idea with major implications. Before a program statement is permitted to successfully enter a given process, it must satisfy the pre-condition for that process. When the process completes, the post-condition must be satisfied. This was one of the most important ideas leading to today's concept of "correctness by design." In the Hoare Triple(t), the assertions could only be checked during run-time, and this is how it is implemented in many of the languages that use it. For example, Eiffel, extended with some additional kinds of assertions, is one of the more successful languages using the Hoare Triplet. However, even in Eiffel, the assertions are trapped at run-time, not as a consequence of some kind of compile-time theorem-proving [Mey92]. More recently, GNAT Ada has adopted this model. Also, there is third-party support for the Hoare Triple(t) in Java [MiM02].

3. Constraints

Wirth states, in his book describing Pascal, that a program is Data + Algorithms [Wir76]. In modern software practice, with its emphasis on abstract data types and software objects, this is a somewhat out-of-date proposition. In modern software, constraints are an important part of the design of both modules and the methods within those modules. Also, constraints have now become a part of the UML in the form of an Object Constraint Language (OCL) [Wak03]. The need for constraints on both methods and data is a current trend that bodes well for the progress toward engineering software. The more carefully constraints are designed by the software engineer, the less opportunity there is for a programmer to violate those constraints. Of course, as with Eiffel and Ada, those constraints must be an inherent part of the source code contract for them to be useful. Comment assertions are not very helpful in software engineering.

Weinberg [op cit., p. 182] suggests that the more strict the constraints, the fewer options one has in for solving a problem. In the movie adaptation of a book by Raphael Sabatini [Sab52], a young fencing student is being instructed on how to hold his foil. The instructor says, "If you hold it too tight you will crush it. If you don't hold it tight

enough, it will fly away.” Constraints are similar. The engineer must be careful to design constraints so they do not prevent the solution from being realized; and they must not inhibit the required efficiency of that realization.

In the engineering definition from Shaw and Garlan [ShG96] in Chapter III, we saw the phrase, “resolution of conflicting constraints”. It is important to keep in mind that engineering is largely about the reconciliation of conflicting constraints and conflicting forces. The very fact of conflicting constraints is an important *raison d’être* for any kind of engineering. Ensuring absence of failure in reconciling these conflicts is a fundamental engineering problem.

a. Data Constraints

Programming language data constraints (as constrained data types) were included in languages such as Ada in the late 1970’s. Those constraints were more like invariants. Prior to the notion of a constrained type, languages in the C family (C, C++, etc), as well as older languages such as COBOL and Fortran, relied on types that mapped directly to the underlying hardware platform on which programs were intended to execute. Therefore, an *int* on a thirty-two bit machine would have a default range of -2^{31} through $2^{31} - 1$ even when the number to be represented was very tiny. A C programmer can insert an if statement to protect values of the type, but that once again relies on the programmer. Relying solely on the programmer creates a potential point of failure in the overall design. Counter-examples can be seen in Ada:

```
type Some_Number is range -200..200;  
for Number'Size use 8;
```

where the type with the name, *Some_Number* is constrained to never be less than -200 nor greater than 200. Further, a representation clause forces this value to be represented in eight bits. This definition will still raise a run-time exception, but no if statements are required in a program to ensure it is within range. Ada does provide a simple if statement to determine whether the value is valid before it is used. The statement, *if X'Valid* (where X is the instance of some scalar type) will produce a true/false result and prevent X from

being used if it is not valid within the constraints given for it in a specification. For this to work properly, the software engineer must have prescribed the constraints for X's data type well in advance of its being used by the programmer.

Using the model just described, any numeric type can be specified with a range bounds, a precision (for floating-point), a storage-size, and other properties that strictly constrain instances of the type. Ada's model of type-safety is more rigorous than that of most other popular languages, and this type model, along with other properties of the language make it useful for satisfying our quest for design metrics and the ability to design to tolerances.

b. Behavioral Constraints

Where invariants are primarily associated with data, other kinds of assertions can be applied to behavior, or as a consequence of behavior. Often these are still expressed in terms of data, but they control or result from behavior.

c. Contract Constraints

A contract constraint is most often associated with object technology, or sometimes, an Abstract Data Type. In this case, the design tolerances include the set of permitted behaviors, set of possible states (e.g., range of values), and set of rules relative to instances of this ADT and instances of other types. Further, a contract is an agreement between two or more parties. Therefore, a contract constraint requires that the interacting software entities concur with each other during that interaction. For example, if the contract specifies that an instance of one data type (e.g., and ADT) is not allowed to interact with an instance of some other data type, the environment (programming language) must enforce that contract. In this case, there is a constraint that proscribes such interaction.

4. Language and Tool Support

There has been no paucity of languages and tools for solving the problems associated with software. The most rigorous of these languages include those mentioned

(e.g., Ada and Eiffel). Other, new languages are emerging with the same level of rigor for the support of an engineering model for software engineering. More and more software professionals are recognizing the need for more rigorous tools and languages. Even as this is being written, new languages are probably on the threshold of being announced. Also, mentioned earlier is SPARK [Bar03], probably the most ambitious attempt in actual use for rigorous engineering of safety-critical software.

E. UNLIKELY SOFTWARE DESIGN TOLERANCES

We first address the issue of what is not currently possible in designing software to tolerances. The binary model used in most computer designs precludes the designer from doing the kind of fine-grained tolerance computations available to other kinds of engineers. When a boolean proposition fails in an algorithm, the entire program can fail. A single wayward bit can destroy months of computation, or cause an entire computer system to crash. The fault-tolerance methods, including exception-handling, are intended to accommodate such anomalies during the execution of a program. Including exception-handling is common in most modern software systems even though it was denounced by none other than C. A. R. Hoare in his Turing Award Lecture [Hoa81] where his principle criticism of Ada was “... features and notational conventions, many of the unnecessary and some of the, like exception-handling, dangerous.” In current programming language design, exception-handling is a standard feature that everyone expects. For a language such as Java, one cannot write simple programs without including a lot of try-catch blocks. While this approach may satisfy some of the fault-tolerance requirements, it does not represent the “correctness by design” approach we would expect from a model of designing to tolerances using design metrics.

Computer programming, absent a discipline of design metrics, rarely exhibits any attention to designing to tolerances. Instead, if..endif statements are (often as an afterthought) sprinkled throughout a program in an effort to redirect an algorithm when it approaches some limit or set of conditions that might be problematic. Yet every

programmer knows that conditional statements in any program are the weak link. A decision made from a conditional statement can often contain non-obvious logic errors that do not show up until after many iterations of program execution.

F. CHAPTER SUMMARY

More could be written on the topic of designing software to tolerances. In addition, more research is required on this topic. At present, there is more to be done than has been done. Eventually, new languages, methods, tools, and notations will come into existence to support this concept more fully.

Even though there is much to do in the development of design to tolerances, it is essential that it be done. Until we have a more substantive set of tools and methods for designing software to tolerances, along with a more effective model for constraints and controls, software will continue to be deprecated as an engineering wannabee. Those who deprecate the progress that has been achieved are simply not well-enough informed about the current opportunities for doing real engineering for software. Those in the software community to continue to ignore those opportunities will eventually find themselves obsolete and out-of-date, still hacking away at their own programs, but not able to cope with the increasing demands that society is placing on software – engineered software.

VII. SUMMARY AND FUTURE WORK

A. DISSERTATION SUMMARY

The central contribution of this dissertation is to present software engineering in a completely different way than it has been presented in the past. This has required us to examine the engineering context as a *reference*. Our conclusions, while controversial, are not so easily dispelled simply because of the soft nature of the product, or the difficulty of engineering a basically discrete environment. We note that it is the very difficulty of engineering software that is its imperative.

The emerging discipline of software engineering must be examined in an engineering context that conforms to the rigor demanded of other engineering practices. That context, while important, is not alone sufficient given the substantial differences between traditional elements of classical engineering and those of software practice. Those differences do not exempt software engineering from conformance to the essential goals, principles, and constraints that characterize more widely accepted engineering practices.

In this work, we have examined the fundamental character of engineering and how it is generally regarded by the community of practicing engineers. On examining the viewpoint of both the practicing engineers, and the academic approach required for engineering education, we concluded that the viewpoint is too narrow. This required that we expand the definition of engineering to encompass a larger number of disciplines, but also required that that expansion not dilute the essential nature of engineering. The goals, principles, and practices remain the same even as the new definition embraces a wider range of practices.

After describing a more comprehensive concept of engineering, we examine software practice in that context. It became clear that much of what passes for engineering in the world of software practice falls short of engineering. This is particularly true of the actual process of computer programming. However, that

examination also revealed that a lot of software engineering practice is not concerned with computer programming. We noted that the creation of software involves three major activities: architecture, engineering, construction. Each of those activities has a set of tasks, some of which overlap or require a kind of incremental repetition. For example, there is a requirements process at each level.

Architectural design provides the construction model for a software product. Engineering design introduces the constraints and controls needed to ensure the construction process is carried out correctly. Construction is required to conform to those controls and constraints as well as remain consistent with the architecture.

One of the key contributions in this dissertation is to identify control as a key artifact of the engineering activity. In the past, control was largely the province of the computer programmer. Once we promote this responsibility to the engineering level, the programmer is constrained by the controls as well as the problem statement. We also identified other responsibilities for the software engineer that are intended to improve software performance and reduce both predictable and unpredictable defects.

In much of software practice, the responsibilities we describe are not yet in place. Even so, they are increasingly important in software for such safety-critical environments as commercial avionics (e.g., Boeing 777/787) and military weapon systems (e.g., cruise missiles). As software engineering practice continues to mature, and as software engineering education improves, we expect that a recognition of these responsibilities will become more widespread. In my personal visits to software developers in Japan and elsewhere, I am encouraged to see an increasing awareness of the need for this kind of responsibility in software engineering.

B. FUTURE WORK

The present work is a starting point for a larger contribution on software engineering. It establishes an engineering context, but not a plan for how that context can be advanced to a more comprehensive model. That kind of plan is the logical next step in my research.

An essential idea in this work has been the emergence of new kinds of engineering in the Twenty-first Century. As noted earlier, the definition of engineering currently in-use originated in the middle of the Twentieth-century, and it has not been updated since 1958. Future work will include research into how those emerging engineering disciplines can become more mature as authentic models of engineering. In this respect, I plan to refine the work on the maturity model for emerging engineering disciplines described in Chapter IV. The engineering maturity model will require input and ideas from the larger engineering community, and I will need to persuade that community that such an effort is necessary to legitimize those emerging engineering disciplines.

I have already begun work on several threads that have importance for that future work. My draft of a book on software evolution has the potential for becoming the medium in which I can plant the ideas from this dissertation. I have also written a book on requirements development that is used at NPS by myself and other instructors, and some of the ideas from this dissertation will find their way into the next edition of that book. Also, I will continue to contribute papers to computer and software engineering publications as my research develops additional new ideas.

Most important, the topic of this dissertation is a culmination of many years of thinking and studying software, software development, and software management. My future work will be to continue that thread of study.

My intention is to continue work on a larger book that derives from the research in this dissertation along with future research into the relationship of software practice to classical engineering. My hope is that such a book will be a larger contribution than this

dissertation, when completed. There will be new chapters in that book that will break new ground. In particular, the topic of software risk management, mentioned several times in this work, will require some new approaches. I have already published papers on this topic and plan several more based on the engineering principles already discussed.

There is also a need for more comprehensive work in the domain of software engineering education. Several initiatives are in place under the sponsorship of the Association for Computing Machinery (ACM) and the Institute for Electrical and Electronics Engineers (IEEE). One of these is Graduate Software Engineering Reference Curriculum (GSwERC). We intend to use the work from this dissertation to assist in the development of a comprehensive engineering model for the GSwERC initiative.

BIBLIOGRAPHY

- [Abr04] Abrial, J.R., "Discrete Systems Models," Laser Summer School on Software Engineering, Elba, Italy, Sponsored by ETH, Zurich, Switzerland, 2004.
- [Agr81] Agresti, W. W., "Software Engineering as Industrial Engineering," *Software Engineering Notes*, vol. 6, no. 5, 1981, pp. 11-12.
- [Ale64] Alexander, Christopher, *Notes on the Synthesis of Form*, Cambridge, MA: Harvard University Press, 1964.
- [Ayu03] Ayyub, Bilal M., *Risk Analysis in Engineering and Economics*, Boca-Raton, FLA: Chapman & Hall/CRC, Chapters 1 and 2, 2003, pp. 127-136.
- [Bar03] Barnes, John G. P., *High Integrity Software*, Menlo Park: CA: Addison-Wesley, 2003.
- [Bau73] Bauer, Fritz, "Software Engineering," *Software Engineering*, also edited by Fritz Bauer, Springer, Heidelberg, Germany, 1973.
- [BCK07] Bass, Len, Paul Clements, and Rick Kazman, *Software Architecture in Practice*, 2nd ed., Addison-Wesley/Pearson Education, Boston, MA, 2007.
- [BL82] Beakley, George and H.W. Leach, *Engineering: An Introduction to a Creative Profession*, 4th ed., Macmillan Publishing Co., 1982.
- [BW77] Becker, Shirley A. and James A. Whittaker, *Cleanroom Software Engineering Practices*, London: Idea Group Publishing, 1977.
- [Bec00] Beck, Kent, *Extreme Programming Explained*, Addison-Wesley, Boston, MA, 2000.
- [Ber92] Berard, Edward V., *Essays on Object-Oriented Software Engineering*, Upper Saddle River, NJ: Prentice-Hall, 1992.
- [BC85] Bergeretti, J. F. and B.A. Carre, "Information-Flow and Data-Flow Analysis of While-Programs," *ACM Transactions on Programming Languages and Systems*, vol. 7, January 1985, pp. 37-61.
- [BT03] Boehm, Barry and Richard Turner, *Balancing Discipline and Agility*, Reading, MA: Addison-Wesley, 2003.

- [Boe84] Boehm, Barry, "Verifying and Validating Software Requirements and Design Specifications," *IEEE Software*, January 1984, pp. 75-88.
- [Boh66] Bohm, Corrado and Jacopini, Giuseppe, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM*, 9 (5), 1966, pp. 366-371.
- [Boo94] Booch, Grady, Bryan, Doug, Peterson, Charles, *Software Engineering with Ada*, Third Edition, Benjamin Cummings, Menlo Park, CA, 1994.
- [BR99] Briand, Loic P. and Daniel M. Roy, *Meeting Deadlines in Hard Real-time Systems: The Rate Monotonic Approach*, New York: IEEE Computer Society, 1999.
- [Bro95] Brooks, Frederick P., *The Mythical Man Month*, Reading, MA: Addison-Wesley, Anniversary ed., 1995.
- [Bry00] Bryant, A., "Metaphor, Myth, and Mimicry: The Bases of Software Engineering," *Annals of Software Eng.*, vol. 10, 2000, pp. 273-292.
- [Burns01] Burns, Alan and Wellings, Andy, *Real-time Systems and Programming Languages*, Third Edition, Addison-Wesley, Harlow, England, 2001.
- [Coc02] Cockburn, Alistair, *Agile Software Development*, Addison-Wesley, Boston, MA, 2002.
- [Cer98] Cerruzi, Paul E., *A History of Modern Computing*, MIT Press, Cambridge, MA, 1998.
- [CE00] Czarnecki, Krzysztof and Ulrich W. Eisenecker, *Generative Programming*, New York: Addison-Wesley, 2000.
- [Den68] Denning, Peter J., "The working set model for program behavior," *Communications of the ACM*, vol. 11 no. 5, May 1968, pp. 323-333.
- [Die83] Dieter, G, *Engineering Design*, New York: McGraw-Hill, 1983.
- [DP00] Dym, Clive L and Patrick Little, *Engineering Design: a Project-based Introduction*, New York: John Wiley & Sons, Inc., 2000, p. 8.

- [ECPD58] Engineer's Council for Professional Development, 26th Annual Report for the Year Ending September 1958 (quoted in Thatcher, Charles M., *Fundamentals of Chemical Engineering*, Charles M. Merrill Books, Columbus, OH, 1962.
- [ER03] Endres, Albert and Dieter Rombach, *A Handbook of Software and Systems Engineering: empirical observations, laws, and theories*, New York: Pearson Addison Wesley, 2003.
- [ET02] Erdogmus, Hakan and Oryal Tanir, Eds., *Advances in Software Engineering*, New York: Springer-Verlag, 2002.
- [Fair85] Fairley, Richard, *Software Engineering Concepts*, McGraw-Hill, 1985.
- [Flo87] Florman, Samuel C., *The Civilized Engineer*, St. Martins Griffin, New York, NY, 1987.
- [Flo96] Florman, Samuel C., *The Introspective Engineer*, St. Martins Griffin, New York, NY, 1996.
- [Fow98] Fowler, A.C., *Mathematical Modeling in the Applied Sciences*, Cambridge University Press, 1998.
- [Fre98] Frezza, Stephen T., "An Undergraduate Software Engineering Program within Electrical Engineering," presented at Frontiers in Education Conference, Tempe, AZ, November 4-7, 1998, <http://fie.engrng.pitt.edu/fie98/papers/1386.pdf> (accessed February 2008).
- [FW96] Fuggetta, Alfonso and Alexander Wolf, *Software Process*, New York: John Wiley & Sons, 1996.
- [GE06] Galorath, Daniel D. and Michael W. Evans, *Software Sizing, Estimation, and Risk Management*, Boca Raton, FL: Auerbach Publications, 2006, pp. 347-395.
- [GJ96] Garg, Pankaj and Mehdi Jazayeri, "Process-Centered Software Engineering Environments: A Grand Tour," in *Software Process* (Alfonso Fuggetta and Alexander Wolf, Eds.). New York: John Wiley & Sons, 1996, pp. 25-51.
- [GaW89] Gause, Donald C. and Weinberg, Gerald M. *Exploring Requirements: Quality before Design*, New York: Dorset House Publishing, 1989, pp. 94-97.

- [Ghezzi2002] Ghezzi, Carlo, Jazayeri, Mehdi, Mandrioli, Dino, *Fundamentals of Software Engineering*, Second Edition, Prentice-Hall, Upper Saddle River, NJ, 2002.
- [Gla03] Glass, Robert L., *Facts and Fallacies of Software Engineering*, Addison-Wesley/Pearson Education, Boston, MA, 2003.
- [Gla99] Glass, Robert L., *Computing Calamities*, Prentice-Hall, Upper Saddle River, NJ, 1999.
- [Gor06] Gorton, Ian, *Essential Software Architecture*, New York: Springer, 2006.
- [GVR02] Glass, R. L., I. Vessey, V. Ramesh, "Research in Software Engineering: an Analysis of the Literature," *Information and Software Technology*, vol. 44, no. 8, 2002, pp. 491-506.
- [Gut77] Guttag, John, "Abstract Data Types and the Development of Data Structures," *Communications of the ACM*, vol. 20, no. 6, June 1977.
- [Hal97] Hall, Elaine, *Managing Risk: Methods for Software and Systems Development*, 1997, Addison-Wesley, New York, NY, 1997.
- [Hoa81] Hoare, C.A.R., "The Emperor's Old Clothes," *Communications of the ACM*, February 1981, pp. 75-83.
- [Hod92] Hodson, William K., Ed., *Maynard's Industrial Engineering Handbook*, 4th ed., New York: McGraw-Hill, Inc., 1992.
- [Hol08] Holtzapple, Mark T and W. Dan Reece, *Concepts in Engineering*, 2nd ed., New York: McGraw-Hill, 2008.
- [Hum89] Humphrey, Watts, S., *Managing the Software Process*, Reading, MA: Addison-Wesley, 1989.
- [Hum95] Humphrey, Watts S., *A Discipline for Software Engineering*, Reading, MA: Addison-Wesley, 1995.
- [Hum 06] Humphrey, Watts S., *TSP – Coaching Development Teams*, Reading, MA: Addison-Wesley, 2006.
- [IEE90] *IEEE Standard Computer Dictionary: a Compilation of IEEE Computer Glossaries: 610*, New York: IEEE Computer Society, 1990.

- [IsZ03] Issary, Valerie and Apostolos Zarras, “Software Architectures and Dependability” in *Formal Methods for Software Architecture: Third International School on Formal Methods for the Design of Computer, Communication and Software Systems (Lecture Notes in Computer Science)*, Marco Benardo and Paola, Inverardi, Eds., New York: Springer-Verlag, 2003, p. 259.
- [Jam99] Jammer, Max, *Concepts of Force*, Mineola, NY: Dover Publications, 1957 and 1999.
- [JeT79] Jensen, Randall W. and Charles C. Tonies, *Software Engineering*, Upper Saddle River, N.J.: Prentice-Hall, 1979.
- [Jon94] Jones, Capers, *Assessment and Control of Software Risks*, Upper Saddle River, N.J.: Prentice-Hall, 1994, pp. 202-208.
- [Jon96] Jones, Capers, *Patterns of Software Systems Failure and Success*, Boston, MA: International Thompson Computer Press, 1995.
- [KWDK90] Kirby, Richard Shelton; Sidney Withington, Arthur B. Darling, and Frederick G. Kilgour, *Engineering in History*, Dover Publications, Mineola, NY, 1990, p. 378.
- [Kol85] Kolence, Kenneth, *An Introduction to Software Physics*, New York: McGraw-Hill, 1985, pp. 11-14.
- [Kru04] Kruchten, Phillippe, “Putting the ‘Engineering’ into Software Engineering,” *Australian Software Eng. Conf. (ASWEC 2004)*, P. Strooper, ed., IEEE CS Press, 2004, pp. 2–8.
- [Lew00] Lewerentz, C and Rust, H “Are Software Engineers True Engineers?” *Ann. Software Eng.*, vol. 10, 2000, pp. 311–328.
- [Lev95] Levenson, Nancy G., *Safeware: Safety System Safety and Computers*, Addison-Wesley, Reading MA, 1995.
- [Mah02] Mahoney, Michael S., “Software as Science – Science as Software,” in *History of Computing: Software Issues* (U. Hahsagen, R. Keil-Slawik, and A. Norberg, Eds.), Berlin: Springer-Verlag, 2002.
- [Mah04] [Mahoney, Michael S., “Finding a History of Software Engineering,” in *IEEE Annals of the History of Computing, No. 26 (January-March)*, 2004, pp. 8-19.

- [Mai97] Maibaum, T. S. E., "What We Teach Software Engineers in the University: Do We Take Engineering Seriously?" *Proceedings of 6th European Conference on the Foundations of Software Engineering*, Zurich, Switzerland, 1997, pp. 22-25.
- [Mar05] Marasco, Joe, *The Software Development Edge*, Addison-Wesley/Pearson Education, Upper Saddle River, NJ, 2005.
- [Mcc04] McConnell, Steve, *Professional Software Development*, Addison-Wesley (Pearson Education), Boston, MA, 2004.
- [Mcb02] McBreen, Pete, *Software Craftsmanship: The New Imperative*, Addison-Wesley, Boston, MA, 2002.
- [Mey00] Meyer, Bertrand, *Object-oriented Software Construction*, 2nd ed., Upper Saddle River, NJ: Prentice-Hall, 2000.
- [Mey92] Meyer, Bertrand, *Eiffel the Language*, Prentice-Hall, Upper Saddle River, NJ, 1992.
- [MiM02] Mitchell, Richard and McKim, Jim, *Design by Contract by Example*, Boston: Addison-Wesley, 2002, pp. 189-213.
- [Mog80] Mogensen, Alan, *Common Sense Applied to Time and Motion Study*, 7th ed., New York: John Wiley & Sons, 1980. (First written in 1932).
- [Moo06] Moore, James W., *The Roadmap to Software Engineering; A Standards Based Guide*, Wiley-IEEE Computer Society, Hoboken, NJ, 2006.
- [Mor01] Morrison, S.J., "Quality Engineering Design," *Manufacturing Engineer*, vol. 80, issue 3, June 2001, pp. 110-112.
- [NaR60] Naur, P, and B. Randell, eds., *Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee*, NATO, 1969.
- [Neu95] Neumann, Peter G., *Computer-Related Risks*, ACM Press, Reading, MA: Addison-Wesley, 1995.
- [Ogata97] Ogata, Katsuhiko, *Modern Control Engineering*, Third Edition, Prentice-Hall, Upper Saddle River, NJ, 1997.
- [Pan 04] Pandian, C. Ravindranath, *Software Metrics: A Guide to Planning, Analysis, and Application*, Boca Raton, FL: CRC Press, 2004.

- [Par99] Parnas, David Lorge, "Software Engineering Programmes are not Computer Science Programmes," *Annals of Software Engineering*, vol. 6, no. 1-4, April 1999, pp. 39-59.
- [Pau02] Paulish, Daniel J., *Architecture-Centric Software Project Management*, Addison-Wesley, Reading, MA, 2002.
- [PeP00] Peters, James and Pedrycz, Witold, *Software Engineering : An Engineering Approach*, Wiley, 2000.
- [PeSV96] Perry, Dewayne E; N. Staudenmeyer, L. G. Votta, "Understanding and Improving Time Usage in Software" *Software Process*, Alfonso Fuggetta and Alexander Wolf, eds., New York: John Wiley & Sons, 1996, pp. 112-135.
- [Pet96] Petroski, Henry, *Invention by Design; How Engineers Get from Thought to Thing*, Cambridge, MA: Harvard University Press, 1996.
- [Pet85] Petroski, Henry, *To Engineer is Human*, New York: St. Martin's Press, 1985.
- [Pfl9191] Pfleeger, Shari Lawrence, *Software Engineering: the Production of Quality Software*, 2nd ed., New York: Macmillan, 1991.
- [Pre05] Pressman, Roger, *Software Engineering: A Practitioner's Approach*, Sixth Edition, McGraw-Hill, New York, NY, 2005.
- [Pro98] Prowell, Stacy J., C.J. Trammell, R.C. Linger, and J.H. Poore, *Cleanroom Software Engineering: Technology and Process*, Reading, MA: Addison-Wesley (for S.E.I.), 1998.
- [RaW63] Rapport, Samuel and Wright, Helen, Eds., *Engineering*, New York: New York University Press, 1963.
- [Rie05] Riehle, Richard D., "Engineering on the Surprise Continuum," *Software Engineering Notes (SEN)*, vol. 30, Number 5, September 2005.
- [RieS07] Riehle, Richard D., "Failure-driven Software Safety," *Software Engineering Notes (SEN)*, vol. 32, no. 5, September 2007.
- [Rie94] Riehle, Richard D., "Ada in Space," *Embedded Systems Programming Magazine*, November 1994.

- [RieJ07] Riehle, Richard D. "Designing Software Components to Tolerances," *Software Engineering Notes (SEN)*, vol. 32, no. 4, July 2007.
- [Rie98] Riehle, Richard D., "The Software Circuit-breaker," *Journal of Object-oriented Programming (JOOP)*, November 1998, pp. 69-73.
- [Rie96] Riehle, Richard D., "Managing Run-time Faults," *Journal of Object-oriented Programming (JOOP)*, September 1996, pp. 73-77.
- [Rie 06] Riehle, Richard D., "Linguistic Continuity in Software Engineering," *Software Engineering Notes (SEN)*, vol. 31, no. 1, January 2006.
- [RieN07] Riehle, Richard D., "Institutional Memory and Risk Management," *Software Engineering Notes (SEN)*, vol. 32, no. 6, November 2007.
- [RGI80] Ross, D.T., J. Goodenough, and C. Irvine, "Software Engineering: Process, Principles, and Goals," in *Tutorial on Software Design Techniques*, P. Freeman and A. Wasserman, Eds., Long Beach, CA: IEEE Computer Society Press, 1980.
- [Roy98] Royce, Walker, *Software Project Management: A Unified Framework*, Reading, MA: Addison-Wesley, 1998.
- [Roy70] Royce, Winston, "Managing the Development of Large Software Systems: Concepts and Techniques," *Proceedings of IEEE WESCON*, August 1970, pp. 1-9.
- [RuJB05] Rumbaugh, James, I. Jacobson, G. Booch, *The Unified Modeling Language Reference Manual*, 2nd ed., Reading, MA: Addison-Wesley, 2005.
- [Sab52] Sabatini, Raphael, *Scaramouche*, screenplay by Ronald Millar, Loew's, 1952.
- [ScM92] Schlaer, Sally and Stephen J. Mellor, *Object Lifecycles: Modeling the World in States*, Englewood Cliffs: Prentice-Hall, 1992, pp. 4-6.
- [Sei00] Seife, Charles, *Zero: The Biography of a Dangerous Idea*, New York: Viking, 2000.

- [ShG96] Shaw, Mary and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Upper-Saddle River, NJ: Prentice-Hall, 1996.
- [Sha90] Shaw, Mary, "Prospects for a Discipline of Software," *IEEE Software*, vol. 7, no. 6, November 1990, pp. 15-24.
- [Shaw01] Shaw, Alan C., *Real-time Systems and Software*, John Wiley & Sons, New York, NY, 2001.
- [Shepard] Shepard, Aland, quoted in <http://www.brainyquote.com/quotes/quotes/a/alanshepar179873.html> (accessed July 2008).
- [Som05] [SOM] Sommerville, Ian, *Software Engineering*, 7th ed., Pearson (Addison-Wesley), 2005.
- [StP03] Steinberg, Daniel H. and Daniel W. Palmer, *Extreme Software Engineering: a Hands-On Approach*, Upper Saddle River: Prentice-Hall, 2003.
- [Ste05] Stern, David, Chapter 16, "Newton and His Laws: in From Stargazers to Starships [a web book], March 2005, <http://www-istp.gsfc.nasa.gov/stargaze/Smap.htm> (accessed July 2008).
- [Tay11] Taylor, Frederick Winslow, *The Principles of Scientific Management*, New York: Harper and brothers, 1911. (*Out-of-print but available electronically for several WWW sources and at university libraries.*)
- [Tek07] <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.27.8684> University of Pennsylvania, (accessed July 2008).
- [Thu89] Thuesen, G. J., Fabrycky, W.J., *Engineering Economy*, Prentice-Hall, Englewood Cliffs, NJ, 1989.
- [Vin90] Vincenti, Walter G., *What Engineers Know and How They Know It*, Baltimore: Johns Hopkins University Press, 1990, p. 6.
- [Vol04] Voland, Gerard, *Engineering by Design*, 2nd ed., Upper Saddle River: Pearson/Prentice-Hall, 2004.
- [WaK03] Warmer, Jos and A. Keppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd ed., Reading, MA: Addison-Wesley, 2003.

- [Wang02] Wang, Wei-lung, "Beware the Engineering Metaphor," *Communications of the ACM*, vol. 45, no. 5, May 2002, p. 27.
- [Wei89] Weinberg, Gerald M. and Gause, Donald C., *Exploring Requirements*, Dorset House Publishing, New York, NY, 1989.
- [Wei98] Weinberg, Gerald M. and Gause, Donald C., *The Psychology of Computer Programming*, (originally 1971), Dorset House Publishing, New York, NY, 1998.
- [WhB97] Whitaker, James A. and Shirley A. Becker, *Cleanroom Software Engineering Practices*, Harrisburg, PA: Idea Publishing Group, 1997.
- [Wie96] Wiegers, Karl E., *Creating a Software Engineering Culture*, Dorset House, New York, NY, 1996.
- [Wir76] Wirth, Niklaus, *Algorithms + Data Structures = Programs*, Englewood Cliffs, NJ: Prentice-Hall, 1976.
- [Wri89] Wright, Paul, H., *Introduction to Engineering*, New York: John Wiley & Sons, 1989, pp. 47 and 49.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California
3. Dr. Peter J. Denning
Computer Science Department
Naval Postgraduate School
Monterey, California
4. Dr. J. Bret Michael
Computer Science Department
Naval Postgraduate School
Monterey, California
5. Dr. Mantak Shing
Computer Science Department
Naval Postgraduate School
Monterey, California
6. Dr. Mikhail Auguston
Computer Science Department
Naval Postgraduate School
Monterey, California
7. Dr. Dan Boger
Naval Postgraduate School
Monterey, California
8. Professor Charles Calvano
Systems Engineering Department
Naval Postgraduate School
Monterey, California
9. Dr. Qioayun Li
Skysurfer Communications, Inc.
San Jose, California