Integrated Software & Systems Engineering Curriculum (iSSEc) Project

Graduate Software Engineering 2009(GSwE2009)

Curriculum Guidelines for Graduate Degree Programs in Software Engineering

Version 1.0

September 30, 2009

This page intentionally left blank.

Table of Contents

List of Figures

List of Tables

Preface

Software engineering (SwE) is "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software."[1] SwE principles and practices are essential for the development of large, complex, or trustworthy systems. In 1989 the Software Engineering Institute (SEI) published a set of recommendations for creating curricula for master's programs in SwE.[2] Those recommendations were highly regarded and used by many universities in shaping their graduate SwE programs.

Since 1989 the way software is developed has changed dramatically. Software's scale, complexity, and criticality have mushroomed, yet no significant effort has been made to revisit and update the original SEI recommendations. (An updated report was published in 1991, but the curriculum recommendations were virtually unchanged.) In 2007, a coalition from academia, industry, government, and professional societies formed the Integrated Software and Systems Engineering Curriculum (iSSEc) project to create a reference curriculum[3] that reflects current development practices and the greater role of software in today's systems. The U.S. Department of Defense's (DoD) Office of the Secretary of Defense (OSD) is the principal iSSEc sponsor, motivated by the many challenges in acquiring, operating, and maintaining defense systems whose functionality and performance depend heavily on tractable and cost-effective software.

Graduate Software Engineering 2009 (GSwE2009): Curriculum Guidelines for Graduate Degree Programs in Software Engineering is the first product of the iSSEc project. Until August 2009 it was called the Graduate Software Engineering Reference Curriculum (GSwERC). GSwE2009 primarily addresses the education of students for a professional master's degree in SwE—that is, a degree intended for someone who is primarily interested in pursuing a career in the practice of SwE and who is not necessarily interested in pursuing a doctorate in SwE or a related field. Typically, such students are already (a) professional software engineers employed by industry or government and who lack a formal graduate education in SwE, or (b) professionals in another field who are making a career change into SwE. In some cases, those students will be fresh graduates with a bachelor's degree with little or no experience. Their lack of experience is a challenge in realizing the educational outcomes identified in GSwE2009—a concern that is explored in several places in this document.

---

[1] IEEE STD 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Computer Society, 1990.

[2] Ardis, M. and Ford, G. *SEI Report on Graduate Software Engineering Education*, CMU/SEI 89-TR-21, Software Engineering Institute, Carnegie Mellon University, June 1989.

[3] A *reference curriculum* is a set of outcomes, entrance expectations, architecture, and a body of knowledge that provide guidance for faculty who are designing and updating their programs. That guidance is intentionally flexible so that faculty can adopt and adapt it based on local programmatic needs. A reference curriculum is not intended to be used directly for program certification or accreditation.

GSwE2009 was created to:

- Improve existing graduate programs in SwE from the viewpoint of universities, students, graduates, software builders, and software buyers;
- Enable the formation of new graduate programs in SwE by providing guidelines on curriculum content and advice on how to implement those guidelines; and
- Support increased enrollment in graduate SwE programs by increasing the value of those programs to potential students and employers.

GSwE2009 builds on the SEI curriculum foundations plus those of other initiatives, such as the *Guide to the Software Engineering Body of Knowledge* (SWEBOK)[4] and *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* (SE2004)[5]. iSSEc followed an iterative, evolutionary approach in creating GSwE2009, beginning with the formation of an Early Start Team (EST) of authors, since renamed the Curriculum Author Team (CAT). First established in July 2007, the CAT is a set of invited experts from industry, government, academia, and professional associations. CAT membership grew as GSwE2009 matured.

The CAT met in workshops approximately every three months between August 2007 and September 2009, leading to the release of GSwERC 0.25 in February 2008, GSwERC 0.5 in October 2008, and GSwE2009 1.0 in September 2009. The SwE community was invited to review versions 0.25 and 0.5 to provide the necessary feedback to develop the current version (1.0). The review of version 0.5 generated more than 800 individual review comments, which were adjudicated for use in creating version 1.0. The detailed comments and their adjudication can be found at www.GSwE2009.org.

Professional society participation in the creation of GSwE2009 has been essential to ensuring that GSwE2009 will have the desired impact on global graduate education. Both the International Council on Systems Engineering (INCOSE) and the U.S. National Defense Industrial Association (NDIA) Systems Engineering Division were early participants in GSwE2009, and each contributed authors. In 2008, the Institute of Electrical and Electronics Engineers (IEEE) Computer Society Education Activities Board became an official participant. In 2009, that participation elevated to the Computer Society level and both the Association for Computing Machinery (ACM) and the Brazilian Computer Society (BCS) also chose to participate. Discussions are underway with the ACM, IEEE Computer Society, and INCOSE in the hope that they will jointly take on the evolution and maintenance of GSwE2009. Finally, GSwE2009's

[4] SWEBOK, *Guide to the Software Engineering Body of Knowledge*, P. Bourque & R. Dupuis (Eds.), IEEE Computer Society Press, 2004.

[5] ACM/IEEE-CS Joint Task Force on Computer Curricula, *Software Engineering 2004, Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering,* August 2004.

success has motivated the start of related efforts to create a Systems Engineering Body of Knowledge and a Graduate Systems Engineering Reference Curriculum—each with an "appropriate" amount of SwE perspective and content. Those efforts should lead to version 1.0 products in 2012.

The following authors contributed to the creation of GSwE2009:

1. *Rick Adcock, Cranfield University and INCOSE representative, UK*
2. *Edward Alef, General Motors, USA*
3. *Bruce Amato, Department of Defense, USA*
4. *Mark Ardis, Stevens Institute of Technology, USA*
5. *Larry Bernstein, Stevens Institute of Technology USA*
6. *Barry Boehm, University of Southern California, USA*
7. *Pierre Bourque, École de Technologie Supérieure and SWEBOK volunteer, Canada*
8. *John Brackett, Boston University, USA*
9. *Murray Cantor, IBM, USA*
10. *Lillian Cassel, Villanova and ACM representative, USA*
11. *Robert Edson, Analytic Services Inc., USA*
12. *Richard Fairley, Colorado Technical University, USA*
13. *Dennis Frailey, Raytheon and Southern Methodist University, USA*
14. *Gary Hafen, Lockheed Martin and NDIA, USA*
15. *Thomas Hilburn, Embry-Riddle Aeronautical University, USA*
16. *Greg Hislop, Drexel University and IEEE Computer Society representative, USA*
17. *David Klappholz, Stevens Institute of Technology, USA*
18. *Philippe Kruchten, University of British Columbia, Canada*
19. *Phil Laplante, Pennsylvania State University, Great Valley, USA*
20. *Qiaoyun (Liz) Li, Wuhan University, China*
21. *Scott Lucero, Department of Defense, USA*
22. *John McDermid, University of York, UK*
23. *James McDonald, Monmouth University, USA*
24. *Ernest McDuffie, National Coordination Office for NITRD, USA*
25. *Bret Michael, Naval Postgraduate School, USA*
26. *William Milam, Ford, USA*
27. *Ken Nidiffer, Software Engineering Institute, USA*
28. *Art Pyster, Stevens Institute of Technology, USA*
29. *Paul Robitaille, Lockheed Martin & INCOSE representative, USA*
30. *Mary Shaw, Carnegie Mellon University, USA*
31. *Sarah Sheard, Third Millenium Systems, USA*
32. *Robert Suritis, IBM, USA*
33. *Massood Towhidnejad, Embry-Riddle Aeronautical University, USA*
34. *Richard Thayer, California State University at Sacramento, USA*
35. *J. Barrie Thompson, University of Sunderland, UK*
36. *Guilherme Travassos, Brazilian Computer Society, Brazil*
37. *Richard Turner, Stevens Institute of Technology, USA*
38. *Joseph Urban, Texas Tech University, USA*
39. *Ricardo Valerdi, MIT & INCOSE representative, USA*
40. *Osmo Vikman, Nokia, Finland*
41. *David Weiss, Avaya, USA*
42. *Mary Jane Willshire, Colorado Technical University, USA*

Acknowledgments

The reference curriculum is the product of many authors from more than 24 organizations who came together selflessly to improve global SwE graduate education. Those authors are listed individually in the preface along with their supporting organizations. Special thanks go to team leaders Dennis Frailey, Tom Hilburn, Jim McDonald, and Bret Michael, who took on the added burden of organizing teleconferences, drafting sections of the document, and helping to pull together the current document and its earlier versions. It is hard to single people out because so many contributed so much. Nevertheless, I want to especially thank Mark Ardis, Larry Bernstein, Barry Boehm, John Brackett, Richard Fairley, Phil Laplante, Mary Shaw, Richard Thayer, J. Barry Thompson, Massood Towhidnejad, Richard Turner, Mary Jane Willshire, and Joe Urban who continuously provided deep and thoughtful comments throughout the CAT's long discussions. Pierre Bourque offered invaluable coordination with independent efforts now underway to update the Software Engineering Body of Knowledge. Appendix C.2, which is a description of systems engineering that is critical to GSwE2009, is largely the writings of Rick Adcock, Barry Boehm, Richard Fairley, Tom Hilburn, and Ricardo Valerdi. We are grateful to the following for hosting workshops that were critical to creating GSwE2009: Analytic Services Inc. and Robert Edson; the National Coordination Office for the Networking and Information Technology Research and Development (NITRD) Program and Ernest McDuffie; Stevens Institute of Technology; the Naval Postgraduate School and Bret Michael; Mary Jane Willshire and Richard Fairley and Colorado Technical University; Massood Towhidnejad, Tom Hilburn and Embry-Riddle Aeronautical University; Mary Shaw and Carnegie Mellon University; and Jim McDonald and Monmouth University. INCOSE and the NDIA Systems Engineering Division both endorsed this effort long ago and provided valued members to the author team. (Originally, Paul Robitaille represented INCOSE, followed by Ricardo Valerdi and Rick Adcock. Gary Hafen has represented the NDIA throughout.) Boots Cassel represented the ACM and Greg Hislop represented the IEEE Computer Society. Guilherme Travassos represented the Brazilian Computer Society. The three graduate students who most supported this effort, Devanandham Henry, Nicole Hutchison, and Kahina Lasfer, all did a terrific job of collecting and analyzing data, consolidating inputs from various authors, handling workshop logistics, and a million other activities. Graduate students Sarah Sheard and Jimmy Gandhi also helped early in the project.

Over the course of the project, more than 100 people contributed reviews. We are grateful for their thoughtful contributions:

1. *K.K. Aditya, Satyam Computer Services Ltd., India*
2. *Da Shasbikant Albal, SSN School of Advanced Software Engineering, India*
3. *Michael Barker, Nara Institute of Science and Technology, Japan*
4. *Adeline Beaulac, Retired, USA*
5. *David Belanger, AT&T, USA*
6. *Oddur Benediktsson, University of Iceland, Iceland*
7. *Brian Berenbach, Siemens Corporate Research, Inc., USA*
8. *Jean Bezivin, University of Nantes, France*
9. *Ilia Bider, IbisSoft, ab, Sweden*

10. *Shawn Bohner, Rose-Hulman Institute of Technology, USA*
11. *Grady Booch, IBM, USA*
12. *Juris Borzovs, University of Latvia, Latvia*
13. *Nick Brixius, Embry-Riddle Aeronautical University, USA*
14. *Ann E. Broihier, Southern Methodist University, USA*
15. *Luigi Buglione, Engineering IT (Italy), Italy*
16. *Robert C. Burns, The Boeing Company, USA*
17. *Joseph Carl, Riverside Research Institute, USA*
18. *Steve Chenoweth, Rose-Hulman Institute of Technology, USA*
19. *Boris Cogan, London Metropolitan University, UK*
20. *Tony Cowling, University of Sheffield, UK*
21. *Brad Crabtree, Raytheon, USA*
22. *David A. Dampier, Mississippi State University, USA*
23. *Yvonne Delaney, University of Limerick, UK*
24. *Audrey Dorofee, Software Engineering Institute, USA*
25. *Weichang Du, University of New Brunswick, Canada*
26. *Sheryl Duggins, Southern Polytechnic State University, USA*
27. *Christof Ebert Vector, Germany*
28. *Amr El-Kadi, The American University in Cairo, Egypt*
29. *Geoff Ewens, AITEC Corporate Education & Consulting, Australia*
30. *Alain Faisandier, INCOSE MAP Systems, USA*
31. *Stuart Faulk, University of Oregon, USA*
32. *John Favaro, Self employed, Italy*
33. *Tim Ferris, University of South Australia (SEEC), Australia*
34. *David Luigi Fuschi, Intelligent Media systems and Services (IMSS)–School of systems engineering, University of Reading, UK*
35. *Keith Garfield, Embry-Riddle Aeronautical University, USA*
36. *Kirti Garg, International Institute of Information Technology, Hyderabad, India*
37. *Robert Glass, Griffith University Brisbane, Australia*
38. *Hassan Gomaa, George Mason University, USA*
39. *Janusz Gorski, Gdansk University of Technology, Poland*
40. *Doug Grant, Swinburne University of Technology, Australia*
41. *Frank Gutcher, Boeing, USA*
42. *Hans Hadderingh, Logica Public Sector CMG, Netherlands*
43. *Gary Hafen, Lockheed Martin, USA Jon Hagar, Lockheed Martin, USA*
44. *Haitham S. Hamza, Cairo University, Egypt*
45. *John Harauz, Jonic Systems Engineering, Inc., USA*
46. *Cecelia Haskins, Geoff Sharman Birkbeck College, London, UK*
47. *Kojun T. Hatta, Carestream Health, Inc. USA*
48. *Orit Hazzan, Technion - Israel Institute of Technology, Israel*
49. *Rick Hefner, Northrop Grumman Corporation, USA*
50. *Peter Henderson, Butler University, USA*
51. *Watts S. Humphrey, Software Engineering Institute, USA*
52. *Mario Jino, University of Campinas, Brazil*
53. *Ron Kenett, KPA Ltd., Israel*
54. *John Klein, Software Engineering Institute, USA*
55. *Peter Knoke, University of Alaska Fairbanks, USA*
56. *Supannika Koolmanojwong, University of Southern California, USA*
57. *Carl Landrum, Honeywell Aerospace, USA*
58. *Richard LeBlanc, Seattle University, USA*
59. *Bob Lechner, University of Massachusetts Lowell, USA*
60. *Cuauhtémoc Lemus Olalde, Centro de Invistigacion en Matematicas, A. C. (CIMAT),Mexico*
61. *Tim Lethbridge, University of Ottawa, Canada*
62. *Hareton Leung, Hong Kong Polytechnic University, Hong Kong*
63. *Fengqi Li, Dalian University of Technology, China*
64. *Qiaoyun (Liz) Li, Wuhan University, China*
65. *David Long, Vitech, USA*
66. *Paul E. MacNeil, Mercer University, USA*
67. *Spiros Mancoridis, Drexel University, USA*
68. *Dino Mandrioli, Politecnico di Milano, Italy*
69. *Frank Maurer, University of Calgary, USA*
70. *Bruce Maxim, University of Michigan Dearborn, USA*
71. *John McDermid, University of York, UK*
72. *Andrew McGettrick, University of Strathclyde, UK*
73. *Nancy Mead, Carnegie Mellon University, USA*

74. *Luisa Mich, University of Trento, Italy*
75. *Alan Milewski, Monmouth University, USA*
76. *Nilofer Mohammed, Satyam Computer Services Ltd., India*
77. *Mike Murphy, Southern Polytechnic State University, USA*
78. *Mary Poppendieck, Poppendieck LLC, USA*
79. *Xue Qiang, Dalian University of Technology, China*
80. *Fabio Queda Bueno Da Silva, Federal University of Pernambuco, Brazil*
81. *Damith C. Rajapaske, National University of Singapore, Singapore*
82. *A.R. Thuasi Ram, Satyam Computer Services Ltd., India*
83. *Ita Richardson, University of Limerick, UK*
84. *Richard Riehle, Naval Postgraduate School, USA*
85. *Steve Roach, University of Texas El Paso, USA*
86. *Pierre N Robillard, Ecole Polytechnique de Montreal, Canada*
87. *Howard Rosenberg, NetSearchers Inc, USA*
88. *Mel Rosso-Llopart, Carnegie Mellon University, USA*
89. *P. Srinivala Ruo, Satyam Computer Services Ltd., India*
91. *Michael Ryan, Dublin City University, UK*
92. *Vladimir O. Safonov, St. Petersburg University, Russia*
93. *Andy Sage, George Mason University, USA*
95. *Salamah Salamah, Embry-Riddle Aeronautical University, USA*
96. *Alberto Sampaio, Instituto Superior de Engenharia do Porto (ISEP), Portugal*
97. *L. Sasidhav, Satyam Computer Services Ltd., India*
98. *Hasan Sayani, University of Maryland University College, USA*
99. *Geoff Sharman, Birkbeck College, London, UK*
100. *Peraphon Sophatsathit, Chulalongkorn University, Bangkok, Thailand*
101. *Diomidis Spinellis, Athens University of Economics and Business, Greece*
102. *Richard Stansbury, Embry-Riddle Aeronautical University, USA*
103. *Sara Stoecklin, Florida State University, USA*
104. *Steve Tockey, Construx Software, USA*
105. *Richard Torkar, Blekinge Institute of Technology, Sweden*
106. *Guilherme Horta Travassos, COPPE/Federal University of Rio de Janeiro, Brazil*
107. *Nicolas Treves, CNAM, France*
108. *Giuseppe Valetto, Drexel University, USA*
109. *Ann Vu, IEEE Computer Society*
110. *Yingxu Wang, University of Calgary, Canada*
111. *Larry Wear, University of Washington Tacoma, USA*
112. *Ye Yang, Institute of Software Chinese Academy of Sciences (ISCAS), China*
113. *Liguo Yu, Indiana University South Bend, USA*
114. *Ming Zhu, School of Software of Dalian University of Technology, China*
115. *Jürgen P. Znotka, University of Applied Sciences Gelsenkirchen, Germany*

We are also grateful to Kristen Baldwin, Bruce Amato, Scott Lucero, and others in the U.S. Office of the Secretary of Defense for their leadership and financial support for this effort. Paramount to our success has been Ms. Baldwin's early and consistent recognition that a reference curriculum for SwE would be of most benefit to the defense community if it were not biased toward defense applications. Finally, we thank Stevens Institute of Technology, especially the School of Systems and Enterprises and its dean, Dinesh Verma, for supporting this effort from the very beginning.

Art Pyster

Art Pyster
GSwE2009 Editor

Executive Summary

The *Graduate Software Engineering 2009 (GSwE2009): Curriculum Guidelines for Graduate Degree Programs in Software Engineering* is a set of recommendations for a master's level graduate program in software engineering (SwE), together with implementation guidance for a university to satisfy those recommendations. Earlier versions of this work were called the *Graduate Software Engineering Reference Curriculum* (GSwERC).

The program described by GSwE2009 is for a professional master's degree, analogous in many ways to a master's of business administration. GSwE2009 is envisioned as a living document that will be revisited regularly and updated when necessary to ensure relevance to the rapidly evolving software engineering discipline. This document includes the curriculum recommendations and materials describing their creation, implementation, and evolution.

GSwE2009 includes the following:

- A set of outcomes to be fulfilled by a student who successfully completes a graduate program based on the curriculum (see summary below)
- A set of student skills, knowledge, and experience assumed by the curriculum, not intended as entrance requirements for a specific program, but as the starting point for the curriculum's outcomes (see summary below)
- An architectural framework to support implementation of the curriculum
- A description of the fundamental or core skills, knowledge, and experience to be taught in the curriculum to achieve the outcomes. This is termed a *Core Body of Knowledge* (CBOK) and includes topic areas and the depth of understanding a student should achieve.

Additional materials included in this document:

- The fundamental philosophy for GSwE2009 development as described in a set of guiding principles (see summary below)
- A discussion of how GSwE2009 will evolve to remain effective
- A mapping of expected outcomes to the CBOK and to the total GSwE2009 program recommendations
- A description of Knowledge Areas (KAs) discussed in GSwE2009 that are not yet fully integrated into the current version of the Software Engineering Body of Knowledge (SWEBOK)
- Glossary, references, and other supporting material.

Summary of Guidance for Creating GSwE2009

The following guidance, established early in the development of GSwE2009, came primarily

from SE2004[6] and the deliberations of the GSwE2009 authors.

- Create a set of recommendations for developing and improving curricula for master's level software engineering education.
- Target a professional degree for practicing software engineers.
- Require about as many credit hours as typical programs do now.
- Recognize software engineering as a distinct discipline with a rich body of knowledge, practice, and theory.
- Recognize that software engineering is founded on a wide variety of disciplines, with deepening ties to Systems Engineering (SE).
- Require that all software engineering students be able to integrate theory and practice.
- Foster ongoing review and revision of the curriculum because of rapid evolution in software engineering.
- Be sensitive to changes in technologies, practices, applications, pedagogy, and the importance of lifelong learning.
- Offer significant guidance in individual curriculum components through a CBOK that all students are expected to master.
- Identify fundamental skills and knowledge that all software engineering master's program graduates must possess.
- Use a flexible curriculum architecture and recognize existing bodies of knowledge, modified and enhanced as required.
- Limit common knowledge required for *all* students to no more than 50% of total knowledge taught.
- Be broadly based and international in scope.
- Include exposure to aspects of professional practice as an integral component of the graduate curriculum.
- State strategies and tactics for implementation.
- Distinguish clearly between SE2004 and GSwE2009.
- Identify expected knowledge and experience for students to enter a master's program in software engineering.

[6] ACM/IEEE-CS Joint Task Force on Computer Curricula, *Software Engineering 2004, Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering,* August 2004.

Summary of Outcomes

Graduates of a master’s program that satisfies GSwE2009 recommendations will:

- Master the CBOK.
- Master software engineering in at least one application domain, such as finance, medical, transportation, or telecommunications, and one application type, such as real-time, embedded, safety-critical, or highly distributed systems. That mastery includes understanding how differences in domain and type manifest themselves in both the software itself and in its engineering, and includes understanding how to learn a new application domain or type.
- Master at least one KA or sub-area from the CBOK to at least the Bloom Synthesis level.
- Be able to make ethical professional decisions and practice ethical professional behavior.
- Understand the relationship between SwE and SE and be able to apply SE principles and practices in the engineering of software.
- Be an effective member of a team, including teams that are international and geographically distributed, effectively communicate both orally and in writing, and lead in one area of project development, such as project management, requirements analysis, architecture, construction, or quality assurance.
- Be able to reconcile conflicting project objectives, finding acceptable compromises within limitations of cost, time, knowledge, existing systems, and organizations.
- Understand and appreciate feasibility analysis, negotiation, and good communications with stakeholders in a typical software development environment, and be able to perform those tasks well; have effective work habits and be a leader.
- Be able to learn new models, techniques, and technologies as they emerge, and appreciate the necessity of such continuing professional development.
- Be able to analyze a current significant software technology, articulate its strengths and weaknesses, compare it to alternative technologies, and specify and promote improvements or extensions to that technology.

Summary of Expected Background

GSwE2009 presumes that an entering student has:

- The equivalent of an undergraduate degree in computing or an undergraduate degree in an engineering or scientific field and a minor in computing,

- The equivalent of an introductory course in software engineering, and
- At least two years of practical experience in some aspect of software engineering or software development.

These presumptions about entering students are designed to achieve the 10 outcomes previously described. However, it is recognized that individual schools may start with a student population that has characteristics that are different from what GSwE2009 presumes here. Such schools will likely have to lengthen their master's programs in order for their students to achieve all 10 outcomes—or the schools will deliberately choose not to adopt some outcomes. In fact, schools may even add other outcomes to favor their particular markets and institutional emphases. GSwE2009 is not intended for use to certify or accredit either programs or individuals. It is a set of recommendations that must be tailored by each adopting university.

The process of developing this report has been highly inclusive. It has been widely reviewed by academics and practitioners through a public draft process. We have also held feedback sessions at conferences and meetings, including the annual American Society for Engineering Education (ASEE) meeting, the International Symposium of the International Council on Systems Engineering (INCOSE), the International Conference on Software Engineering (ICSE), the Conference on Software Engineering Education and Training (CSEET), and various smaller meetings in Europe, Asia, and various parts of the United States. These meetings have provided us with critically important feedback that we have used to shape the final report.

From the beginning it was intended for GSwE2009 to be a living document, with a broad, responsible, and knowledgeable community of practice. It was anticipated that after Version 1.0 was published, Stevens Institute of Technology, which has managed the original development, would identify a steward who would assume responsibility for maintaining and refining the model and expanding and focusing implementation guidance based on experience and feedback from the supporting community and academia, industry, and students. Effort is now underway for a combination of the ACM and the IEEE Computer Society to become that steward. As of the writing of this document, discussions are underway for those two organizations to take over maintenance responsibility for GSwE2009 within the first 6 months of the release of Version 1.0, with INCOSE playing a supporting role.

To support and enable wide acceptance of GSwE2009, two companion documents—*Comparisons of GSwE2009 to Current Master's Programs in Software Engineering* and *Frequently Asked Questions on Implementing GSwE2009*– are being prepared concurrently with the release of GSwE2009. They will be available in Fall 2009 at www.GSwE2009.org and updated regularly.

1. Introduction

Software is a critical component in new products worldwide—often *the* critical component distinguishing products in the marketplace. Software enables technological advances that lead to new, high-performance products and systems in every commercial sector, including medical devices, automobiles, aircraft, power generation systems, mobile phones, and entertainment systems. Automobiles now have millions of lines of embedded code, enabling everything from voice-activated navigation systems for convenience, to anti-lock brake systems for safety, to engine control for fuel efficiency. In fact, one of the primary ways manufacturers now differentiate their cars is through functionality implemented largely through sophisticated software, such as Global Positioning System (GPS) navigation systems.

As product and system functionality grow, so does the need to efficiently and correctly implement the complex software that enables that growth. Sophisticated systems are critical for any large company or government agency in managing their projects and organizations. Such systems support essential business and technical processes, such as recordkeeping and data warehousing, logistics, manufacturing, coordination with suppliers, and customer relationship management. Systems engineers (not always by that name) are typically responsible for the design and development of such complex systems, but because a large part of the functionality is usually implemented in software today, a large part of the implementation responsibility typically falls on software engineers. Indeed, the fields of SwE and systems engineering (SE) are becoming increasingly intertwined and it is vital that corresponding educational programs reflect this.

Because of software complexity and the inherent difficulties of software development, most of the "surprises" that occur during system integration, and after product shipment and system deployment, can be traced back to incorrect software implementation and integration, such as poor requirements or faulty software upgrades in the field. Often these problems show up as interoperability issues between system components that otherwise seem correct in isolation. The common practice of repurposing legacy software for new applications simply adds to the challenges. Robert Glass,[7,8] Nancy Leveson,[9,10] and others have documented countless examples of failures resulting from poor SwE and/or poor communication between systems and software

[7] Glass, R. L., *Computing Calamities: Monumental Computing Disasters*, Prentice Hall Professional Technical Reference, 1998.

[8] Glass, R. L., *Software Runaways: Monumental Software Disasters*, Prentice Hall Professional Technical Reference, 1997.

[9] Leveson, N. G., *Safeware: Systems Safety and Computers*, Addison-Wesley, 1995.

[10] Leveson, N. G., "The Role of Software in Spacecraft Accidents". *AIAA Journal of Spacecraft and Rockets*, *41*(4), July 2004.

engineers. The U.S. Government Accountability Office[11,12,13] regularly issues reports recounting the challenges the U.S. government faces in creating large-scale, reliable, software-intensive systems on schedule, on budget, and with expected functionality.

SE is the discipline through which large-scale, trustworthy, and complex systems are developed, while SwE is the discipline by which the software portions of these systems are developed. Many universities teach SE and SwE at the undergraduate level. Over the years, a few well-known efforts have created guidelines and sample curriculum for SwE. Most notably, more than 100 colleges and universities helped create curriculum guidelines for undergraduate SwE education that the ACM and IEEE Computer Society published in SE2004. Many universities offer a master's degree in SE and/or SwE, but there are no widely accepted curriculum guidelines for graduate education in SE (although INCOSE has published a high-level curriculum[14]). In 1989 the Software Engineering Institute (SEI) of Carnegie Mellon University published a landmark report on graduate education in SwE.[15] A fresh look at curriculum guidance in these fields is in order, considering the reliance of the world economy on the quality of senior SE and SwE professionals, the dramatic changes that the Internet has brought about in how software is created, and continuing global problems in producing satisfactory software.

The iSSEc (Integrated Software and Systems Engineering Curriculum) project was formed in July 2007 to create and promulgate a series of graduate-level reference curricula[16] related to SwE and SE. Led by Stevens Institute of Technology with dozens of experts from other universities, industry, government, and professional societies, iSSEc's first product is this volume, *Graduate Software Engineering 2009 (GSwE2009): Curriculum Guidelines for Graduate Degree Programs in Software Engineering*, reflecting new understandings in how to build software, how SwE depends on SE, and how SwE education is influenced by application domains, such as telecommunications and defense systems. A study of existing graduate programs in SwE

---

[11] GAO, *Defense Acquisitions: Assessment of Selected Major Weapons Programs*, U.S. Government Accountability Office, GAO-08-467SP, March 2008.

[12] GAO, *Information Technology: Inconsistent Software Acquisition Processes at the Defense Logistics Agency Increase Project Risks*, U.S. Government Accountability Office, GAO-02-9, January 2002.

[13] GAO, *Information Security: Agencies Face Challenges in Implementing Effective Software Patch Management Processes*, U.S. Government Accountability Office, GAO-04-816T, June 2004.

[14] Jain, R. and Verma, D., *A Report on Curriculum Content for a Graduate Program in Systems Engineering: A Proposed Framework*, INCOSE International Symposium, 2007.

[15] Ardis, M. and Ford, G., *SEI Report on Graduate Software Engineering Education*, CMU/SEI 89-TR-21, Software Engineering Institute, Carnegie Mellon University, June 1989.

[16] A *reference curriculum* is a set of outcomes, entrance expectations, architecture, and a body of knowledge that provide guidance for faculty who are designing and updating their programs. That guidance is intentionally flexible so that faculty can adopt and adapt it based on local programmatic needs. A reference curriculum is not intended to be used directly for program certification.

revealed their diversity and helps motivate GSwE2009.[17] (A short version of the study report is included in Appendix A.) Prior to August 2009, GSwE2009 was known as the Graduate Software Engineering Reference Curriculum (GSwERC).

GSwE2009 is targeted at a university education leading to a professional master's degree in SwE—that is, a degree intended for someone who will either enter the workforce as a software engineer or someone who is already in the workforce and is seeking to gain a formal education in SwE to advance his or her career. GSwE2009 does not explicitly target graduate programs for those who seek a doctorate and a career in research. In some cases, entering students will be fresh graduates with a bachelor's degree with little or no experience. Their lack of experience is a challenge in realizing the educational outcomes identified in GSwE2009—a challenge that is explored in several places in this document.

There is tremendous diversity in markets that universities serve, educational systems in which universities operate, accreditation programs, size of student body and faculty, and many other factors that affect program content and delivery. GSwE2009 respects and enables that diversity. GSwE2009 is a broad set of recommendations to guide universities in building and updating their graduate programs. As a reference curriculum, it identifies a core that should be included in all programs and an extensive envelope that allows and encourages variation among programs. It provides wide flexibility in how those recommendations are implemented: for example, GSwE2009 includes a core body of knowledge (CBOK) that all students should master (Section 6), but GSwE2009 does not prescribe a particular course[18] packaging to deliver them.

Reflecting its purpose as part of a reference curriculum, the CBOK has been limited in scope so that a student should spend no more than half of his or her class time learning it. This gives both individual universities and students a great deal of flexibility in how they round out the master's program to achieve GSwE2009 outcomes, as well as achieve university and individual outcomes.

Two companion documents aid faculty in adapting and adopting GSwE2009. The first of these documents, *Comparisons of GSwE2009 to Current Master's Programs in Software Engineering*, explains how well current university programs align with GSwE2009 recommendations. The second companion document, *Frequently Asked Questions on Implementing GSwE2009*, offers specific advice on such topics as identifying faculty who are best able to teach classes that implement GSwE2009 recommendations.

[17] Pyster, A., et al., "Master's Degrees in Software Engineering: An Analysis of 28 University Programs," *IEEE Software*, September-October 2009, 94-101.

[18] The term *course* refers to a collection of materials, exercises, and assessments for which academic credit is awarded. A *program* is a collection of courses leading to a degree—the specific interest here being in a master's degree. Often programs have one or more specific orientations called *tracks* that allow a student to specialize in an area of interest such as real-time systems, security, or Web applications.

Despite the freedom that universities have in how they implement GSwE2009, there are several constants. For example, a program that fully satisfies GSwE2009 recommendations will enable its students to achieve *all* the outcomes listed in Section 3 and will follow the curriculum architecture found in Section 5.

This document, which provides the GSwE2009 recommendations, has six sections:

- Section 1 is this introduction.
- Section 2 contains general guidance for those who authored and will maintain GSwE2009.
- Section 3 states the outcomes that a student is expected to achieve immediately upon graduation.
- Section 4 explains the background that students are expected to have when entering a master's program that satisfies GSwE2009 recommendations.
- Section 5 presents the curriculum architecture that any curriculum following these guidelines should satisfy. That architecture supports three levels of knowledge—knowledge that all students should master in every university, knowledge that an individual university program requires for its students, and knowledge that an individual student elects to learn.
- Section 6 is the CBOK that all students in every university should learn. It includes specific knowledge units and the expected Bloom level for each unit. The Bloom Taxonomy offers a six-level scale for competency mastery that is commonly used for curriculum development, further elaborated in Appendix B. Most knowledge units are based on SWEBOK, but others have been added where the Curriculum Author Team (CAT) felt the SWEBOK omitted critical material.

2. Guidance for the Construction and Maintenance of GSwE2009

This section describes the foundational guidance used when developing GSwE2009—the guiding principles, assumptions, and context for the entire GSwE2009 effort. The 17 numbered guidance statements are primarily written in future tense, reflecting their role as *requirements* for the GSwE2009 development effort. The elaboration following each guidance statement speaks to how GSwE2009 satisfies those requirements.

The guidance was strongly influenced by the principles stated in SE2004; in some cases, it repeats verbatim the wording of those principles. Differences arise primarily by distinguishing the higher expectations of graduate education from those of undergraduate education and by more explicitly recognizing how the SwE discipline ties to other disciplines, notably SE. Moreover, we recognize that this guidance is, in many cases, not unique to SwE curricula. It is valid for virtually all engineering disciplines. For example, guidance statement [6] is "All SwE students must learn to integrate theory and practice." Substituting "mechanical," "electrical," or any other engineering discipline for "software" would not change its validity. Nevertheless, these statements were helpful to those developing GSwE2009 and are therefore included here. Note that these statements are numbered for ease of reference only. The numbering does not signify relative importance.

[1] *The principal purpose of GSwE2009 will be to provide a set of tailorable recommendations for developing and improving curricula that provide software engineering education at the master's degree level. It is not intended to be the basis for accreditation.*

GSwE2009 supports the needs of industry and government for software engineers by helping universities equip software engineers with the most current theory and practice, and helping them develop their ability to lead in addressing the future challenges of software development. GSwE2009 is a reference curriculum, not a single definitive curriculum. It provides a set of common recommendations for faculty at different universities to use when constructing individual curricula for a master's degree in SwE—*it should be tailored to each program*. Universities give their degrees different names, often with no pedagogical implications; e.g., Stevens Institute of Technology offers a *Master of Science (MS) in Software Engineering*, while Embry-Riddle Aeronautical University offers a *Master of Software Engineering (MSE)*. GSwE2009 is suitable for programs that educate software engineers regardless of the degree name.

GSwE2009 should not be used to score, appraise, accredit, or certify programs for *compliance*. Phrases such as "*GSwE2009-compliant"* are not used herein. Instead, the softer term "*satisfies GSwE2009 recommendations"* periodically appears. The latter term has no precise meaning, but is intended for a program that follows most GSwE2009 recommendations. The program may deviate from some recommendations: for example, a program may choose to omit some of the 10 outcomes found in Section 3,or even add one or two new outcomes based on local preferences. It may admit students who do not meet the entrance expectations found in Section 4. It may deviate from the recommendations in Section 6 on core knowledge that every graduate should master. Clearly, there is a point at which a university tailors too much. After all, GSwE2009 reflects the collective wisdom of a broad community of authors and reviewers about graduate SwE education. However, that point of "too much" tailoring is not precise and is not prescribed here.

[2] *The master's degree described by GSwE2009 will be a professional degree targeting practicing software engineers. With modification, GSwE2009 may serve as the foundation for those with a research interest who ultimately seek a doctoral degree; however, GSwE2009 is designed specifically to support professional degrees.*

The vast majority of students who earn a master's degree in SwE do not become researchers. They are practicing professionals or aspiring practicing professionals seeking to improve their skills and career opportunities. GSwE2009 will target practicing professionals because that is where the greatest need is.

[3] *A master's program that satisfies GSwE2009 should require about as many credits*[19] *as typical programs do now.*

A minimum number of credits are necessary to convey the breadth and depth of the requisite knowledge and to develop the desired skills in students. Universities will individually decide how to package those credits into courses, although the two companion documents to GSwE2009 offer packaging observations and recommendations. The 2008 study of 28 SwE graduate programs described in Appendix A and more fully in (Pyster, et. Al., 2009) that the average master's program requires between 33 and 36 graduate course credits, typically packaged into 11 or 12 3-credit semester courses using the common U.S. education model. This would roughly equate to 66 to 72 European Credit Transfer and Accumulation System (ECTS) credits for those universities following the Bologna Declaration.[20] One graduate course credit nominally equates to approximately 13 to 14 contact hours between faculty and student, plus homework. Full-time students normally complete such programs in 18 to 24 months. The need to achieve a master's level of professional achievement with this amount of study leads to expectations about what students should be capable of doing when entering a program that attempts to satisfy GSwE2009 recommendations.

[19] Typically, a program requires a certain number of credits for graduation, awarded by taking courses, each of which has an associated number of credits. Historically, the number of credits per course has often aligned with the number of hours of lecture per week, but with online and other non-traditional formats increasingly popular, the rules for assigning credits to a class have become more varied.

[20] Joint Declaration of the European Ministers of Education, "The European Higher Education Area," Convened in Bologna on June 19, 1999.

[4] *Software engineering is a distinct discipline with a rich body of knowledge, practice, and theory.*

SwE does not have the long history of many other engineering disciplines, such as electrical and mechanical engineering. Nevertheless, SwE has matured relatively quickly. Since 1968, when the term "software engineering" was coined, the discipline has spawned numerous journals, conferences, professional societies, undergraduate and graduate programs, professional certifications, bodies of knowledge, and standards—all geared toward establishing broadly achievable levels of performance, all hallmarks of a scientific or engineering discipline.

[5] *Software engineering draws its foundations from a wide variety of disciplines, with deepening ties to systems engineering.*

GSwE2009 concentrates on the knowledge and pedagogy associated with a graduate SwE curriculum. Where appropriate, it shares or overlaps with material contained in other ACM/IEEE Computing Curriculum reports[21] and it offers guidance on its incorporation into other disciplines.

Graduate study in SwE relies on many areas in computer science for its theoretical and conceptual foundations, but it also draws from other fields, including statistics, logic, calculus, discrete mathematics, formal languages, and other mathematical specialties, and from economics, project management, general engineering, and one or more application domains.

SE is a special case. In the past, many universities have made only cursory ties between SwE and SE, reflecting the "old school" view that SE is driven by hardware considerations and SwE is just one of many "specialty" disciplines. As explained in Section 1, that view no longer serves either the SE or the SwE communities well. GSwE2009 provides strong explicit linkage between SE and SwE.

[6] *All software engineering students must learn to integrate theory and practice.*

Students must be able to recognize the importance of abstraction and modeling for software architecture, design, and specification; to be able to acquire special domain knowledge beyond the computing discipline in order to support software development in specific domains of application; and to appreciate the value and attributes of good design. They should be expected to solve real-world problems, relying on the underlying principles taught in their graduate education.

[21] Shackelford, R., et al., *Computing Curricula: 2005 Overview Report*, ACM, 2006.

[7] *The rapid evolution and professional nature of software engineering require ongoing review and revision of the corresponding curriculum.*

Universities, industry, and government, in cooperation with professional associations in this discipline, must establish an ongoing review process that allows individual components of the curriculum recommendations to be updated on a recurring basis. Also, because of the special professional responsibilities of software engineers to the public, the curriculum guidance could support and promote effective external assessment and accreditation of graduate SwE programs. Nevertheless, accreditation is outside the scope of GSwE2009, and it should not be used directly for accreditation. GSwE2009 is a snapshot of recommendations suitable for today and should define mechanisms for ongoing revision as SwE evolves. If current efforts to migrate the maintenance of GSwE2009 to the Association for Computing Machinery (ACM) and IEEE Computer Society are successful, the standards mechanisms those two professional societies operate should provide periodic revision of GSwE2009 as required.

[8] *GSwE2009 will be sensitive to changes in technologies, practices, and applications, new developments in pedagogy, and the importance of lifelong learning.*

The principles underlying SwE change relatively slowly, but the technology through which SwE is practiced keeps changing at breakneck speed. Educational institutions must adopt explicit strategies for responding to changing technology without being caught in the trap of simply "training" the latest technology. A key to this is organizing the curriculum around enduring principles and planning to change the example technologies regularly. Institutions must recognize the importance of remaining abreast of well-established progress in both technology and pedagogy, subject to the constraints of available resources. SwE education, moreover, must seek to prepare students for lifelong learning, which will enable them to move beyond today's technology to meet the challenges of the future. GSwE2009 reinforces that recognition and preparation by specific outcomes identified in Section 3.

[9] *GSwE2009 will go beyond knowledge elements to offer significant guidance on individual curriculum components.*

GSwE2009 assembles knowledge elements into reasonable, easily implemented learning units. Articulating a set of well-defined curriculum units makes it easier for institutions to share pedagogical strategies and tools. It also provides a framework for publishing textbooks and other materials. However, GSwE2009 does not mandate a specific way of aggregating those learning units into courses. Nevertheless, the two companion documents to GSwE2009 describe example implementations that show possible ways of constructing courses that satisfy GSwE2009 recommendations.

[10] *GSwE2009 will identify the fundamental skills and knowledge that all graduates of a SwE master's degree program must possess.*

GSwE2009 defines 10 outcomes that all graduates should achieve. They range from the highly technical to "soft skills" around communication and ethics. Additionally, GSwE2009 defines a specific CBOK that every student should master by graduation. That knowledge contributes to the 10 outcomes. However, only students who go well beyond the content of the CBOK can achieve the 10 outcomes.

[11] *GSwE2009 will be based on a flexible curriculum architecture and on recognized bodies of knowledge. The latter will be modified and enhanced as required.*

The description of CBOK will be concise, appropriate for graduate education, and will use the work of previous studies on relevant bodies of knowledge and curricula, especially (Ardis and Ford, 1989), SWEBOK, SE2004, and INCOSE[22]. *A Guide to the Project Management Body of Knowledge (PMBOK® Guide)*[23] was originally considered as a primary source document. However, in developing the CBOK, the CAT recognized that the SWEBOK already incorporated much of the relevant material from the *PMBOK® Guide* and decided not to use the *PMBOK® Guide* directly as a primary reference.

As new studies emerge, they will be incorporated into subsequent versions of GSwE2009. For example, the IEEE Computer Society is now updating SWEBOK, and subsequent versions of GSwE2009 will incorporate those updates. Bodies of knowledge from related disciplines are included as appropriate, such as INCOSE's Systems Engineering Body of Knowledge[24] and especially the INCOSE Handbook[25].

[12] *GSwE2009 will honor individual program and student flexibility by limiting the common knowledge required for all students to no more than 50% of the total knowledge taught in a master's program.*

The CBOK is recommended for all graduate SwE degrees. That core knowledge has broad acceptance by the SwE education community and related communities. Despite the tendency to squeeze more and more into the required core, GSwE2009 must accommodate wide variations in program focus and individual student interest. No more than 50% of the

[22] Jain, R., and Verma, D., *A Report on Curriculum Content for a Graduate Program in Systems Engineering: A Proposed Framework*, INCOSE International Symposium, 2007.

[23] Project Management Institute, *A Guide to the Project Management Body of Knowledge (PMBOK® Guide)*, 3rd edition, 2004.

[24] INCOSE, *Guide to Systems Engineering Body of Knowledge (G2SEBoK)*, International Council on Systems Engineering, 2004.

[25] Haskins, C. (ed.), *INCOSE Systems Engineering Handbook*, Version 3.1, INCOSE-TP-2003-002-03.1, August 2007.

content of a master's program must be required in order to allow enough flexibility for universities and students.

[13] *GSwE2009 will be broadly based and international in scope.*

Despite the challenge that curricular requirements differ from country to country, GSwE2009 must be useful to SwE educators around the world. Where appropriate, every effort has been made to ensure that the curriculum recommendations are sensitive to national and cultural differences so that they are internationally applicable. The involvement by national computing societies and volunteers from all countries was actively sought and welcomed. Despite this effort, the clear majority of the GSwE2009 authors are from the United States. GSwERC 0.25 was sent for limited review to more than 100 reviewers, who were chosen for their leading roles in SwE education and, in some cases, because they reside outside the United States. Additional international authors were recruited for GSwERC 0.50 to help avoid a U.S.-centric view. Faculty from outside the U.S. contributed analyses of their programs with respect to recommendations. Those analyses are in *Comparisons of GSwE2009 to Current Master's Programs in Software Engineering.* Finally, a review workshop was held in Hyderabad, India in February 2009, bringing in more perspectives from outside the U.S.

[14] *GSwE2009 will include exposure to non-technical aspects of professional practice as an integral component of the graduate curriculum.*

The professional practice of SwE encompasses a wide range of non-technical issues and activities, including general problem solving, management, ethical and legal concerns, written and oral communication, working as part of a team, and recognizing the need for other expertise in a rapidly changing discipline. Those issues and activities are explicitly recognized in several GSwE2009 outcomes expected of all graduating students.

[15] *GSwE2009 will include discussions of strategies and tactics for implementation, along with high-level recommendations.*

Although it is important for GSwE2009 to articulate a broad vision of SwE education, the success of any real university curriculum depends heavily on implementation details. The companion volume, *Frequently Asked Questions on Implementing GSwE2009,* provides institutions with advice on the practical concerns of setting up a curriculum that satisfies GSwE2009 recommendations. That advice recognizes that academic institution and department visions and missions vary widely and may require different approaches to develop and maintain a graduate SwE curriculum. For example, programs may differ in student demographics, teaching/research/professional focus, delivery mechanisms, external constituents, infrastructure, regulations and accreditation issues, and many other program characteristics and constraints. The core knowledge required of all students and the implementation guidance accommodate such differences, including guidance on how programs might extend the core to incorporate institution-specific needs (e.g., focus on a particular applications domain or on particular types of students).

[16] *The distinction between SE2004 and GSwE2009 will be clear and apparent.*

Compared to SE2004, which are the IEEE Computer Society and ACM recommendations for undergraduate SwE curricula, GSwE2009 content is more narrowly focused on SwE and related disciplines. GSwE2009 expects much greater sophistication in student reasoning about SwE principles, and expects students to demonstrate their accumulated skills and knowledge in a more significant capstone experience (project, practicum, or thesis) than does SE2004. The courses, evaluations, and the capstone will generally be more demanding because GSwE2009 is a graduate curriculum—SE2004 is an undergraduate curriculum—and GSwE2009 assumes that students enter the program with at least *two years* of relevant software development experience.

The distinction between GSwE2009 and SE2004 is quite clear when they are viewed through the lens of Bloom's Taxonomy (see Appendix B for more information). SE2004 requires students to master topics at Bloom's Taxonomy levels 1, 2, or 3—*knowledge, comprehension,* or *application.* For several topics, such as *Requirements Analysis,* GSwE2009 recommends mastery at level 4—*analysis*, and for one topic area in which the student specializes it recommends level 5—*synthesis.* SE2004 recommends mastery of many topics at level 1. *Every* topic in GSwE2009 must be mastered at level 2 or higher. Moreover, many more topics in GSwE2009 require mastery at level 3 than does SE2004; e.g., in SE2004, the topic of *software process* is addressed only at levels 1 and 2. In GSwE2009, the same topic is covered at levels 2 and 3.

[17] *GSwE2009 will identify expected knowledge and experience for students to enter a master's program in software engineering.*

Undergraduate computing programs and industry experience in SwE vary greatly. To help institutions build programs that address the needs of the broad SwE community, GSwE2009 recommends minimum prerequisite knowledge and experience. This minimum prerequisite knowledge and experience is determined by the level of proficiency (described in Section 3) expected from students within the limited amount of time, study, and units allowed in a typical master's program. Students who enter a graduate program lacking those prerequisites will generally find it harder to succeed. Of course, universities may choose to offer leveling courses,[26] internships, and other means to help such students at the beginning of their graduate education. Such steps, while valuable, would clearly lengthen the time a student spends seeking a master's degree.

[26] A *leveling course* helps a student who does not have the expected proficiency level in a topic; e.g., a student who lacks the expected background in discrete mathematics could take a course to provide the requisite knowledge, skills, and abilities. Leveling courses are sometimes also called *preparatory* or *bridging courses.*

This page intentionally left blank.

3. Expected Outcomes When a Student Graduates

This section describes what students should be capable of when they graduate from a program that satisfies GSwE2009's recommendations.[27] It establishes the expected outcomes required for a professional practice. It specifies a mix of 10 technical, ethical, learning, and other outcomes, reflecting the diverse skills that graduates require in order to become successful as software engineers. The CBOK Outcome is perhaps the most clearly technical, being tied directly to mastery of the CBOK. As shown in *Comparisons of GSwE2009 to Current Master's Programs in Software Engineering*, many existing programs address this outcome fairly well. In contrast, the Ethics Outcome is among the least technical. It addresses mastery of ethical decision-making, which relatively few existing programs cover well. The curriculum authors deliberated at length about these outcomes to strike the right balance between technical and non-technical skills. Few existing programs cover all 10 outcomes well. That is, of course, not surprising. On the other hand, as articulated in *Comparisons of GSwE2009 to Current Master's Programs in Software Engineering*, over time existing programs should be able to reduce and even eliminate the gap between themselves and GSwE2009.

Several reviewers of GSwE2009 version 0.5 (called *GSwERC* version 0.5 at the time) recommended making these 10 outcomes more objectively testable. The authors weighed those recommendations carefully and provided some additional elaboration in the current version. However, the authors also felt that an extensive elaboration would be too limiting, since there are many ways to achieve these outcomes. Moreover, GSwE2009 is not intended to be used directly for accreditation. Instead, the companion volumes present guidance on achieving the outcomes and provide comparisons between GSwE2009 recommendations and actual master's programs.

The order in which the outcomes are listed is not important. It *does not* reflect a priority among the outcomes. Graduates of a master's program that satisfies GSwE2009 recommendations will:

CBOK *Master the Core Body of Knowledge*

The CBOK specifies a Bloom Taxonomy level for each included knowledge area (KA), subarea, topic, and subtopic. Mastering the CBOK requires learning principles exemplified through practice. A graduating student will have demonstrated that he or she can perform at the specified Bloom level, which ranges from *knowledge* (the lowest level) up through *analysis* (the fourth level). Such performance is the definition of *mastery* used herein. By way of comparison, the undergraduate reference curriculum, SE2004, only expects performance through the third level, *application.*

[27] These outcomes were significantly influenced by a report from Carnegie Mellon University: Shaw, M. (Ed.), *Software Engineering for the 21st Century: A Basis for Rethinking the Curriculum*, Technical Report CMU-ISRI-05-108, Carnegie Mellon University Institute for Software Research, March 2005.

GSwE2009 does not state how the demonstration of mastery will be performed. That decision is left up to the implementing university. However, the hypothetical implementations of GSwE2009 in *Frequently Asked Questions on Implementing GSwE2009* offer approaches for such demonstration. A student who has mastered the CBOK will be able to develop a modest-sized software system of a few thousand lines of code from scratch, be able to modify a pre-existing large-scale software system exceeding 1,000,000 lines of code, and be able to integrate third-party components that are themselves thousands of lines of code. Development and modification include analysis, design, and verification, and should yield high-quality artifacts, including the final software product.

DOMAIN *Master software engineering in one application domain, such as finance, medical, transportation, or telecommunications, and in one application type, such as real-time, embedded, safety-critical, or highly distributed systems. That mastery includes understanding how differences in domain and type manifest themselves in both the software itself and in its engineering, and includes understanding how to learn a new application domain or type.*

Only a student who enters a master's program already an expert or near expert in an application domain will depart that program as an expert. The Domain Outcome does not require a student to become a true expert in an application domain, an achievement that normally takes many years of experience and education. However, SwE only becomes tangible when practiced in an application domain, where software brings real value and where software engineers face, on a daily basis, the characteristics and peculiarities of that domain. Priorities, vocabulary, paradigms, technologies, tools, and a myriad of other factors vary from domain to domain; for example, security and privacy are typically extremely important in financial transactions, but less important in the embedded software of an automobile braking system. For the latter, safety is much more important. Development standards are very important in defense applications, but less important in software used to create special effects in movies. As a reference curriculum, GSwE2009 gives each program the flexibility to emphasize its defining characteristics. Nevertheless, depth in an application domain and application type requires knowing how to apply several of their significant tools and technologies. For example, someone gaining an in-depth understanding of Web applications in 2009 would almost certainly need to be able to use several common Web technologies, such as .Net or Java, and would need to understand at least one toolset for specifying, developing, integrating, modifying, and testing code using those technologies. Additionally, the student should appreciate the intersection of technology with the business/mission drivers of the domain.

There are practical limitations to which application domains an individual university can support in its classrooms. Faculty must be available who understand a domain, often through practice in the industry. Laboratories, case studies, and other artifacts must enable a student to explore a domain. Even large SwE programs will have trouble supporting more than a handful of domains well. Smaller programs might only be able to support one or two application domains.

Section 4 sets expectations for a student entering a master's program. One of those expectations is two years of practical software development experience. That experience is vital to enable mastery of an application domain. It is extremely hard to understand an application domain simply by classroom exercises and readings. Two years experience in an application domain—*any application domain*—will give the student an invaluable practical perspective that can be applied in graduate education to achieve the Domain Outcome.

DEPTH *Master at least one KA or sub-area from the CBOK to the Bloom Synthesis level.*

A student needs to dive deeply into at least one KA or sub-area, such as requirements or architecture. Such depth strengthens the student's analytic skills and enables the student to solve hard problems in at least one KA. This outcome is much more demanding than any in SE2004, the undergraduate SwE curriculum, which only requires mastery up to the *application* level in any topic.

ETHICS *Be able to make ethical professional decisions and practice ethical professional behavior.*

Professionals routinely face ethical, legal, and social dilemmas, such as when is it ethically, legally, and socially acceptable to compromise quality in order to meet schedule? What types of activities constitute a professional conflict of interest or are a breach of ethics, law, or social norms? In some cases, potential violations of the law are clear, but in most situations, there are no black and white rules for resolving such questions. Resolution requires maturity, experience, knowledge, and judgment. A graduate should have demonstrated that he or she has the maturity, knowledge, and judgment to make common professional decisions that have ethical, legal, and social implications. Two years of practical experience before entering the master's program will help enrich the student's ability to understand ethical dilemmas. In two years of work experience, it is quite possible the student will have faced the challenge of deciding whether to ship a product when serious bugs still remain, or whether to discount the views of an important stakeholder because it is "politically" difficult, or other such situations.

SYS ENG *Understand the relationship between software engineering and systems engineering and be able to apply systems engineering principles and practices in the engineering of software.*

As mentioned earlier, SwE and SE have much in common, but are often treated as quite separate disciplines. In some business domains, the term *systems engineer* is not used, instead being replaced by *application engineer, system architect, lead engineer, system analyst,* or many other terms. The student should be able to look past differences in terminology and see the relationship between software and SE, no matter what the latter is called.

Topics such as requirements analysis and architecture should be taught from a common systems/software perspective. For example, the architecture of any large system typically has much of its functionality implemented through a mix of hardware, software, and people. Software engineers should learn how to influence SE decisions to create the right mix for a particular application and should understand how to select the best software technologies to support that mix. The notions of *systems thinking* and *system dynamics*, popularized by people such as Jay Forrester[28], John Sterman[29], and Peter Senge[30], which stress understanding the relationship of the system as a whole to other systems, is an important aspect of SE that should be addressed.

TEAM *Be an effective member of a team, including teams that are multinational and geographically distributed, effectively communicate both orally and in writing, and lead in one area of project development, such as project management, requirements analysis, architecture, construction, or quality assurance.*

Students need to complete tasks that involve work as an individual, but also many other tasks that entail working with a group of individuals. For group work, students ought to be informed of the nature of groups and of group activities/roles as explicitly as possible. This must include an emphasis on the importance of such matters as a disciplined approach, the need to adhere to deadlines, how to communicate both orally and in writing, and how teams are evaluated as a whole rather than just as individuals. Students should have an appreciation of team dynamics and leadership techniques and be able to lead at least one area of project development. Increasingly, system and software development teams are assembled from many geographical sites, often across national boundaries. This

[28] Forrester, J., *Learning Through System Dynamics as Preparation for the 21st Century*, 1994.

[29] Sterman, J., *Business Dynamics: Systems Thinking and Modeling for a Complex World,* McGraw-Hill/Irwin, 2000.

[30] Senge, P., *The Fifth Discipline: The Art and Practice of the Learning Organization,* Broadway Business, 2006.

presents additional challenges of time, language, and culture that students must know how to address. For some programs, establishing geographically dispersed teams will be challenging, but this can be done where necessary by teaming with programs at other universities and campuses.

RECONCILE *Be able to reconcile conflicting project objectives, finding acceptable compromises within limitations of cost, time, knowledge, risk, existing systems, and organizations.*

Students should engage in realistic exercises that expose them to conflicting and changing requirements. For example, end users may prefer a new system that does not require significant change in existing business practice, while the business leaders may be looking to reengineer the business practice in order to realize dramatic gains in efficiency. If not managed well, such conflicts among key stakeholders can lead to requirements churn, product rejection, and many other undesirable consequences. The software engineer should understand techniques to address and resolve such conflicts. As another example, new requirements routinely emerge during the course of most large or complex projects. The graduate of a master's program should be able to reason about the implications of such emergence on technical planning and software architecture, among other considerations. Rigorous techniques to perform tradeoffs should be included as a way of resolving conflicts. Note also the tie between this outcome and the Ethics Outcome. The inability to reconcile conflicts well can lead to ethical dilemmas.

PERSPECTIVE *Understand and appreciate feasibility analysis, negotiation, and good communications with stakeholders in a typical software development environment, and perform those tasks well; have effective work habits and be a leader.*

The presence of a capstone experience, if it is a group project and not an individual activity (such as a thesis), is of considerable importance in this regard. It offers students the opportunity to tackle a major project and demonstrate their ability to bring together topics from a variety of courses and apply them effectively. This mechanism allows students to demonstrate their appreciation of the broad range of SwE topics and their ability to apply their skills to genuine effect. This should also include the ability to offer reflections on their achievements.

LEARN *Be able to learn new models, techniques, and technologies as they emerge, and appreciate the necessity of such continuing professional development.*

In a field as dynamic as SwE, life-long learning is essential to continued success. It is therefore imperative for the graduate student to develop the necessary skills to seek and learn the latest developments—to be able to grow personally. This includes the ability to evaluate and adapt software development processes, metrics, and tools, including the ability to create or assemble satisfactory evidence for that evaluation. A master's program cannot instill the desire for life-long learning, but can teach the skills to know how to do life-long learning.

TECH *Be able to analyze a current significant software technology, articulate its strengths and weaknesses, compare it to alternative technologies, and specify and promote improvements or extensions to that technology.*

Technologies change frequently. A software engineer must be able to select new technologies, understanding their limitations and appropriate uses: that is, to be able to perform tradeoff studies and act as a change agent within his or her professional organization. In 2009, such technologies might include service-oriented architectures and their supporting toolsets. In five years, there will be another set of technologies, just as controversial and complex. A graduate should know how to decide the relative merits of such technologies based on assembled or discovered evidence and be an effective advocate for "winning" technologies. Note, however, that in a university setting, a student will likely only be able to demonstrate their *potential* to be an effective advocate.

The Tech Outcome has a strong tie to the Domain Outcome. The strengths and weaknesses of a technology are generally not absolute, but vary with the application domain and other context.

It is useful to think beyond student competencies at graduation, looking out three to five years later toward the longer-term *objectives* that students might achieve. The CAT considered whether GSwE2009 should include a standard set of objectives that all master's programs should set for their graduates. After lengthy discussion, the CAT concluded no standard set existed. There is simply too much variation among individual master's programs to prescribe a common set in GSwE2009. Nevertheless, it is desirable for each university program to establish its own objectives.

4. Expected Student Background When Entering the Master's Program

One of the hardest decisions to make when constructing a graduate curriculum is determining what a student should be capable of when entering the program. If the bar is set too low, the graduate education will either be shallow or will need to be extended by universities to compensate for low entry capability. If the bar is set too high, too few will qualify and the program will struggle to attract students. According to a recent survey of graduate SwE programs, summarized in Appendix A, the average master's program in SwE requires around 33 to 36 graduate course credits, which in the United States is usually structured into 11 or 12 3-credit semester courses. GSwE2009 is in line with current practice, recommending a program of around 33 to 36 graduate credits. A full-time student could reasonably be expected to complete such a program in 18 to 24 months. The number of recommended credits, combined with expectations for student knowledge and skills when they graduate, determine what students should be capable of when they enter the master's program.

Establishing outcomes cannot be done without considering what students are capable of when entering the master's program. The survey of existing graduate programs described in Appendix A revealed a wide range of expectations for entering students. For example, some universities target students who are making a mid-career shift into being software engineers. For those universities, a student with a bachelor's degree and a "B" grade point average (GPA) is enough for entry. Other universities target students who are software professionals seeking to advance their career with an advanced degree in their current field. Most—but not all—universities presume a student can program in at least one computer language. About a third of the surveyed programs presume a student has professional experience as a software engineer. Nevertheless, GSwE2009 must make some assumptions about what students are capable of at entry or there is no basis for defining what knowledge they can reasonably master while pursuing a degree that is nominally the equivalent of 11 to 12 3-credit semester-long courses.

Note that *expectations* are not *admission requirements*. Individual universities and programs set admission requirements. However, deviations from these expectations may require lengthening the program to compensate and still achieve the 10 outcomes in Section 3. A student can compensate for the lack of a formal education by more extensive experience; a university can offer a student lacking certain knowledge or skills an opportunity to take additional *leveling*[31] courses; or a student lacking experience can take an internship or follow some other means to gain that experience while in the degree program. Of course, the latter two options will increase the number of courses that a student must take to earn a master's degree—a common practice for those entering a graduate program without the expected background. The curriculum architecture, described in the next section, provides a structure by which a university could

[31] Such courses go by many names. Students who lack proficiency in an expected competency normally take them. The objective is to raise their *level* of proficiency to that of their peers.

address students who do not meet the entry expectations. *Frequently Asked Questions on Implementing GSwE2009* provides advice on how to welcome students who lack strong computer science or software development backgrounds.

GSwE2009 presumes that an entering student meets *all* the following:

- The equivalent of an undergraduate degree in computing, or an undergraduate degree in an engineering or scientific field and a minor in computing. Table 1 in Section 6.2, Preparation Knowledge, defines the expected knowledge from the degree.
- The equivalent of an introductory course in SwE. Table 1 in Section 6.2 also defines the expected knowledge from this course.
- At least two years of practical experience in some aspect of SwE or software development. This experience should include participation in teams, development of a program or component that has been successfully delivered, and an update or repair to an existing program or component.

The rationale for these expectations is:

DEGREE Many existing master's programs in SwE expect students to have a bachelor's degree in an engineering or scientific field, but not a degree in computing. Such students generally bring much of the math skills and the ability to think analytically, both of which are essential to SwE. Students often have programming experience, although it is usually programming in the small without the benefit of understanding how to address issues associated with large or complex software.

In order to engineer software, a student must have mastered the fundamentals of computing, including programming, computer hardware, operating systems, data structures, algorithms, discrete math, and a design course that has considered developing a system in which a primary issue has been the integration of several components. Students who do not have at least a minor in computing will generally lack that mastery.

Universities frequently offer leveling courses to students who enter a master's program lacking the expected background in computing.

SWE The majority of master's programs in the 2007 survey of existing programs do not start students in an introductory SwE course. These programs assume that the student has learned the equivalent knowledge either from earlier coursework or from professional experience. GSwE2009 follows the practice of the majority of programs in that regard.

Universities frequently offer an undergraduate course introducing SwE to students who do not have the equivalent knowledge from a prior course or professional experience.

EXPERIENCE SwE is a practical field and it is a truism that there is no substitute for experience. The richness of the discussions in a graduate class and the sophistication of the analysis that students can perform are driven, in part, by the experience of those students. Students with at least two years of practical experience in several aspects of SwE or software development have a significantly deeper appreciation for the issues that are examined in the master's program. Such experience should expose the student to a team environment and to working on *several aspects* of development, as would happen when a student is part of a team modifying, testing, and releasing an existing application. Going through at least one full life cycle of a product release would be ideal. Two years experience in a single development activity, such as performing configuration management, would not support the spirit of this background expectation. The most germane experience would have the student (1) work on a component of a larger system that requires integration; (2) evolve an existing system, such as making it be backward-compatible with previous versions; and (3) address contextual requirements of customers.

Universities could offer internships to students lacking the expected experience, or otherwise involve them in a significant practical experience early in their master's program. However, it should be noted that several CAT members doubt whether an internship can truly compensate for a lack of relevant professional experience. Addition of such internships would probably increase the time required in the program.

This page intentionally left blank.

5. Curriculum Architecture

This section describes the structure of a curriculum into which courses satisfying GSwE2009 recommendations can be packaged. It identifies, via the CBOK, the minimal material that all programs should include and makes provisions for each institution to develop its own distinctive program(s). The curriculum architecture is similar to the one proposed in (Ardis and Ford, 1989) and is compatible with the existing master's programs for which course and curriculum data are described in Appendix A. It is intended to provide a structural basis for programs that deliver the outcomes described in Section 3.

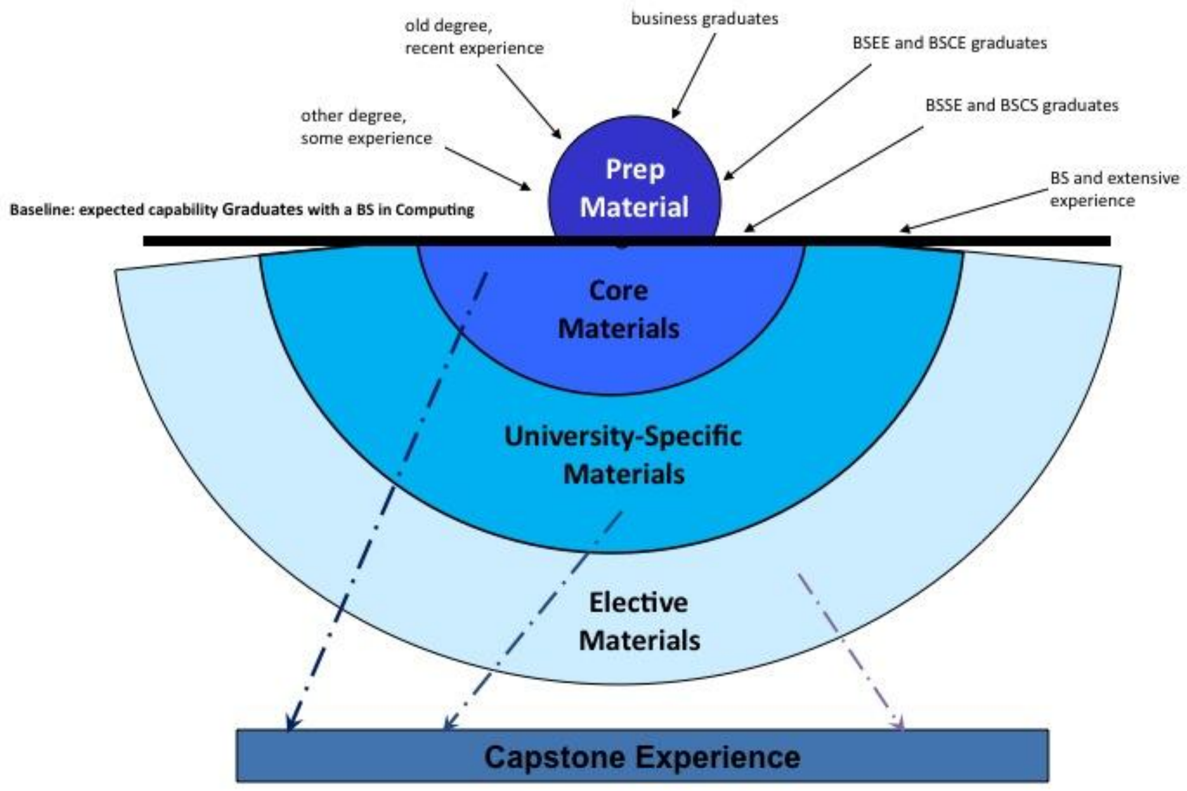


Figure 1. Architectural Structure of a GSwE2009 Master's Program

The curriculum architecture includes preparatory material, core materials, university-specific materials, elective materials, and a mandatory capstone experience. Figure 1 provides an overview of the curriculum architecture. The heavy black line represents the baseline expectations described in Section 4 for students entering the master's program. Thus, material above the heavy black line is mastered before entry into the master's program. Material below the heavy black line is mastered after program entry. A student who satisfies the baseline expectations is ready to begin the program (work *below* the heavy black line). Individual programs will determine how to prepare students whose background falls short. Typically, colleges and universities that wish to admit students who lack the expected background will provide preparatory courses containing materials that those students should take before entering the master's program. Those are the preparatory materials shown above the dark horizontal line in Figure 1.The more deficient the student's background is relative to the baseline entrance expectations, the higher the risk is that the student will not perform satisfactorily, harming both himself and fellow students. It is anticipated that a few students with undergraduate degrees in a variety of fields plus extensive experience, might enter directly into courses that cover only a subset of the core materials, and perhaps occasionally directly into courses that include university-specific and elective materials.

GSwE2009 identifies the fundamental skills and knowledge that all graduates of a master's program in SwE must possess. In Figure 1, this is captured in the half-circle area labeled *Core Materials*. These skills and knowledge include such topics as SE fundamentals, requirements engineering, software design, and ethics and professional conduct, which are listed in Section 6, CBOK. Where appropriate, it defines the common themes of the SwE discipline, including its dependencies on other related disciplines, such as SE, human factors for interface design and testing, and project management, and recommends that all graduate programs include this material. Courses that teach CBOK material would be mandatory or core courses, since taking them would be necessary to learn the core material. The CBOK has been limited to include no more than 50% of the total knowledge conveyed in a complete master's program.

The next half-circle in Figure 1, labeled *University-Specific Materials*, represents materials that an institution might include in order to tailor its program to meet its specific objectives. These will vary by institution or degree program. They may differ widely because of student demographics, teaching/research/professional focus, delivery mechanisms, external constituents, and infrastructure or accreditation issues. Institutions might include material in this part of the curriculum to extend a student's knowledge of their undergraduate field of study with particular emphases on tradeoffs between applications in those fields and the disciplines that are included in other portions of the SwE curriculum. For example, a program that emphasizes safety-critical systems might have a required course on such systems that would be part of the University-Specific Materials. An institution or program might refer to the core materials, as defined in this document, plus its university-specific materials, as its own core.

Elective Materials accommodate different interests of individual students, but may still reflect a program focus. For example, a program may focus on information security, verification and validation (V&V), or health-care systems, providing a series of courses that allow a student to gain depth in a technical area, CBOK KA, or an application domain, respectively. Those courses might be organized into *tracks* or may simply be an unstructured collection of courses. Students may be constrained in what electives they take to foster program educational goals, or the program may allow the student broad freedom in course selection. Elective materials can also include special topics courses that might be used from time to time to introduce experimental topics into the curriculum.

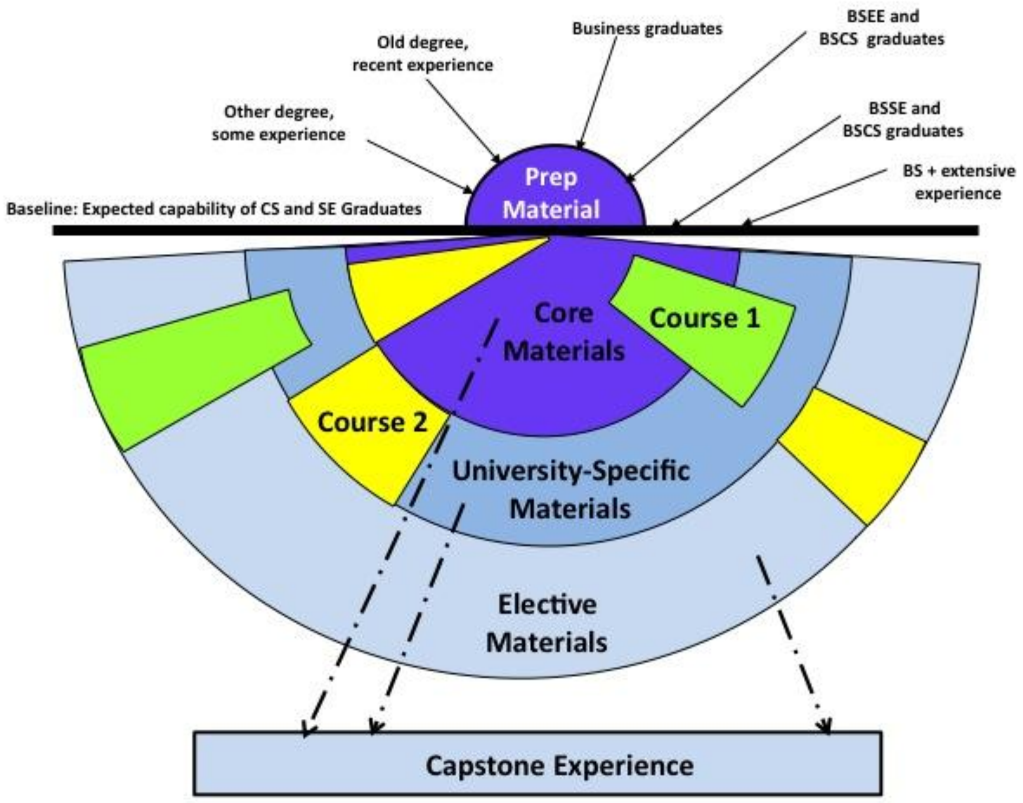

Old degree, recent experience
Business graduates
BSEE and BSCS graduates
Other degree, some experience
BSSE and BSCS graduates
BS + extensive experience
Baseline: Expected capability of CS and SE Graduates
Prep Material
Core Materials
Course 1
Course 2
University-Specific Materials
Elective Materials
Capstone Experience

Figure 2. Course Alignment, Which May or May Not Correspond to Specific Topics or Rings

GSwE2009 recommends that students demonstrate their accumulated skills and knowledge in a capstone experience, which might be a project, a practicum, or a thesis. The capstone experience would likely be between 3 to 6 credit hours, which would count towards the 33 to 36 total credit hours typically required for a master's degree. In this context, a project would be a practically oriented undertaking done by a single student or a group for or with someone within the offering institution. A practicum would be a software development project done for a real external customer by a group of students, perhaps for an employer for whom one or more of them work. A thesis would be SwE research completed by an individual student under the guidance of a research-oriented member of the faculty. Students completing the curriculum must be able to understand and appreciate the importance of negotiation, effective work habits, leadership, and good communication with stakeholders in a typical software development environment. These topics should be integrated into the core materials and perhaps could be reinforced in the university-specific or elective materials. However, the presence of a capstone project, a practicum, or a thesis at the end of the curriculum is of considerable importance in this regard. It offers students the opportunity to tackle a major undertaking and demonstrate their ability to bring together topics from a variety of courses and apply them effectively, as shown by the broken lines connecting the capstone experience back to the materials contained in the various layers of the curriculum.

There is no intent in this architectural specification to require either the content of preparatory courses or the content in core courses to be self-contained in courses with names corresponding to the topics. Figure 2 provides an example showing how this might happen. The yellow wedges in this figure correspond to courses that teach precisely core material, precisely university-specific material or precisely elective material. The green wedges represent courses that integrate material across architectural layers. For example, *Course 1* (shown in green) covers a combination of core and university-specific material. *Course 2* (shown in yellow) covers only university-specific materials. Either, or both, methods of course packaging are appropriate.

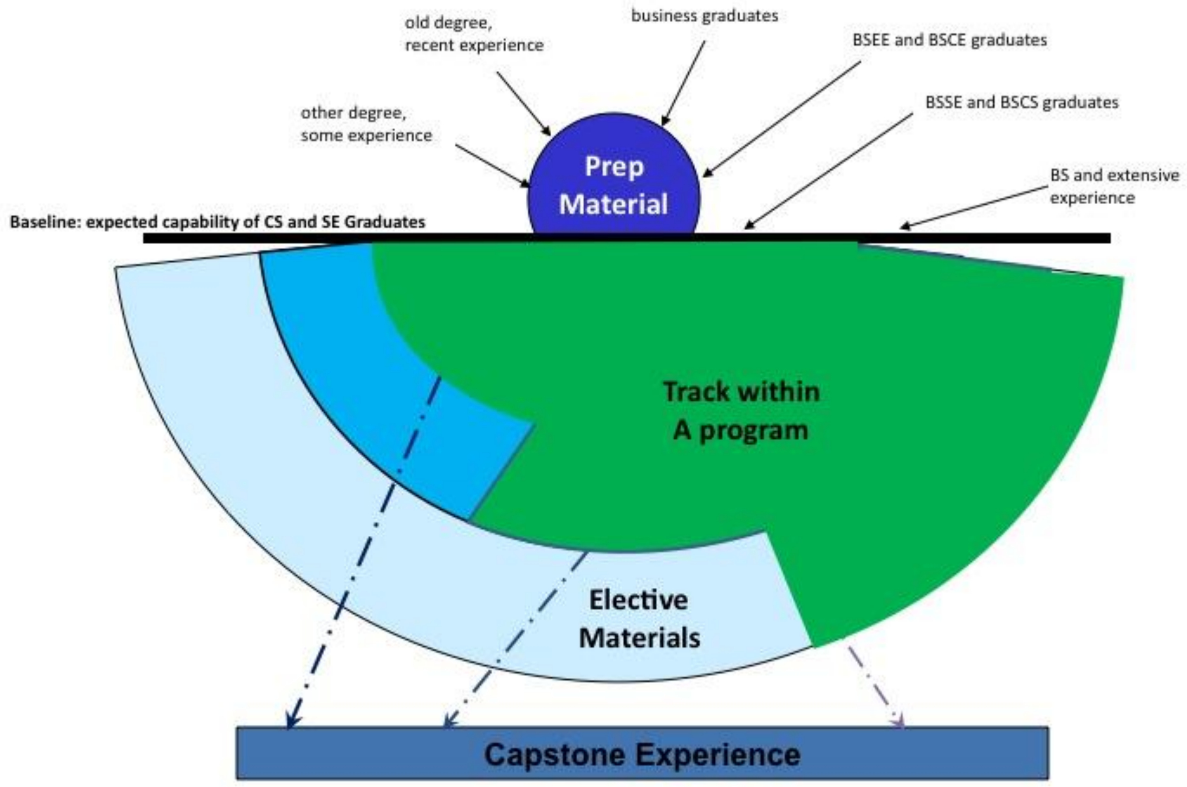

old degree, recent experience
business graduates
BSEE and BSCE graduates
other degree, some experience
BSSE and BSCS graduates
Prep Material
BS and extensive experience
Baseline: expected capability of CS and SE Graduates
Track within A program
Elective Materials
Capstone Experience

Figure 3. Demonstration of How a Specific Track May Fit Within a Program

There is also no intent that all of the courses containing preparatory or core materials must be completed before coursework in the next ring can begin. It is anticipated that the sequencing of courses will be controlled primarily by the prerequisite specifications of each course in a specific institution's curriculum.

Figure 3 offers an example of how a *track* could be constructed within this architectural framework. In this example, the track would include all of the core materials, some university-specific materials from the track, some elective materials related to the track and a capstone experience concentrating on a topic associated with the specific track. Tracks are typically areas of study, such as telecommunications, real-time systems, and information systems.

It is through a combination of Core, University-Specific, and Elective Materials that the 10 outcomes in Section 3 are met. For example, the Domain Outcome requires depth in an application domain, such as telecommunications or finance. A program could offer a track that gives a student depth in telecommunications by emphasizing telecommunications examples in a software architecture class that teaches Core Materials, and that teaches telecommunications principles in an elective course on the general telecommunications field.

This page intentionally left blank.

6. Core Body of Knowledge (CBOK)

6.1 Development of the CBOK

The primary source for developing the CBOK was the SWEBOK. Knowledge elements were also derived from SE2004, (INCOSE, 2003) and especially (Haskins, 2007). In the study and analysis of these sources, it was decided that although the SWEBOK organization and content would dominate, various changes in areas and topics were needed to support the GSwE2009 expected student outcomes and to accommodate the needs and views of academia, industry, and the computing professional societies. For example, two KAs, not in the current version of the SWEBOK, were added: Systems Engineering Fundamentals, and Ethics and Professional Conduct. In addition, some units and topics were added, rearranged or modified. These included:

- Addition of Human Computer Interface design in the Software Design KA
- Addition of an Engineering Economics unit in the Software Engineering Management KA
- Addition of a Risk Management unit in the Software Engineering Management KA
- Addition of a Verification and Validation (V&V) unit in the Software Quality KA
- Changes in the names and the unit/topic organization in three KAs: (a) Software Requirements to Requirements Engineering, (b) Software Testing to Testing and (c) Software Configuration Management to Configuration Management. These changes were made to accommodate and emphasize the role of SE in GSwE2009.

It should be noted that as of the publication date of GSwE2009, the plans for a 2010 refresh of SWEBOK call for a new KA on Professional Practice and four new education KAs: Engineering Economy Foundations, Computing Foundations, Mathematical Foundations, and Engineering Foundations. GSwE2009 has attempted to accommodate the SWEBOK refresh by including these topics in the preparation knowledge (discussed in the next section) and in the additional KAs and units in the CBOK.

Two other proposals for significant re-organization of the SWEBOK KAs were considered:

- Create a KA called Supporting Processes that includes configuration management, V&V, quality assurance, reviews and audits, and software documentation process. This proposal also included recommended changes in the Software Engineering Management area involving units on organizing, staffing, and directing a software project.

- Create a KA called V&V that subsumes the Software Testing KA and includes units from the Software Quality area.

Although both proposals were viewed positively, it was felt that the wide recognition and the common understanding of the organization of the SWEBOK KAs were compelling reasons to maintain the basic SWEBOK outline as a foundation for the GSwE2009 CBOK. However, the first proposal did prompt study, analysis, and modification of the description of some of the knowledge units within the Software Engineering Management KA (in Project Organization and Enactment and in Risk Management).

The CAT has provided a recommended level to which a student should achieve each KA; these are defined in terms of Bloom's taxonomy. Appendix B describes Bloom's cognitive levels[32] and the process used to specify the student cognitive level for both the prerequisite KAs and the CBOK KAs. The following level designations are used in the tables in this section:

- Knowledge (K)
- Comprehension (C)
- Application (AP)
- Analysis (AN)

These level designations are not intended to guide detailed curriculum design, but rather to provide a high-level view of curriculum and student expectations. Students admitted to a program who possess substantial SwE education (e.g., a Bachelor of Science degree in SwE) or experience (e.g., an experienced software project manager) will arrive with knowledge at or above some of the designated Bloom's levels. A university might choose to exempt such students from some of its required courses, giving them the opportunity to take a greater number of advanced courses than afforded the typical student.

6.2 Preparation Knowledge

Table 1 specifies the knowledge students should possess when entering a master's program in order to be best prepared to achieve the GSwE2009 outcomes. SE2004 was the primary source for the knowledge elements. The knowledge may be acquired through undergraduate study, from software development experience, through leveling courses offered by an institution, or through some combination of these. The table is organized hierarchically into three levels, similar to the knowledge organization in the SE2004. The highest level of the hierarchy is the *KA*, such as Mathematical Fundamentals. Each KA is shown in blue and is broken down into smaller numbered divisions called *units,* which represent individual thematic modules within an area. Each unit is further subdivided into an unordered set of *topics*.

Clearly, other preparation knowledge will be needed to support graduate SwE education. For example, students entering a master's program should have a strong background in general education: excellent oral and written communication skills, knowledge of the social sciences, and a solid foundation in continuous mathematics (algebra, pre-calculus, and calculus).

[32] Bloom, B.S. (Ed.), *Taxonomy of educational objectives: The classification of educational goals: Handbook I, cognitive domain,* Longmans, 1956.

Table 1. Preparation Knowledge for Core Body of Knowledge

Knowledge Areas	Bloom Level
Mathematics Fundamentals	
1. Discrete Structures Functions, relations, and sets; basic logic; proof techniques; basics of counting; graphs and trees; discrete probability	**AP**
2. Propositional and Predicate Logic Propositions, operators, and truth tables, laws of logic, predicates and quantifiers, argument and inference	**AP**
3. Probability and Statistics Basic probability theory, random variables and probability distributions, estimation theory, hypothesis testing, regression analysis, analysis of variance	**AP**
Computing Fundamentals	
1. Programming Fundamentals Overview of programming languages; virtual machines; introduction to language translation; declaration and types; abstraction mechanisms; object-oriented programming; functional programming; language translation systems; type systems; programming language semantics; programming language design	**AP**
2. Data Structures and Algorithms Basic algorithmic analysis; algorithmic strategies; fundamentals of computing algorithms; distributed algorithms	**C**
3. Computer Architecture Digital logic and digital systems; machine level representation of data; assembly level machine organization; memory system organization and architecture; interfacing and communication; functional organization; multiprocessing and alternative architectures; performance enhancements; architecture for networks and distributed systems	**C**
4. Operating Systems Operating system overview and principles; concurrency; scheduling and dispatch; memory management; device management; security and protection; file systems; real-time and embedded systems; fault tolerance; system performance evaluation; scripting	**C**
5. Networks and Communications Introduction to net-centric computing; communication and networking; network security; Internet; building Web applications; network management; compression and decompression; multimedia data technologies; wireless and mobile computing	**C**
6. Module Design and Construction Abstraction, information hiding, interface design, procedural design, assertions, exceptions, coupling and cohesion	**AP**
Software Engineering	
1. Software Requirements Software requirements fundamentals; requirements elicitation; requirements analysis; requirements specification; requirements validation	**C**
2. Software Design	**C**

Knowledge Areas	Bloom Level
Software design fundamentals; software structure and architecture; software design notations; software design strategies and methods	
3. Software Construction Software construction fundamentals; software construction practices	**AP**
4. Software Testing Software testing fundamentals; test levels; test techniques	**K**
5. Software Maintenance Software maintenance fundamentals; techniques for maintenance	**K**
6. Software Engineering Management Software project planning; software configuration management	**K**
7. Software Engineering Process Process definition and implementation; product and process measurement	**K**
8. Software Quality Software quality fundamentals; software quality management practices	**K**

6.3 CBOK Concepts and Organization

Table 2 presents the outline of the CBOK that is recommended for the core of a curriculum that supports the GSwE2009 recommendations. It is organized hierarchically in the same manner as Table 1. The CBOK knowledge units and their Bloom level designations were developed in such a way that the core could be covered in the equivalent of approximately 15 credit hours or approximately 200 contact hours (using a North American academic model). The core is designed to comprise a little less than 50% of the total credit hours recommended for a master's degree. Hence, additional time and courses can be allocated to provide additional depth in the core areas (at higher Bloom levels) and to focus on a chosen application domain. An actual workload measure (such as that used in the European Commission's European Credit Transfer System[33]) could have been used, but it was felt that contact hours were sufficient for the intended level of this curriculum guidance.

Table 2. Core Body of Knowledge

Knowledge Area	Systems Eng. Content	Bloom Level
A. Ethics and Professional Conduct		
1. Social, legal, and historical issues	SYS	C
Data confidentiality and security, surveillance and privacy		

[33] European Commission, Education & Training, "European Credit Transfer and Accumulation System (ECTS)" website. http://ec.europa.eu/education/programmes/socrates/ects/index_en.html#1

Knowledge Area	Systems Eng. Content	Bloom Level
Historical developments, and gender, minor, and cultural issues		
Contracts and liability, intellectual property, freedom of information		
Computer crime and law enforcement		
2. Codes of ethics and professional conduct	SYS	C/AP
Responsibilities to society		
Models for professionalism, professional societies		
Codes of ethics and practice		
3. The nature and role of software engineering standards		C
Nature and role of standards		
International standards, standards and harmonization organizations		
Bodies of knowledge, accepted and best practices		
B. System Engineering	**SYS**	
1. Systems Engineering Concepts		C
System context		
People and systems		
System hierarchical relationships		
The role of system engineers		
2. System Engineering Life Cycle Management		C
Lifecycle Management		
Systems engineering and software engineering processes		
3. Requirements		C/AP
Stakeholder requirements		
Requirements analysis		
4. System Design		C/AP
Architectural design		
Implementation		
Trade studies		
5. Integration and Verification		C
6. Transition and Validation		C
7. Operation, Maintenance and Support		C
C. Requirements Engineering	**SYS**	
1. Fundamentals of Requirements Engineering		C/AP
Relationship between systems engineering and software engineering		
Definition of requirements		
System design constraints		

Knowledge Area	Systems Eng. Content	Bloom Level
System design and requirements allocation		
Product and process requirements		
Functional and non-functional requirements		
Emergent properties		
Quantifiable requirements		
2. Requirements Engineering Process		C
Process models		
Process actors		
Process support and management		
Process quality and improvement		
3. Initiation and Scope Definition		AP
Determination and negotiation of requirements		
Feasibility analysis		
Process for requirements review/revision		
4. Requirements Elicitation		AP
Requirements sources		
Elicitation techniques		
5. Requirements Analysis		AN
Requirements classification		
Conceptual modeling		
Heuristic methods		
Formal methods		
Requirements negotiation		
6. Requirements Specification		AP
Requirements specification techniques		
7. Requirements Validation		AP
Requirements reviews		
Prototyping		
Model validation		
Acceptance tests		
8. Practical Considerations		C/AP
Iterative nature of requirements process		
Change management		
Requirements attributes		
Requirements tracing		
Measuring requirements		
D. Software Design		
1. Software Design Fundamentals		C/AP

Knowledge Area	Systems Eng. Content	Bloom Level
General design concepts		
Context of software design		
Software design process		
Enabling techniques		
2. Key Issues in Software Design		AP
Concurrency		
Control and handling of events		
Distribution of components		
Error and exception handling and fault tolerance		
Interaction and presentation		
Data persistence		
3. Software Structure and Architecture		AP/AN
Architectural structures and viewpoints		
Architectural styles (macro architectural patterns)		
Design patterns (micro architectural patterns)		
Human computer interface design		
Families of programs and frameworks		
4. Software Design Quality Analysis and Evaluation		AP
Quality attributes		
Quality analysis and evaluation techniques		
Measures		
5. Software Design Notations		AP
Structural descriptions (static)		
Behavioral descriptions (dynamic)		
6. Software Design Strategies and Methods		AP/AN
General strategies		
Function-oriented (structured) design		
Object-oriented design		
Heuristic methods		
Formal methods		
Component-based design (CBD)		
E. Software Construction		
1. Software Construction Fundamentals		AP
Minimizing complexity		
Anticipating change		
Constructing for verification		
Standards in construction		
2. Managing Construction		AP

Knowledge Area	Systems Eng. Content	Bloom Level
Construction methods		
Construction planning		
Construction measurement		
3. Practical Considerations		AP
Construction design		
Coding		
Construction testing		
Construction quality		
Integration		
F. Testing	**SYS**	
1. Testing Fundamentals		AP
System testing and software testing		
Testing-related terminology		
Key issues		
Relationships of testing to other activities		
2. Test Levels		AP
The target of the tests		
Objectives of testing		
Component testing		
Integration testing		
System testing		
Acceptance testing		
3. Testing Techniques		AP
Based on tester’s intuition and experience		
Specification-based		
Code-based		
Fault-based		
Usage-based		
Based on nature of application		
Selecting and combining techniques		
4. Test-Related Measures		AP/AN
Evaluation of the program or system under test		
Evaluation of the tests performed		
5. Test process		C/AP
Management concerns		
Test activities		
G. Software Maintenance		
1. Software Maintenance Fundamentals		C

Knowledge Area	Systems Eng. Content	Bloom Level
Definitions and terminology		
Nature of maintenance		
Need for maintenance		
Majority of maintenance costs		
Evolution of software		
Categories of maintenance		
2. Key Issues in Software Maintenance		AP
Technical		
- Limited understanding		
- Testing		
- Impact analysis		
- Maintainability		
Management issues		
- Alignment with organizational issues		
Maintenance cost estimation		
- Cost estimation		
- Parametric models		
- Experience		
Software maintenance measurement		
3. Maintenance Process		AP
Maintenance process models		
Maintenance activities		
- Unique activities		
- Supporting activities		
4. Techniques for Maintenance		AP
Program comprehension		
Reengineering		
Reverse engineering		
H. Configuration Management (CM)	SYS	
1. Management of the CM Process		C/AP
Organizational context for CM		
Constraints and guidance for CM		
Planning for CM		
- CM organization and responsibilities		
- CM resources and schedules		
- Vendor/subcontractor control		
- Interface control		
Configuration management plan		

Knowledge Area	Systems Eng. Content	Bloom Level
Surveillance of configuration management		
- CM measures and measurement		
- In-process audits of CM		
2. Configuration Identification		AP
Identifying items to be controlled		
- Configuration items		
- Configuration item relationships		
- Versions		
- Baseline		
-Acquiring configuration items		
Software library		
3. Configuration Control		AP
Requesting, evaluating and approving changes		
- Configuration control board		
- Change request process		
Implementing changes		
Deviations and waivers		
4. Configuration Status Accounting		
Configuration status reporting		
5. Software Release Management and Delivery		AP
Software building		
Software release management		
I. Software Engineering Management		
1. Software Project Planning		AP
Project goals and objectives		
Project policies and standards		
Process planning		
Project assumptions and forecasts		
Project deliverables		
Project staffing		
Effort, schedule, and cost estimation		
Resource allocation		
Quality management		
Project plan/budget development and management		
2. Risk Management	SYS	AP
Risk management concepts		
- Probability, impact		
- Timeframe		

Knowledge Area	Systems Eng. Content	Bloom Level
Risk management process		
- Frameworks, standards, and guidelines		
- Risk identification, analysis and risk prioritization techniques		
- Risk mitigation strategies		
Risk management tools		
- Earned value tracking		
- Technical performance measurement		
- Defect tracking and reporting		
- Project control panels		
Organizational risk management		
Joint supplier/customer risk management		
3. Software Project Organization and Enactment		AP
Project organization		
- Identify and group project functions, activities, and tasks - Determine organizational structure and positions - Define responsibilities, authority relationships, position qualifications		
Project directing		
- Leadership, supervision, delegation of authority, coordination and communication		
- Motivation, conflict resolution, team building		
Project control		
- Implementation of plans, and measurement process		
-Process monitoring		
- Change management		
Reporting		
Supplier contract management (e.g., RFP, cost evaluation, IP rights)		
4. Review and Evaluation		C
Determining satisfaction of requirements		
Reviewing and evaluating performance		
5. Closure	SYS	C
Determining closure		
Closure activities		
6. Software Engineering Measurement		AP
Establish and sustain measurement commitment		
Plan the measurement process		
Perform the measurement process		
Evaluate measurement		
7. Engineering Economics	SYS	C

Knowledge Area	**Systems Eng. Content**	**Bloom Level**
Engineering economics fundamentals		
For-profit decision-making		
Not-for-profit decision-making		
Present economy		
Estimation, risk, and uncertainty		
Multiple attribute decisions		
J. Software Engineering Process		
1. Process Implementation and Change		C/AP
Process infrastructure		
- Software engineering process group		
- Experience factory		
Activities		
Models for process implementation and change		
Practical considerations		
2. Process Definition		C
Life cycle models		
Software life cycle processes		
Notations for process definitions		
Process adaptation		
Automation		
3. Process Assessment		AP
Process assessment models		
Process assessment methods		
4. Product and Process Measurement		AP
Software process measurement		
Software product measurement		
- Size measurement		
- Structure measurement		
- Quality measurement		
Quality of measurement results		
Measurement techniques		
- Analytical techniques	SYS	
- Benchmarking techniques	SYS	
K. Software Quality		
1. Software Quality Fundamentals		AP
Software engineering culture and ethics		
Value and costs of quality	SYS	
Quality models and characteristics	SYS	

Knowledge Area	Systems Eng. Content	Bloom Level
- Software process quality		
- Software product quality		
Quality improvement	SYS	
Application quality requirements	SYS	
- Criticality of systems		
- Dependability		
- Integrity levels of software		
Defect characterization		
2. Software Quality Management Processes		AP
Software quality assurance		
Software quality management techniques		
- Static techniques		
- People-intensive techniques		
- Analytic techniques		
- Dynamic techniques		
Software quality measurement		
3. Verification and Validation (V&V)	SYS	AP
Definitions of V&V		
- System V&V and software V&V		
- Independent V&V		
V&V Techniques		
- Testing		
- Demonstrations		
- Traceability		
- Analysis		
- Inspections		
- Peer reviews		
- Walkthroughs		
- Audits		

Figure 4 depicts the percentages of the curriculum that are recommended for each core KA. These percentages were initially determined by using a quasi Wideband Delphi technique to allocate the 200 contact hours, and then the hours were converted to percentages (of the 50% core) and adjusted to ranges of approximately 1%-2%. As indicated in Figure 4, the percentages for each area apply only to the core, which represents approximately 50% of the curriculum. The percentages should be considered as general high-level guidance, not as precise curriculum specification.

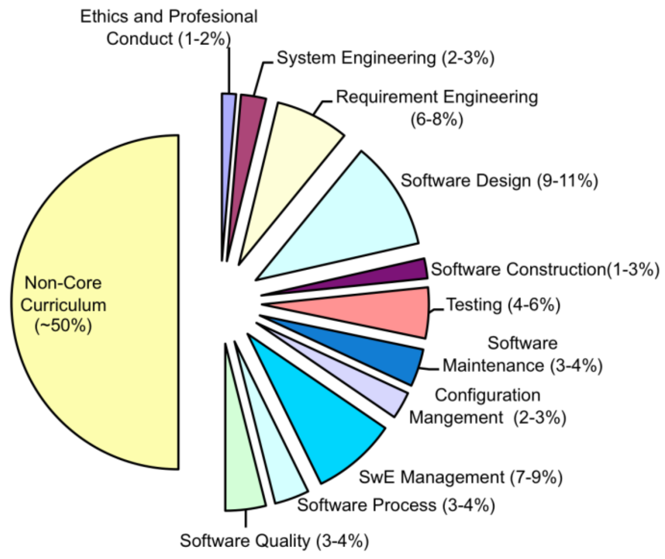


Figure 4. Percentage Devoted to Core Body of Knowledge Areas

As indicated in Section 5, Curriculum Architecture, the university-specific and elective materials will cover many of these KAs in more depth and may cover material outside these KAs completely, such as the study of a specific application domain.

As also explained in Section 5, Curriculum Architecture, the core 15 credit hours could be distributed in many ways. The simplest and most direct way would be as a set of courses dedicated specifically to teaching core material. Using the typical North American model, the entire set of core materials would be taught in five 3-credit semester courses. Alternatively, a program could cover the core material over many more courses. *Comparisons of GSwE2009 to Current Master's Programs in Software Engineering* examines how various universities approximate the coverage of the core material in their programs.

The KAs outlined in Table 2 are intended to characterize the core content of a master's program in SwE; it is not intended to depict or to imply the organization of curricula and courses. Although there are KAs on requirements, design, construction, and testing, this should not be taken to mean that GSwE2009 is recommending a waterfall curriculum: that is, first a course in requirements, then a course on design, and so on. Instead, GSwE2009 supports and encourages a variety of curriculum designs and course organizations that satisfy the GSwE2009 guidelines.

Table 2 provides topic-level outline of the CBOK. For most areas, units, and topics, the SWEBOK provides more in-depth description of the CBOK elements. Appendix C contains more detailed descriptions of three CBOK elements that are not covered sufficiently or at all in the SWEBOK:

- Systems Engineering Fundamentals
- Ethics and Professional Conduct
- Engineering Economics

Both the CBOK and Appendix C should be viewed as extensions of the SWEBOK. They provide depth and detail that support the design and specification of courses and curricula. Please note that Appendix C is not intended to directly influence future versions of SWEBOK.

Appendix E provides a mapping of the ten GSwE2009 Outcomes to the CBOK. The mapping shows where a curriculum, depending on CBOK alone, would fall short of achieving the outcomes, highlighting the importance of the 50% of the curriculum that is not covered by the CBOK.

6.4 Crosscutting Knowledge Elements

One of the concerns with using a hierarchical model for organization of knowledge is that KAs and their units may be incorrectly interpreted as independent of each other. Such misinterpretation can lead to two problems. One problem, highlighted in the previous section, is to view the "development" KAs (requirements engineering, software design, software constructions, testing, and maintenance) as dictating a partitioning of the curriculum into a waterfall model. Similarly, the presence of a KA such as Software Maintenance might lead to the mistaken view that the other development KAs are independent of maintenance and are focused only on new development. A reading of the SWEBOK description of the Software Maintenance KA makes it clear that it is dependent on the other KAs: "this KA description is linked to all other chapters of the Guide."

A related danger with a hierarchical model is that the "crosscutting" nature of certain CBOK elements will be obscured. For example, one might overlook the importance of the "supporting" KAs (process, quality, management, and ethics) as elements that impinge on and relate to the other KAs: for example, the importance of the distribution of software process throughout the development KAs—that is, process cuts across requirements, design, construction, testing, and maintenance.

The next section discusses the crosscutting nature of SE and how the CBOK addresses this. During the development and review of the CBOK, a number of other crosscutting knowledge elements were considered—software security, software safety, software reuse, and human factors and usability. Although the CAT agrees with the importance of these crosscutting elements, it does not feel they needed additional representation within the CBOK, unlike the case of SE. The

SWEBOK refers to each of these elements throughout various KAs (requirements, design, construction, testing, etc.). To illustrate this, Appendix D discusses how software security is related to the SWEBOK and lists KAs and units that support the inclusion of security-related issues in a GSwE2009 curriculum. In addition, the SWEBOK and this document include references to key documents that provide a foundation for providing depth in a topic beyond the CBOK. For example, *Software Assurance: A Curriculum Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software*[34] edited by Sam Redwine, provides a comprehensive view of security throughout the software life cycle and includes sections on the following activities: Ethics, Secure Software Requirements, Secure Software Design, Secure Software Construction, Security V&V, and other practices that span the lifecycle. In effect, (Redwine, 2007) provides a body of knowledge for software security that supports a GSwE2009 curriculum that focuses on the development of secure software systems.

Another missing KA that received considerable comment was *human factors* (ergonomics, human computer interaction, cognitive science, etc.). In the SWEBOK, human factors knowledge (listed as Software Ergonomics) is designated as a related discipline, which means that it is a separate discipline that is important to the development of software products, but its depth and breadth of knowledge is not part of the "generally accepted" knowledge that all software engineers should possess. Specialists might be needed for a software application strongly dependent on human factor issues. This is the position taken by GSwE2009. Of course, a program might use some part of the 50% of the curriculum that goes beyond the core to focus on human factors.

6.5 Systems Engineering Issues

A critical feature of GSwE2009 is the increasing importance of SE to professional SwE education.

6.5.1 Systems Engineering and Software Engineering

Several trends have caused systems engineering and SwE to initially evolve as largely sequential and independent processes. First, SE began as a discipline for determining how best to configure various hardware components into physical systems such as ships, railroads, or telecommunications systems. Once the systems were configured and their component functional and interface requirements were precisely specified, sequential external or internal contracts could be defined for producing the components. When software components began to appear in such systems, the natural thing to do was to treat them sequentially and independently as *computer software configuration items*.

[34] Redwine, S. T. Jr. (Ed.), *Software Assurance: A Curriculum Guide to the Common Body of Knowledge to Produce, Acquire and Sustain Secure Software*, Draft Version 1.2, U.S. Department of Homeland Security, 2007.

Second, the early history of SwE was influenced by a highly formal and mathematical approach to specifying software components, and a reductionist approach to deriving computer software programs that correctly implemented the formal specifications. A "separation of concerns" was practiced, in which the responsibility for producing formalizable software requirements was left to others, most often hardware-oriented systems engineers. Some example quotes illustrating this approach are:

- "The notion of 'user' cannot be precisely defined, and therefore has no place in computer science or software engineering."[35]
- "Analysis and allocation of the system requirements is not the responsibility of the software engineering group but is a prerequisite for their work."[36]

As a result, a generation of SwE education and process improvement goals were focused on reductionist software development practices that assumed that other (mostly non-software people) would furnish appropriate predetermined requirements for the software.

Third, the business practices of contracting for components were well worked out. Particularly in the government sector, acquisition regulations, specifications, and standards were in place and have been traditionally difficult to change. The path of least resistance was to follow a "purchasing agent" metaphor and sequentially specify requirements, establish contracts, formulate and implement solutions, and use the requirements to acceptance-test the solutions.[37,38] When requirements and solutions were not well understood or were changing rapidly, knowledgeable systems and software engineers and organizations could reinterpret the standards to operate more flexibly, concurrently, and pragmatically, and to produce satisfactory systems.[39, 40] But all too frequently, the sequential path of least resistance was followed, leading to the delivery of obsolete or poorly-performing systems.

As the pace of change increased and systems became more user-intensive and software-intensive, serious strains were put on the sequential approach. First, it was increasingly appreciated that the requirements for user-intensive systems generally could not be specified in advance, but emerged with use. This undermined the fundamental assumption of sequential specification and implementation.

---

[35] Dijkstra, E., "Software Engineering: As It Should Be", conference paper, International Conference on Software Engineering 4, September 1979, 442-448. See also EWD 791 at http://www.cs.utexas/users/EWD.

[36] Paulk, M., et al., *Software Capability Maturity Model*, Version 1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1993.

[37] U.S. Department of Defense, *MIL-STD-1521B: Technical Reviews and Audits for Systems, Equipments, and Computer Software,* 1985.

[38] U.S. Department of Defense, *DOD-STD-2167A: Defense System Software Development,* 1988.

[39] Checkland, P., *Systems Thinking, Systems Practice* (2nd ed.), Wiley, 1999.

[40] Royce, W. E., *Software Project Management: A Unified Framework,* Addison-Wesley Professional, 1998.

Second, having people without software experience determine the software specifications often made the software much harder to produce, putting software even more prominently on the system development's critical path. Systems engineers without software experience would minimize computer speed and storage costs and capacities, which caused software costs to escalate rapidly.[41] They would choose best-of-breed system components with software that was incompatible and time-consuming to integrate. They would assume that adding more resources would speed up turnaround time or software delivery schedules, not being aware of slowdown phenomena such as multiprocessor overhead[52] or Brooks' Law (adding more people to a late software project will make it later).[42] The top five critical success factors distinguishing successful from failed software projects in the 2005 Standish Report[43] were primarily in the SE area (lack of user involvement, executive support, clear requirements, proper planning, and realistic expectations), accounting for 71% of the sources of failure.

Third, software people were recognizing that their sequential, reductionist processes were not conducive to producing user-satisfactory software, and were developing alternative SwE processes (evolutionary, spiral, agile) involving more and more SE activities. Concurrently, SE people were coming to similar conclusions about their sequential, reductionist processes, and developing alternative "soft SE" processes, emphasizing the continuous learning aspects of developing successful user-intensive systems. Similarly, the project management field was undergoing questioning about its underlying specification-planning-execution-control theory being obsolete and needing more emphasis on adaptation and value generation.[44]

Many commercial organizations have developed more flexible and concurrent development processes.[45] Also, recent process guidelines and standards such as the Capability Maturity Model Integrated (CMMI)[46], IEEE/EIA Standard 15288-2008, ISO/IEC 12207 for SwE[47], and ISO/IEC

---

[41] Boehm, B., *Software Engineering Economics*, Prentice Hall, 1981.

[42] Brooks, F., *The Mythical Man-Month: Essays on Software Engineering* (2nd ed.), Addison-Wesley Professional, 1995.

[43] Standish Group, Unfinished Voyages, http://www.standishgroup.com/sample_research/unfinished_voyages_1.php, October 19, 2005.

[44] Koskela, L., and Howell, L., "The Underlying Theory of Project Management is Obsolete", *Proceedings of the 2002 PMI Research Conference*, 2002, 293-302.

[45] Womack, J. P., Jones, D.T., and Roos, D., *The Machine that Changed the World: The Story of Lean Production,* Harper Perennial, 1991.

[46] Chrissis, M. B., Konrad, M., and Shrum, S., *CMMI®: Guidelines for Process Integration and Product Improvement (1st edition),* Addison-Wesley Professional, 2003.

[47] ISO (International Standards Organization), *Standard for Information Technology – Software Life Cycle Processes*, ISO/IEC 12207, 1995.

15288 for SE[48] emphasize the need to integrate systems and SwE processes, along with hardware engineering processes and human engineering processes. They emphasize such practices as concurrent engineering of requirements and solutions, integrated product and process development, and risk-driven vs. document-driven processes. New process milestones enable effective synchronization and stabilization of concurrent processes.[49,50]

6.5.2 Systems Engineering and GSwE2009

In (Haskins, 2007) the primary SE professional society, INCOSE, defines *systems engineering* as "an interdisciplinary approach and means to enable the realization of successful systems." "Interdisciplinary" implies that all key disciplines—hardware, software, human factors, economics, application disciplines, legal, and so forth—need to collaborate on defining the system requirements and solution approach. "Enable" implies that good SE cannot guarantee success if its plans and specifications are furnished to incapable developers (but, as previously noted, poor SE can cause even good developers to fail to meet infeasible plans and specifications). "Successful" implies that all of the system's success-critical stakeholders will be no worse off once the system is in operation; otherwise, they will refuse to use or support the system. All of this implies that a project's SE group needs the skills to:

- Identify the success-critical stakeholders
- Determine the stakeholders value propositions
- Help stakeholders collaborate in defining and negotiating a mutually satisfactory set of plans and specifications
- Help adapt the plans and specifications in mutually satisfactory ways to respond to changes (in the environment, technology, competition, or participating organizations).

Each individual does not need to have all of the skills, but needs to understand how the other skills may impact their contributions. These four key SE skill areas are discussed more fully in Appendix C2.

SE is a crosscutting KA, and its units and topics should be integrated throughout the GSwE2009 curriculum components. For example, when knowledge from the Requirements Engineering KA is taught, how the systems context impacts software requirements should be addressed. Similarly, the Requirements Engineering KA should address how the feasibility of implementing specific software requirements influences both system requirements and system architecture.

---

[48] ISO (International Standards Organization), *Systems Engineering – System Life Cycle Processes*, ISO/IEC 15288, 2008.

[49] Boehm, B., "Anchoring the Software Process", *IEEE Software*, July 1996, 73-82.

[50] Kroll, P. and Kruchten, P., *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*, Addison-Wesley Professional, 2003.

The CBOK includes a separate section on SE knowledge that is not part of the original SWEBOK structure. However, this section does not fully capture the crosscutting nature of expected SE knowledge. This was the rationale for the name and terminology changes to the SWEBOK organization in the Requirements Engineering and Configuration Management areas. In addition, Table 2 includes a column labeled "Systems Engineering Content" that indicates (with a SYS designation) that a KA or knowledge unit has content and activities that are part of or relevant to system engineering. This represents a high-level view of system engineering; for a more detailed view, see Appendix C2. It should also be noted that although Figure 4 lists 3-7% for the System Engineering KA, additional system engineering material would be covered under such KAs as Requirements Engineering and Configuration Management.

A key distinction between SE and SwE is that SwE includes development of the software components, while SE excludes the development and manufacturing of software and hardware components, although it includes development and evolution of the plans and specifications for such activities, and integration and test of the resulting components. In particular, the terms in Table 2, such as Architectural Design, are not meant to imply that these activities are done once and sequentially at the beginning, but that they may involve continuing concurrent evolution throughout the system's life cycle.

7. Anticipated GSwE2009 Evolution

From the beginning, it was intended for GSwE2009 to be a living document, with a broad, responsible, and knowledgeable community of practice. It was anticipated that after Version 1.0 was published, Stevens Institute of Technology, which has managed the original development, would identify a steward who would assume responsibility for maintaining and refining the model and expanding and focusing implementation guidance based on experience and feedback from the supporting community and academia, industry, and students. Effort is now underway for a combination of the ACM and the IEEE Computer Society to become that steward. These organizations have played a major role in creating GSwE2009. As of the writing of this document, discussions are underway for those two organizations to take over maintenance responsibility for GSwE2009 within the first 6 months of the release of Version 1.0, with INCOSE playing a supporting role. Some minor changes are expected in the appearance of the document, such as the inclusion of the ACM and IEEE Computer Society logos, when that transition takes place.

This report fits logically within the Computing Curricula series of the ACM and the IEEE Computer Society that started in 2001. Hopefully, it will enjoy the same widespread acceptance and influence as the other reports in that series. To support and enable that acceptance, two companion documents - *Comparisons of GSwE2009 to Current Master's Programs in Software Engineering* and *Frequently Asked Questions on Implementing GSwE2009*– are being prepared concurrently with the release of GSwE2009. They will be available in Fall 2009 at www.GSwE2009.org and updated regularly.

This page intentionally left blank.

Appendix A. Summary of Graduate Software Engineering Programs in 2007

The first step in the iSSEc project was to understand the structure and content of currently implemented master's-level programs. Over 50 universities in the United States and many others globally offer a master's-level degree in SwE. In summer and fall 2007, data from 28 programs was collected and analyzed to enable a reasonable description of the current state of practice. Publicly available data was collected and then augmented and validated with a knowledgeable faculty member from the program.

A.1. Methodology

A list of candidate schools and graduate programs was constructed through Web searches, author contacts, and recommendations from members of the CAT. The range of schools listed included traditional universities, Web-based programs, and government-associated schools. The 28 programs that are included in the study are listed in Table 3.

Table 3. Participating Schools

1. Air Force Institute of Technology
2. Brandeis University
3. California State University – Fullerton
4. California State University – Sacramento
5. Carnegie Mellon University
6. Carnegie Mellon Silicon Valley
7. DePaul University
8. Drexel University
9. Dublin City University (Ireland) *****
10. Embry-Riddle Aeronautical University
11. George Mason University
12. James Madison University
13. Mercer University
14. Monmouth University
15. Naval Postgraduate School
16. Penn State University – Great Valley
17. Quebec University (Canada) *****
18. Rochester Institute of Technology
19. Seattle University
20. Southern Methodist University
21. Stevens Institute of Technology
22. Texas Tech University
23. University of Alabama – Huntsville
24. University of Maryland University College
25. University of Michigan – Dearborn
26. University of Southern California
27. University of York (UK) *****
28. Villanova University

* Non-U.S. schools

A taxonomy was needed to structure the analysis of competencies covered in the program curricula. Rather than create yet another SwE competency model, the team used the SWEBOK as a widely-available, collaboratively-developed and thoroughly-vetted taxonomy.

An Excel-based survey instrument was developed in which to collect and organize the data from the selected programs. The instrument captured data about the program, the courses, and the competencies taught within each course.

The program and course information was collected from public sources—primarily the Web. Using this information, the survey team performed an initial mapping of the course topics to the SWEBOK. Missing information was highlighted and any questions were captured during this initial pass. Subsequently, the survey team contacted a professor at the target institution. The initial data, emailed to and reviewed by that contact or a recommended substitute, was discussed in a telephone conference with members of the survey team. Missing data was filled in, errors were corrected, and in most cases, the contact made changes directly to the instrument and emailed it back to the survey team.

Although attempts were made to standardize the way in which the data was provided, there were still some differences in the level of detail provided and the interpretation of the instructions by the academic program personnel. This led to adjustments to the way the team analyzed the data. To accommodate the differing levels of granularity, as represented by the differing SWEBOK levels of the data, the team decided that, for initial analysis, data would be analyzed at the third SWEBOK level and reported at the first SWEBOK level. This required the team to heuristically aggregate from lower levels to higher levels where the data had been provided at finer granularity.

The findings from the survey fall into two general categories: program characteristics and curriculum characteristics.

A.2. Program Characteristics

The spectrum of programs investigated led to a number of findings about how the programs were managed, their faculty resources, size, longevity, and individual personalities. Some of the more interesting findings are as follows:

[1] Software engineering (SwE) is largely viewed as a specialization of Computer Science—much as SE was often viewed as specialization of industrial engineering or operations research years ago. Data shows that only 26% of the programs are in SwE departments, 44% are within Computer Science departments, and the rest are in a myriad of other academic organizations.

[2] Faculty size is generally small, with few dedicated SwE professors. Forty-eight percent of the programs have five or fewer dedicated full- or part-time faculty members. There is heavy reliance on adjunct faculty for teaching.

[3] Student enrollments are generally small compared to Computer Science and other engineering disciplines. Twenty-nine percent of the programs have 25 or fewer students and 71% have 100 or fewer.

[4] Many programs specialize in specific markets such as defense systems acquisition or safety-critical systems. Those markets are often driven by local businesses.

[5] Admission requirements vary widely. Some will accept anyone with a bachelor's degree and a B average, while others require a computer science degree and at least two years of relevant experience. Leveling courses are widely provided to support students unable to meet all requirements.

[6] Program outcome goals are quite diverse. Programs are set up to produce graduates according to the perceived needs and desires of the target student population. Some programs focus on developing skilled software development team members. Others focus on the skills and knowledge required to manage complex projects. In some cases, the graduates are prepared to be chief engineers and software executives.

[7] Programs continue to be started despite the widespread concern over the decline in computer science majors over the last few years. Of the 28 programs in our study, eight were started since 2001.

[8] On-line offerings are popular, with many programs reaching students far from their physical campuses and some citing a global reach.

A.3. Curriculum Characteristics

The structure and content of courses in existing programs and the relationship of those curricula to standards such as SWEBOK provided valuable insight for GSwE2009 development. Although the survey team collected data on all courses offered by the master's programs, the initial analysis only looked at courses that were required or semi-required, where a semi-required course is one that a student has at least a 50% chance of taking. Some of the more interesting findings are:

[1] Fewer than 40% of all programs required an introductory course on SwE.

[2] There is a wide variation in the depth and breadth of SWEBOK coverage in required and semi-required courses. The well-covered areas are courses in requirements, design, and management. The least well-covered areas are courses in maintenance, configuration management, quality, tools, and methods.

[3] It is clear that the SWEBOK alone does not represent the breadth of many program's required courses. Many programs required courses on specific programming languages (such as C++, Java, and C#), software economics, human factors and user interface design, and legal/ethical issues of software development.

[4] Few surveyed programs explicitly address systems engineering in their required and semi-required courses.

[5] "Object-Oriented" is the standard development paradigm. Almost no one teaches structured methods except for historical interest.

[6] The flexibility of coursework varies widely. For example, one school offered no electives—every course was required. On average, students take 11.6 courses for their degree, 8.3 of which are required or semi-required.

[7] Capstone practicums and projects are frequently required.

A.4. Conclusions

The initial work produced a reasonable profile of master's programs currently offered. The diversity is clearly evident, helping to motivate and inform the GSwE2009 effort.

Appendix B. Bloom Levels for the Body of Knowledge

B. 1 Introduction

Bloom's Taxonomy is a classification system devised in 1956 by group of educators lead by Benjamin Bloom.[51] The taxonomy can be used by educators to set the level of educational/learning objectives required for students engaged in an education unit, course, or program. Bloom's Taxonomy divides educational objectives into three domains: Affective, Psychomotor, and Cognitive. In this document, the focus is on the Cognitive Domain, which is concerned with what we know and how we know it.[52] Conventional education systems tend to stress outcomes in the cognitive domain, particularly the lower-level objectives.

Bloom's taxonomy is hierarchical; i.e., learning at a higher level is dependent on attaining prerequisite knowledge and skills at the lower levels. Table 4 provides a description of the Bloom's Levels for the Cognitive Domain. There is some debate about the ordering of the two highest levels, Synthesis and Evaluation: should their order be reversed or should they be placed at the same level? This is a research area that the GSwE2009 project does not attempt to address, but rather stays with the more traditional view.

Table 4. Explanation of Bloom Taxonomy Cognitive Levels

Level	Competency	Objective Descriptors
Knowledge (K)	Remembering previously learned material. Test observation and recall of information, i.e., "bring to mind the appropriate information"; e.g., dates, events, places, knowledge of major ideas, mastery of subject matter.	List, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name (who, when, where, etc.)
Comprehension (C)	Understanding information and ability to grasp meaning of material presented. For example, translate knowledge into new context, interpret facts, compare, contrast, order, group, infer causes, predict consequences, etc.	Summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend
Application (AP)	Ability to use learned material in new and concrete situations; e.g., use information, methods, concepts, theories to solve problems requiring the skills or knowledge presented.	Apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover

[51] Bloom, B. S. (Ed.), *Taxonomy of educational objectives: The classification of educational goals: Handbook I, cognitive domain,* Longmans, 1956.

[52] Huitt, W., "The cognitive system," *Educational Psychology Interactive*, Valdosta, GA, Valdosta State University, 2006. Retrieved May 22, 2008 from http://chiron.valdosta.edu/whuitt/col/cogsys/cogsys.html

Level	Competency	Objective Descriptors
Analysis (AN)	Ability to decompose learned material into constituent parts in order to understand structure of the whole. This includes seeing patterns, organization of parts, recognition of hidden meanings, and obviously identification of parts.	Analyze, separate, order, explain, connect, classify, arrange, divide, compare, select, explain, infer
Synthesis (S)	Ability to put parts together to form a new whole. This involves the use of existing ideas to create new ones, generalizing from facts, relating knowledge from several areas, and predict, draw conclusions. It may also involve the adaptation of "general" solution principles to the embodiment of a specific problem.	Combine, integrate, modify, rearrange, substitute, plan, create, design, invent, what if?, compose, formulate, prepare, generalize, rewrite
Evaluation (E)	Ability to pass judgment on value of material within a given context or purpose. This involves making comparisons and discriminating between ideas, assessing value of theories, making choices based on reasoned arguments, verify value of evidence, and recognize subjectivity.	Assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarize

Table 5 shows some examples of various Bloom level competencies that might apply to GSwE2009 curricula and courses.

Table 5. Example Cognitive Levels for Software Engineering

Level	Competency
Knowledge (K)	• The student is able to recite the definitions of "class" and "object" and to state the connection between them. • The student is able to define the notion of the waterfall (iterative, incremental, spiral model) software development process.
Comprehension (C)	• The student is able to explain how to decide if something should be modeled as a class or as an object. • The student is able to explain, in a very general way, the conditions under which a software development team might choose to use a waterfall (iterative, incremental, spiral model) software development process as opposed to one of the others.
Application (AP)	• The student is able to start with a simple requirements document and produce a reasonable draft of a UML domain model. • Given the operational concept and requirements of a simple application or system along with a specified budget and required completion time the student is able to choose, and to provide a rudimentary justification for the choice of a particular software development process from among the popular ones, e.g., waterfall, iterative, incremental, and spiral.
Analysis (AN)	• Given a simple requirements document and a UML domain model for an application, the student is able to critique the domain model, e.g., finding classes that should probably have been modeled as objects and vice versa, associations whose names are hard to understand and should be renamed, incorrect multiplicities of associations, etc. • Given the operational concept of a simple application or system along with a requirements document, a budget, a required completion time, a choice of a specific

Level	Competency
	software development process, and a justification of the use of that process on the project, the student is able to find and explain errors in the justification and/or in the choice of process.
Synthesis (S)	• Given a detailed requirements document and a well-constructed UML domain model for an application/system, the student is able to design at least one basic architecture and implementation class diagram(s) for the application/system. • Given an operational concept, requirements, architecture and detailed design documents for an application/system, the student is able to construct a complete implementation plan and provide a cogent argument that if the implementation of the architecture/detailed design is performed according to the plan, then the result will be an application/system that satisfies the requirements, fulfils the operational concept, and will be completed within budget and within schedule.
Evaluation (E)	• Given the operational concept, requirements, architecture, detailed design, and implementation plan, including budget and schedule, for an application/system, as well as a feasibility argument for the implementation plan, the student is able to assess the plan and to either explain why the feasibility argument is valid or why/where it is flawed with regard to any of the claims regarding implementation of the requirements, fulfillment of the operational concept, or ability to be completed within budget and schedule.

B. 2 Frequently Asked Questions

The following questions and answers should help explain why and how Bloom's Taxonomy is used in GSwE2009.

[1] *Why were Bloom's levels used in the GSwE2009 Core Body of Knowledge (CBOK)?*

Bloom's Taxonomy is used in Section 6 (Tables 1 and 2) to indicate the desired minimum level of attainment of knowledge elements for the GSwE2009. The Bloom's classification system was chosen for two reasons: it is a well-recognized and widely-used system in the design and assessment of education components; and in the SWEBOK, Appendix D used Bloom's Taxonomy to specify "general requirements" for practicing software engineers.

[2] *How were the Bloom's levels in Tables 1 and 2 determined?*

Tables 1 and 2 Bloom's level designations were initially determined by a subgroup of the CAT, four SwE educators and practitioners, in a somewhat subjective manner. The process began with the assignment of the SWEBOK Appendix D designations and then, using a quasi-wideband Delphi technique, the CBOK Bloom's designations were adjusted. Subsequent review by the CAT and external reviewers helped finalize the Bloom's level designations.

[3] *Why are there no Synthesis (S) levels or Evaluation (E) levels assigned to the CBOK elements?*

Appendix D of the SWEBOK uses only the Knowledge (K), Comprehension (C), Application (AP), and Analysis (AN) levels for SWEBOK elements. The reason for this is that the SWEBOK targets a SwE graduate with four years of experience and it is intended to specify "generally accepted" knowledge that all software engineers should possess. However, software engineers working in specialized areas (e.g., configuration management or quality assurance) may need to possess knowledge at a higher Bloom's level. This is the philosophy adopted by the GSwE2009 CBOK. That is, the Bloom's levels in the GSwE2009 CBOK specify the minimum knowledge levels that all graduates of GSwE2009 programs should attain. If a graduate program places special focus in its curriculum on a particular area, then it may want to use higher Bloom's levels in that area.

In addition, one might view the S and E levels as more appropriately associated with knowledge acquired with additional experience and education. For example, a senior software architect would certainly be expected to have S and E level attainment in much of the Design area of the CBOK; and likewise, an accomplished researcher in software testing would be expected to have acquired the highest Bloom's levels in this area.

[4] *How should the level designations be used by developers of graduate software engineering curricula?*

The Bloom's level designations can help guide the type of learning activities that should incorporated into a curriculum. First, one should note that all of the GSwE2009 levels in Table 2 are at C level or above. This means that rote learning which results only in abilities to list, define, describe, and so forth is not appropriate.

In developing a course or a unit in the curriculum, the Bloom's level for a specific area implies explicit student capabilities and could be used to develop the learning objectives for the course or unit. For instance, in a course in software or system requirements, the AP designation for Requirements Elicitation could lead to a learning objective such as "Students will be able to use requirements elicitation technique *A* to determine the requirements for a system in domain *B*." The particular technique and application domain involved would vary from program to program. The AN designation for Requirements Analysis would demand that a requirements course incorporate learning objectives and activities that would involve the use of techniques to classify, organize, allocate, model, and analyze the requirements of a software system.

Appendix C. Description of CBOK Elements Not Found in the SWEBOK

This appendix contains material that describes and elaborates on KAs and units that are part of the CBOK (see Section 6 of this document), but are not part of the SWEBOK or which have lesser emphasis in the SWEBOK. There should be no inference that Appendix C material is part of the current SWEBOK or is directly intended to influence future versions.

C.1. Ethics and Professional Conduct

Software engineers develop and maintain products that influence almost every area of human endeavor: medicine and health, transportation and communication, business and finance, education, government and law, and arts and entertainment. In order to deliver software products effectively and efficiently, software engineers must conduct themselves ethically and professionally. This KA outlines the issues and elements of such conduct.

C.1.1. Social, Legal, and Historical issues

Data Confidentiality and Security, Surveillance and Privacy

Issues related to privacy, confidentially, and the security of individual information has become a significant problem in the information age. In particular, the *Software Engineering Code of Ethics and Professional Practice*[53] states that software engineers shall "work to develop software and related documents that respect the privacy of those who will be affected by that software." The use of the Internet and large databases that hold private information (medical, financial, legal, etc.) place a greater responsibility for ethical and professional conduct on the software engineers who develop products that deal with this confidential and private information[54,55].

Contracts and Liability, Intellectual Property, Freedom of Information

Software is typically developed in a society that has laws concerning contracts, intellectual property (copyrights, trademarks, and patents), freedom of information, and employment law. A software engineer must be aware of such laws and be governed in their practice by the requirements and restraints of such laws[56,57,58]. For example, in the U.S., UCITA[59] governs

[53] ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices, *Software Engineering Code of Ethics and Professional Practice*, Version 5.2, 1999. http://www.acm.org/serving/se/code.htm.

[54] Bott, F., et. al., *Professional Issues in Software Engineering (3rd edition)*, Taylor & Francis, 2000.

[55] Tavani, H. T., *Ethics & Technology: Ethical Issues in an Age of Information and Communication Technology.* Wiley, 2003.

[56] Bott, F., et al., *Professional Issues in Software Engineering (3rd edition)*, Taylor & Francis, 2001.

[57] Kaner, C., "Issues in Commercial Law of Interest to Software Engineering Educators," tutorial session at the *Conference on Software Engineering Education & Training (CSEE&T)*, February 2002.

[58] Tavani, H. T., *Ethics & Technology: Ethical Issues in an Age of Information and Communication Technology.* Wiley, 2003.

transactions involving "information rights," which includes "all rights in information created under laws governing patents, copyrights, mask works, trade secrets, trademarks, publicity rights, or any other law that gives a person, independently of contract, a right to control or preclude another person's use of or access to the information on the basis of the rights holder's interest in the information."

Computer Crime and Law Enforcement

Clearly software engineers must not engage in criminal activity; however, with the advent and increase in cybercrime, software engineers must guard against such crime and report crimes or suspicions of criminal activity. This is in accordance with ACM,[71] which states that software engineers shall "disclose to appropriate persons or authorities any actual or potential danger to the user, the public, or the environment, that they reasonably believe to be associated with software or related documents."

Historical Developments, and Gender, Minor, and Cultural Issues

Since software engineers develop and maintain products used by humans, it is important that they understand the historical and cultural aspects of their profession and the related context in which their products will be used.

Software engineers need to be aware of societal diversity and always act without prejudice or discrimination. The British Computer Society (BCS)[60] states, "You shall conduct your professional activities without discrimination against clients or colleagues," and the *Software Engineering Code of Ethics and Professional Practice* asserts that software engineers shall "be fair to and supportive of their colleagues," "consider issues of physical disabilities, allocation of resources, economic disadvantage and other factors that can diminish access to the benefits of software," and "help develop an organizational environment favorable to acting ethically." An article detailing the developments in SwE professionalism from 1996 to 2008 can be found in Encyclopedia of Computer Science and Engineering.[61]

C.1.2. Codes of Ethics and Professional Conduct

Responsibilities to Society

All engineers who create products for society's use have an obligation to perform in a professional manner. Because of the software's criticality and its ubiquitous nature, software engineers have special responsibility.

---

[59] *Uniform Computer Information Transactions Act*. National Conference of Commissioners on Uniform State Laws, 2001.

[60] British Computer Society, *Code of Conduct & Code of Good Practice*, 2004 and 2006, http://www.bcs.org/server.php?show=nav.10967.

[61] Thompson, J.B., "Perspectives on Software Engineering Professionalism," *Wiley Encyclopedia of Computer Science and Engineering*, edited by Benjamin Wah, Wiley, Hoboken, NJ, 2009.

The *Software Engineering Code of Ethics and Professional Practice* states, “Software engineers shall act consistently with the public interest,” “The ultimate effect of the (software engineer’s) work should be to the public good,” and “Moderate the interests of the software engineer, the employer, the client and the users with the public good.”

Models for Professionalism, Professional Societies

Models of a profession depend on society’s understanding of the term “profession.” The *Oxford English Dictionary* defines a profession as “an occupation, vocation or high-status career, usually involving prolonged academic training, formal qualifications and membership of a professional or regulatory body. Professions involve the application of specialized knowledge of a subject, field, or science to fee-paying clientele.” The Accreditation Board for Engineering and Technology (ABET) (www.abet.org) defines engineering as “the profession in which a 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.”

There have been several studies of the professional nature of SwE. In 1996, Mary Shaw traced the history of the evolution of several engineering disciplines and compared them with the evolution of software development practices. In their 1996 study of a SwE profession, Ford and Gibbs[62] concluded that an engineering profession has the following features:

- An initial professional education in a curriculum validated by society through accreditation
- Registration of fitness to practice via voluntary certification or mandatory licensing
- Specialized skill development and continuing professional education
- Communal support via a professional society
- A commitment to norms of conduct often prescribed in a code of ethics

In the past decade there have been significant advances in these areas (Thompson 2009); however, there are no professional societies solely devoted to SwE. There are two international societies that are closely associated with SwE interests and practices:

- Association for Computing Machinery (ACM), http://acm.org
- IEEE Computer Society (IEEE-CS), http://www.computer.org

[62] Ford, G. and Gibbs, N.E., *A Mature Profession of Software Engineering,* CMU/SEI-96-TR-004, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1996.

Codes of Ethics and Practice

A code of ethics and professional conduct is the hallmark of a profession. Medicine, law, and engineering have historically required adherence to such codes in order for an individual to pursue public practice of the profession.

In 1999, the ACM and the IEEE-CS established the *Software Engineering Code of Ethics and Professional Practice*. Other organizations, countries and regions have codes that cover the practice of SwE (e.g., ACM 1192[63], ACS 2008[64], BCS 2008[65], CSI 1993[66]). Views on professionalism and ethical considerations related to professionalism are given in a recent article by Loui and Miller[67].

C.1.3. The Nature and Role of Software Engineering Standards

Nature and Role of Standards

A major role of a profession is to standardize the terminology, measurement methods, and process methods used in national and international practice of the profession. The goal is to enable professionals, educators, and organizations to communicate internationally, and to improve the effectiveness and efficiency of professional practice.

International Standards, Standards and Harmonization Organizations

The IEEE–CS Software and Systems Engineering Standards Committee (S2ESC) has created many of the standards that guide the development and maintenance of software systems. The *IEEE Software Engineering Standards Collection* (http://standards.ieee.org/reading/ieee/std_public/description/se/index.html) contains 40 IEEE standards for software development. This collection is described and discussed in Moore[68]; it also includes an overview of standards-making organizations. The International Standards Organization (ISO) and the American National Standards Institute (ANSI) have created other software, system, and related standards.

[63] ACM Council, *ACM Code of Ethics and Professional Conduct*, October 1992, http://www.acm.org/constitution/code.html.

[64] Australian Computer Society, "Code of Ethics" website, 2008, http://www.acs.org.au/index.cfm?action=show&conID=200509022322219027.

[65] British Computer Society, *Code of Conduct & Code of Good Practice*, 2004 and 2006, http://www.bcs.org/server.php?show=nav.10967.

[66] Computer Society of India, *CSI Code of Ethics*, 2008, http://www.csi-india.org/code-ethics.

[67] Loui M.C. and Miller, K.W., "Ethics and Professional Responsibility in Computing," in *Encyclopedia of Computer Science and Engineering*, edited by Benjamin Wah, Wiley, Hoboken, NJ, 2009.

[68] Moore, J. W., *The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society, 2006.

Bodies of Knowledge, Accepted and Best Practices

A body of knowledge, describing the organization and the principal elements of a discipline, provides a foundation for a profession that supports curriculum development, certification and licensing, continuing professional education, and a code of ethics and professional conduct.

The best-known and most widely-used body of knowledge for SwE is the SWEBOK. In its introduction, SWEBOK presents its objectives:

- To promote a consistent view of SwE worldwide
- To clarify the place—and set the boundary—of SwE with respect to other disciplines such as computer science, project management, computer engineering, and mathematics
- To characterize the contents of the SwE discipline
- To provide a topical access to the Software Engineering Body of Knowledge
- To provide a foundation for curriculum development and for individual certification and licensing material.

Other SwE related bodies of knowledge include the CBOK in Section 6 of this document, and the bodies of knowledge in SE2004 and in the other curriculum guides at http://www.computer.org.

In order to improve software products and to advance the state of SwE as a profession, there have been numerous studies of "best" SwE practices: Beck[69], CMMI[70], Cusumano[71], and Jones[72].

C.2. Systems Engineering

A system is a collection of interconnected components that exist within and interact with an environment. System engineers analyze needs, develop solution concepts, and work with component and system quality attribute specialists (safety, cost, performance, etc.) to synthesize the evolving definition of complex systems consisting of diverse kinds of components. Systems engineers also play a part in the installation and support of these systems in their operational environment, and their eventual removal from service and safe disposal. Virtually all modern-day systems ranging from air traffic control systems to nuclear reactors to financial transaction systems are dependent on software to coordinate the interconnections among system components

[69] Beck, K. and Andres, C., *Extreme Programming Explained: Embrace Change* (2nd edition), Addison-Wesley Professional, 2004.

[70] CMMI Product Team, *Capability Maturity Model, Version 1.1, CMMI for Software Engineering, Staged Representation*, CMU/SEI-2002-TR-029, Software Engineering Institute, Carnegie Mellon University, August 2002.

[71] Cusumano, M., et al., "Software Development Worldwide: The State of the Practice", *IEEE Software 20(6)*, November/December 2003.

[72] Jones, C., "Variations in Software Development Practices", *IEEE Software 20(6)*, November/December 2003, 22-27.

and to provide the functionality of these systems. Software engineers are thus key members of modern system engineering teams.

C.2.1. Systems Engineering Concepts

System Context

The basic concept of a system is very simple: as stated above, it identifies a bounded collection of elements that interact with something larger. This simple idea can be applied to help understand many overlapping real-world relationships; e.g., a commercial aircraft is a system, and contains systems for navigation, propulsion, flight control, catering and entertainment, baggage handling, and so forth. The navigation system contains integrated hardware, software and human components, and has interfaces and relationships with other systems both inside and outside of the aircraft boundary. The aircraft is also part of the airline commercial system, the airport and air traffic control systems, and so forth, all of which can be considered as part of commercial transport, tourism, or commerce systems. To effectively apply the processes and techniques of systems and SwE, we need to select and identify a specific context.

One valuable model that systems engineers use to focus on the systems relationships key in a given situation is to define a System of Interest (SoI) and Wider System context as shown in Figure 5.[73]

The SoI is the selected system for which a life cycle is needed, and over which the organization can exercise some authority and decision-making. This authority may include the specification, design and build of custom-made solution elements and the selection and integration of Commercial Off-the-Shelf (COTS) components; or it may simply be setting the criteria for the selection and use of computing and other services. The Wider SoI is the broader context in which the problem situation and measure of success are defined. The SoI authority will have some influence in the wider system, but may have to negotiate with others. The Environment describes the system relationship and conditions in which the Wider SoI must operate. The SoI must be engineered to deal with a range of possible environments. The Wider Environment includes environmental issues that may not affect system operation directly, but may influence system choices (e.g., consumer market trends).

People and Systems

It is important to distinguish between Users and Operators. Users are those who will interact with the SoI in conjunction with their work activities or personal pursuits. Thus, users are part of the wider SoI, and are concerned with how a SoI enables them to contribute to the effectiveness of that wider system. Operators are people or organizations directly involved in the operation of the SoI. In one way, operators can be treated just like any other system component, making decisions about them to maximize system characteristics. While this view of humans treats them as a

[73] Flood, R. L. and Carson, E.R., *Dealing with Complexity* (2nd edition). Plenum Press, 1993.

technology that can be selected and used within the SoI, it is important to consider people's unique attributes and constraints when they are "designed into the solution." The human elements of a complex system are often hired and trained to perform specific functions. If so, they can be disciplined or dismissed if they fail to perform their duties in a satisfactory manner. They can also be encouraged and incentivized to find ways to improve the system's operation (as with, for example, air traffic controllers, operators of nuclear reactors, and bank tellers).

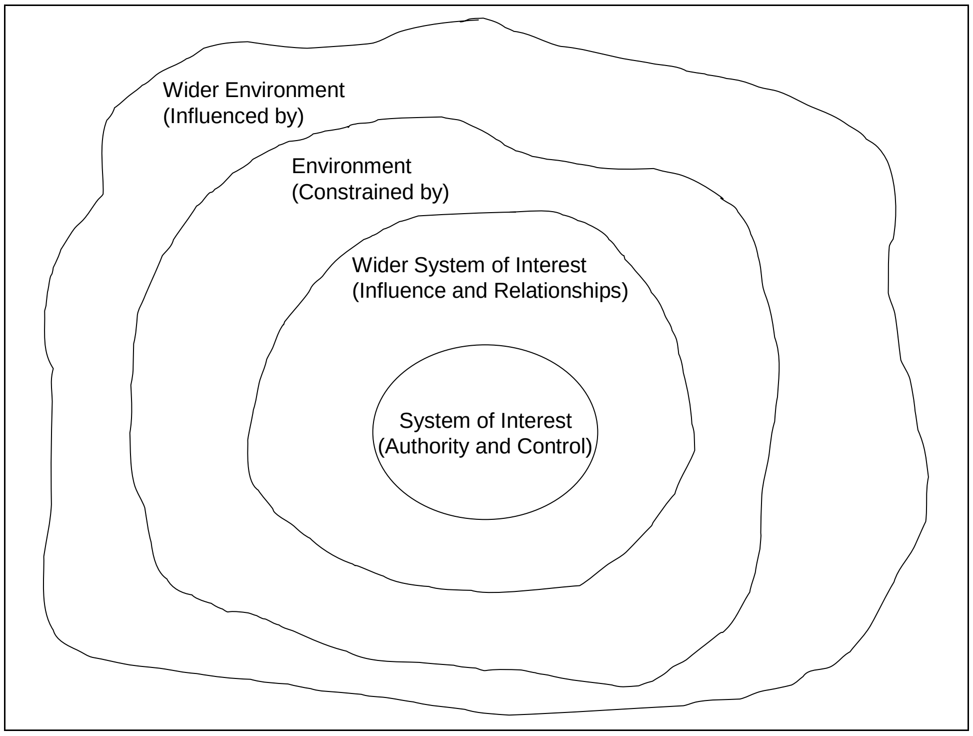


Figure 5. SoI Levels of System Context

Alternatively, users are external to the system. Users of an automated teller machine (ATM), for example, must have an ATM card and a password plus minimal skills to operate the interface, but beyond that they are not within the realm of control of the system or its sponsors, as are bank tellers and system maintainers. There may be different user or operator classes who interact with the system in distinctive ways. Some of those distinctions may be reflected in different modes of system operation (e.g., full operational mode, degraded mode, emergency mode, training mode, and maintenance mode; or perhaps novice, intermediate, and expert modes). User and operator classes and modes of operation must be identified and documented. There may be gray areas in which people must be considered in both user and operator roles, such as nurses as medical device operators or aircraft pilots.

A stakeholder is any individual or organization that affects or is affected by a SoI at some point during its life. Identifying stakeholders is thus a key aspect of SE. The main stakeholder distinctions are between customers, acquirers, and other interested groups. A customer is the individual or organization that specifies the requirements and accepts the delivered system, and, if applicable, pays for system development. In some cases the users and customer are the same individual or organization. In other cases, users and customer may be distinct entities, possibly with a separate acquirer or system integrator role.

Clearly, software engineers and system operators affect the SoI and are key stakeholders. The other stakeholders who may be relevant to a SoI are generally representatives of some constraint or limitation on the system's operation or development: for example, a regulator such as the U.S. Federal Aviation Administration, a trades union or industry body, representatives of local community or environmental lobbies, or venture capitalists providing funds.

System Hierarchical Relationships

Complex systems are comprised of hardware, software, and human elements. The hardware elements include computers and other kinds of hardware devices (e.g., radars). Hardware is procured and/or developed as part of system development. The software element may include legacy software as well as software components to be developed and/or modified. There may be other hardware and software in the environment that provides interfaces to the system.

An Enabling System is a system that complements a SoI during its life but does not necessarily contribute directly to its function during operation.[74] To fully understand the context of a system we may need to consider not only its operational environment, but also how it is transported, stored, maintained, tested, and so forth.

The hardware, software, and human elements of complex systems are also systems (i.e., subsystems): they are collections of interconnected components that exist within and interact with their environments. The environments with which they interact consist of other subsystems and the external system environment. Each hardware and software subsystem contains its subsystems. They are decomposed to whatever level is needed to specify and synthesize the collection of components that constitute a system. Different subsystems may be decomposed to different levels.

A key concern for software engineers is that hierarchical relationships in hardware architectures are often incompatible with good software architecture hierarchies. Hardware architectures tend to have one-to-many "part-of" relationships (the wings are part of the airplane, the ailerons are part of the wings, the aileron controls are part of the ailerons, the aileron control software is part of the aileron controls, the aileron control data management is part of the aileron control software). With many independent aircraft part suppliers, this often leads to incompatible data management, fractionated software management, and slow software response to cross-cutting changes.

[74] Womack, J. P., Jones, D.T., and Roos, D., *The Machine that Changed the World: The Story of Lean Production,* Harper Perennial, 1991.

Modern software architecture practices generally emphasize layered many-to-many "served by" hierarchical relationships, which are extremely difficult to put in place once hardware-oriented systems engineers have committed the project to a "part-of" architecture, work breakdown structure, management structure, and contract structure.[75] This is one of many reasons why software engineers need to be proactive participants in SE.

A System of Systems (SoS) is an interoperating collection of component systems that cooperate together to produce results unachievable by the individual systems alone. What makes this different from the levels of hierarchy described above is that the component systems can operate independently and may be part of more than one context, and that each system will have independent customers and life cycles. These SoS relationships will have a strong influence on the lifecycle choices and commercial framework within which a particular SoI is considered. They also present a challenge in aligning the degree of responsibility of a SoS engineer or manager with their degrees of authority over SoS decisions, another reason why software engineers need to be proactive participants in SE.

The Role of System Engineers

The complex systemic relationships in a real world business or public problem domain are dealt with in two ways:

- *Managed Problem Solving*—making all necessary changes within an organization to provide stakeholder benefits, within the cost and other resource constraints of the organization.
- *Engineering Problem Solving*—identifying, planning, and synchronizing technical activities to deliver identified system changes against agreed measures of effectiveness.

The key skills of a SE team needed to support both of these were identified in Section 6.5, and are further expanded below:

- *Identifying the success-critical stakeholders.* This includes the abilities to diagnose what is unsatisfactory to whom among the stakeholders in the current situation, and what are the root causes of this dissatisfaction, involving the various quantitative and qualitative techniques of operations research. It also involves envisioning opportunities to involve additional stakeholders who provide keys to better system solutions, such as COTS vendors, service providers, strategic partners with needed skills, investors, or new classes of system users.
- *Determining key stakeholders' value propositions.* This will depend on the classes of stakeholders. Users' value propositions can be determined via surveys, brainstorming, prototyping, option ranking techniques, or other preference-gathering techniques. Various

[75] Maier, M. W., "System and Software Architecture Reconciliation," *Systems Engineering* 9(2), Summer 2006, 146-159.

types of business case analysis can determine investors 'and managers' value propositions, including market trend analysis, operational cost savings analysis, critical path analysis, and return on investment analysis. Developers, maintainers, administrators, and operators' value propositions will involve the development, maintenance, and operational feasibility of the system solution options, requiring the formulation and analysis of alternative development approaches (what to make, buy, or subscribe to for services) and operational concepts. Another key stakeholder class is the general public, whose value propositions concerning safety, security, fairness, or privacy may involve regulatory bodies, standards compliance, or legal constraints.

- *Helping stakeholders collaborate in defining and negotiating a mutually satisfactory set of plans and specifications*. This will include identifying the actual or likely conflicts, risks, and uncertainties among the stakeholders' value propositions; analyzing the risks; identifying options for resolving the conflicts and risks; performing tradeoff analyses to evaluate the options with respect to the desired value propositions; combining options to synthesize candidate solutions; helping stakeholders negotiate mutually satisfactory solutions; and verifying and validating the feasibility of the solutions. These activities involve a wide variety of skills, including both technical and interpersonal skills. System engineers are not necessarily component specialists. Their expertise lies in knowing the business milieu or operational domain in which their systems will operate, in performing the top-level activities that must be accomplished to develop a complex system that incorporates various kinds of technologies, and in coordinating the work activities of specialists from various technical disciplines. But they must be conversant with the technologies to be used so they can speak intelligently with the component specialists concerning requirements, capabilities, interfaces, and tradeoffs. In addition, system engineers must work with those in specialty disciplines such as safety, security, reliability, usability, integration, configuration management, verification, and validation.

Not everyone needs to possess all of the skills, but good systems engineers learn how to collaborate with specialists to help achieve good combinations of system requirements, plans, solutions, and evidence of their compatibility and feasibility. At a minimum, software engineers need to participate in SE as technical specialists. But they will be far more effective if they participate as systems engineers with a particular technical specialty.

There are no one-size-fits-all processes for developing and evolving the wide variety of systems in terms of size, criticality, subset-ability, number of component systems, availability of reusable solution elements, degree of legacy constraints, ability to prespecify requirements, and degree of requirements volatility. Systems engineers need to be able to determine what type of process best fits the combination of such characteristics that are involved in the system they are engineering.

- *Adapting the plans and specifications in mutually satisfactory ways to respond to changes.* This is an increasingly important skill, as the speed of change in our world continues to increase. Traditionally, systems engineers would prespecify plans and specifications, and then go on to engineer other systems. But nowadays and increasingly in the future, systems engineers are necessary across a system's entire life cycle to assess changes in the environment, technology, competition, or participating organizations, and to help adapt the systems content, plans and specifications in mutually satisfactory ways to respond to the changes.

The third problem-solving approach that systems engineers must be aware of is:

- *Strategic Problem Solving*—initiating, monitoring, and guiding a collection of managed problem-solving activities to evolve and grow a SoS or enterprise.

Traditional management or engineering approaches cannot directly tackle strategic problems. However, there is a need to understand SoS needs and constraints and propose alternative SoS solution strategies; make trades across project boundaries against SoS criteria and integrate delivered systems into a SoS environment; evaluate SoS performance to feedback into enterprise level investment decisions. One of the main enablers for this strategic approach is the ability to create modular system products integrated through an open, software-intensive infrastructure. Thus, both systems and SwE skills have a role to play in this level of problem solving.

C.2.2. Systems Engineering Life Cycle

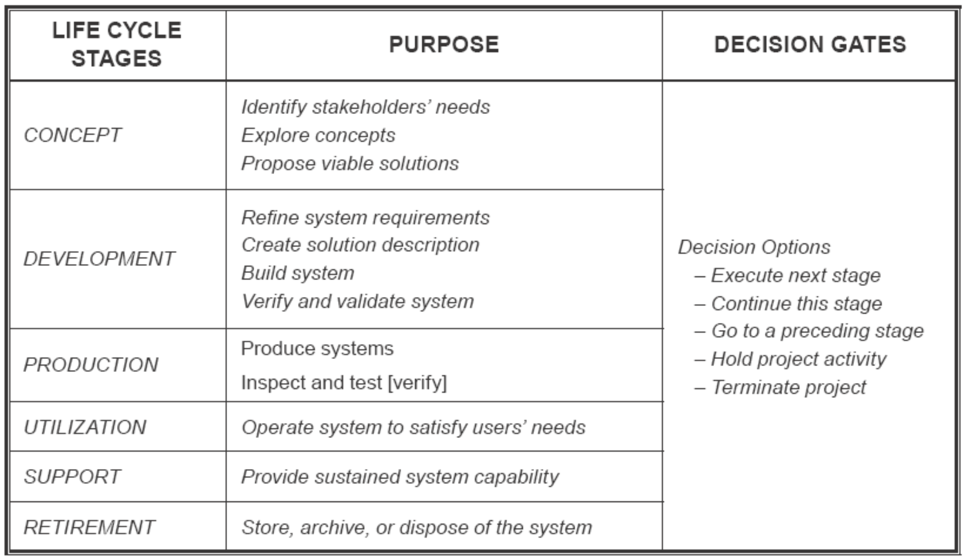
Life Cycle Management

A lifecycle is the organized collection of activities, relationships and contracts that apply to a SoI during its life. As such, it is the relationship to a lifecycle that shapes how and when SE is applied, and sets the context for the relationship between project management, SE and SwE activities.

In standards such as ISO/IEC 15288:2008[76], a lifecycle is described by a set of Life Cycle Stages. Each stage is related to part of a basic problem solving approach. The stage is described by an overall objective, an indication of the activities related to the stage, and a definition of the things to be achieved by the end of the stage. Figure 6 shows the simple set of stages in the ISO 15288:2008[88] standard.

[76] ISO (International Standards Organization), *Systems and Software Engineering – System Life Cycle Processes*, ISO/IEC 15288, 2008.

.



LIFE CYCLE STAGES	PURPOSE	DECISION GATES
CONCEPT	*Identify stakeholders' needs* *Explore concepts* *Propose viable solutions*	*Decision Options* *– Execute next stage* *– Continue this stage* *– Go to a preceding stage* *– Hold project activity* *– Terminate project*
DEVELOPMENT	*Refine system requirements* *Create solution description* *Build system* *Verify and validate system*	
PRODUCTION	Produce systems Inspect and test [verify]	
UTILIZATION	*Operate system to satisfy users' needs*	
SUPPORT	*Provide sustained system capability*	
RETIREMENT	*Store, archive, or dispose of the system*	

Figure 6. ISO 15288:2008 Generic Life Cycle Stages

From a problem-solving viewpoint, the stages are put together in a logical way to move a project forward. At the end of each stage, the project will review its progress against the stage outcomes and make a decision to move to a new stage, return to a previous stage, extend the current stage, or stop or delay the project.

It is important to emphasize that the activities shown against each stage are not confined to that stage. Figure 6 shows the activities critical to the purpose of a stage, and for which there will be significant activity during the stage. However, these activities may also need to be revisited during later stages, or considered and planned for in earlier stages.

In the real world, life cycles are also used for strategic and financial project planning. The extent to which a project needs to fit within a predefined life cycle template or to plan several stages ahead to secure financial resources, depends upon the kind of problem, organization, and sector. To illustrate this we might expect life cycle planning to range from:

- A predefined life cycle template, which a project must fully define and cost before it can gain approval to start any work. This gives a good understanding of cost and risk, but may not be flexible enough to deal with changing objectives and opportunities, and can lead to expensive rework and overruns.
- A scoping study or demonstrator project can be used to run through the life cycle for a simplified problem, to gain confidence prior to full project planning. This helps to identify and remove risks, but it can take time and money that might be better spent on the real problem.
- An incremental or spiral life cycle allows the project to deliver useful outcomes, while dealing with potential risk or uncertainties. This is a good approach for many systems,

particularly unprecedented systems in an environment of rapid change, but may be harder to manage without detailed forward planning and cost prediction.

- The most flexible approach is to fund a single life cycle stage, and at the end of that stage, to review the results obtained so far, the plans for the next stage and the resources required to achieve them. This gives the most flexible approach, but needs an equally flexible view of funding and planning, including a continuous investment in concurrent SE during development stages, to make it work.

These basic life cycle approaches apply to projects, which are focused commercial activities to produce a specific change to an identified SoI. As previously discussed, an enterprise may need to organize a number of projects to deliver synchronized changes. A program is a collection of projects dedicated to development and deployment of a complex SoS, or of a product-line family of systems.

Software engineers working on the development of complex systems need to be able to look at a particular project/program life cycle and be able to relate it back to the spectrum of life cycle approaches above, and the simple life cycle stages and process relationships in a relevant standard. Emerging lifecycle model generators such as the Incremental Commitment Model include teachable decision tables that aid in tailoring these generic approaches to real problems.[77]

Systems Engineering and Software Engineering Processes

System engineers work with stakeholders to identify and understand needs and constraints; and to develop requirements and system architecture, and allocate (and negotiate reallocation of) requirements to the hardware, software, and human elements of a system. In addition, system engineers oversee and coordinate development of the system components; they also oversee integration and verification of the components, system-level validation, and transition of the system into the operational environment. System engineers thus provide technical leadership of projects and programs that are concerned with developing complex systems.

Software engineers play a similar role for the software elements of a project, coordinating software requirements, design, and testing. Software engineers will also contribute to the SE activities and project decision-making. In larger projects, software and system engineers, working closely together, perform the two disciplines. In smaller project teams, individuals with both systems and software skills will be needed.

The coupling and iteration between SE, SwE, and project management can vary greatly for different types of problems and business sectors. At one extreme, a project may need to follow a highly sequential model of system design and software component development, with system design being fixed and software requirements allocated by the systems engineers as an input to software development. At the other extreme, the project may need to take a highly iterative

[77] Boehm, B. and Lane, J.A., "Process and Product Architectures and Practices for Achieving both Agility and High Assurance," *Proceedings of 31st International Conference on Software Engineering,* May 2009.

approach in which an individual or small team, with a mixture of systems, software, and other skills, work with stakeholders to determine the best solution; explore software implementation and operation issues through user involvement; and continue to evolve the solution to deal with a changing environment through its life.

Figure 6 defined the high-level process outcomes associated with each life cycle stage, and stated that this did not imply a one-to-one mapping between process and stages. Figure 7 is a graphical representation of the relationship between process activities and life cycle stages.

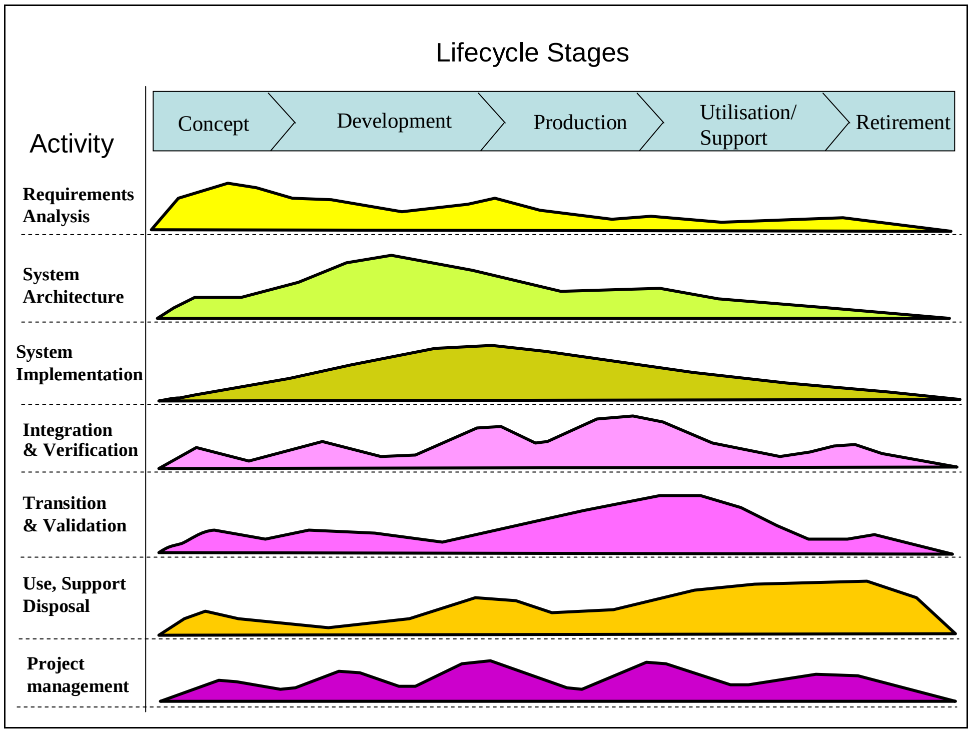


Figure 7. Life Cycle Activity Mapping (Adcock, 2009)

The lines on this diagram represent activity in each of the process areas over a simple life cycle. The life cycle stages along the top of the diagram are purely illustrative, and not drawn to scale. In a real project the Utilization/Support stages would likely be by far the longest part of the lifecycle. Requirements Analysis has a large input during the concept stage, but requirements are refined, reviewed, and reassessed over the rest of the life cycle. Similarly, Integration and Verification (I&V) is conducted during the transition from Development to Production. This is only possible if I&V issues, strategies, and risks are considered in earlier stages. Figure 7 is a schematic representation of these process mappings, sometimes called a "hump diagram."[78, 79]

[78] Kruchten, P., *The Rational Unified Process: An Introduction (3rd edition),* Addison Wesley, 2003.

The System Implementation line represents the hardware, software, human, and organizational changes needed to realize a system change. SwE activities are needed across the whole system life cycle, with the main software outcomes being achieved in the system development and production stages. With the increasing role that software has in integrating systems, and the use of COTS software and software services rather than custom-made code, software must be considered in parallel with system processes. Thus, COTS characteristics might need to be considered in deriving system requirements, making hardware design changes to solve software design problems, or configuring and testing software dynamically during operations and assessing the resulting system safety implications.

The following sections give a brief overview of the systems engineering technical processes described in (Haskins, 2007), the INCOSE handbook. The structure and process descriptions in the handbook are consistent with ISO/IEC 15288:2008 – Systems engineering – System life cycle processes. This standard defines a set of high-level process outcomes and generic activities that can be used as the basis for identifying the detailed activities needed within whichever life cycle framework is selected for a given problem and commercial context. Within this generic framework there is scope to apply specific tools and techniques taken from other standards or development methodologies as needed. The INCOSE handbook also gives guidance on how to tailor the lifecycle and process definitions to an organization, dealing with any conflicts with existing policies, procedures, and standards.[80]

Software engineers should be able to relate these high-level processes to a range of project lifecycle approaches within the spectrum described above. They should understand that the framework is not a one-size-fits-all, sequential process. Very few problems start with a clear and unchallenged customer need. Effective systems cannot be produced based on pre-specified requirements for software elements within the constraints of a top-down system design. The integration and testing of software and system elements, including human elements, is generally done iteratively with a strong stakeholder input as part of an incremental buildup of user benefits.

C.2.3. Requirements

The INCOSE handbook contains two requirements processes. One is stakeholder-focused, and defines need and measures of effectiveness. The second deals with one or more levels of technical requirements for a chosen solution. These processes are closely linked to the Architectural Design process in C.2.4.

---

[79] Boehm, B., and Lane, J.A., "Guide for Using the Incremental Commitment Model (ICM) for Systems Engineering of DoD Projects, Version 0.5," USC-CSSE Technical Report 2009-500, March, 2009, http://csse.usc.edu/csse/TECHRPTS/

[80] Fairley, R.E., *Managing and Leading Software Projects,* Wiley-IEEE Computer Society, 2009.

Stakeholder Requirements

The purpose of the Stakeholder Requirements Definition Process is to elicit, negotiate, document, and maintain stakeholders' requirements for the System of Interest within a defined environment. This is achieved by developing system views that concentrate on system purpose and behavior and are described in the context of the operational environment and conditions.

The outputs of this process consist of formally documented and approved stakeholder requirements that will govern the project. These will include required system capabilities, functions and/or services; quality standards; cost and schedule constraints; concept of operations and support. It is important to understand that stakeholder requirements do not follow the traditional Webster's dictionary definition of requirements as "something claimed or asked for by right and authority." Early stakeholder requirements are generally negotiable statements of wants, needs, goals, and objectives, which are refined and agreed on as our understanding of problems and potential solutions matures.

The activities to achieve the outcomes of this process across the life cycle fall into three groups:

- Capturing requirements, through a process of working with users and other stakeholders to generate a "context of use," and from it, extract needs and constraints.
- Structuring of the needs and constraints into a statement of stakeholder requirements. Documenting, reviewing, configuring, and sharing the requirements as needed.
- Using the requirements to support relevant system decisions across the life cycle.

The stakeholder requirements will form the basis for validating that a system service meets the needs of stakeholders. The process outputs should document the agreed context of use and include measures of effectiveness that will be used for assessing the realized system and enabling systems.

The terms "Stakeholder Requirements" and "context of use" are generic terms used in the INCOSE handbook. Other standards use similar ideas, such as "Operational Requirements," "User Requirements," "Concept of Operations," "Concept of Employment," and so forth.

There may be a number of software engineering stakeholders considered in generating stakeholder requirements. This may include representatives of legacy software systems within the wider SoI or understanding of the implication of the possible software issues raised by how the problem is framed.

Requirements Analysis

The purpose of the Requirements Analysis Process is to review, assess, prioritize, and balance all stakeholder and derived requirements (including constraints); and to transform those requirements into a functional and technical view of a system description capable of meeting the stakeholders' needs. This view can be expressed in a specification, set of drawings or any other means that provides effective communication.

The output of Requirements Analysis is a technical description of characteristics the future system must have in order to meet Stakeholder Requirements, which will be evolved into one or more specific solutions in subsequent development processes. The project team derives additional requirements resulting from analysis of the Stakeholder Requirements as required to meet project and design constraints; defines the functional boundaries for the system to be developed; and identifies and documents any interfaces and information exchange requirements with external systems. The total set of requirements encompasses the functional, performance, non-functional requirements, and the architectural constraints.

The activities to complete this process include requirements capture and structuring as before. The approaches, methods and tools to achieve this will be very similar to those defined in the software requirements section.

System requirements are a basis for verifying compliance of the realized system with its technical description. The process outputs should document relevant standards and interfaces through life constraints, utilization environment, and verification criteria.

Again, the requirements analysis process is a generic one that can be mapped onto any process that deals with requirements derived from solution choices. These requirements are also referred to as "Technical Requirements," "System Specifications," and so forth.

If the requirements are going to be used as the basis of an internal or external contract to develop a system within a given set of cost, schedule, safety, security, or other constraints, the Requirements Analysis activity should provide evidence that a system can be developed to satisfy such constraints.

C.2.4. System Design

System design is concerned with identifying the major hardware and software components of a system that will provide the features and quality attributes of the system in conjunction with the manual operations to be performed by humans. In addition, the interfaces among the hardware and software components and interfaces to and from the operational environment must be specified.

The INCOSE handbook includes an Architectural Design Process, which describes a system solution as a collection of manageable sub-problems, and an Implementation Process, which is really a placeholder for the relevant hardware, software, or human system design standards, as needed. The point at which system design identifies one or more bounded software-intensive system elements forms one of the key relationships between systems and software engineering. The overall system architecture also has a key role in the other key relationship between the two disciplines, the integration and verification of system components.

Architectural Design

The purpose of Architectural Design is to synthesize a solution that satisfies system requirements. To do this we define areas of solution expressed as a set of separate problems of manageable, conceptual, and, ultimately, realizable proportions.

The result of this process is an architectural design that is placed under configuration management. This baseline includes:

- System element detailed descriptions with documented justification for concept selections
- Requirements assigned to system elements and documented
- Interface requirements and a plan for system integration and verification strategy.

Architectural design forms the link between problem understanding, stakeholder value, and a coherent set of realizable solution elements (see Trade Studies below). The process may involve the generation, evaluation, and selection of solution options. The process outputs should document the evaluation process and the resulting logical system architecture; partitioning of system requirements to solution elements; and interfaces and interactions between elements, integration, and verification plans and all relevant decisions to reach an agreed design baseline. Again, if the architecture is to be used as a basis for system development and evolution within a set of resource constraints and performance requirements, the design deliverables should include evidence that a system developed to the specified architecture would be buildable within the resource constraints, and would satisfy the performance and evolution growth requirements.

As discussed in Section 6.5, the overlap between system design and software development is a critical one. The linear system development model, in which software components are bounded and specified by the software engineer and then designed, delivered, and integrated, is increasingly insufficient both for the generation of efficient software and the creation of flexible, agile, and resilient systems. For most system lifecycles, we would expect the selection of software components to be a critical driver on the system architecture, with the overall system design, software allocation and high-level software design decisions being may in parallel by a multi-skilled design team.

Implementation

The purpose of the Implementation Process is to design, create, or fabricate a system element conforming to that element's detailed description. The element is constructed employing appropriate technology and industry practices.

Initial development of requirements and identification of the major system components are accomplished in an iterative manner. Sub-system requirements are then allocated to the major hardware and software components and to the human elements. These requirements are typically at a high level and must be iterated and refined into derived requirements that provide detailed specifications for the hardware and software components and detailed qualifications for the human operators.

Hardware and software components can be obtained in a variety of ways: by purchase, by lease, by building in-house, by modifying existing components, by contracting, and (in the case of software) from open sources. In some cases the customer may provide components that are to be incorporated into the systems, either as-is or after modification.

The human elements of a system can be obtained by recruiting and training, or by retraining current employees of the acquiring organization.

Trade Studies

Trade study describes a process for comparing the appropriateness of different technical solutions. The characteristics of each option are traded against each other. Once a best alternative has been identified, the stakeholders in the decision will want to know how sensitive the recommended selection is to differently evaluated criteria or to different estimates of the alternatives' characteristics—perhaps a different best alternative would result. Therefore, a good trade study provides a disciplined process that justifies the selected approach, and includes sensitivity analysis.

Systems engineers may use trade studies to support any of the key decisions discussed above. This can include helping to identify the problems most important to a stakeholder; to explore their value propositions; and to identify and plan for solutions to deliver those values.

Conflicts among stakeholders' value propositions are frequent, and are built into the various stakeholder roles. Users would like many system capabilities, right away, with high levels of performance, reliability, and ease of use, with the option of frequently changing their minds about features and priorities. Acquirers have limited financial and other resources for developing and operating all the desired capabilities, and would like to keep the acquisition well defined and stable. Developers have limited capabilities to develop, verify, validate, and document large amounts of software on limited budgets and schedules. They would prefer to use their own tools and reusable components, even though these may be incompatible with other developers and the users' other applications. Maintainers want well tested and documented systems, with extensive support for backup, recovery, debugging, and version control, and systems and tools delivered that are compatible with the ones they are already maintaining. Other success-critical stakeholders such as the testers, interoperators, venture capitalists, administrators, and the general public may add further potential conflicts.

Systems engineers need the knowledge, skills, and abilities to collaborate with others to identify, analyze, and prioritize the potential conflicts, risks, and uncertainties among these stakeholder value propositions. They need to identify and evaluate options for resolving the conflicts, risks, and uncertainties, including development and evaluation of candidate architectures, processes, and operational concepts; assessment of reused, purchased, or subscribed-to services; and analysis of tradeoffs among various combinations of options. If none of the available options satisfy all of the stakeholders' value propositions, they need to communicate this to the stakeholders and help manage their expectations and prioritize capabilities to achieve a mutually satisfactory solution, or at least an initial operational capability achievable within the currently available resources. Increasingly, these skills involve a balance of satisficing and multi-criterion optimizing among multiple stakeholders rather than optimizing around a single criterion such as performance.

Trade-studies are also be used to identify overall solution concepts and key solution elements. One critical part of this is deciding on which system functions to allocate to hardware, software, and human system elements, and to understand the full cost and risk implications of design choices. These system trades will require software engineering knowledge and skills to help explore feasible solution directions and to understand the issues and constraints of different types of software-enabled solutions.

C.2.5. Integration and Verification (I&V)

The processes of system I&V take implemented and tested system elements and combine them to realize the SoI. These processes are strongly linked to the equivalent software processes, since some elements of software assessment may only be possible when combined with other system elements. For many software-intensive systems, a continuous integration approach, such as daily or weekly builds of partial unit-tested components, has become preferable to waiting for fully-realized components to be individually developed and unit-tested before beginning integration and test.

Both processes have a strong "through life" aspect, with issues, strategies, and plans for both being an essential part of the requirements and architecture process outcomes. Both may also require the creation of relevant enabling systems, facilities, instrumentation, information repositories, and so forth.

The relevant Planning and Configuration Management process to handle this will be very similar to that described in the CBOK.

<u>Integration</u>

The purpose of the Integration Process is to realize the System of Interest by progressively combining system elements in accordance with the architectural design requirements and the integration strategy.

This process confirms that all boundaries between system elements have been correctly identified and described, including physical, logical, and human-system interfaces and interactions (physical, sensory, and cognitive), and confirms that all functional, performance, and design requirements and constraints are satisfied. If a project is taking an incremental approach to satisfying stakeholder needs, all system configurations need to be tested.

<u>*Verification*</u>

The purpose of the Verification Process is to confirm that all requirements are fulfilled by the system elements and eventual System of Interest—that is, that the system has been built right. This process establishes the procedure for taking remedial actions in the event of non-conformance.

The process confirms that all elements of the SoIwhen working together perform their intended functions and meet the performance requirements allocated to them.

C.2.6. Transition and Validation

The processes of System Transition and Validation are more closely related to the satisfaction of stakeholder needs.

As above, these processes will have a strong "through life" aspect, with much of their impact being in the project strategy and planning activities. These processes deal with external project issues such as links to other projects, availability of customer-owned facilities, or strategies for fitting system release into operational tempo.

The conduct of both Transition and Validation in a particular sector will be constrained by the regulatory frameworks and business practices of that domain. This requires systems engineers and project managers to ensure that any technical activities needed can be properly managed and synchronized with non-technical deliverables.

<u>*Transition*</u>

The purpose of the Transition Process is to transfer custody of the system and responsibility for system support from one organizational entity to another. This includes (but is not limited to) transfer of custody from the development team to the organizations that will subsequently operate and support the system. Successful conclusion of the Transition Process typically marks the beginning of the Utilization Stage of the System of Interest.

The process installs a verified system in the operational environment along with relevant enabling systems, such as operator training systems, as defined in the agreement. As part of this process, the acquirer accepts that the system provides the specified capabilities in the intended operational environment prior to allowing a change in control, ownership, and/or custody.

<u>*Validation*</u>

The purpose of the Validation Process is to confirm that the realized system complies with the stakeholder requirements System validation is subject to approval by the project authority and key stakeholders.

This process is invoked during the Stakeholders Requirements Definition Process to confirm that the requirements properly reflect the stakeholder needs and to establish validation criteria—that is, that the right system has been built. This process is also invoked during the Transition Process to handle the acceptance activities.

C.2.7. Operation, Maintenance, and Disposal

The INCOSE handbook includes technical processes for system Operation, Maintenance, and Disposal. Much like the Implementation process, these processes are a placeholder to recognize that the system users and acquirers will need to have identified and acquired or created all of the Enabling Systems Plans and Resources necessary for the Utilization and Disposal life cycle stages. These processes also describe activities to ensure that systems engineers are aware of and constrained by utilization issues during early lifecycle stages, and that they plan for and integrate all enabling systems and external relationships across the system lifecycle. Recently, efforts have been made to emphasize that Disposal includes the option of planning and preparing for Recycling.

C.2.8. System Engineering of Software

Virtually all of the procedures and techniques of SE can be directly applied to software development. Many people have commented that SwE has more in common with system engineering than with other engineering disciplines. In both system and software development, requirements must be defined, analyzed, and documented (or refined if allocated from system requirements), a design specification must be synthesized, components must be implemented or otherwise obtained, and V&V must be applied throughout the development or modification cycle to ensure the feasibility of proceeding forward, and to find and fix defects in the stage where they are inserted. Both rely on supporting disciplines such as configuration management and user interface design. In some cases, SwE has contributed new approaches to SE: for example, context diagrams, the Concept of Operations, use cases, activity diagrams, and object-oriented design.

Software system engineers play the role of system engineers within the more limited context of software development and modification: they work with users, customers, acquirers, and other stakeholders to identify, analyze, and prioritize operational requirements; they translate operational requirements into technical specifications; they identify the major software system components and allocate requirements to them; they work with software component specialists(e.g., the user interface, database, telecommunications, and algorithm specialists); they work with specialists in disciplines such as safety, security, reliability, usability, configuration management, and quality assurance; and they oversee delivery and installation of software system elements.

It is evident that strong synergy exists between SE and SwE. Increasingly it is useful to think of a software engineer as a systems engineer with a specialist skill in software, rather than as a supplier of bounded software sub-systems. The strong relationship between the two disciplines will become increasingly important as the needs and desires of modern society result in demands for larger, more complex, and more dynamic information-intensive systems, for which software is the preferred medium for adapting to rapid change.

C.3. Engineering Economics

Engineering Economics is about making engineering decisions in a business context[81]. It means aligning technical decisions with the business goals of the organization. Decisions such as “Should we use Extreme Programming or should we use Scrum on this project?” may be easy decisions from a purely technical perspective, but those decisions can have serious implications on the business viability of an engineering project and the resulting product.

C.3.1. Proposals

Making a business decision begins with the notion of a *proposal*. A proposal is a single, separate option that is being considered. For example, carrying out a particular software development project or not, or enhancing an existing program versus redeveloping that same software from scratch. Each proposal represents a unit of choice—choose to carry out a proposal or choose not to. The purpose of business decision-making is to decide, given the current business circumstances, which proposals should be carried out and which should not.

C.3.2. Cash Flow

To make a meaningful business decision about any specific proposal, the proposal must be evaluated from a business perspective. The concepts of cash flow instances and cash flow streams are used to describe the business perspective of a proposal. A *cash flow instance* is a specific amount of money flowing into or out of the organization at a specific time as a direct result of some proposal.

The term *cash flow stream* refers to the set of cash flow instances, over time, which would be caused by carrying out some given proposal. The cash flow stream is, in effect, the complete financial picture of that proposal.

A *cash flow diagram* is a picture of a cash flow stream. The cash flow diagram provides the reader a concise overview of the financial picture of that subject. Figure 8 shows an example cash flow diagram for a proposal.

A cash flow diagram shows the cash flow stream in two dimensions: time runs horizontally, from left to right, and amounts of money run vertically, up and down. Each cash flow instance is drawn on the diagram at a left-to-right position relative to the timing of that cash flow after the start of the proposal. The horizontal axis is divided into units of time that represent years, months, weeks, or other unit as appropriate for the proposal being studied.

[81] Tockey, S., *Return on Software: Maximizing the Return on Your Software Investment (1st edition).* Addison-Wesley Professional, 2004.

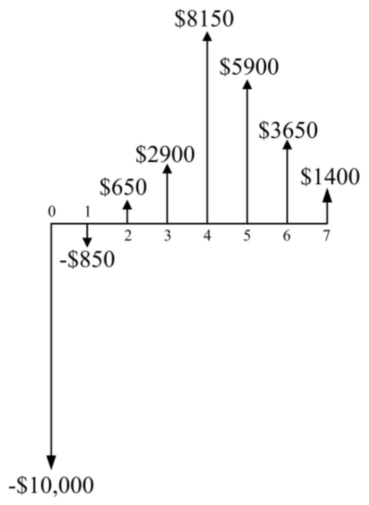


Figure 8. A Cash-Flow Diagram

The Business Decision-Making Process

When evaluating the optimum solution to a problem, both technical issues and business criteria (cost and income) must be considered when make such decisions. Figure 9 depicts a systematic process that could be used to make such decisions.

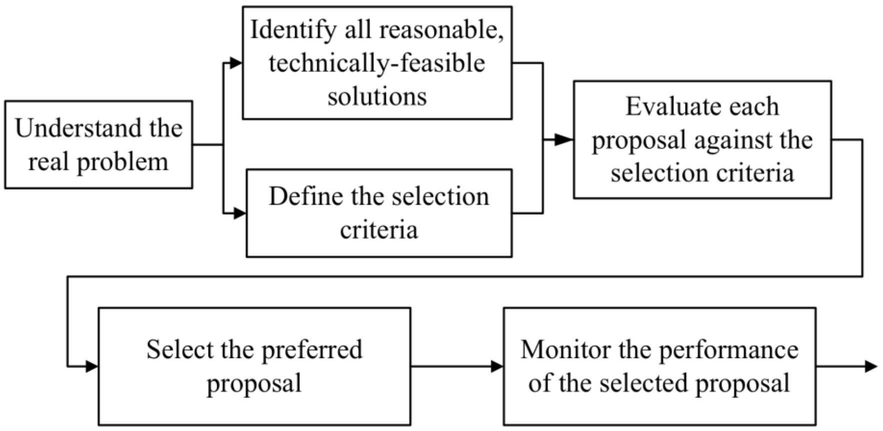

Understand the real problem
Identify all reasonable, technically-feasible solutions
Define the selection criteria
Evaluate each proposal against the selection criteria
Select the preferred proposal
Monitor the performance of the selected proposal

Figure 9. A Business Decision-Making Process

The Time-Value of Money

A fundamental concept in business decision-making is that money has *time-value*: that is, the value of money, because of economic conditions such as inflation, changes over time.

Equivalence

Due to the time-value of money, two or more cash flows are *equivalent* when they equal the same amount of money at a common point in time. Comparing cash flows only makes sense when they are expressed in the same time frame.

Bases for Comparison

A basis for comparison is a common frame of reference for comparing two or more cash flows. It is a way of using equivalence to meaningfully compare proposals. Several bases of comparison are available, including:

- Present worth
- Future worth
- Annual equivalent
- Internal rate of return (IRR)
- (Discounted) Payback period

Mutually Exclusive Alternatives

When an organization is considering carrying out multiple proposals at the same time, the interrelationships and dependencies between proposals can make decision-making complex and risky. Such a situation can be simplified by restructuring and reorganizing the set of proposals into an alternate, but equivalent, set of mutually exclusive proposals.

C.3.3. For-Profit Decision-Making

For-Profit Decision Analysis

Figure 10 describes the process for identifying the best alternative from a set of mutually exclusive alternatives for for-profit organizations.

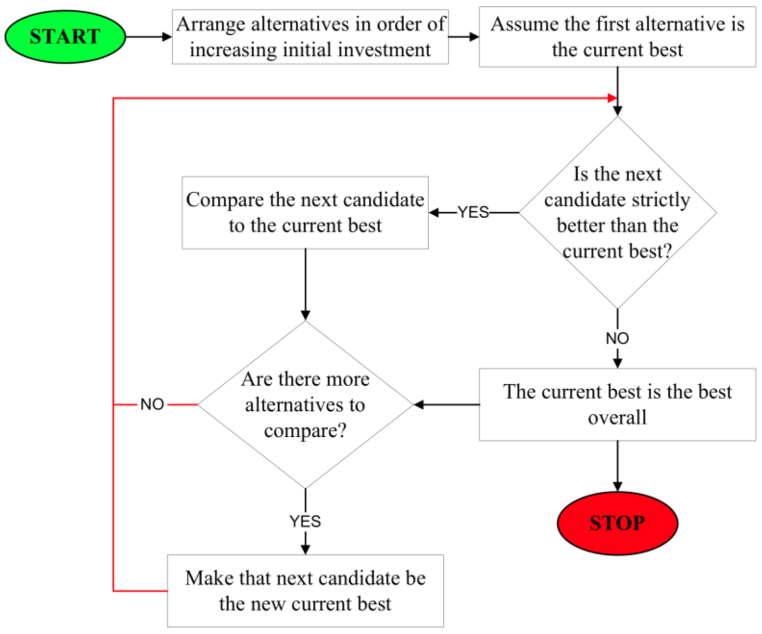

START
Arrange alternatives in order of increasing initial investment
Assume the first alternative is the current best
Is the next candidate strictly better than the current best?
YES
Compare the next candidate to the current best
NO
The current best is the best overall
STOP
Are there more alternatives to compare?
NO
YES
Make that next candidate be the new current best

Figure 10. A For-Profit Decision-Making Process

Minimum Attractive Rate of Return

An organization's *minimum attractive rate of return* (MARR) is the lowest IRR the organization would consider to be a good investment. The MARR is a statement that an organization is confident it can achieve at least that rate of return. It represents an organization's *opportunity cost* for investments. An alternative proposal's present worth evaluated at the MARR shows how much more or less (in present-day cash terms) that alternative is worth than investing at the MARR.

Economic Life

When an organization chooses to invest in a particular proposal, money is bound to that proposal—the money is called "frozen assets." The economic impact of frozen assets ("capital recover with return") tends to start high and decrease over time. On the other hand, operating and maintenance costs of elements associated with the proposal tend to start low but increase over time. The total cost of owning and operating a proposal is the sum of those two costs. Early on, frozen asset costs dominate, and later the operating and maintenance costs dominate. There is a point in time where the sum of the costs is minimized—the economic life, or minimum cost lifetime.

Planning Horizon

The *planning horizon*, also known as the study period, is the consistent time frame over which proposals are considered. Effects such as economic life and the time frame over which reasonable estimates can be made must be factored into establishing a planning horizon. Once the planning horizon is established, several techniques are available for placing proposals with different life spans into the planning horizon.

Replacement and Retirement Decisions

A *replacement decision* is a special case of for-profit decision analysis that happens when an organization already has a particular asset and they are considering replacing it with something else (for instance, replacing a legacy software system with a new application). Replacement decisions use the same decision process but there are additional challenges such as sunk cost and salvage value. A *retirement decision* is concerned with ceasing a current activity, such as when a software company decides to cease selling a software product, or a hardware manufacturer decides to stop building and selling a particular model of computer.

Inflation

Inflation describes long-term trends in prices. If the planning horizon of a business decision is longer than a few years, or if the inflation rate is significant, it can cause noticeable changes in the value of a proposal.

Depreciation

Depreciation addresses how investments in capital assets are charged against income over several years. Depreciation is an important part of after-tax cash flows, which is critical to accurately addressing income taxes. Software itself typically isn't depreciated, but if a proposal has a planning horizon longer than one year, the proposal involves capital assets, and there is a need to accurately reflect the effects of income taxes in the decision analysis, then depreciation is an important analysis factor. Depreciation is also useful when comparing software proposals with non-software proposals.

General Accounting and Cost Accounting

The primary role of *general accounting* is to measure a company's actual financial performance. *Cost accounting* is a specialized branch of general accounting that is used to find the cost of providing the products and services that were sold. Accounting systems are also a rich source of historical data for estimating.

Income Taxes

In the U.S., the federal government and most states charge income taxes, which combined can amount to between 20% and 40% of a corporation's net profit. In some areas, federal, state, and local income taxes can add up to more than 50%. Across the globe, each country establishes its own tax policies. Income tax accounting must be part of business decisions about proposal acceptance and profitability.

C.3.4. Not-for-Profit Decision-Making

The for-profit decision techniques discussed in Section C3.3 do not apply when an organization does not have a profit goal, which is the case in government and in non-profit organizations. In these situations a different set of decision techniques are needed.

Benefit-Cost Analysis

Benefit-cost analysis is a widely used method for evaluating proposals in non-profit organizations. Any proposal with a benefit-cost ratio of less than 1.0 can usually be rejected without any further analysis because it would cost more than it would benefit.

Cost-Effectiveness Analysis

Cost-effectiveness analysis shares similar philosophy and methodology with benefit-cost analysis. There are two versions of cost-effectiveness analysis. The *fixed cost* version maximizes the benefit given some upper bound on cost. The *fixed effectiveness* version minimizes the cost needed to achieve a fixed goal.

C.3.5. Present Economy

Break-Even Analysis

Given functions describing the costs of two or more proposals, *break-even analysis* helps in choosing between them by identifying points where the cost functions are equal. Below a break-even point, one proposal is preferred and above that point, the other is preferred.

Optimization Analysis

The typical use of *optimization analysis* is to study a cost function over a range of values to find the point where overall performance is optimal. Software's classic space-time tradeoff is an example of optimization (e.g., an algorithm that runs faster will typically use more memory).

C.3.6. Estimation, Risk, and Uncertainty

Estimation Techniques

Four families of estimation techniques exist:

- Expert judgment
- Analogy
- Decomposition
- Statistical (or parametric) methods

Addressing Uncertainty

Estimates are inherently uncertain and that uncertainty should be addressed in business decisions. Techniques for addressing uncertainty include:

- Consider ranges of estimates

- Sensitivity analysis
- Delay final decisions

Decisions under Risk

Decisions under risk techniques are used when the decision-maker can assign probabilities to the different possible outcomes. The specific techniques include:

- Expected value decision-making
- Expectation variance and decision-making
- Monte Carlo analysis
- Decision trees
- Expected value of perfect information

Decisions under Uncertainty

Decisions under uncertainty techniques are used when the decision-maker cannot assign probabilities to the different possible outcomes. The specific techniques include:[82]

- Laplace Rule
- Maximin Rule
- Maximax Rule
- Hurwicz Rule
- Minimax Regret Rule

C.3.7. Multiple Attribute Decisions

Most of the topics discussed earlier in Section C.3 are used to make decisions based on a single decision criterion, money. The alternative with the best present worth, the best incremental IRR, the best incremental benefit-cost ratio, and so forth, is the one selected. Aside from technical feasibility, money is almost always the most important decision criterion, but it is not always the only one. Quite often there are other criteria, other "attributes," that need to be considered and those attributes can't be cast in terms of money. *Multiple attribute decision* techniques allow other, non-financial criteria to be factored into the decision.

Value and Measurement Scales

In an abstract sense, the decision making process—be it a financial decision or not—is about maximizing value. The alternative that maximizes total value is chosen (e.g., whether an item is a "name brand" or not can significantly affect its perceived value). Relevant values that cannot

[82] Jordaan, I., *Decisions under Uncertainty: Probabilistic Analysis for Engineering Decisions,* Cambridge University Press, 2005.

be expressed in terms of money need to be measured. A number of measurement scales are available and the scale chosen can limit the kinds of manipulations allowed by that measurement:

- Nominal scales
- Ordinal scales
- Interval scales
- Ratio scales

Compensatory and Non-Compensatory Techniques

There are two families of multiple attribute decision techniques, which differ in how they use the attributes in the decision. One family is the "compensatory," or single-dimensioned, techniques. This family collapses all of the attributes onto a single figure of merit. The family is called compensatory because, for any given alternative, a lower score in one attribute can be compensated by—or traded off against—a higher score in other attributes. The compensatory techniques include:

- Non-dimensional scaling
- Additive Weighting
- Analytic Hierarchy Process (AHP)

In contrast, the other family is the "non-compensatory," or fully dimensioned, techniques. This family does not allow tradeoffs among the attributes. Each attribute is treated as a separate entity in the decision process. The non-compensatory techniques include:

- Dominance
- Satisficing
- Lexicography

Appendix D. Security in the Software Life Cycle

Security has become a widespread and significant issue in the development of software systems. Although the current SWEBOK does not have a specific KA or section devoted to software security, it does have numerous references, throughout the KA chapters, to methods and practices that support the development of secure software systems: requirements, design, construction, testing, maintenance, and so forth. The 2010 revision of the SWEBOK will strengthen the discussion of security throughout the SWEBOK and add a supplementary KA on Software Security. Table 6 points to areas and locations, within the SWEBOK, where security is a relevant issue. The table is not a complete specification of security issues and their relation to the SWEBOK, but rather is intended to illustrate how the SWEBOK KAs are and will be related to security curriculum issues. Additional security issues could be included: further elaboration of the KAs listed and additional KAs such as in maintenance, configuration management, and quality.

Curriculum-related security issues are explored more fully in (Redwine, 2007)[83], which provides a comprehensive guide to software security issues and is similar in style and nature to the SWEBOK. Redwine provides detailed guidance to those who wish to design a GSwE2009 curriculum that focuses on software security; as it states, "The primary audiences for this guide are educators and trainers who can use this guide to help identify both appropriate curricular content and references that detail it." In Table 6, the column headed "Security Topics" uses topics cited in (Redwine, 2007).

Table 6. Software Security and the SWEBOK

Knowledge Area	Security Topics	SWEBOK
Requirements Engineering	Description and classification of "security requirements"	Chapter 2: Sections 1.2, 4.1
	Analysis of security-related needs and expectations	Chapter 2: Sections 3.2, 4.2, 4.3
	Validation of security requirements	Chapter 2: Sections 6.1, 6.2, 6.3
Software Design	Analysis of security data flows and fault tolerance	Chapter 3: Section 2.4, 5.2
	Invocation and utilization of security functionality	Chapter 3: Section 2.2

[83] Redwine, S. T. Jr. (Ed.), *Software Assurance: A Curriculum Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software,* Draft Version 1.2. U.S. Department of Homeland Security, 2007, https://buildsecurityin.us-cert.gov/daisy/bsi/940-BSI/version/default/part/AttachmentData/data/Curriculum GuideToTheCBK.pdf

Knowledge Area	Security Topics	SWEBOK
	Distribution and allocation of software functionality	Chapter 3: Sections 2.3, 3.1
	Analysis of design vulnerability	Chapter 3: Section 4.2
	Design review of security attributes	Chapter 3: Section 4.2
Software Construction	Selection of construction tools and usage standards	Chapter 4: Sections 1.2, 3.2
	Usage of security best design and coding practices, with knowledge of common vulnerabilities	Chapter 4: Sections 3.1, 3.3, 3.4, 3.5
	Performance of security-oriented review and testing	Chapter 4: Sections 3.2, 3.5
Testing	Description of security testing levels	Chapter 5: Section 2.1
	Testing security functionality	Chapter 5: Sections 3.2, 3.6, 3.7
	Attack and penetration testing	Chapter 5: Sections 3.4, 3.6, 3.7
	Classification of security defects	Chapter 5: Section 4.1.2

Appendix E. GSWE2009 Outcomes CBOK Mapping

A mapping of the ten GSwE2009 Outcomes to the CBOK is shown in Table 7. The mapping clarifies where CBOK alone falls short of achieving the outcomes, highlighting the importance of the 50% of the curriculum that is not covered by CBOK. A course that addresses CBOK material could include additional material, concepts, case studies, pedagogical methods, and other facets that substantially address an outcome. However, those additional facets are program-specific and not inherent in the CBOK content. Each outcome is rated *high*, *medium*, *low*, or *none* for how well the CBOK addresses the outcome. For example, a rating of *high* would mean that the CBOK—with its designated Bloom levels—fully supports the outcome; *medium*, *low*, and *none* describe decreasing levels of support of the outcome within CBOK. These ratings are subjective, but are substantiated by the observations column in the table.

Table 7. GSwE2009 Outcomes-CBOK Mapping Table

Outcomes	Supporting Knowledge Areas	Supporting Topics in Knowledge Areas	How Well CBOK Addresses Outcome	Observations
CBOK	All	All	High	By definition, the CBOK addresses this outcome fully.
DOMAIN	All	All	Low	There is no requirement in the CBOK to learn any domain in depth. However, almost any pedagogical approach to teaching the CBOK will cover at least one domain to a minimal level of proficiency.
DEPTH	Any	All	Medium	The CBOK does not require Bloom's Synthesis level. However, the CBOK coverage of such areas as Requirements Analysis to the AN level takes a student significantly towards the Synthesis level.
ETHICS	A. Ethics and Professional Conduct	All	High	The Ethical and Professional Conduct KA specifically addresses this outcome.

Outcomes	Supporting Knowledge Areas	Supporting Topics in Knowledge Areas	How Well CBOK Addresses Outcome	Observations
SYS ENG	Primarily B. System Engineering, but also aspects of the other KAs, especially A. Ethics and Professional Conduct, C. Requirements Engineering, F. Testing, H. Configuration Management. and I. Software Engineering Management	A.1. Social, legal, and historical issues directly supports A.2. Codes of ethics and professional conduct is directly supporting B, C, F, and H topics at least touch on SE I.3. Risk Management directly supports	Medium	Systems Engineering proficiency is required at the Bloom C and AP levels in Table 2. There are many opportunities to incorporate SE into courses when teaching requirements, architecture, and other CBOK topics.
TEAM	A. Ethics and Professional Conduct I. Software Engineering Management Others as needed to demonstrate leadership in a technical area	A.1. Social, legal, and historical issues A.2. Codes of ethics and professional conduct I.3. Software Project Organization and Enactment	Medium	Working in teams is pedagogically straightforward and will typically be covered in courses that teach CBOK material. However, teaching about *"multinational communication and geographically distributed"* teams will be much more challenging for many programs. In order to "lead in one area of project development" the student will need to master that area beyond the CBOK recommended level.

Outcomes	Supporting Knowledge Areas	Supporting Topics in Knowledge Areas	How Well CBOK Addresses Outcome	Observations
RECONCILE	A. Ethics and Professional Conduct C. Requirements Engineering I. Software Engineering Management	A.1. Social, legal, and historical issues A.2. Codes of ethics and professional conduct C.3. Initiation and Scope Definition I.1. Software Project Planning I.2. Risk Management I.3. Software Project Organization and Enactment I.7. Engineering Economics	Medium	There is no KA or topic specifically tied to "*reconcile conflicting project objectives, finding acceptable compromises within limitations of cost, time, knowledge, risk, existing systems, and organizations.*" However, aspects of Ethics, Requirements Engineering and Software Engineering Management should cover this in part.
PERSPECTIVE	C. Requirements Engineering I. Software Engineering Management	C.3. Initiation and Scope Definition C.4. Requirements Elicitation I.3. Software Project Organization and Enactment	Medium	The capstone project and related presentations reinforce communication and leadership.
LEARN	None	None	Low	No specific KA or topic related to this outcome. The capstone and class project could cover aspects related to learning new models and technologies.
TECH	None	*None*	Low	No KA or topic related to this outcome. Elective courses, capstone project and class projects could cover analyzing and testing new technologies.

This page intentionally left blank.

References

ACM Council, *ACM Code of Ethics and Professional Conduct*, October 1992, http://www.acm.org/constitution/code.html.

Shackelford, R., et al., *Computing Curricula: 2005 Overview Report*, ACM, 2006. http://www.acm.org/education/curricula-recommendations.

ACM/IEEE-CS Joint Task Force on Computing Curricula, *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*, August 2004, http://www.acm.org/education/curricula-recommendations.

ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices, *Software Engineering Code of Ethics and Professional Practice*, Version 5.2, 1999, http://www.acm.org/about/se-code/.

Ardis, M. and Ford, G., *SEI Report on Graduate Software Engineering Education*, CMU/SEI 89-TR-21, Software Engineering Institute, Carnegie Mellon University, June 1989.

Australian Computer Society, "ACS Code of Ethics" website, 2008, http://www.acs.org.au/index.cfm?action=show&conID=200509022322219027.

Joint Declaration of the European Ministers of Education, " The European Higher Education Area," Convened in Bologna on June 19, 1999.

Beck, K. and Andres, C., *Extreme Programming Explained: Embrace Change (2nd edition)*, Addison-Wesley Professional, 2004.

Bloom, B. S. (Ed.), *Taxonomy of educational objectives: The classification of Educational goals: Handbook I, cognitive domain*, Longmans, 1956.

Boehm, B., *Software Engineering Economics*, Prentice Hall, 1981.

Boehm, B., "Anchoring the Software Process", *IEEE Software*, July 1996, 73-82.

Boehm, B., and Lane, J.A., "Guide for Using the Incremental Commitment Model (ICM) for Systems Engineering of DoD Projects, Version 0.5," USC-CSSE Technical Report 2009-500, March 2009, http://csse.usc.edu/csse/TECHRPTS/.

Boehm, B, and Lane, J. A., "Process and Product Architectures and Practices for Achieving both Agility and High Assurance," *Proceedings of 31st International Conference on Software Engineering*, May 2009.

Bott, F., et al., *Professional Issues in Software Engineering (3rd edition)*, Taylor & Francis, 2001.

Bourque, P., *SWEBOK Refresh and Continuous Update: A Call for Feedback and Participation*, Conference on Software Engineering Education & Training (CSEET), 2009, pp. 288-289.

British Computer Society, *Code of Conduct & Code of Good Practice*, 2004 and 2006, http://www.bcs.org/server. php?show=nav.10967.

Brooks, F., *The Mythical Man-Month: Essays on Software Engineering (2nd ed.),* Addison-Wesley Professional, 1995.

Checkland, P., *Systems Thinking, Systems Practice* (2nd ed.). Wiley, 1999.

Chrissis, M. B., Konrad, M., and Shrum, S., *CMMI®: Guidelines for Process Integration and Product Improvement (1st edition),* Addison-Wesley Professional, 2003.

CMMI Product Team, *Capability Maturity Model, Version 1.1, CMMI for Software Engineering, Staged Representation*, CMU/SEI-2002-TR-029. Software Engineering Institute, Carnegie Mellon University, August 2002.

Paulk, M., et al., *Software Capability Maturity Model*, Version 1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1993.

Computer Society India, *CSI Code of Ethics*, 2008. http://www.csi-india.org/code-ethics.

Cusumano, M., et al., "Software Development Worldwide: The State of the Practice", *IEEE Software*, November/December 2003.

Dijkstra, E., "Software Engineering: As It Should Be", conference paper, *International Conference on Software Engineering 4*, September 1979, 442-448. See also EWD 791 at http://www.cs.utexas/users/EWD.

European Commission, Education & Training, "European Credit Transfer and Accumulation System (ECTS)" website. http://ec.europa.eu/education/programmes/socrates/ects/index_en.html#1

Fairley, R. E., "A Post-Mortem Analysis of the Software Engineering Programs at Wang Institute of Graduate Studies," *Issues in Software Engineering Education*, Springer Verlag, 1988.

Fairley, R. E., *Managing and Leading Software Projects*. Wiley-IEEE Computer Society, 2009.

Flood, R. L. and Carson, E.R., *Dealing with Complexity* (2nd edition). Plenum Press, 1993.

Ford, G. and Gibbs, N. E., *A Mature Profession of Software Engineering*, CMU/SEI-96-TR-004, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1996.

Ford, G., *SEI Report on Graduate Software Engineering Education,* CMU/SEI 91-TR-002, Software Engineering Institute, Carnegie Mellon University, April 1991.

Forrester, J., "Learning through system dynamics as preparation for the 21st century", Keynote address for the *Systems Thinking and Dynamic Modeling Conference for K-12 Education*, 1994.

GAO, *Defense Acquisitions: Assessment of Selected Major Weapons Programs*, U.S. Government Accountability Office, GAO-08-467SP, March 2008.

GAO, *Information Technology: Inconsistent Software Acquisition Processes at the Defense Logistics Agency Increase Project Risks*, U.S. Government Accountability Office, GAO-02-9, January 2002.

GAO, *Information Security: Agencies Face Challenges in Implementing Effective Software Patch Management Processes*, U.S. Government Accountability Office, GAO-04-816T, June 2004.

Glass, R. L., *Computing Calamities: Monumental Computing Disasters*, Prentice Hall Professional Technical Reference, 1998.

Glass, R. L., *Facts and Fallacies of Software Engineering*, Pearson Education, 2002.

Glass, R. L., *Software Runaways: Monumental Software Disasters*, Prentice Hall Professional Technical Reference, 1997.

Haskins, C. (Ed.), *INCOSE Systems Engineering Handbook*, Version 3.1, INCOSE-TP-2003-002-03.1, August 2007.

Huitt, W. (2006), "The cognitive system", *Educational Psychology Interactive*, Valdosta, GA: Valdosta State University. Retrieved May 22, 2008 from http://chiron.valdosta.edu/whuitt/col/cogsys/cogsys.html.

IEEE STD 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Computer Society, 1990.

INCOSE, *Guide to Systems Engineering Body of Knowledge*, International Council on Systems Engineering, 2004.

ISO (International Standards Organization), *Standard for Information Technology – Software Life Cycle Processe*,. ISO/IEC 12207, 1995.

ISO (International Standards Organization), *Systems and Software Engineering – System Life Cycle Processes*. ISO/IEC 15288, 2008.

Jain, R., and Verma, D., *A Report on Curriculum Content for a Graduate Program in Systems Engineering: A Proposed Framework*. INCOSE International Symposium, 2007.

Jones, C., "Variations in Software Development Practices", *IEEE Software 20(6)*, November/December 2003, 22-27.

Jordaan, I., *Decisions under Uncertainty: Probabilistic Analysis for Engineering Decisions*, Cambridge University Press, 2005.

Kaner, C., "Issues in Commercial Law of Interest to Software Engineering Educators," tutorial session at the *Conference on Software Engineering Education & Training (CSEE&T)*, February 2002.

Kemper, J. D., and Sanders, B. R., *Engineers and Their Profession* (5th Ed.), Oxford University Press, 2000.

Koskela, L., and Howell, L., "The Underlying Theory of Project Management is Obsolete", *Proceedings of the 2002 PMI Research Conference*, 2002, 293-302.

Kroll, P. and Kruchten, P., *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*, Addison-Wesley Professional, 2003.

Kruchten, P., *The Rational Unified Process: An Introduction (3rd edition)*, Addison Wesley Professional, 2003.

Leveson, N. G., *Safeware: Systems Safety and Computers*, Addison-Wesley Professional, 1995.

Leveson, N. G., "The Role of Software in Spacecraft Accidents", *AIAA Journal of Spacecraft and Rockets, 41*(4), July 2004.

Loui M.C. and Miller, K.W., "Ethics and Professional Responsibility in Computing", in *Encyclopedia of Computer Science and Engineering*, edited by Benjamin Wah, Wiley, 2009.

Maier, M. W, "System and Software Architecture Reconciliation," *Systems Engineering, 9*(2), Summer 2006, 146-159.

Moore, J. W., *The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society, 2006.

Project Management Institute, *A Guide to the Project Management Body of Knowledge (PMBOK® Guide)*, 2006.

Pyster, A., Turner, R., Henry, D., Lasfer, K., Bernstein, L. and Baldwin, K., "The Current State of Software Engineering Master's Degree Programs," *Conference on Software Engineering Education and Training*, 2008, 103-109.

Pyster, A., Lasfer, K., Turner, R., Bernstein, L. and Henry, D., "Master's Degrees in Software Engineering: An Analysis of 28 University Programs," *IEEE Software*, September-October 2009, 94-101.

Redwine, S. T., Jr. (Ed), *Software Assurance: A Curriculum Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software*, Draft Version 1.2. U.S. Department of Homeland Security, 2007. https://buildsecurityin.us-cert.gov/daisy/bsi/940-BSI/version/default/part/AttachmentData/data/CurriculumGuideToTheCBK.pdf

Royce, W. E., *Software Project Management: A Unified Framewok*, Addison-Wesley Professional, 1998.

Senge, P., *The Fifth Discipline: The Art and Practice of the Learning Organization*, Doubleday, 2006.

Shaw, M. (Ed.), *Software Engineering for the 21st Century: A Basis for Rethinking the Curriculum*, Technical Report CMU-ISRI-05-108, Carnegie Mellon University Institute for Software Research, March 2005.

Shaw, M., "Prospects for an Engineering Discipline of Software," *IEEE Software*, *7*(6), November 1990, 15-24.

Standish Group, Unfinished Voyages, http://www.standishgroup.com/sample_research/unfinished_voyages_1.php, October 19, 2005.

Sterman, J., *Business Dynamics: Systems Thinking and Modeling for a Complex World.* McGraw-Hill/Irwin, 2000.

SWEBOK, *Guide to the Software Engineering Body of Knowledge*, P. Bourque and R. Dupuis (Eds.). IEEE Computer Society Press, 2004.

Tavani, H. T., *Ethics & Technology: Ethical Issues in an Age of Information and Communication Technology*, Wiley, 2003.

Thompson, J.B., "Perspectives on Software Engineering Professionalism", *Wiley Encyclopedia of Computer Science and Engineering*, edited by Benjamin Wah, Wiley, 2009.

Tockey, S., *Return on Software: Maximizing the Return on Your Software Investment (1st edition)*. Addison-Wesley, 2004.

Uniform Computer Information Transactions Act. National Conference of Commissioners on Uniform State Laws, 2001.

U.S. Department of Defense, *MIL-STD-1521B: Technical Reviews and Audits for Systems, Equipments, and Computer Software*, 1985.

U.S. Department of Defense, *DOD-STD-2167A: Defense System Software Development*, 1988.

Womack, J. P., Jones, D. T., and Roos, D., *The Machine that Changed the World: The Story of Lean Production.* Harper Perennial, 1991.

This page intentionally left blank.

Glossary

Abbreviations

ABET	Accreditation Board for Engineering and Technology
ACM	Association for Computing Machinery
ACS	Australian Computer Society
AHP	Analytic Hierarchy Process
AN	Analysis level in Bloom's taxonomy
ANSI	American National Standards Institute
AP	Application level in Bloom's taxonomy
ASEE	American Society for Engineering Education
ATM	Automated Teller Machine
BCS	British Computer Society
BS	Bachelor of Science
BSCE	Bachelor of Science in Computer Engineering
BSCS	Bachelor of Science in Computer Science
BSEE	Bachelor of Science in Electrical Engineering
BSSE	Bachelor of Science in Systems Engineering
C	Comprehension level in Bloom's taxonomy
CAT	Curriculum Author Team
CBD	Component-Based Design
CBOK	Core Body of Knowledge
CM	Configuration Management

CMMI	Capability Maturity Model Integrated
COTS	Commercial Off-the-Shelf
CS	Computer Science
CSEET	Conference on Software Engineering Education and Training
CSI	Computer Society India
DoD	United States Department of Defense
E	Evaluation level in Bloom's taxonomy
ECTS	European Credit Transfer and Accumulation System
EIA	Electronic Industries Alliance
EST	Early Start Team
GPA	Grade Point Average
GPS	Global Positioning System
GSwE2009	Graduate Software Engineering 2009: Curriculum Guidelines for Graduate Degree Programs in Software Engineering
GSwERC	Graduate Software Engineering Reference Curriculum
I&V	Integration and Verification
ICSE	International Conference on Software Engineering
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IEEE-CS	Institute of Electrical and Electronics Engineers – Computer Society
INCOSE	International Council on Systems Engineering
IP	Intellectual Property
IRR	Internal Rate of Return
ISO	International Standards Organization

iSSEc	Integrated Software and Systems Engineering Curriculum
K	Knowledge level in Bloom's taxonomy
KA	Knowledge Area
MARR	Minimum Attractive Rate of Return
MS	Master of Science
MSE	Master of Software Engineering
NDIA	National Defense Industrial Association—Systems Engineering Division
NITRD	Networking and Information Technology Research and Development Program—specifically, the National Coordination Office for NITRD
OSD	Office of the Secretary of Defense
***PMBOK*®**	Project Management Body of Knowledge
RFP	Request For Proposals
S	Synthesis level in Bloom's taxonomy
SE	Systems Engineering
SE2004	Software Engineering 2004, the ACM/IEEE Computer Society reference curriculum for an undergraduate degree in software engineering, published in 2004
SEI	Software Engineering Institute
SoI	System of Interest
SoS	System of Systems
SwE	Software Engineering
SWEBOK	The IEEE 's Software Engineering Body of Knowledge, published in 2004
SYS	Systems engineering content in CBOK
S2ESC	IEEE–CS Software and Systems Engineering Standards Committee
UCITA	Uniform Computer Information Transactions Act

UML	Unified Modeling Language
V&V	Verification and Validation

Terms

Admission Requirements. Admission requirements are the minimum standards an individual must meet in order to enter an academic program. These requirements are generally mandatory, and waivers require justification. Admission requirements are not specified in GSwE2009. (See *Entrance Expectations*)

Architecture. Architecture refers to the framework used to develop software, which is specifically covered in the Core Body of Knowledge. (For information on the architecture of GSwE2009, please see *Curriculum Architecture.*)

Bloom Taxonomy. A categorization of the intellectual activities associated with learning. The taxonomy has six levels of activity: Knowledge (K), Comprehension (C), Application (AP), Analysis (AN), Synthesis (SYN), and Evaluation (EV). These levels are used to describe the depth to which curricula should cover specific elements in the Core Body of Knowledge (CBOK). The GSwE2009 Curriculum is focused primarily at the K, C, AP, and AN levels, with recommendation of SYN level understanding in an elective area. (Please see Appendix B for more information.)

Bridging Course. See *Leveling Course.*

Capstone Experience. A detailed and work-intensive endeavor that demonstrates the application of knowledge and skills gained in a program to a specific problem. Capstone projects have traditionally been in the form of a thesis. More recently, capstone projects that handle problems relevant to a particular industry segment or area of expertise and develop potential solutions have been included. (For more information on the capstone experience recommended by GSwE2009, please see Section 5, Curriculum Architecture.)

Configuration Management. Generally, the management discipline focused on maintaining consistent structure and performance for a specific product. Configuration management practices help to identify the configuration for a product, track any changes to the product that may alter configuration, and verify that changes do not detrimentally affect performance. For GSwE2009, configuration management refers specifically to software configuration management. (For additional information, please see Appendix C.)

Core Body of Knowledge (CBOK). The recommended knowledge areas that should be obtained within a software engineering master's degree program. In addition, the CBOK provides a recommendation as to the appropriate Bloom's level for each knowledge area. (The CBOK is described in Section 6 of this document.)

Core materials. Fundamental skills and knowledge that all students must master within a given program.

Course. A collection of material, exercises and assessment for which academic credit is awarded, which may be part of a number of programs.

Credit Hours. A unit used to indicate the amount of in-class time for a given course. Generally, this refers to one hour of class time per week per term. This may be affected by the types of terms used (e.g., semesters vs. quarters) and by the instructional mode (e.g., on-line vs. traditional classroom). (Also referenced as course credits.)

Curriculum. All the courses associated with a specific course of study. The curriculum will depend on the level (e.g., graduate or undergraduate) and specificity (i.e., discipline or specialty) of the course of study.

Curriculum Architecture. The structure and framework used to develop a specific course of study. The GSwE2009 Curriculum Architecture is discussed in Section 5.

Degree Program. A collection of courses, delivered by an appropriate authority, leading to an academic degree.

Elective Materials. A set of courses to accommodate different interests and goals of individual students that may include special topics.

Engineering Economics. The application of economic principles to engineering projects. This discipline generally considers the economic implications along with the technical aspects when determining a solution to a particular problem. This discipline is recommended within the GSwE2009 curriculum and is discussed in further detail in Appendix C.

Entrance Expectations. Knowledge and skills expected of students when they enter an academic program. These are often prerequisites to the topics they will study.

Faculty. Academic or teaching staff. These may include both full-time permanent staff who are employed in an academic unit and external staff attached to the program, such as adjuncts..

GSwE2009-Satisfying Program. A university program that offers a master's in software engineering with a curriculum that largely satisfies GSwE2009 recommendations. Reasonable deviation from those recommendations for individual university or program constraints is expected. There is no precise measure of how much deviation is "reasonable".

Human Computer Interface Design. The discipline concerned with providing user-friendly displays that better enable individuals to comprehend electronic information.

Integration and Verification (I&V). The process that combines implemented and tested system elements to realize the System of Interest (SoI).

Lessons Learned. A description of the insights gained when attempting to address a problem and which may prove valid in future situations. In GSwE2009, lessons learned specifically refer to the insights gained when trying to adapt an existing SwE curriculum to the GSwE2009 curriculum. Lessons learned are primarily discussed in the companion document *Frequently Asked Questions on Implementing GSwE2009*.)

Leveling Course. A course designed to allow students who do not meet entrance expectations to enroll in an academic program. In general, these are courses designed to ensure that students have the requisite knowledge, skills, and abilities to succeed in the program. These may also be referred to as *bridging courses* or *preparatory courses*.

Master's Degree. A graduate or professional-level degree intended to follow an undergraduate course of study. Within GSwE2009, a master's degree in software engineering is focused on developing knowledge, skills, and abilities to meet the current and future challenges of complex systems that require software in order to operate properly.

Outcomes. The expected accomplishments of an individual who has completed an academic program. (Please see Section 3, Expected Outcomes When A Student Graduates.)

Pedagogy. The style of instruction and strategies used within a specific course of study. Pedagogy is discussed primarily within the *Frequently Asked Questions on Implementing GSwE2009*.

Preparatory Course. See *Leveling Course.*

Program. See *Degree Program*.

Program track. A specific set of courses within a program that emphasizes different areas of study such as telecommunications, real-time systems, and information systems.

Practical experience. Professional experience that allows a student to be exposed to a team environment and the product life cycle in the context of software engineering.

Reference Curriculum. A set of outcomes, entrance expectations, architecture, and a body of knowledge that provide guidance for faculty who are designing and updating their programs. That guidance is intentionally flexible so that faculty can adopt and adapt it based on local programmatic needs. A reference curriculum is not intended to be used directly for program certification or accreditation.

Requirements Engineering. The process for determining the necessary capabilities and/or functions for a specific product or service. Within GSwE2009, requirements engineering refers specifically to the development of software requirements.

Risk Management. The process of assessing potential threats to an endeavor, developing strategies to both reduce the probability of the problem occurring and counter these threats if they occur, and implementing these strategies using program resources. Risk management in GSwE2009 specifically refers to the processes used to understand and mitigate software-related risks for a project or product.

Software Engineering (SwE). A systematic approach to the development of operational software, and the maintenance of that software.

Software Maintenance. Continued support of a software product after delivery, either to correct deficiencies or errors or to enhance the functionality of the software and its performance.

Software Security. Ensuring that software continues to function correctly in spite of attack or misuse.

Supporting Processes. A GSwE2009-specific supplement to the knowledge areas presented in the SWEBOK. Specifically, this is a knowledge area that includes the activities of configuration management, verification and validation, quality assurance, reviews and audits, and software documentation processes.

Systems Engineering (SE): An interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle and documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem.

Testing. The process by which a product is systematically checked to ensure that a product functions as expected. Within GSwE2009, this specifically refers to the discipline of software testing.

Track. See *Program Track*.

University-specific materials. Specific materials that an institution might include in order to tailor a program to meet specific objectives. For example, university-specific materials may be used for a program with a specific focus in security or human computer interfaces.

Verification and Validation (V&V). The process of ensuring that a product or program meets its specifications and purpose and satisfies the stakeholders' needs. Specifically, this process often ensures that all critical stakeholder requirements are met and that the product will provide the target capabilities specified by the stakeholders. Within GSwE2009, this specifically refers to methods for ensuring that software programs enable appropriate functionality.

Index